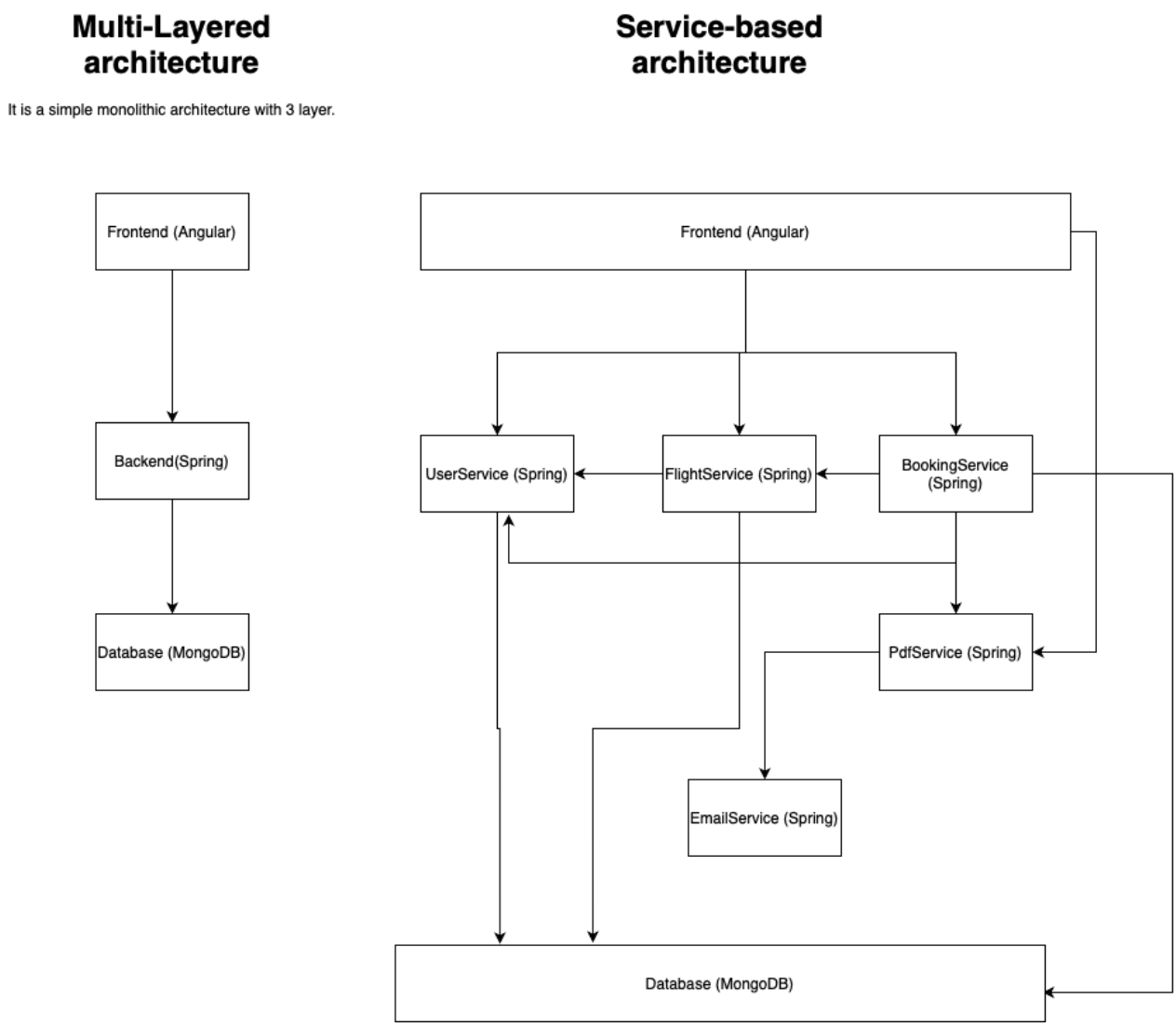
SA5 TICKET SALES

ARCHITECTURES

We provide two architecture for our projects: one monolithic and one distributed:



User

- **Registration and Login**: Users can register and log into the service. This feature allows users to access their saved bookings in the 'MyBooking' section.

```java
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable())
            .exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth ->
                    auth.requestMatchers(new AntPathRequestMatcher("/api/flights")).hasAuthority("admin")
                        .requestMatchers(new AntPathRequestMatcher("/api/bookings/myBookings")).hasAuthority("user")
                        .requestMatchers(new AntPathRequestMatcher("/api/auth/**")).permitAll()
                        .requestMatchers(new AntPathRequestMatcher("/api/flights/**")).permitAll()
                        .requestMatchers(new AntPathRequestMatcher("/api/bookings/**")).permitAll()
                                .anyRequest().authenticated()
            );

    http.authenticationProvider(authenticationProvider());
    http.cors(Customizer.withDefaults());
    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

This is how we manage the requests.

- **Password Security**: Passwords are securely stored by hashing them before saving to the database. This ensures that even if data is accessed unauthorizedly, the actual passwords remain protected.
- **Session Management with JWT**: The system maintains user sessions using JWT (JSON Web Tokens). When a user logs in, they receive a JWT, which is then included in the header of each subsequent request. This token allows the system to recognize and authenticate the user on each request, ensuring secure access to user-specific functionalities.

