Inventory Management System

A Complete SDLC Process

➢ Planning Phase:

1. Objective and Scope Definition:

- **Objective:** To build a scalable and efficient inventory system that manages products, orders, quotations, suppliers, purchases, and customers with real-time stock updates and reporting.
- **Scope:**
  - Categories, Products, and Supplier Management.
  - Order Management and Customer Integration.
  - Quotation, Purchase, and Stock Control.
  - Reporting and Invoicing.
  - Authentication, Permissions, and Notifications.

2. Stakeholder Identification:

- **Primary Stakeholders:** University faculty (for grading), developers, project advisors.
- **End Users:** Employees (inventory handlers), admins (managers), and customers.
- **Technical Team:** Yourself (developer), possibly peers or mentors for code review.

3. Resource Allocation:

- **Technological Resources:**
  - Backend: Laravel (PHP) for server logic.
  - Frontend: Blade templating engine with possible additional packages like **Livewire**.
  - Database: MySQL for data storage.
- **Human Resources:**
  - Solo development effort, requiring time management and code review assistance if necessary.
- **Timeframe:** Based on the tasks provided earlier, allocate around **2-3 weeks** for project completion.

4. Risk Assessment:

- **Potential Risks:**
  - Feature creep (adding too many extra features, delaying submission).
  - Time management issues due to multiple project components.
  - Security risks (lack of proper role-based access, authentication flaws).

    - Dependency management in Laravel (e.g., outdated packages, compatibility issues).
- **Mitigation Strategies:**
    - Stick closely to predefined tasks, ensuring the core functionality is complete before adding extras.
    - Set weekly milestones for deliverables.
    - Conduct regular testing, especially for authentication and access control.
    - Maintain updated Laravel packages and dependencies.

5. Success Criteria and KPIs:

- **Success Criteria:**
    - All core modules (Products, Orders, Purchases, Quotations) are fully functional.
    - Role-based access and authentication system securely implemented.
    - UI/UX is user-friendly with error-free interactions.
    - Real-time stock management and notifications.
- **KPIs:**
    - 100% functionality coverage for all planned features.
    - Unit and functional testing for critical features.
    - Positive feedback from university reviewers and end-users.
    - No security breaches or unauthorized access.

6. Development Methodology:

- **Agile Approach:** Use an agile methodology to handle the various modules iteratively. Each sprint should focus on one major module (e.g., Categories and Products or Orders) with feedback integrated along the way.
- **Sprint Planning:**
    - Sprint 1: Authentication and User Management.
    - Sprint 2: Product, Category, Supplier Management.
    - Sprint 3: Orders, Customers, and Units.
    - Sprint 4: Purchases, Stock Management, Quotations.
    - Sprint 5: Testing, Debugging, and Final Refinements.

➢ Analysis Phase:

1. Functional Requirements:

- **Authentication:**
    - Users can log in, register, and manage their profiles securely.
    - Role-based access control (admin, employee).
- **Inventory Management:**
    - Admins can create, read, update, and delete categories, products, and suppliers.
    - Real-time stock updates upon product purchase or order.

- **Orders and Customers:**
  - Orders can be placed, invoices generated, and order details stored.
  - Integration with customer profiles and management of customer details.
- **Purchases and Stock:**
  - Admin with the request form Supplier can add purchases, and stock levels will adjust accordingly.
  - Low-stock notifications.
- **Quotations:**
  - Users can create quotations for customers with detailed product breakdowns.

2. Non-Functional Requirements:

- **Performance:**
  - The system must handle large data entries efficiently (e.g., hundreds of product entries).
  - All actions should have a minimal response time (< 1 second).
- **Security:**
  - Sensitive data (passwords, financial information) must be securely stored and encrypted.
  - Only authorized users should access sensitive sections (like creating purchases or orders).
- **Usability:**
  - The system should be easy to navigate with a clean user interface.
  - Users should get proper notifications on errors, warnings, or success actions.
- **Scalability:**
  - The system should be scalable to allow new modules or features (e.g., reporting, API integrations) in the future.
- **Reliability:**
  - System uptime should be 99.9%, and downtime, if any, should be minimal and scheduled.

3. Data Requirements:

- **Entities:**
  - **Products, Categories, Suppliers:** Products will belong to categories, and suppliers will provide stock.
  - **Orders and Purchases:** Each order/purchase will store details of products and quantities, and update stock accordingly.
  - **Customers and Users:** Orders will link to customer profiles, and users will manage various functions depending on role.
- **Data Integrity:**
  - Ensure correct associations between entities, such as valid customer IDs for each order.
  - Constraints should be applied to avoid invalid data entries (e.g., order for a product that doesn’t exist).

4. Process Flow and Use Case Analysis:

- **User Authentication Flow:**
  - User login -> Role-based access granted -> Redirect to dashboard (admin for full access, employees for limited features).
- **Order Process Flow:**
  - Customer selection -> Product selection -> Stock check -> Invoice generation.
- **Stock Purchase Flow:**
  - Select supplier -> Choose products -> Add stock quantity -> Update stock levels.
- **Low Stock Notifications Flow:**
  - Products with quantities below a predefined limit will trigger a notification to the admin.

5. Technical Feasibility:

- **Tools and Frameworks:**
  - The use of **Laravel** will streamline the MVC architecture.
  - **MySQL** will handle relational data, and **Blade** will manage the frontend with Laravel **Livewire** Package helpers for dynamic interactions.
- **Dependencies:**
  - Ensure all Laravel packages, including notifications, Enums, and cart management, are up-to-date and compatible.

