OTP Service Technical Documentation

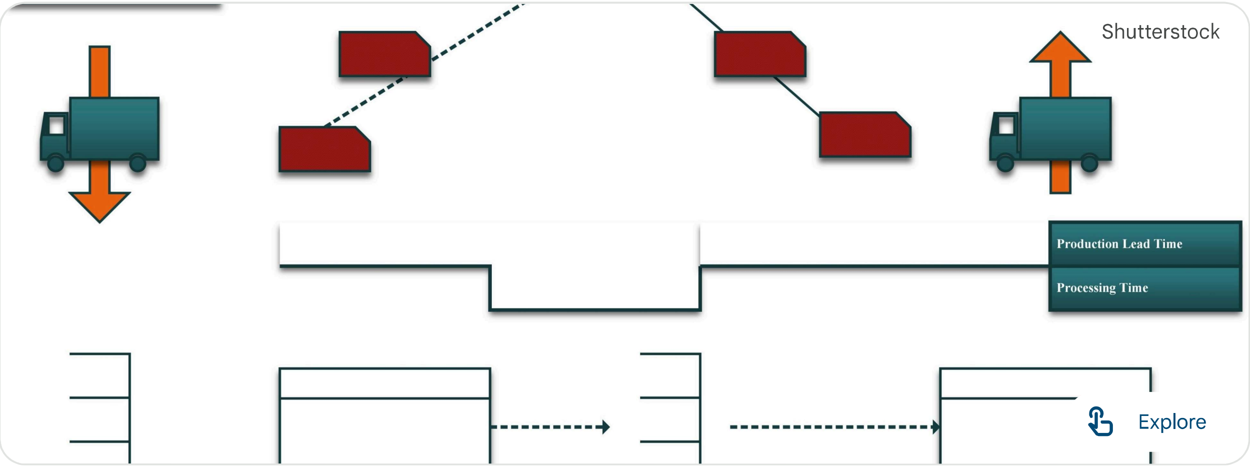
1. Executive Summary

The OTP (One-Time Password) Service is a Spring Boot application designed to handle the generation, verification, and distribution of secure authentication codes. It features a non-blocking architecture for notifications, Redis-based state management for security and scalability, and a resilient integration layer for external delivery providers (Salesforce) protected by a Circuit Breaker.

2. High-Level Architecture

The application is layered into Controllers, Services, Clients, and Configuration components.

- **API Layer (`OtpController`):** REST endpoints for generating, verifying, and resending OTPs.
- **Service Layer (`OtpService` , `ValidationService`):** Business logic for OTP creation, hashing, and request validation.
- **Persistence Layer (Redis):** Stores hashed OTPs and manages resend cooldowns using TTLs.
- **Messaging Layer (`NotificationService` , `AzureServiceBusClient`):** Asynchronous dispatch of OTP notifications.
- **Integration Layer (`SalesforceClient`):** External gateway for delivering messages, protected by Resilience4j.



3. Core Workflows

3.1 OTP Generation Flow

The generation process is split into a synchronous request validation phase and an asynchronous notification phase to ensure low latency for the API consumer.

Sequence:

1. **Request:** Client POSTs to `/otp/generate` .

2. **Validation:** `ValidationService` checks input format (Regex for email/phone) and determining the unique identifier.
3. **Generation:** `OtpService` generates a random 6-digit code.
4. **Storage:**
    - **Hashing:** The OTP is hashed (SHA-256) before storage.
    - **Redis:** The hash is stored with a TTL (180s).
    - **Cooldown:** A separate Redis key is set to prevent immediate re-generation (30s cooldown).
5. **Async Dispatch:** The service hands off the payload to `NotificationService` .
6. **Response:** The API returns 200 OK immediately with the identifier, while the notification process continues in the background.

3.2 Notification & Delivery Flow (Async)

This flow handles the actual delivery of the OTP to the user.

1. **Publishing:** `NotificationService` wraps the request in a `OtpNotificationDto` (with a correlation ID) and calls `AzureServiceBusClient` .
2. **Queue Simulation:** `AzureServiceBusClient` simulates publishing to a queue. It then immediately acts as the "consumer" to process the message.
3. **External Call:** The client calls `SalesforceClient.sendOtp()` .
4. **Resilience:** The `SalesforceClient` call is wrapped in a Circuit Breaker.
    - **Success:** Logged as successful.
    - **Failure:** Caught by `AzureServiceBusClient` . The error is logged, but the operation is **not retried** (fire-and-forget design).

3.3 OTP Verification Flow

1. **Request:** Client POSTs to `/otp/verify` with `identifier` and `otp` .
2. **Retrieval:** System fetches the stored hash from Redis using the identifier.
3. **Hashing:** The incoming OTP is hashed using the same algorithm.
4. **Comparison:**
    - **Match:** The OTP key and Cooldown key are deleted from Redis. Success returned.
    - **Mismatch/Missing:** Throws `InvalidOtpException` or `OtpNotFoundException` .

4. Resilience & Circuit Breaker Logic

The integration with Salesforce is protected by **Resilience4j**. This prevents cascading failures when the downstream service is unstable.

4.1 Configuration (Simplified)

- **Failure Threshold:** 50% (If 5 out of 10 calls fail, the circuit opens).
- **Sliding Window:** 10 calls.
- **Wait Duration:** 30 seconds (Time to stay OPEN before testing recovery).
- **Permitted Probes:** 3 calls (Allowed in HALF-OPEN state to check stability).

4.2 State Machine Behavior

1. **CLOSED (Healthy):** All requests go through. Failures are counted towards the threshold.
2. **OPEN (Failing):**
    - Triggered when the failure rate exceeds 50%.
    - **Behavior:** Requests to `SalesforceClient` are blocked immediately.
    - **Exception:** `CallNotPermittedException` is thrown instantly (Fail Fast).
    - **Fallback:** The application logs the specific error "Salesforce is down - NOT ATTEMPTING".
3. **HALF-OPEN (Recovery):**
    - After 30 seconds, the circuit allows 3 "probe" requests.
    - If these succeed, the circuit resets to **CLOSED**.
    - If they fail, it reverts to **OPEN**.

5. Implementation Details

5.1 Redis Data Model

The service uses two distinct key patterns to manage state:

Key Type	Pattern	Value	TTL	Purpose
OTP Key	`otp:{identifier}`	`SHA-256(123456)`	180s	Stores the valid OTP hash.
Cooldown Key	`otp:resend:{identifier}`	`Timestamp (Long)`	180s	Prevents spamming generate requests.

5.2 Simulation Features

To facilitate testing without real external dependencies, the service includes simulation capabilities configurable via `application.properties` and the `CircuitBreakerTestController` .

- **Salesforce Simulation:** Can be configured to randomly fail (`failure-rate`) or delay responses (`delay-ms`) to test timeouts.
- **Azure Service Bus Simulation:** Mocks the queue behavior, processing messages in-memory immediately after "publishing".

5.3 Error Handling

- **Validation Errors:** 400 Bad Request (e.g., invalid phone format).
- **Logic Errors:** 400/404 (e.g., OTP not found, Cooldown active).
- **Downstream Errors:** 500 Internal Server Error (or handled gracefully in async flows).
  - *Note:* In the async notification flow, errors are logged but do not affect the synchronous HTTP response returned to the user.

6. Testing Guide

Testing the Circuit Breaker

1. **Force Failures:** Use `POST /circuit-breaker/simulate-failure-rate?rate=1.0` to set a 100% failure rate.
2. **Trigger Open State:** Send 5+ generation requests.
3. **Observe Fail Fast:** Subsequent requests will fail instantly without the configured delay, indicating the circuit is OPEN.
4. **Recover:** Set rate to `0.0`, wait 30 seconds, and send requests to observe the transition from HALF-OPEN back to CLOSED.