**Introduction to Spring Framework**

* Spring MVC: mô hình kiến trúc cho phép phát triển các ứng dụng web theo kiểu MVC
* Spring boot: Là dự án con của spring fw, giúp dễ dàng phát triển và triển khai các ứng dụng Spring. Cung cấp các cách cấu hình tự động và starter dependencies giúp giảm thiểu sự phức tạp khi thiết lập dự án
* **DI, IoC:**
* **Dependency Injection(DI):** bản chất của DI là giấu đi cách triển khai của class như nào, chỉ tập chung vào hành vi của nó. Một class có thể được khởi tạo khác nhau phụ thuộc vào context của chương trình theo config hoặc runtime. DI sẽ quản lý việc đó và những bên tiêu dùng sẽ không bị ảnh hưởng, chỉ việc sử dụng những chức năng mà nó cam kết (qua interface, abstract class, hoặc chính khai báo của class đấy).
  + **Constructor Injection**: Phụ thuộc được tiêm vào qua constructor của lớp.
  + **Setter Injection**: Phụ thuộc được tiêm qua các phương thức setter.
  + **Field Injection**: Phụ thuộc được tiêm trực tiếp vào thuộc tính bằng cách dùng annotation @Autowired.
  + Sự khác biệt của **Constructor injecttion và Setter Injecttion:**
    - **Setter:** phụ thuộc được tiêm vào sau khi đối tượng được tạo, hiểu đơn giản là cần gì thì tiêm cái đó
    - **Constructor:**  phụ thuộc được tiêm vào trong khi đối tượng được khởi tạo, hiểu đơn giản là đầy đủ phụ kiện để sử dụng rồi
* **Inversion of Control (IoC):** là nguyên tắc bắt Spring phải tự tạo và quản lý đối tượng và phụ thuộc của nó, sử dụng DI để tiêm phụ thuộc cho đối tượng
* **Spring bean, life cycle bean, bean scope. Các annotation sử dụng để khai báo bean trong Spring.**
* **Bean:** 1 bean nghĩa là một đối tượng được khởi tạo, cấu hình, và quản lý bởi spring container, được định nghĩa trong file cấu hình, hoặc dùng annotation để tự động phát hiện
* Vòng đời của Bean:
  + **Instantiation:** khởi tạo bean thông qua constructor hoặc method static
  + **DI:** Tiêm các phụ thuộc vào bean
  + Hàm được đánh dấu @PostConstruct sẽ được gọi.
  + Bean đã sẵn sàng để hoạt động.
  + Nếu IoC Container không quản lý Bean nữa hoặc bị shutdown, hàm @PreDestroy sẽ được gọi.
  + Bean được xóa.
* **Bean scope (Phạm vi của bean):**
  + **Singleton:** Mặc định, Spring chỉ tạo một đối tượng duy nhất của bean cho toàn bộ ứng dụng.
  + **Prototype:** Mỗi khi có yêu cầu mới, Spring sẽ tạo một đối tượng mới của bean.
  + **Request:** Mỗi yêu cầu HTTP sẽ có một instance của bean.
  + **Session:** Một instance cho mỗi session của người dùng.
* **Khái niệm cơ bản về request, request, httpMethod, JSON, API**
  + **Request:** là yêu cầu gửi lên server từ phía client
  + **Response:** là dữ liệu được trả lại từ server tới người dùng
  + **httpMethod:**
    - **GET: Lấy dữ liệu từ server (ví dụ: xem danh sách sản phẩm).**
    - **POST: Gửi dữ liệu lên server (ví dụ: thêm sản phẩm mới).**
    - **PUT: Cập nhật dữ liệu.**
    - **DELETE: Xóa dữ liệu.**
  + **JSON:**  là định dạng để trao đổi dữ liệu giữa client với server
  + **API:** là cái mà website và server dùng để giao tiếp với nhau, ví dụ website lấy dữ liệu từ API mà server trả về
* **Sơ lược về maven:**
  + **Maven** là một công cụ giúp quản lý dự án, tự động tải các thư viện cần thiết và xử lý quá trình xây dựng (build) dự án.
  + **Pom.xml**: Đây là tệp cấu hình chính của Maven. Có thể thêm các **dependency** (thư viện) vào file này, Maven sẽ tự động tải về và thêm chúng vào dự án.

**RestController & Controller**

* Sự khác nhau giữa @RestController và @Controller
  + @RestController: Là sự kết hợp của @Controller và @ResponseBody để trả về dữ liệu thay vì trả về 1 view, dùng để dựng các RESTfulAPI, trả về dữ liệu dạng XML hoặc JSON
  + @Controller: Thường dùng để trả về **view** trong các ứng dụng web, sử dụng để đánh dấu các lớp là Spring MVC Controller.
  + **Khi nào sử dụng?**
    - **@RestController**: Khi muốn xây dựng các **API** trả về dữ liệu (JSON, XML) cho các ứng dụng frontend như Angular, React, hoặc mobile app.
    - **@Controller**: Khi bạn muốn xây dựng các ứng dụng web **MVC truyền thống** và trả về các **view**.
* **Routing**
  + là việc ánh xạ các yêu cầu HTTP đến các phương thức trong controller
  + Sử dụng các method @GetMapping, @PostMapping, @PutMapping, @DeleteMapping để ánh xạ các phương thức HTTP (GET, POST, PUT, DELETE)
* **Cách controller nhận input**
  + **Request Parameters (Params):** Nhận dữ liệu từ query (?key=value) **VD:** @RequestParam String category
  + **Request Body**: Nhận dữ liệu từ body của HTTP request (thường dùng cho POST, PUT).  
    VD: @RequestBody Product product
  + **Path Variable**: Nhận dữ liệu từ path của URL.(product/{id})  
    VD: @PathVariable Long id
* **Validate input:**
  + Spring hỗ trợ việc **validate input** bằng cách sử dụng @Valid và các annotation từ thư viện javax.validation.
* **Handle Exception:**
  + **@ExceptionHandler**: Được dùng trong từng controller riêng biệt để xử lý các ngoại lệ.
    - Nó sẽ định nghĩa ra 1 method xử lý ngoại lệ ở phương thức nào throw ra nó
    - Method nào throw ra lỗi mà có method xử lý thì sẽ tự gọi method đó
  + **@ControllerAdvice**: Áp dụng cho toàn bộ ứng dụng, xử lý các ngoại lệ ở nhiều controller khác nhau.
    - Nó định nghĩa một class chứa các method **ExceptionHandler,** ở phạm vi toàn cục, có nghĩa ở đâu nó cũng tự bắt lỗi được nếu có thằng ném ra

