

Konibui Platform: Development Plan & Technical Epics

Introduction

This document provides the definitive development plan for the Konibui e-commerce platform. The platform's vision is to create a specialized online marketplace for the Pokémon Trading Card Game (TCG) community, blending the efficiency of a modern web application with the nostalgic charm of a Japanese convenience store, or "konbini".[1, 1]

The purpose of this report is to translate the project's strategic objectives, product requirements, and architectural mandates into a granular, actionable, and sequential development blueprint. It is structured as a series of Jira Epics, each containing a set of well-defined tickets. This work breakdown structure is designed to be consumed directly by the software development team, providing absolute clarity on tasks, acceptance criteria, and technical direction.

The development plan follows a logical progression, beginning with the foundational layers of the application—such as the local development environment and user authentication—before moving to core product features and advanced integrations. This methodical approach ensures a stable, coherent, and efficient development process, aligning with the technical vision of a robust, local-first, and server-rendered application.¹ Each ticket is crafted to be an incremental implementation, facilitating smaller, more manageable pull requests and a consistent development velocity.

Project Epics Overview

The following table provides a high-level roadmap of the entire project, outlining the sequence of development phases and their primary objectives. This summary serves as a quick reference for the project's structure and flow.

Epic ID	Epic Title	Primary Goal
EPIC-1	Project Foundation & Development Environment	To establish a fully functional, containerized local development environment and a clean Laravel 12 application scaffold, enabling any developer to start coding with a single command.

EPIC-2	User Authentication & Role-Based Access Control (RBAC)	To implement a secure system for user registration, login, and session management, and to establish the foundational three-tier RBAC system at both application and database levels.
EPIC-3	TCG Product & Inventory Management System	To build the core data models and administrative interface for managing the complex TCG product catalog, including cards, sets, rarities, condition-based variants, and their associated inventory.
EPIC-4	Product Discovery & Catalog Browsing	To create the public-facing user interface for customers to browse and filter the product catalog, implementing the "Konbini" design theme and ensuring a highly performant experience.
EPIC-5	Advanced Search & Discovery	To implement a high-performance, full-text search engine for the product catalog using Laravel Scout and the TNTSearch driver, fulfilling the local-first search requirement.
EPIC-6	Core Commerce Flow: Shopping Cart & Checkout	To implement the full shopping cart and checkout process, from adding items to the cart to successfully creating an order with Cash on Delivery payment.
EPIC-7	Advanced Database Procedures and Triggers	To implement comprehensive stored procedures and triggers that enhance the e-commerce functionality, improve data integrity, provide audit trails, and automate critical business processes for the Pokemon card trading platform.
EPIC-8	Post-Order User Experience: Order Management	To provide customers with the ability to view their order history and track order status, and to equip staff with the necessary tools to manage and process orders.

EPIC-9	Card Buyback System	To develop the end-to-end feature for users to submit cards for buyback and for staff to review, evaluate, and process these submissions.
EPIC-10	Platform Extensibility: MCP Integration	To transform the application into a programmable platform by exposing core business logic as tools and resources via the Model Context Protocol (MCP) for future AI-driven workflows.

EPIC-1: Project Foundation & Development Environment

Goal

To establish a fully functional, containerized local development environment and a clean Laravel 12 application scaffold, enabling any developer to start coding with a single command, fulfilling the "local-first" mandate.

Context and Rationale

This epic serves as the absolute cornerstone of the Konibui project. A core requirement, explicitly mandated in both the technical and product documentation, is the "local-first" development philosophy.[1, 1] This approach prioritizes a development environment that is consistent, portable, and easy to set up. By containerizing all services using Docker, the project eliminates the classic "it works on my machine" problem, ensuring that every developer and every automated system operates within an identical, version-controlled stack.

The successful completion of this epic is a prerequisite for all future development. It directly addresses the success metric of enabling a new developer to have a complete, running environment in under five minutes.¹ This streamlined onboarding is not only a matter of efficiency but is also essential for the agentic coding system envisioned in the technical spike, which requires a deterministic, command-driven setup process.¹ This epic internalizes all external service dependencies, a key architectural goal that provides stability and control throughout the project lifecycle.

Tickets

TICKET 1.1: Implement Docker Compose Environment

- **Background:** The entire local environment must be orchestrated by a single `docker-compose.yml` file, as specified in the technical architecture.¹ This file will define

and network the four core services required for the application to run: **app** (the PHP-FPM container for Laravel), **db** (the MySQL database), **webserver** (the Nginx public-facing server), and **db-admin** (a phpMyAdmin instance for development). This multi-container approach ensures a clean separation of concerns and mirrors a production-like setup.

- **Acceptance Criteria:**

- A **docker-compose.yml** file is present in the project's root directory.
- Executing the **docker-compose up -d --build** command successfully builds any custom images and starts all four services without any errors.
- A custom bridge network is defined in the **docker-compose.yml** file, and all four services are attached to it.
- Services can communicate with each other using their defined service names as hostnames (e.g., the **app** container can successfully resolve and connect to the **db** service).
- The **webserver** (Nginx) service correctly maps a host port (e.g., 8080) to its internal container port 80.
- The **db-admin** (phpMyAdmin) service correctly maps a separate host port (e.g., 8081) to its internal container port 80, avoiding port conflicts.

- **Technical Suggestions:**

- Use the latest stable official images from Docker Hub for Nginx, MySQL 8.x, and phpMyAdmin as base images.
- Define the custom bridge network at the top level of the **docker-compose.yml** file and reference it in each service definition.

TICKET 1.2: Configure Application (app) and Database (db) Services

- **Background:** This ticket focuses on the detailed configuration of the two most critical services: the application and the database. The **app** service requires a custom **Dockerfile** to extend the official PHP image with necessary extensions like **pdo_mysql** for database connectivity and **gd** for image manipulation, which is required for the buyback system.¹

The

db service must be configured with a persistent named volume to safeguard database data across container restarts, a critical requirement for development consistency.[1, 1]

- **Acceptance Criteria:**

- A **Dockerfile** for the **app** service is created in the project, using an official PHP 8.2+ FPM image as its base.
- The **Dockerfile** successfully installs Composer and the required PHP extensions: **pdo_mysql** and **gd**.
- The **app** service definition in **docker-compose.yml** mounts the local project directory (.) to the container's **/var/www/html** directory, enabling live code changes.
- The **db** service definition uses a named volume (e.g., **konibui_db_data**) mounted to the container's **/var/lib/mysql** directory.

- Data within the database persists after stopping and removing the containers (`docker-compose down`) and subsequently restarting them (`docker-compose up -d`).
- Database credentials (e.g., `MYSQL_DATABASE`, `MYSQL_USER`, `MYSQL_PASSWORD`) are passed to the `db` and `app` services via environment variables sourced from the project's `.env` file.
- **Technical Suggestions:**
 - Consider using a multi-stage build in the `Dockerfile` for the `app` service to keep the final production image lean.
 - Explicitly declare the named volume in the top-level `volumes` key in `docker-compose.yml` for clarity and easier management.

TICKET 1.3: Implement Nginx Configuration and Service Healthchecks

- **Background:** The `webserver` (Nginx) container serves as the public entry point for all HTTP requests and must be configured to properly route traffic. A custom Nginx configuration is required to proxy requests for PHP files to the `app` service.¹ To prevent race conditions and failed database connections during initial startup, a healthcheck mechanism must be implemented. The `app` service must wait until the `db` service is fully initialized and ready to accept connections before it attempts to start.
- **Acceptance Criteria:**
 - A custom `nginx.conf` file is created and mounted as a volume into the `webserver` container.
 - The Nginx configuration correctly sets the `root` directive to `/var/www/html/public`, the entry point for a Laravel application.
 - The configuration includes a `location ~ \.php$` block with a `fastcgi_pass` directive that correctly forwards requests to the `app` service on its internal port 9000 (`app:9000`).
 - The `db` service definition in `docker-compose.yml` includes a `healthcheck` block that periodically checks if the MySQL server is responsive.
 - The `app` service definition includes a `depends_on` directive that explicitly waits for the `db` service to report a `service_healthy` status before starting.
- **Technical Suggestions:**
 - The MySQL healthcheck command can be a simple `mysqladmin ping -h localhost --user=... --password=...`
 - Use the long-form syntax for `depends_on` to specify the health condition: `depends_on: { db: { condition: service_healthy } }`.

TICKET 1.4: Scaffold Laravel 12 Application and Finalize Workflow

- **Background:** With the Docker environment fully configured and operational, the final step is to install a fresh Laravel 12 application and validate that all services are communicating correctly. This involves using Composer to install dependencies, running essential Artisan commands to initialize the application, and confirming that the

application and database administration tool are accessible from the host machine's browser.¹

- **Acceptance Criteria:**

- Executing `docker-compose exec app composer install` inside the project directory successfully installs all PHP dependencies defined in `composer.json`.
- Executing `docker-compose exec app php artisan key:generate` successfully generates a unique application key and saves it to the `.env` file.
- Executing `docker-compose exec app php artisan migrate` successfully connects to the `db` container and runs the default Laravel database migrations without errors.
- The default Laravel welcome page is accessible in a web browser at the host port mapped to the `webserver` service (e.g., `http://localhost:8080`).
- The phpMyAdmin interface is accessible at its configured host port (e.g., `http://localhost:8081`) and can successfully log in to the `db` service using the credentials from the `.env` file.

- **Technical Suggestions:**

- Create a `.env.example` file in the repository root that documents all required environment variables.
- Add a "Getting Started" section to the project's `README.md` file that details the simple, repeatable setup commands (`cp .env.example .env`, `docker-compose up -d --build`, `docker-compose exec app composer install`, etc.).

EPIC-2: User Authentication & Role-Based Access Control (RBAC)

Goal

To implement a secure system for user registration, login, and session management, and to establish the foundational three-tier RBAC system (Customer, Employee, Admin) at both the application and database levels.

Context and Rationale

A robust authentication and authorization system is a non-negotiable foundation for any e-commerce platform, and it is a core functional requirement for Konibui.[1, 1] This epic establishes the mechanisms for managing user identity and enforcing the principle of least privilege across the application. The technical and product requirements are explicit in defining the three primary user roles—Customer, Employee, and Admin—and their distinct capabilities.[1, 1]

Crucially, this epic implements the "defense-in-depth" security strategy mandated by the technical architecture.¹ This involves creating two distinct layers of access control: one at the

application layer using Laravel's native authorization features, and a second, hardened layer directly at the database level using MySQL's role management system. This dual-layer approach significantly strengthens the platform's security posture, ensuring that data access is controlled even in the event of an application-level vulnerability. The implementation sequence is designed to build from the user-facing authentication features inward to the deep infrastructure of database security.

Tickets

TICKET 2.1: Implement Basic User Authentication with Laravel Breeze

- **Background:** The technical specification mandates the use of Laravel Breeze, configured for the Livewire stack, to provide the authentication scaffolding.¹ Breeze offers a minimal, elegant, and secure implementation of all essential authentication features, including registration, login, password reset, and email verification. This server-rendered approach aligns perfectly with the project's architectural choice of Livewire and avoids the complexity of heavier starter kits.
- **Acceptance Criteria:**
 - The Laravel Breeze package is installed and its scaffolding is published using the `--livewire` flag.
 - Public-facing routes for `/register`, `/login`, `/forgot-password`, and `/reset-password` are functional.
 - A new user can successfully register for an account using a form that collects their name, email, and password.
 - Upon registration, the system sends an email verification link. The user's account is inactive until this link is clicked.
 - A registered user with a verified email can successfully log in with their credentials and can also log out.
 - An authenticated user can use the "forgot password" flow to receive a password reset link and successfully update their password.
 - All authentication-related forms are protected against Cross-Site Request Forgery (CSRF).
- **Technical Suggestions:**
 - For local development, configure the `.env` file with `MAIL_MAILER=log`. This will write all outgoing emails, including verification and password reset links, to the `storage/logs/laravel.log` file for easy access and testing without a real SMTP server.

TICKET 2.2: Create Database Schema and Models for RBAC

- **Background:** To support the three distinct user roles, the database schema must be extended beyond the default `users` table. This requires a `roles` table to define the roles themselves ('Admin', 'Employee', 'Customer') and a `role_user` pivot table to establish the many-to-many relationship between users and roles. This structure is explicitly defined in the database schema blueprint.¹

- **Acceptance Criteria:**
 - A new database migration is created for a `roles` table, containing at least an auto-incrementing `id` and a `name` column with a `UNIQUE` constraint.
 - A new database migration is created for a `role_user` pivot table, containing `user_id` and `role_id` columns. These columns should form a composite primary key and have foreign key constraints referencing the `users` and `roles` tables, respectively.
 - The `User` and a new `Role` Eloquent model are created. The `User` model defines a `roles()` method with a `belongsToMany` relationship to the `Role` model.
 - A database seeder is created to populate the `roles` table with the three required roles: 'Admin', 'Employee', and 'Customer'.
 - A helper method, such as `hasRole(string $roleName): bool`, is added to the `User` model to provide a convenient way to check a user's permissions.
- **Technical Suggestions:**
 - Utilize anonymous migrations, a feature available since Laravel 8, to prevent potential class name collisions with future packages.¹
 - Strictly adhere to Laravel's naming conventions for models and pivot tables (e.g., `role_user` for the pivot table between `Role` and `User` models) to allow the framework to automatically resolve relationships.

TICKET 2.3: Implement Application-Layer Authorization with Laravel Policies

- **Background:** The technical architecture specifies Laravel Policies as the preferred method for managing authorization logic, as they provide a more organized and scalable structure for model-based permissions compared to closure-based Gates.¹ This ticket involves creating the foundational Policy classes and a custom middleware to protect application routes and actions based on the user's assigned role.
- **Acceptance Criteria:**
 - A custom `CheckRole` middleware is created and registered in the application's HTTP kernel.
 - The middleware can restrict access to routes or route groups based on one or more roles (e.g., a route protected by `middleware('role:Admin,Employee')` should be accessible to both Admins and Employees, but not Customers).
 - Policy classes are generated for the primary data models that will be created later (e.g., `ProductPolicy`, `OrderPolicy`).
 - The methods within these generated policies (e.g., `create`, `update`, `delete`) are stubbed out with the correct authorization logic, which checks the user's role via the `hasRole()` method.
 - The application's `AuthServiceProvider` is correctly configured to automatically discover and register these model policies.
- **Technical Suggestions:**
 - Generate policies efficiently using the Artisan command: `php artisan make:policy ProductPolicy --model=Product`.

- The `CheckRole` middleware's `handle` method should be designed to accept a variable number of arguments for the role names, providing flexibility.

TICKET 2.4: Implement Database-Layer Security with MySQL Roles

- **Background:** This ticket implements the critical "defense-in-depth" security strategy by creating access controls directly at the database level.¹ This involves creating MySQL roles that mirror the application's roles and granting them the minimum set of privileges required to perform their functions. This provides a crucial second layer of security that protects the data even if the application layer were to be compromised.
- **Acceptance Criteria:**
 - A new Laravel migration is created that uses raw SQL statements to `CREATE ROLE 'konibui_admin', 'konibui_employee', 'konibui_customer';`.
 - The same migration grants the appropriate `SELECT`, `INSERT`, `UPDATE`, and `DELETE` privileges to each of these roles on the relevant application tables, strictly following the privilege matrix defined in the technical spike document.¹
 - The migration's `down()` method correctly drops the created roles.
 - Documentation is added to the project's `README.md` explaining how to create corresponding database users for each role and how to configure the `.env` file to connect to the database as one of these restricted users for testing purposes.
- **Technical Suggestions:**
 - Place the raw SQL for role creation and privilege granting within `DB::unprepared()` calls inside the migration's `up()` and `down()` methods. This is the standard Laravel approach for executing schema-modifying DDL statements that are not supported by the Schema Builder.

EPIC-3: TCG Product & Inventory Management System

Goal

To build the core data models and administrative interface for managing the complex TCG product catalog, including cards, sets, rarities, condition-based variants, and their associated inventory.

Context and Rationale

The core value proposition of the Konibui platform is its specialized focus on Pokémon TCG products.¹ This requires a more sophisticated data model than a typical e-commerce store. A single Pokémon card, such as a "Base Set Charizard," is not a single product but a template for multiple sellable items, each with a unique physical condition and corresponding price. This epic is dedicated to implementing the nuanced data model detailed in both the technical spike and the product requirements.[1, 1]

This model intelligently separates the immutable, definitional data of a card (its name, set, etc.) from the sellable, variable product data (its condition, price, and stock). Getting this data structure right is paramount, as it forms the very heart of the application's business logic. The administrative panels built in this epic will provide the tools necessary for staff to manage this complex catalog, making it the foundational work for all subsequent customer-facing features like browsing, searching, and purchasing.

Tickets

TICKET 3.1: Create Migrations and Models for Core TCG Data

- **Background:** This ticket establishes the database schema for the foundational, immutable data that defines a Pokémon card. This involves creating a `cards` table to store the core information about each unique card (e.g., 'Pikachu', from the 'Base Set'). It also includes the creation of related lookup tables for `sets`, `rarities`, and `categories`, as specified in the database schema design.¹ This normalized structure prevents data duplication and ensures data integrity.
- **Acceptance Criteria:**
 - Database migrations and corresponding Eloquent models are created for `cards`, `sets`, `rarities`, and `categories`.
 - The `cards` table schema includes columns for `name`, `collector_number`, and foreign keys like `set_id` and `rarity_id`.
 - The `sets`, `rarities`, and `categories` tables contain at least `id` and `name` columns.
 - Eloquent relationships (`belongsToMany`) are correctly defined on all models to reflect their relational structure (e.g., a `Card` `belongsToMany` a `Set`).
 - Database seeders are created to populate the `categories` ('Single Card', 'Booster Pack', etc.), `rarities`, and a sample of `sets` to facilitate development and testing.
- **Technical Suggestions:**
 - Define foreign key constraints directly within the migration files using `$table->foreignId('set_id')->constrained();`. This enforces relational integrity at the database level, which is more robust than relying solely on application-level logic.

