Architectural Requirements & Design

Non-Functional Requirements

Quality Requirements

Swift Signals is designed to meet stringent quality standards, ensuring the platform remains extensible, maintainable, reliable, secure, and easy to use in real-world traffic management scenarios. The system is a comprehensive traffic optimization system designed to analyze traffic patterns, simulate intersection performance, and optimize traffic light timing using artificial intelligence algorithms.

Core System Requirements

Usability

Why it matters?

- The system needs to accommodate for both technical experts and non-technical users.

Key Expectations:

- The web interface must provide an intuitive experience for traffic engineers and system administrators.
- Visualization components should clearly present complex traffic data and simulation results.
- Help documentation and user guides should support users of varying technical expertise levels.

Extensibility

Why it matters?

- The system must be capable of seamless integration with existing traffic department infrastructure and real-world data sources.
- Municipal traffic systems vary widely in their technologies and data formats, requiring flexible integration capabilities.
- The system should adapt to different regional traffic management requirements and standards.

Key Expectations:

- The system must provide standardized APIs for integration with external traffic management systems.
- Support for multiple data formats and protocols commonly used by traffic departments (XML, JSON, REST, SOAP, etc.).
- Plugin architecture to support custom integrations without modifying core system functionality.

Maintainability

Why it matters?

- The system is expected to be used indefinitely by traffic departments.

Key Expectations:

- The code base must support long-term maintenance and evolution through clear separation of concerns and comprehensive documentation.
- Each microservice should follow established coding standards and include unit and integration tests.

Reliability

Why it matters?

- The system requires high availability to support continuous traffic analysis and optimization.

Key Expectations:

- Target uptime of 99.5% ensures continuous system availability.
- Health monitoring and alerting systems must provide proactive notification of system issues.

Basic System Requirements

Scalability

Why it matters?

- There are close to 5,500 traffic officers in South Africa and the system needs to be able to handle the necessary volume of users.
- The system is expected to accommodate future growth of traffic departments.
- Database systems should handle growing volumes of time-series traffic data and simulation results.
- The optimization component of the system must accommodate training and inference workloads that increase with system adoption.

Key Expectations:

- The service should be able to handle 10,000 traffic officer requests per second which is close to double the number of traffic officers

---

Performance

Why it matters?

- The system must deliver high-performance capabilities to handle: -> realistic traffic data processing -> effective optimisations of simulations -> effective rendering of simulations.
- The system needs to be able to perform as efficiently as possible to reduce user waiting time.
- The system must support concurrent users and simultaneous simulation runs without degradation in performance.

Key Expectations:

- Web interface responses must complete within 2 seconds under normal load.
- Traffic simulations must process hundreds of vehicles per intersection within acceptable timeframes.
- Artificial Inteligence optimisations for traffic lights must complete within 30 seconds per intersection.

Security

Why it matters?

- Due to municipal integrations the system must enforce strict security standards.

Key Expectations:

- The system must make use of role based access control
- The system must make use of rate limiting and input validation to prevent abuse.

Architectural Strategies

Usability

- Visualization components should clearly present complex traffic data and simulation results.
- Configuration workflows should guide users through intersection setup and optimization processes.
- Help documentation and user guides should support users of varying technical expertise levels.
- Responsiveness

Extensibility

- Standardized API interfaces with comprehensive documentation
- Plugin architecture with well-defined extension points

- Protocol abstraction layers to support multiple communication standards
- Environment-specific configuration management

Maintainability

- Each service should follow established coding standards and include unit and integration tests.
- Configuration management to enable environment-specific deployments without code changes.
- Service interfaces should be well-defined with comprehensive API documentation to facilitate integration and troubleshooting.
- API versioning strategies ensure backward compatibility during system updates.

Reliability

- Continuous monitoring with Elastic Search
- Redundancy and failover mechanisms to protect critical services.
- Circuit breaker patterns in communications to prevent cascading failures.
- Real-time health monitoring and alerts.
- Backup and recovery strategies safeguard historical traffic and simulation data.

