Building the Monolith: Software Software Architecture Lecture 2 2

The Layered Architecture Pattern (Logical View)

Welcome to Lecture 2 of our Software Architecture series. Today, we delve into the foundational concept of the Layered Architecture Pattern, focusing specifically on its logical view. Our journey begins with understanding how to design the internal structure of a robust, yet flexible, monolithic application, using the familiar example of ShopSphere.

This lecture will equip you with the knowledge to structure your applications for maintainability, testability, and clarity, laying the groundwork for more complex architectural discussions in subsequent lectures.

Recap: The Architectural Drivers

Last Week's Focus

In our previous session, we laid the groundwork by defining critical Architectural Significant Requirements (ASRs) that guide our design choices:

- **Scalability:** The ability of the system to handle a growing amount of work by adding resources.
- **Fault Isolation:** Ensuring that a failure in one part of the system does not cause the entire system to crash.
- **Security:** Protecting the system from unauthorized access and data breaches.

We also explored the Use Case View, focusing on a core business process: "Make Purchase." This gave us a clear functional goal to build towards.

Our Current Challenge

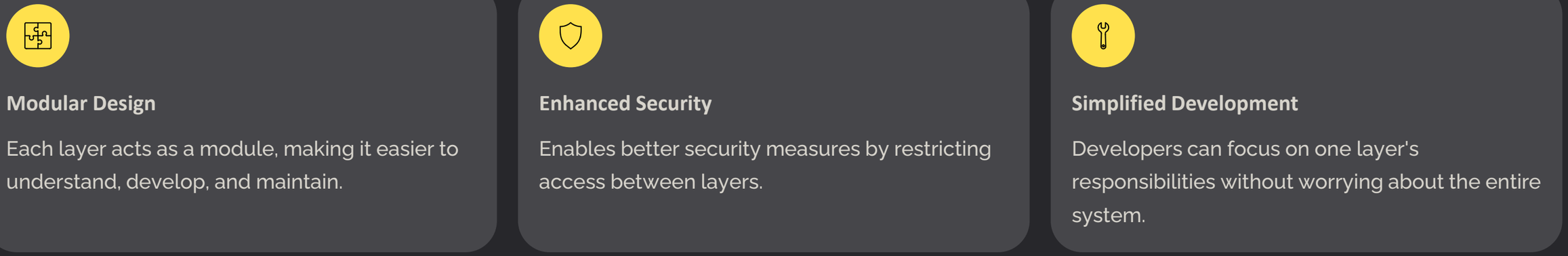
Knowledge Base: The Layered Pattern

The Layered Architecture is a fundamental architectural pattern where components are systematically organized into horizontal layers. Each layer is designed to perform a specific, well-defined role within the system, creating a clear division of responsibilities.

> **Analogy: Building Floors in an Office Building**Think of a multi-story office building. Each floor has a distinct function: the ground floor might be a lobby for public access, the middle floors for general operations, and the top floor for executive offices. There are strict rules about how access is managed between floors; for instance, you wouldn't expect a visitor in the lobby to directly interact with executive decisions without going through proper channels.

Key Advantage: Separation of Concerns (SoC)

The primary benefit of the Layered Architecture is its inherent **Separation of Concerns (SoC)**. Components within a specific layer are solely responsible for their specialized tasks and are oblivious to the internal workings of other layers, especially those above them. For example, a component in the data layer (like a database connector) should know nothing about how a web page is rendered or what specific UI elements are being used.



The Crucial Rule: Strict Downward Dependency

The Constraint: One-Way Interaction

The cornerstone of a well-designed Layered Architecture is the principle of strict downward dependency. This constraint dictates that a component in layer N can **only** call services or interact with components in layer N-1 (the layer immediately below it). It is a one-way street, preventing any "shortcuts" or direct interactions with higher layers.

> **The Prohibited Action:** A component residing in Layer 3 (Persistence) must **never** directly call a component in Layer 1 (Presentation). This would break the fundamental contract of the layered architecture and lead to a tightly coupled, unmanageable system.

