Degree Thesis, Åland University of Applied Sciences, Degree Programme in Information Technology

Development of a Serverless RESTful API

Thomas Hagelberg

2023:23

Date of approval: 28.5.2023
Academic Supervisor: Joakim Isaksson

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Thomas Hagelberg
Title:	Development of a Serverless RESTful API
Academic Supervisor:	Joakim Isaksson
Commissioned by:	Crosskey Banking Solutions

Abstract

The purpose of this thesis is to illustrate the development process of a new serverless RESTful API for managing document-related privileges. The API is hosted in the AWS ecosystem, and based on the serverless compute engine Fargate.

The API is designed to determine the actions that can be taken on a document based on its sub-category, and will be used in Crosskey's new frontend system, which will act as a replacement for static xml configuration.

I have used the agile methodology Scrum to break down the development process into two-week sprints and used various technologies such as Java, Spring, JPA, Oracle, AWS and various other related technologies.

Keywords

Java, Spring, RESTful, Serverless, AWS

Serial number:	ISSN:	Language:	Number of pages:
2023:23	1458-1531	English	46

Handed in:	Date of presentation:	Approved:
3.5.2023	16.5.2023	28.5.2023

EXAMENSARBETE
Åland University of Applied Sciences

Utbildningsprogram:	Informationsteknik
Författare:	Thomas Hagelberg
Arbetets namn:	Utveckling av en serverlös RESTful API
Handledare:	Joakim Isaksson
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Syftet med denna uppsats är att illustrera utvecklingsprocessen av en ny serverlös RESTful API för att hantera dokument-relaterade privilegier. API:et är hyst i AWS-ekosystemet och bygger på den serverlösa beräkningsenheten Fargate.

API:et är utformat för att bestämma vilka åtgärder som kan vidtas på ett dokument baserat på dess underkategori, och kommer att användas i Crosskeys nya frontend-system, vilket kommer att agera som ersättning för statisk XML-konfiguration.

Jag har använt den agila metodologin Scrum för att bryta ner utvecklingsprocessen i två veckors sprintar och använt olika teknologier som Java, Spring, JPA, Oracle, AWS och andra relaterade teknologier.

Nyckelord (sökord)

Java, Spring, RESTful, Serverless, AWS

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
2023:23	1458-1531	Engelska	46

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
3.5.2023	16.5.2023	28.5.2023

INNEHÅLLSFÖRTECKNING/TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

The purpose of this thesis is to illustrate the development process of a new serverless RESTful API for managing document related privileges. The data provided by this API would be used to determine what can be done with a certain type of document, based on its sub-category. This API is intended to be used in Crosskey's new Angular backoffice system. The API would replace an existing xml configuration that provided this functionality in the old Vaadin backoffice system. The backoffice system is a web application providing the bank personnel with a wide variety of functionalities, such as managing users, payments and much more.

The idea of this application was initially brought to light when my colleagues were discussing suitable projects for my thesis, and this turned out to be the best fit based on my own desires. There was no existing microservice in place prior to this in my team, so this was a natural step in our cloud migration journey.

1.2 Method

The initial step was to have several start-up meetings where the overview of the application was discussed. Architectural bullet points were laid out, setting the foundation for how the application should be designed and developed.

The development environment would be of Crosskey standard for Java developers, which included IntelliJ Idea as IDE and Bitbucket as code repository. The programming language was Java and a few of the major frameworks and tools such as Spring Boot and JPA were included in this project.

The development methodology itself is based on Scrum, where the development process relies on increments and two-week sprints, which results in new functionality at the end of each iteration.

1.3 Limitations

One major limitation of this project was to follow API guidelines and security requirements from Crosskey. This resulted in being restricted to use certain versions of frameworks and tools in order to be compliant with other Crosskey systems.

The second limitation was that no front-end functionality was going to be developed for managing the data of this API. This is because it was not clear how the data should be stored and managed, simply that the new backoffice system would fetch the data. This resulted in a gray area as to who was responsible for updating the data as we were initially only responsible for providing the API.

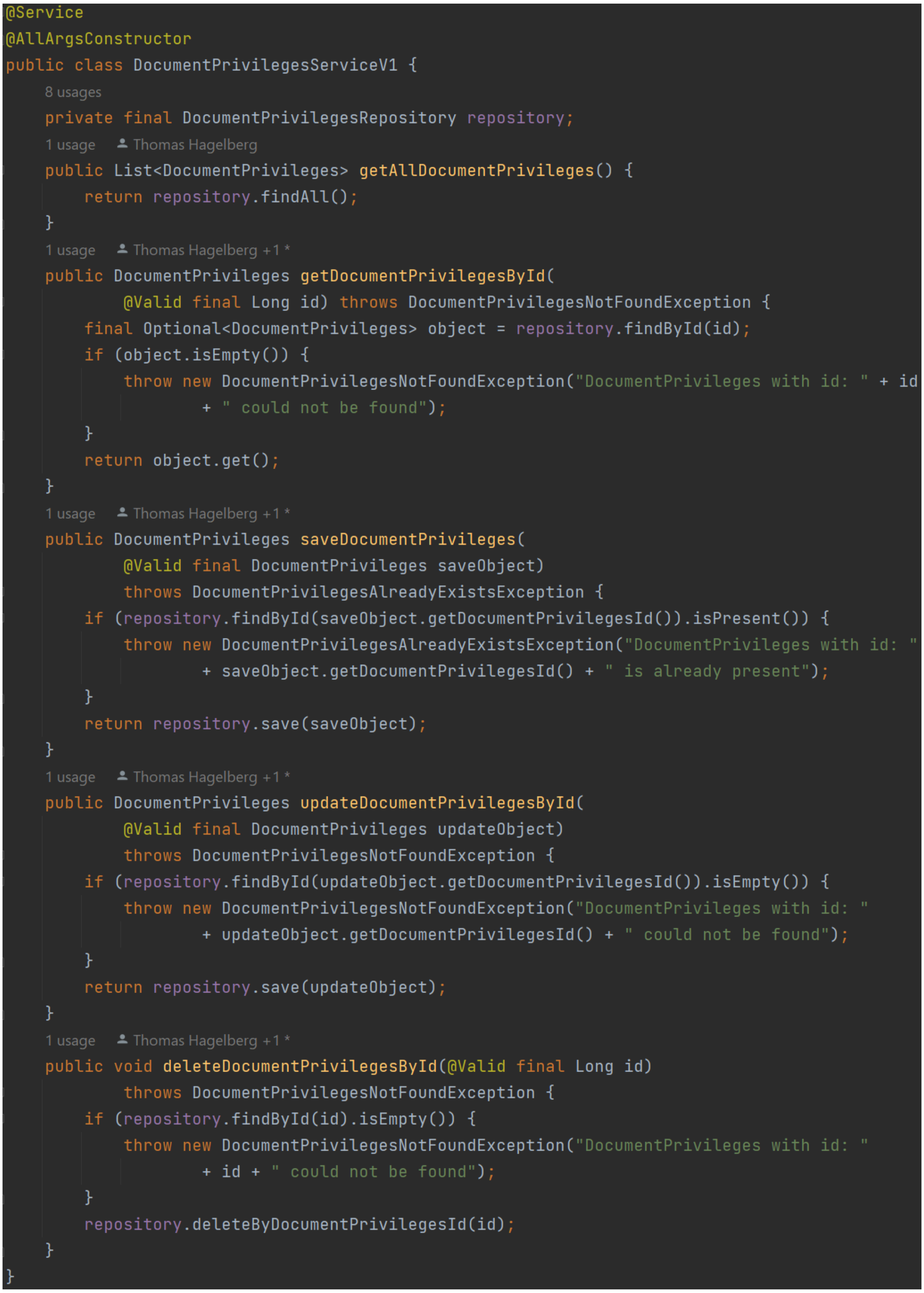
2. PROGRAMMING PRINCIPLES AND PRACTICES

2.1 MVC

MVC (Model-View-Controller) is a design pattern used in software development and used for separating the business logic, front-end and processing of data. MVC divides an application into three main components (Tutorialspoint, n.d):

2.1.1 Model

This component represents the data and the business logic of the application. It is responsible for managing the data and providing an interface to the controller components to access the data. Per good programming principles, most model classes should have exclusive access to their respective repository and only act as an interface towards the controller. Figure 1 illustrates how the model can be represented as a service class in Java, utilizing Spring and Lombok annotations. More explanations of Spring and other used frameworks will be provided in chapter 3.



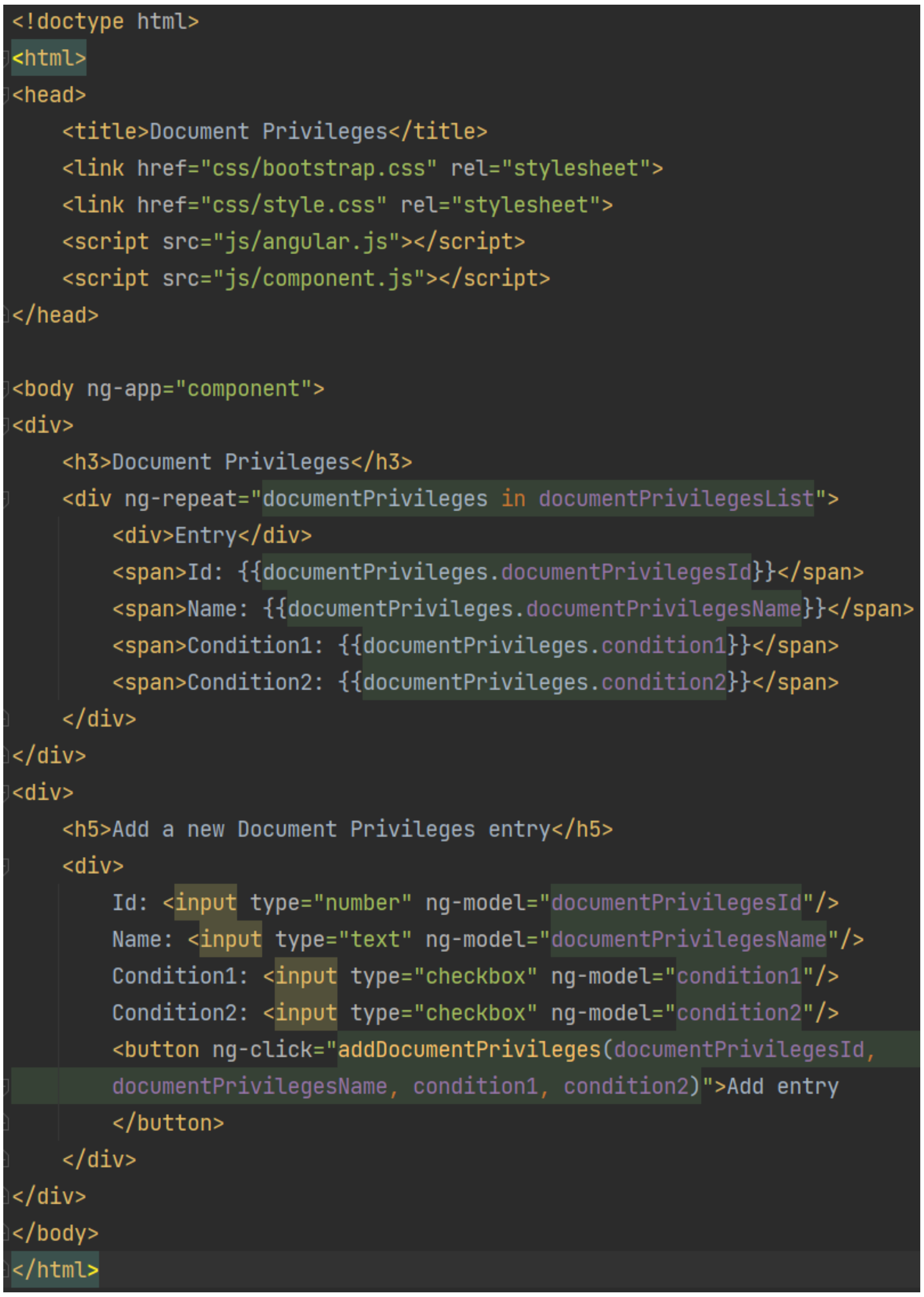
```java
@Service
@AllArgsConstructor
public class DocumentPrivilegesServiceV1 {
    8 usages
    private final DocumentPrivilegesRepository repository;
    1 usage   Thomas Hagelberg
    public List<DocumentPrivileges> getAllDocumentPrivileges() {
        return repository.findAll();
    }
    1 usage   Thomas Hagelberg +1 *
    public DocumentPrivileges getDocumentPrivilegesById(
            @Valid final Long id) throws DocumentPrivilegesNotFoundException {
        final Optional<DocumentPrivileges> object = repository.findById(id);
        if (object.isEmpty()) {
            throw new DocumentPrivilegesNotFoundException("DocumentPrivileges with id: " + id
                    + " could not be found");
        }
        return object.get();
    }
    1 usage   Thomas Hagelberg +1 *
    public DocumentPrivileges saveDocumentPrivileges(
            @Valid final DocumentPrivileges saveObject)
            throws DocumentPrivilegesAlreadyExistsException {
        if (repository.findById(saveObject.getDocumentPrivilegesId()).isPresent()) {
            throw new DocumentPrivilegesAlreadyExistsException("DocumentPrivileges with id: "
                    + saveObject.getDocumentPrivilegesId() + " is already present");
        }
        return repository.save(saveObject);
    }
    1 usage   Thomas Hagelberg +1 *
    public DocumentPrivileges updateDocumentPrivilegesById(
            @Valid final DocumentPrivileges updateObject)
            throws DocumentPrivilegesNotFoundException {
        if (repository.findById(updateObject.getDocumentPrivilegesId()).isEmpty()) {
            throw new DocumentPrivilegesNotFoundException("DocumentPrivileges with id: "
                    + updateObject.getDocumentPrivilegesId() + " could not be found");
        }
        return repository.save(updateObject);
    }
    1 usage   Thomas Hagelberg +1 *
    public void deleteDocumentPrivilegesById(@Valid final Long id)
            throws DocumentPrivilegesNotFoundException {
        if (repository.findById(id).isEmpty()) {
            throw new DocumentPrivilegesNotFoundException("DocumentPrivileges with id: "
                    + id + " could not be found");
        }
        repository.deleteByDocumentPrivilegesId(id);
    }
}
```

Figure 1. Example of a service class working as a model in Java.

