VIETNAM NATIONAL UNIVERSITY – HCM
INTERNATIONAL UNIVERSITY

SEMESTER 1 (2025-2026)

Lecturer: Nguyen Van Sinh

Lab Lecturer: Nguyen Trung Nghia

Web Application Development

TOPIC: Hyper bid web game

Team Members

Full Name	Student ID
Pham Trung Kien	ITCSIU23020
Nguyen Ha An Thanh	ITITWE22051
Tran Thi Kieu My	ITCSIU23026

Github Repository:
https://github.com/Nobodywinsbutme/tyk-webapp

Contents

1 Abstract

This report describes the development of a web-based game item marketplace created for the Web Application Development course. The system is built using Spring Boot 3, Java 17, MySQL, and Thymeleaf, and follows a monolithic architecture combining RESTful APIs with server-side rendering. The application allows users to trade virtual items using in-application currency, ensuring data consistency through transactional processing. This document outlines the system objectives, architecture, core

functionalities, API design, database schema, user workflows, testing approach, and deployment guidelines.

2 Project overview

2.1 Project name

Hyper bid web game (skeleton)

2.2 Game description

This project is a web-based game application that provides a virtual environment where users can register, authenticate, manage in-game items, and trade them through an integrated marketplace. The system uses an in-application currency to support buying and selling activities and ensures data consistency during transactions. It is developed using Spring Boot with Java for the backend, MySQL for data storage, and HTML, CSS, and JavaScript for the frontend, following a monolithic architecture that combines RESTful services with server-side rendering.

3 Requirements

3.1 Functional Requirements

- The system shall allow users to register a new account with unique credentials.
- The system shall allow users to log in and log out using session-based authentication.
- The system shall provide a user profile endpoint to retrieve account information (e.g., coin balance).
- The system shall allow users to browse marketplace listings with optional filtering (type, price range, keyword).
- The system shall allow users to list items for sale from their inventory with a specified price and quantity.
- The system shall allow users to purchase an active listing and automatically update both users' coin balances.
- The system shall transfer purchased item quantities to the buyer's inventory after a successful purchase.
- The system shall allow sellers to cancel their listings and return items to their inventory.
- The system shall allow sellers to update the price of an active listing.
- The system shall allow users to view their inventory with filtering and pagination, including an "On Sale" tab.

- The system shall allow users to upload community designs with images and metadata (title, description, category).
- The system shall allow users to view their own uploaded designs and manage them (edit/delete).
- The system shall allow the public to view approved community designs.
- The system shall allow administrators to review pending designs and approve or reject them.
- The system shall provide a news feed and allow administrators to create, update, and delete news posts.

3.2 Non-Functional Requirements

- **Security:** Passwords shall be stored using strong hashing (BCrypt).
- **Consistency:** Marketplace purchase operations shall be executed within database transactions to prevent partial updates.
- **Usability:** The UI shall provide clear feedback for common actions (login, listing, purchase, upload, moderation).
- **Maintainability:** The application shall follow a layered structure (Controller, Service, Repository, Entity, DTO).
- **Scalability:** The database schema shall support growth in users, inventory items, listings, designs, and news posts.

4 System Architecture

4.1 Architecture Style

The application follows a **monolithic architecture** implemented using Spring Boot. It combines:

- **Server-Side Rendering (SSR)** using Thymeleaf templates for main pages.
- **REST APIs** for asynchronous operations (authentication, inventory, marketplace, designs, news).

Architecture diagram:

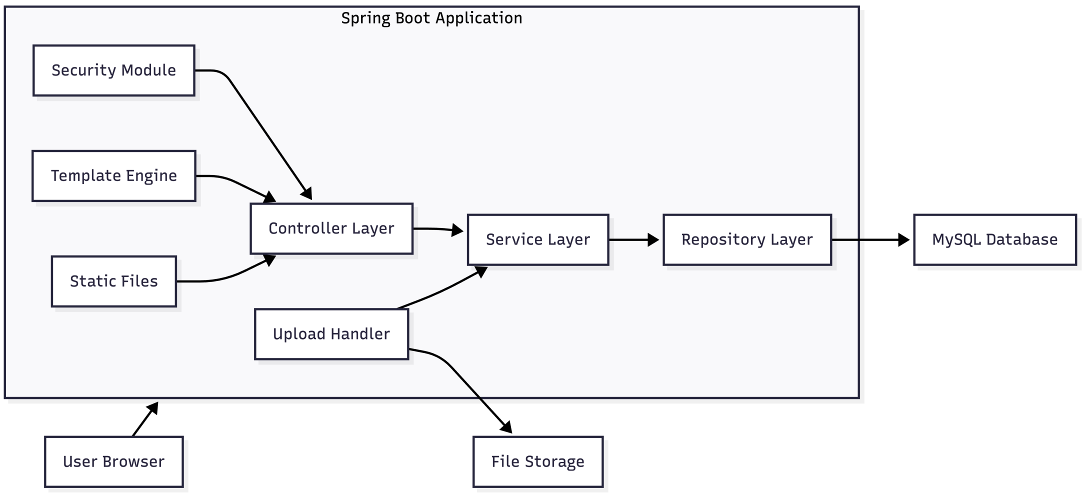


Figure 1: High-level architecture of the web-based game application.

5 Technology stack

Backend	Java 17 with Spring Boot 3, using Spring MVC and Spring Data JPA for business logic and data access.
Frontend	HTML5, CSS (Bootstrap 5), and JavaScript (Vanilla JS with Fetch API), combined with Thymeleaf for server-side rendering.
Database	MySQL relational database, managed through JPA/Hibernate.
Authentication	Spring Security with session-based authentication and BCrypt password hashing.
Build Tools	Apache Maven for dependency management and build automation.
Containerization	Docker (Dockerfile provided for application deployment).
Version Control	Git with GitHub for source code management and collaboration.

6 API list

The following table summarizes the core REST APIs implemented in the system.

