Lecture 4

Spring Model-View-Controller (MVC) Flow and DispatcherServlet

- The Spring MVC framework organizes web application components into three interconnected parts: Model, View, and Controller.

  1. **Controller:** Handles user actions and incoming URLs. The **@Controller** annotation marks a class as a web handler, and **@RequestMapping("/pet")** maps a specific URL to a method.

  2. **Model:** Carries data, typically as key–value pairs, between the Controller and the View.

  3. **View (JSP):** Renders the data provided by the Model, often utilizing Expression Language (EL). A return value like `"pet/form"` from the controller typically resolves to a physical location such as `/WEB-INF/views/pet/form.jsp`.

- The **DispatcherServlet** functions as the "front controller" or the "receptionist at the front desk" of the web application. Every HTTP request must first go to the DispatcherServlet, not directly to a controller. Spring Boot automatically creates and configures the DispatcherServlet, so developers do not write it themselves.

- The DispatcherServlet performs several key steps upon receiving a request (e.g., `/pet`):

  1. It receives the request.

  2. It asks the **HandlerMapping** which method should handle the request.

  3. It calls the identified controller method.

  4. When the controller returns a logical view name (e.g., `"pet/form"`), it asks the **ViewResolver** where to find the corresponding view file.

  5. It merges the Model data with the View (JSP), produces HTML, and sends the final response back to the browser. The DispatcherServlet is essential—it is the traffic controller or post office for every web request.

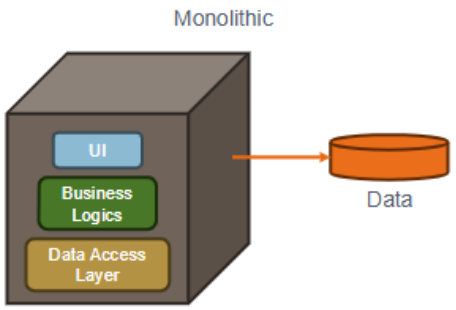
Classic Software Architecture Styles

- The concept of software architecture is derived from principles used in building and construction. Notably, there is **no globally agreed standard** for any architecture, and the same concepts may be referred to by different names.

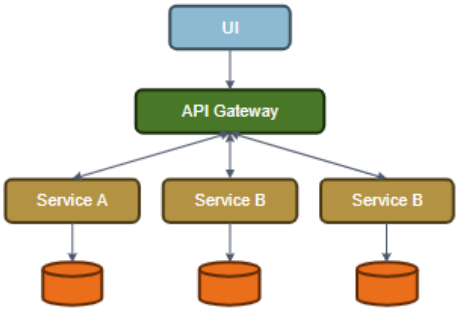
Architectural styles discussed include:

- **Layered (n-tiered) Architecture:** This style organizes components into distinct layers. A common example includes
  - `Layer 1: **Presentation Layer** (UI, customer screens),
  - Layer 2: **Business Layer** (Customer Object, Gamer Interactions),
  - .....
  - Layer N: **Persistence Layer** (DAOs, ORM, Transaction), and the **Database Layer** (SQL, Cloud).
- This architecture is rarely used entirely on its own but is often combined with other patterns. Examples of use include University Intranets, Basic Content Management Systems (CMS), and internal organizational websites.
- **Monolithic vs. Microservices:**

  ◦ A **Monolithic** structure wraps and compiles the UI, Business Logics, Data Access Layer, and Data all together. It is often the best choice for many startups, and the modular-monolithic structure is highly popular.



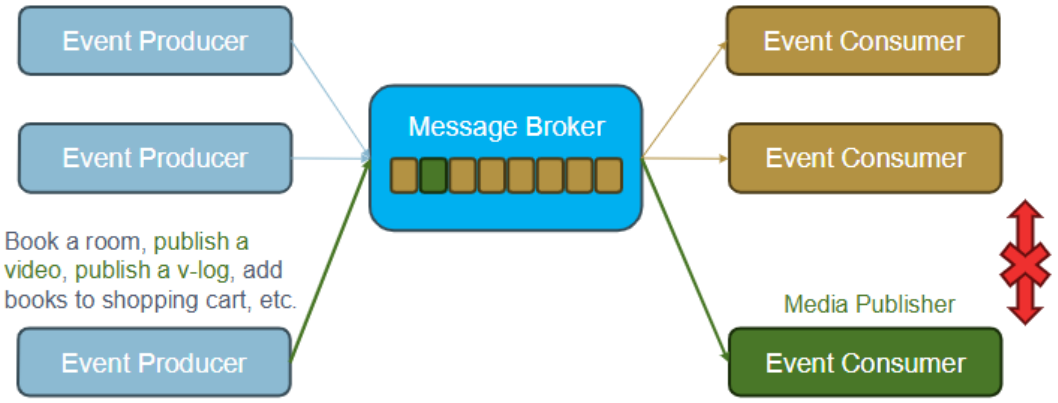
  ◦ **Microservices** break the application into independent services (Service A, Service B), communicating through an API Gateway, with each service often maintaining its own database. Microservices are known for being mighty, powerful, scalable, and secure, though they can be expensive.

