

SYSTEM TECHNICAL DOCUMENTATION

Immigration Management System

Document Version: 1.0

Client: Immigration Department

Project Code: WMJ16414

Status: Final Release

1. SYSTEM OVERVIEW

The Immigration Management System (IMS) is an enterprise-grade digital platform engineered to modernize border control operations through automation, real-time data processing, and robust security protocols. It serves as a centralized hub for managing the end-to-end lifecycle of traveler entry and departure.

1.1 Business Challenges Addressed

- **Manual Bottlenecks:** Replaces inefficient paper-based arrival/departure cards with a high-speed digital submission engine.
- **Border Security:** Mitigates security risks by integrating a real-time "Blacklist" verification system that checks every traveler against restricted databases instantly.
- **Operational Visibility:** Provides government administrators with live analytics on traveler volume, nationality trends, and port traffic patterns.
- **Data Integrity:** Centralizes fragmented immigration records into a unified, secure Oracle-backed data warehouse.

1.2 Target User Personas

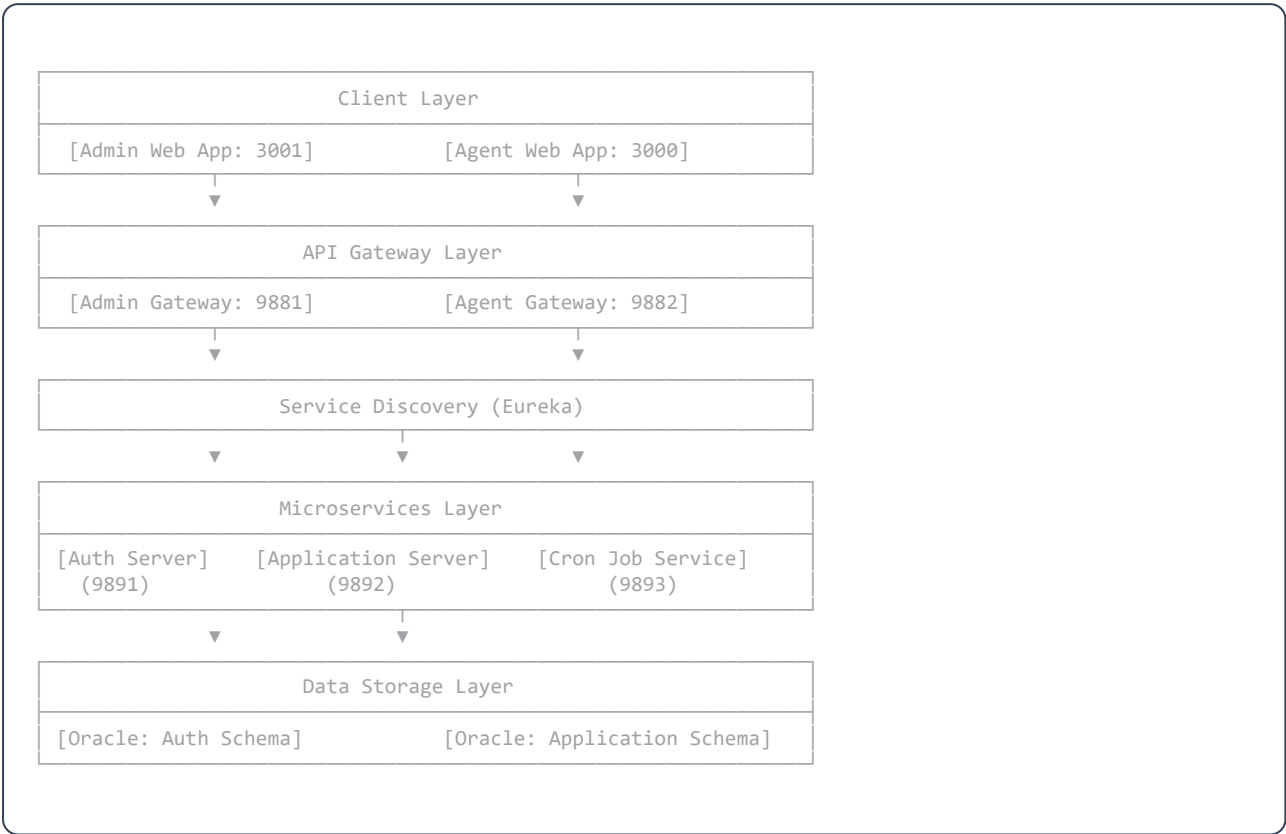
USER TYPE	CORE RESPONSIBILITIES
Travelers	Submit digital arrival/departure forms, provide health/goods declarations, and download QR-coded entry tickets.
Border Agents	Utilize scanner interfaces to verify traveler QR codes, review submission details, and finalize entry/exit status.
Admin Staff	Manage user accounts (Agents/Staff), review flagged applications, and maintain location master data (Cities/Ports).
Super Admins	Complete system oversight, security configuration, audit log analysis, and high-level statistics reporting.

