Technical Requirements Document

Introduction

This Technical Requirements Document (TRD) outlines the technical specifications, architectural decisions, technology stack, and development approach for the GAMEDROP web application. The purpose of this document is to serve as a technical blueprint for the development team, guiding the implementation process.

The functional scope and user-facing features of GAMEDROP are detailed in the accompanying Functional Requirements Document (FRD). This TRD focuses on the how – the technical means by which the functional requirements will be met.

System Architecture

1. High-Level Overview

GAMEDROP will be implemented using a **Client-Server Architecture**. This architectural pattern separates the application into two distinct parts:

1. **Client (Frontend):** A browser-based **Single Page Application (SPA)** built using **React**. This component is responsible for rendering the User Interface (UI), handling user interactions, and communicating with the backend via API calls.
2. **Server (Backend):** An **API server** built using **Node.js** and the **Express.js** framework. This component handles business logic, data persistence (database interactions), user authentication, and exposes functionality through a **RESTful API**.

Communication between the frontend and backend will occur over HTTPS using standard **REST principles**, with data primarily exchanged in **JSON format**. This separation promotes modularity, allows for independent development and deployment of frontend and backend, and provides a foundation for future scalability.

2. Backend Architectural Pattern

The Node.js/Express backend code will be organized following a pattern similar to **Model-View-Controller (MVC)**, adapted for an API server, and will make extensive use of **Middleware**.

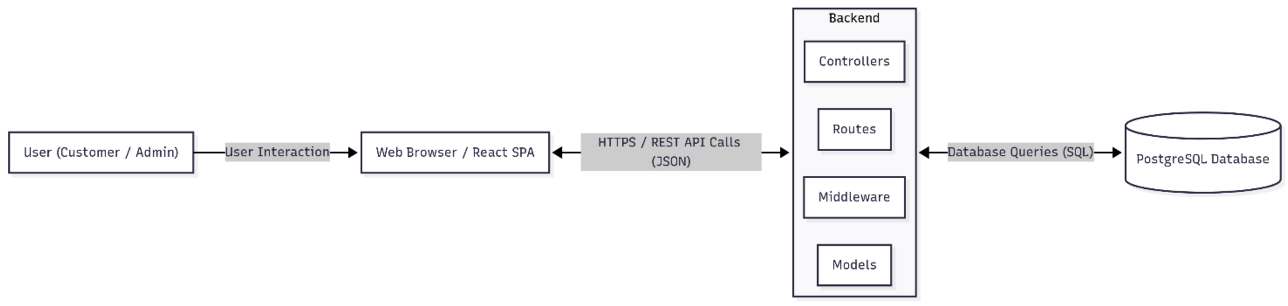
- **Models:** Responsible for defining data structures (e.g., User, Product, Order in `models/user.js`, `models/product.js`) and interacting directly with the PostgreSQL database (performing Create, Read, Update, Delete - CRUD operations). We will use **Prisma ORM** for database interactions.
- **Controllers (Route Handlers):** (e.g., in `controllers/productController.js`, `controllers/authController.js`) These handle the core logic for specific API requests. They receive the request details (after processing by middleware), interact with Models (or Services) to fetch or manipulate data, and then format and send the JSON response back to the client.
- **Routes:** (e.g., in `routes/productRoutes.js`, `routes/authRoutes.js`) These define the API endpoints (URLs like `/api/products` or `/api/auth/login`) and map incoming requests (based on the URL and HTTP method like GET, POST) to the correct Controller function. Routes will be organized into grouped handler files.
- **Middleware:** (e.g., in `middleware/authMiddleware.js`, `middleware/validationMiddleware.js`) These are functions that execute *during* the request-response cycle, typically *before* the main Controller logic runs. Think of them as checkpoints or helpers that process the request sequentially. Middleware will be used for essential tasks, such as:
    - **Request Parsing:** Automatically parsing incoming JSON data from the frontend into usable JavaScript objects (using `express.json()`).

- **Authentication:** Verifying the JWT token sent with requests to protected routes (like placing an order). If the token is invalid or missing, the middleware will stop the request and send an "Unauthorized" response *before* it reaches the intended Controller.
- **Logging:** Recording information about incoming requests for debugging.
- **Validation:** Performing basic checks on incoming data before the Controller processes it.
- **Error Handling:** Catching errors that occur during request processing and sending a standardized error response.

3. Frontend Architectural Pattern

- The **React** frontend will adhere to a **Component-Based Architecture**.
- The UI will be built by composing small, reusable **Components** (e.g., Button, ProductCard, SearchBar, CartItem).
- These components will be assembled into larger **Pages** or **Views** corresponding to different application routes (e.g., HomePage, ProductListPage, ProductDetailPage, CartPage, CheckoutPage, AdminDashboard).
- Client-side routing will be managed by **React Router**.
- State management will primarily use local component state (`useState`) and **React Context API** for global concerns like authentication status.

