System Design Document
YorkU MarketPlace

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

Table of Contents:

CRC Cards

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

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

Class Name: SecurityConfig
Parent Class (if any): java.lang.Object **Subclasses (if any):** None
Responsibilities: ● Configure all web security rules for the application (@Configuration). ● Define which URL paths are public (permitAll) and which require authentication (authenticated). ● Configure the behavior of form login (login page, success URL, failure URL). ● Configure the behavior of logout (logout URL, session invalidation). ● Create and provide the PasswordEncoder (BCryptPasswordEncoder) as a Spring Bean. ● Create and provide the DaoAuthenticationProvider as a Spring Bean, linking the CustomUserDetailsService with the PasswordEncoder. ● Configure all web security rules for the application (@Configuration). ● Define which URL paths are public (permitAll) and which require authentication (authenticated). ● Configure the behavior of form login (login page, success URL, failure URL). ● Configure the behavior of logout (logout URL, session invalidation). ● Create and provide the PasswordEncoder (BCryptPasswordEncoder) as a Spring Bean.

● Create and provide the DaoAuthenticationProvider as a Spring Bean, linking the CustomUserDetailsService with the PasswordEncoder.
Collaborators: ● HttpSecurity (Spring Security class that is configured) ● CustomUserDetailsService (Injected to create the DaoAuthenticationProvider) ● PasswordEncoder (Used by the DaoAuthenticationProvider) ● DaoAuthenticationProvider (Registered with HttpSecurity)

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

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

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

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

Class Name: MarketplaceApplication
Parent Class (if any): java.lang.Object **Subclasses (if any):** None

Responsibilities:
● Serve as the main entry point for the application.
● Bootstrap and launch the entire Spring Boot application (SpringApplication.run).
● Enable Spring Boot's auto-configuration, component scanning, and configuration properties (@SpringBootApplication).

Collaborators:
● SpringApplication (The Spring Boot class used to run the application).

Sprint 2/3:

Class Name:FileController

Parent Class (if any): java.lang.Object
Subclasses (if any): None

Responsibilities:
- Handle HTTP GET requests for retrieving uploaded files (images) via the /uploads/{filename} endpoint (@RestController).
- Dynamically resolve the file path on the server's local file system using the configured file.upload-dir property.
- Validate that the requested file exists on the disk before attempting to serve it.
- Convert the file path into a Spring UrlResource to be sent back to the client.
- Return a ResponseEntity containing the file resource and the appropriate Content-Type (MediaType.IMAGE_JPEG).
- Return a 404 Not Found response if the file does not exist.

Collaborators:
- **Paths / Files** (Java NIO classes used to locate and verify the file path).
- **UrlResource** (Spring class used to wrap the file for the response).
- **ResponseEntity** (Spring class used to build the HTTP response).
- **@Value** (Spring annotation used to inject the storage directory path).

Class Name:ListingController

Parent Class (if any): java.lang.Object
Subclasses (if any): None

Responsibilities:
- Handle user-facing HTTP requests for managing market listings (@Controller).
- Serve the HTML view for creating a new listing (showCreateListing).
- Process the listing creation form (createListing), binding form data to the Listing model.
- Handle MultipartFile image uploads by delegating to the FileStorageService.
- Retrieve the currently authenticated user via AuthService to assign ownership of the listing.
- Display specific listing details and determine if the viewer is the owner (viewListing).
- Enforce security checks (ownership validation) before showing edit forms or processing deletions.
- Delegate business logic for updating and deleting listings to the ListingService.
- Manage navigation flow (redirecting to home, listing details, or access-denied pages).

Collaborators: ● **ListingService** (Handles the core CRUD business logic for listings). ● **AuthService** (Used to retrieve the full User object of the currently logged-in user). ● **FileStorageService** (Used to save uploaded images and generate URLs). ● **Listing** (The model object being created or updated). ● **Model** (Passes data like the listing object or ownership status to the view). ● **Principal / Authentication** (Spring Security interfaces used to identify the current user).

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

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

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)

- **User's Browser**: Initiates HTTP Requests (e.g., POST /login, POST /create-listing, GET /uploads/image.jpg).

Tier 2: Presentation Layer (View/Controller)

- **Spring Security (SecurityConfig)**: The "Gatekeeper." Intercepts **all** requests to specific paths (both Auth and Listing paths) before they reach the controllers.
- **Controllers (C)**:
    1. **AuthController**: Handles login/signup/verify.
    2. **ListingController (NEW)**: Handles /create-listing, /listing/{id}, and editing.
    3. **FileController (NEW)**: Handles serving images via /uploads/{filename}.

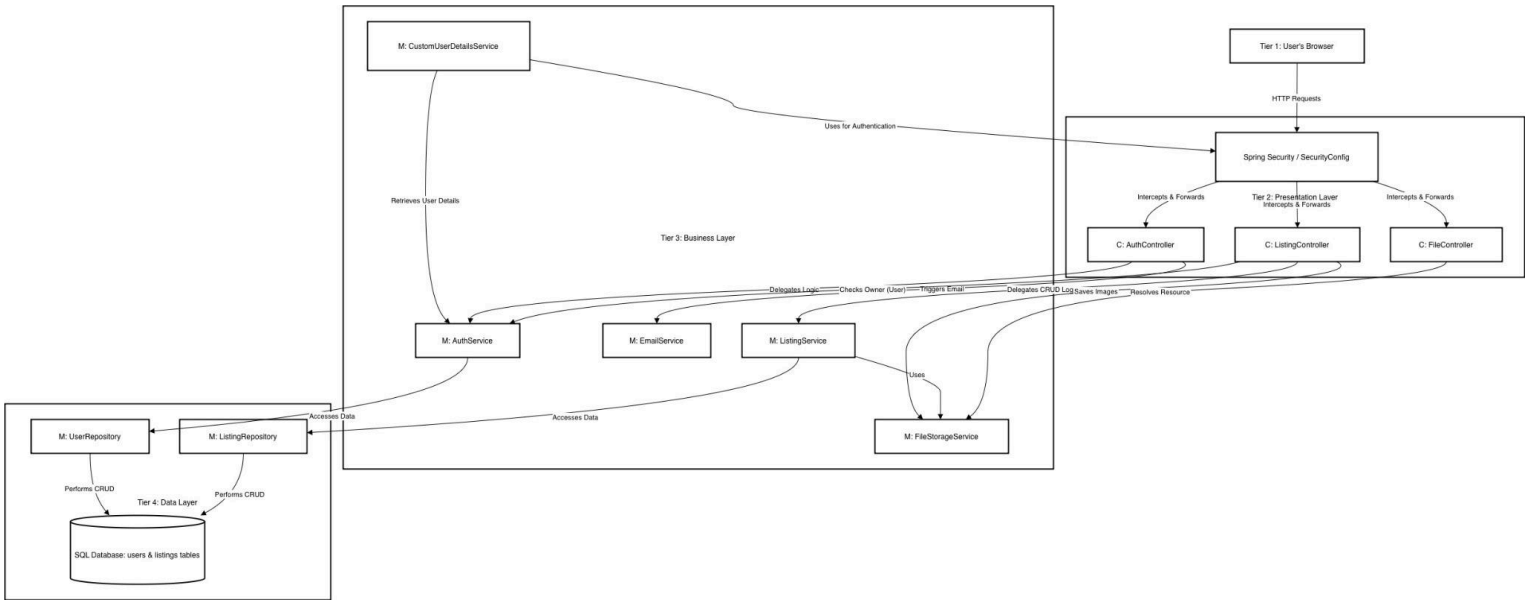
Tier 3: Business Layer (Model/Service)

- **CustomUserDetailsService**: Uses AuthService to bridge User data to Spring Security.
- **Services (M)**:
    1. **AuthService**: Core logic for users/authentication.
    2. **EmailService**: External communication with SMTP.
    3. **ListingService (NEW)**: Contains business logic for creating, updating, and deleting listings.
    4. **FileStorageService (NEW)**: Logic for saving MultipartFile uploads to the disk and resolving paths.

Tier 4: Data Layer (Model/Repository)

- **Repositories**:
    - **UserRepository**: Interface for User CRUD operations.
    - **ListingRepository (NEW)**: Interface for Listing CRUD operations and finding by owner.
- **SQL Database**:
    - Now contains **two** primary tables: users and listings (mapped by the new Listing entity).

Software Architectural Diagram



System Decomposition and Error Strategy

System Decomposition

- **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.

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. SecurityConfigredirects 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.