Why Strictness Matters: Benefits for Maintainability and Testability

Adhering strictly to this rule provides significant long-term benefits for your software project:

Increased Maintainability

When changes are made in one layer, you can be confident that those changes will not unintentionally impact or break functionality in layers above it. This significantly reduces the ripple effect of modifications.

Improved Testability

Each layer can be tested in relative isolation. You can mock or simulate the behavior of the layer below it, allowing for focused unit and integration testing without needing the entire application to be functional. For instance, the Presentation layer can be tested independently by mocking the Business Logic layer's responses.

Reduced Coupling

Strict downward dependency minimizes coupling between layers, meaning components are less dependent on the internal implementation details of other layers. This makes the system more flexible and adaptable to future changes or technology upgrades.

The Four Standard Layers of a Monolith

While the exact number and naming of layers can vary, most monolithic applications built with a layered pattern typically adhere to four core layers:

1. Presentation (UI/Web)	Handles user requests (HTTP/APIs), authentication, UI rendering, and user interaction logic.	`ProductController`: Responsible for receiving HTTP requests for product data, invoking business logic, and returning the appropriate HTTP response (e.g., JSON or rendered HTML).
2. Business Logic (Service/Domain)	Executes core business rules, performs data validation, manages transactions, and orchestrates workflows specific to the application's domain.	`ProductService`: Contains the logic for retrieving, creating, updating, and deleting products, including validation rules and ensuring data consistency.
3. Persistence (Data Access)	Responsible for mapping application objects to database records (e.g., using an ORM or DAO pattern), handling data storage and retrieval operations, and executing CRUD commands.	`ProductRepository`: Interacts directly with the database to save, fetch, and query `Product` objects, abstracting database-specific details from the business logic.
4. Data (Database)	The actual physical storage system where the application's data resides. This includes relational databases, NoSQL databases, file systems, etc.	`products Table` in a PostgreSQL database: Stores all product-related information such as ID, name, description, price, and stock levels.

Understanding these distinct responsibilities is crucial for building a well-structured and maintainable application.

Knowledge Base: Request Flow in Layered Architecture

Process View: Visualizing Data Movement

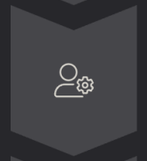
To truly understand the Layered Architecture, it's essential to visualize how a typical request or data interaction flows through these distinct layers. This "Process View" illustrates the sequential communication and dependencies.

Scenario: Customer Clicks "View Product Details"

Let's consider a common scenario in our ShopSphere application: a customer wants to view the details of a specific product.

1. Browser to Presentation

The customer's browser sends an HTTP request (e.g., `GET /products/{id}`) to the Presentation layer of our application.



2. Presentation to Business Logic

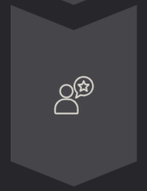
The Presentation layer (e.g., `ProductController`) receives the request, performs initial parsing, and then calls a method in the Business Logic layer (e.g., `ProductService.getProductDetails(id)`).

3. Business Logic to Persistence

The Business Logic layer, after applying any necessary business rules, calls a method in the Persistence layer (e.g., `ProductRepository.findById(id)`) to retrieve the product data.

4. Persistence to Data

The Persistence layer translates the request into a database-specific query (e.g., SQL `SELECT * FROM products WHERE id = ?`) and executes it against the Data layer (the database).



5. Data to Browser (Response)

The database returns the record to the Persistence layer, which maps it back to an object. This object then travels back up through the Business Logic and Presentation layers, eventually being rendered as an HTTP response (e.g., JSON or HTML) back to the customer's browser.

Activity 1: Defining Layer Responsibilities

Now, let's put our knowledge into practice. For the ShopSphere Product Catalog feature, we need to formalize the role of each layer. This activity will help you deeply understand the boundaries and interactions between the Presentation, Business Logic, and Persistence layers.

