IOC&DI

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

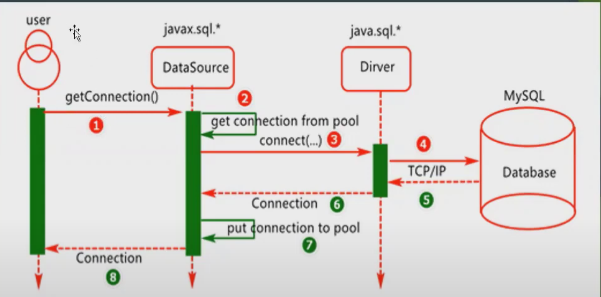
Strategy Pattern /DI (most popular) / Service Locator Pattern/ Template Method.

**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.

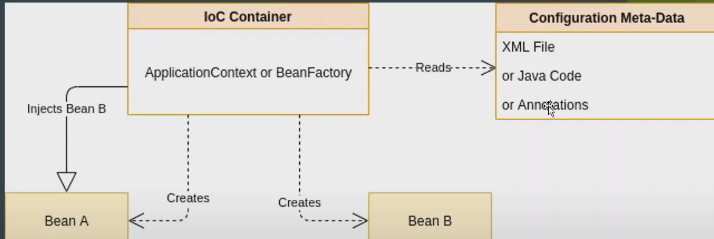
* Constructor injection: Dependencies are provided through a class constructor.
* Setter Injection: Dependencies are set using setter methods.
* Field Injection: Dependencies are directly injected into class fields, often using annotations Spring.
* **Spring JDBC**: framework that simplifies working 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

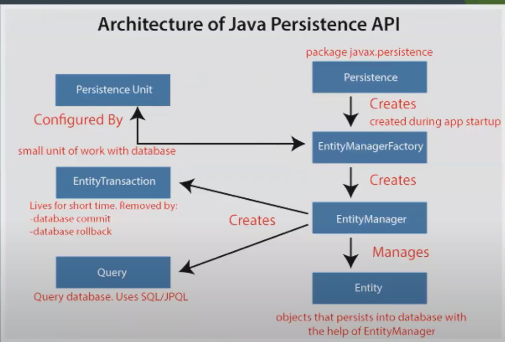
1. @Autowired: dependency injection 🡪used on setter methods, variables, and constructors. Spring manages & creates the necessary objects.
2. @component: class-level annotation indicate a class is a Spring class(as a Bean)) to be discovered & managed by Spring during the classpath.
3. @componentScan: 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.

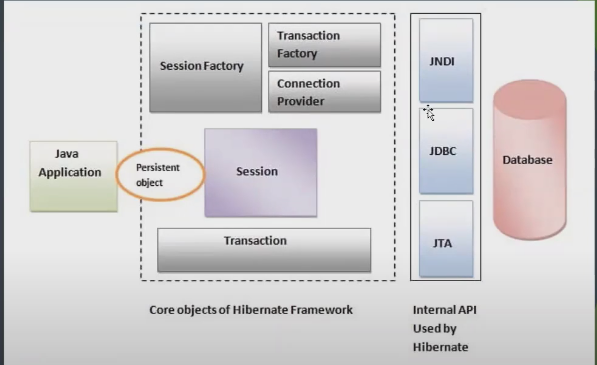
* JPA: persist data between Java object & database.

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.

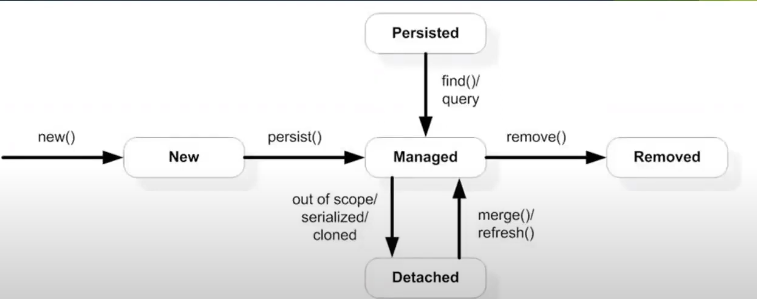
* DETACH:- **Detach** means removing an entity from the persistent context, so it’s no longer tracked by JPA.

**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.

* 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.

JPA Fetch Type:

* Lazy Fetch: The related entities are not loaded immediately. They are only loaded when they are accessed.
  + - * when you want to delay loading related data until it's actually needed, which can improve performance by reducing unnecessary data loading
* OneToMany, ManyToOne.
* Eager Fetch: The related entities are loaded immediately along with the main entity.
  + - * when you know you will always need the related data right away, ensuring it's available without additional database queries.
* OneToOne, ManyToMany.