**Inversion of Control (IoC):** design principle where the control over certain parts of the code is inverted, meaning that instead of the code controlling everything, the control is handed over to an external framework or library. This principle allows for greater flexibility and decoupling in software design.

بدل ما كلاس واحد او جزء واحد من الكود مسؤول عن كل حاجة, بيخلي الكود اقل اعتمادية حيث في جزء خارجي هو اللي بيتحكم في أجزاء معينة من الكود (بستقبل ميثود معينة بدل ما أنشئها ) بيكون عن طريق استخدام

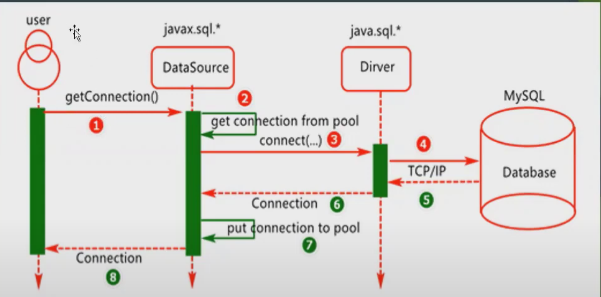
Strategy Pattern /DI (most popular) / Service Locator Pattern/ Template Method.

**Dependency Injection (DI):** a specific technique to implement IoC. DI involves injecting dependencies into a class from the outside, rather than the class creating these dependencies itself. This makes the code easier to maintain, test, and extend.

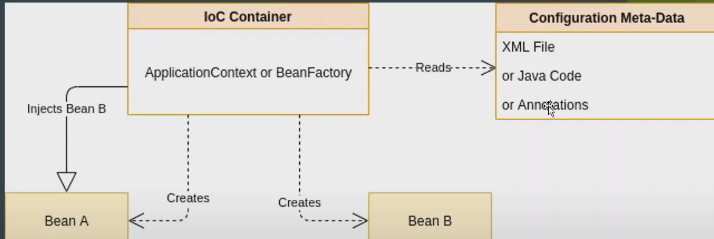
* Constructor injection: Dependencies are provided through a class constructor.
* Setter Injection: Dependencies are set using setter methods.
* Field Injection: Dependencies are directly injected into class fields, often using annotations Spring.
* **Spring JDBC**: a framework that simplifies working with databases using JDBC. includes:
  + **DataSource**: Manages connections to the database.
  + **JdbcTemplate**: Provides an easier interface for executing queries.
  + **NamedParameterJdbcTemplate**: Supports named parameters in queries.

C**onfiguration in “application.properties”:**

* Contains settings for database connection such as URL, username, and password.
* **Connection pooling: mechanism** can be enabled to improve performance by reusing connections, allocating new connections, managing available connections, and closing connections.



* Configuration types:
* XML / Java based / Annotations



Annotations: metadata provides data (supplemental information) about the program

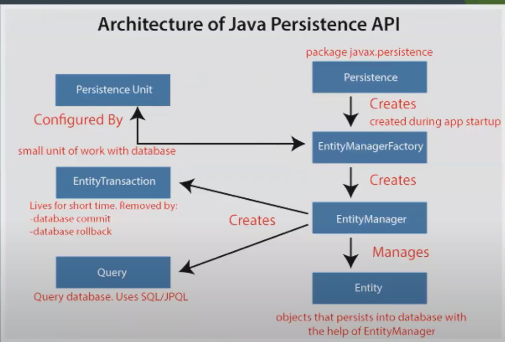
1. @Autowired: dependency injection 🡪used on setter methods, variables, and constructors. Spring manages & creates the necessary objects.
2. @component: class-level annotation indicate a class is a Spring class(as a Bean)) to be discovered & managed by Spring during the classpath.
3. @componentScan: specify the packages that Spring should scan for components to be managed as Beans.
4. @Configuration: indicate that a class contains one or more definitions to be managed by Spring Controller.
5. @Bean: used inside a class with @configration to indicate. that a method returns an object that should be managed as a Bean by spring.
6. @Service: define a class as a service managed by Spring.
7. @Repository: define a class as a Repository managed by Spring.