Endpoint	**Method**	**Description**	**Auth**
/api/auth/register	POST	Register a new user account	No
/api/auth/login	POST	Authenticate user and create session	No
/api/auth/logout	POST	Invalidate current user session	Yes
/api/auth/profile/{username}	GET	Retrieve user profile and coin balance	Yes
/api/market/listings	GET	Retrieve market-place listings with filters	No
/api/market/sell/{userId}	POST	List an item on the market	Yes
/api/market/buy/{userId}/{listingId}	POST	Purchase a market-place listing	Yes
/api/market/{listingId}	DELETE	Cancel an active marketplace listing	Yes
/api/market/{listingId}	PUT	Update the price of a listing	Yes
/api/inventory/{userId}	GET	Retrieve user inven-tory with filters	Yes
/api/inventory/sell/{userId}	POST	Sell item directly from inventory	Yes
/api/designs/upload	POST	Upload a new com-munity design	Yes
/api/designs/public	GET	Retrieve approved public designs	No
/api/designs/my-designs	GET	Retrieve designs uploaded by current user	Yes
/api/designs/pending	GET	Retrieve pending de-signs for moderation	Yes (Admin)
/api/designs/{id}	DELETE	Delete a specific de-sign	Yes
/api/designs/{id}	PUT	Update design de-tails	Yes
/api/designs/{id}/status	PUT	Approve or reject a design submission	Yes (Admin)
/api/news/list	GET	Retrieve published news articles	No
/api/news/create	POST	Create a new news article	Yes (Admin)
/api/news/update/{id}	PUT	Update an existing news article	Yes (Admin)
/api/news/delete/{id}	DELETE	Delete a news article	Yes (Admin)

Table 1: Summary of Implemented REST APIs

6.1 Register

6.1.1 Register Entry Point – AuthController

```java
@PostMapping("/register")
public ResponseEntity<?> register(
        @RequestBody RegisterRequestDTO request) {
    authService.register(request);
    return ResponseEntity.status(HttpStatus.CREATED).body("User
   registered successfully");
}
```

Listing 1: Register endpoint in AuthController

6.1.2 Registration Logic – AuthService

```java
public void register(RegisterRequestDTO request) {

    if (userRepository.existsByUsername(request.getUsername())) {
        throw new RuntimeException("Username already exists");
    }

    if (userRepository.existsByEmail(request.getEmail())) {
        throw new RuntimeException("Email already exists");
    }

    User user = new User();
    user.setUsername(request.getUsername());
    user.setEmail(request.getEmail());
    user.setPassword(passwordEncoder.encode(request.getPassword()));
    user.setRole(Role.USER);
    user.setCoinBalance(1000L);

    userRepository.save(user);
}
```

Listing 2: Register logic implemented in AuthService

6.1.3 Register Request DTO

```java
public class RegisterRequestDTO {
    private String username;
    private String email;
    private String password;
}
```

Listing 3: Register request data transfer object

6.1.4 User Persistence

```java
userRepository.save(user);
```

Listing 4: User repository usage during registration

6.2 Explain workflow

1. The user fills in the registration form with a username, email, and password.
2. The client sends a registration request to the backend endpoint:`POST /api/auth/register`
3. The request is received by the authentication controller (`AuthController`).
4. The controller forwards the request to the authentication service by calling `AuthService.regist`
5. The authentication service performs the following operations:
   - Validates the registration data (e.g., username and email uniqueness).
   - Encrypts the user's password using BCrypt.
   - Creates a new `User` entity with the provided information.
   - Assigns default values such as role and initial coin balance.
6. The new user entity is saved to the database.
7. The database stores the new user record permanently.
8. The system returns a successful response to the client, confirming that the account has been created.

6.3 Login

6.3.1 Login Entry Point – AuthController

```java
@PostMapping("/login")
public ResponseEntity<?> login(
        @RequestBody LoginRequestDTO request,
        HttpServletRequest httpRequest) {
    return authService.login(request, httpRequest);
}
```

Listing 5: Login endpoint in AuthController

6.3.2 Authentication Logic – AuthService

```java
public ResponseEntity<?> login(
        LoginRequestDTO request,
        HttpServletRequest httpRequest) {

    Authentication authentication =
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                request.getUsername(),
                request.getPassword()
            )
        );

    SecurityContextHolder.getContext().setAuthentication(
  authentication);
```

```java
    httpRequest.getSession(true);

    User user = userRepository
        .findByUsername(request.getUsername())
        .orElseThrow();

    return ResponseEntity.ok(
        new LoginResponseDTO(
            user.getUsername(),
            user.getRole().name(),
            user.getCoinBalance()
        )
    );
}
```

Listing 6: Login logic implemented in AuthService

6.3.3 Login Request DTO

```java
public class LoginRequestDTO {
    private String username;
    private String password;
}
```

Listing 7: Login request data transfer object

6.3.4 Security Configuration

6.4 Explain workflow

1. The user enters a username and password in the login form and submits the request.
2. The client sends a login request to the backend endpoint:`POST /api/auth/login`
3. The request is received by the authentication controller (`AuthController`).
4. The controller forwards the request to the authentication service (`AuthService`).
5. The authentication service validates the credentials using Spring Security's authentication manager.
6. If authentication is successful:
   - A server-side HTTP session is created.
   - The security context is updated to mark the user as authenticated.
7. The system retrieves the user's information from the database.
8. A successful response containing user information is returned to the client.
9. The user is now considered logged in and can access protected resources.

6.5 Logout

6.5.1 Logout Entry Point – AuthController

```java
@PostMapping("/logout")
public ResponseEntity<?> logout(HttpServletRequest request) {
    request.getSession().invalidate();
    SecurityContextHolder.clearContext();
    return ResponseEntity.ok("Logged out successfully");
}
```

Listing 8: Logout endpoint in AuthController

6.5.2 Session Termination

```java
request.getSession().invalidate();
SecurityContextHolder.clearContext();
```

Listing 9: Session invalidation during logout

6.5.3 Security Context Cleanup

The logout functionality is implemented by invalidating the server-side session and clearing the Spring Security context, ensuring that the user's authenticated state is fully terminated.

