

System Design Document  
YorkU MarketPlace

Github: <https://github.com/EECS3311F25/Yorku-MarketPlace/tree/main>

Table of Contents:

<b>Title</b>	<b>Page Number</b>
CRC Cards	3-11
System Interaction with the Environment	12
Software Architecture Diagram	13-14
System Decomposition and Error Strategy	15

## CRC Cards

<b>Class Name: User</b>
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"><li>• Represent the state of a user (username, email, password hash, etc.).</li><li>• Map directly to the users table in the database (as a JPA @Entity).</li><li>• Store verification codes and password reset tokens with their expiry.</li></ul>
<b>Collaborators:</b> <ul style="list-style-type: none"><li>• UserRepository (saves/retrieves it)</li><li>• AuthService (creates/updates it)</li><li>• CustomUserDetailsService (reads from it)</li></ul>

<b>Class Name: UserRepository (Interface)</b>
<b>Parent Class (if any):</b> JpaRepository<User, Long> (extends this interface) <b>Subclasses (if any):</b> None (Implementation is provided at runtime by Spring Data JPA)
<b>Responsibilities:</b> <ul style="list-style-type: none"><li>• Define the contract for database operations on the User entity.</li><li>• Provide standard CRUD methods (inherited from JpaRepository).</li><li>• Define custom query methods: findByUsername, findByEmail, findByPasswordResetToken.</li></ul>
<b>Collaborators:</b> <ul style="list-style-type: none"><li>• AuthService (Uses the interface to find and save User objects)</li><li>• Spring Data JPA (Implements the interface)</li></ul>

<b>Class Name: SecurityConfig</b>
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"><li>• Configure all web security rules for the application (@Configuration).</li><li>• Define which URL paths are public (permitAll) and which require authentication (authenticated).</li><li>• Configure the behavior of form login (login page, success URL, failure URL).</li><li>• Configure the behavior of logout (logout URL, session invalidation).</li><li>• Create and provide the PasswordEncoder (BCryptPasswordEncoder) as a Spring Bean.</li><li>• Create and provide the DaoAuthenticationProvider as a Spring Bean, linking the CustomUserDetailsService with the PasswordEncoder.</li><li>• Configure all web security rules for the application (@Configuration).</li><li>• Define which URL paths are public (permitAll) and which require authentication (authenticated).</li><li>• Configure the behavior of form login (login page, success URL, failure URL).</li><li>• Configure the behavior of logout (logout URL, session invalidation).</li><li>• Create and provide the PasswordEncoder (BCryptPasswordEncoder) as a Spring Bean.</li></ul>

<ul style="list-style-type: none"> <li>● Create and provide the DaoAuthenticationProvider as a Spring Bean, linking the CustomUserDetailsService with the PasswordEncoder.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>● HttpSecurity (Spring Security class that is configured)</li> <li>● CustomUserDetailsService (Injected to create the DaoAuthenticationProvider)</li> <li>● PasswordEncoder (Used by the DaoAuthenticationProvider)</li> <li>● DaoAuthenticationProvider (Registered with HttpSecurity)</li> </ul>

<b>Class Name:</b> AuthService
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>● Encapsulate all business logic for authentication (@Service).</li> <li>● Find users by username or email.</li> <li>● Validate user login credentials (password and verification status).</li> <li>● Create new, unverified users and hash their passwords (signupUnverified).</li> <li>● Verify a user's account using a verification code (verifyUser).</li> <li>● Generate, validate, and process password reset tokens.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>● UserRepository (to fetch and save User data)</li> <li>● BCryptPasswordEncoder (to hash and check passwords)</li> <li>● AuthController (delegates business logic to this service)</li> <li>● CustomUserDetailsService (uses this service to find users)</li> </ul>

<b>Class Name:</b> CustomUserDetailsService
<b>Parent Class (if any):</b> Implements UserDetailsService (interface) <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>● Act as the bridge between the application's User model and Spring Security's UserDetails interface.</li> <li>● Implement the loadUserByUsername method to find a user during the login process.</li> <li>● Fetch the application User using the AuthService.</li> <li>● Convert the User object into a UserDetails object that Spring Security can understand (providing username, password hash, and authorities).</li> <li>● Throw UsernameNotFoundException if the user is not found.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>● AuthService (To find the User object by username)</li> <li>● User (The model class it reads from)</li> <li>● UserDetails (The Spring Security object it creates)</li> <li>● SecurityConfig (Injects this service into the DaoAuthenticationProvider)</li> </ul>