ORM: Application 🡪 ORM(JPA🡪vendor(Hibernate))🡪Database.

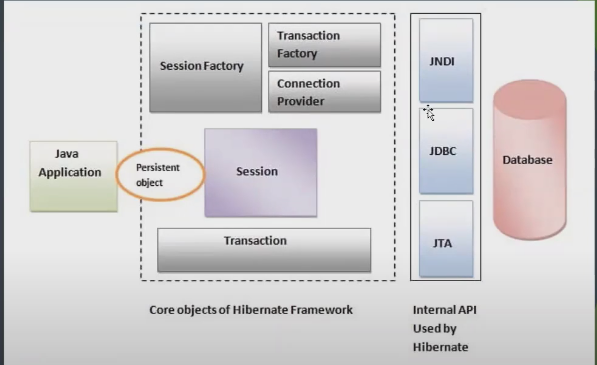
* JPA: persist data between Java object & database.

Acts as a bridge between OO models & database.

Implemented by ORM tools like Hibernate/ TopLink/ iBatis as it’s just a specification, that doesn’t perform any operations by itself.



* Hibernate: open source Java framework simplifies the development of Java applications to interact with the database.
  + lightweight, ORM (Object Relational Mapping) tool.
  + implements the specifications of JPA (Java Persistence API) for data persistence.
  + Fast Performance: because the cache is internally used in the hibernate framework.
  + Generate Database Independent Queries.
  + Automatic Table Creation & Simplifies Complex Join



***Entity Lifecycle Model in JPA & Hibernate***

Cascading Operations:

* PERSIST:- When you **persist** (save) a parent entity, the associated child entities are also automatically saved.

**Example**: If you save a Person object, the Address object linked to that Person will also be saved in the database.

* MERGE:- it updates the existing data in the database with the data from the entity you provided.

**Example**: If you update a Person object and merge it, the database will update the existing record with the new data, and the same will happen for the linked Address object.

* REMOVE:- When you **remove** an entity, it deletes the corresponding row from the database.

**Example**: If you delete a Person object, the Address object linked to it will also be deleted from the database.

* DETACH:- **Detach** means removing an entity from the persistent context, so it’s no longer tracked by JPA.

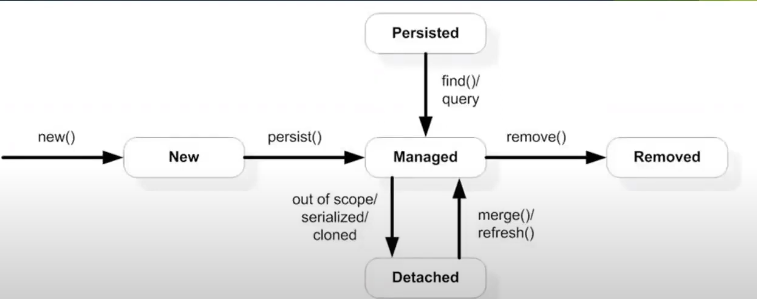
**Example**: If you detach a Person object, the Address object linked to it will also be detached, meaning JPA will stop tracking any changes you make to these objects.

* REFRESH:- means reloading an entity's data from the database, undoing any changes you made since it was last loaded.

**Example**: If you made some changes to a Person object and then use refresh, those changes will be discarded, and the Person object will be restored to its original state from the database.

* **Transient**:- The entity is new and not associated with any persistence context.

* ALL:- ensure that all the cascade operations apply to the child entities, effectively managing the entire object graph as a single unit.



* + - * PERSIST and MERGE are for saving and updating.
      * REMOVE and DETACH are for deleting and stopping tracking.