2.1.2 View

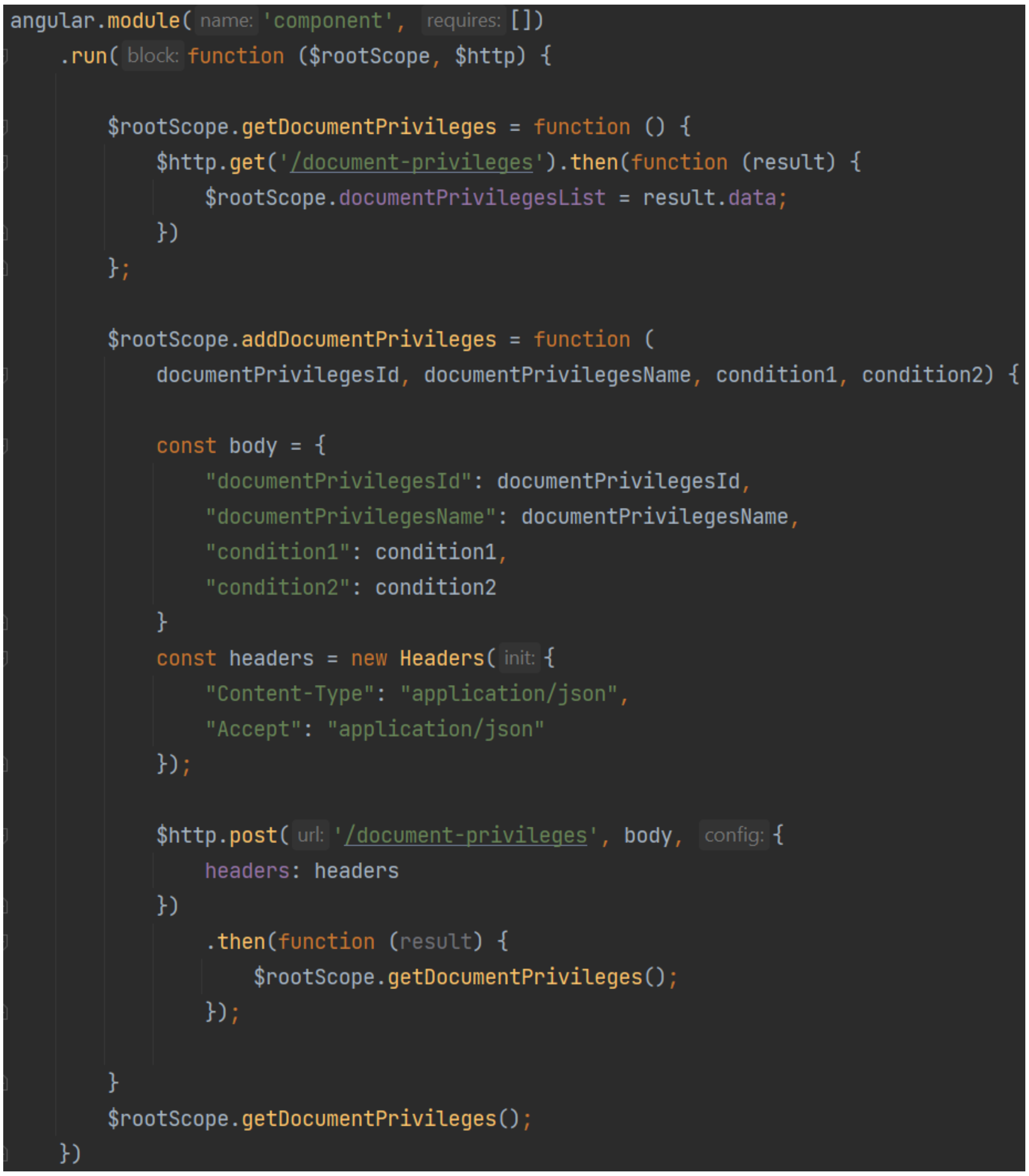
The View is the component that displays the data to the user and provides the user interface for interacting with the application. This includes elements such as buttons, text boxes, images, etc. The View is responsible for converting the Model data into a form that can be displayed to the user.

The View is typically not aware of the Model, and communicates with the Controller to receive updates on changes in the Model. The View updates itself in response to changes in the Model, and informs the Controller of any user inputs. Figure 2 and figure 3 illustrate how a view can be created, using HTML and the Angular framework.



```html
<!doctype html>
<html>
<head>
    <title>Document Privileges</title>
    <link href="css/bootstrap.css" rel="stylesheet">
    <link href="css/style.css" rel="stylesheet">
    <script src="js/angular.js"></script>
    <script src="js/component.js"></script>
</head>

<body ng-app="component">
<div>
    <h3>Document Privileges</h3>
    <div ng-repeat="documentPrivileges in documentPrivilegesList">
        <div>Entry</div>
        <span>Id: {{documentPrivileges.documentPrivilegesId}}</span>
        <span>Name: {{documentPrivileges.documentPrivilegesName}}</span>
        <span>Condition1: {{documentPrivileges.condition1}}</span>
        <span>Condition2: {{documentPrivileges.condition2}}</span>
    </div>
</div>
<div>
    <h5>Add a new Document Privileges entry</h5>
    <div>
        Id: <input type="number" ng-model="documentPrivilegesId"/>
        Name: <input type="text" ng-model="documentPrivilegesName"/>
        Condition1: <input type="checkbox" ng-model="condition1"/>
        Condition2: <input type="checkbox" ng-model="condition2"/>
        <button ng-click="addDocumentPrivileges(documentPrivilegesId,
        documentPrivilegesName, condition1, condition2)">Add entry
        </button>
    </div>
</div>
</body>
</html>
```

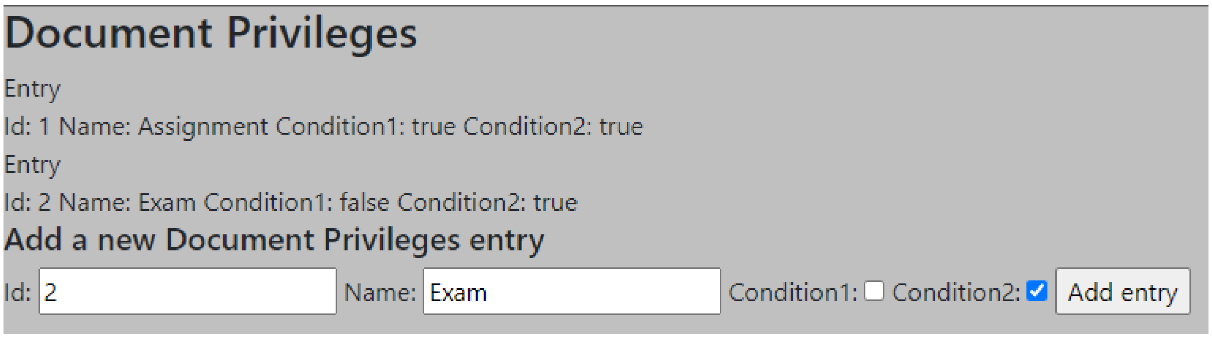
Figure 2. Example of a HTML file working as a view towards the user, utilizing the Angular framework.



```
angular.module( name: 'component',  requires: [])
    .run( block: function ($rootScope, $http) {

        $rootScope.getDocumentPrivileges = function () {
            $http.get('/document-privileges').then(function (result) {
                $rootScope.documentPrivilegesList = result.data;
            })
        };

        $rootScope.addDocumentPrivileges = function (
            documentPrivilegesId, documentPrivilegesName, condition1, condition2) {

            const body = {
                "documentPrivilegesId": documentPrivilegesId,
                "documentPrivilegesName": documentPrivilegesName,
                "condition1": condition1,
                "condition2": condition2
            }
            const headers = new Headers( init: {
                "Content-Type": "application/json",
                "Accept": "application/json"
            });

            $http.post( url: '/document-privileges', body,  config: {
                headers: headers
            })
                .then(function (result) {
                    $rootScope.getDocumentPrivileges();
                });

        }
        $rootScope.getDocumentPrivileges();
    })
```

Figure 3. Example of an angular module that provides enhanced front-end functionality to the view.

Figure 4 show how the actual browser page looks after sending in two entries to the controller, which verifies the data and then forwards the request to the model (service class) which updates the database and returns the updated object to the view.


Document Privileges

Entry

Id: 1 Name: Assignment Condition1: true Condition2: true

Entry

Id: 2 Name: Exam Condition1: false Condition2: true

Add a new Document Privileges entry

Id: 2 Name: Exam Condition1: ☐ Condition2: ☑ Add entry

Figure 4. Browser page based on the HTML file, displaying Document Privileges based example functionality.

2.1.3 Controller

The Controller is responsible for handling user inputs and updating the Model accordingly. This can involve processing user inputs, such as button clicks or form submissions, and updating the Model data based on the inputs received. The Controller is also responsible for updating the View in response to changes in the Model.

The Controller is typically aware of both the Model and the View, and communicates with both to ensure that the Model data is updated and the View is displaying the correct information. It acts as a bridge between the Model and View, ensuring that changes to one component do not affect the other. Figure 5 down below shows an example of how a controller in Java and Spring Boot could look like.

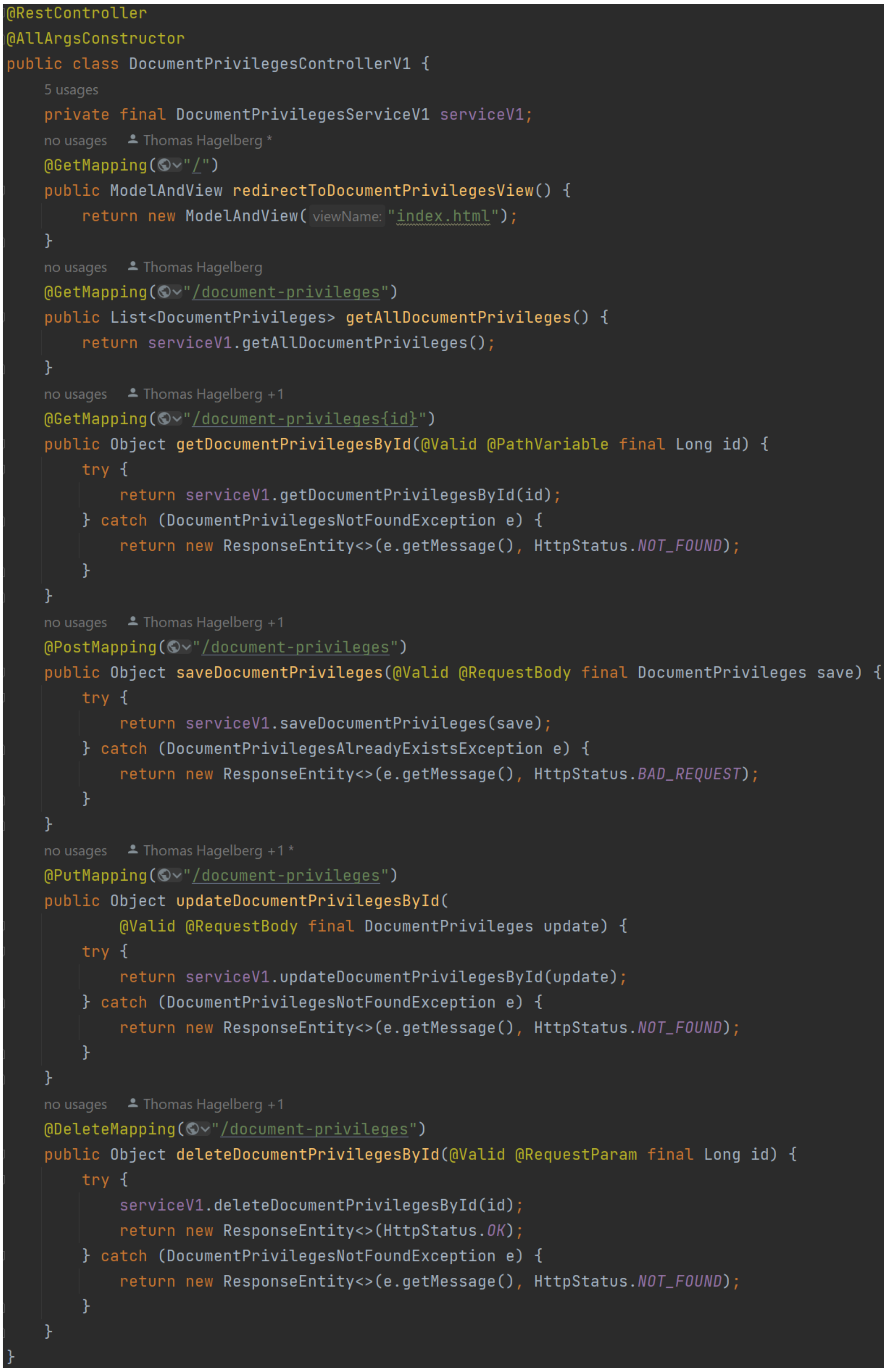

```
@RestController
@AllArgsConstructor
public class DocumentPrivilegesControllerV1 {
    5 usages
    private final DocumentPrivilegesServiceV1 serviceV1;
    no usages   Thomas Hagelberg *
    @GetMapping("/")
    public ModelAndView redirectToDocumentPrivilegesView() {
        return new ModelAndView( viewName: "index.html");
    }
    no usages   Thomas Hagelberg
    @GetMapping("/document-privileges")
    public List<DocumentPrivileges> getAllDocumentPrivileges() {
        return serviceV1.getAllDocumentPrivileges();
    }
    no usages   Thomas Hagelberg +1
    @GetMapping("/document-privileges{id}")
    public Object getDocumentPrivilegesById(@Valid @PathVariable final Long id) {
        try {
            return serviceV1.getDocumentPrivilegesById(id);
        } catch (DocumentPrivilegesNotFoundException e) {
            return new ResponseEntity<>(e.getMessage(), HttpStatus.NOT_FOUND);
        }
    }
    no usages   Thomas Hagelberg +1
    @PostMapping("/document-privileges")
    public Object saveDocumentPrivileges(@Valid @RequestBody final DocumentPrivileges save) {
        try {
            return serviceV1.saveDocumentPrivileges(save);
        } catch (DocumentPrivilegesAlreadyExistsException e) {
            return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
        }
    }
    no usages   Thomas Hagelberg +1 *
    @PutMapping("/document-privileges")
    public Object updateDocumentPrivilegesById(
            @Valid @RequestBody final DocumentPrivileges update) {
        try {
            return serviceV1.updateDocumentPrivilegesById(update);
        } catch (DocumentPrivilegesNotFoundException e) {
            return new ResponseEntity<>(e.getMessage(), HttpStatus.NOT_FOUND);
        }
    }
    no usages   Thomas Hagelberg +1
    @DeleteMapping("/document-privileges")
    public Object deleteDocumentPrivilegesById(@Valid @RequestParam final Long id) {
        try {
            serviceV1.deleteDocumentPrivilegesById(id);
            return new ResponseEntity<>(HttpStatus.OK);
        } catch (DocumentPrivilegesNotFoundException e) {
            return new ResponseEntity<>(e.getMessage(), HttpStatus.NOT_FOUND);
        }
    }
}
```

Figure 5. Example of a controller class in Java.