Scalability

- Independent scaling of system services based on demand
- Efficient data storage and indexing
- Resource reuse

Performance

- Modularisation of system
- Efficient containerization
- Low-latency communication

Security

- TLS encryption
- Tokens
- Certification
- Rate limiting
- Input validation
- Audit logging

Architectural Patterns

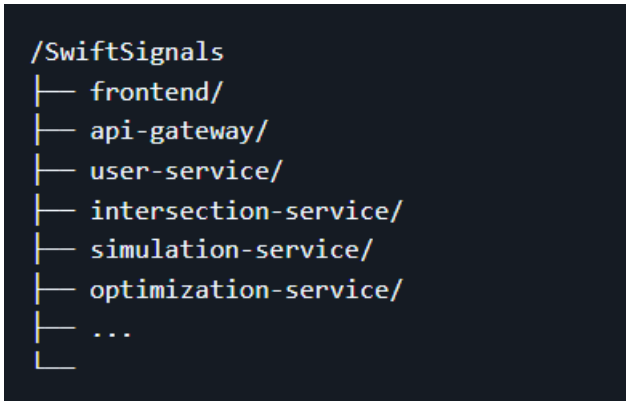
The system makes use of the following patterns:

- Microservices
- Layered Architecture
- API-Gateway Pattern
- Repository Pattern

Microservices

- Enables easy integration with real traffic departments as all that would be needed is to add another microservice for external system connectivity
- Supports modularization of the system by breaking down complex traffic management functionality into smaller, focused services
- Enables independent scaling of system services based on demand, allowing traffic optimization components to scale separately from data processing services
- Provides redundancy and failover mechanisms through service isolation, preventing single points of failure
- Facilitates resource reuse across different traffic management functions while maintaining service boundaries
- Enables each service to follow established coding standards and include comprehensive testing independently

How has it been integrated into Swift Signals?



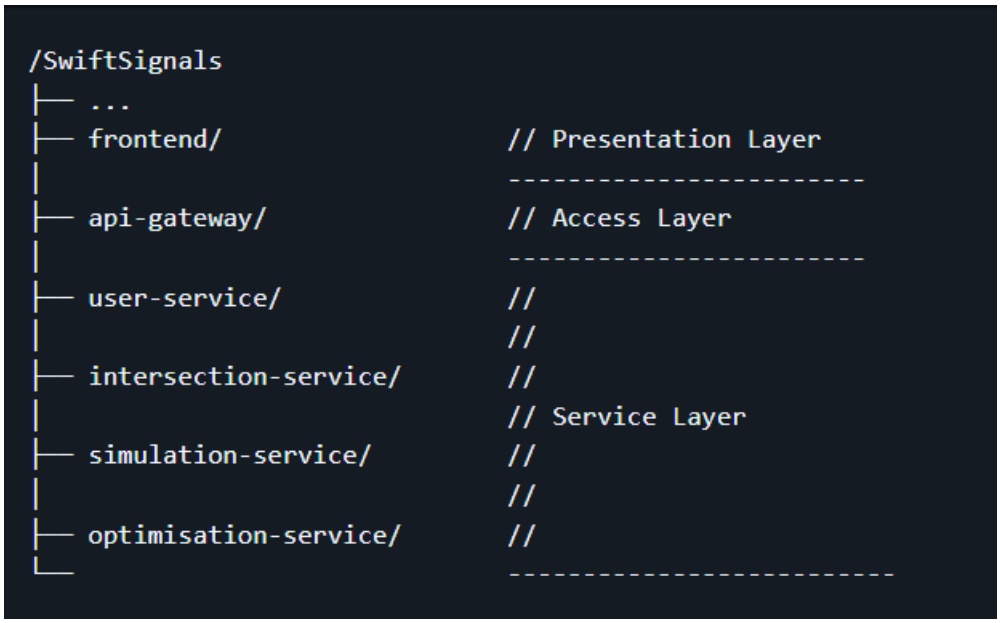
Layered Architecture

- Supports extensibility by providing clear integration points at each layer for external traffic management systems
- Enforces clear separation of concerns between presentation, business logic, and data layers for improved maintainability
- Provides structured security boundaries where each layer can implement appropriate access controls and validation