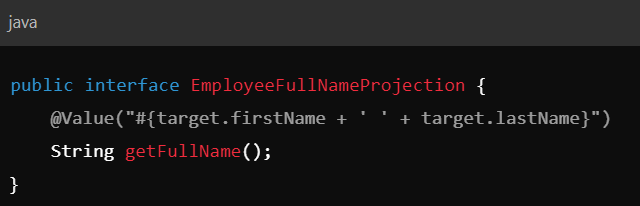
Data JPA Projection: When working with databases, you might not always want to retrieve entire entities; instead, you may need just a subset of data. Projections in Spring Data JPA allow you to fetch specific columns (fields) from an entity. This can optimize performance by reducing the amount of data loaded into memory and simplify the handling of data.

* Interface-based Projection:

specify the data you want to fetch by declaring an interface with methods corresponding to the entity fields you want to retrieve.

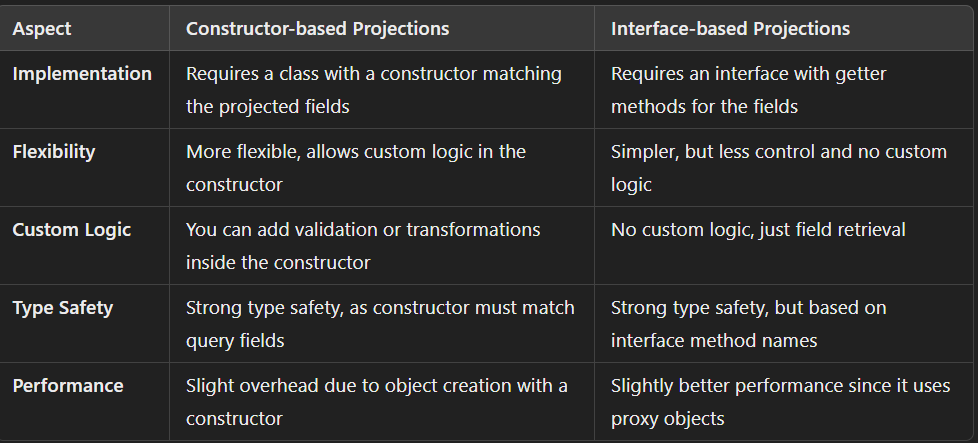
* Interface-Closed Projection: the accessor methods of the projection interface match the properties of the target aggregate (entity). This means all the methods correspond to actual fields in the entity. Like 🡪getName(),getPrice()…
* Interface-Open Projection: allow you to define methods in the projection interface that do not directly map to entity fields. Instead, you can use custom expressions, such as concatenating fields or applying transformations using the “@Value” annotation.

ex: firstName form entity + lastName from entity = fullName in interface.



* Constructor-based Projection: **Constructor-based Projections** allow you to retrieve specific columns from a database and map them directly to a Data Transfer Object (DTO) or a custom class using a constructor. This type of projection is useful when you want more control over the data, such as performing custom transformations, validations, or logic inside the DTO.

Interface & Constructor Based Projection



Transaction: essential for ensuring that a group of operations either all succeed or all fail. If any operation fails, the whole transaction can be rolled back

*“@Transactional”*: Class / Method annotation with attributes:

* **Propagation:** Controls how transactions behave when calling other transactions.
  + REQUIRED**:** Uses the current transaction or starts a new one.
  + REQUIRES\_NEW**:** Always creates a new transaction.
  + SUPPORTS**:** Uses a transaction if one exists; otherwise, runs without.
* **Isolation:** Manages data visibility between transactions.
* READ\_COMMITTED**:** Only sees committed changes (default).
* SERIALIZABLE**:** Ensures full isolation, highest level.
* **Timeout:** Sets the maximum time a transaction can run before it's aborted.

🡪 @Transactional(timeout=30) means the transaction will be aborted if it runs longer than 30 seconds.

* **readOnly:** Optimizes transactions for read operations by marking them as read-only. @Transactional(readOnly=true)
* **rollbackFor/noRollbackFor:** Specifies which exceptions should or shouldn't trigger a rollback.

