Indian Institute of Information Technology, Allahabad

Department of Information Technology

Course - Software Engineering

Software Design Specification

Project - Website for SERL Lab

Report By:

Adeshpal Singh (IIT2021018)

Ujjwal Narayan Pandram (IIT2021034)

Banoth Naveen (IIT2021045)

Ayan Ahmad (IIT2021090)

Rahul Chhabra (IIT2021096)

B.TECH Semester - 4, Section-A (IT)

Contents

Introduction

Purpose of the Document

This is the Software Design Specification (SDS) for the Software Engineering Research Lab Website. The SDS will break down the project into components to describe in detail what the purpose of each component is and how it will be implemented. The SDS will also serve as a tool for verification and validation of the final product.

Scope of the Development Project

We describe what features are in the scope and what are not in the scope of the software to be developed.

In Scope: 1. Information about all the faculty and research scholars in SERL 2. Accessibility Projects and publications by SERL 3. Contacts about SERL

Out of Scope: 1. Applications for internships/projects under SERL 2. Applications for research scholarship under SERL

Definitions, Acronyms, and Abbreviations

Name	Abbreviation
SERL	**Software Engineering Research Laboratory**
REST	**Representational State Transfer**
JPA	**Jakarta Persistence API**

Sections

Section 1.0: An introduction to the document by explaining the purpose of the document, providing terms and references, and a brief overview of each section within the document.

Section 2.0: An overview of the system architecture, system components, detailed description of each software component.

Section 3.0: An explanation of the reusability of existing products and relationships within the system.

Section 4.0: A listing of the design decisions and tradeoffs made during the design phase. This section will help readers and users understand the reasoning for these decisions.

System Architecture Description

This section will provide an outline of the various components and subsystems of the SERL project. Defined in a few phrases, we use a **model view controller** architecture implemented on top of Spring Boot as an **N-tier** architecture.

User Interface Issues

Our primary problem with user interfaces was the management of state on the client side. A large part of state has to be managed on the client side however vanilla JS along with HTML and CSS did not suffice for our needs. One option was to manage the state using newer state of the art technologies such as Redux or other state management solutions. However, we went instead with **server side rendering** and switched to a **model view controller** architecture.

Mustache Templates

Model View Controller Layer

To achieve the MVC architecture, we use **mustache templates**. This means that server side rendering must be done in a logic-less manner. Mustache allows substitution of variables and lists of variables. Mustache, however, does not allow rendering of the form `<th:if>` the way, say, Thymeleaf does. Why did we go with Mustache? Primarily because it deals with a lot less complexity and could handle our use cases easily. These templates provide our **view** and the model aspect needs to be dealt with by the code.

Consider the code for the model of the publications of a given author :

```
@GetMapping("/publications/{authorUsername}")
fun publicationsByAuthor(model: Model, @PathVariable authorUsername: String): String {
    val publications = publicationsRepository.findAllByAuthorUsername(authorUsername)
    val author = userRepository.findByUsername(authorUsername)
    model["title"] = "Publications by ${author?.firstName} ${author?.lastName}"
    model["publications"] = publications.map { it.render() }
    return "publications"
}
```

Here we retrieve all the publications of a given author and render it into a data object that Mustache can understand rather easily. We simply make a few assignments in the model and return the name of the view and Spring handles the rest for us.

HTML and HTTP Controller Layer

Our HTML Controllers allow us to bypass much of the complexity of manually handling `GET` requests for our webpages. Our controllers simply define how to obtain a model for a given view and database state.

However, they cannot handle `POST` requests. The reason for this is rather simple, we did not want to confuse ourselves by having multiple methods on the same endpoint.

Consider for example the handler for the publications request :

```
@PostMapping("/", consumes = [MediaType.APPLICATION_FORM_URLENCODED_VALUE])
fun postByAuthorName(
    requestDto: PublicationHttpPostRequestDto
): RedirectView {
    val authorUsername = requestDto.authorUsername
    val author = userRepository.findByUsername(authorUsername)
        ?: throw ResponseStatusException(HttpStatus.BAD_REQUEST, "Invalid author username")
    val publication = requestDto.toEntity(author)
    author.publications.add(publication)
    publicationsRepository.save(publication)
    return RedirectView("/publications")
}
```

Here the form accepts data about the publication and validates it and sends it to the `pubicationsRepository`.

We define a few similar RESTFul endpoints for our `POST` requests :

1. `/api/user` for users

2. `/api/publications` for publications
3. `/api/subscriptions` for subscriptions
4. `/api/feedback` for feedback

All of these accept `application/json` as well as `application/form-url-encoded-value`. These simply validate the data and insert it into the database.