6.6 Explain workflow

6.7 Marketplace Sell Implementation

6.7.1 Sell Endpoint – MarketRestController

```java
@PostMapping("/sell/{userId}")
public ResponseEntity<?> sellItem(@PathVariable Long userId,
   @RequestBody SellRequestDTO request) {
    try {
        marketService.sellItem(userId, request);
        return ResponseEntity.ok("Item listed successfully");
    } catch (Exception e) {
        return ResponseEntity.badRequest().body(e.getMessage());
    }
}
```

Listing 10: Sell endpoint in MarketRestController

6.7.2 Sell Logic – MarketService

```java
@Transactional
public void sellItem(Long userId, SellRequestDTO request) {
    InventoryItem item = inventoryRepository.findById(request.
   getItemId())
        .orElseThrow(() -> new RuntimeException("Item not found"));

    if (!item.getUser().getId().equals(userId)) {
        throw new RuntimeException("You don't own this item");
    }

    if (item.getQuantity() < request.getQuantity()) {
        throw new RuntimeException("Not enough items!");
    }

    // Create listing
    MarketListing listing = new MarketListing();
    listing.setSeller(item.getUser());
    listing.setItemDefinition(item.getItemDefinition());
    listing.setPrice(request.getPrice());
    listing.setQuantity(request.getQuantity());
    listing.setStatus("ACTIVE");
    listing.setListedAt(LocalDateTime.now());

    marketRepository.save(listing);

    // Deduct from inventory
    int newQuantity = item.getQuantity() - request.getQuantity();
    if (newQuantity > 0) {
        item.setQuantity(newQuantity);
        inventoryRepository.save(item);
    } else {
        inventoryRepository.delete(item);
    }
}
```

Listing 11: Sell logic implemented in MarketService

6.7.3 Sell Request DTO

```java
public class SellRequestDTO {
    private Long itemId;
    private Long price;
    private int quantity;
}
```

Listing 12: SellRequestDTO fields used by the sell workflow

6.7.4 Transactional Consistency

Summary: The marketplace sell feature is implemented through a REST endpoint in `MarketRestController` and business logic in `MarketService`. It validates ownership and quantity, creates an active listing, and updates the seller's inventory within a transactional boundary.

6.8 Explain workflow

6.9 Marketplace Buy Implementation

6.9.1 Buy Endpoint – MarketRestController

```java
@PostMapping("/buy/{userId}/{listingId}")
public ResponseEntity<?> buyItem(@PathVariable Long userId,
   @PathVariable Long listingId) {
    try {
        marketService.buyItem(userId, listingId);
        return ResponseEntity.ok("Purchase successful");
    } catch (Exception e) {
        return ResponseEntity.badRequest().body(e.getMessage());
    }
}
```

Listing 13: Buy endpoint in MarketRestController

6.9.2 Buy Logic – MarketService

```java
@Transactional
public void buyItem(Long buyerId, Long listingId) {
    MarketListing listing = marketRepository.findById(listingId)
        .orElseThrow(() -> new RuntimeException("Listing not found"))
   ;

    if (!"ACTIVE".equals(listing.getStatus())) {
        throw new RuntimeException("Item unavailable");
    }

    if (listing.getSeller().getId().equals(buyerId)) {
        throw new RuntimeException("Cannot buy your own item");
    }

    User buyer = userRepository.findById(buyerId).orElseThrow();
    User seller = listing.getSeller();

    if (buyer.getCoinBalance() < listing.getPrice()) {
        throw new RuntimeException("Not enough coins!");
    }

    // Coin transfer
    buyer.setCoinBalance(buyer.getCoinBalance() - listing.getPrice())
   ;
    seller.setCoinBalance(seller.getCoinBalance() + listing.getPrice
   ());

    // Transfer items to buyer inventory
    inventoryService.addItemToInventory(buyer, listing.
   getItemDefinition(), listing.getQuantity());

    // Finalize listing
    listing.setStatus("SOLD");
    marketRepository.save(listing);

    userRepository.save(buyer);
    userRepository.save(seller);
}
```

Listing 14: Buy logic implemented in MarketService

6.9.3 Inventory Transfer via InventoryService

```java
inventoryService.addItemToInventory(buyer, listing.getItemDefinition
    (), listing.getQuantity());
```

Listing 15: Inventory update call during purchase

6.9.4 Transactional Consistency

Summary: The marketplace buy feature is implemented through a REST endpoint in `MarketRestController` and transactional business logic in `MarketService`. It validates listing availability and buyer balance, performs coin transfer, updates the buyer's inventory, and finalizes the listing status in an atomic transaction.

6.10 Explain workflow

6.11 Inventory View Implementation

6.11.1 Inventory Page Route – InventoryViewController

```java
@GetMapping("/inventory")
public String showInventoryPage(Model model, HttpSession session,
    Principal principal) {
    User currentUser = userService.getAuthenticatedUser(session,
    principal);

    if (currentUser == null) {
        if (session != null) {
            session.invalidate();
        }
        return "redirect:/?message=expired";
    }
    model.addAttribute("user", currentUser);
    return "inventory";
}
```

Listing 16: SSR route for the inventory page

6.11.2 Inventory View API – InventoryRestController

```java
@GetMapping("/{userId}")
public ResponseEntity<?> getUserInventory(
        @PathVariable Long userId,
        @RequestParam(required = false) String type,
        @RequestParam(required = false, defaultValue = "all") String
    tab,
        @RequestParam(required = false) String rarity,
        @RequestParam(required = false) String search,
        @RequestParam(defaultValue = "1") int page,
        @RequestParam(defaultValue = "12") int size
) {
    return ResponseEntity.ok(inventoryService.getMyInventory(userId,
    tab, type, rarity, search, page, size));
}
```

Listing 17: Inventory REST API with filtering and pagination

6.11.3 Inventory Filtering and Tabs – InventoryService

```java
public Page<InventoryResponseDTO> getMyInventory(Long userId, String
    tab, String typeStr, String rarityStr,
                                                 String keyword, int
    page, int size) {

    ItemType type = null;
    if (typeStr != null && !typeStr.isEmpty() && !typeStr.
    equalsIgnoreCase("ALL")) {
        try { type = ItemType.valueOf(typeStr.toUpperCase()); } catch
     (IllegalArgumentException e) { type = null; }
    }

    ItemRarity rarity = null;
    if (rarityStr != null && !rarityStr.isEmpty() && !rarityStr.
    equalsIgnoreCase("ALL")) {
        try { rarity = ItemRarity.valueOf(rarityStr.toUpperCase()); }
     catch (IllegalArgumentException e) { rarity = null; }
    }

    Pageable pageable = PageRequest.of(page - 1, size, Sort.by("id").
    descending());

    // Tab "On Sale" -> read from MarketListing
    if ("on_sale".equalsIgnoreCase(tab)) {
        Page<MarketListing> marketPage =
            marketListingRepository.findBySellerIdAndStatus(userId, "
    ACTIVE", pageable);

        return marketPage.map(listing -> InventoryResponseDTO.builder
    ()
                .id(listing.getId())
                .name(listing.getItemDefinition().getName())
                .imageUrl(listing.getItemDefinition().getImageUrl())
                .type(listing.getItemDefinition().getType().toString
    ())
```

```java
                .rarity(listing.getItemDefinition().getRarity().toString())
                .quantity(listing.getQuantity())
                .description(listing.getItemDefinition().getDescription())
                .price(listing.getPrice())
                .listedAt(listing.getListedAt().toString())
                .isTradable(true)
                .isBookmarked(false)
                .build());
    }

    // Tab "Owned" (default) -> read from InventoryItem
    Page<InventoryItem> rawPage =
        inventoryRepository.findByUserAndFilter(userId, type, rarity, keyword, pageable);

    return rawPage.map(item -> InventoryResponseDTO.builder()
            .id(item.getId())
            .name(item.getItemDefinition().getName())
            .imageUrl(item.getItemDefinition().getImageUrl())
            .type(item.getItemDefinition().getType().toString())
            .rarity(item.getItemDefinition().getRarity() != null ? item.getItemDefinition().getRarity().toString() : "COMMON")
            .quantity(item.getQuantity())
            .description(item.getItemDefinition().getDescription())
            .isTradable(true)
            .isBookmarked(false)
            .build());
}
```

Listing 18: Inventory view logic with Owned and On-Sale tabs

6.11.4 Filtered Inventory Query – InventoryRepository

```java
@Query("SELECT i FROM InventoryItem i " +
       "JOIN i.itemDefinition d " +
       "WHERE i.user.id = :userId " +
       "AND (:type IS NULL OR d.type = :type) " +
       "AND (:rarity IS NULL OR d.rarity = :rarity) " +
       "AND (:keyword IS NULL OR LOWER(d.name) LIKE LOWER(CONCAT('%', :keyword, '%')))")
Page<InventoryItem> findByUserAndFilter(
        @Param("userId") Long userId,
        @Param("type") ItemType type,
        @Param("rarity") ItemRarity rarity,
        @Param("keyword") String keyword,
        @Param("pageable") Pageable pageable
);
```

Listing 19: JPA query for filtering inventory items

6.11.5 Frontend Inventory Loading – inventory.html

```
async function loadInventory(page = 1) {
    const search = document.getElementById('searchInput').value;

    let url = `/api/inventory/${userId}?page=${page}&size=12&tab=${currentTab}`;
    if (currentType) url += `&type=${currentType}`;
    if (currentRarity) url += `&rarity=${currentRarity}`;
    if (search) url += `&search=${search}`;

    const response = await fetch(url);
    const data = await response.json();

    renderItems(data.content);
    renderPagination(data.totalPages, page);
}
```

Listing 20: Frontend fetch call to load inventory data

Summary: The inventory view is implemented using a combination of server-side rendered navigation (`/inventory`) and a REST API (`/api/inventory/{userId}`) for dynamic data loading. The backend supports filtering, pagination, and tab-based views (Owned vs On Sale), while the frontend fetches and renders the paginated results using JavaScript.

6.12 Explain workflow

6.13 Upload a New Community Design Implementation

6.13.1 Upload Endpoint – DesignController

```
@PostMapping(value = "/upload", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public ResponseEntity<?> uploadDesign(
        @RequestParam("title") String title,
        @RequestParam("description") String description,
        @RequestParam("category") String category,
        @RequestParam(value = "image", required = false) MultipartFile image,
        @AuthenticationPrincipal UserDetails userDetails) {

    try {
        DesignResponseDTO response = designService.createDesign(
            title, description, category, image, userDetails.getUsername()
        );
        return ResponseEntity.ok(response);
    } catch (Exception e) {
        return ResponseEntity.badRequest().body("Error uploading design: " + e.getMessage());
    }
}
```

Listing 21: Upload endpoint in DesignController

6.13.2 Upload Logic – DesignService

```java
public DesignResponseDTO createDesign(String title, String desc, String category,
                                      MultipartFile image, String username) throws IOException {
    User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new RuntimeException("User not found"));

    DesignSubmission design = new DesignSubmission();
    design.setTitle(title);
    design.setDescription(desc);
    design.setCreator(user);

    // Admin uploads are approved immediately; user uploads are pending
    if ("ADMIN".equals(user.getRole())) {
        design.setStatus(DesignSubmission.SubmissionStatus.APPROVED);
        design.setApprovedBy(user);
    } else {
        design.setStatus(DesignSubmission.SubmissionStatus.PENDING);
    }

    try {
        design.setCategory(DesignSubmission.DesignCategory.valueOf(category.toUpperCase()));
    } catch (Exception e) {
        throw new RuntimeException("Category not valid");
    }

    if (image != null && !image.isEmpty()) {
        design.setImageUrl(saveFile(image));
    } else {
        throw new RuntimeException("Must have an image!");
    }

    DesignSubmission saved = designRepository.save(design);
    return DesignResponseDTO.fromEntity(saved);
}
```

Listing 22: Create design logic implemented in DesignService

6.13.3 Image Storage – saveFile

```java
private String saveFile(MultipartFile file) throws IOException {
    if (!Files.exists(UPLOAD_DIR)) Files.createDirectories(UPLOAD_DIR);

    String originalName = file.getOriginalFilename();
    String extension = originalName != null && originalName.contains(".")
                        ? originalName.substring(originalName.lastIndexOf("."))
                        : ".jpg";

    String newName = UUID.randomUUID().toString() + extension;
    Path path = UPLOAD_DIR.resolve(newName);
    Files.copy(file.getInputStream(), path, StandardCopyOption.REPLACE_EXISTING);

    return "/uploads/" + newName;
}
```

Listing 23: Image saving logic in DesignService

6.13.4 Serving Uploads – WebConfig

```java
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    Path uploadDir = Paths.get("./uploads");
    String uploadPath = uploadDir.toUri().toString();
    if (!uploadPath.endsWith("/")) {
        uploadPath += "/";
    }
    registry.addResourceHandler("/uploads/**")
            .addResourceLocations(uploadPath);
}
```

Listing 24: Static resource mapping for /uploads/**

6.13.5 Design Entity – DesignSubmission

```java
@Entity
@Table(name = "design_submissions")
public class DesignSubmission {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String title;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Column(length = 500)
    private String imageUrl;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private DesignCategory category;

    @Enumerated(EnumType.STRING)
    private SubmissionStatus status = SubmissionStatus.PENDING;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "creator_id", nullable = false)
    private User creator;

    @Column(updatable = false)
    private LocalDateTime createdAt;

    @PrePersist
    protected void onCreate() {
        this.createdAt = LocalDateTime.now();
    }
}
```

Listing 25: DesignSubmission entity fields

Summary: The community design upload feature is implemented through the multipart upload endpoint in `DesignController` and the business logic in `DesignService`. The system stores uploaded images on the server file system, persists design metadata in the database, and applies a moderation workflow where user uploads are marked as `PENDING` while admin uploads can be approved immediately.

