Authsome Service Interface Contracts

Version: 1.1
Last Updated: October 31, 2025
Purpose: This document defines the functional contracts for all Authsome services. Developers can implement these services in any programming language or framework.

Table of Contents

Overview

Architecture

Each service is **independent** and must implement the functions defined in this contract. Services communicate through well-defined interfaces - the exact communication mechanism (REST, gRPC, direct calls) is implementation-specific.

Key Principles

- **Stateless Operations:** Each function call is independent
- **Clear Responsibilities:** Each service has a single, well-defined purpose
- **JSON Serialization:** All data types must be serializable to JSON
- **Consistent Error Handling:** Use standardized error types across all services

---

Common Data Types

Timestamps

- **Type:** `Long` (64-bit integer)
- **Format:** Milliseconds since Unix epoch
- **Example:** `1698710400000` (represents October 31, 2023, 00:00:00 UTC)

Identifiers

- **Type:** `String`
- **Format:** UUID (36 characters with hyphens)
- **Example:** `"550e8400-e29b-41d4-a716-446655440000"`

Enumerations

IdentityType (Tenant Service)

```
EMAIL       - Email address (e.g., user@example.com)
USERNAME    - Username (e.g., john_doe)
USER_ID     - User's unique identifier (UUID)
```

IdentityType (Notifier Service)

```
EMAIL       - Email notification channel
```

Note: SMS and PUSH_NOTIFICATION may be added in future versions

OtpType

```
NUMERIC       - Only digits 0-9 (e.g., "1234")
ALPHABETIC    - Only letters a-z, A-Z (e.g., "ABCD")
ALPHANUMERIC  - Mix of digits and letters (e.g., "A1B2")
```

Tenant Service

Purpose

Manages tenant (user) accounts including creation, lookup, and identity management.

Function 1: `getTenantByIdentity`

Description: Find a tenant using any of their registered identities.

Parameters:

Parameter	Type	Required	Description
`identityType`	`IdentityType`	Yes	Type of identity to search
`identity`	`String`	Yes	The identity value

Returns: `FetchedTenant` or `null`

FetchedTenant Structure:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "username": "john_doe",
  "createdAt": 1698710400000,
  "updatedAt": 1698710400000
}
```

Example Usage:

```
// Search by email
input: {
  identityType: "EMAIL",
  identity: "user@example.com"
}

// Returns tenant if found, null otherwise
output: {
  id: "550e8400-e29b-41d4-a716-446655440000",
  username: "john_doe",
  createdAt: 1698710400000,
  updatedAt: 1698710400000
}
```

Error Cases:

- `VALIDATION_ERROR` : Invalid `identityType` value
- `VALIDATION_ERROR` : Empty or null `identity`

Function 2: `getTenantByUsername`

Description: Find a tenant by their username.

Parameters:

Parameter	Type	Required	Description

Parameter	Type	Required	Description
`username`	`String`	Yes	Username to search for

Returns: `FetchedTenant` or `null`

Example Usage:

```
input: {
  username: "john_doe"
}

// Returns tenant if found, null otherwise
output: {
  id: "550e8400-e29b-41d4-a716-446655440000",
  username: "john_doe",
  createdAt: 1698710400000,
  updatedAt: 1698710400000
}
```

Error Cases:

- `VALIDATION_ERROR` : Empty or null `username`

Function 3: `createTenant`

Description: Creates a new tenant account with username and password.

Parameters:

Parameter	Type	Required	Description
`username`	`String`	Yes	Unique username for the tenant
`rawPassword`	`String`	Yes	Plain text password (will be hashed)

Returns: `FetchedTenant`

Example Usage:

