OmniCRM Operations Guide

OmniCRM is Omnitouch's comprehensive BSS/OSS solution for mobile and fixed-line service providers. A complete platform that handles everything from customer onboarding to billing, provisioning, and support - all in one integrated system.

Getting Started

For Customer Service Staff

Your first steps:

1. **Learn the interface** - Get familiar with navigation and search
2. **Create a customer** - Step-by-step customer creation
3. **Add a service** - Provision your first service
4. **Process a payment** - Handle customer payments
5. **Top-up services** - Add credit to customer accounts

Daily tasks:

- **Service management** - Manage customer services
- **View usage** - Check balances and usage
- **Modify services** - Change service configurations
- **Generate invoices** - Create and send invoices

For System Administrators

Setup and configuration:

1. **Understand the architecture** - System overview
2. **Configure the system** - System settings
3. **Create products** - Build your catalog
4. **Write playbooks** - Automate provisioning
5. **Set up users** - Create accounts and assign roles

Advanced topics:

- **Inventory management** - Manage assets
- **Customization** - Tailor to your needs
- **API integration** - Connect external systems
- **Security setup** - Configure 2FA and permissions

For Customers

Using the Self-Care Portal:

- **Access your account** - Log in and navigate
- **View services** - See your active services
- **Check usage** - Monitor data and balances
- **Pay invoices** - Make payments online
- **Top-up services** - Add credit instantly

Quick Task Reference

I want to...	Documentation
Create a new customer	Create Customer
Add a service to a customer	Add Service
View service usage and balances	Service Usage
Process a payment	Process Payment
Generate an invoice	Invoice Management
Top-up a service	Top-Up & Recharge
Manage inventory	Inventory System
Create a product	Product Lifecycle
Write a provisioning playbook	Ansible Playbooks
Set up user accounts	RBAC
Enable 2FA	Two-Factor Authentication
Search for anything	Global Search
Understand the system	System Architecture
Use the API	API Documentation

Documentation Library

Essential Reading

- **System Architecture** - Complete technical overview with diagrams
- **Product Lifecycle Guide** - End-to-end product management
- **Ansible Playbooks Guide** - Master provisioning automation

Customer Management

- Customers, Contacts & Sites - Data model and relationships
- Creating Customers - Step-by-step guide
- Customer Tags - Organize with tags
- Customer Attributes - Custom metadata
- Activity Log - Track all changes

Service Operations

- Service Management - Overview
- Adding Services - Provisioning workflow
- Assigning Plans - Plan assignment
- Modifying Services - Making changes
- Service Usage & Balances - Monitor usage
- Top-Up & Recharge - Add credits

Billing & Finance

- Billing Overview - Complete billing guide
- Payment Methods - Managing payment methods
- Transactions - Charges and credits
- Processing Payments - Payment workflows
- Invoice Management - Generate and manage invoices

Administration

- System Configuration - Configure OmniCRM
- Customization - Tailor to your business
- Inventory Management - Asset tracking
- API Keys - API access management

Security & Access

- Authentication Flows - How authentication works
- Two-Factor Authentication - Set up 2FA
- Role-Based Access Control - Users, roles, and permissions

Integrations

- Stripe Integration - Payment processing
- Mailjet Integration - Email service
- API Documentation - REST API reference

Customer-Facing

- Self-Care Portal - Customer portal guide
- Customer Care - Self-service features
- Cell Broadcast System - Emergency alerts

Reference

- Glossary - Terms and definitions
- Changelog - Version history

What Makes OmniCRM Special?

OmniCRM brings together all the tools you need to run a modern telecommunications business:

Automated Service Provisioning

Forget manual configuration - OmniCRM uses **Ansible automation** to provision services in seconds. Whether you're activating a SIM card, configuring customer equipment, or setting up a complex bundle, the system handles it automatically.

What it does:

- Provisions services with a single click
- Configures network equipment automatically
- Creates billing accounts in real-time
- Sends welcome emails and SMS notifications
- Rolls back automatically if something fails

Learn more about Provisioning | **See Ansible Playbooks Guide**

Smart Billing & Payments

Built-in integration with **CGRateS** provides real-time rating and charging, while **Stripe integration** handles payment processing seamlessly.

What it does:

- Real-time usage tracking and rating
- Automatic invoice generation
- Credit card processing via Stripe
- Prepaid and postpaid billing models
- Flexible pricing and promotional codes
- Tax calculation and reporting

Explore Billing Features | **Payment Processing** | **Invoice Management**

Complete Customer Management

Manage customers, contacts, multiple sites, and services with a comprehensive relationship model.

What it does:

- Track customers, contacts, and service locations
- Manage multiple services per customer
- Store custom attributes and metadata
- Tag customers for organization

- Complete activity history and audit logs
- Google Maps integration for site geocoding

Customer Management Guide | **Create a Customer** | **Customer Tags**

Inventory Management

Track and manage all your physical and virtual assets - from SIM cards to routers to IP address blocks.

What it does:

- Track SIM cards, equipment, and accessories
- Manage phone numbers and IP address blocks
- Automated assignment during provisioning
- Batch import and export capabilities
- Equipment configuration storage
- Custom inventory templates

Inventory System Documentation

Customer Self-Care Portal

Empower your customers with a **self-service portal** where they can manage their own services.

What customers can do:

- View services and usage
- Check balances and data allowances
- Pay invoices online
- Download invoices as PDFs
- Update contact information
- **Top-up their services** instantly

Self-Care Portal Guide | **Customer Care Features**

Enterprise Security

Built with security at its core, featuring comprehensive authentication and authorization.

Security features:

- **JWT-based authentication**
- **Two-factor authentication (2FA)** with TOTP and backup codes
- **Role-based access control (RBAC)** with granular permissions
- Email verification for account changes

- Complete audit logging via activity log
- Session management and timeout controls

Authentication Guide | **RBAC Documentation** | **2FA Setup**

Powerful Integrations

OmniCRM integrates seamlessly with industry-leading services:

- **CGRateS** - Real-time billing and rating engine for telecom-grade charging
- **Stripe** - Secure payment processing and card storage
- **Mailjet** - Professional email delivery with templates
- **Google Maps** - Address validation and geocoding for accurate site locations
- **RESTful API** - Integrate with your existing systems

API Documentation | **Stripe Integration** | **Mailjet Integration**

Flexible Product Catalog

Create any type of product offering - from simple standalone services to complex bundles with multiple components.

Product types supported:

- **Standalone** - Single services (mobile plans, internet packages)
- **Bundles** - Combined offerings (internet + TV + phone)
- **Addons** - Supplementary services (data topups, international calling)
- **Promotions** - Special offers and discounts

Each product can have its own provisioning automation, pricing rules, and business logic.

Product Lifecycle Guide | **Products & Services Concepts**

Emergency Broadcast System

For mobile operators, OmniCRM includes a **Cell Broadcast System** for public safety alerts.

What it does:

- Send emergency alerts to geographic areas
- Multi-language message support
- Targeting by mobile network operator
- Compliance with government alert standards

Cell Broadcast Documentation

Powerful Search & Navigation

Find anything instantly with **global search** across customers, services, invoices, and more.

Search features:

- Search across all entities
- Filter and sort results
- Quick navigation shortcuts
- Smart suggestions

Navigation Guide | **Global Search**

System Architecture

View Complete Architecture Documentation

Key Concepts

Understanding Products vs Services

This is one of the most important concepts in OmniCRM:

- **Product** = A template or offering in your catalog (e.g., "Unlimited Mobile Plan")
- **Service** = An active instance of a product for a specific customer (e.g., "John Smith's Unlimited Mobile Plan")

When you provision a product, the system uses **Ansible automation** to create the actual service(s). One product can create multiple services, no services (configuration only), or modify existing services.

Learn more about Products & Services

The Provisioning Magic

OmniCRM's secret sauce is its **automated provisioning system**:

The playbook handles everything - from creating accounts in CGRateS to configuring routers to sending welcome emails. And if anything fails, it automatically rolls back.

Deep Dive: Ansible Playbooks | **Provisioning System**

Billing Made Simple

Every service generates **transactions** (charges or credits). These are grouped into **invoices** and sent to customers. With Stripe integration, payments can be processed automatically.

Billing Overview | **Payment Processing**

Customer Hierarchy

```
Customer (ABC Company)
├── Contacts (John Smith - Billing, Jane Doe - Technical)
├── Sites (Head Office, Branch Office)
└── Services
    ├── Internet - Head Office
    ├── Mobile Plan - John Smith
    └── VoIP - Branch Office
```

Everything is organized hierarchically, making it easy to manage customers with multiple locations and services.

Customers, Contacts & Sites

Why Choose OmniCRM?

All-in-One Platform

Everything you need in a single integrated system - no need to juggle multiple tools or vendors.

Automation First

Ansible-powered provisioning means services are deployed in seconds, not hours. Reduce errors, save time, increase customer satisfaction.

Flexible & Customizable

From product definitions to provisioning playbooks to custom attributes - tailor OmniCRM to match your exact business processes.

Built for Telecom

Designed specifically for service providers with features like CGRateS integration, inventory management, and real-time rating.

Enterprise Security

JWT authentication, 2FA, RBAC, and complete audit logging ensure your data is secure and compliant.

API-Driven

A comprehensive REST API means you can integrate OmniCRM with any existing systems or build custom tools.

Get Support

Documentation Resources

- Start with **System Architecture** for a technical overview
- See **Getting Started guides** for role-specific onboarding
- Check the **Quick Task Reference** for common operations
- Consult the **Glossary** for terminology

Need More Help?

Browse the complete documentation library above or use the **global search** to find specific topics.

OmniCRM - Complete BSS/OSS for Modern Service Providers

Last Updated: 2025-12-23

Two-Factor Authentication (2FA)

Enhanced Security with Time-Based One-Time Passwords

OmniCRM supports **two-factor authentication (2FA)** using time-based one-time passwords (TOTP). This adds an extra layer of security by requiring users to provide both their password and a time-sensitive code from an authenticator app.

See also: `RBAC <rbac>` for 2FA management permissions, `Authentication Flows <authentication_flows>` for login process details.

Purpose

2FA provides:

1. **Enhanced Security** — Protects accounts even if passwords are compromised.
2. **Compliance** — Meets security requirements for regulated industries.
3. **User Choice** — Optional for users, can be enforced per role or globally.
4. **Industry Standard** — Uses TOTP protocol compatible with Google Authenticator, Authy, Microsoft Authenticator, and other standard apps.

How 2FA Works

When 2FA is enabled for a user:

1. **Setup** — User scans a QR code with their authenticator app during enrollment.
2. **Login** — After entering username/password, user provides the 6-digit code from their app.
3. **Verification** — System validates the time-based code matches the expected value.
4. **Access** — User gains access only after both factors are verified.

Enabling 2FA

For Individual Users

Users can enable 2FA for their own account:

1. Navigate to **User Settings** or **Profile**
2. Select **Enable Two-Factor Authentication**
3. Scan the QR code with an authenticator app
4. Enter the verification code to confirm setup
5. Save backup codes in a secure location

2FA Setup

Backup & Recovery

Backup Codes

When enabling 2FA, users receive backup codes that can be used if their device is unavailable:

- Each code is single-use
- Store codes securely offline
- Generate new codes if all are used

Admin Reset

If a user loses access to their authenticator and all backup codes are exhausted, an administrator with database access can manually reset the user's 2FA by clearing the totp_secret field in the database. The user can then re-enable 2FA.

API Integration

Enable 2FA for a user

POST /2fa/enable/user/{user_id}

```json
{
  "password": "current_password"
}
```

Response includes provisioning URI (for QR code) and backup codes.

Verify 2FA setup

POST /2fa/verify-setup/user/{user_id}

```json
{
  "code": "123456"
}
```

Verify 2FA during login

POST /2fa/verify/user/{user_id}

```json
{
  "code": "123456"
}
```

Returns access token, refresh token, and user data upon successful verification.

2FA Login Verify

Regenerate backup codes

POST /2fa/backup-codes/regenerate/user/{user_id}

Requires authentication. Returns new set of backup codes.

Best Practices

- **Backup codes first.** Always save backup codes before completing 2FA setup.
- **Educate users.** Provide clear instructions for setup and recovery.
- **Secure reset process.** Verify user identity before manually resetting 2FA in the database.

FAQ

What authenticator apps are supported? Any TOTP-compatible app (Google Authenticator, Authy, Microsoft Authenticator, 1Password, etc.).

What if I lose my phone? Use a backup code or contact an administrator to reset 2FA.

Can I use SMS instead of an app? Currently, only TOTP authenticator apps are supported.

Is 2FA required? It depends on your organization's policy. 2FA is typically required for administrative and support staff roles but is optional for customer users. The system does not enforce 2FA for customer accounts (customer role users do not see 2FA enrollment prompts).

How long are TOTP codes valid? Codes refresh every 30 seconds and have a small time window for validation (typically accepts codes from current 30-second window plus previous/next windows for clock skew tolerance).

API Key Management

The API Key Management interface provides a **web-based UI** for creating, monitoring, and managing API keys used for programmatic access to the OmniCRM API.

Note

For general API authentication concepts and usage examples, see `concepts_api`.

Overview

API keys enable **secure, long-lived authentication** for:

- Server-to-server integrations
- Automation scripts
- Third-party applications
- Scheduled tasks and cron jobs
- External monitoring systems

Unlike JWT tokens (which expire after minutes/hours), API keys remain valid until manually revoked or until their expiry date.

Accessing API Key Management

Navigate to:

Or directly:

Required Permission: `MANAGE_API_KEYS` (admin role)

API Key List View

The main page displays all API keys in a table format:

Columns:

- **Name** - Descriptive label for the API key (e.g., "Provisioning System", "Monitoring Tool")
- **Created By** - Username of the person who created the key
- **API Key** - The actual key string (partially masked for security)
- **Status** - Active, Expired, or Revoked
- **Created Date** - When the key was generated

- **Expiry Date** - When the key will automatically expire
- **Actions** - Edit, Delete, Regenerate buttons

Example Display:

Dashboard Widgets

At the top of the page, summary statistics are displayed:

- **Total API Keys** - Count of all API keys (active and inactive)
- **Active Keys** - Currently valid keys
- **Expiring Soon** - Keys expiring in the next 30 days
- **Expired Keys** - Keys past their expiry date

Creating an API Key

Step 1: Click "Add API Key"

Click the **+ Add** button at the top right of the API Key list.

Step 2: Fill in Details

A modal form appears requesting:

> Name: ________________________
> (e.g., "Provisioning System")
>
> Description: __________________
> (Optional - purpose of this key)
>
> Expiry Date: [Date Picker]
> (Optional - leave blank for no expiry)
>
> Permissions: ☐ View Customers ☐ Create Customers ☐ View Services ☐ Create Services ☐ Provisioning ☐ View Inventory ☑ Admin (all permissions)
>
> [Cancel] [Generate Key]

Field Guidelines:

Name (required)

- Short, descriptive identifier
- Examples: "Provisioning System", "Billing Integration", "Monitoring"
- Used in audit logs and displayed in the list

Description (optional)

- More detailed explanation
- Examples: "Used by Ansible provisioning server", "Third-party billing sync"
- Helps future administrators understand the key's purpose

Expiry Date (optional)

- If blank: Key never expires (not recommended)
- If set: Key automatically becomes invalid after this date
- Recommended: Set expiry for security (90 days to 1 year)

Permissions

- Select specific permissions or check "Admin" for full access
- Follows same role-based permission model as user accounts
- **Best Practice:** Assign minimum necessary permissions

Step 3: Generate and Copy Key

After clicking **"Generate Key"**, the system displays the newly created API key:

> ⚠ Copy this key now - it will not be shown again!
>
> sk_live_a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0
>
> [Copy to Clipboard]
>
> [Close]

Warning

Save the API key immediately!

Once you close this dialog, the full key cannot be retrieved again. You'll only see a masked version (`sk_live_...XYZ`) in the list view.

If you lose the key, you must **regenerate** it, which invalidates the old key and may break existing integrations.

Step 4: Configure Your Application

Use the API key in your application's requests:

```
curl -X GET "https://yourcrm.com/crm/customers" \
  -H "X-API-KEY: sk_live_a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0"
```

Or in environment variables:

```
export CRM_API_KEY="sk_live_a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0"
```

Managing Existing Keys

Viewing Key Details

Click on any API key name to view full details:

- Full key name and description
- Creation timestamp
- Creator username
- Associated permissions
- Usage statistics (if implemented)
- Recent access logs

Regenerating an API Key

If an API key is compromised or lost, regenerate it:

1. Click the **⋮ (three dots)** in the Actions column
2. Select **"Regenerate Key"**
3. Confirm the action

Warning

Regenerating invalidates the old key immediately.

Any applications using the old key will stop working. Update all integrations with the new key before regenerating.

What Happens:

- Old key is revoked
- New key with same permissions is generated
- New key is displayed (copy it immediately)
- Name, description, and permissions remain unchanged

Revoking (Deleting) an API Key

To permanently remove an API key:

1. Click the **⋮ (three dots)** in the Actions column
2. Select **"Delete"**
3. Confirm deletion

What Happens:

- Key is immediately revoked
- All requests using this key return `401 Unauthorized`
- Key is removed from the database

- **Cannot be undone** - key cannot be recovered

When to Revoke:

- Integration is no longer needed
- Key has been compromised
- System using the key has been decommissioned
- Replacing with a new key with different permissions

Editing API Key Details

To modify an API key's details:

1. Click the **⋮ (three dots)** in the Actions column
2. Select **"Edit"**
3. Update name, description, expiry, or permissions
4. Click **"Save Changes"**

Editable Fields:

- Name
- Description
- Expiry date
- Permissions

Non-Editable:

- The key value itself (use Regenerate to change)
- Created date
- Created by user

API Key Status

API keys can have several statuses:

Active

- Key is valid and can be used
- Within expiry date (or no expiry set)
- Not manually revoked
- Displayed with green badge

Expiring Soon

- Active but will expire in the next 30 days
- Displayed with orange/warning badge
- Consider rotating before expiry

Expired

- Past the expiry date
- No longer accepts authentication
- Displayed with red badge
- Can be deleted or expiry extended

Revoked

- Manually deleted/disabled
- Permanently invalid
- No longer shown in active list

Filtering and Searching

The API Key list supports:

Search:

Search by name, description, or partial key:

Filter by Status:

Filter dropdown to show:

- All Keys
- Active Only
- Expiring Soon (next 30 days)
- Expired

Sort:

Click column headers to sort by:

- Name
- Created Date
- Expiry Date
- Created By

Security Best Practices

API Key Generation

- **Length:** Keys should be at least 32 characters (system enforces this)
- **Randomness:** Generated using cryptographically secure random number generators
- **Format:** Typically prefixed (e.g., `sk_live_`) for identification

API Key Storage

In the CRM:

- Keys are hashed before storage (like passwords)
- Full key only shown once during creation
- Database stores hash for verification
- Even administrators cannot retrieve the full key later

In Your Application:

- Store in environment variables, not code
- Use secret management systems (AWS Secrets Manager, HashiCorp Vault)
- Never commit keys to version control
- Rotate keys periodically (90-365 days)

Permission Management

- **Principle of Least Privilege** - Only grant necessary permissions
- Avoid creating admin keys unless absolutely required
- Use separate keys for different systems/purposes
- Review permissions regularly

Monitoring and Auditing

- Monitor API key usage via activity logs
- Set up alerts for unusual access patterns
- Review "last used" timestamps regularly
- Remove unused keys

Key Rotation

Establish a key rotation policy:

1. **Create new key** with same permissions
2. **Update applications** to use new key
3. **Monitor** to ensure old key is no longer used
4. **Revoke old key** after grace period

Troubleshooting

"401 Unauthorized" when using API key

- **Cause:** Key invalid, expired, or incorrect
- **Fix:**
    - Verify key is copied correctly (no extra spaces)
    - Check key status (Active vs. Expired)

    - Confirm key has required permissions
    - Ensure using `X-API-KEY` header (not `Authorization`)

"API key not found" after creation

- **Cause:** Key may have been created but not properly stored
- **Fix:**
    - Check API key list for the new entry
    - If missing, create a new key
    - Report issue to administrator

API key expiring soon

- **Cause:** Expiry date approaching (within 30 days)
- **Fix:**
    - Create new key with extended expiry
    - Update applications to use new key
    - Revoke old key after migration

Cannot delete API key

- **Cause:** May be protected or in use
- **Fix:**
    - Ensure you have admin permissions
    - Check if key is locked/protected
    - Contact administrator if issue persists

API Endpoints (for Programmatic Management)

API keys can also be managed via API (requires admin permissions):

List API Keys

```
GET /crm/api-keys
Authorization: Bearer <admin-token>
```

Create API Key

```
POST /crm/api-keys
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "name": "New Integration",
  "description": "Third-party billing sync",
  "expiry_date": "2026-01-10",
  "permissions": ["view_customer", "view_service"]
}
```

Response:

```
{
  "api_key_id": 123,
  "name": "New Integration",
  "api_key": "sk_live_a1b2c3d4e5f6g7h8i9j0",
  "status": "active",
  "created": "2025-01-10T10:00:00Z",
  "expiry_date": "2026-01-10T23:59:59Z"
}
```

Revoke API Key

```
DELETE /crm/api-keys/{api_key_id}
Authorization: Bearer <admin-token>
```

Update API Key

```
PATCH /crm/api-keys/{api_key_id}
Authorization: Bearer <admin-token>
Content-Type: application/json

{
  "name": "Updated Name",
  "expiry_date": "2026-12-31"
}
```

Common Use Cases

Use Case 1: Provisioning System Integration

Create an API key for your Ansible provisioning server:

1. Navigate to API Keys → Add
2. Name: "Ansible Provisioning Server"
3. Description: "Used by provisioning automation"
4. Permissions: Provisioning, View/Create Services, View/Update Inventory
5. Expiry: 1 year
6. Copy key and add to Ansible `crm_config.yaml`

Use Case 2: Third-Party Billing Integration

Create a read-only key for billing export:

1. Name: "Billing Sync - QuickBooks"
2. Permissions: View Customers, View Transactions, View Invoices
3. Expiry: 90 days (rotate quarterly)
4. Use in scheduled export script

Use Case 3: Monitoring and Alerting

Create a key for Prometheus/Grafana metrics collection:

1. Name: "Monitoring - Grafana"
2. Permissions: View Services, View Provisioning
3. Expiry: Never (monitoring needs continuous access)
4. Configure in Grafana datasource

Use Case 4: Customer Portal API

Create a key for customer self-service portal:

1. Name: "Customer Portal Backend"
2. Permissions: View Own Customer, View Own Services, Create Payments
3. Expiry: 180 days
4. Use in portal backend API calls

Related Documentation

- `concepts_api` - API authentication concepts and examples
- `rbac` - Role-based access control and permissions
- `2fa` - Two-factor authentication for additional security

Customer Attributes

Customer Attributes are flexible key-value pairs that can be attached to any customer record to store custom metadata, configuration settings, or business-specific information that doesn't fit into standard customer fields.

For visual customer categorization and clickable links, see `Customer Tags <administration_tags>`. For basic customer information, see `Customers, Contacts, Sites & Services <basics_customers>`.

Unlike fixed database fields, attributes allow you to dynamically extend customer records without modifying the database schema. This makes them ideal for storing deployment-specific data, integration parameters, or custom business logic flags.

Purpose and Use Cases

Common use cases for Customer Attributes include:

1. Integration Data

Store external system identifiers or API keys specific to this customer:

- `external_crm_id` = "SF-12345" (Salesforce customer ID)
- `legacy_system_id` = "OLD-CRM-789" (Migration reference)
- `hubspot_contact_id` = "12345678" (HubSpot integration)

2. Custom Business Logic

Store flags or settings that control customer-specific behavior:

- `billing_method` = "quarterly" (Override default monthly billing)
- `auto_provision` = "true" (Enable automatic service provisioning)
- `support_tier` = "premium" (Custom support level)
- `credit_limit` = "10000" (Customer-specific credit limit)

3. Compliance and Regulatory Data

Track compliance-related metadata:

- `gdpr_consent_date` = "2025-01-01" (Data processing consent)
- `tax_exempt` = "true" (Tax exemption status)
- `regulatory_entity` = "FCC-123456" (Regulatory identifier)

4. Operational Metadata

Store operational information:

- `preferred_contact_method` = "email" (Communication preference)
- `account_manager` = "<john.smith@company.com>" (Assigned account manager)
- `onboarding_date` = "2025-01-15" (Customer lifecycle tracking)
- `churn_risk_score` = "0.23" (Predictive analytics)

5. Provisioning Parameters

Store provisioning-specific configuration:

- `radius_username_format` = "email" (Custom RADIUS format)
- `vlan_id` = "100" (Network configuration)
- `ipv6_enabled` = "true" (Feature flags)

Attributes vs. Standard Fields

Use Attributes When:

- Data is deployment-specific or varies by installation
- Requirements change frequently
- Storing integration-specific metadata
- Prototyping new features before adding database fields
- Data doesn't need complex querying or joins

Use Standard Fields When:

- Data is core to the customer model (name, email, address)
- Frequent searching, filtering, or reporting required
- Data has referential integrity constraints
- Performance is critical for large-scale queries

Managing Attributes via the UI

Viewing Customer Attributes

To view attributes for a customer:

1. Navigate to the customer's overview page
2. Click on the **Attributes** tab
3. You will see a table of all attributes for this customer, showing:
    - Attribute Name (key)
    - Attribute Value
    - Created date

    - Last Modified date

Creating a New Attribute

To create a new attribute for a customer:

1. Navigate to the customer's overview page
2. Click on the **Attributes** tab
3. Click the **Add Attribute** button
4. Fill in the required fields:
    - **Attribute Name** (required): The key/name for this attribute (e.g., `external_crm_id`)
    - **Attribute Value** (required): The value to store (e.g., `SF-12345`)
5. Click **Create Attribute**

Naming Conventions:

- Use lowercase with underscores: `external_system_id` ✓
- Avoid spaces: `external system id` ✗
- Keep names descriptive but concise
- Use consistent naming across customers for same attribute types

Editing an Attribute

To edit an existing attribute:

1. Navigate to the customer's overview page
2. Click on the **Attributes** tab
3. Find the attribute you want to edit in the table
4. Click the **Edit** (pencil) button
5. Modify the attribute name or value
6. Click **Update Attribute**

Note

Changing an attribute name creates a new key-value pair. Ensure this doesn't break integrations that depend on the original attribute name.

Deleting an Attribute

To delete an attribute:

1. Navigate to the customer's overview page
2. Click on the **Attributes** tab
3. Find the attribute you want to delete in the table
4. Click the **Delete** (trash) button
5. Confirm the deletion in the popup

Warning

Deleting attributes used by integrations, provisioning workflows, or billing logic may cause failures. Verify dependencies before deletion.

Attribute Field Reference

API Integration

Attributes can be managed programmatically via the API:

Create or Update an Attribute

Endpoint: `PUT /crm/attribute/`

Required Permission: `create_customer_attribute`

Request Body:

```json
{
  "customer_id": 123,
  "attribute_name": "external_crm_id",
  "attribute_value": "SF-12345"
}
```

Response:

```json
{
  "attribute_id": 456,
  "customer_id": 123,
  "attribute_name": "external_crm_id",
  "attribute_value": "SF-12345",
  "created": "2025-01-04 10:30:00",
  "last_modified": "2025-01-04 10:30:00"
}
```

Update an Existing Attribute

Endpoint: `PATCH /crm/attribute/attribute_id/{attribute_id}`

Required Permission: `update_customer_attribute`

Request Body:

```json
{
  "attribute_value": "SF-54321"
}
```

Get Attribute by ID

Endpoint: `GET /crm/attribute/attribute_id/{attribute_id}`

Required Permission: `view_customer_attribute`

Response:


```json
{
  "attribute_id": 456,
  "customer_id": 123,
  "attribute_name": "external_crm_id",
  "attribute_value": "SF-12345",
  "created": "2025-01-04 10:30:00",
  "last_modified": "2025-01-04 10:30:00"
}
```


Get All Attributes by Customer ID

Endpoint: `GET /crm/attribute/customer_id/{customer_id}`

Required Permission: `view_customer_attribute`

Response:


```json
[
  {
    "attribute_id": 456,
    "customer_id": 123,
    "attribute_name": "external_crm_id",
    "attribute_value": "SF-12345",
    "created": "2025-01-04 10:30:00",
    "last_modified": "2025-01-04 10:30:00"
  },
  {
    "attribute_id": 457,
    "customer_id": 123,
    "attribute_name": "billing_method",
    "attribute_value": "quarterly",
    "created": "2025-01-04 10:35:00",
    "last_modified": "2025-01-04 10:35:00"
  }
]
```


Delete an Attribute

Endpoint: `DELETE /crm/attribute/attribute_id/{attribute_id}`

Required Permission: `delete_customer_attribute`

Response:

```
{
  "result": "success"
}
```

Bulk Attribute Operations

Managing Multiple Attributes

To set multiple attributes for a customer at once (e.g., during onboarding or integration sync):

```
import requests

customer_id = 123
attributes = [
    {"attribute_name": "external_crm_id", "attribute_value":
"SF-12345"},
    {"attribute_name": "billing_method", "attribute_value":
"quarterly"},
    {"attribute_name": "support_tier", "attribute_value": "premium"}
]

for attr in attributes:
    attr["customer_id"] = customer_id
    requests.put(
        "https://api.example.com/crm/attribute/",
        json=attr,
        headers={"Authorization": "Bearer YOUR_TOKEN"}
    )
```

Querying Customers by Attribute

While attributes don't have built-in search endpoints, you can filter customers by attribute using the customer search API with custom filtering:

```
# Get all customers, then filter by attribute in application code
customers = requests.get("https://api.example.com/crm/
customer/").json()

for customer in customers:
    attributes = requests.get(
        f"https://api.example.com/crm/attribute/
customer_id/{customer['customer_id']}"
    ).json()
```

```python
    # Find customers with specific attribute
    for attr in attributes:
        if attr['attribute_name'] == 'support_tier' and 
attr['attribute_value'] == 'premium':
            print(f"Premium customer: {customer['customer_name']}")
```

Note

For frequent attribute-based queries, consider adding indexed database fields or implementing a dedicated search endpoint.

Best Practices

1. Naming Conventions

- Use snake_case: `external_system_id` ✓
- Be descriptive: `billing_method` ✓ vs `method` ✗
- Avoid reserved keywords or special characters
- Document attribute meanings in your deployment guide

2. Data Types

- Attributes store values as strings (max 150 characters)
- For booleans, use "true"/"false" (lowercase)
- For dates, use ISO 8601 format: "2025-01-04 10:30:00"
- For large JSON data, consider dedicated database fields instead

3. Validation

- Validate attribute values in application code before saving
- Use consistent value formats across customers
- Document expected values for each attribute name

4. Documentation

- Maintain a registry of attribute names and purposes
- Document which systems/integrations depend on specific attributes
- Include examples of valid values

5. Migration and Cleanup

- Regularly audit unused attributes
- Remove obsolete attributes after system migrations
- Version attribute names when changing schemas (e.g., `api_key_v2`)

Example Workflows

Onboarding Integration

When migrating customers from a legacy system:

```
# Store legacy system reference for debugging
PUT /crm/attribute/
{
  "customer_id": 123,
  "attribute_name": "legacy_crm_id",
  "attribute_value": "OLD-12345"
}

# Track migration date
PUT /crm/attribute/
{
  "customer_id": 123,
  "attribute_name": "migrated_date",
  "attribute_value": "2025-01-04"
}
```

Custom Billing Rules

Override default billing cycle for specific customer:

```
# Set quarterly billing
PUT /crm/attribute/
{
  "customer_id": 123,
  "attribute_name": "billing_cycle",
  "attribute_value": "quarterly"
}

# Then in billing code, check for attribute before processing
attributes = GET /crm/attribute/customer_id/123
billing_cycle = next(
    (a['attribute_value'] for a in attributes if a['attribute_name']
== 'billing_cycle'),
    'monthly'  # default
)
```

Feature Flags

Enable beta features for specific customers:

```
# Enable IPv6 provisioning
```

```
PUT /crm/attribute/
{
  "customer_id": 123,
  "attribute_name": "feature_ipv6_enabled",
  "attribute_value": "true"
}
```

Permissions

Attribute operations require the following permissions:

- `view_customer_attribute` - View attributes
- `create_customer_attribute` - Create new attributes
- `update_customer_attribute` - Modify existing attributes
- `delete_customer_attribute` - Remove attributes

See `rbac` for role-based access control configuration.

Troubleshooting

Attribute Not Appearing in UI

- Verify attribute was created (check API response)
- Refresh the page to reload customer data
- Check user has `view_customer_attribute` permission

Cannot Update Attribute

- Ensure you have `update_customer_attribute` permission
- Verify attribute_id is correct
- Check attribute belongs to specified customer

Integration Failing After Attribute Deletion

- Restore attribute with previous value
- Update integration code to handle missing attributes gracefully
- Audit attribute dependencies before deletion

Attribute Value Truncated

- Attribute values have 150 character limit
- For longer data, split into multiple attributes or use customer notes field
- Consider storing large data in dedicated database fields

System Configuration

OmniCRM uses two main configuration systems: **crm_config.yaml** for backend API settings and **environment variables** for the React UI. This guide covers all configuration options and how to modify them.

Configuration Files Overview

Backend API Configuration:

- **File:** `OmniCRM-API/crm_config.yaml`
- **Format:** YAML
- **Requires:** API restart after changes
- **Used for:** Database, integrations, security, provisioning

Frontend UI Configuration:

- **File:** `OmniCRM-UI/.env`
- **Format:** Environment variables
- **Requires:** UI rebuild after changes
- **Used for:** Branding, features, external services

Backend Configuration (crm_config.yaml)

The `crm_config.yaml` file contains all backend system settings.

Database Configuration

```yaml
database:
  username: omnitouch
  password: omnitouch2024
  server: localhost
```

Fields:

- `username` - MySQL database username
- `password` - MySQL database password
- `server` - Database server hostname or IP (default: localhost)

Database Connection:

- Database name is hardcoded as `omnicrm`
- Default port: 3306 (MySQL default)

- Connection string: `mysql+pymysql://username:password@server/omnicrm`

Security Note: Never commit this file with real credentials to version control. Use environment-specific configuration or secrets management.

Service Types

```yaml
service_types:
  - omnicharge
  - mobile
  - internet
  - iptv
  - voip
```

Purpose: Defines valid service type values for the `service_type` field.

Default Types:

- `mobile` - Mobile/cellular services
- `internet` - Fixed internet (fiber, DSL, wireless)
- `iptv` - Television services
- `voip` - Voice-over-IP services
- `omnicharge` - Billing/charging services

Add custom service types here for your specific use cases.

HSS (Home Subscriber Server) Configuration

```yaml
hss:
  hss_peers:
    - 'http://10.179.2.140:8080'
  apn_list: "1,2,3,4,5,6"
```

Fields:

- `hss_peers` - List of HSS server URLs for subscriber management
- `apn_list` - Comma-separated list of APN (Access Point Name) identifiers

Used for: Mobile network provisioning and subscriber authentication.

Mailjet Email Configuration

```yaml
mailjet:
  api_key: your_mailjet_api_key
  api_secret: your_mailjet_api_secret

  api_crmCommunicationCustomerWelcome:
    from_email: "support@yourcompany.com"
```

```yaml
    from_name: "Your Company Support"
    template_id: 5977509
    subject: "Welcome to Your Company"

  api_crmCommunicationCustomerInvoice:
    from_email: "billing@yourcompany.com"
    from_name: "Your Company Billing"
    template_id: 6759851
    subject: "Your Invoice - "
```

Email Types Configured:

- `api_crmCommunicationCustomerWelcome` - New customer welcome email
- `api_crmCommunicationCustomerInvoice` - Invoice delivery
- `api_crmCommunicationCustomerInvoiceReminder` - Overdue invoice reminders
- `api_crmCommunicationUserWelcome` - New staff user welcome
- `api_crmCommunicationUserPasswordReset` - Password reset requests
- `api_crmCommunicationUserPasswordResetSuccess` - Successful password reset
- `api_crmCommunicationUserPasswordChange` - Password change notifications
- `api_crmCommunicationEmailVerification` - Email address verification
- `api_crmCommunicationsBalanceExpired` - Service expiry notifications
- `api_crmCommunicationsBalanceLow` - Low balance alerts

Template IDs:

Obtain from your Mailjet account after creating email templates. See `integrations_mailjet` for details.

Provisioning Configuration

```yaml
provisioning:
  failure_list: ['admin@yourcompany.com', 'ops@yourcompany.com']
```

Fields:

- `failure_list` - Email addresses to notify when provisioning fails

When Ansible playbooks fail during provisioning, the system sends error details to these addresses.

Invoice Configuration

```yaml
invoice:
  template_filename: 'your_invoice_template.html'
```

Fields:

- `template_filename` - HTML template file for invoice generation

Template file should exist in `OmniCRM-API/templates/` directory.

CRM Base URL

```yaml
crm:
  base_url: 'http://localhost:5000'
```

Purpose: Used by Ansible playbooks to make API calls back to the CRM.

Important:

- Do NOT include trailing slash
- Use publicly accessible URL if playbooks run on different servers
- Update when deploying to production (e.g., `https://api.yourcrm.com`)

OCS (Online Charging System) Configuration

```yaml
ocs:
  ocsApi: 'http://10.179.2.142:8080/api'
  ocsTenant: 'mnc380.mcc313.3gppnetwork.org'
  cgrates: 'localhost:2080'
```

Fields:

- `ocsApi` - OCS REST API endpoint URL
- `ocsTenant` - Tenant identifier for multi-tenant OCS deployments
- `cgrates` - CGRateS JSON-RPC endpoint (host:port)

Used for: Real-time charging, balance management, usage tracking.

SMSC (SMS Center) Configuration

```yaml
smsc:
  source_msisdn: 'YourCompany'
  smsc_url: 'http://10.179.2.216/SMSc/'
  api_key: 'your_smsc_api_key'
```

Fields:

- `source_msisdn` - Sender ID for outbound SMS (company name or short code)
- `smsc_url` - SMS Center API endpoint
- `api_key` - Authentication key for SMSC API

Used for: Sending SMS notifications (balance alerts, OTPs, etc.)

Cell Broadcast Configuration

```
cbc_url: 'http://10.179.1.113:8080'
```

Purpose: Cell Broadcast Center (CBC) API endpoint for emergency alerts.

See `features_cell_broadcast` for usage details.

JWT Secret Key

```
jwt_secret:
'2b93110f723db60172c8e9a1eaa80027a9a9c3f05b44e50dc3fcf38dba68d87e'
```

Purpose: Secret key for signing JWT authentication tokens.

Security:

- Generate using: `openssl rand -hex 32`
- **Never share publicly**
- Changing this invalidates all existing user sessions
- Use different secrets for dev/staging/production

Stripe Payment Configuration

```
stripe:
  secret_key: 'sk_test_...'
  publishable_key: 'pk_test_...'
  currency: 'usd'
  statement_descriptor_suffix: 'YOURCOMPANY'
```

Fields:

- `secret_key` - Stripe secret API key (starts with `sk_`)
- `publishable_key` - Stripe publishable key (starts with `pk_`)
- `currency` - ISO 4217 currency code (usd, gbp, aud, eur, etc.)
- `statement_descriptor_suffix` - Text appearing on customer credit card statements

Key Types:

- Test keys: `sk_test_...` and `pk_test_...` (for development)
- Live keys: `sk_live_...` and `pk_live_...` (for production)

See Payment Vendor Integrations <integrations_payment_vendors> for setup details.

API Keys

```yaml
api_keys:
  "your-secure-api-key-minimum-32-chars":
    roles: ["admin"]
    ips: ["127.0.0.1", "::1"]
  "another-api-key-for-specific-service":
    roles: ["customer_service_agent_1"]
    ips: ["10.0.1.50"]
```

Structure:

- **Key (string):** The actual API key (minimum 32 characters)
- **roles (list):** Role names this key has access to
- **ips (list, optional):** IP addresses allowed to use this key

Generating API Keys:

```
openssl rand -base64 48
```

Roles:

- `admin` - Full access to all endpoints
- Custom roles defined in RBAC system

See `administration_api_keys` and `concepts_api` for details.

IP Whitelist

```yaml
ip_whitelist:
  "10.179.2.142":
    roles: ["admin"]
  "192.168.1.100":
    roles: ["provisioning"]
```

Purpose: Allow specific IP addresses to access the API without authentication.

Structure:

- **IP Address (string):** IPv4 address to whitelist
- **roles (list):** Roles assigned to requests from this IP

Security Warning:

- Only use for trusted internal networks
- Must NOT use localhost IPs (127.0.0.1, ::1)
- Use API keys instead for external access
- Consider firewall rules as additional protection

Frontend Configuration (.env)

The React UI is configured via environment variables in `OmniCRM-UI/.env`.

Branding Configuration

```
REACT_APP_COMPANY_NAME="YourCompany"
REACT_APP_PORTAL_NAME="YourPortal"
REACT_APP_SELF_CARE_NAME="YourCare"
REACT_APP_COMPANY_TAGLINE="Your Company Slogan"
```

Fields:

- `REACT_APP_COMPANY_NAME` - Company name (appears in headers, emails, branding)
- `REACT_APP_PORTAL_NAME` - Admin portal name (page titles, navigation)
- `REACT_APP_SELF_CARE_NAME` - Customer portal name
- `REACT_APP_COMPANY_TAGLINE` - Marketing tagline (appears in login page)

Example:

Regional Configuration

```
REACT_APP_DEFAULT_LOCATION="London, United Kingdom"
REACT_APP_DEFAULT_COUNTRY="United Kingdom"
REACT_APP_DEFAULT_LANGUAGE="en"
REACT_APP_LOCALE="en-GB"
```

Fields:

- `REACT_APP_DEFAULT_LOCATION` - Default location for maps and addresses
- `REACT_APP_DEFAULT_COUNTRY` - Default country for forms
- `REACT_APP_DEFAULT_LANGUAGE` - UI language (ar, ch, en, fr, gr, it, ru, sp)
- `REACT_APP_LOCALE` - Date/number formatting locale (en-GB, en-US, etc.)

Supported Languages:

- `ar` - Arabic
- `ch` - Chinese
- `en` - English (default)
- `fr` - French
- `gr` - Greek
- `it` - Italian
- `ru` - Russian
- `sp` - Spanish

Currency Configuration

```
REACT_APP_CURRENCY_CODE="USD"
REACT_APP_CURRENCY_SYMBOL="$"
```

Fields:

- `REACT_APP_CURRENCY_CODE` - ISO 4217 currency code
- `REACT_APP_CURRENCY_SYMBOL` - Symbol to display (£, $, €, etc.)

Common Currencies:

- USD - $ (US Dollar)
- GBP - £ (British Pound)
- EUR - € (Euro)
- AUD - $ (Australian Dollar)
- CAD - $ (Canadian Dollar)

Note: Must match `stripe.currency` in `crm_config.yaml`.

Color Theme Configuration

```
REACT_APP_PRIMARY_COLOR=#405189
REACT_APP_SECONDARY_COLOR=#2bFFcf
REACT_APP_TERTIARY_COLOR=#1a9fbf
```

Available Colors:

- `REACT_APP_PRIMARY_COLOR` - Main brand color (buttons, links, highlights)
- `REACT_APP_SECONDARY_COLOR` - Accent color
- `REACT_APP_TERTIARY_COLOR` - Additional accent
- `REACT_APP_SUCCESS_COLOR` - Success messages (#28a745)
- `REACT_APP_INFO_COLOR` - Info messages (#17a2b8)
- `REACT_APP_WARNING_COLOR` - Warnings (#ffc107)
- `REACT_APP_DANGER_COLOR` - Errors (#dc3545)
- `REACT_APP_LIGHT_COLOR` - Light backgrounds (#f8f9fa)
- `REACT_APP_DARK_COLOR` - Dark text (#343a40)
- `REACT_APP_PRIMARY_DARK_COLOR` - Dark variant of primary color (for dark mode/hover states)

Format: Hexadecimal color codes (#RRGGBB)

Font Configuration

```
REACT_APP_FONT_FAMILY=Quicksand
```

Purpose: Sets the primary font family for the entire UI.

Important: All fonts are **self-hosted locally** within the OmniCRM-UI application. This means:

- **No external font loading** - Fonts are bundled with the application
- **Walled garden compatible** - No internet access required for fonts to work
- **Offline operation** - Full functionality in air-gapped or restricted network environments
- **Privacy** - No external requests to Google Fonts, Adobe Fonts, or other CDNs
- **Performance** - Faster loading without external dependencies
- **Security** - No third-party tracking or data leakage through font requests

Available Options:

Sans-Serif Fonts:

- Inter
- Roboto
- Open Sans
- Lato
- Quicksand (default)
- Poppins
- Nunito
- Montserrat
- Work Sans
- Source Sans Pro
- Raleway
- Ubuntu
- Josefin Sans
- HKGrotesk

Serif Fonts:

- Merriweather
- Lora
- Playfair Display

System Fonts:

- System - Uses native device fonts for best performance and smallest bundle size

Default: Quicksand

Adding Custom Fonts

Yes, you can add additional fonts! All fonts are stored locally in the application.

To add a new custom font:

1. **Add font files** to `OmniCRM-UI/src/assets/fonts/your-font-name/`
    - Use WOFF2 format for best compression and browser support
    - Include multiple weights (300, 400, 500, 600, 700) for proper rendering
    - Name files: `your-font-name-300.woff2`, `your-font-name-400.woff2`, etc.
2. **Define @font-face rules** in `OmniCRM-UI/src/assets/scss/fonts/_google-fonts.scss`

    ```
    //
    // Your Custom Font - Description
    //

    @font-face {
      font-family: 'Your Font Name';
      font-style: normal;
      font-weight: 400;
      font-display: swap;
      src: url("../../fonts/your-font-name/your-font-name-400.woff2") format('woff2');
    }

    @font-face {
      font-family: 'Your Font Name';
      font-style: normal;
      font-weight: 700;
      font-display: swap;
      src: url("../../fonts/your-font-name/your-font-name-700.woff2") format('woff2');
    }
    ```

3. **Set in .env** file:

    ```
    REACT_APP_FONT_FAMILY=Your Font Name
    ```

Font Weight Guidelines:

- 300 - Light (optional, for subtle headings)
- 400 - Regular (required, default text)
- 500 - Medium (optional, emphasis)
- 600 - Semi-Bold (optional, subheadings)
- 700 - Bold (required, headings and strong text)

Note: All fonts remain self-hosted and work offline. No external CDN or internet connection required.

External Services

```
REACT_APP_GOOGLE_API_KEY=your_google_maps_api_key
REACT_APP_STRIPE_PUBLISHABLE_KEY=pk_test_...
```

Fields:

- `REACT_APP_GOOGLE_API_KEY` - Google Maps API key (for maps, geolocation)
- `REACT_APP_STRIPE_PUBLISHABLE_KEY` - Stripe publishable key (for payments)

Must Match:

`REACT_APP_STRIPE_PUBLISHABLE_KEY` must match `stripe.publishable_key` in `crm_config.yaml`.

Web App Quick Links

```
REACT_APP_WEB_APP_1_NAME="GitHub"
REACT_APP_WEB_APP_1_URL="https://github.com"
REACT_APP_WEB_APP_1_ICON_PATH="resources/webapp_icons/github.png"
```

Purpose: Configure up to 6 quick-access web application links in the UI navigation.

Pattern:

- `REACT_APP_WEB_APP_N_NAME` - Display name
- `REACT_APP_WEB_APP_N_URL` - Target URL
- `REACT_APP_WEB_APP_N_ICON_PATH` - Icon file path (relative to public/)

Example Icons: GitHub, Xero, Monday.com, Gmail, MailJet, Slack

Grafana Integration

```
REACT_APP_GRAFANA_URLS=http://grafana1.local/d/
abc,http://grafana2.local/d/xyz
REACT_APP_GRAFANA_LABELS=Network Monitoring,Service Health
REACT_APP_GRAFANA_API_KEY=your_grafana_api_key
```

Fields:

- `REACT_APP_GRAFANA_URLS` - Comma-separated list of Grafana dashboard URLs
- `REACT_APP_GRAFANA_LABELS` - Comma-separated list of dashboard names
- `REACT_APP_GRAFANA_API_KEY` - Grafana API key for embedding

Usage: Embeds Grafana dashboards in the OmniCRM dashboard page.

Support URLs

```
REACT_APP_FAQS_URL=https://support.yourcompany.com/faqs
REACT_APP_SUPPORT_URL=https://support.yourcompany.com
```

Purpose: Links to external support resources shown in the UI.

Social Logins

```
REACT_APP_ALLOW_SOCIAL_LOGINS=yes
```

Options:

- `yes` - Enable social login buttons (Google, Facebook, etc.)
- `no` - Disable social logins

Note: Social login providers must be configured separately.

Top-Up and Recharge Configuration

```
REACT_APP_TOPUP_PRICE_PER_DAY=10
```

Purpose: Sets the price per day for top-up/recharge services in the self-care portal.

Fields:

- `REACT_APP_TOPUP_PRICE_PER_DAY` - Daily price for recharge services (numeric value)

Example: If set to `10` and currency is USD, customers pay $10 per day of service.

Note: This value must match the backend pricing configuration. See `features_topup_recharge` for complete setup details.

Applying Configuration Changes

Backend (crm_config.yaml)

1. Edit `OmniCRM-API/crm_config.yaml`
2. Save changes
3. Restart the API service:

```
cd OmniCRM-API
sudo systemctl restart omnicrm-api
# or
./restart_api.sh
```

Changes take effect immediately after restart.

Frontend (.env)

1. Edit `OmniCRM-UI/.env`
2. Save changes
3. Rebuild the UI:

```
cd OmniCRM-UI
npm run build
```

4. Restart the UI service or web server

Development Mode:

During development with `npm start`, restart the dev server to apply changes.

Configuration Best Practices

Security

- **Never commit secrets** - Use `.gitignore` for config files with credentials
- **Use environment-specific configs** - Separate dev/staging/production configs
- **Rotate secrets regularly** - Update JWT secrets, API keys periodically
- **Limit API key permissions** - Assign minimal necessary roles
- **Use IP whitelisting sparingly** - Prefer API keys for better security

Maintenance

- **Document changes** - Keep changelog of configuration modifications
- **Backup configs** - Store copies before major changes
- **Test in staging** - Verify config changes before production deployment
- **Version control** - Track config templates (without secrets) in git

Performance

- **Use local database** - Avoid remote database for better performance
- **Configure caching** - Enable OCS caching if available
- **Optimize Grafana** - Limit number of embedded dashboards

Branding

- **Match colors** - Ensure UI colors complement your logo
- **Test contrast** - Verify text readability on colored backgrounds
- **Mobile testing** - Check branding on mobile devices
- **Logo placement** - Place company logos in `OmniCRM-UI/public/`

```
resources/
```

Troubleshooting

Changes not applied

- **Cause:** Service not restarted or UI not rebuilt
- **Fix:** Restart API/UI services after config changes

YAML syntax errors

- **Cause:** Invalid YAML formatting (indentation, quotes, etc.)
- **Fix:** Validate YAML online or use `yamllint crm_config.yaml`

Database connection failed

- **Cause:** Wrong credentials or server unreachable
- **Fix:** Verify database is running, credentials are correct

Stripe payments not working

- **Cause:** Mismatched keys between backend and frontend
- **Fix:** Ensure `publishable_key` matches in both files

Emails not sending

- **Cause:** Invalid Mailjet credentials or template IDs
- **Fix:** Verify Mailjet API key/secret, check template IDs exist

Related Documentation

- `administration_api_keys` - API key management
- `Payment Vendor Integrations <integrations_payment_vendors>` - Payment vendor setup (Stripe, PayPal, etc.)
- `integrations_mailjet` - Email integration
- `concepts_api` - API authentication
- `rbac` - Role-based access control

Customization and Configuration

This guide explains how to customize and configure OmniCRM to match your brand identity, operational requirements, and integration needs.

Environment Variables (.env Files)

OmniCRM uses environment variables to configure both the backend API and frontend UI. There are two separate `.env` files that control different aspects of the system.

Backend API Configuration (.env)

Located at the root of the OmniCRM directory, this file configures database connections and CGRates integration.

Location: `/OmniCRM/.env`

Database Configuration:

```
# MySQL/MariaDB Database (CRM Data)
MYSQL_ROOT_PASSWORD=your_secure_password
MYSQL_USER=omnitouch
MYSQL_PASSWORD=your_database_password
MYSQL_DATABASE=crm
DB_SERVER=db

# PostgreSQL Database (CGRates Data)
POSTGRES_USER=cgrates
POSTGRES_PASSWORD=cgrates_password
POSTGRES_DB=cgrates_db
```

CGRates Configuration:

```
# CGRates API Credentials
CGRATES_API_USER=admin
CGRATES_API_PASS=secret
CGRATES_DB_USER=cgrates
CGRATES_DB_PASS=cgrates_password
CGRATES_DB_NAME=cgrates_db
CGRATES_DB_PORT=5432
```

Security Considerations:

- **Never commit .env files to version control** - Use `.env.example` as a template
- **Use strong passwords** - Minimum 16 characters with mixed case, numbers, and symbols
- **Rotate credentials regularly** - Especially for production deployments
- **Restrict database access** - Use IP whitelisting and firewall rules

Frontend UI Configuration (.env)

Located in the `OmniCRM-UI` directory, this file controls branding, appearance, integrations, and feature flags.

Location: `/OmniCRM/OmniCRM-UI/.env`

API Keys and Integration:

```
# Google Maps API (for address autocomplete and geocoding)
REACT_APP_GOOGLE_API_KEY=your_google_api_key

# Stripe Payment Gateway
REACT_APP_STRIPE_PUBLISHABLE_KEY=pk_test_xxxxx

# Disable browser auto-launch on npm start
BROWSER=none
```

Branding and Company Information:

```
# Company Branding
REACT_APP_COMPANY_NAME="ShellFone"
REACT_APP_PORTAL_NAME="ShellManager"
REACT_APP_SELF_CARE_NAME="ShellCare"
REACT_APP_COMPANY_TAGLINE="Phones with Shells"
```

These values appear throughout the UI:

- `COMPANY_NAME` - Displayed in page titles, emails, and customer communications
- `PORTAL_NAME` - Name of the admin/staff portal (e.g., "ShellManager")
- `SELF_CARE_NAME` - Name of the customer self-service portal (e.g., "ShellCare")
- `COMPANY_TAGLINE` - Appears in login screens and marketing materials

Localization and Regional Settings:

```
# Language and Locale
# Supported languages: ar, ch, en, fr, gr, it, ru, sp
REACT_APP_DEFAULT_LANGUAGE=en
REACT_APP_LOCALE="en-GB"
```

```
# Default Location (for address autocomplete)
REACT_APP_DEFAULT_LOCATION="Sydney, Australia"
REACT_APP_DEFAULT_COUNTRY="Australia"

# Currency Settings
REACT_APP_CURRENCY_CODE="GBP"
REACT_APP_CURRENCY_SYMBOL="£"
```

Color Scheme Customization:

```
# Primary Color (main brand color)
REACT_APP_PRIMARY_COLOR=#405189

# Additional Color Options (commented examples)
# REACT_APP_SECONDARY_COLOR=#2bFFcf
# REACT_APP_TERTIARY_COLOR=#1a9fbf
# REACT_APP_SUCCESS_COLOR=#28a745
# REACT_APP_INFO_COLOR=#17a2b8
# REACT_APP_WARNING_COLOR=#ffc107
# REACT_APP_DANGER_COLOR=#dc3545
```

The primary color is applied to:

- Navigation headers
- Action buttons
- Links and highlights
- Active states
- Brand elements

Web App Integrations:

Configure up to 6 quick-access web applications that appear in the admin dashboard:

```
# Web App 1: GitHub
REACT_APP_WEB_APP_1_NAME="GitHub"
REACT_APP_WEB_APP_1_URL="https://github.com"
REACT_APP_WEB_APP_1_ICON_PATH="resources/webapp_icons/github.png"

# Web App 2: Xero
REACT_APP_WEB_APP_2_NAME="Xero"
REACT_APP_WEB_APP_2_URL="https://go.xero.com/"
REACT_APP_WEB_APP_2_ICON_PATH="resources/webapp_icons/xero.png"

# Web App 3-6: Additional integrations
# (Configure similarly with NAME, URL, and ICON_PATH)
```

Monitoring and Analytics:

```
# Grafana Dashboard Integration
REACT_APP_GRAFANA_URLS=url1,url2,url3
REACT_APP_GRAFANA_LABELS=label1,label2,label3
REACT_APP_GRAFANA_API_KEY=your-api-key
```

Feature Flags:

```
# Support and Documentation URLs
REACT_APP_FAQS_URL=https://docs.yourcompany.com/faqs
REACT_APP_SUPPORT_URL=https://support.yourcompany.com

# Social Login (Google, Facebook, etc.)
REACT_APP_ALLOW_SOCIAL_LOGINS=yes
```

Logo and Splash Image Customization

OmniCRM allows you to replace default branding images with your company's logo and splash screens without modifying code.

Logo Files

Logos are stored in `/OmniCRM-UI/src/assets/images/omnitouch/` and use a fallback system:

Default Logos (always present):

- `DefaultLogoDark.png` - Dark theme logo (used on light backgrounds)
- `DefaultLogoLight.png` - Light theme logo (used on dark backgrounds)

Custom Logos (optional, take precedence when present):

- `logoSm.png` - Small logo for collapsed sidebar (recommended: 100x100px)
- `logoDark.png` - Full-size dark logo for headers (recommended: 200x50px)
- `logoLight.png` - Full-size light logo for authentication screens (recommended: 200x50px)

How Logo Fallback Works:

The system attempts to load custom logos first. If a custom logo file doesn't exist, it falls back to the default:

```
// From Header.js
const tryImport = (filename) => {
    try {
        return require(`../assets/images/omnitouch/${filename}`);
    } catch (err) {
        return null; // Falls back to default
    }
```

```
};

const userLogoSm = tryImport("logoSm.png");
const userLogoDark = tryImport("logoDark.png");
const userLogoLight = tryImport("logoLight.png");
```

Where Logos Appear:

- **logoSm.png** - Collapsed sidebar, mobile navigation, small header displays
- **logoDark.png** - Main header bar (light mode), admin dashboard header
- **logoLight.png** - Login/registration screens, dark backgrounds, authentication carousel

Replacing Logos:

1. **Create Your Logo Files:**

   - Use PNG format for transparency support
   - Match the recommended dimensions above
   - Ensure logos are clear at both regular and retina resolutions

2. **Add to OmniCRM:**

   ```
   # Copy your logo files to the omnitouch images directory
   cp /path/to/your/logoSm.png OmniCRM-UI/src/assets/images/omnitouch/
   cp /path/to/your/logoDark.png OmniCRM-UI/src/assets/images/omnitouch/
   cp /path/to/your/logoLight.png OmniCRM-UI/src/assets/images/omnitouch/
   ```

3. **Rebuild the UI:**

   ```
   cd OmniCRM-UI
   npm run build
   ```

4. **Verify Changes:**

   - Check light mode header (should show `logoDark.png`)
   - Check dark mode header (should show `logoLight.png`)
   - Check collapsed sidebar (should show `logoSm.png`)
   - Check login screen (should show `logoLight.png`)

Logo Design Best Practices:

- **Contrast** - Ensure logos are visible on both light and dark backgrounds
- **Simplicity** - Logos should be recognizable at small sizes
- **Format** - Use PNG with transparent backgrounds
- **Retina** - Provide 2x resolution for high-DPI displays

- **Consistency** - Use the same brand colors across all logo variants

Splash Screens and Authentication Backgrounds

The authentication screens (login, registration, password reset) use a carousel background with customizable images.

Location: `/OmniCRM-UI/src/pages/AuthenticationInner/authCarousel.js`

Default Configuration:

```
import logoLight from "../../assets/images/logo-light.png";

// Logo displayed on authentication screens
<img src={logoLight} alt="" height="18" />
```

Customizing Authentication Screens:

1. Replace `logo-light.png` in `/OmniCRM-UI/src/assets/images/`
2. Add custom background CSS to `.auth-one-bg` class
3. Modify carousel quotes in `authCarousel.js` to match your brand voice

Example Customization:

```
/* Add to your custom CSS */
.auth-one-bg {
    background-image: url('/assets/images/custom-auth-bg.jpg');
    background-size: cover;
    background-position: center;
}
```

CRM Configuration File (crm_config.yaml)

The `crm_config.yaml` file is the central configuration for the OmniCRM API, controlling integrations, provisioning, email templates, and external services.

Location: `/OmniCRM/OmniCRM-API/crm_config.yaml`

Database Configuration

```
database:
  username: omnitouch
  password: omnitouch2024
  server: localhost
```

Note: This should match your `.env` database credentials. In containerized deployments, the server is typically `db` (Docker service name).

Service Types

Define valid service types for your deployment:

```yaml
service_types:
  - omnicharge
  - mobile
  - fixed
  - fixed-voice
  - hotspot
  - dongle
```

These are used throughout the system for:

- Product categorization
- Addon filtering (addons match service types)
- Provisioning workflows
- Reporting and analytics

HSS Integration (Home Subscriber Server)

For mobile network operators with HSS integration:

```yaml
hss:
  hss_peers:
    - 'http://10.179.2.140:8080'
  apn_list: "1,2,3,4,5,6"
```

Configuration:

- `hss_peers` - List of HSS endpoints for subscriber provisioning
- `apn_list` - Comma-separated list of APN IDs available for provisioning

Mailjet Email Template Configuration

OmniCRM uses Mailjet for transactional emails. Each email type has its own template configuration:

```yaml
mailjet:
  api_key: your_mailjet_api_key
  api_secret: your_mailjet_api_secret

  # Customer Welcome Email
  api_crmCommunicationCustomerWelcome:
    from_email: "support@yourcompany.com"
    from_name: "Your Company Support"
    template_id: 5977509
    subject: "Welcome to YourCompany"
```

```yaml
# Customer Invoice Email
api_crmCommunicationCustomerInvoice:
  from_email: "billing@yourcompany.com"
  from_name: "Your Company Billing"
  template_id: 6759851
  subject: "Your Invoice - "

# Invoice Reminder
api_crmCommunicationCustomerInvoiceReminder:
  from_email: "billing@yourcompany.com"
  from_name: "Your Company Billing"
  template_id: 5977570
  subject: "Invoice Payment Reminder"

# User Welcome Email (Staff/Admin)
api_crmCommunicationUserWelcome:
  from_email: "admin@yourcompany.com"
  from_name: "Your Company Admin"
  template_id: 6118112
  subject: "Welcome to the Team"

# Password Reset Request
api_crmCommunicationUserPasswordReset:
  from_email: "security@yourcompany.com"
  from_name: "Your Company Security"
  template_id: 6735666
  subject: "Password Reset Request"

# Password Reset Success Confirmation
api_crmCommunicationUserPasswordResetSuccess:
  from_email: "security@yourcompany.com"
  from_name: "Your Company Security"
  template_id: 6118378
  subject: "Password Reset Successful"

# Password Change Notification
api_crmCommunicationUserPasswordChange:
  from_email: "security@yourcompany.com"
  from_name: "Your Company Security"
  template_id: 6118423
  subject: "Password Changed"

# Email Verification
api_crmCommunicationEmailVerification:
  from_email: "verify@yourcompany.com"
  from_name: "Your Company Verification"
  template_id: 6267350
```

```yaml
    subject: "Verify Your Email Address"

  # Balance Expired Notification
  api_crmCommunicationsBalanceExpired:
    from_email: "alerts@yourcompany.com"
    from_name: "Your Company Alerts"
    template_id: 7238252
    subject: "Service Balance Expired"

  # Low Balance Warning
  api_crmCommunicationsBalanceLow:
    from_email: "alerts@yourcompany.com"
    from_name: "Your Company Alerts"
    template_id: 7238263
    subject: "Low Balance Warning"
```

Creating Mailjet Templates:

1. Log in to Mailjet dashboard (<https://app.mailjet.com>)
2. Navigate to **Transactional** → **Templates**
3. Create a new template or clone an existing one
4. Note the **Template ID** (numeric value)
5. Add template variables matching the OmniCRM data structure
6. Update `crm_config.yaml` with the template ID

Available Template Variables:

Each email type receives specific variables. Common examples:

- `{{customer_name}}` - Customer or user name
- `{{service_name}}` - Service or product name
- `{{invoice_id}}` - Invoice number
- `{{invoice_amount}}` - Total invoice amount
- `{{due_date}}` - Payment due date
- `{{reset_link}}` - Password reset URL
- `{{verification_link}}` - Email verification URL
- `{{balance}}` - Current account balance
- `{{expiry_date}}` - Balance or service expiry date

Provisioning Configuration

```yaml
provisioning:
  failure_list: ['admin@yourcompany.com', 'ops@yourcompany.com']
```

Purpose:

- `failure_list` - Email addresses notified when Ansible provisioning fails
- Notifications include playbook name, error details, and customer information

- Allows ops team to quickly respond to provisioning issues

Invoice Configuration

```yaml
invoice:
  template_filename: 'yourcompany_invoice_template.html'
```

Purpose:

Specifies which Jinja2 HTML template to use for PDF invoice generation.

Template Location: `/OmniCRM-API/invoice_templates/`

See **Invoice PDF Generation** section below for details on creating custom templates.

CRM Base URL

```yaml
crm:
  base_url: 'http://localhost:5000'
```

Purpose:

- Used by Ansible playbooks to make API callbacks
- Used in email templates for generating links to the CRM
- Should be the publicly accessible URL of your API (not internal container names)

Examples:

- Development: `http://localhost:5000`
- Production: `https://api.yourcompany.com`
- Docker: `http://omnicrm-api:5000` (internal container communication)

OCS and CGRates Configuration

```yaml
ocs:
  ocsApi: 'http://10.179.2.142:8080/api'
  ocsTenant: 'mnc380.mcc313.3gppnetwork.org'
  cgrates: 'localhost:2080'
```

Configuration:

- `ocsApi` - OCS API endpoint for subscriber management
- `ocsTenant` - Tenant identifier for multi-tenant OCS deployments
- `cgrates` - CGRates JSON-RPC API endpoint (host:port)

SMSC Configuration (SMS Gateway)

```yaml
smsc:
  source_msisdn: 'YourCompany'
  smsc_url: 'http://10.179.2.216/SMSc/'
  api_key: 'your_smsc_api_key'
```

Purpose:

- Send SMS notifications to customers (low balance, service alerts, 2FA codes)
- `source_msisdn` - Sender ID displayed to recipients (alphanumeric or phone number)
- `smsc_url` - SMSC gateway API endpoint
- `api_key` - Authentication for SMSC API

JWT Secret Key

```yaml
jwt_secret:
'2b93110f723db60172c8e9a1eaa80027a9a9c3f05b44e50dc3fcf38dba68d87e'
```

Security:

- Used to sign and verify authentication tokens
- **MUST be changed from default value** in production
- Generate a secure random string (minimum 64 characters)
- Never share or commit to version control

Generating a New JWT Secret:

```
# Generate a cryptographically secure random key
python3 -c "import secrets; print(secrets.token_hex(32))"
```

Stripe Payment Configuration

```yaml
stripe:
  secret_key: 'sk_live_xxxxxxxxxx'
  publishable_key: 'pk_live_xxxxxxxxxx'
  currency: 'aud'
  statement_descriptor_suffix: 'YOURCOMPANY'
```

Configuration:

- `secret_key` - Stripe secret API key (server-side, keep confidential)
- `publishable_key` - Stripe publishable key (client-side, safe to expose)
- `currency` - ISO 4217 currency code (aud, usd, gbp, eur, etc.)
- `statement_descriptor_suffix` - Appears on customer credit card statements

Statement Descriptor Usage:

- Shown on customer bank statements as "YOURCOMPANY"
- Maximum 22 characters
- Helps customers identify charges
- Also used in invoice PDF filenames (e.g., `YOURCOMPANY_12345.pdf`)

API Keys and IP Whitelisting

Define API keys with role-based access and IP restrictions:

```yaml
api_keys:
  "YOUR_API_KEY_1":
    roles: ["admin"]
    ips: ["127.0.0.1", "::1"]
  "YOUR_API_KEY_2":
    roles: ["customer_service_agent_1"]
    ips: ["127.0.0.1", "::1", "10.0.1.0/24"]

# IP Whitelist (standalone, without API key)
ip_whitelist:
  "10.179.2.142":
    roles: ["admin"]
```

Purpose:

- Allow external systems to authenticate via API key
- Restrict access by IP address
- Grant specific roles to API consumers
- Useful for integrations (billing systems, monitoring, automation)

Security Best Practices:

- Use long, random API keys (minimum 32 characters)
- Restrict IPs to known sources only
- Grant minimum necessary roles
- Rotate API keys regularly
- Monitor API key usage in logs

Invoice PDF Generation

OmniCRM generates professional PDF invoices using Jinja2 HTML templates and WeasyPrint PDF rendering.

How PDF Generation Works

1. **Template Selection:**
    - Template filename is specified in `crm_config.yaml` under

invoice.template_filename
    - Template is loaded from `/OmniCRM-API/invoice_templates/` directory
2. **Data Preparation:**
    - Invoice data (ID, dates, amounts, status) is fetched from database
    - Customer information (name, address) is retrieved
    - Transaction list is compiled (all charges/credits on the invoice)
3. **Template Rendering:**
    - Jinja2 renders the HTML template with invoice data
    - Variables like `{{ invoice_number }}`, `{{ total_amount }}`, etc. are replaced
    - Rendered HTML is saved to `invoice_templates/rendered/` for debugging
4. **PDF Generation:**
    - WeasyPrint converts rendered HTML to PDF
    - PDF supports CSS styling, images, page breaks, headers/footers
    - PDF binary data is generated in memory
5. **Caching:**
    - PDF is Base64-encoded and stored in `Invoice_PDF_Cache` table
    - SHA256 hash is calculated for integrity verification
    - Subsequent requests return cached PDF (instant delivery)
6. **Cache Invalidation:**
    - Cache is invalidated when invoice is modified, voided, or refunded
    - Template changes do not automatically invalidate existing caches

Invoice Template Structure

Invoice templates are Jinja2 HTML files with embedded variables and logic.