2.2 REST

REST (Representational State Transfer) is an architectural style that defines the way of building web services. REST is based on these key principles provided below: (IBM, n.d.)

1. **Uniform interface**. “All API requests for the same resource should look the same, no matter where the request comes from. The REST API should ensure that the same piece of data, such as the name or email address of a user, belongs to only one uniform resource identifier (URI). Resources shouldn’t be too large but should contain every piece of information that the client might need.”

2. **Client-server decoupling**. “In REST API design, client and server applications must be completely independent of each other. The only information the client application should know is the URI of the requested resource; it can't interact with the server application in any other ways. Similarly, a server application shouldn't modify the client application other than passing it to the requested data via HTTP.”

3. **Statelessness**. “REST APIs are stateless, meaning that each request needs to include all the information necessary for processing it. In other words, REST APIs do not require any server-side sessions. Server applications aren’t allowed to store any data related to a client request.”

4. **Cacheability**. “When possible, resources should be cacheable on the client or server side. Server responses also need to contain information about whether caching is allowed for the delivered resource. The goal is to improve performance on the client side, while increasing scalability on the server side.”

5. **Layered system architecture**.” In REST APIs, the calls and responses go through different layers. As a rule of thumb, don’t assume that the client and server applications connect directly to each other. There may be a number of different intermediaries in the communication loop. REST APIs need to be designed so that

neither the client nor the server can tell whether it communicates with the end application or an intermediary.”

2.2.1 RESTful

RESTful refers to web services and applications that implement the principles of REST. The following are key components of RESTful APIs (Amazon, 2023).

2.2.2 Resources and URIs

- A resource is an object or representation of something
- A URI is a unique identifier that represents a resource

2.2.3 HTTP Methods

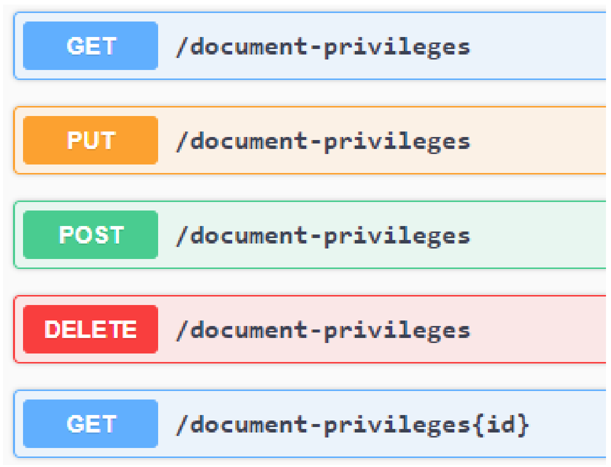
- GET: retrieves a resource
- POST: creates a new resource
- PUT: updates an existing resource
- DELETE: deletes a resource

2.2.4 Common Response Codes

- 200 OK: request was successful
- 201 Created: new resource was successfully created
- 400 Bad Request: request was invalid or cannot be fulfilled
- 401 Unauthorized: authentication failed or user does not have permissions
- 403 Forbidden: authentication succeeded, but the authenticated user does not have access to the resource
- 404 Not Found: resource was not found
- 500 Internal Server Error: an error occurred on the server.

2.2.5 RESTful Example

In figure 6, an example is shown of how a set of RESTful endpoints could look.



```
GET     /document-privileges
PUT     /document-privileges
POST    /document-privileges
DELETE  /document-privileges
GET     /document-privileges{id}
```

Figure 6. RESTful endpoints displaying the HTTP methods and URIs for the document-privileges resource

2.3 Inversion of Control

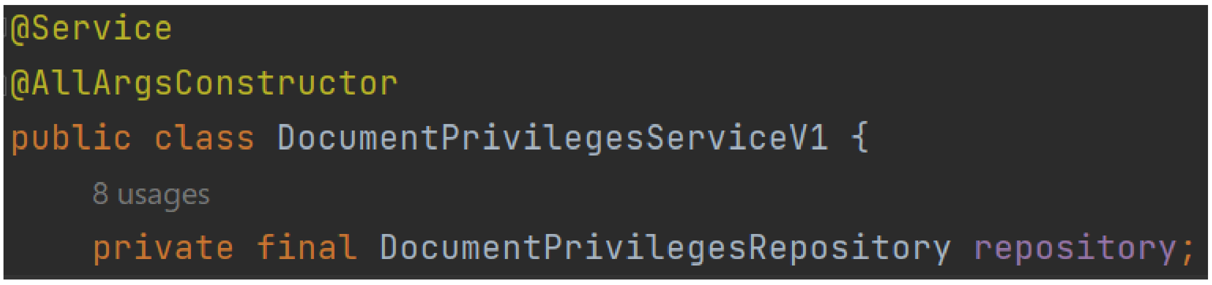
Inversion of Control (IoC) is a design pattern that changes the way in which the control flow of a program is determined. Instead of the program controlling the flow of execution, the flow of execution is controlled by an external entity, such as a framework or container. This pattern is often used to increase flexibility and testability of the code.

There are two main types of IoC:

- Dependency Injection (DI) - the objects that a component depends on are passed to it by an external entity (often a framework) at runtime.
- Service Locator - the component looks up its dependencies from a central registry, rather than having them passed to it.

IoC is often a key part of frameworks that are designed to be extensible and reusable (Baeldung, 2019). It allows developers to create a more flexible and testable codebase, it is often implemented in frameworks such as Spring, AngularJS, and many others.

Figure 7 gives an example of how a Java class is annotated with the Spring Boot "@Service" and Lombok "@AllArgsConstructor" annotations. The Service annotation specifies that the class is a service and a bean should be created for that class which can be used for dependency injection. The AllArgsConstructor annotation on the other hand specifies that all parameters for a class should, if needed, be autowired. Autowiring is a way of connecting parameters with a suitable bean, and stopping the application if no suitable bean can be found.



```
@Service
@AllArgsConstructor
public class DocumentPrivilegesServiceV1 {
    8 usages
    private final DocumentPrivilegesRepository repository;
```

Figure 7. Spring and Lombok annotations

2.4 Serverless

In a serverless architecture, the cloud provider is responsible for the management of the infrastructure, including "provisioning, scaling, and maintaining servers and infrastructure components" (AWS, 2023b). Developers can build and deploy their applications as small, independent functions that run in response to specific events, such as a user request or a message in a queue. The cloud provider then runs these functions as needed, allocating and deallocating resources dynamically to match demand. This means that developers only pay for the resources they actually use, rather than having to provision and manage servers continuously.

The serverless design pattern can be used to build a wide range of applications, including "web and mobile applications, real-time data processing pipelines, and backends for IoT devices" (AWS, 2023b). It provides a highly scalable, flexible, and cost-effective way to build and run applications, and is becoming increasingly popular due to its simplicity and efficiency.

2.4.1 AWS Fargate

AWS Fargate is a serverless compute engine for containers that eliminates the need to manage infrastructure and allows developers to focus on building and deploying their applications.

Fargate is the compute engine that was agreed to use when developing the new API, mostly because the serverless architecture is on the road to become the industry standard. Therefore, learning and experimenting with its features would be really valuable for the team. Figure 8 displays the difference between using AWS' EC2 servers and fargate.

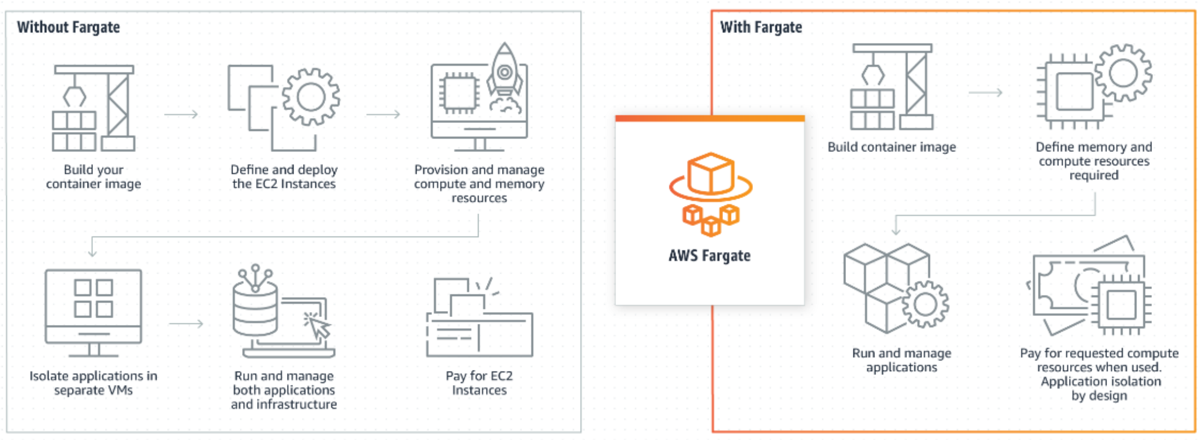

Without Fargate
Build your container image
Define and deploy the EC2 Instances
Provision and manage compute and memory resources
Isolate applications in separate VMs
Run and manage both applications and infrastructure
Pay for EC2 Instances
AWS Fargate
With Fargate
Build container image
Define memory and compute resources required
Run and manage applications
Pay for requested compute resources when used. Application isolation by design

Figure 8. Difference between AWS EC2 and Fargate (AWS, 2023a)

3. FRAMEWORKS AND TOOLS

3.1 Spring

Spring is an open-source, Java-based framework that provides a comprehensive programming and configuration model for modern Java-based enterprise applications. According to the Spring documentation (Spring, 2023a), it is designed to help developers build high-performing, easily testable, and reusable code.

One of the key features of the Spring framework is its dependency injection (DI) and inversion of control (IoC) container. The DI container manages the components that make up an application, and the IoC container manages the flow of control between them (Spring, 2023a). This helps to make the code more modular and easier to test and maintain.

The Spring framework also includes a wide range of modules for tasks such as data access, transaction management, and web services (TechTarget, 2019). These modules are designed to work together seamlessly, and they can be used in any combination to create a customized application.

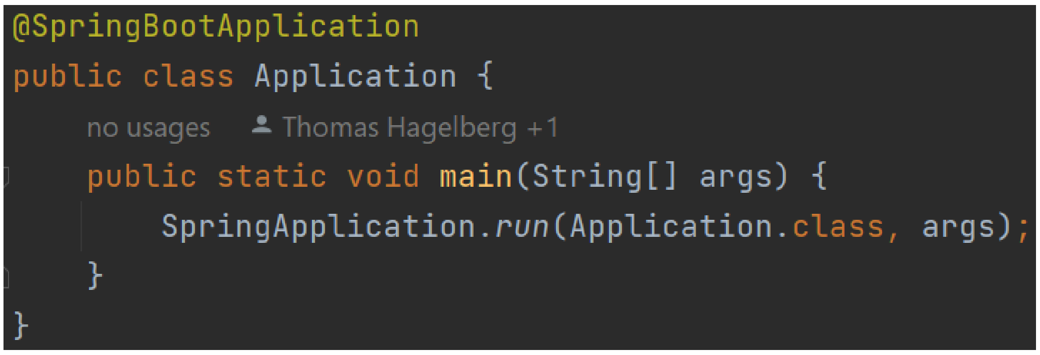
Another important aspect of the Spring framework is its support for aspect-oriented programming (AOP) (Spring, 2023a). AOP allows developers to modularize cross-cutting concerns such as security, logging, and caching, which can be applied to multiple parts of the application without affecting the core business logic. Overall, the Spring framework is a popular choice for enterprise Java development because of its flexibility, ease of use, and ability to integrate with other frameworks and technologies (TechTarget, 2019).

3.1.1 Spring Boot

Spring Boot is a Java-based framework for building and deploying cloud-native applications. It provides an easy way to create standalone, production-grade applications and services efficiently. Spring Boot simplifies the process of creating Spring-powered applications and services by eliminating much of the boilerplate code and configuration that is typically required (Microsoft Azure, 2023).

Spring Boot provides a number of features to simplify the development and deployment of cloud-native applications. These features include auto-configuration, embedded web servers, command-line tools, and pre-configured templates for common use cases (Microsoft Azure, 2023).