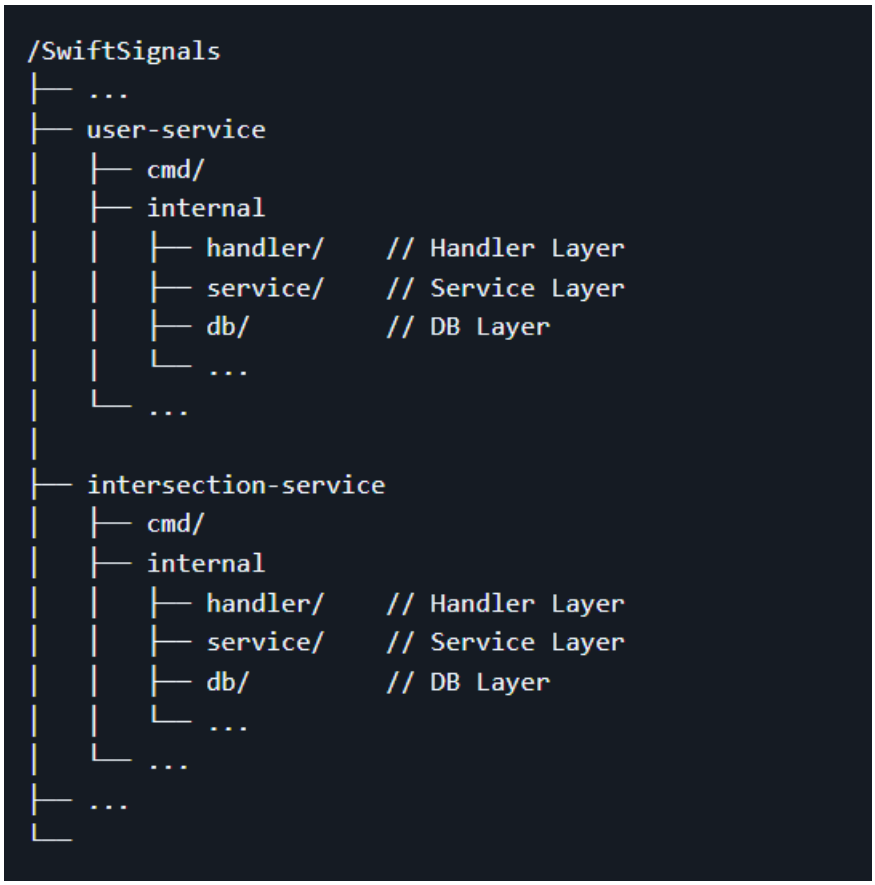
- Supports configuration management by isolating environment-specific settings within designated layers
- Enables well-defined service interfaces through standardized layer communication protocols
- Facilitates comprehensive API documentation by organizing functionality into logical layer groupings

How has it been integrated into Swift Signals?

High-Level Layers



Low-Level Layers



API-Gateway Pattern

- Enables extensibility by providing a centralized integration point for external traffic department systems and protocols
- Implements centralized rate limiting and input validation to prevent system abuse and protect backend services
- Provides single entry point for authentication and authorization, supporting role-based access control requirements
- Enables circuit breaker patterns in communications to prevent cascading failures across the system
- Supports API versioning strategies to ensure backward compatibility during system updates
- Centralizes audit logging for security monitoring and compliance tracking

How has it been integrated into Swift Signals?

```
/SwiftSignals
├── ...
├── api-gateway
│   ├── cmd/
│   ├── internal
│   │   ├── ...
│   │   ├── middleware/     // Common endpoint functionality
│   │   ├── util/           // Uniform tools
│   │   └── model/          // Convert to consistent types
│   └── swagger/            // Consistent API documentation
├── ...
└──
```