6.14 Explain workflow

6.15 Delete a Specific Design Implementation

6.15.1 Delete Endpoint – DesignController

```java
@DeleteMapping("/{id}")
public ResponseEntity<?> deleteDesign(@PathVariable Long id,
                                      @AuthenticationPrincipal
   UserDetails userDetails) {
    try {
        designService.deleteDesign(id, userDetails.getUsername());
        return ResponseEntity.ok("Design deleted successfully");
    } catch (Exception e) {
        return ResponseEntity.badRequest().body("Error deleting
   design: " + e.getMessage());
    }
}
```

Listing 26: Delete endpoint in DesignController

6.15.2 Delete Logic – DesignService

```java
@Transactional
public void deleteDesign(Long designId, String username) {
    User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new RuntimeException("User not found"));

    DesignSubmission design = designRepository.findById(designId)
            .orElseThrow(() -> new RuntimeException("Design not found"));

    // Authorization: allow owner or admin
    boolean isOwner = design.getCreator().getId().equals(user.getId());
    boolean isAdmin = "ADMIN".equalsIgnoreCase(user.getRole());

    if (!isOwner && !isAdmin) {
        throw new RuntimeException("You are not allowed to delete this design");
    }

    // (Optional) delete the uploaded image file if stored locally
    // deleteImageIfExists(design.getImageUrl());

    designRepository.delete(design);
}
```

Listing 27: Delete design logic implemented in DesignService

6.15.3 Database Deletion – DesignRepository

```java
designRepository.delete(design);
```

Listing 28: Repository delete operation

6.15.4 Transactional Consistency

Summary: The design deletion feature is implemented through the REST endpoint `DELETE /api/designs/{id}` in `DesignController` and the authorization-protected business logic in `DesignService`. Only the creator of the design or an administrator is permitted to delete a submission, and the deletion is performed transactionally through

6.16 Explain workflow

7 Database design (ERD)

7.1 Suggested Entities

- **User** (`users`): stores account information such as `id`, `username`, `email`, hashed `password`, `role`, `coinBalance`, and moderation fields (e.g., banned status).
- **ItemDefinition** (`item_definitions`): defines game item metadata including `id`, `name`, `description`, `type`, `rarity`, `imageUrl`, and `basePrice`.

- **InventoryItem** (`inventory_items`): represents items owned by users with `id`, `userId` (FK), `itemId` (FK to ItemDefinition), and `quantity`.
- **MarketListing** (`market_listings`): stores marketplace sale listings with `id`, `sellerId` (FK to User), `itemDefinitionId` (FK), `price`, `quantity`, `status` (e.g., ACTIVE/SOLD), and `listedAt`.
- **DesignSubmission** (`design_submissions`): stores community design uploads with `id`, `title`, `description`, `imageUrl`, `category`, `status` (PENDING/APPROVED/REJECTED), `creatorId` (FK), `approverId` (FK), and `createdAt`.
- **News** (`news`): stores announcements/news posts with `id`, `title`, `description`, `createdAt`, and `authorId` (FK to User).
- **Transaction** (`transactions`): stores trading history with `id`, `buyerId` (FK), `sellerId` (FK), `amount`, `transactionDate`, and a `type` field to classify transaction actions.

ERD:

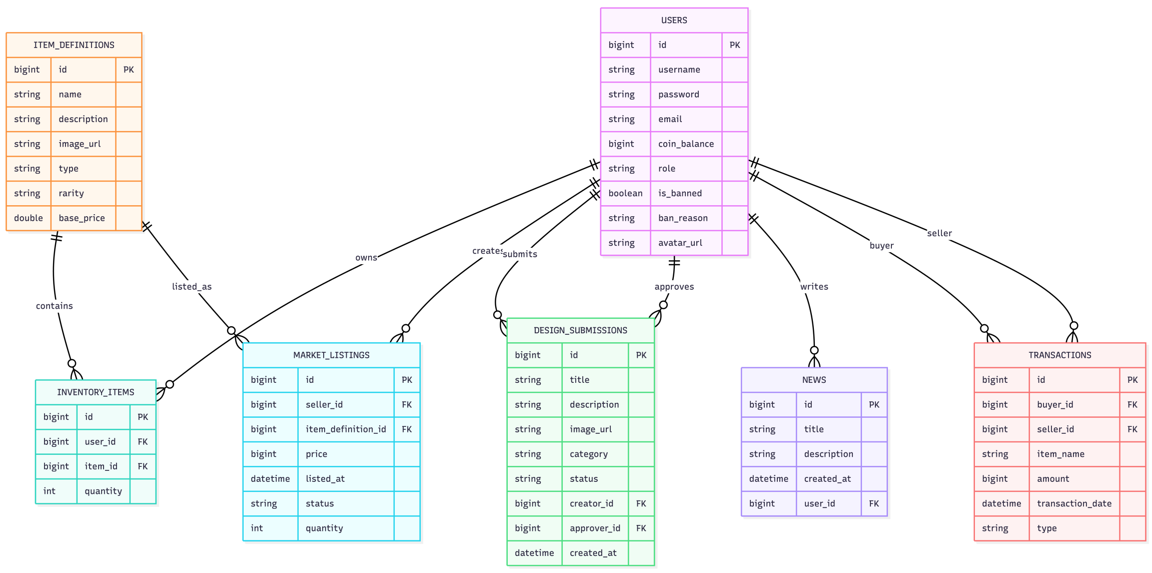


Figure 2: Entity Relationship Diagram (ERD).