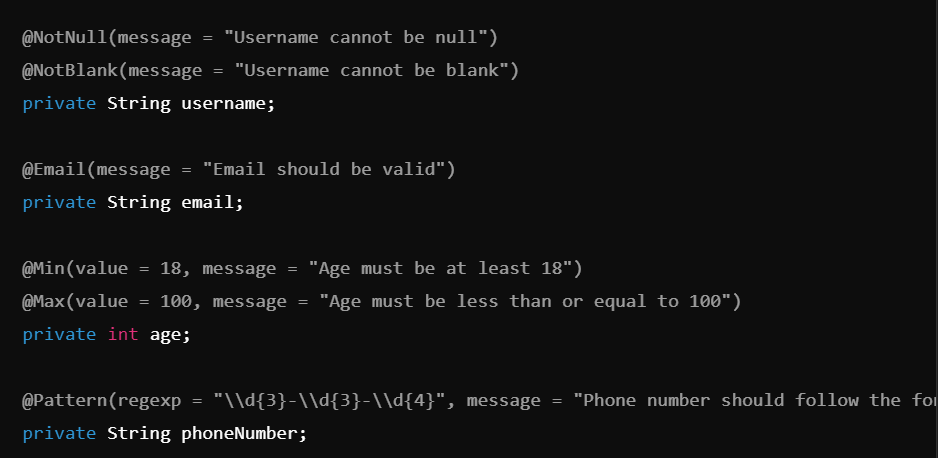
You can prevent rollback on certain exceptions, even if they are unchecked

@Transactional (noRollbackFor=IllegalArgumentException.class).

Validation is Spring: Bean Validation in Spring works by applying constraints to the fields of a class, ensuring that they meet certain conditions. This allows for automatic data validation, ensuring that inputs are valid before processing.

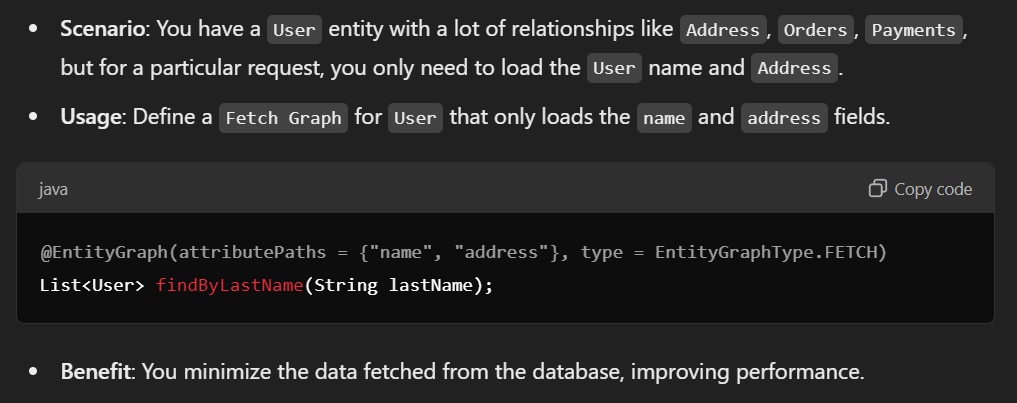
Key Annotations: Spring supports Hibernate validator by adding annotations to fields.

* **@Validated**: class-level annotation tells Spring to validate parameters passed into a method of the annotated class.
* **@Valid**: used at the **method parameters** or **fields level** to validate method inputs.
* **@NotBlank**: ensure that the string input/attribute is not null or empty “must have at least one non-whitespace character”.
* **@Min**: Ensures that a numeric field has a value **greater than or equal to** a specific value.
* **@Max**: Ensures that a numeric field has a value **less than or equal to** a specific value.
* **@Pattern:** Used for fields that need to follow a specific format, like phone numbers, postal codes, and IP address... as matching the given regular (regex).
* @Email: ensure that the string field contains a valid email address according to standard email format.

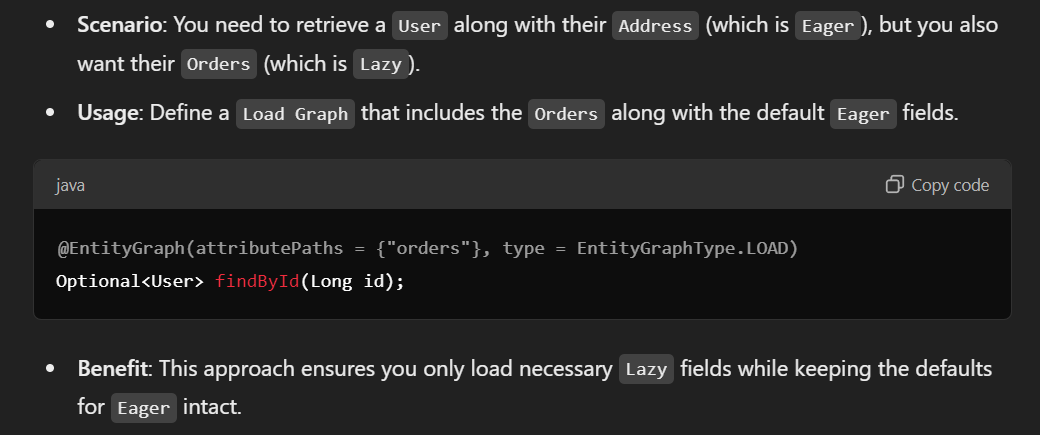


Use Entity Graphs (Fetch and Load): in JPA are used to specify how related entities should be fetched from the database. This allows you to control which associations should be loaded to avoid unnecessary data fetching.

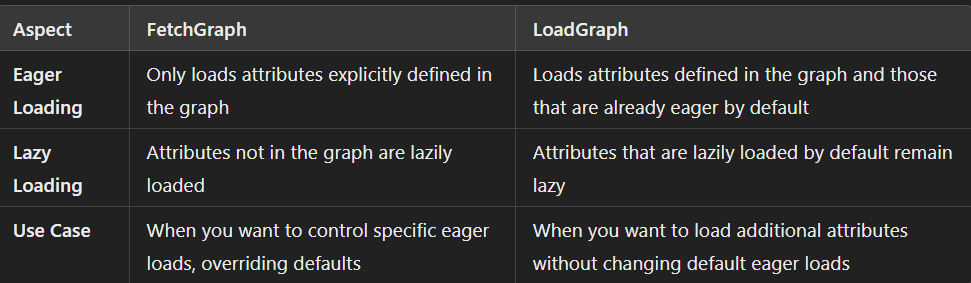
* Fetch Graph: to load only the essential data to optimize performance by fetching only a specific subset of entity relationships and ignoring the default Eager or Lazy loading behaviors.” **limit** the data fetched strictly”



* Load Graph: fetch all Eager fields **and** include(add) extra Lazy fields that are not loaded by default.



@EntityGraph Annotation: to define and apply Fetch or Load Graphs. This allows you to specify which relationships or attributes to load as part of the query execution without changing the default fetch type of the entity mappings, You can define a graph in the entity using @NamedEntityGraph or reference it with @EntityGraph in your queries



Benefits of Fetch and Load Graphs:

1. Performance Optimization: Helps reduce unnecessary loading of large relationships unless explicitly needed, reducing memory and bandwidth overhead.
2. Fine-grained control: You have control over what associations are loaded for each query, making it flexible for different use cases.
3. Simpler than Custom Queries: Instead of writing complex JPQL or SQL queries to load specific associations, you can use entity graphs to control fetching.

JPA Fetch Type: how associated entities are retrieved from the database.

* Lazy Fetch: associated data is fetched only when needed, which is the default for many-to-many and one-to-many relations.

**When to use?**: Use lazy fetching when the associated data is **not always needed** and you want to load data only when it is requested.

**\*** **Can cause the N+1 Query Problem, where multiple queries are executed to fetch related entities, increasing query execution time.**

* Eager Fetch: With eager fetching, related entities are **fetched immediately** along with the primary entity. This means that when the primary entity is retrieved, all associated data is fetched in the same query.
* which is the default for many-to-one and one-to-one relations.