Spring Data JPA Repositories Layer

Spring Data JPA provides us with a very useful layer of abstraction known as "repositories". A repository is defined for a specific entity (such as user) and this `UserRepository` now contains all the possible interactions we will have with our database. We do not interface directly with the database at all!

All the dirty details of connection pooling and query planning and transactions is abstracted away from us.

Consider, for example, the code for the `UserRepository` :

```kotlin
interface UserRepository : CrudRepository<User, Long> {
    fun findByUsername(login: String): User?
    fun findAllByDesignation(designation: String): Iterable<User>
}
```

Even the queries are automatically deduced by the method names!

The Data Transfer Object Layer

Data received by the user in the form of JSON or URL encoded form values are likely to not be directly mappable to the entities defined in the database schema. This means that processing any request and sending responses requires an extra layer to convert them to and from **data transfer objects**. We take an example here :

```kotlin
private fun User.render(): UserHtmlRenderDto = UserHtmlRenderDto(
    firstName = firstName,
    lastName = lastName,
    designation = designation ?: defaultDesignation(),
    description = description ?: defaultDescription(),
    address = address ?: defaultAddress(),
    contactNumber = contactNumber ?: defaultContactNumber(),
    website = website ?: defaultWebsite(),
    imageUrl = imageUrl ?: defaultImageUrl(),
    mails = mails ?: defaultMails()
)
```

To render a `User` entity is to strip it of all referential integrity constraints and cyclic dependencies while also adding handlers for null values.

It is also in this layer that we implement **admin authentication**. The idea is that the admin possesses the API key that is not available to anybody else. This API key is a part of the request data transfer object for POST requests :

```
@PostMapping(path = ["/"], consumes = [MediaType.APPLICATION_FORM_URLENCODED_VALUE])
fun postUser(userDto: UserHttpPostRequestDto): RedirectView {
    if (userDto.apiKey != apiKey) {
        throw ResponseStatusException(HttpStatus.FORBIDDEN, "Invalid API Key")
    }
    val publications = userDto.publications.map { publicationsRepository.findByIdOrNull(it)
    userRepository.save(fromHttpPostDto(userDto, publications.toMutableList()))
    return when (userDto.designation) {
        "faculty" -> RedirectView("/faculty")
        "researcher" -> RedirectView("/researchers")
        else -> throw ResponseStatusException(HttpStatus.BAD_REQUEST, "Designation should be
    }
}
```

Note the usage of `ResponseStatusException` to throw a `HttpStatus.FORBIDDEN` 403 error.

Data Definition Layer

In order to define the database schema we use the classical technique of using Jakarta annotations.

We illustrate this with a simple example. Consider the entity known as feedback :

```
@Entity
class Feedback(
    @Id @GeneratedValue var id: Long? = null,
    var name: String? = null,
    var feedback: String? = null,
)
```

- We mark the class with `@Entity` to tell Jakarta that this class represents a database table
- We specify the primary key as autogenerated using the `@Id` and the `@GeneratedValue` annotations respectively

For a more complicated, we have a more complex entity definition :

```
@Entity
class Resource(
    @Id @GeneratedValue var id: Long? = null,
    var name: String,
    var imageUrl: String,
    var purchaseUrl: String
)
```

When we have relationships involved, we need to annotate the attributes with the kind of relationship.

Consider `user`. `user` has a one-to-many relationship with `publications`.

```
@Entity
open class User(
    var username: String,
    var firstName: String,
    var lastName: String,
    var designation: String? = null,
    var description: String? = null,
    @OneToMany(mappedBy = "author", cascade = [CascadeType.ALL])
    var publications: MutableList<Publication?> = mutableListOf(),
    var address: String? = null,
    var contactNumber: String? = null,
    var website: String? = null,
    var imageUrl: String? = null,
    var mails: String? = null,
    @Id @GeneratedValue var id: Long? = null
)
```

- We use a `@OneToMany` to model the one-to-many relationship on the parent side
- We allow all forms of cascading

For publication, the child side, we have a `@ManyToOne` instead :

```
@Entity
class Publication(
    @Id @GeneratedValue var id: Long? = null,
    var title: String,
    @ManyToOne
    var author: User? = null,
    var journal: String? = null,
    var url: String? = null
)
```

Object Relational Mapping Layer

The layer we use for our ORM is the Jakarta Persistence API. We hook up JPA with an H2 database. Spring Boot automatically takes care of all the hassle of setting up Jakarta.