TICKET 3.2: Implement Product Variant and Inventory Schema

- **Background:** Building upon the base card data, this ticket implements the schema for the actual sellable items. A `products` table will be created to represent a specific variant of a card, linking a record from the `cards` table to a specific `condition` and `price`. A separate `inventory` table will track the stock level for each unique product variant. This two-tiered structure is the key to managing the catalog correctly.[1, 1]
- **Acceptance Criteria:**
 - A database migration and an Eloquent `Product` model are created.
 - The `products` table includes a foreign key to the `cards` table, a `condition` column, a `price` column (DECIMAL type), and a `sku` column with a `UNIQUE` constraint.
 - The `condition` column is defined as an `ENUM` in the database with the allowed values: 'NM', 'LP', 'MP', 'HP', 'DMG'.

- A database migration and an Eloquent **Inventory** model are created. The **inventory** table has a one-to-one relationship with the **products** table (using **product_id** as both primary and foreign key) and contains a **stock** column (UNSIGNED INTEGER).
- The **Product** model correctly defines its relationships to the **Card** and **Inventory** models.
- **Technical Suggestions:**
 - The **condition** column is an ideal candidate for a backed PHP 8.1+ Enum. Create an **App\Enums\ProductCondition:string** enum and cast the **condition** attribute to it in the **Product** model. This provides type-safe, readable code and avoids the use of "magic strings".[1, 1]

TICKET 3.3: Build Admin Panel for Managing Core TCG Data (Cards, Sets, Rarities)

- **Background:** To make the system usable, administrators require a user interface to populate and manage the foundational TCG data. This ticket involves creating a set of protected Livewire components to provide full CRUD (Create, Read, Update, Delete) functionality for the **sets**, **rarities**, and **cards** tables. These panels will be accessible only to users with the 'Admin' role.
- **Acceptance Criteria:**
 - A new route group under the **/admin** prefix is created for managing TCG data (e.g., **/admin/sets**, **/admin/rarities**, **/admin/cards**).
 - The entire route group is protected by middleware that requires both authentication and the 'Admin' role (**middleware(['auth', 'role:Admin'])**).
 - A full-page Livewire component exists for managing **sets**, allowing an admin to list, create, edit, and delete set records.
 - A similar full-page Livewire component exists for managing **rarities**.
 - A full-page Livewire component exists for managing **cards**.
 - All data manipulation actions within these components are authorized using corresponding model policies (e.g., **SetPolicy**, **CardPolicy**).
- **Technical Suggestions:**
 - Leverage pre-built components from TailwindUI, such as tables, modals, and forms, to accelerate the development of a polished and professional-looking admin panel, as recommended in the technical documentation.¹

TICKET 3.4: Build Admin Panel for Managing Product Variants and Inventory

- **Background:** This is the most crucial administrative interface, where staff will perform the daily task of creating and managing the actual sellable products. The UI must allow an admin to select a base **card**, and then create one or more **product** variants for it by assigning a condition, price, and initial stock quantity.
- **Acceptance Criteria:**
 - An admin-only route (e.g., **/admin/products**) and a corresponding Livewire management component are created.

- The interface provides a way to list all existing **products**, showing their linked card name, condition, price, and current stock.
 - The creation form allows an admin to first select a base **card** from the **cards** table.
 - After selecting a card, the admin can create a new **product** variant by choosing a **condition** from a dropdown, setting a **price**, and defining the initial **stock** level.
 - The interface allows an admin to edit the price and update the stock level of any existing product variant.
 - All actions are protected by the **ProductPolicy** and restricted to 'Admin' role users.
 - **Technical Suggestions:**
 - For the card selection UI, a searchable dropdown is essential for usability, especially as the **cards** table grows. A component library or a custom Livewire/Alpine.js component can be used for this.
 - Consider using a modal or a slide-over panel (available in TailwindUI) for the product creation/editing form. This allows the admin to perform the action without losing the context of the main product list.
-

EPIC-4: Product Discovery & Catalog Browsing

Goal

To create the public-facing user interface for customers to browse and filter the product catalog, implementing the "Konbini" design theme and ensuring a highly performant experience.

Context and Rationale

Once the product catalog is managed by administrators, customers need an intuitive and engaging way to discover and browse these products. This epic is dedicated to building the primary user-facing experience of the Konibui store. It translates the specific UX/UI vision from the product brainstorming—the bright, clean, "Konbini" aesthetic—into a tangible, interactive interface.¹

A key focus of this epic is performance. A slow or clunky browsing experience is a primary driver of user abandonment and lost sales. Therefore, the implementation will strictly adhere to the

Livewire performance best practices outlined in the technical spike.¹ This includes minimizing data payloads and reducing unnecessary server requests to ensure the application feels fast and responsive, directly contributing to the conversion rate and page load time success

metrics.¹ The development process will layer the implementation, starting with the foundational application flow and layout, then building the core product discovery features, and finishing with the final visual polish.

Tickets

TICKET 4.1: Implement Core Application Layout, Navigation, and User Flow

- **Background:** This foundational ticket establishes the primary user journey and the visual skeleton of the Konibui application. It defines the overall page structure, the persistent navigation bar, and the distinct user flows for unauthenticated guests, logged-in customers, and staff members. A consistent and intuitive navigation system is critical for user experience, and this ticket ensures that the application's structure adapts intelligently based on the user's role and permissions, as defined in EPIC-2.
- **Acceptance Criteria:**
 - A main application layout (e.g., `layouts/app.blade.php`) is created, defining the header, footer, and main content area for all subsequent pages.
 - A persistent, responsive navigation bar component is created and integrated into the main layout.
 - The navigation bar's content dynamically changes based on the user's authentication status and role:
 - **Guests:** See links to "Login," "Register," and main product categories.
 - **Customers:** See product categories, a "My Account" link/dropdown, and the "Trainer's Bag" cart icon. "Login" and "Register" links are hidden.
 - **Admins/Employees:** See all customer-facing links, plus dedicated navigation links to their respective management dashboards (e.g., "Product Management," "Order Processing").
 - A visually engaging landing page is created at the root URL (`/`). This page features a large hero section or slideshow with high-quality images of featured products to attract users.¹
 - The existing login and registration pages are restyled to match the "Konbini" theme for a consistent look and feel.¹
 - After a successful login, all user roles are redirected to the main store page (`/products`).
 - The navigation bar for customers includes the "Trainer's Bag" cart icon, which will later be used to trigger the cart display.¹
- **Technical Suggestions:**
 - Use Blade's layout inheritance (`@extends`, `@section`) for a consistent page structure.
 - The navigation bar should be a dedicated Blade component. Use `@auth`, `@guest`, and the `User::hasRole()` helper method within the component to conditionally render links.
 - The landing page slideshow can be implemented using a simple Alpine.js component or a lightweight library like Swiper.js to avoid adding heavy dependencies.
 - The post-login redirect behavior can be customized in Laravel's `RouteServiceProvider` or within the authentication controllers.

TICKET 4.2: Implement Main Product Listing Page with Livewire

- **Background:** This ticket involves creating the main shop page, which will display all available products to customers. This page will be powered by a full-page Livewire component responsible for fetching the product data, displaying it in a grid format, and handling pagination for large result sets.
- **Acceptance Criteria:**
 - A public route, such as `/products`, is created and renders a full-page Livewire component (e.g., `ProductListingPage`).
 - The component successfully fetches and displays a paginated grid of `products`.
 - Each item displayed in the grid is an instance of a separate, reusable `ProductCard` component.
 - The main product collection is loaded via a Livewire computed property (`getProductsProperty`). This is a critical performance requirement to avoid sending the entire collection to the client with every request.¹
 - Standard pagination controls are displayed at the bottom of the list and are fully functional, allowing users to navigate between pages of results.
- **Technical Suggestions:**
 - Use Tailwind CSS's grid or flexbox utilities to create a responsive product grid.
 - The use of a computed property for the main data collection is the single most important performance optimization for a data-heavy Livewire component. Ensure this pattern is followed strictly.

TICKET 4.3: Design and Build the Reusable `ProductCard` Component

- **Background:** The `ProductCard` is a key visual element described in detail in the initial product vision.¹ It must act as a clean, information-dense summary of a single product, consistent with the overall "Konbini" theme. It needs to present all essential data points in a clear and visually appealing manner.
- **Acceptance Criteria:**
 - A reusable Blade/Livewire component named `ProductCard` is created.
 - The card prominently displays the product's image, name, and the name of its TCG set.
 - The card displays the custom rarity icon and the rarity name (e.g., 'Secret Rare').
 - The card displays the product's price.
 - The card displays a clear stock status: "In Stock", "Out of Stock", or a "Low Stock" warning when the quantity is less than 5, as required by the PRD.¹
 - A functional "Add to Cart" button is present on the card.
- **Technical Suggestions:**
 - The component should accept a single `$product` Eloquent model as a prop.
 - When the "Add to Cart" button is clicked, it should dispatch a global Livewire event, such as `$this->dispatch('add-to-cart', ['productId' => $this->product->id])`. This decouples the card from the shopping cart component.