**When to use: When you know that the associated data will always be needed when retrieving the primary entity and** want to avoid additional queries later.

JPA Auditing: easily manage the tracking of entity changes in a clean and automated way, reducing the need for manual record-keeping of user actions and timestamps.

* + The auditing feature in Spring Data JPA can be activated using annotations like @CreatedBy, @LastModifiedBy, @CreatedDate, and @LastModifiedDate.
  + **AuditingEntityListener** automatically updates the relevant fields when an entity is persisted or updated.
  + **AuditorAware** must be implemented to provide the current user responsible for changes.

### **@CreatedBy and @LastModifiedBy:** These annotations are used to capture **who** created or modified an entity.

* **@CreatedBy**: store the user who originally created the record.
* **@LastModifiedBy**: store the user who last modified the record.

@CreatedDate and @LastModifiedDate: These annotations capture the timestamps of when an entity was created or last modified.

* **@CreatedDate:** This field will store the **date** and **time** when the entity was created.
* **@LastModifiedDate**: This field will store the **date** and **time** when the entity was last modified.

Sorting & pagination:

**Sorting:** arranging data in a specific order, like ascending or descending.

* **to Implement Sorting:**
  + **Using PageRequest:** When you create a PageRequest object, you can specify sorting along with pagination.
  + **Using Sort Directly:** You can also use the Sort object alone to define sorting criteria.
* **fields you use for sorting must directly match the names of the fields in your domain model class or the aliases used in your JPQL query. This ensures that the sorting operation correctly references the data structure in your code.**

**Pagination**: dividing a large set of results into smaller, manageable pages.

* **To implement Pagination:**
  + **Using Pageable:** The Pageable interface holds details about the requested page, such as:
    - **Page size:** The number of items on each page.
    - **Page number:** Which page you are currently viewing.
    - **Sort information:** Sorting can be combined with pagination using a Sort object.

Locking Mechanisms: Optimistic vs Pessimistic

* **Optimistic Locking**: Uses a version column to prevent concurrent modifications. Throws an OptimisticLockException if another transaction has updated the entity.
* **Pessimistic Locking**: Prevents others from modifying the entity by locking it during a transaction, potentially causing **deadlocks**.
* Use optimistic locking in scenarios with high concurrency to avoid unnecessary database locks.

**Exceptions in Spring**

**Request Flow without an Exception**

1. REST Client sends an HTTP call to the Spring Application, targeting the Controller layer.
2. The Controller handles the HTTP request and forwards the necessary business logic to the Service layer.
3. If the Service layer completes its operation without errors, it returns a response to the Controller.
4. The Controller sends the response back to the REST Client.

Request Flow with an Exception

1. If the Service layer encounters an error (e.g., a business logic exception), it throws an exception.
2. The Controller layer does not handle the exception directly. Instead-by default-, it is passed to a special class known as the RestExceptionHandler (part of the Spring Exception Handling mechanism).

* RestExceptionHandler: responsible for catching the exception and encapsulating the error details into an **ApiError object**.

The **ApiError object** typically contains information such as:

* **HTTP Status Code** (e.g., 400 Bad Request, 404 Not Found)
* **Error Message**: A description of the error that occurred.
* **Timestamp**: When the error happened.
* **Details**: Any additional information related to the error (e.g., validation error details).

Handle Exceptions Components in Spring:

**“@**ControllerAdvice” : Applied at the **class level** and works across all controllers.

* Used to handle exceptions **globally** across all controllers in the application by Combine with **“@ExceptionHandler()”**.
* Allows to centralize exception handling logic, so all controllers benefit from consistent error handling.

**“@ExceptionHandler()”**

* Defines **specific exceptions “Locally”** handled by methods in the controller or in a class annotated with “@ControllerAdvice” .Each method can handle different exceptions.
* For example, one method could handle NullPointerException, while another handles EntityNotFoundException.

Note:

\* You can use @ResponseStatus with @ExceptionHandler to define the HTTP status associated with a specific exception.