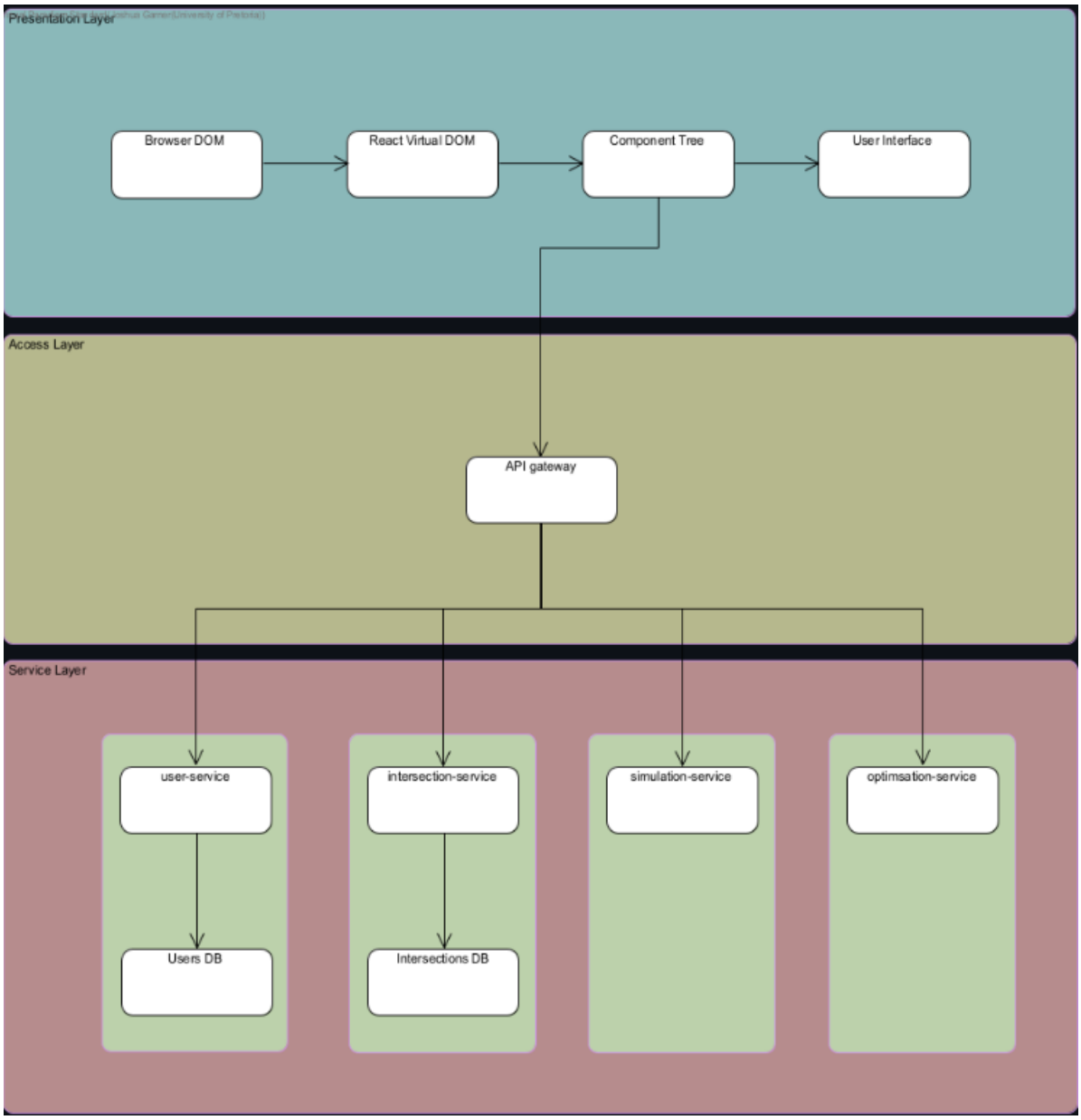
Repository Pattern

- Promotes extensibility by abstracting data access, making it easy to integrate with different traffic department databases and data sources
- Provides efficient data storage and indexing strategies for user and traffic data
- Enables backup and recovery strategies through standardized data access interfaces
- Supports scalable database operations by abstracting data persistence complexity
- Facilitates comprehensive testing through mockable data access layers
- Ensures data consistency and integrity across different traffic management operations