6. Risk Analysis:

- **Potential Issues:**
  - **Database errors:** Incorrect database design could cause relational issues between products, orders, and suppliers.
  - **Authentication bugs:** Flaws in access control might expose sensitive areas to unauthorized users.
  - **Data loss:** Poor error handling can lead to data inconsistencies during orders or purchases.
- **Mitigations:**
  - Conduct database normalization to ensure proper structure.
  - Regular testing of role-based access to ensure compliance.
  - Implement proper error logging and reporting for data issues.

➢ Design Phase:

1. Architectural Design:

- **System Architecture:**
  - **3-Tier Architecture:** The project will follow a 3-tier architecture: Presentation (UI), Logic (Laravel Controllers/Models), and Data (MySQL).

- **MVC Framework:** Laravel's MVC pattern will organize the application into clear models, views, and controllers, ensuring separation of concerns.
- **Blade Templating Engine:** Blade will be used for the front-end templating, ensuring reusable components (such as forms and tables).

2. Database Design:

- **ERD (Entity Relationship Diagram):**
  - Main entities: **Users, Customers, Products, Orders, Purchases, Quotations, Suppliers.**
  - **Relationships:**
    - One-to-many: A customer can have multiple orders.
    - Many-to-many: Products can belong to multiple orders (via `OrderDetails`).
    - **Purchase Details** and **Quotation Details** will maintain relationships with products.
  - **Normalization:** Ensure at least 3NF normalization for avoiding redundancy (e.g., categories linked to products rather than being hardcoded).
- **Key Tables & Relationships:**
  - **Orders, Products, Order Details, Customers**: Establish foreign keys for linking customers to orders, and orders to order details/products.
  - **Suppliers and Products:** Relationship between the supplier and the stock they provide.

3. Interface Design:

- **UI Layout:**
  - **Dashboard:** A single-page dashboard with cards representing key metrics like total sales, orders, stock levels, and low-stock alerts.
  - **Forms and Tables:**
    - Clean forms for adding/editing products, categories, suppliers, etc.
    - Tables with filters and pagination for listing orders, products, purchases, etc.
- **Responsive Design:**
  - Ensure mobile compatibility using **Bootstrap** (or another front-end framework), enabling users to access the platform on various devices.
- **Interactive Components:**
  - **Livewire/JavaScript** for dynamic stock updates, real-time validations (e.g., disabling the "Create Invoice" button for out-of-stock products), and loading data asynchronously.

4. Component Design:

- **Controllers:**

    - **OrderController, ProductController, CustomerController:** Controllers will handle requests for orders, product management, and customer data, ensuring the proper response (e.g., displaying product details or processing orders).
    - **Notifications:** Controllers will handle the logic for sending notifications for low-stock products or successful orders.
- **Models:**
    - **Product, Order, Customer, Supplier:** These will be the primary models interacting with the database, defining relationships such as one-to-many (Orders -> Customers) and many-to-many (Orders -> Products).
- **Views:**
    - **Blade templates** will be utilized for reusability, with templates for navigation bars, product listings, order forms, etc.

5. Security Design:

- **Role-Based Access Control (RBAC):**
    - Different access rights for admin and regular users (e.g., only admins can manage stock).
- **CSRF Protection:**
    - Utilize Laravel's built-in CSRF protection for forms.
- **Data Encryption:**
    - Ensure user-sensitive information (e.g., passwords) is encrypted using Laravel's hashing mechanisms.

➢ Implementation Phase:

1. Code Development:

- **Controllers Implementation:**
    - Develop **Categories, Products, Suppliers, Orders, Purchases, and Quotations Controllers** using Laravel's artisan commands and controllers to handle CRUD operations. Ensure adherence to the design phase and proper usage of **Model-View-Controller (MVC)** architecture.
    - Implement logic for each feature. For example, **OrderController** should process order data and update stock information in the database.
    - Integrate **Livewire/JavaScript** for dynamic data updates, ensuring asynchronous loading for product stock updates and notifications.

2. Database Configuration:

- **Migrations and Seeders:**
    - Use Laravel's **migrations** for defining and creating the database schema, handling table creation for categories, products, suppliers, etc.

    - Use **seeding** to populate the database with initial test data (sample customers, products, and suppliers).
    - Test database relationships (one-to-many and many-to-many) during implementation using tinker or simple routes for data insertion.

3. User Interface Implementation:

- Develop **Blade templates** with components for:
    - **Orders, Products, and Customer Pages** using Bootstrap for a responsive UI.
    - Create forms and tables for data entry and management, with real-time feedback via **AJAX/JS** for form validation and **stock quantity validation** on the client-side.
- Integrate **JavaScript** logic to ensure that the "Create Invoice" button is disabled when the stock quantity is insufficient.

4. Authentication Implementation:

- Implement **Laravel Breeze** or **Jetstream** for the authentication system. Allow role-based access to users and admins, with restrictions on which users can perform specific actions (e.g., only admins can manage product stocks).
- Use middleware to enforce authentication and restrict access to certain routes.

5. Testing:

- **Unit Tests:** Write unit tests for models and controllers to ensure accurate business logic implementation. For example, ensure orders are properly calculated and stock updates occur as expected.
- **Browser Testing:** Use Laravel Dusk to automate browser-based tests, checking user interactions like adding products to orders, checking out, and making purchases.
- Perform manual testing to confirm that stock validation (disabling the button when overstocked) works as intended.

6. Version Control and Deployment:

- Use **Git** for version control, ensuring proper commit messages and code documentation for maintainability.
- Deploy the system on a local development server (e.g., **Laragon**) and later move to production (using services like **Heroku** or **DigitalOcean**).