```
input: {
  username: "john_doe",
  rawPassword: "SecurePass123!"
}

output: {
  id: "550e8400-e29b-41d4-a716-446655440000",
  username: "john_doe",
  createdAt: 1698710400000,
  updatedAt: 1698710400000
}
```

Implementation Notes:

- Password MUST be hashed using bcrypt (cost factor 10+) or argon2id
- Username must be unique across all tenants
- Generate UUID for tenant ID
- Set `createdAt` and `updatedAt` to current timestamp

Error Cases:

- `VALIDATION_ERROR` : Empty or null `username` or `rawPassword`
- `CONFLICT` : Username already exists

Function 4: `addIdentityForTenant`

Description: Associates an identity (email, username) with an existing tenant.

Parameters:

Parameter	Type	Required	Description
`tenantId`	`String`	Yes	UUID of the tenant
`identityType`	`IdentityType`	Yes	Type of identity to add
`identity`	`String`	Yes	Identity value

Returns: `FetchedTenantIdentity`

FetchedTenantIdentity Structure:

```
{
  "id": "identity-uuid-here",
  "tenantId": "550e8400-e29b-41d4-a716-446655440000",
  "identityType": "EMAIL",
  "identity": "user@example.com"
}
```

Example Usage:

```
input: {
  tenantId: "550e8400-e29b-41d4-a716-446655440000",
  identityType: "EMAIL",
  identity: "user@example.com"
}

output: {
  id: "identity-abc123",
  tenantId: "550e8400-e29b-41d4-a716-446655440000",
  identityType: "EMAIL",
  identity: "user@example.com"
}
```

Implementation Notes:

- The combination of `identityType` + `identity` must be unique
- Generate UUID for identity record ID

Error Cases:

- `VALIDATION_ERROR` : Invalid parameters
- `NOT_FOUND` : Tenant with given `tenantId` doesn't exist
- `CONFLICT` : Identity already associated with another tenant

OTP Service

Purpose

Generates, stores, and manages one-time passwords for authentication workflows.

Function 1: `generateAndSaveOtp`

Description: Generates a random OTP and stores it with metadata for later verification.

Parameters:

Parameter	Type	Required	Description
`otpType`	`OtpType`	Yes	Type of OTP to generate
`otpLength`	`Integer`	Yes	Length of OTP (typically 4-8)
`minNumber`	`Integer`	Yes	Min numeric chars (-1 = ignore)

Parameter	Type	Required	Description
`minAlphabet`	`Integer`	Yes	Min alphabetic chars (-1 = ignore)
`maxNumber`	`Integer`	Yes	Max numeric chars (-1 = ignore)
`maxAlphabet`	`Integer`	Yes	Max alphabetic chars (-1 = ignore)
`expiresAfterSecond`	`Integer`	Yes	Expiry time in seconds
`context`	`String`	Yes	Context identifier
`metadata`	`Map<String, String>`	Yes	Additional data to store

Returns: `FetchedOtp`

FetchedOtp Structure:

```
{
  "id": "otp-550e8400-e29b-41d4-a716-446655440000",
  "code": "1234",
  "context": "AUTHSOME_TENANT_SIGNUP",
  "expiresAt": 1698710700000,
  "metadata": {
    "identity": "user@example.com",
    "identityType": "EMAIL",
    "username": "john_doe",
    "password": "encryptedString"
  }
}
```

Common Usage Example (Signup OTP):

```
input: {
  otpType: "NUMERIC",
  otpLength: 4,
  minNumber: -1,      // Not applicable for NUMERIC
  minAlphabet: -1,    // Not applicable for NUMERIC
  maxNumber: -1,      // Not applicable for NUMERIC
  maxAlphabet: -1,    // Not applicable for NUMERIC
  expiresAfterSecond: 300,  // 5 minutes
  context: "AUTHSOME_TENANT_SIGNUP",
  metadata: {
    "identity": "user@example.com",
    "identityType": "EMAIL",
    "username": "john_doe",
    "password": "encrypted_password_here"
  }
}

output: {
  id: "otp-550e8400-e29b-41d4-a716-446655440000",
  code: "1234",
  context: "AUTHSOME_TENANT_SIGNUP",
  expiresAt: 1698710700000,
  metadata: { /* same as input */ }
}
```

Implementation Notes:

- Use cryptographically secure random generator (e.g., `SecureRandom` in Java, `secrets` in Python)
- Calculate `expiresAt` = current timestamp + (expiresAfterSecond * 1000)
- For NUMERIC OTPs, ignore min/max alphabet constraints
- Store the OTP securely (consider hashing if very sensitive)
- The `id` + `context` combination should be unique

Parameter Rules:

- `-1` means "ignore this constraint"
- `0` is invalid (throw VALIDATION_ERROR)

- Positive values apply the constraint

Error Cases:

- `VALIDATION_ERROR`: Invalid `otpType`, `otpLength <= 0`, `expiresAfterSecond <= 0`
- `VALIDATION_ERROR`: Conflicting constraints (e.g., `minNumber > otpLength`)
- `VALIDATION_ERROR`: Empty or null `context`

Function 2: `getOtpById`

Description: Retrieves a stored OTP by its ID.

Parameters:

Parameter	Type	Required	Description
`id`	`String`	Yes	OTP identifier (UUID)

Returns: `FetchedOtp` or `null`

Example Usage:

```
input: {
  id: "otp-550e8400-e29b-41d4-a716-446655440000"
}

output: {
  id: "otp-550e8400-e29b-41d4-a716-446655440000",
  code: "1234",
  context: "AUTHSOME_TENANT_SIGNUP",
  expiresAt: 1698710700000,
  metadata: { /* stored metadata */ }
}
```

Implementation Notes:

- Return `null` if OTP doesn't exist
- Do NOT automatically delete expired OTPs in this function
- Consider implementing automatic cleanup via background job

Error Cases:

- `VALIDATION_ERROR`: Empty or null `id`

Notifier Service

Purpose

Sends notifications to users through various channels (currently email only).

Function 1: `sendNotification`

Description: Sends a notification via the specified channel.

Parameters:

Parameter	Type	Required	Description
`identityType`	`IdentityType`	Yes	Notification channel (EMAIL)
`identity`	`String`	Yes	Recipient address
`subject`	`String`	Yes	Notification subject/title
`content`	`String`	Yes	Notification body/message

Returns: `void` (throws error on failure)

Example Usage:

```
input: {
  identityType: "EMAIL",
  identity: "user@example.com",
  subject: "OTP to create authsome account",
  content: "Your OTP to create your Authsome account is: 1234"
}

// No return value on success
// Throws error on failure
```

Implementation Notes:

- For EMAIL: Use SMTP or email service provider (SendGrid, AWS SES, etc.)
- Should be asynchronous if possible (don't block caller)
- Implement retry logic for transient failures
- Log all sent notifications for audit trail
- Consider rate limiting (max 10 per identity per hour)

Error Cases:

- `VALIDATION_ERROR` : Invalid `identityType`
- `VALIDATION_ERROR` : Empty or null parameters
- `INTERNAL_ERROR` : Failed to send (network error, invalid recipient, etc.)

Error Handling

Error Types

Error Type	When to Use	HTTP Status
`VALIDATION_ERROR`	Invalid input, missing fields, constraint violations	400
`NOT_FOUND`	Entity doesn't exist	404
`CONFLICT`	Duplicate resource (username/email exists)	409
`EXPIRED`	OTP expired, session timeout	410
`UNAUTHORIZED`	Invalid credentials, invalid token	401
`INTERNAL_ERROR`	Database errors, unexpected exceptions	500

Error Response Structure

```
{
  "errorType": "CONFLICT",
  "message": "Username already exists",
  "details": {
    "field": "username",
    "value": "john_doe"
  }
}
```

Example Error Scenarios:

```
// Username already taken
{
  errorType: "CONFLICT",
  message: "Username already exists",
  details: {
    field: "username",
    value: "john_doe"
  }
}

// Invalid OTP
{
  errorType: "VALIDATION_ERROR",
  message: "Invalid OTP code",
  details: {
    field: "otp"
  }
}

