Proposal for Labs and Projects

Contents

Online Shopping Application: Scenario and Requirements

Scenario: "ShopSphere" E-Commerce Platform

ShopSphere is a new, feature-rich online retail platform designed to offer a seamless shopping experience for customers and efficient management tools for administrators. It must be highly available, scalable, and secure, especially concerning financial transactions and user data. The platform needs to support a large and growing number of users, products, and simultaneous transactions.

Key Requirements

Type	Requirement Description	Architectural Driver
Functional	**User Management:** Registration, Login, Profile Management, Order History.	Security, User Interface.
Functional	**Product Catalog:** Browse by category, Search, View product details (description, price, stock).	Performance, Data Management.
Functional	**Shopping Cart:** Add, remove, update quantities, persist cart state across sessions.	State Management, Availability.

Type	Requirement Description	Architectural Driver
Functional	**Checkout & Payment:** Multi-step checkout, secure payment processing (e.g., credit card, PayPal), order confirmation.	Security, Reliability.
Functional	**Order Processing (Admin):** View, update status (Pending, Shipped, Delivered), cancellation.	Transactional Integrity.
Non-Functional	**Performance:** Response time for critical operations (e.g., product search, checkout) must be under **2 seconds**.	Scalability, Load Balancing.
Non-Functional	**Security:** Encrypt sensitive user data (passwords, payment info), secure authentication (e.g., OAuth 2.0).	Data Protection, Authentication.
Non-Functional	**Scalability:** System must be able to scale to support **10,000 concurrent users** and a growing product catalog.	Distribution, Component Isolation.
Non-Functional	**Availability: 99.9% uptime** is required for the shopping and checkout features.	Redundancy, Fault Tolerance.

Architectural Types (for Comparison and Lab Use)

The online shopping system is an excellent case study for comparing different architectural approaches.

1. **Layered Architecture (Monolithic):**
    - **Description:** The system is divided into horizontal layers: **Presentation (UI)**, **Business Logic**, **Persistence (Data Access)**, and **Database**. Each layer interacts only with the layer immediately below it.
    - **Use Case:** Ideal for **initial development** or smaller e-commerce sites where simplicity of deployment and management is prioritized.
    - **Advantage:** Simple to develop, test, and deploy as a single unit.
2. **Microservices Architecture:**
    - **Description:** The application is broken down into a collection of small, independent services, each running in its own process and communicating via lightweight mechanisms (like HTTP APIs or message queues). Key services: **User**

Service, **Product Catalog Service**, **Shopping Cart Service**, **Order Service**, **Payment Service**. An **API Gateway** handles all client requests.

  - **Use Case:** Required for **high-scale, high-availability** e-commerce platforms like Amazon, offering independent deployment, technology diversity, and fault isolation.
  - **Advantage:** High scalability, fault isolation, and faster time-to-market for new features.

3. **Event-Driven Architecture (EDA):**
   - **Description:** Components communicate primarily through events, often using a message broker (e.g., Kafka, RabbitMQ). The state changes (events) are published by one service (the producer) and consumed by others (the consumers) interested in that change.
   - **Use Case:** Excellent for **decoupling transactional and analytical workloads**, such as updating inventory and sending a notification email *after* an order is placed.
   - **Advantage:** Real-time responsiveness, high decoupling, and easier integration of new consumer services.

Software Architecture Course: 8 Lab Practices

The following 8 labs use the **ShopSphere** scenario, progressing from requirements analysis to a partial implementation comparing **Layered** and **Microservices** architectures.

Lab	Title	Architecture Focus	Activity & Step-by-Step Instructions
Lab 1	**Requirements Elicitation & Modeling (Use Case View)**	N/A (Pre-Architecture)	**Activity:** Define Functional/Non-Functional Requirements. Draw UML Use Case Diagrams for the **"Web Customer"** and **"Admin"** actors. **Steps:** 1. Identify all major actors. 2. Define top-level use cases (e.g., Make Purchase, Manage Products). 3. Detail a critical use case (e.g., "Checkout") with pre/post-conditions and a basic flow of events. 4. Identify three ASRs (Architecturally Significant Requirements).