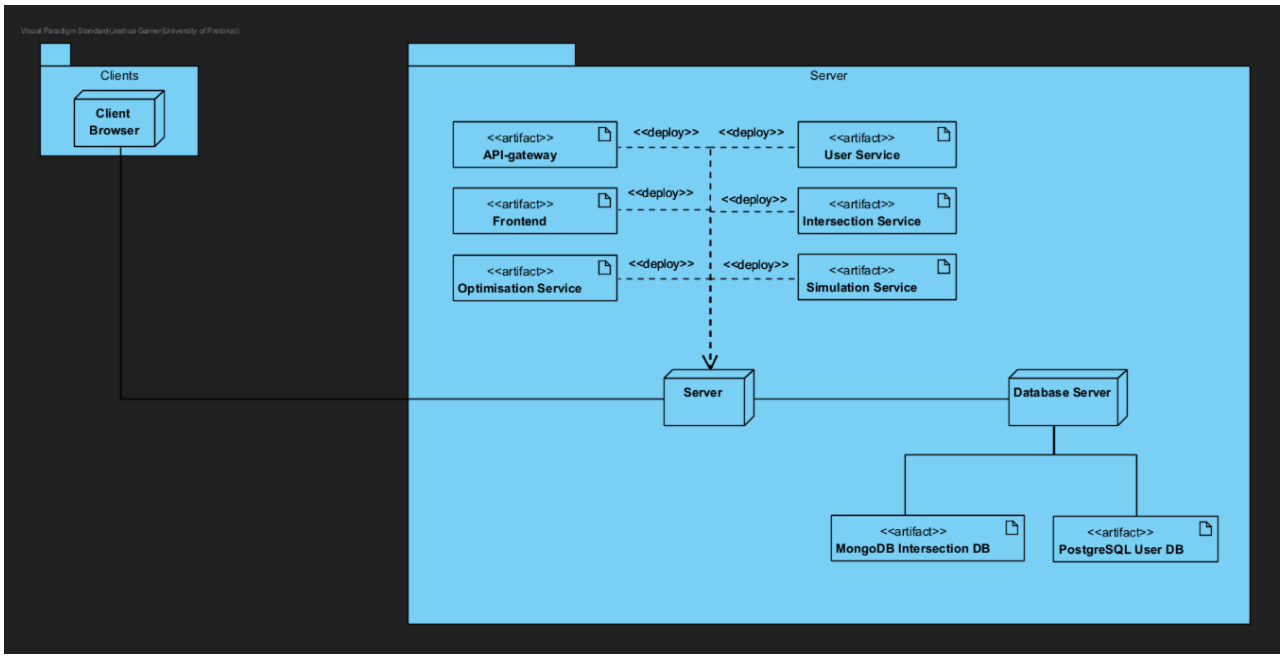
How has it been integrated into Swift Signals?

```
/SwiftSignals
├── ...
├── user-service
│   ├── cmd/
│   ├── internal
│   │   ├── ...
│   │   ├── db/         // Repository Pattern
│   │   └── ...
│   └── ...
│
├── intersection-service
│   ├── cmd/
│   ├── internal
│   │   ├── ...
│   │   ├── db/         // Repository Pattern
│   │   └── ...
│   └── ...
├── ...
└──
```

Architectural Diagram



Deployment Model



Service Contracts

This document defines the service contracts for all microservices in the Swift Signals system. It includes gRPC specifications for internal services and REST specifications for the API Gateway.