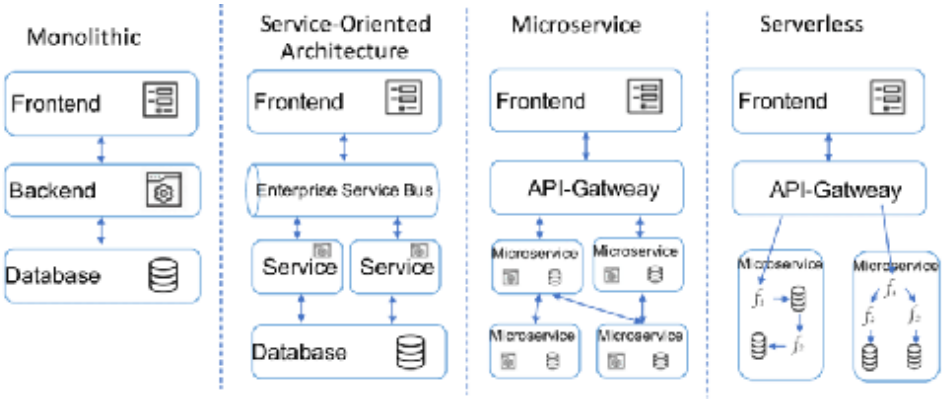


Microservices

- API Gateway
- Each service has it is own database

- **Event-driven Architecture (EDA):** In EDA, **Event Producers** (e.g., booking a room, adding items to a cart) publish events, which are routed by a **Message Broker** and delivered to **Event Consumers**.
  - EDA is characterized by **loose coupling** among event consumers.
  - It offers high **scalability** (services scale independently), strong **decoupling** (components don't need to know about each other), and enables **real-time processing**, which is crucial for applications like IoT or trading. It also provides **fault tolerance** as systems can queue events for later processing.





Communication Styles: Synchronous vs. Asynchronous

The choice of communication style significantly impacts architecture, especially when comparing Microservices and EDA.

Aspect	Microservices	Event-Driven Architecture (EDA)
Communication Style	Synchronous (mainly)	Asynchronous
Coupling	Loosely coupled, but can have dependencies	**Highly decoupled**
Fault Tolerance	Requires circuit breakers/retries	Natural resilience with asynchronous queuing
Common Technologies	Spring Boot, Docker, Kubernetes	Kafka, RabbitMQ, AWS SQS/SNS

Synchronous Communication

- The sender waits for the receiver to process the request and return a response; it is a **blocking interaction** (e.g., a phone call).
- Used in HTTP/REST APIs and database queries (SQL over JDBC).
- Pros: Simple to design, easy error handling (immediate feedback), and predictable order of execution.
- Cons: Can cause delays and bottlenecks if one service is slow, and it leads to **tight coupling** since both systems must be online and responsive simultaneously.

Asynchronous Communication

- The sender sends a message and continues without waiting for a response; it is **non-blocking** (e.g., sending an email).
- Used in Message Queues (Kafka, RabbitMQ) and Event-driven architectures.
- Pros: Leads to **decoupled systems** (sender and receiver do not need to be active simultaneously), offers higher scalability and resilience, and better performance under load.
- Cons: More complex to implement (requires callbacks and message tracking), harder debugging, and often results in **eventual consistency** instead of immediate consistency.

Spring Forms and Validation

Expression Language (EL) and JSTL

- Expression Language (EL) is a mechanism that allows the presentation layer (web pages) to communicate with the application logic (managed beans/classes in the Model). It is used by both JavaServer Faces and JavaServer Pages (JSP) technology.

  Why Use EL:

  - It simplifies communication and provides **Direct Access** to data fields using straightforward syntax like `${user.name}`.
  - It promotes **Cleaner and More Readable Code** by eliminating the need for embedded Java code (`<% %>`) in JSPs.
  - It enhances readability and promotes best practices by encouraging the separation of presentation and business logic.
- **JSTL** (JSP Standard Tag Library) provides XML-like tags (like `<c:forEach>`), which are typically used alongside JSP. The JSTL `c:forEach` loop iterates over a collection defined in the Controller (e.g., `items="${goals}"`), using the EL syntax `${...}` to access the objects.

Spring MVC Forms

- Spring MVC includes features to simplify handling HTML form data. It integrates with the framework to automatically **bind form data to Java objects** (Data Binding).

The **Form Tag Library** provides key tags:

- **<form:form>:** Initiates a Spring form and maps it to a Java object using the `modelAttribute` attribute (e.g., `modelAttribute="user"`).
- **<form:input>** or **<form:select>:** Binds directly to an object field using the `path` attribute (e.g., `path="username"`).
- **<form:errors>:** Displays validation errors next to the corresponding form field.

Spring Form Validation

- Validation in Spring often uses the dedicated **Validator interface**.
  - The `Validator` interface requires two main methods:

  1. **supports(Class<?> clazz):** Checks if the Validator is able to validate instances of the given class.

  2. **validate(Object target, Errors errors):** Executes the validation logic on the `target` object and populates the `errors` object with any findings.

- The **BindingResult** object holds the outcome of the binding and validation process. It is crucial for checking and handling errors in the submitted form data. Key methods for error checking include `hasErrors()`, `hasFieldErrors()`, and `getFieldErrors()`. Developers can manually add errors using `rejectValue()`.

Steps for Validation

1. **Setup the Model:** Define the data structure and use **Validation Annotations** (like `@NotBlank`, `@Size`, `@Min`) on the fields that require checks.

2. **Configure the Controller:** Use the **@Valid** annotation on the model attribute within the controller method to trigger validation. The `BindingResult` object must immediately follow the `@Valid` model attribute in the method signature to capture the validation errors.

3. **Create the JSP Form:** Use the `<form:errors>` tag to display field-specific validation messages and the `<c:if test="${not empty bindingResult}">` tag to display global error messages if errors are present.