Lab	Title	Architecture Focus	Activity & Step-by-Step Instructions
Lab 2	**Layered Architecture Design (Logical View)**	Layered	**Activity:** Design the 4-layer architecture. **Steps:** 1. Define the four layers: Presentation, Business Logic, Persistence, Database. 2. Specify the core components in each layer (e.g., ProductController in Presentation, OrderManager in Business Logic, SQLProductRepo in Persistence). 3. Draw a component diagram showing layer dependencies (strict downward flow).
Lab 3	**Layered Architecture Implementation (CRUD)**	Layered	**Activity:** Implement the **Product Management** feature using the Layered pattern. **Steps:** 1. Set up a basic project structure (e.g., Java/Spring or Python/Flask) with packages for each layer. 2. Implement the Product entity. 3. Code the **Persistence Layer** to connect to a dummy/in-memory database (Create, Read, Update, Delete methods). 4. Implement the **Business Logic** layer for product validation. 5. Implement the **Presentation Layer** (Controller) to expose a REST endpoint (/products).
Lab 4	**Microservices Decomposition & Communication**	Microservices	**Activity:** Decompose the monolith and define service contracts. **Steps:** 1. Identify five core services (**User, Product, Cart, Order, Payment**). 2. Define the key API endpoints (Service Contract) for **Product Service** and **Shopping Cart Service**. 3. Draw a C4 Model (Level 1: System Context) showing the system and its external dependencies (Payment Gateway, Email Service).
Lab 5	**Implementing the Product Microservice**	Microservices	**Activity:** Build an independent Product Microservice. **Steps:** 1. Create a new, standalone service project (different port/repository). 2. Implement the **Product**

Lab	Title	Architecture Focus	Activity & Step-by-Step Instructions
			Service (Catalog and Inventory management logic). 3. Expose the REST API for reading and searching products (GET /api/products). 4. Test the service in isolation using a tool like Postman.
Lab 6	**Introducing the API Gateway Pattern**	Microservices	**Activity:** Implement a simple API Gateway. **Steps:** 1. Use a framework (e.g., Spring Cloud Gateway, Nginx) or write a basic reverse proxy. 2. Configure the gateway to route /api/products requests to the **Product Service** (from Lab 5). 3. Implement a single security check (e.g., token validation stub) on the gateway before forwarding the request.
Lab 7	**Event-Driven Architecture (EDA) & Integration**	Event-Driven	**Activity:** Implement asynchronous communication using events. **Steps:** 1. Set up a local message broker (e.g., RabbitMQ or Kafka dummy setup). 2. Modify the **Order Service** to **publish** an OrderPlacedEvent upon successful checkout. 3. Create a new **Notification Service** that **subscribes** to the OrderPlacedEvent and prints a confirmation message (simulating email). 4. Demonstrate the decoupled nature of the services.
Lab 8	**Deployment View & Quality Attribute Analysis (ATAM)**	Mixed/Deployment	**Activity:** Design the system deployment and evaluate Non-Functional Requirements. **Steps:** 1. Draw a UML Deployment Diagram showing the Microservices on containers/VMs, the API Gateway, and a load balancer. 2. Choose three Non-Functional Requirements (e.g., Scalability, Security, Performance). 3. Conduct a simplified Architecture Trade-off Analysis Method (ATAM) to discuss how the **Microservices** and **Layered**

Lab	Title	Architecture Focus	Activity & Step-by-Step Instructions
			architectures handle one chosen requirement (e.g., Scalability) differently.

Other Course Projects

Here are four other full defined scenarios for course projects, ranging from beginning to medium complexity, including instructional and coding steps.

Project 1: To-Do List Application (Beginner)

- **Scenario:** A simple personal productivity tool for managing daily tasks. Users can add, view, mark as complete, and delete tasks.
- **Architectural Focus: Client-Server Architecture** (1-Tier, all on a single machine/browser initially).
- **Instructional Steps:**
    1. **Define Model:** Create a Task class (or structure) with id, description, and is_completed.
    2. **Develop Interface:** Design a simple web form or console interface for input.
    3. **Implement CRUD:** Write functions to **C**reate, **R**ead (display all), **U**pdate (mark complete), and **D**elete tasks from an in-memory list (or a simple JSON file).
- **Coding Steps (Python/Flask Example):**
    - **Data Structure:** Use a Python list to store dictionaries representing tasks: tasks = [{'id': 1, 'desc': 'Buy groceries', 'complete': False}].
    - **Add Function:** Define a function add_task(description) that generates a new ID and appends the new task to the list.
    - **Display:** Define a function that loops through the list and prints/renders each task.

