ProofIt

Developer Manual

Kings College London
5CCS2SEG - Major Group Project
Team Runtime Terrors

March 2025

Contents

1 Introduction

This document serves as the comprehensive developer manual for the ProofIt application. It provides detailed instructions for installation, configuration, and technical specifications of all system components. This manual is intended for developers, system administrators, and technical contributors who need to understand the system architecture, setup procedures, and API endpoints.

2 Installation Instructions

2.1 Overview

Installing **ProofIt** locally involves setting up several components:

- **Server**: The main application backend.
- **Client**: A web application built with TypeScript (TSX), React, and Vite.
- **Python Microservice**: Part of the embeddings module.

The following dependencies are required to run ProofIt:

- Docker (or Docker Desktop on Windows/Mac)
- Redis
- Ollama
- Java Runtime Environment (JVM)
- Kotlin compiler
- Python 3.10 (command "python3.10" should be available)

We recommend using an IDE with Gradle support, such as JetBrains IntelliJ IDEA, which includes both a JVM and Kotlin compiler, to simplify the development process.

2.2 Installation Options

There are three ways to install **ProofIt** locally, listed in order of simplicity:

1. **Docker Installation (recommended)**: The simplest option, requiring only Ollama to be installed.
2. **Startup Script**: A middle-ground option that automates most of the installation but requires a few manual steps.
3. **Manual Installation**: The most involved option, giving you full control over each component.

3 Docker Installation

This is the easiest option to get the application running locally. You only need Ollama (and your model of choice) running on your host machine, and Docker will take care of the rest.

3.1 Prerequisites

- **Docker**: Install Docker Engine (Linux) or Docker Desktop (Mac/Windows)
- **Docker Compose**: Ensure the Docker Compose V2 plugin is installed as well. This is required for the next step.
- **Ollama**: Install Ollama following the official documentation

3.2 Installation Steps

1. Install a model of your choice from Ollama's official list. In testing, we used mainly qwen2.5-coder:14b.
2. From the root directory, run the following command:

```
docker compose up -d
```

3. This will build and run all services, including installing any other necessary dependencies.

3.3 Troubleshooting Firewall Issues

IMPORTANT FIREWALL WARNING

Firewall rules may prevent the server container from communicating with Ollama on the host machine. Follow these steps to resolve this issue:

Step 1: Access the Docker container

- Find the container ID first:

```
docker ps
```

- Access the container (replace CONTAINER_ID with your actual server container ID):

```
docker exec -it CONTAINER_ID /bin/bash
```

Step 2: Identify the container's network subnet

- Inside the container, run:

```
ip addr show
```

- Look for the IP address (usually something like 172.25.0.5)

Step 3: Configure firewall rules on your host machine

Linux (using UFW):

```
sudo ufw allow from 172.25.0.0/16 to any port 11434
```

Linux (using iptables):

```
sudo iptables -A INPUT -s 172.25.0.0/16 -p tcp --dport 11434 -j ACCEPT
sudo iptables-save
```

macOS:

```
sudo /usr/libexec/ApplicationFirewall/socketfilterfw --add /usr/local/bin/ollama
sudo /usr/libexec/ApplicationFirewall/socketfilterfw --unblock /usr/local/bin/ollama
```

Windows (using PowerShell as Administrator):

```
New-NetFirewallRule -DisplayName "Allow Ollama" -Direction Inbound -Protocol TCP -
    LocalPort 11434 -Action Allow -RemoteAddress 172.25.0.0/16
```

Step 4: Ensure Ollama is listening on all network interfaces

Check the current listening status:

```
# Linux/macOS
netstat -tunl | grep 11434

# Windows PowerShell
netstat -an | findstr 11434
```

If you see 0.0.0.0:11434 or :::11434, Ollama is correctly listening on all interfaces.
If you see 127.0.0.1:11434, you need to configure Ollama to listen on all interfaces:
Linux/macOS: Edit the Ollama service file:

```
# Find the service file location (varies by distribution)
sudo systemctl status ollama

# Edit the service file
sudo nano /etc/systemd/system/ollama.service

# Add or modify the line in [Service] section:
ExecStart=/usr/local/bin/ollama serve --host 0.0.0.0

# Restart Ollama
sudo systemctl daemon-reload
sudo systemctl restart ollama
```

Windows: Modify the Ollama configuration in the Registry or restart Ollama with the appropriate host flag:

```
ollama serve --host 0.0.0.0
```

4 Startup Script Installation

This method provides a balance between simplicity and control. The startup script will automatically install necessary dependencies, run the client, run the server, and run the Python microservice. However, it requires all external components to be installed on the host machine.

4.1 Prerequisites

Ensure the following dependencies are installed:

- **Docker** (for Redis support)
- **Redis**, **Ollama**, and **Python 3.10**
- **Java Virtual Machine (JVM)**, and **Kotlin compiler**

4.2 Installation Steps

On Linux/macOS:

1. Open a terminal and navigate to the root directory.
2. Run:

```
./proofit.sh
```

On Windows:

1. Open PowerShell and navigate to the root directory.
2. Run:

```
./proofit.ps1
```

The top-level script will automatically call the server's startup script and the client's startup script. This will cause all necessary dependencies to be installed, and then run the client, the server, and the Python microservice. The logs for both client and server applications will be visible in the terminal window.

N.B The startup script will **NOT** seed the template library. This must be done manually. Instructions on how to do this are detailed in section 5.6.1

5 Manual Installation

This option gives you complete control over each component of the installation process.

5.1 Installing Dependencies

Before installing **ProofIt**, ensure that the following dependencies are installed and running:

- **Java Runtime Environment (JVM)** and **Kotlin Compiler**:
  - On **Linux/Mac**: Use your package manager or download from the official websites.
  - On **Windows**: Install the JDK (which includes the JVM) and Kotlin from their official distributions.
- **Python 3.10**:
  - On **Linux/Mac**: Install via your package manager or from `https://www.python.org/downloads/`. You may need to include the relevant repository. Instructions on how to do this for your specific distribution can be found online.
  - On **Windows**: Download and install from the official Python website; ensure you select the option to add Python to your PATH.
- **Docker**:
  - On **Linux**: Install Docker Engine following your distribution's instructions. Make sure to include the "compose" plugin. Instructions on how to do that are available on the official Docker documentation.
  - On **Mac/Windows**: Install Docker Desktop.
- **Redis** and **Ollama**: Install these tools following the official documentation. Verify that both services are running before proceeding. You may also wish to install a model at this point. You can choose any model you like from Ollama's official list. Code generation will be impacted by this choice, and we cannot guarantee that the application will work with any given model. In testing, we used mainly qwen2.5-coder:14b.

Installation guides for these dependencies are available online. Once installed, confirm that Redis and Ollama are active; otherwise, **ProofIt** will not function correctly.

5.2 Installing the Client

1. Open a terminal and navigate to the `one-day-poc-client` directory.
2. Install dependencies:

```
npm install
```

3. Start the development server:

```
npm run dev
```

5.3 Installing the Python Microservice

The Python microservice, which is part of the embeddings module, resides in the directory `one-day-poc-server/embeddings/`
It requires Python 3.10. Use the following steps based on your operating system:

On Linux/macOS

1. Open a terminal and navigate to the microservice directory.
2. Create a virtual environment:

```
python3.10 -m venv venv
```

3. Activate the virtual environment:

```
source venv/bin/activate
```

4. Install required dependencies:

```
pip install -r requirements.txt
```

5. Run the microservice:

```
python3.10 -m information_retrieval
```

6. When finished, deactivate the virtual environment:

```
deactivate
```

On Windows

1. Open **Command Prompt** or **PowerShell** and navigate to the microservice directory.
2. Create a virtual environment:

```
python -m venv venv
```

3. Activate the virtual environment:
   - For Command Prompt:

   ```
   venv\Scripts\activate.bat
   ```

   - For PowerShell:

   ```
   venv\Scripts\Activate.ps1
   ```

4. Install required dependencies:

```
pip install -r requirements.txt
```

5. Run the microservice as a module:

```
python3.10 -m information_retrieval
```

6. When finished, deactivate the virtual environment by typing:

```
deactivate
```

5.4 Installing the Server

The server application is built with Gradle. Follow these steps:

On Linux/macOS

1. Open a terminal and navigate to the project root.
2. Build the project using the Gradle wrapper:

```
./gradlew build -x test
```

3. To run the server:

```
./gradlew run
```

On Windows

1. Open **Command Prompt** or **PowerShell** and navigate to the project root.
2. Build the project using the Gradle wrapper:

```
gradlew build -x test
```

3. To run the server:

```
gradlew run
```

5.5 Final Checklist

Before launching **ProofIt**, ensure that:

- All dependencies (Docker, Redis, Ollama, JVM, Kotlin, and Python) are installed and running.
- The appropriate virtual environments are activated where needed.
- Environment-specific commands (especially for Windows) have been used.
- All services are running: Python microservice, client, server, Ollama, and Redis.

5.6 Running the Application

This section covers how to run the various components of the application after setting up all the required dependencies and services.

5.6.1 Seeding the Template Library

Before running the application, you may want to seed the template library with predefined templates. You should **not** have any ProofIT services running to complete this step.

```
# In the one-day-poc-server directory
./gradlew seed        # Linux/macOS
gradlew seed          # Windows
```

5.6.2 Running the Server

To start the server component:

```
# In the one-day-poc-server directory
./gradlew run         # Linux/macOS
gradlew run           # Windows
```

5.6.3 Running the Python Microservice

To run the Python information retrieval microservice:

```
# Navigate to the Python directory and activate the virtual environment
source venv/bin/activate    # Linux/macOS
venv\Scripts\Activate.ps1   # Windoes

# Run the service
python3.10 -m information_retrieval
```

5.6.4 Running the Client

To start the client application:

```
# In the one-day-poc-client directory
# Install dependencies (use either command)
npm install
# OR for clean install based on package-lock.json
npm ci
Start the development server
npm run dev
```

5.6.5 Complete Startup Sequence

For a full deployment, follow these steps in order:

1. Ensure all dependencies and services are running
2. Seed the template library
3. Start the server
4. Start the Python microservice
5. Start the client application

After completing these steps, the application should be fully operational and ready for use.

6 Automated Testing

Comprehensive testing is essential for maintaining code quality and preventing regressions. Our application includes three distinct testing layers: client-side tests, server-side tests, and end-to-end (E2E) tests. This section provides detailed instructions for executing each testing suite and analyzing the resulting coverage reports.

6.1 Client-Side Testing

The client-side test suite validates the functionality of our frontend components, ensuring proper rendering, state management, and user interactions. To execute the client-side tests:

```
# Navigate to the client directory
cd one-day-poc-client

# Execute the test suite
npm run test
```

This command will launch the test runner in interactive watch mode by default. The test runner automatically identifies and executes tests in files with the following naming patterns:

- `*.test.js`
- `*.test.jsx`
- `*.test.ts`
- `*.test.tsx`

6.1.1 Test Coverage Analysis

The client-side test suite automatically generates coverage reports during execution. These reports provide detailed metrics on code coverage, helping identify areas that may require additional testing. To review the coverage report:

```
# Open the coverage report in your default browser
open one-day-poc-client/coverage/index.html  # macOS
xdg-open one-day-poc-client/coverage/index.html  # Linux
start one-day-poc-client/coverage/index.html  # Windows
```

The coverage report includes metrics such as:

- Statement coverage: Percentage of code statements executed during tests
- Branch coverage: Percentage of conditional branches executed during tests
- Function coverage: Percentage of functions called during tests
- Line coverage: Percentage of executable lines executed during tests

6.2 Server-Side Testing

The server-side test suite validates our backend functionality, including API endpoints, business logic, and data persistence. To execute the server-side tests:

```
# Navigate to the server directory
cd one-day-poc-server

# For Linux/macOS
./gradlew clean test jacocoMergedReport

# For Windows
gradlew clean test jacocoMergedReport
```

This command performs three primary operations:

1. `clean`: Removes previous build artifacts to ensure a fresh testing environment
2. `test`: Executes the test suite, which includes unit and integration tests
3. `jacocoMergedReport`: Generates a comprehensive coverage report using JaCoCo (Java Code Coverage)

6.2.1 Test Coverage Analysis

After executing the server-side tests, a detailed coverage report is generated. To review this report:

```
# Open the coverage report in your default browser
open one-day-poc-server/build/reports/jacoco/html/index.html  # macOS
xdg-open one-day-poc-server/build/reports/jacoco/html/index.html  # Linux
start one-day-poc-server/build/reports/jacoco/html/index.html  # Windows
```

The JaCoCo report provides detailed metrics on code coverage across all server-side components, helping identify areas that may require additional testing.

6.3 End-to-End (E2E) Testing

End-to-end tests validate the entire application stack, ensuring that all components work correctly together in a production-like environment. These tests simulate real user interactions and verify that the system behaves as expected. To execute the E2E tests:

```
# Navigate to the project root directory
cd /path/to/project/root

# For Linux/macOS
./test.sh

# For Windows
./test.ps1
```

The E2E test scripts perform several operations:

1. Start the server in a test environment
2. Start the client in a test environment
3. Execute automated test scenarios that interact with the application
4. Report test results and shut down the test environment

6.3.1 Important Considerations

When working with E2E tests, consider the following:

- E2E tests may take significantly longer to run than unit or integration tests
- The test environment requires both frontend and backend components to be operational
- Tests simulate browser interactions, which may occasionally produce intermittent failures due to timing issues
- Debugging E2E test failures often requires examining browser logs and screenshots captured during test execution

6.4 Continuous Integration

All test suites are automatically executed in our CI/CD pipeline when changes are pushed to the repository. This ensures that code changes do not introduce regressions before being deployed to production environments.

The CI pipeline executes:

1. Client-side tests with coverage analysis
2. Server-side tests with coverage analysis
3. End-to-end tests to validate the complete application

6.5 Troubleshooting Common Testing Issues

6.5.1 Client-Side Testing Issues

If client-side tests fail:

- Verify that all dependencies are correctly installed (`npm install`)
- Check for environment-specific configuration issues
- Review test output for specific component failures
- Examine browser console logs for JavaScript errors

6.5.2 Server-Side Testing Issues

If server-side tests fail:

- Verify that the database configuration is correct for the test environment
- Check for resource conflicts, such as port bindings
- Review test output for specific exceptions or assertion failures
- Examine server logs for detailed error information

6.5.3 E2E Testing Issues

If E2E tests fail:

- Verify that both client and server components start correctly
- Check for network connectivity issues between components
- Review browser logs and screenshots for visual clues
- Consider timing issues that may require adjusting wait periods in test scripts

7 Server-Side API

This section provides detailed documentation of the server-side API modules and their endpoints.

7.1 Authentication Module

7.1.1 Overview

The Authentication Module provides robust authentication and authorization capabilities for the ProofIt application. It implements a decoupled architecture that separates authentication concerns from the rest of the application, allowing for flexible integration with various authentication providers while maintaining clean code organization.

The module handles:

- User authentication via OAuth 2.0 (primarily with Amazon Cognito)
- Session management through secure HTTP-only cookies
- JWT validation for secure API access
- Role-based access control via JWT claims
- Token caching using Redis for performance optimization

This design ensures that authentication logic remains independent of other application components, promoting maintainability and allowing the authentication provider to be changed with minimal impact on the rest of the system. For instance, while the current implementation uses Amazon Cognito, switching to another OAuth provider would require changes only to the configuration, with no modifications to the core application logic.

The Authentication Module serves as the foundation for the application's security model, ensuring that only authenticated users can access protected resources while providing a seamless user experience through proper session management and efficient token validation.

7.1.2 Architecture

The Authentication Module employs a layered architecture that separates concerns and promotes maintainability:

Structural Components

- **Authentication Builder**: Configures authentication providers and JWT validation (implemented in `AuthenticatorBuil`
- **Route Configuration**: Sets up authentication-related endpoints (in `AuthenticationRoutes.kt`)
- **Session Management**: Handles user session creation and validation
- **Helper Components**: Utilities for token validation, user info retrieval, and caching

Architectural Patterns The module employs several design patterns:

- **Builder Pattern**: Used in `AuthenticatorBuilder` to configure authentication settings
- **Façade Pattern**: The `Authentication` object provides a simplified interface to the authentication system
- **Extension Functions**: Kotlin extension functions are used extensively to enhance the Ktor framework's authentication capabilities
- **Singleton Pattern**: Used for shared resources like Redis connections

Integration with Ktor The authentication module is tightly integrated with the Ktor web framework, which provides the underlying infrastructure for:

- OAuth authentication via the `io.ktor.server.auth` package
- JWT validation through the `io.ktor.server.auth.jwt` package
- Session management using the `io.ktor.server.sessions` package

Data Flow The authentication process follows a clear flow:

1. User initiates login via the authentication endpoint
2. Ktor routes the request to the configured OAuth provider
3. After successful authentication, the callback endpoint receives the user's tokens
4. JWT claims are extracted and validated
5. User session is created and stored in a secure cookie
6. Session data is cached in Redis for faster subsequent validations
7. Protected routes verify the session token before granting access

This architecture ensures a clean separation between authentication logic and the rest of the application, making the system easier to maintain and test. It also provides flexibility to adapt to different authentication requirements or providers in the future.

7.1.3 Core Components

The Authentication Module consists of several key components that work together to provide a comprehensive authentication solution:

Authenticators The `Authenticators` object contains methods for configuring OAuth settings and JWT validation:

```
object Authenticators {
    private lateinit var jwkProvider: JwkProvider

    /**
     * Configures the OAuth settings for the application.
     */
    internal fun AuthenticationConfig.configureOAuth(config: JsonObject) {
        val providerLookupData = config["providerLookup"]!!.jsonObject
        oauth(config["name"]!!.jsonPrimitive.content) {
            urlProvider = { config["urlProvider"]!!.jsonPrimitive.content }
            providerLookup = {
                OAuthServerSettings.OAuth2ServerSettings(
                    name = providerLookupData["name"]!!.jsonPrimitive.content,
                    authorizeUrl = providerLookupData["authorizeUrl"]!!.jsonPrimitive.content,
                    accessTokenUrl = providerLookupData["accessTokenUrl"]!!.jsonPrimitive.
                        content,
                    clientId = providerLookupData["clientId"]!!.jsonPrimitive.content,
                    clientSecret = providerLookupData["clientSecret"]!!.jsonPrimitive.content,
                    defaultScopes = providerLookupData["defaultScopes"]!!.jsonArray
                        .map { it.jsonPrimitive.content },
                    requestMethod = HttpMethod.Post,
                )
            }
```

```kotlin
                client = HttpClient(CIO)
            }
    }

    /**
     * Configures the JWT settings for the application.
     */
    fun AuthenticationConfig.configureJWTValidator(config: JsonObject) {
        val issuer = config["jwtIssuer"]!!.jsonPrimitive.content
        jwkProvider =
            JwkProviderBuilder(issuer)
                .cached(JWTConstants.JWK_PROVIDER_CACHE_SIZE, JWTConstants.
                    JWK_PROVIDER_EXPIRES_IN, TimeUnit.HOURS)
                .rateLimited(JWTConstants.JWK_PROVIDER_BUCKET_SIZE, 1, TimeUnit.MINUTES)
                .build()

        generateVerifier(jwkProvider, issuer)
    }

    // Additional implementation details...
}
```

Listing 1: Authenticators Object Implementation

JWTConstants The `JWTConstants` object defines important configuration values for JWT validation:

```kotlin
internal object JWTConstants {
    const val LEEWAY: Long = 10
    const val JWK_PROVIDER_CACHE_SIZE: Long = 10
    const val JWK_PROVIDER_EXPIRES_IN: Long = 24
    const val JWK_PROVIDER_BUCKET_SIZE: Long = 10
}
```

Listing 2: JWTConstants Object

AuthenticatedSession The `AuthenticatedSession` data class represents a user's authenticated session:

```kotlin
@Serializable
internal data class AuthenticatedSession(
    val userId: String,
    val token: String,
    val admin: Boolean?,
)
```

Listing 3: AuthenticatedSession Data Class

This class contains:

- `userId`: The unique identifier for the authenticated user
- `token`: The JWT token for authentication
- `admin`: A flag indicating whether the user has administrative privileges

AuthenticationRoutes The `AuthenticationRoutes` object defines the endpoints for authentication operations:

```kotlin
object AuthenticationRoutes {
    const val AUTHENTICATION_ROUTE: String = "/api/auth"
    const val AUTHENTICATION_CHECK_ROUTE: String = "/api/auth/check"
    const val CALL_BACK_ROUTE: String = "/api/auth/callback"
    const val LOG_OUT_ROUTE: String = "/api/auth/logout"
    const val JWT_VALIDATION_ROUTE: String = "/api/auth/validate"
    const val USER_INFO_ROUTE: String = "api/auth/me"
}
```

Listing 4: AuthenticationRoutes Object

JWTValidationResponse The `JWTValidationResponse` data class represents the result of validating a JWT token:

```kotlin
@Serializable
internal data class JWTValidationResponse(
    val userId: String,
    val admin: Boolean?,
)
```

Listing 5: JWTValidationResponse Data Class

CognitoUserInfo The `CognitoUserInfo` data class represents user information retrieved from the identity provider:

```kotlin
@Serializable
internal data class CognitoUserInfo(
    val name: String,
    val email: String,
    val dob: String,
)
```

Listing 6: CognitoUserInfo Data Class

Helper Functions The module includes various helper functions that support authentication operations:

- `validateJWT`: Validates a JWT token and extracts user information

```kotlin
    internal fun validateJWT(token: String?): JWTValidationResponse? =
        runCatching { JWT.decode(token) }
            .getOrNull()
            ?.takeIf { it.expiresAt.after(Date.from(Instant.now())) }
            ?.let { decoded ->
                val userId = decoded.getClaim("sub").asString()
                val admin = decoded.getClaim("cognito:groups").asList(String::class.java)?.
                    contains("admin_users") == true
                userId?.let { JWTValidationResponse(it, admin) }
            }
```

Listing 7: JWT Validation Function

- `cacheSession`: Stores authentication data in Redis for faster validation

```kotlin
    internal fun cacheSession(
        token: String,
        authData: JWTValidationResponse,
        expirySeconds: Long = 3600,
    ) {
        Redis.getRedisConnection().use { jedis ->
            jedis.setex("auth:$token", expirySeconds, Json.encodeToString(authData))
        }
    }
```

Listing 8: Session Caching Function

- `checkCache`: Checks Redis for cached authentication data

```kotlin
    internal fun checkCache(token: String): JWTValidationResponse? {
        Redis.getRedisConnection().use { jedis ->
            val cachedData = jedis["auth:$token"] ?: return null
            return Json.decodeFromString<JWTValidationResponse>(cachedData)
        }
    }
```

Listing 9: Cache Checking Function

- `generateUserInfo`: Extracts user attributes from authentication responses

```kotlin
    internal fun generateUserInfo(response: Response): CognitoUserInfo {
        val jsonResponse = Json.parseToJsonElement(response.body.string()).jsonObject
        val attributes = jsonResponse["UserAttributes"]?.jsonArray ?: return CognitoUserInfo
            ("", "", "")

        return CognitoUserInfo(
            name = PoCJSON.findCognitoUserAttribute(attributes, "name") ?: "Unknown",
            email = PoCJSON.findCognitoUserAttribute(attributes, "email") ?: "Unknown",
            dob = PoCJSON.findCognitoUserAttribute(attributes, "birthdate") ?: "Unknown",
```

```
        )
    }
```

Listing 10: User Info Generation Function

These core components work together to provide a cohesive authentication system that handles user authentication, session management, and access control in a secure and efficient manner.

7.1.4 Authentication Mechanism

The Authentication Module implements a comprehensive authentication mechanism based on industry-standard protocols and best practices.

OAuth 2.0 Flow The primary authentication mechanism uses the OAuth 2.0 authorization code flow:

1. The user initiates authentication by accessing the `/api/auth` endpoint
2. The application redirects the user to the identity provider's authorization page (Amazon Cognito)
3. The user authenticates with their credentials on the provider's page
4. Upon successful authentication, the provider redirects back to the application's callback endpoint with an authorization code
5. The application exchanges this code for access and ID tokens
6. The tokens are validated, and user information is extracted from the JWT claims
7. A session is created for the authenticated user

JWT-Based Authentication JSON Web Tokens (JWTs) are used for secure authentication after the initial OAuth flow:

- **Token Structure**: JWTs contain three parts: header, payload, and signature
- **Token Validation**: The module verifies JWTs using the JSON Web Key Set (JWKS) provided by the identity provider
- **Claims Extraction**: User information and permissions are extracted from the JWT claims
- **Token Storage**: The token is stored in an HTTP-only, secure cookie to prevent client-side access

Provider Agnosticism While the current implementation uses Amazon Cognito as the identity provider, the module is designed to be provider-agnostic. The OAuth configuration is loaded from a JSON file, allowing different providers to be integrated by modifying the configuration file without changing the application code.

```json
{
  "name": "Cognito",
  "jwtIssuer": "https://cognito-idp.region.amazonaws.com/pool-id",
  "urlProvider": "http://localhost:8000/api/auth/callback",
  "providerLookup": {
    "name": "Amazon Cognito",
    "authorizeUrl": "https://domain.auth.region.amazoncognito.com/oauth2/authorize",
    "accessTokenUrl": "https://domain.auth.region.amazoncognito.com/oauth2/token",
    "clientId": "client-id",
    "clientSecret": "client-secret",
    "defaultScopes": [
      "email",
      "openid",
      "profile"
    ]
  }
}
```

Listing 11: Example OAuth Configuration

Multiple Authentication Methods The module supports multiple authentication methods for API requests:

- **Cookie-Based**: Using the `AuthenticatedSession` cookie
- **Bearer Token**: Using the Authorization header with a JWT token

This is implemented in the JWT verifier configuration:

```
jwt("jwt-verifier") {
    authHeader { call ->
        val sessionCookie =
            call.request.cookies["AuthenticatedSession"]
                ?: return@authHeader call.request.headers["Authorization"]?.let {
                    parseAuthorizationHeader(it) }

        return@authHeader try {
            val session = Json.decodeFromString<AuthenticatedSession>(sessionCookie)
            parseAuthorizationHeader("Bearer ${session.token}")
        } catch (_: SerializationException) {
            null
        }
    }
    // Additional configuration...
}
```

Listing 12: JWT Authentication Header Configuration

This dual approach provides flexibility for different client applications while maintaining a consistent security model.

Role-Based Access Control The authentication mechanism includes support for role-based access control through JWT claims. The module extracts role information from the `cognito:groups` claim:

```
val admin = decoded.getClaim("cognito:groups").asList(String::class.java)?.contains("
    admin_users") ?: false
```

Listing 13: Role Extraction from JWT

This information is stored in the session and can be used for authorization decisions in protected routes.

JWT Verification with JWKS The module verifies JWT signatures using the JSON Web Key Set (JWKS) provided by the identity provider:

```
val jwkProvider =
    JwkProviderBuilder(issuer)
        .cached(JWTConstants.JWK_PROVIDER_CACHE_SIZE, JWTConstants.JWK_PROVIDER_EXPIRES_IN,
            TimeUnit.HOURS)
        .rateLimited(JWTConstants.JWK_PROVIDER_BUCKET_SIZE, 1, TimeUnit.MINUTES)
        .build()
```

Listing 14: JWKS Configuration

The JWKS endpoint is cached to improve performance and protected against denial-of-service attacks through rate limiting.

The authentication mechanism is designed to be secure, flexible, and compliant with modern web standards, ensuring that users can authenticate safely while providing developers with the tools needed to protect application resources.

7.1.5 Available Endpoints

The Authentication Module exposes several endpoints that handle different aspects of the authentication process. Each endpoint serves a specific purpose in the authentication flow.

Authentication Initiation GET /api/auth

Initiates the OAuth 2.0 authentication flow by redirecting the user to the identity provider's login page. This endpoint is the entry point for user authentication and requires no authentication itself.

Implementation Details:

```
private fun Route.setAuthenticationEndpoint(route: String) {
    get(route) {
        call.respondRedirect("/authenticate")
    }
}
```

The endpoint redirects to a central authentication route that is configured with the appropriate OAuth provider.

Authentication Callback GET /api/auth/callback

Handles the callback from the OAuth provider after successful authentication. This endpoint:

- Receives the authorization code from the provider
- Exchanges it for access and ID tokens
- Extracts user information from the JWT claims
- Creates a session for the authenticated user
- Redirects the user to the appropriate application page

Implementation Details:

```
fun Route.setUpCallbackRoute(
    route: String,
    redirectDomain: String = "http://localhost:5173",
) {
    get(route) {
        val principal: OAuthAccessTokenResponse.OAuth2? = call.authentication.principal()
        if (principal == null) {
            call.respond(HttpStatusCode.Unauthorized)
            return@get
        }
        val token: String? = principal.extraParameters["id_token"]
        try {
            val decoded = JWT.decode(token ?: return@get call.respond(HttpStatusCode.
                Unauthorized))
            val userId: String =
                decoded.getClaim("sub").asString() ?: return@get call.respond(HttpStatusCode.
                    Unauthorized)
            val admin: Boolean =
                decoded.getClaim("cognito:groups").asList(String::class.java)?.contains("
                    admin_users") ?: false

            call.sessions.set(AuthenticatedSession(userId, principal.accessToken, admin))
            cacheSession(token, JWTValidationResponse(userId, admin))
            val redirectUrl = call.request.queryParameters["redirect"] ?: "/"
            call.respondRedirect("$redirectDomain$redirectUrl")
        } catch (e: Exception) {
            return@get call.respond(HttpStatusCode.Unauthorized)
        }
    }
}
```

The implementation validates the received token, extracts the user ID and role information, creates a session, caches the session data in Redis, and redirects the user to the appropriate page.

Authentication Validation GET /api/auth/check

Verifies the validity of the current authentication session. This endpoint:

- Checks for the presence of an `AuthenticatedSession` cookie
- Validates the session token
- Checks Redis for cached session data to optimize performance
- Returns an appropriate response indicating the authentication status

Implementation Details:

```
fun Route.setUpCheckEndpoint(checkRoute: String) {
    get(checkRoute) {
        val sessionCookie =
            call.request.cookies["AuthenticatedSession"]?.let { cookie ->
                kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(cookie) }.
                    getOrNull()
            } ?: return@get call.respond(HttpStatusCode.Unauthorized, "Invalid or missing
                session cookie")

        checkCache(sessionCookie.token)?.let { cachedSession ->
            return@get call.respond(HttpStatusCode.OK, cachedSession)
        }

        call.response.headers.append(HttpHeaders.Authorization, "Bearer ${sessionCookie.token}
            ")
        call.response.headers.append(HttpHeaders.Location, "http://localhost:8000/api/auth/
            validate")
        call.respond(HttpStatusCode.TemporaryRedirect)
    }
}
```

The endpoint first checks for a cached session in Redis to avoid redundant validation. If no cached data is found, it redirects to the JWT validation endpoint.

JWT Validation GET /api/auth/validate

Validates a JWT token provided either in the `AuthenticatedSession` cookie or in the Authorization header. This endpoint returns the user ID and role information if the token is valid.

Implementation Details:

```kotlin
fun Route.setUpJWTValidation(validationRoute: String) {
    get(validationRoute) {
        val token: String? =
            call.request.cookies["AuthenticatedSession"]?.let {
                runCatching { Json.decodeFromString<AuthenticatedSession>(it) }.getOrNull()?.
                    token
            } ?: call.request.headers["Authorization"]?.removePrefix("Bearer ")

        val response = validateJWT(token)
        return@get respondAuthenticationCheckRequest(response, token)
    }
}
```

The implementation extracts the JWT token from either the cookie or the Authorization header, validates it using the `validateJWT` function, and responds with the validation result.

User Information GET /api/auth/me

Retrieves detailed profile information for the authenticated user. This endpoint:

- Extracts the access token from the session cookie
- Makes a request to the identity provider's user info endpoint
- Processes and returns user attributes in a structured format

Implementation Details:

```kotlin
fun Route.setUpUserInfoRoute(
    route: String,
    verifierUrl: String = "https://cognito-idp.eu-west-2.amazonaws.com/",
    contentType: String = "application/x-amz-json-1.1",
    amzTarget: Boolean = true,
    amzApi: String = "AWSCognitoIdentityProviderService.GetUser",
) {
    get(route) {
        val sessionJson =
            call.request.cookies["AuthenticatedSession"] ?: return@get call.respond(
                HttpStatusCode.Unauthorized,
                "Missing authentication cookie",
            )

        val token =
            try {
                Json.decodeFromString<AuthenticatedSession>(sessionJson).token
            } catch (e: Exception) {
                return@get call.respond(HttpStatusCode.Unauthorized, "Invalid token format")
            }

        val response = buildUserInfoRequest(token, verifierUrl, contentType, amzTarget, amzApi
            ).sendRequest()
        if (!response.isSuccessful) {
            return@get call.respond(HttpStatusCode.Unauthorized, "Invalid token")
        }

        val userInfo = generateUserInfo(response)
        if (userInfo == CognitoUserInfo("", "", "")) {
            return@get call.respondText(
                "Internal Server Error",
                status = HttpStatusCode.InternalServerError,
            )
        }

        call.respondText(
            Json.encodeToString<CognitoUserInfo>(userInfo),
            status = HttpStatusCode.OK,
            contentType = ContentType.Application.Json,
        )
```

```
    }
}
```

The endpoint retrieves user attributes from the identity provider and returns them in a standardized format.

Logout POST /api/auth/logout

Terminates the user's session and invalidates the authentication token. This endpoint:

- Clears the session data
- Expires the session cookie
- Removes cached session data from Redis

Implementation Details:

```kotlin
private fun Route.setLogOutEndpoint(route: String) {
    post(route) {
        val cookie =
            call.request.cookies["AuthenticatedSession"]?.let {
                kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(it) }.
                    getOrNull()
            } ?: return@post call.respond(HttpStatusCode.OK)

        call.sessions.clear<AuthenticatedSession>()
        call.response.cookies.append(
            Cookie(
                "AuthenticatedSession",
                "",
                path = "/",
                httpOnly = true,
                secure = true,
                expires = GMTDate(0),
            ),
        )
        // Remove cached session
        Redis.getRedisConnection().use { jedis ->
            jedis.del("auth:${cookie.token}")
        }
        call.respond(HttpStatusCode.OK)
    }
}
```

The logout endpoint ensures that the session is completely terminated both client-side and server-side.

These endpoints work together to provide a complete authentication flow, from initial login to session validation and logout, ensuring secure access to the application's resources.

7.1.6 Available Endpoints

The Authentication Module exposes several endpoints that handle different aspects of the authentication process. Each endpoint serves a specific purpose in the authentication flow.

Authentication Initiation GET /api/auth

Initiates the OAuth 2.0 authentication flow by redirecting the user to the identity provider's login page. This endpoint is the entry point for user authentication and requires no authentication itself.

Implementation Details:

```kotlin
private fun Route.setAuthenticationEndpoint(route: String) {
    get(route) {
        call.respondRedirect("/authenticate")
    }
}
```

The endpoint redirects to a central authentication route that is configured with the appropriate OAuth provider.

Authentication Callback GET /api/auth/callback

Handles the callback from the OAuth provider after successful authentication. This endpoint:

- Receives the authorization code from the provider
- Exchanges it for access and ID tokens
- Extracts user information from the JWT claims

- Creates a session for the authenticated user
- Redirects the user to the appropriate application page

Implementation Details:

```
fun Route.setUpCallbackRoute(
    route: String,
    redirectDomain: String = "http://localhost:5173",
) {
    get(route) {
        val principal: OAuthAccessTokenResponse.OAuth2? = call.authentication.principal()
        if (principal == null) {
            call.respond(HttpStatusCode.Unauthorized)
            return@get
        }
        val token: String? = principal.extraParameters["id_token"]
        try {
            val decoded = JWT.decode(token ?: return@get call.respond(HttpStatusCode.
                Unauthorized))
            val userId: String =
                decoded.getClaim("sub").asString() ?: return@get call.respond(HttpStatusCode.
                    Unauthorized)
            val admin: Boolean =
                decoded.getClaim("cognito:groups").asList(String::class.java)?.contains("
                    admin_users") ?: false

            call.sessions.set(AuthenticatedSession(userId, principal.accessToken, admin))
            cacheSession(token, JWTValidationResponse(userId, admin))
            val redirectUrl = call.request.queryParameters["redirect"] ?: "/"
            call.respondRedirect("$redirectDomain$redirectUrl")
        } catch (e: Exception) {
            return@get call.respond(HttpStatusCode.Unauthorized)
        }
    }
}
```

The implementation validates the received token, extracts the user ID and role information, creates a session, caches the session data in Redis, and redirects the user to the appropriate page.