**ORM**

* **ORM(Object-Relational Mapping):**
  + là kĩ thuật cho phép ta ánh xạ đối tượng trong code với các bảng trong cơ sở dữ liệu
  + ORM giúp **chuyển đổi dữ liệu giữa cơ sở dữ liệu** và các đối tượng trong ứng dụng mà không cần viết truy vấn SQL thủ công.
* Lợi ích
  + Giảm công việc phải viết các truy vấn SQL thủ công
  + Tăng năng suất vì chỉ phải làm việc với các đối tượng
  + dễ bảo trì
* Cơ chế hoạt động
  + **Map các đối tượng với các bảng**: ORM tự động ánh xạ các lớp (entity) trong ứng dụng với các bảng trong CSDL.
  + **Sinh SQL tự động**: ORM có thể tạo ra các câu truy vấn SQL dựa trên các thao tác với các đối tượng trong mã nguồn.
  + **Quản lý vòng đời của đối tượng**: ORM quản lý vòng đời của các đối tượng (entity) và tự động đồng bộ hóa dữ liệu giữa chúng với CSDL.

**So sánh ORM vs JDBC:**

* **ORM**:
  + Tự động sinh truy vấn, dễ dùng hơn.
  + Có khả năng cache, giúp cải thiện hiệu suất trong nhiều trường hợp.
  + Dễ bảo trì, ít mã lặp.
  + Tích hợp tốt với framework như Spring.
* **JDBC**:
  + Hiệu suất cao hơn khi sử dụng trong các trường hợp cần tối ưu hóa từng câu truy vấn SQL cụ thể.
  + Phải viết thủ công các câu lệnh SQL, điều này có thể dẫn đến mã nguồn dài hơn và dễ lỗi.
  + Khả năng bảo trì thấp hơn do có nhiều mã SQL được viết trực tiếp trong mã nguồn.

**Spring Data JPA** là một phần mở rộng của **JPA (Java Persistence API)**, và JPA là một **chuẩn ORM**.

**Quan hệ trong JPA:**

1. Many To One: là nhiều đối tượng tham chiếu đến 1 đối tượng  
   vd: Ví dụ: Nhiều sản phẩm (Product) thuộc về một danh mục (Category).
2. One To Many: 1 đối tượng tham chiếu đến nhiều đối tượng  
   VD: Một danh mục (Category) có thể chứa nhiều sản phẩm (Product).
3. Many To Many: nhiều đối tượng liên kết đến nhiều đối tượng khác  
   VD: Một sinh viên (Student) có thể tham gia nhiều khóa học (Course) và một khóa học có thể có nhiều sinh viên.

**Cascade type trong JPA:**

Cascade type quyết định hành động của một entity sẽ ảnh hưởng thế nào đến các entity liên quan:

* **CascadeType.PERSIST**: Khi lưu một entity, các entity liên quan cũng sẽ được lưu.
* **CascadeType.MERGE**: Khi hợp nhất (merge) một entity, các entity liên quan cũng sẽ được hợp nhất.
* **CascadeType.REMOVE**: Khi xóa một entity, các entity liên quan cũng sẽ bị xóa.
* **CascadeType.ALL**: Áp dụng tất cả các hành động cascade.

**Fetch type trong JPA:**

Fetch type xác định cách mà dữ liệu được tải khi truy vấn:

* **FetchType.EAGER**: Dữ liệu liên quan được tải ngay lập tức khi truy vấn.
* **FetchType.LAZY**: Dữ liệu liên quan chỉ được tải khi cần thiết (khi truy cập).

**Chuyển DTO <-> Entity**

* Sử dụng thư viện **MapStruct** hoặc **ModelMapper**.
  + Với **MapStruct**, chỉ cần định nghĩa các phương thức mapping trong interface, và MapStruct sẽ tự động sinh mã
* Sử dụng Java Reflection
  + Viết phương thức thủ công để lặp qua các thuộc tính và set giá trị tương ứng.

**Advance**

* **Native query:**
  + Native query cho phép viết các câu SQL thuần mà không thông qua JPA, thường được sử dụng khi cần tối ưu hóa hiệu suất hoặc viết các truy vấn phức tạp.  
    VD: @Query(value = "SELECT \* FROM products WHERE name = :name", nativeQuery = true)
* **Specification**
  + cho phép xây dựng các truy vấn động trong JPA dựa trên các tiêu chí khác nhau.
* **Paging**
  + giúp phân trang dữ liệu trong các truy vấn, sử dụng đối tượng Pageable.