1.3 Core Capabilities

- **Smart Form Submission:** Logic-driven forms that adapt based on travel route (Air, Land, Sea) and age (Guardian logic for minors).
- **Secure Verification:** QR and Barcode generation for tickets, enabling contactless verification at ports of entry.
- **Automated Notifications:** Real-time email delivery for credentials and travel tickets via background workers.
- **Dynamic Reporting:** On-demand generation of Excel and PDF reports for immigration traffic and security audits.

2. HIGH LEVEL ARCHITECTURE

The system utilizes a **Microservices Architecture** pattern based on the Spring Cloud ecosystem. This ensures independent scaling, high availability, and loose coupling between business domains.



2.1 Service Boundaries

SERVICE NAME	FUNCTIONAL BOUNDARY & RESPONSIBILITY
Config Server (9888)	Acts as the central configuration repository. Provides encrypted properties to all services from a Git-based back-end.
Discover (9811)	Netflix Eureka server for service registration, heartbeating, and client-side load balancing.
Authorization (9891)	OAuth2 Server. Manages user authentication, JWT generation, and client-credential security for inter-service calls.
Application (9892)	The core business engine. Manages forms, blacklist logic, master data, PDF generation, and dashboard statistics.
CronJob (9893)	Scheduled task executor. Handles asynchronous email queuing and periodic data cleanup.

2.2 ARCHITECTURE FLOW

The following logical flows describe the interaction between system components during critical business operations.

Form Submission Flow

Traveler → Public API → Application Service → Database → Email Service → Traveler (Ticket)

The traveler accesses the public submission endpoint via the Application Service. The system validates the input, generates a unique reference code, persists the data to Oracle, and triggers an asynchronous email containing the PDF ticket with a QR code.

Authentication Flow

User → Gateway → Authorization Service → Database → JWT Token → User

Users authenticate via the gateway. The Authorization service verifies credentials against the Auth Schema, generates a signed JWT containing roles (e.g., ROLE_AGENT), and returns it. This token is used for subsequent authorized requests.

Form Review Flow

Agent → Agent Gateway → Application Service → Blacklist Check → Database → Status Update

When an agent scans a QR code, the request passes through the Agent Gateway. The Application service retrieves the form, performs a real-time blacklist check, and updates the status to "ARRIVED" or "DEPARTED" upon confirmation.

Statistics & Reporting Flow

Admin → Admin Gateway → Application Service → Data Analytics → Dashboard/Excel

Administrative users request dashboard metrics. The application service performs aggregation queries on the transactional data, calculates trends (Previous month comparison), and serves the data as JSON for the UI or as a downloadable Excel file.

3. TECHNOLOGY STACK

The IMS utilizes a modern, robust stack designed for high throughput and enterprise security requirements.

Backend & Cloud Frameworks

CATEGORY	TECHNOLOGY	ROLE IN SYSTEM
Language	Java	Core programming language (Standard for reliability).
Framework	Spring Boot	Foundation for all microservices logic.
Microservices	Spring Cloud	Distributed configuration, discovery, and routing.
Security	OAuth2 / JWT	Stateless authentication and role-based access.

Data, Storage & Utilities

CATEGORY	TECHNOLOGY	ROLE IN SYSTEM
Database	Oracle Database	Primary storage for transactional and security data.
ORM	Spring Data JPA	Mapping Java objects to Oracle tables (Hibernate).
PDF Engine	wkhtmltopdf	Renders HTML/CSS templates into PDF tickets.
DevOps	Docker	Containerization for environment consistency.