Services Overview

Service Name	Description	Protocol	Public?
user-service	Handles user registration, login, and admin privileges	gRPC	❌
intersection-service	Handles intersection creation, fetching and storing	gRPC	❌
simulation-service	Runs intersection simulations	gRPC	❌
optimization-service	Optimises simulations using AI models	gRPC	❌
api-gateway	Single entry point for frontend requests (REST → gRPC)	REST/gRPC	✅

user-service

Overview

Handles user account creation, login, and access control. Only this service interacts with the user database and JWT creation.

Proto File: user.proto

```
service UserService {
  rpc RegisterUser(RegisterUserRequest) returns (UserResponse);
  rpc LoginUser(LoginUserRequest) returns (LoginUserResponse);
  rpc LogoutUser(UserIDRequest) returns (google.protobuf.Empty);

  rpc GetUserByID(UserIDRequest) returns (UserResponse);
  rpc GetUserByEmail(GetUserByEmailRequest) returns (UserResponse);
  rpc GetAllUsers(GetAllUsersRequest) returns (stream UserResponse);
  rpc UpdateUser(UpdateUserRequest) returns (UserResponse);
  rpc DeleteUser(UserIDRequest) returns (google.protobuf.Empty);

  rpc GetUserIntersectionIDs(UserIDRequest)
      returns (stream IntersectionIDResponse);
  rpc AddIntersectionID(AddIntersectionIDRequest)
      returns (google.protobuf.Empty);
  rpc RemoveIntersectionIDs(RemoveIntersectionIDRequest)
      returns (google.protobuf.Empty);

  rpc ChangePassword(ChangePasswordRequest) returns (google.protobuf.Empty);
  rpc ResetPassword(ResetPasswordRequest) returns (google.protobuf.Empty);

  rpc MakeAdmin(AdminRequest) returns (google.protobuf.Empty);
  rpc RemoveAdmin(AdminRequest) returns (google.protobuf.Empty);
}
```

Testing

```
/user-service
├── /cmd
├── /internal
│   ├── /handler
│   │   ├── /test        //unit tests for handler layer
│   │   └── handler.go
│   ├── /service
│   │   ├── /test        //unit tests for service layer
│   │   ├── contract.go
│   │   └── service.go
│   ├── /db
│   │   ├── /test        //unit tests for db layer
│   │   ├── contract.go
│   │   └── repository.go
│   └── /test            //integration tests for user-service
│
```

intersection-service

Overview

Handles creation and management of intersection instances.

Proto File: intersection.proto

```
service IntersectionService {
  rpc CreateIntersection(CreateIntersectionRequest)
      returns (IntersectionResponse);
  rpc GetIntersection(IntersectionIDRequest) returns (IntersectionResponse);
  rpc GetAllIntersections(GetAllIntersectionsRequest)
      returns (stream IntersectionResponse);
  rpc UpdateIntersection(UpdateIntersectionRequest)
      returns (IntersectionResponse);
  rpc DeleteIntersection(IntersectionIDRequest) returns (google.protobuf.Empty);
  rpc PutOptimisation(PutOptimisationRequest) returns (PutOptimisationResponse);
}
```

Testing

```
/intersection-service
├── /cmd
├── /internal
│   ├── /handler
│   │   ├── /test        //unit tests for handler layer
│   │   └── handler.go
│   ├── /service
│   │   ├── /test        //unit tests for service layer
│   │   ├── contract.go
│   │   └── service.go
│   ├── /db
│   │   ├── /test        //unit tests for db layer
│   │   ├── contract.go
│   │   └── repository.go
│   └── /test            //integration tests for intersection-service
│
```

simulation-service

Overview

Handles running simulations of intersections.

Proto File: simulation.proto

```
service SimulationService {
  rpc GetSimulationResults(SimulationRequest)
      returns (SimulationResultsResponse);
  rpc GetSimulationOutput(SimulationRequest) returns (SimulationOutputResponse);
}
```

optimisation-service

Overview

Receives simulation parameters and performs optimization using AI.

Proto File: optimisation.proto

```
service OptimisationService {
  rpc RunOptimisation(OptimisationParameters) returns (OptimisationParameters);
}
```

api-gateway

Overview

The only public-facing service. Handles REST requests from the frontend, translates them to gRPC requests for backend services, and returns responses.