<b>Class Name:</b> EmailService
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>● Abstract the details of sending emails (@Service).</li> <li>● Construct and send the account verification email (sendVerificationEmail).</li> <li>● Construct and send the password reset link email (sendPasswordResetEmail).</li> <li>● Handle exceptions that may occur during the email sending process.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>● JavaMailSender (A Spring utility that is injected to send the email).</li> <li>● SimpleMailMessage (A Spring class used to build the email message).</li> <li>● AuthController (Calls this service to send emails).</li> </ul>

<b>Class Name:</b> AuthController
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>● Handle user-facing HTTP requests for authentication (@Controller).</li> <li>● Map URL paths (/login, /signup, /verify, etc.) to specific Java methods.</li> <li>● Serve the HTML view names for forms (e.g., return "login", "signup").</li> <li>● Process form submissions (@PostMapping), validating simple user input (e.g., email domain, matching passwords).</li> <li>● Delegate all complex business logic (like creating users or validating tokens) to the AuthService.</li> <li>● Trigger emails by calling the EmailService.</li> <li>● Manage temporary user state during verification using HttpSession.</li> <li>● Add error/success messages to the Model or RedirectAttributes for the view to display.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>● AuthService (To perform all authentication and user management logic)</li> <li>● EmailService (To send verification and password reset emails)</li> <li>● HttpSession (To store the username during the verification process)</li> <li>● Model (To pass data to the view on the current request)</li> <li>● RedirectAttributes (To pass data (e.g., errors) across a redirect)</li> <li>● HttpServletRequest (To get server info like the base URL)</li> </ul>

<b>Class Name:</b> MarketplaceApplication
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None

<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Serve as the main entry point for the application.</li> <li>• Bootstrap and launch the entire Spring Boot application (SpringApplication.run).</li> <li>• Enable Spring Boot's auto-configuration, component scanning, and configuration properties (@SpringBootApplication).</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• SpringApplication (The Spring Boot class used to run the application).</li> </ul>

Sprint 2:

<b>Class Name:</b> FileController
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Handle HTTP GET requests for retrieving uploaded files (images) via the /uploads/{filename} endpoint (@RestController).</li> <li>• Dynamically resolve the file path on the server's local file system using the configured file.upload-dir property.</li> <li>• Validate that the requested file exists on the disk before attempting to serve it.</li> <li>• Convert the file path into a Spring UriResource to be sent back to the client.</li> <li>• Return a ResponseEntity containing the file resource and the appropriate Content-Type (MediaType.IMAGE_JPEG).</li> <li>• Return a 404 Not Found response if the file does not exist.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• <b>Paths / Files</b> (Java NIO classes used to locate and verify the file path).</li> <li>• <b>UriResource</b> (Spring class used to wrap the file for the response).</li> <li>• <b>ResponseEntity</b> (Spring class used to build the HTTP response).</li> <li>• <b>@Value</b> (Spring annotation used to inject the storage directory path).</li> </ul>

<b>Class Name:</b> ListingController
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Handle user-facing HTTP requests for managing market listings (@Controller).</li> <li>• Serve the HTML view for creating a new listing (showCreateListing).</li> <li>• Process the listing creation form (createListing), binding form data to the Listing model.</li> <li>• Handle MultipartFile image uploads by delegating to the FileStorageService.</li> <li>• Retrieve the currently authenticated user via AuthService to assign ownership of the listing.</li> <li>• Display specific listing details and determine if the viewer is the owner (viewListing).</li> <li>• Enforce security checks (ownership validation) before showing edit forms or processing deletions.</li> <li>• Delegate business logic for updating and deleting listings to the ListingService.</li> <li>• Manage navigation flow (redirecting to home, listing details, or access-denied pages).</li> </ul>