By using Spring Boot, developers can focus on writing the business logic for their application, while the framework takes care of the underlying infrastructure and configurations. This makes it easier to get started with developing and deploying cloud-native applications and services, and reduces the time and effort required to get an application up and running. Figure 9 shows the “@SpringBootApplication” annotation that is used to enable the Spring Boot functionality in a Java application class.



```
@SpringBootApplication
public class Application {
    no usages   Thomas Hagelberg +1
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Figure 9. The "@SpringBootApplication" annotation in a Java application class

3.1.2 Spring Native

Spring Native is a set of tools and technologies for building cloud-native applications using the Spring framework (Spring, 2023b). It provides a way for developers to build and run Spring-based applications in a cloud-native environment, taking advantage of the scalability, reliability, and performance of cloud computing.

The Spring Native documentation states that it provides a number of features and tools to simplify the development and deployment of cloud-native applications such as support for microservices architecture, containerization, and orchestration (Spring, 2023b). This makes it easier for developers to build and deploy cloud-native applications that are scalable, reliable, and performant.

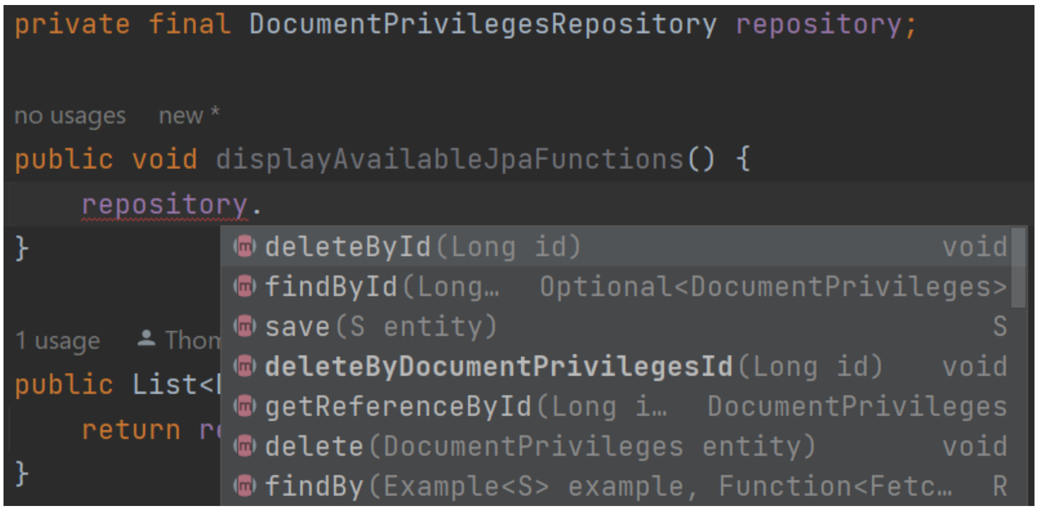
Ahead of Time (AOT) processing is a key feature of the Spring Native framework. It is designed to generate machine code that can be executed by the operating system, without the need for a separate runtime environment (Spring, 2021). This allows for faster startup times and reduced memory usage, as the application does not have to load the runtime environment into memory before it can start processing requests.

3.2 Java Persistence API

The Java Persistence API (JPA) is a Java application programming interface (API) specification that provides a standard way to access relational databases in Java applications. JPA is an API specification that provides a way to access relational databases in a standard way for Java applications (JavaTPoint, n.d.). JPA is part of the Java Enterprise Edition (Java EE) platform and is designed to simplify the storage and retrieval of Java objects from relational databases.

JPA provides a set of APIs for defining the structure of a database, as well as for querying and updating the data in that database (IBM, 2023). JPA can be used with a variety of relational databases, including relational databases such as Oracle, MySQL, and Microsoft SQL Server.

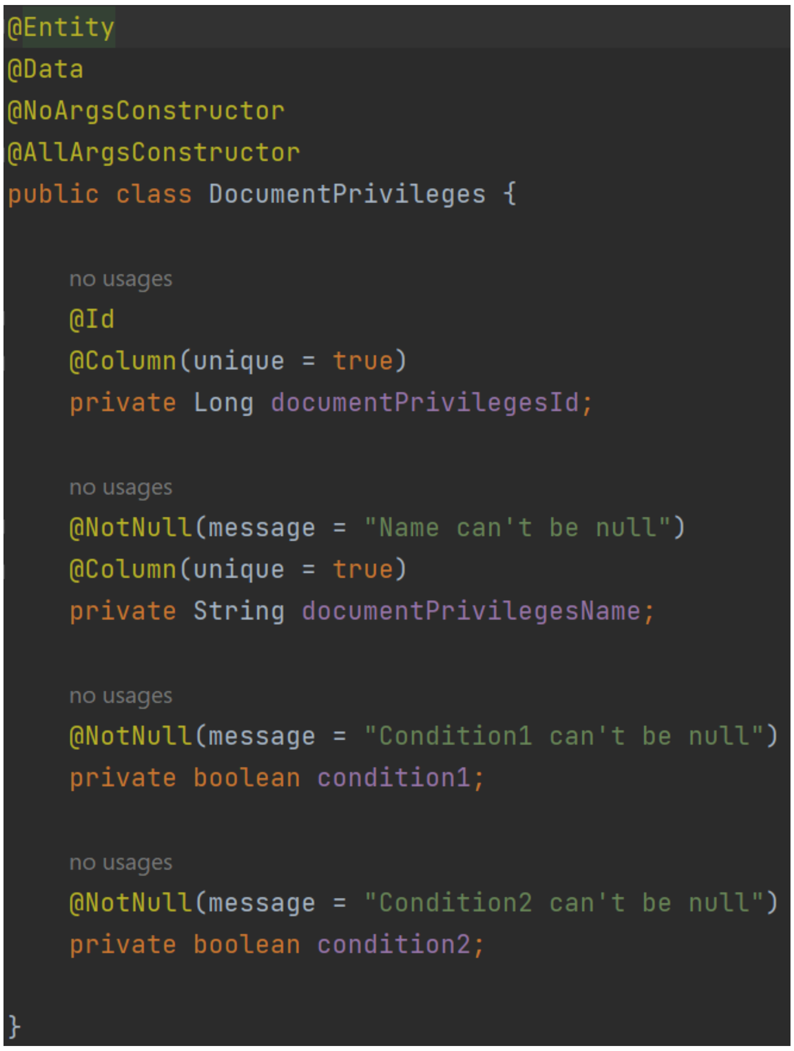
JPA is implemented through Object Relational Mapping (ORM), which is a technique for mapping data from a relational database to objects in a Java program, and vice versa. ORM allows developers to write Java code to interact with the database, rather than writing SQL statements, which can simplify the development process (IBM, 2023). In figure 10, a few examples are shown of what functionality is available per default in JPA without creating any custom SQL functions and queries.



```
private final DocumentPrivilegesRepository repository;

no usages   new *
public void displayAvailableJpaFunctions() {
    repository.
}             deleteById(Long id)                               void
              findById(Long…   Optional<DocumentPrivileges>
1 usage  Thon save(S entity)                                       S
public List<l deleteByDocumentPrivilegesId(Long id)             void
    return r  getReferenceById(Long i…   DocumentPrivileges
}             delete(DocumentPrivileges entity)                 void
              findBy(Example<S> example, Function<Fetc…            R
```

Figure 10. Displaying a few functions for handling queries using a Repository that extends JpaRepository

The repository itself is created by a Java class utilizing the “@Entity” annotation, that is shown in Figure 11. The “@Data” annotation provides basic getter and setter methods, toString and hash functions, while the “@NoArgsConstructor” and “@AllArgsConstructor” creates class constructor functions. The “@Data”, “@NoArgsConstructor” and “@AllArgsConstructor” annotations are part of the Lombok artifact, and are not included with JPA.

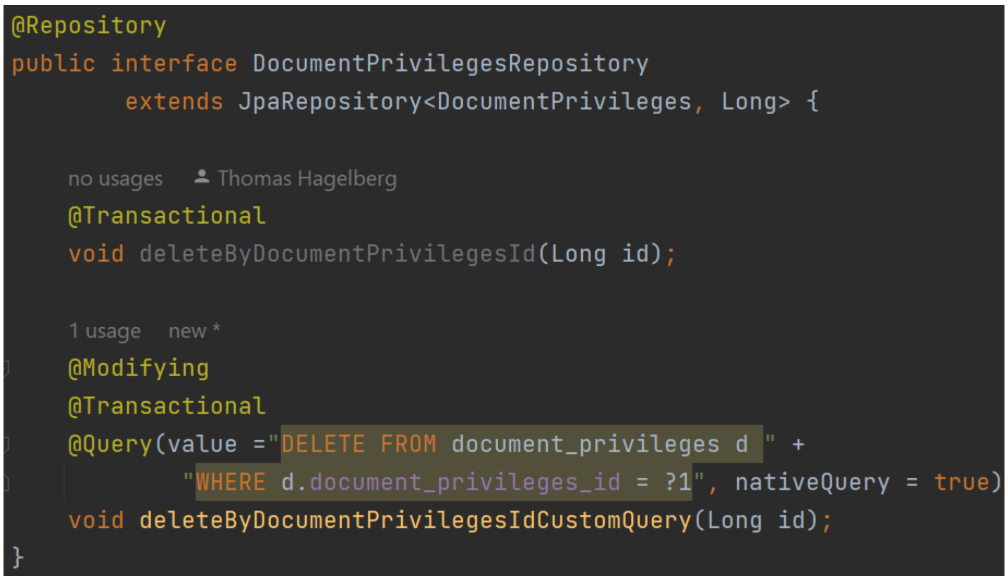


```
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class DocumentPrivileges {

    no usages
    @Id
    @Column(unique = true)
    private Long documentPrivilegesId;

    no usages
    @NotNull(message = "Name can't be null")
    @Column(unique = true)
    private String documentPrivilegesName;

    no usages
    @NotNull(message = "Condition1 can't be null")
    private boolean condition1;

    no usages
    @NotNull(message = "Condition2 can't be null")
    private boolean condition2;

}
```

Figure 11. Example of an Entity class

When using a repository that should use JPA we extend the ‘JpaRepository’ class and specify what entity should be used, and what class should be used as a primary key.

A few included functions were already shown in figure 10, but it is also possible to create custom queries. They can simply be structured using text, but that is not “best practices” from a security perspective, due to the risk of potential SQL injection attacks. Therefore we can also create custom queries with “prepared statements”, which is much safer. Figure 12 shows an example of this.



```
@Repository
public interface DocumentPrivilegesRepository
        extends JpaRepository<DocumentPrivileges, Long> {

    no usages   Thomas Hagelberg
    @Transactional
    void deleteByDocumentPrivilegesId(Long id);

    1 usage   new *
    @Modifying
    @Transactional
    @Query(value ="DELETE FROM document_privileges d " +
            "WHERE d.document_privileges_id = ?1", nativeQuery = true)
    void deleteByDocumentPrivilegesIdCustomQuery(Long id);
}
```

Figure 12. Example of a Repository class that is extending the JpaRepository, with two custom queries.

3.3 Swagger

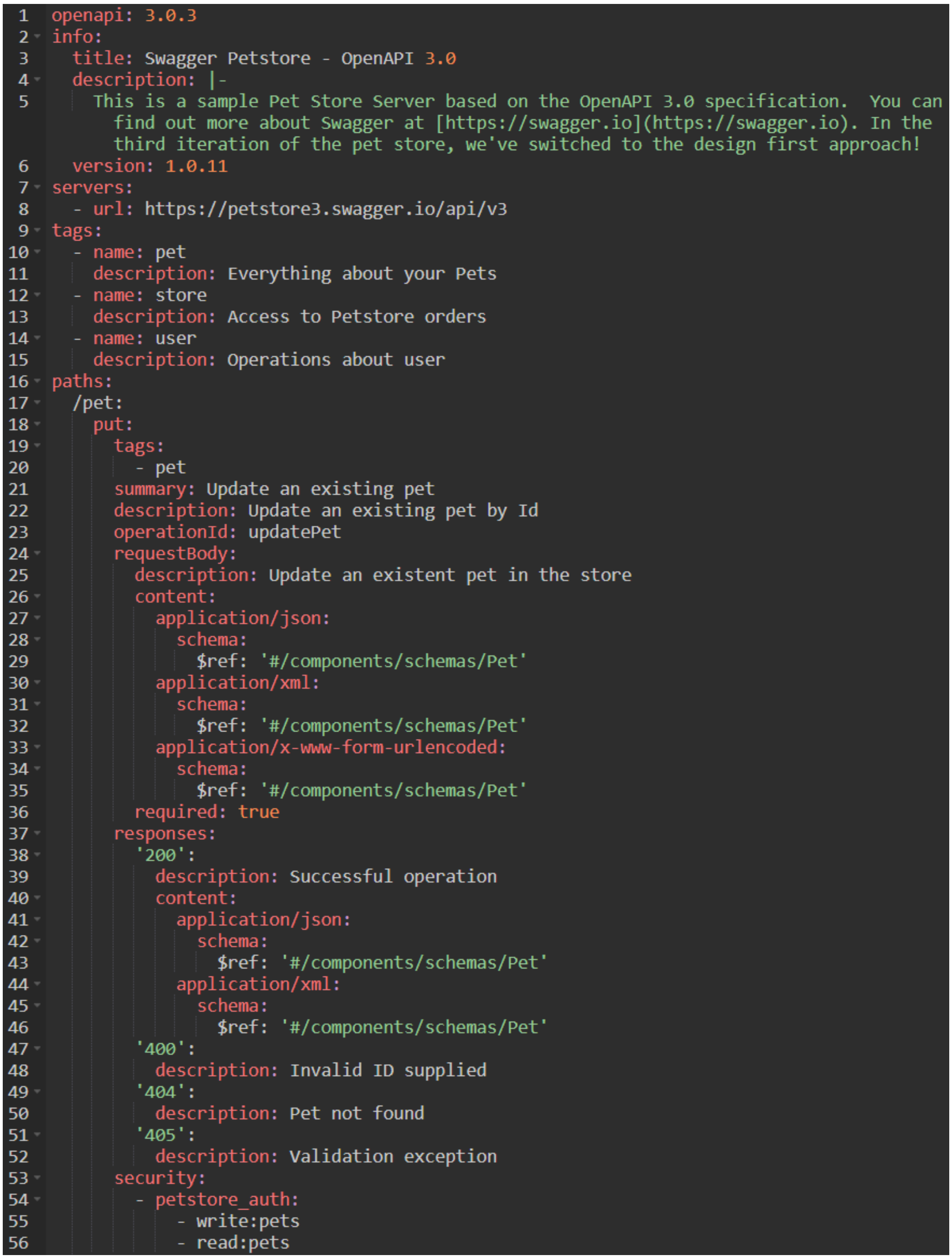
Swagger is a set of open-source tools built around the OpenAPI Specification that can help the user design, build, document and consume REST APIs (Swagger, n.d.-d). Swagger provides a user-friendly interface for creating and testing APIs, as well as generating API documentation automatically. This makes it easier for developers to collaborate and communicate about the APIs they are building, and for API consumers to understand how to use the APIs.

Swagger is a machine-readable representation of a RESTful Web API that enables a client to understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic (TechTarget, n.d.).

3.3.1 OpenAPI Specification

The OpenAPI Specification (OAS) provides a uniform way of describing RESTful APIs, regardless of the programming technology or framework utilized to build the API. This makes it more convenient for developers to work with APIs and for API consumers to comprehend how to use the API (Swagger, n.d.-a).

The OAS comprises various elements that can be used to describe the API, such as endpoints, request and response bodies, parameters, and security definitions. The specification is adaptable, allowing developers to include various levels of detail in describing their API. This feature allows for customization and tailored description of the API based on specific needs and requirements. Figure 13 shows an example of an openAPI specification, or contract, as some may call it.