```java
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
    try {
        String jwt = parseJwt(request);
        if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
            String username = jwtUtils.getUserNameFromJwtToken(jwt);

            UserDetails userDetails = userService.loadUserByUsername(username);
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails, credentials: null,userDetails.getAuthorities());
            authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
    } catch (Exception e) {
        logger.error("Cannot set user authentication: {}", e);
    }

    filterChain.doFilter(request, response);
}
```

This is how we check for each request if the JWT Token is still valid.

Admin

- **Role Management**: The system incorporates role management, utilizing Spring Security's **UserDetails** interface. This feature is crucial for differentiating between regular users and administrators (admins).
- **Admin-Specific Requests**: Requests that require administrative privileges, such as creating or deleting flights, are filtered and processed only if made by an admin. This ensures that critical functionalities are securely managed and accessed only by authorized personnel.

Flight

- **Admin-Only Creation and Deletion**: Flights can only be created and deleted by admins, ensuring control over flight management.
- **User Access and Interaction**: Regular users can search for flights based on criteria like departure and arrival cities, date, and travel class (economy, business, or first-class). Additionally, users have the option to search for round-trip flights.
- **Booking Process**: After selecting a flight, users can fill out a form with their personal details to finalize the booking.

- **How we save the flights?**

saveFlight Method

```java
public Flight saveFlight(Flight flight, String size) throws IOException {
    int c=0,r=0;
    double price = priceCalculation(flight.getArrival(), flight.getDeparture(),flight.getTravelClass());
    double childrenPrice = (price*3)/4;
    System.out.println("Il prezzo "+price);
    if(size.equals("small")){
        c=10;
        r=4;

    }else if(size.equals("big")){
        c=25;
        r=6;
    }else{
        c=15;
        r=6;
    }
    ArrayList<Seat> seats= new ArrayList<>();
    char a='A';
    for(int i=0;i<c;i++){
        for(int j=0;j<r;j++){
            Seat s = new Seat( name: a+""+j, isAvailable: true,price,childrenPrice);
            seats.add(s);
        }
        int charValue = a;
        charValue++;
        a = (char) charValue;
    }
    flight.setSeats(seats);
    return flightRepository.save(flight);
}
```

Main Method

- **Purpose**: This method saves a flight object into the system.
- Parameters:
    - **Flight flight**: The flight object to be saved.
    - **String size**: A string indicating the size of the flight (small, medium, or big), which affects the number of seats.
- Seat Generation:
    - Based on the **size** parameter, the method determines the number of columns (**c**) and rows (**r**) for the seats in the flight.

    - Seats are generated in a loop and added to an **ArrayList<Seat>**. Each seat has a unique identifier (**a+""+j**), a boolean indicating availability, and prices for adults and children.
    - The seat identifier combines a letter (representing a row) and a number (representing a column).
- Price Calculation:
    - The method calls **priceCalculation** to determine the base price for a seat, based on departure and arrival locations and travel class.
- Saving the Flight:
    - The generated seats are set into the **flight** object.
    - The flight object is then saved to a repository (presumably a database) and returned.

priceCalculation Method

- **Purpose**: Calculates the price of a flight based on departure and arrival locations and travel class.
    - Procedure:
    - Retrieves geographic coordinates (latitude and longitude) for both departure and arrival cities using the **getCoordinates** method.
    - Calculates the distance between these two points using the Haversine formula (a formula that determines the great-circle distance between two points on a sphere given their longitudes and latitudes).
    - Adjusts the price based on the travel class. Business class increases the price by 40%, and first class by 80%.

getCoordinates Method

- **Purpose**: Retrieves geographic coordinates for a given city.
    - Procedure:

- Constructs a URL to query the Nominatim API (OpenStreetMap's API for geocoding) for the given city.
- Makes an HTTP GET request to the API and parses the JSON response to extract latitude and longitude.
- Returns the coordinates as a **double** array.

PDF

- **Ticket Generation**: Upon booking, the system generates a unique PDF ticket for each reservation. This ticket contains all details of the booking.
- **User Accessibility**: Users can download their unique PDF ticket, which serves as a record of their booking.

Email

- **Automatic Email on Booking**: Once a booking is made, the system automatically sends an email to the first email address provided by the user. This feature ensures that even non-registered users receive a copy of their booking confirmation, reducing the risk of losing important booking information.

Database

- **Data Storage with MongoDB**: All data, including user details, bookings, and flight information, is stored in a database managed by MongoDB. This DBMS (Database Management System) choice offers flexibility and scalability, accommodating the dynamic data needs of the system.

Security with Spring

- The security of the application is fortified using Spring Security, which provides comprehensive security services for Java applications.
- The implementation of **UserDetails** and JWT for session management are central to the system's security framework, ensuring that only authenticated and authorized users can access sensitive functionalities and data.

- This architecture provides a robust and secure framework for managing flight bookings, user authentication, and role-based access control, while ensuring data integrity and security with MongoDB as the database solution.