- The "Add to Cart" button could briefly change its text to "Caught!" upon a successful click, adding a delightful, on-theme micro-interaction as suggested in the brainstorming document.¹

TICKET 4.4: Implement Filtering by Category, Set, and Rarity

- **Background:** A large catalog is useless without powerful filtering tools. This ticket involves adding a filter sidebar or panel to the `ProductListingPage` component, allowing users to narrow down the product list by core TCG attributes like the product's category, set, and rarity, as specified in the functional requirements.¹
- **Acceptance Criteria:**
 - The `ProductListingPage` Livewire component is updated with public properties to hold the state of the selected filters (e.g., `public array $selectedSets =;`, `public?int $selectedRarityId = null;`).
 - The view is updated with UI elements (e.g., checkboxes, dropdowns) for filtering by `Category`, `Set`, and `Rarity`.
 - Any change to a filter's value triggers a re-render of the product list, applying the new filter criteria to the database query.
 - The filtering logic correctly utilizes Eloquent's relational query methods, such as `whereIn` for multiple selections and `whereHas` for filtering based on relationships.
 - All form input elements used for filtering must use `wire:model.defer` to prevent a network request on every single keystroke or click, batching the update instead.¹
- **Technical Suggestions:**
 - The data used to populate the filters (e.g., the list of all available sets and rarities) should also be loaded via computed properties to keep the component's serialized state small and performant.
 - The patterns found in the `spatie/laravel-query-builder` package provide an excellent reference for how to dynamically build an Eloquent query from an array of filter parameters.¹

TICKET 4.5: Implement Sorting and Product Detail Page

- **Background:** In addition to filtering, standard e-commerce user experience dictates that users should be able to sort results. Furthermore, while the `ProductCard` provides a summary, a dedicated product detail page is required for users to view more comprehensive information and larger images before making a purchase decision.
- **Acceptance Criteria:**
 - A sort dropdown is added to the `ProductListingPage` UI.
 - The dropdown provides options to sort the product list by Price (ascending and descending), Name (A-Z), and Set.
 - Selecting a sort option updates the component's state and re-queries the product list with the appropriate `orderBy` clause.

- Clicking on any **ProductCard** navigates the user to a unique, SEO-friendly URL for that specific product (e.g., `/products/{sku}`).
- The product detail page successfully loads and displays all information for the selected product, including a larger image, a full description, and details about other available conditions for the same base card.
- **Technical Suggestions:**
 - The product detail page can be implemented as either a standard server-rendered Blade view using a controller or as another full-page Livewire component, depending on the desired level of interactivity.
 - Use Laravel's route model binding to automatically resolve the **Product** model from the SKU or ID in the URL.

TICKET 4.6: Implement "Konbini" Theming and Responsive Design

- **Background:** This ticket focuses on applying the specific visual identity that differentiates Konibui from generic stores. It involves implementing the color palette, typography, and custom iconography described in the product vision to create the unique "Modern Konbini" user experience.¹ This includes ensuring the design is fully responsive across all target devices.
- **Acceptance Criteria:**
 - The application's layout uses a predominantly white or very light-gray background, ensuring product art is the main focus.
 - Primary action colors (for buttons, links, highlights) are configured to use the iconic Pokémon Red, Black, White, and Grey.
 - Secondary accent colors (Konbini green and orange) are used for non-critical elements like info boxes or sale tags.
 - The specified fonts ("Nunito" for headings, "Inter" for body text) are loaded and applied correctly.
 - The default shopping cart icon is replaced with a "Trainer's Bag" icon.
 - All loading indicators (e.g., for Livewire updates) are replaced with a spinning Poké Ball animation.
 - The entire product browsing interface is fully responsive and usable on mobile (320px+), tablet (768px+), and desktop (1024px+) screen sizes, as defined in the PRD.¹
- **Technical Suggestions:**
 - Define the custom color palette and font families within the `tailwind.config.js` file. This makes the theme values available as standard Tailwind utility classes (e.g., `bg-brand-white`, `text-pokemon-red`, `font-display`).
 - Use Alpine.js for purely client-side UI interactions, such as toggling the visibility of a mobile navigation menu or a filter panel. This avoids unnecessary server round-trips for actions that don't require server state, a key performance strategy.¹

EPIC-5: Advanced Search & Discovery

Goal

To implement a high-performance, full-text search engine for the product catalog using Laravel Scout and the TNTSearch driver, fulfilling the local-first search requirement.

Context and Rationale

For an e-commerce store with a potentially massive catalog of individual cards, a simple database **LIKE** query is insufficient for providing a satisfactory user experience. Customers expect fast, relevant, and typo-tolerant search functionality. This epic addresses the requirement for a dedicated full-text search engine.¹

In alignment with the project's "local-first" architectural principle, the chosen technology is Laravel Scout with the TNTSearch driver.¹ Unlike cloud-based services like Algolia, TNTSearch runs entirely within the PHP ecosystem and uses a local, file-based index. This eliminates external dependencies and costs, simplifies the development setup, and perfectly matches the project's self-contained architecture. This epic will build the search functionality from the ground up, from indexing the data to creating the user-facing search interface.

Tickets

TICKET 5.1: Install and Configure Laravel Scout with TNTSearch

- **Background:** The first step is to integrate the necessary software packages into the Laravel application. This involves installing both the Laravel Scout core package and the specific TNTSearch driver via Composer, and then configuring the application to use them.
- **Acceptance Criteria:**
 - The `laravel/scout` and `teamtnt/laravel-scout-tntsearch-driver` packages are successfully installed via Composer.
 - The Scout service provider is registered in `config/app.php`.
 - The Scout configuration file is published (`scout.php`), and the default driver is set to `tntsearch`.
 - The `.env` file is updated with `SCOUT_DRIVER=tntsearch`.
 - The TNTSearch configuration specifies a storage path within the `storage` directory for the search index files.
- **Technical Suggestions:**
 - Ensure the directory specified for the TNTSearch index (`storage/tnt-indexes`) is writable by the web server user within the Docker container. Add this directory to the `.gitignore` file.

TICKET 5.2: Configure and Index Searchable Models

- **Background:** Once Scout is installed, the application needs to be told which data should be searchable. This is done by adding the `Searchable` trait to the relevant Eloquent models and customizing the data that gets sent to the search index. For Konibui, both the base `Card` information and the `Product` variant information are relevant to search.
- **Acceptance Criteria:**
 - The `Laravel\Scout\Searchable` trait is added to the `App\Models\Card` and `App\Models\Product` Eloquent models.
 - The `toSearchableArray()` method is implemented on both models to define the indexed data.
 - The searchable array for a `Product` includes its own data (condition, price) and related `Card` data (name, set name, rarity name).
 - Running the command `docker-compose exec app php artisan scout:import "App\Models\Product"` successfully creates a search index file and populates it with all existing products.
 - Creating, updating, or deleting a `Product` record in the application automatically and correctly updates the search index in real-time.
- **Technical Suggestions:**
 - In the `Product` model's `toSearchableArray()`, eager load the card relationship to avoid N+1 query problems during indexing. The array should be a flattened structure of all text-based attributes a user might search for.

TICKET 5.3: Build Livewire Search Component

- **Background:** With the data indexed, a user interface is needed to perform searches. This ticket involves creating a Livewire component that provides a search input field and displays the results returned from Scout. This component will be a central part of the site's header or navigation.
- **Acceptance Criteria:**
 - A new Livewire component, `ProductSearch`, is created.
 - The component contains a public property (e.g., `public string $query = ''`).
 - An input field in the component's view is bound to the `$query` property using `wire:model`.
 - As the user types, the component executes a search using `Product::search($this->query)->get()`.
 - The search results are displayed to the user in a dropdown or a dedicated results list.
 - If the query is empty, no results are shown.
- **Technical Suggestions:**
 - To improve user experience, use `wire:model.debounce.500ms` on the search input. This will wait for the user to stop typing for half a second before sending the search request, preventing a flood of requests on every keystroke.
 - The search results display should be handled with Alpine.js for showing/hiding the dropdown to provide instant UI feedback without a server round-trip.

TICKET 5.4: Implement Fuzzy Search and Combine with Filters

- **Background:** A key benefit of TNTSearch is its support for fuzzy searching, which can handle typos and minor variations in search terms. This ticket focuses on enabling this feature and integrating the search functionality with the existing filtering logic from the product listing page.
 - **Acceptance Criteria:**
 - The TNTSearch configuration is updated to enable fuzzy search ('fuzzy' => true).
 - Searching for a term with a minor typo (e.g., "Charzard" instead of "Charizard") still returns the correct results.
 - On the main product listing page, the search input and the filter controls work in tandem.
 - When a user performs a search, the results are a list of product IDs from Scout. This list of IDs is then used to constrain the main Eloquent query, to which the other filters (set, rarity, etc.) are then applied.
 - The final displayed list of products respects both the full-text search query and the selected filters.
 - **Technical Suggestions:**
 - The logic for combining search and filters should be: 1) Get an array of IDs from `Product::search($query)->keys()`. 2) If the array is not empty, add a `whereIn('id', $keys)` clause to the main Eloquent query builder instance. 3) Apply the rest of the filters to the query builder as before. This ensures the speed of full-text search is combined with the precision of relational filtering.
-

EPIC-6: Core Commerce Flow: Shopping Cart & Checkout

Goal

To implement the full shopping cart and checkout process, from adding items to the cart to successfully creating an order with Cash on Delivery payment.