Goal

To define the primary function, expected input, and anticipated output for the top three layers involved in our Product Catalog feature. This exercise reinforces the concept of Separation of Concerns.

Presentation	Process API requests, manage user interaction, and render responses.	HTTP Request (JSON/Form Data), User Actions	HTTP Response (JSON/HTML), UI Updates
Business Logic	Enforce business rules, validate data, and orchestrate domain operations.	Data Model Object (e.g., `Product`), Primitive Data Types	Validated Data Model Object, Operation Results
Persistence	Map application objects to database records, handle CRUD operations, and manage data integrity.	SQL/ORM Command, Data Model Object	Database Record/Tuple, Hydrated Data Model Object

This structured approach ensures that each layer is self-contained and has a clear contract with its adjacent layers, fostering a more robust and scalable architecture.

Knowledge Base: The Logical View (UML Component)

Purpose: Static System Structure

The Logical View in software architecture is concerned with the static structure of the system. It helps us understand the significant components, their responsibilities, and how they interact without diving into implementation details. The primary artifact for depicting the logical view is the **UML Component Diagram**.

What is a Component?

In the context of UML, a component is:

> "A self-contained, replaceable, and encapsulated part of a system that provides services through its interfaces."

Think of a component as a well-defined building block in your software. It could be a class, a module, a microservice, or even a subsystem. The key characteristics are its clear boundaries and the services it offers to others, and the services it requires from others.

Self-Contained

A component manages its own internal state and logic, independent of external factors.

Replaceable

Due to its well-defined interfaces, a component can be swapped out for another compliant component without affecting the rest of the system.

Encapsulated

Its internal implementation details are hidden, exposing only its public interfaces.

Provides Services

A component offers specific functionalities that other parts of the system can utilize.

Component Modeling: Interfaces

Interfaces are fundamental to component-based design as they define the contract between components. They clearly specify **what** services a component offers and **what** services it requires to function, without revealing **how** those services are implemented.

Provided Interface (Lollipop)

Represented by a circle (lollipop), a provided interface signifies a set of services that a component offers to its environment. These are the public methods or functions that other components can call. It's the "what I can do for you" aspect of a component.

- Example: A `ProductService` component provides an `IProductService` interface with a method like `getProductDetails(id)`.

Required Interface (Socket)

Represented by a semi-circle (socket), a required interface indicates a set of services that a component needs from another component to perform its own functions. It's the "what I need from you" aspect.

- Example: A `ProductController` component requires an `IProductService` interface to fulfill requests related to product details.

The Connection

The magic happens when a component's required interface (socket) plugs into another component's provided interface (lollipop). This visual representation clearly shows a dependency: the consumer component relies on the producer component to perform a specific task, adhering to the contract defined by the interface. This mechanism decouples the components, allowing them to evolve independently as long as the interface contract is maintained.

Activity 2: Component Identification (Product Catalog)

Goal

To translate the conceptual layers into concrete, identifiable components for our ShopSphere Product Catalog feature. This involves naming specific components within each layer and outlining their primary roles and provided interfaces.

Presentation	`ProductController`	Receives HTTP GET requests for `/products/{id}` and orchestrates the response.
Business Logic	`ProductService`	Provides: `IProductService` (e.g., method: `getProductDetails(id)` to retrieve product data and apply business rules).
Persistence	`ProductRepository`	Provides: `IProductRepository` (e.g., method: `findById(id)` to fetch product data from the database).

Proof of Concept: Component Dependencies

- The `ProductController` **requires** the services of the `ProductService` to process product-related requests.
- The `ProductService` **requires** the services of the `ProductRepository` to interact with the underlying data store.

This explicit chaining of required interfaces enforces our strict downward dependency rule. It means that the `ProductController` is only aware of the `ProductService`'s contract, not its internal implementation, and similarly for the `ProductService` with the `ProductRepository`. This design promotes maintainability, testability, and flexibility in our architecture.