Authentication Validation GET /api/auth/check

Verifies the validity of the current authentication session. This endpoint:

- Checks for the presence of an `AuthenticatedSession` cookie
- Validates the session token
- Checks Redis for cached session data to optimize performance
- Returns an appropriate response indicating the authentication status

Implementation Details:

```
fun Route.setUpCheckEndpoint(checkRoute: String) {
    get(checkRoute) {
        val sessionCookie =
            call.request.cookies["AuthenticatedSession"]?.let { cookie ->
                kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(cookie) }.
                    getOrNull()
            } ?: return@get call.respond(HttpStatusCode.Unauthorized, "Invalid or missing
                session cookie")

        checkCache(sessionCookie.token)?.let { cachedSession ->
            return@get call.respond(HttpStatusCode.OK, cachedSession)
        }

        call.response.headers.append(HttpHeaders.Authorization, "Bearer ${sessionCookie.token}
            ")
        call.response.headers.append(HttpHeaders.Location, "http://localhost:8000/api/auth/
            validate")
        call.respond(HttpStatusCode.TemporaryRedirect)
    }
}
```

The endpoint first checks for a cached session in Redis to avoid redundant validation. If no cached data is found, it redirects to the JWT validation endpoint.

JWT Validation GET /api/auth/validate

Validates a JWT token provided either in the `AuthenticatedSession` cookie or in the Authorization header. This endpoint returns the user ID and role information if the token is valid.

Implementation Details:

```
fun Route.setUpJWTValidation(validationRoute: String) {
    get(validationRoute) {
        val token: String? =
            call.request.cookies["AuthenticatedSession"]?.let {
                runCatching { Json.decodeFromString<AuthenticatedSession>(it) }.getOrNull()?.
                    token
            } ?: call.request.headers["Authorization"]?.removePrefix("Bearer ")

        val response = validateJWT(token)
        return@get respondAuthenticationCheckRequest(response, token)
    }
}
```

The implementation extracts the JWT token from either the cookie or the Authorization header, validates it using the `validateJWT` function, and responds with the validation result.

User Information GET /api/auth/me

Retrieves detailed profile information for the authenticated user. This endpoint:

- Extracts the access token from the session cookie
- Makes a request to the identity provider's user info endpoint
- Processes and returns user attributes in a structured format

Implementation Details:

```
fun Route.setUpUserInfoRoute(
    route: String,
    verifierUrl: String = "https://cognito-idp.eu-west-2.amazonaws.com/",
    contentType: String = "application/x-amz-json-1.1",
    amzTarget: Boolean = true,
    amzApi: String = "AWSCognitoIdentityProviderService.GetUser",
) {
    get(route) {
        val sessionJson =
            call.request.cookies["AuthenticatedSession"] ?: return@get call.respond(
                HttpStatusCode.Unauthorized,
                "Missing authentication cookie",
            )

        val token =
            try {
                Json.decodeFromString<AuthenticatedSession>(sessionJson).token
            } catch (e: Exception) {
                return@get call.respond(HttpStatusCode.Unauthorized, "Invalid token format")
            }

        val response = buildUserInfoRequest(token, verifierUrl, contentType, amzTarget, amzApi
            ).sendRequest()
        if (!response.isSuccessful) {
            return@get call.respond(HttpStatusCode.Unauthorized, "Invalid token")
        }

        val userInfo = generateUserInfo(response)
        if (userInfo == CognitoUserInfo("", "", "")) {
            return@get call.respondText(
                "Internal Server Error",
                status = HttpStatusCode.InternalServerError,
            )
        }

        call.respondText(
            Json.encodeToString<CognitoUserInfo>(userInfo),
            status = HttpStatusCode.OK,
            contentType = ContentType.Application.Json,
        )
    }
}
```

The endpoint retrieves user attributes from the identity provider and returns them in a standardized format.

Logout POST /api/auth/logout

Terminates the user's session and invalidates the authentication token. This endpoint:

- Clears the session data
- Expires the session cookie
- Removes cached session data from Redis

Implementation Details:

```
private fun Route.setLogOutEndpoint(route: String) {
    post(route) {
        val cookie =
            call.request.cookies["AuthenticatedSession"]?.let {
                kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(it) }.
                    getOrNull()
            } ?: return@post call.respond(HttpStatusCode.OK)

        call.sessions.clear<AuthenticatedSession>()
        call.response.cookies.append(
            Cookie(
                "AuthenticatedSession",
                "",
                path = "/",
                httpOnly = true,
                secure = true,
                expires = GMTDate(0),
            ),
        )
        // Remove cached session
        Redis.getRedisConnection().use { jedis ->
            jedis.del("auth:${cookie.token}")
        }
        call.respond(HttpStatusCode.OK)
    }
}
```

The logout endpoint ensures that the session is completely terminated both client-side and server-side.

These endpoints work together to provide a complete authentication flow, from initial login to session validation and logout, ensuring secure access to the application's resources.

7.1.7 Session Management

The Authentication Module implements robust session management to maintain user authentication state securely across multiple requests.

Session Implementation Sessions are managed using Ktor's built-in session management capabilities. The application configures sessions in the `authModule` function:

```
private fun Application.configureSessions() {
    install(Sessions) {
        cookie<AuthenticatedSession>("AuthenticatedSession") {
            cookie.maxAgeInSeconds = AuthenticationConstants.DEFAULT_EXPIRATION_SECONDS
            cookie.secure = true
            cookie.httpOnly = true
            cookie.path = "/"
            cookie.extensions["SameSite"] = "None"
        }
    }
}
```

Listing 15: Session Configuration

This configuration sets up several important security features:

- **Session Expiration**: Sessions expire after a configurable period, defined by `AuthenticationConstants.DEFAULT_EXPIRAT` This is set to 3600 seconds (1 hour) by default.
- **Secure Flag**: The `secure` flag ensures that cookies are only sent over HTTPS connections
- **HTTP-Only Flag**: The `httpOnly` flag prevents JavaScript from accessing the cookie, protecting against cross-site scripting (XSS) attacks
- **Path Setting**: The `path` parameter ensures the cookie is available across the entire application
- **SameSite Policy**: The `SameSite` attribute is set to "None" to allow cross-origin requests, which is necessary for the OAuth flow

Session Data Structure The session data is stored in the `AuthenticatedSession` data class:

```
@Serializable
data class AuthenticatedSession(
    val userId: String,
    val token: String,
    val admin: Boolean?,
)
```

Listing 16: AuthenticatedSession Data Class

This class contains the essential information needed to identify and authorize the user:

- `userId`: Uniquely identifies the authenticated user
- `token`: Contains the JWT used for API authorization
- `admin`: Indicates whether the user has administrative privileges

Session Lifecycle **Session Creation** Sessions are created during the OAuth callback process after successful authentication:

```
call.sessions.set(AuthenticatedSession(userId, principal.accessToken, admin))
```

Listing 17: Session Creation

Session Access In protected routes, the session is accessed to retrieve authentication information:

```
val sessionCookie = call.request.cookies["AuthenticatedSession"]?.let { cookie ->
    kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(cookie) }.getOrNull()
}
```

Listing 18: Session Access

Session Termination Sessions are terminated during logout:

```
call.sessions.clear<AuthenticatedSession>()
call.response.cookies.append(
    Cookie(
        "AuthenticatedSession",
        "",
        path = "/",
        httpOnly = true,
        secure = true,
        expires = GMTDate(0),
    ),
)
```

Listing 19: Session Termination

Error Handling The module implements robust error handling for session operations:

- **Deserialization Errors**: Session cookie parsing is wrapped in `runCatching` to safely handle malformed session data
- **Missing Sessions**: All code paths check for null session data and respond with appropriate error messages
- **Invalid Tokens**: Even with a valid session cookie, token validation is still performed to ensure the session has not been compromised

Session Persistence Session data exists only in the cookie and is not stored server-side in a traditional session store. However, validation results are cached in Redis to improve performance:

```
fun cacheSession(
    token: String,
    authData: JWTValidationResponse,
    expirySeconds: Long = 3600,
) {
    Redis.getRedisConnection().use { jedis ->
        jedis.setex("auth:$token", expirySeconds, Json.encodeToString(authData))
    }
}
```

Listing 20: Session Caching

The session management implementation provides a secure, performance-optimized approach to maintaining user authentication state throughout the application.

7.1.8 Security Considerations

The Authentication Module implements numerous security measures to protect user authentication data and prevent common web security vulnerabilities.

JWT Security JSON Web Tokens (JWTs) are central to the module's security model and are protected through several mechanisms:

- **Signature Verification**: All JWTs are cryptographically verified using the JWKS endpoint provided by the identity provider:

```
    jwkProvider =
        JwkProviderBuilder(issuer)
            .cached(10, 24, TimeUnit.HOURS)
            .rateLimited(10, 1, TimeUnit.MINUTES)
            .build()
```

Listing 21: JWKS Configuration

- **Expiration Validation**: Tokens are checked for expiration during validation:

```
    if (decoded.expiresAt.before(Date.from(Instant.now()))) {
        return null
    }
```

Listing 22: Expiration Check

- **Required Claims**: Essential claims like `sub` (subject identifier) are required for successful validation:

```
    val userId = decoded.getClaim("sub").asString() ?: return null
```

Listing 23: Required Claims Check

Cookie Security Session cookies are secured through multiple protective measures:

- **HTTP-Only Flag**: Prevents JavaScript access to the cookie, protecting against XSS attacks
- **Secure Flag**: Ensures cookies are only transmitted over HTTPS connections
- **SameSite Policy**: Configures cross-origin behavior to prevent CSRF attacks while allowing OAuth flows
- **Limited Lifetime**: Cookies have a default expiration of 10 minutes, reducing the window of opportunity for attacks
- **Minimal Data**: Only essential authentication data is stored in the cookie

Role-Based Access Control The module implements role-based access control through JWT claims:

```
val admin = decoded.getClaim("cognito:groups").asList(String::class.java)?.contains("
    admin_users") ?: false
```

Listing 24: Role Extraction

This approach allows protected routes to enforce authorization based on user roles:

```
// Example of how a route might use the admin flag for authorization
get("/admin/dashboard") {
    val session = call.principal<JWTPrincipal>()
    val userId = session?.payload?.getClaim("sub")?.asString()
    val admin = session?.payload?.getClaim("cognito:groups")?.asList(String::class.java)?.
        contains("admin_users") ?: false

    if (!admin) {
        call.respond(HttpStatusCode.Forbidden, "Admin access required")
        return@get
    }

    // Admin-only functionality
    call.respond(HttpStatusCode.OK, "Admin dashboard")
}
```

Listing 25: Role-Based Authorization Example

Rate Limiting The JWKS endpoint used for token verification is protected against denial-of-service attacks through rate limiting:

```
jwkProvider =
    JwkProviderBuilder(issuer)
        .cached(10, 24, TimeUnit.HOURS)
        .rateLimited(10, 1, TimeUnit.MINUTES)
        .build()
```

Listing 26: JWKS Rate Limiting

This configuration limits requests to the JWKS endpoint to 10 per minute and caches responses for 24 hours, reducing the load on both the application and the identity provider.

Error Handling and Logging The module implements secure error handling practices:

- **Generic Error Messages**: Authentication failures return standardized error messages without revealing detailed information that could aid attackers
- **Graceful Error Recovery**: Exceptions during authentication are caught and handled to prevent information leakage
- **Secure Logging**: Authentication events are logged for audit purposes without exposing sensitive data

Defense in Depth The module employs a defense-in-depth strategy with multiple security layers:

- **Multiple Validation Points**: Token validation occurs at both the `/api/auth/check` endpoint and within protected routes
- **Secure Transport**: All authentication traffic uses HTTPS (enforced by the `secure` cookie flag)
- **Short-Lived Tokens**: Session expiration limits the impact of compromised credentials
- **Separation of Concerns**: Authentication logic is isolated from application logic, reducing the attack surface

Security Testing The authentication module includes comprehensive tests that verify security mechanisms:

```
@Test
fun `Test JWT validation when token has no sub claim`() =
    testApplication {
        application {
            this@application.install(ContentNegotiation) {
                json()
            }
            this@application.routing {
                setUpJWTValidation("/validate")
            }
        }
        val response =
            client.get("/validate") {
                header(HttpHeaders.Authorization, "Bearer ${AuthenticationTestHelpers.
                    generateTestJwtTokenNoSub()}")
            }
        assertEquals(HttpStatusCode.Unauthorized, response.status)
    }
```

Listing 27: Security Test Example

These tests ensure that security mechanisms work as expected and are not bypassed during code changes.

By implementing these security measures, the Authentication Module provides robust protection for user credentials and session data while maintaining a seamless user experience.

7.1.9 Configuration

The Authentication Module uses a flexible configuration approach that allows for easy customization and adaptation to different authentication providers and requirements.

Configuration File Authentication settings are loaded from a JSON configuration file, which contains the necessary details for connecting to the OAuth provider and validating JWTs:

```
{
  "name": "Cognito",
  "jwtIssuer": "",
  "urlProvider": "http://localhost:8000/api/auth/callback",
```

```
  "providerLookup": {
    "name": "Cognito",
    "authorizeUrl": "",
    "accessTokenUrl": "",
    "clientId": "",
    "clientSecret": "",
    "defaultScopes": [
      "email",
      "openid",
      "profile"
    ]
  }
}
```

Listing 28: Example Configuration File (cognito.json)

This configuration includes:

- **name**: The identifier for the authentication provider
- **jwtIssuer**: The URL of the JWT issuer, used for token validation
- **urlProvider**: The callback URL for the OAuth flow
- **providerLookup**: Detailed OAuth provider settings, including:
  - **authorizeUrl**: The authorization endpoint URL
  - **accessTokenUrl**: The token endpoint URL
  - **clientId**: The OAuth client ID
  - **clientSecret**: The OAuth client secret
  - **defaultScopes**: The OAuth scopes requested during authentication

Configuration Loading The configuration is loaded during application startup in the `authModule` function:

```
fun Application.authModule(
    configFilePath: String = AuthenticationConstants.CONFIGURATION_FILE_PATH,
    authName: String = AuthenticationConstants.DEFAULT_AUTHENTICATOR,
) {
    configureAuthentication(configFilePath)
    configureSessions()
    configureAuthenticationRoutes(authName = authName)
}

private fun Application.configureAuthentication(configFilePath: String) {
    val config = PoCJSON.readJsonFile(configFilePath)
    try {
        install(Authentication) {
            configureOAuth(config)
            configureJWTValidator(config)
        }
    } catch (e: DuplicatePluginException) {
        print("")
    }
}
```

Listing 29: Configuration Loading

The configuration file path can be customized through the `configFilePath` parameter, with a default value defined in `AuthenticationConstants`:

```
object AuthenticationConstants {
    const val DEFAULT_EXPIRATION_SECONDS = 600L
    const val CONFIGURATION_FILE_PATH = "auth/src/main/resources/cognito.json"
    const val DEFAULT_AUTHENTICATOR = "Cognito"
}
```

Listing 30: Configuration Constants

Configuration Application The configuration is applied to the Ktor authentication system using extension functions in the `Authenticators` object:

```
fun AuthenticationConfig.configureOAuth(config: JsonObject) {
    val providerLookupData = config["providerLookup"]!!.jsonObject
    oauth(config["name"]!!.jsonPrimitive.content) {
        urlProvider = { config["urlProvider"]!!.jsonPrimitive.content }
        providerLookup = {
```

```
            OAuthServerSettings.OAuth2ServerSettings(
                name = providerLookupData["name"]!!.jsonPrimitive.content,
                authorizeUrl = providerLookupData["authorizeUrl"]!!.jsonPrimitive.content,
                accessTokenUrl = providerLookupData["accessTokenUrl"]!!.jsonPrimitive.content,
                clientId = providerLookupData["clientId"]!!.jsonPrimitive.content,
                clientSecret = providerLookupData["clientSecret"]!!.jsonPrimitive.content,
                defaultScopes = providerLookupData["defaultScopes"]!!.jsonArray
                    .map { it.jsonPrimitive.content },
                requestMethod = HttpMethod.Post,
            )
        }
        client = HttpClient(CIO)
    }
}
```

Listing 31: OAuth Configuration Application

```
fun AuthenticationConfig.configureJWTValidator(config: JsonObject) {
    val issuer = config["jwtIssuer"]!!.jsonPrimitive.content
    jwkProvider =
        JwkProviderBuilder(issuer)
            .cached(10, 24, TimeUnit.HOURS)
            .rateLimited(10, 1, TimeUnit.MINUTES)
            .build()

    generateVerifier(jwkProvider, issuer)
}
```

Listing 32: JWT Configuration Application

Multiple Authentication Providers The architecture supports multiple authentication providers through named authentication configurations:

```
fun Application.configureAuthenticationRoutes(authName: String) {
    routing {
        authenticate(authName) {
            setAuthenticationEndpoint(AUTHENTICATION_ROUTE)
            setUpCallbackRoute(CALL_BACK_ROUTE)
        }
        authenticate("jwt-verifier") {
            setUpJWTValidation(JWT_VALIDATION_ROUTE)
            setUpUserInfoRoute(USER_INFO_ROUTE)
        }
        setUpCheckEndpoint(AUTHENTICATION_CHECK_ROUTE)
        setLogOutEndpoint(LOG_OUT_ROUTE)
    }
}
```

Listing 33: Authentication Provider Selection

The `authName` parameter allows different authentication providers to be used, with a default value defined in `AuthenticationConstants`.

Configuration Testing The module includes tests specifically for configuration handling:

```
@Test
fun `Test authModule with default params`() =
    testApplication {
        application {
            try {
                authModule()
            } catch (e: Exception) {
                print("Entered catch block")
            }
            assertTrue(true)
        }
    }
```

Listing 34: Configuration Test Example

This flexible configuration approach allows the Authentication Module to be easily adapted to different environments and authentication requirements without modifying the core implementation.

7.1.10 Redis Caching

The Authentication Module leverages Redis as a distributed caching system to optimize authentication performance and reduce the computational overhead of token validation.

Caching Purpose JWT validation can be a computationally expensive operation that involves:

- Cryptographic signature verification
- Token structure validation
- Claims extraction and validation

To avoid performing these operations for every request, the module caches validation results in Redis. This approach significantly improves performance, especially for applications with high traffic volumes.

Redis Connection Management Redis connectivity is managed through the `Redis` object, which provides a connection pool for efficient resource utilization:

```
internal object Redis {
    private val REDIS_HOST = EnvironmentLoader.get("REDIS_HOST")
    private const val REDIS_PORT = 6379
    private val redisPool = JedisPool(JedisPoolConfig(), REDIS_HOST, REDIS_PORT)

    fun getRedisConnection(): Jedis = redisPool.resource
}
```

Listing 35: Redis Connection Management

The `Redis` object:

- Creates a connection pool to manage Redis connections efficiently
- Loads the Redis host from environment variables for configuration flexibility
- Provides a method to obtain a connection from the pool

Session Caching Implementation When a user successfully authenticates, their session information is cached in Redis:

```
internal fun cacheSession(
    token: String,
    authData: JWTValidationResponse,
    expirySeconds: Long = 3600,
) {
    Redis.getRedisConnection().use { jedis ->
        jedis.setex("auth:$token", expirySeconds, Json.encodeToString(authData))
    }
}
```

Listing 36: Cache Storage Implementation

Key aspects of this implementation:

- The token itself is used as the cache key (with an "auth:" prefix)
- The `JWTValidationResponse` object is serialized to JSON for storage
- An expiration time of 1 hour (3600 seconds) is set by default
- The Redis connection is automatically returned to the pool after use via the `use` function

Cache Retrieval When a request needs to validate a token, the module first checks the Redis cache:

```
internal fun checkCache(token: String): JWTValidationResponse? {
    Redis.getRedisConnection().use { jedis ->
        val cachedData = jedis["auth:$token"] ?: return null
        return Json.decodeFromString<JWTValidationResponse>(cachedData)
    }
}
```

Listing 37: Cache Retrieval Implementation

This function:

- Attempts to retrieve cached data using the token as the key
- Returns null if no cache entry is found
- Deserializes the cached JSON data to a `JWTValidationResponse` object if found
- Manages the Redis connection through the `use` function

Cache Usage in Authentication Flow The cache is used in the authentication check endpoint to optimize token validation:

```
internal fun Route.setUpCheckEndpoint(checkRoute: String) {
    get(checkRoute) {
        val sessionCookie =
            call.request.cookies["AuthenticatedSession"]?.let { cookie ->
                kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(cookie) }.
                    getOrNull()
            } ?: return@get call.respond(HttpStatusCode.Unauthorized, "Invalid or missing
                session cookie")

        checkCache(sessionCookie.token)?.let { cachedSession ->
            return@get call.respond(HttpStatusCode.OK, cachedSession)
        }

        call.response.headers.append(HttpHeaders.Authorization, "Bearer ${sessionCookie.token}
            ")
        call.response.headers.append(HttpHeaders.Location, "http://localhost:8000/api/auth/
            validate")
        call.respond(HttpStatusCode.TemporaryRedirect)
    }
}
```

Listing 38: Cache Usage in Authentication Check

The flow:

1. Extract the session token from the cookie
2. Check if validation results are cached in Redis
3. If found, return the cached results immediately
4. If not found, redirect to the full validation endpoint

Cache Invalidation Cache entries are invalidated in two ways:

1. **Automatic Expiration**: Each cache entry has a time-to-live (TTL) of 1 hour by default
2. **Explicit Deletion**: During logout, the cache entry is explicitly removed:

```
    // Remove cached session
    Redis.getRedisConnection().use { jedis ->
        jedis.del("auth:${cookie.token}")
    }
```

Listing 39: Cache Invalidation During Logout

Testing with Redis The module includes tests that verify Redis caching functionality:

```
@Test
fun `Test check route with valid cookie, cached`() =
    testApplication {
        install(ContentNegotiation) {
            json()
        }
        routing {
            setUpCheckEndpoint("/check")
        }
        val jwt = AuthenticationTestHelpers.generateTestJwtToken()
        val sessionCookie =
            AuthenticatedSession(
                userId = "user123",
                token = jwt,
                admin = true,
            )
        val validationResponse = JWTValidationResponse("user123", true)
        cacheSession(jwt, validationResponse)
        println("REDIS CACHE: ${Redis.getRedisConnection().get("auth:$jwt")}")
        val response =
            client.get("/check") {
                cookie("AuthenticatedSession", Json.encodeToString(sessionCookie))
            }
        assertEquals(HttpStatusCode.OK, response.status)
        assertEquals(validationResponse, Json.decodeFromString<JWTValidationResponse>(response
            .bodyAsText()))
```

```
        Redis.getRedisConnection().del("auth:$jwt") // Clean up Redis
    }
```

Listing 40: Redis Caching Test

This test:

- Manually caches a session
- Verifies that the endpoint returns the cached data
- Cleans up the Redis entry after the test

Redis Configuration For the Redis caching to function, Redis must be properly configured:

- Redis should be installed and running on the port specified in the configuration
- The `REDIS_HOST` environment variable must be correctly set
- For production deployments, Redis should be configured with appropriate security settings

The Redis caching implementation significantly improves authentication performance by reducing the computational overhead of token validation while maintaining security through appropriate cache expiration and invalidation.

7.1.11 Best Practices and Usage Guidelines

This section provides best practices and guidelines for working with the Authentication Module, covering usage patterns, security considerations, and maintenance recommendations.

Integrating Authentication into Routes To protect routes with authentication, follow these best practices:

- **Use the `authenticate` Middleware**:

```
routing {
    // Public routes
    get("/public") {
        call.respondText("This is a public endpoint")
    }

    // Protected routes
    authenticate("jwt-verifier") {
        get("/protected") {
            // Access principal for user information
            val principal = call.principal<JWTPrincipal>()
            val userId = principal?.payload?.getClaim("sub")?.asString()
            call.respondText("Hello, $userId!")
        }
    }
}
```

Listing 41: Authentication Middleware Usage

- **Access User Information Consistently**:

```
val principal = call.principal<JWTPrincipal>()
val userId = principal?.payload?.getClaim("sub")?.asString()
val isAdmin = principal?.payload?.getClaim("cognito:groups")
    ?.asList(String::class.java)
    ?.contains("admin_users") ?: false
```

Listing 42: Accessing User Information

- **Handle Authentication Failures Gracefully**:

```
// The authenticate middleware automatically handles failures,
// but you can add custom status pages for a better user experience
install(StatusPages) {
    status(HttpStatusCode.Unauthorized) { call, _ ->
        call.respondText(
            "You need to be logged in to access this resource",
            status = HttpStatusCode.Unauthorized
        )
    }
}
```

Listing 43: Authentication Failure Handling

Security Best Practices Follow these security best practices when working with the Authentication Module:

- **Protect Sensitive Configuration**: Never commit the `cognito.json` file with real credentials to version control. Use environment variables or secure parameter stores for production deployments.
- **Implement Proper Authorization**: Authentication verifies identity, but authorization controls access. Always implement proper authorization checks:

```
get("/admin-dashboard") {
    val principal = call.principal<JWTPrincipal>()
    val isAdmin = principal?.payload?.getClaim("cognito:groups")
        ?.asList(String::class.java)
        ?.contains("admin_users") ?: false

    if (!isAdmin) {
        call.respond(HttpStatusCode.Forbidden, "Admin access required")
        return@get
    }

    // Admin-only functionality
}
```

Listing 44: Authorization Example

- **Use Appropriate Token Scopes**: Request only the OAuth scopes your application needs. The default configuration includes `email`, `openid`, and `profile`.
- **Set Appropriate Session Expiration**: Adjust `AuthenticationConstants.DEFAULT_EXPIRATION_SECONDS` based on your security requirements. Shorter expiration times are more secure but may impact user experience.
- **Configure Redis Securely**: In production environments, ensure Redis is properly secured:
  - Enable authentication
  - Use encrypted connections
  - Restrict network access
  - Regularly rotate Redis credentials

Performance Optimization To optimize authentication performance:

- **Use the `/api/auth/check` Endpoint**: This endpoint leverages Redis caching for efficient validation.
- **Configure Appropriate Cache TTL**: The default cache TTL is 1 hour (3600 seconds). Adjust this based on your security requirements and traffic patterns:

```
// For shorter cache lifetime
cacheSession(token, validationResponse, 1800) // 30 minutes

// For longer cache lifetime
cacheSession(token, validationResponse, 7200) // 2 hours
```

Listing 45: Custom Cache TTL

- **Monitor Redis Performance**: For high-traffic applications, monitor Redis performance and scale as needed.

Testing Authentication Properly test authentication functionality:

- **Mock Authentication for Tests**: Use the test helpers provided in `AuthenticationTestHelpers`:

```
@Test
fun `Test protected endpoint`() = testApplication {
    // Configure test authentication
    authentication {
        mock<OAuthAccessTokenResponse.OAuth2>("test") {
            principal = // mock principal
        }
    }

    // Test your protected routes
}
```

Listing 46: Mocking Authentication in Tests

- **Test Both Positive and Negative Cases**: Ensure you test both successful authentication and various failure scenarios.

Maintenance and Troubleshooting For ongoing maintenance:

- **Monitor Token Validation Failures**: A high rate of validation failures could indicate configuration issues or potential attacks.
- **Regular Configuration Review**: Periodically review authentication configuration, especially when updating identity provider settings.
- **Keep Dependencies Updated**: Regularly update the authentication libraries to benefit from security patches and new features.
- **Redis Connection Management**: Ensure Redis connections are properly closed using the `use` pattern to prevent connection leaks:

```
    Redis.getRedisConnection().use { jedis ->
        // Redis operations
    } // Connection automatically returned to pool
```

Listing 47: Proper Redis Connection Usage

By following these best practices, you can ensure that your implementation of the Authentication Module is secure, performant, and maintainable, providing a reliable foundation for your application's security model.

7.2 Chat Module

The Chat Module provides a robust framework for handling real-time chat interactions, message processing, file uploads, and conversation management. Operating behind JWT authentication middleware, it ensures secure access to all endpoints while maintaining efficient message handling and storage.

7.2.1 Overview

The Chat Module serves as the central component for managing all chat-related interactions within the ProofIt application. It provides a secure and efficient interface for handling chat messages, file uploads, and conversation history management. The module is designed to support both synchronous and asynchronous communication patterns, enabling users to engage with the system through a conversational interface.

Key features of the Chat Module include:

- **Secure Message Processing**: All chat endpoints operate behind JWT authentication middleware to ensure that only authenticated users can access chat functionality.
- **Structured Message Format**: The module uses a well-defined message structure that supports various content types, including text, code snippets, and references to uploaded files.
- **File Upload Handling**: Comprehensive support for file uploads, including multi-part form data processing, metadata handling, and secure storage management.
- **Conversation Management**: The module maintains conversation contexts and histories, allowing for persistent and coherent interaction sequences.
- **Database Integration**: Seamless integration with the database module for storing message history, conversation metadata, and file references.
- **Input Sanitization**: Robust security measures including input sanitization to prevent injection attacks and other security vulnerabilities.
- **Prototype Generation**: Integration with the prototype generation functionality, allowing chat interactions to initiate the creation of application prototypes.

The Chat Module is a critical component of the application that facilitates natural and intuitive interactions between users and the system. It is designed to be scalable, secure, and maintainable, with a focus on providing a seamless user experience.

7.2.2 Architecture

The Chat Module follows a well-structured architecture that separates concerns and promotes maintainability. This architecture ensures that different aspects of chat functionality—such as message processing, file handling, and storage—are modularized and cohesive.

Modular Organization The Chat Module is organized into several key components:

- **Route Definitions**: Central endpoint definitions that establish the HTTP routes for chat operations.
- **Request Handlers**: Specialized components that process incoming requests for different chat operations.

- **Data Models**: Structured data classes that represent chat messages, requests, and responses.
- **Storage Integrations**: Components that interface with the database module for persistence.
- **Security Middleware**: Authentication and authorization mechanisms that protect chat endpoints.
- **File Processing**: Components dedicated to handling file uploads and processing.

Request Flow The typical request flow through the Chat Module involves the following steps:

1. An authenticated request arrives at one of the chat endpoints.
2. The JWT authentication middleware validates the request's credentials.
3. The appropriate route handler processes the request, parsing parameters and validating inputs.
4. For message requests, the content is sanitized and processed before being passed to the relevant business logic.
5. For file uploads, the multipart data is parsed, and files are securely stored.
6. The response is generated based on the operation's outcome and returned to the client.
7. In parallel, messages and metadata are stored in the database for persistence.

Integration with Other Modules The Chat Module integrates with several other system components:

- **Authentication Module**: Leverages JWT authentication for securing endpoints.
- **Database Module**: Uses repository patterns for storing and retrieving chat data.
- **Prompting Module**: Interfaces with the language model integration for generating responses.
- **Prototype Module**: Enables chat interactions to trigger prototype generation.

Architectural Patterns The Chat Module implements several architectural patterns:

- **Middleware Pattern**: Uses request processing middleware for cross-cutting concerns like authentication and logging.
- **Repository Pattern**: Abstracts database operations through specialized repository interfaces.
- **Factory Pattern**: Employs factory methods for creating and managing complex objects.
- **Command Pattern**: Structures message processing as discrete command operations.

Extensibility The modular architecture allows the Chat Module to be extended in several ways:

- New message types can be added by extending the message processing pipeline.
- Additional file formats can be supported by implementing new file processors.
- New storage strategies can be integrated by implementing the appropriate repository interfaces.
- Security mechanisms can be enhanced by adding middleware components.

This well-structured architecture ensures that the Chat Module can evolve to meet changing requirements while maintaining robustness and security.

7.2.3 Core Components

The Chat Module consists of several core components that work together to provide a comprehensive chat functionality. These components handle different aspects of chat operations, from message processing to file handling.

ChatEndpoint The `ChatEndpoint` object defines the base routes for the chat API and manages the upload directory configuration:

```
object ChatEndpoint {
    internal var UPLOAD_DIR: String = "uploads" // Configurable for testing
}

// Constants for route paths
const val CHAT = "/api/chat"
const val GET = "$CHAT/history"
const val JSON = "$CHAT/json"
const val UPLOAD = "$CHAT/upload"
```

Listing 48: ChatEndpoint Definition

This component centralizes the route definitions and provides a configurable upload directory setting that can be adjusted for different environments, including testing scenarios.

Request The `Request` data class represents a chat message request from a user:

```
@Serializable
data class Request(
    val userID: String,
    val time: String,
    val prompt: String,
    val conversationId: String = "default-conversation",
)
```

Listing 49: Request Data Class

This class encapsulates:

- `userID`: Identifier for the user sending the message
- `time`: Timestamp indicating when the request was created
- `prompt`: The actual content/message submitted by the user
- `conversationId`: Optional identifier for the conversation context, with a default value

Response The `Response` data class represents the structured response to file upload operations:

```
@Serializable
data class Response(
    val time: String,
    val message: String,
)
```

Listing 50: Response Data Class

This class contains:

- `time`: Timestamp indicating when the response was created
- `message`: Response message providing feedback about the operation

UploadData The `UploadData` class tracks information during the file upload process:

```
private data class UploadData(
    var fileDescription: String = "",
    var fileName: String = "",
    var message: Request? = null,
    var response: Response? = null,
)
```

Listing 51: UploadData Class

This class manages:

- `fileDescription`: Optional description text for the uploaded file
- `fileName`: Name of the file being uploaded (with timestamp modification)
- `message`: Optional structured message submitted with the upload
- `response`: The response object to be sent back to the client

Chat Message and Conversation Models The module defines data models for chat messages and conversations:

```
@Serializable
data class MessageDto(
    val id: String,
    val conversationId: String,
    val senderId: String,
    val content: String,
    val timestamp: String
)

@Serializable
data class Conversation(
    val id: String,
    val name: String,
    val lastModified: String,
    val messageCount: Int,
    val userId: String
)

```

```
@Serializable
data class ConversationHistory(
    val conversations: List<Conversation>
)
```

Listing 52: Chat Message and Conversation Models

These models represent:

- Individual chat messages with sender information and content
- Conversation metadata including names and message counts
- Collections of conversations for history retrieval

Route Handlers The module includes several specialized route handlers for different chat operations:

- `chatRoutes`: Handles conversation history and message retrieval
- `jsonRoutes`: Processes incoming JSON message requests
- `uploadRoutes`: Manages file upload operations

```
fun Application.chatModule() {
    routing {
        authenticate("jwt-verifier") {
            chatRoutes()
            jsonRoutes()
            uploadRoutes(ChatEndpoint.UPLOAD_DIR)
        }
    }
}
```

Listing 53: Chat Module Registration

This setup ensures that all chat routes are protected by the JWT authentication middleware, providing a secure chat environment. The module's organization allows for easy extension and maintenance, with each component focused on a specific aspect of chat functionality.

7.2.4 Available Endpoints

The Chat Module exposes several HTTP endpoints for different chat operations. All endpoints are protected by JWT authentication to ensure secure access.

Conversation History Endpoints GET /api/chat/history

Retrieves a list of conversations for the authenticated user.

Query Parameters:

- `userId` (optional): Filter conversations by user ID. Defaults to "user" if not provided.

Response: A JSON object containing a list of conversation objects with metadata.

Implementation Details:

```
internal fun Route.chatRoutes() {
    get(GET) {
        try {
            println("Fetching conversations")
            val userId = call.request.queryParameters["userId"] ?: "user"

            val conversations = getConversationHistory(userId).map {
                Conversation(
                    id = it.id,
                    name = it.name,
                    lastModified = it.lastModified,
                    messageCount = it.messageCount,
                    userId = it.userId
                )
            }
            println("Fetched ${conversations.size} conversations")
            call.respond(ConversationHistory(conversations))
        } catch (e: Exception) {
            return@get call.respondText(
                "Error: ${e.message}",
                status = HttpStatusCode.InternalServerError
            )
        }
```

```
    }
}
```

Listing 54: Conversation History Endpoint

The endpoint leverages the database storage layer to retrieve conversation histories and transforms the database entities into DTO objects suitable for API responses.