Context and Rationale

This epic represents the commercial heart of the application. It builds the critical path that turns a browsing user into a paying customer. The implementation must be seamless, intuitive, and robust to maximize conversion rates and build user trust. The requirements specify a dual-mode shopping cart: database-backed for logged-in users to ensure persistence across devices, and session-based for guests for a low-friction initial experience.¹

The checkout process will be implemented as a single, stateful Livewire component, creating a smooth, single-page-application-like experience without full page reloads, which can be jarring for users.[1, 1] The initial payment method will be Cash on Delivery (COD), but the architecture must be extensible to accommodate future payment gateways like PayPal. This is achieved by encapsulating logic within a dedicated service layer, a core principle of the project's backend architecture.¹

Tickets

TICKET 6.1: Implement Database-Backed Shopping Cart Service

- **Background:** To handle cart logic in a centralized and reusable way, a dedicated `CartService` class will be created. This service will manage all cart operations (add, update, remove, clear) and handle the logic for both database-backed carts for authenticated users and session-based carts for guests, as required.¹
- **Acceptance Criteria:**
 - A `CartService` class is created in the `app/Services` directory.
 - The service can be resolved from Laravel's service container.
 - The service contains methods like `add(Product $product, int $quantity)`, `update(int $productId, int $quantity)`, `remove(int $productId)`, `getCartItems()`, and `clear()`.
 - When a user is authenticated, the service stores and retrieves cart data from a dedicated `cart_items` database table.
 - When a user is a guest, the service stores and retrieves cart data from the user's session.
 - When a guest with items in their session-based cart logs in, the cart items are automatically migrated to the database.
- **Technical Suggestions:**
 - For the database-backed cart, a simple table with `user_id`, `product_id`, and `quantity` is sufficient.
 - The service can use `auth()->check()` to determine whether to use the session or the database for storage. A popular and robust package like `darryldecode/cart` can be used as a reference for a feature-rich implementation.¹

TICKET 6.2: Build Shopping Cart UI Component

- **Background:** A visual representation of the shopping cart is needed, often as a slide-out panel or a dedicated page. This component will listen for events dispatched from `ProductCard` components and update itself to reflect the cart's current state.
- **Acceptance Criteria:**
 - A reusable Livewire component, `ShoppingCart`, is created.
 - The component can be displayed as a slide-out panel or a dropdown, triggered by clicking the "Trainer's Bag" cart icon in the site header.
 - The component listens for the `add-to-cart` event and calls its own `refresh()` method or a method on the `CartService` to update its state.
 - It displays a list of all items in the cart, including their image, name, quantity, and price.
 - Users can update the quantity of an item or remove an item directly from this UI.
 - The component displays the cart subtotal and a clear "Checkout" button.
 - A badge on the main cart icon in the header displays the total number of items in the cart.
- **Technical Suggestions:**

- Use Livewire's event system (`$this->dispatch()` and `#[On('event-name')]`) for communication between the `ProductCard` and `ShoppingCart` components.¹
- The visibility of the slide-out panel should be controlled by Alpine.js to provide an instant open/close effect without a server request.

TICKET 6.3: Implement Multi-Step Checkout Component

- **Background:** The entire checkout process will be managed by a single, stateful Livewire component to provide a seamless user experience.¹ This component will guide the user through the required steps: reviewing their cart, providing shipping information, and selecting a payment method.
- **Acceptance Criteria:**
 - A full-page Livewire component, `CheckoutPage`, is created and accessible via a `/checkout` route.
 - The component manages the checkout state, including the current step (e.g., `shipping`, `payment`, `review`).
 - The first step displays a form for the user to enter or confirm their shipping address. Address fields are validated using Livewire's real-time validation.
 - The second step allows the user to select a payment method. Initially, this will only show "Cash on Delivery".
 - The final step shows a full order summary: all items, quantities, prices, shipping address, and the total amount.
 - All form inputs within the component use `wire:model.defer` to ensure optimal performance.[1, 1]
- **Technical Suggestions:**
 - Use a public property like `public string $step = 'shipping';` to control which part of the checkout view is rendered.
 - Validation rules should be defined in a dedicated `rules()` method on the component. Use TailwindUI's application shell layouts to structure the multi-step form cleanly.

TICKET 6.4: Implement COD Payment and Order Creation Logic

- **Background:** This is the final step of the checkout flow, where the user's cart is converted into a formal order in the database. The entire process must be wrapped in a database transaction to ensure atomicity—either all steps succeed, or everything is rolled back, preventing data corruption.¹
- **Acceptance Criteria:**
 - A `submitOrder()` method exists on the `CheckoutPage` component, triggered by the final confirmation button.
 - The entire logic within this method is wrapped in a `DB::transaction()` closure.
 - A new record is inserted into the `orders` table with the user's ID, total amount, a status of 'pending', and a `payment_method` of 'cod'.

- The system iterates through the cart items and creates a corresponding record for each in the `order_items` table, linking them to the new order ID.
 - The user's shopping cart is cleared (from both session and database).
 - Upon successful transaction commit, the user is redirected to a dedicated "Order Successful" confirmation page.
 - If any step within the transaction fails, all database changes are rolled back, and the user is shown an appropriate error message.
 - **Technical Suggestions:**
 - The inventory decrement logic should not be handled here in the application layer. This will be handled automatically by a database trigger that will be implemented in a subsequent epic, ensuring atomicity and preventing race conditions.¹
 - Abstract the core order creation logic into an `OrderProcessingService` to keep the Livewire component lean and focused on handling the request/response.
-

EPIC 7: Advanced Database Procedures and Triggers

Goal

To implement comprehensive stored procedures and triggers that enhance the e-commerce functionality, improve data integrity, provide audit trails, and automate critical business processes for the Pokemon card trading platform.

Context and Rationale

Building on the existing EPIC-6 commerce flow and database structure, EPIC 7 focuses on database-level automation and business logic. This epic implements stored procedures for complex operations and triggers for automatic data management, logging, and validation. These database features will improve performance, ensure data consistency, and provide comprehensive audit trails for business operations.

TICKET 7.1: Core Business Stored Procedures

Background: Implement essential stored procedures for critical business operations including order management, inventory control, and user operations.

Stored Procedures to Implement:

1. `sp_PlaceOrder` - Complete Order Processing

SQL

None

```
-- Handles complete order placement with inventory validation,  
stock updates, and transaction safety
```

```
DELIMITER //
```

```
CREATE PROCEDURE sp_PlaceOrder(  
  
    IN p_user_id INT,  
  
    IN p_shipping_first_name VARCHAR(50),  
  
    IN p_shipping_last_name VARCHAR(50),  
  
    IN p_shipping_email VARCHAR(100),  
  
    IN p_shipping_phone VARCHAR(20),  
  
    IN p_shipping_address_line_1 VARCHAR(255),  
  
    IN p_shipping_address_line_2 VARCHAR(255),  
  
    IN p_shipping_city VARCHAR(100),  
  
    IN p_shipping_state VARCHAR(50),  
  
    IN p_shipping_postal_code VARCHAR(20),  
  
    IN p_shipping_country VARCHAR(50),  
  
    IN p_payment_method ENUM('cod', 'paypal', 'stripe'),  
  
    IN p_special_instructions TEXT,  
  
    IN p_tax_rate DECIMAL(5,4),  
  
    IN p_shipping_cost DECIMAL(10,2),  
  
    OUT p_order_id INT,  
  
    OUT p_order_number VARCHAR(20),  
  
    OUT p_total_amount DECIMAL(10,2),
```

```

    OUT p_status VARCHAR(20),

    OUT p_message TEXT
)

BEGIN

    DECLARE v_subtotal DECIMAL(10,2) DEFAULT 0;

    DECLARE v_tax_amount DECIMAL(10,2) DEFAULT 0;

    DECLARE v_total_amount DECIMAL(10,2) DEFAULT 0;

    DECLARE v_product_id INT;

    DECLARE v_quantity INT;

    DECLARE v_unit_price DECIMAL(10,2);

    DECLARE v_stock INT;

    DECLARE v_product_name VARCHAR(255);

    DECLARE v_product_sku VARCHAR(50);

    DECLARE v_done INT DEFAULT FALSE;

    DECLARE v_order_number VARCHAR(20);

    -- Cursor for cart items

    DECLARE cart_cursor CURSOR FOR

        SELECT ci.product_id, ci.quantity, p.price, i.stock,
        p.name, p.sku

        FROM cart_items ci

        JOIN products p ON ci.product_id = p.id

```



```

        JOIN inventory i ON p.id = i.product_id

        WHERE ci.user_id = p_user_id;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

    ROLLBACK;

    SET p_status = 'ERROR';

    SET p_message = 'Order placement failed due to database
error';

    GET DIAGNOSTICS CONDITION 1 p_message = MESSAGE_TEXT;

END;

START TRANSACTION;

-- Check if cart has items

IF (SELECT COUNT(*) FROM cart_items WHERE user_id =
p_user_id) = 0 THEN

    SET p_status = 'ERROR';

    SET p_message = 'Cart is empty';

    ROLLBACK;

ELSE

```

```

-- Validate inventory for all items

OPEN cart_cursor;

read_loop: LOOP

    FETCH cart_cursor INTO v_product_id, v_quantity,
v_unit_price, v_stock, v_product_name, v_product_sku;

    IF v_done THEN

        LEAVE read_loop;

    END IF;

    IF v_stock < v_quantity THEN

        SET p_status = 'ERROR';

        SET p_message = CONCAT('Insufficient stock for
product: ', v_product_name, ' (Available: ', v_stock, ',
Requested: ', v_quantity, ')');

        CLOSE cart_cursor;

        ROLLBACK;

    END IF;

    SET v_subtotal = v_subtotal + (v_unit_price *
v_quantity);

END LOOP;

CLOSE cart_cursor;