We simply define the details of the H2 database in the `application.properties` :

```
spring.jpa.properties.hibernate.
globally_quoted_identifiers=
true
spring.jpa.properties.hibernate.
globally_quoted_identifiers_skip_column_definitions=
true
spring.datasource.url=jdbc:h2:./app
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
```

- We use globally quoted identifiers to enforce case sensitivity
- We specify the URL of the database along with credentials
- We specify the driver and dialect class
- We set the Hibernate DDL as `update`; this means that our web server will *update* the given server rather than create a new one.

Database Layer

For the database we use H2 database. Any relational database would do and indeed, H2 is not particularly scalable. We would have to switch to MySQL or Postgres for anything even slightly more complicated than our current codebase.

Nonetheless, we chose H2 precisely because it is

Spring Boot (Web Server) Layer

The lowest layer in our architecture is the Spring Boot framework. Spring Boot embeds and abstracts away the nitty gritty details of setting up a

Reuse and Relationships

The technologies we chose for our project, namely MVC architecture, Spring Data JPA, Bootstrap, and H2 database, played a significant role in enabling code reuse and promoting a modular development approach.

The MVC architecture inherently promotes code reuse by separating the application into three distinct layers: the Model, View, and Controller. This clear separation allows developers to reuse and share code across different components of the application. For example, business logic defined in the Controller layer can be reused across multiple views, ensuring consistent behavior and reducing code duplication. Similarly, the Model layer encapsulates data and business rules, making it easier to reuse data-related code and validations.

Spring Data JPA further enhances code reuse by providing a set of standardized

APIs and abstractions for working with databases. It offers a repository pattern that eliminates the need for developers to write repetitive CRUD (Create, Read, Update, Delete) operations for each entity. Instead, developers can define generic repository interfaces that automatically generate the necessary SQL queries based on method signatures. This approach greatly reduces boilerplate code and encourages code reuse across different entities and data access operations.

Bootstrap, as a front-end framework, offers a library of reusable UI components and styles. Developers can leverage these components to build consistent and visually appealing user interfaces without reinventing the wheel. By using Bootstrap's predefined CSS classes and responsive design features, developers can create reusable layouts and styles that can be easily applied to different parts of the application. This saves time and effort in designing and styling individual components, allowing for efficient code reuse throughout the project.

Additionally, the H2 database's embeddable nature facilitates code reuse by simplifying the setup and configuration process. Since H2 can be embedded within the application, developers can distribute the entire application as a single executable, ensuring consistent and predictable behavior across different environments. This eliminates the need for separate database installations and configurations, making it easier to reuse and share the application codebase without complex setup instructions or dependencies.

Overall, the combination of MVC architecture, Spring Data JPA, Bootstrap, and H2 database enables code reuse by providing standardized patterns, abstractions, and pre-built components. These technologies simplify the development process, reduce code duplication, and facilitate modular design, resulting in a more maintainable and reusable codebase.

Design Decisions and Trade-offs

In designing our project, we prioritized ease of use and developer familiarity to ensure a seamless and efficient development process. One of the key choices we made was to adopt the Model-View-Controller (MVC) architecture. This design pattern promotes a clear separation of concerns, allowing developers to focus on specific components of the application without affecting others. By using MVC, developers can easily understand and maintain the codebase, leading to improved productivity and faster development cycles.

To simplify data access and management, we incorporated Spring Data JPA into our project. This powerful framework provides a high-level abstraction over the underlying database, eliminating the need for developers to write complex SQL queries manually. Spring Data JPA leverages the familiar Java Persistence API (JPA) standard, enabling developers to work with entities, relationships, and queries using object-oriented paradigms. This familiar approach reduces the learning curve and allows developers to quickly and efficiently interact with the database.

For the user interface, we leveraged the popular Bootstrap framework. Bootstrap offers a wide range of pre-designed components and responsive layouts, enabling developers to create visually appealing and user-friendly interfaces with minimal effort. Its extensive documentation and vast community support ensure that developers can easily find solutions and guidance when facing design challenges. By utilizing Bootstrap, we aimed to provide an intuitive and consistent user experience across different devices and screen sizes.

In terms of the database, we chose the H2 database for its simplicity and ease of setup. H2 is an in-memory, lightweight database that can be embedded within the application. This eliminates the need for external database installations during development, making it convenient for developers to get started quickly. H2 also supports easy configuration and data migration, allowing for smooth transitions between development, testing, and production environments.

By considering ease of use and developer familiarity as the design basis of our project and incorporating MVC architecture, Spring Data JPA, Bootstrap, and H2 database, we aimed to create a development environment that promotes efficient coding practices, simplifies data management, and provides an intuitive user interface, ultimately resulting in a user-friendly and robust application.