GET /api/chat/history/{conversationId}

Retrieves messages from a specific conversation.

Path Parameters:

- `conversationId`: The unique identifier for the conversation.

Query Parameters:

- `limit` (optional): Maximum number of messages to retrieve. Defaults to 50.
- `offset` (optional): Number of messages to skip. Defaults to 0.

Response: A JSON array of message objects ordered by timestamp.
Implementation Details:

```
get("$GET/{conversationId}") {
    try {
        val conversationId = call.parameters["conversationId"] ?: return@get call.respondText(
            "Missing conversation ID",
            status = HttpStatusCode.BadRequest
        )

        val limit = call.request.queryParameters["limit"]?.toIntOrNull() ?: 50
        val offset = call.request.queryParameters["offset"]?.toIntOrNull() ?: 0

        println("Fetching messages")
        val messages = getMessageHistory(conversationId, limit)
        println("Fetched $messages messages")

        val messageDtos = messages.map { message ->
            MessageDto(
                id = message.id,
                conversationId = message.conversationId,
                senderId = message.senderId,
                content = message.content,
                timestamp = message.timestamp.toString()
            )
        }

        call.respond(messageDtos)
    } catch (e: Exception) {
        println("Error getting messages: ${e.message}")
        e.printStackTrace()
        return@get call.respondText(
            "Error: ${e.message}",
            status = HttpStatusCode.InternalServerError
        )
    }
}
```

Listing 55: Message History Endpoint

This endpoint retrieves message histories for specific conversations, with support for pagination through the limit and offset parameters.

GET /api/chat/history/{conversationId}/{messageId}

Retrieves prototype data associated with a specific message.

Path Parameters:

- `conversationId`: The unique identifier for the conversation.
- `messageId`: The unique identifier for the message.

Response: A JSON object containing the prototype data if available.
Implementation Details:

```
get("$GET/{conversationId}/{messageId}") {
    try {
        println("Fetching prototype")
        val conversationId = call.parameters["conversationId"] ?: return@get call.respondText(
            "Missing conversation ID",
```

```
            status = HttpStatusCode.BadRequest
        )
        val messageId = call.parameters["messageId"] ?: return@get call.respondText(
            "Missing message ID",
            status = HttpStatusCode.BadRequest
        )

        val prototype = retrievePrototype(conversationId, messageId)

        if (prototype != null) {
            call.respond(PrototypeDto(files = prototype.filesJson))
        }
    } catch (e: Exception) {
        println("Error getting messages: ${e.message}")
        e.printStackTrace()
        return@get call.respondText(
            "Error: ${e.message}",
            status = HttpStatusCode.InternalServerError
        )
    }
}
```

Listing 56: Prototype Retrieval Endpoint

This endpoint retrieves prototype data that may be associated with specific chat messages, enabling the application to display and manage generated prototypes.

Message Processing Endpoints POST /api/chat/json

Processes a new chat message and generates a response.

Request Body: A JSON object conforming to the `Request` data class:

- `userID`: The user's identifier
- `time`: Timestamp for the message
- `prompt`: The message content
- `conversationId`: The conversation context

Response: A `ServerResponse` object containing the generated response and any associated prototype data.

Implementation Details:

```
fun Route.jsonRoutes() {
    post(JSON) {
        println("Received JSON request")
        val request: Request =
            runCatching {
                call.receive<Request>()
            }.getOrElse {
                return@post call.respondText(
                    "Invalid request ${it.message}",
                    status = HttpStatusCode.BadRequest,
                )
            }
        handleJsonRequest(request, call)
    }
}

private suspend fun handleJsonRequest(
    request: Request,
    call: ApplicationCall,
) {
    println("Handling JSON request: ${request.prompt} from ${request.userID} for conversation
        ${request.conversationId}")
    saveMessage(request.conversationId, request.userID, request.prompt)

    val response = getPromptingMain().run(request.prompt)

    val savedMessage = saveMessage(request.conversationId, "LLM", response.chat.message)
    response.prototype?.let { prototypeResponse ->
        val prototype = Prototype(
            messageId = savedMessage.id,
            filesJson = prototypeResponse.files.toString(),
            version = 1,
            isSelected = true
        )
        storePrototype(prototype)
    }
```

```
    println("MessageId: ${savedMessage.id}")

    val responseWithId = response.copy(
        chat = response.chat.copy(messageId = savedMessage.id)
    )

    println("RECEIVED RESPONSE")
    val jsonString = Json.encodeToString(ServerResponse.serializer(), responseWithId)
    println("ENCODED RESPONSE: $jsonString")

    call.respondText(jsonString, contentType = ContentType.Application.Json)
}
```

Listing 57: JSON Message Endpoint

This endpoint processes chat messages, saves them to the database, generates responses using the prompting module, and handles any prototype data that may be generated.

File Upload Endpoints POST /api/chat/upload

Handles file uploads with support for multi-part form data.

Request Form Data:

- `file`: The file to upload
- `description` (optional): Description of the file
- `message` (optional): JSON string conforming to the `Request` data class

Response: Either a `Response` object or a text message confirming the upload.

Implementation Details:

```
fun Route.uploadRoutes(uploadDir: String) {
    post(UPLOAD) {
        val uploadData = UploadData()
        val uploadDir = createUploadDirectory(uploadDir)

        val multipartData = call.receiveMultipart()
        multipartData.forEachPart { part ->
            handlePart(part, uploadDir, uploadData, call)
        }

        respondToUpload(call, uploadData)
    }
}
```

Listing 58: File Upload Endpoint

The implementation involves several helper functions:

- `createUploadDirectory`: Ensures the upload directory exists
- `handlePart`: Processes each part of the multipart request
- `handleFormItem`: Processes text form items like descriptions
- `handleFileItem`: Processes file items, saving them to the filesystem
- `respondToUpload`: Generates an appropriate response after upload

This endpoint provides comprehensive file upload capabilities, with support for metadata and integration with the chat message system.

Conversation Management Endpoints POST /api/chat/json/{conversationId}/rename

Renames an existing conversation.

Path Parameters:

- `conversationId`: The unique identifier for the conversation to rename.

Request Body: A JSON object with a `name` field containing the new conversation name.

Response: A success or error message.

Implementation Details:

```
post("$JSON/{conversationId}/rename") {
    try {
        println("Received conversation rename request")
        val conversationId = call.parameters["conversationId"] ?: throw
            IllegalArgumentException("Missing ID")
        val requestBody = call.receive<Map<String, String>>()
```

```
        val name = requestBody["name"] ?: throw IllegalArgumentException("Missing name")
        val success = updateConversationName(conversationId, name)
        println("Renamed conversation $conversationId to $name")
        if (success) {
            call.respondText("Conversation renamed successfully", status = HttpStatusCode.OK)
        } else {
            call.respondText("Failed to update name", status = HttpStatusCode.
                InternalServerError)
        }
    } catch (e: Exception) {
        call.respondText(
            "Error: ${e.message}",
            status = HttpStatusCode.BadRequest
        )
    }
}
```

Listing 59: Conversation Rename Endpoint

This endpoint allows users to rename their conversations for better organization and context management.

These endpoints collectively provide a comprehensive API for chat operations, enabling message exchange, file uploads, and conversation management within the application.

7.2.5 Message Processing

The Chat Module implements a sophisticated message processing pipeline that handles everything from receiving raw messages to generating responses and storing conversation history. This section details the key components and workflows involved in message processing.

Message Flow When a message is received through the `/api/chat/json` endpoint, it goes through the following processing steps:

1. **Message Validation**: The incoming request is validated against the `Request` data class schema.
2. **Message Persistence**: The validated message is stored in the database to maintain conversation history.
3. **Response Generation**: The message content is sent to the Prompting Module to generate an appropriate response.
4. **Response Persistence**: The generated response is also stored in the database, linked to the original message.
5. **Prototype Handling**: If the response includes prototype data, it is extracted and stored separately.
6. **Response Formatting**: The final response is formatted as a `ServerResponse` object and sent back to the client.

```
private suspend fun handleJsonRequest(
    request: Request,
    call: ApplicationCall,
) {
    // Step 1: Log the request
    println("Handling JSON request: ${request.prompt} from ${request.userID} for conversation
        ${request.conversationId}")

    // Step 2: Save the user's message to the database
    saveMessage(request.conversationId, request.userID, request.prompt)

    // Step 3: Generate a response using the Prompting Module
    val response = getPromptingMain().run(request.prompt)

    // Step 4: Save the generated response to the database
    val savedMessage = saveMessage(request.conversationId, "LLM", response.chat.message)

    // Step 5: Store any prototype data if present
    response.prototype?.let { prototypeResponse ->
        val prototype = Prototype(
            messageId = savedMessage.id,
            filesJson = prototypeResponse.files.toString(),
            version = 1,
            isSelected = true
        )
        storePrototype(prototype)
    }

```

```
    // Step 6: Prepare and send the response
    val responseWithId = response.copy(
        chat = response.chat.copy(messageId = savedMessage.id)
    )

    val jsonString = Json.encodeToString(ServerResponse.serializer(), responseWithId)
    call.respondText(jsonString, contentType = ContentType.Application.Json)
}
```

Listing 60: Message Processing Flow

Message Storage Message storage is handled through the `saveMessage` function, which interfaces with the database module:

```
private suspend fun saveMessage(conversationId: String, senderId: String, content: String):
    ChatMessage {
    val message = ChatMessage(
        conversationId = conversationId,
        senderId = senderId,
        content = content
    )
    println("Saving message: $message")
    storeMessage(message)
    println("Stored message: ${message.id}")
    return message
}
```

Listing 61: Message Storage Function

This function:

- Creates a new `ChatMessage` object with appropriate metadata
- Invokes the storage layer's `storeMessage` function to persist the message
- Returns the stored message with its assigned ID

Response Generation The Chat Module delegates response generation to the Prompting Module through the `getPromptingMain().run()` method:

```
/**
 * This function serves as a getter for the singleton promptingMainInstance
 * to ensure consistent access throughout the application.
 *
 * @return The current PromptingMain instance
 */
private fun getPromptingMain(): PromptingMain = promptingMainInstance
```

Listing 62: Prompting Module Integration

The Prompting Module:

- Processes the input message
- Utilizes language models to generate appropriate responses
- Incorporates conversation context for coherent exchanges
- Optionally generates prototype data based on the input

Prototype Integration When a message response includes prototype data, it is extracted and stored using the `storePrototype` function:

```
response.prototype?.let { prototypeResponse ->
    val prototype = Prototype(
        messageId = savedMessage.id,
        filesJson = prototypeResponse.files.toString(),
        version = 1,
        isSelected = true
    )
    storePrototype(prototype)
}
```

Listing 63: Prototype Storage

This integration enables:

- Automatic prototype generation based on chat messages
- Persistent storage of prototype data linked to specific messages
- Version tracking for prototypes
- Selection status tracking for multi-prototype scenarios

Error Handling The message processing pipeline includes robust error handling at multiple levels:

```
val request: Request =
    runCatching {
        call.receive<Request>()
    }.getOrElse {
        return@post call.respondText(
            "Invalid request ${it.message}",
            status = HttpStatusCode.BadRequest,
        )
    }
```

Listing 64: Error Handling in Message Processing

Key error handling mechanisms include:

- Request validation with detailed error reporting
- Exception catching during response generation
- Database error handling during message storage
- Response formatting error prevention

Testing The message processing pipeline is thoroughly tested to ensure reliability:

```
@Test
fun `Test successful json route with valid request`() =
    testApplication {
        val mockPromptingMain = mock<PromptingMain>()
        runBlocking {
            whenever(mockPromptingMain.run(any())).thenReturn(
                ServerResponse(
                    chat =
                        ChatResponse(
                            message = "This is a test response",
                            timestamp = "2025-01-01T12:00:00",
                        ),
                ),
            )
        }

        try {
            setPromptingMain(mockPromptingMain)
            setupTestApplication()

            val response =
                client.post("/api/chat/json") {
                    header(HttpHeaders.Authorization, "Bearer ${createValidToken()}")
                    contentType(ContentType.Application.Json)
                    setBody(
                        """
                        {
                            "userID": "testUser",
                            "time": "2025-01-01T12:00:00",
                            "prompt": "Test prompt"
                        }
                        """.trimIndent(),
                    )
                }

            assertEquals(HttpStatusCode.OK, response.status)
            val responseBody = response.bodyAsText()
            val serverResponse = Json.decodeFromString<ServerResponse>(responseBody)
            assertEquals("This is a test response", serverResponse.chat.message)
        } finally {
            resetPromptingMain()
        }
    }
```

Listing 65: Message Processing Test

Tests cover various scenarios including:

- Successful message processing with valid inputs
- Error handling for invalid JSON
- Responses to empty or malformed prompts
- Integration between message processing and prototype generation
- Authentication validation for protected routes

Performance Considerations The message processing pipeline is designed with several performance optimizations:

- **Asynchronous Processing**: All database operations are performed as suspending functions to avoid blocking the main thread.
- **Efficient Serialization**: The Kotlinx Serialization library is used for efficient JSON serialization and deserialization.
- **Stateless Design**: The message processing pipeline is stateless, allowing for horizontal scaling of the application.
- **Minimal Logging**: Logging is strategic and focused on key events to minimize overhead while maintaining debuggability.

Extension Mechanisms The message processing pipeline is designed to be extensible in several ways:

- **Custom Message Types**: The `Request` and `Response` classes can be extended to support additional fields for new message types.
- **Processing Hooks**: The `handleJsonRequest` function can be modified to include additional processing steps or hooks.
- **Alternative Response Generators**: The Prompting Module instance can be replaced with alternative implementations through the `setPromptingMain` function, which is particularly useful for testing.

```
    /**
     * Sets a custom PromptingMain instance.
     *
     * This function is primarily used for testing purposes to inject a mock or
     * customized PromptingMain implementation.
     *
     * @param promptObject The PromptingMain instance to use for processing requests
     */
    internal fun setPromptingMain(promptObject: PromptingMain) {
        promptingMainInstance = promptObject
    }

    /**
     * Resets the PromptingMain instance to a new default instance.
     *
     * This function is used to restore the default behavior of the prompting
     * workflow, typically after testing or when a fresh state is required.
     */
    internal fun resetPromptingMain() {
        promptingMainInstance = PromptingMain()
    }
```

Listing 66: Prompting Module Configuration

The message processing component of the Chat Module serves as the core functionality for enabling interactive conversations within the application. Its robust design ensures reliability, performance, and extensibility while maintaining a clean integration with other system components.

7.2.6 File Uploads

The Chat Module includes comprehensive support for file uploads, enabling users to share files within chat conversations. This functionality is implemented through the upload route, which handles multipart form data and provides robust file processing capabilities.

Upload Route Configuration The file upload functionality is configured in the `uploadRoutes` function:

```
/**
 * Configures a POST route that handles multipart file upload requests.
 *
 * This route processes uploaded files along with optional description and message data.
 * It stores files in the specified upload directory and generates a response with
 * information about the uploaded content.
 *
 * @receiver The Route on which this endpoint will be registered
 * @param uploadDir Base directory path where uploaded files will be stored
 */
fun Route.uploadRoutes(uploadDir: String) {
    post(UPLOAD) {
        val uploadData = UploadData()
        val uploadDir = createUploadDirectory(uploadDir)

        val multipartData = call.receiveMultipart()
        multipartData.forEachPart { part ->
            handlePart(part, uploadDir, uploadData, call)
        }

        respondToUpload(call, uploadData)
    }
}
```

Listing 67: Upload Route Configuration

This function:

- Sets up a POST endpoint at the `UPLOAD` path
- Creates an `UploadData` object to track upload state
- Ensures the upload directory exists
- Processes each part of the multipart request
- Generates an appropriate response after processing

Upload Directory Management The upload directory is managed through the `createUploadDirectory` function:

```
/**
 * Creates the upload directory if it doesn't exist.
 *
 * @param dir Path to the directory where uploaded files should be stored
 * @return A File object representing the upload directory
 */
private fun createUploadDirectory(dir: String): File {
    val uploadDir = File(dir)
    if (!uploadDir.exists()) {
        uploadDir.mkdirs()
    }
    return uploadDir
}
```

Listing 68: Upload Directory Creation

This ensures that:

- The specified directory exists before attempting to write files
- Multiple levels of directories can be created if needed
- The directory can be configured for different environments

Multipart Processing The multipart form data is processed by handling each part according to its type:

```
/**
 * Processes each part of the multipart data according to its type.
 *
 * This function dispatches different part types to appropriate handlers and ensures
 * that resources are properly disposed after processing.
 *
 * @param part The multipart data part to be processed
 * @param uploadDir The directory where files will be saved
 * @param uploadData Container for tracking upload processing state
 * @param call The ApplicationCall for responding if needed
 */
```

```
private suspend fun handlePart(
    part: PartData,
    uploadDir: File,
    uploadData: UploadData,
    call: ApplicationCall,
) {
    when (part) {
        is PartData.FormItem -> handleFormItem(part, uploadData, call)
        is PartData.FileItem -> handleFileItem(part, uploadDir, uploadData)
        else -> { // Empty because we do nothing here!
        }
    }
    part.dispose()
}
```

Listing 69: Multipart Data Processing

This function:

- Categorizes parts as form items or file items
- Dispatches each part to an appropriate handler
- Ensures parts are properly disposed to prevent resource leaks

Form Item Processing Form items, which contain textual data, are processed by the `handleFormItem` function:

```
/**
 * Processes form items from the multipart request.
 *
 * Handles text data like descriptions and JSON messages.
 *
 * @param part The form item part to process
 * @param uploadData Container for tracking upload processing state
 * @param call The ApplicationCall for responding in case of errors
 */
private suspend fun handleFormItem(
    part: PartData.FormItem,
    uploadData: UploadData,
    call: ApplicationCall,
) {
    when (part.name) {
        "description" -> uploadData.fileDescription = part.value
        "message" -> handleMessagePart(part.value, uploadData, call)
    }
}
```

Listing 70: Form Item Processing

This function:

- Identifies form items by their name attribute
- Updates the `fileDescription` for description items
- Delegates message processing to a specialized handler

Message Part Processing When a message is included in the multipart request, it is processed specially:

```
/**
 * Processes a JSON message submitted with the upload.
 *
 * Attempts to parse the message string into a Request object and generate a
 * response. If parsing fails, responds with an error message.
 *
 * @param value The JSON string to parse
 * @param uploadData Container for tracking upload processing state
 * @param call The ApplicationCall for responding in case of errors
 */
private suspend fun handleMessagePart(
    value: String,
    uploadData: UploadData,
    call: ApplicationCall,
) {
    runCatching {
        uploadData.message = Json.decodeFromString(value)
        uploadData.response =
```

```
            uploadData.message?.let {
                Response(
                    time = LocalDateTime.now().toString(),
                    message = "${it.prompt}, ${it.userID}!",
                )
            }
    }.onFailure {
        call.respondText(
            text = "Invalid request: ${it.message}",
            status = HttpStatusCode.BadRequest,
        )
    }
}
```

Listing 71: Message Part Processing

This function:

- Attempts to parse the JSON string into a `Request` object
- Creates a `Response` based on the parsed message
- Handles parsing errors with an appropriate error response

File Item Processing File items, which contain binary data, are processed by the `handleFileItem` function:

```
/**
 * Processes an uploaded file item from the multipart request.
 *
 * Reads the file data, generates a timestamped filename to prevent conflicts,
 * and saves the file to the upload directory.
 *
 * @param part The file item part to process
 * @param uploadDir The directory where the file will be saved
 * @param uploadData Container for tracking upload processing state
 */
private suspend fun handleFileItem(
    part: PartData.FileItem,
    uploadDir: File,
    uploadData: UploadData,
) {
    uploadData.fileName = generateTimestampedFileName(part.originalFileName)
    val fileBytes = part.provider().readRemaining().readByteArray()
    File("$uploadDir/${uploadData.fileName}").writeBytes(fileBytes)
}
```

Listing 72: File Item Processing

This function:

- Generates a unique filename for the uploaded file
- Reads the file data into memory
- Writes the file to the specified upload directory

Filename Generation To prevent filename conflicts, uploaded files are given unique timestamped names:

```
/**
 * Generates a unique filename by appending a timestamp to the original file name.
 *
 * This prevents filename conflicts in the upload directory and preserves the original
 * file extension if present. If the original filename is null or blank, a generic
 * name with timestamp will be used.
 *
 * @param originalFileName The original name of the uploaded file, potentially null
 * @return A new timestamped filename that preserves the original extension
 */
internal fun generateTimestampedFileName(originalFileName: String?): String {
    val timestamp = System.currentTimeMillis()
    if (originalFileName.isNullOrBlank()) return "unknown_$timestamp"

    val lastDotIndex = originalFileName.lastIndexOf('.')
    return if (lastDotIndex != -1) {
        val name = originalFileName.substring(0, lastDotIndex)
        val extension = originalFileName.substring(lastDotIndex)
        "${name}_$timestamp$extension"
    } else {
```

```
        "${originalFileName}_$timestamp"
    }
}
```

Listing 73: Filename Generation

This function:

- Handles null or blank filenames with a default "unknown" prefix
- Preserves the original file extension
- Appends a timestamp to ensure uniqueness

Response Generation After processing the upload, an appropriate response is generated:

```
/**
 * Sends an appropriate response after processing the upload.
 *
 * If a structured response object is available, it will be sent as JSON.
 * Otherwise, a simple text response is sent containing the file description
 * and storage location.
 *
 * @param call The ApplicationCall used to send the response
 * @param uploadData Container with the upload state and response information
 */
private suspend fun respondToUpload(
    call: ApplicationCall,
    uploadData: UploadData,
) {
    uploadData.response?.let { response ->
        call.respond(response)
    } ?: call.respondText("${uploadData.fileDescription} is uploaded to 'uploads/${uploadData.
        fileName}'")
}
```

Listing 74: Upload Response Generation

This function:

- Sends a structured `Response` object if available
- Falls back to a text response with file details if no structured response is available

The file upload functionality provides a comprehensive solution for integrating file sharing into the chat experience. Its robust design handles various edge cases, prevents resource leaks, and ensures consistent responses for different upload scenarios.

7.2.7 Storage Management

The Chat Module implements comprehensive storage management for chat messages, conversations, and associated data. This section details how chat data is persisted, retrieved, and managed throughout the application.

Storage Factory The module uses a factory pattern to access chat repositories through the `ChatStorageFactory`:

```
object ChatStorageFactory {
    private val repository by lazy {
        DatabaseManager.externalInit()
        DatabaseManager.chatRepository()
    }

    fun getChatRepository(): ChatRepository = repository
}
```

Listing 75: Chat Storage Factory

This factory:

- Ensures lazy initialization of the chat repository
- Initializes the database connection if needed
- Provides a single access point for chat storage operations

Message Storage Messages are stored using the `storeMessage` function:

```
suspend fun storeMessage(message: ChatMessage): Boolean {
    println("Storing message: ${message.content} from ${message.senderId} in ${message.
        conversationId}")
    return runCatching {
        println("Storing message: ${message.content} from ${message.senderId} in ${message.
            conversationId}")
        ChatStorageFactory.getChatRepository().saveMessage(message)
    }.getOrElse { e ->
        println("Error storing message: ${e.message}")
        false
    }
}
```

Listing 76: Message Storage Function

This function:

- Takes a `ChatMessage` object as input
- Delegates to the chat repository for the actual storage operation
- Catches and logs any errors that occur during storage
- Returns a boolean indicating success or failure

Message Retrieval Messages are retrieved using the `getMessageHistory` function:

```
suspend fun getMessageHistory(conversationId: String, limit: Int = 50, offset: Int = 0): List<
    ChatMessage> {
    return runCatching {
        ChatStorageFactory.getChatRepository().getMessagesByConversation(conversationId, limit
            , offset)
    }.getOrElse { e ->
        println("Error retrieving message history: ${e.message}")
        emptyList()
    }
}
```

Listing 77: Message Retrieval Function

This function:

- Takes a conversation ID and optional pagination parameters
- Delegates to the chat repository for message retrieval
- Handles errors by returning an empty list
- Returns the retrieved messages for the specified conversation

Conversation Management Conversations are retrieved using the `getConversationHistory` function:

```
suspend fun getConversationHistory(userId: String): List<Conversation> {
    return runCatching {
        ChatStorageFactory.getChatRepository().getConversationsByUser(userId)
    }.getOrElse { e ->
        println("Error retrieving conversation history: ${e.message}")
        emptyList()
    }
}
```

Listing 78: Conversation Retrieval Function

This function:

- Takes a user ID to filter conversations
- Delegates to the chat repository for conversation retrieval
- Handles errors by returning an empty list
- Returns the retrieved conversations for the specified user

Conversation metadata can be updated using the `updateConversationName` function:

```
suspend fun updateConversationName(conversationId: String, name: String): Boolean {
    return runCatching {
        ChatStorageFactory.getChatRepository().updateConversationName(conversationId, name)
    }.getOrElse { e ->
        println("Error updating conversation name: ${e.message}")
```

```
        false
    }
}
```

Listing 79: Conversation Update Function

This function:

- Takes a conversation ID and a new name
- Delegates to the chat repository for the update operation
- Handles errors by returning false
- Returns a boolean indicating success or failure

Prototype Storage Prototypes generated during chat interactions are stored using the `storePrototype` function:

```
suspend fun storePrototype(prototype: Prototype): Boolean {
    return runCatching {
        ChatStorageFactory.getChatRepository().savePrototype(prototype)
        true
    }.getOrElse { e ->
        println("Error storing prototype: ${e.message}")
        false
    }
}
```

Listing 80: Prototype Storage Function

This function:

- Takes a `Prototype` object as input
- Delegates to the chat repository for the storage operation
- Handles errors by returning false
- Returns a boolean indicating success or failure

Prototypes can be retrieved using the `retrievePrototype` function:

```
suspend fun retrievePrototype(conversationId: String, messageId: String): Prototype? {
    return runCatching {
        ChatStorageFactory.getChatRepository().getSelectedPrototypeForMessage(conversationId,
            messageId)
    }.getOrElse { e ->
        println("Error retrieving prototype: ${e.message}")
        null
    }
}
```

Listing 81: Prototype Retrieval Function

This function:

- Takes a conversation ID and message ID to identify the prototype
- Delegates to the chat repository for prototype retrieval
- Handles errors by returning null
- Returns the retrieved prototype if available

Data Modeling The chat storage implementation relies on several data models:

```
data class ChatMessage(
    val id: String = UUID.randomUUID().toString(),
    val conversationId: String,
    val senderId: String,
    val content: String,
    val timestamp: Instant = Instant.now()
)

data class Conversation(
    val id: String,
    val name: String,
    val lastModified: String,
    val messageCount: Int,
    val userId: String
)
```

```
data class Prototype(
    val messageId: String,
    val filesJson: String,
    val version: Int,
    val isSelected: Boolean
)
```

Listing 82: Chat Data Models

These models:

- Represent the core data structures for chat functionality
- Define the relationships between messages, conversations, and prototypes
- Include appropriate metadata for tracking and organization

Error Handling The storage management functions implement consistent error handling:

```
return runCatching {
    // Storage operation
}.getOrElse { e ->
    println("Error message: ${e.message}")
    fallbackValue // e.g., false, emptyList(), null
}
```

Listing 83: Error Handling Pattern

This pattern:

- Wraps storage operations in `runCatching` for structured error handling
- Logs errors for debugging purposes
- Returns appropriate fallback values to prevent application crashes
- Isolates database errors from the rest of the application

The storage management component of the Chat Module provides a robust foundation for persisting chat data throughout the application. Its consistent error handling and clean abstractions ensure reliable data operations while isolating the rest of the application from database complexities.

7.2.8 Security

The Chat Module implements comprehensive security measures to protect chat data and prevent unauthorized access or malicious use. This section details the security features and best practices employed throughout the module.

Authentication Integration All chat endpoints are protected by JWT authentication middleware, ensuring that only authenticated users can access chat functionality:

```
fun Application.chatModule() {
    routing {
        authenticate("jwt-verifier") {
            chatRoutes()
            jsonRoutes()
            uploadRoutes(ChatEndpoint.UPLOAD_DIR)
        }
    }
}
```

Listing 84: Authentication Integration

This integration:

- Ensures all chat routes are protected by the JWT authentication middleware
- Leverages the Authentication Module's token validation capabilities
- Prevents unauthorized access to chat data and functionality
- Maintains user context throughout chat operations

Input Sanitization The module implements robust input sanitization to prevent injection attacks and other security vulnerabilities:

```
val request: Request =
    runCatching {
        call.receive<Request>()
    }.getOrElse {
        return@post call.respondText(
            "Invalid request ${it.message}",
            status = HttpStatusCode.BadRequest,
        )
    }
```

Listing 85: Input Validation Example

Key input validation measures include:

- Strict validation of JSON request bodies against defined data models
- Sanitization of message content before processing
- Validation of path and query parameters
- Careful handling of file uploads to prevent path traversal attacks

Secure File Handling The file upload functionality includes several security measures:

```
uploadData.fileName = generateTimestampedFileName(part.originalFileName)
val fileBytes = part.provider().readRemaining().readByteArray()
File("$uploadDir/${uploadData.fileName}").writeBytes(fileBytes)
```

Listing 86: Secure File Handling

File security measures include:

- Sanitization of file names to prevent path traversal attacks
- Generation of unique file names to prevent overwriting
- Isolation of file storage in a dedicated upload directory
- Careful error handling to prevent information leakage

Conversation Access Control The module implements user-based access control for conversations:

```
fun getConversationHistory(userId: String): List<Conversation> {
    return runCatching {
        ChatStorageFactory.getChatRepository().getConversationsByUser(userId)
    }.getOrElse { e ->
        println("Error retrieving conversation history: ${e.message}")
        emptyList()
    }
}
```

Listing 87: Conversation Access Control

This ensures that:

- Users can only access their own conversations
- User IDs are validated through authentication
- Conversation data is properly isolated between users

Error Handling and Logging The module implements secure error handling and logging practices:

```
catch (e: Exception) {
    println("Error storing message: ${e.message}")
    return@post call.respondText(
        "Error: ${e.message}",
        status = HttpStatusCode.InternalServerError,
    )
}
```

Listing 88: Secure Error Handling

Security considerations in error handling include:

- Avoiding exposure of sensitive information in error messages
- Logging appropriate details for debugging without revealing security-sensitive data
- Providing generic error messages to clients
- Handling exceptions at appropriate levels to prevent information leakage

Content Security The module includes measures to protect against malicious content:

- **Content-Type Validation**: File uploads are validated against expected content types
- **Size Limitations**: Appropriate size limits are applied to prevent denial-of-service attacks
- **Content Isolation**: Uploaded files are stored in isolated locations to prevent execution

Transport Security While the module itself doesn't implement transport security, it is designed to work with secure transport:

- The application should be deployed behind HTTPS
- Cookies used for authentication should have the secure flag set
- Sensitive data should only be transmitted over encrypted connections

Session Management The module integrates with the Authentication Module's session management:

```
val sessionCookie =
    call.request.cookies["AuthenticatedSession"]?.let { cookie ->
        kotlin.runCatching { Json.decodeFromString<AuthenticatedSession>(cookie) }.getOrNull()
    }
```

Listing 89: Session Handling Example

Session security measures include:

- Secure cookie handling for session data
- Proper validation of session tokens
- Session expiration and renewal
- Protection against session hijacking

Testing and Validation The module includes security-focused tests:

```
@Test
fun `Test unauthorized access`() =
    testApplication {
        setupTestApplication()
        val response = client.get(GET)
        assertEquals(HttpStatusCode.Unauthorized, response.status)
    }
```

Listing 90: Security Test Example

Security testing includes:

- Verification of authentication requirements
- Testing of input validation mechanisms
- Verification of access control logic
- Testing of error handling for security implications

These comprehensive security measures ensure that the Chat Module provides a secure environment for chat interactions, protecting user data while maintaining a smooth user experience.

7.2.9 Configuration

The Chat Module provides several configuration options that can be customized to adapt to different deployment environments and requirements. This section details the available configuration settings and how to adjust them.

Upload Directory Configuration The base directory for file uploads can be configured through the `ChatEndpoint` object:

```
object ChatEndpoint {
    internal var UPLOAD_DIR: String = "uploads" // Do not use val for testing!
}
```

Listing 91: Upload Directory Configuration

This configuration:

- Defines the default upload directory as "uploads"
- Can be modified programmatically for different environments

- Is particularly useful for testing scenarios where a temporary directory might be preferred

To customize the upload directory in a production environment, set it during application startup:

```
fun Application.configureChat() {
    // Set custom upload directory based on environment
    val configuredUploadDir = environment.config.propertyOrNull("chat.uploadDir")?.getString()
    if (!configuredUploadDir.isNullOrBlank()) {
        ChatEndpoint.UPLOAD_DIR = configuredUploadDir
    }

    // Ensure the directory exists
    val uploadDir = File(ChatEndpoint.UPLOAD_DIR)
    if (!uploadDir.exists()) {
        uploadDir.mkdirs()
    }

    // Install the chat module
    chatModule()
}
```

Listing 92: Customizing Upload Directory

Route Configuration The base routes for chat functionality are defined as constants:

```
const val CHAT = "/api/chat"
const val GET = "$CHAT/history"
const val JSON = "$CHAT/json"
const val UPLOAD = "$CHAT/upload"
```

Listing 93: Route Configuration

While these constants are not directly configurable at runtime, they can be modified in the source code if different route paths are required. Any changes should be coordinated with the client application to ensure proper communication.

Module Installation The Chat Module is installed in the application through the `chatModule` function:

```
fun Application.chatModule() {
    routing {
        authenticate("jwt-verifier") {
            chatRoutes()
            jsonRoutes()
            uploadRoutes(ChatEndpoint.UPLOAD_DIR)
        }
    }
}
```

Listing 94: Module Installation

This function:

- Sets up all chat-related routes within an authenticated scope
- Passes the configured upload directory to the upload routes
- Integrates with the application's routing system

To customize module installation, the function can be modified or wrapped:

```
fun Application.customizableChatModule(
    authenticationName: String = "jwt-verifier",
    uploadDirectory: String = ChatEndpoint.UPLOAD_DIR
) {
    routing {
        authenticate(authenticationName) {
            chatRoutes()
            jsonRoutes()
            uploadRoutes(uploadDirectory)
        }
    }
}
```

Listing 95: Custom Module Installation

Prompting Module Configuration The Chat Module integrates with the Prompting Module, which can be configured through the `setPromptingMain` and `resetPromptingMain` functions:

```
/**
 * Sets a custom PromptingMain instance.
 *
 * This function is primarily used for testing purposes to inject a mock or
 * customized PromptingMain implementation.
 *
 * @param promptObject The PromptingMain instance to use for processing requests
 */
internal fun setPromptingMain(promptObject: PromptingMain) {
    promptingMainInstance = promptObject
}

/**
 * Resets the PromptingMain instance to a new default instance.
 *
 * This function is used to restore the default behavior of the prompting
 * workflow, typically after testing or when a fresh state is required.
 */
internal fun resetPromptingMain() {
    promptingMainInstance = PromptingMain()
}
```

Listing 96: Prompting Module Configuration

These functions:

- Allow for custom Prompting Module implementations to be injected
- Provide a way to reset to the default implementation
- Are particularly useful for testing scenarios where mock implementations are needed

Database Integration Configuration The Chat Module integrates with the Database Module through the `ChatStorageFactory`:

```
object ChatStorageFactory {
    private val repository by lazy {
        DatabaseManager.externalInit()
        DatabaseManager.chatRepository()
    }

    fun getChatRepository(): ChatRepository = repository
}
```

Listing 97: Database Integration Configuration

This factory:

- Initializes the database connection when first needed
- Provides access to the chat repository
- Uses lazy initialization to defer database setup until actually required

Database-specific configuration (connection strings, pool sizes, etc.) is handled by the Database Module itself and is not directly configurable through the Chat Module.