<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• <b>ListingService</b> (Handles the core CRUD business logic for listings).</li> <li>• <b>AuthService</b> (Used to retrieve the full User object of the currently logged-in user).</li> <li>• <b>FileStorageService</b> (Used to save uploaded images and generate URLs).</li> <li>• <b>Listing</b> (The model object being created or updated).</li> <li>• <b>Model</b> (Passes data like the listing object or ownership status to the view).</li> <li>• <b>Principal / Authentication</b> (Spring Security interfaces used to identify the current user).</li> </ul>

<b>Class Name:</b> Listing
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Represent the state of an item for sale (title, description, price, image URL).</li> <li>• Map directly to the listings table in the database (as a JPA @Entity).</li> <li>• Maintain the relationship with the User who owns/posted the item (Many-to-One).</li> <li>• Store metadata such as the creation timestamp (createdAt).</li> <li>• Provide boilerplate code (getters, setters, constructors) via Lombok annotations to other layers.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• <b>User</b> (The owner object associated with this listing).</li> <li>• <b>ListingRepository</b> (Saves and retrieves Listing objects).</li> <li>• <b>ListingService</b> (Contains business logic that manipulates Listing objects).</li> <li>• <b>ListingController</b> (Binds form data to Listing objects).</li> </ul>

<b>Class Name:</b> ListingRepository (Interface)
<b>Parent Class (if any):</b> JpaRepository<Listing, Long> (extends this interface) <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Define the contract for database operations on the Listing entity.</li> <li>• Provide standard CRUD methods (inherited from JpaRepository).</li> <li>• Define a custom query method (findByOwner) to retrieve a list of items posted by a specific user.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• <b>ListingService</b> (Uses the interface to find and save Listing objects)</li> <li>• <b>Spring Data JPA</b> (Implements the interface)</li> <li>• <b>Listing</b> (The entity managed by this repository)</li> <li>• <b>User</b> (Used as a parameter for finding listings)</li> </ul>

<b>Class Name:</b> WebSocketConfig
<b>Parent Class (if any):</b> java.lang.Object (Implements WebSocketMessageBrokerConfigurer) <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Configure the message broker to handle real-time messaging using Spring's WebSocket support (@EnableWebSocketMessageBroker).</li> <li>• Enable a simple memory-based message broker to send messages back to the client on destinations prefixed with "/topic".</li> <li>• Define the application prefix "/app" to filter messages destined for @RequestMapping methods in the Controller.</li> <li>• Register the "/ws-chat" STOMP endpoint to allow clients to initiate the handshake.</li> <li>• Enable SockJS fallback options to ensure connectivity for browsers that do not support native WebSockets.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• <b>MessageBrokerRegistry</b> (Configures message broker settings and prefixes).</li> <li>• <b>StompEndpointRegistry</b> (Registers STOMP endpoints and enables SockJS)</li> <li>• <b>ChatController</b> (Implicitly relies on the "/app" prefix configured here to route messages).</li> </ul>

<b>Class Name:</b> WebSocketConfig
<b>Parent Class (if any):</b> java.lang.Object (Implements WebSocketMessageBrokerConfigurer) <b>Subclasses (if any):</b> None
<b>Responsibilities:</b> <ul style="list-style-type: none"> <li>• Configure the message broker to handle real-time messaging using Spring's WebSocket support (@EnableWebSocketMessageBroker).</li> <li>• Enable a simple memory-based message broker to send messages back to the client on destinations prefixed with "/topic".</li> <li>• Define the application prefix "/app" to filter messages destined for @RequestMapping methods in the Controller.</li> <li>• Register the "/ws-chat" STOMP endpoint to allow clients to initiate the handshake.</li> <li>• Enable SockJS fallback options to ensure connectivity for browsers that do not support native WebSockets.</li> </ul>
<b>Collaborators:</b> <ul style="list-style-type: none"> <li>• <b>MessageBrokerRegistry</b> (Configures message broker settings and prefixes).</li> <li>• <b>StompEndpointRegistry</b> (Registers STOMP endpoints and enables SockJS)</li> <li>• <b>ChatController</b> (Implicitly relies on the "/app" prefix configured here to route messages).</li> </ul>

<b>Class Name:</b> ChatController
<b>Parent Class (if any):</b> java.lang.Object <b>Subclasses (if any):</b> None
<b>Responsibilities:</b>



- Handle user-facing HTTP requests for the messaging interface (@Controller).
- Map URL paths (/inbox, /chat) to specific Java methods for rendering views.
- Serve the inbox HTML page and populate it with the user's active conversation threads (getInbox).
- Serve the chat room HTML page and generate a unique, deterministic room ID for participants (getChatPage).
- Process incoming real-time messages sent via WebSockets (@MessageMapping).
- Parse the composite chat identifier (ListingID\_UserA\_UserB) to extract listing information.
- Delegate saving/persisting new messages to the ChatService.
- Broadcast received messages to the correct topic subscription using SimpMessagingTemplate.

**Collaborators:**

- **ChatService** → Handles business logic for fetching chat history and saving new messages.
- **AuthService** → Retrieves the current authenticated user.
- **SimpMessagingTemplate** → Sends messages to subscribed WebSocket clients.
- **Model** → Passes attributes (e.g., chat history) to the HTML view.
- **Principal** → Identifies the currently logged-in user.

**Class Name: ChatMessageDTO**

**Parent Class (if any):** java.lang.Object

**Subclasses (if any):** None

**Responsibilities:**

- Encapsulates raw message data sent from the client via WebSockets.
- Contains required payload fields: chatId, senderId, receiverId, content.
- Serves as a DTO to decouple the WebSocket API layer from internal ChatMessage domain entities.
- Uses Lombok @Data to auto-generate getters, setters, toString, equals, and hashCode.

**Collaborators:**

- ChatController → Receives this object as the @Payload in WebSocket message-handling methods.
- ChatService → Extracts DTO data to create and persist ChatMessage entities.
- Lombok → Provides compile-time annotation processing for boilerplate code.

**Class Name: InboxConversationDTO**

**Parent Class (if any):** java.lang.Object

**Subclasses (if any):** None

**Responsibilities:**

- Bundle the information needed to display an inbox conversation preview.
- Hold the listing associated with the conversation.
- Store the other user involved in the chat (besides the current user).
- Contain the last message exchanged in the conversation for preview purposes.
- Use Lombok @Data + @AllArgsConstructor to auto-generate boilerplate (getters, setters, constructor, toString, etc.).

**Collaborators:**

- **Listing** (domain model representing the associated marketplace listing).
- **User** (domain model representing the other participant in the conversation).
- **InboxController / ChatController** (uses this DTO to populate inbox UI).
- **Lombok** (generates boilerplate code at compile time).

**Class Name:** ChatMessage

**Parent Class (if any):** java.lang.Object

**Subclasses (if any):** None

**Responsibilities:**

- Represent a persistent chat message entity stored in the database.
- Store message content, timestamp, sender, recipient, and listing association.
- Map to the chat\_messages table via JPA annotations.
- Automatically generate boilerplate (getters, setters, constructors, etc.) using Lombok (@Data, @NoArgsConstructor, @AllArgsConstructor).
- Maintain relationships to:
  - User (sender)
  - User (recipient)
  - Listing (associated listing)
- Automatically initialize timestamp with LocalDateTime.now() if not set.

**Collaborators:**

- User model (for sender and recipient associations).
- Listing model (identifies the listing tied to the conversation).
- ChatService (creates, saves, and retrieves ChatMessage entities).
- ChatController (invokes the service to send or load messages).
- JPA/Hibernate (handles ORM mapping and persistence).

**Class Name:** ChatMessageRepository

**Parent Class (if any):** JpaRepository<ChatMessage, Long>

**Subclasses (if any):** None

**Responsibilities:**

- Provide database access methods for ChatMessage entities.
- Inherit CRUD operations (save, delete, findById, findAll) from JpaRepository.
- Fetch full chat history between two users for a specific listing using a compound derived query: findByIdAndSenderIdAndRecipientIdOrListingIdAndSenderIdAndRecipientIdOrderByTimestampAsc(..)
- Handles both directions ( $A \rightarrow B$  and  $B \rightarrow A$ ).
- Retrieve all messages where a given user is involved (sender or recipient) for building the inbox view.
- Use custom JPQL (@Query) to return messages ordered by newest first.

