Spring

Spring Framework

- Beans are Java objects.
- Entities such as `Students` in JPA/Hibernate represent data objects, not services or components. They are meant to be transient and managed by the `EntityManager` or `Session` when interacting with the database.
- Spring typically manages stateless, reusable components (like services, repositories, etc.), while entities are often transient and tied to specific database operations.
- `@Autowired` is used to inject Spring-managed beans, which are typically singleton, prototype, or other scoped beans. Since entities are not Spring-managed beans, they should not be injected with `@Autowired`.
- Instead, entities are created and managed within the context of transactions and specific database operations.
- Entities should be created and manipulated within the scope of a method or a transaction, not held as long-lived fields.

▼ **JPA**

▼ **entity Annotations**

- `@Transient` : Will not be mapped to any column in the database.
- `@Column` == NON Annotation
- `@GeneratedValue`(strategy = GenerationType.***AUTO***) **or** (strategy = GenerationType.***IDENTITY***)
- `@Temporal`(TemporalType.***DATE***)
- `@Enumerated`(EnumType.***STRING***) **or** (EnumType.**ORDINAL**)

▼ **Advantages and Disadvantages of Using Ordinals**

Advantages:

1. **Space Efficiency**: Storing the ordinal as an integer is usually more space-efficient compared to storing the string representation, especially when dealing with large datasets.
2. **Faster Comparisons**: Integer comparisons are faster than string comparisons, which could make querying slightly faster.

Disadvantages:

1. **Lack of Readability**: The stored integer doesn't directly convey the meaning of the enum value. For example, `2` doesn't inherently tell you that it corresponds to `SHIPPED`.
2. **Risk with Enum Changes**: If the enum declaration is modified (e.g., a new value is inserted in the middle), it will shift the ordinal values of subsequent enum constants, potentially corrupting the data. For example, if you insert a new status `IN_TRANSIT` between `PROCESSING` and `SHIPPED`, the ordinal of `SHIPPED` would change from `2` to `3`.

- `@Entity`
- `@Table` == NON Annotation
- `@Id`
  - ▼ **ID auto generator problem**
    - Hibernate uses a concept called "hi/lo" or "pooled" strategies, where a block of IDs is reserved in memory, and this can lead to gaps if the application is restarted or the session ends before all IDs in the current block are used.

▼ DTO

- DTOs are indeed a form of abstraction that helps manage complexity, improve security, and maintain flexibility in your software design.

▼ foreign key constraint

- ```
  CREATE TABLE corrupteditems (
      id BIGINT AUTO_INCREMENT PRIMARY KEY,
      item_id BIGINT NOT NULL,
  ```
  ```

```
    FOREIGN KEY (item_id) REFERENCES items(id)
);
```

The `@JoinColumn(name = "item_id", referencedColumnName = "id", nullable = false)` annotation in Hibernate not only specifies the column name for the foreign key but also ensures that a foreign key constraint is created in the database.

- In the case of a unidirectional `ManyToOne` relationship, the `@JoinColumn` annotation does not necessarily need the `referencedColumnName` attribute if it is referencing the primary key of the target entity.
- By default, `@JoinColumn` will reference the primary key of the target entity.
- When you use `@OneToOne` by itself (without `@JoinColumn`), Hibernate will still handle the foreign key constraint by default

- **Service Layer Validation**

```
import org.springframework.beans.factory.annotation.Autowir
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transacti

@Service
public class CorruptedItemService {

    @Autowired
    private ItemRepository itemRepository;

    @Autowired
    private CorruptedItemRepository corruptedItemRepository

    @Transactional
    public void addCorruptedItem(Long itemId) {
        // Check if the item exists
        Item item = itemRepository.findById(itemId).orElseT

        // Create and save the corrupted item
```

```
            CorruptedItem corruptedItem = new CorruptedItem();
            corruptedItem.setItem(item);
            corruptedItemRepository.save(corruptedItem);
        }
    }
```

- the default fetch type for `@ManyToOne` is `FetchType.EAGER`

▼ **Relationship**

- In the context of object-relational mapping (ORM) with Hibernate, relationships between entities can be categorized as unidirectional or bidirectional, but there are also different types of relationships based on cardinality (`@OneToOne` ,...)
- **Unidirectional**: Only one entity knows about the relationship. Navigation is possible from one side only.
- **Bidirectional**: Both entities know about the relationship. Navigation is possible from both sides.
- **SQL Schema**: Remains the same for both unidirectional and bidirectional relationships.
- Unidirectional To Bidirectional just need `mappedBy`
  - the `mappedBy` attribute is used in the `@OneToMany` or `@ManyToMany` or `@OneToOne`
- `@OneToOne` **Without** `@JoinColumn` **: TRUE but not suggested**
- `@JoinColumn` **Without** `@OneToOne` **: FALSE**

▼ **REST (Spring MVC)**

▼ **The hole structure**

▼ **annotations**

▼ **@ResponseBody**

▼ **defenation**

- This annotation tells Spring Boot to directly write the return value of the method into the HTTP response body.

▼ **Serialization**

▼ ResponseEntity

Type	Usage	Purpose
ResponseEntity<CorruptedItems>	Returns the CorruptedItems object in the response body when the item is found.	Best for returning a specific object that will be serialized (e.g., JSON) in the response.
ResponseEntity<String>	Returns a plain string message in the response body (e.g., "Item found" or "Item not found").	Best for simple textual responses like messages, error descriptions, or debugging info.
ResponseEntity<>	Returns a response without specifying the body type, often used when only the status matters.	Appropriate when you don't care about the body (e.g., just returning 404 NOT FOUND status).

- Serialization takes place inside Spring's **DispatcherServlet** and **HttpMessageConverters**.
- **Jackson** (for JSON): By default, Spring Boot uses the **Jackson library** to convert Java objects into JSON. Jackson inspects the fields of the `CorruptedItems` class and generates a corresponding JSON representation.
- **JAXB** (for XML): If the API or the client prefers XML, Spring can use the **JAXB** library to convert the Java object into an XML format.
- The `@ResponseBody` annotation (or using `ResponseEntity` directly) tells Spring Boot that the return value of the controller method should be written directly to the HTTP response body. Spring then uses the appropriate message converter (like Jackson) to serialize the return object

(`CorruptedItems`) into JSON or XML based on the `Content-Type` header or the client's `Accept` header.

- If the request's `Accept` header specifies `application/json`, Spring will use Jackson to serialize the object to JSON.
- If `application/xml` is requested, Spring might use JAXB to serialize the object to XML (if properly configured).
- If needed, you can customize the serialization process, for example, by adding **Jackson annotations** to the `CorruptedItems` class to control how the object is serialized:
- `@JsonIgnore`: Ignore certain fields during serialization.
- `@JsonProperty`: Customize the names of fields in the serialized output.
- `@JsonFormat`: Format date fields in a specific way.
- Serialization ensures that the internal Java representation of an object can be safely transmitted over the network in a format that other systems (whether they are written in Java or other languages) can understand. For example, a JavaScript client (like a web browser) can easily parse a JSON response and use its data.

▼ **Wrapping**