8 Workflows and User Journeys

This section describes the main user flows implemented in the system:

- User registration and login (session-based authentication)
- Inventory browsing and filtering (Owned vs On Sale, search, pagination)
- Marketplace trading flow (sell item, buy item, cancel listing, update price)
- Community design submission and moderation (upload, pending review, approve/reject)
- Admin content management (news create/update/delete)

Workflow diagram:

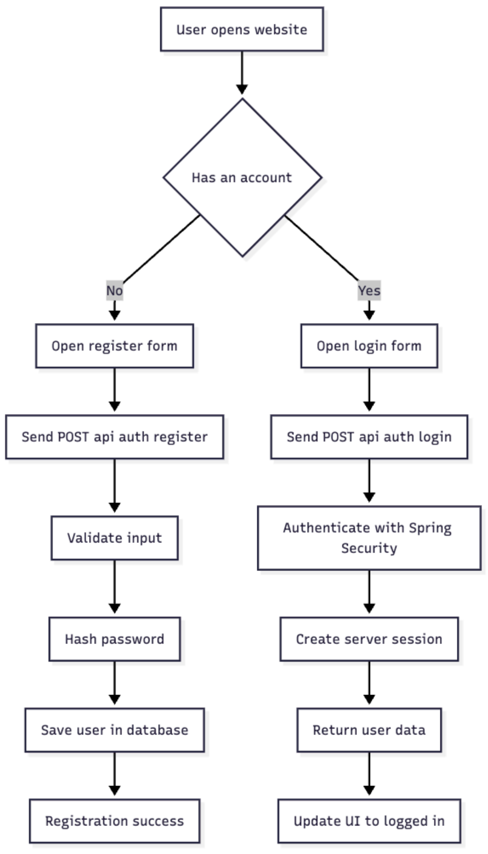


Figure 3: Auth flow (Register + Login)

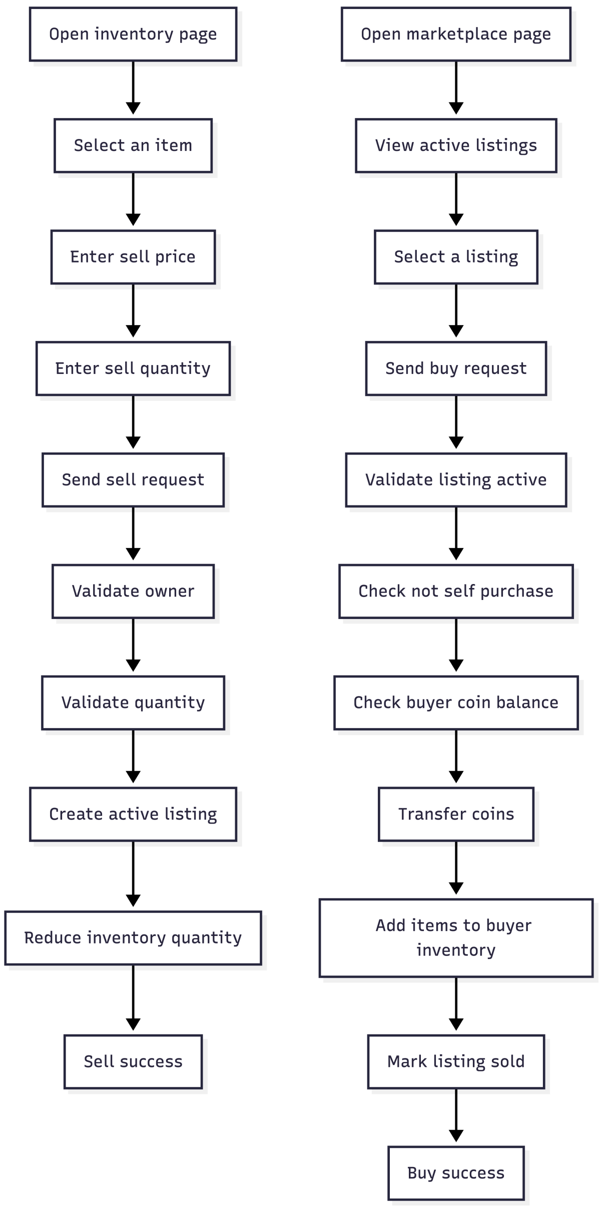


Figure 4: Marketplace trading workflow (sell and buy)

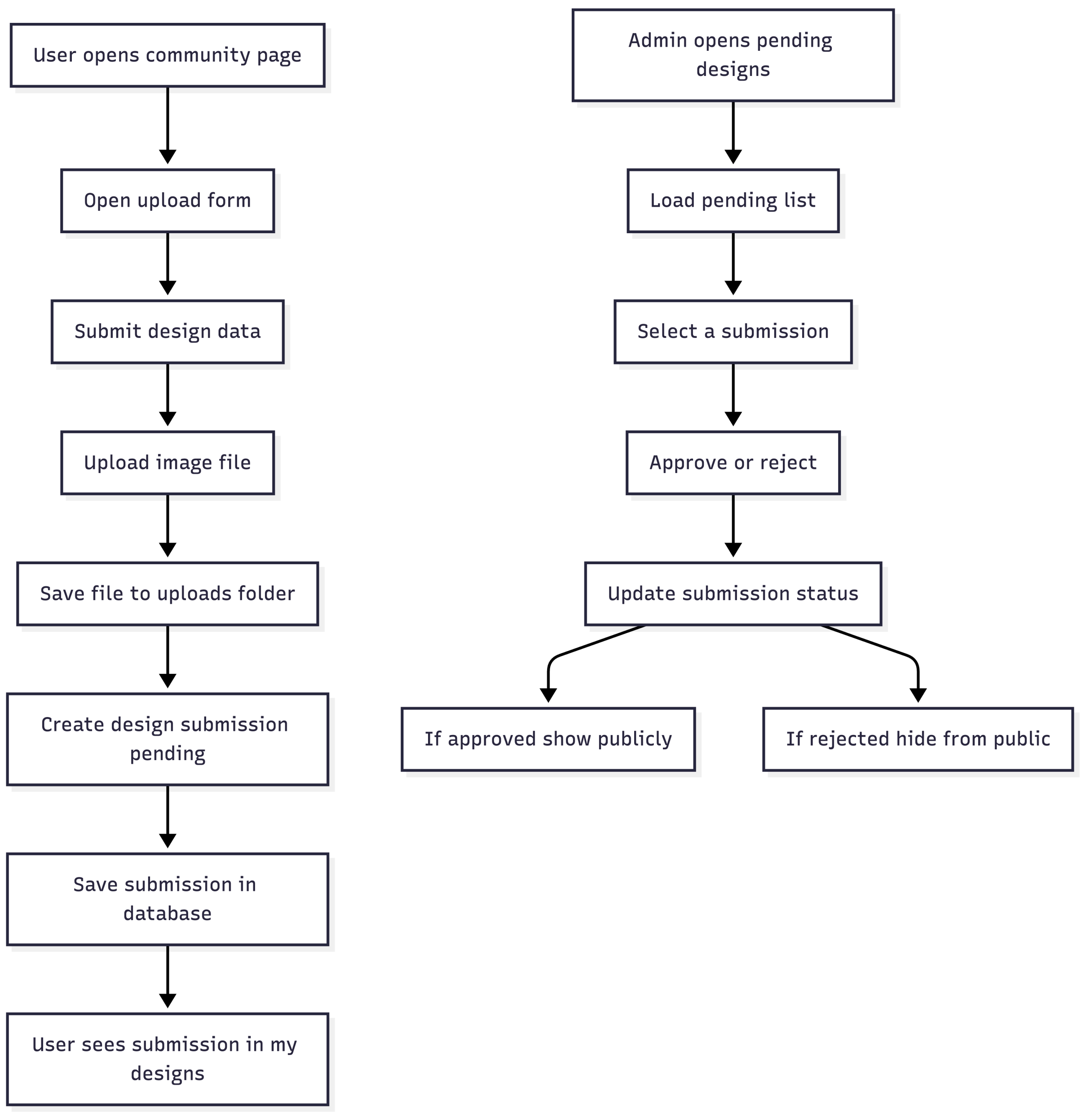
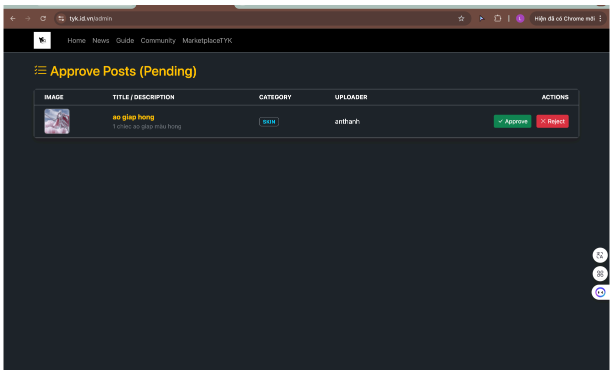