Template Location: `/OmniCRM-API/invoice_templates/yourcompany_invoice_template.html`

Available Variables:

```
{
    'invoice_number': 12345,
    'date': '2025-01-04',
    'client': {
        'name': 'John Smith',
        'address': {
            'address_line_1': '123 Main St',
            'city': 'Sydney',
            'state': 'NSW',
            'zip_code': '2000',
            'country': 'Australia'
        }
    },
    'transaction_list': [
```

```
        [
            {
                'transaction_id': 1,
                'title': 'Mobile Service - Monthly Fee',
                'retail_cost': 30.00,
                'wholesale_cost': 10.00,
                'created': '2025-01-01'
            },
            {
                'transaction_id': 2,
                'title': 'Data Addon - 5GB',
                'retail_cost': 15.00,
                'wholesale_cost': 5.00,
                'created': '2025-01-15'
            }
        ]
    ],
    'total_amount': 45.00,
    'due_date': '2025-01-31',
    'start_date': '2025-01-01',
    'end_date': '2025-01-31',
    'paid': False,
    'void': False
}
```

Example Template Snippet:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Invoice {{ invoice_number }}</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 40px;
        }
        .header {
            text-align: center;
            margin-bottom: 30px;
        }
        .invoice-details {
            margin-bottom: 20px;
        }
        table {
            width: 100%;
            border-collapse: collapse;
        }
        th, td {
```

```
            border: 1px solid #ddd;
            padding: 12px;
            text-align: left;
        }
        th {
            background-color: #405189;
            color: white;
        }
        .total {
            text-align: right;
            font-size: 18px;
            font-weight: bold;
            margin-top: 20px;
        }
    </style>
</head>
<body>
    <div class="header">
        <img src="../absolute/path/to/your-logo.png" alt="Company
Logo" height="60">
        <h1>INVOICE</h1>
    </div>

    <div class="invoice-details">
        <p><strong>Invoice Number:</strong> {{ invoice_number }}</p>
        <p><strong>Date:</strong> {{ date }}</p>
        <p><strong>Due Date:</strong> {{ due_date }}</p>
        <p><strong>Billing Period:</strong> {{ start_date }} to {{
end_date }}</p>
    </div>

    <div class="customer-details">
        <h3>Bill To:</h3>
        <p>{{ client.name }}</p>
        <p>{{ client.address.address_line_1 }}</p>
        <p>{{ client.address.city }}, {{ client.address.state }} {{
client.address.zip_code }}</p>
        <p>{{ client.address.country }}</p>
    </div>

    <table>
        <thead>
            <tr>
                <th>Description</th>
                <th>Date</th>
                <th>Amount</th>
            </tr>
        </thead>
```

```
        <tbody>
            {% for transaction in transaction_list[0] %}
            <tr>
                <td>{{ transaction.title }}</td>
                <td>{{ transaction.created }}</td>
                <td>${{ "%.2f"|format(transaction.retail_cost)
}}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>

    <div class="total">
        <p>Total Amount Due: ${{ "%.2f"|format(total_amount) }}</p>
    </div>

    {% if paid %}
    <div style="text-align: center; color: green; font-weight:
bold;">
        PAID
    </div>
    {% endif %}

    {% if void %}
    <div style="text-align: center; color: red; font-weight: bold;">
        VOID
    </div>
    {% endif %}
</body>
</html>
```

Template Best Practices:

- **Use absolute paths for images** - `file:///absolute/path/to/image.png`
- **Inline CSS** - WeasyPrint doesn't load external stylesheets reliably
- **Test with sample data** - Use `invoice_templates/rendered/` to inspect HTML
- **Page breaks** - Use `<div style="page-break-after: always;"></div>` for multi-page invoices
- **Headers and footers** - Use `@page` CSS rules for repeating elements
- **Currency formatting** - Use Jinja2 filters: `{{ "%.2f"|format(amount) }}`

Creating a Custom Invoice Template

1. **Copy Example Template:**

```
cd /OmniCRM/OmniCRM-API/invoice_templates
cp norfone_invoice_template.html
```

```
yourcompany_invoice_template.html
```

2. **Edit Template:**

    - Replace company name, logo, contact information
    - Adjust styling (colors, fonts, layout) to match brand
    - Add or remove sections as needed (tax breakdowns, payment instructions, etc.)

3. **Update Configuration:**

    Edit `crm_config.yaml`:

    ```yaml
    invoice:
      template_filename: 'yourcompany_invoice_template.html'
    ```

4. **Test Invoice Generation:**

    - Create a test invoice in the CRM
    - Download the PDF and verify formatting
    - Check `invoice_templates/rendered/{invoice_id}.html` for debugging

5. **Invalidate Old Caches (if needed):**

    If you've changed the template and want to regenerate existing invoices:

    ```sql
    -- Clear all cached PDFs (forces regeneration)
    DELETE FROM Invoice_PDF_Cache;
    ```

PDF Caching System

To improve performance, OmniCRM caches generated PDFs:

Cache Behavior:

- **First Request** - PDF is generated, cached, and returned
- **Subsequent Requests** - Cached PDF is returned immediately (no regeneration)
- **Cache Invalidation** - Occurs when invoice is modified, voided, or refunded
- **Cache Cleanup** - Old caches are automatically purged after 30 days of inactivity

Cache Storage:

- Base64-encoded PDF stored in `Invoice_PDF_Cache` table
- SHA256 content hash for integrity verification
- Includes filename, creation timestamp, last accessed timestamp

Manual Cache Management:

```python
# In OmniCRM API or Python shell
from services.invoice_service import cleanup_old_pdf_cache,
invalidate_invoice_cache
from utils.db_helpers import get_db_session

session = get_db_session()

# Clean up caches older than 30 days
result = cleanup_old_pdf_cache(session, days_old=30)
print(result)  # {'status': 'success', 'deleted_count': 15}

# Invalidate specific invoice cache
invalidate_invoice_cache(session, invoice_id='12345')
```

API Endpoints:

Generate/download invoice PDF:

GET /invoice/pdf/{invoice_id}

Response: PDF file download with filename from Stripe statement descriptor

Cache Headers:

- First request: Slower response (generation time)
- Cached requests: Instant response
- Cache hit/miss is transparent to the user

Troubleshooting

PDF Generation Fails:

- Check that WeasyPrint is installed: `pip install weasyprint`
- Verify template filename matches `crm_config.yaml`
- Check `invoice_templates/rendered/` for HTML rendering errors
- Review API logs for Jinja2 template errors

Images Not Appearing in PDF:

- Use absolute file paths: `file:///full/path/to/image.png`
- Ensure image files exist and are readable
- Check image format (PNG and JPEG work best)
- Verify image paths don't contain special characters

Styling Issues:

- Inline all CSS (external stylesheets not supported)

- Avoid complex CSS features (flexbox, grid may not render correctly)
- Test with simple layouts first, add complexity gradually
- Use tables for layout instead of divs where possible

Cache Not Invalidating:

- Verify `invalidate_invoice_cache()` is called when invoice is modified
- Check that transaction updates trigger cache invalidation
- Manually delete from `Invoice_PDF_Cache` table if needed

Configuration Checklist

Use this checklist when deploying OmniCRM:

Backend Configuration

- [ ] Copy `.env.example` to `.env`
- [ ] Set strong database passwords
- [ ] Configure CGRates credentials
- [ ] Update `crm_config.yaml` with your settings:
    - [ ] Database connection
    - [ ] Service types
    - [ ] Mailjet API keys and template IDs
    - [ ] Provisioning failure notification emails
    - [ ] Invoice template filename
    - [ ] CRM base URL (publicly accessible)
    - [ ] OCS/CGRates endpoints
    - [ ] SMSC configuration
    - [ ] Generate new JWT secret key
    - [ ] Stripe keys (live, not test)
    - [ ] API keys and IP whitelisting

Frontend Configuration

- [ ] Copy `OmniCRM-UI/.env.example` to `OmniCRM-UI/.env`
- [ ] Set Google Maps API key
- [ ] Set Stripe publishable key
- [ ] Update company branding:
    - [ ] Company name
    - [ ] Portal name
    - [ ] Self-care name
    - [ ] Company tagline
- [ ] Configure localization:
    - [ ] Default language
    - [ ] Locale
    - [ ] Default location and country
    - [ ] Currency code and symbol
- [ ] Set primary brand color

- [ ] Configure web app integrations (optional)
- [ ] Add support and FAQ URLs (optional)

Branding Assets

- [ ] Create logo files (logoSm.png, logoDark.png, logoLight.png)
- [ ] Upload logos to `OmniCRM-UI/src/assets/images/omnitouch/`
- [ ] Create custom invoice template HTML
- [ ] Upload invoice template to `OmniCRM-API/invoice_templates/`
- [ ] Update `crm_config.yaml` with invoice template filename
- [ ] Test invoice PDF generation
- [ ] Rebuild UI: `npm run build`

Security

- [ ] Change all default passwords
- [ ] Generate unique JWT secret
- [ ] Use production Stripe keys (not test keys)
- [ ] Rotate Mailjet API keys
- [ ] Enable firewall rules
- [ ] Configure IP whitelisting for API access
- [ ] Set up SSL/TLS certificates
- [ ] Enable HTTPS for all endpoints
- [ ] Review CORS settings
- [ ] Implement rate limiting
- [ ] Configure backup and recovery procedures

Testing

- [ ] Test customer registration flow
- [ ] Test service provisioning end-to-end
- [ ] Verify email notifications are sent correctly
- [ ] Test invoice generation and PDF download
- [ ] Verify payment processing (Stripe)
- [ ] Check user authentication and 2FA
- [ ] Test impersonation and audit logging
- [ ] Verify usage data syncs from OCS
- [ ] Test ActionPlan creation and renewal
- [ ] Confirm inventory allocation works correctly

Deployment

- [ ] Build Docker images or deploy to servers
- [ ] Start database containers (MySQL, PostgreSQL)
- [ ] Start CGRates
- [ ] Start OmniCRM API
- [ ] Start OmniCRM UI
- [ ] Configure reverse proxy (nginx, traefik)

- [ ] Set up monitoring (Grafana, Prometheus)
- [ ] Configure log aggregation
- [ ] Set up automated backups
- [ ] Document deployment architecture
- [ ] Train staff on system usage

Related Documentation

- `RBAC and User Management </rbac>`
- `Products and Services </concepts_products_and_services>`
- `Ansible Provisioning </concepts_ansible>`
- `Inventory Management </administration_inventory>`
- `Customer Invoices </payments_invoices>`
- `Two-Factor Authentication </2fa>`
- `Customer Care and Impersonation </customer_care>`

Inventory Overview in OmniCRM

The **Inventory** system in OmniCRM is designed to manage and track both physical and virtual items used by network operators and customers.

This means we can track all sorts of items, such as modems, phone numbers, IP address blocks, or even physical hardware like GPON ONTs or Fixed Wireless CPEs.

See also: Customer Attributes <administration_attributes> for storing custom metadata, and Customer Tags <administration_tags> for visual categorization.

To support customers with a fixed network footprint, the inventory system can also track homes passed for a given service, allowing operators to do service qualification remotely, and for those operating a fixed wireless network, we can track the CPEs deployed in the field with their locations.

Note

Inventory items are linked to products during provisioning through the inventory_items_list field. For a complete walkthrough of how inventory integrates with product provisioning, including the inventory picker UI and Ansible playbook integration, see Complete Product Lifecycle Guide - Inventory Requirements <guide_product_lifecycle>.

Purpose

The OmniCRM Inventory serves several key purposes:

1. **Provisioning Services**: When a customer signs up for a service, items like modems, SIM cards, or phone numbers may need to be allocated. The inventory system tracks these items and associates them with customers.
2. **Stock Management**: For physical stock, such as hardware or other equipment, the inventory helps operators maintain visibility into what is available, where it is stored, and what has been allocated or sold to customers.
3. **Customer Allocation**: The system allows for items to be allocated to customers, whether for use in a service (e.g., assigning a modem to a customer's internet account) or for direct sale.
4. **Service Qualification / Network Footprint**: By storing information about network footprint, such as each home passed for a GPON service, or each Fixed Wireless CPE deployed, allows staff to do service qualification remoteley and see if there is a netowrk footprint in a specific area.

Example Inventory Lifecycle

To illustrate how the Inventory system works, consider the following examples

SIM Card Example

A batch of 1000 SIM cards are ordered from Omnitouch. Firstly an Inventory Template is created for SIM cards (if it doesn't already exist) and the ordered SIMs are loaded into the Inventory in the state In Transit.

Once the SIMs are received, they are marked as In Stock, and can be allocated out to different retail stores, with the Location of the inventory item updated to reflect which retail store each SIM card is at - This is useful for tracking stock levels at each store and ensuring that each store has enough stock to meet customer demand.

When a customer signs up for a mobile service in-store, a SIM card is allocated to the customer and the status is changed to Allocated. The SIM card inventory item is then assigned to the customer, and the status is updated to In Use.

If the customer cancels the service or the service goes dormant, the SIM card is marked as Decommissioned.

GPON Homes Passed Example

For a GPON network, the inventory system can track each home passed for a given service.

When a new area is built out, each address passed can be added to the inventory.

This allows operators to see which homes are passed for a given service, and which homes are not yet passed.

When a customer signs up for a service, OmniCRM can automatically run a service qualification against the customer's address, to see if the address is in the homes passed inventory and what services can be offered.

Inventory Templates

The **InventoryTemplate** feature enables the creation of any number of item types with predefined fields. These templates act as blueprints that define the essential characteristics of different items, such as:

- **Modems** with a MAC address (`itemtext1`) and a serial number (`itemtext2`).
- **Homes Passed** for a given service, with a location and status (e.g., passed or not passed).

- **Phone Numbers** with a primary number (`itemtext1`) and a geographic location (`itemtext2`).
- **Virtual Resources** like IP address blocks, with identifiers mapped through the template.

Each inventory template defines up to 20 customizable text fields (`itemtext1` through `itemtext20`) with corresponding labels (`itemtext1_label` through `itemtext20_label`) that describe what each field represents. For example, a Modem template might set `itemtext1_label` to "MAC Address" and `itemtext2_label` to "Serial Number".

Operators can customize the fields for each item type using **InventoryTemplates**. These templates allow items to be categorized and managed in a structured way, ensuring consistency in how items are tracked.

Linking to Products:

Inventory template names are referenced in product definitions via the `inventory_items_list` field. When provisioning a product, the system displays an inventory picker showing only items matching the required template types.

Example: A product with `inventory_items_list: "['SIM Card', 'Mobile Number']"` requires two inventory templates named exactly "SIM Card" and "Mobile Number" to exist. Template names are case-sensitive.

For complete details on how inventory templates connect to product provisioning, see `Product Lifecycle - Inventory Requirements <guide_product_lifecycle>`.

Viewing templates in OmniCRM

Creating Inventory Templates via the UI

To create a new inventory template:

1. Navigate to **Inventory → Templates** from the main menu

2. Click the **Add Template** button

3. Fill in the required fields:

   Basic Information:

   - **Icon** (optional): Icon class name for visual identification (e.g., `fa-solid fa-sim-card`)
   - **Item** (required): The template name (must match exactly what's used in `inventory_items_list` for products)

   Cost Information (Required):

- **Wholesale Cost** (required): Your cost to purchase or provision this item type
- **Retail Cost** (required): Standard retail price if sold separately to customers

Note

Wholesale and retail costs set here serve as default values when creating new inventory items from this template. Individual inventory items can have different costs if needed.

Field Labels:

- **Item Text 1 Label** (required): Label for the first customizable field (defaults to "Model Number")
  - Common examples: "ICCID" for SIM cards, "MAC Address" for modems, "Phone Number" for numbers
- **Item Text 2 Label** (required): Label for the second customizable field (defaults to "Serial Number")
  - Common examples: "IMSI" for SIM cards, "Serial Number" for hardware, "Geographic Region" for numbers
- **Item Text 3-20 Labels** (optional): Additional field labels as needed
  - Click **Add Field** to add more custom fields
  - Only add fields you will actually use for this item type

Visibility Settings:

- **Allow Dropdown Staff**: Enable staff to select this inventory type in dropdowns
- **Allow Dropdown Customer**: Enable customers to see/select this inventory type (customer portal)

4. Click **Save** to create the template

Editing Inventory Templates

To edit an existing template:

1. Navigate to **Inventory → Templates**
2. Find the template in the list
3. Click the **Edit** button
4. Modify fields as needed
5. Click **Save**

Warning

Changing field labels (e.g., `itemtext1_label`) only affects new items created after the change. Existing inventory items keep their data but will display with the new label names.

Caution

Template names referenced in product `inventory_items_list` fields are case-sensitive. Renaming a template will break the link to products using the old name.

Common Template Examples

SIM Card Template:

- Item: "SIM Card"
- Wholesale Cost: 2.50
- Retail Cost: 10.00
- Item Text 1 Label: "ICCID"
- Item Text 2 Label: "IMSI"
- Item Text 3 Label: "SIM Type" (Physical/eSIM)

Mobile Number Template:

- Item: "Mobile Number"
- Wholesale Cost: 1.00
- Retail Cost: 0.00
- Item Text 1 Label: "Phone Number"
- Item Text 2 Label: "Geographic Region"
- Item Text 3 Label: "Number Type" (Mobile/Landline)

Fixed Wireless CPE Template:

- Item: "Fixed Wireless CPE"
- Wholesale Cost: 250.00
- Retail Cost: 450.00
- Item Text 1 Label: "MAC Address"
- Item Text 2 Label: "Serial Number"
- Item Text 3 Label: "Firmware Version"
- Item Text 4 Label: "Manufacturer"
- Item Text 5 Label: "Model"

GPON ONT Template:

- Item: "GPON ONT"
- Wholesale Cost: 45.00
- Retail Cost: 0.00 (included with service)
- Item Text 1 Label: "Serial Number"
- Item Text 2 Label: "MAC Address"
- Item Text 3 Label: "PON Location"
- Item Text 4 Label: "Model"

Creating an Item from a Template in OmniCRM

Creating and Managing Inventory Items

Once an **InventoryTemplate** is defined, individual **Inventory** items can be created. Each inventory item represents a specific instance of an item type (e.g., a specific modem or phone number) that can be:

- **Allocated to Customers**: Items are linked to customers for service provisioning (e.g., assigning hardware for an internet connection).
- **Tracked for Stock**: Operators can monitor available inventory, such as unsold or unassigned items.
- **Sold or Decommissioned**: Once sold, items are marked with relevant timestamps (e.g., `sold_date`) and can no longer be considered available stock.

Inventory List View Edit Inventory Item

Through this system, OmniCRM facilitates efficient stock management, helps allocate resources to customers, and provides detailed visibility into the status and history of every item.

Services can be linked to a given **Inventory** item, allowing for easy tracking of which items are associated with which customers or services.

Once an inventory item has been assigned to a customer, the Ansible plays can update the item's status and history to reflect the allocation. This ensures that operators have an accurate record of which items are in use and which are available for allocation, as well as knowing which customer is using which item.

We can view items allocated to a customer from the customer's profile page in the **Inventory** tab.

Customer Inventory View

For Inventory items linked to a **Service**, we can see that by editing the service, to see the linked Inventory items.

Viewing Inventory Items linked to a Service in OmniCRM

Inventory Item Fields

Each inventory item contains comprehensive information organized into several categories:

Basic Item Information

- **inventory_id** - Unique identifier for the inventory item (auto-generated)
- **item** - Type of item (matches Inventory Template name, e.g., "SIM Card", "Modem", "Phone Number")

- **inventory_template_id** - Link to the Inventory Template that defines this item type
- **customer_id** - If assigned to a customer, the customer's ID (nullable)
- **service_id** - If linked to a specific service, the service ID (nullable)
- **item_location** - Physical or logical location of the item:
  - For physical items: building, warehouse, shelf location, store name, etc.
  - For virtual items: geographic location, IP block location, number range region, etc.
- **item_state** - Current state of the inventory item (enumerated values):
  - **New** - Brand new, unused item
  - **Used** - Previously used but functional
  - **Internal Use** - Allocated for internal testing or staff use
  - **Assigned** - Currently assigned to a customer or service
  - **Damaged** - Non-functional, requires repair or disposal
  - **Out Of Service** - Temporarily unavailable
  - **Lost** - Item cannot be located
  - **Stolen** - Item was stolen

Customizable Item Fields (from Template)

The inventory system supports up to 20 customizable text fields whose meaning is defined by the Inventory Template:

- **itemtext1** - First customizable field (required, label defined by template's `itemtext1_label`)
  - Example: For modems, might be "MAC Address"
  - Example: For SIM cards, might be "ICCID"
  - Example: For phone numbers, might be "Phone Number"
- **itemtext2** through **itemtext20** - Additional customizable fields (optional, labels defined by template)
  - Example: itemtext2 for modems might be "Serial Number"
  - Example: itemtext2 for SIM cards might be "IMSI"
  - Example: itemtext3 for modems might be "Firmware Version"

Each Inventory Template defines which of these fields are used and what they represent via the corresponding label fields (`itemtext1_label`, `itemtext2_label`, etc.).

Cost Information

- **wholesale_cost** - Your cost to purchase/provision this item (float)
- **retail_cost** - Price charged to customer if sold separately (float)
- **sold_date** - Timestamp when item was sold or assigned to customer

Physical Address (for Network Equipment and Sites)

Used for tracking physical deployment locations, particularly for fixed network equipment (CPEs, ONTs, modems) or homes passed:

- **address_line_1** - Street address, building number, unit number
- **address_line_2** - Additional address information (suite, apartment, floor)
- **city** - City or town
- **state** - State, province, or region
- **zip_code** - Postal/ZIP code
- **country** - Country name

Geographic Location (Auto-populated from Web UI)

When creating inventory items via the Web UI with address autocomplete, these fields are automatically populated:

- **google_maps_place_id** - Google Maps Place ID for the address
- **plus_code** - Google Maps Plus Code (Open Location Code) for precise location
- **latitude** - Geographic latitude (decimal degrees)
- **longitude** - Geographic longitude (decimal degrees)

These fields enable:

- Mapping inventory locations on a map view
- Proximity calculations for service qualification
- Coverage analysis for network planning
- Route optimization for field technician dispatching

Device Management and Access URLs

The `management_url` field provides quick access to device interfaces and provisioning URLs:

- **management_url** - Access URL for the inventory item
    - **Network Equipment**: Web interface URL (e.g., `https://192.168.1.1` for routers, switches, ONTs, CPEs)
    - **eSIM Profiles**: LPA (Local Profile Assistant) address for eSIM activation (e.g., `LPA:1$smdp.example.com$ACTIVATION-CODE-HERE`)
    - **Other Use Cases**: Any URL that needs to be easily accessible via mobile device

    QR Code Generation

    When viewing inventory items with a `management_url`, the system automatically generates a **scannable QR code**:

    - **Inventory item detail view**: 128x128 QR code displayed alongside the URL
    - **Service inventory table**: 64x64 QR code shown for assigned items
    - **Format**: Both QR code and clickable hyperlink displayed together

Common Use Cases

    - **Network Technicians**: Scan QR code to access device management interface without typing IP addresses
    - **eSIM Activation**: Customers scan QR code from the CRM to install eSIM profile on their device
    - **Customer Self-Service**: Provide easy access to device configuration or customer portals

- **management_username** - Admin username for device access

- **management_password** - Admin password for device access (encrypted at rest)

Configuration Management

For devices with configuration files:

- **config_content** - Complete configuration file content (stored as text)
    - Useful for backup, versioning, and disaster recovery
    - Can store router configs, switch configs, CPE configs, etc.
- **config_file_path** - Path to external configuration file if stored separately
    - Alternative to storing full config in database
    - Path to network share, version control repository, or config management system

Notes and Metadata

- **inventory_notes** - Free-form notes about the inventory item
    - Installation notes
    - Maintenance history
    - Quirks or special handling requirements
    - Vendor information
    - Warranty details
- **created** - Timestamp when inventory item was created in the system (auto-set)
- **last_modified** - Timestamp of last update to the inventory item (auto-updated)

Field Usage Examples

Example 1: Mobile SIM Card

```json
{
  "inventory_id": 1001,
  "item": "SIM Card",
  "inventory_template_id": 5,
  "itemtext1": "8961234567890123456",
```

```json
  "itemtext2": "310120123456789",
  "itemtext3": "Physical",
  "item_location": "Warehouse A, Shelf 3",
  "item_state": "Assigned",
  "customer_id": 456,
  "service_id": 789,
  "wholesale_cost": 2.50,
  "retail_cost": 10.00,
  "sold_date": "2025-01-15T10:30:00Z",
  "inventory_notes": "Activated on 2025-01-15"
}
```

Example 2: Mobile eSIM Profile

```json
{
  "inventory_id": 1002,
  "item": "eSIM",
  "inventory_template_id": 6,
  "itemtext1": "8961234567890123457",
  "itemtext2": "310120123456790",
  "itemtext3": "eSIM",
  "item_location": "Virtual Inventory",
  "item_state": "Assigned",
  "customer_id": 457,
  "service_id": 790,
  "management_url": "LPA:1$smdp.example.com$ACTIVATION-CODE-
ABC123XYZ",
  "wholesale_cost": 0.00,
  "retail_cost": 0.00,
  "sold_date": "2025-01-16T14:20:00Z",
  "inventory_notes": "eSIM profile ready for activation"
}
```

When viewing this eSIM inventory item, the UI displays a QR code containing the LPA address. Customers scan this QR code with their mobile device to install the eSIM profile.

Example 3: Customer Premises Equipment (CPE) - Fixed Wireless

```json
{
  "inventory_id": 2001,
  "item": "Fixed Wireless CPE",
  "inventory_template_id": 10,
  "itemtext1": "AA:BB:CC:DD:EE:FF",
  "itemtext2": "FW2024-12345",
  "itemtext3": "v2.4.1",
  "itemtext4": "Ubiquiti",
  "itemtext5": "LiteBeam AC Gen2",
  "item_location": "Customer Site",
```

```json
  "item_state": "Assigned",
  "customer_id": 789,
  "service_id": 1234,
  "address_line_1": "123 Main Street",
  "address_line_2": "Apt 4B",
  "city": "Sydney",
  "state": "NSW",
  "zip_code": "2000",
  "country": "Australia",
  "latitude": "-33.8688",
  "longitude": "151.2093",
  "management_url": "https://192.168.100.1",
  "management_username": "admin",
  "management_password": "encrypted_password_here",
  "config_file_path": "/configs/cpe/fw2024-12345.conf",
  "inventory_notes": "Installed 2025-01-10. Customer reports
excellent signal. Pointing: Azimuth 45°, Elevation 15°"
}
```

Example 3: GPON ONT with Full Address

```json
{
  "inventory_id": 3001,
  "item": "GPON ONT",
  "inventory_template_id": 15,
  "itemtext1": "ALCL12345678",
  "itemtext2": "AA:BB:CC:DD:EE:FF",
  "itemtext3": "OLT-1, PON 3, ONT 42",
  "itemtext4": "Nokia G-010G-A",
  "item_location": "Customer Premises",
  "item_state": "Assigned",
  "customer_id": 321,
  "service_id": 654,
  "address_line_1": "456 Fiber Lane",
  "city": "Melbourne",
  "state": "VIC",
  "zip_code": "3000",
  "country": "Australia",
  "google_maps_place_id": "ChIJ1234567890",
  "plus_code": "4RRH+2C Melbourne VIC",
  "latitude": "-37.8136",
  "longitude": "144.9631",
  "management_url": "https://192.168.1.1",
  "management_username": "admin",
  "config_content": "# ONT Configuration\nwlan-ssid: HomeNetwork\
nwlan-password: encrypted...",
  "wholesale_cost": 45.00,
  "retail_cost": 0.00,
  "inventory_notes": "Provisioned 2025-01-20. Optical power: -22dBm"
```

```
}
```

Note

When viewing inventory items with a `management_url` (like Examples 2, 3, and 4 above), the UI automatically displays:

- A scannable QR code containing the URL or LPA address
- A clickable hyperlink (for web URLs)

Use Cases:

- **eSIM Activation** (Example 2): Customers scan the QR code to install the eSIM profile on their device
- **Network Equipment Access** (Examples 3 & 4): Technicians scan to access device management interfaces without manually typing IP addresses

Example 5: Phone Number (Virtual Inventory)

```json
{
  "inventory_id": 4001,
  "item": "Phone Number",
  "inventory_template_id": 20,
  "itemtext1": "+61412345678",
  "itemtext2": "Melbourne",
  "itemtext3": "Mobile",
  "item_location": "Australia - VIC",
  "item_state": "Assigned",
  "customer_id": 555,
  "service_id": 888,
  "wholesale_cost": 1.00,
  "retail_cost": 0.00,
  "inventory_notes": "Ported from Telstra on 2025-01-05"
}
```

Inventory Item States Explained

The `item_state` field tracks the lifecycle of inventory items:

- **New → Assigned** - Normal flow when provisioning to a customer
- **Assigned → Used** - After service deactivation, item can be reused
- **New → Internal Use** - Allocated for testing, demos, or staff use
- **Assigned → Damaged** - Item failed, requires RMA or disposal
- **Any State → Lost** - Item cannot be located (triggers audit)
- **Any State → Stolen** - Item was stolen (triggers security report)
- **Damaged/Used → New** - After refurbishment or repair

Filtering and searching inventory by state allows operators to:

- Track available stock (New items)
- Identify items assigned to customers (Assigned)
- Find items available for reuse (Used)
- Monitor equipment issues (Damaged, Out Of Service)
- Audit missing items (Lost, Stolen)

Customer Tags

Tags are handy colour-coded links that can be added to a customer to help categorize them, for example, a customer might have a tag for "Open Support Ticket" or "Super Late Invoice" or "Jerk".

For storing structured metadata and custom key-value data, see Customer Attributes <administration_attributes>.

The tags are displayed as pills on the customer's profile page, and the colour of the pill is customizable along with the link.

One common use case is to tag customers who have an open support ticket, so that the support team can easily jump to the open ticket from the customer's profile page.

Tags can be created in the system by an administrator through the UI or by 3rd party systems via the API and can have start and end dates, so they can be automatically removed after a certain period.

Tags

Managing Tags via the UI

Viewing Customer Tags

To view tags for a customer:

1. Navigate to the customer's overview page
2. Click on the **Tags** tab
3. You will see a list of all active tags for the customer, showing:
    - Tag preview with the configured color
    - Tag text
    - Active date (when the tag becomes visible)
    - Deactivate date (when the tag will be hidden)
    - Link (if configured)

Creating a New Tag

To create a new tag for a customer:

1. Navigate to the customer's overview page
2. Click on the **Tags** tab
3. Click the **Add Tag** button

4. Fill in the required fields:
    - **Tag Text** (required): The text that will be displayed on the tag
    - **Tag Color** (required): Choose a color using the color picker or enter a hex code
    - **Tag Link** (optional): URL that will open when the tag is clicked
    - **Active Date** (required): Date when the tag should start being displayed
    - **Deactivate Date** (required): Date when the tag should stop being displayed (defaults to 2099-01-01)
5. Preview your tag in the preview section
6. Click **Create Tag**

Create Customer Tag Form Customer Tags Display

Editing a Tag

To edit an existing tag:

1. Navigate to the customer's overview page
2. Click on the **Tags** tab
3. Find the tag you want to edit in the list
4. Click the **Edit** (pencil) button
5. Modify the fields as needed
6. Click **Update Tag**

Deleting a Tag

To delete a tag:

1. Navigate to the customer's overview page
2. Click on the **Tags** tab
3. Find the tag you want to delete in the list
4. Click the **Delete** (trash) button
5. Confirm the deletion in the popup

Tag Field Reference

API Integration

Tags can also be managed programmatically via the API:

Create a Tag:

```
PUT /crm/tag/
{
  "tag_text": "VIP Customer",
  "tag_hex_color": "FFD700",
```

```
  "tag_link": "https://example.com/vip",
  "tag_active_date": "2025-01-01 00:00:00",
  "tag_deactivate_date": "2099-12-31 23:59:59",
  "customer_id": 12
}
```

Update a Tag:

```
PATCH /crm/tag/tag_id/{tag_id}
{
  "tag_text": "Updated Tag Text",
  "tag_hex_color": "FF0000"
}
```

Get Tags by Customer:

```
GET /crm/tag/customer_id/{customer_id}
```

Delete a Tag:

```
DELETE /crm/tag/tag_id/{tag_id}
```

OmniCRM System Architecture

This document provides an overview of the OmniCRM system architecture, including component relationships and data flow.

High-Level System Overview

OmniCRM is a comprehensive BSS/OSS platform that integrates several key components to provide complete service management for telecom providers.

Core Components

1. Frontend Applications

OmniCRM Web UI

- React single-page application
- Staff interface for customer management, service provisioning, billing
- Real-time provisioning status updates
- Role-based access control

Self-Care Portal

- Customer-facing portal
- Service management and usage tracking
- Invoice viewing and payment
- Shared codebase with staff UI, different views

2. API Layer

OmniCRM API (Flask/Python)

- RESTful API for all operations
- OpenAPI/Swagger documentation
- JWT-based authentication
- Rate limiting and caching
- WebSocket support for real-time updates

Key API Routes:

- `/crm/customer/*` - Customer management
- `/crm/service/*` - Service operations
- `/crm/product/*` - Product catalog

- `/crm/provision/*` - Provisioning operations
- `/crm/transaction/*` - Billing transactions
- `/crm/invoice/*` - Invoice management

3. Provisioning System

4. Billing & Rating

OCS/CGRateS Integration

- Real-time charging and rating
- Account management
- Balance tracking (monetary, data, voice, SMS)
- Action plans for recurring charges
- Threshold-based notifications

Billing Workflow:

5. Data Model

OmniCRM uses a relational database with the following core models. For visual entity relationship diagrams, see [Customers, Contacts, Sites & Services].

Customer & Related Entities

Customer - Central entity representing a company or individual

Field	Type	Description
customer_id	Integer (PK)	Unique identifier
customer_name	String	Company or individual name
customer_account_type	Enum	'Individual' or 'Business'
customer_status	Enum	'Open', 'Closed', 'Suspended', 'Archived'
customer_payment_type	String	'prepaid' or 'postpaid'
customer_enabled	Boolean	Is account active
tax_identifier	String	VAT/GST number
contract_start_date	DateTime	Contract start
contract_end_date	DateTime	Contract end

Contact - People associated with a customer

Field	Type	Description
contact_id	Integer (PK)	Unique identifier
customer_id	Integer (FK)	Parent customer
contact_firstname	String	First name
contact_lastname	String	Last name
contact_email	String	Email address

Field	Type	Description
contact_phone	String	Phone number
contact_types	String	'Primary', 'Billing', 'Technical'

Site - Physical service delivery locations

Field	Type	Description
site_id	Integer (PK)	Unique identifier
customer_id	Integer (FK)	Parent customer
site_name	String	Location name
address_line_1	String	Street address
city, state, zip_code	String	Location details
latitude, longitude	Float	GPS coordinates
google_maps_place_id	String	Google Maps reference
plus_code	String	Open Location Code

Service & Product Models

Service - Active service instances

Field	Type	Description
service_id	Integer (PK)	Unique identifier
customer_id	Integer (FK)	Parent customer
product_id	Integer (FK)	Product template
site_id	Integer (FK)	Service location
service_name	String	Display name
service_uuid	String	Billing system identifier
service_status	Enum	Current status
service_billed	Boolean	Generate transactions
wholesale_cost	Float	Provider cost
retail_cost	Float	Customer price
bundled_parent	Integer (FK)	Parent service for bundles

Product - Service offering templates

Field	Type	Description
product_id	Integer (PK)	Unique identifier
product_name	String	Display name
product_slug	String	URL-friendly name
category	Enum	'standalone', 'bundle', 'addon', 'promo'
provisioning_play	String	Ansible playbook name
provisioning_json_vars	JSON	Playbook variables
inventory_items_list	String	Required inventory
retail_cost	Float	Monthly price

Field	Type	Description
retail_setup_cost	Float	One-time fee
enabled	Boolean	Available for sale

Billing Models

Transaction - Individual charges/credits

Field	Type	Description
transaction_id	Integer (PK)	Unique identifier
customer_id	Integer (FK)	Parent customer
invoice_id	Integer (FK)	Parent invoice (optional)
service_id	Integer (FK)	Related service
title	String	Transaction description
retail_cost	Float	Amount
tax_percentage	Float	Tax rate
tax_amount	Float	Calculated tax
void	Boolean	Cancelled transaction

Invoice - Grouped transactions for billing

Field	Type	Description
invoice_id	Integer (PK)	Unique identifier
customer_id	Integer (FK)	Parent customer
paid	Boolean	Payment received
void	Boolean	Cancelled invoice
payment_reference	String	Stripe transaction ID
start_date, end_date	Date	Billing period
due_date	Date	Payment deadline
retail_cost	Float	Total amount

Inventory Models

Inventory - Physical and virtual assets

Field	Type	Description
inventory_id	Integer (PK)	Unique identifier
customer_id	Integer (FK)	Assigned customer
service_id	Integer (FK)	Linked service
inventory_template_id	Integer (FK)	Item type template
item	String	Item type (SIM Card, Router, etc.)
item_state	Enum	'New', 'Assigned', 'Used', etc.
itemtext1-20	String	Flexible fields
management_url	String	Equipment admin URL

Field	Type	Description
config_content	Text	Configuration file

Inventory_Template - Defines inventory item structure

Field	Type	Description
inventory_template_id	Integer (PK)	Unique identifier
item	String	Template name
itemtext1_label	String	Label for itemtext1 field
itemtext2_label	String	Label for itemtext2 field

Provisioning Models

Provision - Provisioning job tracking

Field	Type	Description
provision_id	Integer (PK)	Unique identifier
product_id	Integer (FK)	Product being provisioned
customer_id	Integer (FK)	Target customer
service_id	Integer (FK)	Created/modified service
provisioning_play	String	Ansible playbook name
provisioning_status	Integer	0=Success, 1=Running, 2=Failed

Provision_Event - Individual provisioning steps

Field	Type	Description
provision_event_id	Integer (PK)	Unique identifier
provision_id	Integer (FK)	Parent provision job
event_name	String	Task name
event_number	Integer	Sequence number
provisioning_status	Integer	0=Success, 1=Running, 2=Failed
provisioning_result_json	JSON	Full task output

User & Security Models

User - User accounts

Field	Type	Description
id	Integer (PK)	Unique identifier
username	String	Login username
email	String	Email address
email_verified	Boolean	Email confirmed
is_2fa_enabled	Boolean	Two-factor auth enabled
totp_secret	String	TOTP secret key

Role - User roles

Field	Type	Description
id	Integer (PK)	Unique identifier
name	String	Role name
description	String	Role description

Permission - Granular permissions

Field	Type	Description
id	Integer (PK)	Unique identifier
name	String	Permission name (resource.action)
description	String	Permission description

Relationships:

- Users have many Roles (many-to-many)
- Roles have many Permissions (many-to-many)
- Users can link to one Contact (for customer portal access)

Integration Points

Stripe Payment Gateway

- Tokenized payment methods
- PCI-compliant card storage
- Automated invoice payment
- Refund processing
- Expiring card notifications

Mailjet Email Service

- Transactional emails (invoices, welcome, notifications)
- Contact synchronization
- Template management
- Delivery tracking

Google Maps

- Address autocomplete
- Geocoding and reverse geocoding
- Plus Code generation
- Site location mapping

OCS/CGRateS

- Account provisioning
- Real-time rating
- Balance management
- CDR processing
- Action plans and schedules

Security Architecture

Security Features:

- JWT-based authentication
- Two-factor authentication (TOTP)
- Role-based access control (RBAC)
- Permission-based resource access
- Session management with Redis
- Password hashing (bcrypt)
- Email verification
- Audit logging via Activity Log

Deployment Architecture

Recommended Production Setup:

Technology Stack

Backend:

- Python 3.x
- Flask web framework
- SQLAlchemy ORM
- Alembic migrations
- Ansible for automation

Frontend:

- React
- State management
- React Router
- Axios for API calls

Database:

- Relational database (primary data store)
- Redis (caching & sessions)

External Services:

- CGRateS (billing/rating)
- Stripe (payments)
- Mailjet (email)
- Google Maps (geocoding)

Scalability Considerations

Horizontal Scaling:

- Stateless API design allows multiple instances
- Load balancer distributes requests
- Redis for shared session state

Database Scaling:

- Read replicas for reporting queries
- Connection pooling
- Query optimization and indexing

Provisioning at Scale:

- Background job processing
- Multiple Ansible workers
- Job queue management
- Retry logic for failed provisions

Monitoring & Observability

Key Metrics:

- API response times
- Provisioning success/failure rates
- Database connection pool usage
- Cache hit rates
- Active sessions

Logging:

- Application logs (Flask)
- Provisioning logs (Ansible output)
- Activity log (audit trail)
- Error tracking and alerts

Related Documentation

- Ansible Playbooks - Provisioning automation
- Provisioning System - Workflow details
- Products and Services - Product architecture
- API Documentation - API reference
- RBAC - Security and permissions

Authentication Flows and Admin Controls

OmniCRM provides comprehensive authentication features including login, two-factor authentication (2FA), password management, and admin controls for managing user security. This guide focuses on the UI workflows for both end users and administrators.

See also: `Self-Care Portal <self_care_portal>` for customer login and portal access, `RBAC <rbac>` for staff permissions.

Overview

OmniCRM authentication includes:

- **Email/Password Login** - Standard credential-based authentication
- **Two-Factor Authentication (2FA)** - Optional TOTP-based second factor
- **Remember Me** - Extended session up to 30 days
- **Password Reset** - Self-service password recovery via email
- **Admin Controls** - Administrative tools for resetting 2FA and passwords
- **Social Logins** - Optional Google, Apple, Facebook integration (if enabled)
- **Role-Based Navigation** - Automatic routing based on user roles

Login Flow

The login page is the entry point for all users (staff and customers).

Standard Login

Login Screen

Login Process:

1. Enter **email address** (staff or customer email)
2. Enter **password**
3. Optional: Check **"Remember me for 30 days"** for extended session
4. Click **"Login"**

What Happens Next:

- **Without 2FA:** User logged in immediately, navigated based on role:
    - **Customers** → Self-Care portal (`/self-care`)
    - **Staff/Admins** → Customers dashboard (`/customers`)
    - **CBC Mode** → Cell Broadcast interface (`/create-cell-broadcast`)
- **With 2FA Enabled:** Redirected to 2FA verification screen

Remember Me Feature:

When enabled, session persists for **30 days** instead of expiring when browser closes. Uses secure HTTP-only cookies.

Show/Hide Password:

Click the **eye icon (�)** to toggle password visibility.

Login with 2FA

If user has 2FA enabled, after entering email/password, the 2FA challenge screen appears:

Two-Factor Authentication Screen

Using Authenticator Code:

1. Open authenticator app (Google Authenticator, Authy, etc.)
2. Find OmniCRM entry
3. Enter the 6-digit code
4. Code auto-submits when all 6 digits entered
5. If valid, user logged in and navigated to appropriate dashboard

Using Recovery Code:

If authenticator app unavailable:

1. Click **"Recovery Code"** tab
2. Enter one of your saved backup codes (e.g., `3fa5b9c2`)
3. Click **"Verify"**
4. Code is consumed (can only be used once)

Cancel:

Click **"Cancel"** to return to login page.

Social Logins (Optional)

If enabled (`REACT_APP_ALLOW_SOCIAL_LOGINS=yes`), social login buttons appear:

> [◈ Sign in with Google] [● Sign in with Apple] [◈ Sign in with Facebook]

Click any button to authenticate via that provider. Currently displays "coming soon" message (social login implementation in progress).

Forgot Password Link

Click **"Forgot password?"** link to initiate password reset flow.

Two-Factor Authentication (2FA) Setup

Users can enable 2FA for enhanced account security. 2FA uses TOTP (Time-Based One-Time Password) compatible with standard authenticator apps.

Accessing 2FA Setup

From user profile or settings:

Note for Customers:

Customer role users do not see 2FA prompts. 2FA is typically required only for staff and administrative users.

Step 1: Confirm Password

> Current Password

[Cancel] [Continue]

Enter your current password to proceed. This verifies your identity before enabling 2FA.

Step 2: Scan QR Code

[Cancel] [Confirm]

Setup Instructions:

1. **Download Authenticator App** (if you don't have one):
    - iOS: Apple App Store → "Google Authenticator"
    - Android: Google Play → "Google Authenticator"
    - Alternatives: Authy, Microsoft Authenticator, 1Password
2. **Scan QR Code:**
    - Open authenticator app
    - Tap "+" or "Add account"
    - Choose "Scan QR code"
    - Point camera at QR code on screen
    - App adds "OmniCRM" entry with 6-digit code
3. **Save Backup Codes:**
    - **CRITICAL:** Write down or copy these 8 codes
    - Store in secure location (password manager, safe, etc.)
    - Each code single-use only
    - Used if you lose access to authenticator app
    - Click **"Copy Codes"** to copy all codes to clipboard
4. **Verify Setup:**
    - Enter current 6-digit code from authenticator app
    - Click **"Confirm"**
    - If valid, 2FA is now enabled

Step 3: 2FA Enabled

Success message appears:

From now on, login requires both password and 2FA code.

2FA Setup Interface

Password Reset Flow (Self-Service)

Users who forget their password can reset it via email.

Step 1: Request Reset Link

From login page, click **"Forgot password?"**

Password Reset Request

1. Enter **email address**
2. Click **"Send Reset Link"**

What Happens:

- System checks if email exists in database
- If found, sends password reset email via Mailjet
- Email contains time-limited reset link (typically 1 hour expiry)

- Success message appears: "Reset instructions have been sent to your email"

If Email Not Found:

Error message: "No account found with that email address"

Step 2: Check Email

User receives email with subject like:

> Hi [Name],
>
> You requested a password reset for your OmniCRM account.
>
> Click the link below to reset your password: <https://yourcompany.com/reset-password/abc123token456>
>
> This link expires in 1 hour.
>
> If you didn't request this, ignore this email.

Click the reset link to proceed.

Step 3: Set New Password

Reset link opens password creation page:

> ◈ (lock icon)
>
> Password
>
> Confirm Password
>
> [Reset Password]

1. Enter **new password**
2. Re-enter in **Confirm Password** field
3. Click **"Reset Password"**

Password Requirements:

- Minimum length (typically 8+ characters)
- Passwords must match

Success:

- Success message: "Password has been reset successfully"
- Automatically redirected to login page
- User can now login with new password

Expired/Invalid Token:

If reset link is expired or invalid:

> [Request New Reset Link]

Admin Controls for User Management

Administrators with appropriate permissions can manage user authentication settings from the User Management interface.

Accessing User Management

Displays table of all users with action buttons.

> Name Email Phone Actions John Smith <john@example.com> +44 123... ✏ ⯑ ⯑ ⯑ Jane Doe <jane@example.com> +44 456... ✏ ⯑ ⯑ Bob Wilson <bob@example.com> +44 789... ✏ ⯑ ⯑ ⯑ ✉

Action Icons:

- ✏ **Edit** - Modify user details, roles, permissions
- ⯑ **Delete** - Remove user account
- ⯑ **Reset Password** - Generate temporary password
- ⯑ **Reset 2FA** - Disable 2FA for user (only shown if 2FA enabled)
- ✉ **Send Welcome Email** - Resend welcome email (only shown if user never logged in)

Admin: Reset User Password

When user forgets password and admin needs to help:

Step 1: Click Reset Password Icon (⯑)

Confirmation modal appears:

> Are you sure you want to reset the password for:
>
> User: John Smith (<john@example.com>)
>
> A temporary password will be generated and displayed. The user must change this password on next login.
>
> [Cancel] [Reset Password]

Step 2: Confirm Reset

Click **"Reset Password"**. System generates secure temporary password.

Step 3: Temporary Password Displayed

> Temporary password for John Smith:
>
> [⯑ Copy Password]
>
> ⚠ IMPORTANT: • Send this password to the user via secure channel • Do not send via email or unsecured messaging • User will be forced to change password on next login
>
> [Close]

Admin Action:

- Copy temporary password
- Call user or communicate via secure method

- Provide temporary password verbally
- Instruct user to login and change password

User Experience:

When user logs in with temporary password:

1. Login succeeds
2. Immediately redirected to "Change Password" screen
3. Must set new password before accessing system
4. Cannot skip password change

Admin: Reset User 2FA

When user loses access to authenticator app and all backup codes:

Step 1: Click Reset 2FA Icon (⛉)

Only appears for users with 2FA currently enabled.

Confirmation modal appears:

Admin Reset User 2FA Confirmation

Step 2: Confirm Reset

Click **"Reset 2FA"**

Step 3: Confirmation

Success message:

> John Smith can now login with just their password. They can re-enable 2FA from their user settings.

User Experience:

- User can now login with password only (no 2FA code required)
- 2FA shield icon (⛉) disappears from user's row in admin table
- User can voluntarily re-enable 2FA from their settings

Important Security Note:

Before resetting 2FA, admin should:

1. Verify user identity through alternative means:
    - Government ID verification
    - Security questions
    - Recent transaction verification
    - In-person verification (if applicable)
2. Document the reset in customer notes
3. Inform user to re-enable 2FA after regaining access

Admin: Send Welcome Email

For users who haven't received or lost their welcome email:

When Available:

Paper plane icon (✉) only appears for users who have **never logged in** (`login_count = 0`).

Click Send Welcome Email Icon (✉)

> Send welcome email to:
>
> User: Bob Wilson (<bob@example.com>)
>
> Email will include: • Welcome message • Login instructions • Link to set initial password (if applicable) • Support contact information
>
> [Cancel] [Send Email]

Click **"Send Email"**

Success message:

Email Sent via Mailjet:

Uses template: `api_crmCommunicationUserWelcome`

Admin: Edit User

Click **Edit icon (✏)** to modify user details:

> First Name
>
> Last Name
>
> Email
>
> Phone Number
>
> Roles ☑ admin ☐ customer_service_agent_1 ☐ customer
>
> [Cancel] [Save Changes]

Editable Fields:

- Name, email, phone
- **Roles** - Assign/remove roles (affects permissions)
- Active/inactive status

Admin: Delete User

Click **Delete icon (�)** to remove user:

> Are you sure you want to delete:
>
> User: John Smith (<john@example.com>)
>
> ⚠ WARNING: This action cannot be undone.
>
> This will permanently delete: • User account and credentials • 2FA settings • Session history
>
> Customer data and transactions will NOT be deleted.

[Cancel] [Delete User]

Click **"Delete User"** to confirm.

Success message:

Best Practices

For End Users

Login Security:

- Use strong, unique passwords
- Enable "Remember me" only on personal devices
- Always logout on shared computers
- Enable 2FA for additional security

2FA Management:

- Save backup codes immediately after enabling 2FA
- Store codes in password manager or secure location
- Test a backup code to ensure they work
- Re-generate backup codes if you use several
- Contact admin if you lose both authenticator and backup codes

Password Management:

- Use password manager to generate and store passwords
- Never share passwords via email or messaging
- Change password if you suspect compromise
- Use unique password for OmniCRM (don't reuse passwords)

For Administrators

User Security Management:

- Verify user identity before resetting 2FA or passwords
- Never send temporary passwords via email
- Document all security resets in user notes
- Encourage staff to enable 2FA
- Monitor for unusual login patterns

Password Resets:

- Communicate temporary passwords via phone or in-person only
- Generate strong temporary passwords (system does this automatically)
- Ensure user changes password on first login
- Don't reset passwords unnecessarily - use email reset flow when possible

2FA Resets:

- Treat 2FA resets as high-security actions
- Verify identity through multiple channels before resetting
- Document reason for reset
- Encourage user to re-enable 2FA immediately after regaining access
- Consider requiring 2FA for all administrative users

User Management:

- Regularly review user list for inactive accounts
- Remove users who have left organization
- Ensure appropriate role assignments
- Monitor users who have never logged in
- Audit user permissions quarterly

Troubleshooting

"Invalid email or password" error

- **Cause:** Incorrect credentials
- **Fix:**
  - Verify email address is correct
  - Check caps lock is off
  - Try password reset if forgotten
  - Contact admin if account locked

2FA code not accepted

- **Cause:** Time sync issue or incorrect code
- **Fix:**
  - Ensure device time is correct (Settings → Date & Time → Automatic)
  - Wait for code to refresh (codes change every 30 seconds)
  - Try next code that appears
  - Use backup code if authenticator not working
  - Contact admin to reset 2FA if all else fails

"Remember me" not working

- **Cause:** Cookies disabled or cleared
- **Fix:**
  - Enable cookies in browser settings
  - Don't clear cookies when closing browser
  - Disable privacy extensions for OmniCRM domain
  - Try different browser

Password reset email not received

- **Cause:** Email not sent, spam filter, or wrong email
- **Fix:**
  - Check spam/junk folder
  - Verify email address is correct
  - Wait 5-10 minutes (email delivery can be delayed)
  - Check Mailjet integration is working (admin)
  - Contact admin for manual password reset

Password reset link expired

- **Cause:** Token expired (typically 1 hour)
- **Fix:**
  - Request new password reset
  - Check email and click link promptly
  - Contact admin if repeated issues

Cannot enable 2FA (incorrect password)

- **Cause:** Current password entered incorrectly
- **Fix:**
    - Verify current password
    - Reset password first if uncertain
    - Contact admin for assistance

Lost authenticator app and backup codes

- **Cause:** Phone lost/reset, backup codes not saved
- **Fix:**
    - Contact administrator immediately
    - Admin will verify identity and reset 2FA
    - Login with password only after reset
    - Re-enable 2FA and SAVE backup codes this time

Admin: "Failed to reset 2FA" error

- **Cause:** Insufficient permissions
- **Fix:**
    - Ensure you have admin role
    - Check API permissions
    - Contact system administrator

Admin: Temporary password not generated

- **Cause:** API error or permissions issue
- **Fix:**
    - Refresh page and try again
    - Verify admin permissions
    - Check API logs for errors
    - Ensure database is accessible

Security Considerations

Session Management:

- Sessions expire after inactivity period
- "Remember me" extends session to 30 days
- Sessions stored as HTTP-only cookies (not accessible to JavaScript)
- Secure flag ensures cookies only sent over HTTPS

Password Security:

- Passwords hashed using industry-standard algorithms
- Plain text passwords never stored
- Temporary passwords automatically expired after first use
- Failed login attempts tracked (potential rate limiting)

2FA Security:

- TOTP secrets encrypted in database
- QR codes generated client-side when possible
- Backup codes hashed before storage
- Each backup code single-use only

Admin Actions:

- 2FA resets logged in activity log
- Password resets create audit trail
- Admin actions require appropriate role permissions
- IP addresses logged for security events

Related Documentation

- `2fa` - Detailed 2FA API reference (API-focused)
- `rbac` - Role-based access control and permissions
- `administration_configuration` - Mailjet email configuration for password reset
- `integrations_mailjet` - Email template configuration
- `customer_care` - Self-Care portal for customers

2FA Login Verification

Create a Customer

0fT52ZvoZBE

Customers, Contacts, Sites & Services

We have a simple model of a **Customer** under this Customer, can have multiple **Contacts** and multiple **Sites**, **Services**, etc.

A **Customer** is a company or individual who has a relationship with us, to who we send an invoice / bill.

A **Contact** is a person who works with the customer, for an individual, it's probably the same as the customer themselves, a single person, but we might have family members or other contacts, and each contact has a type, for example a billing contact, a technical contact, etc, which influences how we handle the contact.

A **Site** is a physical location where we deliver services, it could be a home, office, or other location. This allows us to have multiple sites for a single customer, for example, a customer with multiple offices, and know which services are associated with which site.

A **Service** is something we bill a customer for, it could be a home internet service, mobile service, or even abstract services like leasing a subnet or providing metered electricity to a rack. Each service is linked to a customer and a site, and can have multiple charges associated with it.

Customers also have an Activity Log <csa_activity_log>, which is a record of all the changes made, Tags <administration_tags>, Attributes <administration_attributes> for storing custom metadata, Inventory Items <administration_inventory> and financial information like Transactions <payments_transaction>, Invoices <payments_invoices> & Payment Methods <basics_payment>.

Once we've created a customer we can then add a service <csa_add_service> to that customer, which is the thing we bill them for.

For information on creating a customer, see [Create a Customer].

Data Model Overview

OmniCRM uses a relational data model organized around customers and their services. The model is broken down into focused sections below.

Customer Core Relationships

The customer is the central entity, with related contacts, sites, and services.

Key Points:

- One customer can have multiple contacts (billing, technical, etc.)
- One customer can have multiple sites (branch offices, locations)
- Services are delivered to sites
- Contacts can have portal access via linked user accounts

Billing & Financial Data

Transactions and invoices track all financial activity.

Key Points:

- Transactions can be standalone or grouped into invoices
- Services automatically generate monthly transactions
- Stripe cards are tokenized and stored securely
- Invoices group multiple transactions for billing

Product & Provisioning

Products define service offerings; provisioning creates actual services.

Key Points:

- Products are templates; services are active instances
- Provisioning creates or modifies services via Ansible playbooks
- Each provision job has multiple events for tracking progress
- One product provision can create multiple services (bundles)

Inventory System

Track physical and virtual assets assigned to customers.

Key Points:

- Inventory templates define the structure (fields) for each item type
- Flexible itemtext1-20 fields adapt to different inventory types
- Items can be assigned to customers and linked to services
- Stores equipment configurations and management credentials

User & Security

User accounts with role-based access control.

Key Points:

- Users can have multiple roles; roles have multiple permissions
- Two-factor authentication (2FA) is optional per user
- Staff users are standalone; customer users link to contacts
- Granular permissions control access to resources

Customer List

The customer list provides a searchable, filterable table of all customers in the system.

Customer List View

Features:

- **Search** - Filter customers by name or ID
- **Bulk Actions** - Select multiple customers for batch operations
- **Pagination** - Navigate through large customer lists
- **Quick Actions** - View or delete customers directly from the list

Customer - Detail

The customer object itself does not contain much information, it's just a name and a reference to the contacts and sites.

Your specific deployment may include additional fields or customizations, but the basic customer object is very simple.

On the overview page is also a graph showing the Average Revenue Per User (`ARPU`) for the customer, which is the total revenue divided by the number of services, and a comparison as to how this customer compares to the average ARPU for all customers in the system.

The Customer Status options are tailored based on the specific needs of your business, but typically include options like Active, Inactive, Pending, etc, with different rules in each that control the behavior of the customer in the system in that state.

Deleting a customer can only be done if the customer has no active services, unpaid invoices or uninvoiced transactions. If the customer has any of these, you will need to close the active services and ensure the payments are made before you can delete the customer, which in turn will archive the customer and all associated data, which can later be restored if needed.

Customer - Detail

Site - Detail

Sites are physical locations where services are delivered, and can have multiple services associated with them.

They are predominantly used for business customers, where a single customer might have multiple sites, for example, a customer with multiple offices.

Having multiple sites allows us to track which services are associated with which site, for example if a customer orders a new service for a new office, we need to make sure that we deliver the correct service to the correct location. This allows us to track services by site and to bill them separately if needed.

Google Maps Integration and Geocoding

Each site is integrated with **Google Maps** to ensure accurate address data and geolocation. The UI automatically geocodes addresses and generates location data.

How Address Geocoding Works:

When adding or editing a site, the interface provides two methods for setting location:

1. **Address Search (Recommended)**
    - Use the search bar at the top of the form
    - Type an address and Google Maps Autocomplete suggests matches
    - Select the correct address from the dropdown
    - The system automatically populates:
        - **Site Name** - Place name from Google Maps
        - **Address Line 1** - Street number and name
        - **Address Line 2** - Subpremise (unit/suite number)
        - **City** - Locality
        - **State/Region** - Administrative area
        - **Post Code** - Postal code
        - **Country** - Country name
        - **Latitude & Longitude** - Precise coordinates
        - **Plus Code** - 11-character Open Location Code (e.g., `8C3MFJV8+2F`)
        - **Google Maps Place ID** - Unique place identifier
2. **Draggable Map Marker (Manual)**
    - Drag the pin on the map to the exact location

- System performs **reverse geocoding** to get address from coordinates
- Same fields auto-populate based on pin location
- Useful for rural areas or when address is imprecise

Plus Code Generation:

Plus Codes are automatically generated from latitude/longitude using the Open Location Code library. A Plus Code is a short code (11 characters) that represents a precise location anywhere in the world.

Example:

> Auto-populated fields: ✓ Site Name: "123 Main Street" ✓ Address Line 1: "123 Main Street" ✓ City: "London" ✓ State: "Greater London" ✓ Country: "United Kingdom" ✓ Post Code: "SW1A 1AA" ✓ Latitude: 51.5074 ✓ Longitude: -0.1278 ✓ Plus Code: "9C3XGPHC+3Q" ✓ Google Place ID: "ChIJdd4hrwug2EcRmSrV3Vo6llI"

Validation Requirements:

Before saving a site, the system validates:

- **Latitude & Longitude** must be set (either via search or dragging marker)
- **Country** must be populated (defaults to `REACT_APP_DEFAULT_COUNTRY` if not set)
- **Plus Code** must be 12 characters (11 + 1 for padding)

If validation fails, you'll see an error:

Visual Feedback:

The interface shows real-time feedback:

or

How Location Data is Used

The geocoded location data (latitude, longitude, Plus Code) is used throughout OmniCRM for:

1. Service Delivery and Installation

- **Field Technicians** - Access Plus Code to navigate to exact site location
- **Installation Scheduling** - Assign techs based on geographic proximity
- **Equipment Deployment** - Ensure correct equipment delivered to correct site

2. Outage Notifications

- **Geofenced Alerts** - If network outage in specific area, query sites by lat/long radius
- **Targeted Communications** - Send outage notifications only to affected customers via `Mailjet <integrations_mailjet>`
- **Status Pages** - Display outage map with affected sites

Example:

> Query: SELECT * FROM Customer_Site
> WHERE distance(latitude, longitude, 51.5074, -0.1278) < 5
>
> Result: 47 affected sites Action: Send outage notification to 47 customers

3. Reporting and Analytics

- **Geographic Revenue** - Revenue by city, state, region
- **Service Density Maps** - Heatmap of service locations
- **Expansion Planning** - Identify underserved areas

4. Multi-Site Business Customers

- **Site Management** - Track which services at which locations
- **Separate Billing** - Invoice by site if required
- **Service Assignment** - Link services to specific sites during provisioning

Rural and Remote Sites

For customers in rural areas where street addresses may not exist or be inaccurate:

1. **Use Map Drag**
    - Zoom into the approximate area
    - Drag the pin to the exact property/building
    - System generates Plus Code for that precise location
2. **Plus Code Sharing**
    - Share Plus Code with customer (e.g., `8C3MFJV8+2F`)
    - Customer can enter this in Google Maps to see exact location
    - Field techs use Plus Code for navigation
3. **Address Notes**
    - Use "Address Note" field for additional directions
    - Example: "Turn left at red barn, 500m past cattle grid"
    - Notes visible to installation teams

Site - Add with Google Maps

Tip

You can drag the pin on the map to the correct location if the address is not accurate. The system will reverse-geocode the location and populate all address fields automatically.

Site - Detail View

Contact - Detail

The Contacts are people associated with the customer. They can be billing contacts, technical contacts, or other types, and each contact has a type that influences how we handle the contact.

Customer Contacts List

We can have multiple contacts for a single customer, for example, a customer with multiple billing contacts, or a customer with multiple technical contacts.

A good example would be a company with a managed service provider, who handles the technical side of things, and a separate billing contact who handles the financial side of things, or a family where each member has their own contact but not all are authorized to make changes.

Likewise we may only want to send outage notifications to the technical contact, or only send invoices to the billing contact, and the contact type allows us to control this.

The exact logic of how contact types are used is up to your business, but the basic idea is that each contact has a type that influences how we handle them, and that each person who is associated with the customer is a contact.

Contacts are synced with the Mailjet <integrations_mailjet> integration, allowing us to send targeted email campaigns based on the contact type, location, monthly spend, or purchased services, and to manage all email templates used for transactional communications.

Contact - Overview Contact - Edit Form Contact - Detail

Navigation

OmniCRM is designed from the ground up to be responsive and intuitive.

This guide will help you navigate the system and find the information you need.

Once logged in, the navbar on the left side of the screen will show you the main sections of the system, and the sub-sections within each.

Navigation - Navbar

In the top right of the screen, you will see the user menu, which allows you to log out, change your password, or access your user settings.

Navigation - User Menu

You've got links to all of your organization's most commonly used webapps in the WebApp bar (this can be tailored to your organization's needs).

Navigation - WebApps

Any alerts you have outstanding are visible in the top right of the screen, and you can click on the alert to view more information.

Navigation - Alerts

We can change the language of the system by clicking on the language dropdown in the top right of the screen.

Navigation - Language

If you're a prince of darkness, you can switch to dark mode by clicking on the moon icon in the top right of the screen.

Navigation - Dark Mode

Billing Tab Overview

The Billing tab provides a unified view of all financial information for a customer, combining payment methods, transactions, and invoices into a single interface for efficient billing management.

Related documentation: Payment Methods <payment_methods>, Transactions <payments_transaction>, Invoices <payments_invoices>, Payment Processing <payments_process>.

Accessing the Billing Tab

Per-Customer View:

1. Navigate to **Customers → [Select Customer]**
2. Click **Billing** tab
3. View all three sections: Payment Methods, Transactions, and Invoices

System-Wide Views:

System-wide billing data can be accessed separately:

- **Billing → Transactions** - All transactions across all customers
- **Billing → Invoices** - All invoices across all customers

Self-Care Portal:

Customers accessing the Self-Care Portal <self_care_portal> see the same Billing tab structure:

- View and manage their payment methods
- View transaction history
- View and pay invoices online

Billing Tab Structure

The Billing tab is organized into three main sections, displayed as cards:

Section 1: Payment Methods

Purpose: Manage how customer pays for services

Key Features:

- View all saved credit cards
- Set default payment method
- Add new payment methods (via Stripe)
- Remove expired or unused cards

Customer Billing - Payment Methods
Payment Methods Card Detail

Documentation: basics_payment

Section 2: Transactions

Purpose: Track all charges and credits for customer

Key Features:

- View transaction statistics (Total, Uninvoiced, This Month, Last Month)
- List all transactions with filtering by void/invoice status
- Add manual transactions (charges or credits)
- Void incorrect transactions
- See which transactions are invoiced vs uninvoiced

Documentation: `payments_transaction`

Section 3: Invoices

Purpose: Group transactions into bills for customer to pay

Key Features:

- View invoice statistics (Total, Unpaid, This Month, Last Month)
- List all invoices with filtering by paid/void status
- Generate new invoices from uninvoiced transactions
- Download invoice PDFs
- Email invoices to customers
- Pay invoices online (Stripe or manual payment methods)

- Void or refund invoices

Documentation: `payments_invoices`

Data Flow Between Sections

Understanding how data flows between the three sections is crucial for effective billing management.

Flow Diagram

Transaction → Invoice Relationship

1. Transaction Creation:

When a service is provisioned or a manual charge is added:

- Transaction created in **Transactions** section
- Transaction status: **Uninvoiced**
- Transaction's `invoice_id` field is `null`

Example:

2. Invoice Generation:

When staff generates an invoice:

- Invoice created in **Invoices** section

- All uninvoiced transactions within date range grouped into invoice
- Transaction's `invoice_id` field populated
- Transaction status changes to: **Invoiced**

Example:

3. Transaction Statistics Update:

- **Uninvoiced Transactions** total decreases
- **Total Invoices** statistic increases
- **Unpaid Invoices** total increases

Invoice → Payment Relationship

1. Invoice Payment:

When customer pays invoice:

- Payment processed using saved **Payment Method** (Stripe card)
- Or manual payment method selected (cash, POS, bank transfer)
- Invoice status changes to: **Paid**

2. Payment Transaction Created:

For manual payments (non-Stripe):

- Negative transaction created automatically
- Transaction title: "Payment for Invoice #1234"
- Transaction amount: -$45.00 (negative, credits customer)
- Transaction's `invoice_id` field: Links to paid invoice

Example:

3. Statistics Update:

- **Unpaid Invoices** total decreases
- **Total Invoices This Month** unchanged (invoice already existed)

Payment Method → Invoice Relationship

Stripe Payment Flow:

1. Customer adds credit card in **Payment Methods**
2. Card tokenized via Stripe, stored securely
3. When paying invoice, customer selects saved card
4. Stripe charges card
5. Invoice marked as paid
6. `payment_reference` field populated with Stripe payment intent ID

Manual Payment Flow:

1. Customer pays via cash/POS/bank transfer (no payment method needed)
2. Staff selects payment method in Pay Invoice modal
3. Staff enters reference number (optional)
4. Negative transaction created for payment amount
5. Invoice marked as paid

Complete Billing Workflows

These workflows demonstrate how the three sections work together to accomplish common tasks.

Workflow 1: New Customer Setup and First Invoice

Goal: Set up billing for new customer and collect first payment

1. **Add Payment Method:**
    - Navigate to customer → **Billing** tab
    - **Payment Methods** section → Click **"Add Payment Method"**
    - Customer adds credit card via Stripe
    - Card saved as default payment method
2. **Verify Transactions:**
    - **Transactions** section shows uninvoiced transactions:
        - Service setup fee: $50.00
        - First month service: $45.00
        - Total Uninvoiced: $95.00
3. **Generate Invoice:**
    - **Invoices** section → Click **"Generate Proforma Invoice"**
    - Set date range to include setup and first month
    - Click **"Generate Invoice"**
    - Invoice #INV-2025-001234 created for $95.00
4. **Transactions Update:**
    - Both transactions now show: Invoice #INV-2025-001234
    - **Uninvoiced Transactions** total now $0.00
5. **Email Invoice:**
    - Click email icon next to invoice
    - Customer receives invoice email with PDF and pay link
6. **Customer Pays Online:**
    - Customer clicks pay link in email
    - Redirected to Self-Care portal
    - Click **"Pay Invoice"** button
    - Select default payment method
    - Click **"Pay Invoice"**
    - Stripe charges card
7. **Invoice Update:**
    - Invoice status changes to **"Paid"**
    - **Unpaid Invoices** total decreases by $95.00

Result: Customer fully set up with payment method, first invoice paid.

Workflow 2: Monthly Recurring Billing

Goal: Bill all customers for monthly service at end of month

1. **Services Auto-Charge:**
    - End of month arrives (January 31)
    - Billing system automatically creates transactions for all recurring services
    - **Transactions** section shows new uninvoiced transactions
2. **Review Uninvoiced Transactions:**
    - Navigate to **Transactions** section
    - Filter: **Invoice Status: Not Invoiced**
    - Review list of all transactions ready for billing
    - Verify amounts and descriptions correct
3. **Generate Invoices:**
    - Navigate to **Billing → Invoices** (system-wide)

    - For each customer (or use batch process):
        - Click **"Generate Proforma Invoice"**
        - Select customer
        - Start Date: 2025-01-01
        - End Date: 2025-01-31
        - Due Date: 2025-02-15
        - Click **"Generate Invoice"**
4. **Transactions Update:**
    - All transactions now linked to invoices
    - **Uninvoiced Transactions** totals reset to $0.00
5. **Email All Invoices:**
    - For each invoice, click email icon
    - All customers receive monthly invoices
6. **Customers Pay:**
    - Customers with saved payment methods pay online via Self-Care
    - Staff processes cash/POS payments for customers who pay in person
    - **Unpaid Invoices** total decreases as payments received

Result: All customers billed for January, invoices sent, payments processed.

Workflow 3: Handling Service Issue Credit

Goal: Credit customer for service outage, apply to unpaid invoice

1. **Customer Reports Issue:**
    - Service was down for 2 days
    - Customer deserves $10 credit
2. **Add Credit Transaction:**
    - Navigate to customer → **Billing** tab → **Transactions** section
    - Click **"+ Add Transaction"**
    - Transaction Type: **Credit**
    - Credit Type: **Cash Payment** (or appropriate type)
    - Title: "Service Outage Credit"
    - Description: "Compensation for 2-day outage 8-9 Jan"
    - Retail Cost: 10.00
    - Click **"Add Transaction"**
3. **Transaction Created:**
    - Transaction appears in list with amount: **-$10.00**
    - Transaction status: **Uninvoiced**
    - **Uninvoiced Transactions** total now includes -$10.00
4. **Apply to Invoice:**
    - If customer already has unpaid invoice:
        - Invoice remains unpaid with original amount
        - Credit will be applied to next invoice generation
    - If generating new invoice:
        - **Invoices** section → Click **"Generate Proforma Invoice"**
        - Include date range with credit transaction
        - Invoice generated with credit applied:
5. **Customer Pays:**
    - Customer pays reduced amount: $35.00
    - Invoice marked as paid

Result: Customer credited for outage, credit applied to next invoice, lower payment collected.

Workflow 4: Payment Method Expired - Update and Retry

Goal: Customer's card expired, causing payment failure - update card and retry payment

1. **Payment Failure Notification:**
    - Customer attempts to pay invoice
    - Stripe returns error: "Card expired"
    - Payment fails, invoice remains unpaid
2. **Update Payment Method:**
    - Customer navigates to **Billing** tab
    - **Payment Methods** section → Click **"Add Payment Method"**
    - Enter new card details (updated expiry date)
    - New card saved
3. **Set as Default:**
    - Customer clicks **"Set as Default"** on new card
    - Old card removed automatically (if desired)
4. **Retry Payment:**
    - Navigate to **Invoices** section
    - Locate unpaid invoice
    - Click **"Pay"** icon
    - Payment modal opens with new default card pre-selected
    - Click **"Pay Invoice"**
    - Stripe charges new card successfully
5. **Invoice Update:**
    - Invoice status changes to **"Paid"**
    - `payment_reference` field populated with new Stripe payment intent ID

Result: Customer updated payment method, invoice successfully paid with new card.

Workflow 5: Voiding Incorrect Invoice and Re-Billing

Goal: Staff generated invoice with wrong transactions - void and regenerate correctly

1. **Error Discovered:**
    - Invoice #INV-2025-001234 generated with wrong date range
    - Included transactions from wrong month
    - Invoice is unpaid
2. **Void Invoice:**
    - Navigate to **Billing** tab → **Invoices** section
    - Locate incorrect invoice
    - Click delete icon (�)
    - Confirm void
    - Invoice voided
3. **Transactions Released:**
    - Navigate to **Transactions** section
    - All transactions from voided invoice now show: **Uninvoiced**
    - **Uninvoiced Transactions** total increases
    - Transactions available for new invoice
4. **Generate Correct Invoice:**
    - **Invoices** section → Click **"Generate Proforma Invoice"**
    - Set correct date range
    - Apply filter if needed (e.g., "Mobile" for mobile-only invoice)
    - Click **"Generate Invoice"**
    - New invoice created with correct transactions
5. **Verify and Email:**
    - Review new invoice details
    - Verify correct transactions included
    - Click email icon to send to customer

Result: Incorrect invoice voided, transactions re-invoiced correctly, customer receives corrected invoice.

Workflow 6: Processing Cash Payment for Multiple Invoices

Goal: Customer pays multiple unpaid invoices with single cash payment

1. **Customer Arrives with Cash:**
    - Customer brings $300 cash to pay outstanding invoices
    - Navigate to customer → **Billing** tab
2. **Review Unpaid Invoices:**
    - **Invoices** section → Filter: **Paid: Not yet Paid**
    - View unpaid invoices:
3. **Pay First Invoice:**
    - Click pay icon on invoice #1234
    - Payment modal opens
    - Select **"Cash"** payment method
    - Enter reference: "Cash paid 2025-02-10 - Receipt #001"
    - Click **"Pay Invoice"**
    - Invoice #1234 marked as **"Paid"**
4. **Pay Remaining Invoices:**
    - Repeat process for invoice #1235:
        - Reference: "Cash paid 2025-02-10 - Receipt #001"
    - Repeat for invoice #1236:
        - Reference: "Cash paid 2025-02-10 - Receipt #001"
5. **Verify Transactions:**
    - Navigate to **Transactions** section
    - Three new payment transactions created:
    - All linked to respective invoices
6. **Update Statistics:**
    - **Invoices** section → **Unpaid Invoices** total decreased by $300.00
    - All invoices now show **"Paid"** status

Result: Customer paid all outstanding invoices with cash, payment transactions recorded with receipt reference.

Best Practices

For Staff Users

Transaction Management:

- Add manual transactions immediately (don't delay)
- Use descriptive titles and descriptions for clarity
- Link transactions to services and sites when applicable
- Void incorrect transactions before they're invoiced

Invoice Generation:

- Generate invoices at consistent intervals (e.g., monthly on 1st of month)
- Use date ranges carefully to avoid overlap or gaps
- Use filters to create service-specific invoices when needed
- Email invoices immediately after generation
- Review invoice PDFs before sending to customers

Payment Processing:

- Verify payment method valid before attempting charge
- Always enter reference numbers for manual payments (cash, POS, bank transfer)
- Mark invoices as paid immediately after receiving payment

- Refund via Stripe only (create manual credit for other payment methods)

Data Hygiene:

- Regularly review uninvoiced transactions
- Investigate void transactions to understand billing errors
- Monitor unpaid invoices and follow up with customers
- Keep payment methods current (remove expired cards)

For Customers (Self-Care Portal)

Payment Methods:

- Keep at least one valid payment method on file
- Update payment methods before cards expire
- Set your preferred card as default

Invoice Payments:

- Pay invoices before due date to avoid late fees
- Review invoice details and transactions before paying
- Download invoice PDFs for your records
- Contact support immediately if invoice appears incorrect

Transaction Review:

- Regularly review transaction history
- Report any unexpected charges immediately
- Understand which transactions are invoiced vs uninvoiced

For Administrators

System Configuration:

- Configure Mailjet email templates for professional invoice delivery
- Customize invoice PDF templates to match branding
- Set up Stripe integration for secure payment processing
- Configure payment terms and due dates

Monitoring and Reporting:

- Use statistics widgets to monitor billing health
- Track **Uninvoiced Transactions** total - should decrease after billing cycle
- Monitor **Unpaid Invoices** total - follow up on overdue payments
- Review **This Month** vs **Last Month** statistics for trends

Automation:

- Automate recurring service charges via product configuration
- Set up automatic invoice generation for recurring billing (if available)
- Configure email reminders for overdue invoices

Common Issues and Solutions

Issue: Customer Cannot Pay Invoice

Symptoms:

- Customer clicks pay button but nothing happens
- Error message: "No payment methods found"

Diagnosis:

1. Navigate to customer → **Billing** tab → **Payment Methods** section
2. Check if customer has any saved payment methods
3. Check if saved cards are expired

Solution:

- Customer must add valid payment method before paying invoices
- Guide customer to **Payment Methods** page to add credit card
- Verify card accepted (Visa, Mastercard, Amex, etc.)
- Retry payment after card added

Issue: Invoice Generated with Wrong Transactions

Symptoms:

- Invoice includes transactions from wrong period
- Invoice missing expected transactions
- Invoice total incorrect

Diagnosis:

1. Open invoice in **Invoices** section
2. Review transactions included in invoice
3. Check transaction dates vs invoice date range
4. Check if filter was applied during generation

Solution:

- **If invoice unpaid:** Void invoice, verify transactions uninvoiced, regenerate with correct date range
- **If invoice paid:** Cannot void - create credit transaction for incorrect amount, generate corrected invoice
- **Prevention:** Always review **Transactions** section before generating invoice to verify correct transactions will be included

Issue: Uninvoiced Transactions Total Not Decreasing

Symptoms:

- **Uninvoiced Transactions** widget shows high amount
- Transactions list shows many uninvoiced transactions
- Monthly invoices already generated

Diagnosis:

1. Filter transactions by **Invoice Status: Not Invoiced**
2. Review list of uninvoiced transactions
3. Check transaction dates - may be recent charges after last invoice generation
4. Check if some transactions are voided (should not count toward uninvoiced total)

Solution:

- **Expected behavior:** Transactions created after last invoice generation remain uninvoiced

until next billing cycle
- **If old transactions uninvoiced:** Generate invoice for those transactions with appropriate date range
- **If voided transactions counted:** System should exclude voided transactions automatically - report bug if not

Issue: Payment Method Keeps Declining

Symptoms:

- Stripe payment fails with "Card declined"
- Customer reports card should work
- Multiple retry attempts fail

Diagnosis:

1. Check Stripe dashboard for decline reason
2. Common decline reasons:
    - Insufficient funds
    - Card expired
    - Card reported lost/stolen
    - Bank fraud prevention
    - International card blocked

Solution:

- Ask customer to contact their bank to authorize payment
- Try different payment method (different card)
- For staff: Process as manual payment (cash, POS) and mark invoice paid
- Verify billing address matches card billing address

Keyboard Shortcuts and Tips

Navigation:

- Access billing tab quickly: Customer page → **B** key (if keyboard shortcuts enabled)

Filtering:

- **Transactions:** Click **"Not Invoiced"** filter before generating invoices to see exactly what will be included
- **Invoices:** Click **"Not yet Paid"** filter to see all outstanding invoices requiring follow-up

Bulk Operations:

- Select multiple invoices (checkbox) to delete/void multiple at once (staff only)
- Use search bar to quickly find specific transaction or invoice by ID

Quick Actions:

- Click transaction/invoice title to view full details in modal
- Right-click actions menu (⋮) for quick access to download/email/pay options

Related Documentation

- `basics_payment` - Payment methods and Stripe integration
- `payments_transaction` - Transactions in detail

- `payments_invoices` - Invoices in detail
- `integrations_mailjet` - Email invoice delivery
- `csa_activity_log` - Viewing billing activity history

Ansible Playbooks: Detailed Guide

OmniCRM products are provisioned using **Ansible**, allowing for automated service management based on the specific requirements of each product and its associated inventory.

How Playbooks and Products Work Together

Critical Concept: Playbooks are what actually create services in OmniCRM. When you assign a playbook to a product, you're defining **what happens** when that product is provisioned - but that can mean different things for different products.

Products Trigger Playbooks

When a product is provisioned in OmniCRM:

1. The product definition specifies which playbook to run (via `provisioning_play` field)
2. The product passes variables to the playbook (via `provisioning_json_vars` and inventory selections)
3. The playbook executes and does whatever it's programmed to do
4. The playbook determines what gets created (if anything)

What Playbooks Can Do

A single provisioning playbook can:

Create Multiple Services
A bundled product playbook might create:

- A main internet service record
- An IPTV addon service record
- A VoIP service record
- All with one product provision action

Create No Services
Some playbooks don't create service records at all:

- A playbook that just configures CPE equipment
- A playbook that sends configuration to network equipment
- A playbook that updates external systems

Create One Service

The most common pattern:

- Create a single service record for the customer
- Link inventory to that service
- Set up billing for that service

Modify Existing Services
Topup and addon playbooks:

- Don't create new services
- Update existing service records (add data, extend expiry, etc.)
- Add balances to existing billing accounts

Perform Actions Without Service Records
Some playbooks are purely operational:

- Reset account balances
- Swap inventory items between customers
- Generate reports or configurations

Example: Different Playbook Behaviors