Message Size Limitations Message size limitations can be implemented in the Chat Module:

```
private const val MAX_MESSAGE_SIZE = 4000 // Maximum characters per message

private suspend fun handleJsonRequest(
    request: Request,
    call: ApplicationCall,
) {
    // Validate message size
    if (request.prompt.length > MAX_MESSAGE_SIZE) {
        return@post call.respondText(
            "Message exceeds maximum length of $MAX_MESSAGE_SIZE characters",
            status = HttpStatusCode.BadRequest,
        )
    }

    // Continue with normal processing
    // ...
```

```
}
```

Listing 98: Message Size Configuration

This configuration:

- Defines a maximum character count for messages
- Validates incoming messages against this limit
- Returns an appropriate error response for oversized messages

Error Response Configuration Error responses can be customized for different environments:

```
private suspend fun handleError(
    call: ApplicationCall,
    exception: Exception,
    isDevelopment: Boolean = false
) {
    val message = if (isDevelopment) {
        "Error: ${exception.message}\n${exception.stackTraceToString()}"
    } else {
        "An error occurred while processing your request"
    }

    call.respondText(
        message,
        status = HttpStatusCode.InternalServerError,
    )
}
```

Listing 99: Error Response Configuration

This approach:

- Provides detailed error information in development environments
- Returns generic error messages in production environments
- Prevents exposure of sensitive information to end users

The Chat Module's configuration options provide flexibility for adapting to different environments and requirements while maintaining a consistent API for chat functionality. By adjusting these settings, developers can optimize the module for specific deployment scenarios and integration with other system components.

7.2.10 Best Practices and Usage Guidelines

This section provides best practices and usage guidelines for developers working with the Chat Module. Following these recommendations will ensure efficient, secure, and maintainable chat functionality.

Message Processing When working with chat messages, follow these best practices:

- **Validate message format**: Always validate incoming messages against the expected data model:

```
    val request: Request =
        runCatching {
            call.receive<Request>()
        }.getOrElse {
            return@post call.respondText(
                "Invalid request ${it.message}",
                status = HttpStatusCode.BadRequest,
            )
        }
```

Listing 100: Message Validation Example

- **Sanitize message content**: Sanitize user-provided content to prevent XSS and other injection attacks:

```
    val sanitizedContent = Jsoup.clean(request.prompt, Safelist.basic())
    val safeRequest = request.copy(prompt = sanitizedContent)
```

Listing 101: Content Sanitization Example

- **Maintain conversation context**: Always include the conversation ID to ensure messages are properly associated with conversations:

```
    val conversationId = request.conversationId.takeIf { it.isNotBlank() } ?: UUID.
        randomUUID().toString()
    saveMessage(conversationId, request.userID, request.prompt)
```

Listing 102: Conversation Context Example

- **Handle large messages efficiently**: Process large messages in chunks to avoid memory issues:

```
    if (request.prompt.length > MAX_MESSAGE_SIZE) {
        val chunks = request.prompt.chunked(MAX_MESSAGE_SIZE)
        chunks.forEach { chunk ->
            saveMessage(request.conversationId, request.userID, chunk)
        }
    } else {
        saveMessage(request.conversationId, request.userID, request.prompt)
    }
```

Listing 103: Large Message Handling

File Upload Handling When working with file uploads, adhere to these guidelines:

- **Verify file types**: Validate uploaded files against allowed MIME types:

```
    val allowedTypes = listOf("text/plain", "application/pdf", "image/jpeg", "image/png")
    val contentType = part.contentType ?: ContentType.Any
    if (contentType.toString() !in allowedTypes) {
        call.respondText(
            "Unsupported file type: ${contentType}",
            status = HttpStatusCode.BadRequest
        )
        return@forEachPart
    }
```

Listing 104: File Type Validation

- **Limit file sizes**: Enforce reasonable file size limits to prevent resource exhaustion:

```
    val MAX_FILE_SIZE = 10 * 1024 * 1024 // 10 MB
    val channel = part.provider()
    if (channel.availableForRead > MAX_FILE_SIZE) {
        call.respondText(
            "File exceeds maximum size of 10 MB",
            status = HttpStatusCode.PayloadTooLarge
        )
        return@forEachPart
    }
```

Listing 105: File Size Limiting

- **Use streaming for large files**: Process large files as streams rather than loading them entirely into memory:

```
    val fileBytes = ByteArray(8192)
    val fileOutputStream = FileOutputStream(File("$uploadDir/${uploadData.fileName}"))

    try {
        val channel = part.provider()
        while (true) {
            val bytesRead = channel.readAvailable(fileBytes, 0, fileBytes.size)
            if (bytesRead < 0) break
            fileOutputStream.write(fileBytes, 0, bytesRead)
        }
    } finally {
        fileOutputStream.close()
    }
```

Listing 106: File Streaming Example

- **Store files outside the web root**: Store uploaded files in a location that cannot be directly accessed by web requests:

```
    val secureUploadDir = File("/var/data/uploads")
    if (!secureUploadDir.exists()) {
        secureUploadDir.mkdirs()
    }
```

Listing 107: Secure File Storage

Error Handling and Logging Implement consistent error handling and logging practices:

- **Use structured error handling**: Employ Kotlin's `runCatching` for cleaner error handling:

```
return runCatching {
    ChatStorageFactory.getChatRepository().saveMessage(message)
}.onFailure { e ->
    logger.error("Failed to save message: ${e.message}")
}.getOrDefault(false)
```

Listing 108: Structured Error Handling

- **Provide informative error messages**: Return helpful error messages without exposing sensitive information:

```
call.respondText(
    "Unable to process request. Please try again later.",
    status = HttpStatusCode.InternalServerError
)
```

Listing 109: Error Message Example

- **Log at appropriate levels**: Use appropriate log levels for different types of events:

```
logger.debug("Processing upload for user ${request.userID}")
logger.info("File ${filename} uploaded successfully")
logger.warn("Invalid file format attempted: ${contentType}")
logger.error("Failed to save message: ${e.message}", e)
```

Listing 110: Logging Best Practices

- **Include request context in logs**: Add relevant context to log messages for easier debugging:

```
logger.info("Chat message [conversationId=${request.conversationId}, userId=${request.
    userID}] processed successfully")
```

Listing 111: Contextual Logging

Performance Optimization Optimize the Chat Module for better performance:

- **Use pagination for history retrieval**: Always implement pagination for message history to avoid loading too many messages at once:

```
val limit = call.request.queryParameters["limit"]?.toIntOrNull() ?: 50
val offset = call.request.queryParameters["offset"]?.toIntOrNull() ?: 0
val messages = getMessageHistory(conversationId, limit, offset)
```

Listing 112: Pagination Implementation

- **Implement response caching**: Cache frequently accessed data like conversation lists:

```
val cacheKey = "conversations:${userId}"
val cachedData = cacheManager.get(cacheKey)

if (cachedData != null) {
    return Json.decodeFromString<List<Conversation>>(cachedData)
}

val conversations = chatRepository.getConversationsByUser(userId)
cacheManager.set(cacheKey, Json.encodeToString(conversations), expirationSeconds = 300)
return conversations
```

Listing 113: Response Caching

- **Use asynchronous processing**: Leverage Kotlin's coroutines for non-blocking operations:

```
coroutineScope {
    launch {
        saveMessage(conversationId, userId, content)
    }

    launch {
        updateConversationLastModified(conversationId)
    }
}
```

Listing 114: Asynchronous Processing

- **Optimize database queries**: Ensure database operations are efficient and properly indexed:

```
// Inefficient query
val messages = getAllMessages().filter { it.conversationId == conversationId }

// Optimized query
val messages = getMessagesByConversation(conversationId)
```

Listing 115: Query Optimization

Security Considerations Maintain strong security practices:

- **Validate user permissions**: Ensure users can only access their own conversations:

```
val userId = call.principal<JWTPrincipal>()?.payload?.getClaim("sub")?.asString()
if (userId != conversation.userId) {
    call.respond(HttpStatusCode.Forbidden, "You don't have permission to access this
        conversation")
    return@get
}
```

Listing 116: Permission Validation

- **Use parameterized queries**: Always use parameterized queries to prevent SQL injection:

```
chatRepository.getMessagesByConversation(conversationId, limit, offset)

// Instead of something like:
connection.execute("SELECT * FROM messages WHERE conversation_id = '$conversationId'")
```

Listing 117: Parameterized Query Example

- **Implement rate limiting**: Prevent abuse by implementing rate limits on chat endpoints:

```
install(RateLimiting) {
    rateLimiter = RedisRateLimiter(
        Redis.getRedisConnection(),
        RateLimitConf(
            keyPrefix = "ratelimit:",
            limit = 100,
            refillRate = 10.0,
            refillTimeUnit = TimeUnit.MINUTES
        )
    )
}
```

Listing 118: Rate Limiting Implementation

- **Scan uploaded files**: Implement virus scanning for uploaded files:

```
val scanResult = virusScanner.scan(fileBytes)
if (!scanResult.isClean) {
    call.respondText(
        "File failed security scan",
        status = HttpStatusCode.BadRequest
    )
    return@forEachPart
}
```

Listing 119: File Scanning

Testing Implement thorough testing strategies:

- **Write unit tests**: Test individual components in isolation:

```
@Test
fun `Test generateTimestampedFileName with null input`() {
    val result = generateTimestampedFileName(null)
    assertTrue(result.startsWith("unknown_"))
    assertTrue(result.substring(8).toLongOrNull() != null)
}
```

Listing 120: Unit Test Example

- **Implement integration tests**: Test the interaction between multiple components:

```kotlin
@Test
fun `Test file upload and retrieval`() = testApplication {
    // Setup test environment
    val testDir = File("test_uploads")
    testDir.mkdirs()

    // Perform file upload
    val response = client.post(UPLOAD) {
        // Setup multipart request
    }

    // Verify response
    assertEquals(HttpStatusCode.OK, response.status)

    // Verify file was saved correctly
    val uploadedFile = testDir.listFiles()?.firstOrNull { it.name.startsWith("test_") }
    assertNotNull(uploadedFile)
}
```

Listing 121: Integration Test Example

- **Mock external dependencies**: Use mocking for testing components that depend on external services:

```kotlin
@Test
fun `Test message processing with mocked prompting`() {
    // Create mock
    val mockPromptingMain = mock<PromptingMain>()
    whenever(mockPromptingMain.run(any())).thenReturn(
        ServerResponse(chat = ChatResponse(message = "Mock response", timestamp = "
            2025-01-01T12:00:00"))
    )

    // Use mock in test
    setPromptingMain(mockPromptingMain)

    // Test logic
    // ...
}
```

Listing 122: Dependency Mocking

- **Test error scenarios**: Ensure error handling works as expected:

```kotlin
@Test
fun `Test invalid JSON handling`() = testApplication {
    setupTestApplication()

    val response = client.post("/api/chat/json") {
        header(HttpHeaders.Authorization, "Bearer ${createValidToken()}")
        contentType(ContentType.Application.Json)
        setBody("This is not valid JSON")
    }

    assertEquals(HttpStatusCode.BadRequest, response.status)
    assertTrue(response.bodyAsText().contains("Invalid request"))
}
```

Listing 123: Error Handling Test

Following these best practices will ensure that implementations of the Chat Module are secure, performant, and maintainable. These guidelines promote clean code organization, proper error handling, and robust security, while providing practical examples for common development scenarios.

7.3 Database Module

7.3.1 Overview

The Database Module provides a robust foundation for database interactions within the application. It implements a clean architecture pattern that separates concerns between database connection management, entity representation, and data access operations. This module is built using Kotlin with the Exposed framework for object-relational mapping, HikariCP for connection pooling, and Flyway for database migrations.

The module serves as the data persistence layer for the application, handling:

- Database connection establishment and management
- Database schema creation and migration
- Entity mapping between database and application models
- Transaction management and error handling
- Repository interfaces for CRUD operations

The Database Module is designed to be modular and extensible, allowing for easy addition of new entities and repositories as the application evolves. Its architecture promotes clean separation of concerns, making it easier to maintain and test.

7.3.2 Architecture

Module Structure The database module is organized into the following key components:

- **Core**: Contains the foundational components responsible for database connectivity, including `DatabaseManager` and `PoCDatabase`.
- **Tables**: Defines data entities and their corresponding database schemas.
- **Repositories**: Implements data access patterns that encapsulate database operations for specific entity types.

Architectural Layers The module follows a layered architecture that separates concerns:

- **Connection Layer**: Manages database connections and pooling via HikariCP.
- **Schema Layer**: Defines table structures and relationships using Exposed's DSL.
- **Entity Layer**: Maps database rows to Kotlin objects using Exposed's DAO pattern.
- **Repository Layer**: Provides high-level data access operations with error handling.

This layered approach ensures that each component has a single responsibility, making the code more maintainable and testable. The separation also allows for changes in one layer without affecting others, such as modifying a database schema without changing the repository interface.

Architecture Diagram

The database module architecture follows a vertical organization, with components arranged in increasing levels of abstraction:

- **Bottom Layer**: Connection management (DatabaseManager)
- **Middle Layer**: Schema definitions and entity mappings
- **Top Layer**: Repository interfaces for application use

Each entity type (e.g., Prototype) has its own vertical slice through these layers, with its own table definition, entity class, and repository implementation.

Core Components The Database Module consists of several key components that work together to provide a robust database interaction layer for the application.

DatabaseManager The `DatabaseManager` object is responsible for setting up and initializing the database connection. It performs several key functions:

- Loads database configuration from environment variables
- Configures the connection pool via HikariCP
- Sets up transaction isolation levels
- Executes database migrations using Flyway

Key implementation details:

```kotlin
object DatabaseManager {
    private var database: Database? = null
    private var templateRepository: TemplateRepository? = null
    private var dataSource: HikariDataSource? = null
    private var chatRepository: ChatRepository? = null

    /**
     * Resets the database manager state by closing connections and clearing references.
```

```
     */
    fun reset() {
        dataSource?.close()
        database = null
        dataSource = null
        templateRepository = null
        chatRepository = null
    }

    /**
     * Initializes the database connection and runs all necessary migrations.
     */
    internal fun init(): Database {
        val credentials = getDatabaseCredentials()

        return try {
            configureFlyway(credentials)
            val newDatabase = setupDatabase(credentials)
            database = newDatabase
            newDatabase
        } catch (e: Exception) {
            throw e
        }
    }

    private fun setupDatabase(credentials: DatabaseCredentials): Database {
        val config = HikariConfig().apply {
            jdbcUrl = credentials.url
            username = credentials.username
            password = credentials.password
            maximumPoolSize = max(1, credentials.maxPoolSize)
            isAutoCommit = false
            transactionIsolation = "TRANSACTION_REPEATABLE_READ"
            driverClassName = "org.postgresql.Driver"
        }
        dataSource = HikariDataSource(config)
        return Database.connect(dataSource!!)
    }

    // Other implementation methods...
}
```

Listing 124: DatabaseManager Implementation

The `DatabaseManager` leverages HikariCP for efficient connection pooling, which is crucial for handling concurrent database access. The connection pool is configured with appropriate settings like maximum pool size and transaction isolation level to ensure reliable database operations.

DatabaseCredentials The `DatabaseCredentials` data class encapsulates the information needed to connect to the database:

```
internal data class DatabaseCredentials(
    val url: String,
    val username: String,
    val password: String,
    val maxPoolSize: Int = 10
)
```

Listing 125: DatabaseCredentials Data Class

This class provides a clean interface for passing connection parameters to the database setup functions. It includes a default value for `maxPoolSize` to simplify configuration when only basic settings are needed.

PoCDatabase The `PoCDatabase` object provides a singleton pattern for accessing the database connection. It uses lazy initialization to ensure the database is only set up once when first accessed:

```
internal object PoCDatabase {
    internal val database: Database by lazy { DatabaseManager.init() }

    fun init() {
        database
    }
}
```

Listing 126: PoCDatabase Implementation

This pattern ensures that:

- The database connection is initialized only when needed
- The same database connection instance is reused throughout the application
- Connection management is centralized and consistent

The singleton design pattern is essential here as it prevents the creation of multiple database connections, which would waste resources and potentially lead to connection leaks. All application components should access the database through this single point of entry.

Repository Interfaces The `DatabaseManager` provides accessor methods for repository interfaces:

```
fun templateRepository(): TemplateRepository {
    val db = checkNotNull(database) { "Database connection not initialized. Call init() first.
        " }
    return templateRepository ?: TemplateRepository(db).also { templateRepository = it }
}

fun chatRepository(): ChatRepository {
    val db = checkNotNull(database) { "Database connection not initialized. Call init() first.
        " }
    return chatRepository ?: ChatRepository(db).also { chatRepository = it }
}
```

Listing 127: Repository Accessor Methods

These methods follow the lazy initialization pattern for repositories:

- They verify that the database is initialized before creating a repository
- They cache repository instances to avoid creating multiple instances
- They inject the database connection into the repository, enabling better testability

This approach creates a clean separation of concerns while maintaining efficient resource management.

Entity Framework The Database Module uses Exposed, a SQL framework for Kotlin, to define database schemas and map database rows to objects. This framework provides a type-safe and concise way to work with database entities while maintaining the flexibility of SQL.

Table Definitions The module uses Exposed's DSL to define database tables. Below is an example from the `Prototypes` table definition:

```
internal object Prototypes : UUIDTable("prototypes") {
    val userId = text("userId")
    val userPrompt = text("prompt")
    val fullPrompt = text("fullPrompt")
    val path = varchar("path", 255).nullable()
    val createdAt = timestamp("created_at")
    val projectName = text("name")
}
```

Listing 128: Prototype Table Definition

Key aspects of the table definition:

- It extends `UUIDTable`, indicating it uses UUID as primary key
- The constructor parameter specifies the table name in the database
- Column definitions include type information via Exposed's DSL
- Special column types like nullable fields and timestamps are supported

The module includes several table definitions for different entity types:

```
object ChatMessageTable : UUIDTable("chat_messages") {
    val conversationId = reference("conversation_id", ConversationTable, onDelete =
        ReferenceOption.CASCADE)
    val isFromLLM = bool("is_from_llm")
    val content = text("content")
    val timestamp = timestamp("timestamp")
}
```

Listing 129: Chat Message Table Definition

These definitions establish the database schema and relationships between entities, such as foreign key references and cascade delete operations.

Entity Classes The module implements the Data Access Object (DAO) pattern using Exposed's `Entity` and `EntityClass` abstractions. These provide an object-oriented way to interact with database rows:

```
class PrototypeEntity(
    id: EntityID<UUID>,
) : UUIDEntity(id) {
    companion object : UUIDEntityClass<PrototypeEntity>(Prototypes)

    var userId by Prototypes.userId
    var userPrompt by Prototypes.userPrompt
    var fullPrompt by Prototypes.fullPrompt
    var s3Key by Prototypes.path
    var createdAt by Prototypes.createdAt
    var projectName by Prototypes.projectName

    /**
     * Converts the entity to a Prototype object
     */
    fun toPrototype() =
        Prototype(
            id = this.id.value,
            userId = this.userId,
            userPrompt = this.userPrompt,
            fullPrompt = this.fullPrompt,
            s3key = this.s3Key,
            createdAt = this.createdAt,
            projectName = this.projectName,
        )
}
```

Listing 130: Prototype Entity Class

The entity class provides:

- Property delegation to map object fields to table columns
- A companion object for entity creation and querying
- A conversion method to transform database entities to domain objects

Exposed's entity system allows for intuitive database operations:

```
// Creating a new entity
val prototype = PrototypeEntity.new(UUID.randomUUID()) {
    userId = "user123"
    userPrompt = "Generate a weather app"
    fullPrompt = "Create a React weather application with the following features..."
    s3Key = "prototypes/weather-app.zip"
    createdAt = Instant.now()
    projectName = "Weather App"
}

// Finding an entity by ID
val existingPrototype = PrototypeEntity.findById(id)

// Updating an entity
existingPrototype?.apply {
    projectName = "Advanced Weather App"
}

// Deleting an entity
existingPrototype?.delete()
```

Listing 131: Entity Operations Example

Data Transfer Objects The module separates database entities from domain models using Data Transfer Objects (DTOs). This provides a clean boundary between database representation and application logic:

```
data class Prototype(
    val id: UUID,
    var userId: String,
    var userPrompt: String,
    var fullPrompt: String,
    val s3key: String?,
    val createdAt: Instant,
    val projectName: String,
)
```

Listing 132: Prototype Data Class

Similarly, the module defines DTOs for chat-related entities:

```
data class ChatMessage(
    val id: String = UUID.randomUUID().toString(),
    val conversationId: String,
    val senderId: String, // Will be converted to isFromLLM in the database
    val content: String,
    val timestamp: Instant = Instant.now()
)
```

Listing 133: ChatMessage Data Class

This separation of concerns ensures that:

- Changes to the database schema don't directly impact application code
- Domain models can be shaped to fit application needs without database constraints
- Database-specific details remain isolated in the database layer

The DTO pattern is a crucial aspect of maintaining a clean architecture, allowing the database implementation details to be abstracted away from the rest of the application.

Repository Pattern The module implements the Repository pattern to encapsulate data access logic. Each entity type has a corresponding repository class responsible for performing CRUD operations.

Core Principles The Repository pattern applied in this module follows these principles:

- Provides a collection-like interface for accessing domain objects
- Abstracts the underlying data source implementation details
- Centralizes data access logic for specific entity types
- Handles error conditions and wraps operations in Result types

Prototype Repository The `PrototypeRepository` class demonstrates how the module handles data access:

```
class PrototypeRepository(
    private val db: Database,
) {
    /**
     * Function to save a Prototype to the database
     * @param prototype The Prototype to save
     * @return A Result object containing the success or failure of the operation
     */
    suspend fun createPrototype(prototype: Prototype): Result<Unit> =
        runCatching {
            newSuspendedTransaction(Dispatchers.IO, db) {
                PrototypeEntity.new(prototype.id) {
                    userId = prototype.userId
                    userPrompt = prototype.userPrompt
                    fullPrompt = prototype.fullPrompt
                    s3Key = prototype.s3key
                    createdAt = prototype.createdAt
                    projectName = prototype.projectName
                }
            }
        }

    /**
     * Function to retrieve a Prototype from the database by its ID
     * @param id The UUID of the Prototype to retrieve
     * @return A Result object containing the Prototype if it exists, or null if it does not
     */
    suspend fun getPrototype(id: UUID): Result<Prototype?> =
        runCatching {
            newSuspendedTransaction(Dispatchers.IO, db) {
                PrototypeEntity.findById(id)?.toPrototype()
            }
        }

    /**
     * Function to retrieve a list of Prototypes from the database by the user ID
```

```
     * @param userId The ID of the user
     * @param page The page number of the results (for pagination)
     * @param pageSize The number of Prototypes to retrieve per page
     * @return A Result object containing a list of Prototypes if they exist
     */
    suspend fun getPrototypesByUserId(
        userId: String,
        page: Int = 1,
        pageSize: Int = 20,
    ): Result<List<Prototype>> =
        runCatching {
            newSuspendedTransaction(Dispatchers.IO, db) {
                PrototypeEntity
                    .find { Prototypes.userId eq userId }
                    .orderBy(Prototypes.createdAt to SortOrder.DESC)
                    .limit(pageSize, offset = ((page - 1) * pageSize).toLong())
                    .map { it.toPrototype() }
            }
        }
}
```

Listing 134: Prototype Repository Implementation

Chat Repository The module also includes a `ChatRepository` for managing chat messages and conversations:

```
suspend fun saveMessage(message: ChatMessage): Boolean {
    return try {
        newSuspendedTransaction(IO_DISPATCHER, db) {
            val conversationId = if (message.conversationId.isNullOrBlank()) {
                println("DEBUG - Using random UUID since conversationId was empty")
                UUID.randomUUID()
            } else {
                try {
                    UUID.fromString(message.conversationId)
                } catch (e: Exception) {
                    println("DEBUG - UUID parse failed: ${e.message}")
                    UUID.randomUUID()
                }
            }

            val conversation = ConversationEntity.findById(conversationId) ?:
                ConversationEntity.new(conversationId) {
                    name = "New Conversation"
                    lastModified = message.timestamp
                    userId = if (message.senderId != "LLM") message.senderId else "user"
                }

            conversation.lastModified = message.timestamp

            val messageId = UUID.fromString(message.id)
            ChatMessageEntity.new(messageId) {
                this.conversation = conversation
                this.isFromLLM = message.senderId == "LLM"
                this.content = message.content
                this.timestamp = message.timestamp
            }
        }
        true
    } catch (e: Exception) {
        println("Error saving message: ${e.message}")
        false
    }
}
```

Listing 135: Chat Repository Example Method

Template Repository Another example is the `TemplateRepository`, which implements a different error handling approach:

```
class TemplateRepository(
    private val db: Database,
) {
    val logger = LoggerFactory.getLogger(TemplateRepository::class.java)
```

```

    /**
     * Function to save a Template to the database
     * @param template The Template to save
     * @return A Result object containing the success or failure of the operation
     */
    suspend fun saveTemplateToDB(template: Template): Result<Unit> =
        runCatching {
            require(!template.id.isBlank())

            newSuspendedTransaction(IO_DISPATCHER, db) {
                val existingTemplate = TemplateEntity.findById(template.id)
                if (existingTemplate != null) {
                    existingTemplate.fileURI = template.fileURI
                } else {
                    TemplateEntity.new(template.id) {
                        fileURI = template.fileURI
                    }
                }
            }
        }

    /**
     * Function to retrieve a Template from the database by its ID
     * @param id The ID of the Template to retrieve
     * @return The Template if it exists, or null if it does not
     */
    suspend fun getTemplateFromDB(id: String): Template? =
        runCatching {
            newSuspendedTransaction(IO_DISPATCHER, db) {
                TemplateEntity.findById(id)?.toTemplate()
            }
        }.onFailure { e ->
            logger.error("Error retrieving template with ID $id: ${e.message}", e)
        }.getOrNull()
}
```

Listing 136: Template Repository Implementation

Key Repository Features The repository implementation includes several important design elements:

Dependency Injection Each repository accepts a database instance as a constructor parameter, enabling:

- Easier unit testing with mock or test databases
- Flexibility to use different database connections if needed
- Clearer dependency flow in the application

Coroutine Support All database operations are implemented as suspending functions that:

- Ensure database operations don't block the main thread
- Allow for integration with asynchronous application flows
- Leverage structured concurrency

Error Handling with Result Type Most methods return the `Result` type to provide structured error handling:

- Success and failure cases are explicitly represented
- Callers can handle errors without try-catch blocks
- Database exceptions are encapsulated within the repository layer

Pagination Support For methods that might return large result sets, pagination parameters are included:

- Limits the number of records retrieved per request
- Allows for efficient navigation through large datasets
- Prevents potential out-of-memory issues with large result sets

These design decisions ensure the repository layer is robust, testable, and follows best practices for modern database applications.

Transaction Management The Database Module employs a sophisticated approach to transaction management using Exposed's coroutine-based transaction API. This ensures operations are performed efficiently and safely in a non-blocking manner.

Suspended Transactions The module uses Exposed's `newSuspendedTransaction` function to manage database transactions in a coroutine context:

```
suspend fun getPrototype(id: UUID): Result<Prototype?> =
    runCatching {
        newSuspendedTransaction(Dispatchers.IO, db) {
            PrototypeEntity.findById(id)?.toPrototype()
        }
    }
```

Listing 137: Suspended Transaction Example

This approach provides several important benefits:

- **Non-blocking execution**: Database operations run on the IO dispatcher, avoiding blocking the main thread
- **Automatic transaction boundaries**: The transaction is properly started and either committed or rolled back
- **Clean coroutine integration**: Works naturally with other suspending functions in the application
- **Exception safety**: Transactions are rolled back if exceptions occur

Transaction Isolation The module configures specific transaction isolation levels to ensure data consistency:

```
private fun setupDatabase(credentials: DatabaseCredentials): Database {
    val config = HikariConfig().apply {
        // Other configuration...
        isAutoCommit = false
        transactionIsolation = "TRANSACTION_REPEATABLE_READ"
        // More configuration...
    }
    // ...
}
```

Listing 138: Transaction Isolation Configuration

The use of `TRANSACTION_REPEATABLE_READ` isolation level ensures:

- Consistent reads within a transaction
- Prevention of non-repeatable reads
- Balance between consistency and concurrency

Exception Handling in Transactions The module uses two approaches for handling exceptions in transactions:

1. **Result-based approach**: Most repository methods use the `runCatching` function to wrap transactions in a `Result` type:

```
    suspend fun createPrototype(prototype: Prototype): Result<Unit> =
        runCatching {
            newSuspendedTransaction(Dispatchers.IO, db) {
                // Database operations...
            }
        }
```

Listing 139: Result-based Exception Handling

2. **Try-catch approach**: Some methods use traditional try-catch blocks for more control over error handling:

```
    suspend fun saveMessage(message: ChatMessage): Boolean {
        return try {
            newSuspendedTransaction(IO_DISPATCHER, db) {
                // Database operations...
            }
            true
        } catch (e: Exception) {
            println("Error saving message: ${e.message}")
            false
        }
    }
```

Listing 140: Try-catch Exception Handling

Both approaches ensure:

- Database exceptions are captured and handled appropriately
- Transactions are automatically rolled back on exception
- External callers are protected from internal database errors

Nested Transactions The Exposed framework supports nested transactions, which are used in more complex operations:

```
newSuspendedTransaction(Dispatchers.IO, db) {
    // Outer transaction
    val conversation = // Retrieve or create conversation...

    transaction {
        // Nested transaction
        // Update related entities...
    }

    // Continue outer transaction
}
```

Listing 141: Nested Transaction Example

Nested transactions provide:

- Greater control over transaction boundaries
- Ability to commit or roll back parts of a larger operation
- Better organizing of complex database operations

Transaction Context Features Inside a transaction block, developers can use the full range of Exposed's DSL and entity operations:

- Entity creation, retrieval, and updates
- Complex queries with filtering, ordering, and joins
- Batch operations for performance optimization
- Database functions and custom SQL when needed

The transaction management approach in the Database Module ensures robust and efficient database operations while maintaining clean, readable code that integrates well with the application's coroutine-based architecture.

7.3.3 Configuration and Environment

The Database Module uses a configuration approach based on environment variables, providing flexibility across different deployment environments while maintaining security best practices.

Environment Variables The module loads its configuration from environment variables, using the `EnvironmentLoader` utility:

```
private fun getDatabaseCredentials(): DatabaseCredentials {
    return DatabaseCredentials(
        url = EnvironmentLoader.get("DB_URL"),
        username = EnvironmentLoader.get("DB_USERNAME"),
        password = EnvironmentLoader.get("DB_PASSWORD"),
        maxPoolSize = EnvironmentLoader.get("DB_MAX_POOL_SIZE").toInt()
    )
}
```

Listing 142: Environment Configuration

Required Environment Variables The following environment variables must be configured for the Database Module to function properly:

- `DB_URL`: JDBC URL for the PostgreSQL database
- `DB_USERNAME`: Database username
- `DB_PASSWORD`: Database password
- `DB_MAX_POOL_SIZE`: Maximum number of connections in the pool

Environment File The repository includes a template `.env.example` file that can be used as a starting point for configuration:

```
DB_URL=jdbc:postgresql://localhost:5432/poc_database
DB_USERNAME=postgres
DB_PASSWORD=postgres
DB_MAX_POOL_SIZE=10
```

Listing 143: Example .env File

Configuration Best Practices When working with the Database Module, follow these configuration practices:

- **Never commit sensitive information**: Ensure the actual `.env` file is listed in `.gitignore`
- **Use different credentials per environment**: Development, testing, and production should use separate database instances and credentials
- **Set appropriate pool sizes**: Configure the connection pool size based on the expected load and available resources
- **Document any custom environment variables**: If extending the module with additional configuration, update the documentation accordingly

Connection Pool Configuration The connection pool is configured via HikariCP with several important parameters:

```
val config = HikariConfig().apply {
    jdbcUrl = credentials.url
    username = credentials.username
    password = credentials.password
    maximumPoolSize = max(1, credentials.maxPoolSize)
    isAutoCommit = false
    transactionIsolation = "TRANSACTION_REPEATABLE_READ"
    driverClassName = "org.postgresql.Driver"
}
```

Listing 144: HikariCP Configuration

Key HikariCP configuration options:

- `maximumPoolSize`: Controls the maximum number of connections in the pool
- `isAutoCommit`: Disabled to ensure explicit transaction control
- `transactionIsolation`: Set to ensure consistent read behavior
- `driverClassName`: Explicitly specified to ensure the correct JDBC driver is used

The configuration approach balances flexibility, security, and performance considerations, allowing the Database Module to be easily adapted to different deployment environments.

7.3.4 Database Migrations

The Database Module uses Flyway to manage database schema migrations, providing a reliable and version-controlled approach to evolving the database schema over time.

Flyway Configuration Flyway is configured and executed during the database initialization process:

```
private fun configureFlyway(credentials: DatabaseCredentials) {
    Flyway.configure()
        .dataSource(credentials.url, credentials.username, credentials.password)
        .locations("database/src/main/resources/db/migrations")
        .load()
        .migrate()
}
```

Listing 145: Flyway Configuration

Key aspects of the Flyway configuration:

- **Data source**: Uses the same credentials as the main database connection
- **Migration location**: Scripts are stored in a dedicated directory within the module
- **Automatic execution**: Migrations run automatically during database initialization

Migration Scripts Migration scripts should be placed in the `database/src/main/resources/db/migrations` directory and follow Flyway's naming convention:

```
V<version>__<description>.sql
```

Listing 146: Migration Script Naming Convention

For example:

- `V1__Create_prototypes_table.sql`
- `V2__Add_index_to_userId.sql`
- `V3__Add_new_column_to_prototypes.sql`

Migration Best Practices When working with database migrations, follow these best practices:

- **Version sequentially**: Ensure migration versions increment sequentially
- **Never modify existing migrations**: Once a migration is committed, treat it as immutable
- **Use descriptive names**: Make the script name clearly describe its purpose
- **Include both up and down changes**: Where possible, include statements to both apply and revert changes
- **Test migrations thoroughly**: Verify that migrations work correctly before committing
- **Keep migrations atomic**: Each migration should handle a single, cohesive change

Example Migration Script Below is an example of a migration script that creates the `prototypes` table:

```sql
-- V1__Create_prototypes_table.sql

CREATE TABLE prototypes (
    id UUID PRIMARY KEY,
    userId TEXT NOT NULL,
    prompt TEXT NOT NULL,
    fullPrompt TEXT NOT NULL,
    s3_key VARCHAR(255),
    created_at TIMESTAMP NOT NULL
);

-- Create index on userId for faster lookups
CREATE INDEX idx_prototypes_user_id ON prototypes(userId);

-- Create index on creation timestamp for efficient sorting
CREATE INDEX idx_prototypes_created_at ON prototypes(created_at DESC);
```

Listing 147: Example Flyway Migration Script

Handling Migration Failures If a migration fails, Flyway will throw an exception which will be captured by the `DatabaseManager.init()` method. This ensures that the application won't start with an inconsistent database state. To resolve migration failures:

1. Check the logs for the specific error message
2. Fix the issue in the database manually if necessary
3. If the migration needs to be modified, create a new migration that applies the correction
4. Restart the application to retry the migration process

The migration system ensures that database schema changes are applied consistently across all environments and provides a clear history of how the schema has evolved over time.

7.3.5 Setting Up the Development Environment

The Database Module supports running the database in a Docker container for development, providing a consistent and isolated environment for database operations.

Docker Setup The project includes Docker configuration for running a PostgreSQL database:

```bash
# Start the database
docker compose up -d

# Stop the database
docker compose down
```

Listing 148: Docker Commands

The Docker configuration automatically sets up a PostgreSQL instance with the necessary configuration for development. This approach ensures that all developers work with the same database configuration, eliminating environment-specific issues.