```yaml
openapi: 3.0.3
info:
  title: Swagger Petstore - OpenAPI 3.0
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification.  You can
      find out more about Swagger at [https://swagger.io](https://swagger.io). In the
      third iteration of the pet store, we've switched to the design first approach!
  version: 1.0.11
servers:
  - url: https://petstore3.swagger.io/api/v3
tags:
  - name: pet
    description: Everything about your Pets
  - name: store
    description: Access to Petstore orders
  - name: user
    description: Operations about user
paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
      operationId: updatePet
      requestBody:
        description: Update an existent pet in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
          application/x-www-form-urlencoded:
            schema:
              $ref: '#/components/schemas/Pet'
        required: true
      responses:
        '200':
          description: Successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
            application/xml:
              schema:
                $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid ID supplied
        '404':
          description: Pet not found
        '405':
          description: Validation exception
      security:
        - petstore_auth:
            - write:pets
            - read:pets
```

Figure 13. Example of an openAPI specification (Swagger, n.d.-b)

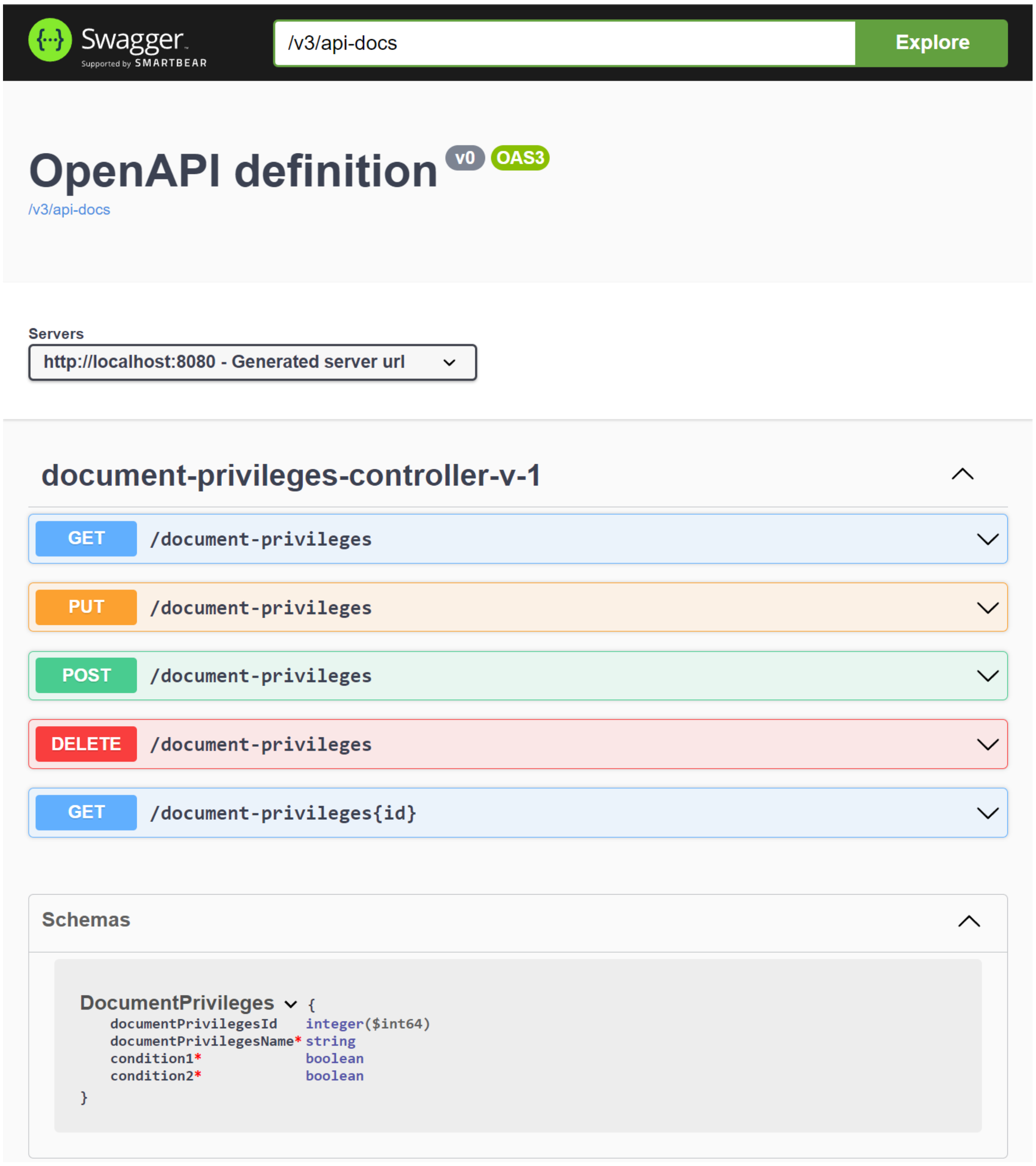
In addition to its fundamental features, the OAS provides a structure for testing APIs and generating API documentation. This simplifies the process for developers to create and maintain dependable, well-documented APIs that are user-friendly and easy to comprehend for others.

3.3.2 Swagger UI

Swagger UI is a collection of HTML, JavaScript, and CSS assets that dynamically generate documentation from a Swagger-compliant API. This documentation provides a comprehensive overview of the API's resources, operations, and parameters, making it easier for developers to understand how to use the API (Swagger, n.d.-c).

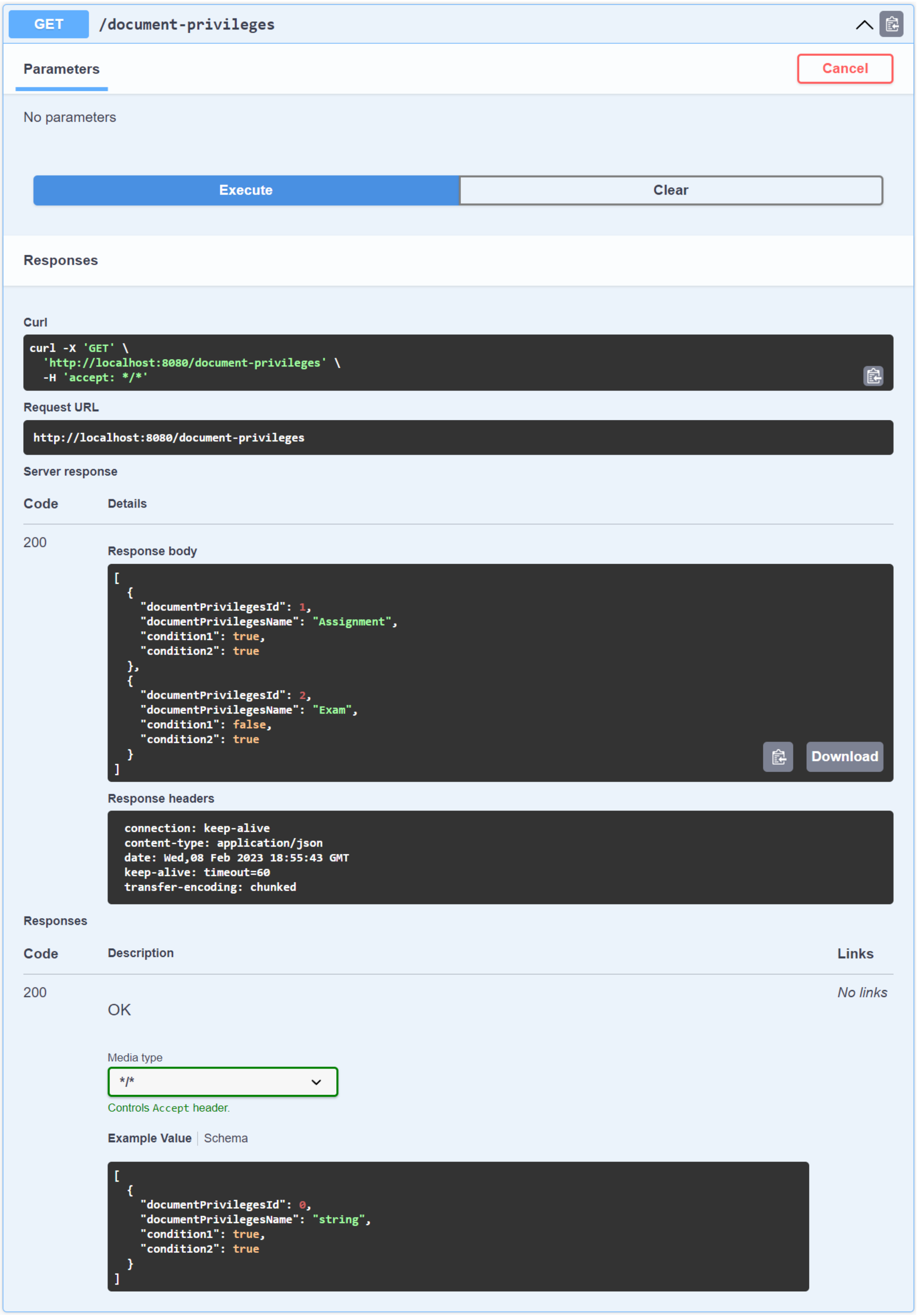
Swagger UI provides an intuitive, interactive way to explore the API and its capabilities, making it an essential tool for any developer working with Swagger-based APIs. With Swagger UI, developers can quickly test and debug the API, reducing the time and effort required to implement the API in their own applications.

Figure 14 shows how the Swagger UI overview looks for the Document Privileges example RESTful API endpoints, and the respective schema (object) that is available. Figure 15 illustrates how the user can interact and try out the specific endpoints.



```
DocumentPrivileges {
    documentPrivilegesId    integer($int64)
    documentPrivilegesName* string
    condition1*             boolean
    condition2*             boolean
}
```

Figure 14. Swagger UI overview for the endpoints of Document Privileges example functionality.



Curl

```
curl -X 'GET' \
  'http://localhost:8080/document-privileges' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/document-privileges
```

Server response

Code | Details

200

Response body

```
[
  {
    "documentPrivilegesId": 1,
    "documentPrivilegesName": "Assignment",
    "condition1": true,
    "condition2": true
  },
  {
    "documentPrivilegesId": 2,
    "documentPrivilegesName": "Exam",
    "condition1": false,
    "condition2": true
  }
]
```

Response headers

```
connection: keep-alive
content-type: application/json
date: Wed,08 Feb 2023 18:55:43 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Responses

Code	Description	Links
200	OK	*No links*

Media type

/

Controls Accept header.

Example Value | Schema

```
[
  {
    "documentPrivilegesId": 0,
    "documentPrivilegesName": "string",
    "condition1": true,
    "condition2": true
  }
]
```

Figure 15. Swagger UI interaction with the /document-privileges endpoint.

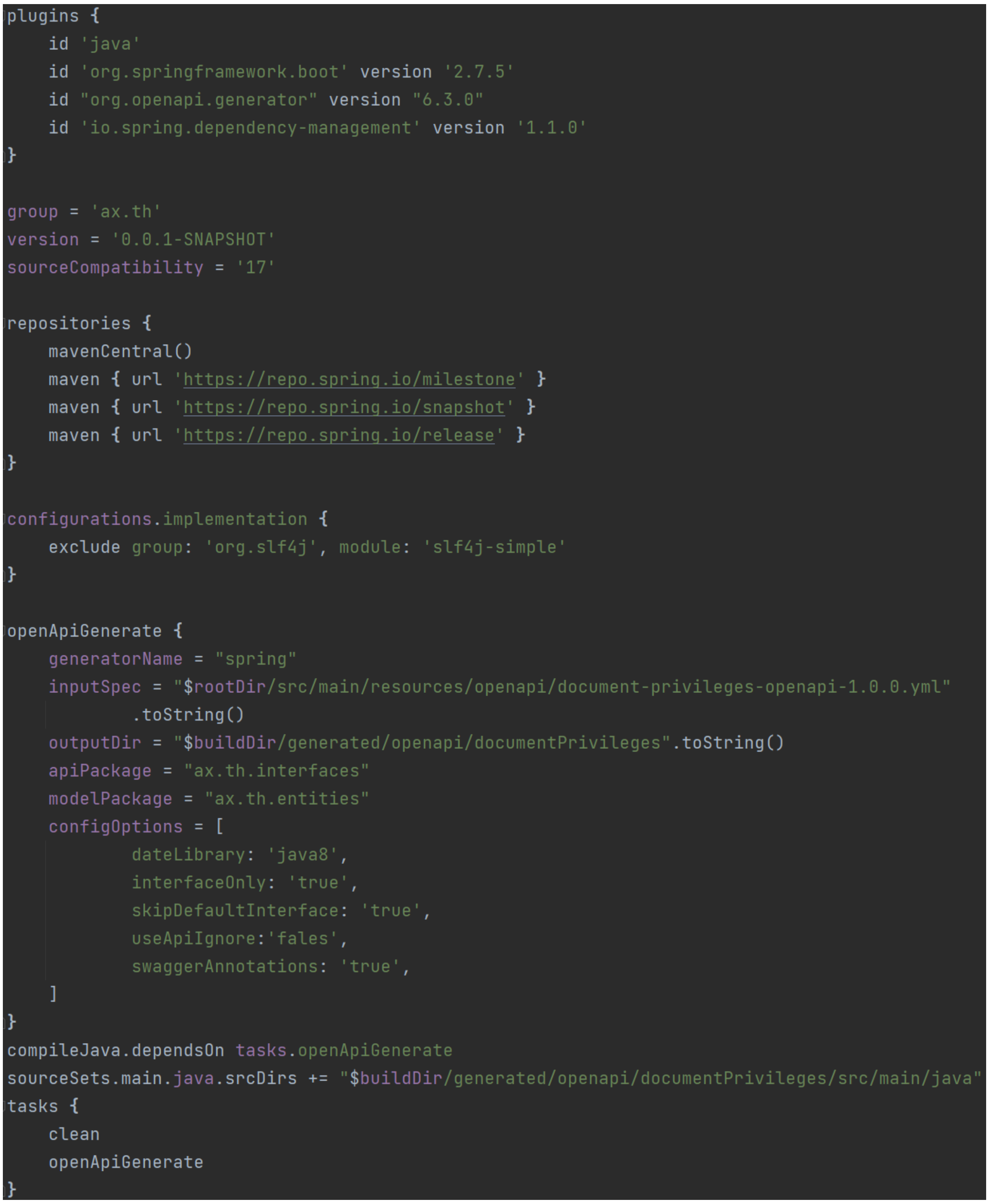
3.4 Gradle

Gradle is an open-source build automation tool for Java projects that is designed to be flexible, efficient, and easy to use. It builds upon the concepts of Apache Ant and Apache Maven and introduces a Groovy-based domain-specific language (DSL) instead of the XML form used by Apache Maven for declaring the project configuration (Gradle, n.d.).

Gradle helps automate the process of building, testing, and deploying Java applications by providing a flexible and scalable build system that can be easily integrated with other tools. The tool is particularly well suited to large, complex projects with many dependencies, as it can handle the complexity of managing these dependencies and building the project efficiently.