```
# Product 1: Mobile SIM Service (creates 1 service)
# provisioning_play: play_simple_service
- Creates service record in CRM
- Creates billing account in OCS
- Assigns SIM card and phone number inventory
- Sends welcome email

# Product 2: Internet Bundle (creates 3 services)
# provisioning_play: play_bundle_internet_tv_voice
- Creates internet service record
- Creates IPTV service record
- Creates VoIP service record
- Links all to same customer
- Single billing account for the bundle

# Product 3: Data Topup (creates 0 services)
# provisioning_play: play_topup_no_charge
- Finds existing service by service_id
- Adds data balance to existing OCS account
- Updates service expiry date
- NO new service created

# Product 4: CPE Configuration (creates 0 services)
# provisioning_play: play_prov_cpe_mikrotik
- Generates router configuration
- Updates inventory record with config
```

```
- Emails config to support team
- NO service created (just equipment setup)
```

The key point: **The playbook defines the behavior, the product is just a trigger.**

Plays vs Tasks

Understanding the distinction between Plays and Tasks is fundamental to working with OmniCRM playbooks.

Play (Playbook)
A complete provisioning workflow that orchestrates multiple tasks to achieve a business objective. Plays are the top-level playbooks stored in `OmniCRM-API/Provisioners/plays/` and are referenced in product definitions.

Examples:

- `play_simple_service.yaml` - Provision a basic service
- `play_topup_no_charge.yaml` - Apply a free topup to a service
- `play_prov_cpe_mikrotik.yaml` - Configure customer premises equipment

Task (Reusable Component)
A self-contained, reusable set of operations that can be included by multiple plays. Tasks are prefixed with `task_` and live in the same directory.

Examples:

- `task_welcome_email.yaml` - Send a welcome email to a customer
- `task_activate_olt.yaml` - Activate OLT equipment
- `task_notify_ocs.yaml` - Send notifications to the billing system

The relationship between them:

```yaml
# play_simple_service.yaml (A Play)
- name: Simple Provisioning Play
  hosts: localhost
  tasks:
    - name: Main provisioning block
      block:
        - name: Create service
          uri: ...

        - name: Configure billing
          uri: ...

        # Include reusable task
        - include_tasks: task_welcome_email.yaml
```

```yaml
        # Include post-provisioning tasks
        - include_tasks: post_provisioning_tasks.yaml
```

Playbook Structure and Anatomy

All OmniCRM playbooks follow a consistent structure. Understanding this structure is essential for creating and maintaining playbooks.

Basic Structure

Every playbook starts with these standard headers:

```yaml
- name: Descriptive Name of the Playbook
  hosts: localhost           # Always localhost for OmniCRM
  gather_facts: no           # Disabled for performance
  become: False              # Don't escalate privileges

  tasks:
    - name: Main block
      block:
        # Provisioning tasks go here

      rescue:
        # Rollback/cleanup tasks go here
```

Header Explanation

name
Descriptive name shown in provisioning logs and UI. This appears as `playbook_description` in the provision record.

hosts: localhost
All OmniCRM playbooks run on localhost since they interact with remote systems via APIs, not SSH.

gather_facts: no
Ansible's fact gathering is disabled because:

- We don't need system information
- It adds unnecessary overhead
- Can crash browsers if displayed in debug output

become: False
No privilege escalation is needed since we're making API calls, not modifying system files.

Configuration Loading

Every playbook must load the central configuration file:

```yaml
tasks:
  - name: Include vars of crm_config
    ansible.builtin.include_vars:
      file: "../../crm_config.yaml"
      name: crm_config
```

This makes the configuration available as `crm_config.ocs.cgrates`, `crm_config.crm.base_url`, etc.

The `crm_config.yaml` typically contains:

```yaml
ocs:
  cgrates: "10.0.1.100:2080"
  ocsTenant: "default_tenant"
crm:
  base_url: "https://crm.example.com"
```

Variable Access Patterns

Variables can come from several sources:

From the Product Definition:

```yaml
- name: Access product_id passed by OmniCRM
  debug:
    msg: "Provisioning product {{ product_id }}"
```

From Inventory Selection:

```yaml
- name: Get inventory ID for SIM Card
  set_fact:
    sim_card_id: "{{ hostvars[inventory_hostname]['SIM Card'] | int
}}"
  when: "'SIM Card' in hostvars[inventory_hostname]"
```

From API Responses:

```yaml
- name: Get Product information from CRM API
  uri:
    url: "http://localhost:5000/crm/product/product_id/{{ product_id
}}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
```

```yaml
    return_content: yes
  register: api_response_product

- name: Use the product name
  debug:
    msg: "Product name is {{ api_response_product.json.product_name
}}"
```

Common Playbook Patterns

Service Provisioning Pattern

This is the most common pattern for creating new services.

```yaml
- name: Service Provisioning Playbook
  hosts: localhost
  gather_facts: no
  become: False

  tasks:
    - name: Main block
      block:

      # 1. Load configuration
      - name: Include vars of crm_config
        ansible.builtin.include_vars:
          file: "../../crm_config.yaml"
          name: crm_config

      # 2. Get product information
      - name: Get Product information from CRM API
        uri:
          url: "http://localhost:5000/crm/product/product_id/{{
product_id }}"
          method: GET
          headers:
            Authorization: "Bearer {{ access_token }}"
          return_content: yes
          validate_certs: no
        register: api_response_product

      # 3. Get customer information
      - name: Get Customer information from CRM API
        uri:
          url: "http://localhost:5000/crm/customer/customer_id/{{
customer_id }}"
          method: GET
          headers:
```

```
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_customer

    # 4. Set facts from retrieved data
    - name: Set package facts
      set_fact:
        package_name: "{{ api_response_product.json.product_name
}}"
        package_comment: "{{ api_response_product.json.comment }}"
        setup_cost: "{{ api_response_product.json.retail_setup_cost
}}"
        monthly_cost: "{{ api_response_product.json.retail_cost }}"

    # 5. Generate unique identifiers
    - name: Generate UUID
      set_fact:
        uuid: "{{ 99999999 | random | to_uuid }}"

    - name: Generate Service UUID
      set_fact:
        service_uuid: "Service_{{ uuid[0:8] }}"

    # 6. Create account in billing system
    - name: Create account in OCS/CGRateS
      uri:
        url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
        method: POST
        body_format: json
        headers:
          Content-Type: "application/json"
        body:
          {
            "method": "ApierV2.SetAccount",
            "params": [{
                "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                "Account": "{{ service_uuid }}",
                "ActionPlanIds": [],
                "ActionPlansOverwrite": true,
                "ExtraOptions": {
                  "AllowNegative": false,
                  "Disabled": false
                },
                "ReloadScheduler": true
            }]
          }
        status_code: 200
      register: ocs_response
```

```yaml
    - name: Verify OCS account creation
      assert:
        that:
          - ocs_response.status == 200
          - ocs_response.json.result == "OK"

    # 7. Add initial balance
    - name: Add 0 Monetary Balance
      uri:
        url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
        method: POST
        body_format: json
        body:
          {
            "method": "ApierV1.AddBalance",
            "params": [{
                "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                "Account": "{{ service_uuid }}",
                "BalanceType": "*monetary",
                "Categories": "*any",
                "Balance": {
                    "ID": "Initial Balance",
                    "Value": 0,
                    "ExpiryTime": "+4320h",
                    "Weight": 1,
                    "Blocker": true
                }
            }]
          }
        status_code: 200
      register: balance_response

    # 8. Create service record in CRM
    - name: Get current date and time in ISO 8601 format
      command: date --utc +%Y-%m-%dT%H:%M:%S%z
      register: current_date_time

    - name: Add Service via API
      uri:
        url: "http://localhost:5000/crm/service/"
        method: PUT
        body_format: json
        headers:
          Content-Type: "application/json"
          Authorization: "Bearer {{ access_token }}"
        body:
          {
```

```
              "customer_id": "{{ customer_id }}",
              "product_id": "{{ product_id }}",
              "service_name": "{{ package_name }} - {{ service_uuid
}}",
              "service_type": "generic",
              "service_uuid": "{{ service_uuid }}",
              "service_billed": true,
              "service_taxable": true,
              "service_provisioned_date": "{{
current_date_time.stdout }}",
              "service_status": "Active",
              "wholesale_cost": "{{
api_response_product.json.wholesale_cost | float }}",
              "retail_cost": "{{ monthly_cost | float }}"
            }
          status_code: 200
        register: service_creation_response

      # 9. Add setup cost transaction
      - name: Add Setup Cost Transaction via API
        uri:
          url: "http://localhost:5000/crm/transaction/"
          method: PUT
          headers:
            Content-Type: "application/json"
            Authorization: "Bearer {{ access_token }}"
          body_format: json
          body:
            {
              "customer_id": {{ customer_id | int }},
              "service_id": {{
service_creation_response.json.service_id | int }},
              "title": "{{ package_name }} - Setup Costs",
              "description": "Setup costs for {{ package_comment }}",
              "invoice_id": null,
              "retail_cost": "{{ setup_cost | float }}"
            }
          return_content: yes
        register: transaction_response

      # 10. Include post-provisioning tasks
      - include_tasks: post_provisioning_tasks.yaml

      rescue:

      # Rollback/cleanup section
      - name: Print all vars for debugging
        debug:
```

```
          var: hostvars[inventory_hostname]

      - name: Remove account in OCS
        uri:
          url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
          method: POST
          body_format: json
          body:
            {
              "method": "ApierV2.RemoveAccount",
              "params": [{
                  "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                  "Account": "{{ service_uuid }}",
                  "ReloadScheduler": true
              }]
            }
          status_code: 200
        ignore_errors: True
        when: service_uuid is defined

      - name: Delete Service from CRM if it was created
        uri:
          url: "http://localhost:5000/crm/service/service_id/{{
service_creation_response.json.service_id }}"
          method: DELETE
          headers:
            Authorization: "Bearer {{ access_token }}"
          status_code: 200
        ignore_errors: True
        when: service_creation_response is defined

      - name: Fail if not intentional deprovision
        assert:
          that:
            - action == "deprovision"
```

Topup/Recharge Pattern

Used for adding credits, data, or time to existing services.

```
- name: Service Topup Playbook
  hosts: localhost
  gather_facts: no
  become: False

  tasks:
    - name: Include vars of crm_config
      ansible.builtin.include_vars:
```

```yaml
        file: "../../crm_config.yaml"
        name: crm_config

    # 1. Get service information
    - name: Get Service information from CRM API
      uri:
        url: "http://localhost:5000/crm/service/service_id/{{
service_id }}"
        method: GET
        headers:
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_service

    # 2. Get product information (what to topup)
    - name: Get Product information from CRM API
      uri:
        url: "http://localhost:5000/crm/product/product_id/{{
product_id }}"
        method: GET
        headers:
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_product

    # 3. Extract service details
    - name: Set service facts
      set_fact:
        service_uuid: "{{ api_response_service.json.service_uuid }}"
        customer_id: "{{ api_response_service.json.customer_id }}"
        package_name: "{{ api_response_product.json.product_name }}"
        topup_value: "{{ api_response_product.json.retail_cost }}"

    # 4. Execute action in billing system (free topup)
    - name: Execute Action to add credits
      uri:
        url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
        method: POST
        body_format: json
        body:
          {
            "method": "APIerSv1.ExecuteAction",
            "params": [{
                "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                "Account": "{{ service_uuid }}",
                "ActionsId": "Action_Topup_Standard"
            }]
          }
```

```yaml
        status_code: 200
      register: action_response

    - name: Verify action executed successfully
      assert:
        that:
          - action_response.status == 200
          - action_response.json.result == "OK"

    # 5. Reset any triggered limits
    - name: Reset ActionTriggers
      uri:
        url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
        method: POST
        body_format: json
        body:
          {
            "method": "APIerSv1.ResetAccountActionTriggers",
            "params": [{
                "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                "Account": "{{ service_uuid }}",
                "Executed": false
            }]
          }
        status_code: 200

    # 6. Update service dates
    - name: Calculate new expiry date
      command: "date --utc +%Y-%m-%dT%H:%M:%S%z -d '+30 days'"
      register: new_expiry_date

    - name: Update Service with new expiry
      uri:
        url: "http://localhost:5000/crm/service/{{ service_id }}"
        method: PATCH
        headers:
          Authorization: "Bearer {{ access_token }}"
          Content-Type: "application/json"
        body_format: json
        body:
          {
            "service_deactivate_date": "{{ new_expiry_date.stdout
}}",
            "service_status": "Active"
          }

    # 7. Optional: Send notification
    - name: Send Notification SMS
```

```yaml
      uri:
        url: "http://sms-gateway/api/send"
        method: POST
        body_format: json
        body:
          {
            "source": "CompanyName",
            "destination": "{{ customer_phone }}",
            "message": "Your service has been topped up. New expiry:
{{ new_expiry_date.stdout }}"
          }
        status_code: 201
      ignore_errors: True
```

CPE Provisioning Pattern

Used for configuring customer premises equipment (routers, modems, ONTs).

```yaml
- name: CPE Provisioning Playbook
  hosts: localhost
  gather_facts: no
  become: False

  tasks:
    - name: Include vars of crm_config
      ansible.builtin.include_vars:
        file: "../../crm_config.yaml"
        name: crm_config

    # 1. Get inventory item for CPE
    - name: Set CPE inventory ID from hostvars
      set_fact:
        cpe_inventory_id: "{{ hostvars[inventory_hostname]['WiFi
Router CPE'] | int }}"
      when: "'WiFi Router CPE' in hostvars[inventory_hostname]"

    # 2. Get CPE details from inventory
    - name: Get Inventory data for CPE
      uri:
        url: "{{ crm_config.crm.base_url }}/crm/inventory/
inventory_id/{{ cpe_inventory_id }}"
        method: GET
        headers:
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_cpe

    # 3. Get customer site information
```

```
    - name: Get Site info from API
      uri:
        url: "{{ crm_config.crm.base_url }}/crm/site/customer_id/{{
customer_id }}"
        method: GET
        headers:
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_site

    # 4. Update CPE inventory with location
    - name: Patch CPE inventory item with location
      uri:
        url: "{{ crm_config.crm.base_url }}/crm/inventory/
inventory_id/{{ cpe_inventory_id }}"
        method: PATCH
        body_format: json
        headers:
          Authorization: "Bearer {{ access_token }}"
        body:
          {
            "address_line_1": "{{
api_response_site.json.0.address_line_1 }}",
            "city": "{{ api_response_site.json.0.city }}",
            "state": "{{ api_response_site.json.0.state }}",
            "latitude": "{{ api_response_site.json.0.latitude }}",
            "longitude": "{{ api_response_site.json.0.longitude }}"
          }
        status_code: 200

    # 5. Generate credentials
    - name: Set CPE hostname
      set_fact:
        cpe_hostname: "CPE_{{ cpe_inventory_id }}"
        cpe_username: "admin_{{ cpe_inventory_id }}"

    - name: Generate random password
      set_fact:
        cpe_password: "{{ lookup('pipe', 'cat /dev/urandom | tr -dc
a-zA-Z0-9 | head -c 16') }}"

    # 6. Generate WiFi credentials
    - name: Set WiFi SSID
      set_fact:
        wifi_ssid: "Network_{{ cpe_inventory_id }}"

    - name: Generate WiFi password
      set_fact:
```

```yaml
        word_list:
          - apple
          - cloud
          - river
          - mountain
          - ocean

    - name: Create WiFi PSK
      set_fact:
        random_word: "{{ word_list | random }}"
        random_number: "{{ 99999 | random(start=10000) }}"

    - name: Combine WiFi PSK
      set_fact:
        wifi_psk: "{{ random_word }}{{ random_number }}"

    # 7. Generate configuration file
    - name: Set config filename
      set_fact:
        config_name: "{{ cpe_hostname }}_{{ lookup('pipe', 'date
+%Y%m%d%H%M%S') }}.cfg"
        config_dest: "/tmp/{{ cpe_hostname }}_{{ lookup('pipe', 'date
+%Y%m%d%H%M%S') }}.cfg"

    - name: Create config from template
      template:
        src: "templates/cpe_router_config.j2"
        dest: "{{ config_dest }}"

    # 8. Read generated config
    - name: Read config file
      ansible.builtin.slurp:
        src: "{{ config_dest }}"
      register: config_content

    # 9. Update inventory with provisioning info
    - name: Patch CPE inventory with config
      uri:
        url: "{{ crm_config.crm.base_url }}/crm/inventory/
inventory_id/{{ cpe_inventory_id }}"
        method: PATCH
        body_format: json
        headers:
          Authorization: "Bearer {{ access_token }}"
        body:
          {
            "itemtext3": "{{ wifi_ssid }}",
            "itemtext4": "{{ wifi_psk }}",
```

```
            "management_url": "{{ cpe_hostname }}",
            "management_username": "{{ cpe_username }}",
            "management_password": "{{ cpe_password }}",
            "config_content": "{{ config_content.content | b64decode
}}",
            "inventory_notes": "Provisioned: {{ lookup('pipe', 'date
+%Y-%m-%d') }}"
          }
        status_code: 200

    # 10. Send config to support team
    - name: Email configuration to support
      uri:
        url: "https://api.mailjet.com/v3.1/send"
        method: POST
        body_format: json
        headers:
          Content-Type: "application/json"
        body:
          {
            "Messages": [{
              "From": {
                "Email": "provisioning@example.com",
                "Name": "Provisioning System"
              },
              "To": [{
                "Email": "support@example.com",
                "Name": "Support Team"
              }],
              "Subject": "CPE Config - {{ cpe_hostname }}",
              "Attachments": [{
                "ContentType": "text/plain",
                "Filename": "{{ config_name }}",
                "Base64Content": "{{ config_content.content }}"
              }]
            }]
          }
        user: "{{ mailjet_api_key }}"
        password: "{{ mailjet_api_secret }}"
        force_basic_auth: true
        status_code: 200
```

Auto-Renewal Pattern

Configure automatic recurring charges or renewals using CGRateS ActionPlans.

```
# Part of a topup playbook that sets up auto-renewal
```

```
# 1. Normalize auto_renew parameter
- name: Normalize auto_renew to boolean
  set_fact:
    auto_renew_bool: "{{ (auto_renew | string | lower) in ['true',
'1', 'yes'] }}"

# 2. Create action for auto-renewal
- name: Create Action for AutoRenew
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.SetActions",
        "params": [{
          "ActionsId": "Action_AutoTopup_{{ service_uuid }}_{{
product_id }}",
          "Overwrite": true,
          "Actions": [
            {
              "Identifier": "*http_post",
              "ExtraParameters": "{{ crm_config.crm.base_url }}/crm/
provision/simple_provision_addon/service_id/{{ service_id
}}/product_id/{{ product_id }}"
            },
            {
              "Identifier": "*cdrlog",
              "BalanceType": "*generic",
              "ExtraParameters":
"{\"Category\":\"^activation\",\"Destination\":\"Auto Renewal\"}"
            }
          ]
        }]
      }
    status_code: 200
  register: action_response
  when: auto_renew_bool

# 3. Create monthly ActionPlan
- name: Create ActionPlan for Monthly Renewal
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.SetActionPlan",
```

```yaml
        "params": [{
          "Id": "ActionPlan_Monthly_{{ service_uuid }}_{{ product_id
}}",
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "ActionPlan": [{
            "ActionsId": "Action_AutoTopup_{{ service_uuid }}_{{
product_id }}",
            "Years": "*any",
            "Months": "*any",
            "MonthDays": "*any",
            "WeekDays": "*any",
            "Time": "*monthly",
            "StartTime": "*now",
            "Weight": 10
          }],
          "Overwrite": true,
          "ReloadScheduler": true
        }]
      }
    status_code: 200
  when: auto_renew_bool

# 4. Assign ActionPlan to account
- name: Assign ActionPlan to account
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV2.SetAccount",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}",
          "ActionPlanIds": ["ActionPlan_Monthly_{{ service_uuid }}_{{
product_id }}"],
          "ActionPlansOverwrite": true,
          "ReloadScheduler": true
        }]
      }
    status_code: 200
  when: auto_renew_bool

# 5. Remove ActionPlan if auto-renew disabled
- name: Remove ActionPlan from account
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
```

```
    body_format: json
    body:
      {
        "method": "ApierV1.RemoveActionPlan",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Id": "ActionPlan_Monthly_{{ service_uuid }}_{{ product_id
}}"
        }]
      }
    status_code: 200
  ignore_errors: true
  when: not auto_renew_bool
```

Reusable Tasks

Reusable tasks are small, self-contained playbooks that can be included by multiple plays. They promote code reuse and consistency.

Welcome Email Task

`task_welcome_email.yaml` - Sends a welcome email to new customers.

```
# This task expects these variables to be set by the parent play:
# - api_response_customer (customer details)
# - package_name (product name)
# - monthly_cost (recurring cost)
# - setup_cost (one-time cost)

- name: Set email configuration
  set_fact:
    mailjet_api_key: "{{ lookup('env', 'MAILJET_API_KEY') }}"
    mailjet_api_secret: "{{ lookup('env', 'MAILJET_SECRET') }}"
    email_from: "noreply@example.com"
    recipients: []

- name: Set email subject and sender name
  set_fact:
    email_subject: "Welcome to our service!"
    email_from_name: "Customer Service Team"

- name: Prepare list of recipients from customer contacts
  loop: "{{ api_response_customer.json.contacts }}"
  set_fact:
    recipients: "{{ recipients + [{'Email': item.contact_email,
'Name': item.contact_firstname ~ ' ' ~ item.contact_lastname}] }}"

- name: Get first contact name
```

```yaml
  set_fact:
    first_contact: "{{
api_response_customer.json.contacts[0].contact_firstname }}"

- name: Send welcome email
  uri:
    url: "https://api.mailjet.com/v3.1/send"
    method: POST
    body_format: json
    headers:
      Content-Type: "application/json"
    body:
      {
        "Messages": [{
          "From": {
            "Email": "{{ email_from }}",
            "Name": "{{ email_from_name }}"
          },
          "To": "{{ recipients }}",
          "Subject": "{{ email_subject }}",
          "TextPart": "Dear {{ first_contact }}, welcome! Your
service is ready.",
          "HTMLPart": "Dear {{ first_contact
}},<br/><h3>Welcome!</h3><br/>Your {{ package_name }} service is now
active.<br/>Monthly cost: ${{ monthly_cost }}<br/>Setup fee: ${{
setup_cost }}<br/>If you have any questions, contact
support@example.com"
        }]
      }
    user: "{{ mailjet_api_key }}"
    password: "{{ mailjet_api_secret }}"
    force_basic_auth: true
    status_code: 200
  register: email_response
```

Post Provisioning Tasks

`post_provisioning_tasks.yaml` - Standard cleanup and notifications run after every provision.

```yaml
# This file is included at the end of most provisioning playbooks
# It handles common post-provisioning operations

- include_tasks: task_notify_ocs.yaml
```

The `task_notify_ocs.yaml` might contain:

```yaml
- name: Notify OCS of provisioning completion
```

```yaml
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "APIerSv1.ReloadCache",
        "params": [{
          "ArgsCache": "*all"
        }]
      }
    status_code: 200
  ignore_errors: true
```

Common Operations

Working with Inventory

Retrieving inventory details:

```yaml
- name: Get SIM Card inventory ID
  set_fact:
    sim_inventory_id: "{{ hostvars[inventory_hostname]['SIM Card'] | int }}"
  when: "'SIM Card' in hostvars[inventory_hostname]"

- name: Get SIM Card details
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{ sim_inventory_id }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: sim_response

- name: Extract SIM details
  set_fact:
    iccid: "{{ sim_response.json.iccid }}"
    imsi: "{{ sim_response.json.imsi }}"
    ki: "{{ sim_response.json.ki }}"
```

Assigning inventory to customer:

```yaml
- name: Assign SIM to customer
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{ sim_inventory_id }}"
```

```yaml
    method: PATCH
    headers:
      Authorization: "Bearer {{ access_token }}"
    body_format: json
    body:
      {
        "customer_id": {{ customer_id }},
        "service_id": {{ service_id }},
        "item_state": "Assigned"
      }
    status_code: 200
```

Date and Time Operations

Getting current date/time:

```yaml
- name: Get current date and time in ISO 8601 format
  command: date --utc +%Y-%m-%dT%H:%M:%S%z
  register: current_date_time

- name: Get today's date only
  set_fact:
    today: "{{ lookup('pipe', 'date +%Y-%m-%d') }}"
```

Calculating future dates:

```yaml
- name: Calculate expiry date 30 days from now
  command: "date --utc +%Y-%m-%dT%H:%M:%S%z -d '+30 days'"
  register: expiry_date

- name: Calculate date 90 days in future
  command: "date --utc +%Y-%m-%d -d '+{{ days }} days'"
  register: future_date
  vars:
    days: 90
```

Generating Random Values

UUIDs and identifiers:

```yaml
- name: Generate UUID
  set_fact:
    uuid: "{{ 99999999 | random | to_uuid }}"

- name: Generate service identifier
  set_fact:
    service_uuid: "SVC_{{ uuid[0:8] }}"
```

Random passwords:

```yaml
- name: Generate secure password
  set_fact:
    password: "{{ lookup('pipe', 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 16') }}"
```

Memorable passphrases:

```yaml
- name: Set word list
  set_fact:
    words:
      - alpha
      - bravo
      - charlie
      - delta
      - echo

- name: Generate passphrase
  set_fact:
    word: "{{ words | random }}"
    number: "{{ 99999 | random(start=10000) }}"

- name: Combine into passphrase
  set_fact:
    passphrase: "{{ word }}{{ number }}"
```

Working with CGRateS/OCS

Creating accounts:

```yaml
- name: Create billing account
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV2.SetAccount",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}",
          "ActionPlanIds": [],
          "ActionPlansOverwrite": true,
          "ExtraOptions": {
            "AllowNegative": false,
            "Disabled": false
          },
```

```
          "ReloadScheduler": true
        }]
      }
    status_code: 200
  register: account_response
```

Adding balances:

```
- name: Add data balance
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.AddBalance",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}",
          "BalanceType": "*data",
          "Categories": "*any",
          "Balance": {
            "ID": "Data Package",
            "Value": 10737418240,
            "ExpiryTime": "+720h",
            "Weight": 10
          }
        }]
      }
    status_code: 200
```

Executing actions:

```
- name: Execute charging action
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "APIerSv1.ExecuteAction",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}",
          "ActionsId": "Action_Standard_Charge"
        }]
      }
    status_code: 200
```

Getting account information:

```
- name: Get account details
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV2.GetAccount",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}"
        }]
      }
    status_code: 200
  register: account_info
```

Working with Attribute Profiles:

```
- name: Get AttributeProfile
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "APIerSv1.GetAttributeProfile",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "ID": "ATTR_{{ service_uuid }}"
        }]
      }
    return_content: yes
    status_code: 200
  register: attr_response
  ignore_errors: true

- name: Extract attribute value
  set_fact:
    phone_number: "{{ attr_response.json.result.Attributes |
json_query(\"[?Path=='*req.PhoneNumber'].Value[0].Rules\") | first
}}"
  when: attr_response is defined
```

Conditional Logic

Checking if variables exist:

```yaml
- name: Use custom value or default
  set_fact:
    monthly_cost: "{{ custom_cost | default(50.00) }}"

- name: Only run if variable is defined
  debug:
    msg: "Service UUID is {{ service_uuid }}"
  when: service_uuid is defined
```

Boolean conditions:

```yaml
- name: Provision equipment
  include_tasks: configure_cpe.yaml
  when: provision_cpe | default(false) | bool

- name: Skip if deprovision
  assert:
    that:
      - action != "deprovision"
  when: action is defined
```

Multiple conditions:

```yaml
- name: Complex conditional task
  uri:
    url: "{{ endpoint }}"
    method: POST
  when:
    - service_uuid is defined
    - customer_id is defined
    - action != "deprovision"
    - enable_feature | default(true) | bool
```

Loops and Iteration

Simple loops:

```yaml
- name: Create multiple balances
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.AddBalance",
        "params": [{
          "Account": "{{ service_uuid }}",
          "BalanceType": "{{ item.type }}",
```

```yaml
          "Balance": {
            "Value": "{{ item.value }}"
          }
        }]
      }
  loop:
    - { type: "*voice", value: 3600 }
    - { type: "*data", value: 10737418240 }
    - { type: "*sms", value: 100 }
```

Looping over API responses:

```yaml
- name: Get all customer sites
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/site/customer_id/{{
customer_id }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
  register: sites_response

- name: Configure equipment at each site
  debug:
    msg: "Configuring site at {{ item.address_line_1 }}"
  loop: "{{ sites_response.json }}"
```

Error Handling

Using ignore_errors:

```yaml
- name: Optional SMS notification
  uri:
    url: "http://sms-gateway/send"
    method: POST
    body: {...}
  ignore_errors: true
```

Assertions for validation:

```yaml
- name: Verify API response
  assert:
    that:
      - response.status == 200
      - response.json.result == "OK"
    fail_msg: "API call failed: {{ response.json }}"
```

Conditional error handling:

```yaml
- name: Try to get existing service
```

```yaml
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/service/service_uuid/{{
service_uuid }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
  register: service_lookup
  failed_when: false

- name: Create service if it doesn't exist
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/service/"
    method: PUT
    body: {...}
  when: service_lookup.status == 404
```

Best Practices

Variable Naming

Use descriptive, consistent names:

```yaml
# Good
service_uuid: "SVC_12345"
customer_name: "John Smith"
monthly_cost: 49.99

# Bad
svc: "SVC_12345"
name: "John Smith"
cost: 49.99
```

Prefix variables by source:

```yaml
api_response_customer: {...}
api_response_product: {...}
cgr_account_info: {...}
```

Debugging

Print variables for troubleshooting:

```yaml
- name: Print all variables
  debug:
    var: hostvars[inventory_hostname]

- name: Print specific variable
  debug:
```

```yaml
    msg: "Service UUID: {{ service_uuid }}"

- name: Print API response
  debug:
    var: api_response_product.json
```

Validation

Always validate critical API responses:

```yaml
- name: Create account
  uri:
    url: "{{ billing_endpoint }}"
    method: POST
    body: {...}
  register: response

- name: Verify account creation
  assert:
    that:
      - response.status == 200
      - response.json.result == "OK"
    fail_msg: "Failed to create account: {{ response.json }}"
```

Idempotency

Design tasks to be safely re-runnable:

```yaml
# Check if resource exists first
- name: Check if account exists
  uri:
    url: "{{ ocs_endpoint }}/get_account"
    method: POST
    body: {"Account": "{{ service_uuid }}"}
  register: account_check
  failed_when: false

# Only create if doesn't exist
- name: Create account
  uri:
    url: "{{ ocs_endpoint }}/create_account"
    method: POST
    body: {...}
  when: account_check.status == 404
```

Security

Never hardcode credentials:

```yaml
# Bad
mailjet_api_key: "abc123def456"

# Good - use environment variables
mailjet_api_key: "{{ lookup('env', 'MAILJET_API_KEY') }}"

# Good - use config file
mailjet_api_key: "{{ crm_config.email.api_key }}"
```

Always use HTTPS and authentication:

```yaml
- name: Call external API
  uri:
    url: "https://api.example.com/endpoint"
    method: POST
    headers:
      Authorization: "Bearer {{ access_token }}"
    validate_certs: yes
```

Documentation

Document complex logic:

```yaml
# Calculate pro-rata charge for partial month
# If customer signs up on the 15th and billing is on 1st,
# charge 50% of monthly cost for remaining days
- name: Calculate days until end of month
  command: "date -d 'last day of this month' +%d"
  register: days_in_month

- name: Get current day
  command: "date +%d"
  register: current_day

- name: Calculate pro-rata amount
  set_fact:
    days_remaining: "{{ (days_in_month.stdout | int) -
(current_day.stdout | int) }}"
    pro_rata_cost: "{{ (monthly_cost | float) * (days_remaining |
float) / (days_in_month.stdout | float) }}"
```

Testing Playbooks

Testing Approach

1. **Dry Run First**: Test with non-production systems
2. **Verify Variables**: Use debug tasks to confirm all required variables are present

3. **Check Responses**: Validate API responses before proceeding
4. **Rollback Testing**: Intentionally fail tasks to verify rescue blocks work
5. **Deprovision Testing**: Test with `action: "deprovision"` to verify cleanup

Example test playbook:

```yaml
- name: Test Service Provisioning
  hosts: localhost
  gather_facts: no

  tasks:
    - name: Verify required variables
      assert:
        that:
          - product_id is defined
          - customer_id is defined
          - access_token is defined
        fail_msg: "Missing required variables"

    - name: Test API connectivity
      uri:
        url: "http://localhost:5000/crm/health"
        method: GET
      register: health_check

    - name: Verify health check
      assert:
        that:
          - health_check.status == 200
```

Common Pitfalls

Missing type conversions:

```yaml
# Wrong - may be string
customer_id: "{{ customer_id }}"

# Correct - ensure integer
customer_id: {{ customer_id | int }}
```

Not handling undefined variables:

```yaml
# Wrong - fails if not defined
service_uuid: "{{ service_uuid }}"

# Correct - provide default
service_uuid: "{{ service_uuid | default('') }}"
```

Forgetting validation:

```yaml
# Wrong - doesn't check response
- name: Create account
  uri: ...
  register: response

# Correct - validate response
- name: Create account
  uri: ...
  register: response

- name: Verify creation
  assert:
    that:
      - response.json.result == "OK"
```

Provisioning Workflow

Generally, Omnitouch staff will work with the customer to:

1. Define the product requirements
2. Develop the necessary Ansible playbooks to automate the provisioning process
3. Test the playbooks in a staging environment
4. Deploy to production

This ensures that each service is deployed consistently and reliably, reducing the risk of errors and ensuring that all necessary steps are completed in the correct order.

Ansible Variables

The variables passed to Ansible playbooks include:

Product Variables
Derived from the OmniCRM product configurations and define how the service should be set up.

Inventory Variables
Selected from the inventory, these include items such as modems, SIM cards, IP address blocks, or phone numbers that are required for provisioning.

System Variables
Automatically added by OmniCRM:

- `product_id` - The product being provisioned

- `customer_id` - The customer receiving the service
- `service_id` - The service being modified (for topups/changes)
- `access_token` - JWT for API authentication

Deprovisioning

When a service is no longer needed, the **Ansible Playbooks** are also used to deprovision the service using the `rescue` block pattern. This:

- Removes any configurations
- Releases inventory back to the pool
- Deletes billing accounts
- Ensures the system is kept clean

Rollback and Error Handling

Ansible's **block/rescue** feature is employed during both provisioning and deprovisioning to handle errors gracefully. If a task fails at any point during provisioning, the rescue section automatically rolls back changes to return to a consistent state. This ensures reliability and reduces the risk of partial or failed deployments.

For complete details on the provisioning system, workflows, and authentication, see `concepts_provisioning`.

OmniCRM API

All functions within OmniCRM are accessible via the API - There is no functionality that is only available in the UI.

This allows you to integrate OmniCRM with other systems or automate tasks.

The API is a RESTful API, and is secured using multiple authentication methods including JWT tokens, API keys, and IP whitelisting.

The API is documented using Swagger, a tool that allows easy reading, understanding, and testing of API functionality.

The API documentation is available at the following URL:

> <https://yourcrm/crm/docs/>

OmniCRM Swagger Page

Authentication Methods

OmniCRM supports three authentication methods, each designed for different use cases:

1. **JWT Bearer Tokens** - For interactive user sessions (Web UI, mobile apps)
2. **API Keys** - For server-to-server integrations and automation scripts
3. **IP Whitelisting** - For trusted internal systems (provisioning servers, monitoring tools)

JWT Bearer Token Authentication

This is the primary authentication method for user sessions. Users log in with email and password, receive a JWT token, and use it for subsequent requests.

Use Cases:

- Web UI authentication
- Mobile app authentication
- Short-lived programmatic access

How to Authenticate:

To log in, send a JSON body with the following structure to `/crm/auth/login` as a POST request:

```json
{
    "email": "youruser@yourdomain.com",
    "password": "yourpassword"
}
```

The API will return a JSON object containing a `token` field, which is used to authenticate all future requests. Additionally, the response includes a `refresh_token` that can be used to refresh the token when it expires, along with the user's permissions and roles.

You can test this from the Swagger page by selecting the `/auth/login` endpoint, filling in your Username and Password, and clicking the `Try it out` button.

OmniCRM Swagger Page

To authorize the session, copy the token value and click the "Authorize" button at the top-right of the Swagger page. Paste the token into the "Value" field, prefixed by `Bearer` and click "Authorize".

OmniCRM Swagger Bearer Auth

Now, all subsequent requests will be authenticated with this token.

API Key Authentication

API keys provide secure, long-lived authentication for server-to-server integrations and automation scripts without requiring user passwords.

Use Cases:

- Automated provisioning systems
- Monitoring and alerting tools
- Integration with external systems
- Scheduled tasks and cron jobs

How API Keys Work:

API keys are configured in the `crm_config.yaml` file and are associated with specific roles and permissions. Each API key is a secure random string (minimum 32 characters) that authenticates requests when passed in the `X-API-KEY` header.

Configuring API Keys:

API keys must be added to `crm_config.yaml` by an administrator with server access:

```yaml
api_keys:
  your-secure-api-key-here-minimum-32-chars:
    roles:
```

```yaml
      - admin
    description: "Provisioning automation system"
  another-api-key-for-monitoring-system:
    roles:
      - view_customer
      - view_service
    description: "Monitoring and alerting"
```

Using API Keys:

Include the API key in the `X-API-KEY` header of your requests:

```
curl -X GET "https://yourcrm.com/crm/customers" \
  -H "X-API-KEY: your-secure-api-key-here-minimum-32-chars"
```

Python Example:

```python
import requests

crm_url = 'https://yourcrm.com'
api_key = 'your-secure-api-key-here-minimum-32-chars'

headers = {
    "Content-Type": "application/json",
    "X-API-KEY": api_key
}

# Get Customers
response = requests.get(crm_url + '/crm/customers', headers=headers)
for customer in response.json()['data']:
    print(customer)
```

Best Practices:

- Generate API keys using cryptographically secure random generators (`openssl rand -base64 48`)
- Use different API keys for different systems
- Document the purpose of each API key in the `description` field
- Rotate API keys periodically
- Never commit API keys to version control
- Assign minimal necessary permissions to each API key

IP Whitelist Authentication

IP whitelisting allows specific IP addresses to access the API without authentication. This is useful for trusted internal systems on private networks.

Use Cases:

- Internal provisioning servers
- Network monitoring systems on management VLANs
- Ansible playbooks running on controlled infrastructure

Configuring IP Whitelist:

Add trusted IP addresses to `crm_config.yaml`:

```yaml
ip_whitelist:
  - 192.168.1.100
  - 10.0.0.0/24
  - 172.16.50.10
```

Security Considerations:

- Only use IP whitelisting on private, secured networks
- Never whitelist public IP addresses
- Use the most specific IP ranges possible
- Document why each IP is whitelisted
- Regularly audit whitelisted IPs

Example API Calls with Python

Here is an example of how to log in and retrieve a list of customers using JWT token authentication:

```python
import requests

crm_url = 'https://yourcrm.com'
session = requests.Session()

print("Provisioning data to server: " + str(crm_url))

headers = {
    "Content-Type": "application/json"
}

# Get Auth Token
response = session.post(crm_url + '/crm/auth/login', json={
    "email": "youruser@yourdomain.com",
    "password": "yourpassword"
}, headers=headers)

print(response.status_code)
print(response.json())
assert response.status_code == 200

headers['Authorization'] = 'Bearer ' + response.json()['token']
```

```python
print("Authenticated to CRM successfully")

# Get Customers
response = session.get(crm_url + '/crm/customers', headers=headers)
for customer in response.json()['data']:
    print(customer)
```

CRM Service / Product Charging Notes

Note

For a complete end-to-end guide covering product definition, provisioning, addons, and deprovisioning with detailed Ansible examples and pricing strategy, see Complete Product Lifecycle Guide <guide_product_lifecycle>.

Products & Services Overview

Product (Menu Item):

A Product is like a specific dish on a restaurant menu, such as a "Spaghetti Carbonara."

It has a clear description, a list of ingredients (like pasta, cream, eggs, cheese, and bacon), and a price.

In OmniCRM, a Product similarly contains the details of what is included — features, specifications, and pricing.

Often, customers may want modifications, like "hold the onions" or "add extra cheese" to their meal. Within OmniCRM, this corresponds to customizing a service before creation. The level of customizations or modifications to a service are up to you (the operator) to define.

In OmniCRM, customers or staff might modify a Product to better suit a specific customer's needs, such as upgrading their Internet speed or adding specific features. This customization is reflected in the specific Service provided.

A product is essentially an offering that customers can choose to order from, similar to reading and picking a dish from the menu.

OmniCRM Product Defintions

Product Catalog (Restaurant Menu):

The Product Catalog is like the entire menu in a restaurant, which lists all the dishes available — from appetizers to desserts.

It is the complete collection of everything the restaurant (or in your case, the Service Provider) has to offer.

In the business context, the Product Catalog provides customers with all available Products, so they can choose the one that best meets their needs.

Product Management Interface Product Catalog Edit View

Service (Prepared Dish):

When a customer orders an item from the menu, the dish is prepared in the kitchen. This is akin to creating a Service from a Product.

In OmniCRM, when a customer selects a Product, an instance of that Product is created and delivered as a Service.

It is customized and prepared specifically for that customer, just like a meal prepared for a diner.

For example, when someone selects the "Internet Bronze Plan" from the Product Catalog, the provisioning system "cooks" up an instance of that plan from the ingredients (IP addresses, Modems and Ports) — i.e., activates the plan and delivers it to the specific customer.

Bundled Products (Combo Meals):

The Product Catalog might also offer bundles, like a combo meal that includes an appetizer, main course, and dessert together for a special price.

In OmniCRM, bundled Products combine multiple individual Products into one convenient package — like a "Home Essentials Bundle" that includes Internet, cable, and phone services at a discounted rate.

Once selected, this bundle is turned into multiple Services tailored for the customer.

Product Definitions

A product is a template that is used to create a service / addon / discount / bolt on, etc.

Inside the definition we include:

- Information about the product (features, inclusions, T&Cs, contract length, icon, etc) that is displayed to the user of the CRM (Customer or *Staff*).
- The business logic around who can purchase the product (*Business* or *Residential*), if it relies on having a parent service provisioned (like mobile addons only available to customers with a mobile service), if it can be ordered directly by a customer via self care or only by a customer service agent, and when the product can be purchased (Allowing for a product to only be available for a set period of time).
- When Inventory items are to be included (such as Modems or SIM Cards) these are specified as Inventory Items List, for example the below service requires a SIM Card and a Phone Number to be assigned:

  `['SIM Card', 'Phone Number']` These correlate to the `Inventory Items <administration_inventory>` defined in the CRM.
- Reference an Ansible Playbook to provision the service `Provisioning Play <concepts_ansible>` as well as the variables to pass to Ansible. These variables to pass are magic, in that they may be variables like `service_id` that are defined by the product we're adding it to, or they may be like `ICCID` & `MSISDN` where we have selected inventory items that are passed over when assigning the inventory. Bundling is handled in the provisioning play to contain multiple services, for example a bundled home internet, TV & Voice product, may provision a service for each.

OmniCRM Product Definitions

Product Categories and Service Types

Products use two classification fields to help organize and filter offerings:

Product Categories

The `category` field controls where products are displayed in the UI. Common values include:

- **standalone** - Shown as a base service option when creating a new service
- **addon** - Shown when adding to an existing service
- **bundle** - Shown as a bundled service option (provisioned like an addon to existing services)
- **promo** - Special promotional offerings

These categories are purely organizational and don't dictate what gets provisioned. The actual provisioning behavior is determined entirely by the Ansible playbook referenced in `provisioning_play`.

For example: - A `standalone` product typically creates a new service object - An `addon` or `bundle` product is typically added to an existing service - But this is up to the implementer writing the playbook - you could create multiple service objects from an addon, or modify existing services from a standalone product if needed

The category simply controls the UI flow and where customers/staff see the product option.

Service Types

The `service_type` field categorizes what kind of service is being provided.

These are entirely defined by the user, but common values include:

- **mobile** - Mobile phone services with voice, SMS, and data
- **fixed** - Fixed wireless or wired internet services
- **fixed-voice** - Fixed-line voice services (VoIP, landline)
- **hotspot** - Mobile hotspot or rental devices
- **dongle** - USB modem or dongle services
- **voice** - Voice-only services
- **data** - Data-only services

Like categories, service types are customizable based on your offerings. They help in:

- Filtering which addons apply to which base services
- Organizing products in the customer portal
- Matching inventory requirements
- Determining provisioning workflows

Example: A customer with a `mobile` service can see mobile addons, while a customer with a `fixed` service sees fixed-line addons.

Managing Products

Products are managed through the Product Management page, where you can view, search, filter, and edit all available products.

Product List Page

Product Modal Interface

Clicking on any product opens an enhanced tabbed interface that organizes all product settings into logical groups for easier navigation and editing.

Product Modal - Basic Info Tab

The product management modal features five organized tabs:

1. 🔷 **Basic Info** - Core product information (name, slug, category, icon, features, terms)
2. 🔷 **Pricing** - All cost-related fields including recurring costs, setup costs, and tax percentage
3. ⚙ **Configuration** - Renewal settings, customer types, and dependencies
4. 🔷 **Provisioning** - Ansible playbook configuration and inventory requirements
5. 🔷 **Availability** - Date ranges and system timestamps

Product Modal - Pricing Tab

Pricing Tab Organization:

The Pricing tab groups cost fields into logical sections:

- **Recurring Costs** - Monthly retail and wholesale costs side-by-side
- **Setup Costs** - One-time activation fees for retail and wholesale
- **Tax** - Tax percentage configuration with automatic calculation

Edit Mode Features:

- **Icon Picker** - Search and select FontAwesome icons visually
- **Inventory Items Picker** - Select from available inventory item types
- **Date/Time Picker** - Easy selection of availability windows
- **Currency Formatting** - Automatic $ prefix for cost fields
- **Dropdown Selectors** - Pre-defined options for categories and boolean fields

Product Modal - Basic Info Edit Mode

Icon Picker:

When editing the icon field, a searchable icon picker interface appears allowing you to visually browse and select from thousands of FontAwesome icons.

Product Modal - Icon Picker

Features: * Search icons by keyword (e.g., "wrench", "mobile", "wifi") * Preview icon appearance in real-time * Shows icon class name for reference * Dropdown selection for quick access

Configuration Tab:

The Configuration tab organizes product behavior settings into logical groups.

Product Modal - Configuration Tab

Configuration Sections:

- **Renewal Settings:**
  - Auto Renew - Default renewal behavior (Prompt/Yes/No)
  - Allow Auto Renew - Whether customers can enable auto-renewal
  - Contract Days - Minimum contract length (e.g., 30 for monthly, 365 for annual)
- **Customer Types:**
  - Residential - Available to consumer customers
  - Business - Available to commercial customers
- **Dependencies:**
  - Relies On List - Product IDs or service types required before this product can be

    added
    - Used for addon dependencies (e.g., mobile addons require active mobile service)

Provisioning Tab:

The Provisioning tab handles Ansible automation and inventory requirements.

Product Modal - Provisioning Tab

Provisioning Fields:

- **Provisioning Play:**
    - Name of Ansible playbook (without .yaml extension)
    - Must exist in OmniCRM-API/Provisioners/plays/ directory
    - Called when service is created, updated, or deprovisioned
- **Provisioning JSON Vars:**
    - Default variables passed to Ansible playbook as JSON
    - Can be overridden during provisioning
    - Playbook receives these plus customer_id, product_id, service_id, access_token
- **Inventory Items List:**
    - Multi-select picker showing available inventory item types
    - Examples: SIM Card, Phone Number, Modem Router, IPv4 Address
    - Customer/staff selects specific items from available inventory during ordering
    - Selected inventory IDs passed to playbook with inventory type as variable name

Availability Tab:

The Availability tab controls when the product can be purchased and displays system metadata.

Product Modal - Availability Tab

Availability Settings:

- **Available From:**
    - Date/time when product becomes available for purchase
    - Leave empty for immediate availability
    - Useful for pre-announcing new products
- **Available Until:**
    - Date/time when product is no longer available for purchase
    - Leave empty for indefinite availability
    - Perfect for limited-time promotions or end-of-life products
- **System Metadata (Read-Only):**
    - Created - Timestamp when product was first created
    - Last Modified - Timestamp of most recent update
    - Automatically maintained by the system

Modal Actions:

- **View Mode:**
    - Close - Dismiss modal
    - Clone Product - Create a copy with "_clone" suffix
    - Edit Product - Switch to edit mode
- **Edit/Create Mode:**
    - Cancel - Discard changes and close
    - Save Changes - Create or update product (large button for emphasis)

Product Fields

The Product model contains all the information needed to define an offering and how it should be provisioned. These fields are managed through the Product Management modal interface described above.

Basic Information

- **product_id** - Unique identifier automatically assigned by the system
- **product_name** - Display name shown to customers and staff in the UI
- **product_slug** - Unique identifier used in URLs and API calls (lowercase, no spaces, use hyphens)
- **category** - Controls where this product appears in the UI:
    - `standalone` - Shown as a base service option when creating a new service
    - `addon` - Shown when adding to an existing service
    - `bundle` - Shown as a bundled service option
    - `promo` - Special promotional offerings
- **service_type** - Type of service being provided (e.g., mobile, fixed, fixed-voice, hotspot, dongle, voice, data). Used to filter which addons apply to which services.
- **comment** - Internal notes about the product for staff reference only (not shown to customers)
- **icon** - FontAwesome icon class displayed in the UI (e.g., `fa-solid fa-sim-card`)

Pricing Fields

- **retail_cost** - Monthly recurring charge billed to the customer (set to 0 for one-time purchases or prepaid products)
- **wholesale_cost** - Your monthly cost for providing this service (used for margin calculations)
- **retail_setup_cost** - One-time activation or setup fee charged to the customer
- **wholesale_setup_cost** - Your one-time cost for setting up the service
- **tax_percentage** - Tax percentage applied to this product (e.g., 10 for 10%, 12.5 for 12.5%). Set to 0 for tax-exempt products. This tax rate is automatically applied to transactions created from this product.

Product Tax Configuration

Tax Application:

When a transaction is created from this product, the tax percentage is automatically copied to the transaction and the tax amount is calculated. For example:

- Product with 10% tax, $50.00 retail cost → Transaction has $5.00 tax
- Product with 0% tax (tax-exempt) → Transaction has $0.00 tax
- Manual transaction override → Staff can change tax percentage per transaction

Customer Visibility and Access

- **enabled** - Whether this product is active and available for purchase (set to false to hide without deleting)
- **residential** - Whether residential (consumer) customers can purchase this product
- **business** - Whether business (commercial) customers can purchase this product
- **customer_can_purchase** - Whether customers can self-purchase via the portal (true) or if only staff can add it (false)
- **available_from** - Date/time when this product becomes available for purchase (optional)
- **available_until** - Date/time when this product is no longer available for purchase (optional, useful for limited-time offers)

Contract and Renewal

- **contract_days** - Minimum contract length in days (e.g., 30 for monthly, 365 for annual, 0 for no minimum contract)
- **auto_renew** - Default renewal behavior:
    - `prompt` - Asks customer each time whether to renew
    - `true` - Automatically renews without asking
    - `false` - Requires manual renewal
- **allow_auto_renew** - Whether customers can enable automatic renewal (set to false for one-time purchases)

Customer-Facing Content

- **terms** - Terms and conditions displayed to customers before purchase (include limitations, expiry rules, usage conditions)
- **features_list** - List of features and inclusions shown to customers (Python list format: `['Feature 1', 'Feature 2']`)

Provisioning Configuration

- **provisioning_play** - Name of the Ansible playbook that provisions this service (without .yaml extension). Must exist in `OmniCRM-API/Provisioners/plays/` directory.
- **provisioning_json_vars** - Default variables passed to the Ansible playbook as JSON. These can be overridden when provisioning. The playbook receives these along with `customer_id`, `product_id`, `service_id`, and `access_token`.
- **inventory_items_list** - List of inventory items required for this product (e.g., `['SIM Card', 'Mobile Number']`). When a customer orders, they'll be prompted to select specific items from available inventory. Selected inventory IDs are passed to the provisioning playbook with the inventory type as the variable name.
- **relies_on_list** - List of product IDs or service types that must exist before this product can be added. Used for addon dependencies (e.g., mobile addons require an active mobile service).

System Metadata

- **created** - Timestamp when the product was created (automatically set)
- **last_modified** - Timestamp when the product was last updated (automatically updated)

Example Product Definitions

Standalone Product (Mobile SIM)

OmniCRM Product Definitions

```
{
  "product_id": 1,
  "product_slug": "Mobile-SIM",
  "product_name": "Mobile SIM Only",
  "category": "standalone",
  "service_type": "mobile",
  "provisioning_play": "play_psim_only",
  "provisioning_json_vars": "{\"iccid\": \"\", \"msisdn\": \"\"}",
  "inventory_items_list": "['SIM Card', 'Mobile Number']",
  "retail_cost": 0,
  "retail_setup_cost": 0,
  "wholesale_cost": 3,
  "wholesale_setup_cost": 1,
```

```
  "contract_days": 0,
  "residential": true,
  "business": true,
  "enabled": true,
  "customer_can_purchase": true,
  "icon": "fa-solid fa-sim-card",
  "features_list": "['Australian Phone Number (04xxx)', 'Fastest speeds', 'Best coverage', 'Roaming on the Mainland']",
  "terms": "Must be activated within 6 months. All credit lost if service is not used for 12 months.",
  "comment": "Physical SIM card for use with Mobile Phones"
}
```

This standalone product requires two inventory items (SIM Card and Mobile Number) and creates a new service when provisioned.

Addon Product (Monthly Data Plan)

```
{
  "product_slug": "norfone-mobile-prepaid-mini",
  "product_name": "Norfone Mini Plan",
  "category": "addon",
  "service_type": "mobile",
  "provisioning_play": "play_topup_charge_then_action",
  "provisioning_json_vars": "",
  "inventory_items_list": "[]",
  "retail_cost": 30,
  "retail_setup_cost": 0,
  "wholesale_cost": 5.84,
  "contract_days": 30,
  "residential": true,
  "business": false,
  "enabled": true,
  "customer_can_purchase": true,
  "auto_renew": "prompt",
  "icon": "fa-solid fa-sim-card",
  "features_list": "['8GB of Ultra fast data', 'Unlimited Calls & Texts to Norfone users', '100 Minutes of Talk to Australia', '100 SMS to Australia', '30 Day Expiry']",
  "terms": "Credit expires after 30 days. Once data, voice or sms is used up, you will need to top up to continue using the service.",
  "comment": "Our smallest plan for light users"
}
```

This addon product doesn't require inventory and is applied to an existing service. It charges the customer and adds credits/balances to their service.

Bundle Product (Seniors Package)

```
{
  "product_slug": "Bundle-Seniors",
  "product_name": "Seniors Bundle",
  "category": "bundle",
  "service_type": "fixed",
  "provisioning_play": "play_seniors_package",
  "provisioning_json_vars": "{\"IPTV_Service_ID\": \"SeniorBundle\"}",
  "inventory_items_list": "['Modem Router']",
  "retail_cost": 30,
```

```
  "retail_setup_cost": 0,
  "wholesale_cost": 10,
  "wholesale_setup_cost": 11,
  "contract_days": 180,
  "residential": true,
  "business": false,
  "enabled": true,
  "icon": "fa-solid fa-person-walking-with-cane",
  "features_list": "['20Mbps Download', '5Mbps Upload', 'Unlimited Data', 'Home
Voice', 'TV: Extra +£5 per month', '£60 Installation Fee']",
  "terms": "6 Month Contract, must show senior citizen's card to qualify",
  "comment": "20Mbps/2Mbps GPON Service + IPTV + Phone"
}
```

This bundle product provisions multiple services (Internet + IPTV + Phone) using a single playbook. It requires one inventory item (Modem Router).

Addon Product (Simple Top-up)