Environment Configuration To configure the development environment:

1. Copy the example environment file: `cp .env.example .env`
2. Update the `.env` file with appropriate values for your local setup
3. Ensure the `.env` file is in the project root and not committed to version control

A typical development environment file might look like:

```
DB_URL=jdbc:postgresql://localhost:5432/poc_database
DB_USERNAME=postgres
DB_PASSWORD=postgres
DB_MAX_POOL_SIZE=5
```

Listing 149: Development .env File

Database Initialization When the application starts for the first time with a new database, several initialization steps occur automatically:

1. Flyway creates its metadata table to track migration versions
2. All pending migrations are executed in version order
3. The database schema is created according to the migration scripts

7.4 Embeddings Module

The Embeddings Module provides functionality for generating text embeddings, storing them in vector databases, and performing semantic search operations. It serves as a bridge between natural language text and numerical vector representations, enabling powerful semantic search capabilities within the application.

7.4.1 Overview

The Embeddings Module is a critical component of the ProofIt application that transforms textual data into vector representations suitable for semantic search and similarity matching. It enables the application to understand the meaning and context of text beyond simple keyword matching.

Purpose and Functions The primary functions of the Embeddings Module include:

- **Text Embedding Generation**: Converting natural language text into numerical vector representations (embeddings) that capture semantic meaning.
- **Vector Storage**: Storing and indexing these vector embeddings efficiently for fast retrieval.
- **Semantic Search**: Enabling search operations based on semantic similarity rather than exact text matching.
- **Hybrid Search**: Combining vector-based semantic search with traditional keyword-based search for comprehensive results.
- **Template Management**: Managing component templates and their associated metadata for retrieval in UI component generation.

Key Capabilities The Embeddings Module provides several key capabilities:

- **Context-Aware Understanding**: The module captures the semantic meaning of text, allowing it to understand context, synonyms, and related concepts.
- **High-Dimensional Vector Operations**: The module efficiently handles operations on high-dimensional vector spaces, using techniques like cosine similarity to measure semantic relatedness.
- **Efficient Indexing**: The module employs FAISS (Facebook AI Similarity Search) for high-performance similarity search in dense vector spaces.
- **Text Indexing**: In addition to vector search, the module also implements traditional text indexing using Pyserini (built on Apache Lucene) for keyword-based search.
- **Cross-Language Integration**: The module bridges Kotlin/JVM code with Python machine learning libraries through a REST API-based microservice architecture.

Integration with Other Modules The Embeddings Module integrates with several other components of the application:

- **Template System**: Provides semantic search capabilities for finding relevant UI components and templates based on natural language descriptions.
- **Prompting Module**: Enhances prompt construction with semantically relevant context and information.
- **Chat Module**: Enables context-aware retrieval of information that can be incorporated into chat responses.
- **Database Module**: Coordinates with the database module for persistent storage of embedding-related metadata.

Technical Implementation The Embeddings Module is implemented as a hybrid system:

- **Python Microservice**: A Flask-based service that handles the computationally intensive embedding generation and vector operations, using libraries like sentence-transformers and FAISS.
- **Kotlin Client**: A set of Kotlin classes that interface with the Python microservice, providing a clean API for the rest of the JVM-based application.
- **REST API**: The communication between the Kotlin application and Python microservice occurs through a well-defined REST API, enabling language-agnostic integration.

This architecture leverages the strengths of both languages: Kotlin for application development and integration with the JVM ecosystem, and Python for its rich machine learning and natural language processing capabilities.

7.4.2 Architecture

The Embeddings Module employs a hybrid architectural pattern that bridges JVM and Python ecosystems. This approach combines the robustness of Kotlin for application development with the rich machine learning capabilities of Python.

High-Level Architecture The module is structured around a client-server architecture:

- **Python Microservice** (Server): A Flask-based service responsible for embedding generation, vector storage, and similarity search operations.
- **Kotlin Client**: Classes that communicate with the Python microservice through HTTP requests, providing a clean API for the rest of the application.

This separation allows each part of the system to leverage the strengths of its respective language ecosystem while communicating through a well-defined REST API.

Component Diagram The major components and their relationships are outlined below:

Embeddings Module Architecture

JVM Ecosystem (Kotlin)	**Python Ecosystem**
TemplateService	Flask API Server
TemplateRepository	Embedding Service
TemplateEmbedResponse	Vector Store
StoreTemplateResponse	Keyword Search
TemplateSearchResponse	Pyserini Indexer
JSON over HTTP	

Data Flow The typical data flow through the Embeddings Module involves several steps:

1. **Embedding Generation**: Text data is sent from the Kotlin application to the Python microservice, which generates vector embeddings using machine learning models.
2. **Vector Storage**: The generated embeddings are stored in a FAISS index, while the original text data is indexed using Pyserini for keyword search.
3. **Search Operations**: Queries are processed by combining the results of vector-based semantic search and keyword-based text search.
4. **Result Aggregation**: The results are aggregated and returned to the Kotlin application for further processing or presentation to the user.

This flow enables powerful semantic search capabilities while maintaining a clean architectural separation.

Python Microservice Architecture The Python microservice follows a layered architecture:

- **API Layer**: Flask routes and endpoints that handle HTTP requests and responses.
- **Service Layer**: Core functionality for embedding generation, vector operations, and search.
- **Storage Layer**: FAISS and Pyserini indexes for storing and retrieving vectors and text data.
- **Utility Layer**: Helper functions for data processing, normalization, and error handling.

This layered approach promotes separation of concerns and maintainability.

Kotlin Client Architecture The Kotlin client follows a similar layered architecture:

- **API Client**: Classes responsible for HTTP communication with the Python microservice.
- **Data Models**: Serializable data classes for request and response payloads.
- **Service Layer**: High-level services that abstract away the details of the HTTP communication and provide a clean API for the rest of the application.
- **Repository Layer**: Integration with the broader application's data access patterns.

Communication Protocol The Kotlin client and Python microservice communicate through a JSON-based REST API:

- **Embedding Endpoint** (`/embed`): Generates vector embeddings for input text.
- **Storage Endpoint** (`/new`): Stores text data and its embedding for future search.
- **Search Endpoint** (`/search`): Performs semantic search based on input embeddings and/or keywords.

Each endpoint accepts and returns JSON data, making the communication language-agnostic.

Persistence Strategy The Embeddings Module maintains several forms of persistent data:

- **FAISS Index**: Stores vector embeddings for efficient similarity search.
- **Pyserini Index**: Stores text data for keyword-based search.
- **Mapping File**: Maintains the association between vector IDs and document identifiers.
- **Database Records**: Metadata about embeddings and their associations with templates is stored in the application's database.

This multi-layered persistence strategy ensures robust data management while optimizing for search performance.

Error Handling and Resilience The architecture includes several mechanisms for error handling and resilience:

- **Service Initialization Check**: The system verifies the availability of the Python microservice before operations and gracefully handles initialization failures.
- **Request Timeouts**: HTTP requests include appropriate timeouts to prevent hanging operations.
- **Result Validation**: Responses from the microservice are validated before being used by the application.
- **Fallback Mechanisms**: If vector search fails, the system can fall back to keyword search and vice versa.

This architecture provides a robust foundation for the Embeddings Module, enabling powerful semantic search capabilities while maintaining clean separation of concerns and technological stacks.

7.4.3 Core Components

The Embeddings Module consists of several core components that work together to provide semantic search capabilities. These components are distributed between the Kotlin client and Python microservice parts of the architecture.

Kotlin Components

- **TemplateService**: The main service class that coordinates embedding operations and communicates with the Python microservice.

```
object TemplateService {
    internal var httpClient = HttpClient(CIO)

    /**
     * Embeds the given data and returns the embedding.
     */
    suspend fun embed(
        data: String,
        name: String,
    ): TemplateEmbedResponse {
        val payload = mapOf("text" to data, "name" to name)
        val response =
            httpClient
                .post(EmbeddingConstants.EMBED_URL) {
                    header(HttpHeaders.ContentType, "application/json")
                    setBody(Json.encodeToString(payload))
                }
        val responseText = response.bodyAsText()
        return runCatching { Json.decodeFromString<TemplateEmbedResponse>(responseText)
            }.getOrElse {
            error("Failed to store template!")
        }
    }

    /**
     * Stores the given template in both the embedding service and local storage.
     */
    suspend fun storeTemplate(
        fileURI: String,
        data: String,
    ): StoreTemplateResponse {
        // Create local template record
        val templateId = TemplateStorageService.createTemplate(fileURI) ?: error("Failed
             to store template!")

        // Store in embedding service
        val remoteResponse = storeTemplateEmbedding(templateId, data)
        val success = remoteResponse.status == HttpStatusCode.OK
        return if (success) {
            Json.decodeFromString<StoreTemplateResponse>(remoteResponse.bodyAsText()).
                copy(id = templateId)
        } else {
            error("Failed to store template!")
        }
    }

    /**
     * Performs a semantic search against stored templates using an embedding vector.
     */
    suspend fun search(
        embedding: List<Float>,
        query: String,
    ): TemplateSearchResponse {
        val payload = SearchData(embedding, query)
        val response =
            httpClient
                .post(EmbeddingConstants.SEMANTIC_SEARCH_URL) {
                    header(HttpHeaders.ContentType, "application/json")
                    setBody(Json.encodeToString(payload))
                }

        val responseText = response.bodyAsText()
        return runCatching { Json.decodeFromString<TemplateSearchResponse>(responseText)
             }.getOrElse {
            error("Failed to store template!")
        }
    }
}
```

Listing 150: TemplateService Object

- **EmbeddingConstants**: Contains URL constants for the embedding service endpoints.

```
internal object EmbeddingConstants {
    private const val EMBEDDING_SERVICE_URL = "http://localhost:7000/"
    const val EMBED_URL = "$EMBEDDING_SERVICE_URL/embed"
```

```
    const val EMBED_AND_STORE_URL = "$EMBEDDING_SERVICE_URL/new"
    const val SEMANTIC_SEARCH_URL = "$EMBEDDING_SERVICE_URL/search"
}
```

Listing 151: EmbeddingConstants Object

- **TemplateEmbedResponse**: Data class representing a response from the embedding generation endpoint.

```
@Serializable
data class TemplateEmbedResponse(
    val status: String,
    val embedding: List<Float> = emptyList(),
)
```

Listing 152: TemplateEmbedResponse Data Class

- **StoreTemplateResponse**: Data class representing a response from storing a template with its embedding.

```
@Serializable
data class StoreTemplateResponse(
    val status: String,
    val id: String? = null,
    val message: String? = null,
)
```

Listing 153: StoreTemplateResponse Data Class

- **TemplateSearchResponse**: Data class representing a response from a semantic search operation.

```
@Serializable
data class TemplateSearchResponse(
    val status: String,
    val matches: List<String> = emptyList(),
)
```

Listing 154: TemplateSearchResponse Data Class

- **SearchData**: Data class representing a request for semantic search.

```
@Serializable
internal data class SearchData(
    val embedding: List<Float>,
    val query: String,
)
```

Listing 155: SearchData Data Class

- **TemplateStorageService**: Service for managing template storage and retrieval in the database.

```
object TemplateStorageService {
    var logger = LoggerFactory.getLogger(TemplateService::class.java)

    /**
     * Creates a new template and stores it in the database.
     */
    suspend fun createTemplate(fileURI: String): String? {
        val templateId = UUID.randomUUID().toString()
        val template = Template(id = templateId, fileURI = fileURI)
        val result = runCatching {
            DatabaseManager.templateRepository().saveTemplateToDB(template)
        }

        return if (result.isSuccess) {
            templateId
        } else {
            null
        }
    }

    /**
     * Retrieves a template by its ID.
     */
    suspend fun getTemplateById(templateId: String): Template? {
        val template = runCatching {
            DatabaseManager.templateRepository().getTemplateFromDB(templateId)
        }.getOrElse {
            println("Error retrieving template with ID $templateId: ${it.message}")
            null
```

```
            }

            return template ?: run {
                println("Failed to get template with the following id: $templateId")
                null
            }
        }
    }
```

Listing 156: TemplateStorageService Object

Python Components

- **Embedding Service**: The main Flask application that serves as the entry point for the Python microservice.

```
    from flask import Flask, jsonify, request
    from flask_cors import CORS

    from information_retrieval.data_handler import load_data, save_data
    from information_retrieval.keyword_search import pyserini_indexer as pi
    from information_retrieval.vector_search import embedder as emb, vector_store as vs

    app = Flask(__name__)
    CORS(app)

    first_request = True
    @app.before_request
    def startup_once():
        global first_request
        if first_request:
            vs.index, vs.store = load_data()
            first_request = False
```

Listing 157: Embedding Service Initialization

- **Embedder**: Responsible for converting text into vector embeddings using a pre-trained model.

```
    from sentence_transformers import SentenceTransformer
    import numpy as np

    model = SentenceTransformer('all-MiniLM-L6-v2')

    def embed(text: str):
        """Converts input text into vector embeddings using a huggingface sentence
            transformer."""
        if not isinstance(text, str):
            return None
        return normalize(model.encode(text).tolist())


    def normalize(embedding: list[float]) -> list[float]:
        """Normalize the embedding vectors so that they sum up to 1 (or very close to)."""
        embedding = np.array(embedding)
        norm = np.linalg.norm(embedding)
        return (embedding / norm).tolist() if norm != 0 else embedding.tolist()
```

Listing 158: Embedder Implementation

- **Vector Store**: Manages storage and retrieval of vector embeddings using FAISS.

```
    import numpy as np

    index, store = None, {}

    def semantic_search(embedding: list, top_k: int):
        base = np.array(embedding, dtype=np.float32).reshape(1, -1)
        if index.ntotal == 0:
            return []

        distances, indices = index.search(base, top_k)
        results = [store[idx] for idx in indices[0] if idx in store]
        return results

    def store_embedding(name: str, vector: np.array) -> bool:
        if not index.is_trained:
            return False
```

```
        vector = vector.reshape(1, -1)
        index.add(vector)
        store[len(store)] = name
        return True
```

Listing 159: Vector Store Implementation

- **Pyserini Indexer**: Handles keyword-based search using Pyserini (a Python wrapper for Apache Lucene).

```
    import json
    import os
    from pyserini.index.lucene import LuceneIndexer
    from pyserini.search.lucene import LuceneSearcher
    from information_retrieval.data_handler import LUCENE_INDEX_DIR

    def store_jsonld(name:str, data: dict) -> bool:
        """Stores JSON-LD metadata and indexes it with Pyserini."""
        if not isinstance(data, dict):
            return False

        # Ensure the directory exists
        os.makedirs(LUCENE_INDEX_DIR, exist_ok=True)

        # Create or append to the index
        indexer = LuceneIndexer(LUCENE_INDEX_DIR)
        indexer.add_doc_dict({
            "id": name,
            "contents": json.dumps(data),
        })

        indexer.close()
        return True


    def keyword_search(query: str, top_k: int = 5):
        """Performs a keyword-based search using Pyserini (BM25 ranking)."""
        if not os.path.exists(LUCENE_INDEX_DIR) or not os.listdir(LUCENE_INDEX_DIR):
            return []

        try:
            searcher = LuceneSearcher(LUCENE_INDEX_DIR)
            hits = searcher.search(query, k=top_k)

            results = []
            for hit in hits:
                results.append(hit.docid)

            return results
        except Exception as e:
            print(f"Error during keyword search: {e}")
            return []
```

Listing 160: Pyserini Indexer Implementation

- **Data Handler**: Manages loading and saving of FAISS index and mapping data.

```
    import faiss
    import pickle
    import os
    import pathlib

    # Get the absolute path of the current directory
    BASE_DIR = str(pathlib.Path(__file__).parent.parent.absolute())

    # Define file paths relative to the base directory
    FAISS_FILE = os.path.join(BASE_DIR, "faiss.index")
    MAPPINGS_FILE = os.path.join(BASE_DIR, "mappings.pkl")
    LUCENE_INDEX_DIR = os.path.join(BASE_DIR, "jsonld_index")
    VECTOR_DIMENSION = 384

    def load_data():
        """
        Retrieve persisted data from disk. These will be embeddings and corresponding
            mappings.
        """
        try:
            index = faiss.read_index(FAISS_FILE)
```

```
            if index is None:
                index = faiss.IndexFlatIP(VECTOR_DIMENSION)
        except Exception as e:
            print(f"Could not load FAISS index from {FAISS_FILE} ({e}). Creating new index."
                )
            index = faiss.IndexFlatIP(VECTOR_DIMENSION)

        try:
            with open(MAPPINGS_FILE, "rb") as f:
                vector_store = pickle.load(f)
            if vector_store is None:
                vector_store = {}
        except Exception as e:
            print(f"Could not load mapping from {MAPPINGS_FILE} ({e}). Creating new mapping.
                ")
            vector_store = {}

        # Ensure the Lucene index directory exists
        os.makedirs(LUCENE_INDEX_DIR, exist_ok=True)

        return index, vector_store

    def save_data(index, vector_store):
        """
        Save data into disk for persistence.
        """
        faiss.write_index(index, FAISS_FILE)
        with open(MAPPINGS_FILE, "wb") as f:
            pickle.dump(vector_store, f)
```

Listing 161: Data Handler Implementation

API Endpoints The Python microservice exposes several endpoints:

- **/embed**: Generates embeddings for input text.

```
    @app.route('/embed', methods=['POST'])
    def embed_route():
        data = request.json
        if not "text" in data:
            return jsonify({
                "status": "error",
                "message": "No prompt provided"
            })
        text = data["text"]
        embedding = emb.embed(text)
        if not embedding:
            return jsonify({"status": "error", "message": f"Error embedding: {text}"})
        return jsonify({"status": "success", "embedding": embedding})
```

Listing 162: Embed Endpoint Implementation

- **/new**: Stores text data and its embedding for future search.

```
    @app.route('/new', methods=['POST'])
    def new_template_route():
        data = request.json
        if not "text" in data:
            return jsonify({"status": "error", "message": "No prompt provided"})

        jsonld = data["text"]
        name = data["name"]
        embedding = emb.embed(jsonld)
        vector_success = vs.store_embedding(name, vector = np.array(embedding))
        keyword_success = pi.store_jsonld(name, json.loads(jsonld))
        if not vector_success or not keyword_success:
            return jsonify({"status": "error", "message": f"Failed to store template: Vector
                 DB: {vector_success}, Keyword DB: {keyword_success}"})

        # Save data to disk after successful storage
        save_data(vs.index, vs.store)

        return jsonify({"status": "success", "message": "New template stored successfully!"
            })
```

Listing 163: New Template Endpoint Implementation

- **/search**: Performs semantic search based on input embeddings and/or keywords.

```
    @app.route('/search', methods=['POST'])
    def search_route():
        data = request.json
        if not "embedding" in data:
            return jsonify({"status": "error", "message": "No embedding provided for
                semantic search!"})
        if not "query" in data:
            return jsonify({"status": "error", "message": "No query provided for keyword
                search!"})

        top_k = data.get("top_k", 5)
        vector_results = vs.semantic_search(data["embedding"], top_k=top_k)
        keyword_results = pi.keyword_search(data["query"], top_k=top_k)
        results = list(set(vector_results + keyword_results))
        return jsonify({"status": "success", "matches": results})
```

Listing 164: Search Endpoint Implementation

These core components work together to provide the complete embedding and semantic search functionality. The Kotlin components handle integration with the rest of the application, while the Python components provide the specialized machine learning and vector search capabilities.

7.4.4 Vector Search

The vector search component of the Embeddings Module enables efficient semantic similarity search using vector embeddings. This component leverages the FAISS (Facebook AI Similarity Search) library to perform high-performance similarity operations in dense vector spaces.

Vector Embedding Generation Before vector search can be performed, text must be converted into numerical vector embeddings. This is handled by the `embedder` component:

```
from sentence_transformers import SentenceTransformer
import numpy as np

model = SentenceTransformer('all-MiniLM-L6-v2')

def embed(text: str):
    """Converts input text into vector embeddings using a huggingface sentence transformer."""
    if not isinstance(text, str):
        return None
    return normalize(model.encode(text).tolist())

def normalize(embedding: list[float]) -> list[float]:
    """Normalize the embedding vectors so that they sum up to 1 (or very close to)."""
    embedding = np.array(embedding)
    norm = np.linalg.norm(embedding)
    return (embedding / norm).tolist() if norm != 0 else embedding.tolist()
```

Listing 165: Embedding Generation Process

Key aspects of the embedding generation process:

- The module uses the `all-MiniLM-L6-v2` sentence transformer model to generate embeddings
- The model produces 384-dimensional vector embeddings for input text
- Embeddings are normalized to unit length to ensure cosine similarity calculations are consistent
- The embedding process handles input validation, ensuring only string inputs are processed

FAISS Index Management The heart of the vector search functionality is the FAISS index, which enables efficient similarity search in high-dimensional spaces:

```
import faiss
import pickle
import os

# Define file paths and dimensions
FAISS_FILE = os.path.join(BASE_DIR, "faiss.index")
MAPPINGS_FILE = os.path.join(BASE_DIR, "mappings.pkl")
VECTOR_DIMENSION = 384

def load_data():
```

```
    """Retrieve persisted data from disk. These will be embeddings and corresponding mappings.
        """
    try:
        index = faiss.read_index(FAISS_FILE)
        if index is None:
            index = faiss.IndexFlatIP(VECTOR_DIMENSION)
    except Exception as e:
        print(f"Could not load FAISS index from {FAISS_FILE} ({e}). Creating new index.")
        index = faiss.IndexFlatIP(VECTOR_DIMENSION)

    try:
        with open(MAPPINGS_FILE, "rb") as f:
            vector_store = pickle.load(f)
        if vector_store is None:
            vector_store = {}
    except Exception as e:
        print(f"Could not load mapping from {MAPPINGS_FILE} ({e}). Creating new mapping.")
        vector_store = {}

    return index, vector_store

def save_data(index, vector_store):
    """Save data into disk for persistence."""
    faiss.write_index(index, FAISS_FILE)
    with open(MAPPINGS_FILE, "wb") as f:
        pickle.dump(vector_store, f)
```

Listing 166: FAISS Index Initialization

Key aspects of FAISS index management:

- The module uses `IndexFlatIP`, which is a flat index optimized for inner product (cosine similarity)
- The index is persisted to disk as a `faiss.index` file
- A separate mapping file maintains the relationship between vector IDs and document IDs
- Robust error handling ensures that a new index is created if the existing one cannot be loaded
- The system supports both loading existing indices and creating new ones as needed

Vector Store Operations The `vector_store` module provides the core functionality for storing and searching vector embeddings:

```
import numpy as np

index, store = None, {}

def semantic_search(embedding: list, top_k: int):
    """
    Performs a semantic search using the provided embedding vector.

    Args:
        embedding: The query vector for similarity search
        top_k: Maximum number of results to return

    Returns:
        A list of document IDs sorted by similarity to the query vector
    """
    base = np.array(embedding, dtype=np.float32).reshape(1, -1)
    if index.ntotal == 0:
        return []

    distances, indices = index.search(base, top_k)
    results = [store[idx] for idx in indices[0] if idx in store]
    return results

def store_embedding(name: str, vector: np.array) -> bool:
    """
    Stores a vector embedding in the FAISS index.

    Args:
        name: The document ID associated with the vector
        vector: The vector embedding to store

    Returns:
        True if the vector was successfully stored, False otherwise
    """
    if not index.is_trained:
```

```
        return False
    vector = vector.reshape(1, -1)
    index.add(vector)
    store[len(store)] = name
    return True
```

Listing 167: Vector Store Operations

The vector store provides two main operations:

- **Semantic Search**: Finds similar documents based on vector similarity
  - Reshapes the input embedding into a format suitable for FAISS
  - Handles empty index cases gracefully
  - Performs similarity search using the FAISS index
  - Maps the resulting indices back to document IDs
  - Returns a list of document IDs sorted by similarity
- **Store Embedding**: Adds new embeddings to the index
  - Verifies that the index is properly trained and initialized
  - Reshapes the input vector to ensure consistent dimensionality
  - Adds the vector to the FAISS index
  - Updates the mapping between vector indices and document IDs
  - Returns a success indicator

Similarity Computation The vector search component uses cosine similarity to measure the semantic relatedness between documents:

```
# FAISS uses inner product for IndexFlatIP, which is equivalent to cosine similarity
# when vectors are normalized to unit length
base = np.array(embedding, dtype=np.float32).reshape(1, -1)
distances, indices = index.search(base, top_k)
```

Listing 168: Cosine Similarity Computation

Key features of the similarity computation:

- Vectors are normalized to unit length during the embedding process
- FAISS's `IndexFlatIP` uses inner product, which is equivalent to cosine similarity for normalized vectors
- The search function returns both distances (similarity scores) and indices
- Higher similarity scores indicate greater semantic relatedness

Performance Considerations The vector search implementation includes several performance optimizations:

- **Efficient Indexing**: FAISS is designed for fast similarity search in high-dimensional spaces
- **Dimensionality**: The `all-MiniLM-L6-v2` model generates relatively compact 384-dimensional embeddings, balancing quality and performance
- **Flat Index**: The current implementation uses a flat index, which is simple and accurate but may not scale as well as more complex index types
- **Caching**: The index is loaded once during initialization and kept in memory for subsequent operations
- **Batch Operations**: The implementation supports adding vectors individually but could be extended for batch operations

For larger-scale deployments, several additional optimizations could be considered:

- Using quantized indices (e.g., `IndexIVFPQ`) for better space efficiency
- Implementing sharding for distributed search across multiple machines
- Adding a memory cache layer to reduce disk I/O operations
- Pre-computing embeddings for common queries

Integration with Search Workflow The vector search component is integrated into the overall search workflow through the search endpoint:

```
@app.route('/search', methods=['POST'])
def search_route():
    data = request.json
    if not "embedding" in data:
```

```
        return jsonify({"status": "error", "message": "No embedding provided for semantic
            search!"})
    if not "query" in data:
        return jsonify({"status": "error", "message": "No query provided for keyword search!"
            })

    top_k = data.get("top_k", 5)
    vector_results = vs.semantic_search(data["embedding"], top_k=top_k)
    keyword_results = pi.keyword_search(data["query"], top_k=top_k)
    results = list(set(vector_results + keyword_results))
    return jsonify({"status": "success", "matches": results})
```

Listing 169: Search Endpoint Integration

This integration ensures that vector search results are combined with keyword search results to provide comprehensive and relevant matches for user queries.

The vector search component is a critical part of the Embeddings Module, enabling semantic understanding and similarity matching beyond what traditional keyword search can provide. By leveraging the power of neural embeddings and efficient similarity search algorithms, this component significantly enhances the application's search capabilities.

7.4.5 Keyword Search

In addition to vector-based semantic search, the Embeddings Module implements traditional keyword-based search using Pyserini, a Python wrapper for Apache Lucene. This complementary approach provides robust search capabilities that combine the strengths of both semantic understanding and lexical matching.

Pyserini Indexer The `pyserini_indexer` component handles indexing and search operations for text data:

```
import json
import os
from pyserini.index.lucene import LuceneIndexer
from pyserini.search.lucene import LuceneSearcher
from information_retrieval.data_handler import LUCENE_INDEX_DIR

JSONL_FILE = "jsonld_docs.jsonl"

def store_jsonld(name:str, data: dict) -> bool:
    """Stores JSON-LD metadata and indexes it with Pyserini."""
    if not isinstance(data, dict):
        return False

    # Ensure the directory exists
    os.makedirs(LUCENE_INDEX_DIR, exist_ok=True)

    # Check if there are existing documents in the index
    existing_docs = []
    if os.path.exists(LUCENE_INDEX_DIR) and os.listdir(LUCENE_INDEX_DIR):
        try:
            # Try to search for existing documents to check if the index is valid
            searcher = LuceneSearcher(LUCENE_INDEX_DIR)
            searcher.close()
        except Exception as e:
            print(f"Error with existing index: {e}. Creating a new one.")
            # If there's an error, we'll create a new index
            for item in os.listdir(LUCENE_INDEX_DIR):
                item_path = os.path.join(LUCENE_INDEX_DIR, item)
                if os.path.isfile(item_path):
                    os.remove(item_path)
                elif os.path.isdir(item_path):
                    import shutil
                    shutil.rmtree(item_path)

    # Create or append to the index
    indexer = LuceneIndexer(LUCENE_INDEX_DIR)
    indexer.add_doc_dict({
        "id": name,
        "contents": json.dumps(data),
    })

    indexer.close()

    print(f"Saved document '{name}' to Lucene index at {LUCENE_INDEX_DIR}")
    return True
```

Listing 170: Pyserini Indexer Implementation

Key aspects of the indexing process:

- The function accepts a document identifier and a dictionary of data to index
- It ensures the index directory exists before attempting to add data
- It validates the existing index to ensure it's in a consistent state
- It creates a new index if necessary or appends to the existing one
- It converts the data to a JSON string for storage in the Lucene index
- The document is indexed with an ID field and a contents field

Keyword Search Implementation The `keyword_search` function implements lexical search using Pyserini:

```
def keyword_search(query: str, top_k: int = 5):
    """Performs a keyword-based search using Pyserini (BM25 ranking)."""
    if not os.path.exists(LUCENE_INDEX_DIR) or not os.listdir(LUCENE_INDEX_DIR):
        return []

    try:
        searcher = LuceneSearcher(LUCENE_INDEX_DIR)
        hits = searcher.search(query, k=top_k)

        results = []
        for hit in hits:
            results.append(hit.docid)

        return results
    except Exception as e:
        print(f"Error during keyword search: {e}")
        return []
```

Listing 171: Keyword Search Implementation

Key aspects of the search implementation:

- The function accepts a query string and a parameter for the maximum number of results
- It checks for the existence of the index directory and returns an empty list if it doesn't exist
- It creates a Lucene searcher to query the index
- It performs the search and extracts document IDs from the search hits
- It returns a list of document IDs sorted by relevance
- It handles exceptions gracefully, returning an empty list and logging errors

BM25 Ranking The keyword search implementation uses the BM25 ranking algorithm, which is the default in Pyserini and Lucene:

```
searcher = LuceneSearcher(LUCENE_INDEX_DIR)
hits = searcher.search(query, k=top_k)
```

Listing 172: BM25 Ranking

BM25 (Best Matching 25) is a probabilistic ranking function used to rank documents by relevance based on the query terms they contain. It considers:

- Term frequency: How often a query term appears in a document
- Inverse document frequency: How rare or common a term is across all documents
- Document length normalization: Longer documents tend to use the same terms more often

This ranking algorithm provides robust text retrieval capabilities based on lexical matching, complementing the semantic understanding provided by vector search.

Index Management The Pyserini indexer includes several features for managing the Lucene index:

- **Index Validation**: The indexer verifies that the existing index is valid before using it:

```
    if os.path.exists(LUCENE_INDEX_DIR) and os.listdir(LUCENE_INDEX_DIR):
        try:
            # Try to search for existing documents to check if the index is valid
            searcher = LuceneSearcher(LUCENE_INDEX_DIR)
            searcher.close()
```

```
        except Exception as e:
            print(f"Error with existing index: {e}. Creating a new one.")
            # If there's an error, we'll create a new index
            # ...
```

Listing 173: Index Validation

- **Index Cleanup**: If an invalid or corrupted index is detected, the system cleans up the directory before creating a new index:

```
    for item in os.listdir(LUCENE_INDEX_DIR):
        item_path = os.path.join(LUCENE_INDEX_DIR, item)
        if os.path.isfile(item_path):
            os.remove(item_path)
        elif os.path.isdir(item_path):
            import shutil
            shutil.rmtree(item_path)
```

Listing 174: Index Cleanup

- **Resource Management**: The system properly closes indexers and searchers to prevent resource leaks:

```
    searcher = LuceneSearcher(LUCENE_INDEX_DIR)
    # Use the searcher...
    searcher.close()
```

Listing 175: Resource Management

Integration with Search Workflow The keyword search component is integrated into the overall search workflow through the search endpoint:

```
@app.route('/search', methods=['POST'])
def search_route():
    data = request.json
    if not "embedding" in data:
        return jsonify({"status": "error", "message": "No embedding provided for semantic
            search!"})
    if not "query" in data:
        return jsonify({"status": "error", "message": "No query provided for keyword search!"
            })

    top_k = data.get("top_k", 5)
    vector_results = vs.semantic_search(data["embedding"], top_k=top_k)
    keyword_results = pi.keyword_search(data["query"], top_k=top_k)
    results = list(set(vector_results + keyword_results))
    return jsonify({"status": "success", "matches": results})
```

Listing 176: Search Endpoint Integration

This integration ensures that keyword search results are combined with vector search results to provide comprehensive matches for user queries.

Error Handling and Robustness The keyword search implementation includes robust error handling to ensure reliable operation:

- **Input Validation**: The system verifies that input data is of the correct type before processing
- **Index Existence Check**: The search function checks for the existence of the index directory before attempting to search
- **Exception Handling**: All operations are wrapped in try-except blocks to handle unexpected errors gracefully
- **Error Logging**: Errors are logged with descriptive messages to aid in troubleshooting
- **Fallback Behavior**: In case of errors, the system provides sensible fallback behavior (empty results) rather than failing completely

The keyword search component provides a valuable complement to the vector-based semantic search, offering precise lexical matching capabilities alongside semantic understanding. By combining these approaches, the Embeddings Module delivers more comprehensive and accurate search results than either approach could provide on its own.

7.4.6 Python Integration

The Embeddings Module bridges Kotlin and Python ecosystems through a microservice architecture. This integration enables the application to leverage Python's rich machine learning ecosystem while maintaining the robustness of Kotlin for the main application logic.

Architecture Overview The integration between Python and Kotlin follows a client-server model:

- **Python Service**: A Flask-based REST API that provides embedding generation, storage, and search functionality
- **Kotlin Client**: HTTP client code that communicates with the Python service
- **Communication Protocol**: JSON over HTTP for language-agnostic data exchange

This architecture allows each part to leverage the strengths of its language ecosystem while maintaining a clean separation of concerns.