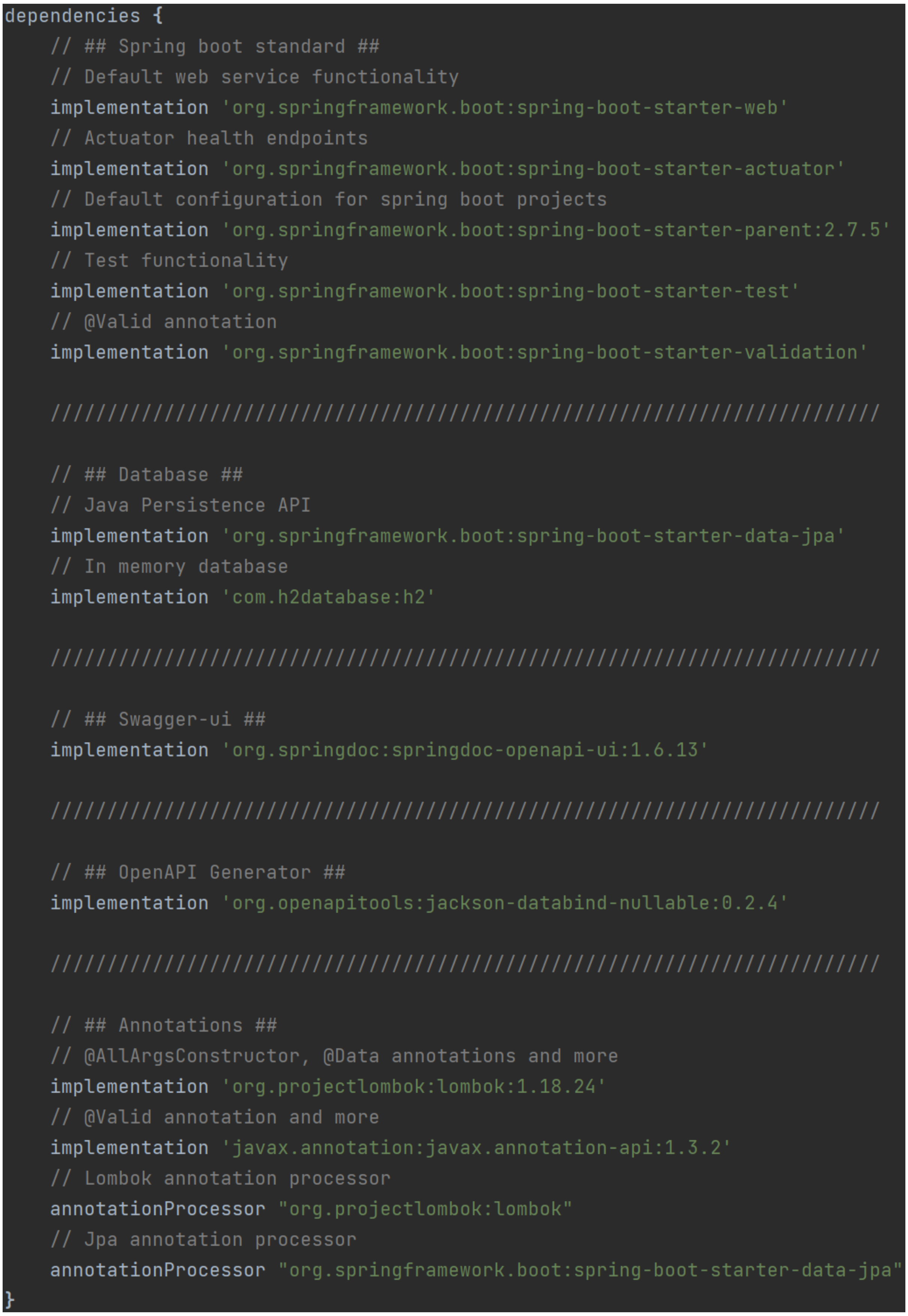
Gradle is designed to be highly configurable and extensible, allowing developers to customize its behavior to fit their specific needs. This is achieved through the use of plugins, which can be used to add new features or extend the capabilities of Gradle. Gradle plugins are the preferred way to add new functionality to Gradle. With plugins, it is possible to extend the core build logic and add new features, tasks and configurations to Gradle (Gradle, n.d.).

Overall, Gradle is a powerful and versatile tool that can help streamline the build process for Java projects and make it easier for developers to focus on coding, rather than worrying about the build process. Figure 16 and 17 displays how the Document Privileges example project is structured. Figure 16 shows the basic configuration including plugins, repositories, Java version and more, while figure 17 shows how Gradle can be used to load external artifacts into the application using dependencies.



```
plugins {
    id 'java'
    id 'org.springframework.boot' version '2.7.5'
    id "org.openapi.generator" version "6.3.0"
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'ax.th'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/milestone' }
    maven { url 'https://repo.spring.io/snapshot' }
    maven { url 'https://repo.spring.io/release' }
}

configurations.implementation {
    exclude group: 'org.slf4j', module: 'slf4j-simple'
}

openApiGenerate {
    generatorName = "spring"
    inputSpec = "$rootDir/src/main/resources/openapi/document-privileges-openapi-1.0.0.yml"
            .toString()
    outputDir = "$buildDir/generated/openapi/documentPrivileges".toString()
    apiPackage = "ax.th.interfaces"
    modelPackage = "ax.th.entities"
    configOptions = [
            dateLibrary: 'java8',
            interfaceOnly: 'true',
            skipDefaultInterface: 'true',
            useApiIgnore:'fales',
            swaggerAnnotations: 'true',
    ]
}
compileJava.dependsOn tasks.openApiGenerate
sourceSets.main.java.srcDirs += "$buildDir/generated/openapi/documentPrivileges/src/main/java"
tasks {
    clean
    openApiGenerate
}
```

Figure 16. Basic Gradle configuration for the Document Privileges example application.



```
dependencies {
    // ## Spring boot standard ##
    // Default web service functionality
    implementation 'org.springframework.boot:spring-boot-starter-web'
    // Actuator health endpoints
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    // Default configuration for spring boot projects
    implementation 'org.springframework.boot:spring-boot-starter-parent:2.7.5'
    // Test functionality
    implementation 'org.springframework.boot:spring-boot-starter-test'
    // @Valid annotation
    implementation 'org.springframework.boot:spring-boot-starter-validation'

    //////////////////////////////////////////////////////////////////////////

    // ## Database ##
    // Java Persistence API
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    // In memory database
    implementation 'com.h2database:h2'

    //////////////////////////////////////////////////////////////////////////

    // ## Swagger-ui ##
    implementation 'org.springdoc:springdoc-openapi-ui:1.6.13'

    //////////////////////////////////////////////////////////////////////////

    // ## OpenAPI Generator ##
    implementation 'org.openapitools:jackson-databind-nullable:0.2.4'

    //////////////////////////////////////////////////////////////////////////

    // ## Annotations ##
    // @AllArgsConstructor, @Data annotations and more
    implementation 'org.projectlombok:lombok:1.18.24'
    // @Valid annotation and more
    implementation 'javax.annotation:javax.annotation-api:1.3.2'
    // Lombok annotation processor
    annotationProcessor "org.projectlombok:lombok"
    // Jpa annotation processor
    annotationProcessor "org.springframework.boot:spring-boot-starter-data-jpa"
}
```

Figure 17. Gradle dependencies towards external artifacts.

3.5 Jenkins

Jenkins is an open-source automation server designed to support continuous integration and continuous delivery (CI/CD) of software projects. Jenkins provides hundreds of plugins to support building, deploying, and automating any project (Jenkins, n.d.).

Jenkins provides a web interface for configuring, managing and viewing the logs of builds and allows developers to automate various stages of the software development lifecycle, including building, testing, and deployment. This helps to ensure that software is delivered quickly and efficiently, while also reducing the risk of errors and bugs. The use of Jenkins also allows developers to focus on writing code, as opposed to manual and repetitive tasks such as building and testing. Jenkins provides a web interface for configuring, managing and viewing the logs of builds.

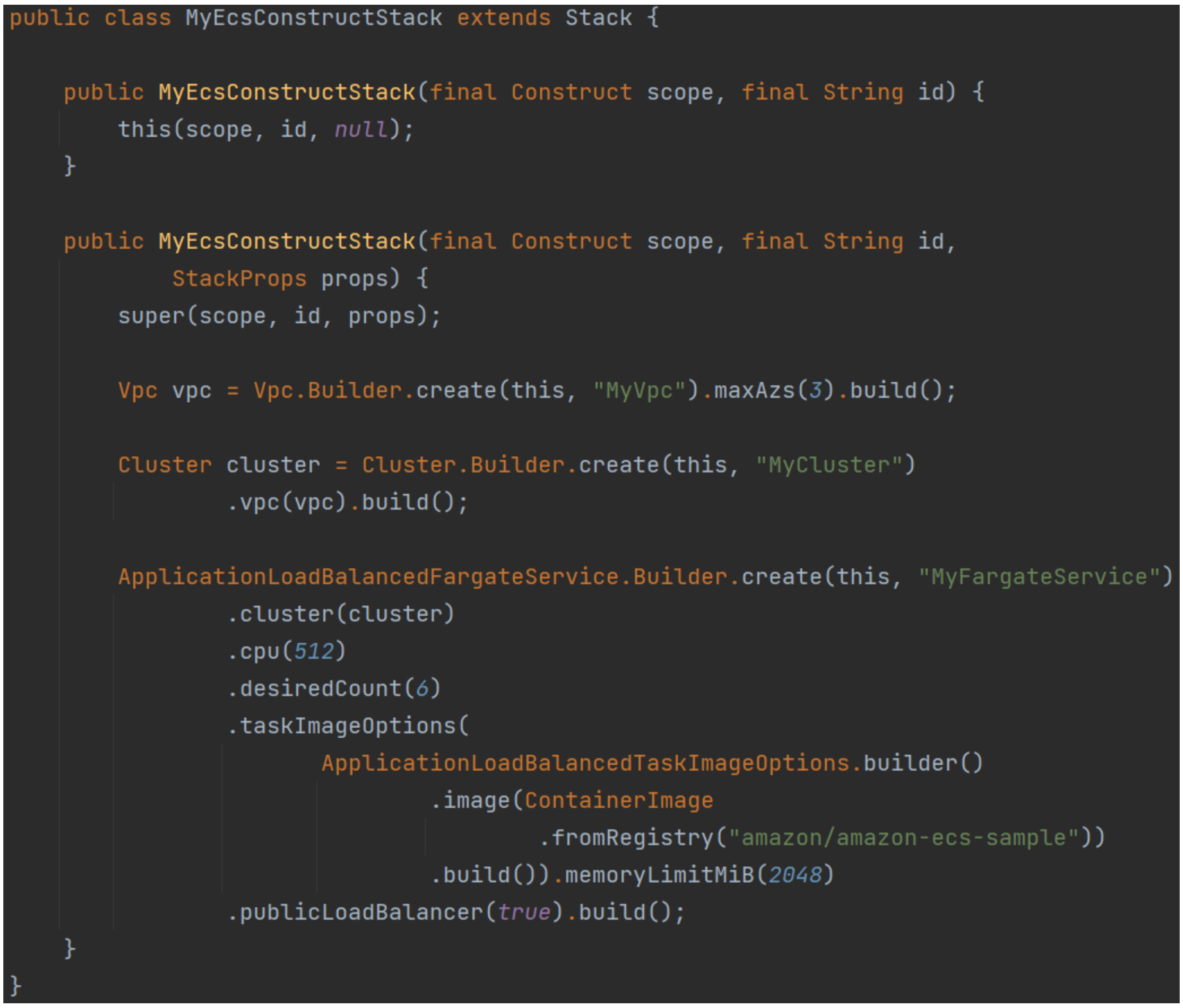
Jenkins supports a wide range of programming languages and technologies, making it a flexible tool that can be used in a variety of environments. The platform can be easily integrated with other tools, such as Git, JIRA, and Selenium, which allows for a seamless and streamlined software development process.

3.6 AWS Cloud Development Kit

The AWS Cloud Development Kit (AWS CDK) is a software development framework for defining cloud infrastructure as code and provisioning it through AWS CloudFormation. It uses familiar programming languages, including JavaScript, TypeScript, Python, Java, and C#, and deploys everything as a cloud formation stack (AWS, n.d.).

The AWS CDK allows developers to define and manage cloud resources using familiar programming constructs, making it easier for developers to define, provision, and update cloud infrastructure. The AWS CDK also provides a high level of abstraction, allowing developers to define cloud infrastructure in terms of cloud components and relationships, rather than low-level infrastructure as code.

The AWS CDK supports a range of AWS services, including AWS RDS, AWS EC2, AWS ECS and AWS Fargate, as well as custom resources and AWS CloudFormation macros. Figure 18 shows an example of how an AWS ECS service with AWS Fargate launch type can be defined using Java code, which is utilizing the AWS CDK framework (AWS, n.d.).



```java
public class MyEcsConstructStack extends Stack {

    public MyEcsConstructStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyEcsConstructStack(final Construct scope, final String id,
            StackProps props) {
        super(scope, id, props);

        Vpc vpc = Vpc.Builder.create(this, "MyVpc").maxAzs(3).build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                .vpc(vpc).build();

        ApplicationLoadBalancedFargateService.Builder.create(this, "MyFargateService")
                .cluster(cluster)
                .cpu(512)
                .desiredCount(6)
                .taskImageOptions(
                        ApplicationLoadBalancedTaskImageOptions.builder()
                                .image(ContainerImage
                                        .fromRegistry("amazon/amazon-ecs-sample"))
                                .build()).memoryLimitMiB(2048)
                .publicLoadBalancer(true).build();
    }
}
```

Figure 18. Example of how a stack can be defined in Java using AWS CDK framework (AWS, n.d.).

Overall, the AWS CDK is a powerful and flexible tool for defining and provisioning cloud infrastructure, and it makes it easier for developers to manage and automate the deployment of their cloud infrastructure (AWS, n.d.).

4. DEVELOPMENT PROCESS

4.1 Preparation

4.1.1 Initial meetings

This project began as a discussion meeting in late September 2022, where the product owner, application managers, and system architect were present. The objective of the meeting was to identify a suitable project for me. The most relevant work identified was to develop a new microservice - a RESTful API that would be based on a serverless architecture.

Although the exact description of what the API should do was not defined, document-related privileges were the general idea. This resulted in a follow-up meeting with the application managers, where we discussed the other architectural aspects of the application. The primary objectives were to discuss:

- The purpose of the API
- Who will use it
- The technologies that should be used

As a result of the meeting, the API was designed to provide Crosskey's new Backoffice system with different types of privileges for documents based on their sub-category. In other words, it defined what users were allowed to do with different types of documents, such as deleting them, reinstating them, or identifying the document's creator.

The technology stack was thoroughly discussed, and the output was:

- Spring Boot
- RESTful API
- Oracle Database
- Serverless (AWS Fargate)
- AWS Secrets Manager

4.1.2 Scope definition

The project’s vision was now clear, and I had a better understanding of what we were aiming for. The next step was to break down the project into smaller tasks in an agile environment where Scrum was the methodology, and Jira was the technology used for managing and tracking the development process. The entire project was defined as an “epic” in Jira terminology, which contained the project description and all tasks, or “stories” that needed to be done in order to be ready for release. The epic is used as an overview to see the result of a project and simplifies the administration.