```
{
  "product_slug": "Mobile-Topup-5",
  "product_name": "PAYG £5 Topup",
  "category": "addon",
  "service_type": "mobile",
  "provisioning_play": "play_topup_monetary",
  "provisioning_json_vars": "{\"service_id\": \"\"}",
  "inventory_items_list": "[]",
  "retail_cost": 5,
  "retail_setup_cost": 0,
  "wholesale_cost": 0,
  "contract_days": 0,
  "residential": true,
  "business": false,
  "enabled": true,
  "customer_can_purchase": true,
  "icon": "fa-solid fa-coins",
  "features_list": "['£5 credit', 'Valid for 180 days']",
  "terms": "Valid for 180 days or until all credit is used. See our website for
full rates",
  "comment": "£5 to use for Calls, SMS & Data"
}
```

This addon simply adds monetary credit to an existing service. No inventory required, and it uses `service_id` to identify which service to top up.

How Variables are Passed to Ansible

Understanding how variables flow from the product definition through the API to the Ansible playbook is critical for writing effective provisioning playbooks.

Variable Sources and Merging

When a provisioning job is created, variables come from multiple sources and are merged together in this order (later sources override earlier ones):

1. **Product's provisioning_json_vars** - Default variables from the product definition
2. **Request body** - Variables passed in the API call (can override product defaults)

3. **System-added variables** - Automatically added by the provisioning system
4. **Inventory selections** - IDs of selected inventory items (if `inventory_items_list` is not empty)

Variable Merging Process

The system merges variables from all sources, with later sources overriding earlier ones. This allows for flexible customization at provision time.

For example, if your product has:

```
"provisioning_json_vars": "{\"monthly_cost\": 50, \"data_gb\": 100}"
```

And your API request includes:

```
{
  "product_id": 10,
  "customer_id": 456,
  "monthly_cost": 45,
  "custom_param": "value"
}
```

The final `extra_vars` passed to Ansible will be:

```
{
    "monthly_cost": 45,        # Overridden from request
    "data_gb": 100,            # From provisioning_json_vars
    "product_id": 10,          # From request
    "customer_id": 456,        # From request
    "custom_param": "value",   # From request
    "access_token": "eyJ..."   # Added by system
}
```

System-Added Variables

The provisioning system automatically adds:

- `access_token` - JWT token for authenticating API calls back to the CRM (from `g.access_token` for IP/API key auth, or generated from `refresh_token` for user auth)
- `initiating_user` - The user ID who triggered the provisioning (or first admin for automated systems)
- Any fields from the request body (`product_id`, `customer_id`, `service_id`, etc.)

Inventory Variables

When a product requires inventory items (e.g., `inventory_items_list: "['SIM Card', 'Mobile Number']"`), the process works as follows:

1. **UI/API prompts for selection** - User selects specific inventory items from available stock
2. **Inventory IDs are added to variables** - The selected inventory item IDs are added with the inventory type as the variable name
3. **Playbook accesses inventory IDs** - The provisioning playbook can then retrieve full inventory details from the CRM API

For example, if a user selects: - SIM Card with inventory_id: 789 - Mobile Number with inventory_id: 101

The variables passed to the playbook include: - `SIM Card`: 789 - `Mobile Number`: 101

The playbook can then use these IDs to fetch the complete inventory records (ICCID, IMSI, MSISDN, etc.) from the CRM API and use that information to provision the service on network equipment.

How Ansible Receives Variables

The provisioning system passes all merged variables to the Ansible playbook as `extravars`. Inside the playbook, these variables are available through Ansible's standard variable system and can be used in tasks.

Variables can be referenced directly in playbook tasks using the `{{ variable_name }}` syntax. For example, `{{ product_id }}`, `{{ customer_id }}`, `{{ monthly_cost }}`, etc.

Variables Passed to Addon Products

When an addon product is provisioned, the system automatically passes:

- `product_id` - The ID of the addon product being provisioned
- `customer_id` - The customer who owns the service
- `service_id` - The ID of the service this addon is being added to (critical for addons)
- `access_token` - Authentication token for API calls
- Any variables from `provisioning_json_vars`
- Any additional variables from the API request

Example Addon Provisioning Flow

When a customer adds the "£5 Topup" addon to their mobile service (service_id: 123), the playbook receives variables including:

- `product_id`: 45 (the topup product)
- `customer_id`: 456 (the customer)
- `service_id`: 123 (the service to add credit to)
- `access_token`: Authentication token
- Plus any variables from the product's `provisioning_json_vars`

The playbook then uses these variables to:

1. **Fetch service details** from the CRM API using the `service_id`
2. **Extract the service UUID** and other information from the service record
3. **Add credit to the billing system** (OCS) using the service UUID
4. **Record the transaction** in the CRM for billing purposes

This flow allows the addon to identify exactly which service to modify and apply the changes appropriately.

Difference Between Standalone and Addon Variables

Standalone Products receive:

- `product_id` - The product being provisioned
- `customer_id` - The customer ordering the service
- Inventory item IDs (e.g., `SIM Card`, `Mobile Number`) if the product requires them
- `access_token` - For API authentication

Addon Products receive:

- `product_id` - The addon product being provisioned

- `customer_id` - The customer who owns the service
- `service_id` - **The ID of the existing service to modify** (this is the key difference)
- `access_token` - For API authentication

The key difference is `service_id` - this tells the playbook which existing service to modify or add to.

Bundle Products

Bundle products are provisioned like addons but their playbook may create multiple service records. They receive the same variables as addons, including:

- `product_id` - The bundle product
- `customer_id` - The customer
- `service_id` - Parent service (if applicable)
- Inventory item IDs (e.g., `Modem Router`) if required
- `access_token` - For API authentication

The bundle playbook (e.g., `play_seniors_package`) then creates multiple related services (Internet, IPTV, Phone) and links them together.

Services

A service is an instance of a product that belongs to a customer, that they get billed for.

It is essentially a link to an `OCS </glossary>` (Online Charging System) account which handles the charge generation and the actual balances and usages for the account. The OCS is powered by CGRateS and manages monetary balances, unitary balances (data, voice, SMS), ActionPlans for auto-renewal, and ThresholdS for spending limits.

Adding a Service: Product Selection and Filtering

When adding a service to a customer (either a new standalone service or an addon to an existing service), the system displays available products in a carousel interface. The products shown are filtered based on several criteria:

Product Filtering for Standalone Services

When creating a new service for a customer, the UI displays products filtered by:

1. **Customer Type** - Products are categorized as:
    - **Individual (Residential)**: Products where `residential = true` or `business = false`
    - **Business**: Products where `business = true`
2. **Category** - Products are separated into:
    - **Service Plans**: Products with `category` = `standalone` or `bundle`
    - **Addons**: Products with `category` = `addon` (shown in separate carousel)
3. **Availability** - Products are only shown if:
    - `enabled = true` - Product is active and not disabled
    - Current date is between `available_from` and `available_until` - Product is within its availability window
    - `customer_can_purchase = true` (if customer is self-purchasing) - Product allows direct customer purchase

Note

API-Level Filtering: The API automatically filters products by enabled status and availability dates at two levels:

- **Purchase/Selection Endpoints** (`/crm/product/`) - Used by Addons modal and PlanList for product selection. Automatically filters to show ONLY enabled products within their availability date range. This ensures customers and staff can only select products that are currently available for purchase.
- **Management Endpoints** (`/crm/product/paginated`) - Used by Product Management page. Shows ALL products including disabled and outside availability dates, allowing administrators to manage the full product catalog including inactive products.

Pass `include_disabled=true` to the base product endpoint to bypass filtering (only for administrative use).

The UI displays separate carousels for:

- **Individual Service Plans** - Residential products for consumer customers
- **Business Service Plans** - Commercial products for business customers
- **Individual Addons** - Residential addon packs
- **Business Addons** - Commercial addon packs

Product Filtering for Addon Services

When adding an addon to an **existing service**, additional filtering is applied:

1. **Service Type Matching** - Only addons with matching `service_type` are shown:
    - If the existing service has `service_type = "mobile"`, only addons with `service_type = "mobile"` are displayed
    - This ensures mobile customers only see mobile addons, internet customers only see internet addons, etc.
2. **Dependency Checking** - If an addon has a `relies_on_list`:
    - The system checks if the customer has the required products/services
    - Only addons whose dependencies are satisfied are shown
3. **Same Customer Type Filter** - Addons are still filtered by `residential` vs `business` to match the customer type

Example Filtering Scenario

For a business customer with an existing mobile service (`service_type = "mobile"`):

- **Standalone Products Shown**: All business standalone/bundle products (`business = true`, `category != "addon"`)
- **Addon Products Shown**: Only business mobile addons (`business = true`, `category = "addon"`, `service_type = "mobile"`)
- **Products Hidden**: Residential products, addons for other service types (internet, voice, etc.), disabled products

Service Fields

The Service model contains fields that track the provisioned service instance and its relationship to the customer, product, and billing system.

Basic Service Information

- **service_id** - Unique identifier automatically assigned by the system (read-only)
- **customer_id** - Link to the customer who owns this service (read-only after creation)
- **product_id** - Link to the product this service was created from (read-only after creation)

- **service_name** - Display name shown to customers (editable)
- **service_type** - Type of service: mobile, internet, voip, iptv, bundle, etc. (editable)
- **service_uuid** - Unique identifier used in OCS/CGRateS for billing (read-only, auto-generated)
- **icon** - FontAwesome icon class for display in self-care portal (editable)

Service Status and Dates

- **service_status** - Current status: Active, Inactive, Suspended, etc. (editable)
- **service_provisioned_date** - When the service was first provisioned (auto-set, read-only)
- **service_active_date** - When the service became active (editable)
- **service_deactivate_date** - When the service expires or will be deactivated (editable)
- **contract_end_date** - End date of contract commitment (editable)

Billing and Pricing

- **retail_cost** - Monthly recurring charge to customer (editable)
- **wholesale_cost** - Your cost for providing the service (editable)
- **service_billed** - Whether this service appears on invoices (editable, default: true)
- **service_taxable** - Whether taxes apply to this service (editable, default: true)
- **invoiced** - Whether the service has been invoiced (auto-set by billing system)
- **promo_code** - Promotional code used when service was created (editable)

Customer Visibility

- **service_visible_to_customer** - Whether customer can see this service in self-care portal (editable, default: true)
- **service_usage_visible_to_customer** - Whether customer can view usage/balance details (editable, default: true)

Provisioning Configuration

- **provisioning_play** - Ansible playbook used to provision this service (inherited from product, read-only)
- **provisioning_json_vars** - Variables passed to provisioning playbook (inherited from product, read-only)
- **deprovisioning_play** - Ansible playbook to run when service is deprovisioned (read-only)
- **deprovisioning_json_vars** - Variables for deprovisioning playbook (read-only)

Service Relationships

- **bundled_parent** - If this service is part of a bundle, the service_id of the parent service (read-only)
- **site_id** - Link to the physical site/location where service is delivered (editable)

Notes and Metadata

- **service_notes** - Internal notes about the service for staff reference (editable)
- **created** - Timestamp when service was created (auto-set, read-only)
- **last_modified** - Timestamp of last update (auto-updated, read-only)

Editable vs Read-Only Fields

Editable via API/UI:

Services can be updated via `PATCH /crm/service/{service_id}` with these fields:

- service_name, service_type, service_status

- service_notes
- retail_cost, wholesale_cost
- service_billed, service_taxable
- service_visible_to_customer, service_usage_visible_to_customer
- service_active_date, service_deactivate_date, contract_end_date
- icon, promo_code, site_id

Read-Only (Auto-Set):

These fields cannot be directly modified after creation:

- service_id, customer_id, product_id
- service_uuid (generated during provisioning)
- service_provisioned_date
- provisioning_play, provisioning_json_vars
- deprovisioning_play, deprovisioning_json_vars
- bundled_parent
- invoiced (managed by billing system)
- created, last_modified (automatically managed)

Provisioning Products into Services

The provisioning process converts a Product (template) into a Service (customer-specific instance) through a series of coordinated steps involving the Web UI, API, and Ansible playbooks.

High-Level Provisioning Flow

1. **Pre-provisioning Setup** - Product created in API with provisioning configuration, and corresponding Ansible playbooks written and tested
2. **Service Selection** - From the Customer Page, staff or customer selects "Add Service"
3. **Product Filtering** - Displayed products filtered based on:
    - Customer type (residential/business)
    - Existing services (for addon dependencies in `relies_on_list`)
    - Availability dates (`available_from`/`available_until`)
    - `enabled` and `customer_can_purchase` flags
4. **Customization** - Option to override provisioning variables (for price adjustments, custom configurations, etc.)
5. **Inventory Selection** - If product requires inventory (`inventory_items_list` is not empty), user selects specific items (e.g., which SIM card, which phone number)
6. **Provision Initiation** - When "Provision" button is clicked, the API creates a provisioning job

Detailed API and Ansible Integration Flow

When a service is provisioned, the following sequence occurs:

Step 1: Provision Job Creation (`/routes/service.py`)

The API receives the provision request and calls `create_provisioning_job()` from `services/provisioning_service.py` with:

- `provisioning_play` - Name of the Ansible playbook (e.g., `play_psim_only`)
- `provisioning_json_vars` - JSON string of variables from the product or overridden by request
- `customer_id` - ID of the customer ordering the service
- `product_id` - ID of the product being provisioned
- `service_id` - (Optional) ID of existing service for addons

- Inventory selections - IDs of selected inventory items

Step 2: Variable Assembly (`services/provisioning_service.py`)

The provisioning service merges variables from multiple sources in this order:

1. Product's `provisioning_json_vars` (defaults from product definition)
2. Request body parameters (can override product defaults)
3. System-added variables:
    - `access_token` - JWT token for API authentication back to CRM
    - `initiating_user` - User ID who triggered provisioning
    - `customer_id`, `product_id`, `service_id`
4. Inventory selections - Added as `{inventory_type: inventory_id}` pairs

Example merged variables:

```
{
    "customer_id": 123,
    "product_id": 456,
    "service_id": 789,          # Only for addons
    "SIM Card": 1001,           # From inventory selection
    "Mobile Number": 1002,      # From inventory selection
    "monthly_cost": 30,         # From provisioning_json_vars
    "data_gb": 50,              # From provisioning_json_vars
    "access_token": "eyJ...",   # System-added for API callbacks
    "initiating_user": 5        # System-added
}
```

Step 3: Provision Record Creation (`models.py` - Provision model)

A `Provision` record is created in the database with:

- `provision_id` - Unique identifier for tracking
- `provisioning_play` - Playbook filename
- `provisioning_json_vars` - Merged variables as JSON string
- `task_count` - Number of tasks in the playbook (extracted from YAML)
- `provisioning_status` - Status code (initially set to 1 = running, then updated to 0 = success, 2 = failed, or may remain 1 if still in progress)
- `product_id`, `customer_id`, `service_id` - Context references

Step 4: Background Playbook Execution (`Provisioners/playbook_runner_v2.py`)

The API spawns a background thread that:

1. Loads the playbook YAML from `OmniCRM-API/Provisioners/plays/{playbook_name}.yaml`
2. Calls `ansible_runner.run()` with:
    - `playbook` - Path to the loaded YAML file
    - `extravars` - All merged variables (passed to Ansible)
    - `inventory` - Set to `'localhost,'` (local execution)
    - `event_handler` - Custom handler to capture task execution events
3. Monitors playbook execution in real-time

Step 5: Event Capture and Logging (`ProvisioningEventHandler`)

As each Ansible task executes, events are captured and stored as `Provision_Event` records:

- `event_name` - Task name from playbook

- `event_number` - Sequence number
- `provisioning_status` - Status code indicating task outcome:
    - **0** = Success - Task completed successfully
    - **1** = Running - Task is currently executing
    - **2** = Failed - Critical failure that stops provisioning
    - **3** = Failed (ignored) - Task failed but errors were ignored (`ignore_errors: true` in playbook)
- `provisioning_result_json` - Task results with sensitive data redacted

The event handler automatically strips passwords, keys, secrets, and other sensitive data from logs.

Step 6: Ansible Playbook Execution (`Provisioners/plays/*.yaml`)

The Ansible playbook runs locally and typically performs these actions:

1. **Fetch Product Definition** - GET request to `/crm/product/product_id/{{ product_id }}` using `{{ access_token }}`

2. **Fetch Customer Information** - GET request to `/crm/customer/customer_id/{{ customer_id }}`

3. **Process Inventory Items** (if required) - GET request to `/crm/inventory/inventory_id/{{ inventory_id }}` for each selected item to retrieve full details (ICCID, MSISDN, serial numbers, etc.)

4. **Configure External Systems** - Make API calls to:

    - HSS (Home Subscriber Server) for subscriber provisioning
    - IMS (IP Multimedia Subsystem) for voice registration
    - CGRateS/OCS for account creation, charging configuration, rate plans
    - ENUM servers for phone number mapping
    - Network equipment (routers, switches, etc.)

5. **Add Setup Costs** (if applicable) - POST to `/crm/transaction/` to record one-time charges

6. **Charge the Customer** - POST to OCS/CGRateS to charge retail_setup_cost if configured

7. **Create OCS Account** - POST to OCS/CGRateS to create billing account with UUID

8. **Configure Recurring Charges** - Create Actions and ActionPlans in OCS/CGRateS for monthly recurring charges

9. **Create Service Record** - PUT/POST to `/crm/service/` to create the service record in CRM:

```json
{
  "customer_id": 123,
  "product_id": 456,
  "service_name": "Mobile SIM - 0412345678",
  "service_uuid": "generated-uuid-for-ocs",
  "service_status": "Active",
  "service_type": "mobile",
  "retail_cost": 30,
  "wholesale_cost": 5,
  "provisioning_play": "play_psim_only",
  "provisioning_json_vars": "{...}"
}
```

10. **Assign Inventory** - PATCH to `/crm/inventory/inventory_id/{{ inventory_id }}` to mark inventory as "Assigned" to the service

11. **Send Notifications** (optional) - Email or SMS to customer with service details

Step 7: Completion and Status Update

When playbook completes:

- **Success**: `Provision.provisioning_status` updated to **0** (Success)
- **Critical Failure**: `Provision.provisioning_status` updated to **2** (Failed), and failure email sent to `crm_config.provisioning.failure_list`
- **Non-Critical Failures**: Tasks that fail with `ignore_errors: true` are marked with status **3** (Failed but ignored) and do not stop provisioning

The provisioned service is now visible in the CRM and active for the customer (if provisioning succeeded).

Key Differences: Standalone vs Addon vs Bundle Provisioning

Standalone Products (`category: standalone`):

- Receive `customer_id` and `product_id`
- Typically require inventory items (SIM cards, phone numbers, modems)
- Create **new** service record via API PUT `/crm/service/`
- Provision new resources on external systems (HSS, OCS, network equipment)
- Example: New mobile SIM activation, new internet connection

Addon Products (`category: addon`):

- Receive `customer_id`, `product_id`, and ``**service_id**`` (existing service to modify)
- Typically do NOT require inventory (or minimal inventory)
- **Modify existing** service or add charges to existing OCS account
- May execute actions on OCS (add data pack, add credit, enable feature)
- Do not create new service records (or create child service records linked to parent)
- Example: Monthly data plan top-up, international roaming pack, extra credit

Bundle Products (`category: bundle`):

- Similar to addons in terms of variables received
- May require some inventory items (e.g., modem for home bundle)
- Create **multiple** related service records (Internet + TV + Phone)
- Provision multiple resources across different systems
- Link services together in CRM for unified billing/management
- Example: Home bundle (Internet + IPTV + VoIP phone)

Provisioning Playbook Requirements

For a playbook to work correctly, it must:

1. **Be located at** `OmniCRM-API/Provisioners/plays/{playbook_name}.yaml`
2. **Accept variables** via Ansible's `extravars` (accessed as `{{ variable_name }}`)
3. **Authenticate API calls** using `Authorization: Bearer {{ access_token }}` header
4. **Handle failures gracefully** using `rescue` blocks and `ignore_errors` where appropriate
5. **Create service record** for standalone products, or modify existing service for addons
6. **Assign inventory** if inventory items were selected
7. **Return meaningful error messages** via `fail` module when critical errors occur

Common Variables Available in Playbooks

Every playbook receives these variables:

- `customer_id` - Integer, customer ordering the service
- `product_id` - Integer, product being provisioned
- `service_id` - Integer (addons/bundles only), existing service to modify
- `access_token` - String, JWT token for CRM API authentication
- `initiating_user` - Integer, user who triggered provisioning
- Plus any inventory item IDs: `{{ inventory_type }}: inventory_id`
- Plus any variables from `provisioning_json_vars`
- Plus any variables passed in the provision request

Playbooks can use these to:

- Fetch full product details: `GET /crm/product/product_id/{{ product_id }}`
- Fetch customer details: `GET /crm/customer/customer_id/{{ customer_id }}`
- Fetch inventory details: `GET /crm/inventory/inventory_id/{{ SIM_Card }}`
- Create transactions: `POST /crm/transaction/`
- Create services: `PUT /crm/service/`
- Update services: `PATCH /crm/service/{{ service_id }}`
- Assign inventory: `PATCH /crm/inventory/inventory_id/{{ inventory_id }}`

Example: Simple Addon Playbook Flow

For a mobile data top-up addon:

1. Playbook receives: `customer_id`, `product_id`, `service_id`, `access_token`
2. Fetch service details: `GET /crm/service/{{ service_id }}` to get `service_uuid`
3. Fetch product details: `GET /crm/product/product_id/{{ product_id }}` to get pricing and data amount
4. Charge customer in OCS: `POST to CGRateS` to deduct retail_cost from balance
5. Add data credit in OCS: `POST to CGRateS` to add data balance with expiry
6. Record transaction in CRM: `POST /crm/transaction/` with charge details
7. Complete successfully

The entire process is tracked in the `Provision` and `Provision_Event` tables for debugging and audit purposes.

OCS Involvement

`OCS` (Online Charging System), implemented via CGRateS, handles all real-time charging and usage tracking for services. The CRM service record acts as a pointer to the OCS account, which manages:

- **Recurring charges** - Monthly fees, DID rental, subscription charges
- **Usage-based charging** - Per-minute voice calls, per-MB data, per-SMS charges
- **Balance management** - Monetary balances (prepaid credit) and unitary balances (data GB, voice minutes, SMS count)
- **Balance conversions** - Converting monetary balances into unitary balances (e.g., spending $30 to get 10GB data pack)
- **Account state** - Active, suspended, disabled based on credit limits and thresholds

The CRM service record contains metadata and configuration (customer, product, pricing, visibility), while OCS contains the live billing state (balances, usage, charges).

Retrieving Service Usage and Balances

Service usage information is retrieved from OCS/CGRateS and displayed to customers and staff in real-time.

How Usage is Retrieved

When a service's usage is requested (via UI or API), the following flow occurs:

1. **API Request** - Frontend calls `GET /crm/service/{service_id}` or views service details in UI

2. **Service Lookup** - API retrieves service record from database, extracts `service_uuid`

3. **CGRateS API Calls** - The `cgrates_service.py` module makes two calls to CGRateS:

   1. **Get_Balance(service_uuid)** - Retrieves account balance with `BalanceMap`
      - Returns balances organized by type: DATA, VOICE, SMS, MONETARY, DATA_DONGLE
      - Each balance includes: ID, Value, ExpirationDate, Weight, DestinationIDs
      - System adds human-readable fields: `custom_Name_hr`, `custom_Expiration`, `custom_Description_String`
   2. **Get_ActionPlans(service_uuid)** - Retrieves active auto-renewal action plans (covered in next section)

4. **Response Merging** - CGRateS data is merged into the service response:

```
{
  "service_id": 123,
  "service_name": "Mobile Service",
  "service_uuid": "abc-123-def",
  "cgrates": {
    "BalanceMap": {
      "DATA": [{
        "ID": "DATA_10GB",
        "Value": 5368709120,
        "ExpirationDate": "2025-02-01T00:00:00Z",
        "custom_Name_hr": "10GB Data Pack",
        "custom_Expiration": "Feb 1, 2025",
        "custom_Description_String": "5 GB remaining"
      }],
      "VOICE": [{
        "ID": "VOICE_UNLIMITED",
        "Value": 999999999,
        "custom_Name_hr": "Unlimited Calls",
        "custom_Description_String": "Unlimited minutes"
      }],
      "MONETARY": [{
        "ID": "PREPAID_CREDIT",
        "Value": 25.50,
        "custom_Description_String": "$25.50 credit"
      }]
    },
    "ActionPlans": [...]
  }
}
```

5. **UI Display** - Frontend components display the usage data:
    - **ServiceUsage.js** - Main usage display component with automatic refresh every 3 seconds
    - **UsageCard.js** - Summary cards for each balance type
    - **UsageProgress.js** - Progress bars showing percentage used/remaining
    - Balances are color-coded and formatted for readability

Usage Data Structure

Each balance in the `BalanceMap` contains:

CGRateS Native Fields:

- `ID` - Unique identifier for the balance (e.g., "DATA_10GB_2025_01")
- `Value` - Balance amount:
    - For DATA: bytes (5368709120 = 5 GB)
    - For VOICE: seconds (3600 = 1 hour)
    - For SMS: count (100 = 100 messages)
    - For MONETARY: currency units (25.50 = $25.50)
- `ExpirationDate` - ISO 8601 timestamp when balance expires
- `Weight` - Priority for balance consumption (higher weight consumed first)
- `DestinationIDs` - Destinations this balance applies to (e.g., ["AU", "INTERNATIONAL"])

Human-Readable Fields (added by CRM):

- `custom_Name_hr` - Human-readable name extracted from ID
- `custom_Expiration` - Formatted expiration date (e.g., "Jan 15, 2025" or "in 11 days")
- `custom_Description_String` - Human-readable balance description:
    - DATA: "5 GB remaining" or "10 GB total"
    - VOICE: "60 minutes remaining" or "Unlimited"
    - SMS: "50 SMS remaining"
    - MONETARY: "$25.50 credit"

Usage Visibility Control

Service usage visibility is controlled by two fields:

- **service_visible_to_customer** - If false, service is hidden entirely from customer's self-care portal
- **service_usage_visible_to_customer** - If false, service is visible but usage/balance details are hidden (customer can see they have the service, but not how much they've used)

This allows operators to:

- Hide internal/test services from customers
- Show service exists without revealing usage (useful for unlimited plans or sensitive services)
- Fully transparent usage display (default)

Real-Time Usage Updates

The Web UI automatically refreshes usage data:

- **Interval**: Every 3 seconds (configurable in ServiceUsage component)
- **Method**: Polls `GET /crm/service/{service_id}` which fetches live data from CGRateS
- **Efficiency**: Only active service views refresh; list views use cached data

This ensures customers and staff see near-real-time balance updates as usage occurs.

Recurring Charges / AutoRenew

Recurring charges, such as a Monthly Service Charge, or a Per-DID-Charge are first created as Actions inside `OCS` and take the format `Action_ServiceUUID_ServiceName_WhatitDoes`.

For say a $60 per month GPON service that includes 1TB of usage, the Action would look something like this:

`Action_kj49-adsf-1234-9742_60_GPON_1TB_MonthlyExpiry`

1. Reset Monetary Balance to $0
2. Send an HTTP POST to `/simple_provision` on the CRM to provision something
3. Add a Credit for 1TB Usage expiring in 1 month

If we want to make the MRC recurring (we do) then we would create an `ActionPlan` named `ActionPlan_{{ service_uuid }}_Monthly_Charge` which would have the time set to *monthly* to trigger every month, and assign the `ActionPlan` to the account.

We can set based on the *Year / Months / Days* parameter an expiry date for when the MRC will stop also, for example for a fixed 12 month service that stopped after this point.

Because the Actions and `ActionPlans` are both unique to the service, they do not share anything with any other services.

This means that we can provision them with adjusted values, and it will not impact any other services.

Addons & Bolt Ons

Addons / Bolt-Ons such as buying extra data, roaming packs, international minutes, etc, are handled in much the same way. An Action is created to do what is needed, such as charging a monetary value and then granting a unitary balance with a set expiry.

Rather than using `ActionPlans` to automatically add this to reoccur on the account, we just trigger `ExecuteAction` for the Action we just created once off from within Ansible.

Adding Prepaid Monetary Balances

For prepaid monetary balances, such as a $10 PAYG plan, this is added as a monetary balance, but with a higher priority.

The credit limit on these services for the default balance would be $0.

Credit Limits / Preventing Excessive Overspend

`ThresholdS` are used on each account to set the maximum spend for a given time period.

For PAYG / Prepaid customers, this is $0.

Interacting with OCS via CRM

For each Service you can see the `Balances` and `ActionPlans`, `Actions` and `ThresholdS` from `OCS` from within the CRM API.

`ActionPlans` can be removed as needed from the CRM API, actioned via Ansible Playbooks. `ActionPlans` can be added as needed, from the CRM, by adding an Addon/Service and actioned via Ansible Playbooks.

`OCS` accounts can be disabled, which will stop `ActionPlans` from being executed and from services being able to be consumed.

For Credit Limits, a `ThresholdS` value is set per the policy for the product.

Viewing and Managing ActionPlans in the CRM

ActionPlans (auto-renewal configurations) are displayed and managed through the CRM interface, allowing staff and customers to see upcoming automatic renewals and manage them.

How ActionPlans are Retrieved and Displayed

When viewing a service in the CRM, ActionPlans are automatically fetched and displayed:

1. **API Call** - When `GET /crm/service/{service_id}` is called, the API:

    - Retrieves the service record from the database
    - Extracts the `service_uuid` (OCS account identifier)
    - Calls `get_cgrates_action_plans_by_service_uuid(service_uuid)` from `cgrates_service.py`
    - This internally calls `ocs.Get_ActionPlans(service_uuid)` to fetch ActionPlans from CGRateS

2. **ActionPlan Data Structure** - Each ActionPlan returned contains:

    ```
    {
        "ActionPlanId": "ServiceID_abc-123-def__ProductID_456__...",
        "PlanName": "Monthly_Renewal_Plan",
        "NextExecTime": "2025-02-01T00:00:00Z",
        "custom_NextExecTime_hr": "in 11 days",
        "ActionPlanId_split_dict": {
            "ServiceID": "abc-123-def",
            "ProductID": 456,
            "CustomerID": 789,
            ...
        }
    }
    ```

    - `ActionPlanId` - Unique identifier containing encoded service/product/customer information
    - `PlanName` - Name of the action plan (typically the renewal playbook name)
    - `NextExecTime` - ISO timestamp when the ActionPlan will next execute
    - `custom_NextExecTime_hr` - Human-readable time until execution (e.g., "in 11 days", "tomorrow", "Feb 1, 2025")
    - `ActionPlanId_split_dict` - Dictionary with parsed components from the ActionPlanId

3. **UI Display** - ActionPlans are shown in the **ActionPlansTable** component:

    Table Columns:

    - **Product Name** - Retrieved by looking up `ProductID` from the ActionPlanId
    - **Cost** - Shows `retail_cost` from the product definition

    - **Renewal Date** - Displays `custom_NextExecTime_hr` (human-readable)
    - **Actions** - Two buttons:
        - **Renew Now** - Immediately provision the addon/renewal (bypasses waiting for scheduled execution)
        - **Remove Auto Renew** - Cancels the automatic renewal

  When No ActionPlans Exist:

    - Table shows message: "No auto-renew enabled for this service"
    - Customer can add auto-renewing addons to enable auto-renewal

Managing ActionPlans

Staff and customers can manage ActionPlans through the UI:

Removing an ActionPlan (Canceling Auto-Renewal):

1. Click "Remove Auto Renew" button in ActionPlansTable
2. Confirmation modal appears: "Are you sure you want to remove this auto-renewal?"
3. On confirmation, frontend calls: `DELETE /crm/oam/remove_action_plan/{action_plan_id}`
4. API removes the ActionPlan from CGRateS via `ocs.Remove_ActionPlan()`
5. Activity is logged: "Removed ActionPlan {ActionPlanId} from service {service_id}"
6. ActionPlan disappears from the table

Renewing Immediately (Manual Renewal):

1. Click "Renew Now" button in ActionPlansTable
2. Confirmation modal appears: "Are you sure you want to renew this now?"
3. On confirmation, the system:
    - Extracts product_id from ActionPlanId
    - Creates a new provisioning job for that product
    - Provisions the addon immediately (runs the provisioning playbook)
    - Service receives the addon benefits without waiting for scheduled renewal
4. Provisioning status modal shows progress
5. On success, balances are immediately updated

Adding Auto-Renewal:

Auto-renewal is enabled by provisioning an addon product that has `auto_renew` set:

- Products with `auto_renew = "true"` - Automatically create ActionPlans during provisioning
- Products with `auto_renew = "prompt"` - Ask customer if they want auto-renewal (modal dialog)
- Products with `auto_renew = "false"` - Never create ActionPlans (one-time purchase)

The provisioning playbook creates the ActionPlan in CGRateS with:

- Unique ActionPlanId encoding service, product, and customer IDs
- Renewal schedule (monthly, yearly, custom interval)
- Action to execute (typically reprovisioning the same addon)
- Expiration date (if contract has fixed term)

ActionPlan Naming Convention

ActionPlans follow a standardized naming convention to encode metadata:

Format:

```
ServiceID_{service_uuid}__ProductID_{product_id}__CustomerID_{customer_id}__...
```

Example:

```
ServiceID_abc-123-def__ProductID_456__CustomerID_789__MonthlyRenewal
```

This encoding allows the CRM to:

- Identify which service the ActionPlan belongs to
- Look up product details (name, pricing) for display
- Track customer ownership
- Parse renewal type and schedule

The CRM automatically parses these components into `ActionPlanId_split_dict` for easy access.

ActionPlans in Service View

When viewing a service in the CRM, the ActionPlans table is displayed in the service details:

Staff View (`ServiceView.js`):

- Shows full ActionPlans table with all management options
- Displays product names, costs, renewal dates
- Allows removal and immediate renewal

Customer Self-Care View:

- Shows upcoming renewals in a simplified view
- Displays next renewal date and amount
- May allow customers to disable auto-renewal (configurable per product)

Empty State:

- If no ActionPlans exist: "No auto-renew enabled for this service"
- Suggests adding an auto-renewing addon to enable automatic renewals

ActionPlans and Expiring Services

The CRM includes an expiring services endpoint that identifies services needing renewal:

`GET /crm/service/expiring?threshold=7`

This returns services expiring within the threshold days and includes:

- Services with balances expiring soon
- Services without active ActionPlans (manual renewal required)
- Services with ActionPlans scheduled to execute soon

This allows operators to:

- Send renewal reminders to customers
- Identify at-risk customers (expiring without auto-renewal)
- Monitor upcoming auto-renewals
- Proactively manage service continuity

Provisioning System

OmniCRM uses **Ansible** to automate the provisioning, configuration, and deprovisioning of customer services. The provisioning system is designed to be flexible, allowing for complex workflows while maintaining consistency and reliability.

Recent Provisioning Events

Note

For a complete walkthrough of the product-to-service journey with detailed Ansible playbook examples, pricing strategies, and real-world scenarios, see Complete Product Lifecycle Guide <guide_product_lifecycle>.

Overview

When a product is ordered or a service needs to be configured, OmniCRM creates a **Provisioning Job** that executes one or more Ansible playbooks. These playbooks interact with various backend systems (OCS/CGRateS, network equipment, APIs, etc.) to fully provision the service.

The provisioning system supports two main workflows:

1. **Standard Provisioning** - Triggered by staff or customers through the UI/API
2. **Simple Provisioning** - Triggered by external systems like OCS for automated operations

Provisioning Status Values

Provisioning jobs and individual tasks can have the following statuses:

- **Status 0 (Success)** - The provisioning job completed successfully
- **Status 1 (Running)** - The provisioning job or task is currently executing
- **Status 2 (Failed - Critical)** - A critical failure occurred that caused the provisioning to fail
- **Status 3 (Failed - Ignored)** - A task failed but had `ignore_errors: true`, so provisioning continued

When a provisioning job fails, OmniCRM sends email notifications to the configured failure notification list with detailed error information.

How Products Drive Provisioning

The **Product** definition is the blueprint for what gets provisioned and how. When a user selects a product to provision, the system reads several key fields from the product definition to determine what to do.

Product Fields Used in Provisioning

A product definition contains:

- `provisioning_play` - The name of the Ansible playbook to execute (without the .yaml extension)
- `provisioning_json_vars` - JSON string containing default variables to pass to Ansible
- `inventory_items_list` - List of inventory types that must be assigned (e.g., `['SIM Card', 'Mobile Number']`)
- `product_id`, `product_name`, pricing fields - Automatically passed to the playbook

Example Product Definition

```
{
  "product_id": 1,
  "product_slug": "Mobile-SIM",
  "product_name": "Mobile SIM Only",
  "provisioning_play": "play_psim_only",
  "provisioning_json_vars": "{\"iccid\": \"\", \"msisdn\": \"\"}",
  "inventory_items_list": "['SIM Card', 'Mobile Number']",
  "retail_cost": 0,
  "retail_setup_cost": 0,
  "wholesale_cost": 3,
  "wholesale_setup_cost": 1
}
```

From Product to Provisioning Job

When provisioning is initiated, the system:

1. **Loads the playbook** specified in `provisioning_play`

   The system looks for `OmniCRM-API/Provisioners/plays/play_psim_only.yaml`

2. **Merges variables** from multiple sources into `extra_vars`:

   1. **From provisioning_json_vars**: `{"iccid": "", "msisdn": ""}`
   2. **From request body**: Any additional variables the user/API provides

    3. **From product fields**: `product_id`, `customer_id`, etc.
    4. **From authentication**: `access_token` or setup for `refresh_token`

3. **Assigns inventory** based on `inventory_items_list`

   Before running the playbook, the UI/API prompts for inventory selection:

    - **SIM Card** - User selects an available SIM from inventory
    - **Mobile Number** - User selects an available phone number

   The selected inventory IDs are added to `extra_vars` with the inventory type as the key:

```
extra_vars = {
    "product_id": 1,
    "customer_id": 456,
    "SIM Card": 789,        # inventory_id of selected SIM
    "Mobile Number": 101,   # inventory_id of selected phone number
    "iccid": "",            # From provisioning_json_vars
    "msisdn": "",           # From provisioning_json_vars
    "access_token": "eyJ..."
}
```

4. **Passes everything to Ansible** via `hostvars[inventory_hostname]`

   Inside the playbook, variables are accessible as:

```
- name: Get inventory_id for SIM Card
  set_fact:
    inventory_id_sim_card: "{{ hostvars[inventory_hostname]['SIM Card'] | int }}"
  when: "'SIM Card' in hostvars[inventory_hostname]"
```

How Playbooks Use Inventory Variables

Once the playbook has the inventory IDs, it retrieves the full inventory details from the API:

```
- name: Get SIM Card Details from Inventory
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{ inventory_id_sim_card }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: sim_card_response
```

```yaml
- name: Extract ICCID and IMSI from inventory
  set_fact:
    iccid: "{{ sim_card_response.json.iccid }}"
    imsi: "{{ sim_card_response.json.imsi }}"

- name: Get Phone Number Details from Inventory
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
inventory_id_phone_number }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: phone_number_response

- name: Extract MSISDN
  set_fact:
    msisdn: "{{ phone_number_response.json.msisdn }}"
```

The playbook can then use these values to:

- Provision the SIM card on the HSS with the IMSI
- Configure the phone number in the billing system
- Assign the inventory items to the customer
- Create the service record with these details

Real-World Example: Mobile SIM Provisioning

From `play_psim_only.yaml`, here's how it uses product and inventory data:

```yaml
- name: Get Product information from CRM API
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/product/product_id/{{
product_id }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: api_response_product

- name: Set package facts from product
  set_fact:
    package_name: "{{ api_response_product.json.product_name }}"
    package_comment: "{{ api_response_product.json.comment }}"
    setup_cost: "{{ api_response_product.json.retail_setup_cost }}"
    monthly_cost: "{{ api_response_product.json.retail_cost }}"

- name: Set inventory_id_sim_card if SIM Card was selected
```

```yaml
  set_fact:
    inventory_id_sim_card: "{{ hostvars[inventory_hostname]['SIM
Card'] | int }}"
  when: "'SIM Card' in hostvars[inventory_hostname]"

- name: Set inventory_id_phone_number if Mobile Number was selected
  set_fact:
    inventory_id_phone_number: "{{
hostvars[inventory_hostname]['Mobile Number'] | int }}"
  when: "'Mobile Number' in hostvars[inventory_hostname]"

- name: Get SIM Card details from inventory
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
inventory_id_sim_card }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: sim_inventory_response

- name: Get Phone Number details from inventory
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
inventory_id_phone_number }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: phone_inventory_response

- name: Extract values from inventory
  set_fact:
    iccid: "{{ sim_inventory_response.json.iccid }}"
    imsi: "{{ sim_inventory_response.json.imsi }}"
    msisdn: "{{ phone_inventory_response.json.msisdn }}"
    ki: "{{ sim_inventory_response.json.ki }}"
    opc: "{{ sim_inventory_response.json.opc }}"

- name: Provision subscriber on HSS
  uri:
    url: "http://{{ hss_server }}/subscriber/{{ imsi }}"
    method: PUT
    body_format: json
    body:
      {
        "imsi": "{{ imsi }}",
        "msisdn": "{{ msisdn }}",
```

```
        "ki": "{{ ki }}",
        "opc": "{{ opc }}",
        "enabled": true
      }
    status_code: 200

- name: Assign inventory to customer
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
inventory_id_sim_card }}"
    method: PATCH
    headers:
      Authorization: "Bearer {{ access_token }}"
    body_format: json
    body:
      {
        "customer_id": {{ customer_id }},
        "item_state": "Assigned"
      }
    status_code: 200
```

This demonstrates the complete flow:

1. Product definition specifies `provisioning_play: "play_psim_only"`
2. Product requires `inventory_items_list: ['SIM Card', 'Mobile Number']`
3. User selects inventory items during provisioning
4. Inventory IDs are passed to playbook as `extra_vars`
5. Playbook retrieves full inventory details from API
6. Playbook uses inventory data to configure network equipment
7. Playbook marks inventory as assigned to the customer

Rollback and Cleanup: Best Practice Pattern

Critical Best Practice: The same playbook should handle both failed provision rollback and intentional deprovisioning using Ansible's `block` and `rescue` structure.

Playbook Structure

From `play_psim_only.yaml`:

```
- name: OmniCore Service Provisioning 2024
  hosts: localhost
  gather_facts: no
  become: False

  tasks:
```

```yaml
    - name: Main block
      block:
        # --- PROVISIONING TASKS ---
        - name: Get Product information
          uri: ...

        - name: Create account in OCS
          uri: ...

        - name: Provision subscriber on HSS
          uri: ...

        - name: Create service record
          uri: ...

        # ... many more provisioning tasks ...

      rescue:
        # --- CLEANUP TASKS ---
        # This section runs when:
        # 1. Any task in the block fails (rollback)
        # 2. action == "deprovision" (intentional cleanup)

        - name: Get Inventory items linked to this service
          uri:
            url: "{{ crm_config.crm.base_url }}/crm/inventory/
customer_id/{{ customer_id }}"
            method: GET
          register: inventory_api_response
          ignore_errors: True

        - name: Return inventory to pool
          uri:
            url: "{{ crm_config.crm.base_url }}/crm/inventory/
inventory_id/{{ item.inventory_id }}"
            method: PATCH
            body_format: json
            body:
              service_id: null
              customer_id: null
              item_state: "Used"
          with_items: "{{ inventory_api_response.json.data }}"
          ignore_errors: True

        - name: Delete Account from Charging
          uri:
            url: "http://{{ crm_config.ocs.OCS }}/jsonrpc"
            method: POST
```

```
            body:
              {
                "method": "ApierV1.RemoveAccount",
                "params": [{
                    "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                    "Account": "{{ service_uuid }}"
                }]
              }
          ignore_errors: True

        - name: Delete Attribute Profile
          uri:
            url: "http://{{ crm_config.ocs.OCS }}/jsonrpc"
            method: POST
            body:
              {
                "method": "APIerSv1.RemoveAttributeProfile",
                "params": [{
                    "ID": "ATTR_ACCOUNT_{{ service_uuid }}"
                }]
              }
          ignore_errors: True

        - name: Remove Resource Profile
          uri: ...
          ignore_errors: True

        - name: Remove Filters
          uri: ...
          ignore_errors: True

        - name: Deprovision Subscriber from HSS
          uri:
            url: "{{ item.key }}/subscriber/{{
item.value.subscriber_id }}"
            method: DELETE
          loop: "{{ hss_subscriber_data | dict2items }}"
          ignore_errors: True
          when:
            - deprovision_subscriber | bool == true

        - name: Patch Subscriber to Dormant State
          uri:
            url: "{{ item.key }}/subscriber/{{
item.value.subscriber_id }}"
            method: PATCH
            body:
              {
```

```
                "enabled": true,
                "msisdn": "9999{{ imsi[-10:] }}",  # Dummy number
                "ue_ambr_dl": 9999999,              # Unusably high
                "ue_ambr_ul": 9999999
              }
          loop: "{{ hss_subscriber_data | dict2items }}"
          when:
            - deprovision_subscriber | default(false) | bool == false

        # Final assertion determines success or failure
        - name: Set status to "Success" if Manual deprovision / Fail
if failed provision
          assert:
            that:
              - action == "deprovision"
```

Why This Pattern is Best Practice

1. No Code Duplication

The same cleanup tasks handle both scenarios:

- **Failed Provision (Rollback)**: If any task in the `block` fails, the `rescue` section executes automatically
- **Intentional Deprovision**: When called with `action: "deprovision"`, the playbook immediately jumps to `rescue`

2. Complete Cleanup Guaranteed

When a provision fails partway through, the rescue section ensures:

- All created OCS accounts are deleted
- All configured network equipment entries are removed
- Assigned inventory is returned to the pool
- HSS subscribers are deleted or set to dormant
- No partial provisioning remains in any system

This prevents "orphaned" resources that:

- Consume inventory without being tracked
- Create billing accounts that aren't linked to services
- Cause confusion during troubleshooting
- Waste network resources

3. Graceful Failure Handling with ignore_errors

Notice every cleanup task uses `ignore_errors: True`. This is intentional because:

- During rollback, some resources may not have been created yet
- We want to attempt all cleanup tasks even if some fail
- The final assertion determines overall success/failure

For example, if provisioning fails at "Create account in OCS", the cleanup will try to:

- Delete the OCS account (will fail, but ignored)
- Remove attribute profiles (will fail, but ignored)
- Return inventory (succeeds)
- Delete HSS subscriber (may not exist, ignored)

4. Distinguishing Deprovision from Rollback

The final assertion at the end of `rescue` is clever:

```
- name: Set status to "Success" if Manual deprovision / Fail if failed provision
  assert:
    that:
      - action == "deprovision"
```

This means:

- **If ``action == "deprovision"``**: Assertion passes, playbook succeeds (status 0)
- **If ``action`` is not set or != "deprovision"**: Assertion fails, playbook fails (status 2)

So the same cleanup code results in different provisioning job statuses depending on intent.

5. Conditional Cleanup Based on Service Type

Some cleanup tasks use conditionals to handle different scenarios:

```
- name: Deprovision Subscriber from HSS
  uri: ...
  when:
    - deprovision_subscriber | bool == true

- name: Patch Subscriber to Dormant State
  uri: ...
  when:
    - deprovision_subscriber | default(false) | bool == false
```

This allows for flexible cleanup:

- **Full deletion**: When SIMs are dedicated to customers

(`deprovision_subscriber: true`)
- **Dormant state**: When SIMs are reusable and should remain in HSS (`deprovision_subscriber: false`)

How to Use This Pattern

For Provisioning:

```json
{
  "product_id": 1,
  "customer_id": 456,
  "provisioning_play": "play_psim_only"
}
```

If provisioning fails, automatic rollback occurs via `rescue`.

For Deprovisioning:

```json
{
  "service_id": 123,
  "service_uuid": "Service_abc123",
  "action": "deprovision",
  "provisioning_play": "play_psim_only"
}
```

The playbook skips directly to `rescue` section, runs all cleanup, and succeeds.

Benefits Summary

� **Single source of truth**: One playbook handles provision and deprovision � **Atomic operations**: Either fully provisioned or fully cleaned up � **No orphaned resources**: Failed provisions leave no trace � **Easier maintenance**: Changes to provisioning logic automatically apply to cleanup � **Reduced errors**: No chance of provision and deprovision code getting out of sync � **Testable**: Can test deprovision logic by running with `action: "deprovision"`

This pattern should be followed in all provisioning playbooks to ensure reliability and consistency.

Overriding Product Variables

The `provisioning_json_vars` can be overridden at provision time. For example, a product might define:

```json
{
  "provisioning_json_vars": "{\"monthly_cost\": 50,
\"data_limit_gb\": 100}"
}
```

But when provisioning, you can override these:

```json
{
  "product_id": 1,
  "customer_id": 456,
  "monthly_cost": 45,
  "data_limit_gb": 150
}
```

The merged `extra_vars` will use the overridden values. This allows for:

- Custom pricing for specific customers
- Different data limits based on promotions
- Testing with different parameters without modifying the product

Products Without Inventory

Not all products require inventory. For example, a data addon or a feature toggle might have:

```json
{
  "product_id": 10,
  "product_name": "Extra 10GB Data",
  "provisioning_play": "play_local_data_addon",
  "provisioning_json_vars": "{\"data_gb\": 10}",
  "inventory_items_list": "[]"
}
```

In this case, the playbook receives:

```python
extra_vars = {
    "product_id": 10,
    "customer_id": 456,
    "service_id": 123,  # Service to add data to
    "data_gb": 10,
    "access_token": "eyJ..."
}
```

The playbook simply adds the data to the existing service without needing any inventory items.

Standard Provisioning Workflow

Standard provisioning is initiated when:

- A staff member adds a service to a customer from the UI
- A customer orders a service through the self-care portal
- The API is called directly with `PUT /crm/provision/`

When You Click "Provision"

Here's the complete flow that occurs when a user clicks the "Provision" button:

1. UI Displays Product Selection

User selects a product from the product catalog. Product contains:

- `provisioning_play` - Which Ansible playbook to run
- `inventory_items_list` - Required inventory (e.g., `['SIM Card', 'Mobile Number']`)
- `provisioning_json_vars` - Default variables

2. Inventory Picker (If Required)

If `inventory_items_list` is not empty, a modal appears showing dropdowns for each inventory type. User must select available inventory items before proceeding.

3. Provision Button Clicked

JavaScript sends `PUT /crm/provision/` request:

```
PUT /crm/provision/
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json

{
  "product_id": 42,
  "customer_id": 123,
  "SIM Card": 5001,
  "Mobile Number": 5002
}
```

4. API Receives Request

The provisioning endpoint (`routes/provisioning.py`):

- Validates authentication (Bearer token, API key, or IP whitelist)
- Checks user has `CREATE_PROVISION` permission
- Extracts `initiating_user` from token
- Loads product definition from database
- Retrieves playbook path: `OmniCRM-API/Provisioners/plays/play_psim_only.yaml`

5. Variables Merged

System combines variables from multiple sources:

```python
# From product
product_vars = json.loads(product['provisioning_json_vars'])
# From request body
request_vars = request.json
# System-added
system_vars = {
    'product_id': 42,
    'customer_id': 123,
    'access_token': g.access_token,  # See authentication section
below
    'initiating_user': 7
}
# Final merged
extra_vars = {**product_vars, **request_vars, **system_vars}
```

6. Provision Record Created

Database record created with status 1 (Running):

```python
provision = {
    'provision_id': 456,
    'customer_id': 123,
    'product_id': 42,
    'provisioning_play': 'play_psim_only',
    'provisioning_json_vars': json.dumps(extra_vars),
    'provisioning_status': 1,  # Running
    'task_count': 85,
    'initiating_user': 7,
    'created': '2025-01-10T14:30:00Z'
}
```

7. Background Thread Spawned

```python
run_playbook_in_background(
    playbook='plays/play_psim_only.yaml',
    extra_vars=extra_vars,
    provision_id=456,
    refresh_token=refresh_token  # For token refresh during execution
)
```

8. API Returns Immediately

Response returned to UI with `provision_id`:


```json
{
  "provision_id": 456,
  "provisioning_status": 1,
  "message": "Provisioning job created"
}
```

9. UI Polls for Updates

UI starts polling `GET /crm/provision/provision_id/456` every 3 seconds to check status. The response includes:


```json
{
  "provision_id": 456,
  "provisioning_status": 1,
  "task_count": 12,
  "provisioning_result_json": [
    {
      "event_number": 1,
      "event_name": "Get Product information from CRM API",
      "provisioning_status": 0,
      "timestamp": "2024-01-15T10:30:05"
    },
    {
      "event_number": 2,
      "event_name": "Assign SIM Card from inventory",
      "provisioning_status": 1,
      "timestamp": "2024-01-15T10:30:07"
    }
  ]
}
```


10. Ansible Executes in Background

Playbook runs tasks sequentially:

- Each task completion creates `Provision_Event` record in database
- Event includes: task name, status (0=success, 2=failed, 3=failed but ignored), result JSON
- UI displays real-time progress showing completed tasks and currently running task
- Failed tasks show error messages in the event details

Tracking in the UI:

While provisioning is running (status 1), users can view:

- **Service Details Page** - Shows provisioning status badge (Running/Success/Failed)
- **Activity Log** - Lists all provisioning events with timestamps
- **Provision Details View** - Shows task-by-task progress with expand/collapse for details

Example display:

Provisioning Status: Running (8 of 12 tasks completed)

✓ Get Product information from CRM API ✓ Fetch Customer details ✓ Assign SIM Card from inventory (ICCID: 8991101200003204510) ✓ Assign Mobile Number (555-0123) ⟳ Create account in OCS/CGRateS (in progress...) ⯑ Configure network policies ⯑ Create service record ...

11. Provisioning Completes

Final status set:

- `provisioning_status: 0` - Success
- `provisioning_status: 2` - Failed (critical error)

UI stops polling and displays result:

- **Success**: Green checkmark, service marked Active, user can view service details
- **Failure**: Red X, error message displayed, option to retry or contact support
- **Email notification**: If failure, email sent to `provisioning.failure_list` in config

Authentication and Authorization

User Tracking

Every provisioning job tracks which user initiated it:

- **User-initiated**: The `initiating_user` field is set to the user's ID from their JWT token
- **API Key auth**: Uses the first admin user ID
- **IP whitelist auth**: Uses the first admin user ID

Permission Checks

The system checks permissions before allowing provisioning:

- Staff need the `CREATE_PROVISION` permission
- Customers can only provision services for their own account (`VIEW_OWN_PROVISION` permission)

How Ansible Authenticates with the CRM API

Ansible playbooks need to make authenticated API calls back to the CRM (to fetch product details, create services, update inventory, etc.). Authentication is handled through **Bearer tokens** passed to the playbook.

The source of the `access_token` depends on the authentication method used to call the provisioning API:

Method 1: User Login (Bearer Token)

When a user logs in via web UI:

1. User authenticates: `POST /crm/auth/login`
2. Receives JWT `access_token` (short-lived, 15-30 min) and `refresh_token` (long-lived)
3. Makes provisioning request with Bearer token in header
4. Provisioning API extracts token from `Authorization: Bearer ...` header
5. Stores in `g.access_token` (Flask request context)
6. Passes to Ansible as `access_token` variable

Code (permissions.py):

```
# Extract Bearer token from header
auth_header = request.headers.get('Authorization', '')
if auth_header.startswith('Bearer '):
    bearer_token = auth_header[7:]
    # Validate and decode
    decoded_token = jwt.decode(bearer_token, secret_key,
algorithms=['HS256'])
    # Store for provisioning
    g.access_token = bearer_token
```

Code (provisioning.py):

```
if "access_token" in g:
    json_data['access_token'] = g.access_token
run_playbook(playbook_path, extra_vars=json_data,
provision_id=provision_id)
```

Method 2: API Key (X-API-KEY Header)

For automated systems using API keys:

1. System makes request: `PUT /crm/provision/` with `X-API-KEY: your-api-key...` header
2. Provisioning API validates API key against `crm_config.yaml`
3. **Generates a new JWT token on-the-fly** for the first admin user
4. Stores in `g.access_token`
5. Passes to Ansible

Why Generate a Token?

API keys are strings, not JWTs. Playbooks call API endpoints expecting JWT authentication. So:

- Validate API key
- If valid and has `admin` role, generate temporary JWT
- Use first admin user's ID as JWT subject
- Token allows playbook to make authenticated API calls

Code (permissions.py):

```
def handle_api_key_auth(f, api_key, *args, **kwargs):
    if not secure_compare_api_key(api_key):
        return {'message': 'Invalid API key'}, 401

    API_KEYS = yaml_config.get('api_keys', {})
    if api_key in API_KEYS:
        if 'admin' in API_KEYS[api_key].get('roles', []):
            admin_user_id = retrieve_first_admin_user_id()
            access_token =
create_access_token(identity=str(admin_user_id))
            g.access_token = access_token
```

Method 3: IP Whitelist

For trusted internal systems on private networks:

1. System makes request from whitelisted IP (e.g., 192.168.1.100)
2. Provisioning API checks client IP against `ip_whitelist` in `crm_config.yaml`
3. If whitelisted, **generates a new JWT token** for first admin user
4. Stores in `g.access_token`
5. Passes to Ansible

Code (permissions.py):

```
def handle_ip_auth(f, *args, **kwargs):
    client_ip = get_real_client_ip()
    if not is_ip_whitelisted(client_ip):
        return {'message': 'Access denied'}, 403

    admin_user_id = retrieve_first_admin_user_id()
    access_token = create_access_token(identity=str(admin_user_id))
    g.access_token = access_token
```

Using the Token in Playbooks

Every API call in the playbook includes the token:

```
- name: Get Product Details
  uri:
    url: "http://localhost:5000/crm/product/product_id/{{ product_id
}}"
    headers:
      Authorization: "Bearer {{ access_token }}"

- name: Create Service Record
  uri:
```

```yaml
    url: "http://localhost:5000/crm/service/"
    method: PUT
    headers:
      Authorization: "Bearer {{ access_token }}"
    body:
      customer_id: "{{ customer_id }}"
      service_name: "Mobile Service"
```

Token Expiration and Refresh

Long-running playbooks (5-10 minutes) may outlive the `access_token` (15-30 min expiry). For user-initiated provisions, the system passes both `access_token` and `refresh_token`:

```
refresh_token = request.cookies.get('refresh_token')
run_playbook(playbook_path, extra_vars, provision_id,
refresh_token=refresh_token)
```

If `access_token` expires, the playbook runner can:

1. Detect 401 Unauthorized response
2. Call `POST /crm/auth/refresh` with `refresh_token`
3. Receive new `access_token`
4. Retry failed request

For API key/IP whitelist auth, generated tokens can have longer expiration (1-2 hours) since these are trusted automated systems.

The Provisioning Process

1. **Job Creation**

   When a provisioning request is received, the system:

   - Validates the request and checks permissions
   - Loads the Ansible playbook specified in the product definition
   - Creates a `Provision` record in the database with status 1 (Running)
   - Extracts variables from the product definition and request body
   - Captures authentication tokens for API access

2. **Token Handling**

   Ansible playbooks need to authenticate with the CRM API to retrieve data and make changes. The provisioning system handles this in two ways:

   - **Bearer Token (JWT)**: For user-initiated provisioning, the `refresh_token` from the request is used to generate fresh access tokens during playbook execution
   - **API Key/IP Auth**: For automated systems, an `access_token` is

passed directly to the playbook via `g.access_token`

3. **Background Execution**

   The playbook runs in a background thread using `playbook_runner_v2`. This allows the API to return immediately while provisioning continues asynchronously.

   During execution:

   - Each task completion/failure creates a `Provision_Event` record
   - The event handler monitors for critical vs. ignored failures
   - Real-time status updates are written to the database
   - The UI can poll for updates via `GET /crm/provision/provision_id/<id>`

4. **Playbook Execution**

   The Ansible playbook typically performs these operations:

   - Retrieves product information from the API
   - Retrieves customer information from the API
   - Assigns inventory items (SIM cards, IP addresses, phone numbers, etc.)
   - Creates accounts in OCS/OCS
   - Configures network equipment
   - Creates the service record in the CRM API
   - Adds setup cost transactions
   - Sends welcome emails/SMS to customers

5. **Error Handling**

   Ansible playbooks use `block` and `rescue` sections for rollback:

   - If a critical task fails, the rescue section removes partial provisioning
   - Tasks with `ignore_errors: true` are marked as status 3 and don't fail the job
   - Fatal errors (YAML syntax, connection failures) create a special error event with debugging information

Example: Standard Provisioning Playbook

Here's an example from `play_simple_service.yaml`:

```yaml
- name: Simple Provisioning Play
  hosts: localhost
  gather_facts: no
  become: False
```

```
  tasks:
    - name: Main block
      block:
      - name: Get Product information from CRM API
        uri:
          url: "http://localhost:5000/crm/product/product_id/{{
product_id }}"
          method: GET
          headers:
            Authorization: "Bearer {{ access_token }}"
          return_content: yes
          validate_certs: no
        register: api_response_product

      - name: Set package facts
        set_fact:
          package_name: "{{ api_response_product.json.product_name
}}"
          setup_cost: "{{ api_response_product.json.retail_setup_cost
}}"
          monthly_cost: "{{ api_response_product.json.retail_cost }}"

      - name: Generate Service UUID
        set_fact:
          service_uuid: "Service_{{ 99999999 | random | to_uuid }}"

      - name: Create account in OCS
        uri:
          url: "http://{{ crm_config.ocs.OCS }}/jsonrpc"
          method: POST
          body_format: json
          body:
            {
              "method": "ApierV2.SetAccount",
              "params": [{
                  "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                  "Account": "{{ service_uuid }}",
                  "ActionPlanIds": [],
                  "ExtraOptions": { "AllowNegative": false,
"Disabled": false }
              }]
            }
          status_code: 200
        register: response

      - name: Add Service via API
        uri:
          url: "http://localhost:5000/crm/service/"
```

```
          method: PUT
          body_format: json
          headers:
            Authorization: "Bearer {{ access_token }}"
          body:
            {
              "customer_id": "{{ customer_id }}",
              "product_id": "{{ product_id }}",
              "service_name": "Service: {{ service_uuid }}",
              "service_uuid": "{{ service_uuid }}",
              "service_status": "Active",
              "retail_cost": "{{ monthly_cost | float }}"
            }
          status_code: 200
        register: service_creation_response

      - name: Add Setup Cost Transaction
        uri:
          url: "http://localhost:5000/crm/transaction/"
          method: PUT
          headers:
            Authorization: "Bearer {{ access_token }}"
          body_format: json
          body:
            {
              "customer_id": {{ customer_id | int }},
              "service_id": {{
service_creation_response.json.service_id | int }},
              "title": "{{ package_name }} - Setup Costs",
              "retail_cost": "{{ setup_cost | float }}"
            }
        register: api_response_transaction

      rescue:
        - name: Remove account in OCS
          uri:
            url: "http://{{ crm_config.ocs.OCS }}/jsonrpc"
            method: POST
            body_format: json
            body:
              {
                "method": "ApierV2.RemoveAccount",
                "params": [{
                    "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                    "Account": "{{ service_uuid }}"
                }]
              }
            status_code: 200
```

```yaml
      - name: Fail the provision
        assert:
          that:
            - false
```

This playbook demonstrates the typical flow:

1. Retrieve product details from the CRM API
2. Generate a unique service UUID
3. Create the billing account in OCS
4. Create the service record via the CRM API
5. Add setup cost transactions
6. If anything fails, the `rescue` section removes the OCS account

Simple Provisioning Workflow

Simple provisioning is designed for automated systems that need to trigger provisioning without user interaction. The most common use case is OCS triggering addon provisioning via ActionPlans.

Simple Provisioning Endpoints

OmniCRM provides two simple provisioning endpoints:

- `POST /crm/provision/simple_provision_addon/service_id/<id>/product_id/<id>`

  For automated addon provisioning (e.g., recurring charges, automatic top-ups)

- `POST /crm/provision/simple_provision_addon_recharge/service_id/<id>/product_id/<id>`

  For quick recharge operations that need immediate feedback

Authentication for Simple Provisioning

Simple provisioning endpoints use **IP whitelisting** or **API keys** for authentication:

- The request's source IP is checked against `ip_whitelist` in `crm_config.yaml`
- Or an API key from `api_keys` in `crm_config.yaml` can be provided
- An access token is generated and passed to the playbook via `g.access_token`

Example: OCS ActionPlan Callback

OCS can be configured to call the simple provisioning endpoint when executing recurring actions:

```json
{
  "method": "ApierV1.SetActionPlan",
  "params": [{
    "Id": "ActionPlan_Service123_Monthly_Charge",
    "ActionsId": "Action_Service123_Add_Monthly_Data",
    "Timing": {
      "Years": [],
      "Months": [],
      "MonthDays": [1],
      "Time": "00:00:00Z"
    },
    "Weight": 10,
    "ActionTriggers": [
      {
        "ThresholdType": "*min_event_counter",
        "ThresholdValue": 1,
        "ActionsID": "Action_Service123_HTTP_Callback"
      }
    ]
  }]
}
```

The action makes an HTTP POST to:

This triggers the associated playbook (e.g., `play_topup_no_charge.yaml`) which adds data/credits to the service.

Example: Simple Topup Playbook

From `play_topup_monetary.yaml`:

```yaml
- name: Mobile Topup Monetary - 2024
  hosts: localhost
  gather_facts: no
  become: False

  tasks:
    - name: Get Product information from CRM API
      uri:
        url: "http://localhost:5000/crm/product/product_id/{{
product_id }}"
        method: GET
        headers:
```

```yaml
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_product

    - name: Get Service information from CRM API
      uri:
        url: "http://localhost:5000/crm/service/service_id/{{
service_id }}"
        method: GET
        headers:
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_service

    - name: Set service facts
      set_fact:
        service_uuid: "{{ api_response_service.json.service_uuid }}"
        customer_id: "{{ api_response_service.json.customer_id }}"
        package_name: "{{ api_response_product.json.product_name }}"
        monthly_cost: "{{ api_response_product.json.retail_cost }}"

    - name: Get Customer Payment Method
      uri:
        url: "http://localhost:5000/crm/stripe/customer_id/{{
customer_id }}"
        method: GET
        headers:
          Authorization: "Bearer {{ access_token }}"
        return_content: yes
      register: api_response_stripe

    - name: Charge customer
      uri:
        url: "http://localhost:5000/crm/stripe/charge_card/{{
customer_stripe_id }}"
        method: POST
        headers:
          Authorization: "Bearer {{ access_token }}"
        body_format: json
        body:
          {
            "retail_cost": "{{ monthly_cost }}",
            "description": "{{ package_name }} topup",
            "customer_id": "{{ customer_id | int }}",
            "service_id": "{{ service_id | int }}"
          }
      register: api_response_stripe
```

```yaml
    - name: Add monetary balance to OCS
      uri:
        url: "http://{{ crm_config.ocs.OCS }}/jsonrpc"
        method: POST
        body_format: json
        body:
          {
            "method": "ApierV1.AddBalance",
            "params": [{
                "Tenant": "{{ crm_config.ocs.ocsTenant }}",
                "Account": "{{ service_uuid }}",
                "BalanceType": "*monetary",
                "Balance": {
                    "Value": "{{ monthly_cost | float * 100 }}",
                    "ExpiryTime": "+4320h"
                }
            }]
          }
        status_code: 200

    - name: Add Transaction to CRM
      uri:
        url: "http://localhost:5000/crm/transaction/"
        method: PUT
        headers:
          Authorization: "Bearer {{ access_token }}"
        body_format: json
        body:
          {
            "customer_id": {{ customer_id | int }},
            "service_id": {{ service_id | int }},
            "title": "{{ package_name }}",
            "retail_cost": "{{ monthly_cost | float }}"
          }

    - name: Send Notification SMS
      uri:
        url: "http://sms-gateway/SMS/plaintext/{{ api_key }}"
        method: POST
        body_format: json
        body:
          {
            "source_msisdn": "YourCompany",
            "destinatination_msisdn": "{{ customer_phone }}",
            "message_body": "Thanks for topping up {{ monthly_cost
}}!"
          }
        status_code: 201
```

```
      ignore_errors: True
```

This playbook:

1. Gets service and product details from the API
2. Retrieves the customer's payment method
3. Charges the customer via Stripe API
4. Adds monetary balance to OCS
5. Records the transaction in the CRM
6. Sends a confirmation SMS (with `ignore_errors: True` so failures don't fail the job)

Provisioning Chains

For complex products that require multiple provisioning steps, OmniCRM supports **provisioning chains**. A chain executes multiple playbooks sequentially, passing context between them.

Example use case: A bundled service that provisions:

1. Base internet service (creates the primary service record)
2. IPTV addon (uses the service_id from step 1)
3. Static IP addon (uses the service_id from step 1)

The provisioning service automatically:

- Queries the database for the `service_id` created by the first playbook
- Injects it into the `extra_vars` for subsequent playbooks
- Tracks each playbook as a separate `Provision` record

Failure Reasons and Debugging

When provisioning fails, the system captures detailed information to help diagnose the issue.

Common Failure Scenarios

Critical Task Failures (Status 2)

These cause the entire provisioning job to fail:

- API calls returning unexpected status codes
- Assertions failing (e.g., `assert: that: response.status == 200`)
- Missing required inventory items
- Network equipment unreachable
- Invalid credentials or expired tokens
- OCS/OCS errors

Ignored Failures (Status 3)

These are logged but don't fail the job:

- Optional SMS/email notifications failing
- Non-critical data lookups (marked with `ignore_errors: True`)
- Cleanup operations during deprovisioning

Fatal Errors

These prevent the playbook from running at all:

- YAML syntax errors in the playbook
- Undefined variables in the playbook
- Missing playbook files
- Connection failures to the Ansible controller

When a fatal error occurs, the system creates a special error event containing:

- The Ansible exit code
- Full stdout (contains syntax error details)
- Full stderr (contains runtime errors)
- A list of common causes for that type of failure
- All variables passed to the playbook

Error Notification Emails

When provisioning fails (status 2), an email is automatically sent to the configured failure notification list (`provisioning.failure_list` in `crm_config.yaml`).

The email includes:

- Customer information
- Product/service details
- Color-coded task results:
  - **Green**: Successful tasks
  - **Orange**: Failed but ignored tasks
  - **Red**: Critical failures
- For critical failures: Full debug output including request/response bodies
- For fatal errors: Ansible output, error messages, and common causes

Monitoring Provisioning Jobs

Provisioning Status API

To check the status of a provisioning job:

```
GET /crm/provision/provision_id/<id>
```

```
Authorization: Bearer <token>
```

Response includes:


```json
{
  "provision_id": 123,
  "customer_id": 456,
  "customer_name": "John Smith",
  "product_id": 10,
  "provisioning_status": 0,
  "provisioning_play": "play_psim_only",
  "playbook_description": "OmniCore Service Provisioning 2024",
  "task_count": 85,
  "provisioning_result_json": [
    {
      "event_number": 1,
      "event_name": "Get Product information from CRM API",
      "provisioning_status": 1,
      "provisioning_result_json": "{...}"
    },
    {
      "event_number": 2,
      "event_name": "Create account in OCS",
      "provisioning_status": 1,
      "provisioning_result_json": "{...}"
    }
  ]
}
```


Listing Provisioning Jobs

To get a paginated list of all provisioning jobs:

```
GET /crm/provision/?page=1&per_page=20&sort=provision_id&order=desc
Authorization: Bearer <token>
```

Supports filtering:

```
GET /crm/provision/?filters={"provisioning_status":[2]}&search=Mobile
Authorization: Bearer <token>
```

This returns only failed jobs (status 2) where the description contains "Mobile".

Best Practices

Playbook Design

- **Always use block/rescue**: Ensure partial provisioning can be rolled back

- **Use ignore_errors judiciously**: Only for truly optional operations
- **Log important variables**: Use `debug` tasks to log key values for troubleshooting
- **Validate responses**: Use `assert` to check API responses are as expected
- **Idempotency**: Design playbooks to be safely re-runnable

Authentication

- **User-initiated provisioning**: Always use refresh_token for long-running playbooks
- **Automated provisioning**: Use IP whitelist or API keys with generated access tokens
- **Token expiry**: The refresh_token ensures access tokens are regenerated as needed

Error Handling

- **Provide context**: Include customer_id, service_id, and operation details in error messages
- **Notify appropriately**: Critical failures trigger emails, but don't spam for expected failures
- **Debugging info**: Capture full request/response bodies in Provision_Event records

Security

- **Validate inputs**: Check customer_id, product_id, service_id before provisioning
- **Permission checks**: Verify users can only provision for authorized customers
- **Sensitive data**: Use the redaction system to strip passwords/keys from logs
- **IP whitelisting**: Restrict simple_provision endpoints to trusted systems only

Performance

- **Background execution**: Never block API responses waiting for provisioning
- **Polling intervals**: UI should poll for status updates every 2-5 seconds
- **Parallel tasks**: Use Ansible's native parallelism for independent operations
- **Database updates**: Event handler updates the database in real-time, no need to query during execution

Related Documentation

- `concepts_ansible` - General Ansible provisioning concepts

- `concepts_api` - CRM API authentication and usage
- `concepts_products_and_services` - Product and service definitions
- `administration_inventory` - Inventory management for provisioning

Customer Activity Log

Every change made to a customer, contact, site, service, and financial parameters, like Transactions, Invoices & Payment Methods, is recorded in the Activity Log.

This allows us to track changes made to the system, who made them, and when they were made, and is useful for auditing changes, and tracking down issues, for example, if a customer says they never received an invoice, we can check the Activity Log to see if it was sent, or if a contact was removed, we can see who removed it and when.

The Activity Log is a chronological list of changes, with the most recent changes at the top, and older changes further down the list, which can be filtered from the tabs.

Activity log records cannot be deleted, but they can be filtered, and the details can be viewed to see what was changed, and by whom.

Customer Activity Log

Adding a Service

Services are the things we bill the customer for, they could be home internet services, mobile services, or even abstract services like leasing a subnet or providing metered electricity to a rack.

A Service is just an instance of a `Product <concepts_products_and_services>` for a given customer, that's picked out of the product catalog and provisioned for the customer.

If you haven't already `created a customer <basics_create_customer>`, you'll need to do that first, as services are linked to customers, you'll also need to define a payment method for the customer, as services generate charges that need to be paid.

Customers can provision their own services (if we allow them to), or Customer Service staff can provision services for the customer.

Based on the product definition, there are rules regarding who can purchase a product, such as only allowing business customers to purchase a business product, or only allowing customers with a mobile service to purchase a mobile addon.

Services can also have usage components, like data usage, call minutes, or other usage based charges, and can have multiple charges associated with them, such as monthly charges, one-off charges, or usage charges, we can view this from the "Usage" button.

Many services support Addons, for example a mobile service has all the topups available from the Addons menu, and a home internet service might have a static IP address or extra data available as an addon, again all of this is defined in the `product catalog<concepts_products_and_services>`.

Service - Provisioning and Charges

Assign Plans Workflow

The Assign Plan feature allows staff to **provision services for customers** by selecting products from the catalog and initiating the provisioning process. This is the primary method for creating new services when customers don't self-provision.

Overview

Assigning a plan involves:

1. Selecting a customer
2. Choosing a product from the catalog
3. Configuring inventory requirements (SIM cards, equipment, etc.)
4. Setting service parameters (auto-renewal, custom fields)
5. Initiating provisioning
6. Monitoring provisioning progress

This workflow is used for all service types: mobile, internet, IPTV, and VoIP.

Accessing Assign Plan

From Product Catalog:

From Customer Page:

From Add-ons Page:

The assign plan interface opens in a modal or dedicated page.

Step-by-Step Workflow

Step 1: Browse Product Catalog

The product catalog displays available products grouped by category and customer type.

Product Catalog - Customer View
Addon Carousel Modal

View Toggles:

Categories: • Service Plans • Add-ons

Toggling between Individual and Business filters products to show only those available to the selected customer type.

Product Cards:

Products are displayed in a carousel or grid:

Click **"Assign to Customer"** to proceed.

Step 2: Select Customer

If not already on a customer page, you'll be prompted to select a customer.

Customer Search:

> Search Customers: [John ▼]
>
> Matching customers: • John Smith (ID: 123) • John Doe (ID: 456) • Johnson Enterprises (ID: 789)

Type to search by:

- Customer name
- Customer ID
- Email address
- Phone number

Select the customer from the dropdown.

Warning

Ensure you've selected the correct customer before proceeding. Assigning a plan to the wrong customer requires manual intervention to correct.

Step 3: Configure Inventory (if required)

If the product requires inventory items (defined in `inventory_items_list`), inventory pickers appear.

Service Inventory Picker

Inventory Picker Example:

> Required Inventory Items:
>
> SIM Card *
>
> Available SIM Cards: • SIM-00123 - ICCID: 8944...0001 (New) • SIM-00124 - ICCID: 8944...0002 (New) • SIM-00125 - ICCID: 8944...0003 (New)
>
> Mobile Number *
>
> Available Numbers: • +44 7700 900123 (Reserved) • +44 7700 900124 (Available) • +44 7700 900125 (Available)

Inventory Selection Rules:

- Asterisk (*) indicates required field
- Only available inventory items shown (status: "In Stock" or "New")
- Dropdowns dynamically load based on inventory template names
- Once selected, items are temporarily reserved

What Happens:

- Selected inventory items are passed as variables to the provisioning playbook
- During provisioning, items are assigned to the service and customer
- Item status changes from "In Stock" to "Assigned"

Step 4: Configure Auto-Renewal (optional)

For recurring services, you may be prompted to set auto-renewal:

> Would you like to enable auto-renewal for this service?
>
> When enabled, this service will automatically renew at the end of each billing period and charge the customer's default payment method.
>
> [No] [Yes]

Auto-Renewal Behavior:

- **Yes**: Service renews automatically, customer charged monthly
- **No**: Service expires at end of contract period, manual renewal required

Best Practice: Default to "Yes" for consumer services, "No" for one-time services or when customer requests manual control.

Step 5: Review and Confirm

Review screen shows all selections before provisioning:

> Customer: John Smith (ID: 123) Product: Mobile - 20GB Plan
>
> Inventory: • SIM Card: SIM-00123 (ICCID: 8944...0001) • Mobile Number: +44 7700 900123
>
> Pricing: • Setup Fee: £0.00 • Monthly Cost: £15.00
>
> Auto-Renew: Yes
>
> [Cancel] [Confirm & Provision]

Click **"Confirm & Provision"** to initiate the provisioning process.

Step 6: Provisioning Progress

The provisioning modal displays real-time progress:

> ✓ Validating customer account ✓ Assigning SIM Card (ICCID: 8944...0001) ✓ Assigning Mobile Number (+44 7700 900123) ⟳ Configuring OCS account (in progress...) ⯑ Creating service record ⯑ Sending welcome email
>
> Progress: 3 of 6 tasks completed

Provisioning Progress - Detailed View

The modal polls the provisioning API every 0.2 seconds for status updates.

Progress Indicators:

- ☑ Completed successfully
- ⟳ Currently running
- ⯑ Pending (not started)
- ✗ Failed (if errors occur)

Provisioning Failed Example

Step 7: Completion

Success:

> Service successfully provisioned for John Smith
>
> Service ID: 789 Service Name: Mobile - +44 7700 900123 Status: Active
>
> [View Service] [Close]

Click **"View Service"** to open the service details page.

Failure:

If provisioning fails:

> Error: Unable to connect to OCS
>
> The service record has been created but provisioning did not complete. Please review the error and retry.
>
> Provision ID: 456
>
> [View Logs] [Retry] [Close]

- **View Logs**: Opens provisioning details with error messages
- **Retry**: Attempts provisioning again
- **Close**: Exits modal (service record remains but not activated)

Special Cases

Adding Add-ons to Existing Service

When assigning an add-on (category: "addon") to a customer who already has a service:

1. **Automatic Service Detection:**
    - System finds customer's existing services
    - Filters by service_type (mobile add-on only shows for mobile services)
    - If customer has multiple matching services, prompts to select which one
2. **No New Service Created:**
    - Add-on provisions against existing service_id
    - Uses existing service's OCS account
    - Inventory (if any) assigned to existing service
3. **Provisioning Playbook:**
    - Different playbook than standalone services
    - Typically adds balance, features, or equipment to existing account

Example:

> Existing Mobile Services: • Mobile - +44 7700 900123 (ID: 789) • Mobile - +44 7700 900456 (ID: 790)
>
> Which service should receive this add-on? [Mobile - +44 7700 900123 ▼]
>
> [Cancel] [Continue]

Provisioning for Business Customers

Business customers may have additional requirements:

- **Site Selection** - Choose which business location receives service
- **Contact Assignment** - Designate billing/technical contacts
- **Custom Fields** - Account numbers, cost centers, PO numbers

Example Business Flow:

Select Installation Site: [London Office - 123 Main St ▼]

Billing Contact: [Jane Doe - <jane@acme.com> ▼]

Technical Contact: [Bob Smith - <bob@acme.com> ▼]

Purchase Order Number: [PO-2025-001234________]

[Cancel] [Continue]

Bulk Service Assignment

For assigning the same plan to multiple customers (e.g., mass migrations):

1. Use CSV import (if available)
2. Or assign individually with template settings
3. Inventory must be available in bulk
4. Monitor provisioning queue to avoid overload

Common Workflows

Workflow 1: New Mobile Service

1. Customer walks into store wanting mobile service
2. Staff opens **Products → Plans**
3. Toggles to **Individual** customer type
4. Selects **"Mobile - 20GB Plan"**
5. Clicks **"Assign to Customer"**
6. Searches for customer by phone: "+1234567890"
7. Selects **John Smith** from results
8. Chooses SIM Card from inventory picker
9. Chooses available mobile number
10. Enables **auto-renewal**
11. Confirms and provisions
12. Watches progress until complete
13. Hands SIM to customer with welcome packet

Workflow 2: Adding Internet to Existing Customer

1. Navigate to customer page: **Customers → John Smith**
2. Click **Services** tab
3. Click **"Add Service"** button
4. Browse internet plans
5. Select **"Fiber - 100Mbps"**
6. Select installation site (if business/multiple sites)
7. Choose CPE modem from inventory
8. Set installation date (if required)

9. Provision service
10. Create installation ticket

Workflow 3: Assigning Data Top-Up Add-on

1. Customer calls: "I need more data"
2. Staff searches customer in global search
3. Opens customer services tab
4. Clicks "Add-ons" next to mobile service
5. Selects "5GB Data Boost"
6. Payment authorized
7. Add-on provisions immediately
8. Customer receives instant data boost

Troubleshooting

"No products available"

- **Cause:** Filters exclude all products
- **Fix:**
    - Toggle customer type (Individual vs Business)
    - Check product catalog has enabled products
    - Verify products match customer eligibility

"No inventory available"

- **Cause:** Inventory items out of stock or all assigned
- **Fix:**
    - Add more inventory items to system
    - Check item status (should be "New" or "In Stock")
    - Verify inventory template names match product requirements

"Customer not found"

- **Cause:** Customer doesn't exist or search term incorrect
- **Fix:**
    - Create customer first
    - Try different search terms (ID, email, phone)
    - Check for typos

Provisioning gets stuck

- **Cause:** Playbook error or external system unreachable
- **Fix:**
    - Wait for timeout (typically 5 minutes)
    - Check provisioning logs for specific error
    - Verify OCS, network systems are online
    - Retry provisioning after fixing issue

Provisioning succeeds but service doesn't work

- **Cause:** OCS account created but network not updated
- **Fix:**
    - Check OCS has account
    - Verify SIM activated in HLR/HSS
    - Check network provisioning (RADIUS, DPI, etc.)
    - Review playbook tasks for missed steps

Best Practices

Before Assigning:

- Verify customer has valid payment method on file
- Confirm customer eligibility for product (residential vs business)
- Ensure required inventory is available
- Review product terms and pricing with customer

During Assignment:

- Double-check customer selection before confirming
- Select correct inventory items (check serial numbers)
- Enable auto-renewal for convenience (unless customer objects)
- Monitor provisioning progress until completion

After Assignment:

- Verify service appears in customer's service list
- Check service status is "Active"
- Confirm inventory assigned correctly
- Send welcome email or instructions to customer
- Test service if possible (make test call, check data)

For Add-ons:

- Confirm add-on is compatible with existing service
- Explain billing (one-time vs recurring)
- Verify payment before provisioning
- Check balance updated immediately after add-on provision

Related Documentation

- `csa_add_service` - Overview of services
- `concepts_products_and_services` - Product catalog concepts
- `administration_inventory` - Managing inventory items
- `concepts_provisioning` - Provisioning system details
- `guide_product_lifecycle` - Complete product lifecycle including provisioning

Modifying a Service

Services can be modified by the end customer via the Self-Care Portal <self_care_portal>, or by an administrator via the admin portal.

Once a service is provisioned, you can modify its parameters, add enhancements, or change settings.

Editing Service Parameters

Basic service parameters can be modified by clicking the **Edit** button on the service details page.

Service Edit Modal

Editable Fields:

- Service Name
- Service Status (Active, Inactive, Suspended)
- Service Notes
- Auto-Renewal settings
- Linked inventory items
- Custom fields

Changes are saved immediately and reflected in the customer's self-care portal.

Adding Service Add-ons

Add-ons enhance existing services with additional features, data allowances, or hardware. Common add-on types include:

- **Data top-ups** - Additional data allowance (e.g., "5GB Data Boost")
- **Feature upgrades** - Extra capabilities (e.g., "International Calling")
- **Hardware rentals** - Equipment additions (e.g., "WiFi 6 Modem")
- **Premium services** - Enhanced features (e.g., "Static IP Address")

Accessing the Add-on Catalog

From a service details page:

1. Navigate to the service you want to enhance
2. Click the **"Add Add-on"** or **"Browse Add-ons"** button
3. The add-on catalog opens, filtered to show only compatible add-ons

Automatic Filtering:

The system automatically filters add-ons based on:

- **Service Type** - Only shows add-ons matching the service type (mobile, internet, voip, etc.)
- **Customer Type** - Filters by residential vs. business customer
- **Service Compatibility** - Checks if service meets add-on requirements
- **Availability** - Only shows enabled products

For example, if viewing a residential mobile service, you'll only see mobile add-ons marked for residential customers.

Add-on Display

Add-ons are displayed in an interactive carousel showing:

Service Add-ons Catalog

Product Card Display:

Information Displayed:

- Product icon
- Product name
- Feature list (bullet points)
- Setup cost
- Monthly/recurring cost
- Terms and conditions link
- "Add to Service" button

Provisioning an Add-on

Step 1: Select Add-on

Click on the desired add-on card, then click **"Add to Service"**

Step 2: Select Inventory (if required)

If the add-on requires physical inventory (e.g., hardware rental), an inventory picker appears:

> Available Modems: • Modem-12345 - TP-Link AX1800 (New) • Modem-12346 - TP-Link AX1800 (New) • Modem-12347 - Netgear RAX40 (New)

Select the specific inventory item to assign to this service.

Step 3: Configure Auto-Renewal (optional)

For recurring add-ons, you may be prompted:

> Would you like to enable auto-renewal for this add-on?
>
> [No] [Yes]

- **Yes** - Add-on renews automatically each billing period
- **No** - One-time purchase, customer must manually renew

Step 4: Confirm and Provision

Review the add-on details and click **"Confirm"**

The provisioning modal appears showing real-time progress:

> ✓ Validating payment ✓ Assigning inventory ⟳ Configuring service (in progress...) ⯑ Creating transaction ⯑ Sending confirmation email

Once complete, the add-on appears in the service's add-on list.

Provisioning Behind the Scenes:

When you add an add-on, the system:

1. Validates customer can purchase add-on
2. Runs add-on's Ansible playbook (`provisioning_play`)
3. Assigns any required inventory items to the service
4. Creates transaction record for billing
5. Updates service configuration (OCS, network systems, etc.)
6. Sends confirmation notification to customer

For technical details on add-on provisioning, see `Complete Product Lifecycle Guide - Adding Addons <guide_product_lifecycle>`.

Viewing Active Add-ons

Active add-ons appear on the service details page in the **"Add-ons"** section:

Example Display:

> ⯑ 5GB Data Boost
> Added: 10 Jan 2025 Expires: 17 Jan 2025 Status: Active Cost: £5.00
>
> ⯑ WiFi 6 Modem Rental
> Added: 01 Dec 2024 Inventory: Modem-12345 Auto-Renew: Yes Monthly: £10.00

Removing Add-ons

To remove an add-on from a service:

1. Navigate to the service details page
2. Locate the add-on in the **"Active Add-ons"** section
3. Click the **"Remove"** or **trashcan icon** next to the add-on
4. Confirm the removal

What Happens:

- **Hardware add-ons** - Deprovision playbook runs, inventory marked for return
- **Virtual add-ons** - Benefits removed immediately
- **Auto-renewing add-ons** - Cancels future renewals
- **Transaction created** - Credits any pro-rated amount if applicable

Warning

Removing hardware add-ons (modems, CPE, etc.) typically requires equipment to be returned. The system will mark inventory for return and may send return instructions to the customer.

Common Add-on Scenarios

Scenario 1: Customer Running Out of Data

1. Customer contacts support: "I've used my monthly allowance"
2. Staff navigates to customer's mobile service
3. Clicks "Add Add-on"
4. Selects "5GB Data Boost"
5. Provisions immediately
6. Customer receives instant data top-up

Scenario 2: Business Needs Static IP

1. Business customer requests static IP for VPN
2. Staff opens customer's internet service
3. Browses add-ons, selects "Static IP Address"
4. System provisions IP from available pool (inventory)
5. Configures routing in network equipment
6. Business receives IP configuration details

Scenario 3: Equipment Rental for Fiber Service

1. Customer signs up for fiber internet
2. During provisioning, customer opts for modem rental
3. Staff adds "WiFi 6 Modem" add-on

4. Selects available modem from inventory
5. Modem shipped to customer
6. Recurring £10/month charge added to account

Troubleshooting

"No compatible add-ons available"

- **Cause:** No add-ons match the service type or customer type
- **Fix:** Check product catalog has add-ons with matching `service_type` and `residential/business` settings

Add-on provisioning fails

- **Cause:** Provisioning playbook error or inventory unavailable
- **Fix:**
  - Check provisioning logs for specific error
  - Verify inventory items are in stock (if required)
  - Review playbook logs in provisioning system

Inventory picker shows no items

- **Cause:** No available inventory items of required type
- **Fix:**
  - Add inventory items to the system
  - Check existing items aren't all assigned or damaged
  - Verify inventory template name matches `inventory_items_list` exactly

Auto-renewal not working

- **Cause:** Auto-renewal flag not set or payment method expired
- **Fix:**
  - Verify service has `auto_renew: true`
  - Check customer has valid payment method on file
  - Review scheduled jobs in billing system

Related Documentation

- `guide_product_lifecycle` - Complete add-on provisioning flow
- `csa_add_service` - Creating new services
- `concepts_products_and_services` - Product and service concepts
- `administration_inventory` - Inventory management for hardware add-ons

Service Management

The Service Management interface provides comprehensive tools for viewing, filtering, and managing customer services across your organization.

Related documentation: `Adding Services <csa_add_service>`, `Service Usage <csa_service_usage>`, `Modifying Services <csa_modify>`, `Products & Services <concepts_products_and_services>`.

Accessing the Service List

Navigate to:

Or directly:

This displays all services with filtering, sorting, and bulk management capabilities.

Service List Overview

The service list displays services in a table with:

Service List View

Columns:

- **Service ID** - Unique identifier
- **Service Name** - Descriptive name (e.g., "Mobile - +44 7700 900123")
- **Customer Name** - Associated customer (clickable link)
- **Service Type** - mobile, iptv, internet, voip
- **Status** - Active, Inactive, Suspended
- **Start Date** - When service began
- **End Date** - When service expires/ended
- **Monthly Cost** - Retail price
- **Actions** - View, Edit, Add-ons, Usage, Delete

Filtering Services

Status Tabs

Quick-filter services by status using the tab bar:

- **All Results** - Shows all services regardless of status
- **Active** - Currently active services (default view)
- **Inactive** - Cancelled or expired services
- **Suspended** - Temporarily disabled services (non-payment, fraud, etc.)

Clicking a tab updates the list instantly. The active tab is highlighted.

Service Type Filter

Filter by service type:

Select one or multiple types to show only matching services.

Use Cases:

- **Mobile** - View all mobile SIM services
- **Internet** - Show fiber, DSL, and fixed wireless
- **IPTV** - Display TV subscription services
- **VoIP** - List voice-over-IP services

Customer Filter

Search and filter by customer:

Start typing a customer name to see matching results:

Select a customer to show only their services.

Use Case: Quickly view all services for a specific customer.

Search

Global search across all service fields:

Searches:

- Service name
- Service ID
- Customer name
- Phone numbers (for mobile services)
- Service UUID

Example: Search "0770" to find all services with phone numbers containing "0770".

Sorting Services

Click any column header to sort by that field:

Sortable Columns:

- Service ID (default: newest first)
- Service Name (alphabetical)
- Customer Name (alphabetical)
- Service Type (alphabetical)
- Start Date (chronological)
- End Date (chronological)
- Monthly Cost (numerical)

Sort Direction:

- Click once: Ascending (A-Z, oldest-newest, lowest-highest)
- Click twice: Descending (Z-A, newest-oldest, highest-lowest)
- Active sort column shows ▲ or ▼ indicator

Sort Dropdown:

Top right dropdown provides quick sorting presets:

Viewing Service Details

Click on any service name to open the detailed service view.

Service Detail Tabs:

1. **Overview** - Service summary, status, dates, pricing
2. **Inventory** - Assigned equipment (SIM cards, modems, etc.)
3. **Transactions** - Charges, credits, payments
4. **Usage** - Data/voice/SMS usage statistics
5. **Add-ons** - Active and available add-ons
6. **Activity Log** - Change history

Quick Actions

From the service list, click the action menu (⋮) for quick operations:

- **View** - Opens service details
- **Edit** - Modify service parameters
- **Add-ons** - Browse and add service enhancements
- **Usage** - View current usage and balance
- **Delete** - Cancel/remove service

Bulk Operations

Select multiple services using checkboxes to perform bulk actions.

Selecting Services

Individual Selection:

Click the checkbox next to each service you want to select.

Select All:

Click the checkbox in the table header to select all visible services.

    ☑ Service 1 ☑ Service 2 ☑ Service 3

Selection Counter:

The interface shows how many services are selected:

Bulk Actions

Once services are selected, bulk action buttons appear:

Delete Multiple Services:

1. Select services to delete
2. Click **"Delete Selected"** button
3. Confirm deletion in modal

Warning

Bulk deletion is permanent and will:

- Cancel all selected services
- Mark inventory as unassigned
- Stop all recurring charges
- Create activity log entries

Use Cases:

- Clean up test services
- Cancel services for closed customer accounts
- Remove duplicate or erroneous entries

Best Practice: Use filters to narrow the list before bulk operations to avoid accidental deletions.

Service Status Management

Status Types

Services can have three statuses:

Active

- Service is operational
- Charges apply
- Customer can use service
- Displayed with green badge

Inactive

- Service has been cancelled or expired
- No charges applied
- Customer cannot use service
- Displayed with gray badge
- Inventory marked for return/refurbishment

Suspended

- Service temporarily disabled
- May or may not charge (configurable)
- Customer cannot use service
- Displayed with orange/yellow badge
- Common reasons: Non-payment, fraud investigation, customer request

Changing Service Status

To change a service's status:

1. Open service details
2. Click **"Edit"** button
3. Change **"Service Status"** dropdown
4. Click **"Save"**

What Happens:

- **Active → Inactive:** Triggers deprovisioning playbook (if configured)
- **Active → Suspended:** Disables service in OCS/network but keeps record
- **Suspended → Active:** Re-enables service, resumes billing
- **Inactive → Active:** May trigger re-provisioning (use caution)

Service Usage View

Click **"Usage"** in the actions menu to open the usage modal.

Service Usage Modal

Information Shown:

- **Balance** - Remaining credit or prepaid value
- **Data Usage** - Used vs. allocated (progress bar)
- **Voice Usage** - Minutes used vs. plan allowance
- **SMS Usage** - Messages sent vs. allowance
- **Expiry Date** - When current balance/plan expires

Actions:

- **Top Up** - Add credit or data (for prepaid services)
- **View Details** - See detailed usage breakdown
- **Export** - Download usage report (if available)

Upcoming Auto-Renews

The Upcoming Auto-Renews view provides a centralized interface for monitoring and managing all scheduled service renewals across your organization.

Accessing Auto-Renews

Navigate to:

Or directly:

This displays all services with scheduled auto-renewal, sorted by next renewal date.

Auto-Renews Overview

The auto-renews list displays scheduled renewals in a table with:

Columns:

- **Customer** - Customer name (clickable link to customer overview)
- **Service** - Service name (clickable link to customer overview)
- **Product** - Product/plan name being renewed
- **Cost** - Renewal cost (from product retail price)
- **Renews On** - Date and time of next renewal with human-readable format
- **Status** - Service status (Active, Suspended, etc.)
- **Actions** - Renew now or remove auto-renewal

Example Display:

Upcoming Auto-Renewals Dashboard

How Auto-Renewal Works

Auto-renewal is scheduled in CGRateS (the billing system) when a service is created or modified. The system:

1. **Schedules Action** - Creates an ActionPlan in CGRateS with the renewal date
2. **Monitors Balance** - Checks if customer has sufficient balance before renewal
3. **Executes Renewal** - On the scheduled date, automatically renews the service
4. **Updates Records** - Creates transactions, updates service dates, and logs activity

Data Source:

The Upcoming Auto-Renews view queries CGRateS directly using the `ApierV1.GetScheduledActions` API and enriches the data with customer and service information from the CRM database.

Renew Now

To manually trigger a renewal before the scheduled date:

1. Click the ◈ **(Renew Now)** button for the service

2. Review the renewal details in the confirmation modal:

   > Customer: Acme Corp Service: FixedWireless_75628fa5 Product: Home Internet Mega Cost: $89.99
   >
   > Next scheduled renewal: Nov 11, 2025 10:45 AM
   >
   > [Cancel] [Confirm Renewal]

3. Click **"Confirm Renewal"** to process immediately

4. A provisioning job is created and executed

5. Monitor progress in the provisioning status modal

What Happens:

- Service is topped up with the product allowances
- Balance is updated in CGRateS
- Transaction record is created
- Service end date is extended
- Activity log entry is created

Use Cases:

- Customer requests early renewal
- Resolve service issues by resetting allowances
- Testing renewal processes
- Customer wants to use service before scheduled renewal

Remove Auto-Renewal

To cancel automatic renewal for a service:

1. Click the ◈ **(Remove Auto-Renew)** button for the service

2. Confirm deletion in the modal:

   > This will remove the scheduled auto-renewal for:
   >
   > Customer: Acme Corp Service: FixedWireless_75628fa5 Next Renewal: Nov 11,

2025 10:45 AM

The service will not automatically renew. You will need to manually renew or the service will expire on the end date.

[Cancel] [Remove Auto-Renewal]

3. Click **"Remove Auto-Renewal"** to confirm
4. The ActionPlan is removed from CGRateS
5. Service will expire naturally unless manually renewed

What Happens:

- ActionPlan is deleted from CGRateS
- Service status remains unchanged
- Service will expire on its current end date
- Customer must manually renew or the service will stop

Use Cases:

- Customer is cancelling service
- Switching to manual renewal process
- Service plan is changing
- Customer requested to stop automatic charges

Warning

Removing auto-renewal means the service will expire unless manually renewed. Customers will not receive automatic service continuation.

Understanding the Display

Customer Column:

- Shows customer name as a clickable link
- Links to customer overview page
- Shows "N/A" if service is not linked to a customer (orphaned service)

Service Column:

- Shows service name/UUID
- Links to customer overview page
- Shows service UUID if name is not set
- Shows "N/A" if service cannot be found in database

Product Column:

- Product name from CGRateS ActionPlan
- Extracted from the scheduled action metadata
- Always shows product name even if service is not found

Cost Column:

- Retail cost from the Product table in CRM
- Amount that will be charged on renewal
- Shows "N/A" if product is not found in database

Renews On Column:

- Exact date and time of next renewal
- Human-readable relative time (e.g., "3 hours from now", "2 days from now")
- Time zone aware (uses CGRateS server timezone)

Status Column:

- Current service status from CRM database
- **Active** (Green) - Service is operational
- **Suspended** (Orange) - Service is temporarily disabled
- **Unknown** (Gray) - Service not found in database or status not set

Troubleshooting

Services showing "N/A" for Customer/Service

- **Cause:** Service UUID in CGRateS doesn't match service_uuid in CRM database
- **Fix:**
  - Verify service exists in database
  - Check that service_uuid format matches: `ServiceType_UUID` (e.g., `FixedWireless_75628fa5`)
  - Service may have been deleted from CRM but still scheduled in CGRateS

Cost showing "N/A"

- **Cause:** Product ID in CGRateS doesn't exist in CRM Product table
- **Fix:**
  - Verify product exists in database
  - Check product_id in the ActionPlan matches a product in CRM
  - Product may have been deleted

"Renews On" not showing date

- **Cause:** NextRunTime not present in CGRateS response
- **Fix:**
  - Check CGRateS ActionPlan configuration
  - Verify ActionTiming is correctly configured
  - Check CGRateS logs for errors

"Renew Now" fails

- **Cause:** Various provisioning errors
- **Troubleshooting:**
  - Check provisioning status modal for error details
  - Verify Ansible playbooks are configured correctly
  - Check OCS connectivity
  - Review provisioning logs

Auto-renewal removed but still showing

- **Cause:** Cache delay or CGRateS sync issue
- **Fix:**
  - Click **"Refresh"** button to reload data
  - Verify ActionPlan was actually removed in CGRateS
  - Check CGRateS API connectivity

Pagination

Services are displayed in pages for performance:

> ◄ Previous 1 [2] 3 4 5 Next ►
>
> Items per page: [10 ▼]
>
> - 10
> - 25
> - 50
> - 100

Controls:

- **Previous/Next** - Navigate pages
- **Page Numbers** - Jump to specific page
- **Items per Page** - Adjust how many services show per page

Performance Tip: Use filters to reduce total results rather than increasing items per page.

Service Badges and Indicators

Visual indicators help identify service states quickly:

Status Badges:

Auto-Renewal Indicator:

Services with auto-renewal enabled show:

Expiring Soon:

Services expiring within 7 days show:

Overdue:

Services with outstanding balance show:

CGRateS Integration (Advanced)

For services integrated with CGRateS (the billing and rating engine), administrators can manage advanced configurations directly from the service view. This includes attributes, filters, and viewing active sessions.

Note

CGRateS integration features require the **cgrates_api_access** permission. Only administrators have access to these features by default. See `rbac` for permission configuration.

Automatic Provisioning vs. Manual Management

In normal operation, CGRateS attributes and filters are automatically provisioned by Ansible during the initial service provisioning workflow. When a new service is created, the provisioning playbooks:

- Create the service account in CGRateS
- Configure attributes (IMSI, MSISDN, account identifiers, speed profiles, etc.)
- Set up filters to ensure correct rating
- Apply the appropriate rating plans

However, there are cases where you may need to modify these configurations after the service has been provisioned:

- **Speed Profile Changes** - Customer upgrades/downgrades bandwidth (MaxBitrateDL/UL)
- **Policy Adjustments** - Change QoS policies or traffic shaping rules (PcefPolicyName)
- **Phone Number Changes** - Update MSISDN or other identifiers
- **Troubleshooting** - Fix misconfigurations or test different settings
- **Special Configurations** - Apply custom attributes not part of standard provisioning

The manual management interface allows administrators to make these changes directly without re-running the entire provisioning workflow. This is particularly useful for:

- **Quick Changes** - Modify a single attribute without waiting for provisioning
- **Testing** - Experiment with different configurations
- **Customer Support** - Resolve issues on the fly during support calls
- **Custom Configurations** - Apply service-specific settings not covered by templates

Warning

Manual changes to CGRateS configuration bypass the standard provisioning workflow. Ensure you understand the impact of your changes, as incorrect configurations can affect billing and service functionality. All changes are logged to the customer activity feed for audit purposes.

Accessing CGRateS Features

When viewing or editing a service that's provisioned in CGRateS, three collapsible sections appear at the bottom of the service form:

- **CGRateS Attributes** - Configure service-specific attributes
- **CGRateS Filters** - Define filtering rules for the service
- **Active Sessions** - View real-time active sessions

Each section is collapsed by default to keep the interface clean. Click the section header to expand and view/edit the configuration.

CGRateS Sections Collapsed View

The collapsed sections show count badges indicating how many attributes, filters, or active sessions exist for the service.

CGRateS Attributes

Attributes allow you to define custom fields and transformations that are applied to rating events for this specific service.

Attribute ID Format: `ATTR_ACCOUNT_{service_uuid}`

Example: For service with UUID `Mobile_SIM_c2880638`, the attribute profile ID is `ATTR_ACCOUNT_Mobile_SIM_c2880638`

Managing Attributes:

1. Open service edit view

2. Expand **"CGRateS Attributes"** section
3. Click **"Edit Attributes"** button
4. Add/modify/remove attributes as needed
5. Click **"Save Attributes"**

CGRateS Attributes Edit View

Attribute Fields:

- **Path** - The field to modify (e.g., `*req.Account`, `*req.IMSI`)
- **Type** - How the value is set:
    - `*constant` - Sets a fixed value
    - `*variable` - Captures value from event fields using RSRParser
    - `*composed` - Appends value instead of overwriting
    - `*usage_difference` - Calculates duration between two fields
    - `*sum` - Sums multiple values
    - `*value_exponent` - Computes exponent of a field
- **Rules** - The value(s) to apply (can have multiple rules per attribute)

Example Attribute Configuration:

```
{
  "Path": "*req.Account",
  "Type": "*constant",
  "Value": [{"Rules": "Mobile_SIM_474a380a"}]
}
```

Common Use Cases:

- Set account identifier for rating
- Map IMSI/MSISDN to service
- Configure bandwidth limits (MaxBitrateDL/UL)
- Set policy names (PcefPolicyName)
- Transform or enrich rating events

Activity Logging:

All attribute modifications are logged to the customer's activity feed with full details of the changes made.

CGRateS Filters

Filters define matching rules that determine when this service's configuration should be applied during rating.

Filter ID Format: `FLTR_ACCOUNT_{service_uuid}`

Example: For service with UUID `Mobile_SIM_c2880638`, the filter ID is `FLTR_ACCOUNT_Mobile_SIM_c2880638`

Managing Filters:

1. Open service edit view
2. Expand **"CGRateS Filters"** section
3. Click **"Edit Filters"** button
4. Add/modify/remove filter rules
5. Click **"Save Filters"**

Filter Rule Fields:

- **Element** - The field to match against (e.g., `~*req.Account`, `~*req.Destination`)
- **Type** - Match type:
  - `*string` - Exact string match
  - `*prefix` - Starts with specified value
  - `*suffix` - Ends with specified value
  - `*empty` - Field is empty
  - `*exists` - Field exists
  - `*notexists` - Field does not exist
  - `*timings` - Match time/date patterns
  - `*destinations` - Match destination patterns
  - `*rsr` - RSR field matching
  - `*gt` / `*gte` / `*lt` / `*lte` - Numeric comparisons
- **Values** - The value(s) to match (can have multiple values per rule)

Example Filter Configuration:

```json
{
  "Element": "~*req.Account",
  "Type": "*string",
  "Values": ["Mobile_SIM_474a380a"]
}
```

Common Use Cases:

- Ensure attributes only apply to specific account
- Filter by destination (national vs. international)
- Time-based filtering (peak vs. off-peak)
- Filter by service type or category

Activity Logging:

All filter modifications are logged to the customer's activity feed.

Active Sessions

View real-time active sessions for this service. This shows ongoing calls, data sessions, or other billable events currently in progress.

Viewing Active Sessions:

1. Open service edit view
2. Expand **"Active Sessions"** section
3. View list of active sessions
4. Click **"View Details"** on any session to see full session data
5. Click **"Refresh"** to reload session list

CGRateS Active Sessions View

Session Information Displayed:

- **Setup Time** - When the session started
- **Usage** - Current session duration (in seconds)
- **Destination** - Called number or destination

Session Details Modal:

Clicking "View Details" opens a modal showing:

- **Basic Information:**
  - CGRID (session ID)
  - Account (service UUID)
  - Setup time
  - Current usage/duration
  - Destination
  - Category
- **Complete Session Data:**
  - Full JSON representation of the session
  - All CGRateS session fields
  - Real-time session state
  - Scrollable JSON viewer for inspection

CGRateS Session Details Modal

Use Cases:

- Monitor active calls or data sessions
- Troubleshoot billing issues
- Verify session is being rated correctly
- Check session attributes and values
- Audit active service usage

Refresh Rate:

Sessions are fetched on-demand when you expand the section. Click "Refresh" to get the latest session data.

Note

Only sessions matching this service's account (service UUID) are displayed. The filter `*string:~*req.Account:{service_uuid}` is automatically applied.

CGRateS API Proxy

All CGRateS operations (attributes, filters, sessions) use the OmniCRM API proxy endpoint:

Endpoint: `POST /crm/ocs/proxy`

Required Fields:

- `method` - CGRateS API method (e.g., `APIerSv1.GetAttributeProfile`)
- `params` - Array of parameters for the method
- `customer_id` - Customer ID (for activity logging)
- `service_id` - Service ID (for activity logging)

Optional Fields:

- `tenant` - CGRateS tenant (defaults to config value)

Example Request:

```
{
  "method": "APIerSv1.GetAttributeProfile",
  "params": [{"ID": "ATTR_ACCOUNT_Mobile_SIM_c2880638"}],
  "customer_id": 123,
```

```
  "service_id": 456
}
```

Tenant Configuration:

The tenant is automatically set from the OmniCRM configuration file (`crm_config.yaml`) under `ocs.ocsTenant`. This ensures all CGRateS operations use the correct tenant without hardcoding values in the frontend.

Permission Requirement:

The `cgrates_api_access` permission is required. This permission is granted to the `admin` role by default.

Activity Logging:

All non-GET CGRateS API operations are automatically logged to the customer's activity feed, including:

- API method called
- Tenant used
- Full parameters sent
- Service ID the operation was performed on
- User who performed the operation
- Timestamp

This creates a complete audit trail of all CGRateS configuration changes.

Troubleshooting CGRateS Integration

"Permission Denied" when accessing CGRateS features

- **Cause:** User lacks `cgrates_api_access` permission
- **Fix:** Grant permission to user's role (typically admin-only feature)

Attributes or Filters not loading

- **Cause:** CGRateS connectivity issue or profile doesn't exist
- **Fix:**
    - Check CGRateS server connectivity in config
    - Verify tenant configuration is correct
    - Check browser console for API errors
    - Profile may not exist yet (will show empty form)

Changes not saving

- **Cause:** Validation error or CGRateS API error
- **Fix:**
    - Check for required fields (Path, Type, Element)
    - Verify JSON format is correct
    - Check activity log for error details
    - Review CGRateS logs

No active sessions showing

- **Cause:** No sessions currently active for this service
- **Fix:**
    - This is normal if service is not in use

    - Try refreshing after initiating a session (call, data, etc.)
    - Verify service UUID matches the account in CGRateS

Session details not updating in real-time

- **Cause:** Session data is fetched on-demand, not live
- **Fix:** Click "Refresh" button to get latest session data

Activity log not showing CGRateS changes

- **Cause:** Only non-GET operations are logged (reads are not logged)
- **Fix:** This is by design - only writes/modifications create activity entries

Common Workflows

Workflow 1: Find Customer's Services

1. Click **Service Type filter** (optional)
2. Click **Customer filter**
3. Type customer name
4. Select customer from dropdown
5. Review customer's services

Workflow 2: Identify Expiring Services

1. Click **"Active"** tab
2. Sort by **"End Date"** (ascending)
3. Services expiring soonest appear first
4. Contact customers for renewal

Workflow 3: Clean Up Test Services

1. Search for "test" in search box
2. Review results to confirm they're test data
3. Select all test services
4. Click **"Delete Selected"**
5. Confirm deletion

Workflow 4: Suspend Non-Paying Customer

1. Navigate to customer account
2. View services tab
3. Select all active services
4. Change status to "Suspended"
5. Save changes

Workflow 5: View Mobile Service Usage

1. Filter by **Service Type: Mobile**
2. Click service name to open details
3. Click **"Usage"** tab
4. Review data/voice/SMS consumption
5. Identify heavy users or overages

Troubleshooting

Services not appearing in list

- **Cause:** Status filter hiding results
- **Fix:** Click "All Results" tab to show all statuses

Cannot find service by search

- **Cause:** Search term doesn't match stored data
- **Fix:**
  - Try partial search (e.g., "0770" instead of full number)
  - Use customer filter instead
  - Check for typos

Bulk delete button disabled

- **Cause:** No services selected
- **Fix:** Check boxes next to services you want to delete

Sort not working

- **Cause:** Column not sortable
- **Fix:** Only columns with ▲▼ icons are sortable

Page loads slowly

- **Cause:** Too many services to display
- **Fix:**
  - Apply filters to reduce result set
  - Reduce items per page
  - Use search to narrow results

Related Documentation

- `csa_add_service` - Adding new services
- `csa_modify` - Modifying services and adding add-ons
- `concepts_products_and_services` - Product and service concepts
- `basics_customers` - Customer management

Service Usage and Balance Tracking

The Service Usage system provides **real-time monitoring** of customer consumption for data, voice, SMS, and monetary balances. This feature integrates with the OCS (Online Charging System) to display current usage, remaining allowances, and balance expiry information to both customers (via `Self-Care Portal <self_care_portal>`) and staff.

Overview

Usage tracking enables:

- **Real-time Balance Display** - View current usage and remaining allowances
- **Multiple Balance Types** - Track data, voice, SMS, and monetary balances simultaneously
- **Expiry Monitoring** - See when balances expire
- **Balance Breakdown** - Detailed view of individual balance buckets
- **Auto-Refresh** - Usage updates every 3 seconds automatically

Accessing Service Usage

From Service List:

1. Navigate to **Services → Service List**
2. Click the **actions menu (⋮)** next to a service
3. Select **"Usage"**

From Service Details:

1. Open a service's detail page
2. Click the **"Usage"** tab

From Customer Page:

1. Open customer overview
2. Navigate to **Services** tab
3. Click **"View Usage"** next to any service

The usage modal or page opens showing real-time consumption data.

Usage Display

The usage interface shows summary cards and detailed progress bars for each balance type.

Service Usage Display
Service Balance Modal

Summary Cards

Top row displays quick-view cards for each balance type:

Card Information:

- **Balance Type** - Icon and label (Data, Voice, SMS, Monetary)

- **Remaining Amount** - Current balance in appropriate units
- **Expiry Time** - Days/hours until balance expires
- **More Info Button** - Click to expand detailed breakdown

Progress Bars

Below the cards, progress bars show consumption visually with filled portions indicating remaining balance.

Progress Bar Features:

- **Visual Indicator** - Filled portion shows remaining balance
- **Percentage** - Numeric percentage of balance remaining
- **Absolute Values** - Shows used vs. total (e.g., "12.5GB / 20GB")
- **Color Coding**:
  - Green: >50% remaining
  - Yellow: 20-50% remaining
  - Red: <20% remaining
- **Clickable** - Click to expand detailed breakdown

Balance Types

Data Balance

Tracks internet data consumption.

Units: Gigabytes (GB) or Megabytes (MB)

Display Format:

    Progress: 12.5GB / 20GB (62%)

Common Scenarios:

- **Mobile data plans** - 5GB, 10GB, 20GB monthly allowances
- **Fixed wireless** - Unlimited or capped at high amounts (500GB, 1TB)
- **Top-ups** - Additional data purchased mid-cycle
- **Dongle services** - Prepaid data for hotspot devices

Multiple Buckets:

Services often have multiple data balances:

- Monthly allowance (expires monthly)
- Bonus data (expires after campaign period)
- Top-up data (shorter expiry, consumed first)

Voice Balance

Tracks phone call minutes.

Units: Minutes (min)

Display Format:

    Progress: 125 min / 500 min (25%)

Call Duration Tracking:

- Incoming calls (if charged)
- Outgoing calls
- International calls (separate bucket if applicable)
- Premium numbers

Calculation:

Voice usage is calculated by call duration in nanoseconds internally, converted to minutes for display.

SMS Balance

Tracks text message usage.

Units: Messages (msgs)

Display Format:

> Progress: 45 / 250 (18%)

Message Types:

- Standard SMS (160 characters)
- Long SMS (multiple segments)
- MMS (if separately tracked)

Monetary Balance

Tracks prepaid credit or account balance.

Units: Currency (£, $, €, etc.)

Display Format:

> Progress: £15.50 / £20.00 (77%)

Usage:

- Prepaid accounts use monetary balance to pay for usage
- Credit decrements as customer uses services
- Can be topped up via payment or voucher
- May expire if not used within validity period

Detailed Balance Breakdown

Click **"More Info"** on any card or click a progress bar to expand the detailed breakdown.

Expanded View:

> ⯑ Monthly Allowance 20GB
> Remaining: 12.5 GB Used: 7.5 GB Expires: 25 Jan 2025 (15 days) Weight: 10
>
> ⯑ Bonus Data 5GB
> Remaining: 5.0 GB Used: 0 GB Expires: 31 Jan 2025 (21 days) Weight: 20

⯑ Top-Up Data 3GB
Remaining: 0 GB Used: 3.0 GB Expires: 18 Jan 2025 (EXPIRED) Weight: 30

Total Remaining: 17.5 GB

Balance Bucket Fields:

- **ID/Name** - Identifier for the balance bucket
- **Remaining** - Amount left in this specific bucket
- **Used** - Amount consumed from this bucket
- **Expiry Date** - When this balance expires
- **Weight** - Priority order (higher weight consumed first)

Weight System

Balances have a **weight** value that determines consumption order:

- **Higher weight = consumed first**
- **Lower weight = consumed last**

Example Weights:

- Top-up data: Weight 30 (consumed first, shorter expiry)
- Bonus data: Weight 20 (consumed second)
- Monthly allowance: Weight 10 (consumed last, longest expiry)

This ensures that expiring balances are used before longer-lasting ones.

Real-Time Updates

Usage data refreshes automatically every **3 seconds** via polling.

What Updates:

- Current balance amounts
- Usage progress bars
- Expiry timers
- Individual bucket details

User Experience:

- No page reload required
- Smooth updates without flicker
- Loading overlay during refresh
- Status badge shows current service state

Use Cases:

- Monitor customer usage during call
- Watch balance decrement in real-time as customer uses service
- Verify top-up immediately after purchase

Usage in Different Service Types

Mobile Services

Display all four balance types:

- Data (GB)
- Voice (minutes)
- SMS (messages)
- Monetary (currency)

Example:

> DATA: 12.5GB remaining VOICE: 125 min remaining SMS: 45 msgs remaining MONETARY: £15.50 remaining

Fixed Wireless / Internet

Typically shows only:

- Data (GB or TB)
- Monetary (if prepaid)

Example:

> DATA: 450GB / 500GB remaining MONETARY: £45.00 (prepaid credit)

Hotspot / Dongle Services

Shows dongle-specific data tracking:

- Data (consumed vs. prepaid)
- Monetary (prepaid balance)

Display Mode:

When `dongle=true`, the component hides voice and SMS, showing only relevant data and monetary balances.

Troubleshooting

Usage showing as 0 / 0

- **Cause:** Service not integrated with OCS or CGRateS
- **Fix:**
    - Verify service is provisioned in OCS
    - Check OCS API connectivity
    - Review service UUID mapping

Usage not updating

- **Cause:** Polling stopped or OCS unreachable
- **Fix:**
    - Refresh the page
    - Check browser console for errors
    - Verify OCS API is online

Balances show incorrect amounts

- **Cause:** OCS data mismatch or caching issue
- **Fix:**
    - Force OCS balance refresh
    - Check for pending transactions
    - Verify OCS configuration

Expiry dates missing

- **Cause:** Balance has no expiry set
- **Fix:**
    - Some balances are set to never expire (unlimited validity)
    - Check balance configuration in OCS

Multiple balances confusing

- **Cause:** Multiple top-ups or bonus data added
- **Fix:**
    - Use detailed breakdown view to see all buckets
    - Sort by weight to see consumption order
    - Review individual expiry dates

Integration with OCS/CGRateS

Usage data comes from the **OCS (Online Charging System)**, typically CGRateS.

Data Flow:

1. User opens usage view
2. OmniCRM calls `GET /crm/service/{service_id}`
3. API queries OCS via service UUID
4. OCS returns balance map:

```json
{
  "BalanceMap": {
    "*data": [
      {
        "ID": "monthly_data_20GB",
        "Value": 13421772800,
        "ExpiryTime": "2025-01-25T23:59:59Z",
        "Weight": 10
      }
    ],
    "*voice": [
      {
        "ID": "monthly_voice_500min",
        "Value": 7500000000000,
        "ExpiryTime": "2025-01-25T23:59:59Z",
        "Weight": 10
      }
    ],
    "*sms": [
      {
        "ID": "monthly_sms_250",
        "Value": 250,
        "ExpiryTime": "2025-01-25T23:59:59Z",
```

```
        "Weight": 10
      }
    ],
    "*monetary": [
      {
        "ID": "prepaid_credit",
        "Value": 1550,
        "ExpiryTime": "2025-02-25T23:59:59Z",
        "Weight": 10
      }
    ]
  }
}
```

5. UI converts values to display units (bytes → GB, nanoseconds → minutes)
6. Progress bars and cards rendered
7. Polling continues every 3 seconds

OCS Balance Types Mapping:

The OCS returns balance data with type prefixes that map to UI display:

- `*data` → **DATA** card (internet usage)
- `*voice` → **VOICE** card (call minutes)
- `*sms` → **SMS** card (text messages)
- `*monetary` → **MONETARY** card (prepaid credit)

Each balance type can have multiple buckets (e.g., monthly allowance + bonus data + top-up data), all displayed in the detailed breakdown view.

Balance Value Conversions:

- **Data:** Bytes → GB (divide by 1024^3)
- **Voice:** Nanoseconds → Minutes (divide by 60×10^9)
- **SMS:** Count (no conversion)
- **Monetary:** Cents → Currency (divide by 100)

Auto-Renew and ActionPlans

Services with auto-renew enabled have **ActionPlans** scheduled in the OCS.

What are ActionPlans?

ActionPlans are scheduled tasks in CGRateS that automatically execute at specific times to:

- Add balance to an account (auto top-up)
- Renew monthly allowances
- Apply recurring charges
- Expire old balances

How Auto-Renew Works:

1. **Service Provisioning:**
    - When service created with `auto_renew = true`
    - Provisioning playbook creates ActionPlan in OCS
    - ActionPlan configured to run monthly (or per billing cycle)

2. **ActionPlan Configuration:**

   ActionPlan contains:

   - **Account ID** - Service UUID
   - **Actions** - What to do (add data, voice, SMS, monetary balance)
   - **Schedule** - When to execute (e.g., monthly on 1st at 00:00 UTC)
   - **Amount** - How much balance to add

3. **Automatic Execution:**

   - OCS executes ActionPlan at scheduled time
   - Adds balance to account (e.g., 20GB data, 500 minutes voice)
   - Sets expiry date for new balance (e.g., 30 days)
   - Customer charged via payment method on file

4. **Viewing ActionPlans:**

   - Navigate to service details in OCS view
   - ActionPlans listed with next execution time
   - Shows: Plan name, next run date, action details

Example ActionPlan:

```
{
  "ActionPlanId":
"ProductID_MonthlyPlan__ProductName_20GB_Mobile__ActionPlan_Monthly_Renew",
  "NextExecTime": "2025-02-01T00:00:00+00:00",
  "ActionName_hr": "Monthly Renew",
  "PlanName": "20GB Mobile",
  "ActionFrequency_hr": "Every MonthlyPlan",
  "custom_NextExecTime_hr": "in 22 days"
}
```

Managing Auto-Renew:

- **Enable** - Set during service creation or modification
- **Disable** - Remove ActionPlan from OCS (service keeps existing balance but won't auto-renew)
- **Modify** - Change renewal amount or frequency via service modification

Manual Renewal:

If auto-renew disabled, customer must manually:

- Top-up before balance expires
- Or service suspends when balance depletes

Viewing in UI:

Services tab shows auto-renew status:

> Next Renewal: 1 Feb 2025 (in 22 days) Renewal Amount: £15.00

Best Practices

For Support Staff:

- Check usage before answering "Why is my service slow?" calls
- Verify balance after top-ups to confirm success
- Use detailed breakdown to identify expired buckets
- Monitor high-usage customers to prevent overages

For Customers (Self-Care):

- Check usage regularly to avoid running out
- Top up before balance expires
- Understand weight system to know which balance is consumed first
- Contact support if usage seems incorrect

For Administrators:

- Configure appropriate balance expiries
- Set weight values to prioritize expiring balances
- Monitor OCS connectivity for accurate reporting
- Review balance configurations match product offerings

Related Documentation

- `features_topup_recharge` - Top-up system for adding balance
- `csa_service_management` - Managing services
- `csa_modify` - Adding add-ons to increase allowances
- `concepts_products_and_services` - Product configuration

Customer Care

User Impersonation for Support and Troubleshooting

Impersonation allows authorized staff to temporarily log in as another user to troubleshoot issues, verify configurations, or see exactly what the user experiences. This feature is essential for customer support but requires appropriate permissions and is fully audited.

When impersonating a customer, staff access the Self-Care Portal <self_care_portal> exactly as the customer sees it, allowing for accurate troubleshooting and support.

See also: RBAC <rbac> for permission configuration, Customers <basics_customers> for customer management, Self-Care Portal <self_care_portal> for customer portal features.

Purpose

User impersonation provides:

1. **Troubleshooting** — See exactly what the customer sees to diagnose issues
2. **Verification** — Confirm service configurations and permissions work correctly
3. **Training** — Demonstrate features from the customer's perspective
4. **Support** — Help customers navigate the system without requiring screen sharing
5. **Audit Trail** — All impersonation sessions are logged for security and compliance

Required Permissions

To impersonate users, you must have one of the following permissions:

- can_impersonate — Dedicated impersonation permission for support staff
- admin — Full administrative access (includes impersonation rights)

Users without these permissions cannot access the impersonation feature.

How to Impersonate a User

Via Web UI:

1. **Navigate to Customer** — Find the customer in the CRM
2. **Select Contacts** — View the customer's contact list
3. **Click "Login as User"** — Button appears next to each contact that has a user account
4. **Confirm Impersonation** — System may prompt for confirmation
5. **Session Starts** — You are now logged in as that user

Logging in as a User in OmniCRM

Via API:

Start impersonation session:

Endpoint: `POST /auth/impersonate`

Required Permission: `can_impersonate` or `admin`

Request:

```json
{
  "user_id": 42
}
```

Response:

```json
{
  "success": true,
  "impersonating_user_id": 1,
  "target_user_id": 42,
  "impersonation_start": "2025-01-04T15:30:00Z",
  "access_token": "new_token_for_impersonated_user",
  "refresh_token": "new_refresh_token"
}
```

The returned tokens are for the impersonated user's session.

What Happens During Impersonation

When you impersonate a user:

- **Full Context** — You see the system exactly as the target user sees it:
  - Their dashboard and navigation
  - Their customer data (if customer user)
  - Their permissions and access controls

    - Their services, invoices, and usage
- **Session Tracking** — The system tracks both identities:
    - `impersonating_user_id` — Your real user ID
    - `target_user_id` — The user you're impersonating
    - `impersonation_start` — When impersonation began
- **Indicator** — The UI displays a banner showing you are impersonating someone:
    - "You are currently logged in as [Username]"
    - "Click here to stop impersonation"
- **Audit Logging** — All actions are logged with both user IDs:
    - Actions appear as performed by the target user
    - Audit logs record who was actually performing them (impersonating user)
    - Complete audit trail maintained in `ImpersonationLog` table

Stopping Impersonation

Via Web UI:

1. **Click Banner** — Click the impersonation banner at the top of the page
2. **Or Navigate** — Go to user menu and select "Stop Impersonation"
3. **Confirmation** — Session ends and you return to your own account

Via API:

Endpoint: `POST /auth/stop_impersonation`

Request: No body required (authenticated request)

Response:

```json
{
  "success": true,
  "impersonation_end": "2025-01-04T15:45:00Z",
  "duration_seconds": 900,
  "access_token": "your_original_token",
  "refresh_token": "your_original_refresh_token"
}
```

Your original session is restored.

Impersonation Audit Logging

All impersonation sessions are logged in the `ImpersonationLog` table with:

- **impersonating_user_id** — Staff member who performed impersonation
- **target_user_id** — Customer or user who was impersonated

- **impersonation_start** — Start timestamp
- **impersonation_end** — End timestamp (when session was stopped)
- **impersonation_duration** — Duration in seconds

This provides complete accountability for all impersonation sessions and allows:

- **Security audits** — Review who impersonated whom and when
- **Compliance reporting** — Demonstrate proper use of elevated access
- **Investigation** — Track actions during impersonation sessions
- **Monitoring** — Identify unusual impersonation patterns

Viewing Impersonation Logs:

Administrators can query impersonation logs via API:

```
GET /auth/
impersonation_logs?user_id={user_id}&start_date={date}&end_date={date}
```

Filter by:

- Impersonating user (who performed the impersonation)
- Target user (who was impersonated)
- Date range
- Duration

Best Practices

1. **Minimize Duration** — Impersonate only as long as needed to resolve the issue
2. **Document Purpose** — Note why impersonation was needed in customer activity log
3. **Inform Customers** — Let customers know you may need to view their account (privacy policy)
4. **Verify Identity** — Confirm user identity before impersonating via their account
5. **Review Logs** — Regularly audit impersonation logs for unusual patterns
6. **Limit Permissions** — Only grant `can_impersonate` to support staff who need it
7. **Training** — Ensure staff understand the responsibility and audit implications

Security Considerations

- **Full Access** — Impersonation grants complete access to the target user's account
- **No Password Required** — Impersonation bypasses authentication (permission-based only)

- **Logged Actions** — All actions during impersonation are attributed to target user in application logs (but audit logs show real actor)
- **Session Isolation** — Impersonation creates a new session; doesn't affect target user's active sessions
- **Time Limits** — Impersonation sessions should be time-limited (configurable)
- **MFA Bypass** — Impersonation bypasses 2FA requirements (uses impersonator's authentication)

Restrictions

- **Cannot Impersonate Admins** — Depending on configuration, may not be able to impersonate other administrators
- **Same Permissions** — You get target user's permissions, not union of both users' permissions
- **Session Limits** — Only one impersonation session per staff member at a time
- **Audit Requirements** — Cannot disable or hide impersonation logging

Troubleshooting Common Issues

Issue: "Login as User" button doesn't appear

- Solution: Verify you have `can_impersonate` or `admin` permission
- Solution: Confirm the contact has a linked user account

Issue: Impersonation fails with permission error

- Solution: Check if target user is an administrator (may be restricted)
- Solution: Verify your impersonation permission is active

Issue: Cannot stop impersonation

- Solution: Use API endpoint to stop impersonation: `POST /auth/stop_impersonation`
- Solution: Clear browser cookies and log in again with your credentials

Issue: Actions not logging correctly

- Solution: Verify impersonation session is active (check banner)
- Solution: Review audit logs - actions are logged with both user IDs

Cell Broadcast System

The Cell Broadcast System in OmniCRM enables mobile network operators to send emergency alerts and public warnings to mobile devices within specific geographic areas. Cell Broadcast is a critical public safety feature used for AMBER alerts, weather warnings, tsunami alerts, and other emergency notifications.

Key Advantage: Unlike standard SMS messaging, Cell Broadcast messages will audibly alert on phones that are on silent, out of credit or roaming. As this is a broadcast message, it is possible to send an alert to every member of the population carrying a mobile phone in a matter of seconds.

Omnitouch Warning Link (OWL)

The Omnitouch Warning Link (OWL) platform provides a comprehensive solution for disaster management professionals and mobile network operators:

- **Cell Broadcast Entity (CBE)** - Secure web-based application for authorized users to create and broadcast emergency warning messages
- **Cell Broadcast Center (CBC)** - Standards-compliant network integration component that connects to cellular networks (2G/3G/4G/5G) to distribute messages

OWL is designed for use on any device with a web browser (Chrome/Firefox/Safari/Edge), such as computers, laptops, tablets or mobile phones.

Overview

Cell Broadcast (also known as Public Warning System or PWS) allows operators to:

- **Send Emergency Alerts** - Distribute critical safety information to all devices in an area
- **Target Geographic Regions** - Broadcast to specific tracking areas or network cells by controlling which cell towers broadcast messages
- **Support Multiple Languages** - Provide alert messages in multiple languages simultaneously (up to 500 characters per language)
- **Manage Alert Lifecycle** - Create, update, approve, monitor and delete broadcast messages
- **Integrate with External Systems** - Connect with CBC (Cell Broadcast Center) infrastructure via multiple cellular network interfaces
- **Two-Factor Authentication** - Secure approval process using Time-based

One-Time Passwords (TOTP)
- **Two-Person Rule** - Optional requirement for approval from a second person before message transmission

Unlike SMS, Cell Broadcast does not require subscriber lists and can reach all capable devices in a geographic area instantly, making it ideal for time-critical public safety alerts. In most networks, a broadcast to all devices takes under 10 seconds.

Use Cases

Cell Broadcast is used for:

- **Emergency Warnings** - Natural disasters (earthquakes, tsunamis, floods, fires)
- **AMBER Alerts** - Child abduction notifications
- **Weather Alerts** - Severe weather warnings, tornado alerts
- **Public Safety** - Terrorist threats, chemical spills, evacuations
- **Test Messages** - System testing and public awareness campaigns (Monthly Test Messages do not alert users but will be received)

Geographic targeting is often of paramount importance in emergency situations. For example, Tsunami alerts advising those near coastal areas to seek higher ground should not be sent to people in inland areas far from the hazard. By controlling which cell towers broadcast emergency warning messages, the scope of transmission can be limited to the appropriate geographical area.

Emergency Warning Message Lifecycle

Transmitting an Emergency Warning message requires quick action while providing accurate information and authentication to ensure message validity.

The message lifecycle consists of four stages:

1. **Message Definition** - Type of message, message content, expiry settings
2. **Message Targeting** - Geographic areas and cell towers to broadcast to
3. **Message Approval** - Authorization/verification of operator identity and second person approval (if required)
4. **Message Review** - Final confirmation before transmission

After broadcast, messages can be monitored, updated as situations evolve, and stopped when immediate danger subsides.

Stage 1: Message Definition

This step defines the basic parameters of the emergency warning message to be broadcast:

Message Identifier
Different types of messages have different identifiers, which are treated differently by receiving phones. For example, a Monthly Test Message should not alert actual users but they will still receive the message.

Note: Each message template available in the OWL system already has the appropriate Message Identifier (MI) embedded, so system users do not need to manually select this when creating an alert message.

Message Text
Text body limited to 500 characters containing the message to be displayed to end users. Messages can be provided in **multiple languages** by adding the second language text underneath the first. Remember the 500-character limit applies to the total message including all languages.

Message Templates
Hazard alert messages can be predefined in advance as "templates" for different envisioned scenarios such as:

- Floods
- Tsunamis
- Earthquakes
- Periodic tests
- Other disaster scenarios

Templates save time during emergencies. These templates can be modified as needed when defining the message, or messages can be scripted from scratch.

Message Expiry and Repetitions
Emergency messages have a finite lifetime for which they are relevant. When defining the message:

- **Expires (minutes)** - How long the message will continue to be broadcast
- **Message Repetitions** - How many times it will be retransmitted

Each phone will only show the message to the user once. However, cell sites will continue to transmit messages until the expiry time is reached to ensure people entering the coverage area from outside will receive the message.

Stage 2: Targeting

Cell Broadcast messages are sent at the cell tower level, and geographic reach can be limited by selecting which towers broadcast the message.

Optional Targeting
This step is optional. Not entering any targeting information means all cell towers will transmit the emergency warning message.

Predefined Target Areas

The OWL system has a database of all cell towers and can define target areas on a map. Areas can be targeted using predefined zones (determined in advance for quick selection) or by drawing custom areas on the map.

Map Drawing Tools
Custom target areas can be created using:

- Polygon tool - Draw precise coverage boundaries
- Circle tool - Quick radius-based alerts
- Rectangle tool - Grid-aligned coverage

The "Add New Zone" feature allows defining custom target areas that can be saved for future use.

Stage 3: Approval

An approval process validates that the person issuing the Emergency Warning message is authorized to do so.

Two-Factor Authentication
Uses Time-based One-Time Passwords (TOTP) via:

- **Physical token** (like an RSA SecurID)
- **App-based solution** (Google Authenticator, Authy, Microsoft Authenticator, or other TOTP-compatible apps)

Users who will be creating or approving draft message alerts must have an authenticator app on their smartphone to generate the authorization code that the system will request.

Setting Up 2FA
When first configuring 2FA:

1. Install an authenticator app on your smartphone (Google Authenticator, Authy, Microsoft Authenticator, etc.)
2. Navigate to your OWL account settings and scan the QR code with your authenticator app
3. Enter the verification code to confirm setup
4. Save backup codes in a secure location
5. Test code generation before emergencies

For detailed 2FA setup instructions, see Two-Factor Authentication <2fa>.

If you change mobile devices or the app stops syncing with your OWL account, contact your System Administrator for help. Administrators can reset 2FA tokens from the **Users and Roles → Users** page.

Two-Person Rule
Where the process requires approval from a second person, the person issuing

the alert must input the Time-based One-Time Password of the other person before the process will be allowed to proceed. This provides oversight and minimizes risk of misuse.

Granular User Roles
Individual user roles can be configured to:

- Allow only certain users to send predefined messages
- Restrict message targeting to specific regions
- Require additional approval workflows

Stage 4: Review

Once Message Definition, Targeting, and Approval stages are complete, the operator must review the message before final broadcast. Once satisfied with the message details, they can transmit the message.

Transmission Speed: In most networks, a broadcast to all devices in the network takes under 10 seconds.

Stage 5: Monitoring and Updates

Once the message broadcast is started, operators can monitor and manage transmitted messages.

Network Feedback
Cellular networks return information about cell sites that have broadcast the message. If a cell site is offline or unavailable, this will be reported back to the operator.

Automatic Retransmission
Should any offline cell sites become available again while the Emergency Warning is still active, all phones connected to that cell will receive the message.

In-Flight Updates
Once broadcast, the message can be:

- Updated as the situation evolves
- Modified with new message body content
- Recalled/stopped at any time

Historical Records
All information about historical messages can be viewed and reviewed for audit purposes.

Message Structure

Each Cell Broadcast message consists of:

Message Configuration

- **Message Identifier** - Unique identifier for the alert type (e.g., 4370 for ETWS Earthquake, 4371 for ETWS Tsunami)
- **Category** - Alert category (normal, emergency, high, extreme)
- **Repetition Period** - Seconds between broadcast repetitions
- **Number of Broadcasts** - How many times to broadcast the message
- **Warning Period** - Duration in seconds the warning is valid
- **Channel Indicator** - Type of channel used for broadcast

Localized Messages

Each CBC message can include multiple language variants:

- **Language** - ISO language code (en, es, fr, zh, etc.)
- **Message Body** - Alert text in that language (up to 1395 characters)

The system automatically broadcasts all language variants, allowing recipients to view alerts in their preferred language.

Tracking Areas

Defines geographic targeting for the alert:

- **Tracking Area** - Geographic identifier (cell ID, tracking area code)
- **Operator** - Mobile network operator code (MCC-MNC)
- **RAT Type** - Radio Access Technology (LTE, 5G, UMTS, GSM)

Multiple tracking areas can be specified to cover larger regions or multiple operators.

Creating a Cell Broadcast Message

Via Web UI:

1. **Navigate to Cell Broadcast** - Access the CBC management interface from the main navigation
2. **Click "Create Alert"** - Opens the message creation form
3. **Configure Message Parameters:**
    - Message Identifier (e.g., 4370 for earthquake alerts)
    - Category (normal, high, extreme)
    - Repetition Period (typically 5-60 seconds)
    - Number of Broadcasts (999 for continuous, or specific count)
    - Warning Period (duration in seconds)
    - Channel Indicator (typically "basic")

Cell Broadcast Message Definition

4. **Add Localized Messages:**
    - Click "Add Language"
    - Select language from dropdown
    - Enter message text (max 1395 characters for GSM7, less for Unicode)
    - Repeat for additional languages
5. **Define Tracking Areas:**
    - Click "Add Tracking Area"
    - Enter tracking area code
    - Select operator (MCC-MNC combination)
    - Choose RAT type (LTE, 5G, etc.)
    - Repeat for additional geographic areas

Cell Broadcast Tracking Area Selection

6. **Review and Create** - Verify all details and click "Create Alert"

Cell Broadcast Message Review

Via API:

Endpoint: `PUT /crm/cbc/`

Required Permission: `CREATE_CBC_MESSAGE`

Request Body:

```json
{
  "messageIdentifier": "4370",
  "category": "emergency",
  "repetitionPeriod": 10,
  "numberOfBroadcasts": 999,
  "warningPeriodSec": 3600,
  "channelIndicator": "basic",
  "localized_messages": [
    {
      "language": "en",
      "messageBody": "EARTHQUAKE WARNING: Magnitude 6.5 earthquake detected. Take cover immediately. Drop, Cover, Hold On."
    },
    {
      "language": "es",
      "messageBody": "ADVERTENCIA DE TERREMOTO: Terremoto de magnitud 6.5 detectado. Cúbrase inmediatamente. Agáchese, Cúbrase, Agárrese."
    }
  ],
  "tracking_areas": [
    {
      "tracking_area": "12345",
```

```
      "operator": "310-410",
      "rat_type": "LTE"
    },
    {
      "tracking_area": "12346",
      "operator": "310-410",
      "rat_type": "5G"
    }
  ]
}
```

Response:

```
{
  "cbc_message_id": 123,
  "cbc_unique_id": "550e8400-e29b-41d4-a716-446655440000",
  "messageIdentifier": "4370",
  "category": "emergency",
  "repetitionPeriod": 10,
  "numberOfBroadcasts": 999,
  "warningPeriodSec": 3600,
  "channelIndicator": "basic",
  "initiating_user": 5,
  "approving_user": null,
  "created": "2025-01-10T14:30:00Z",
  "localized_messages": [...],
  "tracking_areas": [...]
}
```

The message is immediately sent to the Cell Broadcast Center for transmission.

Managing Existing Messages

View All Messages

Endpoint: `GET /crm/cbc/`

Required Permission: `VIEW_CBC_MESSAGE`

Returns list of all CBC messages with their status, timestamps, and configuration.

Update a Message

Endpoint: `PATCH /crm/cbc/{cbc_message_id}`

Required Permission: `UPDATE_CBC_MESSAGE`

Updates message content, tracking areas, or broadcast parameters. Updated messages are re-sent to the CBC.

Request Body:

```
{
  "cbc_message_id": 123,
  "numberOfBroadcasts": 500,
  "localized_messages": [
    {
      "language": "en",
      "messageBody": "UPDATED: Earthquake warning still in effect.
Aftershocks possible."
    }
  ]
}
```

Delete a Message

Endpoint: `DELETE /crm/cbc/{cbc_message_id}`

Required Permission: `DELETE_CBC_MESSAGE`

Removes the message from the database and attempts to cancel it on the CBC.

Approval Workflow

Cell Broadcast messages support an optional approval workflow for high-stakes alerts:

1. **Initiating User** - Staff member who creates the alert (`initiating_user` field)
2. **Approving User** - Manager who approves the alert before broadcast (`approving_user` field)

If approval is required:

- Message is created with `approving_user = null`
- Message is held in "pending approval" state
- Approving user reviews message and either approves or rejects
- On approval, `approving_user` is set and message is broadcast

This workflow is configurable based on organization policy.

Message Identifiers

Standard message identifiers follow 3GPP TS 23.041:

Earthquake and Tsunami Warning System (ETWS):

- **4370** - ETWS Earthquake Warning
- **4371** - ETWS Tsunami Warning
- **4372** - ETWS Earthquake and Tsunami Combined Warning
- **4373-4378** - ETWS Other Emergency Types
- **4379** - ETWS Test Message

Commercial Mobile Alert System (CMAS) / Wireless Emergency Alerts (WEA):

- **4352** - Presidential Alert
- **4353-4355** - Extreme Alerts
- **4356-4359** - Severe Alerts
- **4360-4363** - AMBER Alerts
- **4364-4367** - Public Safety Messages
- **4368-4369** - State/Local Tests
- **4380-4381** - Test Messages

Custom Ranges:

- **0-999** - Reserved for operator-specific alerts
- **1000-4095** - Custom message types

Integration with Cell Broadcast Center

The Cell Broadcast Entity (CBE) needs a mechanism to deliver messages to individual cellular networks. The OWL Cell Broadcast Center (CBC) connects to each cellular network to send Emergency Warning messages to the public.

Multi-Network Redundancy
In the event that one cellular network operator is unavailable (outage or no coverage), if another operating cellular network is available, users will still receive Emergency Warning messages via the other available networks.

Per-Operator CBC Instances
To ensure there is no connection between competing networks, OWL runs a separate CBC instance for each MNO; it is not shared between operators.

OWL CBC Configuration

The CBC URL is configured in `crm_config.yaml`:

```yaml
cbc_url: "http://cbc.example.com:8080"
```

Message Transmission:

When a CBC message is created or updated:

1. OmniCRM CBE stores the message in its database
2. Message is formatted for the CBC API
3. HTTP POST request sent to `{cbc_url}/alerts/send`
4. CBC connects to cellular network via appropriate interface (CBSP, SBc-AP, or N50)
5. CBC acknowledges receipt and begins broadcast
6. Devices in the target tracking areas receive the alert

Message Deletion:

When a message is deleted, the CBE attempts to cancel it on the CBC to stop ongoing broadcasts.

Cellular Network Integration Points

Different generations of cellular networks (2G/3G/4G/5G) each have unique interfaces for connecting to/from the Cell Broadcast Center. Based on the technologies used by each cellular network, the correct interface must be configured.

OWL Cell Broadcast Center supports 2G, 3G, 4G and 5G Cell Broadcast interfaces and has integrated with numerous commonly used cellular network components.

CBSP - 2G/3G - Base Station Controller (BSC)

The Cell Broadcast Service Protocol (CBSP) interface links the CBC to the Base Station Controller (BSC) controlling 2G (GSM) base stations.

- Used for 2G and 3G Cell Broadcast messaging with combined Radio Network Controller/Base Station Controller deployments
- Can be configured either as a client or server arrangement depending on the BSC vendor
- A connection must be established between all BSCs in the network and the OWL CBC
- Interfaces are continuously monitored with alerting to indicate if a CBSP link has gone down

Note: 3GPP defined Service Area Broadcast Protocol (SABP) for use in standalone RNC deployments. This can be used if required when CBSP is not supported for 3G cells on a combined RNC/BSC, however additional testing and support from the RNC vendor may be required.

SBc-AP - 4G/5G Non-Standalone - MME/IWF

The SBc-AP interface links the OWL CBC to the MME (Mobility Management Entity) serving 4G and 5G eNodeB/gNodeBs.

- Used for 4G LTE networks
- Also used for Non-Standalone 5G (the majority of deployments as of 2025)

- A connection must be established between all MMEs in the network and the OWL CBC
- Interfaces are continuously monitored with alerting to indicate if a SBc-AP link has gone down

N50 - 5G Standalone - AMF

For Standalone 5G networks, the N50 interface connects the OWL CBC to the AMF (Access and Mobility Management Function) serving 5G gNodeBs.

- Interface is present in the OWL CBC
- Has not been extensively tested with 3rd party AMFs due to the small number of commercially available 5G SA networks in 2025
- Will be fully supported as 5G SA deployments become more common

MNO Networking Requirements

Networking must be in place between the OWL CBC and the Mobile Network Operator's network to reach the interfaces outlined above.

This is handled case-by-case, but generally requires:

- Dedicated cross-connect/fiber between the CBC and MNO network
- Each interface logically separated
- Connectivity to each integration point (MME, RNC, BSC) in the cellular network

Supported Network Equipment

OWL CBC has been tested and integrates with commonly used cellular network components from major vendors:

Cell Site Data Integration

OWL supports automatic data scraping from:

- **Nokia NetAct**
- **Huawei U2000 / U2020**
- **ZTE NetNumen / ZXPOS**
- **Ericsson ENM**

Alternatively, cell site data can be provided to the Omnitouch operations team periodically via email.

User Management and Access Control

Role-Based Access Control (RBAC)

The OWL system uses role-based access control (RBAC): people (Registered Users) are assigned one or more Roles, and each Role is a bundle of Permissions. Permissions are the smallest unit of access (e.g., create draft message alert). A Registered User's effective access is the union of Permissions from all assigned Roles.

RBAC Components:

- **Users** - Real people who sign-in to the OWL system
- **Permissions** - Micro capabilities (e.g., approve draft message, create message, view reports)
- **Roles** - Named sets of permissions (e.g., Message Approvers, Message Creators)
- **Assignment** - Users receive one or more Roles; permissions aggregate

RBAC Benefits:

1. **Data Protection** - Users only see and do what they're allowed to
2. **Operational Fit** - Roles mirror job functions (Admin, Message Creator, Message Approver)
3. **Simple Admin** - Grant access by assigning roles; avoid per-user micromanagement

System Permissions

System permissions generally follow CRUD patterns with four options:

- **View** - Read or browse messages and reports
- **Create** - Create or add a message alert
- **Update** - Edit or modify a draft message alert
- **Delete** - Delete or remove a draft message alert

Core CBC Permissions:

- `CREATE_CBC_MESSAGE` - Create new broadcast messages
- `VIEW_CBC_MESSAGE` - View existing messages and their status
- `UPDATE_CBC_MESSAGE` - Modify message content or broadcast parameters
- `DELETE_CBC_MESSAGE` - Delete messages and cancel broadcasts

Assign these permissions to roles based on your organization's public safety responsibilities.

For comprehensive information on user roles and permissions management, see `Users and Roles <rbac>`.

User Account Management

First-Time Login

When a new user is set up in the system, they are provided with login credentials by the System Administrator:

1. Navigate to the OWL login URL provided by your System Administrator
2. Enter your username and password
3. If 2FA is enabled for your account, you will be prompted to set it up (see 2FA section above)

Best Practice: Users should change their initial password immediately after first login for security. Navigate to your user profile settings to change your password.

For more details on authentication flows, see `Authentication Flows and Admin Controls <authentication_flows>`.

Password Reset

Users can reset their own passwords:

1. From the system login page, select "Forgot Password"
2. Follow the email verification process
3. Set a new password

Admin Password Reset:

System Administrators can reset passwords for users from the **Users and Roles → Users** page:

1. Navigate to **Users and Roles → Users**
2. Select the user requiring password reset
3. Click **Reset Password**
4. System generates a secure random temporary password (12 characters)
5. Temporary password is displayed to the admin
6. Provide the temporary password to the user via a secure channel (phone or in-person, never via email)
7. User can login with the temporary password
8. **Important:** User should change this password immediately after logging in

For more details, see `Users and Roles <rbac>`.

Adding New System Users

System Administrators can add new users by navigating to **Users and Roles → Users**:

Required information for each new Registered User:

- First Name
- Last Name
- Email
- Password (temporary)
- Mobile Number
- Role(s)

Best Practice: Implement a New System User Application Form that aligns with your organization's existing policies and procedures. Only add users after proper approval.

Editing User Profiles

System Administrators can change permissions and access for existing users:

1. Navigate to **Users and Roles → Users**
2. Select the user to edit
3. Modify assigned roles as needed
4. Save changes

Deleting System Users

System Administrators can delete users from **Users and Roles → Users**.

Warning: Once a user has been deleted from the OWL system, the action cannot be undone. Take great care before executing this task.

Viewing All System Users

At any time, the System Administrator can view a list of all Registered Users and see what permissions and roles each one has by navigating to **Users and Roles → Users**.

Alert Message History

The OWL system keeps a record of all alert messages sent for auditing purposes and to support operational review.

Accessing Message History:

Navigate to **Manage Cell Broadcasts** to view:

- All previously sent messages
- Message status (active, expired, cancelled)
- Timestamp of transmission
- Initiating and approving users
- Geographic targeting details
- Message content and parameters

This helps with:

- Auditing the system
- Reviewing operating procedures used by the NDMO and other alerting agencies
- Understanding message effectiveness
- Compliance and reporting requirements

Cell Broadcast Map Visualization

The Cell Broadcast Map provides a **visual interface** for planning geographic coverage areas and identifying which cell towers will broadcast alerts.

Accessing the Map

Navigate to:

Or directly:

Required Permission: `VIEW_CBC_MESSAGE` or `CREATE_CBC_MESSAGE`

Map Features

The map displays:

Cell Tower Locations:

- **Tower Icons** - Cell site positions displayed as cell tower icons
- **Sector Indicators** - Directional arrows showing antenna sector orientation
- **Tower Names** - Site identifiers and names
- **Coverage Data** - Loaded from `cellSites.txt` file

Drawing Tools:

- **Polygon Tool** - Draw custom coverage areas
- **Circle Tool** - Create circular broadcast zones
- **Rectangle Tool** - Define rectangular coverage areas

Coverage Analysis:

- **Site List** - Table showing all cell sites within the drawn area
- **Zone Management** - Save and load predefined coverage zones
- **Site Count** - Real-time count of towers in coverage area

Using the Map

Step 1: Load Cell Site Data

The map automatically loads cell tower locations from `/cellSites.txt` on page load. This file contains:

- **lat/lng** - Tower coordinates
- **site_name** - Identifier
- **sectors** - Antenna directions in degrees

Step 2: Draw Coverage Area

Select a drawing tool:

1. Click **Polygon** to draw custom shapes
   - Click to add points
   - Double-click to complete
   - Creates precise coverage boundaries
2. Click **Circle** to draw circular areas
   - Click center point
   - Drag to set radius
   - Useful for quick radius-based alerts
3. Click **Rectangle** to draw rectangular areas
   - Click one corner
   - Drag to opposite corner
   - Efficient for grid-aligned coverage

Step 3: Review Covered Sites

After drawing, the system:

- Calculates which cell sites fall within the area
- Displays list in table below the map
- Shows site names and coordinates
- Provides count of affected towers

Example Coverage Table:

Step 4: Export Coverage Data

Use the coverage data to:

- **Identify tracking area codes** for the broadcast
- **Estimate alert reach** based on tower count
- **Plan emergency response** by understanding affected areas
- **Save zone definitions** for future use

Workflow Integration

The map visualization integrates with broadcast creation:

Planning an Alert:

1. **Open CBC Map** to visualize affected area
2. **Draw coverage zone** using polygon tool
3. **Review covered sites** and adjust boundaries
4. **Note tracking area codes** for affected cell sites
5. **Create CBC message** with appropriate targeting

Example: Coastal Tsunami Warning

Zone Management

Saving Zones:

Predefined coverage zones can be saved for common scenarios:

- **Coastal Region** - For tsunami/storm surge warnings
- **Wildfire Zones** - For evacuation alerts
- **Urban Centers** - For AMBER alerts in populated areas
- **Highway Corridors** - For traffic/weather alerts

Zone Data Format:

Zones are stored in `site_data.json`:

```json
[
  {
    "area": "Coastal Region",
    "polygon": [
      {
        "coord": [
          {"lat": -33.8688, "lng": 151.2093},
          {"lat": -33.8650, "lng": 151.2070},
          {"lat": -33.8600, "lng": 151.2150},
          {"lat": -33.8688, "lng": 151.2093}
        ]
      }
    ]
  }
]
```

Loading Saved Zones:

1. Select zone from dropdown menu
2. Polygon displays on map automatically
3. Site list updates to show covered towers
4. Modify polygon if needed

Map Configuration

Google Maps API:

The map requires Google Maps API key configured in environment:

```
REACT_APP_GOOGLE_API_KEY=your_google_maps_api_key_here
```

Cell Site Data:

Update `public/cellSites.txt` with your network's cell tower locations:

```
# Format: lat,lng,name,sector1,sector2,sector3
-33.8688,151.2093,SYD_CENTRAL,0,120,240
-33.8650,151.2070,SYD_HARBOUR,45,165,285
```

Obtain cell site coordinates from:

- Network planning tools
- Tower installation records
- Field surveys with GPS
- Radio network controller (RNC) configuration

Map Styling:

Custom map styles can be configured in `mapStyles.js` to:

- Highlight emergency services
- Show topographic features
- Emphasize population centers
- Match organizational branding

Use Cases

Emergency Planning:

- Pre-plan coverage zones for known hazard areas
- Test different targeting strategies
- Estimate population reach
- Coordinate with emergency services

Alert Verification:

- Confirm message will reach intended area
- Identify coverage gaps
- Avoid over-alerting adjacent regions
- Validate tracking area codes

Network Analysis:

- Visualize cell tower distribution
- Identify coverage overlaps
- Plan tower deployments for better alert coverage
- Optimize broadcast efficiency

Best Practices

Based on experience with customers around the world, Omnitouch recommends the following best practices for all OWL deployments.

Message Content:

- Keep messages concise and clear (under 360 characters for single-page alerts)
- Use all capital letters for emergency alerts (improves readability)
- Include specific action items ("Evacuate immediately", "Seek shelter")
- Avoid technical jargon
- Test messages with actual devices before emergencies

Language Support:

- Always provide messages in the primary language of the region
- Include additional languages for multicultural areas
- Ensure translations are culturally appropriate
- Test special characters and Unicode support

Geographic Targeting:

- Use smallest necessary tracking areas to avoid alert fatigue
- Consider population density when setting repetition period
- Test geographic targeting before emergencies
- Maintain accurate tracking area documentation

Testing:

- Use test message identifiers (4379, 4380-4381) for drills
- Schedule regular system tests
- Verify CBC integration is functioning
- Train staff on emergency procedures

Alert Fatigue:

- Only use for genuine emergencies
- Avoid over-broadcasting
- Set appropriate repetition periods
- Use severity levels appropriately

User Account Security

2FA Token Security:

- Protect physical 2FA tokens like building access cards
- Report lost or stolen tokens immediately
- Test 2FA regularly to ensure it's functioning
- Save backup codes when setting up 2FA and store them securely offline
- Contact System Administrator if you need 2FA reset

For more information on 2FA setup and recovery, see Two-Factor Authentication <2fa>.

Data Maintenance

Predefined Target Areas:

As geographic boundaries change, development occurs, and risk areas shift, there is a need to review the Predefined Target Areas. Omnitouch suggests this data is reviewed **annually by the NDMO**, with support from other alerting agencies where applicable.

Updates to boundaries can be defined using several common GIS platforms or Google Earth, then provided via email to the Omnitouch Operations team who will make the changes to the system.

Predefined Message Templates:

The predefined message templates should be reviewed **at least annually by the NDMO**, with support from other alerting agencies where applicable, to ensure:

- Message content still accurately reflects the hazard and call to action
- Contact information and instructions are current
- Language translations remain accurate
- Message tone and urgency are appropriate
- References to emergency services or procedures are up to date

Updates can be provided via email to the Omnitouch Operations team who will make the changes to the system.

Cell Site Data:

- Review and update cell site data whenever network changes occur
- Typical update frequency: monthly or quarterly
- Coordinate with MNO network planning teams
- Verify accuracy after major network upgrades or expansions

System Architecture

All Omnitouch products are designed to support geographically distributed deployments.

Deployment Options

All components can run as:

- **Containers (K8s)** - Kubernetes orchestrated containerized deployments
- **Virtual Machines** - VMware, Proxmox, HyperV
- **Private Cloud** - On-premises cloud infrastructure
- **Public Cloud** - AWS, GCP
- **Bare Metal** - Direct hardware deployment

Distributed Architecture

The distributed architecture allows:

- Local Disaster Management Offices to access OWL CBE and distribute messages even if a region becomes isolated from the national network
- Multiple Cell Broadcast Entities and multiple Cell Broadcast Centers per operator/per country
- Local disaster response agencies (Municipal Government, Police, Fire, etc.) to issue alerts to their region even if the main NDMO (National Disaster Management Office) loses access

This is particularly valuable when MNOs have distributed their cellular network with local BSC/MME resources.

CBE and CBC Networking

To ensure there is no connection between competing networks, each MNO has a separate CBC instance (not shared).

Networking requirements between CBE and CBC:

- All traffic is encrypted between CBE & CBC
- Authentication based on mutual certificates
- Connectivity from CBE to CBC over TLS on TCP port 443
- Coordination required between MNOs and the NDMO or agency hosting the CBE

Access Considerations

The agency hosting the CBE will need to define access procedures for end users (ie. Citrix, VPN, etc.), keeping in mind that the system must be accessible in non-ideal scenarios such as:

- Large scale outages of public power networks
- Telecommunications network failures
- Natural disasters affecting infrastructure

Deployment Requirements

CBE VM Requirements (NDMO / Government)

3x Virtual Machines:

- **2x CBE VMs** (Ideally in different datacenters/availability zones)
- **1x Monitoring VM**

Each VM requires:

- **Storage:** 50GB
- **CPU:** 2x Virtual CPU
- **RAM:** 8GB
- **OS:** Base OS provided by Omnitouch
- **Networking:** Allow traffic to CBC VMs on TCP port 443 for TLS traffic to control the CBCs

CBC VM Requirements (MNO)

3x Virtual Machines:

- **2x CBC VMs** (Ideally in different datacenters/availability zones)
- **1x Monitoring VM**

Each VM requires:

- **Storage:** 50GB
- **CPU:** 2x Virtual CPU
- **RAM:** 8GB
- **OS:** Base OS provided by Omnitouch
- **Connectivity:** To each of the Integration Points in the cellular network (MME, RNC, BSC)
- **Networking:** Allow traffic from CBE VMs on TCP port 443 for TLS traffic to be controlled by the CBEs

Integration Steps

Deploying OWL involves the following steps:

1. **Source new operator/s** - Identify participating mobile network operators
2. **Project admin (set-up)** - Establish project governance and administration
3. **Select hosting location for CBE** - Determine where CBE will be hosted

4. **Define Users & Message Flows/Procedures** - Establish approval workflows and user roles
5. **Define Polygons for Target Areas and Message Templates** - Pre-configure common scenarios
6. **IP Address allocations** - Allocate IP addresses for NDMO and MNOs
7. **Set up site-to-site VPN** - Establish secure connection to Omnitouch team
8. **Deploy CBE VMs to NDMO** - Install Cell Broadcast Entity
9. **Deploy CBC VMs to MNO** - Install Cell Broadcast Center at each operator
10. **Configure Network Elements** - Set up connectivity to CBC from cellular network equipment
11. **Networking between CBE and CBC VMs** - Establish secure communication
12. **Networking between CBC VMs and Network Elements** - Connect to BSC/MME/AMF
13. **Setup API access to NMS** - Configure cell site data integration
14. **Monitoring setup & tested** - Verify monitoring and alerting
15. **Verification/Testing of test alerts** - Conduct system testing
16. **Public warning test** - Perform end-to-end public test

Character Limits

Cell Broadcast messages have strict character limits based on encoding:

GSM 7-bit Encoding (English, basic Latin characters):

- Single page: 93 characters
- Multi-page: 15 pages × 93 = 1395 characters maximum

Unicode UCS-2 Encoding (non-Latin scripts, emojis):

- Single page: 41 characters
- Multi-page: 15 pages × 41 = 615 characters maximum

OWL Platform:

- Message text limited to 500 characters
- Web UI displays remaining character count and warns when approaching limits

Monitoring and Logs

Cell Broadcast activity is logged for audit purposes:

- Message creation, update, and deletion events
- Initiating and approving users
- Timestamps and message identifiers
- CBC API responses and errors

- Geographic targeting details

Access logs via the Activity Log or database queries:

```sql
SELECT * FROM cbc
WHERE created >= '2025-01-01'
ORDER BY created DESC;
```

Integration with Mobile Devices

Cell Broadcast messages are received by compatible mobile devices:

Device Support:

- Most smartphones from 2015 onwards support Cell Broadcast
- Feature phones may have limited support
- Device must be connected to the network (no data/SMS credits required)
- Works even during network congestion when SMS fails

User Experience:

- Alert displays as full-screen notification
- Unique alert tone plays
- Alert persists until acknowledged
- No user subscription required
- Cannot be blocked by users for presidential/extreme alerts

Testing Device Reception:

To verify devices can receive alerts:

1. Send test message (identifier 4379 or 4380-4381)
2. Ensure device is in the target tracking area
3. Check device has Cell Broadcast enabled in settings
4. Verify with multiple device models and OS versions

Additional Functionality

The OWL platform can be extended with optional features to complement Cell Broadcast messaging:

Mass Text / SMS

Sending regular SMS to individuals for supplementary notifications.

- Often used for downgraded alerts to inform people the immediate threat has passed, in a less-obtrusive way

- Much slower than Cell Broadcast but can include a confirmation mechanism to verify the message was received
- Useful for targeted follow-up communications

Voice Calling for Fixed Line

Automatically call fixed line numbers and play emergency warning messages.

- Pre-recorded message playback
- Text-to-speech rendition of emergency warning message
- Reaches populations without mobile phones
- Can verify message delivery through call completion

Cross-post to Social Media

Automatically post Emergency Warning messages to official social media channels.

- Extends reach beyond cellular network
- Provides reference for those who missed initial alert
- Allows for extended messaging beyond character limits

Automatic Radio/TV Transmission

Automatically broadcast emergency messages via radio and television.

- Pre-recorded message playback
- Text-to-speech rendition of emergency warning message
- Reaches populations during network outages
- Complements Cell Broadcast for comprehensive coverage

External Warning Devices

The OWL CBC can connect to a variety of external sources:

- Social media platforms
- Public APIs
- Voice call systems
- Radio broadcast systems
- Physical alarms and sirens
- Electronic signage

Customization options can be explored as part of the design phase.

Customization and Maintenance

Periodic Testing

Periodic testing of the solution should be performed at regular intervals to ensure:

- The solution and all components are functioning correctly
- All staff are familiar with the processes and procedures required for issuing Emergency Warning Messages
- Integration points remain operational
- Message templates are current and effective

Recommended Testing:

- Monthly test messages using identifiers 4379, 4380-4381
- Quarterly full system tests including approval workflows
- Annual public warning tests with advance notification
- Regular training sessions for authorized users

Cell Site Data Maintenance

When operators add or remove cell sites, or change Tracking Areas/Identifiers of cell sites, this information must be shared with the Omnitouch team to ensure the mapping tool data remains accurate.

Automatic Data Integration

OWL supports automatic data scraping from:

- Nokia NetAct
- Huawei U2000 / U2020
- ZTE NetNumen / ZXPOS
- Ericsson ENM

Manual Data Updates

Alternatively, cell site data can be provided to the Omnitouch operations team periodically via email in various formats.

Update Frequency: Review and update cell site data whenever network changes occur, typically monthly or quarterly.

Predefined Target Areas

As geographic boundaries change, development occurs, and risk areas shift, there is a need to review the Predefined Target Areas used in the Targeting stage.

Annual Review: Predefined target areas should be reviewed annually by the NDMO (National Disaster Management Office).

Update Process: Updates to boundaries can be defined using several common GIS platforms or Google Earth, then provided to the Omnitouch operations team.

Predefined Message Templates

Annual Review: Predefined message templates should be reviewed annually by the NDMO to ensure:

- Message content reflects current emergency procedures
- Language translations are accurate
- Message identifiers are appropriate
- Contact information and instructions are current

Update Process: Updates can be provided via email to the Omnitouch operations team.

Message Approval Flows

Different regions have different requirements regarding issuing of messages and approval flows.

Two-Person Rule: Use of the two-person rule is advocated wherever practical to ensure oversight in message submission.

Granular User Roles: Individual user roles can be configured to:

- Allow only certain users to send predefined messages
- Restrict targeting to specific regions
- Require additional approval steps
- Minimize risk of misuse

2FA / Security Maintenance

The Omnitouch operations team can support with:

- Resetting 2FA tokens
- Re-issuing lost/damaged/expired tokens
- Security audit and token management

Token Security Procedures:

When tokens are issued, a procedure is detailed for steps to take should a token become lost or unaccounted for. The person accepting the token must follow these procedures to ensure the system is not abused.

Immediate Actions for Lost Tokens:

1. Report lost token immediately to authorized personnel
2. Token is deactivated in the system
3. Security review conducted
4. New token issued following security verification

External API Integration

A full suite of APIs is available for the Cell Broadcast Entity to allow 3rd party systems to integrate and interact with the CBE.

API Capabilities:

- **Reporting/Monitoring** - Verify status and reach of transmitted messages
- **Message Creation** - Create and broadcast new messages programmatically
- **Health Checks** - Periodically check system health with routine test message traffic generation
- **Status Queries** - Retrieve message status, delivery statistics, and system metrics

See the API Reference section below for detailed endpoint documentation.

API Reference

All CBC endpoints require authentication and appropriate permissions.

Create Message:

```
PUT /crm/cbc/
```

Get All Messages:

```
GET /crm/cbc/
```

Update Message:

```
PATCH /crm/cbc/{cbc_message_id}
```

Delete Message:

```
DELETE /crm/cbc/{cbc_message_id}
```

See the Swagger documentation at `/crm/docs/` for detailed API specifications.

Global Search

The Global Search feature provides a **unified search interface** to quickly find customers, contacts, services, inventory, and sites across the entire OmniCRM database.

See also: `Customers <basics_customers>`, `Inventory <administration_inventory>`, `Service Management <csa_service_management>`.

Access Global Search

From anywhere in the CRM:

Click the search icon in the top navigation bar or navigate to:

The global search page appears with a large search box and filter options.

How It Works

Global search performs a **cross-entity search** across five data types:

What Gets Searched:

1. **Customers** - Customer name
2. **Contacts** - First name, last name, email address, phone number
3. **Sites** - Site name
4. **Inventory** - Serial numbers, ICCIDs, identifiers (itemtext1, itemtext2)
5. **Services** - Service name, service UUID

Search Behavior:

- **Partial matching** - Searches for terms containing your query (e.g., "Smith" matches "John Smith" and "Smithson")
- **Case-insensitive** - "john" matches "John", "JOHN", and "john"
- **Multiple entities** - Single search returns results from all entity types
- **Paginated results** - Shows 10 results per page by default

Performing a Search

Basic Search

1. Enter your search term in the search box
2. Click **"Search"** or press Enter

Global Search Results

Example search terms:

- Customer name: `"Acme Corp"`
- Phone number: `"+1234567890"` or `"1234567890"`
- Email: `"john@example.com"` or `"john"`
- Serial number: `"ICCID8944"` or just `"8944"`
- Service UUID: `"123e4567-e89b"`

Include Closed Accounts

By default, search only returns results from **Open** customer accounts.

To search across all accounts including closed ones:

1. Check the **"Include Closed Accounts"** checkbox
2. Click **"Search"** again

This will search:

- Customers with `customer_status = "Closed"`
- Contacts, services, sites, and inventory linked to closed customers

Use cases for closed account search:

- Finding historical customer records
- Locating equipment from deprovisioned services
- Looking up old phone numbers or services
- Recovering customer data for re-activation

Understanding Search Results

Result Display Format

Results are displayed in a scrollable list showing:

> John Smith Customer ID: 123 Type: customer
>
> John Smith (Contact) Customer ID: 123 Type: contact
>
> Mobile - +44 7700 900123 Customer ID: 123 Type: service

Each result shows:

- **Name/Title** - The primary identifier (clickable link)
- **Customer ID** - The parent customer this belongs to
- **Type** - The entity type (customer, contact, site, inventory, service)

Result Types Explained

Customer Results:

Clicking opens the customer overview page showing all details, services, contacts, etc.

Contact Results:

Clicking opens the customer page with the Contacts tab active, scrolling to the specific contact.

Site Results:

Clicking opens the customer page with the Sites tab active.

Inventory Results:

Clicking opens the customer page with the Inventory tab active. If inventory is unassigned (no customer_id), it links to the main inventory list instead.

Service Results:

Clicking opens the customer page with the Services tab active, highlighting the specific service.

Navigation from Results

All search results are **clickable links** that navigate directly to the relevant page:

Link Pattern:

- `/customers/{customer_id}` - Customer records
- `/customers/{customer_id}#4` - Contacts (tab 4)
- `/customers/{customer_id}#2` - Sites (tab 2)
- `/customers/{customer_id}#8` - Inventory (tab 8)
- `/customers/{customer_id}#3` - Services (tab 3)
- `/inventory-items-list` - Unassigned inventory

The hash (#) fragment automatically selects the correct tab when the customer page loads.

Pagination

Results are paginated with 10 items per page:

> Showing results 11-20 of 47

Navigate through pages using:

- **Previous/Next** buttons
- **Page numbers** - Click specific page
- **Keyboard** - Left/right arrows (if implemented)

Common Search Scenarios

Scenario 1: Find Customer by Phone

User calls in, provides phone number.

> Results: • John Smith (Contact) - Customer ID: 123 • Mobile - 555-0123 (Service) - Customer ID: 123

Click either result to access customer account.

Scenario 2: Locate SIM Card

Technician needs to find which customer has a specific SIM.

> Results: • 8944538000000001234 (Inventory) - Customer ID: 456

Click result to see SIM assignment, customer details.

Scenario 3: Find Inactive Customer

Need to locate a customer who closed their account 6 months ago.

> ☑ Include Closed Accounts

> Results: • Acme Corporation (customer) - Customer ID: 789

Scenario 4: Search by Email

Customer emails support, staff needs to find their account.

> Results: • John Smith (Contact) - Customer ID: 123

Scenario 5: Find Service by UUID

Provisioning log shows service UUID, need to find which customer.

> Results: • Mobile - +44 7700 900123 (Service) - Customer ID: 456

Search Tips

For Best Results:

- **Use partial terms** - "Smith" is better than "John Smith" for broader results
- **Try variations** - If "John" doesn't work, try phone or email
- **Include closed accounts** - When searching historical data
- **Be specific for equipment** - Use full serial numbers for inventory
- **Search service UUID** - When other identifiers aren't known

What Gets Searched (by Entity):

Customers:

- Customer name only (not address, notes, or other fields)

Contacts:

- First name
- Last name
- Email address
- Phone number

Sites:

- Site name only

Inventory:

- itemtext1 (typically ICCID, serial number, MAC address)
- itemtext2 (typically IMSI, secondary identifier)
- *Note: Does not search itemtext3-20 or inventory notes*

Services:

- Service name
- Service UUID

What Doesn't Get Searched:

- Customer addresses
- Customer notes
- Transaction descriptions
- Invoice details
- Provisioning logs
- Activity log entries

- Inventory notes (beyond itemtext1/2)

API Reference

Global Search Endpoint

```
GET /utilities/
search_everything?search=Smith&page=1&per_page=10&search_closed_records=false
Authorization: Bearer <token>
```

Query Parameters:

- `search` (required) - The search term
- `page` (optional) - Page number (default: 1)
- `per_page` (optional) - Results per page (default: 10)
- `search_closed_records` (optional) - Include closed accounts (default: false)

Response:


```json
{
  "data": [
    {
      "id": 123,
      "name": "John Smith",
      "customer_id": 123,
      "type": "customer"
    },
    {
      "id": 456,
      "name": "John Smith",
      "customer_id": 123,
      "type": "contact"
    },
    {
      "id": 789,
      "name": "Mobile - +44 7700 900123",
      "customer_id": 123,
      "type": "service"
    }
  ],
  "pagination": {
    "current_page": 1,
    "per_page": 10,
    "total_pages": 5,
    "total_items": 47
  }
}
```


Search Logic (Backend):

The backend performs a SQL UNION across all entity tables:

```sql
-- Customers
SELECT customer_id AS id,
       customer_name AS name,
       customer_id,
       'customer' AS type
```

```sql
FROM customer
WHERE customer_name LIKE '%Smith%'
  AND customer_status = 'Open'

UNION ALL

-- Contacts
SELECT contact_id AS id,
       CONCAT(contact_firstname, ' ', contact_lastname) AS name,
       customer_id,
       'contact' AS type
FROM customer_contact
WHERE (contact_firstname LIKE '%Smith%' OR
       contact_lastname LIKE '%Smith%' OR
       contact_email LIKE '%Smith%' OR
       contact_phone LIKE '%Smith%')

UNION ALL

-- Sites
SELECT site_id AS id,
       site_name AS name,
       customer_id,
       'site' AS type
FROM customer_site
WHERE site_name LIKE '%Smith%'

UNION ALL

-- Inventory
SELECT inventory_id AS id,
       itemtext1 AS name,
       customer_id,
       'inventory' AS type
FROM inventory
WHERE itemtext1 LIKE '%Smith%' OR
      itemtext2 LIKE '%Smith%'

UNION ALL

-- Services
SELECT service_id AS id,
       service_name AS name,
       customer_id,
       'service' AS type
FROM customer_service
WHERE service_name LIKE '%Smith%' OR
      service_uuid LIKE '%Smith%'
```

Results are then paginated and returned.

Performance Considerations

Search Performance:

- Searches use LIKE queries with wildcards (`%term%`)
- No full-text indexing currently implemented

- Large databases (>100k customers) may experience slower searches
- Results limited to 10 per page for performance

Optimization Tips:

- Be specific with search terms to reduce result set
- Use closed account filter to reduce search scope
- Consider adding database indexes on frequently searched fields

Troubleshooting

No results found (but record exists)

- **Cause:** Search term doesn't match stored data format
- **Examples:**
  - Phone stored as "+44 7700 900123", searching "07700900123" won't match
  - Email stored as "<john.smith@example.com>", searching "john" won't match
- **Fix:** Try variations, use partial matches that definitely exist

Search too slow

- **Cause:** Large database, complex query across multiple tables
- **Fix:**
  - Use more specific search terms
  - Limit to open accounts only (uncheck closed accounts)
  - Contact administrator about database indexing

Results link to wrong customer

- **Cause:** Multiple customers/contacts with same name
- **Fix:** Use Customer ID to differentiate, or search by unique identifier (email, phone)

Closed accounts not appearing

- **Cause:** "Include Closed Accounts" checkbox not checked
- **Fix:** Check the box and search again

Related Documentation

- `basics_customers` - Customer management
- `basics_navigation` - General navigation
- `administration_inventory` - Inventory searches

Top-Up and Recharge System

The OmniCRM Top-Up system provides a **self-service prepaid recharge portal** for customers to add credit or extend service validity via the Self-Care Portal <self_care_portal>. This feature is commonly used for:

- **Mobile data services** - Prepaid SIM cards and data-only services
- **Hotspot services** - WiFi hotspot dongles and portable internet devices
- **Fixed wireless services** - Prepaid internet access

Overview

The top-up system allows customers to purchase additional days of service through a streamlined, multi-step checkout process with integrated Stripe payment processing.

Key Features:

- Self-service customer portal (no staff intervention required)
- Flexible duration selection (1-30 days)
- Real-time usage display before purchase
- Stripe-powered secure payment processing
- Automatic refunds if top-up fails
- Invoice and transaction generation
- Provisioning system integration for service activation

Access the Top-Up Portal

The top-up portal is accessed via a **public URL** that customers can visit without logging into the CRM:

How Customers Access It:

- Direct link sent via SMS when balance is low
- QR code on printed materials
- Link on self-care portal
- Shared via customer support

The portal automatically detects the customer's service based on their requesting IP address or IMSI.

Top-Up Process

The top-up flow consists of **4 steps**:

Step 1: Data Selection

Customers select how many days of service they want to purchase.

Interface:

- **Slider control** - Select 1 to 30 days
- **Live price calculation** - Shows total cost based on selection
- **Expiry date display** - Calculates and shows when service will expire
- **Current usage display** - Shows remaining balance/expiry before top-up

Example Display:

Pricing Configuration:

- Price per day is configured via environment variable `REACT_APP_TOPUP_PRICE_PER_DAY`
- Default: $10 USD per day
- Currency is set via `REACT_APP_CURRENCY_CODE`

Step 2: Billing Information

Customers provide their contact details for the transaction:

- **First Name**
- **Last Name**
- **Email Address**

This information is used for:

- Invoice generation
- Payment receipt email
- Transaction records
- Refund processing (if needed)

Step 3: Payment

Secure payment processing via **Stripe Elements**.

Payment Methods Supported:

- Credit cards (Visa, Mastercard, Amex)
- Debit cards
- Digital wallets (Apple Pay, Google Pay) *if enabled in Stripe*

Security Features:

- PCI-compliant Stripe integration
- No card details stored in OmniCRM
- 3D Secure authentication support
- Encrypted payment transmission

Payment Flow:

1. Stripe Elements form displayed with card input
2. Customer enters payment details
3. Payment Intent created for the exact amount
4. Card charged immediately
5. Payment success/failure handled

Note

If the payment succeeds but the top-up provisioning fails (e.g., network error, OCS unreachable), the system automatically initiates a **full refund** to the customer's payment method.

Step 4: Completion

Success Screen:

> Your service has been extended. New expiry date: 17 Jan 2025
>
> Receipt sent to: <customer@example.com> Transaction ID: TXN-123456

Failure Screen:

If top-up fails, the system displays an error and automatically processes a refund:

> We were unable to complete your top-up. Your payment has been refunded.
>
> Error: Unable to connect to billing system
>
> Please try again or contact support.

Backend Processing

When a customer completes payment, the following happens automatically:

1. Payment Validation

The system validates:

- Payment Intent status is `succeeded`
- Payment amount matches selected days (`days × price_per_day`)
- Payment Intent hasn't been processed before (prevents double top-up)

2. Top-Up Operation

```
- API endpoint: POST /oam/topup_dongle
- Validates service_uuid and IMSI
- Calls OCS/CGRateS to add balance
- Creates provisioning job (play_topup_hotspot)
```

3. Record Creation

The system creates multiple database records:

- **HotspotTopup record** - Tracks the top-up transaction
    - payment_intent_id
    - service_uuid
    - imsi
    - days purchased
    - topup_amount
    - status (Success/Failed/Refunded)
- **Transaction record** - Financial transaction
    - Title: "Hotspot Topup - 7 Days"
    - Amount: topup_amount (positive)
    - Linked to service_id and customer_id
- **Invoice record** - Payment invoice
    - Contains the top-up transaction
    - Marked as paid immediately
    - Payment reference: Stripe payment_intent_id
- **Payment transaction** - Offsetting credit transaction
    - Title: "Payment for [Invoice Title]"
    - Amount: topup_amount (negative - credit)
    - Links invoice payment to customer account

4. Provisioning Job

A provisioning job is created with playbook `play_topup_hotspot` which:

- Connects to OCS/CGRateS API
- Adds balance to the account
- Extends expiry date
- Creates activity log entry
- Sends confirmation notification (if configured)

The API waits for provisioning to complete (polling with 0.2s intervals, max 25 iterations) before returning success to the customer.

5. Automatic Refund on Failure

If any step fails after payment:

```python
if topup_provisioning_failed:
    refund = stripe.Refund.create(
        payment_intent=payment_intent_id,
        reason='requested_by_customer'  # Automatic system refund
    )
    status_message = "Topup Failed. Refunding payment..."
```

The refund is processed automatically and the customer is notified on-screen.

API Endpoints

Top-Up Endpoint

```
POST /oam/topup_dongle
Content-Type: application/json

{
  "service_uuid": "123e4567-e89b-12d3-a456-426614174000",
  "imsi": "310120123456789",
  "days": 7,
  "payment_intent_id": "pi_1234567890abcdef",
  "topup_amount": 70.00
}
```

Response (Success):

```json
{
  "result": "OK",
  "status": 200,
  "provision_id": 456,
  "payment_intent_id": "pi_1234567890abcdef",
  "service_uuid": "123e4567-e89b-12d3-a456-426614174000",
  "invoice_id": 789
}
```

Response (Failure):

```json
{
  "result": "Failed",
  "Reason": "OCS connection timeout",
  "status": 500
}
```

Validation Checks:

- All required fields present (service_uuid, imsi, days, payment_intent_id,

topup_amount)
- topup_amount matches days: `topup_amount × 100 == days × 1000` (in cents)
- Payment Intent exists in Stripe
- Payment Intent amount matches: `payment_intent.amount == topup_amount × 100`
- Payment Intent status is `succeeded`
- Payment Intent not already processed (checks `HotspotTopup` table)

Usage Endpoint

Retrieves current usage and service information for the customer:

```
GET /oam/usage
```

Response:

```json
{
  "imsi": "310120123456789",
  "service": {
    "service_uuid": "123e4567-e89b-12d3-a456-426614174000",
    "service_name": "Mobile Data - 0412345678",
    "service_status": "Active"
  },
  "balance": {
    "expiry": "2025-01-10T23:59:59Z",
    "unlimited": true
  },
  "requestingIp": "203.0.113.45"
}
```

This endpoint uses the requesting IP address to identify the customer's service automatically.

Configuration

Environment Variables

Configure these in the OmniCRM-UI `.env` file:

```
# Top-Up Portal Configuration
REACT_APP_TOPUP_PRICE_PER_DAY=10
REACT_APP_CURRENCY_CODE=AUD
REACT_APP_SELF_CARE_NAME="YourCompany"

# Stripe Configuration
REACT_APP_STRIPE_PUBLISHABLE_KEY=pk_live_...
```

Stripe Configuration

The top-up system uses Stripe Payment Intents:

1. **Enable Payment Intents** in your Stripe Dashboard
2. **Configure Webhook** to receive payment status updates (optional but recommended)
3. **Set up payment methods** (cards, wallets, etc.)
4. **Test mode** - Use test keys for development

```
# Development
REACT_APP_STRIPE_PUBLISHABLE_KEY=pk_test_...

# Production
REACT_APP_STRIPE_PUBLISHABLE_KEY=pk_live_...
```

Playbook Configuration

The provisioning playbook `play_topup_hotspot.yaml` must be configured to:

- Accept `days` variable
- Connect to OCS/CGRateS API
- Add balance to account
- Update service expiry date

Example playbook structure:

```yaml
- name: Top up hotspot service
  hosts: localhost
  tasks:
    - name: Add balance to OCS
      uri:
        url: "{{ ocs_api_url }}/add_balance"
        method: POST
        body:
          imsi: "{{ imsi }}"
          days: "{{ days }}"
          service_uuid: "{{ service_uuid }}"
```

Low Balance Notifications

The system can send automatic notifications when customer balance is low:

SMS Notifications:

When triggered by OCS events (`Action_Balance_Low`, `Action_Balance_Out`, `Action_Balance_Expired`):

Email Notifications:

Configured in the OCS/CGRateS action plans to send balance alerts.

Notification Triggers:

- `Action_Balance_Low` - Balance below threshold (e.g., 2 days remaining)
- `Action_Balance_Out` - Balance exhausted
- `Action_Balance_Expired` - Service expired

Each notification includes the top-up portal link for easy customer access.

Troubleshooting

Common Issues

"Payment system unavailable"

- **Cause:** Stripe library failed to load or invalid publishable key
- **Fix:**
  - Check `REACT_APP_STRIPE_PUBLISHABLE_KEY` is set correctly
  - Verify Stripe account is active
  - Check browser console for JavaScript errors

"Top-up failed. Refunding payment..."

- **Cause:** Provisioning job failed (OCS unreachable, playbook error, etc.)
- **Fix:**
  - Check provisioning logs: `GET /crm/provision/provision_id/<id>`
  - Verify OCS/CGRateS API is accessible
  - Review playbook `play_topup_hotspot.yaml` for errors
  - Check Ansible logs

"Payment intent already processed"

- **Cause:** Customer attempting to reuse same payment (e.g., refresh after success)
- **Fix:** This is working as designed to prevent double billing. Customer should start a new top-up if needed.

"Payment intent amount does not match"

- **Cause:** Mismatch between UI calculation and backend validation
- **Fix:**
  - Verify `REACT_APP_TOPUP_PRICE_PER_DAY` matches backend expectation (default $10)
  - Check currency configuration is consistent
  - Clear browser cache and retry

Monitoring Top-Ups

View Top-Up Records:

Query the `HotspotTopup` table to see all top-up attempts:

```sql
SELECT
  hotspot_topup_id,
  service_uuid,
  days,
  topup_amount,
  status,
  payment_intent_id,
  created
FROM hotspot_topup
WHERE status = 'Failed'
ORDER BY created DESC;
```

Check Provisioning Status:

```
GET /crm/provision/provision_id/<provision_id>
```

Shows the detailed status of the top-up provisioning job.

Stripe Dashboard:

Monitor payments, refunds, and failed transactions in your Stripe Dashboard at <https://dashboard.stripe.com>

Security Considerations

Payment Security:

- All card data handled by Stripe (PCI Level 1 compliant)
- No sensitive payment data stored in OmniCRM database
- Payment Intents prevent unauthorized charges
- Amount validation on both client and server side

Fraud Prevention:

- Duplicate payment Intent detection prevents double billing
- IP address tracking for usage correlation
- IMSI validation ensures top-up goes to correct service
- Automatic refunds limit financial exposure

Access Control:

- Top-up portal is public (by design - customers need access)

- Usage endpoint requires valid service identification (IP or IMSI)
- Backend validation prevents arbitrary service top-ups
- Admin can view all top-up records via CRM interface

Best Practices

For Operators:

1. **Test refund flow** - Regularly test failure scenarios to ensure refunds work
2. **Monitor failed top-ups** - Set up alerts for high failure rates
3. **Keep playbooks simple** - Top-up playbooks should be fast and reliable
4. **Verify OCS connectivity** - Ensure OCS API is always accessible
5. **Review pricing** - Update `REACT_APP_TOPUP_PRICE_PER_DAY` as needed

For Customers:

1. **Bookmark the top-up URL** - Quick access when needed
2. **Save low balance notifications** - SMS contains direct link
3. **Keep email updated** - Receipts sent to email on file
4. **Check expiry before travel** - Top up before leaving coverage area

For Developers:

1. **Handle Stripe webhooks** - Implement webhook handlers for payment status updates
2. **Implement idempotency** - Always check payment_intent_id before processing
3. **Log extensively** - Top-up failures need detailed troubleshooting info
4. **Test error paths** - Verify refund automation works correctly
5. **Monitor performance** - Provisioning polling should complete in <5 seconds

Related Documentation

- `payments_process` - General payment processing
- `concepts_provisioning` - Provisioning system overview
- Payment Vendor Integrations <integrations_payment_vendors> - Payment vendor integration details
- `payments_transaction` - Transaction management
- `payments_invoices` - Invoice handling

Complete Product Lifecycle Guide

This guide provides an end-to-end walkthrough of the product lifecycle in OmniCRM, from creating a product definition through provisioning services, adding addons, and deprovisioning. We'll cover pricing strategy, Ansible integration, and provide real-world examples throughout.

Overview: The Product-to-Service Journey

The lifecycle of a product in OmniCRM follows these stages:

1. **Product Definition** - Administrator creates product template with pricing and provisioning rules
2. **Service Creation** - Customer orders product, system provisions service instance
3. **Service Lifecycle** - Customer uses service, adds addons/topups, modifies service
4. **Deprovisioning** - Service is terminated, resources are released

Understanding Pricing: Wholesale vs Retail

Every product and service in OmniCRM has two pricing dimensions: **wholesale** and **retail**.

Wholesale Cost

The wholesale cost represents the actual cost to deliver the service:

- Infrastructure and bandwidth costs
- Licensing fees
- Equipment costs
- Operational expenses

Retail Cost

The retail cost is the amount charged to the customer.

Setup Costs

Both wholesale and retail have setup cost variants for one-time provisioning charges:

- `wholesale_setup_cost` - Your cost to provision

- `retail_setup_cost` - Amount charged to customer for activation

Example:

```
{
  "retail_cost": 15.00,
  "wholesale_cost": 5.00,
  "retail_setup_cost": 0.00,
  "wholesale_setup_cost": 1.00
}
```

Stage 1: Creating a Product Definition

Products are templates that define what gets provisioned and how customers are charged.

Creating a Mobile SIM Product

Let's create a prepaid mobile SIM product with 20GB data per month.

Step 1: Navigate to Product Management

From the admin UI, go to **Products → Create Product**.

Step 2: Define Basic Information

```
{
  "product_name": "Prepaid Mobile 20GB",
  "product_slug": "prepaid-mobile-20gb",
  "category": "standalone",
  "service_type": "mobile",
  "enabled": true,
  "icon": "fa-solid fa-sim-card",
  "comment": "Prepaid mobile SIM with 20GB data, unlimited calls & texts"
}
```

Field Explanations:

- `product_name` - Customer-facing name shown in catalog
- `product_slug` - URL-safe identifier used in API calls and links
- `category` - "standalone" means this creates a new service (vs addon/bundle)
- `service_type` - Groups related products, used for addon filtering
- `enabled` - Must be true for product to be orderable
- `icon` - FontAwesome icon displayed in UI
- `comment` - Internal notes for staff reference

Step 3: Set Pricing

```json
{
  "retail_cost": 15.00,
  "wholesale_cost": 5.00,
  "retail_setup_cost": 0.00,
  "wholesale_setup_cost": 1.00,
  "contract_days": 30
}
```

Pricing Breakdown:

- Monthly revenue per customer: £15.00
- Monthly cost to deliver: £5.00
- Monthly profit margin: £10.00 (200% markup, 67% margin)
- Setup profit: -£1.00 (subsidized to attract customers)
- Contract length: 30 days (monthly renewal)

Step 4: Define Customer Eligibility

```json
{
  "residential": true,
  "business": false,
  "customer_can_purchase": true,
  "available_from": "2025-01-01T00:00:00Z",
  "available_until": null
}
```

- Residential customers can order
- Business customers cannot (different product line)
- Self-service purchase enabled
- Available from Jan 1, 2025 onwards
- No end date (ongoing offer)

Step 5: Configure Auto-Renewal

```json
{
  "auto_renew": "prompt",
  "allow_auto_renew": true
}
```

- `"prompt"` - Ask customer if they want auto-renewal at purchase
- `"true"` - Automatically renew without asking
- `"false"` - Never auto-renew (manual top-up only)
- `allow_auto_renew: true` - Customer can enable/disable auto-renewal later

Step 6: Specify Inventory Requirements

Inventory requirements define which physical or virtual resources must be allocated when provisioning this product. This is a critical step that connects your product catalog to your `Inventory Management System <administration_inventory>`.

```json
{
  "inventory_items_list": "['SIM Card', 'Mobile Number']"
}
```

What Are Inventory Items?

Inventory items are trackable resources stored in the OmniCRM inventory system. Each item has:

- **Type** - Defined by the Inventory Template (e.g., "SIM Card", "Mobile Number", "Modem")
- **Unique attributes** - Serial numbers, MAC addresses, phone numbers, etc.
- **State** - In Stock, Assigned, Decommissioned, etc.
- **Location** - Physical or logical location

How Inventory Requirements Work:

The `inventory_items_list` is a Python list (as a string) containing inventory type names. Each name must exactly match an existing `Inventory Template <administration_inventory>` name.

Example Inventory Requirements:

```
# Mobile SIM product
inventory_items_list: "['SIM Card', 'Mobile Number']"

# Fixed internet service
inventory_items_list: "['Modem Router', 'Static IP Address']"

# Digital service (no physical items)
inventory_items_list: "[]"

# Fixed wireless with CPE
inventory_items_list: "['Fixed Wireless CPE', 'IPv4 Address', 'IPv6 Prefix']"
```

The Inventory Picker Process

When a user provisions a product with inventory requirements, the system enforces a mandatory selection process:

1. Provision Button Clicked

After selecting the product, the user clicks "Provision". Instead of immediately provisioning, the system checks `inventory_items_list`.

2. Inventory Picker Modal Appears

If inventory is required, a modal dialog appears with a separate dropdown for each inventory type:

3. Filtering Available Inventory

The dropdown for each inventory type only shows items that are:

- **Correct Type** - Matches the inventory template name exactly
- **Available Status** - `item_state` is "New" or "In Stock" (not "Assigned" or "Damaged")
- **Not Assigned** - `service_id` and `customer_id` are NULL
- **In Stock at Location** - Optionally filtered by warehouse/store location

Example Dropdown Options:

For "SIM Card" inventory type, the dropdown might show:

Each option displays:

- Inventory ID or reference number
- Primary identifier (`itemtext1` - e.g., ICCID for SIM, number for phone)
- Current location (`item_location`)

4. Selection Required to Proceed

Critical Rule: Provisioning CANNOT proceed without selecting all required inventory items.

- "Continue" button is disabled until all dropdowns have selections
- User must select one item for each inventory type
- System validates selections before proceeding

5. Selected Inventory Passed to Ansible

Once user clicks "Continue", the selected inventory IDs are passed to the Ansible playbook as variables:

```
# User selected:
# - SIM Card inventory_id: 5001
# - Mobile Number inventory_id: 5002

# Variables passed to Ansible:
{
    "product_id": 42,
    "customer_id": 123,
    "SIM Card": 5001,           # Inventory ID
    "Mobile Number": 5002,       # Inventory ID
```

```
    "access_token": "eyJ..."
}
```

Note: The variable name matches the inventory type exactly. The playbook uses `hostvars[inventory_hostname]['SIM Card']` to access the inventory ID.

6. Playbook Fetches Full Inventory Details

The Ansible playbook uses the inventory ID to fetch complete details:

```
- name: Get SIM Card details from inventory
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
hostvars[inventory_hostname]['SIM Card'] }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
  register: api_response_sim

- name: Extract ICCID and IMSI
  set_fact:
    iccid: "{{ api_response_sim.json.itemtext1 }}"
    imsi: "{{ api_response_sim.json.itemtext2 }}"
```

Now the playbook has all SIM details (ICCID, IMSI, etc.) to provision the subscriber in the HSS.

7. Inventory State Changed to "Assigned"

After the service record is created, the playbook updates inventory to link it to the service:

```
- name: Assign SIM Card to Service
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
hostvars[inventory_hostname]['SIM Card'] }}"
    method: PATCH
    body:
      {
        "service_id": "{{ service_creation_response.json.service_id
}}",
        "customer_id": "{{ customer_id }}",
        "item_state": "Assigned"
      }
```

Important: Inventory assignment happens **during playbook execution** as a specific task, NOT when the provision button is clicked. This means:

- **Risk of Double-Allocation**: Between clicking "Provision" and inventory

being assigned, another user could theoretically select the same inventory item
- **Best Practice**: For high-volume operations, implement inventory locking or use database transactions
- **Rollback on Failure**: If the playbook fails before inventory assignment, inventory remains unassigned and available for reuse

Why Not Assign Earlier?

Inventory isn't assigned when "Provision" is clicked because:

1. **Service ID Needed**: The `service_id` doesn't exist until the service is created in the playbook
2. **Rollback Simplicity**: If provisioning fails early (e.g., OCS account creation fails), inventory doesn't need cleanup
3. **Flexibility**: Playbook can decide not to assign inventory based on conditional logic

Handling Failed Provisions:

When a provision fails after inventory is assigned, the rescue block should release inventory:

```yaml
rescue:
  - name: Release inventory on failure
    uri:
      url: "{{ crm_config.crm.base_url }}/crm/inventory/
inventory_id/{{ hostvars[inventory_hostname]['SIM Card'] }}"
      method: PATCH
      body:
        {
          "service_id": null,
          "customer_id": null,
          "item_state": "In Stock"
        }
    when: service_id is defined  # Only if service was created
```

This ensures inventory isn't left in an "Assigned" state for a non-existent or failed service.

When Inventory List is Empty

If `inventory_items_list: "[]"` (empty list), the inventory picker is skipped entirely and provisioning proceeds immediately. This is common for:

- **Digital products** - Software licenses, VPN accounts
- **Service addons** - Data top-ups that don't need new hardware
- **Virtual services** - That don't consume trackable resources

Example: A "5GB Data Boost" addon has `inventory_items_list: "[]"` because it just adds balance to an existing service without needing new hardware.

Inventory Template Setup

Before using an inventory type in `inventory_items_list`, you must create the Inventory Template:

1. Navigate to **Administration → Inventory → Templates**
2. Create template with exact name (e.g., "SIM Card")
3. Define fields:
    - `itemtext1_label`: "ICCID"
    - `itemtext2_label`: "IMSI"
    - `itemtext3_label`: "PUK Code"
4. Add inventory items of this type to stock

For complete details on creating and managing inventory templates, see Inventory Management <administration_inventory>.

Multiple Items of Same Type

While the `inventory_items_list` is an array, having duplicate types (e.g., `"['SIM Card', 'SIM Card']"`) is **not recommended** as it may cause confusion in the UI and playbook variable naming.

For scenarios requiring multiple similar items:

Option 1: Create distinct inventory template names

```yaml
# Dual-SIM phone service
inventory_items_list: "['Primary SIM Card', 'Secondary SIM Card', 'Mobile Number']"
```

Create separate templates: "Primary SIM Card" and "Secondary SIM Card" with same fields but different names.

Option 2: Use a single bundled inventory item

```yaml
# Dual-SIM kit
inventory_items_list: "['Dual SIM Kit', 'Mobile Number']"
```

Where "Dual SIM Kit" inventory template has fields for both SIMs (`itemtext1`: Primary ICCID, `itemtext2`: Secondary ICCID, etc.).

Common Inventory Scenarios

Mobile Service:

```yaml
inventory_items_list: "['SIM Card', 'Mobile Number']"
```

- SIM Card: Physical or eSIM with ICCID/IMSI
- Mobile Number: Phone number (MSISDN)

Fixed Internet:

```
inventory_items_list: "['Modem Router', 'Static IP Address']"
```

- Modem Router: CPE device with MAC address
- Static IP Address: IPv4 from address pool

Fixed Wireless:

```
inventory_items_list: "['Fixed Wireless CPE', 'IPv4 Address', 'IPv6 Prefix']"
```

- CPE: Customer premises equipment (antenna, modem)
- IPv4: Public IP address
- IPv6 Prefix: /56 or /64 prefix

Note: Appointments and scheduling are **not** inventory items. Use separate scheduling/calendar systems for installation appointments.

VoIP Service:

```
inventory_items_list: "['DID Number']"
```

- DID Number: Direct Inward Dialing phone number

Note: SIP usernames, passwords, and account configurations are **generated programmatically** by the provisioning playbook, not selected from inventory.

GPON/Fiber:

```
inventory_items_list: "['ONT Device', 'GPON Port', 'IPv4 Address', 'Fiber Drop Cable']"
```

- ONT Device: Optical Network Terminal with serial number
- GPON Port: Specific port on OLT with fiber connection
- IPv4 Address: Public or private IP
- Fiber Drop Cable: Physical fiber cable from street to premises (tracked for asset management)

Equipment Rental:

```
inventory_items_list: "['Rental Modem']"
```

- Tracks which modem is with which customer
- Important for recovering equipment on cancellation

Why Inventory Requirements Matter

1. Prevent Double-Allocation

Without inventory tracking, you could accidentally:

- Assign same SIM card to two customers
- Allocate same IP address to multiple services
- Ship same equipment serial to different locations

Inventory picker ensures each item is assigned to exactly one service.

2. Audit Trail

Inventory assignment creates complete audit trail:

- Which SIM card is with which customer
- When was it assigned
- Which service is using which phone number
- Equipment history (who had it, when, for what service)

3. Resource Planning

Track inventory levels:

- Alert when SIM cards running low
- Reorder before stockout
- Plan technician schedules based on CPE availability
- Manage IP address space allocation

4. Cost Tracking

Link wholesale cost to specific item:

- Track cost of each SIM card
- Calculate equipment depreciation
- Identify lost or stolen items
- Accurate COGS (Cost of Goods Sold)

5. Deprovisioning

When service is cancelled, inventory can be:

- Released back to stock (SIM cards, modems)
- Retired (damaged equipment)
- Returned to vendor (rental equipment)
- Kept for grace period (phone numbers before release)

Troubleshooting Inventory Picker Issues

Problem: "No inventory available" message appears

Causes:

- No inventory items of required type exist in database
- All items are already "Assigned" to other services
- Items are marked as "Damaged" or "Out Of Service"
- Inventory template name doesn't match exactly (case-sensitive)

Solution:

1. Verify inventory template exists: **Administration → Inventory → Templates**
2. Check template name matches exactly (including spaces, case)
3. Add inventory items of this type: **Administration → Inventory → Add Item**
4. Verify items are in "New" or "In Stock" state
5. Check items aren't already assigned (`service_id` should be NULL)

Problem: Inventory picker doesn't appear

Causes:

- `inventory_items_list` is empty: `"[]"`
- `inventory_items_list` is NULL or not set
- Product category is "addon" and inherits parent service inventory

Solution:

- If inventory is needed, set `inventory_items_list: "['Type1', 'Type2']"`
- Verify product definition saved correctly
- Check API response for product includes inventory_items_list

Problem: Playbook fails with "inventory not found"

Causes:

- Playbook references wrong variable name
- Inventory ID not passed correctly
- Inventory was deleted between selection and provisioning

Solution:

- Verify playbook uses correct variable: `hostvars[inventory_hostname]['SIM Card']`
- Check variable is integer: `{{ hostvars[inventory_hostname]['SIM Card'] | int }}`
- Add error handling in playbook for missing inventory

See Inventory Management <administration_inventory> for complete details on creating templates, adding items, and managing stock levels.

Step 7: Define Features and Terms

Features and terms are customer-facing marketing and legal content that helps customers understand what they're buying and the obligations involved.