```

```

-- Calculate totals

SET v_tax_amount = v_subtotal * p_tax_rate;

SET v_total_amount = v_subtotal + v_tax_amount +
p_shipping_cost;

-- Generate unique order number

SET v_order_number = CONCAT('ORD-', YEAR(NOW()), '-',
UPPER(SUBSTRING(MD5(CONCAT(p_user_id, NOW(), RAND())), 1, 8)));

-- Create order

INSERT INTO orders (

    order_number, user_id, status, payment_method,
payment_status,

    subtotal, tax_amount, shipping_cost, total_amount,

    shipping_first_name, shipping_last_name,
shipping_email, shipping_phone,

    shipping_address_line_1, shipping_address_line_2,
shipping_city,

    shipping_state, shipping_postal_code,
shipping_country,

    special_instructions, created_at, updated_at

) VALUES (

    v_order_number, p_user_id, 'pending',
p_payment_method, 'pending',

```

```

        v_subtotal, v_tax_amount, p_shipping_cost,
v_total_amount,

        p_shipping_first_name, p_shipping_last_name,
p_shipping_email, p_shipping_phone,

        p_shipping_address_line_1, p_shipping_address_line_2,
p_shipping_city,

        p_shipping_state, p_shipping_postal_code,
p_shipping_country,

        p_special_instructions, NOW(), NOW()
    );

SET p_order_id = LAST_INSERT_ID();

-- Reset cursor

SET v_done = FALSE;

-- Create order items and update inventory

OPEN cart_cursor;

insert_loop: LOOP

    FETCH cart_cursor INTO v_product_id, v_quantity,
v_unit_price, v_stock, v_product_name, v_product_sku;

    IF v_done THEN

        LEAVE insert_loop;

```

```

        END IF;

        -- Insert order item

        INSERT INTO order_items (
            order_id, product_id, product_name, product_sku,
            unit_price, quantity, total_price, created_at,
updated_at
        ) VALUES (
            p_order_id, v_product_id, v_product_name,
v_product_sku,
            v_unit_price, v_quantity, (v_unit_price *
v_quantity), NOW(), NOW()
        );

        -- Update inventory

        UPDATE inventory SET stock = stock - v_quantity WHERE
product_id = v_product_id;

    END LOOP;

    CLOSE cart_cursor;

    -- Clear cart

    DELETE FROM cart_items WHERE user_id = p_user_id;

```

```

        SET p_order_number = v_order_number;

        SET p_total_amount = v_total_amount;

        SET p_status = 'SUCCESS';

        SET p_message = 'Order placed successfully';

    COMMIT;

END IF;

END //

DELIMITER ;

```

2. **sp_CancelOrder** - Order Cancellation with Inventory Restoration SQL

```

None
DELIMITER //

CREATE PROCEDURE sp_CancelOrder(

    IN p_order_id INT,

    IN p_user_id INT,

    IN p_cancel_reason TEXT,

    OUT p_status VARCHAR(20),

    OUT p_message TEXT

)

BEGIN

```

```
DECLARE v_order_status VARCHAR(50);

DECLARE v_order_user_id INT;

DECLARE v_done INT DEFAULT FALSE;

DECLARE v_product_id INT;

DECLARE v_quantity INT;


DECLARE order_items_cursor CURSOR FOR

    SELECT product_id, quantity

    FROM order_items

    WHERE order_id = p_order_id;


DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

    ROLLBACK;

    SET p_status = 'ERROR';

    GET DIAGNOSTICS CONDITION 1 p_message = MESSAGE_TEXT;

END;


START TRANSACTION;
```

```
-- Validate order exists and belongs to user

SELECT status, user_id INTO v_order_status, v_order_user_id

FROM orders

WHERE id = p_order_id;

IF v_order_user_id IS NULL THEN

    SET p_status = 'ERROR';

    SET p_message = 'Order not found';

    ROLLBACK;

ELSEIF v_order_user_id!= p_user_id THEN

    SET p_status = 'ERROR';

    SET p_message = 'Unauthorized: Order does not belong to
user';

    ROLLBACK;

ELSEIF v_order_status IN ('cancelled', 'shipped',
'delivered') THEN

    SET p_status = 'ERROR';

    SET p_message = CONCAT('Cannot cancel order with status:
', v_order_status);

    ROLLBACK;

ELSE

    -- Restore inventory for all order items
```



```

OPEN order_items_cursor;

restore_loop: LOOP

    FETCH order_items_cursor INTO v_product_id,
v_quantity;

    IF v_done THEN

        LEAVE restore_loop;

    END IF;

    UPDATE inventory

    SET stock = stock + v_quantity

    WHERE product_id = v_product_id;

END LOOP;

CLOSE order_items_cursor;

-- Update order status

UPDATE orders

SET status = 'cancelled',

    notes = IFNULL(CONCAT(notes, '\n'), '') +
CONCAT('Cancelled: ', p_cancel_reason),

    updated_at = NOW()

WHERE id = p_order_id;

```

```
        SET p_status = 'SUCCESS';

        SET p_message = 'Order cancelled successfully';

    COMMIT;

END IF;

END //

DELIMITER ;
```

3. **sp_UpdateProductStock** - Inventory Management with Validation SQL

```
None
DELIMITER //

CREATE PROCEDURE sp_UpdateProductStock(

    IN p_product_id INT,

    IN p_new_stock INT,

    IN p_reason VARCHAR(255),

    IN p_updated_by INT,

    OUT p_status VARCHAR(20),

    OUT p_message TEXT

)

BEGIN

    DECLARE v_current_stock INT;
```

```
DECLARE v_product_name VARCHAR(255);

DECLARE v_min_stock INT DEFAULT 5;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

    ROLLBACK;

    SET p_status = 'ERROR';

    GET DIAGNOSTICS CONDITION 1 p_message = MESSAGE_TEXT;

END;

START TRANSACTION;

-- Validate product exists

SELECT i.stock, p.name

INTO v_current_stock, v_product_name

FROM inventory i

JOIN products p ON i.product_id = p.id

WHERE p.id = p_product_id;

IF v_product_name IS NULL THEN

    SET p_status = 'ERROR';
```

```
        SET p_message = 'Product not found';

        ROLLBACK;

ELSEIF p_new_stock < 0 THEN

    SET p_status = 'ERROR';

    SET p_message = 'Stock cannot be negative';

    ROLLBACK;

ELSE

    -- Update inventory

    UPDATE inventory

    SET stock = p_new_stock, updated_at = NOW()

    WHERE product_id = p_product_id;

    -- Log the change

    INSERT INTO inventory_logs (

        product_id, old_stock, new_stock, change_amount,

        reason, updated_by, created_at

    ) VALUES (

        p_product_id, v_current_stock, p_new_stock,

        (p_new_stock - v_current_stock), p_reason,

        p_updated_by, NOW()

    );
```

```

        SET p_status = 'SUCCESS';

        SET p_message = CONCAT('Stock updated for ',
v_product_name,
                                ' from ', v_current_stock, ' to
', p_new_stock);

        -- Check for low stock warning

        IF p_new_stock <= v_min_stock THEN

            SET p_message = CONCAT(p_message, ' (WARNING: Low
stock threshold reached)');

        END IF;

        COMMIT;

    END IF;

END //

DELIMITER ;

```

4. **sp_GetUserOrderHistory** - Comprehensive Order History

SQL

```

None
DELIMITER //

CREATE PROCEDURE sp_GetUserOrderHistory(

    IN p_user_id INT,

```

```
    IN p_limit INT DEFAULT 50,

    IN p_offset INT DEFAULT 0,

    IN p_status VARCHAR(50) DEFAULT NULL

)

BEGIN

    SELECT

        o.id,

        o.order_number,

        o.status,

        o.payment_method,

        o.payment_status,

        o.total_amount,

        o.created_at,

        o.updated_at,

        COUNT(oi.id) as item_count,

        SUM(oi.quantity) as total_quantity

    FROM orders o

    LEFT JOIN order_items oi ON o.id = oi.order_id

    WHERE o.user_id = p_user_id

    AND (p_status IS NULL OR o.status = p_status)

    GROUP BY o.id
```

```

ORDER BY o.created_at DESC

LIMIT p_limit OFFSET p_offset;

-- Also return total count for pagination

SELECT COUNT(*) as total_orders

FROM orders

WHERE user_id = p_user_id

AND (p_status IS NULL OR status = p_status);

END //

DELIMITER ;

```

5. **sp_ProcessRefund** - Handle Order Refunds

SQL

```

None
DELIMITER //

CREATE PROCEDURE sp_ProcessRefund(

    IN p_order_id INT,

    IN p_refund_amount DECIMAL(10,2),

    IN p_refund_reason TEXT,

    IN p_processed_by INT,

    OUT p_status VARCHAR(20),

    OUT p_message TEXT

```

)

BEGIN

DECLARE v_order_total DECIMAL(10,2);

DECLARE v_order_status VARCHAR(50);

DECLARE v_done INT DEFAULT FALSE;

DECLARE v_product_id INT;

DECLARE v_quantity INT;