4.2 Development

4.2.1 Initial Startup

The initial story was to set up the project, create a repository, generate a code skeleton, and create some basic endpoint. Although initially considered easy, it was challenging because the thought was to copy the structure from an existing project into this one. However, different projects have different setups and configurations, and all these configurations were unclear to me at the time. I had never developed an entire enterprise application from scratch before, so this was a considerable challenge.

Numerous discussions were held with colleagues who had expertise in the difficulties I was facing. For each problem, I searched for a detailed explanation to understand the root cause behind it.

4.2.2 Spring Native Investigation

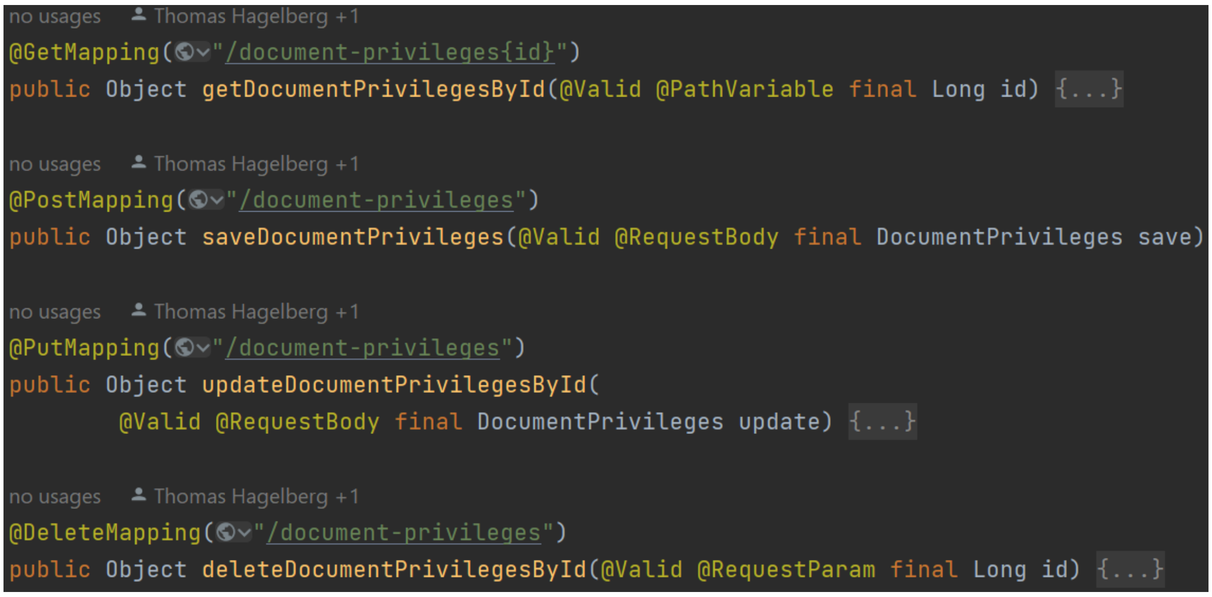
In the process of setting up the new microservice project, I came across the concept of Spring Native and decided to investigate it. I read up on the advantages of Spring Native, but was unsure of how to configure it. After much effort, I successfully configured Gradle with the native plugins and dependencies, only to be faced with error messages related to downloading necessary dependencies. These errors were due to Crosskey’s firewall blocking untrusted traffic, as it should.

After discussing the issue with the platform managers, I discovered that I needed to point all necessary dependencies towards our own internal repository. However, I was unable to find a suitable solution online in our use-case and ended up posting a detailed question on Stack Overflow. A member of the Spring team at VMware replied with a solution involving the use of a proxy, but I ultimately decided against using Spring Native due to my concerns about its enterprise readiness (Stack Overflow, 2022).

4.2.3 Contract-first approach

The software development methodology known as the contract-first approach prioritizes the creation of the openAPI contract, which includes endpoints, schemas, responses and security specific to the API, among other options. Although I initially wanted to explore the possibility of not using a contract, I discovered that doing so came with challenges.

Figure 19 shows that it is possible to create RESTful endpoints without a contract, relying solely on annotations like “@GetMapping”, “@PostMapping”, and so on. However, it is not ideal to develop an API with manually defined API annotations in the controller, as it can end up with significant functionality and descriptions missing.

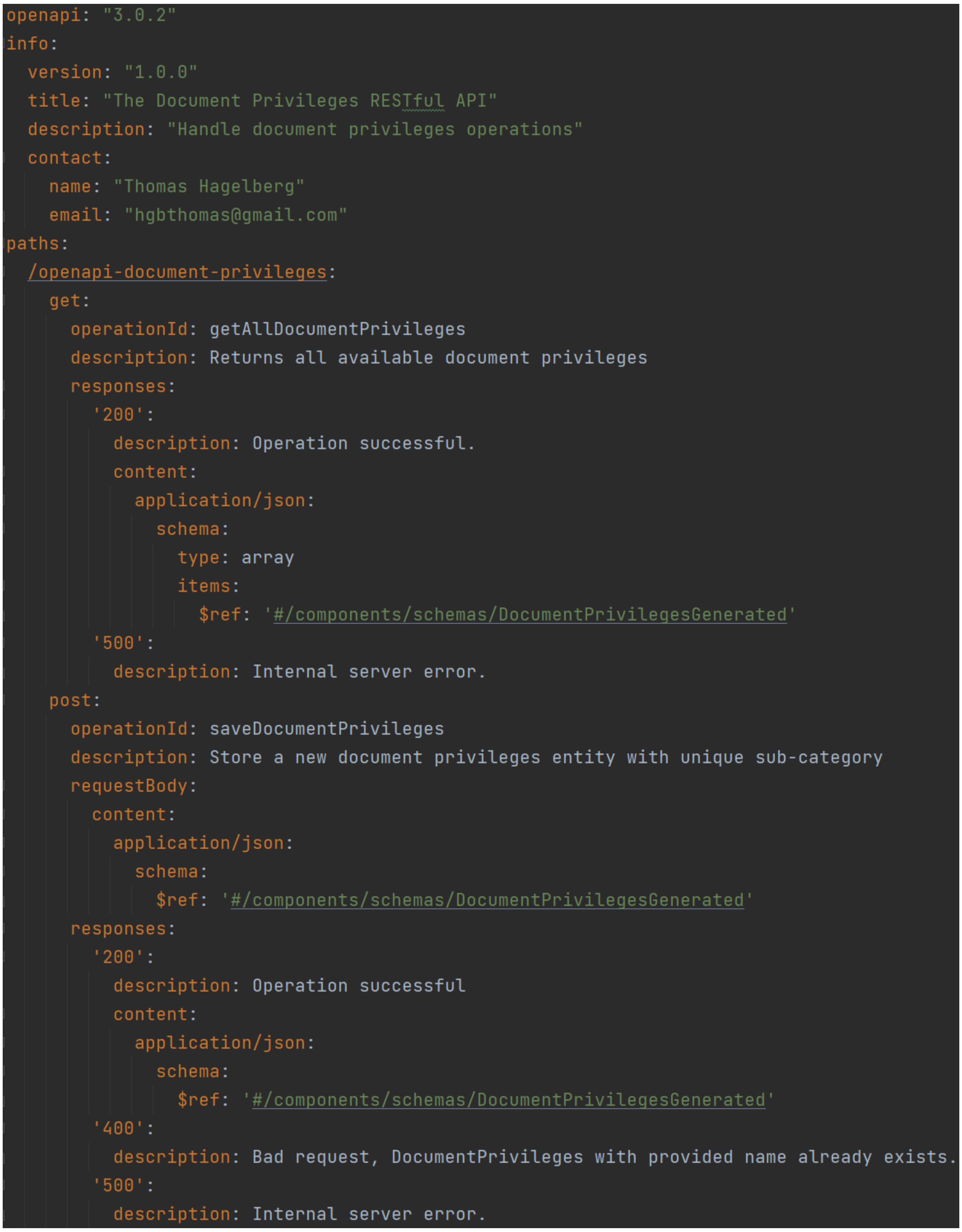


```
no usages   Thomas Hagelberg +1
@GetMapping("/document-privileges{id}")
public Object getDocumentPrivilegesById(@Valid @PathVariable final Long id) {...}

no usages   Thomas Hagelberg +1
@PostMapping("/document-privileges")
public Object saveDocumentPrivileges(@Valid @RequestBody final DocumentPrivileges save)

no usages   Thomas Hagelberg +1
@PutMapping("/document-privileges")
public Object updateDocumentPrivilegesById(
        @Valid @RequestBody final DocumentPrivileges update) {...}

no usages   Thomas Hagelberg +1
@DeleteMapping("/document-privileges")
public Object deleteDocumentPrivilegesById(@Valid @RequestParam final Long id) {...}
```

Figure 19. Example of RESTful endpoints in Spring boot using only annotations.

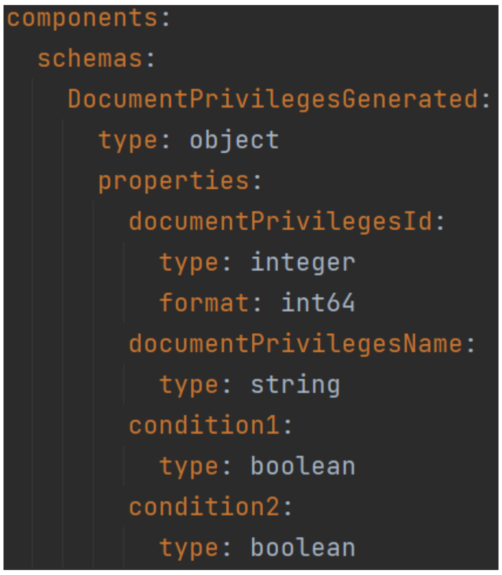
Not using the contract-first approach can thus result in a significant increase in time and effort, particularly when developing an enterprise product. The implementation could become more complicated, there could be increased lack of documentation and higher development costs. Those are just a few possible drawbacks of not using the contract-first approach.

As a result, I began creating the necessary contract after realizing the drawbacks of not using one. The required changes are illustrated in figures 20-23. The schema is generated as a separate class, while the contract is created as its own API interface. The generated interface is then implemented in the controller class, and its functions are overridden, as illustrated in figure 24.



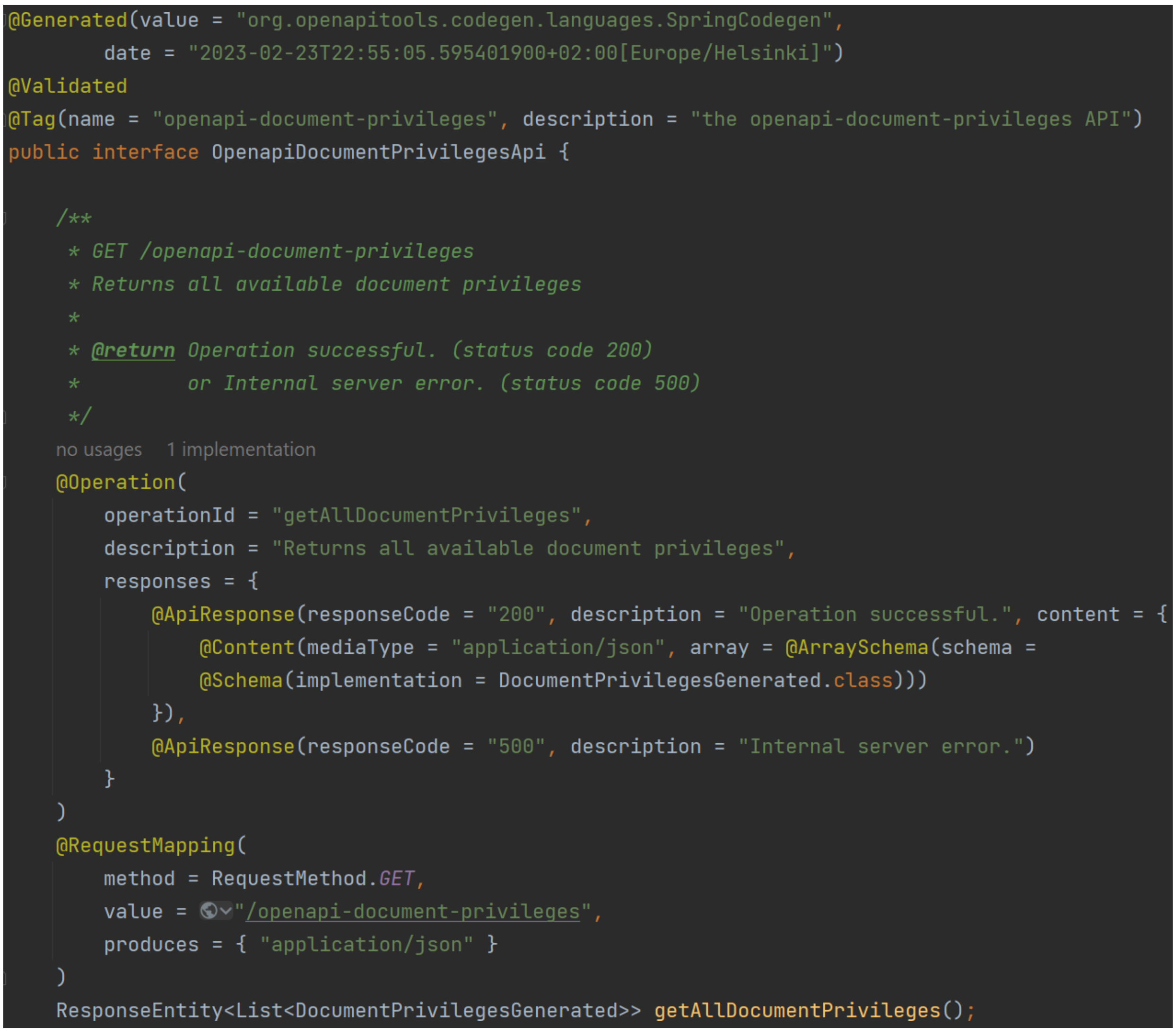
```yaml
openapi: "3.0.2"
info:
  version: "1.0.0"
  title: "The Document Privileges RESTful API"
  description: "Handle document privileges operations"
  contact:
    name: "Thomas Hagelberg"
    email: "hgbthomas@gmail.com"
paths:
  /openapi-document-privileges:
    get:
      operationId: getAllDocumentPrivileges
      description: Returns all available document privileges
      responses:
        '200':
          description: Operation successful.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/DocumentPrivilegesGenerated'
        '500':
          description: Internal server error.
    post:
      operationId: saveDocumentPrivileges
      description: Store a new document privileges entity with unique sub-category
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/DocumentPrivilegesGenerated'
      responses:
        '200':
          description: Operation successful
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/DocumentPrivilegesGenerated'
        '400':
          description: Bad request, DocumentPrivileges with provided name already exists.
        '500':
          description: Internal server error.
```

Figure 20. Example of an openAPI contract.