4. REPOSITORY STRUCTURE

The project follows a **Maven Multi-Module** design to enforce separation of concerns and maximize code reuse across the microservices ecosystem.

```
ImmigrationBackend/ └─ ImmigrationParent/ # Master POM (Dependencies & Versions) └─
ImmigrationProperties/ # Shared Property Libraries └─ ImmigrationCore/ # Core Framework
(Shared Logic) | └─ call_api/ # External API Integration Utils | └─ config/ # Common Security
Configs | └─ exception/ # Unified Error Handling | └─ model/ # Global Data Models └─
ImmigrationConfigServer/ # Distributed Config Management └─ ImmigrationDiscover/ # Eureka
Service Registry └─ ImmigrationAuthorization/ # OAuth2 Auth & User Security └─
ImmigrationApplication/ # Main Business & Public API | └─ controller/ # REST Endpoints | └─
factory/ # Business Logic Factories | └─ pdfgenerator/ # Ticket Generation Engine └─
ImmigrationCronJob/ # Background Task Runner └─ ImmigrationGatewayAdmin/ # Entry point for
Admin UI └─ ImmigrationGatewayAgent/ # Entry point for Agent UI └─ DockerDir/ # Docker
Deployment Orchestration
```

4.1 Directory Purpose Explanation

- **ImmigrationCore:** A shared library dependency used by all microservices. It contains the `Validation.java` engine for sanitizing traveler input and `InValidInputFormatException` for standardized error reporting.
- **ImmigrationProperties:** Houses the `application.properties` files used during development and specialized shared configuration strings that are common across environments.
- **ImmigrationApplication:** The largest module containing the actual immigration domain logic. It uses a **Factory Pattern** in the `factory/` directory to separate complex business rules (like Blacklist checking) from the controllers.
- **ImmigrationAuthorization:** Specifically manages the security context. It connects to the Auth schema and handles user registration, password strength validation (Passay), and token lifecycle.
- **DockerDir:** This directory is structured for orchestration. It contains three distinct ``docker-compose.yml`` files (Config, Microservices, Gateways) to manage the strict startup order required by the Spring Cloud environment.

5. APPLICATION COMPONENTS AND MODULES

The Immigration Application is divided into specialized modules to handle the complex requirements of border management.

5.1 Authentication & Authorization Module

Handles secure login via OAuth2. Supports **Password Grant** for user UI login and **Client Credentials** for secure inter-service API calls.

- **Core Files:** OAuth2AuthorizationConfig.java, AuthController.java
- **Logic:** JWT token issuance, refresh logic, and BCrypt credential verification.

5.2 Immigration Form Management Module

The system's heart, managing the digital "Arrival/Departure" cards. Features deep validation for passport data, flight details, and travel purposes.

- **Core Files:** ImmigrationFormDetailEntity.java, ImmigrationFormFactory.java
- **Logic:** Conditional field validation based on Route (Air/Road/Sea).

5.3 User Management Module

Granular RBAC for Super Admins, Staff, and Agents. Allows city-based assignment for border agents.

5.4 Blacklist Management Module

Security filter that stores passport numbers of restricted individuals. Applications are cross-referenced during submission.

5.5 Form Review & Verification

Agent-side interface for scanning QR codes and marking travelers as "Arrived" or "Departed".

5.6 Ticket Generation

Thymeleaf-based PDF engine creating secure tickets with embedded QR/Barcodes.

5.7 Statistics & Reporting

Aggregates transactional data into dashboard metrics and downloadable Excel reports.

5.8 Email Notifications

Asynchronous engine for sending credentials and traveler tickets.

5.9 Master Data

Global registry for Nationalities, States, and Cities (init via GeoJSON).

5.10 Cron Job Module

Background scheduler for system maintenance and email queue processing.

5.11 API Gateway Module

Centralized routing with CORS management and Swagger UI aggregation for the entire microservice cluster.

6. API DOCUMENTATION

All APIs are routed through the Gateways (Ports 9881/9882) and require appropriate OAuth2 scopes.

6.1 Authentication APIs

POST /oauth/token

Generates access/refresh tokens. Requires Basic Auth (Client ID/Secret).

6.2 Immigration Form APIs

POST /app/public/immigration/api/submit-form

Public endpoint for traveler submission. Returns unique reference code.

GET /app/public/immigration/api/get-form-details

Retrieves form data using Reference Code and Passport Number.

6.3 User Management APIs

ENDPOINT	METHOD	PERMISSION REQUIRED
/admin/secured/api/agent-user-management/agentadminuser	POST	ROLE_USER_MANAGEMENT
/admin/secured/api/agent-user-management/getAgentUserList	POST	ROLE_STAFF

6.4 Blacklist Management APIs

POST /admin/secured/api/blacklist-travellers/add

Adds a passport number to the restricted database.

6.5 Form Review APIs (Agent)

POST /agent/secured/api/mark-status

Updates traveler status to "ARRIVED" or "DEPARTED".

6.6 Statistics & Master Data APIs

- **Dashboard Stats:** POST /admin/secured/api/statistics/dashboard
- **Nationality List:** GET /app/public/master/nationality-list

6.8 Ticket APIs

GET /app/public/ticket/download

Returns a PDF byte stream of the traveler ticket.

7. DATA ARCHITECTURE

The system utilizes **Oracle Database 18c** with a two-schema design to separate security credentials from transactional business data.

7.1 Schema Separation

- **c##immiuserAuth:** Stores OAuth2 tokens (access/refresh), client details, and user credentials.
- **c##immiuserApp:** Stores all immigration forms, blacklist records, and master location data.

7.2 Key Entity Relationships

The architecture emphasizes data integrity via strict foreign key constraints:

- **Form ↔ City:** Each submission links to a City Entity for accommodation tracking.
- **Form ↔ Agent:** Verified forms link to the specific Agent ID who scanned the QR code.
- **City ↔ State:** Hierarchical location mapping for reporting by region.

7.3 Immigration Form Details

CATEGORY	PRIMARY FIELDS
Personal	Names, DOB, Passport No, Nationality, Gender, Marital Status.
Travel	Route (Air/Sea/Road), Flight No, Company, Port of Entry.
Health/Customs	Declarations (Money, Plants, Animals), Symptoms tracking.
Audit	Form Reference Code, Review Status, Reviewed By, Timestamp.

7.4 Indexing & Validation Strategy

Indexing for Performance

- **Unique Index:** FormReferenceCode for instant ticket retrieval.
- **Search Index:** PassportNumber (App Table) and Email (Auth Table).
- **Security Index:** PassportNumber on Blacklist table for real-time submission checks.

Multi-Layer Validation

1. **JSR-303 (Bean Validation):** Enforces `@NotNull` and `@Email` constraints at the controller level.
2. **Service Layer Logic:** Validates age-specific logic (e.g., Minors must have guardian data).
3. **Database Constraints:** Enforces uniqueness and referential integrity via Oracle constraints.

8. SECURITY ARCHITECTURE

Security is the system's core pillar, implementing a defense-in-depth approach across all layers.

8.1 Authentication & Token Management

- **Method:** OAuth2 2.0 with JWT (JSON Web Tokens).
- **Token Storage:** Tokens are stored in the `oauth_access_token` table, allowing for centralized revocation and audit.
- **Encryption:** All user passwords are hashed using **BCrypt** with a high salt cost.

8.2 Authorization & Role Management (RBAC)

Permissions are assigned via Authorities attached to the JWT:

- **ROLE_SUPER_ADMIN:** Universal access.
- **ROLE_AGENT:** Access to scan and review APIs only.
- **ROLE_BLACKLIST_USER_MGMT:** Access restricted to security staff.

8.3 API Protection & Endpoint Security

- **Scope Guarding:** Gateways verify that the token has either `admin_server` or `agent_server` scopes.
- **Rate Limiting:** Public submission endpoints are protected by **Resilience4j** to prevent DDoS or brute-force form submissions.
- **CORS:** Restricted to authorized domains via Gateway configuration.

8.4 Sensitive Data Protection (PII)

PII (Personally Identifiable Information) such as Passport Numbers and Addresses are handled with strict care:

- **Encryption in Transit:** TLS/SSL is required for all API communications.
- **Logging:** The system is configured to avoid logging PII in standard application logs.
- **Access Control:** Database access is restricted to application service accounts with minimal required privileges.