```json
{
  "features_list": "20GB High-Speed Data. Unlimited Calls & Texts. EU Roaming Included. No Contract. 30-Day Expiry",
  "terms": "Credit expires after 30 days. Data, calls, and texts valid only within expiry period. Fair use policy applies. See website for full terms."
}
```

Purpose and Business Value

Features List - Marketing & Sales:

The features list serves multiple critical business functions:

1. **Product Differentiation** - Helps customers quickly compare products and choose the right one
   - "Prepaid Mobile 20GB" vs "Prepaid Mobile 50GB" - features clearly show the difference
   - Without features, customers only see price, missing value proposition
2. **Marketing Communication** - Key selling points prominently displayed
   - "EU Roaming Included" attracts international travelers
   - "No Contract" appeals to commitment-averse customers
   - Features drive purchase decisions
3. **Customer Expectations** - Sets clear expectations about what's included
   - Reduces support calls ("Does this include calls?" → clearly listed)
   - Prevents misunderstandings and refund requests
   - Builds trust through transparency
4. **Self-Service** - Enables customers to self-select appropriate products
   - Customer reads features, understands offering, makes informed choice
   - Reduces need for sales staff explanation
   - Speeds up purchase process
5. **SEO and Discoverability** - Features can be indexed for search
   - Customer searches "unlimited calls mobile plan" → product appears
   - Improves product catalog searchability

Terms and Conditions - Legal & Compliance:

Terms serve legal and operational purposes:

1. **Legal Protection** - Protects business from disputes and liability
    - "Credit expires after 30 days" - customer cannot demand refund at 31 days
    - "Fair use policy applies" - prevents abuse (tethering entire office on mobile plan)
    - Creates binding agreement
2. **Expectation Management** - Prevents customer dissatisfaction
    - "Valid only within expiry period" - customer knows usage deadline
    - "Cannot be refunded" (for addons) - prevents fraudulent purchases
    - Reduces chargebacks and complaints
3. **Regulatory Compliance** - Meets legal requirements
    - Consumer protection laws require clear terms
    - Telecommunications regulations mandate disclosure
    - GDPR/privacy terms can be referenced
4. **Operational Boundaries** - Defines service scope and limitations
    - "Subject to network coverage" - not liable for dead zones
    - "Speed may vary" - manages expectations on "up to" speeds
    - "Equipment must be returned" - ensures rental equipment recovery
5. **Audit Trail** - Proves customer was informed
    - Customer accepted terms at purchase
    - System logs acceptance timestamp
    - Defensible in disputes or legal proceedings

Real-World Example:

Customer buys "Unlimited Calls & Texts" plan, then uses it for telemarketing (10,000 calls/day). Without terms:

- Customer: "You said unlimited!"
- Provider: "We meant personal use..."
- Customer: "That's not what you advertised!"
- Result: Dispute, potential regulator complaint, brand damage

With terms: "Fair use policy applies. Service is for personal use only. Commercial use prohibited."

- Provider: Points to terms customer accepted
- Customer cannot claim ignorance
- Legal basis to suspend service
- Dispute resolved in provider's favor

Features List Format:

Understanding the correct format is critical because **improper formatting breaks the UI display**. Features might appear as one long string instead of bullet points, or not display at all.

The `features_list` field can be formatted in two ways:

Option 1: Period-Separated String (Recommended)

Features are separated by a period and space (". "). The UI splits on this delimiter and renders each feature as a bullet point.

Why this format?

- Simple to edit - just type features with periods between them
- No special characters to escape
- Works reliably across all UI components
- Easy to update without breaking JSON syntax

Correct vs Incorrect:

Option 2: JSON Array String

```
"['20GB High-Speed Data', 'Unlimited Calls & Texts', 'EU Roaming Included']"
```

The UI can also parse JSON arrays. Note this is a **string containing JSON**, not an actual JSON array in the database.

Why this format exists?

- Allows features with periods in them (e.g., "Up to 100Mbps. Subject to availability.")
- Programmatic generation from scripts/API is easier
- Imported from external product catalogs that use arrays

Important: This must be valid Python list syntax as a string. Single quotes around each item, double quotes around the whole string.

Which Format to Use?

- **Period-separated** - For manual product creation in UI (simpler, less error-prone)
- **JSON array** - For API/script-based product creation (more robust for complex features)

Both formats produce identical output in the UI - they just affect how you input the data.

Where Features Appear in the UI:

1. Product Catalog (Customer View)

When customers browse available products, features are displayed as bullet points on each product card:

2. Product Details Page

Clicking "View Details" shows full product information including:

- Product name and icon
- Pricing (monthly cost, setup cost)
- Full features list (bullet points)
- Terms and conditions (see below)
- Availability and eligibility

3. Provisioning Confirmation

During provisioning, features are shown for user to review before confirming:

> Features: • 20GB High-Speed Data • Unlimited Calls & Texts • EU Roaming Included • No Contract • 30-Day Expiry
>
> Cost: £15.00/month Setup: £0.00
>
> [Cancel] [Confirm & Provision]

4. Service Details (After Provisioning)

After service is active, features are displayed on the service detail page for customer reference.

Terms and Conditions Format:

The `terms` field is plain text that can include newlines:

Where Terms Appear in the UI:

1. Product Details Page

Terms are displayed in a collapsed section that expands when clicked:

2. Order Confirmation

During provisioning, a checkbox requires user to accept terms:

> [Provision] button disabled until checked

3. Invoices

Service terms may be included on invoices as footnotes for clarity.

Best Practices:

- **Features:** Keep concise (under 50 characters each), focus on key benefits

- **Terms:** Include critical legal requirements, expiration policies, fair use policies
- **Both:** Update when product changes to keep customers informed

Step 8: Link Ansible Provisioning Playbook

```
{
  "provisioning_play": "play_local_mobile_sim",
  "provisioning_json_vars": "{
    \"days\": 30,
    \"data_gb\": 20,
    \"voice_minutes\": \"unlimited\",
    \"sms_count\": \"unlimited\"
  }"
}
```

- `provisioning_play` - Name of Ansible playbook (without .yaml extension)
- `provisioning_json_vars` - Default variables passed to playbook
- Playbook must exist at: `OmniCRM-API/Provisioners/plays/play_local_mobile_sim.yaml`

Complete Product Definition

```
{
  "product_name": "Prepaid Mobile 20GB",
  "product_slug": "prepaid-mobile-20gb",
  "category": "standalone",
  "service_type": "mobile",
  "enabled": true,
  "icon": "fa-solid fa-sim-card",
  "comment": "Prepaid mobile SIM with 20GB data, unlimited calls & texts",

  "retail_cost": 15.00,
  "wholesale_cost": 5.00,
  "retail_setup_cost": 0.00,
  "wholesale_setup_cost": 1.00,
  "contract_days": 30,

  "residential": true,
  "business": false,
  "customer_can_purchase": true,
  "available_from": "2025-01-01T00:00:00Z",
  "available_until": null,

  "auto_renew": "prompt",
  "allow_auto_renew": true,

  "inventory_items_list": "['SIM Card', 'Mobile Number']",
```

```
  "features_list": "[
    '20GB High-Speed Data',
    'Unlimited Calls & Texts',
    'EU Roaming Included',
    'No Contract',
    '30-Day Expiry'
  ]",
  "terms": "Credit expires after 30 days. Data, calls, and texts
valid only within expiry period. Fair use policy applies.",

  "provisioning_play": "play_local_mobile_sim",
  "provisioning_json_vars": "{
    \"days\": 30,
    \"data_gb\": 20,
    \"voice_minutes\": \"unlimited\",
    \"sms_count\": \"unlimited\"
  }"
}
```

Creating an Addon Product

Addons enhance or modify existing services. They come in two types: **virtual addons** (no physical resources) and **hardware addons** (require inventory).

Example 1: Virtual Addon (5GB Data Boost)

A digital addon that adds data to an existing mobile service:

```
{
  "product_name": "5GB Data Boost",
  "product_slug": "5gb-data-boost",
  "category": "addon",
  "service_type": "mobile",
  "enabled": true,
  "icon": "fa-solid fa-plus",
  "comment": "Add 5GB extra data to existing mobile service",

  "retail_cost": 5.00,
  "wholesale_cost": 1.50,
  "retail_setup_cost": 0.00,
  "wholesale_setup_cost": 0.00,
  "contract_days": 0,

  "residential": true,
  "business": true,
  "customer_can_purchase": true,
```

```
  "auto_renew": "false",
  "allow_auto_renew": false,

  "inventory_items_list": "[]",
  "relies_on_list": "",

  "features_list": "5GB High-Speed Data. Valid for 7 Days",
  "terms": "Data expires after 7 days or when exhausted. Cannot be
refunded.",

  "provisioning_play": "play_topup_charge_then_action",
  "provisioning_json_vars": "{
    \"data_gb\": 5,
    \"days\": 7
  }"
}
```

Example 2: Hardware Addon (Modem Rental)

An addon that provides physical equipment for an existing fiber service:

```
{
  "product_name": "WiFi 6 Modem Rental",
  "product_slug": "wifi6-modem-rental",
  "category": "addon",
  "service_type": "internet",
  "enabled": true,
  "icon": "fa-solid fa-router",
  "comment": "Add WiFi 6 modem to fiber service - rental",

  "retail_cost": 10.00,
  "wholesale_cost": 3.00,
  "retail_setup_cost": 0.00,
  "wholesale_setup_cost": 45.00,
  "contract_days": 30,

  "residential": true,
  "business": true,
  "customer_can_purchase": true,

  "auto_renew": "true",
  "allow_auto_renew": true,

  "inventory_items_list": "['Rental Modem']",
  "relies_on_list": "",

  "features_list": "WiFi 6 (802.11ax). Dual-band 2.4GHz + 5GHz. Up to
40 devices. Parental controls",
  "terms": "Equipment rental. Must be returned on service
```

```
cancellation or £150 replacement fee applies. Equipment remains
property of provider.",

  "provisioning_play": "play_addon_assign_modem",
  "provisioning_json_vars": "{
    \"device_type\": \"modem_router\",
    \"requires_configuration\": true
  }"
}
```

Key Differences for Addons:

- `category: "addon"` - Applied to existing service, not standalone
- `contract_days: 0` (virtual) or `30` (recurring rental) - Billing frequency
- `inventory_items_list: "[]"` (virtual) or `"['Rental Modem']"` (hardware) - Physical resources
- `auto_renew: "false"` (one-time) or `"true"` (rental) - Recurring behavior
- `relies_on_list: ""` - Empty means applies to any service of matching `service_type`

Why Hardware Addons Need Inventory:

Hardware addons require `inventory_items_list` because:

1. **Track Equipment** - Know which modem is with which customer
2. **Prevent Stockouts** - Can't provision addon if no modems in stock
3. **Recovery** - When customer cancels, know which equipment to recover
4. **Cost Tracking** - Link wholesale cost to specific serial number
5. **Depreciation** - Track equipment value over rental period
6. **Warranty** - Identify defective units by serial number

Addon Provisioning Flow with Inventory:

When a customer adds "WiFi 6 Modem Rental" to their fiber service:

1. **Addon Selected** - Customer clicks "Add to Service"
2. **Inventory Picker Appears** - Just like standalone services:
3. **Payment Processed** - £10.00 monthly rental charged
4. **Modem Assigned** - Inventory updated:
    - `service_id`: Linked to fiber service
    - `customer_id`: Linked to customer
    - `item_state`: "Assigned"
5. **Shipping Triggered** - Fulfillment system notified to ship modem
6. **Installation** - Customer receives modem, plugs into ONT
7. **Recurring Billing** - £10/month charged until addon cancelled

Deprovisioning Hardware Addons:

When customer cancels modem rental:

1. **Cancellation Initiated** - Customer clicks "Remove Addon"
2. **Return Process Started**:
   - Email sent with return instructions
   - Prepaid shipping label generated
   - 14-day grace period before penalty
3. **Equipment Returned**:
   - Inventory updated: `item_state` = "In Stock" (after refurbishment)
   - Or `item_state` = "Damaged" (if defective)
   - Linked to next customer once refurbished
4. **No Return**:
   - After 14 days, £150 replacement fee charged
   - Inventory marked: `item_state` = "Lost"
   - Wholesale cost (£45) + replacement value recovered

Pricing for Addons:

Addons can be priced differently from standalone services:

- Virtual addons typically have no setup costs
- Hardware addons may have wholesale setup costs for equipment
- Recurring rental addons use `contract_days` for billing frequency

Stage 2: The Provisioning Process

When a customer orders the "Prepaid Mobile 20GB" product, OmniCRM orchestrates provisioning through Ansible.

Provisioning Flow Diagram

Customer Orders → Inventory Selection → Provisioning Job Created ↓ ↓ Payment Authorized ← Variables Assembled ← Ansible Playbook Executed ↓ ↓ Service Record Created → OCS Account Setup → Inventory Assigned → Service Active

Step-by-Step Provisioning Flow

1. Customer Initiates Order

From customer page:

- Staff clicks "Add Service"
- Selects "Prepaid Mobile 20GB" from product carousel
- Product details and pricing displayed

2. Inventory Selection

System prompts for required inventory:

- **SIM Card** - Dropdown shows available SIM cards in stock

        ◦ Example: "SIM-00123 - ICCID: 8944..."
- **Mobile Number** - Dropdown shows available phone numbers
    - Example: "+44 7700 900123"

Staff or customer selects items from available inventory.

3. Pricing Confirmation

System displays final pricing:

- Setup cost: £0.00 (free activation)
- Monthly cost: £15.00
- Due today: £15.00 (first month)
- Renewal date: 30 days from today

If auto-renew prompting enabled, customer chooses:

- ☐ Automatically renew this service every 30 days

4. Provision Button Clicked

When "Provision" is clicked, the API:

- Creates `Provision` record with status "Running" (status=1)
- Merges variables from product + request + inventory selections
- Spawns background thread to execute Ansible playbook
- Returns provision_id to UI for status tracking

5. Variables Assembled

System merges variables from multiple sources:

From Product:

```
{
  "days": 30,
  "data_gb": 20,
  "voice_minutes": "unlimited",
  "sms_count": "unlimited"
}
```

From Request:

```
{
  "product_id": 42,
  "customer_id": 123,
  "SIM Card": 5001,
  "Mobile Number": 5002
}
```

System-Added:


```json
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "initiating_user": 7
}
```


Final Variables Passed to Ansible:


```json
{
    "product_id": 42,
    "customer_id": 123,
    "SIM Card": 5001,
    "Mobile Number": 5002,
    "days": 30,
    "data_gb": 20,
    "voice_minutes": "unlimited",
    "sms_count": "unlimited",
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "initiating_user": 7
}
```


6. Ansible Playbook Execution

The playbook `play_local_mobile_sim.yaml` executes with these variables.

Understanding the Ansible Provisioning Playbook

Let's examine a real provisioning playbook to understand what happens behind the scenes.

Mobile SIM Provisioning Playbook Example

Location: `OmniCRM-API/Provisioners/plays/play_local_mobile_sim.yaml`

High-Level Structure:

```yaml
- name: Mobile SIM Provisioning
  hosts: localhost
  gather_facts: no
  become: False

  tasks:
    - name: Main block
      block:
        # 1. Load configuration
        # 2. Fetch product details from API
        # 3. Fetch customer details from API
```

```
        # 4. Fetch inventory details from API
        # 5. Create account in OCS (CGRateS)
        # 6. Add balances and allowances to OCS
        # 7. Create service record in CRM
        # 8. Assign inventory to service
        # 9. Record transactions
        # 10. Send welcome notifications

      rescue:
        # Rollback on failure
        # - Remove OCS account
        # - Release inventory
        # - Log error
```

Detailed Playbook Walkthrough:

Task 1: Load Configuration

```
- name: Include vars of crm_config
  ansible.builtin.include_vars:
    file: "../../crm_config.yaml"
    name: crm_config
```

Loads system configuration including:

- OCS/CGRateS URL and credentials
- CRM base URL
- Tenant configuration

Task 2: Fetch Product Details

```
- name: Get Product information from CRM API
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/product/product_id/{{
product_id }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
    return_content: yes
  register: api_response_product
```

What This Does:

- Calls `GET /crm/product/product_id/42`
- Retrieves complete product definition
- Stores in `api_response_product` variable

Why: Even though we have `provisioning_json_vars` from the product, we fetch the full product to get:

- Latest pricing (may have changed since order started)
- Product name for service naming
- Features list for documentation
- Wholesale costs for margin tracking

Task 3: Set Package Facts

```yaml
- name: Set package facts
  set_fact:
    package_name: "{{ api_response_product.json.product_name }}"
    monthly_cost: "{{ api_response_product.json.retail_cost }}"
    setup_cost: "{{ api_response_product.json.retail_setup_cost }}"
```

Extracts commonly-used values into simple variables for readability.

Task 4: Fetch Inventory Details

```yaml
- name: Get SIM information from CRM API
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
hostvars[inventory_hostname]['SIM Card'] }}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
  register: api_response_sim

- name: Set IMSI from Inventory response
  set_fact:
    imsi: "{{ api_response_sim.json.itemtext2 }}"
    iccid: "{{ api_response_sim.json.itemtext1 }}"
```

What This Does:

- Looks up SIM Card inventory ID 5001
- Retrieves SIM details:
  - `itemtext1` = ICCID (SIM card number)
  - `itemtext2` = IMSI (subscriber identity)
- Does same for Mobile Number inventory (retrieves phone number)

Why This Matters:

- IMSI is needed to provision subscriber in HSS (Home Subscriber Server)
- ICCID is recorded in service notes for troubleshooting
- Phone number (MSISDN) is displayed to customer and used for routing

Task 5: Generate Service UUID

```yaml
- name: Generate UUID Fact
  set_fact:
```

```yaml
    uuid: "{{ 99999999 | random | to_uuid }}"

- name: Set Service UUID
  set_fact:
    service_uuid: "Local_Mobile_SIM_{{ uuid[0:8] }}"
```

What This Does:

- Generates random UUID
- Creates service_uuid like `Local_Mobile_SIM_a3f2c1d8`

Why:

- Service UUID is the unique identifier in OCS/CGRateS
- Used for all charging operations
- Must be globally unique across all services

Task 6: Create OCS Account

```yaml
- name: Create account in OCS
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    headers:
      Content-Type: "application/json"
    body:
      {
        "method": "ApierV2.SetAccount",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}",
          "ActionPlanIds": [],
          "ExtraOptions": {
            "AllowNegative": false,
            "Disabled": false
          },
          "ReloadScheduler": true
        }]
      }
  register: ocs_create_response
```

What This Does:

- Calls CGRateS JSON-RPC API
- Creates new account with service_uuid
- Sets account to active (not disabled)
- Prevents negative balance (prepaid mode)

Why:

- OCS account is where all charging happens
- Balances (data, voice, SMS, money) are stored here
- Usage is tracked and rated in real-time

Task 7: Add Data Balance

```
- name: Add 20GB Data Balance
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.AddBalance",
        "params": [{
          "Tenant": "{{ crm_config.ocs.ocsTenant }}",
          "Account": "{{ service_uuid }}",
          "BalanceType": "*data",
          "Balance": {
            "ID": "DATA_20GB_Monthly",
            "Value": 21474836480,
            "ExpiryTime": "+720h",
            "Weight": 10,
            "DestinationIDs": "*any"
          }
        }]
      }
```

What This Does:

- Adds 20GB data balance to account
- Value: 21474836480 bytes (20 * 1024 * 1024 * 1024)
- Expires in 720 hours (30 days)
- Weight 10 (higher weight consumed first)

Task 8: Add Unlimited Voice & SMS

```
- name: Add Unlimited Voice
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.AddBalance",
        "params": [{
          "Account": "{{ service_uuid }}",
```

```
          "BalanceType": "*voice",
          "Balance": {
            "ID": "VOICE_Unlimited",
            "Value": 999999999,
            "ExpiryTime": "+720h"
          }
        }]
      }
```

- Adds 999,999,999 seconds of voice (essentially unlimited)
- Expires in 30 days

Task 9: Create Service Record in CRM

```
- name: Add Service via API
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/service/"
    method: PUT
    body_format: json
    headers:
      Authorization: "Bearer {{ access_token }}"
    body:
      {
        "customer_id": "{{ customer_id }}",
        "product_id": "{{ product_id }}",
        "service_name": "Mobile - {{ phone_number }}",
        "service_type": "mobile",
        "service_uuid": "{{ service_uuid }}",
        "service_status": "Active",
        "service_provisioned_date": "{{ provision_datetime }}",
        "retail_cost": "{{ monthly_cost }}",
        "wholesale_cost": "{{
api_response_product.json.wholesale_cost }}",
        "icon": "fa-solid fa-sim-card"
      }
  register: service_creation_response
```

What This Creates:

- Service record linked to customer
- Links to OCS via `service_uuid`
- Stores retail and wholesale costs
- Sets status to "Active"
- Returns `service_id` for subsequent operations

Task 10: Assign Inventory to Service

```
- name: Assign SIM Card to Service
  uri:
```

```
    url: "{{ crm_config.crm.base_url }}/crm/inventory/inventory_id/{{
hostvars[inventory_hostname]['SIM Card'] }}"
    method: PATCH
    body_format: json
    headers:
      Authorization: "Bearer {{ access_token }}"
    body:
      {
        "service_id": "{{ service_creation_response.json.service_id
}}",
        "customer_id": "{{ customer_id }}",
        "item_state": "Assigned"
      }
```

What This Does:

- Updates SIM Card inventory record
- Sets `service_id` to link SIM to service
- Changes state from "In Stock" to "Assigned"
- Repeats for Mobile Number inventory

Why:

- Tracks which SIM is assigned to which customer
- Prevents double-allocation of inventory
- Enables inventory reporting and auditing

Task 11: Record Setup Cost Transaction

```
- name: Add Setup Cost Transaction
  uri:
    url: "{{ crm_config.crm.base_url }}/crm/transaction/"
    method: PUT
    body_format: json
    headers:
      Authorization: "Bearer {{ access_token }}"
    body:
      {
        "customer_id": "{{ customer_id }}",
        "service_id": "{{ service_creation_response.json.service_id
}}",
        "title": "{{ package_name }} - Setup",
        "description": "Activation fee",
        "retail_cost": "{{ setup_cost }}",
        "wholesale_cost": "{{
api_response_product.json.wholesale_setup_cost }}"
      }
```

What This Does:

- Records £0.00 setup charge to customer (retail)
- Records £1.00 wholesale cost
- Creates transaction record for invoicing

Task 12: Rescue Block (Error Handling)

```yaml
rescue:
  - name: Remove account in OCS on failure
    uri:
      url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
      method: POST
      body:
        {
          "method": "ApierV2.RemoveAccount",
          "params": [{
            "Account": "{{ service_uuid }}"
          }]
        }

  - name: Fail the provisioning
    fail:
      msg: "Provisioning failed, rolled back OCS account"
```

What This Does:

- If any task fails, rescue block executes
- Deletes OCS account that was partially created
- Releases inventory back to "In Stock"
- Fails the provision job with error message

Why:

- Prevents orphaned accounts in OCS
- Ensures clean rollback on errors
- Maintains data consistency

Provisioning Complete: What Was Created

After successful provisioning, the system has:

1. OCS Account (CGRateS):

- Account ID: `Local_Mobile_SIM_a3f2c1d8`
- Balances:
    - 20GB data (expires in 30 days)
    - Unlimited voice (999M seconds, expires in 30 days)
    - Unlimited SMS (999M messages, expires in 30 days)

2. CRM Service Record:

- Service ID: 1234
- Customer: John Doe (customer_id: 123)
- Product: Prepaid Mobile 20GB (product_id: 42)
- Service Name: "Mobile - +44 7700 900123"
- Service UUID: `Local_Mobile_SIM_a3f2c1d8`
- Status: Active
- Monthly Cost: £15.00 (retail), £5.00 (wholesale)
- Profit: £10.00/month

3. Inventory Assignments:

- SIM Card 5001: Assigned to service 1234, customer 123
- Mobile Number 5002: Assigned to service 1234, customer 123

4. Transaction Records:

- Setup cost transaction created
- First month charge recorded

5. Customer Can Now:

- View service in self-care portal
- See 20GB data balance
- Make calls and send SMS
- Top up or add addons
- View usage in real-time

Stage 3: Adding Addons and Topups

After a service is active, customers can purchase addons to enhance their service.

Addon Provisioning Flow

Let's say customer has used 18GB of their 20GB allowance and wants to buy the "5GB Data Boost" addon.

1. Customer Navigates to Service

- Opens "Mobile - +44 7700 900123" service page
- Sees current usage: 18GB of 20GB used (90%)
- Clicks "Add Addon" or "Top Up"

2. System Filters Available Addons

Only shows addons where:

- `category = "addon"`
- `service_type = "mobile"` (matches service type)

- `residential = true` (if customer is residential)
- `enabled = true`

Customer sees: "5GB Data Boost - £5.00"

3. Customer Selects Addon

- Clicks "5GB Data Boost"
- Confirms purchase for £5.00
- System captures payment authorization

4. Addon Provisioning Initiated

System calls `play_topup_charge_then_action.yaml` with variables:

```
{
    "product_id": 43,  # 5GB Data Boost product
    "customer_id": 123,
    "service_id": 1234,  # Existing service
    "access_token": "eyJ...",
    "data_gb": 5,  # From provisioning_json_vars
    "days": 7      # From provisioning_json_vars
}
```

Key Difference from Standalone:

- `service_id` is included (existing service to modify)
- No inventory required
- No service creation (modifies existing)

Addon Provisioning Playbook Walkthrough

Task 1: Fetch Service Details

```
- name: Get Service information from CRM API
  uri:
    url: "http://localhost:5000/crm/service/service_id/{{ service_id
}}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
  register: api_response_service

- name: Set service facts
  set_fact:
    service_uuid: "{{ api_response_service.json.service_uuid }}"
    customer_id: "{{ api_response_service.json.customer_id }}"
```

Why:

- Need `service_uuid` to add balance to correct OCS account
- Verifies service exists and is active
- Ensures service belongs to the customer

Task 2: Charge Customer

```yaml
- name: Get Customer's Default Payment Method
  uri:
    url: "http://localhost:5000/crm/stripe/customer_id/{{ customer_id
}}"
    method: GET
    headers:
      Authorization: "Bearer {{ access_token }}"
  register: api_response_stripe

- name: Get default card ID
  set_fact:
    customer_stripe_id: "{{ api_response_stripe.json |
json_query(query) }}"
  vars:
    query: "data[?default_payment_method==`true`].customer_stripe_id
| [0]"

- name: Charge card
  uri:
    url: "http://localhost:5000/crm/stripe/charge_card/{{
customer_stripe_id }}"
    method: POST
    body_format: json
    headers:
      Authorization: "Bearer {{ access_token }}"
    body:
      {
        "retail_cost": 5.00,
        "description": "5GB Data Boost",
        "customer_id": "{{ customer_id }}",
        "service_id": "{{ service_id }}",
        "product_id": "{{ product_id }}",
        "wholesale_cost": 1.50,
        "invoice": true
      }
  register: charge_response

- name: Assert payment successful
  assert:
    that:
      - charge_response.status == 200
```

What This Does:

- Finds customer's default Stripe payment method
- Charges £5.00 to the card
- Records wholesale cost £1.50 for margin tracking
- Creates transaction linked to service
- Adds to next invoice
- Fails provision if payment fails

Why Charge First:

- No credit delivered until payment confirmed
- Prevents fraud
- Matches payment to addon provision

Task 3: Add Data Balance to OCS

```yaml
- name: Add 5GB Data Balance
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body_format: json
    body:
      {
        "method": "ApierV1.AddBalance",
        "params": [{
          "Account": "{{ service_uuid }}",
          "BalanceType": "*data",
          "Balance": {
            "ID": "DATA_5GB_Boost_{{ uuid }}",
            "Value": 5368709120,
            "ExpiryTime": "+168h",
            "Weight": 20
          }
        }]
      }
```

What This Does:

- Adds 5GB (5368709120 bytes) to account
- Expires in 168 hours (7 days)
- Weight 20 (higher weight consumed first - boost before monthly allowance)

Customer Balance After Addon:

- Original monthly: 2GB remaining (expires in 25 days)
- New boost: 5GB (expires in 7 days)
- Total available: 7GB
- Usage order: Boost consumed first, then monthly

Task 4: Record Transaction

```yaml
- name: Add Addon Transaction
  uri:
    url: "http://localhost:5000/crm/transaction/"
    method: PUT
    body_format: json
    headers:
      Authorization: "Bearer {{ access_token }}"
    body:
      {
        "customer_id": "{{ customer_id }}",
        "service_id": "{{ service_id }}",
        "title": "5GB Data Boost",
        "description": "Additional 5GB data valid for 7 days",
        "retail_cost": 5.00,
        "wholesale_cost": 1.50
      }
```

What This Does:

- Records £5.00 charge to customer
- Records £1.50 wholesale cost
- Links transaction to service for reporting

Complete Addon Flow Summary

1. Customer selects addon from filtered list
2. Payment authorized and charged
3. Data balance added to OCS account
4. Transaction recorded in CRM
5. Customer immediately sees updated balance: 7GB available

Financial Tracking:

- Service monthly charge: £15 retail, £5 wholesale
- Addon purchase: £5 retail, £1.50 wholesale

Auto-Renewal: Recurring Addons

Some addons can be set to auto-renew (monthly data plans, subscriptions, etc).

Product Configuration:

```json
{
  "product_name": "Monthly 10GB Data Plan",
  "category": "addon",
  "retail_cost": 10.00,
  "contract_days": 30,
  "auto_renew": "true",
```

```
  "provisioning_play": "play_topup_charge_then_action"
}
```

Provisioning Creates ActionPlan:

```
- name: Create ActionPlan for Auto-Renewal
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body:
      {
        "method": "ApierV1.SetActionPlan",
        "params": [{
          "Id": "ServiceID_{{ service_uuid }}__ProductID_{{
product_id }}__MonthlyRenewal",
          "ActionPlan": [{
            "ActionsId": "Action_{{ product_slug }}",
            "Years": "*any",
            "Months": "*any",
            "MonthDays": "*any",
            "WeekDays": "*any",
            "Time": "00:00:00",
            "Weight": 10
          }],
          "Overwrite": false,
          "ReloadScheduler": true
        }]
      }
```

What This Does:

- Creates scheduled task in OCS
- Executes `Action_{{ product_slug }}` every 30 days
- Action charges customer and re-applies data balance
- Continues until customer cancels

Customer Management:

- Customer sees "Next Renewal: Feb 1, 2025 - £10.00" in service view
- Can click "Cancel Auto-Renewal" to stop future charges
- Can click "Renew Now" to immediately apply next month's allowance

Stage 4: Deprovisioning Services

When a customer cancels service, the system must cleanly remove all resources.

Deprovisioning Triggers

Deprovisioning can be triggered by:

1. **Customer cancellation** - Customer clicks "Cancel Service"
2. **Administrative action** - Staff marks service for deactivation
3. **Non-payment** - Service expires due to lack of renewal
4. **Contract end** - Fixed-term contract reaches end date

Deprovisioning Flow

1. Customer Initiates Cancellation

- Navigates to service
- Clicks "Cancel Service"
- System prompts: "Are you sure? Any remaining balance will be forfeited."
- Customer confirms

2. Grace Period (Optional)

Some operators implement grace period:

- Service marked "Pending Cancellation"
- Remains active for 7-30 days
- Customer can reverse cancellation during grace period
- Automatic deprovisioning after grace period

3. Deprovisioning Job Created

System creates provision job with:

```json
{
    "action": "deprovision",
    "service_id": 1234,
    "customer_id": 123,
    "service_uuid": "Local_Mobile_SIM_a3f2c1d8"
}
```

Calls playbook specified in `service.deprovisioning_play` or rescue block of original playbook.

4. Ansible Deprovision Playbook

```yaml
- name: Deprovision Mobile Service
  hosts: localhost
  tasks:
    - name: Disable OCS Account
      uri:
```

```yaml
        url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
        method: POST
        body:
          {
            "method": "ApierV2.SetAccount",
            "params": [{
              "Account": "{{ service_uuid }}",
              "ExtraOptions": { "Disabled": true }
            }]
          }

    - name: Remove ActionPlans (stop auto-renewals)
      uri:
        url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
        method: POST
        body:
          {
            "method": "ApierV1.RemoveActionPlan",
            "params": [{
              "Id": "ServiceID_{{ service_uuid }}__*"
            }]
          }

    - name: Update Service Status in CRM
      uri:
        url: "http://localhost:5000/crm/service/{{ service_id }}"
        method: PATCH
        body:
          {
            "service_status": "Deactivated",
            "service_deactivate_date": "{{ current_datetime }}"
          }

    - name: Release Inventory to Stock
      uri:
        url: "http://localhost:5000/crm/inventory/inventory_id/{{
sim_card_id }}"
        method: PATCH
        body:
          {
            "service_id": null,
            "customer_id": null,
            "item_state": "Decommissioned"
          }
```

What This Does:

1. **Disables OCS account** - Stops all charging, usage blocked
2. **Removes ActionPlans** - Cancels auto-renewals

3. **Updates CRM service** - Status "Deactivated", date recorded
4. **Releases inventory** - SIM marked "Decommissioned", available for reuse (after refurbishment)

5. Post-Deprovisioning

System performs cleanup:

- Customer no longer sees service in self-care portal
- Service remains in CRM for historical reporting
- Transactions and invoices preserved for accounting
- Inventory can be refurbished and reused
- OCS account can be archived after retention period

Partial vs Full Deprovisioning

Partial Deprovisioning (Suspension):

- Used for non-payment or temporary suspension
- OCS account disabled but not deleted
- Balances preserved
- Can be re-enabled when payment received

```yaml
- name: Suspend Service
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body:
      {
        "method": "ApierV2.SetAccount",
        "params": [{
          "Account": "{{ service_uuid }}",
          "ExtraOptions": { "Disabled": true }
        }]
      }
```

Full Deprovisioning (Permanent Cancellation):

- Used for permanent cancellation
- OCS account deleted entirely
- Balances forfeit
- Cannot be re-enabled

```yaml
- name: Remove OCS Account
  uri:
    url: "http://{{ crm_config.ocs.cgrates }}/jsonrpc"
    method: POST
    body:
      {
```

```
      "method": "ApierV2.RemoveAccount",
      "params": [{
        "Account": "{{ service_uuid }}"
      }]
    }
```

Best Practices for Product Management

Product Lifecycle Management

Product States:

- `enabled: true` - Product available for new orders
- `enabled: false` - Product disabled, existing services continue

Disabling Products:

- Mark product as `enabled: false` to prevent new orders
- Existing services remain active
- Customers can still renew/modify existing services
- Useful for sunsetting old products

Inventory Management

Inventory States:

- `New` - Fresh stock, ready to assign
- `In Stock` - Available for provisioning
- `Assigned` - Linked to customer service
- `Decommissioned` - Can be refurbished and reused
- `Damaged` - Needs repair or disposal

Reusing Inventory:

After deprovisioning:

- SIM cards: Refurbish and mark "In Stock"
- Phone numbers: Release after porting period (30 days)
- Equipment: Test, refurbish, mark "Used"

Provisioning Metrics

Monitor:

- Provisioning success rate
- Average provisioning time
- Common failure points
- Inventory turnover

Mailjet Integration with OmniCRM

OmniCRM integrates with **Mailjet** to manage all email communication with customers and staff, ensuring professional, branded, and reliable email delivery for both transactional emails and marketing campaigns.

Overview

The Mailjet integration provides:

- **Automated Transactional Emails** - Password resets, invoices, welcome emails, notifications
- **Contact Syncing** - Customer contacts automatically synced to Mailjet for campaigns
- **Email Templates** - 10+ pre-configured email types with customizable Mailjet templates
- **Marketing Campaigns** - Segmented email campaigns based on customer data
- **Reliable Delivery** - Professional email infrastructure with tracking and analytics

Configuration

Mailjet is configured in `OmniCRM-API/crm_config.yaml` under the `mailjet` section.

Basic Configuration

```yaml
mailjet:
  api_key: your_mailjet_api_key
  api_secret: your_mailjet_api_secret
```

Obtaining API Credentials:

1. Create account at <https://www.mailjet.com>
2. Navigate to **Account Settings → API Keys**
3. Copy **API Key** and **Secret Key**
4. Paste into `crm_config.yaml`

Email Template Configuration

OmniCRM uses **10 distinct email template types** for automated communications. Each template is configured with:

- **from_email** - Sender email address
- **from_name** - Sender display name
- **template_id** - Mailjet template ID (numeric)
- **subject** - Email subject line

Template Types and Configuration

Customer Welcome Email

Sent when a new customer account is created.

```yaml
api_crmCommunicationCustomerWelcome:
  from_email: "support@yourcompany.com"
  from_name: "Your Company Support"
  template_id: 5977509
  subject: "Welcome to Your Company"
```

When Sent:

- New customer signs up via Self-Care portal
- Staff creates new customer account
- Customer activates service for first time

Template Variables Available:

- `{{ var:customer_name }}` - Customer's full name
- `{{ var:email }}` - Customer's email address
- `{{ var:company_name }}` - Your company name
- `{{ var:login_url }}` - Link to Self-Care portal
- `{{ var:support_url }}` - Link to support page

Customer Invoice Email

Sent when an invoice is generated and ready for payment.

```yaml
api_crmCommunicationCustomerInvoice:
  from_email: "billing@yourcompany.com"
  from_name: "Your Company Billing"
  template_id: 6759851
  subject: "Your Invoice - "
```

When Sent:

- Invoice automatically generated for billing period
- Manual invoice created by staff
- Customer requests invoice copy

Template Variables Available:

- `{{ var:customer_name }}` - Customer's full name
- `{{ var:invoice_number }}` - Invoice ID/number
- `{{ var:invoice_date }}` - Invoice issue date
- `{{ var:due_date }}` - Payment due date
- `{{ var:total_amount }}` - Total amount due
- `{{ var:invoice_url }}` - Link to view/download invoice PDF
- `{{ var:pay_url }}` - Link to pay invoice online

Invoice Attachment:

The invoice PDF is automatically attached to the email.

Customer Invoice Reminder

Sent to remind customers of overdue invoices.

```yaml
api_crmCommunicationCustomerInvoiceReminder:
  from_email: "billing@yourcompany.com"
  from_name: "Your Company Billing"
  template_id: 6759852
  subject: "Payment Reminder - Invoice Overdue"
```

When Sent:

- Invoice is X days past due (configurable)
- Manual reminder triggered by staff
- Automated reminder workflow (if configured)

Template Variables Available:

- `{{ var:customer_name }}`
- `{{ var:invoice_number }}`
- `{{ var:due_date }}`
- `{{ var:days_overdue }}`
- `{{ var:total_amount }}`
- `{{ var:pay_url }}`

Staff User Welcome Email

Sent when a new staff user account is created.

```yaml
api_crmCommunicationUserWelcome:
  from_email: "admin@yourcompany.com"
  from_name: "Your Company Admin"
  template_id: 5977510
  subject: "Welcome to the Team"
```

When Sent:

- Admin creates new staff user
- "Send Welcome Email" button clicked in user management

Template Variables Available:

- `{{ var:user_name }}` - Staff user's full name
- `{{ var:email }}` - Staff user's email
- `{{ var:role }}` - Assigned role(s)
- `{{ var:login_url }}` - Link to admin portal login
- `{{ var:temp_password }}` - Temporary password (if applicable)
- `{{ var:support_email }}` - IT support contact

User Password Reset

Sent when a user requests to reset their password.

```yaml
api_crmCommunicationUserPasswordReset:
  from_email: "noreply@yourcompany.com"
  from_name: "Your Company Security"
  template_id: 5977511
  subject: "Password Reset Request"
```

When Sent:

- User clicks "Forgot Password" on login page
- User submits password reset request

Template Variables Available:

- `{{ var:user_name }}`
- `{{ var:reset_url }}` - Time-limited password reset link (typically 1 hour)
- `{{ var:expiry_time }}` - When reset link expires

Security Note:

Reset links expire after configured time period (default 1 hour).

User Password Reset Success

Sent to confirm password was successfully changed.

```yaml
api_crmCommunicationUserPasswordResetSuccess:
  from_email: "noreply@yourcompany.com"
  from_name: "Your Company Security"
  template_id: 5977512
  subject: "Password Changed Successfully"
```

When Sent:

- User successfully completes password reset
- Immediately after new password is set

Template Variables Available:

- `{{ var:user_name }}`
- `{{ var:change_date }}` - Date/time password was changed
- `{{ var:ip_address }}` - IP address of change (optional)
- `{{ var:support_email }}` - Contact if change was unauthorized

User Password Change

Sent when a user changes their password from settings.

```yaml
api_crmCommunicationUserPasswordChange:
  from_email: "noreply@yourcompany.com"
  from_name: "Your Company Security"
  template_id: 5977513
  subject: "Password Change Notification"
```

When Sent:

- User changes password from profile/settings
- Admin resets user password

Template Variables Available:

- `{{ var:user_name }}`
- `{{ var:change_date }}`
- `{{ var:changed_by }}` - "Self" or admin name
- `{{ var:support_email }}`

Email Verification

Sent to verify a user's email address.

```yaml
api_crmCommunicationEmailVerification:
  from_email: "noreply@yourcompany.com"
  from_name: "Your Company"
  template_id: 5977514
  subject: "Verify Your Email Address"
```

When Sent:

- New account created (customer or staff)
- User changes email address

- Email verification required for security

Template Variables Available:

- `{{ var:user_name }}`
- `{{ var:verification_url }}` - Link to verify email
- `{{ var:verification_code }}` - Code to enter manually (alternative to link)

Balance Expired Notification

Sent when a customer's service balance/allowance has expired.

```yaml
api_crmCommunicationsBalanceExpired:
  from_email: "support@yourcompany.com"
  from_name: "Your Company Support"
  template_id: 5977515
  subject: "Your Service Balance Has Expired"
```

When Sent:

- Prepaid balance expires
- Monthly allowance renewal date passed
- Service expiry date reached

Template Variables Available:

- `{{ var:customer_name }}`
- `{{ var:service_name }}` - Name of expired service
- `{{ var:expiry_date }}`
- `{{ var:balance_type }}` - "Data", "Voice", "Monetary", etc.
- `{{ var:renewal_url }}` - Link to renew/top-up

Low Balance Alert

Sent when a customer's balance falls below configured threshold.

```yaml
api_crmCommunicationsBalanceLow:
  from_email: "support@yourcompany.com"
  from_name: "Your Company Support"
  template_id: 5977516
  subject: "Low Balance Alert"
```

When Sent:

- Balance drops below threshold (e.g., 20% remaining)
- Configured in service plan or OCS
- Real-time monitoring triggers alert

Template Variables Available:

- `{{ var:customer_name }}`
- `{{ var:service_name }}`
- `{{ var:current_balance }}`
- `{{ var:threshold }}`
- `{{ var:balance_type }}`
- `{{ var:topup_url }}` - Link to add balance

Creating Mailjet Email Templates

For each email type, you need to create a corresponding template in Mailjet.

Step 1: Create Template in Mailjet

1. Login to Mailjet dashboard
2. Navigate to **Email Templates → Transactional Templates**
3. Click **Create a New Template**
4. Choose **Code your own template** (for advanced users) or **Use template builder**

Step 2: Design Template

Use Mailjet's drag-and-drop builder or HTML editor to design your email.

Essential Elements:

- **Header** - Company logo and branding
- **Greeting** - Personalized with `{{ var:customer_name }}` or `{{ var:user_name }}`
- **Content** - Main message body
- **Variables** - Insert template variables from list above
- **Call to Action** - Buttons/links for user actions
- **Footer** - Unsubscribe link, company address, support contact

Example Template (Password Reset):

```
<!DOCTYPE html>
<html>
<head>
    <style>
        body { font-family: Arial, sans-serif; }
        .button { background-color: #4CAF50; color: white; padding: 14px 28px; }
    </style>
</head>
<body>
    <img src="https://yourcompany.com/logo.png" alt="Logo"
```

```html
width="200">

    <h2>Password Reset Request</h2>

    <p>Hello {{ var:user_name }},</p>

    <p>We received a request to reset your password. Click the button below to create a new password:</p>

    <a href="{{ var:reset_url }}" class="button">Reset Password</a>

    <p>This link expires in {{ var:expiry_time }}.</p>

    <p>If you didn't request this, please ignore this email.</p>

    <hr>
    <p style="font-size: 12px; color: #666;">
        Your Company | support@yourcompany.com<br/>
        123 Business St, City, Country
    </p>
</body>
</html>
```

Step 3: Get Template ID

1. Save template in Mailjet
2. Note the **Template ID** (numeric, e.g., `5977509`)
3. Copy this ID to `crm_config.yaml`

Step 4: Test Template

1. In Mailjet, use **Test Email** feature
2. Provide sample values for all variables
3. Send test email to yourself
4. Verify formatting, links, and branding

Step 5: Configure in OmniCRM

Add template configuration to `crm_config.yaml`:

```yaml
mailjet:
  api_key: your_api_key
  api_secret: your_secret

  api_crmCommunicationUserPasswordReset:
    from_email: "noreply@yourcompany.com"
    from_name: "Your Company Security"
    template_id: 5977511
```

```yaml
    subject: "Password Reset Request"
```

Restart OmniCRM API for changes to take effect:

```
cd OmniCRM-API
sudo systemctl restart omnicrm-api
```

Contact Syncing

All customer contacts in OmniCRM are automatically synced to Mailjet.

What Gets Synced:

- Contact name
- Email address
- Contact type (billing, technical, etc.)
- Customer location
- Custom fields

Sync Frequency:

Contacts sync in real-time when:

- New customer created
- Contact added/updated
- Customer details modified

Troubleshooting

Email not sending

- **Cause:** Invalid API credentials, Mailjet account suspended, or template ID wrong
- **Fix:**
    - Verify `api_key` and `api_secret` in `crm_config.yaml`
    - Check Mailjet account status and billing
    - Verify template ID exists in Mailjet
    - Check API logs for errors

Template variables not substituting

- **Cause:** Variable name mismatch or missing data in OmniCRM
- **Fix:**
    - Verify variable names match exactly (case-sensitive)
    - Use `{{ var:variable_name }}` format
    - Check OmniCRM is passing variable data in API call
    - Test with sample data in Mailjet

Invoice PDF not attaching

- **Cause:** PDF generation failed or file size too large
- **Fix:**
    - Check invoice generation logs
    - Verify invoice template renders correctly
    - Ensure PDF under 15MB (Mailjet limit)
    - Test invoice PDF generation separately

Contacts not syncing to Mailjet

- **Cause:** API rate limit exceeded or sync service not running
- **Fix:**
    - Check Mailjet API rate limits (200 calls/minute)
    - Verify OmniCRM-API service is running
    - Review sync logs for errors
    - Manually trigger sync for testing

Related Documentation

- `administration_configuration` - Complete Mailjet configuration reference
- `payments_invoices` - Invoice generation and email delivery
- `authentication_flows` - Password reset and verification emails
- `customer_care` - Self-Care portal welcome emails

Further Reading

- Mailjet Documentation: <https://dev.mailjet.com/>
- Mailjet API Reference: <https://dev.mailjet.com/email/reference/>

Changelog

This contains the last 50 changes made to the OmniCRM software stack or it's dependencies.

Note: This does not track changes to inididual customer setups.

Payment Methods Management

OmniCRM's Payment Methods system allows customers and staff to securely manage payment cards using **multi-vendor payment processing** (Stripe, PayPal, etc.). Payment methods enable automatic billing for services, one-time payments, and recurring charges without storing sensitive card data in OmniCRM.

See also: Payment Vendor Integrations <integrations_payment_vendors>, Payment Processing Architecture <payment_processing_architecture>, Billing Overview <billing_overview>, Payment Processing <payments_process>, Invoices <payments_invoices>.

Overview

The payment methods system provides:

- **Secure Card Storage** - Cards tokenized by payment vendors (Stripe, PayPal), never stored in OmniCRM
- **Multi-Vendor Support** - Stripe and PayPal payment methods supported
- **Multiple Cards** - Customers can store multiple payment methods
- **Default Selection** - Designate preferred payment method for automatic charges
- **Expiry Tracking** - Monitor and update expiring cards
- **Self-Service** - Customers can manage their own cards via Self-Care Portal <self_care_portal>
- **Staff Management** - Support staff can add/remove cards on behalf of customers

Supported Payment Methods:

- **Cards** (via Stripe or PayPal)
  - Credit Cards (Visa, Mastercard, American Express, Discover)
  - Debit Cards
  - Prepaid Cards (if supported by card network)
- **PayPal Accounts** (via PayPal integration)

Not Stored in OmniCRM:

Card details are tokenized by payment vendors and stored securely. OmniCRM only stores:

- Payment vendor (stripe, paypal)
- Card brand (Visa, Mastercard, etc.)
- Last 4 digits
- Expiry month/year
- Cardholder name/nickname
- Vendor-specific payment method token

Accessing Payment Methods

From Customer Page:

1. Navigate to **Customers → [Select Customer]**
2. Click **Billing** tab
3. Scroll to **Payment Methods** section

Or directly:

From Expiring Cards Dashboard:

View all customers with expiring cards:

This shows a system-wide list of cards expiring within the next 60 days.

Payment Methods List

The payment methods table displays all stored cards for a customer:

Payment Methods List

Column Descriptions:

- **Nickname** - Friendly name for the card (e.g., "Personal Card", "Work Visa")
- **Issuer** - Card brand and last 4 digits
- **Expiry** - Expiration month/year (MM/YYYY format)
- **Added** - Date card was added to account
- **Default** - Checkmark indicates default payment method for automatic charges

Actions Per Card:

Each row has an actions menu (⋮) with options:

- **Set as Default** - Make this the default payment method
- **Delete** - Remove card from account

Adding a Payment Method

Click **"Add Payment Method"** to open the secure payment modal.

Step 1: Enter Card Details

The secure payment form appears (powered by Stripe Elements or PayPal SDK):

Add Payment Method Modal

Required Fields:

- **Card Information** - Card number, expiry, CVC (validated by Stripe)
- **Cardholder Name** - Name on the card
- **Country/Region** - Billing country

Optional Fields:

- **Card Nickname** - Friendly label to distinguish between cards

Security:

- Card details entered directly into vendor-hosted secure iframe (Stripe Elements / PayPal SDK)
- OmniCRM never sees or stores full card numbers
- PCI DSS compliance handled by payment vendor
- Real-time validation prevents invalid card numbers

Step 2: Submit and Tokenize

When you click **"Add Payment Method"**:

1. **Client-Side Validation:**
    - Payment vendor validates card number format
    - Checks expiry date is in the future
    - Verifies CVC format
2. **Tokenization:**
    - Card details sent directly to payment vendor (not OmniCRM)
    - Vendor creates a secure token (e.g., `pm_1A2B3C4D` for Stripe)
    - Token returned to OmniCRM
3. **Server Processing:**
    - OmniCRM saves token to customer record with vendor identifier
    - Stores last 4 digits, brand, expiry, and vendor name for display
    - No full card number ever touches OmniCRM servers

Step 3: Confirmation

Success message appears:

> Your Visa ending in 1234 has been added to your account.

The new card appears in the payment methods table.

Automatic Default Selection:

- If this is the customer's first card, it's automatically set as default
- If customer already has cards, new card is added as non-default
- Customer can change default after adding

Setting Default Payment Method

The default payment method is used for:

- Automatic recurring service charges
- Invoice payments
- Top-ups and recharges
- One-time transactions (unless specified otherwise)

To Change Default:

1. Locate the card you want to set as default in the payment methods table

2. Click the **actions menu (⋮)** next to the card

3. Select **"Set as Default"**

4. Confirmation appears

    > Visa ending in 5678 is now your default payment method.

The checkmark moves to the newly selected card.

Visual Indicator:

Default cards show:

in the Default column, typically with a green checkmark badge.

Deleting a Payment Method

Remove cards that are expired, lost, or no longer needed.

Step 1: Initiate Deletion

1. Find the card to delete in the payment methods table
2. Click the **actions menu (⋮)**
3. Select **"Delete"**

Step 2: Confirm Deletion

A confirmation modal appears:

> Are you sure you want to delete this payment method?
>
> Card: Visa ending in 1234 Expiry: 12/2026
>
> ⚠ Warning: If this is your only payment method, you will need to add a new one to continue using services that require automatic billing.
>
> [Cancel] [Delete Payment Method]

Click **"Delete Payment Method"** to confirm.

Step 3: Deletion Complete

Success message:

The card is removed from the table and deleted from the payment vendor.

Important Restrictions:

- **Cannot delete default if other cards exist** - Set a different card as default first
- **Warning if deleting last card** - Services requiring payment may be suspended
- **No undo** - Deletion is permanent; customer must re-add card if needed

Managing Expiring Cards

OmniCRM tracks card expiry dates and provides tools to proactively update expiring cards.

Expiring Cards Dashboard

Navigate to **Billing → Expiring Cards** to see a system-wide list:

> Customer Card Expiry Days Until Action John Smith Visa ****1234 02/2025 12 days Update Acme Corp MC5678 03/2025 45 days Update Jane Doe Amex****9012 01/2025 EXPIRED Update**

Expiring Cards Dashboard

Filters:

- **Expiry Range** - Next 30/60/90 days or already expired

- **Customer Type** - Individual vs Business
- **Service Type** - Filter by service requiring payment method

Actions:

- **Update** - Opens customer's payment methods page to add new card
- **Notify** - Send email reminder to customer (if Mailjet configured)

Expiry Notifications

If Mailjet is configured, automatic emails are sent:

- **60 days before expiry** - First reminder
- **30 days before expiry** - Second reminder
- **7 days before expiry** - Final warning
- **At expiry** - Card has expired notice

Customers can click a link in the email to update their payment method via the Self-Care portal.

Email Template Variables:

Mailjet templates receive:

- Customer name
- Card brand and last 4 digits
- Expiry date
- Link to Self-Care payment methods page

See `integrations_mailjet` for email template configuration.

Updating an Expiring Card

Recommended Workflow:

1. Customer receives expiry notification email
2. Customer logs into Self-Care portal
3. Navigates to **Billing → Payment Methods**
4. Clicks **"Add Payment Method"**
5. Enters new card details (same card with updated expiry, or replacement card)
6. Sets new card as default
7. Deletes old/expired card

Staff Workflow:

If customer calls support:

1. Staff opens customer account
2. Navigates to **Billing → Payment Methods**
3. Adds new card on customer's behalf (customer provides details over phone)
4. Sets new card as default
5. Deletes expired card
6. Confirms with customer

Warning

Never ask customers to email or text card details. Always use:

- Secure Self-Care portal for self-service

- Phone with staff entering details directly into system
- In-person at retail location

What Happens When Cards Expire

When a payment card reaches its expiry date and is not updated:

Immediate Effects:

1. **Automatic Payments Fail**
    - Payment vendor rejects transactions with expired cards
    - Monthly service renewals fail to process
    - Auto-top-ups fail
    - Invoice auto-payments fail
2. **Customer Notifications**
    - System attempts to charge card
    - Payment failure notification sent
    - "Update Payment Method" email sent with link to Self-Care portal
3. **Service Status Changes**
    - **Postpaid Services** - May continue temporarily with outstanding balance
    - **Prepaid Services** - Service suspension when balance depletes
    - **Auto-Renew Services** - Renewal fails, service may expire

Subsequent Actions:

Day 1-3 (Grace Period):

- Service continues normally
- Customer receives first payment failure notice
- System attempts retry (depending on configuration)

Day 4-7:

- Second payment attempt (if configured)
- Warning email sent
- Customer service may contact customer

Day 8-14:

- Service may be suspended for non-payment
- Suspended status prevents usage but preserves account
- Customer can restore by updating payment method and paying outstanding balance

Day 15+:

- Service may be terminated for non-payment
- Inventory (SIM cards, equipment) marked for return
- Final notice sent
- Account referred to collections (if applicable)

Preventing Service Interruption:

To avoid service disruption:

- Update cards **30 days before expiry**
- Add multiple payment methods for redundancy
- Enable payment failure alerts
- Monitor Expiring Cards dashboard weekly

Restoring Service After Expiry:

If service suspended due to expired card:

1. Add new valid payment method
2. Set as default
3. Pay outstanding balance (if any)
4. Contact support to reactivate service
5. Service restored within minutes to hours

Payment Method Security

Tokenization

OmniCRM uses vendor tokenization to ensure security:

1. **Customer enters card** → Sent directly to payment vendor servers
2. **Vendor validates and tokenizes** → Creates unique token
3. **Token stored in OmniCRM** → Full card number never stored
4. **Payment processing** → Token sent to vendor, vendor charges card

What OmniCRM Stores:

```json
{
  "vendor": "stripe",
  "vendor_payment_method_id": "pm_1A2B3C4D5E6F",
  "payment_type": "card",
  "brand": "visa",
  "last4": "1234",
  "exp_month": 12,
  "exp_year": 2026,
  "name": "John Smith",
  "nickname": "Personal Card",
  "is_default": true
}
```

What OmniCRM Does NOT Store:

- Full card number
- CVV/CVC code
- Magnetic stripe data
- PIN numbers

PCI Compliance

By using vendor-hosted payment forms:

- **Reduced PCI scope** - Card data never touches OmniCRM servers
- **Vendor-hosted fields** - Card entry happens in vendor's secure iframe
- **No card storage** - Tokens used instead of raw card data
- **Secure transmission** - All communication over HTTPS/TLS

See Payment Vendor Integrations <integrations_payment_vendors> for vendor-specific security details.

Common Workflows

Workflow 1: Customer Adds First Payment Method

Scenario: New customer signing up for service

1. Customer creates account
2. Selects service plan
3. Prompted to add payment method during checkout
4. Enters card details in Stripe modal
5. Card tokenized and saved
6. Automatically set as default
7. Service provisioned
8. First charge processed

Workflow 2: Customer Updates Expiring Card

Scenario: Credit card about to expire

1. Customer receives email notification (60 days before expiry)
2. Logs into Self-Care portal
3. Navigates to **Billing → Payment Methods**
4. Reviews current card expiring 12/2025
5. Clicks **"Add Payment Method"**
6. Enters replacement card with expiry 12/2028
7. Sets new card as default
8. Deletes old card
9. Confirmation email sent

Workflow 3: Staff Helps Customer Over Phone

Scenario: Customer calls: "My card was declined"

1. Customer calls support
2. Staff verifies identity (security questions)
3. Staff checks payment methods: Card expired 01/2025
4. Staff: "Your card has expired. Do you have a new card?"
5. Customer provides new card details over phone
6. Staff navigates to **Customers → [Customer] → Billing**
7. Clicks **"Add Payment Method"**
8. Enters card details as customer reads them
9. Sets new card as default
10. Deletes expired card
11. Retries failed payment
12. Confirms with customer: "Payment successful, service restored"

Workflow 4: Business Customer with Multiple Cards

Scenario: Company wants different cards for different purposes

1. Business customer adds primary card (Visa ending 1111)
2. Sets as default for monthly service charges
3. Adds backup card (Mastercard ending 2222) for top-ups
4. Adds purchasing card (Amex ending 3333) for equipment purchases
5. When making top-up, selects Mastercard manually at checkout
6. Default Visa still used for automatic monthly billing

Workflow 5: Managing Expiring Cards (Admin)

Scenario: Proactive expiry management

1. Admin navigates to **Billing → Expiring Cards**
2. Filters: "Next 30 days"
3. Sees 15 customers with expiring cards
4. Selects all → **"Send Reminder Emails"**
5. Mailjet sends personalized emails to each customer
6. Customers update cards via Self-Care
7. Admin reviews list 1 week later
8. Calls remaining customers who haven't updated
9. Assists with card updates over phone

Troubleshooting

"Card declined" when adding payment method

- **Cause:** Payment vendor rejected card (insufficient funds, fraud prevention, issuer decline)
- **Fix:**
  - Try a different card
  - Contact card issuer to authorize transaction
  - Ensure card supports online purchases
  - Check billing address matches card on file

"Error adding payment method" (generic error)

- **Cause:** Payment vendor API error or network issue
- **Fix:**
  - Refresh page and try again
  - Check internet connection
  - Verify payment vendor configuration is correct in system settings
  - Check browser console for specific error message
  - Try different browser (disable ad blockers)

Cannot delete payment method (button disabled)

- **Cause:** Trying to delete the default card, or it's the only card
- **Fix:**
  - Set a different card as default first
  - If it's the only card, add a new card before deleting

Card shows as expired but not in "Expiring Cards" list

- **Cause:** Card expired recently, cache not refreshed
- **Fix:**
  - Refresh the page
  - Check filters on Expiring Cards dashboard
  - Expired cards may move to different view

New card not appearing immediately

- **Cause:** Page hasn't refreshed after adding card
- **Fix:**
  - Payment methods table should auto-refresh
  - If not, manually refresh browser
  - Check if error occurred during add process

Payment modal won't load

- **Cause:** Payment vendor SDK not loading, API key issue, or browser extension blocking
- **Fix:**
  - Check browser console for errors
  - Disable ad blockers and tracking protection
  - Verify payment vendor configuration in system settings
  - Ensure vendor SDK script loads (check Network tab)
  - Try incognito/private browsing mode

Customer doesn't receive expiry notifications

- **Cause:** Mailjet not configured or email template missing
- **Fix:**
  - Verify Mailjet credentials in crm_config.yaml
  - Check email template exists for card expiry
  - Confirm customer email address is valid
  - Check Mailjet logs for delivery failures

Best Practices

For Customers:

- Add payment method before service activation to avoid delays
- Keep at least 2 cards on file for redundancy
- Update expiring cards 30+ days before expiry
- Delete old/expired cards to avoid confusion
- Use descriptive nicknames ("Personal Visa", "Work Amex")
- Verify default payment method is correct for automatic billing

For Support Staff:

- Verify customer identity before accessing payment methods
- Never ask customers to send card details via email/SMS/chat
- Process card additions immediately during calls (don't defer)
- Confirm new card is set as default after adding
- Delete old cards only after confirming new card works
- Test payment after updating expired card (process £0.01 authorization)

For Administrators:

- Monitor Expiring Cards dashboard weekly
- Send reminder emails 60/30/7 days before expiry
- Keep payment vendor test/live keys separate for dev vs production
- Ensure Mailjet templates are configured for expiry notifications
- Review failed payment reports to identify expired cards
- Train staff on secure card handling procedures

Security Best Practices:

- Only use public/publishable keys in frontend (never secret keys)
- Ensure all payment pages load over HTTPS
- Regularly review payment vendor dashboards for suspicious activity
- Enable vendor fraud prevention rules (Stripe Radar, PayPal risk checks)
- Require CVC for all card-not-present transactions
- Log payment method changes in activity log

Related Documentation

- `Payment Vendor Integrations <integrations_payment_vendors>` - Multi-vendor payment configuration
- `Payment Processing Architecture <payment_processing_architecture>` - Complete architecture details
- `payments_process` - Processing payments with stored payment methods
- `payments_invoices` - Automatic invoice payment using default card
- `features_topup_recharge` - Top-up system using payment methods
- `basics_payment` - General payment and billing concepts
- `customer_care` - Self-Care portal for customers to manage their own cards

Payment Methods Management Interface

Customer Invoices

Transactions </payments_transaction> are grouped together to form an invoice, which is sent to the customer for payment.

Invoices have a start and end date, which is the period the invoice covers, and a due date, which is the date the invoice is due for payment.

Generate a Proforma Invoice

Invoices can be automatically generated by the system, for example, when a service is billed, an invoice is created for the retail cost, or they can be manually created, for example, if a customer requests a copy of an invoice, or if a customer is billed for a one-time charge.

Customer invoices are fully templated with Mailjet <integrations_mailjet> and can be customized to include the company logo, address, and payment details, and can be sent to the customer via email, or downloaded as a PDF.

Activity Log

Customizing Invoice Templates

OmniCRM uses HTML templates with Jinja2 templating to generate invoices. You can fully customize the invoice design, branding, colors, and layout.

Invoice Template Location

Invoice templates are stored in `OmniCRM-API/invoice_templates/`

Default Templates:

- `norfone_invoice_template.html` - Sample invoice template
- `cifi_invoice_template.html` - Alternative template example

Configuration:

The active invoice template is specified in `OmniCRM-API/crm_config.yaml`:

```yaml
invoice:
  template_filename: 'norfone_invoice_template.html'
```

Available Template Variables

Invoice templates have access to the following Jinja2 variables:

Invoice Information:

- `{{ invoice_number }}` - Unique invoice ID (e.g., `INV-2025-001234`)
- `{{ date }}` - Invoice issue date (ISO format: `2025-01-10T12:00:00`)
- `{{ due_date }}` - Payment due date (e.g., `2025-02-10`)
- `{{ start_date }}` - Billing period start date
- `{{ end_date }}` - Billing period end date
- `{{ total_amount }}` - Total invoice amount before tax (numeric)

- `{{ total_tax }}` - Total tax amount calculated from all transactions (numeric)

Customer Information:

- `{{ client.name }}` - Customer's full name or company name
- `{{ client.address.address_line_1 }}` - Address line 1
- `{{ client.address.address_line_2 }}` - Address line 2
- `{{ client.address.city }}` - City
- `{{ client.address.state }}` - State/province
- `{{ client.address.zip_code }}` - Postal/ZIP code
- `{{ client.address.country }}` - Country

Transaction Line Items:

Loop through transactions using:

```
{% for sub_transaction in transactions %}
    <tr>
        <td>{{ sub_transaction.transaction_id }}</td>
        <td>{{ sub_transaction.created.split("T")[0] }}</td>
        <td>{{ sub_transaction.title }}</td>
        <td>{{ sub_transaction.description }}</td>
        <td>${{ "%.2f"|format(sub_transaction.retail_cost) }}</td>
    </tr>
{% endfor %}
```

Transaction Fields:

- `sub_transaction.transaction_id` - Transaction ID
- `sub_transaction.created` - Transaction date/time
- `sub_transaction.title` - Transaction title
- `sub_transaction.description` - Detailed description
- `sub_transaction.retail_cost` - Line item amount
- `sub_transaction.tax_percentage` - Tax percentage applied (e.g., 10 for 10%)
- `sub_transaction.tax_amount` - Calculated tax amount in dollars

Displaying Tax in Templates:

```
<td>
    {% if sub_transaction.tax_amount and sub_transaction.tax_amount > 0 %}
        ${{ "%.2f"|format(sub_transaction.tax_amount) }} ({{
sub_transaction.tax_percentage }}%)
    {% else %}
        -
    {% endif %}
</td>
```

Creating a Custom Invoice Template

Step 1: Copy Existing Template

```
cd OmniCRM-API/invoice_templates/
cp norfone_invoice_template.html your_company_invoice_template.html
```

Step 2: Customize HTML/CSS

Edit `your_company_invoice_template.html` to match your branding:

Key Customization Areas:

1. **Company Logo and Branding**

```html
<!-- Replace with your logo URL -->
<img src="https://yourcompany.com/logo.png" alt="Your Company" width="200">

<!-- Update company name -->
<h1>Your Company Name</h1>
```

2. **Color Scheme**

```html
<style>
    /* Primary brand color */
    .navbar {
        background: linear-gradient(to bottom right, #your-color-1, #your-
color-2);
    }

    /* Table headers */
    .table thead th {
        background-color: #your-brand-color !important;
        color: white !important;
    }

    /* Buttons and links */
    .btn-primary {
        background-color: #your-brand-color;
    }
</style>
```

3. **Company Information Footer**

```html
<footer>
    <p>Your Company Name</p>
    <p>123 Business Street, City, Country</p>
    <p>Phone: +1-555-123-4567 | Email: billing@yourcompany.com</p>
    <p>ABN/Tax ID: 12345678900</p>
</footer>
```

4. **Payment Instructions**

```html
<div class="payment-info">
    <h3>Payment Methods</h3>
    <p><strong>Online:</strong> Pay at https://yourcompany.com/pay</p>
    <p><strong>Bank Transfer:</strong></p>
    <ul>
        <li>Account Name: Your Company Ltd</li>
        <li>BSB: 123-456</li>
        <li>Account Number: 987654321</li>
        <li>Reference: {{ invoice_number }}</li>
    </ul>
</div>
```

5. **Terms and Conditions**

```html
<div class="terms">
    <h4>Payment Terms</h4>
```

```
    <p>Payment due within 30 days of invoice date.</p>
    <p>Late payment fees: 2% per month on overdue balances.</p>
    <p>For billing inquiries: billing@yourcompany.com</p>
</div>
```

Step 3: Update Configuration

Edit `OmniCRM-API/crm_config.yaml`:

```yaml
invoice:
  template_filename: 'your_company_invoice_template.html'
```

Step 4: Restart API

```
cd OmniCRM-API
sudo systemctl restart omnicrm-api
```

Step 5: Test Invoice Generation

1. Navigate to a customer with transactions
2. Generate a test invoice
3. Download PDF to verify formatting
4. Email invoice to yourself to test email delivery

Advanced Customization

Conditional Content:

Use Jinja2 conditionals to show/hide content:

```
{% if total_amount > 1000 %}
    <div class="high-value-notice">
        <p><strong>Note:</strong> Large balance - Payment plan available upon request.</p>
    </div>
{% endif %}

{% if client.address.country == "Australia" %}
    <p>GST Included: ${{ "%.2f"|format(total_amount * 0.10) }}</p>
{% endif %}
```

Multi-Language Support:

Create language-specific templates:

```
invoice_template_en.html
invoice_template_es.html
invoice_template_fr.html
```

Configure based on customer's language preference.

Custom Calculations:

```
<!-- Display subtotal and tax breakdown -->
<tr>
    <td colspan="4" class="text-right"><strong>Subtotal:</strong></td>
    <td>${{ "%.2f"|format(total_amount) }}</td>
</tr>
```

```
<tr>
    <td colspan="4" class="text-right"><strong>Tax:</strong></td>
    <td>${{ "%.2f"|format(total_tax) }}</td>
</tr>
<tr>
    <td colspan="4" class="text-right"><strong>Total:</strong></td>
    <td>${{ "%.2f"|format(total_amount + total_tax) }}</td>
</tr>
```

Note: The `total_tax` variable is automatically calculated by summing the `tax_amount` from all transactions in the invoice. Each transaction's tax is calculated based on its `tax_percentage` field, which defaults to the product's `tax_percentage` or 0% if not specified.

QR Code for Payment:

Generate QR code for mobile payment:

```
<div class="qr-payment">
    <img src="https://api.qrserver.com/v1/create-qr-code/?size=150x150&data={{ payment_url }}"
         alt="Scan to Pay">
    <p>Scan with your phone to pay instantly</p>
</div>
```

PDF Styling Best Practices

OmniCRM uses **WeasyPrint** to convert HTML to PDF. Follow these guidelines:

Supported CSS:

- Most CSS 2.1 properties
- Limited CSS3 (flexbox, some transforms)
- Web fonts via `@font-face`

Not Supported:

- JavaScript
- CSS Grid (use tables instead)
- Complex animations
- Some modern CSS properties

Page Size and Margins:

```
@page {
    size: A4;
    margin: 1cm;
}

body {
    font-family: Arial, sans-serif;
    font-size: 10pt;
}
```

Print-Specific Styling:

```
@media print {
    .no-print {
        display: none;
```

```
    }

    .page-break {
        page-break-after: always;
    }
}
```

Table Layout:

```
.table {
    table-layout: fixed;
    width: 100%;
}

.table th, .table td {
    word-wrap: break-word;
    padding: 4px;
}
```

Font Embedding:

For custom fonts, use web-safe fonts or embed:

```
@font-face {
    font-family: 'YourFont';
    src: url('https://yourcompany.com/fonts/yourfont.woff2') format('woff2');
}

body {
    font-family: 'YourFont', Arial, sans-serif;
}
```

Testing Invoice Templates

Test Checklist:

1. **Visual Inspection:**
    - Logo displays correctly
    - Colors match brand guidelines
    - Text is readable (not too small)
    - Tables align properly
    - All sections present
2. **Data Accuracy:**
    - Customer details correct
    - Transaction amounts sum correctly
    - Dates formatted properly
    - All variables substituting correctly
3. **PDF Quality:**
    - File size reasonable (<5MB)
    - Images sharp and clear
    - No text cutoff or overflow
    - Page breaks in appropriate places
4. **Multi-Page Invoices:**
    - Headers repeat on each page
    - Page numbers display
    - Long transaction lists paginate correctly
5. **Email Delivery:**
    - PDF attaches to email

    - File size under Mailjet limit (15MB)
    - Renders in Gmail, Outlook, Apple Mail

Test Command (Manual Generation):

You can test invoice generation via API:

```
curl -X GET "http://localhost:5000/crm/invoice/{invoice_id}/pdf" \
     -H "Authorization: Bearer YOUR_TOKEN" \
     --output test_invoice.pdf
```

Common Template Issues

Variables not substituting:

- **Cause:** Typo in variable name or missing data
- **Fix:** Check spelling exactly (case-sensitive), verify data exists in database

PDF styling broken:

- **Cause:** Unsupported CSS property
- **Fix:** Use CSS 2.1 properties, test with WeasyPrint-compatible CSS

Images not showing:

- **Cause:** Relative URLs or blocked external resources
- **Fix:** Use absolute HTTPS URLs, ensure images publicly accessible

Tables overflowing page:

- **Cause:** Fixed column widths too wide
- **Fix:** Use percentage widths, `table-layout: fixed`

Fonts not rendering:

- **Cause:** Font not embedded or unavailable
- **Fix:** Use web-safe fonts (Arial, Times New Roman, etc.) or properly embed custom fonts

PDF generation fails:

- **Cause:** HTML syntax errors or WeasyPrint crash
- **Fix:** Validate HTML, check WeasyPrint logs, simplify complex layouts

Invoice PDF Caching

To improve performance and reduce redundant PDF generation, OmniCRM includes an Invoice PDF caching system. When an invoice PDF is first generated, it is cached in the database for subsequent requests.

How PDF Caching Works:

1. **First Request** - When an invoice PDF is requested (download or email), the system:
    - Generates the PDF from the invoice template
    - Encodes the PDF as Base64
    - Calculates a SHA256 hash of the PDF content
    - Stores in `Invoice_PDF_Cache` table with:
        - Invoice ID reference

        - PDF data (Base64-encoded)
        - Filename
        - Content hash (for integrity verification)
        - Creation timestamp
2. **Subsequent Requests** - When the same invoice is requested again:
    - System checks for cached PDF by invoice_id
    - If cache exists and is valid, returns cached PDF immediately
    - Updates `last_accessed` timestamp to track cache usage
3. **Cache Invalidation** - Cached PDFs are invalidated when:
    - Invoice is modified (transactions added/removed, details changed)
    - Invoice template is updated
    - Manual cache clearing is triggered

Benefits:

- **Performance** - Instant PDF delivery for repeat requests (no regeneration delay)
- **Consistency** - Same PDF for all downloads of an invoice (unless invoice is modified)
- **Server Load** - Reduces CPU usage from PDF generation
- **User Experience** - Loading indicator appears during initial generation, subsequent requests are instant

Cache Management:

The Invoice PDF Cache is automatically managed by the system. Old or unused cache entries can be purged periodically based on:

- Age (e.g., remove cache entries older than 90 days)
- Access patterns (remove entries not accessed in 30 days)
- Storage limits (implement cache size limits if needed)

API Behavior:

When downloading an invoice via API or UI:

- First request: Shows loading indicator while PDF generates, then caches
- Subsequent requests: Immediate download from cache
- Cache hit/miss is transparent to the user

Important: When you update your invoice template, clear the cache to ensure new invoices use the updated design:

```sql
-- Clear all cached invoice PDFs (run in MySQL)
DELETE FROM Invoice_PDF_Cache;
```

Or update `crm_config.yaml` to automatically invalidate cache on template change.

Accessing Invoices

Invoices can be viewed at the system level or per-customer:

Per-Customer View:

1. Navigate to **Customers → [Select Customer]**
2. Click **Billing** tab
3. View invoices list in the third card

System-Wide View:

1. Navigate to **Billing → Invoices** (from main menu)
2. View all invoices across all customers

Invoice Statistics Widgets

At the top of the invoices page, four statistics cards display financial summaries.

Invoice Statistics and List

Widget Descriptions:

- **Total Invoices** - Sum of all invoice retail costs (all time) and count of invoices sent
- **Unpaid Invoices** - Sum of invoices not yet paid and count of unpaid invoices
- **Invoices This Month** - Sum of invoices created this calendar month with count
- **Invoices Last Month** - Sum of invoices created last calendar month with count

Value Formatting:

- Values over 1,000: Display as "k" suffix (e.g., $1.5k)
- Values over 1,000,000: Display as "M" suffix (e.g., $2.3M)
- Values over 1,000,000,000: Display as "B" suffix (e.g., $1.1B)

Trend Indicators:

- Widgets for "This Month" and "Last Month" show percentage change
- Green arrow up: Increase from previous period
- Red arrow down: Decrease from previous period
- Gray arrow right: No change

Invoices List

The invoices table displays all invoices with the following columns:

Global Invoices List

Column Descriptions:

- **ID** - Unique invoice ID
- **Title** - Invoice title/description
- **Period** - Billing period (start date - end date) or "N/A" for one-time invoices
- **Due Date** - Payment due date
- **Created** - Invoice creation date
- **Amount** - Total invoice amount (retail cost)
- **Status** - Paid, Unpaid, or Refunded
- **Actions** - Available actions (varies by status)

Action Icons:

- ⬇ **(Download)** - Download invoice PDF
- ⯑ **(Delete)** - Void invoice (only if not paid)
- ⯑ **(Pay)** - Pay invoice online (only if unpaid)
- ✉ **(Email)** - Send invoice email to customer
- ⯑ **(Refund)** - Refund Stripe payment (only for paid Stripe invoices)

Generating an Invoice

Click **"+ Generate Proforma Invoice"** to create a new invoice.

Generate Invoice Modal with Transaction Preview

Field Descriptions:

- **Search Customers** - Select customer (only shown in system-wide view, pre-filled in customer view)
- **Title** - Invoice title/name (optional, defaults to "Invoice for [Period]")
- **Start Date** - Beginning of billing period (defaults to 14 days ago)
- **End Date** - End of billing period (defaults to today)
- **Due Date** - Payment deadline (defaults to today)
- **Transaction Preview** - Shows all uninvoiced transactions in date range with ability to include/exclude specific transactions

Transaction Selection:

- **✓ (Green Plus)** - Click to exclude a transaction from the invoice
- **× (Red X)** - Click to include a previously excluded transaction
- **Select All** - Include all displayed transactions
- **Clear All** - Exclude all transactions
- Excluded transactions appear grayed out with strikethrough text
- Real-time totals update as you select/deselect transactions

What Happens:

1. System finds all uninvoiced transactions for customer within date range
2. Displays transaction preview with ability to include/exclude individual transactions
3. Shows real-time calculation of subtotal, tax, and total based on selected transactions
4. Only selected (included) transactions are added to the invoice
5. Generates invoice PDF and caches it
6. Marks selected transactions as invoiced (`invoice_id` field populated)
7. Excluded transactions remain uninvoiced and available for future invoices
8. Invoice appears in list with "Unpaid" status

Example Use Cases:

Monthly Billing: Set start date to first of month, end date to last day of month, preview shows all uninvoiced transactions from that period. Select all or manually exclude specific ones.

Service-Specific Invoice: Use same date range, then manually exclude unwanted transactions (e.g., exclude non-mobile transactions to create mobile-only invoice).

One-Time Invoice: Set both start and end date to the same day, preview shows only transactions from that date. Exclude any charges not relevant to this specific invoice.

Viewing Invoice Details

Click on any invoice row in the table to view full invoice details including all transactions, totals, and available actions.

Invoice Details View

Invoice Details Modal:

- **Invoice Information** - Shows invoice ID, title, dates, payment status, and void status
- **Transactions List** - Displays all transactions included in the invoice with:
  - Transaction date
  - Title and description
  - Retail cost
  - Tax amount and percentage (formatted as `$10.00 (10%)`)
  - Tax-exempt transactions show "-" in Tax column
- **Totals Summary** - Real-time calculation showing:
  - Transaction count
  - Subtotal (sum of all retail costs)
  - Tax (sum of all tax amounts)
  - Invoice Total (subtotal + tax)
- **Action Buttons** - Same actions available as in the table:
  - **Download PDF** - Download invoice PDF (always available)
  - **Send Email** - Email invoice to customer (non-voided invoices)
  - **Pay Invoice** - Process payment (unpaid, non-voided invoices only)
  - **Refund** - Refund Stripe payment (paid Stripe invoices only)
  - **Delete** - Void invoice (unpaid, non-voided invoices only)

Downloading Invoice PDFs

Click the **download icon (⬇)** in the table or **"Download PDF"** button in the invoice details modal to download an invoice as PDF.

Download Process:

1. Click download icon next to invoice
2. Loading spinner appears during generation (first time only)
3. Browser prompts to save file: `Invoice_01234.pdf`
4. PDF opens or saves to Downloads folder

PDF Caching Behavior:

- **First Download** - PDF generated from template, cached in database (may take 2-3 seconds)
- **Subsequent Downloads** - Instant download from cache
- **Cache Invalidation** - Cache cleared if invoice modified or template updated

Troubleshooting Download Issues:

- **Spinner never stops** - Check browser console, API may be down
- **PDF blank or corrupted** - Check invoice template for syntax errors
- **Download fails** - Check popup blocker settings, try different browser

Paying Invoices

Click the **pay icon (⛢)** to pay an invoice online.

Pay Invoice Modal

Payment Process:

1. Click pay icon on unpaid invoice
2. Payment modal opens showing invoice details
3. Select payment method:
   - **Stripe Transaction** - Charge saved credit card (available to all users)
   - **Cash** - Manual cash payment (staff only)

    - **Refund** - Apply refund as payment (staff only)
    - **POS Transaction** - Point-of-sale terminal (staff only)
    - **Bank Transfer** - Manual bank transfer (staff only)
4. If Stripe selected:
    - Select card from saved payment methods
    - Default card pre-selected
    - Click to select different card
5. If other method selected:
    - Enter reference number (optional)
6. Click **"Pay Invoice"** to process
7. System processes payment:
    - **Stripe** - Charges card via Stripe API
    - **Other methods** - Creates negative transaction for payment amount
8. Invoice status changes to "Paid"
9. Success notification displayed

Self-Care vs Staff Payment:

:doc:`Self-Care Portal <self_care_portal>` (Customers):

- Only Stripe payment available
- Must have saved payment method
- Warning shown if no payment methods exist
- Link to add payment method provided

Staff Portal (Admins):

- All payment methods available
- Can mark invoice paid manually (cash, POS, bank transfer)
- Can enter reference numbers for tracking

Payment Method Warning:

If customer has no saved payment methods, a warning is displayed prompting them to add a payment method before they can pay invoices.

Missing Payment Method Warning

Emailing Invoices

Click the **email icon (✉)** to send invoice to customer.

What Happens:

1. Click email icon next to invoice
2. System retrieves invoice PDF from cache (or generates if not cached)
3. Sends email via `Mailjet <integrations_mailjet>` using `api_crmCommunicationCustomerInvoice` template
4. Email includes:
    - Invoice PDF as attachment
    - Customer name
    - Invoice number and due date
    - Total amount due
    - Link to pay invoice online
    - Link to view/download invoice
5. Success notification: "Invoice email successfully sent"

Email Recipients:

Email sent to all customer contacts with type "billing" or primary contact if no billing contact exists.

Email Template Variables:

- `{{ var:customer_name }}` - Customer's full name
- `{{ var:invoice_number }}` - Invoice ID
- `{{ var:invoice_date }}` - Invoice issue date
- `{{ var:due_date }}` - Payment due date
- `{{ var:total_amount }}` - Total amount due
- `{{ var:invoice_url }}` - Link to view/download PDF
- `{{ var:pay_url }}` - Link to pay invoice online

Troubleshooting Email Issues:

- **Email not sent** - Check Mailjet API credentials in `crm_config.yaml`
- **Customer not receiving** - Verify customer contact email addresses
- **PDF not attaching** - Check PDF generation succeeded (try downloading first)

Voiding Invoices

Click the **delete icon (�)** to void an invoice.

Requirements:

- Invoice must be **Unpaid**
- Paid invoices cannot be voided (must be refunded instead)

How to Void:

1. Locate unpaid invoice in list
2. Click delete icon (�)
3. Confirm in modal:

Void Invoice Confirmation Modal

What Happens:

- Invoice marked as `void = true`
- All transactions unlinked from invoice (`invoice_id` set to null)
- Transactions become "uninvoiced" again
- Transactions can be included in new invoice
- Invoice appears in list with "Void:" prefix in title
- Invoice actions disabled (no download, pay, or email)
- Can be viewed by filtering for "Void" invoices

Important Notes:

- Voiding is NOT the same as refunding
- **Void** = "This invoice should never have existed" (billing error, duplicate)
- **Refund** = "Reverse a valid paid invoice" (return money to customer)

Refunding Invoices

Click the **refund icon (�)** to refund a paid invoice.

Requirements:

- Invoice must be **Paid**
- Invoice must be paid via **Stripe**
- Invoice must have a valid `payment_reference` (Stripe payment intent ID)
- Only available to staff users (not Self-Care)

How to Refund:

1. Locate paid Stripe invoice
2. Click refund icon (⯑)
3. Refund confirmation modal opens:

Refund Invoice Confirmation Modal

4. Click **"Confirm Refund"**
5. System processes Stripe refund:
    - Calls Stripe API to refund payment
    - Creates refund transaction in Stripe
    - Updates invoice with `refund_reference`
6. Invoice status changes to "Refunded"
7. Success notification displayed

What Happens After Refund:

- Invoice remains in system (not voided)
- Status shows "Refunded"
- Transactions remain linked to invoice
- Customer receives refund to original payment method (3-7 business days)
- Stripe dashboard shows refund transaction

Refund Restrictions:

- Cannot refund invoices paid via cash, POS, or bank transfer (manual reversal required)
- Cannot partial refund (full invoice amount only)
- Cannot refund twice

Searching and Filtering Invoices

Search

Use the search bar to find invoices. Searches across:

- Invoice ID
- Invoice title
- Customer name (system-wide view only)

Filters

Apply filters to narrow invoice list:

Available Filters:

- **Void Status** - All, Void, Not Void
- **Paid Status** - All, Paid, Not yet Paid

Filter Actions:

- **Apply Filters** - Apply selected filters to list
- **Reset Filters** - Clear all filters and show all invoices

Sorting

Click any column header to sort:

- **ID** - Sort by invoice ID (newest/oldest)
- **Title** - Sort alphabetically
- **Due Date** - Sort by due date
- **Created** - Sort by creation date
- **Amount** - Sort by retail cost (highest/lowest)
- **Status** - Sort by paid status (paid first or unpaid first)

Click again to reverse sort direction (ascending ↔ descending).

Pagination

Navigate through large invoice lists with page controls showing current page, total pages, and items per page selector (10, 25, 50, or 100 items).

Common Invoice Workflows

Workflow 1: Monthly Billing with Transaction Preview

1. End of month arrives (e.g., January 31)
2. Navigate to **Billing → Invoices**
3. Click **"+ Generate Proforma Invoice"**
4. Select customer (or do per-customer if billing many customers)
5. Set dates:
    - Start Date: 2025-01-01
    - End Date: 2025-01-31
    - Due Date: 2025-02-15 (15 days from now)
    - Title: "January 2025 Services" (optional)
6. **Transaction Preview** section appears showing all uninvoiced transactions from January
7. Review the preview:
    - All transactions are included by default
    - Check totals: Subtotal, Tax, and Invoice Total
    - Verify all charges are correct
8. Click **"Generate Invoice"** (button shows transaction count, e.g., "Generate Invoice (15)")
9. Invoice created with all selected transactions
10. Click invoice row to view details and verify
11. Click **"Send Email"** button in details modal or email icon in table
12. Customer receives invoice email with PDF and pay link

Workflow 2: Selective Transaction Invoicing

1. Customer has multiple services (Mobile + Internet) and misc charges
2. Wants separate invoices for each service
3. **Generate first invoice (Mobile Services):**
    - Click **"+ Generate Proforma Invoice"**
    - Title: "Mobile Services - January 2025"
    - Start/End: Jan 1-31
    - Due Date: Feb 15
    - In transaction preview, **exclude** all non-mobile transactions:
        - Click **X** button next to Internet transactions

        - Click **X** button next to miscellaneous charges
        - Only Mobile service transactions remain selected
    - Verify totals reflect only mobile services
    - Click **"Generate Invoice"** (shows count of mobile transactions)
4. **Generate second invoice (Internet Services):**
    - Click **"+ Generate Proforma Invoice"** again
    - Title: "Internet Services - January 2025"
    - Start/End: Jan 1-31 (same period)
    - In transaction preview:
        - Mobile transactions already invoiced (don't appear)
        - Exclude miscellaneous charges using **X** button
        - Only Internet service transactions remain
    - Click **"Generate Invoice"**
5. **Generate third invoice (Additional Charges):**
    - Click **"+ Generate Proforma Invoice"** again
    - Title: "Additional Charges - January 2025"
    - Only uninvoiced misc charges appear in preview
    - Click **"Select All"** to include all
    - Click **"Generate Invoice"**
6. Email all three invoices to customer

Workflow 3: Excluding Disputed or Pending Transactions

1. End of billing period arrives
2. Navigate to customer **Billing** tab
3. Click **"+ Generate Proforma Invoice"**
4. Set billing period dates
5. Transaction preview shows 20 transactions
6. Customer has disputed one charge and another is pending investigation
7. In transaction preview:
    - Locate disputed transaction (e.g., "Data overage charge")
    - Click **X** button to exclude it
    - Locate pending transaction (e.g., "Installation fee")
    - Click **X** button to exclude it
    - Transaction count updates: "18 Transactions selected"
    - Totals recalculate automatically
8. Review updated totals (excludes disputed amounts)
9. Click **"Generate Invoice (18)"**
10. Invoice generated with only approved transactions
11. Disputed/pending transactions remain uninvoiced for next billing cycle

Workflow 4: Quick Invoice Review and Adjustment

1. Staff generates monthly invoice
2. Transaction preview shows unexpected high total
3. Review each transaction in the preview:
    - Notice duplicate charge for same service
    - Click **X** to exclude the duplicate
    - Notice test transaction that shouldn't be billed
    - Click **X** to exclude test transaction
4. Totals update in real-time
5. Verify new total matches expected amount
6. Click **"Generate Invoice"** with corrected transactions
7. Go back and void/delete the excluded transactions if needed
8. Email invoice to customer with confidence

Workflow 5: One-Time Installation Invoice

1. Field tech completes installation
2. Staff adds installation transaction manually
3. Navigate to customer **Billing** tab
4. Click **"+ Generate Proforma Invoice"**
5. Set dates:
   - Start Date: today
   - End Date: today
   - Due Date: today + 7 days
   - Title: "Installation Services"
6. Transaction preview shows only today's transactions
7. Verify installation charge appears
8. Exclude any recurring charges using **X** button (if present)
9. Click **"Generate Invoice"**
10. Email to customer immediately
11. Customer pays online via Stripe

Workflow 6: Reviewing Invoice Before Customer Contact

1. Customer calls with billing question
2. Staff navigates to customer's invoice list
3. **Click on invoice row** to open Invoice Details modal
4. Review invoice information:
   - Invoice ID, dates, status
   - All transactions included with descriptions
   - Tax breakdown per transaction
   - Subtotal, Tax, and Total amounts
5. Answer customer's questions with exact details
6. If customer requests PDF, click **"Download PDF"** button in modal
7. If customer requests email resend, click **"Send Email"** button
8. Close modal when done

Workflow 7: Correcting Billing Error

1. Customer reports incorrect charge
2. Staff clicks on invoice row to view details
3. Reviews transaction list in Invoice Details modal
4. Identifies incorrect transaction
5. Invoice is unpaid, so can be voided
6. Click **"Delete"** button in modal footer
7. Confirm void
8. Transactions become uninvoiced again
9. Staff modifies or removes incorrect transaction from transaction list
10. Generate new invoice with corrected transactions:
    - Use transaction preview to exclude corrected transaction if needed
    - Include only valid charges
11. Email corrected invoice to customer

Workflow 8: Processing Multiple Payments

1. Customer brings cash to pay multiple invoices
2. Navigate to customer **Billing** tab
3. View unpaid invoices
4. Click on first invoice row to view details
5. Verify amount and transactions
6. Click **"Pay Invoice"** button in modal footer

7. Select **"Cash"** payment method
8. Enter reference: "Cash paid 2025-01-15"
9. Click **"Pay Invoice"**
10. Modal closes, invoice marked as "Paid"
11. Repeat for remaining invoices
12. All invoices now marked as "Paid"

Workflow 9: Handling Refund Request

1. Customer requests refund for overpayment
2. Staff verifies invoice was paid via Stripe
3. Navigate to invoice in list
4. Click on invoice row to view details
5. Verify payment information and amount
6. Click **"Refund"** button in modal footer (only appears for Stripe invoices)
7. Confirm refund
8. System processes Stripe refund
9. Invoice status changes to "Refunded"
10. Customer receives refund in 3-7 business days
11. Staff follows up with customer to confirm receipt

Troubleshooting

Cannot generate invoice - No transactions found

- **Cause:** No uninvoiced transactions in specified date range
- **Fix:** Check transaction list, verify transactions exist and are not already invoiced. Adjust date range or remove filter.

Invoice PDF generation fails

- **Cause:** Template syntax error, WeasyPrint crash, or missing customer data
- **Fix:** Check invoice template HTML for errors, verify customer address fields populated, review API logs.

Payment fails with Stripe error

- **Cause:** Card declined, insufficient funds, expired card, or Stripe API issue
- **Fix:** Try different payment method, verify card valid, check Stripe dashboard for decline reason.

Cannot void invoice

- **Cause:** Invoice already paid
- **Fix:** Paid invoices cannot be voided. If refund needed, use refund function for Stripe invoices or create credit transaction manually.

Invoice email not sending

- **Cause:** Mailjet API credentials invalid, customer has no billing contact, or email template missing
- **Fix:** Verify Mailjet configuration in `crm_config.yaml`, check customer contacts, verify invoice email template exists.

Refund button not appearing

- **Cause:** Invoice paid via cash/POS/bank transfer (not Stripe), or invoice not paid

- **Fix:** Refund button only appears for Stripe payments. For other payment methods, create manual credit transaction.

Download PDF shows old template design

- **Cause:** PDF cached before template update
- **Fix:** Clear invoice PDF cache: `DELETE FROM Invoice_PDF_Cache WHERE invoice_id = X;`

Customer cannot pay invoice (no payment methods)

- **Cause:** No saved payment methods in Self-Care portal
- **Fix:** Customer must add credit card at **Payment Methods** page before paying invoices.

Multiple invoices generated for same period

- **Cause:** Staff generated invoice twice, or date ranges overlap
- **Fix:** Void duplicate invoice. Adjust date ranges to prevent overlap. Use transaction preview to ensure unique transaction sets.

Transaction preview shows no transactions

- **Cause:** All transactions in date range are already invoiced or no transactions exist
- **Fix:** Verify date range is correct. Check transaction list to confirm transactions exist. Filter invoices to see which invoice contains the transactions.

Cannot exclude transaction from invoice generation

- **Cause:** Transaction already invoiced or browser issue
- **Fix:** Verify transaction shows in preview with checkmark. Refresh page and try again. Clear browser cache if issue persists.

Invoice total doesn't match expected amount

- **Cause:** Unexpected transactions included, tax not calculated, or excluded transactions still counted
- **Fix:** Review transaction preview carefully. Check each transaction's retail cost and tax. Verify excluded transactions are grayed out. Check transaction count badge on Generate Invoice button.

Generate Invoice button is disabled

- **Cause:** No transactions selected or invalid date range
- **Fix:** Ensure at least one transaction is included (not excluded). Verify Start Date is before End Date. Check that Due Date is set.

Invoice Details modal not opening

- **Cause:** JavaScript error or page not fully loaded
- **Fix:** Refresh page. Check browser console for errors. Try different browser. Verify internet connection.

Transaction tax not displaying in Invoice Details

- **Cause:** Transaction has 0% tax or tax_amount is null
- **Fix:** Verify transaction has tax_percentage set. Check that tax_amount was calculated when transaction was created. Update transaction if needed.

Action buttons missing in Invoice Details modal

- **Cause:** Invoice is voided or user lacks permissions
- **Fix:** Voided invoices only show Download PDF button. Verify invoice status. Check user role and permissions.

Related Documentation

- `integrations_mailjet` - Email invoice delivery and templates
- `administration_configuration` - Invoice template configuration
- `payments_transaction` - Creating transactions that appear on invoices
- `payments_process` - Processing invoice payments
- `basics_payment` - Payment methods and Stripe integration

Process Payments

The majority of payments will be processed automatically by the system, but there are times when you may need to process a payment manually.

To pay an invoice, select the unpaid invoice, and click on the "Pay" button.

This will open a payment form, where you can enter the payment method, and click "Submit" to process the payment.

The customer will automatically receive a receipt for the payment, and the invoice will be marked as paid.

For bank transfers, you can enter the Payment reference and the date the payment was made (if different from the current date).

Pay an Invoice Pay an Invoice

Customer Transactions

Anything that costs money in the system is recorded as a transaction under the customer.

Every transaction has a monetary amount for wholesale cost and retail cost, and a description of what the transaction is for.

Transactions can be automatically generated by the system, for example, when a service is provisioned, a transaction is created for the setup cost, and when a service is billed, a transaction is created for the retail cost.

Transactions can also be manually created, for example, if a customer is given a credit, a transaction is created for the credit amount, or an installation fee is charged, a transaction is created for the installation fee.

Transactions are grouped together to form `Invoices <payments_invoices>`, which is sent to the customer for payment.

Transactions Management

Accessing Transactions

Transactions can be viewed at the system level or per-customer:

Per-Customer View:

1. Navigate to **Customers → [Select Customer]**
2. Click **Billing** tab
3. View transactions list in the first card

System-Wide View:

1. Navigate to **Billing → Transactions** (from main menu)
2. View all transactions across all customers

Transaction Statistics Widgets

At the top of the transactions page, four statistics cards display financial summaries:

Transaction Statistics and List

Widget Descriptions:

- **Total Transactions** - Sum of all transaction retail costs (all time)
- **Total Uninvoiced Transactions** - Sum of transactions not yet included in an invoice
- **Total Transactions This Month** - Sum of transactions created this calendar month
- **Total Transactions Last Month** - Sum of transactions created last calendar month

Value Formatting:

- Values over 1,000: Display as "k" suffix (e.g., $1.5k)
- Values over 1,000,000: Display as "M" suffix (e.g., $2.3M)
- Values over 1,000,000,000: Display as "B" suffix (e.g., $1.1B)

Transactions List

The transactions table displays all transactions with the following columns:

Column Descriptions:

- **ID** - Unique transaction ID
- **Date** - Transaction creation date
- **Title** - Short transaction name
- **Description** - Detailed description of what the transaction is for
- **Amount** - Retail cost (positive for charges, negative for credits)
- **Invoice** - Invoice ID if transaction has been invoiced (clickable link)
- **Status** - Checkmark if invoiced, dash if not yet invoiced

Actions Per Row:

Each row has an actions menu (⋮) with options:

- **View Details** - Opens transaction detail modal
- **Download Invoice PDF** - Download PDF (only if invoiced)
- **Void Transaction** - Mark transaction as void (only if not invoiced)

Transaction Types

Transactions fall into two main categories:

Debit Transactions (Charges)

Positive amounts that increase customer balance owed:

- **Service Setup Fees** - One-time charges when service provisioned
- **Monthly Service Fees** - Recurring charges for services
- **Installation Fees** - Charges for field technician visits
- **Equipment Charges** - Charges for modems, routers, SIM cards
- **Late Payment Fees** - Penalties for overdue invoices
- **Manual Charges** - Custom charges added by staff

Credit Transactions (Payments/Refunds)

Negative amounts that decrease customer balance owed:

- **Cash Payments** - Customer paid by cash
- **Card Payments** - Customer paid by credit/debit card
- **Bank Transfer Payments** - Customer paid via bank transfer
- **Account Credits** - Goodwill credits, compensation
- **Refunds** - Money returned to customer
- **Discounts** - Promotional or loyalty discounts

Adding a Transaction Manually

Click **"+ Add Transaction"** to open the add transaction modal.

Debit Transaction (Charge):

Add Debit Transaction Modal

Credit Transaction (Payment/Refund):

Add Credit Transaction Modal

Field Descriptions:

- **Transaction Type** - Select Debit (charge) or Credit (payment/refund)
- **Credit Type** - If Credit selected, choose payment method (Cash, Card, Bank Transfer)
- **Title** - Short name for transaction (required)
- **Description** - Detailed explanation (optional)
- **Retail Cost** - Amount customer pays (required, positive number)
- **Wholesale Cost** - Your cost (optional, for margin tracking)
- **Tax Percentage** - Tax rate applied to this transaction (optional, defaults to product tax or 0%)
- **Service** - Link transaction to specific service (optional)
- **Site** - Link transaction to specific site (optional)
- **Transaction Date** - Date of transaction (defaults to today)

Validation:

- Title and retail cost are required
- Retail cost must be a positive number
- If Credit type selected, a credit type must be chosen

What Happens:

1. Transaction created in database
2. Appears in customer's transactions list
3. Included in "Uninvoiced Transactions" count
4. Available for inclusion in next invoice generation
5. Activity log entry created

Searching and Filtering Transactions

Search

Use the search bar to find transactions. Searches across:

- Transaction ID
- Title
- Description
- Invoice ID

Filters

Apply filters to narrow transaction list:

Available Filters:

- **Void Status** - All, Void, Not Void
- **Invoice Status** - All, Invoiced, Not Invoiced

Filter Actions:

- **Apply Filters** - Apply selected filters to list
- **Reset Filters** - Clear all filters and show all transactions

Sorting

Click any column header to sort:

- **ID** - Sort by transaction ID (newest/oldest)
- **Date** - Sort by transaction date
- **Title** - Sort alphabetically
- **Amount** - Sort by retail cost (highest/lowest)
- **Invoice** - Sort by invoice ID

Click again to reverse sort direction (ascending ↔ descending).

Voiding Transactions

Transactions added in error can be **voided** (marked as deleted).

Requirements:

- Transaction must NOT be invoiced
- Once invoiced, transactions cannot be voided (must be refunded instead)

How to Void:

1. Locate transaction in list
2. Click actions menu (⋮)
3. Select **"Void Transaction"**
4. Confirm in modal

Void Transaction Confirmation Modal

What Happens:

- Transaction marked as `void = true`
- No longer appears in default transaction list
- Excluded from invoice generation
- Can be viewed by filtering for "Void" transactions
- Deducted from "Uninvoiced Transactions" total

Note: Voiding is NOT the same as refunding. Void means "this transaction should never have existed." Refund means "reverse a valid transaction."

Tax on Transactions

Transactions can include tax, which is automatically calculated based on the product's tax configuration or manually specified per transaction.

Transaction Tax Display

Tax Behavior:

- **Debit Transactions (Charges)** - Tax is applied to charges based on:
  - **Product Tax Percentage** - If the transaction is linked to a product, the product's tax percentage is automatically applied
  - **Manual Override** - Staff can override the tax percentage when creating a transaction
  - **Tax Amount** - Calculated as: `retail_cost × (tax_percentage / 100)`
  - **Display Format** - Shown as: `$10.00 (10%)` in transaction lists

- **Credit Transactions (Payments/Refunds)** - No tax is applied to credits
  - Tax percentage field is hidden for credit transactions
  - Tax is automatically set to 0% for all payments and refunds
  - Credits reduce the customer's outstanding balance without tax implications

Tax Calculation Example:

- Product: Mobile Plan with 10% tax, $50.00 retail cost
- Automatic Tax Calculation: $50.00 × 0.10 = $5.00
- Display: $5.00 (10%)

Zero Tax (NIL/Exempt):

- Products can be tax-exempt by setting tax percentage to 0
- Tax defaults to 0% if not specified
- Tax-exempt transactions show "-" in the Tax column

Zero Tax Transaction

Transaction Details View

Click a transaction to view full details:

Invoiced vs Uninvoiced Transactions

Uninvoiced Transactions:

- Not yet included in any invoice
- Available for next invoice generation
- Can be voided
- Count toward "Uninvoiced Transactions" total
- Status shows dash (-)

Invoiced Transactions:

- Included in an invoice
- Cannot be voided (must refund if needed)
- Invoice ID clickable (links to invoice details)
- Status shows checkmark (✓)
- Cannot be modified

Invoice Generation:

When you generate an invoice for a customer:

1. System finds all uninvoiced transactions for that customer
2. Optionally filter by date range
3. Transactions included in new invoice
4. Transaction `invoice_id` field populated
5. Transaction now marked as "invoiced"

See `payments_invoices` for invoice generation details.

Common Workflows

Workflow 1: Manual Credit for Service Outage

1. Customer calls: "Service was down for 2 days"
2. Staff decides to credit £10
3. Navigate to customer **Billing** tab
4. Click **"+ Add Transaction"**
5. Select **Credit** transaction type
6. Select **Cash Payment** credit type
7. Enter title: "Service Outage Credit"
8. Enter description: "Compensation for 2-day outage 8-9 Jan"
9. Enter retail cost: 10.00
10. Select affected service from dropdown
11. Click **"Add Transaction"**
12. Transaction appears with -£10.00 amount
13. Will be included in next invoice as credit

Workflow 2: Manual Installation Fee

1. Field tech installs service
2. Staff needs to charge £75 installation fee
3. Navigate to customer **Billing** tab
4. Click **"+ Add Transaction"**
5. Select **Debit** transaction type
6. Enter title: "Installation Fee"
7. Enter description: "Field technician visit for fiber installation"
8. Enter retail cost: 75.00
9. Enter wholesale cost: 45.00 (optional, for margin tracking)
10. Select service installed
11. Select site where installed
12. Click **"Add Transaction"**
13. Transaction appears in uninvoiced list
14. Will be included in next invoice

Workflow 3: Voiding Duplicate Transaction

1. Staff notices duplicate transaction
2. Verify transaction NOT yet invoiced
3. Click actions menu (⋮) on duplicate transaction
4. Select **"Void Transaction"**
5. Confirm in modal
6. Transaction removed from list
7. Uninvoiced total decreases accordingly

Workflow 4: Finding Transactions for Invoice

1. Need to generate monthly invoice
2. Click **Invoice filter: "Not Invoiced"**
3. Click **Apply Filters**
4. View all uninvoiced transactions
5. Note total amount from widgets
6. Navigate to generate invoice
7. Select date range (e.g., 1-31 Jan)
8. Transactions in range included in invoice

Troubleshooting

Cannot void transaction

- **Cause:** Transaction already invoiced
- **Fix:** Transaction is part of invoice history. If refund needed, create a Credit transaction instead.

Duplicate transactions appearing

- **Cause:** Service charged multiple times or provisioning error
- **Fix:** Void the duplicate transaction(s) if not invoiced. If invoiced, issue credit.

Transaction not appearing in list

- **Cause:** Filters applied or transaction voided
- **Fix:** Click "Reset Filters" to show all transactions. To see voided transactions, filter by "Void: Void".

Uninvoiced total doesn't match expected

- **Cause:** Some transactions already invoiced, or voided transactions excluded
- **Fix:** Apply filter "Invoice: Not Invoiced" to see only uninvoiced. Check voided transactions separately.

Cannot add transaction (customer field disabled)

- **Cause:** Viewing customer-specific transactions page
- **Fix:** Customer is pre-selected. If you need to add transaction for different customer, go to system-wide Transactions page.

Related Documentation

- `payments_invoices` - Invoice generation and management
- `payments_process` - Processing payments against invoices
- `basics_payment` - Payment methods overview
- `csa_activity_log` - Viewing transaction history in activity log

Rule Based Access Control

Roles, Permissions & Users in OmniCRM

OmniCRM uses **role-based access control (RBAC)**: people (Users) are assigned one or more Roles, and each Role is a bundle of Permissions. Permissions are the smallest unit of access (e.g., view_customer, create_inventory). A user's effective access is the **union** of permissions from all assigned roles.

Purpose

RBAC enables:

1. **Data Protection** — Users only see and do what they're allowed to.
2. **Operational Fit** — Roles mirror job functions (Admin, Support, Finance, Customer Admin).
3. **Simple Admin** — Grant access by assigning roles; avoid per-user micromanagement.
4. **Tenant Isolation** — "view own …" permissions limit visibility to a user's own customer/tenant data.

RBAC Permissions

How Users, Roles, and Permissions Fit Together

- **Users** — Real people who sign in to OmniCRM.
- **Permissions** — Atomic capabilities (e.g., view_customer, delete_product).
- **Roles** — Named sets of permissions (e.g., *Admin*, *Support*, *Finance*).
- **Assignment** — Users receive one or more roles; permissions aggregate.

RBAC Permissions

Authentication proves *who you are* (JWT, API key, or whitelisted IP). Authorization (roles/permissions) decides *what you can do*.

Managing Users

The OmniCRM user management system allows administrators to create and manage staff users (administrators, customer service agents), view and modify user roles, reset passwords, manage two-factor authentication, and control user access.

User Types

Customer Users - Created via self-signup or by administrators. Automatically assigned the "Customer" role. These users access the self-care portal to manage their services, view usage, pay invoices, etc.

Staff Users - Created by administrators with appropriate permissions. Can be assigned roles like Admin, Support, Finance, etc. These users access the CRM interface to manage customers, provision services, handle billing, etc.

Administrative Users - Users with the `admin` permission. Have full access to the system including user management, role management, and all protected endpoints.

The initial administrative user is created by the Omnitouch team when the system is deployed.

Adding New Users (Administrators and CSAs)

Administrators can create new staff users through the Web UI or API.

Via Web UI

1. **Navigate to Users & Roles** - Access the user management interface from the administration menu
2. **Click "Add User"** - Opens the user creation form

Adding a User

3. **Fill in User Details:**
    - **Username** - Unique username for login (required)
    - **Email** - User's email address (required, must be unique)
    - **Password** - Temporary password (required, user should change on first login)
    - **First Name** - User's first name (required)
    - **Middle Name** - User's middle name (optional)
    - **Last Name** - User's last name (required)
    - **Phone Number** - Contact phone number (optional)
    - **Role** - Select one or more roles to assign (required)
    - **Customer Contact** - Optionally link user to a customer contact record (for customer users)
4. **Click "Create User"** - User is created and can immediately log in with the provided credentials
5. **User receives notification** - Optionally send welcome email with login instructions

Best Practices:

- Use a temporary password like `TempP@ssw0rd!` and require user to change it on first login
- Assign appropriate roles based on job function (see Typical Role Designs below)
- Enable 2FA for all administrative and support staff
- Link customer users to their customer contact record for proper data scoping

Via API

Create a user programmatically:

Endpoint: `POST /auth/users`

Required Permission: `admin`

Request Body:

```json
{
  "username": "john.smith",
  "email": "john.smith@company.com",
  "password": "TempP@ssw0rd!",
  "first_name": "John",
  "middle_name": "D",
  "last_name": "Smith",
  "phone_number": "+61412345678",
  "role": "Support"
}
```

Response:

```json
{
  "id": 123,
  "username": "john.smith",
  "email": "john.smith@company.com",
  "first_name": "John",
  "last_name": "Smith",
  "roles": ["Support"],
  "created": "2025-01-04T10:30:00Z"
}
```

Assigning Multiple Roles:

Users can have multiple roles. Permissions are additive (union of all assigned role permissions).

To assign multiple roles, include them in the request:

```json
{
```

```
  "username": "jane.doe",
  "email": "jane.doe@company.com",
  "password": "TempP@ssw0rd!",
  "first_name": "Jane",
  "last_name": "Doe",
  "role": "Support,Finance"
}
```

Or use the role assignment endpoint after user creation:

```
POST /auth/roles/{role_id}/users/{user_id}
```

Viewing and Searching Users

List All Users (Admin):

```
GET /auth/users
```

Returns paginated list of all users with their roles and basic information.

Search Users:

```
GET /auth/users/
search?search={query}&filters={"role":["Support"]}&page=1&per_page=50
```

Filter by:

- Role name
- Email domain
- Active/deleted status
- 2FA enabled status
- Last login date

Get Specific User:

```
GET /auth/users/{user_id}
```

Returns full user details including:

- Personal information
- Assigned roles and effective permissions
- 2FA status
- Last login and session information
- Linked customer contact (if applicable)

Creating and Managing Roles

Roles are collections of permissions that can be assigned to users. Instead of assigning permissions individually to each user, you create roles that bundle related permissions and assign those roles to users.

Creating a New Role

Via Web UI:

1. Navigate to **Users & Roles** → **Roles** tab
2. Click **"Create Role"**
3. Enter role details:
    - **Name** - Short, descriptive name (e.g., "Tier2_Support")
    - **Description** - Explain the role's purpose and responsibilities
4. Click **"Create"**
5. Role is created with no permissions; add permissions in the next step

Via API:

Endpoint: `POST /auth/roles`

Required Permission: `admin`

Request:

```json
{
  "name": "Tier2_Support",
  "description": "Level 2 support team with elevated provisioning access"
}
```

Response:

```json
{
  "id": 45,
  "name": "Tier2_Support",
  "description": "Level 2 support team with elevated provisioning access",
  "permissions": [],
  "users": []
}
```

Adding Permissions to a Role

Once a role is created, assign permissions to define what users with that role can do.

Via Web UI:

1. Navigate to **Users & Roles** → **Roles** tab
2. Click on the role name to view details

3. In the **Permissions** section, click **"Add Permission"**
4. Select one or more permissions from the list
5. Click **"Add"** - Permissions are immediately assigned

Managing Role Permissions Assigning Permissions to Role

Via API:

Endpoint: `POST /auth/roles/{role_id}/permissions`

Request:

```json
{
  "permission_id": 123
}
```

Or add multiple permissions:

```json
{
  "permission_ids": [123, 124, 125]
}
```

Example: Creating a "Provisioning Specialist" Role

This role can view customers, manage services, and provision:

1. Create the role:

   ```
   POST /auth/roles
   {
     "name": "Provisioning_Specialist",
     "description": "Can provision services and manage customer
   services"
   }
   ```

2. Add permissions:

   ```
   POST /auth/roles/45/permissions
   {
     "permission_ids": [
       1,   # view_customer
       20,  # view_customer_service
       21,  # create_customer_service
       22,  # update_customer_service
       30,  # view_provision
       31,  # create_provision
       40,  # view_inventory
       50,  # view_product
     ]
   ```

```
}
```

Removing Permissions from a Role

Via Web UI:

1. Navigate to role details
2. In **Permissions** list, click the **"X"** or **"Remove"** button next to the permission
3. Confirm removal

Via API:

Endpoint: `DELETE /auth/roles/{role_id}/permissions/{permission_id}`

Example:

```
DELETE /auth/roles/45/permissions/31
```

This removes `create_provision` permission from the role.

Editing Role Details

Update role name or description:

Via Web UI:

1. Navigate to **Users & Roles** → **Roles** tab
2. Click on the role to edit
3. Modify the role name or description
4. Click **"Save"**

Editing Role Details

Via API:

Endpoint: `PUT /auth/roles/{role_id}`

```json
{
  "name": "Senior_Support",
  "description": "Senior support team with full customer access"
}
```

Deleting a Role

Warning: Deleting a role removes it from all assigned users. Ensure users have alternative roles or they will lose access.

Via API:

`DELETE /auth/roles/{role_id}`

Best Practice: Instead of deleting, consider archiving or renaming roles that are no longer needed.

Assigning Roles to Users

During User Creation:

Include role in the user creation request (see "Adding New Users" above).

For Existing Users:

Via Web UI:

1. Navigate to **Users & Roles** → **Users** tab
2. Click on the user to edit
3. In the **Roles** section, select/deselect roles
4. Click **"Save"**

Editing User Roles

Via API:

Update user's roles:

Endpoint: `PUT /auth/users/{user_id}`

```
{
  "role": "Support,Finance"
}
```

Or assign a single role to a user via role endpoint:

Endpoint: `POST /auth/roles/{role_id}/users/{user_id}`

Viewing Role Assignments

All users in a role:

`GET /auth/roles/{role_id}/users`

Returns list of all users assigned to that role.

All roles for a user:

`GET /auth/users/{user_id}`

The response includes `roles` array with all assigned roles.

Managing User Passwords

OmniCRM provides multiple methods for password management depending on the context.

User Self-Service Password Reset

Users who forgot their password can reset it themselves via the login page:

1. **Click "Forgot Password"** on the login page
2. **Enter email address** - System sends a password reset email
3. **Check email** - Email contains a secure reset link with token (valid for 1 hour)
4. **Click link** - Opens password reset form
5. **Enter new password** - Must meet password complexity requirements:
    - Minimum 8 characters
    - At least one uppercase letter
    - At least one lowercase letter
    - At least one number
    - At least one special character
6. **Submit** - Password is immediately updated; user can log in with new password

API Flow:

1. **Request reset:**

   Endpoint: `POST /auth/forgot_password`

   ```json
   {
     "email": "user@example.com"
   }
   ```

   System generates reset token and sends email.

2. **Reset with token:**

   Endpoint: `POST /auth/reset_password`

   ```json
   {
     "token": "abc123...",
     "new_password": "NewSecureP@ssw0rd!"
   }
   ```

Administrator Password Reset

Administrators can reset a user's password without requiring email verification. This sets a temporary password that the user should change on next login.

Via Web UI:

1. Navigate to **Users & Roles → Users**
2. Find the user and click the **"Reset Password"** button
3. Enter a temporary password
4. Click **"Reset"**
5. Notify the user of their temporary password (via secure channel)
6. User should change password on next login

Via API:

Endpoint: `POST /auth/users/{user_id}/admin_reset_password`

Required Permission: `admin`

Request:

```json
{
  "new_password": "TempP@ssw0rd!",
  "force_change": true
}
```

Parameters:

- `new_password` - Temporary password to set
- `force_change` (optional) - If true, user must change password on next login

User Change Own Password

Authenticated users can change their own password from their profile:

Endpoint: `POST /auth/change_password`

Request:

```json
{
  "current_password": "OldP@ssw0rd!",
  "new_password": "NewSecureP@ssw0rd!"
}
```

System validates the current password before allowing the change.

Password Security

- Passwords are hashed using bcrypt (werkzeug security)
- Never stored in plaintext
- Reset tokens expire after 1 hour
- Failed login attempts can trigger account lockout (configurable)
- Password history tracking prevents reuse (configurable)

- Complexity requirements enforced

Managing Two-Factor Authentication (2FA)

OmniCRM supports TOTP-based two-factor authentication for enhanced security. Administrators can enable, disable, and reset 2FA for users.

OmniCRM 2FA nag popup shown to admin users

Enabling 2FA for a User

Via Web UI:

1. Navigate to **Users & Roles → Users**
2. Click on the user to view details
3. In the **Security** section, click **"Enable 2FA"**
4. System generates:
   - TOTP secret (QR code displayed)
   - 10 backup codes (one-time use)
5. User scans QR code with authenticator app (Google Authenticator, Authy, etc.)
6. User enters verification code from app to confirm setup
7. User saves backup codes in secure location
8. 2FA is now enabled; required for all future logins

OmniCRM 2FA enrollment page

Via API:

1. **Generate TOTP secret:**

   Endpoint: `POST /2fa/enable/user/{user_id}`

   Response:

   ```
   {
     "totp_secret": "JBSWY3DPEHPK3PXP",
     "qr_code_url": "otpauth://totp/OmniCRM:user@example.com?secret=JBSWY3DPEHPK3PXP&issuer=OmniCRM",
     "backup_codes": [
       "12345678",
       "23456789",
       "34567890",
       ...
     ]
   }
   ```

2. **Verify setup:**

Endpoint: `POST /2fa/verify-setup/user/{user_id}`

```json
{
  "code": "123456"
}
```

Returns `{"verified": true}` on success.

2FA Login Flow

Once 2FA is enabled, the login process changes:

1. User enters username and password
2. System validates credentials
3. If valid, prompts for 2FA code
4. User enters code from authenticator app OR backup code
5. System verifies code
6. On success, user is logged in

OmniCRM 2FA Login

Backup Codes:

- 10 codes generated during 2FA setup
- Single-use only (consumed after use)
- Used if authenticator app is unavailable
- Can be regenerated by user or admin

Verifying 2FA Code

Endpoint: `POST /2fa/verify/user/{user_id}`

```json
{
  "code": "123456"
}
```

Accepts both:

- **TOTP code** (6 digits from authenticator app)
- **Backup code** (8 digits from backup code list)

Regenerating Backup Codes

If a user exhausts backup codes or loses them, generate new ones:

Via Web UI:

1. Navigate to user details
2. Click **"Regenerate Backup Codes"**

3. Display or send new codes to user
4. Old codes are invalidated

Via API:

Endpoint: `POST /2fa/backup-codes/regenerate/user/{user_id}`

Response:

```json
{
  "backup_codes": [
    "98765432",
    "87654321",
    "76543210",
    ...
  ]
}
```

Administrator 2FA Reset

If a user loses access to their authenticator app and all backup codes, an administrator can disable and re-enable 2FA.

Via Web UI:

1. Navigate to **Users & Roles → Users**
2. Click on the user
3. Click **"Reset 2FA"** button
4. Confirm the reset
5. 2FA is disabled; user can log in with just password
6. Guide user to set up 2FA again with new secret

OmniCRM 2FA Reset

Via API:

Endpoint: `POST /2fa/admin/disable/user/{user_id}`

Required Permission: `admin`

This completely disables 2FA for the user:

- TOTP secret cleared
- Backup codes cleared
- `is_2fa_enabled` flag set to false

User can then re-enable 2FA to get new secret and backup codes.

User Self-Service 2FA Reset (New Device)

If a user gets a new device but still has access to backup codes:

Endpoint: `POST /2fa/reset-for-new-device/user/{user_id}`

```json
{
  "backup_code": "12345678"
}
```

System validates the backup code, then generates new TOTP secret and backup codes. User can set up authenticator app on new device.

2FA Best Practices

- **Require 2FA for all admin and support staff**
- **Store backup codes securely** (password manager or secure note)
- **Regenerate backup codes** after using several
- **Use reputable authenticator apps** (Google Authenticator, Authy, Microsoft Authenticator)
- **Document 2FA reset procedures** for support staff
- **Audit 2FA usage** - monitor which users have 2FA enabled

Updating User Information

Administrators can update user details at any time.

Via Web UI:

1. Navigate to **Users & Roles → Users**
2. Click on user to edit
3. Modify any editable fields:
   - First name, middle name, last name
   - Email address (requires verification)
   - Phone number
   - Roles
   - Customer contact linkage
4. Click **"Save"**

Via API:

Endpoint: `PUT /auth/users/{user_id}`

```json
{
  "first_name": "Jane",
  "last_name": "Doe-Smith",
  "email": "jane.doesmith@newcompany.com",
  "phone_number": "+61498765432",
```

```
  "role": "Support,Finance"
}
```

Email Changes:

When email is changed, the new email is marked as pending until verified:

- `pending_email` field stores new email
- Verification email sent to new address
- User clicks link to verify
- `email` field updated to new value
- `email_verified` flag set to true

Deleting Users

OmniCRM uses **soft deletes** for users - they are marked as deleted but not removed from the database. This preserves audit trails and historical data.

Deleting a User

Via Web UI:

1. Navigate to **Users & Roles** → **Users**
2. Find the user to delete
3. Click **"Delete"** button
4. Confirm deletion
5. User is immediately logged out and cannot log in again

Via API:

Endpoint: `DELETE /auth/users/{user_id}`

Required Permission: `admin`

What Happens:

- `deleted` flag set to `True`
- `deleted_at` timestamp recorded
- User cannot log in
- All active sessions invalidated
- User still appears in audit logs and historical records
- Linked data (customer contacts, activities) preserved

Viewing Deleted Users

Filter for deleted users:

```
GET /auth/users/search?filters={"deleted":[true]}
```

Restoring a Deleted User

If a user was deleted by mistake, administrators can restore them:

Endpoint: `PUT /auth/users/{user_id}`

```json
{
  "deleted": false
}
```

This clears the `deleted` flag and allows the user to log in again.

Note: User's password remains unchanged, so they can use their previous password.

Permanently Deleting a User

Warning: This is irreversible and removes all user data from the database.

Not exposed via UI. Only available via direct database access for compliance reasons (e.g., GDPR data deletion requests).

Best Practices for User Deletion

- **Soft delete by default** - Preserves audit trails
- **Document deletion reasons** - Add note in activity log before deleting
- **Transfer ownership** - Reassign user's open tickets, tasks before deleting
- **Review access** - Ensure no critical processes depend on the user
- **Archive data** - Export user's work history if needed
- **Notify relevant teams** - Inform managers/colleagues of deletion

Permission Catalog

Permissions generally follow CRUD patterns:

- view_* — read/browse
- create_* — create/add
- update_* — edit/modify
- delete_* — delete/remove

Some entities also include **“view own …”** variants that restrict visibility to the current user’s customer/tenant.

RBAC Permissions

Global / Administrative

- `admin` — Full administrative access (manage users, roles, and permissions; access all protected endpoints).
- `can_impersonate` — Temporarily act as another user (audited; for support/troubleshooting).

Customers & Related Records

- **Customer**
  - view_customer, create_customer, update_customer, delete_customer
  - **Own-scope:** *view own customer*
- **Customer Site**
  - view_customer_site, create_customer_site, update_customer_site, delete_customer_site
  - **Own-scope:** *view own customer site*
- **Customer Contact**
  - view_customer_contact, create_customer_contact, update_customer_contact, delete_customer_contact
  - **Own-scope:** *view own customer contact*
- **Customer Attribute** (see `Customer Attributes <administration_attributes>`)
  - view_customer_attribute, create_customer_attribute, update_customer_attribute, delete_customer_attribute
  - **Own-scope:** *view own customer attribute*
- **Customer Tag** (see `Customer Tags <administration_tags>`)
  - view_customer_tag, create_customer_tag, update_customer_tag, delete_customer_tag
  - **Own-scope:** *view own customer tag*
- **Customer Service**
  - view_customer_service, create_customer_service, update_customer_service, delete_customer_service
  - **Own-scope:** *view own customer service*
- **Customer Activity**
  - view_customer_activity, create_customer_activity, update_customer_activity, delete_customer_activity
  - **Own-scope:** *view own customer activity*

Billing

- **Stripe Card**
  - `view_customer_stripe_card`, `create_customer_stripe_card`, `update_customer_stripe_card`, `delete_customer_stripe_card`
  - **Own-scope:** *view own customer stripe card*
- **Transactions**
  - `view_customer_transaction`, `create_customer_transaction`, `update_customer_transaction`, `delete_customer_transaction`

    - **Own-scope:** *view own customer transaction*
- **Invoices**
    - `view_customer_invoice`, `create_customer_invoice`, `update_customer_invoice`, `delete_customer_invoice`
    - **Own-scope:** *view own customer invoice*

Communications

- view_communication, create_communication, update_communication, delete_communication
- **Own-scope:** *view own communication*

Inventory & Templates

- **Inventory**
    - `view_inventory`, `create_inventory`, `update_inventory`, `delete_inventory`
    - **Own-scope:** *view own inventory*
- **Inventory Template**
    - `view_inventory_template`, `create_inventory_template`, `update_inventory_template`, `delete_inventory_template`
    - **Own-scope:** *view own inventory template*

Products

- `view_product`, `create_product`, `update_product`, `delete_product`

Cell Broadcast (CBC)

- `view_cbc_message`, `create_cbc_message`, `update_cbc_message`, `delete_cbc_message`

Provisioning

- **Provision**
    - `view_provision`, `create_provision`, `update_provision`, `delete_provision`
    - **Own-scope:** *view own provision*
- **Provision Event**
    - `view_provision_event`, `create_provision_event`, `update_provision_event`, `delete_provision_event`

"View Own" Access

"View own …" permissions scope reads (and optionally edits, where implemented) to data associated with the user's **own customer/tenant**. For example, a

Customer Admin role can manage *their* tenant's contacts, sites, invoices, and services, but cannot see other tenants.

Typical Role Designs

Role	Typical Permissions	Notes
System Admin	`admin`, optionally `can_impersonate`; plus broad CRUD as needed	Full control over users/roles/permissions
Support	`view_customer`, `view_customer_service`, `view_communication`, `view_provision`; optional updates	Add `can_impersonate` if permitted
Finance	`view_customer_invoice`, `view_customer_transaction`, `view_product`; optional `create_customer_invoice`	Read-heavy; limited write
Customer Admin (tenant)	"view own ..." across customer, sites, contacts, services, inventory, invoices, transactions, communications, provisioning	Tenant-scoped management
Read-only Auditor	Broad `view_*` only	No create/update/delete

Example Roles and Included Permissions (summary)

Managing Roles & Permissions via API

All endpoints require `admin` permission.

List permissions

Endpoint: `GET /auth/permissions`

Create a permission (rare)

Endpoint: `POST /auth/permissions`

Request Body:

```json
{
    "name": "view_example",
    "description": "Read-only access to example objects"
}
```

List roles

Endpoint: `GET /auth/roles`

Create a role

Endpoint: `POST /auth/roles`

Request Body:

```json
{
"name": "Support",
"description": "Tier-1 support team"
}
```

Add a permission to a role

Endpoint: `POST /auth/roles/{role_id}/permissions`

Request Body:

```json
{
  "permission_id": 123
}
```

Remove a permission from a role

Endpoint: `DELETE /auth/roles/{role_id}/permissions/{permission_id}`

Assigning Roles to Users

Create a user with role

Endpoint: `POST /auth/users`

Request Body:

```json
{
  "username": "sara",
  "email": "sara@example.com",
  "password": "TempP@ssw0rd!",
  "first_name": "Sara",
  "last_name": "Ng",
  "phone_number": "+61...",
  "role": "Support"
}
```

Update a user's role

Endpoint: `PUT /auth/users/{user_id}`

Request Body:

```json
{
  "role": "Finance"
}
```

List users (Admin-only)

Endpoint: `GET /auth/users`

Impersonation (Controlled)

- **Required:** `can_impersonate` or `admin`

Start impersonation

Endpoint: `POST /auth/impersonate`

Request Body:

```json
{ "user_id": 42 }
```

Stop impersonation

Endpoint: `POST /auth/stop_impersonation`

Best Practices

- **Least privilege first.** Start with minimal roles; add permissions as needed.
- **Prefer "view own …".** Use tenant-scoped permissions for customer-facing roles.
- **Keep roles stable.** Update role permissions when features change—don't edit each user.
- **Audit regularly.** Review who has `admin` or `can_impersonate`.

FAQ

Can a user have multiple roles? Yes. Permissions are additive.

Do I need custom permissions? Usually not. The built-in catalog covers most needs.

How do "view own …" rules know what's mine? They evaluate the linkage between your user/contact and your customer (tenant).

Self-Care Portal

The Self-Care Portal is a customer-facing interface that allows end users to manage their own accounts, view usage, pay invoices, and modify services without requiring assistance from customer service staff.

Access Methods:

- Direct login via customer credentials
- Staff impersonation <customer_care> for troubleshooting (from Customers → Contacts → "Login as User")

See also: Customer Care <customer_care> for impersonation details, Authentication Flows <authentication_flows> for login process.

Purpose

The Self-Care Portal provides customers with:

1. **Account Management** - View and update personal information, contacts, and addresses
2. **Service Overview** - See all active services, usage, and expiry dates
3. **Usage Tracking** - Monitor data, voice, SMS, and monetary balances
4. **Billing Access** - View and pay invoices, manage payment methods
5. **Service Modifications** - Add top-ups, purchase add-ons, modify services
6. **24/7 Availability** - Access account information anytime without calling support

Portal Overview

When customers log in to the Self-Care Portal, they see a personalized dashboard with:

Navigation Sections:

- **Dashboard** - Quick overview of services and recent activity
- **Account** - Personal details, contacts, sites
- **Services** - List of all services with status and details
- **Usage** - Balance consumption and expiry information
- **Billing** - Invoices, transactions, payment methods
- **Top-Up** - Purchase data, voice, SMS, or monetary credit

Account Details

The Account section displays customer information and allows limited self-service updates.

Self-Care Portal - Account Details

Editable Fields:

Customers can update:

- Email address (requires verification)
- Phone number
- Password
- Notification preferences

View-Only Information:

- Customer ID
- Account creation date
- Customer type (Individual/Business)
- Sites (addresses)
- Linked contacts

Updating Account Information:

1. Navigate to **Account → Details**
2. Click **"Edit"** next to the field to update
3. Enter new information
4. Click **"Save Changes"**
5. For email changes, verify via link sent to new address

Security Features:

- Password changes require current password
- Email changes require verification
- Activity logged for audit trail
- 2FA settings (if enabled)

Services Overview

The Services section shows all active and inactive services for the customer.

Self-Care Portal - Services List

Service Card Display:

Each service displays:

- **Service Name** - Human-readable identifier (e.g., "Mobile - +44 7700 900123")
- **Product** - Plan or product name
- **Status** - Active, Suspended, Expired, Cancelled
- **Created Date** - Service activation date
- **Expiry Date** - When service expires (if applicable)
- **Monthly Cost** - Recurring charge
- **Auto-Renewal** - Enabled/Disabled indicator

Service Actions:

- **View Usage** - See balance consumption (Data, Voice, SMS, Monetary)
- **Top-Up** - Add credit or data
- **Add-ons** - Purchase additional features
- **Modify** - Change service parameters (if allowed)
- **View Details** - See full service configuration

Service Status Indicators:

- � **Active** - Service is operational
- � **Expiring Soon** - Renews or expires in <7 days
- � **Suspended** - Service temporarily disabled (payment issue, manual suspension)
- ● **Expired** - Service no longer active

Usage Tracking

Customers can monitor their usage across all balance types in real-time.

Data Usage

View data consumption with detailed breakdowns by bucket and expiry.

Self-Care Portal - Data Usage

Data Usage Display:

- **Total Balance** - All data buckets combined
- **Used This Period** - Consumption since last renewal
- **Progress Bar** - Visual representation of consumption
- **Expiry Information** - When each bucket expires
- **Bucket Breakdown** - Multiple data buckets with priority order

Bucket Priority:

When multiple data buckets exist (e.g., base plan + top-ups), they consume in weight order:

- **Weight 10** - Consumed first (typically promotional/bonus data)
- **Weight 20** - Consumed second (typically top-up data)
- **Weight 30** - Consumed last (typically base plan data)

Voice Usage

Track call minutes remaining across all voice buckets.

Self-Care Portal - Voice Usage

Voice Usage Display:

- **Remaining Minutes** - Total voice balance
- **Used Minutes** - Consumption this period
- **Call History** - Recent calls (if enabled)
- **Expiry Dates** - When voice buckets expire
- **International Minutes** - Separate tracking (if applicable)

Usage Breakdown:

- On-Net Calls - Calls within same network
- Off-Net Calls - Calls to other networks
- International Calls - Calls outside country
- Premium Numbers - Special rate numbers

SMS Usage

Monitor SMS message allowances and consumption.

SMS Usage Display:

- **Remaining Messages** - SMS balance
- **Used This Month** - Messages sent
- **MMS Included** - Whether MMS counts toward balance
- **International SMS** - Separate tracking (if applicable)

Monetary Balance

View prepaid credit balance for pay-as-you-go services.

Monetary Display:

- **Current Balance** - Available credit
- **Last Top-Up** - Most recent recharge amount and date
- **Expiry Date** - When balance expires (if applicable)
- **Auto-Recharge** - Enabled/Disabled status

Billing Management

Customers can view invoices, transactions, and manage payment methods.

Invoices

View and pay outstanding invoices directly from the portal.

Self-Care Portal - Invoices

Invoice List Display:

- **Invoice Number** - Unique identifier
- **Date** - Invoice creation date
- **Due Date** - Payment deadline
- **Amount** - Total invoice amount
- **Status** - Paid, Unpaid, Overdue
- **Actions** - Download PDF, Pay Online

Paying an Invoice:

1. Navigate to **Billing → Invoices**
2. Find unpaid invoice in list
3. Click **"Pay Now"** button
4. Select payment method (saved card or new card)
5. Confirm payment
6. Receive confirmation email

Downloading Invoices:

1. Click **"Download"** icon next to invoice
2. PDF downloads with full invoice details
3. Saved for tax records and documentation

Transactions

View complete transaction history including charges, credits, and payments.

Transaction Display:

- **Date** - Transaction creation date
- **Description** - What the charge/credit is for
- **Amount** - Charge (positive) or credit (negative)
- **Invoice** - Which invoice includes this transaction
- **Status** - Invoiced or Uninvoiced

Payment Methods

Manage saved credit cards for automatic billing and online payments.

Payment Method Management:

- **Add Card** - Securely add new credit/debit card via Stripe
- **Set Default** - Choose primary payment method
- **Remove Card** - Delete expired or unused cards
- **Card Details** - View last 4 digits, expiry, card brand

See Payment Methods <payment_methods> for detailed payment management documentation.

Top-Up / Recharge

Purchase additional data, voice, SMS, or monetary credit instantly.

Top-Up Process:

1. Navigate to **Services** → Select service → **"Top-Up"**
2. Choose top-up product from catalog
3. Select amount (preset options or custom)
4. Review cost and expiry information
5. Select payment method
6. Confirm purchase
7. Balance updated immediately

Available Top-Ups:

- **Data Top-Ups** - 1GB, 5GB, 10GB, 20GB, 50GB options
- **Voice Top-Ups** - Additional call minutes
- **SMS Bundles** - Message packs
- **Monetary Credit** - Prepaid balance (£5, £10, £20, £50, £100)

See Top-Up & Recharge <features_topup_recharge> for detailed top-up workflows.

Service Add-Ons

Purchase additional features and enhancements for existing services.

Available Add-Ons:

- **International Roaming** - Enable service abroad
- **Static IP Address** - Fixed IP for home internet
- **Premium Content** - IPTV channels, streaming services

- **Hardware Upgrades** - Modem rental, set-top boxes
- **Speed Boosts** - Temporary bandwidth increases

Purchasing Add-Ons:

1. Navigate to **Services** → Select service → **"Add-Ons"**
2. Browse available add-ons for this service type
3. Click **"Add to Service"** on desired add-on
4. Review cost (one-time + recurring)
5. Confirm purchase
6. Add-on provisioned automatically

See `Modifying Services - Add-ons <csa_modify>` for add-on management details.

Notifications & Alerts

Customers receive automated notifications for important events:

Email Notifications:

- Invoice generated and ready for payment
- Payment received confirmation
- Service expiry warnings (7 days, 3 days, 1 day)
- Low balance alerts (data, voice, monetary)
- Service activation/deactivation
- Password reset requests
- Security alerts (login from new device)

In-Portal Alerts:

- Unpaid invoices
- Expiring services
- Low data warnings (10% remaining)
- Payment method expiring
- Required actions (verify email, update payment method)

Notification Preferences:

Customers can configure:

- Email notification frequency
- SMS alerts (if enabled)
- Alert thresholds (e.g., notify when <20% data remaining)
- Notification categories (billing, usage, service)

Self-Service Limitations

Some operations require staff assistance:

Requires Customer Service:

- Changing customer type (Individual ↔ Business)
- Transferring services between customers
- Cancelling services (may be self-service if enabled)
- Disputing invoices
- Requesting refunds
- Changing primary contact
- Complex provisioning issues

Security Restrictions:

- Cannot view or modify other users' accounts
- Limited to own customer data (tenant isolation)
- Cannot access administrative functions
- Cannot void invoices or transactions
- Cannot modify service configuration (only add-ons/top-ups)

Staff Impersonation Access

Support staff can access the Self-Care Portal as a customer for troubleshooting.

Impersonation Process:

1. Navigate to **Customers** → Select customer
2. Go to **Contacts** tab
3. Find the contact linked to user account
4. Click **"Login as User"** button
5. New tab opens with customer's Self-Care Portal view
6. All actions are logged and attributed to impersonated user
7. Staff sees exactly what customer sees

Login as User - Impersonation

Use Cases:

- **Troubleshooting** - Reproduce customer-reported issues
- **Verification** - Confirm service configurations appear correctly
- **Training** - Demonstrate portal features
- **Support** - Guide customer through portal while viewing their screen

Security & Auditing:

- Requires `can_impersonate` or `admin` permission

- All actions logged to audit trail
- Customer sees impersonation in activity log
- Session timeout after inactivity
- Cannot change customer password while impersonating

See Customer Care - User Impersonation <customer_care> for complete impersonation documentation.

Mobile Responsiveness

The Self-Care Portal is fully responsive and optimized for mobile devices.

Mobile Features:

- Touch-optimized navigation
- Simplified layouts for small screens
- Swipe gestures for carousel navigation
- Mobile-friendly forms and inputs
- QR code scanning for eSIM provisioning
- One-tap phone calling
- GPS integration for address autocomplete

Progressive Web App (PWA):

- Install as app on home screen
- Offline viewing of recent data
- Push notifications (if enabled)
- Fast loading with service workers

Password Reset & Account Recovery

Customers can reset forgotten passwords without calling support.

Self-Service Password Reset:

1. Click **"Forgot Password"** on login page
2. Enter email address
3. Receive password reset email (valid 1 hour)
4. Click link in email
5. Enter new password (must meet complexity requirements)
6. Submit and login with new password

Password Requirements:

- Minimum 8 characters
- At least one uppercase letter
- At least one lowercase letter

- At least one number
- At least one special character (!@#$%^&*)

Account Lockout:

After 5 failed login attempts:

- Account locked for 30 minutes
- Password reset email sent automatically
- Security notification sent to registered email

Best Practices for Customers

Security Recommendations:

1. Enable 2FA for enhanced security
2. Use unique, strong password
3. Keep email address current for notifications
4. Set up default payment method for auto-renewals
5. Monitor usage regularly to avoid overage charges
6. Save backup codes in secure location (if 2FA enabled)
7. Logout after using shared/public computers

Usage Management:

1. Enable low balance alerts (10-20% remaining)
2. Top-up before balances expire to avoid service interruption
3. Review monthly invoices for unexpected charges
4. Update payment methods before cards expire
5. Monitor data usage throughout month to avoid throttling

Support Escalation:

If self-service doesn't resolve the issue:

1. Check knowledge base / help articles (if available)
2. Review activity log for recent changes
3. Contact customer service via phone, email, or chat
4. Provide customer ID and service details for faster resolution

API Access

Customers with technical requirements can use the API directly.

API Key Generation:

Available for business customers or upon request:

1. Navigate to **Account → API Access**
2. Click **"Generate API Key"**
3. Set permissions (read-only or read-write)
4. Set expiry date
5. Save API key securely (shown only once)

API Use Cases:

- Automated usage monitoring
- Integration with internal billing systems
- Programmatic top-ups
- Service provisioning via scripts
- Data exports for analytics

See `API Documentation <concepts_api>` for endpoint details and examples.

Frequently Asked Questions

Q: Why can't I see all my services?

A: Ensure you're logged in with the correct account. If you have multiple customer accounts, each has separate services. Contact support to merge accounts if needed.

Q: My payment failed but I was charged. What do I do?

A: Check your bank statement for pending charges. If charge appears but invoice still shows unpaid, contact support with transaction reference number.

Q: How do I cancel a service?

A: Navigate to service details and click "Cancel Service" (if self-service enabled). Otherwise, contact customer service to process cancellation.

Q: Can I transfer a service to another person?

A: No, service transfers require customer service assistance for security and compliance reasons.

Q: Why is my data balance decreasing faster than expected?

A: Check background app updates, video streaming quality, and automatic cloud backups. Review usage breakdown in portal for detailed consumption.

Q: I lost my 2FA device. How do I regain access?

A: Use backup codes to login, then disable and re-enable 2FA. If no backup codes, contact support for 2FA reset (requires identity verification).

Q: Can I pay an invoice without logging in?

A: Guest invoice payment may be available via direct invoice link. Otherwise, login required for security.

Q: How do I download all my invoices at once?

A: Currently requires downloading each invoice individually. For bulk downloads, contact support or use API if available.

Related Documentation

- Customer Care - User Impersonation <customer_care> - Staff troubleshooting access
- Payment Methods <payment_methods> - Managing cards and payment
- Top-Up & Recharge <features_topup_recharge> - Purchasing additional credit
- Service Usage <csa_service_usage> - Understanding balance tracking
- Billing Overview <billing_overview> - Billing concepts and invoices
- Authentication Flows <authentication_flows> - Login and security
- 2FA <2fa> - Two-factor authentication setup
- Service Modifications <csa_modify> - Adding features and add-ons