DECLARE order_items_cursor CURSOR FOR

SELECT product_id, quantity

FROM order_items

WHERE order_id = p_order_id;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET v_done = TRUE;

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

ROLLBACK;

SET p_status = 'ERROR';

GET DIAGNOSTICS CONDITION 1 p_message = MESSAGE_TEXT;

END;


```
START TRANSACTION;

-- Validate order

SELECT total_amount, status

INTO v_order_total, v_order_status

FROM orders

WHERE id = p_order_id;

IF v_order_total IS NULL THEN

    SET p_status = 'ERROR';

    SET p_message = 'Order not found';

    ROLLBACK;

ELSEIF v_order_status = 'refunded' THEN

    SET p_status = 'ERROR';

    SET p_message = 'Order already refunded';

    ROLLBACK;

ELSEIF p_refund_amount > v_order_total THEN

    SET p_status = 'ERROR';

    SET p_message = 'Refund amount cannot exceed order
total';

    ROLLBACK;

ELSE
```

```

-- Restore inventory if full refund

IF p_refund_amount = v_order_total THEN

    OPEN order_items_cursor;

    refund_loop: LOOP

        FETCH order_items_cursor INTO v_product_id,
v_quantity;

        IF v_done THEN

            LEAVE refund_loop;

        END IF;

        UPDATE inventory

        SET stock = stock + v_quantity

        WHERE product_id = v_product_id;

    END LOOP;

    CLOSE order_items_cursor;

END IF;

-- Update order status

UPDATE orders

    SET status = IF(p_refund_amount = v_order_total,
'refunded', 'partially_refunded'),

        payment_status = 'refunded',

```

```

        notes = IFNULL(CONCAT(notes, '\n'), '') +
                CONCAT('Refund processed: $', p_refund_amount,
' - ', p_refund_reason),
        updated_at = NOW()
WHERE id = p_order_id;

-- Record refund

INSERT INTO refunds (
        order_id, amount, reason, processed_by, created_at
) VALUES (
        p_order_id, p_refund_amount, p_refund_reason,
p_processed_by, NOW()
);

SET p_status = 'SUCCESS';

SET p_message = CONCAT('Refund of $', p_refund_amount, '
processed successfully');

COMMIT;

END IF;

END //

DELIMITER ;

```

6. `sp_GetLowStockProducts` - Inventory Monitoring SQL

None

```
DELIMITER //
```

```
CREATE PROCEDURE sp_GetLowStockProducts(
```

```
    IN p_threshold INT DEFAULT 5
```

```
)
```

```
BEGIN
```

```
    SELECT
```

```
        p.id,
```

```
        p.name,
```

```
        p.sku,
```

```
        c.name as card_name,
```

```
        s.name as set_name,
```

```
        r.name as rarity_name,
```

```
        i.stock,
```

```
        p.price,
```

```
        p.is_active
```

```
    FROM products p
```

```
    JOIN inventory i ON p.id = i.product_id
```

```
    JOIN cards c ON p.card_id = c.id
```

```
    JOIN sets s ON c.set_id = s.id
```

```
    JOIN rarities r ON c.rarity_id = r.id
```

```
WHERE i.stock <= p_threshold

AND p.is_active = 1

ORDER BY i.stock ASC, p.name ASC;

END //

DELIMITER ;
```

7. **sp_ArchiveOldOrders** - Data Management

SQL

```
None
DELIMITER //

CREATE PROCEDURE sp_ArchiveOldOrders(

    IN p_days_old INT DEFAULT 365,

    OUT p_archived_count INT

)

BEGIN

    DECLARE EXIT HANDLER FOR SQLEXCEPTION

    BEGIN

        ROLLBACK;

        SET p_archived_count = -1;

    END;

    START TRANSACTION;
```

```
-- Archive orders older than specified days with completed  
status
```

```
INSERT INTO archived_orders
```

```
SELECT *, NOW() as archived_at
```

```
FROM orders
```

```
WHERE status IN ('delivered', 'cancelled', 'refunded')
```

```
AND created_at < DATE_SUB(NOW(), INTERVAL p_days_old DAY);
```

```
SET p_archived_count = ROW_COUNT();
```

```
-- Archive corresponding order items
```

```
INSERT INTO archived_order_items
```

```
SELECT oi.*, NOW() as archived_at
```

```
FROM order_items oi
```

```
JOIN orders o ON oi.order_id = o.id
```

```
WHERE o.status IN ('delivered', 'cancelled', 'refunded')
```

```
AND o.created_at < DATE_SUB(NOW(), INTERVAL p_days_old DAY);
```

```
-- Delete from main tables
```

```
DELETE oi FROM order_items oi
```

```
JOIN orders o ON oi.order_id = o.id
```

```
WHERE o.status IN ('delivered', 'cancelled', 'refunded')

AND o.created_at < DATE_SUB(NOW(), INTERVAL p_days_old DAY);


DELETE FROM orders

WHERE status IN ('delivered', 'cancelled', 'refunded')

AND created_at < DATE_SUB(NOW(), INTERVAL p_days_old DAY);


COMMIT;

END //

DELIMITER ;
```

TICKET 7.2: Automated Triggers for Data Integrity and Logging

Background: Implement triggers that automatically handle data validation, logging, and business rule enforcement.

Triggers to Implement:

1. `tr_orders_inventory_update` - Automatic Inventory Management

SQL

```
None
DELIMITER //

CREATE TRIGGER tr_orders_inventory_update

AFTER UPDATE ON orders

FOR EACH ROW

BEGIN
```

```
-- If order status changes to 'shipped', log the shipment
IF OLD.status!= NEW.status AND NEW.status = 'shipped' THEN

    INSERT INTO order_status_log (

        order_id, old_status, new_status, changed_by,

        change_reason, created_at

    ) VALUES (

        NEW.id, OLD.status, NEW.status, NEW.updated_by,

        'Order shipped', NOW()

    );

END IF;
```

-- If order is cancelled, restore inventory (backup to stored procedure)

```
IF OLD.status!= NEW.status AND NEW.status = 'cancelled' THEN

    -- This is handled by sp_CancelOrder, but serves as
    backup

    INSERT INTO inventory_adjustments (

        order_id, adjustment_type, reason, created_at

    ) VALUES (

        NEW.id, 'restore', 'Order cancelled', NOW()

    );

END IF;
```



```
END //
```

```
DELIMITER ;
```

2. `tr_products_price_history` - Price Change Tracking SQL

None

```
DELIMITER //
```

```
CREATE TRIGGER tr_products_price_history
```

```
AFTER UPDATE ON products
```

```
FOR EACH ROW
```

```
BEGIN
```

```
-- Log price changes
```

```
IF OLD.price!= NEW.price THEN
```

```
    INSERT INTO price_history (
```

```
        product_id, old_price, new_price, change_amount,
```

```
        change_percentage, changed_by, created_at
```

```
    ) VALUES (
```

```
        NEW.id, OLD.price, NEW.price,
```

```
        (NEW.price - OLD.price),
```

```
        ROUND(((NEW.price - OLD.price) / OLD.price) * 100,  
2),
```

```
        NEW.updated_by, NOW()
```

```
    );
```

```

END IF;

-- Log status changes
IF OLD.is_active!= NEW.is_active THEN

    INSERT INTO product_status_log (
        product_id, old_status, new_status, changed_by,
created_at
    ) VALUES (
        NEW.id,
        IF(OLD.is_active = 1, 'active', 'inactive'),
        IF(NEW.is_active = 1, 'active', 'inactive'),
        NEW.updated_by, NOW()
    );

END IF;

END //

DELIMITER ;

```

3. **tr_cart_items_validation** - Cart Item Validation

SQL

```

None
DELIMITER //

CREATE TRIGGER tr_cart_items_validation

BEFORE INSERT ON cart_items

```

```
FOR EACH ROW
BEGIN
    DECLARE v_stock INT;
    DECLARE v_is_active BOOLEAN;
    DECLARE v_price DECIMAL(10,2);

    -- Validate product exists and is active
    SELECT i.stock, p.is_active, p.price
    INTO v_stock, v_is_active, v_price
    FROM inventory i
    JOIN products p ON i.product_id = p.id
    WHERE p.id = NEW.product_id;

    IF v_stock IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Product not
found';
    END IF;

    IF v_is_active = 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Product is
not active';
    END IF;
```

```

        IF NEW.quantity <= 0 THEN

            SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Quantity must
be greater than 0';

        END IF;

        IF NEW.quantity > v_stock THEN

            SIGNAL SQLSTATE '45000'

            SET MESSAGE_TEXT = CONCAT('Insufficient stock. Available:
', v_stock, ', Requested: ', NEW.quantity);

        END IF;

        -- Set timestamps

        SET NEW.created_at = NOW();

        SET NEW.updated_at = NOW();

    END //

DELIMITER ;

```

4. **tr_inventory_low_stock_alert** - Stock Threshold Monitoring SQL

```

None
DELIMITER //

CREATE TRIGGER tr_inventory_low_stock_alert