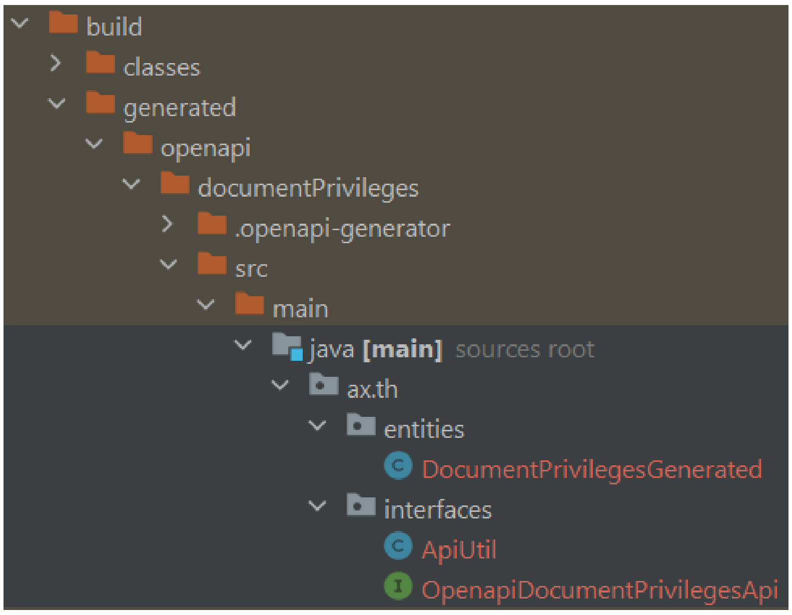
```yaml
components:
  schemas:
    DocumentPrivilegesGenerated:
      type: object
      properties:
        documentPrivilegesId:
          type: integer
          format: int64
        documentPrivilegesName:
          type: string
        condition1:
          type: boolean
        condition2:
          type: boolean
```

Figure 21. OpenAPI component schema for the contract.



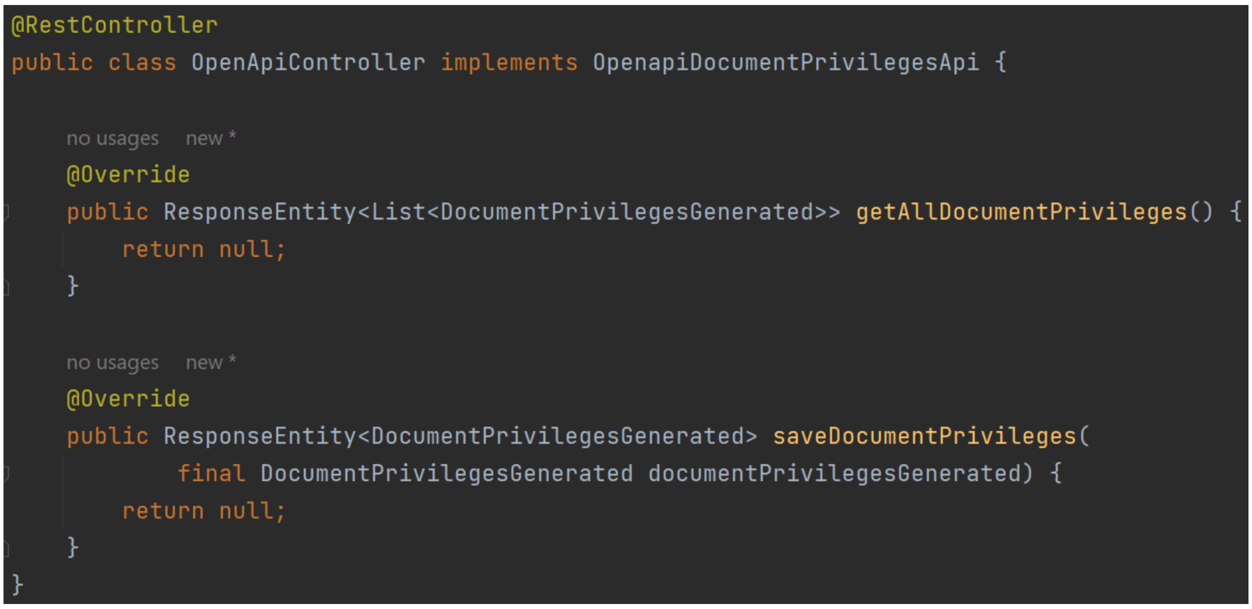
```java
@Generated(value = "org.openapitools.codegen.languages.SpringCodegen",
        date = "2023-02-23T22:55:05.595401900+02:00[Europe/Helsinki]")
@Validated
@Tag(name = "openapi-document-privileges", description = "the openapi-document-privileges API")
public interface OpenapiDocumentPrivilegesApi {

    /**
     * GET /openapi-document-privileges
     * Returns all available document privileges
     *
     * @return Operation successful. (status code 200)
     *         or Internal server error. (status code 500)
     */
    no usages   1 implementation
    @Operation(
        operationId = "getAllDocumentPrivileges",
        description = "Returns all available document privileges",
        responses = {
            @ApiResponse(responseCode = "200", description = "Operation successful.", content = {
                @Content(mediaType = "application/json", array = @ArraySchema(schema =
                @Schema(implementation = DocumentPrivilegesGenerated.class)))
            }),
            @ApiResponse(responseCode = "500", description = "Internal server error.")
        }
    )
    @RequestMapping(
        method = RequestMethod.GET,
        value = "/openapi-document-privileges",
        produces = { "application/json" }
    )
    ResponseEntity<List<DocumentPrivilegesGenerated>> getAllDocumentPrivileges();
```

Figure 22. Generated API interface from the openAPI contract.



```
build
  classes
  generated
    openapi
      documentPrivileges
        .openapi-generator
        src
          main
            java [main] sources root
              ax.th
                entities
                  DocumentPrivilegesGenerated
                interfaces
                  ApiUtil
                  OpenapiDocumentPrivilegesApi
```

Figure 23. Image displaying the generated interface and classes using "org.openapi.generator" plugin



```
@RestController
public class OpenApiController implements OpenapiDocumentPrivilegesApi {

    no usages   new *
    @Override
    public ResponseEntity<List<DocumentPrivilegesGenerated>> getAllDocumentPrivileges() {
        return null;
    }

    no usages   new *
    @Override
    public ResponseEntity<DocumentPrivilegesGenerated> saveDocumentPrivileges(
            final DocumentPrivilegesGenerated documentPrivilegesGenerated) {
        return null;
    }
}
```

Figure 24. Example of a controller class implementing the generated interface

4.2.4 JPA integration towards the database

As mentioned in chapter 3.2, the JPA framework/specification is used for providing enhanced functionality when managing entities directly towards a database. This makes the overall code quality better, and results in non-redundant coding when we can use already existing functions for managing and persisting data.

JPA has built in functionality for creating schema tables, based on the existing entities. This is possible because JPA uses the ORM (Object Relational Mapping) tool Hibernate, which handles all the conversions from the POJO (Plain Old Java Object) entities to SQL compliant queries. This table generation option is called: “jpa.hibernate.ddl-auto” in the application.yml or application.properties file. I tried it out in the development environment, but I did not find it suitable for the actual test, stage and production environment.

The reason for this was caused by the lack of overview when making updates to the POJO entities. All the SQL queries being performed can be seen and accessed, but I still believed that using a migration tool with specific script files was more clear, easy to understand and added traceability of what changes had been made.

4.3 CI/CD to AWS Fargate

CI/CD to AWS Fargate included many third-party tools for building, storing, deploying and running the application. Initially the whole process felt complex, but after a few tries all the steps became clear and very manageable.

4.3.1 Jenkins

The initial tool to set up was the CI/CD Jenkins pipeline. This pipeline was responsible for building the application jar and dockerfile, and publishing them both to the Sonatype Nexus Repository, which will be explained more in chapter 4.3.2. This process was relatively straightforward. Gradle was configured with the name of the jar, and where it should be published to. The same goes for the dockerfile but with one addition, a dockerfile specification with required arguments, that copied in the built jar and specified the path to start the application.

4.3.2 Sonatype Nexus Repository

Sonatype Nexus Repository was used for storing artifacts (jar files) and dockerfiles that would be available to be fetched from different deployment tools, which in my case was the AWS CodeBuild. CodeBuild fetches and stores a mirrored dockerfile in the AWS ECR (Elastic Container Registry). The mirrored version in ECR would then be used when deploying the Fargate service.

4.3.3 AWS CDK

AWS CDK (Cloud Development Kit) is the tool for defining AWS IaC (Infrastructure as Code) using a supported programming language. In the CDK we define everything that is needed in order to deploy the actual fargate service, which includes the new fargate stack, network configuration, environment variables, path to the docker image and more.

The implementation of all shared infrastructure was already done by another team, and my task at hand was to configure the new fargate stack, choose which network configuration I wanted to use, specify the path to my docker image in nexus, set up which environment variables should be used in the different environments, and lastly after the deployment was successful, configure OP5 monitoring for the new fargate service.

4.4 Automated Testing

Alongside the development of the application itself, the testing of the application is just as important in order to make sure that any changes, internally and externally are compatible with the latest deployed version.

The core testing of an application is based on Unit and Integration tests. In my Spring Boot application I decided to use JUnit and Mockito as testing frameworks, alongside Spring Boot's own testing framework. The core testing was split into two parts: integration testing and unit testing.

4.4.1 Integration Testing

Integration testing is a technique used to verify that all layers of the application are functioning correctly in concert with each other. This is accomplished by executing the full Spring Boot application with specific test configurations and performing a series of requests to the RESTful endpoints, while expecting a predetermined response. (Baeldung, 2023)

4.4.2 Unit testing

Unit testing is a technique used to test individual classes in isolation from other components. This is achieved by creating mock return values from the functions of other classes. This ensures that the encapsulated class independently does what it is supposed to. The advantage of having extensive unit test coverage is that it eases the troubleshooting by pinpointing the root cause of the problem in the application. (Baeldung, 2023)

4.4.3 Postman and Newman

Conducting core testing within the application is a good practice. However, it is advisable to also incorporate automated testing in a real test and/or staging environment to eliminate all forms of mocking. This ensures that, if the automated tests are successful, there is a high probability that the application will work in production environments. However, it is important to note that this cannot be taken for granted.

To achieve this, I utilized Postman and Newman for the automated testing of the application in the test and stage environments. Postman, which is an API testing tool, was used together with Newman, a command-line collection runner for Postman, to run the tests from within Jenkins pipelines.

Initially, I created a development collection in Postman to test the application locally on localhost. This enabled me to manually test the application during the development phase, thereby ensuring higher durability at the deployment phase. Thereafter, I duplicated the development collection into a test and stage collection, and updated the endpoint URLs for the new collections. Finally, I exported the collections and environment configuration to a Bitbucket repository that contained the Newman and Jenkins configuration files needed to run the pipeline.

By investing enough time to develop thorough test cases for the application, the need for manual testing throughout the application's lifecycle is significantly reduced. This often proves to be a valuable investment for all parties involved.

5. CONCLUSION

5.1 The result

Looking at the end result of this API, it can be concluded that it was a great success. It has all the functionality that was planned from the start, it is compliant with Crosskey API standards and security requirements. Turning an empty repository into a fully scaled serverless API hosted in AWS Fargate for multiple environments, I can say I am very proud of the work that has been done.

5.2 Reflections

The whole process of planning, developing, testing and deploying has more or less been a big collaboration. In other words, a project at this scale, with my beforehand knowledge would not have been possible without the help from all my colleagues, providing support in new areas that I had no prior experience in. And that is what I believe is the most valuable thing that I can carry on in the future. Broadening your network and collaborating with different teams and people is what makes a great product.

REFERENCES

Amazon. (2023). *RESTful API.* https://aws.amazon.com/what-is/restful-api/

AWS. (2023a). *Serverless Compute Engine-AWS Fargate*. https://aws.amazon.com/fargate/

AWS. (2023b). *Serverless Computing*. https://aws.amazon.com/serverless/

AWS. (n.d.). *What is the AWS CDK?* https://docs.aws.amazon.com/cdk/v2/guide/home.html

Baeldung. (2019). *Inversion of Control And Dependency Injection with Spring*. https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Baeldung. (2023). *Testing in Spring Boot*. https://www.baeldung.com/spring-boot-testing

Gradle. (n.d.). *What is Gradle?* https://docs.gradle.org/current/userguide/what_is_gradle.html

IBM. (2023). *Java Persistence API*. https://www.ibm.com/docs/en/was-liberty/nd?topic=overview-java-persistence-api-jpa

IBM. (n.d.). *What is a REST API?* https://www.ibm.com/topics/rest-apis

JavaTPoint. (n.d.). *JPA Introduction*. https://www.javatpoint.com/jpa-introduction

Jenkins. (n.d). https://www.jenkins.io/

Microsoft Azure. (2023). *What is Java Spring Boot?* https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring-boot/

Spring. (2023a). *Core Technologies*. https://docs.spring.io/spring-framework/docs/current/reference/html/core.html

Spring. (2023b). *GraalVM Native Image Support*. https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html#native-image

Spring. (2021). *AOT Engine*. https://spring.io/blog/2021/12/09/new-aot-engine-brings-spring-native-to-the-next-level

Stack Overflow. (2022). *Native question*. https://stackoverflow.com/questions/74399883/how-to-set-dependency-mapping-binding-in-gradle-bootbuildimage-spring-boot-2-7

Swagger. (n.d.-a). *About Swagger Specification | Documentation*. https://swagger.io/docs/specification/about/

Swagger. (n.d.-b). *Swagger Editor*. https://editor.swagger.io/

Swagger. (n.d.-c). *Swagger UI*. https://swagger.io/tools/swagger-ui/

Swagger. (n.d.-d). *What is Swagger?*
https://swagger.io/docs/specification/2-0/what-is-swagger/

TechTarget. (2019). *What is Spring Framework?*
https://www.techtarget.com/searchapparchitecture/definition/Spring-Framework

TechTarget (n.d.). *What is Swagger?*
https://www.techtarget.com/searchapparchitecture/definition/Swagger/

Tutorialspoint. (n.d.). *MVC-Framework.*
https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm