Session-based Travelling Agency

Inocan Sergiu-Robert

January 8, 2024

Abstract

This document presents "Session-based Travelling Agency," a comprehensive web application developed by Inocan Sergiu-Robert for organizing travel itineraries. Utilizing Java Spring Framework, JDBC, and Session-based programming, the platform offers features such as real-time booking, detailed journey information, and user feedback. It blends advanced backend technologies with a user-friendly HTML+CSS+JS frontend. The document includes UML diagrams to elucidate the system's architecture and user interaction. Additionally, it provides a detailed look at implementation strategies, code examples, and a mini-project showcasing the Java Spring framework's application.

1 Introduction

The Session-based Travel Agency application is a multifaceted web-based platform designed to offer users a top-tier experience in planning, booking, and managing their travel itineraries. This application capitalizes on cutting-edge technologies to provide an extensive range of features, including real-time flight and hotel booking, itinerary customization, travel recommendations, user preferences tracking, and dynamic pricing.

1.1 Technology Stack

- Java Spring Framework: The core of this application is built upon the Java Spring framework, a robust and scalable platform that facilitates modular development, dependency injection, and efficient management of application components. Spring provides the foundation for building a highly maintainable and extensible codebase.
- JDBC (Java Database Connectivity): JDBC is employed to establish a secure and efficient connection with the backend database system. Through JDBC, the application interacts with the relational database management system (RDBMS) to store and retrieve critical journey-related data, including journey details, booking records, pricing information, and user feedback.
- HttpSession: The HttpSession mechanism is utilized to manage user sessions effectively. By leveraging HttpSession, the application ensures seamless user interactions by maintaining session state, user authentication, and authorization, enabling the implementation of personalized experiences tailored to individual user preferences.

- HTML+CSS Frontend: The user interface (UI) is developed using a combination of HTML and CSS, resulting in an aesthetically pleasing and highly responsive web design. This frontend technology stack ensures a visually engaging and intuitive user experience, with rich interactions and user-friendly navigation.

1.2 Key Features

- Real-time Booking for Diverse Journeys: Users can access real-time information about a wide range of journeys, from scenic road trips to adventurous treks and cultural explorations. The application allows for instant booking and reservations for these diverse travel experiences.
- Viewing Journey Details: Users have access to comprehensive journey details, including itineraries, destinations, highlights, and available dates. This feature enables users to explore and understand the journey they are interested in before making a reservation.
- User Feedback and Reviews: The application provides a platform for users to check feedback and reviews from fellow travelers who have embarked on similar journeys. This user-generated content assists travelers in making well-informed decisions about their upcoming journeys.

2 Design

2.1 UML Diagrams:

- Use Case Diagram:
  - In my use case diagram for a Traveling Agency Application, I've outlined several interactions that a "Client" - the actor in this scenario - can have with the system.
  - As the user, I start by logging into the application. If I enter invalid credentials, the system will exclude me from proceeding. Once logged in, I can perform several actions. I can see journey details to explore various travel options provided by the agency. I also have the option to see complete feedback, which means I can view reviews and experiences from other clients regarding their journeys.
  - If I decide on a journey that suits my preferences, I can proceed to choose that journey. An inclusive action within choosing a journey is to fill out a date form, which is necessary to finalize the booking. There's an exclusion condition here where the arrival date must be greater than the departure date, which is a logical requirement to ensure that the booking dates make sense.
  - Finally, once I've completed my interactions with the application, or if I decide not to book at the moment, I can log out of the system.
  - This use case diagram serves as a high-level representation of the functionalities available to me as a client using the travel agency application.

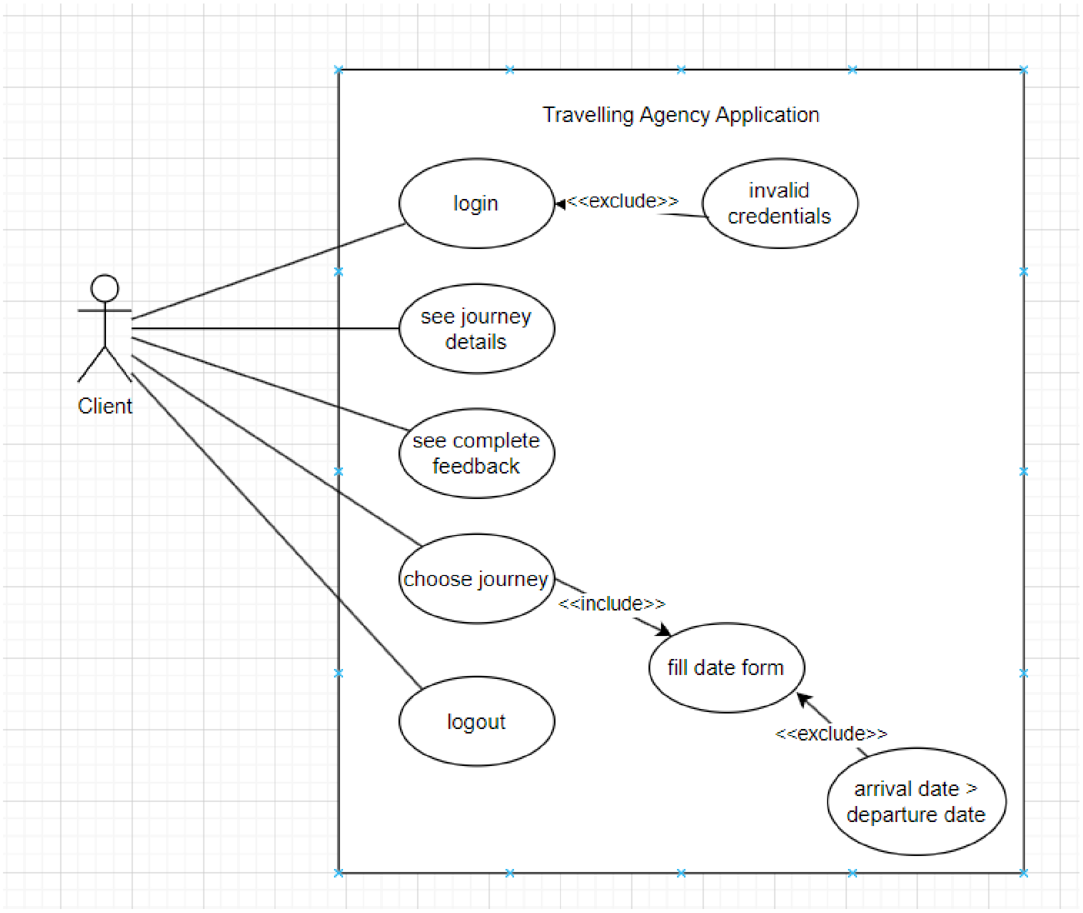


Fig 1: Use case diagram

- Class Diagram:

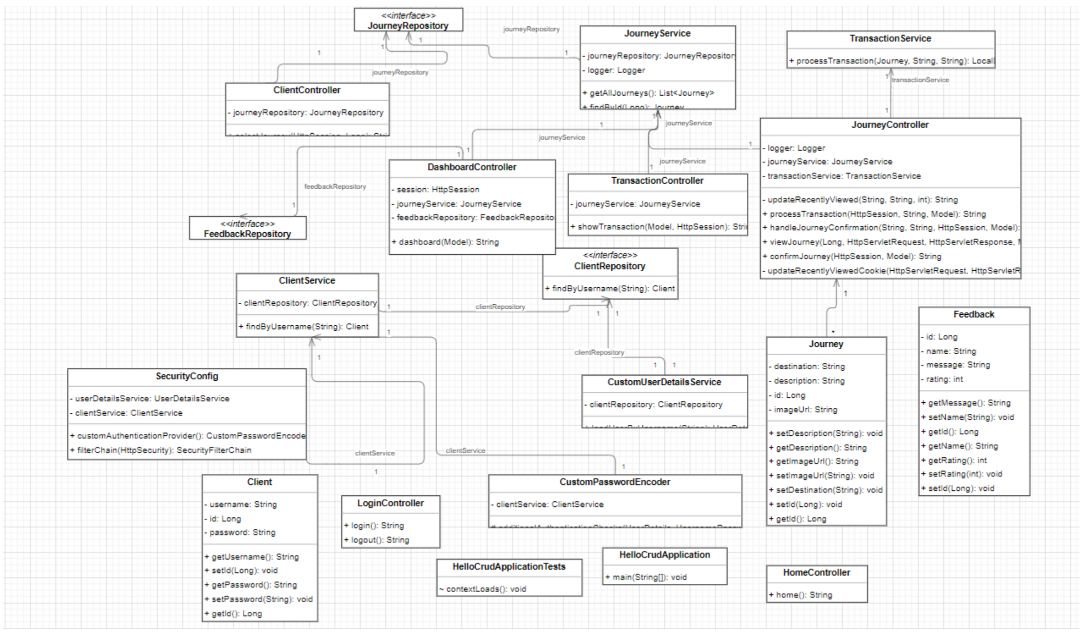
  In the class diagram for my travel agency application, the classes are organized into logical groups that represent different layers and components of the application, which primarily uses a MVC architecture, having also a Service layer:

  – Controllers:
    * ClientController: Manages client interactions with journey data.
    * DashboardController: Handles the presentation of the dashboard view to the client.
    * LoginController: Manages client authentication processes.
    * TransactionController: Deals with the financial transactions of the client.
    * JourneyController: Directly manages journey-related requests.
    * HomeController: Manages the main landing page or home view of the application.

  – Services:

    - JourneyService: Contains business logic pertaining to journey operations. TransactionService: Processes the business logic for transactions. ClientService: Encapsulates business logic for client-related operations. CustomUserDetailsService: Custom service for user details, for security purposes.
- Repositories:
    - JourneyRepository: Interface for data access operations on journey data. FeedbackRepository: Interface for data access operations on feedback data. ClientRepository: Interface for data access operations related to client information.
- Security Configurations:
    - SecurityConfig: Configuration class for security, detailing the security filters and rules. CustomPasswordEncoder: Defines the password encoding strategy.
- Models:
    - Client: Represents the client entity with attributes and behaviors. Journey: Represents a journey with details such as destination and ratings. Feedback: Represents feedback entries with details like message and rating.
- Main:
    - HelloCrudApplication: The entry point of the application, containing the main method.

Each of these groups plays a specific role within the application's architecture, with controllers handling HTTP requests, services encapsulating business logic, repositories providing data access, and models representing the data structure. Security configurations ensure secure operations, and the testing and entry point classes are for running and verifying the application.



<<interface>> JourneyRepository
journeyRepository
JourneyService
- journeyRepository: JourneyRepository
- logger: Logger
+ getAllJourneys(): List<Journey>
+ findById(Long): Journey
TransactionService
+ processTransaction(Journey, String, String): Local
transactionService
ClientController
- journeyRepository: JourneyRepository
journeyService
JourneyController
- logger: Logger
- journeyService: JourneyService
- transactionService: TransactionService
- updateRecentlyViewed(String, String, int): String
+ processTransaction(HttpSession, String, Model): String
+ handleJourneyConfirmation(String, String, HttpSession, Model):
+ viewJourney(Long, HttpServletRequest, HttpServletResponse,
+ confirmJourney(HttpSession, Model): String
- updateRecentlyViewedCookie(HttpServletRequest, HttpServletR
DashboardController
- session: HttpSession
- journeyService: JourneyService
- feedbackRepository: FeedbackReposito
+ dashboard(Model): String
feedbackRepository
<<interface>> FeedbackRepository
TransactionController
- journeyService: JourneyService
+ showTransaction(Model, HttpSession): Stri
<<interface>> ClientRepository
+ findByUsername(String): Client
ClientService
- clientRepository: ClientRepository
+ findByUsername(String): Client
clientRepository
Journey
- destination: String
- description: String
- id: Long
- imageUrl: String
+ setDescription(String): void
+ getDescription(): String
+ getImageUrl(): String
+ setImageUrl(String): void
+ setDestination(String): void
+ setId(Long): void
+ getId(): Long
Feedback
- id: Long
- name: String
- message: String
- rating: int
+ getMessage(): String
+ setName(String): void
+ getId(): Long
+ getName(): String
+ getRating(): int
+ setRating(int): void
+ setId(Long): void
SecurityConfig
- userDetailsService: UserDetailsService
- clientService: ClientService
+ customAuthenticationProvider(): CustomPasswordEncode
+ filterChain(HttpSecurity): SecurityFilterChain
CustomUserDetailsService
- clientRepository: ClientRepository
clientService
Client
- username: String
- id: Long
- password: String
+ getUsername(): String
+ setId(Long): void
+ getPassword(): String
+ setPassword(String): void
+ getId(): Long
LoginController
+ login(): String
+ logout(): String
CustomPasswordEncoder
- clientService: ClientService
HelloCrudApplicationTests
~ contextLoads(): void
HelloCrudApplication
+ main(String[]): void
HomeController
+ home(): String

Fig 2: Class diagram

- Package diagram:

  In this package diagram, which outlines the structure the software application, the packages are organized as follows:

  - main: This is the entry point of the application. It has dependencies on all other packages, suggesting that it orchestrates the overall flow of the application by leveraging the functionalities provided by these packages.
  - controller: This package contains the classes responsible for handling incoming HTTP requests, manipulating data through the services, and returning the response to the client. It depends on the service and model packages, indicating that it uses services to process business logic and models to represent the data it handles.
  - model: This package includes the domain models or entities of the application. These are the core classes that represent the application's data structure. The model has a unidirectional relationship with the main package, indicating that it's utilized by the application's entry point.
  - service: The service package encapsulates the business logic of the application. It relies on the repo (repository) package to persist and retrieve data.
  - repo: Short for repository, this package is responsible for data access and storage. It contains interfaces and classes that interact with the database or other persistence mechanisms. It has a direct association with the model package because it would need to understand the structure of the data it's persisting or retrieving.

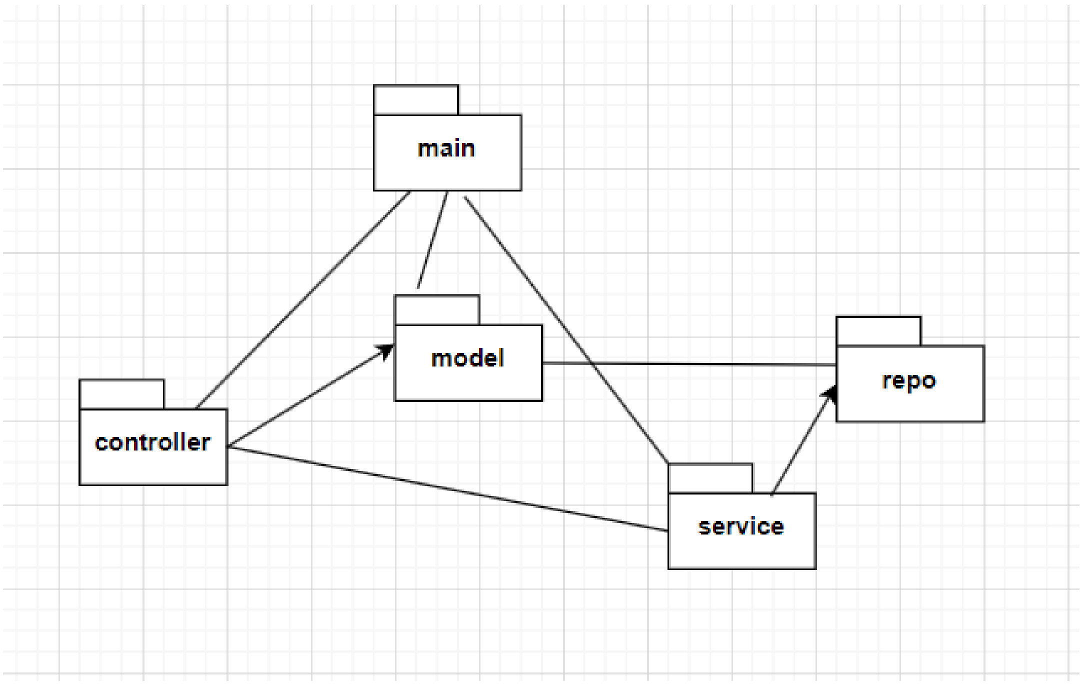


Fig 3: Package diagram

- State transition diagram:

  In my state transition diagram for the application, the process flow is this one:

  - Starts at the HomePage state. From there, I can transition to the Login state, where two possible outcomes are considered: if I input invalid credentials, I remain in the Login state, but if I provide valid credentials, a session is created, and I transition to the Dashboard state.
  - While in the Dashboard, I have several options. I can choose to log out, which transitions me to the Logout state and presumably ends my session, bringing me back to the HomePage.
  - Alternatively, I can choose a journey, which transitions me to the SetDetails state, where I can set the specifics of my journey. If I input invalid dates (e.g., the arrival date is before the departure date), I stay in the SetDetails state. If the dates are valid, I can confirm my selection, which takes me to the Checkout state to complete my booking.

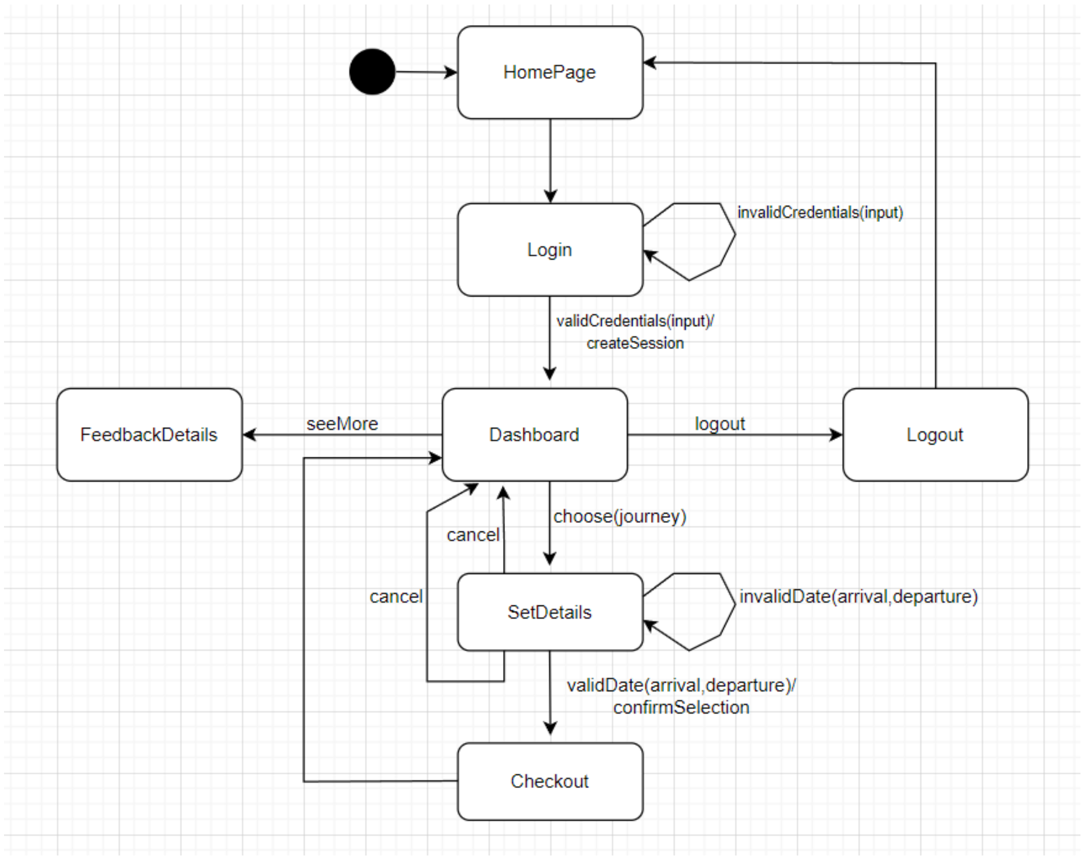


Fig 4: State transition diagram

- Deployment diagram:

In my deployment diagram for the travel agency application, there are three primary devices that interact with each other: the Web Server, the Database Server, and the Web Browser.

- The Web Server hosts the Java Spring application, which is the backend of my application. It handles HTTP requests on port 8080 and manages user sessions. It approves session requests from the client and communicates with the Database Server.
- The Database Server contains the Agency DB, an artifact that I've chosen to implement with PostgreSQL. It has two components, Client and Journey, which represent the data structure for clients and their journey details within the database.
- The Web Browser represents the client-side of the application, where users interact with the system. It displays the Client Website, an HTML5 artifact, which is the user interface. Within this interface, there are components such as confirm-journey, dashboard, and transaction-success, which correspond to different pages.

Overall, the deployment diagram illustrates how my application's components are distributed across different devices and how they interact. The Web Server processes logic and communicates with the Database Server for data persistence, while the Web Browser serves as the front-end that clients use to interact with my application.

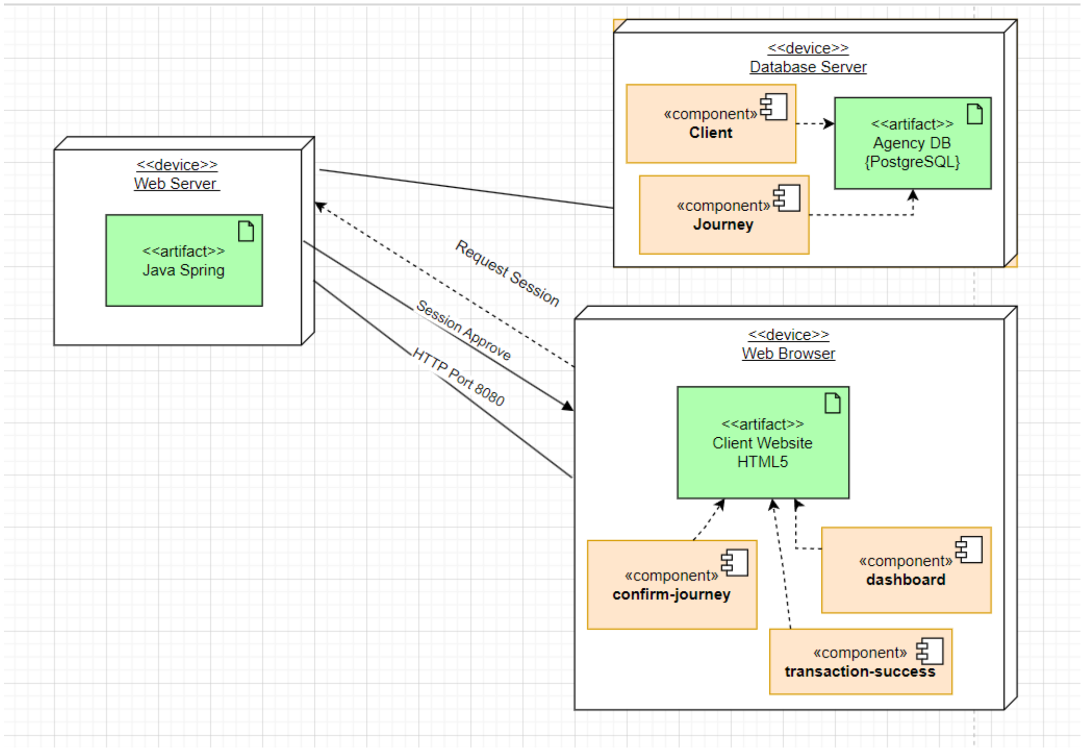


Fig 5: Deployment diagram

2.2 Session sequence diagram:

This sequence diagram describes the interactions between different components within my project.

- Firstly, when the client logs in, a session request is sent to the Agency, the Agency then creates a new session request to the service, which creates the session and stores the client's data within the attribute, then it sends the attribute to the Agency which will respond to the Client, approving its logging in and assigning a session ID for him.
- After that, if the Client chooses a journey, a new session is created and the 2nd session id is set for the client, he then has to choose a proper date, composed of 2 dates: the arrival date and the departure date, if those 2 dates pass the requirements, the service will confirm the client's booking.

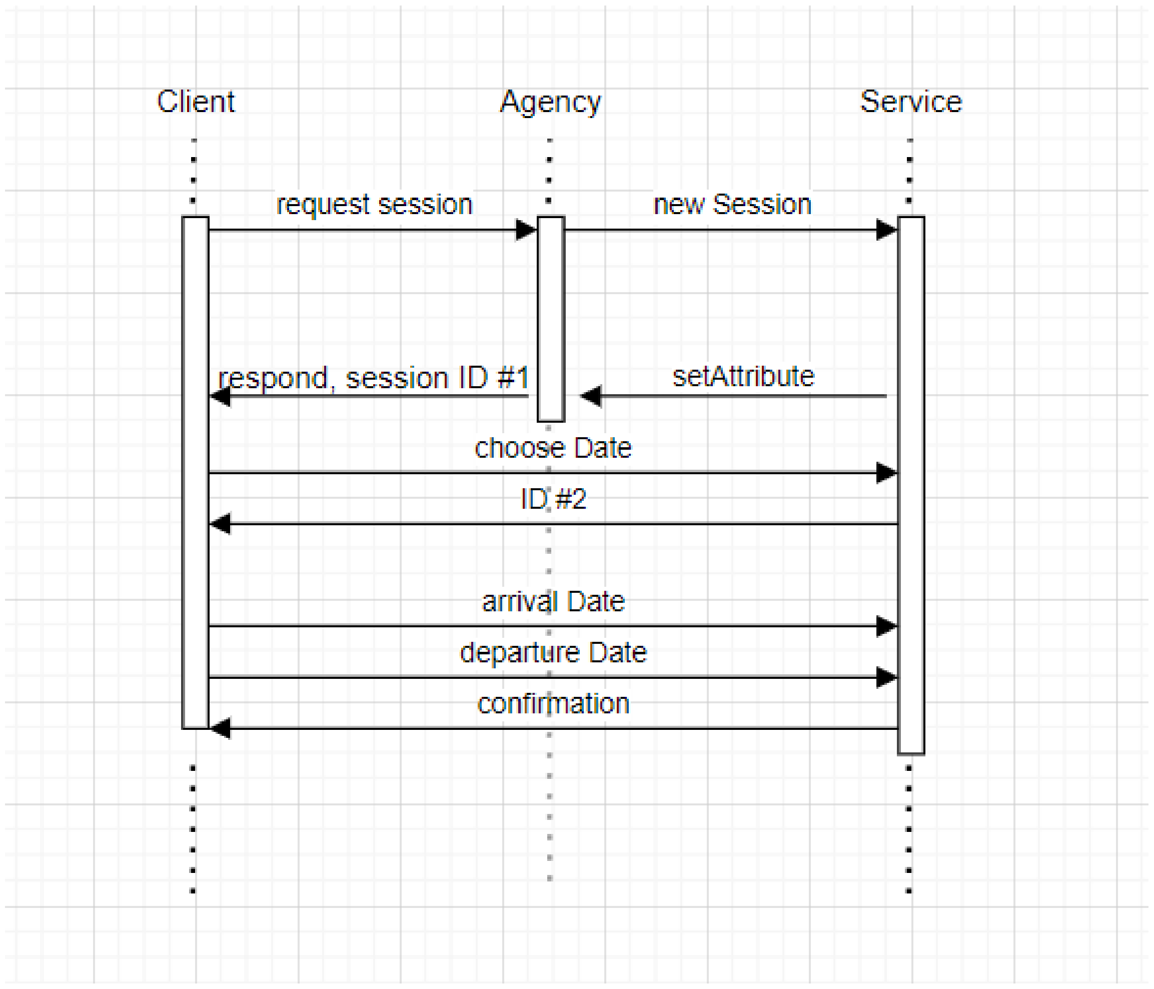


Fig 6: Sequence diagram

2.3 Session Types Used:

In my Java Spring project, I'm using **Cookies** as a crucial part of session management. **Cookies** are small pieces of data that are stored on the client-side, typically within the user's web browser. They serve as a means of maintaining state information between the client and the server.

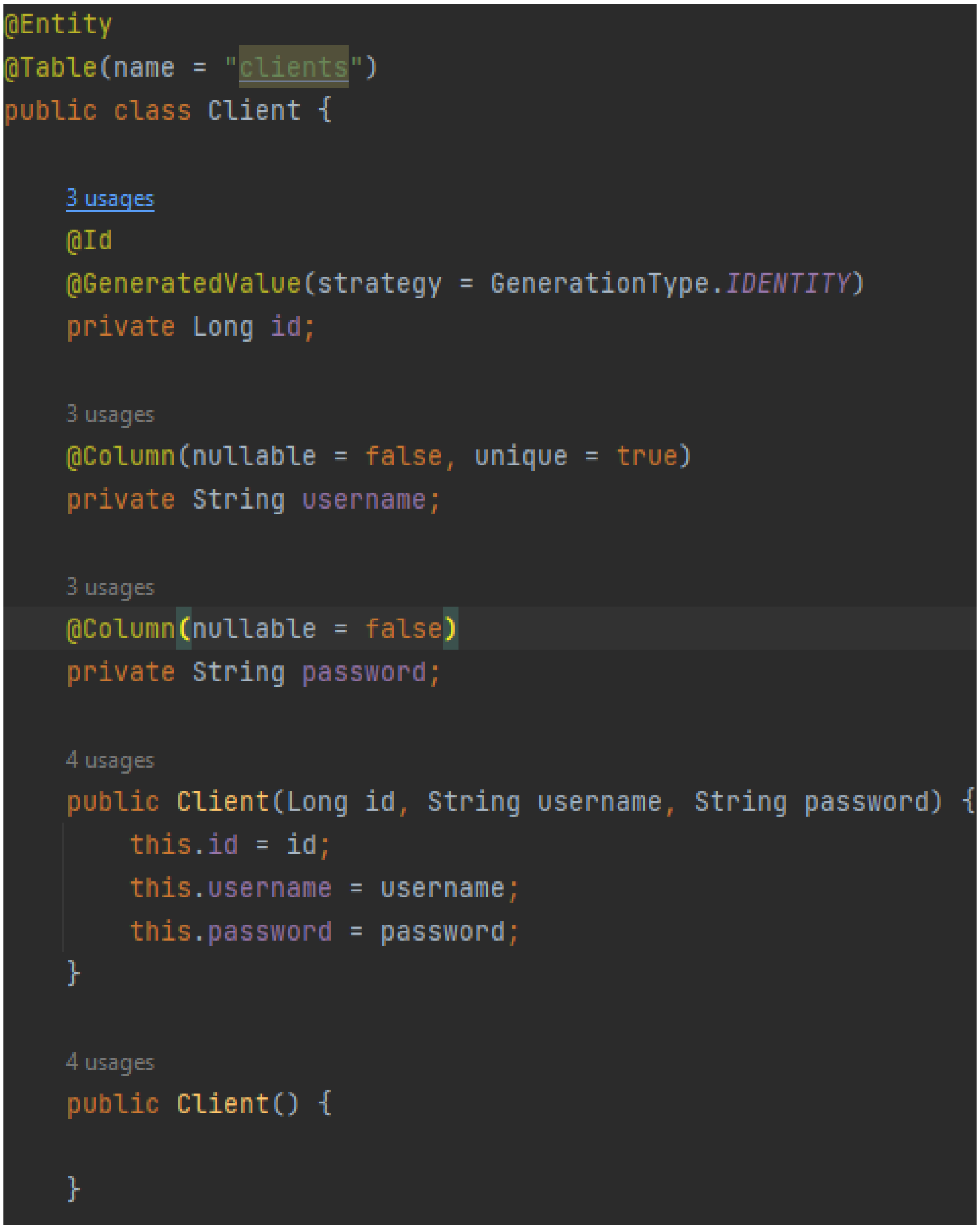
Here's how cookies work in my project:

- Session Identifier Storage: When a user first visits my website and establishes a session, a unique session identifier is generated on the server. This session identifier is a sort of token that represents the user's session. Instead of storing all session data on the server, which would be inefficient, we just store this identifier in a cookie on the user's browser.
- Associating Requests with Sessions: Every time the user makes a new request to my website, the browser automatically includes this session identifier cookie in the HTTP request header. This is crucial because it allows the server to identify which session the request belongs to.
- Server-Side Session Management: On the server-side, Spring takes care of managing these session identifiers. It knows how to map a received session identifier to the corresponding session data, effectively associating each request with the correct session. This way, I can maintain user-specific data and state across multiple requests without the user having to constantly log in or provide their credentials.

3 Implementation

In this section, I will present only the most relevant parts of code, or the most important ones, as the whole code itself is too much.

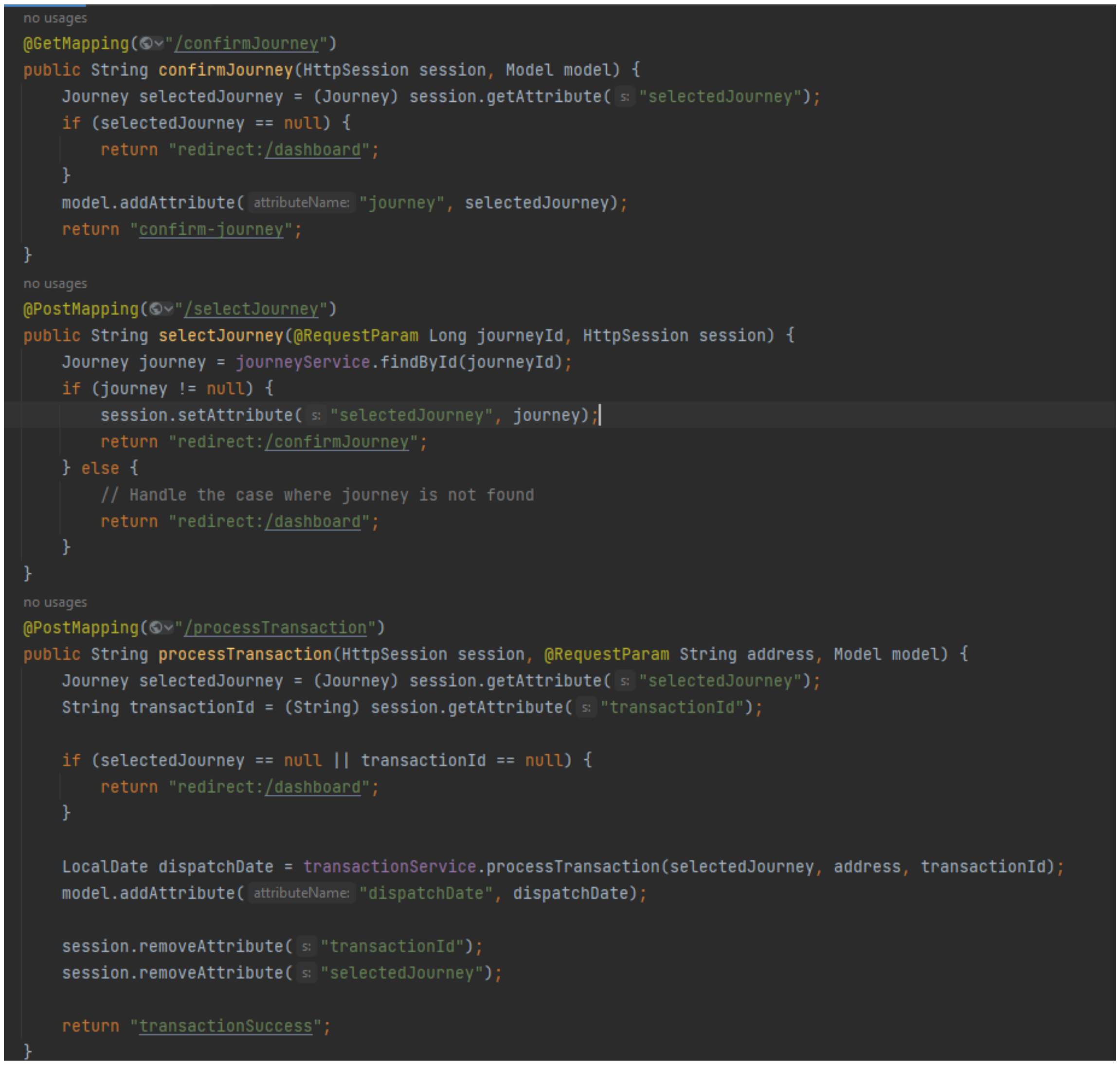
Firstly, the models, which are the objects representing the actual data from the database, whose values are injected using Spring's tag @Entity. These models are used for the services and controllers so that they can work with the actual data from the database.



```
@Entity
@Table(name = "clients")
public class Client {

    3 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    3 usages
    @Column(nullable = false, unique = true)
    private String username;

    3 usages
    @Column(nullable = false)
    private String password;

    4 usages
    public Client(Long id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    4 usages
    public Client() {

    }
```

Fig 7: The Client model

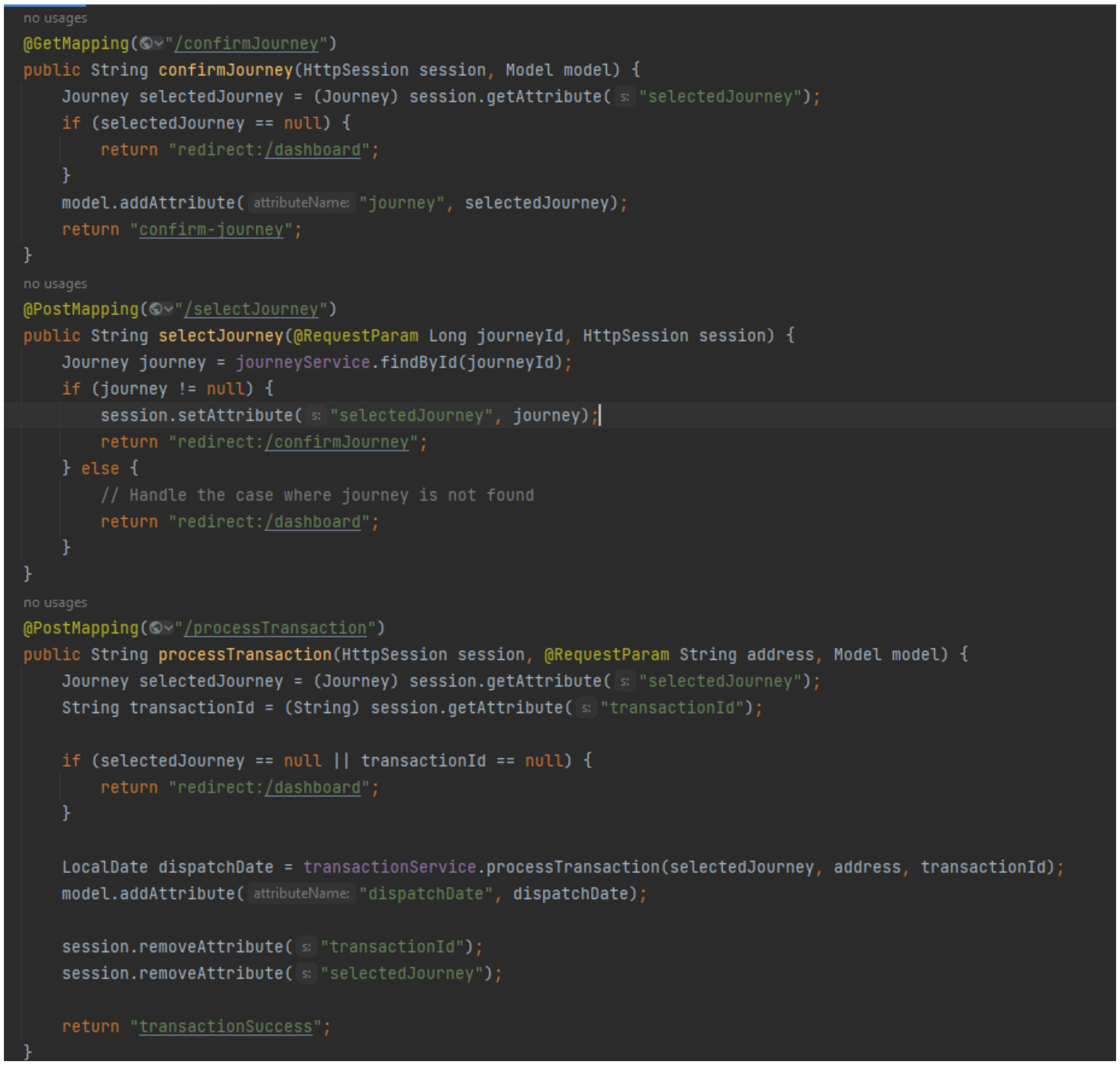
Secondly, the controllers of this project, which are ensuring the handling of the session requests and responses, together with the functionalities needed upon such requests. The whole Controller package is described in the design section, here I will just present 2 relevant methods from the most important controller, namely the JourneyController, which is responsible for everything that needs to be done related to the journeys, things like: confirming the journey, selecting one, processing the data of a journey, or showing the selected journey at the end. The methods that will be shown in the picture are 3 methods, 1 getMap type for confirming the journey and storing its data, and other 2 postMap type, the first one selects a certain journey and the second one gets the journey data and gets the transaction done.



```
no usages
@GetMapping("/confirmJourney")
public String confirmJourney(HttpSession session, Model model) {
    Journey selectedJourney = (Journey) session.getAttribute( s: "selectedJourney");
    if (selectedJourney == null) {
        return "redirect:/dashboard";
    }
    model.addAttribute( attributeName: "journey", selectedJourney);
    return "confirm-journey";
}
no usages
@PostMapping("/selectJourney")
public String selectJourney(@RequestParam Long journeyId, HttpSession session) {
    Journey journey = journeyService.findById(journeyId);
    if (journey != null) {
        session.setAttribute( s: "selectedJourney", journey);
        return "redirect:/confirmJourney";
    } else {
        // Handle the case where journey is not found
        return "redirect:/dashboard";
    }
}
no usages
@PostMapping("/processTransaction")
public String processTransaction(HttpSession session, @RequestParam String address, Model model) {
    Journey selectedJourney = (Journey) session.getAttribute( s: "selectedJourney");
    String transactionId = (String) session.getAttribute( s: "transactionId");

    if (selectedJourney == null || transactionId == null) {
        return "redirect:/dashboard";
    }

    LocalDate dispatchDate = transactionService.processTransaction(selectedJourney, address, transactionId);
    model.addAttribute( attributeName: "dispatchDate", dispatchDate);

    session.removeAttribute( s: "transactionId");
    session.removeAttribute( s: "selectedJourney");

    return "transactionSuccess";
}
```

Fig 8: Some journey controller methods

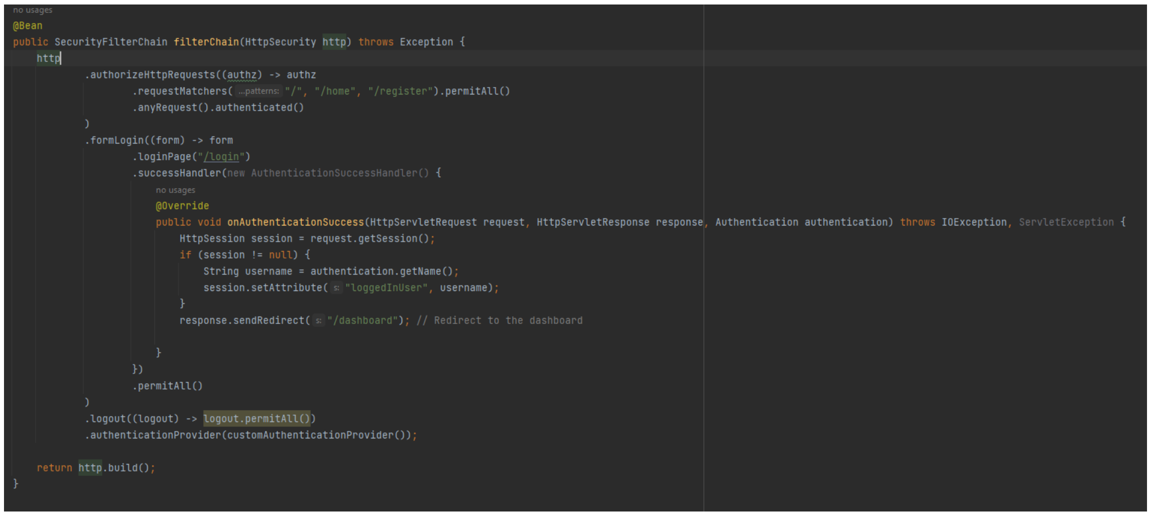
Another important part of the project is the Service package, which holds the classes responsible for calling methods that use SQL queries, in order to manipulate the data stored in the database. One such service is JourneyService, which has a method like getAllJourneys() which gets all the journeys from the journeys table inside the database.



```java
no usages
@GetMapping("/confirmJourney")
public String confirmJourney(HttpSession session, Model model) {
    Journey selectedJourney = (Journey) session.getAttribute( s: "selectedJourney");
    if (selectedJourney == null) {
        return "redirect:/dashboard";
    }
    model.addAttribute( attributeName: "journey", selectedJourney);
    return "confirm-journey";
}
no usages
@PostMapping("/selectJourney")
public String selectJourney(@RequestParam Long journeyId, HttpSession session) {
    Journey journey = journeyService.findById(journeyId);
    if (journey != null) {
        session.setAttribute( s: "selectedJourney", journey);
        return "redirect:/confirmJourney";
    } else {
        // Handle the case where journey is not found
        return "redirect:/dashboard";
    }
}
no usages
@PostMapping("/processTransaction")
public String processTransaction(HttpSession session, @RequestParam String address, Model model) {
    Journey selectedJourney = (Journey) session.getAttribute( s: "selectedJourney");
    String transactionId = (String) session.getAttribute( s: "transactionId");

    if (selectedJourney == null || transactionId == null) {
        return "redirect:/dashboard";
    }

    LocalDate dispatchDate = transactionService.processTransaction(selectedJourney, address, transactionId);
    model.addAttribute( attributeName: "dispatchDate", dispatchDate);

    session.removeAttribute( s: "transactionId");
    session.removeAttribute( s: "selectedJourney");

    return "transactionSuccess";
}
```

Fig 9: Journey Service

A very important class, is The SecurityConfig class which is a configuration class for setting up Spring Security in the application. It's responsible for customizing authentication and authorization processes. This class defines how the application handles user logins, specifying which URLs are publicly accessible and which require authentication. It also configures a custom form login process, determining the behavior upon successful authentication, such as redirecting to a dashboard and storing the username in the session. For example, the method that decides which pages are accesible publicly or not, while also handling the scenario where the user logged in succesfully is done by the filterChain() method inside of this class.



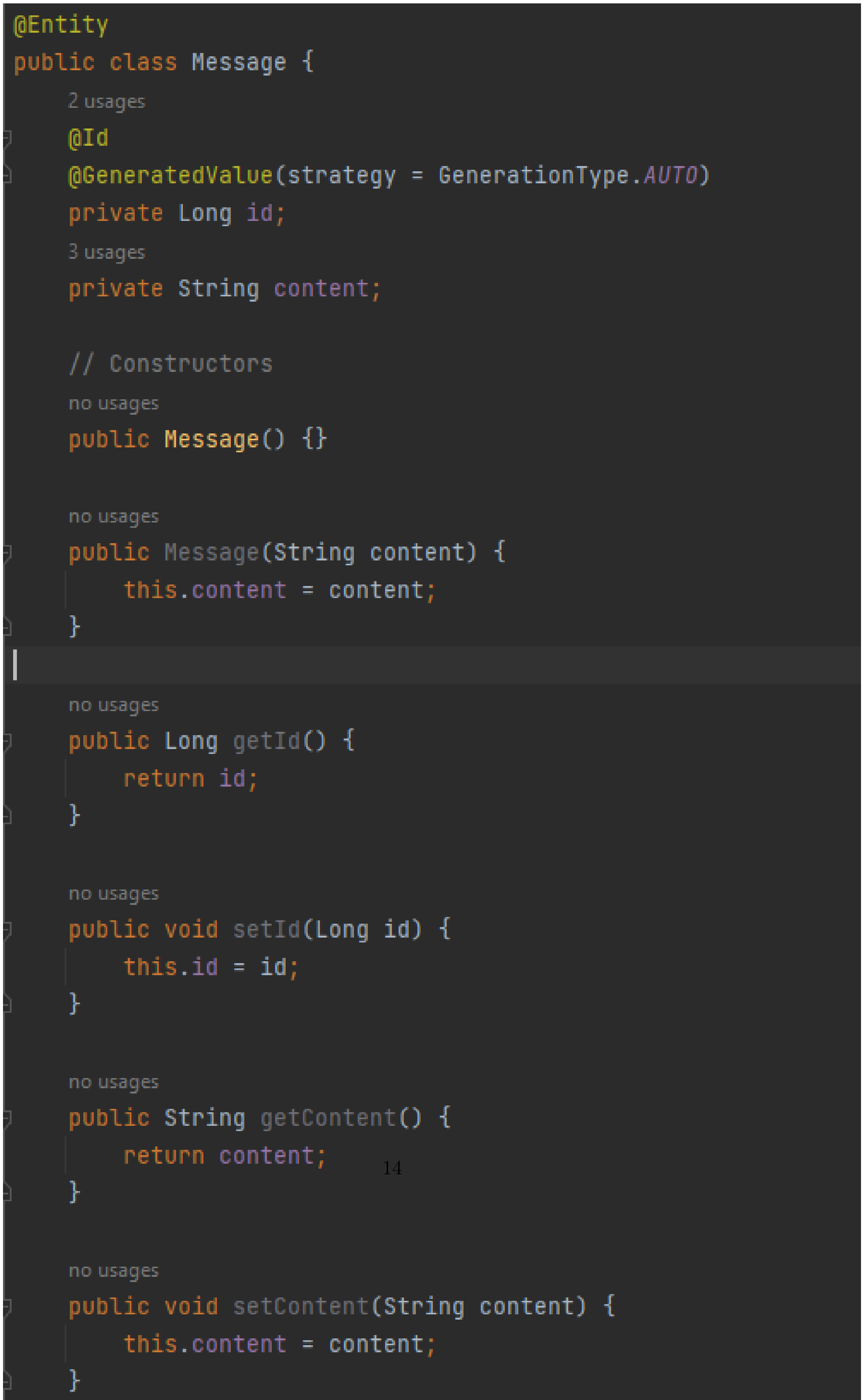
```
no usages
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
            .authorizeHttpRequests((authz) -> authz
                    .requestMatchers( ...patterns: "/", "/home", "/register").permitAll()
                    .anyRequest().authenticated()
            )
            .formLogin((form) -> form
                    .loginPage("/login")
                    .successHandler(new AuthenticationSuccessHandler() {
                        no usages
                        @Override
                        public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) throws IOException, ServletException {
                            HttpSession session = request.getSession();
                            if (session != null) {
                                String username = authentication.getName();
                                session.setAttribute( s: "loggedInUser", username);
                            }
                            response.sendRedirect( s: "/dashboard"); // Redirect to the dashboard

                        }
                    })
                    .permitAll()
            )
            .logout((logout) -> logout.permitAll())
            .authenticationProvider(customAuthenticationProvider());

    return http.build();
}
```