**Collaborators:**

- **ChatMessage** (entity being queried).
- **ChatService** (calls repository methods to load histories and inbox messages).

- **Spring Data JPA** (parses method names, generates queries, and executes JPQL).
- **User** (domain model representing the other participant in the conversation).
- **InboxController / ChatController** (uses this DTO to populate inbox UI).
- **Lombok** (generates boilerplate code at compile time).

**Class Name:** ChatMessage

**Parent Class (if any):** JpaRepository<ChatMessage, Long>

**Subclasses (if any):** None

**Responsibilities:**

- Represent a persistent chat message entity stored in the chat\_messages table.
- Store message data, including:
  - content (text of the message)
  - timestamp (auto-initialized using LocalDateTime.now())
  - sender (User who sent the message)
  - recipient (User receiving the message)
  - listing (Marketplace listing associated with the conversation)
- Define ORM mappings using JPA annotations (@Entity, @Table, @ManyToOne, etc.).
- Generate boilerplate code (getters, setters, constructors, equals, hashCode, toString) via Lombok annotations (@Data, @NoArgsConstructor, @AllArgsConstructor).

**Collaborators:**

- **User** (sender and recipient references).
- **Listing** (attaches each message to a specific listing context).
- **ChatMessageRepository** (queries and stores ChatMessage entities).
- **ChatService** (creates and retrieves messages from the database).
- **ChatController** (requests message persistence and retrieval).

**Class Name:** ChatService

**Parent Class (if any):** java.lang.Object

**Subclasses (if any):** None

**Responsibilities:**

- Handle all business logic related to chat messaging.
- Save new chat messages to the database (saveMessage).
- Retrieve ordered chat history between two users for a specific listing (getChatHistory).
- Fetch and build a user's inbox conversation list (getUserConversations).
- Identify the "other user" involved in each conversation.
- Produce InboxConversationDTO objects for display in the inbox UI.
- Prevent duplicate conversation entries using a composite key (listingId + "\_" + otherUserId).

**Collaborators:**

- **ChatMessageRepository** – Performs database operations for retrieving and storing messages.
- **AuthService** – Fetches sender and recipient User objects.
- **ListingService** – Retrieves the Listing associated with the conversation.
- **ChatMessage** – Domain entity created, populated, and persisted by the service.

- **InboxConversationDTO** – Structures conversation preview data for the inbox.

## System Interaction with the Environment

This section outlines the dependencies and assumptions made about the operating environment for the Marketplace application to function correctly.

- **Operating System (OS):** The application is built using Java and the Spring Boot framework, which are platform-independent. It is assumed the system will be deployed on any modern operating system (like Linux, Windows Server, or macOS) that has a compatible **Java Virtual Machine (JVM)**.
- **Programming Language & Virtual Machine:**
  - **Java:** The application is written in the Java programming language.
  - **JVM:** It requires a compatible Java Virtual Machine to run. The system requires **Java 17 or newer**.
- **Core Frameworks & Libraries:** The system is fundamentally dependent on the **Spring Boot** framework, which orchestrates all components. Key dependencies include:
  - **Spring Web:** To handle HTTP requests, controllers (AuthController), and REST-like interactions.
  - **Spring Security:** For all authentication, authorization, password hashing (BCryptPasswordEncoder), and web security configuration (SecurityConfig).
  - **Spring Data JPA:** To manage database interactions. It provides the implementation for the UserRepository.
  - **Hibernate:** This is the default JPA implementation used by Spring Data JPA to map the Userobject (@Entity) to the database.
  - **Spring Mail:** Required by the EmailService to send emails via JavaMailSender.
  - **Lombok:** Used at compile-time to reduce boilerplate code in model classes like User (e.g., @Data).
- **Databases (DBs):**
  - The system requires a **SQL-compatible relational database** (e.g., PostgreSQL, MySQL, or H2 for development) to persist the User entity.
  - The database **connection details** (URL, username, password, and driver) are provided in an external configuration file.
- **External Services & Network Configuration:**
  - **SMTP Mail Server:** The EmailService has a critical external dependency on a functioning **SMTP server**. The application's environment must be configured with the correct host, port, and credentials for this server to successfully send verification and password reset emails.
  - **HTTP/HTTPS Port:** The application runs an embedded web server (Tomcat by default) and must be able to bind to a specific network port to listen for incoming user requests.
- **Build & View:**
  - **Build Tool:** The e project uses a standard Java build tool like **Maven** to manage its dependencies and package the application into a runnable .jar file.
  - **View Technology:** The AuthController returns String view names (e.g., "login", "home").
  -

## Software Architecture Diagram:

The architecture of this system is a classic **3-Tier (or N-Tier) Web Application Architecture**. This is a well-known and widely used pattern that separates the application into distinct logical layers:

1. **Presentation Tier (Web Layer):** This is the user-facing layer, responsible for handling HTTP requests and presenting information.
2. **Business Tier (Service Layer):** This layer contains the core business logic and rules of the application.
3. **Data Tier (Data Layer):** This layer is responsible for all data storage and retrieval.

This design directly implements the **Model-View-Controller (MVC)** pattern, which is the foundation of the Spring framework:

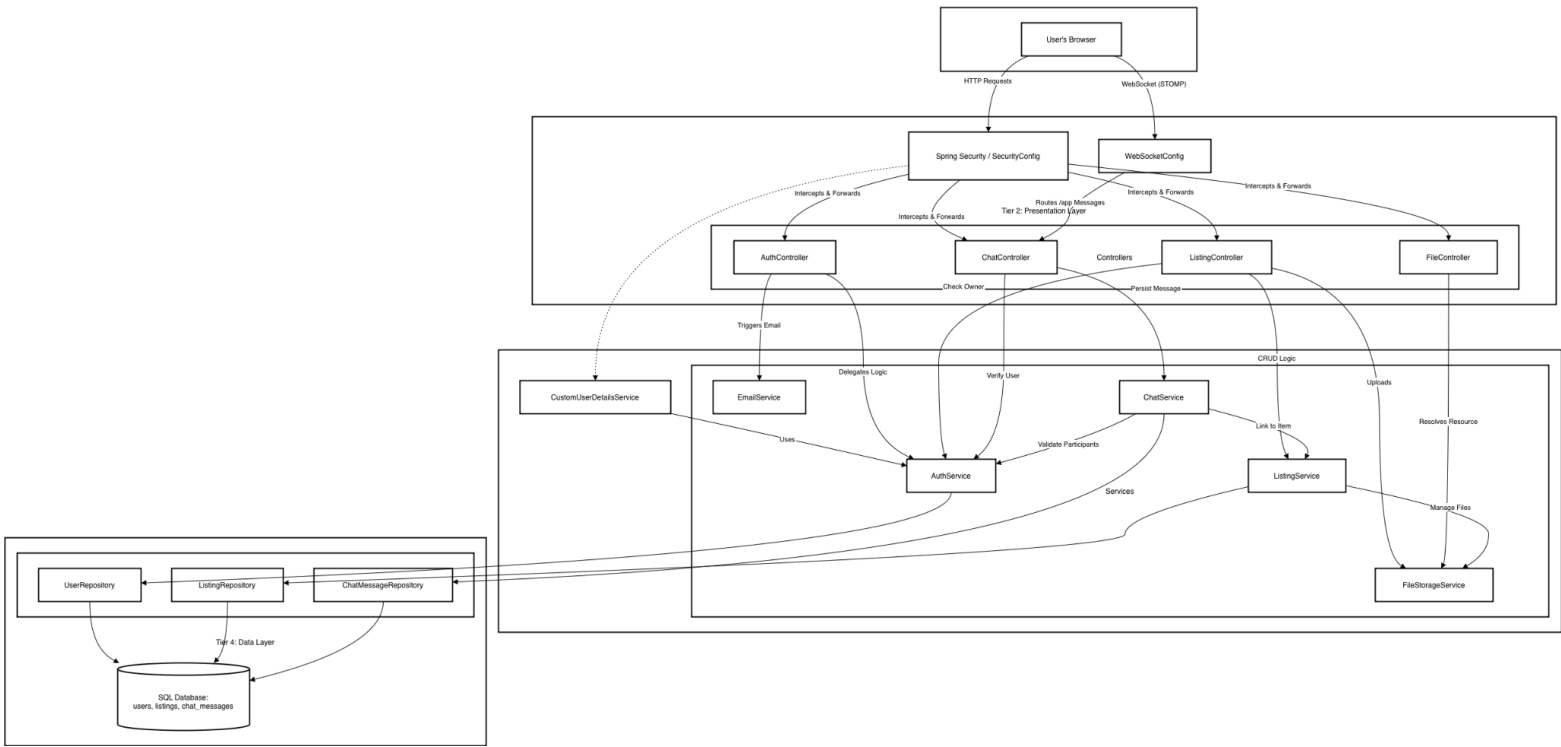
- **Model (M):** The Business and Data Tiers (AuthService, EmailService, User, UserRepository).
- **View (V):** The Thymeleaf/HTML files (e.g., login.html, home.html).
- **Controller (C):** The AuthController in the Presentation Tier.

A **Security Layer** (provided by Spring Security) also wraps the Presentation Tier to intercept and protect all incoming requests.

Explanation of Software Architecture:

- **Tier 1: User (Client):** This is the user's web browser, which sends **HTTP Requests** (e.g., GET /login, POST /signup).
- **Tier 2: Presentation Layer:**
  - **SecurityConfig:** This component acts as the first "gate." It intercepts all requests, deciding which are public (like /login) and which require authentication (like /home). It also defines how to log in (by using the UserDetailsService).
  - **AuthController:** This is the **Controller (C)**. Its role is to map URL paths to specific Java methods. It handles web-specific data (HttpSession, Model) and delegates all business logic to the service layer.
- **Tier 3: Business Layer:**
  - **AuthService:** This is the primary **Model (M)** component and the "brain" of the application. It contains all the core logic (password hashing, token generation, account verification) and is completely independent of the web.
  - **EmailService:** A specialized **Model (M)** component. Its single responsibility is to manage the external communication with an SMTP mail server.
  - **CustomUserDetailsService:** A special **Model (M)** component that acts as a bridge between the Business Layer and the Security Layer. Its job is to load user data for Spring Security.
- **Tier 4: Data Layer:**
  - **UserRepository:** A **Model (M)** component that defines the contract for data access (e.g., "we need to findByUsername").
  - **Database (and JPA):** The physical database stores the data. Spring Data JPA is the (unseen) tool that writes the SQL to execute the UserRepository's methods. The User class is the **@Entity** that maps to this database.

Software Architectural Diagram



## System Decomposition and Error Strategy

### System Decomposition

#### Strategy for Errors and Exceptional Cases

The system uses a layered strategy to handle errors, responding differently based on the type of error.

##### 1. Invalid User Input:

- **Examples:** Passwords do not match (POST /reset-password), email does not end in @yorku.ca (POST /signup), username is already taken.
- **Strategy:** These errors are caught in the AuthController before any service logic is run.
- **Response:** The method adds an "error" message to the Model or RedirectAttributes and returns the user to the same form (e.g., returns "signup"). This allows the user to see the error and correct their input.

##### 2. Authentication & Authorization Failures:

- **Examples:** User enters an incorrect password, user's verification token is expired or invalid, a non-logged-in user tries to access /home.
- **Strategy:** These are handled by the SecurityConfig or by explicit validation in the AuthController (e.g., authService.validatePasswordResetToken).
- **Response:** The user is securely redirected to a "safe" page, typically the login page. SecurityConfig redirects to /login?error=true, and the AuthController redirects with a flash attribute explaining the token was invalid.

##### 3. External System & Network Failures:

- **Examples:** The EmailService fails to connect to the SMTP server, the database is offline.
- **Strategy:** These runtime exceptions are caught using try-catch blocks inside the AuthController (e.g., the try-catch around emailService.sendVerificationEmail).
- **Response:** The application does not crash. It catches the exception, logs it, and adds a generic error to the Model (e.g., "Could not send verification email. Please try again later."). The user is returned to the form, understanding that a system-level (not input-level) error occurred.