8.5 Audit and Logging

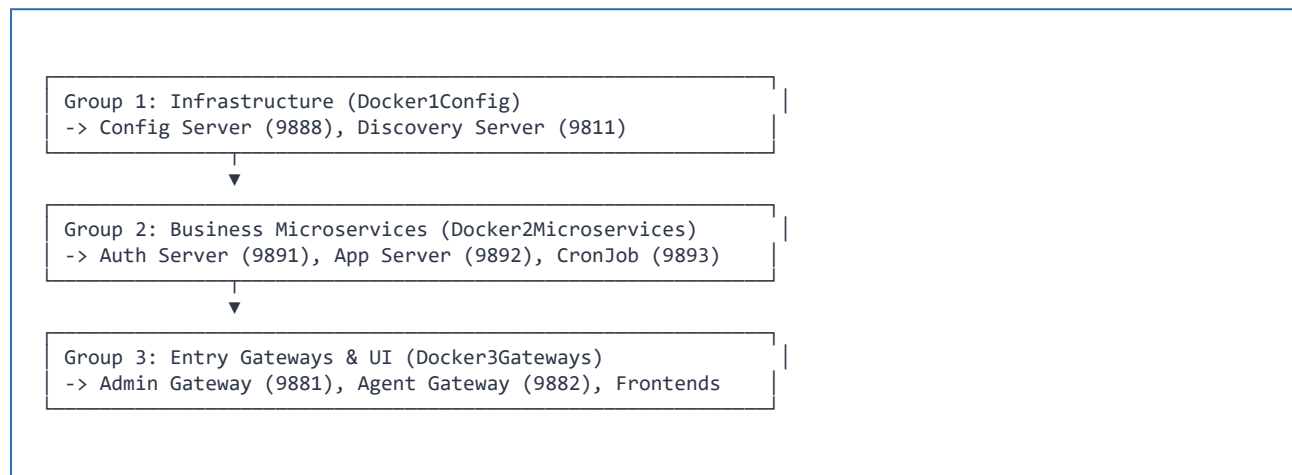
All sensitive operations (Blacklist additions, User creation, Status changes) are logged with a **Trace ID** provided by Spring Cloud Sleuth, allowing developers to trace a single request across multiple microservices.

9. DEPLOYMENT ARCHITECTURE

The Immigration Management System utilizes **Docker Containerization** with **Docker Compose** to achieve environment parity and modular scalability. This architecture abstracts the host operating system, ensuring consistent behavior from developer machines to production servers.

9.1 Docker Architecture & Compose Groups

The deployment is structured into three logical groups based on initialization dependencies. Services must be started in this specific sequence to ensure the microservice mesh stabilizes correctly.



9.2 Deployment Workflow

The standard workflow follows a **Build-Transfer-Start** pattern:

1. **Artifact Generation:** Compile the project using `mvn clean install` to generate SNAPSHOT JARs.
2. **Staging:** Transfer JAR files to their respective `DockerDir` directories on the host server.
3. **Infra Boot:** Run `docker-compose up -d` in `Docker1Config`. Verify Eureka dashboard is accessible at port 9811.
4. **Service Boot:** Deploy `Docker2Microservices`. This group maps host volumes for `/root/applogs` and the `geo.json` config.
5. **Edge Boot:** Deploy `Docker3Gateways` once core services are registered in Eureka.

9.3 Runtime Dependencies

- **External:** Oracle Database 18c (Standard JDBC connectivity), SMTP Relay for email, and `wkhtmltopdf` binary.
- **Internal:** Every service requires the Config Server to be active to resolve properties via `spring.config.import`.

9.4 Port Mapping & Environment Configuration

The following table outlines the networking configuration for the microservice cluster.