Python Microservice Setup The Python microservice is implemented as a Flask application with CORS support:

```
from flask import Flask, jsonify, request
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

first_request = True
@app.before_request
def startup_once():
    global first_request
    if first_request:
        vs.index, vs.store = load_data()
        first_request = False

if __name__ == '__main__':
    atexit.register(save_data, vs.index, vs.store)
    app.run(host="0.0.0.0", port=7000)
```

Listing 177: Flask App Setup

Key aspects of the setup:

- The application uses Flask for routing and request handling
- CORS support is enabled to allow cross-origin requests
- The FAISS index and store are loaded during initialization
- The application runs on port 7000 and accepts connections from any host
- A shutdown hook ensures data is saved when the application terminates

Python Dependencies The Python microservice depends on several libraries:

```
# Core dependencies
faiss-cpu==1.10.0
numpy==1.25.2
flask==3.1.0
flask-cors==5.0.0
sentence-transformers==3.4.1

# Pyserini and related dependencies
pyserini==0.44.0
pyjnius==1.6.1
Cython==3.0.12

# Utility libraries
werkzeug==3.1.3
pytest==8.3.4
```

Listing 178: Python Dependencies

These dependencies provide:

- Vector operations and similarity search (FAISS, NumPy)
- Web service functionality (Flask, Werkzeug)

- Embedding generation (sentence-transformers)
- Text search capabilities (Pyserini, Lucene via JNI)
- Testing framework (pytest)

Python Initialization and Shutdown The microservice implements careful initialization and shutdown procedures:

```python
# Initialization
@app.before_request
def startup_once():
    global first_request
    if first_request:
        vs.index, vs.store = load_data()
        first_request = False

# Shutdown
if __name__ == '__main__':
    atexit.register(save_data, vs.index, vs.store)
    app.run(host="0.0.0.0", port=7000)
```

Listing 179: Initialization and Shutdown

This ensures:

- Data is loaded only once, during the first request
- Resources are properly initialized before use
- Data is saved when the application shuts down
- Resources are cleaned up properly on exit

Kotlin HTTP Client The Kotlin side implements an HTTP client using Ktor:

```kotlin
object TemplateService {
    internal var httpClient = HttpClient(CIO)

    suspend fun embed(
        data: String,
        name: String,
    ): TemplateEmbedResponse {
        val payload = mapOf("text" to data, "name" to name)
        val response =
            httpClient
                .post(EmbeddingConstants.EMBED_URL) {
                    header(HttpHeaders.ContentType, "application/json")
                    setBody(Json.encodeToString(payload))
                }
        val responseText = response.bodyAsText()
        return runCatching { Json.decodeFromString<TemplateEmbedResponse>(responseText) }.
            getOrElse {
            error("Failed to store template!")
        }
    }

    // Other methods...
}
```

Listing 180: HTTP Client Implementation

Key aspects of the HTTP client:

- Uses Ktor's CIO engine for asynchronous HTTP communication
- Implements suspending functions for non-blocking operation
- Properly sets content types and headers
- Handles JSON serialization and deserialization
- Implements error handling with proper error propagation

Request and Response Format The communication between Python and Kotlin uses JSON for data exchange:

```
// Request to /embed endpoint
{
  "text": "A responsive login form with email, password, and Google OAuth.",
  "name": "LoginForm"
```

```
}

// Response from /embed endpoint
{
  "status": "success",
  "embedding": [0.123, 0.456, ...]
}
```

Listing 181: Example JSON Payload

The JSON format provides:

- Language-agnostic data representation
- Structured and self-describing data
- Support for nested objects and arrays
- Compatibility with both Python and Kotlin serialization libraries

Error Handling Across Languages The integration implements consistent error handling across language boundaries:

```
@app.route('/embed', methods=['POST'])
def embed_route():
    data = request.json
    if not "text" in data:
        return jsonify({
            "status": "error",
            "message": "No prompt provided"
        })
    text = data["text"]
    embedding = emb.embed(text)
    if not embedding:
        return jsonify({"status": "error", "message": f"Error embedding: {text}"})
    return jsonify({"status": "success", "embedding": embedding})
```

Listing 182: Python-Side Error Handling

```
suspend fun embed(
    data: String,
    name: String,
): TemplateEmbedResponse {
    val payload = mapOf("text" to data, "name" to name)
    val response =
        httpClient
            .post(EmbeddingConstants.EMBED_URL) {
                header(HttpHeaders.ContentType, "application/json")
                setBody(Json.encodeToString(payload))
            }
    val responseText = response.bodyAsText()
    return runCatching { Json.decodeFromString<TemplateEmbedResponse>(responseText) }.
        getOrElse {
        error("Failed to store template!")
    }
}
```

Listing 183: Kotlin-Side Error Handling

This approach ensures:

- Errors are properly communicated across language boundaries
- Both Python and Kotlin code handle errors appropriately
- Error messages are propagated to the appropriate level
- The system degrades gracefully when errors occur

Docker Integration The Python microservice includes Docker support for containerized deployment:

```
FROM python:3.10-slim

# Install OpenJDK 21 (required for Pyserini)
RUN apt-get update && apt-get install -y --no-install-recommends \
    wget \
    ca-certificates \
    gnupg \
    build-essential \
```

```
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Add Adoptium repository (provides OpenJDK packages)
RUN mkdir -p /etc/apt/keyrings && \
    wget -O - https://packages.adoptium.net/artifactory/api/gpg/key/public | tee /etc/apt/
        keyrings/adoptium.asc && \
    echo "deb [signed-by=/etc/apt/keyrings/adoptium.asc] https://packages.adoptium.net/
        artifactory/deb $(awk -F= '/^VERSION_CODENAME/{print$2}' /etc/os-release) main" | tee
        /etc/apt/sources.list.d/adoptium.list

# Install OpenJDK 21
RUN apt-get update && apt-get install -y --no-install-recommends \
    temurin-21-jdk \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Set JAVA_HOME environment variable
ENV JAVA_HOME=/usr/lib/jvm/temurin-21-jdk-amd64

# Create and set working directory
WORKDIR /app

# Copy requirements first to leverage Docker cache
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Set Python path
ENV PYTHONPATH=/app

# Command to run the application
CMD ["python3.10", "-m", "information_retrieval"]

EXPOSE 7000
```

Listing 184: Dockerfile for Python Microservice

The Docker setup ensures:

- Consistent environment across deployments
- All required dependencies are properly installed
- Both Python and Java environments are available (needed for Pyserini)
- The application is properly exposed on port 7000
- The Python path is correctly configured

Testing Across Language Boundaries The integration includes tests that verify the communication between Python and Kotlin:

```
@Test
fun `Test embed returns correct response on success`() =
    runBlocking {
        val engine =
            MockEngine { request ->
                when (request.url.toString()) {
                    EmbeddingConstants.EMBED_URL ->
                        respond(
                            content = embedResponseSuccessJson,
                            status = HttpStatusCode.OK,
                            headers = headersOf("Content-Type" to listOf(ContentType.
                                Application.Json.toString())),
                        )

                    else -> error("Unhandled ${request.url}")
                }
            }

        val client = HttpClient(engine)
        TemplateService.httpClient = client

        val response = TemplateService.embed("Test text", "Test name")
        assertEquals("success", response.status)
        val floats = response.embedding
```

```
        assertEquals(listOf(0.1f, 0.2f, 0.3f), floats)
    }
```

Listing 185: Python Service Test

These tests ensure:

- The communication protocol works as expected
- Request and response formats are correctly understood by both sides
- Error conditions are properly handled
- The integration is robust and reliable

The Python integration provides a powerful foundation for the Embeddings Module, leveraging the strengths of both Python and Kotlin to deliver advanced machine learning capabilities within a robust application framework. This hybrid approach demonstrates how modern applications can transcend language boundaries to utilize the best tools for each task.

7.4.7 API Endpoints

The Embeddings Module exposes several REST API endpoints that enable embedding generation, storage, and semantic search capabilities. These endpoints form the interface between the Python microservice and the Kotlin application.

Endpoint Overview The module exposes three primary endpoints:

- `/embed`: Generates vector embeddings for input text
- `/new`: Stores text data and its embedding for future search
- `/search`: Performs semantic search based on input embeddings and keywords

These endpoints are defined in the Flask application:

```
app = Flask(__name__)
CORS(app)

@app.route('/embed', methods=['POST'])
def embed_route():
    # Implementation...

@app.route('/new', methods=['POST'])
def new_template_route():
    # Implementation...

@app.route('/search', methods=['POST'])
def search_route():
    # Implementation...
```

Listing 186: Flask Endpoint Registration

Embed Endpoint The `/embed` endpoint generates vector embeddings for input text:

```
@app.route('/embed', methods=['POST'])
def embed_route():
    data = request.json
    if not "text" in data:
        return jsonify({
            "status": "error",
            "message": "No prompt provided"
        })
    text = data["text"]
    embedding = emb.embed(text)
    if not embedding:
        return jsonify({"status": "error", "message": f"Error embedding: {text}"})
    return jsonify({"status": "success", "embedding": embedding})
```

Listing 187: Embed Endpoint Implementation

Request Format

```
POST /embed
Content-Type: application/json

{
  "text": "The text to generate embeddings for",
  "name": "Optional identifier for the text"
}
```

Listing 188: Embed Endpoint Request

Response Format

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "success",
  "embedding": [0.123, 0.456, ...]  // Array of 384 floating-point values
}
```

Listing 189: Successful Embed Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "error",
  "message": "Error message describing the issue"
}
```

Listing 190: Error Embed Response

Error Conditions The endpoint may return error responses for several conditions:

- Missing `text` field in the request
- Invalid input type (not a string)
- Failure in the embedding generation process

New Template Endpoint The `/new` endpoint stores text data and its embedding for future search:

```python
@app.route('/new', methods=['POST'])
def new_template_route():
    data = request.json
    if not "text" in data:
        return jsonify({"status": "error", "message": "No prompt provided"})

    jsonld = data["text"]
    name = data["name"]
    embedding = emb.embed(jsonld)
    vector_success = vs.store_embedding(name, vector = np.array(embedding))
    keyword_success = pi.store_jsonld(name, json.loads(jsonld))
    if not vector_success or not keyword_success:
        return jsonify({"status": "error", "message": f"Failed to store template: Vector DB: {
            vector_success}, Keyword DB: {keyword_success}"})

    # Save data to disk after successful storage
    save_data(vs.index, vs.store)

    return jsonify({"status": "success", "message": "New template stored successfully!"})
```

Listing 191: New Template Endpoint Implementation

Request Format

```
POST /new
Content-Type: application/json

{
  "text": "JSON-LD or text content to store",
  "name": "Unique identifier for the content"
}
```

Listing 192: New Template Endpoint Request

Response Format

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "success",
  "message": "New template stored successfully!"
}
```

Listing 193: Successful New Template Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "error",
  "message": "Failed to store template: Vector DB: false, Keyword DB: true"
}
```

Listing 194: Error New Template Response

Processing Steps The endpoint performs several operations:

1. Extracts the text and name from the request
2. Generates a vector embedding for the text
3. Stores the embedding in the vector database (FAISS)
4. Stores the text in the keyword database (Pyserini)
5. Persists the updated databases to disk
6. Returns a success or error response

Error Conditions The endpoint may return error responses for several conditions:

- Missing `text` or `name` fields in the request
- Invalid JSON-LD format (if applicable)
- Failure in the embedding generation process
- Failure to store in the vector database
- Failure to store in the keyword database

Search Endpoint The `/search` endpoint performs semantic search based on input embeddings and keywords:

```
@app.route('/search', methods=['POST'])
def search_route():
    data = request.json
    if not "embedding" in data:
        return jsonify({"status": "error", "message": "No embedding provided for semantic
            search!"})
    if not "query" in data:
        return jsonify({"status": "error", "message": "No query provided for keyword search!"
            })

    top_k = data.get("top_k", 5)
    vector_results = vs.semantic_search(data["embedding"], top_k=top_k)
    keyword_results = pi.keyword_search(data["query"], top_k=top_k)
    results = list(set(vector_results + keyword_results))
    return jsonify({"status": "success", "matches": results})
```

Listing 195: Search Endpoint Implementation

Request Format

```
POST /search
Content-Type: application/json

{
  "embedding": [0.123, 0.456, ...],  // Vector embedding for semantic search
  "query": "Search query text",       // Text query for keyword search
  "top_k": 5                         // Optional, number of results to return
}
```

Listing 196: Search Endpoint Request

Response Format

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "success",
  "matches": ["id1", "id2", "id3"]   // Array of matching document IDs
}
```

Listing 197: Successful Search Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "status": "error",
  "message": "No embedding provided for semantic search!"
}
```

Listing 198: Error Search Response

Processing Steps The endpoint performs several operations:

1. Extracts the embedding vector, query text, and optional parameters from the request
2. Performs vector-based semantic search using the embedding
3. Performs keyword-based search using the query text
4. Combines and deduplicates the results from both search methods
5. Returns the combined result set

Error Conditions The endpoint may return error responses for several conditions:

- Missing `embedding` field in the request
- Missing `query` field in the request
- Invalid embedding format (not an array of numbers)
- Failures in the search process

Kotlin Client Integration On the Kotlin side, these endpoints are accessed through the `TemplateService` class:

```
object TemplateService {
    internal var httpClient = HttpClient(CIO)

    suspend fun embed(
        data: String,
        name: String,
    ): TemplateEmbedResponse {
        val payload = mapOf("text" to data, "name" to name)
        val response =
            httpClient
                .post(EmbeddingConstants.EMBED_URL) {
                    header(HttpHeaders.ContentType, "application/json")
                    setBody(Json.encodeToString(payload))
                }
        val responseText = response.bodyAsText()
        return runCatching { Json.decodeFromString<TemplateEmbedResponse>(responseText) }.
            getOrElse {
            error("Failed to store template!")
        }
    }

    suspend fun storeTemplate(
        fileURI: String,
        data: String,
    ): StoreTemplateResponse {
        // Implementation...
    }

    suspend fun search(
        embedding: List<Float>,
        query: String,
    ): TemplateSearchResponse {
```

```
        // Implementation...
    }
}
```

Listing 199: Kotlin Client Integration

The `EmbeddingConstants` object defines the URLs for these endpoints:

```
internal object EmbeddingConstants {
    private const val EMBEDDING_SERVICE_URL = "http://localhost:7000/"
    const val EMBED_URL = "$EMBEDDING_SERVICE_URL/embed"
    const val EMBED_AND_STORE_URL = "$EMBEDDING_SERVICE_URL/new"
    const val SEMANTIC_SEARCH_URL = "$EMBEDDING_SERVICE_URL/search"
}
```

Listing 200: Embedding Constants

API Error Handling The API implements consistent error handling across all endpoints:

- **Input Validation**: Each endpoint validates required fields and input formats
- **Structured Error Responses**: Error responses follow a consistent format with a status field and error message
- **Error Propagation**: Errors are properly propagated from the Python service to the Kotlin client
- **Exception Handling**: Both sides implement proper exception handling to prevent crashes
- **Status Codes**: While the API uses HTTP 200 for both success and error responses, it differentiates between them using the status field in the response body

API Testing The API endpoints are thoroughly tested to ensure correct behavior:

```
@Test
fun `Test embed no text`(client):
    """
    Test the /embed route with no "text" key in the payload.
    """
    response = client.post("/embed", json={})
    data = response.get_json()
    assert data["status"] == "error"
    assert data["message"] == "No prompt provided"

@Test
fun `Test embed success`(client, monkeypatch):
    """
    Test the /embed route with a successful embedding response.
    """
    monkeypatch.setattr(information_retrieval.embedding_service.emb, "embed", lambda text:
        [0.1, 0.2, 0.3])

    payload = {"text": "test prompt"}
    response = client.post("/embed", json=payload)
    data = response.get_json()
    assert data["status"] == "success"
    assert "embedding" in data
    assert data["embedding"] == [0.1, 0.2, 0.3]
```

Listing 201: API Endpoint Test

The API endpoints provide a clean, well-defined interface between the Python microservice and the Kotlin application, enabling seamless integration of advanced machine learning capabilities into the application while maintaining a clean separation of concerns.

7.4.8 Storage

The Embeddings Module implements a comprehensive storage strategy that spans both the Python microservice and the Kotlin application. This multi-layered approach ensures robust persistence of embedding data while maintaining efficient search capabilities.

Storage Components The storage system consists of several components:

- **FAISS Index**: Stores vector embeddings for efficient similarity search
- **Vector-Document Mapping**: Maps vector indices to document identifiers

- **Lucene Index**: Stores text data for keyword-based search
- **Database Records**: Stores metadata about templates and their file locations

This distributed storage approach leverages the strengths of each storage technology for its specific purpose.

FAISS Index Storage The FAISS index is managed by the Python microservice:

```python
import faiss
import pickle
import os

# Define file paths
FAISS_FILE = os.path.join(BASE_DIR, "faiss.index")
MAPPINGS_FILE = os.path.join(BASE_DIR, "mappings.pkl")

def load_data():
    """Retrieve persisted data from disk."""
    try:
        index = faiss.read_index(FAISS_FILE)
        if index is None:
            index = faiss.IndexFlatIP(VECTOR_DIMENSION)
    except Exception as e:
        print(f"Could not load FAISS index from {FAISS_FILE} ({e}). Creating new index.")
        index = faiss.IndexFlatIP(VECTOR_DIMENSION)

    try:
        with open(MAPPINGS_FILE, "rb") as f:
            vector_store = pickle.load(f)
        if vector_store is None:
            vector_store = {}
    except Exception as e:
        print(f"Could not load mapping from {MAPPINGS_FILE} ({e}). Creating new mapping.")
        vector_store = {}

    return index, vector_store

def save_data(index, vector_store):
    """Save data into disk for persistence."""
    faiss.write_index(index, FAISS_FILE)
    with open(MAPPINGS_FILE, "wb") as f:
        pickle.dump(vector_store, f)
```

Listing 202: FAISS Index Storage

Key aspects of the FAISS storage:

- The index is stored as a binary file using FAISS's native serialization
- The vector-to-document mapping is stored as a Python dictionary serialized with pickle
- The storage system handles file I/O errors gracefully, creating new structures when needed
- Both index and mapping are loaded into memory during initialization for efficient operation

Lucene Index Storage The Lucene index for keyword search is managed by the Pyserini component:

```python
import os
from pyserini.index.lucene import LuceneIndexer
from information_retrieval.data_handler import LUCENE_INDEX_DIR

def store_jsonld(name:str, data: dict) -> bool:
    """Stores JSON-LD metadata and indexes it with Pyserini."""
    if not isinstance(data, dict):
        return False

    # Ensure the directory exists
    os.makedirs(LUCENE_INDEX_DIR, exist_ok=True)

    # Create or append to the index
    indexer = LuceneIndexer(LUCENE_INDEX_DIR)
    indexer.add_doc_dict({
        "id": name,
        "contents": json.dumps(data),
    })

    indexer.close()
    return True
```

Listing 203: Lucene Index Storage

Key aspects of the Lucene storage:

- The index is stored as a directory of files in the Lucene format
- Each document is stored with an ID field and a contents field
- The contents field contains the serialized JSON data
- The indexer properly manages resources, closing the index after use

Database Integration On the Kotlin side, template metadata is stored in the application database:

```
/**
 * Creates a new template and stores it in the database.
 *
 * @param fileURI The URI of the template file
 * @return template id or null
 */
suspend fun createTemplate(fileURI: String): String? {
    val templateId = UUID.randomUUID().toString()
    val template = Template(id = templateId, fileURI = fileURI)
    val result =
        runCatching {
            DatabaseManager.templateRepository().saveTemplateToDB(template)
        }

    return if (result.isSuccess) {
        templateId
    } else {
        null
    }
}
```

Listing 204: Database Integration

This integration:

- Creates a new template record with a unique identifier
- Stores the file URI in the database
- Uses the database module's repository pattern for data access
- Handles errors gracefully, returning null if the operation fails

Storage Initialization The storage components are initialized during application startup:

```
# Python side (Flask app)
first_request = True
@app.before_request
def startup_once():
    global first_request
    if first_request:
        vs.index, vs.store = load_data()
        first_request = False

# Kotlin side (TemplateService)
object TemplateService {
    internal var httpClient = HttpClient(CIO)
    // ...
}
```

Listing 205: Storage Initialization

This initialization ensures:

- Storage components are ready before any requests are processed
- Resources are loaded only once to prevent redundant initialization
- Error handling is in place to recover from initialization failures

Storage Persistence The system ensures data persistence through several mechanisms:

```
# Save data after successful storage
@app.route('/new', methods=['POST'])
def new_template_route():
```

```
    # Process request...

    # Save data to disk after successful storage
    save_data(vs.index, vs.store)

    return jsonify({"status": "success", "message": "New template stored successfully!"})

# Save data on shutdown
if __name__ == '__main__':
    atexit.register(save_data, vs.index, vs.store)
    app.run(host="0.0.0.0", port=7000)
```

Listing 206: Storage Persistence

This approach ensures:

- Data is saved to disk after each storage operation
- Data is also saved when the application shuts down
- Even in case of unexpected termination, the system will recover on next startup

File Structure The storage system uses a well-defined file structure:

```
BASE_DIR = str(pathlib.Path(__file__).parent.parent.absolute())

# Vector search files
FAISS_FILE = os.path.join(BASE_DIR, "faiss.index")
MAPPINGS_FILE = os.path.join(BASE_DIR, "mappings.pkl")

# Keyword search files
LUCENE_INDEX_DIR = os.path.join(BASE_DIR, "jsonld_index")
```

Listing 207: Storage File Structure

This structure:

- Keeps all storage files within the application directory
- Uses appropriate file formats for each storage type
- Maintains a clean separation between different storage components
- Uses absolute paths to avoid working directory issues

Backup and Recovery The storage system implements basic backup and recovery mechanisms:

```
def load_data():
    """Retrieve persisted data from disk."""
    try:
        index = faiss.read_index(FAISS_FILE)
        if index is None:
            index = faiss.IndexFlatIP(VECTOR_DIMENSION)
    except Exception as e:
        print(f"Could not load FAISS index from {FAISS_FILE} ({e}). Creating new index.")
        index = faiss.IndexFlatIP(VECTOR_DIMENSION)

    # Similar recovery for other components...
```

Listing 208: Recovery Mechanism

If data files are corrupted or missing, the system will:

- Log the error for diagnostic purposes
- Create new, empty storage structures
- Continue operation with the new structures
- Rebuild indices as new data is added

Data Model The storage system is built around several key data models:

```
data class Template(
    val id: String,
    val fileURI: String
)
```

Listing 209: Template Data Model

```kotlin
@Serializable
data class TemplateEmbedResponse(
    val status: String,
    val embedding: List<Float> = emptyList(),
)
```

Listing 210: Embedding Response Model

```kotlin
@Serializable
data class TemplateSearchResponse(
    val status: String,
    val matches: List<String> = emptyList(),
)
```

Listing 211: Search Response Model

These models ensure:

- Consistent data representation across storage components
- Clear interfaces between storage and application logic
- Proper serialization and deserialization of data
- Type safety through Kotlin's strong typing

The multi-layered storage approach of the Embeddings Module provides a robust foundation for persistent, high-performance semantic search capabilities. By leveraging specialized storage technologies for different aspects of the data, the system achieves both flexibility and efficiency while maintaining data integrity.

7.4.9 Security

The Embeddings Module implements several security measures to protect data and ensure proper operation. This section outlines the security considerations and implementations within the module.

Input Validation The module implements thorough input validation to prevent security issues:

```python
@app.route('/embed', methods=['POST'])
def embed_route():
    data = request.json
    if not "text" in data:
        return jsonify({
            "status": "error",
            "message": "No prompt provided"
        })
    text = data["text"]
    embedding = emb.embed(text)
    if not embedding:
        return jsonify({"status": "error", "message": f"Error embedding: {text}"})
    return jsonify({"status": "success", "embedding": embedding})
```

Listing 212: Input Validation in Python Service

```kotlin
suspend fun embed(
    data: String,
    name: String,
): TemplateEmbedResponse {
    // Input validation
    if (data.isBlank()) {
        error("Cannot embed empty data")
    }

    val payload = mapOf("text" to data, "name" to name)
    // Rest of implementation...
}
```

Listing 213: Input Validation in Kotlin Client

Key validation measures:

- Checking for required fields in requests
- Validating data types (string vs. non-string)
- Preventing empty or blank inputs
- Validating JSON data structure

Type Safety The Kotlin client leverages Kotlin's strong type system to prevent type-related security issues:

```kotlin
@Serializable
data class TemplateEmbedResponse(
    val status: String,
    val embedding: List<Float> = emptyList(),
)

@Serializable
data class TemplateSearchResponse(
    val status: String,
    val matches: List<String> = emptyList(),
)
```

Listing 214: Type Safety in Kotlin

This approach:

- Defines clear data structures for requests and responses
- Uses Kotlin's serialization framework for type-safe JSON handling
- Prevents type confusion and related vulnerabilities
- Detects serialization/deserialization errors early

Error Handling and Information Exposure The module implements careful error handling to prevent information leakage:

```
// Python side
def keyword_search(query: str, top_k: int = 5):
    """Performs a keyword-based search using Pyserini (BM25 ranking)."""
    if not os.path.exists(LUCENE_INDEX_DIR) or not os.listdir(LUCENE_INDEX_DIR):
        return []

    try:
        searcher = LuceneSearcher(LUCENE_INDEX_DIR)
        hits = searcher.search(query, k=top_k)

        results = []
        for hit in hits:
            results.append(hit.docid)

        return results
    except Exception as e:
        print(f"Error during keyword search: {e}")
        return []

// Kotlin side
suspend fun storeTemplate(
    fileURI: String,
    data: String,
): StoreTemplateResponse {
    val templateId =
        TemplateStorageService.createTemplate(fileURI)
            ?: error("Failed to store template!")

    val remoteResponse = storeTemplateEmbedding(templateId, data)
    val success = remoteResponse.status == HttpStatusCode.OK
    return if (success) {
        Json.decodeFromString<StoreTemplateResponse>(remoteResponse.bodyAsText()).copy(id =
            templateId)
    } else {
        error("Failed to store template!")
    }
}
```

Listing 215: Secure Error Handling

Key security aspects:

- Errors are caught and logged without exposing sensitive details
- Generic error messages are returned to clients
- Detailed error information is logged for debugging
- The system gracefully handles failures without crashing

File System Security The module implements measures to protect file system operations:

```
# Define file paths relative to the base directory
BASE_DIR = str(pathlib.Path(__file__).parent.parent.absolute())
FAISS_FILE = os.path.join(BASE_DIR, "faiss.index")
MAPPINGS_FILE = os.path.join(BASE_DIR, "mappings.pkl")
LUCENE_INDEX_DIR = os.path.join(BASE_DIR, "jsonld_index")

def load_data():
    """Retrieve persisted data from disk."""
    try:
        # File operations...
    except Exception as e:
        print(f"Could not load FAISS index from {FAISS_FILE} ({e}). Creating new index.")
        # Recovery operations...

    # Ensure the Lucene index directory exists
    os.makedirs(LUCENE_INDEX_DIR, exist_ok=True)

    return index, vector_store
```

Listing 216: File System Security

Key security measures:

- Using absolute paths to prevent path traversal issues
- Careful handling of file operations with proper error checking
- Creating directories with appropriate permissions
- Avoiding user-controlled paths in file operations

Cross-Origin Resource Sharing (CORS) The Python microservice configures CORS to control which origins can access the API:

```
from flask import Flask, jsonify, request
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
```

Listing 217: CORS Configuration

In a production environment, CORS should be configured more restrictively:

```
# Production CORS Configuration
CORS(app, resources={r"/*": {"origins": "https://yourapporigin.com"}})
```

Listing 218: Production CORS Configuration

This would:

- Restrict API access to specific origins
- Prevent cross-site request forgery (CSRF) attacks
- Control which HTTP methods are allowed
- Specify which headers can be used

Authentication and Authorization While the Embeddings Module itself does not implement authentication, it is designed to be used behind appropriate authentication middleware:

```
// In the main application routing
routing {
    authenticate("jwt-verifier") {
        // Protected routes that use the embeddings module
        post("/api/templates/search") {
            // Extract user information from authentication
            val principal = call.principal<JWTPrincipal>()
            val userId = principal?.payload?.getClaim("sub")?.asString()

            // Use the embeddings module with user context
            val request = call.receive<SearchRequest>()
            val embedResponse = TemplateService.embed(request.query, "search-${UUID.randomUUID
                ()}")
            val searchResponse = TemplateService.search(embedResponse.embedding, request.query
                )

```

```
            call.respond(searchResponse)
        }
    }
}
```

Listing 219: Integration with Authentication

This integration ensures:

- Only authenticated users can access embedding functionality
- User context can be incorporated into search operations
- Authorization checks can be performed before operations
- Audit trails can be maintained for embedding operations

Resource Protection The module implements measures to protect against resource exhaustion:

```
def semantic_search(embedding: list, top_k: int):
    """
    Performs a semantic search using the provided embedding vector.

    Args:
        embedding: The query vector for similarity search
        top_k: Maximum number of results to return

    Returns:
        A list of document IDs sorted by similarity to the query vector
    """
    # Limit top_k to a reasonable value to prevent resource exhaustion
    top_k = min(top_k, 100)

    base = np.array(embedding, dtype=np.float32).reshape(1, -1)
    if index.ntotal == 0:
        return []

    distances, indices = index.search(base, top_k)
    results = [store[idx] for idx in indices[0] if idx in store]
    return results
```

Listing 220: Resource Protection

Key protection measures:

- Limiting the number of results that can be requested
- Ensuring efficient operation even with large indices
- Proper resource cleanup after operations
- Graceful handling of empty or invalid inputs
- Prevention of excessive memory usage

Data Sanitization The module implements data sanitization to prevent injection attacks:

```
def store_jsonld(name:str, data: dict) -> bool:
    """Stores JSON-LD metadata and indexes it with Pyserini."""
    if not isinstance(data, dict):
        return False

    # Create or append to the index
    indexer = LuceneIndexer(LUCENE_INDEX_DIR)
    indexer.add_doc_dict({
        "id": name,
        "contents": json.dumps(data),
    })

    indexer.close()
    return True
```

Listing 221: Data Sanitization

Key sanitization measures:

- Validating input types to prevent injection attacks
- Using proper serialization methods (e.g., `json.dumps`) to escape special characters
- Avoiding the use of raw inputs in critical operations
- Controlling how data is stored and indexed

Network Security The HTTP communication between the Kotlin client and Python microservice should be secured in a production environment:

```
object TemplateService {
    internal var httpClient = HttpClient(CIO) {
        install(HttpTimeout) {
            requestTimeoutMillis = 10000
            connectTimeoutMillis = 5000
        }
        expectSuccess = true
    }

    // Methods...
}
```

Listing 222: HTTP Client Configuration

In a production environment, additional security measures should be implemented:

```
object TemplateService {
    internal var httpClient = HttpClient(CIO) {
        install(HttpTimeout) {
            requestTimeoutMillis = 10000
            connectTimeoutMillis = 5000
        }
        expectSuccess = true

        // Add TLS configuration for HTTPS
        engine {
            https {
                // Configure TLS
                serverCertificateVerification = true
                // Specify trusted certificates if needed
            }
        }

        // Add authentication if needed
        install(Auth) {
            // Configure authentication method
        }
    }

    // Methods...
}
```

Listing 223: Production HTTP Client Configuration

These measures would:

- Encrypt traffic between the client and server using TLS
- Verify server certificates to prevent man-in-the-middle attacks
- Implement timeouts to prevent resource exhaustion from hanging connections
- Add authentication to prevent unauthorized access to the microservice

Error Logging and Monitoring The module implements logging for security events:

```
try:
    # Operation...
except Exception as e:
    print(f"Error during keyword search: {e}")
    return []
```

Listing 224: Security Logging

In a production environment, more sophisticated logging should be implemented:

```
import logging

logger = logging.getLogger(__name__)

try:
    # Operation...
except Exception as e:
    logger.error(f"Security error: {e}", exc_info=True)
    return []
```

Listing 225: Production Security Logging

This would:

- Create structured logs for security events
- Include stack traces for debugging
- Allow integration with centralized logging systems
- Enable alerts for security incidents

Dependency Security The module manages dependencies to prevent security vulnerabilities:

```
# requirements.txt
faiss-cpu==1.10.0
numpy==1.25.2
flask==3.1.0
flask-cors==5.0.0
sentence-transformers==3.4.1
# Other dependencies...
```

Listing 226: Python Dependencies

Key dependency security practices:

- Pinning dependency versions to prevent unexpected changes
- Using well-maintained and actively supported libraries
- Regularly updating dependencies to include security patches
- Minimizing the number of dependencies to reduce the attack surface

Security Testing The module includes tests for security-related functionality:

```
@Test
fun `Test embed with invalid input`(client):
    """
    Test the /embed route with invalid input types.
    """
    response = client.post("/embed", json={"text": 123})  # Non-string input
    data = response.get_json()
    assert data["status"] == "error"
```

Listing 227: Security Testing

Additional security testing should include:

- Penetration testing of the API endpoints
- Fuzzing to find input handling vulnerabilities
- Static code analysis to identify potential security issues
- Regular security audits of the codebase

Docker Security The module implements basic Docker security in its Dockerfile:

```
FROM python:3.10-slim

# Create and set working directory
WORKDIR /app

# Copy requirements first to leverage Docker cache
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Set Python path
ENV PYTHONPATH=/app

# Command to run the application
CMD ["python3.10", "-m", "information_retrieval"]

EXPOSE 7000
```

Listing 228: Docker Security

For production deployments, additional security measures should be considered:

- Running the container as a non-root user

- Implementing read-only file systems where possible
- Using security scanning tools for container images
- Setting resource limits to prevent denial-of-service attacks
- Implementing network policies to control container communication

The security measures implemented in the Embeddings Module provide a strong foundation for protecting data and preventing common vulnerabilities. By following security best practices in input validation, error handling, resource protection, and other areas, the module maintains a robust security posture while delivering powerful embedding and search capabilities.

7.4.10 Best Practices and Usage Guidelines

This section provides best practices and guidelines for developers working with the Embeddings Module. Following these recommendations will ensure efficient, secure, and maintainable usage of the module's capabilities.

