e-commerce web application for shivshambhoeshop.in

Following a top-down approach. Let's break it down into a high-level design and schema

Contents

[**High-Level Design** 2](#_Toc189738935)

[Requirements Analysis 2](#_Toc189738936)

[Architecture Design 2](#_Toc189738937)

[Component Design 2](#_Toc189738938)

[**Database Schema** 3](#_Toc189738939)

[Users Table 3](#_Toc189738940)

[Products Table 3](#_Toc189738941)

[Categories Table 3](#_Toc189738942)

[Orders Table 3](#_Toc189738943)

[OrderItems Table 3](#_Toc189738944)

[ShoppingCart Table 4](#_Toc189738945)

[CartItems Table 4](#_Toc189738946)

[**High-Level Design Diagram** 4](#_Toc189738947)

[User Interface Layer 4](#_Toc189738948)

[Application Layer 4](#_Toc189738949)

[Data Layer 4](#_Toc189738950)

[API Layer 4](#_Toc189738951)

[**Steps to Implement** 4](#_Toc189738952)

[Set Up the Project 4](#_Toc189738953)

[Design the Database 5](#_Toc189738954)

[Implement the Business Logic 5](#_Toc189738955)

[Build the User Interface 5](#_Toc189738956)

[Expose REST APIs 5](#_Toc189738957)

[Testing and Deployment 5](#_Toc189738958)

[**Scalability Considerations** 5](#_Toc189738959)

[Dependency Injection 5](#_Toc189738960)

[REST APIs 5](#_Toc189738961)

[Database Optimization 5](#_Toc189738962)

[Load Balancing 6](#_Toc189738963)

[Caching 6](#_Toc189738964)

[Microservices Architecture 6](#_Toc189738965)

# **High-Level Design**

## Requirements Analysis

* + **Functional Requirements**: User registration and login, product catalog, shopping cart, order processing, payment gateway integration, order tracking, admin panel for inventory management.
* **Non-Functional Requirements**: Scalability, security, performance, maintainability.

## Architecture Design

* + **Presentation Layer**: ASP.NET Core MVC for the web interface.
  + **Business Logic Layer**: .NET 8 services to handle business rules and logic.
  + **Data Access Layer**: Entity Framework Core for database interactions.
  + **Database**: SQL Server for storing data.
  + **Hosting**: On-premise server with IIS (Internet Information Services).
  + **Dependency Injection:** Use built-in DI in ASP.NET Core to manage service lifetimes and dependencies.
  + **REST APIs:** Expose business logic through RESTful APIs for better scalability and integration.

## Component Design

* + **User Interface**: HTML, CSS, JavaScript, and Razor views.
  + **Controllers**: Handle HTTP requests and responses.
  + **Services**: Business logic implementation.
  + **Repositories**: Data access logic.
  + **Models**: Data transfer objects and entity models.

# **Database Schema**

## Users Table

* + UserId (Primary Key)
  + Username
  + PasswordHash
  + Email
  + Role (e.g., Admin, Customer)

## Products Table

* + ProductId (Primary Key)
  + ProductName
  + Description
  + Price
  + StockQuantity
  + CategoryId (Foreign Key)

## Categories Table

* + CategoryId (Primary Key)
  + CategoryName

## Orders Table

* + OrderId (Primary Key)
  + UserId (Foreign Key)
  + OrderDate
  + TotalAmount
  + OrderStatus

## OrderItems Table

* + OrderItemId (Primary Key)
  + OrderId (Foreign Key)
  + ProductId (Foreign Key)
  + Quantity
  + UnitPrice

## ShoppingCart Table

* + CartId (Primary Key)
  + UserId (Foreign Key)
  + CreatedDate

## CartItems Table

* + CartItemId (Primary Key)
  + CartId (Foreign Key)
  + ProductId (Foreign Key)
  + Quantity

# **High-Level Design Diagram**

## User Interface Layer

* + Web Pages (HTML, CSS, JavaScript)
  + Razor Views

## Application Layer

* + Controllers
  + Services
  + Repositories

## Data Layer

* + Entity Framework Core
  + SQL Server

## API Layer

* + RESTful APIs for business logic

# **Steps to Implement**

## Set Up the Project

* + Create a new ASP.NET Core MVC project.
  + Set up Entity Framework Core with SQL Server.

## Design the Database

* + Define the database schema using Entity Framework Core models.
  + Create migrations and update the database.

## Implement the Business Logic

* + Create services for handling business logic.
  + Implement repositories for data access.

## Build the User Interface

* + Design the web pages using Razor views.
  + Implement controllers to handle user interactions.

## Expose REST APIs

* + Create RESTful APIs for key business logic operations.
  + Use controllers to handle API requests and responses.

## Testing and Deployment

* + Write unit tests for the business logic.
  + Deploy the application on an on-premise server with IIS.

# **Scalability Considerations**

## Dependency Injection

* + Use ASP.NET Core's built-in DI to manage service lifetimes (transient, scoped, singleton).
  + Ensure services and repositories are injected where needed to promote loose coupling and testability.

## REST APIs

* + Expose business logic through RESTful APIs to allow for easy integration with other systems and services.
  + Design APIs to be stateless and scalable.

## Database Optimization

* + Use indexing and query optimization techniques.
  + Consider database sharding or partitioning if needed.

## Load Balancing

* + Use load balancers to distribute traffic across multiple servers.

## Caching

* Implement caching strategies to reduce database load.

## Microservices Architecture

* Consider breaking down the application into microservices for better scalability and maintainability.