User opens community page
Open upload form
Submit design data
Upload image file
Save file to uploads folder
Create design submission pending
Save submission in database
User sees submission in my designs
Admin opens pending designs
Load pending list
Select a submission
Approve or reject
Update submission status
If approved show publicly
If rejected hide from public

Figure 5: Design upload + admin moderation flow

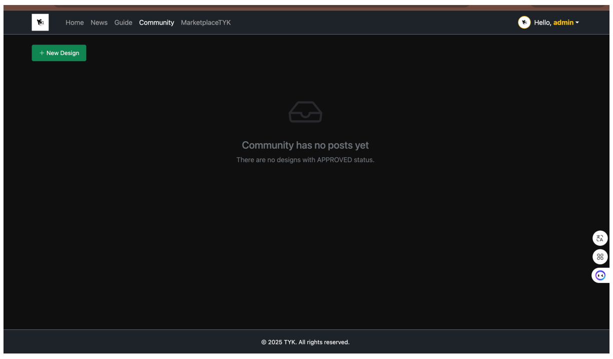
9 UI / Screenshots



Approve Posts (Pending)

IMAGE	TITLE / DESCRIPTION	CATEGORY	UPLOADER	ACTIONS
	ao giap hong 1 chiec ao giap màu hong	SKIN	anthanh	✓ Approve ✕ Reject

Figure 6: admin

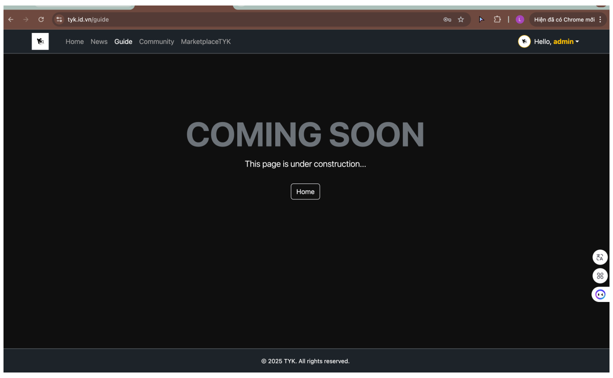


Community has no posts yet

There are no designs with APPROVED status.

© 2025 TYK. All rights reserved.

Figure 7: Community



COMING SOON

This page is under construction...

© 2025 TYK. All rights reserved.

Figure 8: Guide

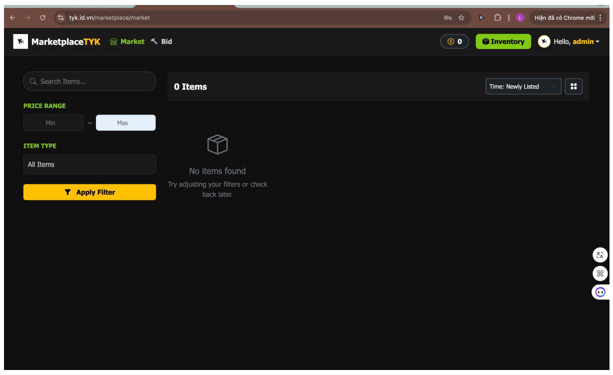


Figure 9: Market

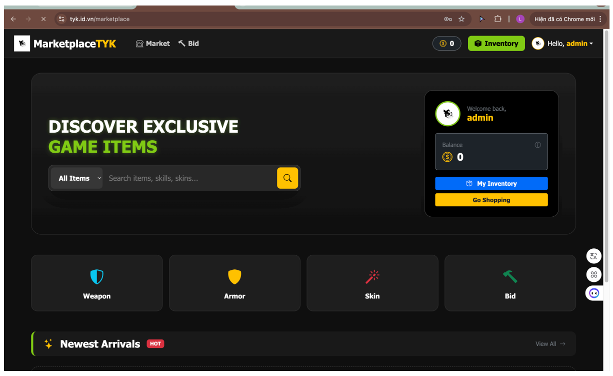


Figure 10: Marketplace

10 Deployment and Operations

10.1 Local Development

1. Clone the repository: `https://github.com/Nobodywinsbutme/tyk-webapp`
2. Open the project in an IDE (e.g., IntelliJ IDEA) or use the terminal.
3. Ensure MySQL is running locally (or start it using Docker). Update database credentials in `application.properties` or `application.yml` if needed.
4. Build and run the backend using Maven: `mvn clean install`
5. Start the Spring Boot application: `mvn spring-boot:run`
   Alternatively, run the main class directly from the IDE.
6. Access the application in a browser: `http://localhost:8080`

10.2 Production

- **Containerization:** Package the Spring Boot application as a Docker image and run it as a single service (the frontend is served by Thymeleaf and static resources within the same application).