Embedding Generation When generating embeddings, follow these practices:

- **Clean Text Input**: Remove unnecessary formatting, markup, and noise from text before generating embeddings:

```kotlin
    // Remove HTML tags and normalize whitespace
    val cleanText = Jsoup.parse(rawText).text().trim()

    // Generate embedding for the cleaned text
    val embeddingResponse = TemplateService.embed(cleanText, name)
```

Listing 229: Text Cleaning Example

- **Batch Similar Requests**: When generating multiple embeddings, batch similar requests to reduce overhead:

```kotlin
    // Instead of this:
    val embeddings = texts.map { text ->
        TemplateService.embed(text, "id-${UUID.randomUUID()}")
    }

    // Consider this approach for similar texts:
    val combinedText = texts.joinToString("\n\n---\n\n")
    val combinedEmbedding = TemplateService.embed(combinedText, "batch-${UUID.randomUUID()}"
        )
    // Then process the combined embedding as needed
```

Listing 230: Batching Example

- **Cache Embeddings**: Cache embeddings for frequently used texts to reduce computational load:

```kotlin
    // Simple in-memory cache
    private val embeddingCache = ConcurrentHashMap<String, List<Float>>()

    suspend fun getEmbedding(text: String): List<Float> {
        val cacheKey = text.hashCode().toString()

        return embeddingCache.getOrPut(cacheKey) {
            val response = TemplateService.embed(text, "cached-${UUID.randomUUID()}")
            response.embedding
        }
    }
```

Listing 231: Embedding Caching

- **Normalize Text Length**: For more consistent embeddings, normalize text length before processing:

```kotlin
    fun normalizeText(text: String, maxLength: Int = 512): String {
        return text.take(maxLength)
    }
```

Listing 232: Text Length Normalization

Template Storage When storing templates with embeddings, follow these practices:

- **Use Meaningful Identifiers**: Choose descriptive identifiers for templates to aid in debugging and maintenance:

```
// Poor practice:
val templateId = UUID.randomUUID().toString()

// Better practice:
val templateType = "login-form"
val templateId = "$templateType-${UUID.randomUUID()}"
```

Listing 233: Meaningful Template IDs

- **Include Metadata**: Store relevant metadata with templates to provide context:

```
val templateData = """
{
    "@context": "https://schema.org/",
    "@type": "SoftwareSourceCode",
    "name": "LoginForm",
    "description": "A responsive login form with email, password, and Google OAuth.",
    "programmingLanguage": {
        "@type": "ComputerLanguage",
        "name": "TypeScript"
    },
    "keywords": ["login", "authentication", "oauth"]
}
"""

val response = TemplateService.storeTemplate(fileURI, templateData)
```

Listing 234: Template with Metadata

- **Validate Template Data**: Ensure template data is valid before storage:

```
fun validateTemplate(template: String): Boolean {
    try {
        val json = Json.parseToJsonElement(template).jsonObject
        return json.containsKey("name") && json.containsKey("description")
    } catch (e: Exception) {
        return false
    }
}

if (validateTemplate(templateData)) {
    val response = TemplateService.storeTemplate(fileURI, templateData)
    // Process response...
} else {
    println("Invalid template data")
}
```

Listing 235: Template Validation

- **Handle Duplicate Templates**: Implement strategies for handling duplicate or similar templates:

```
suspend fun isDuplicate(newTemplate: String): Boolean {
    val embedding = TemplateService.embed(newTemplate, "duplicate-check")
    val searchResults = TemplateService.search(embedding.embedding, newTemplate)

    if (searchResults.matches.isEmpty()) {
        return false
    }

    // Check similarity threshold
    // Implementation depends on similarity metrics

    return false // Default to not a duplicate
}
```

Listing 236: Duplicate Detection

Semantic Search When performing semantic search operations, follow these practices:

- **Combine with Keyword Search**: Use both semantic and keyword search for comprehensive results:

```
suspend fun searchTemplates(query: String): List<String> {
    // Generate embedding for semantic search
    val embedResponse = TemplateService.embed(query, "search-${UUID.randomUUID()}")

```

```
    // Perform combined search
    val searchResponse = TemplateService.search(embedResponse.embedding, query)

    return searchResponse.matches
}
```

Listing 237: Combined Search Approach

- **Consider Result Diversity**: Implement strategies to ensure diverse search results:

```
suspend fun getDiverseResults(query: String, diversityFactor: Float = 0.3f): List<String
    > {
    val results = searchTemplates(query)

    // Implementation would filter results to ensure diversity
    // based on embeddings, categories, or other factors

    return results
}
```

Listing 238: Result Diversity

- **Tune Result Count**: Adjust the number of results based on the use case:

```
suspend fun searchWithContext(query: String, context: SearchContext): List<String> {
    // Determine appropriate result count based on context
    val resultCount = when (context) {
        SearchContext.BROWSING -> 20
        SearchContext.SPECIFIC_LOOKUP -> 5
        SearchContext.SUGGESTION -> 3
    }

    // Include result count in search parameters
    // Implementation details...

    return results
}
```

Listing 239: Result Count Tuning

- **Implement Pagination**: For large result sets, implement pagination:

```
suspend fun paginatedSearch(query: String, page: Int, pageSize: Int): SearchPage {
    // Generate embedding
    val embedResponse = TemplateService.embed(query, "search-${UUID.randomUUID()}")

    // Get total result count
    val allResults = TemplateService.search(embedResponse.embedding, query)

    // Apply pagination
    val start = page * pageSize
    val end = minOf(start + pageSize, allResults.matches.size)
    val pageResults = if (start < allResults.matches.size) {
        allResults.matches.subList(start, end)
    } else {
        emptyList()
    }

    return SearchPage(
        results = pageResults,
        totalCount = allResults.matches.size,
        currentPage = page,
        pageSize = pageSize,
        totalPages = (allResults.matches.size + pageSize - 1) / pageSize
    )
}
```

Listing 240: Search Pagination

Python Microservice Management For managing the Python microservice, follow these practices:

- **Implement Health Checks**: Add health check endpoints to monitor the microservice status:

```
@app.route('/health', methods=['GET'])
def health_check():
    # Check if essential components are working
```

```
    try:
        # Verify FAISS index
        if vs.index is None:
            return jsonify({"status": "error", "message": "FAISS index not initialized"
                })

        # Verify embedding model
        sample_text = "Health check"
        embedding = emb.embed(sample_text)
        if embedding is None:
            return jsonify({"status": "error", "message": "Embedding model not working"
                })

        return jsonify({"status": "healthy"})
    except Exception as e:
        return jsonify({"status": "error", "message": str(e)})
```

Listing 241: Health Check Implementation

- **Monitor Resource Usage**: Implement monitoring for resource usage:

```
@app.route('/metrics', methods=['GET'])
def metrics():
    import psutil
    import gc

    memory_usage = psutil.Process().memory_info().rss / (1024 * 1024)  # MB
    cpu_percent = psutil.Process().cpu_percent()
    index_size = vs.index.ntotal if vs.index else 0

    return jsonify({
        "memory_usage_mb": memory_usage,
        "cpu_percent": cpu_percent,
        "index_size": index_size,
        "python_objects": len(gc.get_objects())
    })
```

Listing 242: Resource Monitoring

- **Implement Graceful Shutdown**: Ensure data is saved during shutdown:

```
import signal
import sys

def signal_handler(sig, frame):
    print("Shutting down, saving data...")
    save_data(vs.index, vs.store)
    print("Data saved, exiting.")
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)
signal.signal(signal.SIGTERM, signal_handler)
```

Listing 243: Graceful Shutdown

- **Implement Backup Mechanisms**: Regularly back up the vector and text indices:

```
def backup_data():
    """Create a timestamped backup of index data."""
    import datetime
    import shutil

    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

    # Backup FAISS index
    backup_faiss = f"{FAISS_FILE}.{timestamp}.backup"
    shutil.copy2(FAISS_FILE, backup_faiss)

    # Backup mappings
    backup_mappings = f"{MAPPINGS_FILE}.{timestamp}.backup"
    shutil.copy2(MAPPINGS_FILE, backup_mappings)

    # Backup Lucene index
    backup_lucene = f"{LUCENE_INDEX_DIR}.{timestamp}.backup"
    shutil.copytree(LUCENE_INDEX_DIR, backup_lucene)

    return {
```

```
            "timestamp": timestamp,
            "backups": [backup_faiss, backup_mappings, backup_lucene]
        }
```

Listing 244: Backup Implementation

Error Handling Implement robust error handling throughout the Embeddings Module:

- **Use Structured Error Handling**: Wrap operations in structured error handling to provide clear error messages:

```
    suspend fun searchWithErrorHandling(query: String): Result<List<String>> {
        return runCatching {
            val embedResponse = TemplateService.embed(query, "search-${UUID.randomUUID()}")
            val searchResponse = TemplateService.search(embedResponse.embedding, query)
            searchResponse.matches
        }
    }

    // Usage
    val searchResult = searchWithErrorHandling(query)
    searchResult.fold(
        onSuccess = { matches ->
            // Process matches
        },
        onFailure = { error ->
            // Handle error
            log.error("Search failed", error)
            emptyList<String>()
        }
    )
```

Listing 245: Structured Error Handling

- **Implement Circuit Breaker**: Use a circuit breaker pattern to handle service unavailability:

```
    class EmbeddingServiceCircuitBreaker {
        private var failureCount = 0
        private var lastFailureTime = 0L
        private val maxFailures = 3
        private val resetTimeoutMs = 60000 // 1 minute

        fun recordSuccess() {
            failureCount = 0
        }

        fun recordFailure() {
            failureCount++
            lastFailureTime = System.currentTimeMillis()
        }

        fun isOpen(): Boolean {
            // Reset if enough time has passed
            if (System.currentTimeMillis() - lastFailureTime > resetTimeoutMs) {
                failureCount = 0
                return false
            }

            return failureCount >= maxFailures
        }
    }
```

Listing 246: Circuit Breaker Pattern

- **Implement Fallbacks**: Provide fallback mechanisms for when the embedding service is unavailable:

```
    suspend fun searchWithFallback(query: String): List<String> {
        return runCatching {
            // Try semantic search first
            val embedResponse = TemplateService.embed(query, "search-${UUID.randomUUID()}")
            val searchResponse = TemplateService.search(embedResponse.embedding, query)
            searchResponse.matches
        }.getOrElse { error ->
            // Log the error
            log.error("Semantic search failed, falling back to keyword search", error)
```

```kotlin

            // Fallback to simple keyword matching
            fallbackKeywordSearch(query)
        }
    }

    fun fallbackKeywordSearch(query: String): List<String> {
        // Simple keyword matching implementation
        // This would be a simpler, more reliable fallback
        // ...
    }
```

Listing 247: Fallback Implementation

- **Log Errors Appropriately**: Implement proper error logging for debugging and monitoring:

```kotlin
    suspend fun embed(data: String, name: String): TemplateEmbedResponse {
        return try {
            val payload = mapOf("text" to data, "name" to name)
            val response = httpClient.post(EmbeddingConstants.EMBED_URL) {
                // Request setup...
            }
            val responseText = response.bodyAsText()
            Json.decodeFromString<TemplateEmbedResponse>(responseText)
        } catch (e: Exception) {
            log.error("Failed to generate embedding: ${e.message}", e)
            TemplateEmbedResponse("error", emptyList())
        }
    }
```

Listing 248: Error Logging

Testing Implement thorough testing for the Embeddings Module:

- **Unit Test Core Functions**: Test individual functions in isolation:

```kotlin
    @Test
    fun `Test embedding normalization`() {
        val vector = floatArrayOf(3.0f, 4.0f).toList()
        val normalized = normalize(vector)

        val expectedLength = 1.0f
        val actualLength = kotlin.math.sqrt(normalized.sumOf { it * it })

        assertEquals(expectedLength, actualLength, 0.0001f)
    }
```

Listing 249: Unit Testing

- **Mock External Dependencies**: Use mocking for testing components that depend on external services:

```kotlin
    @Test
    fun `Test template search with mocked service`() = runBlocking {
        // Create mock service
        val mockHttpClient = mockk<HttpClient>()

        // Configure mock behavior
        coEvery {
            mockHttpClient.post(any(), any())
        } returns mockk {
            every { status } returns HttpStatusCode.OK
            every { bodyAsText() } returns """
                {"status":"success","matches":["template1","template2"]}
            """
        }

        // Set the mock client
        TemplateService.httpClient = mockHttpClient

        // Test the function
        val result = TemplateService.search(listOf(0.1f, 0.2f), "test query")

        // Verify results
        assertEquals("success", result.status)
        assertEquals(listOf("template1", "template2"), result.matches)
```

```
    }
```

Listing 250: Mock Testing

- **Integration Testing**: Test the interaction between components:

```
    @Test
    fun `Test end-to-end embedding and search`() = runBlocking {
        // Ensure the Python service is running
        val healthCheck = runCatching {
            HttpClient().get("http://localhost:7000/health")
        }.getOrNull()

        if (healthCheck == null) {
            println("Skipping integration test: Python service not available")
            return@runBlocking
        }

        // Perform an embedding operation
        val embedResponse = TemplateService.embed("Test template", "test-id")

        // Verify embedding generation
        assertEquals("success", embedResponse.status)
        assertTrue(embedResponse.embedding.isNotEmpty())

        // Perform a search operation
        val searchResponse = TemplateService.search(embedResponse.embedding, "Test template"
            )

        // Verify search results
        assertEquals("success", searchResponse.status)
        // Other assertions...
    }
```

Listing 251: Integration Testing

- **Test Edge Cases**: Ensure the system handles edge cases correctly:

```
    @Test
    fun `Test empty input handling`() = runBlocking {
        // Test with empty string
        val embedResponse = TemplateService.embed("", "empty-test")
        assertEquals("error", embedResponse.status)

        // Test with very long input
        val longInput = "a".repeat(10000)
        val longEmbedResponse = TemplateService.embed(longInput, "long-test")

        // Check truncation behavior
        // Assertions depend on the expected behavior
    }
```

Listing 252: Edge Case Testing

Following these best practices will ensure the Embeddings Module is used effectively, securely, and efficiently. These guidelines promote clean code organization, proper error handling, and robust implementation throughout the application.

7.5 Prompting Module

7.5.1 Overview

The Prompting module is responsible for handling user prompts, sanitizing inputs, constructing specialized prompts for Large Language Models (LLMs), and generating appropriate responses including functional code prototypes. This module orchestrates the entire workflow from receiving a raw user prompt to delivering a structured response with executable code.

The module serves as a critical bridge between user intentions and the LLM's capabilities, implementing a multi-step process that enhances the quality and safety of the generated outputs. It performs prompt sanitization, functional requirements extraction, template matching, and prototype generation, while incorporating security checks throughout the process.

Key functionalities include:

- Sanitizing user input to remove potentially malicious content

- Extracting functional requirements from user prompts
- Retrieving relevant code templates based on requirements
- Constructing specialized prompts for the LLM
- Processing LLM responses into structured outputs
- Ensuring security and safety of generated code
- Formatting responses for client consumption

The module is designed with a clean separation of concerns, with specialized components handling different aspects of the workflow, making it maintainable and extensible.

7.5.2 Architecture

The Prompting module is structured around several key components that work together to process user prompts and generate appropriate responses. The architecture follows a modular design with clear separation of concerns, allowing for easier maintenance and future extensions.

Component Interaction The module's components interact in a sequential workflow:

1. User prompt is received by `PromptingMain`
2. `SanitisationTools` cleans and validates the input
3. `PromptingTools` creates a specialized prompt for requirements extraction
4. `PrototypeInteractor` sends the prompt to the LLM and receives a response
5. `PromptingTools` processes the response to extract functional requirements
6. `TemplateInteractor` fetches relevant templates based on the requirements
7. `PromptingTools` creates a prototype prompt with requirements and templates
8. `PrototypeInteractor` sends the prompt to the LLM and receives the prototype
9. Security checks are performed on the generated code
10. `ResponseHandler` formats the final response for the client

Core Dependencies The module relies on several key external dependencies:

- **Kotlin Serialization**: For JSON processing and serialization
- **Ktor**: For HTTP client and server functionality
- **JSoup**: For HTML sanitization
- **Prototype module**: For interacting with the LLM services
- **Embeddings module**: For template similarity search
- **Database module**: For persistent storage of templates
- **Utils module**: For common utilities and environment configuration

Data Flow The data flow through the module is primarily linear, with the following key transformations:

1. Raw user prompt → Sanitized prompt with extracted keywords
2. Sanitized prompt → Functional requirements prompt → LLM response
3. LLM response → Structured requirements JSON
4. Requirements → Template query → Matching templates
5. Requirements + Templates → Prototype prompt → LLM response
6. LLM response → Security-checked, formatted server response

This architecture ensures that each component has a clear responsibility and that the data flows logically through the system, with appropriate transformations at each step.

7.5.3 Main Components

PromptingMain The `PromptingMain` class serves as the central orchestrator for the entire prompting workflow. It manages the multi-step process from receiving a user prompt to delivering a structured response with generated code.

```kotlin
class PromptingMain(
    private val model: String = EnvironmentLoader.get("OLLAMA_MODEL")
) {
    suspend fun run(userPrompt: String): ServerResponse
    private fun prototypePrompt(userPrompt: String, freqsResponse: JsonObject, templates: List
        <String>): String
    private fun promptLlm(prompt: String, options: OllamaOptions = OllamaOptions()):
        JsonObject
```

```
    private fun serverResponse(response: JsonObject): ServerResponse
    private fun onSiteSecurityCheck(llmResponse: LlmResponse)
}
```

Key Methods

run The primary entry point that executes the complete workflow:

1. Sanitizes the user prompt using `SanitisationTools`
2. Generates a specialized prompt for functional requirements
3. Makes a first LLM call to extract requirements
4. Fetches matching templates based on the requirements
5. Creates a comprehensive prototype prompt with requirements and templates
6. Makes a second LLM call to generate the final prototype
7. Validates and formats the response

prototypePrompt Creates a specialized prompt for generating a prototype based on:

- The original user prompt
- Extracted functional requirements
- Optional templates for suggested components

promptLlm Sends a prompt to the LLM and processes the response:

- Calls the LLM through `PrototypeInteractor`
- Formats the response using `PromptingTools`
- Throws exceptions for errors or null responses

serverResponse Extracts data from the LLM response and formats it into a structured server response:

- Creates a `ChatResponse` with the LLM's message
- Extracts prototype files if available
- Returns a combined `ServerResponse`

onSiteSecurityCheck Performs security validation on generated code:

- Checks each file for security issues
- Throws exceptions for unsafe code

Response Data Models The module defines several data classes for structuring responses:

```
@Serializable
data class ChatResponse(
    val message: String,
    val role: String = "LLM",
    val timestamp: String,
    val messageId: String
)

@Serializable
data class ServerResponse(
    val chat: ChatResponse,
    val prototype: PrototypeResponse? = null
)

@Serializable
data class PrototypeResponse(
    val files: JsonObject
)
```

These classes provide a consistent structure for responses to the client, combining both the textual explanation (`chat`) and the generated code files (`prototype`).

7.5.4 Input Sanitisation

SanitisationTools The `SanitisationTools` object is responsible for cleaning and securing user input. It performs extensive sanitization to prevent injection attacks, remove potentially harmful content, and ensure the input is safe for processing.

```
object SanitisationTools {
    const val MAX_PROMPT_LENGTH = 1000

    internal fun sanitisePrompt(prompt: String): SanitisedPromptResult
    private fun cleanPrompt(prompt: String): String
    private fun extractKeywords(prompt: String): List<String>
}
```

Key Methods

sanitisePrompt The main entry point for prompt sanitization that combines cleaning and keyword extraction:

- Calls `cleanPrompt` to sanitize the raw input
- Calls `extractKeywords` to identify significant terms
- Returns both in a structured `SanitisedPromptResult`

cleanPrompt Performs thorough sanitization on the raw input:

- Removes all HTML tags using JSoup
- Removes leading and trailing whitespace
- Replaces special characters and HTML entities
- Limits input to `MAX_PROMPT_LENGTH` characters
- Removes potentially malicious patterns like "ignore", "pretend", "disregard", etc.

extractKeywords Identifies significant terms in the sanitized prompt:

- Loads a predefined list of keywords from `KeywordLoader`
- Matches these keywords against the sanitized prompt
- Returns a list of matched keywords for further processing

SanitisedPromptResult The result of the sanitization process is encapsulated in a `SanitisedPromptResult` data class:

```
@Serializable
data class SanitisedPromptResult(
    val prompt: String,
    val keywords: List<String>
)
```

This class combines:

- The cleaned prompt text with potentially harmful content removed
- A list of extracted keywords for further processing or matching

Security Considerations The sanitization process implements several security measures:

- HTML sanitization to prevent XSS attacks
- Maximum length enforcement to prevent DoS attacks
- Pattern matching to detect prompt injection attempts
- Character filtering to remove special characters
- Entity replacement to normalize input

These measures collectively help ensure that the input is safe for processing by subsequent components and especially for sending to the LLM, where prompt injection could otherwise be a significant risk.

7.5.5 Prompt Engineering

PromptingTools The `PromptingTools` object is responsible for creating specialized prompts for LLM interactions and processing LLM responses. It constructs carefully engineered prompts tailored to specific purposes and handles the parsing and cleaning of JSON responses.

```
object PromptingTools {
    fun functionalRequirementsPrompt(prompt: String, keywords: List<String>): String
    fun prototypePrompt(userPrompt: String, requirements: String, templates: List<String>):
        String
    fun formatResponseJson(response: String): JsonObject
    private fun cleanLlmResponse(response: String): String
    fun String.removeEscapedQuotations(): String
    fun String.removeComments(): String
}
```

Key Methods

functionalRequirementsPrompt Creates a specialized prompt for extracting functional requirements:

- Constructs a system message with detailed instructions for the LLM
- Includes the user's sanitized prompt as input
- Incorporates extracted keywords for additional context
- Formats the entire prompt as a JSON array of message objects
- Returns a stringified JSON array ready to send to the LLM

The prompt instructs the LLM to respond with a structured JSON containing requirements and keywords. It provides detailed guidelines for crafting high-quality functional requirements, including specificity, measurability, and clarity.

prototypePrompt Creates a comprehensive prompt for generating a code prototype:

- Constructs a detailed system message with code generation rules
- Includes the original user prompt
- Incorporates the extracted functional requirements
- Adds available reference templates if any
- Specifies the expected response format and schema
- Returns a stringified JSON array of message objects

This method builds a prompt that guides the LLM to generate production-quality code based on the requirements and templates. It includes detailed specifications for the response format, technology stack, and code quality guidelines.

formatResponseJson Processes the raw LLM response into a structured JsonObject:

- Calls `cleanLlmResponse` to sanitize the raw response
- Attempts to parse the cleaned string as a JSON object
- Handles exceptions and provides error information
- Returns the parsed JsonObject for further processing

cleanLlmResponse Extracts and cleans a JSON object from an LLM response string:

- Finds the first opening '{' and last closing '}' brace
- Extracts the JSON object from the response
- Removes comments using `removeComments`
- Handles escaped quotations with `removeEscapedQuotations`
- Normalizes newlines and whitespace
- Returns a clean JSON string ready for parsing

String Extensions The object also provides string extension functions:

- `removeEscapedQuotations`: Replaces escaped quotes with regular quotes
- `removeComments`: Removes C-style, and Python-style comments from strings

Prompt Engineering Strategy The module employs a sophisticated prompt engineering strategy:

- **Two-step approach**: First extracting requirements, then generating code
- **Detailed instructions**: Providing clear guidelines for the LLM
- **Structured formats**: Specifying exact response formats
- **Context enrichment**: Including keywords and templates
- **Technology guidance**: Suggesting appropriate tech stacks
- **Quality guidelines**: Specifying code quality standards

This approach significantly improves the quality and relevance of the generated code by first establishing clear requirements and then using those requirements to drive the code generation process.

7.5.6 Template Management

TemplateInteractor The `TemplateInteractor` object manages the retrieval and storage of code templates. It provides an interface for fetching relevant templates based on a prompt and storing new templates for future use.

```
object TemplateInteractor {
    suspend fun fetchTemplates(prompt: String): List<String>
    suspend fun storeNewTemplate(templateID: String, templateCode: String, jsonLD: String):
        Boolean
    private suspend fun getTemplateContent(id: String): String?
}
```

Key Methods

fetchTemplates Retrieves templates that match a given prompt:

- Calls the `TemplateService` to create an embedding of the prompt
- Uses the embedding to search for matching templates
- Retrieves the content of each matched template
- Returns a list of template contents as strings

This method leverages semantic embeddings to find templates that are conceptually related to the prompt, even if they don't share exact keywords.

storeNewTemplate Stores a new template with its associated metadata:

- Stores the template code in the configured storage location
- Stores the JSON-LD metadata in a separate file
- Creates a database entry for the template
- Registers the template with the `TemplateService` for future searching
- Returns a boolean indicating success or failure

getTemplateContent A helper method that retrieves the content of a template by its ID:

- Looks up the template in the `TemplateStorageService`
- Gets the file URI for the template
- Retrieves the file content using `TemplateStorageUtils`
- Decodes the content to a string and returns it

TemplateStorageUtils The `TemplateStorageUtils` object provides utilities for template storage operations. It handles file operations for templates, including retrieving and storing template files in either local or remote storage.

```
object TemplateStorageUtils {
    suspend fun retrieveFileContent(templateHandle: String): ByteArray
    suspend fun storeFile(content: String, filePrefix: String, fileSuffix: String,
        storageConfig: StorageConfig): String
    private fun retrieveLocalFileContent(path: String): ByteArray
    private suspend fun retrieveRemoteFileContent(url: String): ByteArray
    private fun parseS3Url(url: String): Pair<String, String>

    data class StorageConfig(val path: String, val key: String, val bucket: String)
}
```

7.5.7 Key Methods

retrieveFileContent Retrieves file content from either local or remote storage:

- Checks the environment configuration to determine storage type
- Calls either `retrieveLocalFileContent` or `retrieveRemoteFileContent`
- Returns the file content as a ByteArray
- Throws `TemplateRetrievalException` if the file cannot be retrieved

storeFile Stores a file in either local or remote storage:

- Creates a temporary file with the provided content
- Stores the file in the appropriate location based on environment settings
- Returns the path to the stored file
- Cleans up the temporary file after storage

Storage Helpers The object includes several helper methods:

- `retrieveLocalFileContent`: Gets a file from local storage
- `retrieveRemoteFileContent`: Gets a file from remote storage (S3)
- `parseS3Url`: Extracts bucket and key from an S3 URL

StorageConfig A data class for configuring file storage options:

- `path`: The local filesystem path
- `key`: The filename or object key
- `bucket`: The S3 bucket name for remote storage

Template Integration Strategy Templates play a crucial role in the code generation process:

- They provide real-world code examples for the LLM to learn from
- They improve consistency and quality of generated code
- They enable reuse of common patterns and components
- They accelerate development by providing starting points

The module's approach to templates includes:

- Semantic matching to find relevant templates
- Including templates directly in prompts to the LLM
- Instructing the LLM to adapt and combine templates
- Storing templates with metadata for better searchability

This integration significantly enhances the quality and relevance of generated code by providing high-quality examples tailored to the user's requirements.

7.5.8 Response Handling

ResponseHandler The `ResponseHandler` object provides utilities for processing HTTP responses from prompting services and converting them into standardized application responses. It ensures consistent handling of both successful and failed responses.

```
object ResponseHandler {
    suspend fun handlePromptResponse(response: HttpResponse, call: ApplicationCall)
    private suspend fun handleSuccessResponse(prototypeResponse: HttpResponse, call:
        ApplicationCall)
    private suspend fun handleFailureResponse(prototypeResponse: HttpResponse, call:
        ApplicationCall)
    private fun createResponse(message: String): Response
}
```

Key Methods

handlePromptResponse The main entry point for handling HTTP responses:

- Checks if the response status indicates success
- Routes to either `handleSuccessResponse` or `handleFailureResponse`
- Ensures all responses are properly handled regardless of status

handleSuccessResponse Processes a successful HTTP response:

- Extracts the response body as text
- Creates a standardized `Response` object with the current timestamp
- Sends the response to the client via the provided `ApplicationCall`

handleFailureResponse Processes a failed HTTP response:

- Logs the error with the response status
- Creates a standardized `Response` object with an error message
- Includes both the status code and response body in the error message
- Sends the error response to the client

createResponse Creates a standardized `Response` object:

- Adds the current timestamp
- Includes the provided message
- Returns a consistent `Response` structure

Response Data Structure The module uses a simple `Response` data class for standardized responses:

```
@Serializable
data class Response(
    val time: String,
    val message: String
)
```

This structure provides:

- A timestamp indicating when the response was generated
- The main content of the response (success message or error details)

Error Handling Strategy The module implements a comprehensive error handling strategy:

- **Uniform response format**: Both successful and failed responses use the same structure
- **Detailed error information**: Error responses include both status codes and error messages
- **Graceful failure**: All errors are caught and converted to informative responses
- **Timestamp inclusion**: All responses include a timestamp for debugging and logging
- **Consistent client experience**: The client receives a structured response even in error cases

This approach ensures that clients can reliably process responses and handle errors appropriately, while also providing enough information for debugging and troubleshooting.

7.5.9 Keyword Management

KeywordLoader The `KeywordLoader` object is responsible for loading and providing access to the application's predefined keywords list. It uses lazy initialization to ensure the keywords file is read only once when first needed.

```
object KeywordLoader {
    private val keywords: List<String> by lazy { /* initialization logic */ }
    fun getKeywordsList(): List<String>
}
```

Key Features

Lazy Loading The keywords are loaded only when first requested:

- Improves application startup time
- Conserves memory if keywords aren't needed
- Ensures consistent state across the application

Resource-Based Configuration Keywords are loaded from a JSON resource file:

- Allows easy updates without code changes
- Supports environment-specific configurations
- Centralizes keyword management

Immutable Access The keywords are provided as an immutable list:

- Prevents accidental modifications
- Ensures thread safety
- Supports functional programming patterns

KeywordList The `KeywordList` class provides a container for deserializing the keywords JSON file:

```
@Serializable
data class KeywordList(
    @SerialName("keywords")
    private val keywordsIn: List<String>
) {
    val keywords: List<String> get() = Collections.unmodifiableList(keywordsIn)
    override fun toString(): String = "KeywordList(keywords=$keywordsIn)"
}
```

7.5.10 Key Features

Serialization Support The class supports JSON deserialization:

- Uses Kotlin Serialization annotations
- Maps the JSON "keywords" field to the internal list
- Provides clean serialization/deserialization

Encapsulation The class protects the keywords list:

- Uses a private field for internal storage
- Exposes an immutable view via a getter
- Prevents external modification of the list

Standard Data Class Features As a Kotlin data class, it provides:

- Automatic `equals()` and `hashCode()` implementations
- Copy functionality
- Component functions for destructuring

Keywords in Prompt Processing Keywords play a crucial role in the prompt processing pipeline:

- **Extraction**: `SanitisationTools` extracts keywords from user prompts
- **Enrichment**: Keywords enrich prompts sent to the LLM
- **Context**: They provide additional context for requirements generation
- **Template matching**: Keywords help find relevant templates

The predefined keywords list includes common software development terms, component names, and domain concepts. This list helps identify the user's intent and guide the LLM toward generating more relevant and appropriate responses.

For example, a user prompt containing the keyword "authentication" would be recognized, and this information would be passed to the LLM to indicate that authentication functionality should be considered in the generated requirements and code.

7.6 Prototype Module

7.6.1 Overview

The Prototype module is responsible for interfacing with Large Language Models (LLMs) to generate code prototypes based on user prompts. It serves as the bridge between user requirements and executable code, handling the communication with Ollama (a local LLM server), processing responses, and ensuring the security of generated code.

This module plays a crucial role in the application's code generation workflow by:

- Providing a clean interface for sending prompts to LLMs
- Processing and validating LLM responses
- Converting unstructured LLM outputs into structured file representations
- Implementing security checks to ensure generated code is safe

- Handling error conditions gracefully

The Prototype module is designed to be agnostic to the specific requirements of user prompts, focusing instead on the technical aspects of LLM communication and code generation. It works closely with the Prompting module, which handles the construction of specialized prompts, while the Prototype module focuses on the efficient and secure delivery of these prompts to the LLM and the processing of responses.

Key functionalities include:

- Sending prompts to Ollama with configurable parameters
- Handling streaming and non-streaming responses
- Converting LLM output to structured JSON representations
- Extracting code files from LLM responses
- Validating generated code for security issues
- Error handling for LLM communication issues

7.6.2 Architecture

The Prototype module follows a clean architectural pattern with clear separation of concerns, organized around several key components:

Core Components

- **PrototypeMain**: The entry point for the module, responsible for coordinating the prompt-response workflow
- **OllamaService**: Handles communication with Ollama, including sending requests and receiving responses
- **Security Package**: Contains utilities for validating and securing generated code
- **Data Models**: A collection of serializable data classes for structured data exchange

Component Interaction Flow The typical flow through the module follows these steps:

1. A client (typically the Prompting module) calls `PrototypeMain.prompt()` with a text prompt
2. `PrototypeMain` creates an `OllamaRequest` with the prompt and model information
3. `OllamaService` sends the request to the Ollama server and awaits a response
4. The response is received and converted into an `OllamaResponse` object
5. `PrototypeMain` extracts the response content and performs validation
6. If code is generated, security checks are performed on each file
7. The final structured response is returned to the caller

Dependencies The module relies on several external dependencies:

- **Ktor Client**: For HTTP communication with Ollama
- **Kotlinx Serialization**: For JSON serialization/deserialization
- **JTidy**: For HTML validation and cleaning
- **JSoup**: For HTML parsing and sanitization
- **Utils Module**: For environment configuration and common utilities

Error Handling Strategy The module implements a comprehensive error handling strategy using Kotlin's `Result` type and custom exceptions:

- `OllamaService` returns `Result<OllamaResponse>` to encapsulate success or failure
- `OllamaException` provides specific error information for Ollama-related issues
- `PromptException` handles issues with prompt processing or response parsing
- All network errors are caught and converted to appropriate exceptions

This architecture provides a robust foundation for LLM communication, with clear concerns separation, strong error handling, and a flexible design that can accommodate different models and security requirements.

7.6.3 Main Components

PrototypeMain The `PrototypeMain` class serves as the primary entry point for the Prototype module. It coordinates the communication with the LLM and handles the processing of responses.

```
class PrototypeMain(
    private val model: String,
) {
    suspend fun prompt(
        prompt: String,
        options: OllamaOptions,
    ): OllamaResponse?
}
```

Key Methods **prompt** The main method that sends a prompt to the language model and returns the generated response:

- Creates an `OllamaRequest` with the provided prompt and model identifier
- Configures options like temperature, top-k, and top-p parameters
- Calls `OllamaService` to send the request to the LLM
- Validates the response and throws an exception if the request fails
- Returns the `OllamaResponse` object or null if processing fails

```
suspend fun prompt(
    prompt: String,
    options: OllamaOptions,
): OllamaResponse? {
    val request = OllamaRequest(
        prompt = prompt,
        model = model,
        stream = false,
        options = options
    )
    val llmResponse = OllamaService.generateResponse(request)
    check(llmResponse.isSuccess) { "Failed to receive response from the LLM" }
    return llmResponse.getOrNull()
}
```

Configuration The `PrototypeMain` class is configured with:

- A model identifier string that specifies which language model to use
- Optional parameters through the `OllamaOptions` class for fine-tuning model behavior

The model identifier is typically loaded from environment variables using the `EnvironmentLoader`:

```
val prototypeMain = PrototypeMain(
    model = EnvironmentLoader.get("OLLAMA_MODEL")
)
```

Error Handling `PrototypeMain` implements a robust error handling strategy:

- Uses Kotlin's `check` function to validate responses
- Throws an `IllegalStateException` if the LLM response fails
- Preserves the original error information through the Result pattern
- Logs errors for debugging purposes

This approach ensures that failures are clearly communicated to the caller while preserving detailed error information for troubleshooting.

7.6.4 Ollama Service

OllamaService The `OllamaService` object is responsible for communication with the Ollama server, which provides access to local Large Language Models. It handles the HTTP communication, request formatting, and response parsing.

```
object OllamaService {
    private val jsonParser = Json { ignoreUnknownKeys = true }
    private const val OLLAMA_PORT = 11434
    private val OLLAMA_HOST = EnvironmentLoader.get("OLLAMA_HOST")
    private const val REQUEST_TIMEOUT_MILLIS = 600_000_000L
    private const val CONNECT_TIMEOUT_MILLIS = 30_000_000L
    private const val SOCKET_TIMEOUT_MILLIS = 600_000_000L
```

```

    var client = HttpClient(CIO) {
        install(HttpTimeout) {
            requestTimeoutMillis = REQUEST_TIMEOUT_MILLIS
            connectTimeoutMillis = CONNECT_TIMEOUT_MILLIS
            socketTimeoutMillis = SOCKET_TIMEOUT_MILLIS
        }
    }

    suspend fun isOllamaRunning(): Boolean
    suspend fun generateResponse(request: OllamaRequest): Result<OllamaResponse>
    private suspend fun callOllama(request: OllamaRequest): OllamaResponse
}
```

Key Methods **isOllamaRunning** Checks if the Ollama server is accessible:

- Sends a simple GET request to the Ollama server root endpoint
- Returns true if the server responds with a 200 OK status
- Returns false if any error occurs during the check
- Uses a try-catch block with a Result type for graceful error handling

generateResponse Sends a prompt to an LLM via Ollama and returns the generated response:

- First checks if Ollama is running using `isOllamaRunning()`
- If Ollama is not running, returns a failure Result with an appropriate message
- If Ollama is running, calls `callOllama()` to send the request
- Wraps the response in a Success Result or catches exceptions and returns them in a Failure Result
- Returns a `Result<OllamaResponse>` to encapsulate the success or failure state

callOllama Makes a call to Ollama and parses the response:

- Constructs the Ollama API URL using the configured host and port
- Sends a POST request with the OllamaRequest serialized to JSON
- Reads and parses the response using the Kotlinx JSON parser
- Throws exceptions for network errors or invalid JSON responses
- Returns a parsed `OllamaResponse` object

Configuration The service is configured with several key parameters:

- `OLLAMA_HOST`: The hostname or IP address where Ollama is running (loaded from environment)
- `OLLAMA_PORT`: The port number for the Ollama API (default 11434)
- Timeout parameters for HTTP requests:
  - Request timeout: Maximum time for the entire request (10 minutes)
  - Connect timeout: Maximum time to establish a connection (30 seconds)
  - Socket timeout: Maximum time for inactivity between data packets (10 minutes)

HTTP Client The service uses Ktor Client for HTTP communication:

- Uses the CIO engine for efficient asynchronous I/O
- Configures timeouts for handling long-running LLM operations
- Allows dependency injection for testing (the client can be replaced)

JSON Parsing The service uses Kotlinx Serialization for JSON handling:

- Configures the parser to ignore unknown keys for forward compatibility
- Uses a pretty printer for debugging output
- Handles serialization exceptions with clear error messages

7.6.5 Security

PrototypeSecurity The `PrototypeSecurity` package provides utilities for validating and securing generated code. It includes checks for potentially dangerous patterns and ensures that the generated code adheres to expected standards.

```
fun secureCodeCheck(
    code: String,
    language: String,
): Boolean

fun checkCodeSizeLimit(
    code: String,
    maxBytes: Int,
): Boolean

fun runCompilerCheck(
    code: String,
    language: String,
): Boolean

fun checkCssSyntax(cssCode: String): Boolean

fun checkHtmlSyntaxWithJTidy(htmlCode: String): Boolean

fun checkJavaScriptSyntax(jsCode: String): Boolean
```

Key Functions **secureCodeCheck** The main function that runs multiple security checks:

- Performs syntax/compile check using language-specific validators
- Returns true if all checks pass, false otherwise
- Provides detailed logging for failed checks

checkCodeSizeLimit Enforces a maximum code size in bytes:

- Converts the code to UTF-8 bytes and checks the size
- Returns true if the size is within the limit, false otherwise
- Helps prevent denial-of-service attacks through extremely large code

runCompilerCheck Performs a syntax check for different languages:

- Routes to language-specific checkers based on the provided language
- Currently supports JavaScript, CSS, and HTML
- Returns false for unsupported languages

Language-Specific Checkers The package includes specialized checkers for different languages:
checkHtmlSyntaxWithJTidy Validates HTML code using JTidy:

- Uses the JTidy library to parse and validate HTML
- Captures errors during parsing
- Returns true if no errors are found, false otherwise

checkCssSyntax Validates CSS using Stylelint:

- Creates a temporary file with the CSS code
- Runs the Stylelint tool on the file using Node.js
- Captures and logs any errors
- Returns true if the exit code is 0 (no errors), false otherwise

checkJavaScriptSyntax Validates JavaScript using Node.js:

- Creates a temporary file with the JavaScript code
- Runs Node.js with the –check flag to validate syntax
- Captures and logs any errors
- Returns true if the exit code is 0 (no errors), false otherwise

Security Strategy The module implements a multi-layered security approach:

- **Syntax validation**: Ensures code is syntactically correct
- **Size limits**: Prevents excessively large code
- **Commented-out blocklist**: Infrastructure for future pattern-based blocking
- **External tooling**: Leverages established tools like Node.js and JTidy
- **Process isolation**: Runs validation in separate processes where appropriate

This approach helps protect against various security risks, including:

- Malicious code injection
- Denial-of-service attacks
- Cross-site scripting (XSS)
- Command injection
- Resource exhaustion

7.6.6 Data Models

The Prototype module defines several serializable data classes for structured representation of requests, responses, and file content. These models ensure type safety and provide clear interfaces for data exchange.

