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]

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 Logic Implementation	To implement the mandated advanced MySQL features—triggers and stored procedures—to enforce data integrity and automate critical business logic at the database level.
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] 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]
- **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] 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]

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]

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 (belongsTo, hasMany) are correctly defined on all models to reflect their relational structure (e.g., a Card belongsTo 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]
- **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]

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 core structure, adding interactive elements, and finishing with the final visual polish.

Tickets

TICKET 4.1: 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 4.2: 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 4.3: 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 4.4: 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 4.5: 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] 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]
- **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 Logic Implementation

Goal

To implement the mandated advanced MySQL features—triggers and stored procedures—to enforce data integrity and automate critical business logic at the database level.

Context and Rationale

A key architectural mandate for Konibui is the strategic use of advanced database features to enhance data integrity, performance, and reliability.[1] This epic moves critical business logic from the application layer directly into the database layer. Offloading tasks like inventory management to a database trigger ensures they are executed atomically with order creation, completely eliminating the risk of race conditions where two users might simultaneously buy the last item in stock.

Furthermore, encapsulating the entire multi-step order creation process within a stored procedure reduces network latency between the application and the database from multiple queries to a single call, improving performance. These features are not merely for academic demonstration; they are strategically chosen to build a more robust and resilient e-commerce

platform by leveraging the power and reliability of the database itself. All implementations will be managed via Laravel Migrations to ensure they are version-controlled and repeatable.

Tickets

TICKET 7.1: Implement Inventory Decrement Database Trigger

- **Background:** To guarantee that inventory levels are always accurate and that stock adjustments are atomic with order creation, this logic will be handled by a database trigger.[1] An AFTER INSERT trigger on the order_items table will automatically decrement the stock for the corresponding product in the inventory table. This is the most reliable way to prevent overselling.
- **Acceptance Criteria:**
  - A new Laravel migration is created to define the database trigger.
  - The migration contains raw SQL to create an AFTER INSERT trigger named after_order_item_insert on the order_items table.
  - The trigger's logic correctly executes an UPDATE statement on the inventory table, decrementing the stock column by the quantity of the newly inserted order_item.
  - The UPDATE statement includes a WHERE clause to ensure stock is not decremented below zero (stock >= NEW.quantity).
  - Manually inserting a record into the order_items table (via phpMyAdmin or a test) correctly and automatically reduces the stock in the inventory table.
  - The migration's down() method correctly drops the trigger.
- **Technical Suggestions:**
  - Use the DELIMITER command in the raw SQL to allow for multi-statement trigger bodies. The entire CREATE TRIGGER statement should be wrapped in a DB::unprepared() call within the migration.

TICKET 7.2: Implement Order Cancellation/Return Inventory Trigger

- **Background:** To complement the decrement trigger, a corresponding mechanism is needed to return stock to inventory when an order is cancelled or an item is returned. This can be implemented as another trigger, for instance, an AFTER UPDATE trigger on the orders table that checks for a status change.
- **Acceptance Criteria:**
  - A new migration is created to define an inventory increment trigger.
  - An AFTER UPDATE trigger is created on the orders table (or alternatively, an AFTER DELETE trigger on order_items if cancellations involve deleting line items).
  - The trigger logic checks if the order's status has changed to 'Cancelled' or 'Returned'.
  - If the status matches, the trigger logic finds the associated order_items and executes UPDATE statements on the inventory table to increment the stock for each returned item.

    - Manually updating an order's status to 'Cancelled' correctly and automatically restores the stock levels for all items in that order.
- **Technical Suggestions:**
    - This logic can be complex. An alternative to an UPDATE trigger on orders is to have a separate order_item_cancellations table. An INSERT trigger on this new table would then handle the inventory increment. This approach often provides a clearer audit trail. The team should evaluate the best approach.

TICKET 7.3: Develop CreateNewOrder Stored Procedure

- **Background:** For a complex, multi-step operation like creating a complete order (inserting into orders, then looping to insert into order_items, then logging the transaction), a stored procedure is the ideal tool.[1] It encapsulates the entire business transaction within the database, ensuring atomicity and reducing network overhead to a single CALL from the Laravel application.
- **Acceptance Criteria:**
    - A new migration is created to define a stored procedure named CreateNewOrder.
    - The procedure accepts input parameters such as in_user_id, in_payment_method, in_total_amount, and in_order_items (as a JSON string).
    - The procedure's body begins with START TRANSACTION.
    - It correctly inserts a new record into the orders table.
    - It parses the in_order_items JSON array and iterates through it, inserting a record into the order_items table for each item.
    - The after_order_item_insert trigger (from Ticket 7.1) fires correctly for each item insertion.
    - It includes an error handler that executes a ROLLBACK if any SQL exception occurs.
    - If all steps succeed, it executes a COMMIT.
    - The Laravel application's OrderProcessingService is refactored to call this stored procedure via DB::statement('CALL CreateNewOrder(...)') instead of using Eloquent.
- **Technical Suggestions:**
    - MySQL's native JSON functions (JSON_TABLE, JSON_EXTRACT) can be used within the stored procedure to parse the input array of items.
    - Use DECLARE EXIT HANDLER FOR SQLEXCEPTION to implement robust error handling and rollback logic within the procedure.

TICKET 7.4: Implement Data Archiving Strategy

- **Background:** To maintain high query performance on "hot" operational tables like orders and transaction_logs as they grow over time, a data archiving strategy is required.[1] This involves automatically moving old, completed records to separate archive tables, keeping the primary tables lean and fast.
- **Acceptance Criteria:**

    - Archive tables (e.g., orders_archive, transaction_logs_archive) are created via migrations. Their structure is identical to the primary tables, with an added archived_at timestamp.
    - A BEFORE DELETE trigger is placed on the primary orders table.
    - When a record is deleted from orders, this trigger first copies the entire row into the orders_archive table before the deletion proceeds.
    - A scheduled Laravel Artisan command (e.g., app:archive-old-records) is created.
    - This command queries for orders with a 'Completed' or 'Cancelled' status that are older than a defined retention period (e.g., 2 years).
    - The command executes a DELETE statement on these old orders, which in turn fires the trigger to archive them.
    - The command is registered with Laravel's scheduler to run on a nightly basis.
- **Technical Suggestions:**
    - The retention period should be configurable in an application config file (e.g., config/konibui.php).
    - Process the records to be deleted in chunks within the Artisan command (->chunkById(...)) to avoid consuming too much memory if a large number of records need to be archived at once.

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.