- **Database:** Use a managed MySQL service (e.g., Amazon RDS or Google Cloud SQL) and configure credentials via environment variables or a secure secret manager.
- **CI/CD:** Configure a CI pipeline (e.g., GitHub Actions) to build the project with Maven, execute automated tests, build the Docker image, and push it to a container registry.
- **Reverse Proxy and TLS:** Deploy a reverse proxy (e.g., Nginx) in front of the application to terminate TLS (HTTPS), handle routing, and optionally enable gzip compression and caching for static assets.
- **Operations:** Enable structured logging, basic health checks, and resource monitoring to support troubleshooting and maintain stable operation in production.

11 Testing Plan

- **Unit Testing (Backend):** Use JUnit 5 and Spring Boot Test to validate core service logic, including authentication validation, inventory filtering, and marketplace transaction rules (sell, buy, cancel, update price).
- **Integration Testing:** Test REST endpoints with Spring MockMvc (or TestRestTemplate) and execute database-related tests against a MySQL test database (local instance or Docker) to verify persistence and query behavior.
- **Transaction Consistency Testing:** Verify that purchase operations are atomic by testing failure scenarios (e.g., insufficient coin balance, inactive listing) to ensure no partial updates occur.
- **UI and Workflow Testing (Manual):** Manually validate end-to-end user journeys on the web interface, including registration/login/logout, inventory browsing with filters and pagination, marketplace selling and buying, design upload and deletion, admin approval/rejection, and news management.
- **File Upload Testing:** Validate multipart upload behavior for community designs, including file storage, URL accessibility under `/uploads/**`, and correct rendering on public pages.

12 Security Considerations

- **Password Storage:** User passwords are stored as salted hashes using BCrypt via Spring Security's password encoder.
- **Transport Security:** HTTPS/TLS should be enforced in production to protect credentials and session cookies in transit.
- **Session Security:** Authentication is session-based; therefore, secure cookie settings should be enabled (e.g., `HttpOnly`, `Secure`, and appropriate `SameSite` policy) to reduce the risk of session hijacking.
- **Access Control:** Role-based authorization should restrict administrative endpoints (e.g., moderation actions and news management) to administrators only.

- **Transactional Safety:** Marketplace purchase operations should be executed within database transactions to prevent inconsistent states (e.g., coin deducted without receiving items) and to reduce race-condition risks during concurrent purchases.
- **Input Validation:** All client inputs (query parameters and request bodies) should be validated on the server side to prevent invalid data and reduce injection risks.
- **File Upload Security:** Uploaded images should be validated for type and size limits, stored outside the application classpath, and served only through controlled static mappings (e.g., `/uploads/**`).
- **Operational Protections:** Rate-limiting and logging for authentication endpoints are recommended to mitigate brute-force login attempts and to support incident investigation.

13 Project Plan and Timeline

The project was developed by a team of three members. The work was divided into three main tracks: (1) authentication and administration (Dev 1), (2) item and inventory system (Dev 2), and (3) marketplace trading logic (Dev 3). Table 2 summarizes the planned timeline.

Week	Planned Activities (Dev 1 / Dev 2 / Dev 3)
Week 1	**Dev 1:** Define requirements, set up repository structure, project dependencies, and basic UI layout. **Dev 2:** Draft item and inventory data model (ItemDefinition, InventoryItem). **Dev 3:** Draft marketplace data model (MarketListing, Transaction) and trading rules.
Week 2	**Dev 1:** Implement authentication (register/login/logout), user entity, and Spring Security configuration. **Dev 2:** Implement ItemDefinition and InventoryItem entities and repositories. **Dev 3:** Implement MarketListing entity and marketplace repository queries (listing filters).
Week 3	**Dev 1:** Implement admin-related foundations (role handling, navigation updates, basic admin pages). **Dev 2:** Implement inventory service and inventory REST API (filters, pagination, data mapping DTOs). **Dev 3:** Implement marketplace service core flows (sell, cancel, update price) with transactional handling.
Week 4	**Dev 1:** Implement community features (design submission pages and moderation views) and news module endpoints. **Dev 2:** Integrate inventory UI with Fetch API and connect inventory-to-sell flow. **Dev 3:** Implement purchase workflow (buy) including coin transfer and inventory updates; integrate marketplace UI.
Week 5	**Dev 1:** Finalize admin moderation (approve/reject designs) and admin news management (CRUD). **Dev 2:** Add validation, edge-case handling, and improve inventory UI (tabs: Owned / On Sale). **Dev 3:** Add robustness checks (self-buy prevention, inactive listing handling) and transaction consistency tests.
Week 6	**Dev 1:** Prepare deployment configuration (Docker build/run) and finalize documentation sections. **Dev 2:** Perform system-wide testing and UI polishing. **Dev 3:** Perform end-to-end testing of marketplace flows and assist with final report and screenshots.

Table 2: Proposed project timeline for a three-developer team.

14 Contributions

- **Dev 1 -Pham Trung Kien- (Team Lead) – Authentication and Administration:** Implemented user registration, login/logout, Spring Security configuration, and core user management. Developed the base Thymeleaf UI structure (navigation and shared layout), and implemented administrative features including design moderation and news management.
- **Dev 2 -Tran Thi Kieu My- Items and Inventory System:** Designed and implemented the item and inventory modules, including database entities and repositories for item definitions and user inventory. Developed inventory APIs with filtering and pagination, and integrated the inventory UI for browsing and managing items.
- **Dev 3 -Nguyen Ha An Thanh- Marketplace Trading Logic:** Designed and implemented marketplace functionality, including listing creation (sell), purchasing (buy), cancellation, and price updates. Ensured transactional consistency for coin transfers and inventory updates, and integrated marketplace APIs with the frontend

user flows.

15 References

- Project repo: https://github.com/Nobodywinsbutme/tyk-webapp
- Course materials and slides
- Library docs: Spring Boot