OllamaRequest Represents a request to the Ollama API for generating text from a language model.

```
@Serializable
data class OllamaRequest(
    val model: String,
    val prompt: String,
    val stream: Boolean,
    val options: OllamaOptions = OllamaOptions(),
)
```

Fields

- `model`: The identifier of the language model to use (e.g., "llama2")
- `prompt`: The text prompt to send to the language model
- `stream`: Whether to stream the response or return it all at once
- `options`: Additional parameters for controlling model behavior (optional)

OllamaOptions Represents configuration options for an Ollama request to control model behavior.

```
@Serializable
data class OllamaOptions(
    val temperature: Double? = null,
    val top_k: Int? = null,
    val top_p: Double? = null,
    val num_predict: Int? = null,
    val stop: List<String>? = null,
)
```

Fields

- `temperature`: Controls randomness (higher = more random, lower = more deterministic)
- `top_k`: Limits vocabulary choices to the K most likely next tokens
- `top_p`: Uses nucleus sampling, selecting from tokens that comprise the top P probability mass
- `num_predict`: Maximum number of tokens to generate
- `stop`: List of strings to stop generation when encountered

OllamaResponse Represents a response from the Ollama API containing generated text.

```
@Serializable
data class OllamaResponse(
    val model: String,
    val created_at: String,
    val response: String,
    val done: Boolean,
    val done_reason: String,
)
```

Fields

- `model`: The model identifier that generated the response
- `created_at`: Timestamp when the response was created
- `response`: The generated text output from the model
- `done`: Whether the generation is complete
- `done_reason`: The reason for completion (e.g., "stop", "length")

LlmResponse Represents a structured response from the LLM containing prototype file information.

```
@Serializable
data class LlmResponse(
    val mainFile: String,
    val files: Map<String, FileContent>,
)
```

Fields

- `mainFile`: The entry point file for the prototype (e.g., "index.js")
- `files`: A map of filenames to their contents, representing the complete prototype file structure

FileContent Represents the content of a file in a serializable format.

```
@Serializable
data class FileContent(
    val content: String,
)
```

Fields

- `content`: The string content of the file

Custom Exceptions The module defines two custom exception classes for error handling:
OllamaException

```
class OllamaException(
    message: String,
) : RuntimeException(message)
```

Used for errors related to Ollama communication or operation.
PromptException

```
class PromptException(
    message: String,
) : RuntimeException(message)
```

Used for errors related to prompt processing or response parsing.

JSON Conversion Utilities The module includes a helper function for converting JsonObject to LlmResponse:

```
fun convertJsonToLlmResponse(json: JsonObject): LlmResponse
```

This function:

- Extracts the "files" field from the JSON
- Converts each file entry to a FileContent object
- Sets a default mainFile if not provided
- Handles different JSON formats (code/content fields)
- Throws PromptException for missing or malformed fields

7.7 Utils Module

7.7.1 Overview

The Utils module provides core utility functionality used throughout the application. It offers abstractions for common operations such as file storage, environment configuration, JSON processing, and AWS service interactions. This module serves as a foundation for other modules by handling low-level implementation details and providing clean interfaces for common tasks.

Key responsibilities of the Utils module include:

- Providing a unified storage interface that supports both local and S3 remote storage
- Loading and managing environment variables from various sources
- Handling JSON processing, serialization, and deserialization
- Facilitating secure interactions with AWS services such as S3 and STS

- Abstracting implementation details to provide simple, consistent APIs

The Utils module is designed with reliability and testability in mind, featuring comprehensive exception handling, clear separation of concerns, and extensive test coverage. It follows the principle of dependency injection where appropriate, allowing for easy mocking during testing.

7.7.2 Storage Service

StorageService The `StorageService` object provides a unified interface for interacting with both local and remote file storage. It acts as a facade over the `LocalStorage` and `S3Storage` implementations, allowing the rest of the application to work with files without directly dealing with the storage implementation details.

```
object StorageService {
    private var localStorage = true

    private fun updateStorageLocation()
    private fun getStorageLocation(): String

    fun storeFileLocal(path: String, key: String, file: File): String
    suspend fun storeFileRemote(bucket: String, key: String, file: File): String
    fun getFileLocal(path: String): ByteArray?
    suspend fun getFileRemote(bucket: String, key: String): ByteArray?
    fun deleteFileLocal(path: String): Boolean
    suspend fun deleteFileRemote(bucket: String, key: String): Boolean
}
```

Configuration Methods `updateStorageLocation`

- Resets and updates the environment configuration
- Retrieves the current value of the `LOCAL_STORAGE` environment variable
- Updates the internal `localStorage` flag accordingly

`getStorageLocation`

- Calls `updateStorageLocation` to ensure current configuration is used
- Returns "local" or "remote" based on the `localStorage` flag

Local Storage Methods These methods provide a clean interface for local file storage operations by delegating to the `LocalStorage` object:

`storeFileLocal`

- Stores a file in the local filesystem at the specified path
- Returns the full path to the stored file if successful, or an empty string on failure

`getFileLocal`

- Retrieves a file from the local filesystem at the specified path
- Returns the file content as a ByteArray, or null if the file doesn't exist

`deleteFileLocal`

- Deletes a file from the local filesystem
- Returns true if the deletion was successful, false otherwise

Remote Storage Methods These methods provide a clean interface for remote (S3) file storage operations by delegating to the `S3Storage` object:

`storeFileRemote`

- Stores a file in the specified S3 bucket with the given key
- Returns the full S3 URL to the stored file if successful, or an empty string on failure
- Implemented as a suspend function to support asynchronous operations

`getFileRemote`

- Retrieves a file from the specified S3 bucket with the given key
- Returns the file content as a ByteArray, or null if the file doesn't exist
- Implemented as a suspend function to support asynchronous operations

`deleteFileRemote`

- Deletes a file from the specified S3 bucket with the given key
- Returns true if the deletion was successful, false otherwise
- Implemented as a suspend function to support asynchronous operations

7.7.3 Local Storage

LocalStorage The `LocalStorage` object provides functionality for storing, retrieving, and deleting files in the local filesystem. It handles all the low-level file operations and includes robust error handling to ensure reliability.

```
object LocalStorage {
    fun storeFile(path: String, name: String, file: File): String
    fun getFile(path: String): ByteArray?
    fun deleteFile(path: String): Boolean
}
```

Key Methods `storeFile`

- Copies a file to the specified path with the given name
- Uses Kotlin's `Path` API to handle filesystem operations
- Deletes the original file after a successful copy
- Wraps operations in `runCatching` to handle exceptions gracefully
- Returns the full path to the stored file if successful, or an empty string on failure

```
fun storeFile(path: String, name: String, file: File): String =
    runCatching {
        file.copyTo(Path(path, name).toFile(), overwrite = true)
        file.delete()
    }.getOrNull()?.let { Path(path, name).toString() } ?: ""
```

`getFile`

- Retrieves the content of a file at the specified path
- Uses Kotlin's `Path` API to handle filesystem operations
- Wraps operations in `runCatching` to handle exceptions gracefully
- Returns the file content as a ByteArray if successful, or null on failure

```
fun getFile(path: String): ByteArray? =
    runCatching {
        Path(path).toFile().readBytes()
    }.getOrNull()
```

`deleteFile`

- Deletes a file at the specified path
- Uses Kotlin's `Path` API to handle filesystem operations
- Wraps operations in `runCatching` to handle exceptions gracefully
- Returns true if the deletion was successful, false otherwise

```
fun deleteFile(path: String) = runCatching { Path(path).toFile().delete() }.isSuccess
```

Error Handling The `LocalStorage` implementation prioritizes robustness through comprehensive error handling:

- All file operations are wrapped in `runCatching` blocks to catch any exceptions
- Failed operations return null or empty values rather than throwing exceptions
- This design allows callers to handle failures gracefully without complex try-catch logic
- The implementation handles various failure cases, including:
  - File not found
  - Permission denied
  - I/O errors
  - Insufficient disk space

Path Handling The implementation uses Kotlin's `Path` API for reliable path handling:

- Correctly joins paths and filenames using platform-specific separators
- Handles both absolute and relative paths
- Abstracts away platform-specific path differences
- Provides consistent behavior across different operating systems

7.7.4 S3 Storage

S3Storage The `S3Storage` object provides functionality for storing, retrieving, and deleting files in Amazon S3 storage. It acts as a thin wrapper around the `S3Service`, adding a layer of abstraction and consistent error handling.

```
object S3Storage {
    suspend fun storeFile(bucket: String, key: String, file: File): String
    suspend fun getFile(bucket: String, key: String): ByteArray?
    suspend fun deleteFile(bucket: String, key: String): Boolean
}
```

Key Methods `storeFile`

- Uploads a file to the specified S3 bucket with the given key
- Delegates to `S3Service.uploadFile` for the actual operation
- Returns the full S3 path to the stored file if successful, or an empty string on failure
- Implemented as a suspend function to support asynchronous operations

```
suspend fun storeFile(bucket: String, key: String, file: File): String =
    S3Service.uploadFile(bucket, file, key)
```

`getFile`

- Retrieves a file from the specified S3 bucket with the given key
- Delegates to `S3Service.getFile` for the actual operation
- Returns the file content as a ByteArray if successful, or null on failure
- Implemented as a suspend function to support asynchronous operations

```
suspend fun getFile(bucket: String, key: String): ByteArray? =
    S3Service.getFile(bucket, key)
```

`deleteFile`

- Deletes a file from the specified S3 bucket with the given key
- Delegates to `S3Service.deleteObject` for the actual operation
- Returns true if the deletion was successful, false otherwise
- Implemented as a suspend function to support asynchronous operations

```
suspend fun deleteFile(bucket: String, key: String): Boolean =
    S3Service.deleteObject(bucket, key)
```

S3Service The `S3Service` object provides the core functionality for interacting with Amazon S3. It handles the actual AWS SDK operations, authentication, and error handling.

```
object S3Service {
    private val dispatcher = Dispatchers.IO
    private lateinit var s3Manager: S3Manager
    private lateinit var s3client: S3Client

    fun init(s3Manager: S3Manager)
    private suspend fun ensureClient()
    suspend fun uploadFile(bucketName: String, file: File, key: String): String
    suspend fun getFile(bucketName: String, key: String): ByteArray?
    suspend fun deleteObject(bucketName: String, key: String): Boolean
    suspend fun listBucket(bucketName: String): List<String>
}
```

Initialization and Client Management `init`

- Initializes the S3Service with an S3Manager instance
- The S3Manager is responsible for handling authentication and client creation

`ensureClient`

- Checks if the S3 client is already initialized
- If not, it obtains a new client from the S3Manager
- This lazy initialization approach improves startup time

S3 Operations `uploadFile`

- Detects the content type of the file using `Files.probeContentType`
- Creates a `PutObjectRequest` with the file and metadata
- Calls the S3 client to upload the file
- Returns the S3 URL if successful, or an empty string on failure

`getFile`

- Creates a `GetObjectRequest` for the specified bucket and key
- Calls the S3 client to download the file
- Returns the file content as a ByteArray if successful, or null on failure

`deleteObject`

- Creates a `DeleteObjectRequest` for the specified bucket and key
- Calls the S3 client to delete the file
- Returns true if the operation was successful, false otherwise

`listBucket`

- Creates a `ListObjectsV2Request` for the specified bucket
- Calls the S3 client to list the objects in the bucket
- Returns a list of object keys if successful, or an empty list on failure

7.7.5 Environment Management

EnvironmentLoader The `EnvironmentLoader` object provides functionality for loading and accessing environment variables from various sources. It abstracts the details of environment variable retrieval, supporting both system environment variables and environment files (.env).

```
object EnvironmentLoader {
    private var env: Dotenv? = null

    fun loadEnvironmentFile(fileName: String)
    fun get(key: String): String
    fun reset()
}
```

Key Methods `loadEnvironmentFile`

- Loads environment variables from a specified file
- Checks if the file exists before attempting to load it
- Uses the Dotenv library to parse and load the variables
- Configures the loader to use the current directory and specified filename

```
fun loadEnvironmentFile(fileName: String) {
    if (File(fileName).exists()) {
        env = Dotenv
            .configure()
            .directory("./")
            .filename(fileName)
            .load()
    }
}
```

`get`

- Retrieves the value of an environment variable by its key
- First checks the loaded environment file (if any)
- Falls back to system environment variables via `SystemEnvironment`
- Returns an empty string if the variable is not found in either source

```
fun get(key: String): String = env?.get(key) ?: SystemEnvironment.readSystemVariable(key) ?: "
    "
```

`reset`

- Resets the environment by setting the internal Dotenv reference to null
- This effectively unloads any previously loaded environment file
- Useful for testing and for forcing a reload of environment variables

```
fun reset() {
    env = null
}
```

SystemEnvironment The `SystemEnvironment` internal object provides a wrapper around system environment variable access. This abstraction improves testability by allowing the system environment access to be mocked during tests.

```
internal object SystemEnvironment {
    fun readSystemVariable(variableName: String): String?
}
```

Key Methods `readSystemVariable`

- Reads a system environment variable by name
- Simply delegates to `System.getenv`
- Returns the value of the variable, or null if it doesn't exist
- Encapsulated in an object to allow for mocking in tests

```
fun readSystemVariable(variableName: String): String? = System.getenv(variableName)
```

Environment Usage Throughout the application, environment variables are used for configuration in a consistent way:

```
// Example: Configuring the LLM model
val model: String = EnvironmentLoader.get("OLLAMA_MODEL")

// Example: Checking storage location
val localStorageEnabled = EnvironmentLoader.get("LOCAL_STORAGE").toBoolean()

// Example: Configuring AWS region
val awsRegion = EnvironmentLoader.get("AWS_REGION")
```

Environment File Format The environment file format is a simple key-value format, with one variable per line:

```
DB_URL=testUrl
DB_USER=testUser
DB_PASSWORD=testPassword
AWS_REGION=eu-west-2
LOCAL_STORAGE=true
OLLAMA_HOST=localhost
```

This format aligns with the standard .env file format used in many applications and is compatible with various development tools and deployment environments.

7.7.6 JSON Processing

PoCJSON The `PoCJSON` object provides utilities for JSON processing tasks, including reading JSON files and extracting attributes from JSON structures. It encapsulates common JSON operations to provide a consistent interface across the application.

```
object PoCJSON {
    fun readJsonFile(name: String): JsonObject
    fun findCognitoUserAttribute(array: JsonArray, attribute: String): String?
}
```

Key Methods `readJsonFile`

- Reads a JSON file from the classpath resources
- Uses the class loader to find and load the file
- Reads the file content as a string
- Parses the string into a JsonObject using Kotlin's serialization library
- Returns the parsed JsonObject for further processing

```
fun readJsonFile(name: String): JsonObject {
    val file = javaClass.classLoader.getResourceAsStream(name)
    val content: String = file?.readBytes()?.toString(Charsets.UTF_8) ?: ""
    return Json.parseToJsonElement(content).jsonObject
}
```

`findCognitoUserAttribute`

- Searches for a specific attribute in a Cognito user attribute array
- Takes an array of JSON objects and an attribute name to search for
- Performs a case-insensitive search by converting the attribute name to lowercase
- Returns the value of the attribute if found, or null otherwise
- Handles various error cases, including missing keys, non-primitive values, and type mismatches

```
fun findCognitoUserAttribute(array: JsonArray, attribute: String): String? =
    try {
        array
            .find { it.jsonObject["Name"]?.toString() == "\"${attribute.lowercase()}\"" }
            ?.let {
                kotlin.runCatching { it.jsonObject["Value"]!!.jsonPrimitive.content }.
                    getOrNull()
            }
    } catch (e: IllegalArgumentException) {
        null
    }
```

JSON Configuration Files The application uses several JSON configuration files, including:

s3_config.json

- Contains configuration for S3 storage
- Specifies the IAM role to assume for S3 access
- Lists the allowed S3 buckets
- Defines the AWS region for S3 operations

```
{
  "s3": {
    "role": "arn:aws:iam::977098998083:role/S3SafeRole",
    "buckets": [
      "one-day-poc-chat",
      "one-day-poc-prompts"
    ],
    "region": "eu-west-2"
  }
}
```

aws_config.json

- Contains configuration for AWS services
- Provides templates for AWS configuration in different environments
- Serves as an example for the required configuration structure

JSON Serialization The application uses Kotlin's serialization library for JSON processing:

- `kotlinx.serialization.json` package for JSON operations
- `Json` for parsing and formatting JSON
- `JsonObject`, `JsonArray`, and `JsonPrimitive` for representing JSON structures
- `jsonObject` and `jsonPrimitive` for accessing JSON elements

This provides type-safe JSON processing with strong integration with Kotlin's type system and language features.

7.7.7 AWS Services

S3Manager The `S3Manager` class manages the Amazon S3 client and handles the necessary authentication for S3 operations. It coordinates with the AWS Security Token Service (STS) to assume roles with appropriate permissions.

```
class S3Manager(
    private val s3Config: JsonObject?,
    private val stsClient: StsClient,
) {
    private var s3client: S3Client? = null

    suspend fun assumeS3SafeRole(): Credentials
    suspend fun getClient(): S3Client
    fun buildClient(s3Config: JsonObject?, credentialsProvider: StaticCredentialsProvider):
        ConfiguredS3Client
}
```

Key Methods `assumeS3SafeRole`

- Assumes an IAM role to obtain temporary credentials for S3 access
- Creates an AssumeRoleRequest with the role ARN from the configuration
- Uses the STS client to request temporary credentials
- Returns a Credentials object containing the temporary credentials
- Handles errors and returns empty credentials if the operation fails

`getClient`

- Provides an S3 client, either creating a new one or returning an existing one
- Initializes the client lazily upon first request
- Obtains temporary credentials by assuming the role
- Creates a StaticCredentialsProvider with the temporary credentials
- Builds and returns an S3 client configured with the credentials and region

`buildClient`

- Creates and configures an S3 client with the given parameters
- Extracts the region from the configuration, defaulting to eu-west-2
- Creates an S3Client instance with the specified region and credentials
- Returns a ConfiguredS3Client containing the client and configuration details

STSInteractor The `STSInteractor` object provides functionality for interacting with the AWS Security Token Service (STS). It facilitates assuming IAM roles to obtain temporary credentials for AWS service access.

```
object STSInteractor {
    suspend fun assumeRole(stsClient: StsClient, request: AssumeRoleRequest): Credentials
}
```

Key Methods `assumeRole`

- Assumes an IAM role using the provided STS client and request
- Validates that the request contains a role ARN
- Calls the STS client's assumeRole method to obtain temporary credentials
- Converts the STS credentials to a Credentials object
- Returns empty credentials if the role ARN is missing
- Uses the Kotlin use function to ensure proper resource cleanup

```
suspend fun assumeRole(stsClient: StsClient, request: AssumeRoleRequest): Credentials {
    if (request.roleArn == null) {
        return Credentials("", "", "")
    }

    stsClient.use { sts ->
        val response = sts.assumeRole(request)
        return Credentials(
            response.credentials!!.accessKeyId,
            response.credentials!!.secretAccessKey,
```

```
            response.credentials!!.sessionToken,
        )
    }
}
```

AWSCredentialsProvider The `AWSCredentialsProvider` interface provides a consistent way to obtain AWS credentials throughout the application. It abstracts the source of credentials, allowing for different implementations such as environment variables, assumed roles, or static credentials.

```
interface AWSCredentialsProvider {
    fun getAccessKeyId(): String
    fun getSecretAccessKey(): String
    fun getRegion(): String
}
```

Implementations `DefaultAWSCredentialsProvider`

- Uses the AWS SDK's DefaultChainCredentialsProvider to resolve credentials
- Follows the standard AWS credential resolution chain
- Handles the case when credentials cannot be resolved by providing default values
- Uses Kotlin coroutines to resolve credentials asynchronously

`AWSUserCredentials`

- Singleton object that provides AWS credentials to the application
- Uses a DefaultAWSCredentialsProvider by default
- Allows for setting a custom provider for testing
- Provides methods to reset the provider to the default

AWS Configuration The application uses a JSON configuration file for AWS settings:

- Loaded using the `PoCJSON.readJsonFile` method
- Contains information such as role ARNs, bucket names, and regions
- Used by the S3Manager to configure the S3 client
- Example configuration:

```
{
  "s3": {
    "role": "arn:aws:iam::977098998083:role/S3SafeRole",
    "buckets": [
      "one-day-poc-chat",
      "one-day-poc-prompts"
    ],
    "region": "eu-west-2"
  }
}
```

Initialization The AWS services are initialized in the `UtilsModule` class:

```
fun Application.configureUtils() {
    val awsRegion = EnvironmentLoader.get("AWS_REGION")
    val s3Manager =
        S3Manager(PoCJSON.readJsonFile("s3_config.json"), StsClient { region = awsRegion })
    S3Service.init(s3Manager)
}
```

This initialization:

- Loads the AWS region from environment variables
- Creates an STS client configured with the region
- Loads the S3 configuration from the JSON file
- Creates an S3Manager with the configuration and STS client
- Initializes the S3Service with the S3Manager

8 Client Application

8.1 Overview

The client application is a React-based frontend for a prototype code generation system. It provides an interactive chat interface where users can request code prototypes, which are then automatically generated and run within a sandboxed WebContainer environment. The application is designed to facilitate rapid prototyping and experimentation with code snippets, complete web applications, and UI components.

8.2 Architecture

The application follows a modern React architecture with the following key components:

8.2.1 Core Structure

- **State Management:** Uses React Context API for global state (Authentication, Conversations)
- **Routing:** Implements React Router for navigation between pages
- **UI Components:** Utilizes a combination of custom components and adapted Shadcn UI components
- **API Communication:** Uses fetch API wrapped in service modules

8.2.2 Key Directories

- `/src/components/`: UI components organized by feature or functionality
- `/src/contexts/`: React context providers for state management
- `/src/hooks/`: Custom React hooks for shared functionality
- `/src/pages/`: Top-level page components
- `/src/services/`: Business logic and API interaction
- `/src/styles/`: Global styles and variables
- `/src/types/`: TypeScript type definitions
- `/src/__tests__/`: Test files organized by feature

8.3 Key Features

8.3.1 Authentication System

The application implements a complete authentication flow using the AuthContext provider:

- User login/logout via external authentication service
- Session persistence with server-side verification
- Role-based access control (user vs. admin)
- Protected routes and conditional UI elements

```
const checkAuth = async () => {
  try {
    const response = await fetch('http://localhost:8000/api/auth/check', {
      method: 'GET',
      credentials: 'include',
    });

    if (!response.ok) {
      setIsAuthenticated(false);
      setIsAdmin(false);
      UserService.clearUser();
      return;
    }

    const data = await response.json();
    if (data.userId) {
      await UserService.fetchUserData();
      setIsAuthenticated(true);
      setIsAdmin(data.isAdmin || false);
      // Handle saved prompt if exists
    }
  } catch (error) {
    // Error handling
  }
};
```

Listing 253: From AuthContext.tsx

8.3.2 Conversation Management

The application manages chat conversations through the ConversationContext:

- Creating, storing, and retrieving conversation history
- Managing active conversation state
- Conversation naming and organization
- Message synchronization between UI and server

8.3.3 Chat Interface

The chat interface provides an interactive medium for users to request code prototypes:

- Real-time message exchange
- Markdown support with code syntax highlighting
- Automatic scrolling to recent messages
- Error handling and feedback

8.3.4 WebContainer Integration

One of the most powerful features is the integration with WebContainer for running generated code:

- Sandbox environment for executing code safely in the browser
- File system virtualization
- Real-time code execution and preview
- Support for npm packages and dependencies

```
async function loadFiles() {
  if (!webcontainerInstance || !files) return;

  await resetWebContainer();
  setStatus('Normalising files...');
  const normalisedFiles = normaliseFiles(files);

  setStatus('Mounting files...');
  try {
    await webcontainerInstance.mount(normalisedFiles);
    console.log('Files mounted successfully');

    setStatus('Installing dependencies...');
    const installProcess = await webcontainerInstance.spawn('npm', ['install']);
    activeProcessesRef.current.push(installProcess);

    const exitCode = await installProcess.exit;
    if (exitCode !== 0) {
      throw new Error(`npm install failed with exit code ${exitCode}`);
    }

    setStatus('Starting development server...');
    const startProcess = await webcontainerInstance.spawn('npm', ['run', 'start']);
    activeProcessesRef.current.push(startProcess);

    // Additional setup and configuration
  } catch (error) {
    console.error('Error:', error);
    setStatus(`Error: ${getErrorMessage(error)}`);
  }
}
```

Listing 254: From PrototypeFrame.tsx

8.3.5 Responsive UI

The interface adapts to different screen sizes using custom hooks and responsive design:

- Mobile detection with `useIsMobile` hook
- Collapsible sidebar for efficient use of screen space
- Responsive layouts with Tailwind CSS

```
export function useIsMobile() {
  const [isMobile, setIsMobile] = React.useState<boolean | undefined>(undefined)

  React.useEffect(() => {
    const mql = window.matchMedia(`(max-width: ${MOBILE_BREAKPOINT - 1}px)`)
    const onChange = () => {
      setIsMobile(window.innerWidth < MOBILE_BREAKPOINT)
    }
    mql.addEventListener("change", onChange)
    setIsMobile(window.innerWidth < MOBILE_BREAKPOINT)
    return () => mql.removeEventListener("change", onChange)
  }, [])

  return !!isMobile
}
```

Listing 255: From UseMobile.tsx

8.4 Page Structure

8.4.1 Landing Page

- Welcome message and value proposition
- Suggested prompts for quick starting
- Input box for entering custom prompts
- Previous prompt history for authenticated users

8.4.2 Generate Page

- Split-view interface with chat and prototype panels
- Collapsible chat panel
- Live code preview
- Project management (naming, export options)

8.4.3 Profile Page

- User information display
- Account settings

8.4.4 Error Pages

- Consistent error messaging
- Different templates for common HTTP errors (401, 403, 404, 500)
- Navigation back to safe states

8.5 Component Hierarchy

8.5.1 App Component

The root component that sets up routing and global providers:

```
const App: React.FC = () => {
  return (
    <AuthProvider>
      <ConversationProvider>
        <NavBar />
        <Routes>
          <Route path="/" element={<LandingPage />} />
          <Route path="/profile" element={<ProfilePage />} />
          <Route path="/generate" element={<Generate />} />
```

```
          <Route path="/*" element={<ErrorRoutes />} />
        </Routes>
      </ConversationProvider>
    </AuthProvider>
  );
};
```

Listing 256: App Component Structure

8.5.2 Generate Page Components

- `SidebarWrapper` - Provides the layout structure with a collapsible sidebar
- `ChatScreen` - Manages the chat interface and message exchange
- `PrototypeFrame` - Handles the WebContainer integration and code preview

8.5.3 Chat Components

- `ChatBox` - Input interface for entering messages
- `MessageBox` - Displays conversation messages with formatting
- `MessageBubble` - Individual message styling with sender-based variants

8.5.4 Sidebar Components

- `AppSidebar` - Main sidebar container with navigation items
- `NavMain` - Conversation history navigation
- `NavUser` - User profile and authentication controls

8.6 Type System

The application uses TypeScript with a well-defined type system centered around:

8.6.1 Message Types

```
export type MessageRole = 'User' | 'LLM';

export interface Message {
    role: MessageRole;
    content: string;
    timestamp: string;
    conversationId?: string;
    id?: string;
}
```

Listing 257: From Types.ts

8.6.2 File System Types

```
export interface WebContainerFile {
    file: {
        contents: string;
    };
}

export interface WebContainerDirectory {
    directory: {
        [fileName: string]: WebContainerFile | WebContainerDirectory;
    };
}

export type FileSystemEntry = WebContainerFile | WebContainerDirectory;

export interface FileTree {
    [path: string]: FileSystemEntry;
}
```

Listing 258: File System Types

8.6.3 API Response Types

```
export interface ServerResponse {
    chat?: ChatResponse;
    prototype?: PrototypeResponse;
}

export interface ChatResponse {
    message: string;
    role: MessageRole;
    timestamp: string;
    messageId?: string;
}

export interface PrototypeResponse {
    files: FileTree;
}
```

Listing 259: API Response Types

8.6.4 Component Props Types

```
export interface ChatScreenProps {
    showPrototype: boolean;
    setPrototype: Dispatch<SetStateAction<boolean>>;
    setPrototypeFiles: Dispatch<SetStateAction<FileTree>>;
    initialMessage?: string | null;
}

export interface ChatHookReturn {
    message: string;
    setMessage: (message: string) => void;
    sentMessages: Message[];
    handleSend: (messageToSend?: string) => Promise<void>;
    errorMessage: string | null;
    setErrorMessage: (error:string | null) => void;
}
```

Listing 260: Component Props Types

8.7 API Integration

The application communicates with a backend server for several functionalities:

8.7.1 Authentication Endpoints

- `GET /api/auth/check` - Verify authentication status
- `POST /api/auth/logout` - Log out the current user
- `GET /api/auth/me` - Retrieve current user information

8.7.2 Chat Endpoints

- `POST /api/chat/json` - Send a chat message and get response
- `GET /api/chat/history` - Retrieve conversation history
- `GET /api/chat/history/:conversationId` - Get messages for a specific conversation
- `GET /api/chat/history/:conversationId/:messageId` - Get prototype files for a specific message
- `POST /api/chat/json/:conversationId/rename` - Rename a conversation

8.8 Custom Hooks

The application utilizes several custom hooks for shared functionality:

8.8.1 useWebContainer

Manages the WebContainer instance lifecycle:

```
export function useWebContainer() {
  const [instance, setInstance] = useState<WebContainer | null>(null);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<Error | null>(null);

  useEffect(() => {
    let mounted = true;

    async function initWebContainer() {
      try {
        if (!isCrossOriginIsolated()) {
          throw new Error(
            'Cross-Origin Isolation is not enabled. WebContainer requires the following HTTP
                headers:\n' +
            '- Cross-Origin-Embedder-Policy: require-corp\n' +
            '- Cross-Origin-Opener-Policy: same-origin'
          );
        }

        // Initialize WebContainer instance
        // ...
      } catch (err) {
        // Error handling
      }
    }

    initWebContainer();

    return () => {
      mounted = false;
    };
  }, []);

  return {
    instance,
    loading,
    error,
    isReady: !!instance && !loading
  };
}
```

Listing 261: useWebContainer Hook

8.8.2 useIsMobile

Detects and responds to mobile viewport sizes:

```
export function useIsMobile() {
  const [isMobile, setIsMobile] = React.useState<boolean | undefined>(undefined)

  React.useEffect(() => {
    const mql = window.matchMedia(`(max-width: ${MOBILE_BREAKPOINT - 1}px)`)
    const onChange = () => {
      setIsMobile(window.innerWidth < MOBILE_BREAKPOINT)
    }
    mql.addEventListener("change", onChange)
    setIsMobile(window.innerWidth < MOBILE_BREAKPOINT)
    return () => mql.removeEventListener("change", onChange)
  }, [])

  return !!isMobile
}
```

Listing 262: useIsMobile Hook

8.8.3 ChatMessage

Manages chat state and API interactions:

```
const ChatMessage = ({
  setPrototype,
  setPrototypeFiles,
}: ChatMessageProps): ChatHookReturn => {
```

```
  const [message, setMessage] = useState<string>('');
  const [sentMessages, setSentMessages] = useState<Message[]>([]);
  const [llmResponse, setLlmResponse] = useState<string>('');
  const [chatResponse, setChatResponse] = useState<ChatResponse | null>(null);
  const [errorMessage, setErrorMessage] = useState<string | null>(null);

  const { activeConversationId, createConversation, messages, loadingMessages } =
      useConversation();

  // ... implementation of message handling logic

  return {
    message,
    setMessage,
    sentMessages,
    handleSend,
    errorMessage,
    setErrorMessage,
  };
};
```

Listing 263: ChatMessage Hook

8.9 Testing Strategy

The application uses Vitest with React Testing Library for component testing:

8.9.1 Component Tests

Tests for individual UI components verify:

- Correct rendering of UI elements
- Component behavior with different props
- Event handling (click, input, etc.)
- Conditional rendering logic

8.9.2 Hook Tests

Tests for custom hooks verify:

- Correct state initialization
- State updates in response to actions
- Side effect management
- Error handling

8.9.3 Context Tests

Tests for context providers verify:

- Context initialization
- State updates across components
- API interactions
- Error states

8.9.4 Page Tests

Integration tests for pages verify:

- Correct assembly of components
- Routing behavior
- User flows

8.10 Styling System

The application uses a component-based styling approach with Tailwind CSS and CSS modules:

8.10.1 Utility-First Approach

Tailwind classes are used for most styling needs:

```
<div className="flex flex-col items-center justify-center text-center mb-5 px-6">
  <h1 className="text-5xl md:text-6xl font-light">Enabling you from</h1>
  <h2 className="text-6xl md:text-7xl font-bold">day one</h2>
</div>
```

Listing 264: Tailwind CSS Example

8.10.2 Component Variants

The application uses class-variance-authority (cva) for component variants:

```
const messageBubbleVariant = cva(
  "flex gap-2 items-end break-words relative group",
  {
    variants: {
      variant: {
        llm: "self-start",
        user: "self-end flex-row-reverse max-w-[60%]",
      },
      layout: {
        default: "",
      },
    },
    defaultVariants: {
      variant: "llm",
      layout: "default",
    },
  },
);
```

Listing 265: Component Variants with cva

8.10.3 Global Variables

CSS variables are used for theme consistency:

```
:root {
    --background: url('../assets/background.svg');
    --foreground: 247 55% 20%;
    --card: 237, 19%, 100%;
    --card-foreground: 0 0% 96%;
    --popover: 247 35% 20%;
    --popover-foreground: 0 0% 96%;
    --primary: 240, 10%, 89%;
    --primary-foreground: 247 45% 30%;
    /* ... more variables */
}
```

Listing 266: CSS Variables

8.11 Deployment Considerations

8.11.1 Cross-Origin Isolation

WebContainer requires Cross-Origin Isolation with specific HTTP headers:

- Cross-Origin-Embedder-Policy: require-corp
- Cross-Origin-Opener-Policy: same-origin

8.11.2 Environment Configuration

The application is configured for different environments:

- Development: Local server on http://localhost:8000
- Production: Would require proper API endpoints configuration

8.11.3 Performance Optimizations

Several optimizations are implemented:

- Memoization of expensive component renders
- Lazy loading of WebContainer
- Efficient state management with context API
- Conditional rendering to reduce DOM size

8.12 Conclusion

The client application represents a sophisticated React-based solution for interactive code generation and prototyping. It combines modern frontend technologies with innovative WebContainer integration to provide a seamless development experience. The architecture follows best practices for maintainability, scalability, and user experience, making it a powerful tool for rapid application development and experimentation.