SERVICE NAME	INTERNAL	EXTERNAL	CONFIGURATION VARIABLES
Config Server	9888	9888	<code>spring.cloud.config.server.git.uri</code>
Discovery	9811	9811	<code>eureka.client.service-url.defaultZone</code>
Authorization	9891	9891	<code>spring.datasource.url</code> (Auth Schema)
Application	9892	9892	<code>WKHTML_RUN</code> , <code>app.init.geo-json-path</code>
CronJob	9893	9893	<code>spring.profiles.active</code>
Admin Gateway	9881	9881	<code>CORS allowed-origins</code>

9.5 Environment Separation

- **Development (dev):** Local Oracle instance, h2 for testing, DEBUG logging level enabled for rapid debugging.
- **Alpha (alpha):** Staging environment. Uses specific Git branch configurations. Log level set to INFO.
- **Production (prod):** Hardened environment with Oracle 18c Enterprise. Log level set to WARN. HTTPS is enforced at the Gateway level.

9.6 Container & Logging Configuration

Resource Allocation

To ensure stability, Docker resource limits are recommended:

- **App Server:** 2.0 CPUs / 2.0GB Memory.
- **Auth Server:** 1.0 CPU / 1.0GB Memory.

Logging Strategy

Containers are ephemeral, so logs are mounted to host volumes (`./applogs:/root/applogs`). Logs are formatted using SLF4J and Logback, rotating daily to prevent disk overflow.

10. DEVOPS AND CI/CD

While current deployment is managed through sequential scripts, the architecture is primed for a fully automated CI/CD lifecycle.

10.1 Version Control & Artifact Generation

- **Bitbucket Integration:** Code is managed in a multi-module Git repository. Configuration properties are stored in a separate `immigration-properties.git` repository for security separation.
- **Maven Lifecycle:** Uses `ImmigrationParent` as the master POM to maintain version consistency across all dependencies (Spring Boot, Resilience4j, etc.).
- **Docker Images:** Built using `openjdk:8-jdk-alpine` to minimize image size and attack surface.

10.2 Recommended CI/CD Pipeline

To transition to modern DevOps, the following 7-stage pipeline is recommended:

STAGE	ACTION
1. Source	Trigger on Git push to develop or main.
2. Build	<code>mvn clean compile</code> all modules.
3. Test	Execute JUnit 5 suite and JaCoCo coverage reports.
4. Package	Generate JARs and build Docker images tagged by version.
5. Artifact	Push images to a private Docker Registry (e.g., Nexus, ECR).
6. Staging	Deploy to Alpha environment for QA verification.
7. Prod	Blue-Green deployment to production after manual approval.

10.3 Configuration Management (Git-Ops)

The **Config Server** follows a Git-Ops philosophy. Changes to database passwords, feature flags, or API keys are committed to the property repository and refreshed across the microservice mesh via the `/actuator/refresh` endpoint without requiring a full service restart.

11. LOGGING AND OBSERVABILITY

In a distributed environment, observing the health of individual nodes is not enough; the system tracks the journey of traveler requests across the entire service mesh.

11.1 Application Logging & Error Tracking

- **Standard:** Logback implementation with RollingFileAppender.
- **Structure:** [timestamp] [level] [traceId] [spanId] [message].
- **Exceptions:** Global error handling via `@ControllerAdvice` ensures business exceptions (e.g., `CustomResponseException`) return meaningful HTTP status codes and JSON payloads instead of stack traces.

11.2 Distributed Tracing with Sleuth & Zipkin

The system integrates **Spring Cloud Sleuth** to inject Trace IDs into every request header. These IDs propagate through Feign clients to downstream services.

- **Zipkin (2.2.2.RELEASE):** Visualizes the latency of each service call. If a ticket generation fails, Zipkin identifies exactly which hop (Gateway → App → PDF Generator) caused the failure.

11.3 Metrics & Troubleshooting

Spring Boot Actuator

Active monitoring of JVM memory, GC, and HTTP throughput via `/actuator/metrics`. Health checks at `/actuator/health` are used by Docker Compose for service health probes.

Troubleshooting Matrix

SCENARIO	DIAGNOSTIC ENDPOINT	COMMON SOLUTION
Token Invalid	<code>/oauth/check_token</code>	Check Auth Schema connectivity; verify BCrypt salt.
PDF Failed	<code>/actuator/health</code>	Check WKHTML_RUN binary path and permissions.
Service Invisible	Eureka Dashboard (9811)	Verify Config Server is providing the Eureka URL.