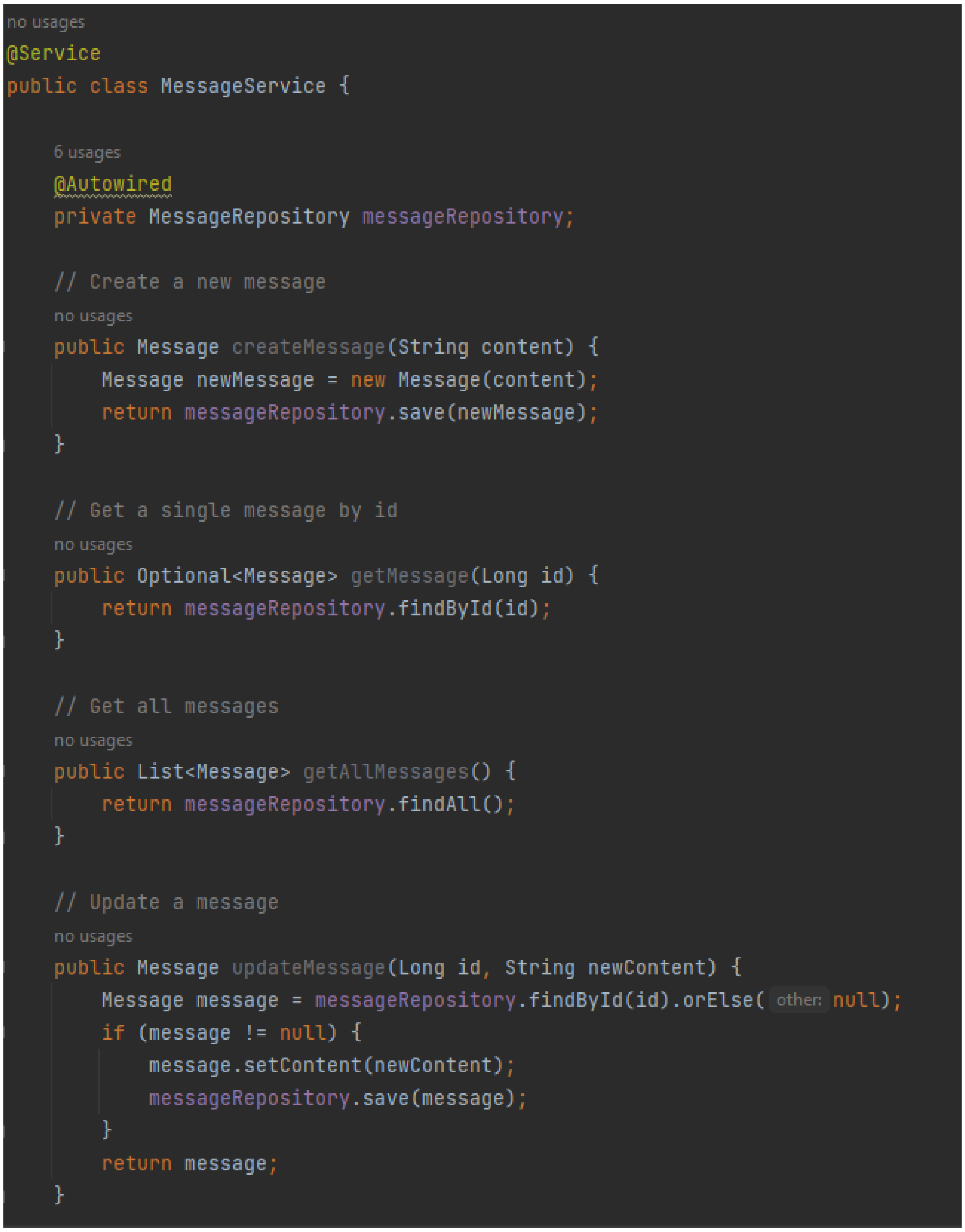
Fig 10: Security Method

4 Appendix

4.1 Mini project

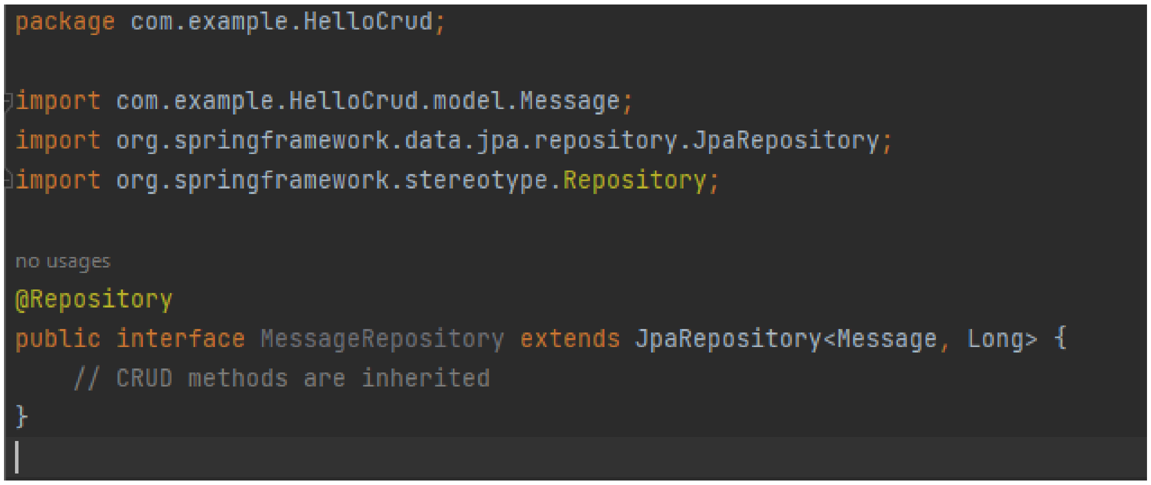
The mini-project consists of a basic hello world application in the used framework, java spring in my case, it has 4 parts: entity (which is the message ”hello world”), the repository (which makes the crud operations for the database), the service (which contains the business logic of the application) and the main class, which runs the program.


```
@Entity
public class Message {
    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    3 usages
    private String content;

    // Constructors
    no usages
    public Message() {}

    no usages
    public Message(String content) {
        this.content = content;
    }


    no usages
    public Long getId() {
        return id;
    }

    no usages
    public void setId(Long id) {
        this.id = id;
    }

    no usages
    public String getContent() {
        return content;
    }

    no usages
    public void setContent(String content) {
        this.content = content;
    }
```



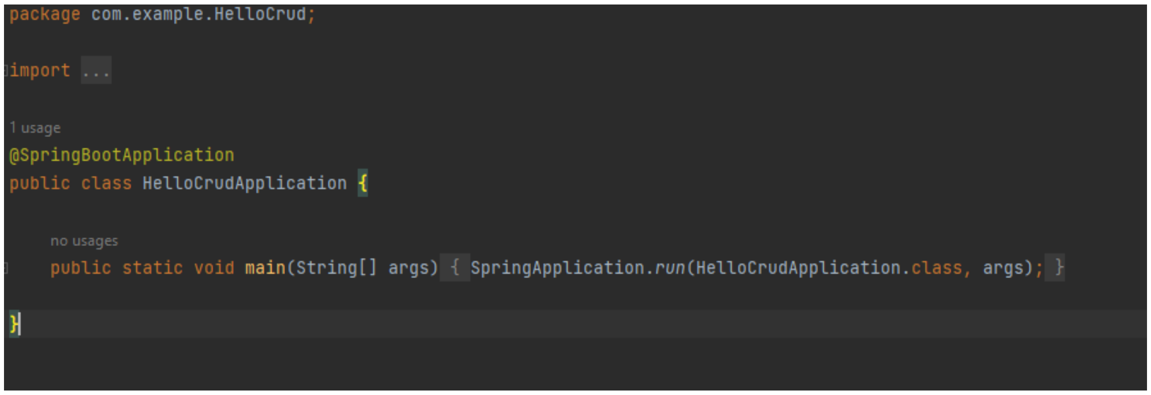
```
no usages
@Service
public class MessageService {

    6 usages
    @Autowired
    private MessageRepository messageRepository;

    // Create a new message
    no usages
    public Message createMessage(String content) {
        Message newMessage = new Message(content);
        return messageRepository.save(newMessage);
    }

    // Get a single message by id
    no usages
    public Optional<Message> getMessage(Long id) {
        return messageRepository.findById(id);
    }

    // Get all messages
    no usages
    public List<Message> getAllMessages() {
        return messageRepository.findAll();
    }

    // Update a message
    no usages
    public Message updateMessage(Long id, String newContent) {
        Message message = messageRepository.findById(id).orElse( other: null);
        if (message != null) {
            message.setContent(newContent);
            messageRepository.save(message);
        }
        return message;
    }
```

Fig 12: Service class (hello application)



```
package com.example.HelloCrud;

import com.example.HelloCrud.model.Message;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

no usages
@Repository
public interface MessageRepository extends JpaRepository<Message, Long> {
    // CRUD methods are inherited
}
```

Fig 13: Repository interface (hello application)



```
package com.example.HelloCrud;

import ...

1 usage
@SpringBootApplication
public class HelloCrudApplication {

    no usages
    public static void main(String[] args) { SpringApplication.run(HelloCrudApplication.class, args); }

}
```

Fig 14: Main class (hello application)

References

[1] *Session-Based Distributed Programming in Java* https://www.doc.ic.ac.uk/~rhu/sessionj/hyh08session-based.pdf

[2] *Spring Boot + Session Management Hello World Example* https://www.javainuse.com/spring/springboot_session

[3] *Spring Boot JDBC implementation- Hello World Example* https://www.javainuse.com/spring/bootjdbc

[4] *Deployment Diagram Tutorial* https://online.visual-paradigm.com/diagrams/tutorials/deployment-diagram-tutorial/

[5] *Diagrams tool* https://www.drawio.com/