```

```
AFTER UPDATE ON inventory

FOR EACH ROW

BEGIN

    DECLARE v_product_name VARCHAR(255);

    DECLARE v_threshold INT DEFAULT 5;

    -- Check if stock dropped below threshold

    IF OLD.stock > v_threshold AND NEW.stock <= v_threshold THEN

        SELECT name INTO v_product_name

        FROM products

        WHERE id = NEW.product_id;

        INSERT INTO stock_alerts (

            product_id, product_name, current_stock,

threshold_value,

            alert_type, message, created_at

        ) VALUES (

            NEW.product_id, v_product_name, NEW.stock,

v_threshold,

            'low_stock',

            CONCAT('Product "', v_product_name, '" is below stock

threshold. Current stock: ', NEW.stock),
```

```

        NOW()

    );

END IF;

-- Check if stock reaches zero

IF OLD.stock > 0 AND NEW.stock = 0 THEN

    SELECT name INTO v_product_name

    FROM products

    WHERE id = NEW.product_id;

    INSERT INTO stock_alerts (

        product_id, product_name, current_stock,
threshold_value,

        alert_type, message, created_at

    ) VALUES (

        NEW.product_id, v_product_name, NEW.stock, 0,

        'out_of_stock',

        CONCAT('Product "', v_product_name, '" is now out of
stock'),

        NOW()

    );

END IF;

```

```
END //
```

```
DELIMITER ;
```

5. `tr_user_activity_log` - User Action Tracking SQL

None

```
DELIMITER //
```

```
CREATE TRIGGER tr_user_activity_log_insert
```

```
AFTER INSERT ON orders
```

```
FOR EACH ROW
```

```
BEGIN
```

```
    INSERT INTO user_activity_log (
```

```
        user_id, activity_type, description,
```

```
        related_table, related_id, ip_address, created_at
```

```
    ) VALUES (
```

```
        NEW.user_id, 'order_placed',
```

```
        CONCAT('Order placed: ', NEW.order_number, ' - Total: $',  
NEW.total_amount),
```

```
        'orders', NEW.id, NEW.user_ip, NOW()
```

```
    );
```

```
END //
```

```
DELIMITER ;
```

```

CREATE TRIGGER tr_user_activity_log_login

AFTER UPDATE ON users

FOR EACH ROW

BEGIN

    IF OLD.last_login_at!= NEW.last_login_at THEN

        INSERT INTO user_activity_log (

            user_id, activity_type, description,

            ip_address, created_at

        ) VALUES (

            NEW.id, 'login',

            CONCAT('User logged in: ', NEW.email),

            NEW.last_login_ip, NOW()

        );

    END IF;

END //

DELIMITER ;

```

6. **tr_order_total_calculation** - Automatic Order Total Updates

SQL

```

None

DELIMITER //

CREATE TRIGGER tr_order_total_recalculation

```



```
AFTER INSERT ON order_items

FOR EACH ROW

BEGIN

    DECLARE v_new_subtotal DECIMAL(10,2);

    DECLARE v_tax_rate DECIMAL(5,4) DEFAULT 0.08;

    DECLARE v_shipping_cost DECIMAL(10,2);

    DECLARE v_tax_amount DECIMAL(10,2);

    DECLARE v_total_amount DECIMAL(10,2);


    -- Recalculate order totals

    SELECT SUM(total_price), shipping_cost

    INTO v_new_subtotal, v_shipping_cost

    FROM order_items oi

    JOIN orders o ON oi.order_id = o.id

    WHERE oi.order_id = NEW.order_id

    GROUP BY o.shipping_cost;


    SET v_tax_amount = v_new_subtotal * v_tax_rate;

    SET v_total_amount = v_new_subtotal + v_tax_amount +
v_shipping_cost;


    -- Update order totals
```

```
UPDATE orders

SET subtotal = v_new_subtotal,

    tax_amount = v_tax_amount,

    total_amount = v_total_amount,

    updated_at = NOW()

WHERE id = NEW.order_id;

END //

DELIMITER ;
```

7. **tr_product_search_index_update** - Search Index Maintenance SQL

```
None
DELIMITER //

CREATE TRIGGER tr_product_search_index_update

AFTER UPDATE ON products

FOR EACH ROW

BEGIN

    -- Update search index when product details change

    IF OLD.name!= NEW.name OR OLD.description!= NEW.description
    OR OLD.is_active!= NEW.is_active THEN

        INSERT INTO search_index_updates (

            table_name, record_id, action, created_at

        ) VALUES (
```

```

        'products', NEW.id, 'update', NOW()
    );
END IF;
END //
DELIMITER ;

CREATE TRIGGER tr_product_search_index_insert
AFTER INSERT ON products
FOR EACH ROW
BEGIN
    INSERT INTO search_index_updates (
        table_name, record_id, action, created_at
    ) VALUES (
        'products', NEW.id, 'insert', NOW()
    );
END //
DELIMITER ;

```

TICKET 7.3: Supporting Tables and Views

Background: Create necessary supporting tables for logging and create useful views for reporting.

Required Tables:

SQL

None

-- Inventory change logging

```
CREATE TABLE inventory_logs (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    product_id INT NOT NULL,  
    old_stock INT NOT NULL,  
    new_stock INT NOT NULL,  
    change_amount INT NOT NULL,  
    reason VARCHAR(255),  
    updated_by INT,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (product_id) REFERENCES products(id),  
    FOREIGN KEY (updated_by) REFERENCES users(id)  
);
```

-- Price change history

```
CREATE TABLE price_history (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    product_id INT NOT NULL,  
    old_price DECIMAL(10,2) NOT NULL,  
    new_price DECIMAL(10,2) NOT NULL,  
    change_amount DECIMAL(10,2) NOT NULL,
```

```
change_percentage DECIMAL(5,2),

changed_by INT,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

FOREIGN KEY (product_id) REFERENCES products(id),

FOREIGN KEY (changed_by) REFERENCES users(id)

);

-- Order status logging

CREATE TABLE order_status_log (

    id INT AUTO_INCREMENT PRIMARY KEY,

    order_id INT NOT NULL,

    old_status VARCHAR(50),

    new_status VARCHAR(50) NOT NULL,

    changed_by INT,

    change_reason TEXT,

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (order_id) REFERENCES orders(id),

    FOREIGN KEY (changed_by) REFERENCES users(id)

);

-- Stock alerts
```

```
CREATE TABLE stock_alerts (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    product_id INT NOT NULL,  
    product_name VARCHAR(255),  
    current_stock INT NOT NULL,  
    threshold_value INT NOT NULL,  
    alert_type ENUM('low_stock', 'out_of_stock') NOT NULL,  
    message TEXT,  
    is_resolved BOOLEAN DEFAULT FALSE,  
    resolved_at TIMESTAMP NULL,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (product_id) REFERENCES products(id)  
);
```

-- User activity logging

```
CREATE TABLE user_activity_log (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    user_id INT NOT NULL,  
    activity_type VARCHAR(50) NOT NULL,  
    description TEXT,  
    related_table VARCHAR(50),
```

```
related_id INT,  
  
ip_address VARCHAR(45),  
  
user_agent TEXT,  
  
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  
FOREIGN KEY (user_id) REFERENCES users(id),  
  
INDEX idx_user_activity (user_id, created_at),  
  
INDEX idx_activity_type (activity_type, created_at)  
);  
  
  
-- Refunds tracking  
  
CREATE TABLE refunds (  
  
    id INT AUTO_INCREMENT PRIMARY KEY,  
  
    order_id INT NOT NULL,  
  
    amount DECIMAL(10,2) NOT NULL,  
  
    reason TEXT,  
  
    processed_by INT,  
  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  
    FOREIGN KEY (order_id) REFERENCES orders(id),  
  
    FOREIGN KEY (processed_by) REFERENCES users(id)  
);
```

```
-- Archive tables

CREATE TABLE archived_orders LIKE orders;

ALTER TABLE archived_orders ADD COLUMN archived_at TIMESTAMP
DEFAULT CURRENT_TIMESTAMP;

CREATE TABLE archived_order_items LIKE order_items;

ALTER TABLE archived_order_items ADD COLUMN archived_at TIMESTAMP
DEFAULT CURRENT_TIMESTAMP;
```

TICKET 7.4: Testing and Validation

Background: Create comprehensive tests to validate all stored procedures and triggers are working correctly.

Acceptance Criteria:

- All 7+ stored procedures execute without errors
- All 7+ triggers fire correctly on their respective events
- Data integrity is maintained across all operations
- Error handling works as expected
- Performance benchmarks are met
- Documentation is complete

Test Cases Required:

1. Stored Procedure Tests:

- Order placement with sufficient inventory
- Order placement with insufficient inventory
- Order cancellation and inventory restoration
- Stock updates with validation
- Refund processing
- Low stock product identification
- Order archiving

2. Trigger Tests:

- Inventory updates on order status changes
- Price change logging
- Cart validation on item insertion

- Stock alert generation
 - User activity logging
 - Order total recalculation
-

Summary

This expanded EPIC 7 provides:

- **7 Comprehensive Stored Procedures** for core business operations
- **10+ Triggers** for automated data management
- **Complete audit trail** for all critical operations
- **Data integrity validation** at database level
- **Performance optimization** through database-level logic
- **Business rule enforcement** through constraints and triggers
- **Comprehensive logging** for debugging and compliance

The implementation focuses on the Pokemon card e-commerce platform's specific needs while maintaining scalability and data integrity.

The remaining epics (Order Management, Buyback System, MCP Integration) will be detailed with the same structure and level of exhaustive detail, ensuring full coverage of all product and technical requirements.