4. High-Level Design



Technology Stack

❖ Backend:

- **Runtime:** Node.js
- **Framework:** Express.js
- **Database:** PostgreSQL
- **Database Interaction:** Prisma ORM
- **Authentication:** `jsonwebtoken` (for JWT handling), `bcryptjs` (for password hashing).

❖ Frontend:

- **Library/Framework:** React
- **Routing:** `react-router-dom`
- **Form Handling:** `formik`
- **State Management:** React's built-in state (`useState`, `useContext`)
- **API Calls:** Browser `Workspace` API or `axios`.
- **CSS / UI Framework:** Material UI

- **Package Manager:** `npm` (comes with Node.js)
- **Version Control:** Git

Database Design

This section outlines the database design approach. A separate, more detailed **Database Schema Document** will be created to specify all tables, columns, data types, relationships, primary keys, foreign keys, and constraints.

- **Main Entities/Tables (Conceptual - details in Schema Doc):**

Table	Attributes
User	`id` (Primary Key, UUID/Serial), `username` (Unique, Text), `email` (Unique, Text), `password_hash` (Text), `role` (Text - e.g., 'CUSTOMER', 'ADMIN'), `created_at` (Timestamp), `updated_at` (Timestamp).
Product	`id` (PK), `title` (Text), `description` (Text), `platform` (Text - could reference a `Category` table), `genre` (Text - could reference a `Category` table), `price` (Decimal), `stock_quantity` (Integer), `cover_image_url` (Text), `release_date` (Date), `created_at`, `updated_at`.
Cart	`id` (PK), `user_id` (Foreign Key referencing `User.id`), `created_at`, `updated_at`.
CartItem	`id` (PK), `cart_id` (FK referencing `Cart.id`), `product_id` (FK referencing `Product.id`), `quantity` (Integer).
Order	`id` (PK), `user_id` (FK referencing `User.id`), `total_amount` (Decimal), `status` (Text - e.g., 'PENDING', 'PROCESSING', 'SHIPPED', 'DELIVERED', 'CANCELLED'), `shipping_address` (JSON or Text fields), `order_date` (Timestamp), `created_at`, `updated_at`.
OrderItem	`id` (PK), `order_id` (FK referencing `Order.id`), `product_id` (FK referencing `Product.id`), `quantity` (Integer), `price_at_purchase` (Decimal).

Review	`id` (PK), `user_id` (FK referencing `User.id`), `product_id` (FK referencing `Product.id`), `rating` (Integer, 1-5), `comment` (Text), `review_date` (Timestamp).

- **Relationships (Conceptual):**
  - A `User` can have many `Order`s. An `Order` belongs to one `User`.
  - A `User` can have one `Cart`. A `Cart` belongs to one `User`.
  - A `Cart` can have many `CartItem`s. A `CartItem` belongs to one `Cart` and one `Product`.
  - An `Order` can have many `OrderItem`s. An `OrderItem` belongs to one `Order` and one `Product`.
  - A `Product` can have many `Review`s. A `Review` belongs to one `Product` and one `User`.
- **Password Storage:** As stated previously, user passwords will be securely stored by hashing them with `bcryptjs` and including a salt.
- **Data Integrity:** Appropriate constraints (NOT NULL, UNIQUE, foreign key constraints) will be defined in the Database Schema Document to ensure data integrity.

API Design

The backend will expose a **RESTful API** (Representational State Transfer Application Programming Interface). This is a standard way for the frontend (React app) and backend (Node.js server) to communicate over the internet (using HTTPS).

- **Core Idea:** We use specific URLs (Uniform Resource Locators) to represent different "resources" in our application (like products, orders, users). We use standard HTTP methods (like GET, POST, PUT, DELETE) to tell the server what action we want to perform on those resources.
- **Data Format:** All data exchanged between the frontend and backend will be in **JSON** (JavaScript Object Notation) format. JSON is a simple, text-based format using key-value pairs that JavaScript understands easily.
    - Example JSON for a product:

      ```
      {
          "id": 123,
          "title": "Awesome Game",
          "platform": "PS5",
          "price": 59.99,
          "stock_quantity": 10
      }
      ```
- **Authentication:** For actions that require a user to be logged in (like placing an order or viewing order history) or require admin privileges, the frontend must send the user's **JWT** (JSON Web Token) to the backend. This token is typically sent in the `Authorization` header of the HTTP request, like this: `Authorization: Bearer <your_jwt_token_here>`. Backend **middleware** will check this token before allowing access to protected routes.
- **Key Resource Endpoints (Examples):** This is not the full list, but shows the pattern:
    - Authentication:
        - `POST /api/auth/register` - Create a new user account.
        - `POST /api/auth/login` - Log in and receive a JWT.

    - Products:
        - `GET /api/products` - Get a list of all available products (can include filters like `/api/products?genre=RPG`).
        - `GET /api/products/:id` - Get details for a single product (e.g., `/api/products/123`).
        - `POST /api/products` - (*Admin only*) Add a new product.
    - Orders:
        - `POST /api/orders` - (*Customer logged in*) Place a new order (cart details in request body).
        - `GET /api/orders/my-history` - (*Customer logged in*) Get the logged-in user's order history.
        - `GET /api/orders` - (*Admin only*) Get a list of all orders (can include filters).
        - `GET /api/orders/:id` - (*Admin only*) Get details of a specific order.
        - `PUT /api/orders/:id/status` - (*Admin only*) Update the status of an order.
    - Cart:
        - `GET /api/cart` - (*Customer logged in*) Get the user's current cart contents.
        - `POST /api/cart/items` - (*Customer logged in*) Add an item to the cart.
        - `PUT /api/cart/items/:productId` - (*Customer logged in*) Update quantity of an item.
        - `DELETE /api/cart/items/:productId` - (*Customer logged in*) Remove item from cart.
- **Responses:** The API will use standard HTTP status codes to indicate success or failure (e.g., `200 OK`, `201 Created`, `400 Bad Request`, `401 Unauthorized`, `404 Not Found`, `500 Internal Server Error`). Successful responses containing data will have a JSON body. Error responses will also typically have a JSON body explaining the error.

Frontend Design

- **Core Architecture: Component-Based Architecture** using **React**. The UI will be constructed from reusable, modular components.
- **Routing:**
    - **Library:** `react-router-dom`
    - **Strategy:** Client-side routing will be implemented to create a seamless SPA experience.
    - **Key Routes:**
        - `/` (Homepage / Product Catalog)
        - `/products/:productId` (Product Detail Page)
        - `/category/:categoryName` (Product listing by category/platform/genre)
        - `/cart` (Shopping Cart Page)
        - `/checkout` (Checkout Process Page)
        - `/login` (Login Page)
        - `/register` (Registration Page)
        - `/account/profile` (User Profile Page)
        - `/account/orders` (User Order History Page)
        - `/account/orders/:orderId` (Specific Order Detail Page)
        - `/admin/dashboard` (Admin Dashboard - entry point)
        - `/admin/products` (Admin Product Management)
        - `/admin/products/new` (Admin Add New Product Form)
        - `/admin/products/edit/:productId` (Admin Edit Product Form)
        - `/admin/orders` (Admin Order Management)
        - `/admin/users` (Admin User Management)
- **Key Conceptual Components (Examples):**
    - `Navbar`: Site navigation.
    - `Footer`: Site footer.
    - `ProductCard`: Displays summary info of a game in lists.
    - `ProductList`: Renders a collection of `ProductCard` components.
    - `ProductDetailView`: Displays full details for a single product.
    - `SearchBar`: Component for text-based product search.
    - `FilterPanel`: Component for category/platform/genre/price filters.
    - `ShoppingCartView`: Displays items in the cart.
    - `CartItem`: Represents a single item within the shopping cart.
    - `CheckoutForm`: Multi-step form for shipping, payment simulation.
    - `LoginForm`, `RegistrationForm`: User authentication forms.

    - `ReviewForm`: For submitting product reviews.
    - `ReviewList`: Displays reviews for a product.
    - `AdminProductTable`, `AdminOrderTable`, `AdminUserTable`: For admin management views.
    - `AdminProductForm`: Form for admins to add/edit products.
- **Form Handling:**
    - **Library:** `formik` (as required).
    - **Usage:** All forms (user registration, login, checkout shipping details, product reviews, admin forms) will utilize `Formik` for managing form state, handling submission, and implementing client-side input validation. Error messages for invalid inputs will be displayed appropriately.
- **State Management:**
    - **Local State:** `useState` and `useReducer` hooks will be used for managing state within individual components or closely related component trees.
    - **Global State: React Context API** will be used for managing global application state such as user authentication status (e.g., logged-in user details, JWT token) and potentially the shopping cart contents before checkout. This adheres to the rubric's preference for built-in state management.
- **API Interaction:**
    - **Library:** `axios` (recommended for its ease of use, promise-based API, and features like request/response interceptors for tasks like automatically attaching JWT tokens to requests).
    - **Usage:** Frontend components will use `axios` to make asynchronous calls to the backend REST API endpoints to fetch data (e.g., product lists, order history) and submit data (e.g., new orders, user registrations, product reviews).
- **Error Handling:**
    - User-friendly error messages will be displayed when API calls fail (e.g., network errors, server errors indicated by status codes).
    - Form validation errors will be displayed near the respective input fields.
    - Basic error boundaries might be implemented around major sections of the application to prevent a JavaScript error in one part from crashing the entire application.

Testing Strategy

To ensure code quality, functionality, and adherence to requirements, the following testing strategies will be employed:

- **Backend Testing:**
    - **Framework: Jest** (A popular JavaScript testing framework, often used with Node.js).
    - **Unit Tests:**
    - **Test Execution:** Tests will be runnable via the `npm test` script.
- **Frontend Testing:**
    - **Framework: Jest** and **React Testing Library**.
- **Manual Testing:**
    - Throughout the development, team members will continuously perform manual testing of the application's features in web browsers (Chrome, Firefox).
    - This includes testing all user flows defined in the FRD, checking UI responsiveness, form submissions, and overall user experience.