// OTP expired
{
  errorType: "EXPIRED",
  message: "OTP has expired",
  details: {
    expiresAt: 1698710700000,
    currentTime: 1698711000000
  }
}
```

Implementation Guide

Quick Start Checklist

Tenant Service

- [ ] Implement database schema for tenants and identities
- [ ] Hash passwords using bcrypt (cost 10+) or argon2id
- [ ] Ensure username uniqueness constraint
- [ ] Ensure identity + identityType uniqueness constraint
- [ ] Implement all 4 functions

OTP Service

- [ ] Implement database schema for OTPs
- [ ] Use cryptographically secure random generator
- [ ] Implement OTP generation logic with constraints
- [ ] Set up background job for expired OTP cleanup
- [ ] Consider rate limiting (5 OTPs per identity per hour)

Notifier Service

- [ ] Configure email SMTP settings
- [ ] Implement email template system (optional)
- [ ] Add retry logic for failures
- [ ] Set up notification logging
- [ ] Consider rate limiting (10 notifications per identity per hour)

Security Requirements

Password Security

```
✓ NEVER store passwords in plain text
✓ Use bcrypt (cost 10+) or argon2id
✗ Don't use MD5, SHA1, or simple hashing
```

OTP Security

```
✓ Use SecureRandom (Java) or secrets (Python)
✓ Generate unpredictable codes
✗ Don't use Math.random() or predictable patterns
```

Data Encryption

```
✓ Use TLS/HTTPS for network communication
✓ Encrypt sensitive data at rest (AES-256)
✓ Store encryption keys securely (not in code)
```

Example Signup Flow

This shows how all three services work together:

```
// Step 1: User submits signup form
POST /api/v1/authsome-service/signup
{
  identityType: "EMAIL",
  identity: "user@example.com",
  username: "john_doe",
  password: "SecurePass123!"
}

// Backend orchestration:
// 1. Check if identity exists
tenant = tenantService.getTenantByIdentity("EMAIL", "user@example.com")
if (tenant != null) throw CONFLICT

// 2. Check if username exists
tenant = tenantService.getTenantByUsername("john_doe")
if (tenant != null) throw CONFLICT

// 3. Encrypt password temporarily
encryptedPassword = encrypt("SecurePass123!")

// 4. Generate OTP
otp = otpService.generateAndSaveOtp(
  "NUMERIC", 4, -1, -1, -1, -1, 300,
  "AUTHSOME_TENANT_SIGNUP",
  {
    identity: "user@example.com",
    identityType: "EMAIL",
    username: "john_doe",
    password: encryptedPassword
  }
)

// 5. Send OTP email
notifierService.sendNotification(
  "EMAIL",
  "user@example.com",
  "OTP to create authsome account",
  "Your OTP is: " + otp.code
)

// 6. Return token to client
return { token: otp.id }
```

```
// ==================================

// Step 2: User submits OTP for verification
PUT /api/v1/authsome-service/signup/1234
Header: Signup-Token: otp-550e8400-...

// Backend orchestration:
// 1. Get stored OTP
otp = otpService.getOtpById(token)
if (otp == null) throw VALIDATION_ERROR

// 2. Verify OTP code
if (otp.code != "1234") throw VALIDATION_ERROR

// 3. Check if expired
if (currentTime > otp.expiresAt) throw EXPIRED

// 4. Verify context
if (otp.context != "AUTHSOME_TENANT_SIGNUP") throw VALIDATION_ERROR

// 5. Extract signup data
identity = otp.metadata.identity
username = otp.metadata.username
password = decrypt(otp.metadata.password)

// 6. Create tenant
tenant = tenantService.createTenant(username, password)

// 7. Add identity
tenantService.addIdentityForTenant(
  tenant.id,
  "EMAIL",
  identity
)

// Done! User account created
```

Testing Guide

Unit Tests

- Test each function with valid inputs
- Test each function with invalid inputs (null, empty, wrong type)
- Test all error cases
- Test edge cases (expired OTPs, duplicate usernames)

Integration Tests

- Test with real database
- Test with real email provider (or mock)
- Test complete signup flow end-to-end

Performance Benchmarks

- OTP generation: < 100ms
- Tenant lookup: < 50ms
- Email sending: Non-blocking (async)

Future Enhancements

Planned features for future versions:

- SMS notifications (`IdentityType.SMS`)
- Push notifications (`IdentityType.PUSH`)
- Multi-factor authentication (MFA)

- OTP resend functionality
- Password reset workflow
- Tenant profile updates
- Comprehensive audit logging
- Dedicated rate limiting service

Document Version: 1.1
Last Updated: October 31, 2025
Status: Active