Authentication

Required for all endpoints (except signup/login): **JWT Bearer token**

REST Endpoints

Endpoint	Method	Description	Input (Request Body)	Output (Response)
/admin/users	GET	Retrieves a paginated list of all users.	N/A	model.User array
/admin/users/{id}	GET	Retrieves a user by their ID.	N/A	model.User
/admin/users/{id}	PATCH	Updates a user's details by their ID.	model.UpdateUserRequest	model.User
/admin/users/{id}	DELETE	Deletes a user by their ID.	N/A	No Content (204)
/intersections	GET	Retrieves all intersections associated with the user.	N/A	model.Intersections

Endpoint	Method	Description	Input (Request Body)	Output (Response)
/intersections	POST	Creates a new intersection.	model.CreateIntersectionRequest	model.CreateIntersectionResponse
/intersections/{id}	GET	Retrieves a single intersection by its ID.	N/A	model.Intersection
/intersections/{id}	PATCH	Partially updates an existing intersection.	model.UpdateIntersectionRequest	model.Intersection
/intersections/{id}	DELETE	Deletes an intersection by its ID.	N/A	No Content (204)
/intersections/{id}/optimise	GET	Generates and returns optimized simulation data.	N/A	model.SimulationResponse
/intersections/{id}/optimise	POST	Runs an optimization for an intersection.	N/A	model.OptimisationResponse
/intersections/{id}/simulate	GET	Generates and returns	N/A	model.SimulationResponse

Endpoint	Method	Description	Input (Request Body)	Output (Response)
		simulation data.		
/login	POST	Authenticates a user and returns a token.	model.LoginRequest	model.LoginResponse
/logout	POST	Invalidates the user's session or token.	N/A	model.LogoutResponse
/me	GET	Retrieves the profile of the authenticated user.	N/A	model.User
/me	PATCH	Updates the profile of the authenticated user.	model.UpdateUserRequest	model.User
/me	DELETE	Deletes the profile of the authenticated user.	N/A	No Content (204)

Endpoint	Method	Description	Input (Request Body)	Output (Response)
/register	POST	Registers a new user.	model.RegisterRequest	model.RegisterResponse
/reset-password	POST	Resets a user's password.	model.ResetPasswordRequest	model.ResetPasswordResponse

Testing

```
/api-gateway
├── /cmd
├── /internal
│   ├── /handler
│   │   ├── /test        //unit tests for handler layer
│   │   └── ...
│   ├── /service
│   │   ├── /test        //unit tests for service layer
│   │   └── ...
│   ├── /client
│   │   ├── /test        //unit tests for client layer
│   │   └── ...
│   └── /test            //integration tests for api-gateway
│
```

Uniform Error Handling

```go
type ServiceError struct {
    Code    ErrorCode      `json:"code"`
    Message string         `json:"message"`
    Cause   error          `json:"-"`
    Context map[string]any `json:"context,omitempty"`
}

const (
    ErrValidation    ErrorCode = "VALIDATION_ERROR"
    ErrNotFound      ErrorCode = "NOT_FOUND"
    ErrAlreadyExists ErrorCode = "ALREADY_EXISTS"
    ErrUnauthorized  ErrorCode = "UNAUTHORIZED"
    ErrForbidden     ErrorCode = "FORBIDDEN"
    ErrConflict      ErrorCode = "CONFLICT_ERROR"
    ErrUnavailable   ErrorCode = "UNAVAILABLE_ERROR"
    ErrDatabase      ErrorCode = "DB_ERROR"
    ErrInternal      ErrorCode = "INTERNAL_ERROR"
    ErrExternal      ErrorCode = "EXTERNAL_ERROR"
)
```

Notes

- All internal services communicate using **gRPC**
- External clients use a **REST API** via the API Gateway
- Fields and message formats follow .proto definitions
- All endpoints are secured with **JWT auth**
- Admin-only actions (e.g., banning users) are restricted by role