Project 2: Simple Student Grade Management System (Beginner-Medium)

- **Scenario:** A system for a small department to manage student enrollment, course records, and assign/view grades. An Admin user can add students and courses, and faculty can assign grades.

- **Architectural Focus: Layered Architecture (2-Tier)**: **Client (Web UI)** and **Server (Business Logic + Persistence in a file/DB)**.
- **Instructional Steps:**
    1. **Define Models:** Create Student, Course, and Enrollment (linking students, courses, and storing the grade).
    2. **Database Design:** Sketch a simple relational database schema (3 tables) and primary/foreign keys.
    3. **Implement Persistence:** Use a file (CSV/JSON) or a simple SQLite database to store the data.
    4. **Implement Business Logic:** Develop functions for calculating a student's GPA based on their enrolled courses.
- **Coding Steps (SQL/Python Example):**
    - **SQL Schema:** CREATE TABLE Students (id INT PRIMARY KEY, name TEXT); CREATE TABLE Courses (code TEXT PRIMARY KEY, name TEXT); CREATE TABLE Enrollments (student_id INT, course_code TEXT, grade REAL, FOREIGN KEY...);
    - **GPA Function:** Write a function calculate_gpa(student_id) that executes an SQL query (or file reading logic) to retrieve all grades for a student and calculates the weighted average.

Project 3: URL Shortener Service (Medium)

- **Scenario:** A service like Bitly or tinyurl.com that takes a long URL and generates a short, unique code (e.g., bit.ly/AbCde12). When the short link is accessed, the user is redirected to the original long URL. Must track click count.
- **Architectural Focus: Client-Server Architecture (3-Tier/N-Tier)**: **Client**, **Web Server/Application Logic**, **Database**. Focus on high read performance.
- **Instructional Steps:**
    1. **Core Feature:** Implement the URL shortener logic (generating a unique, short, alphanumeric key).
    2. **Redirection:** Configure the web server/application to handle the short code as a route and look up the long URL.
    3. **Analytics:** Add a column to the database to track the number of times the short link has been clicked.

    4. **Security/Input:** Implement basic URL validation to prevent non-HTTP/HTTPS URLs.

- **Coding Steps (Any Backend Framework Example):**
    - **Key Generation:** Use a function like base62_encode or a simple hash to generate the unique short code from an incrementing ID.
    - **Database:** CREATE TABLE urls (id INT PRIMARY KEY, short_code VARCHAR(10) UNIQUE, long_url TEXT, click_count INT DEFAULT 0);
    - **Redirection Route:** Define a route (e.g., /u/<short_code>) that looks up the long_url, increments click_count, and returns a **301/302 HTTP Redirect** response.

Project 4: Real-Time Chat Application (Medium)

- **Scenario:** A simple multi-user chat application where users can join a room and send messages to all other users in that room. Messages must appear instantly without page refresh.
- **Architectural Focus: Event-Driven Architecture (using WebSockets)**. Messages are events pushed from the server to all connected clients.
- **Instructional Steps:**
    1. **Set up WebSocket Server:** Configure a backend to support the WebSocket protocol (e.g., Socket.io, Flask-SocketIO, or raw WebSockets).
    2. **Client Connection:** Implement a basic HTML/JavaScript client to open a WebSocket connection.
    3. **Message Broadcasting:** The server must receive a message from one client and **broadcast** it to all other active clients in the room.
    4. **Persistence (Optional):** Add persistence to save the last 10 messages to a database.
- **Coding Steps (Node.js/Socket.io Example):**
    - **Server Logic:** On receiving a message event: io.on('connection', (socket) => { socket.on('chat message', (msg) => { io.emit('chat message', msg); }); }); (The io.emit broadcasts to everyone).
    - **Client Logic (JS):** Use const socket = io(); to connect and socket.on('chat message', function(msg) { // append message to screen }); to receive and display.