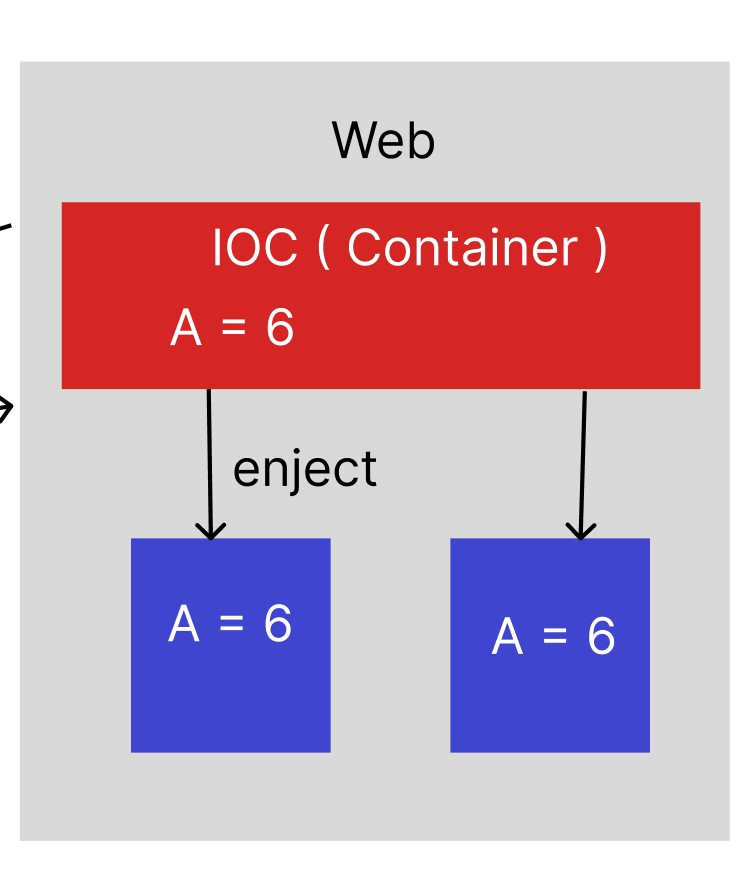
Trong controller, mình ko tạo connection cho mỗi lần thực thi câu query mà chúng ta lưu connection variable lưu vào trong thanh heap memory. Chứ ta ko new connection mỗi lần tạo connection.

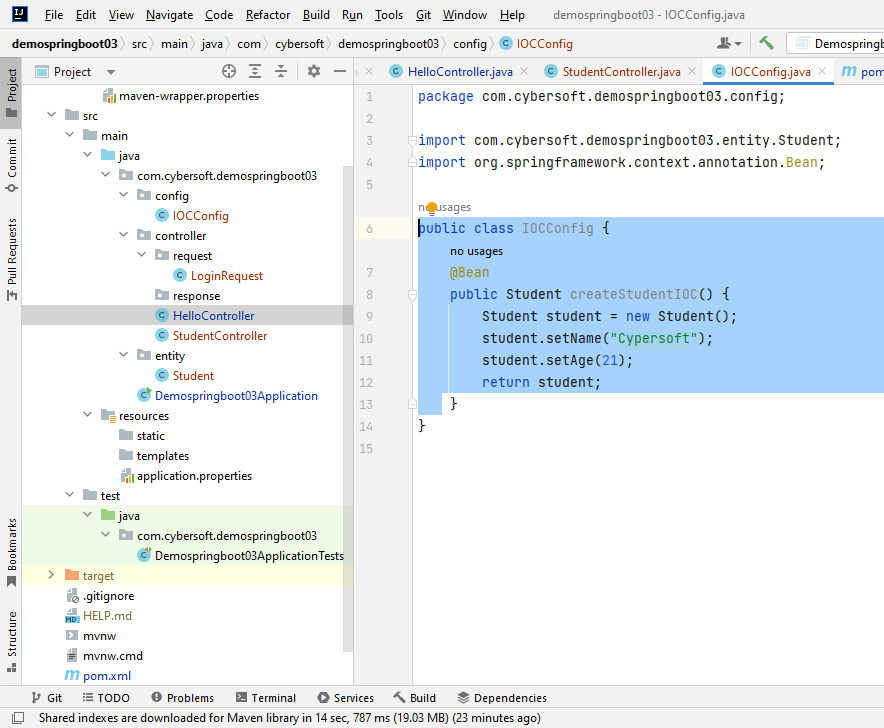
IOC (Container) là nơi lưu trữ các object sau khi được instance từ class thay vì lưu trữ trong heap memory (ko security).

Nếu muốn tạo ra class gán sẵn giá trị và sử dụng lại class này cho class khác thì phải đưa class này lưu trữ trên container (IOC), thông qua annotation @Bean, @Service, @Component, @Repository

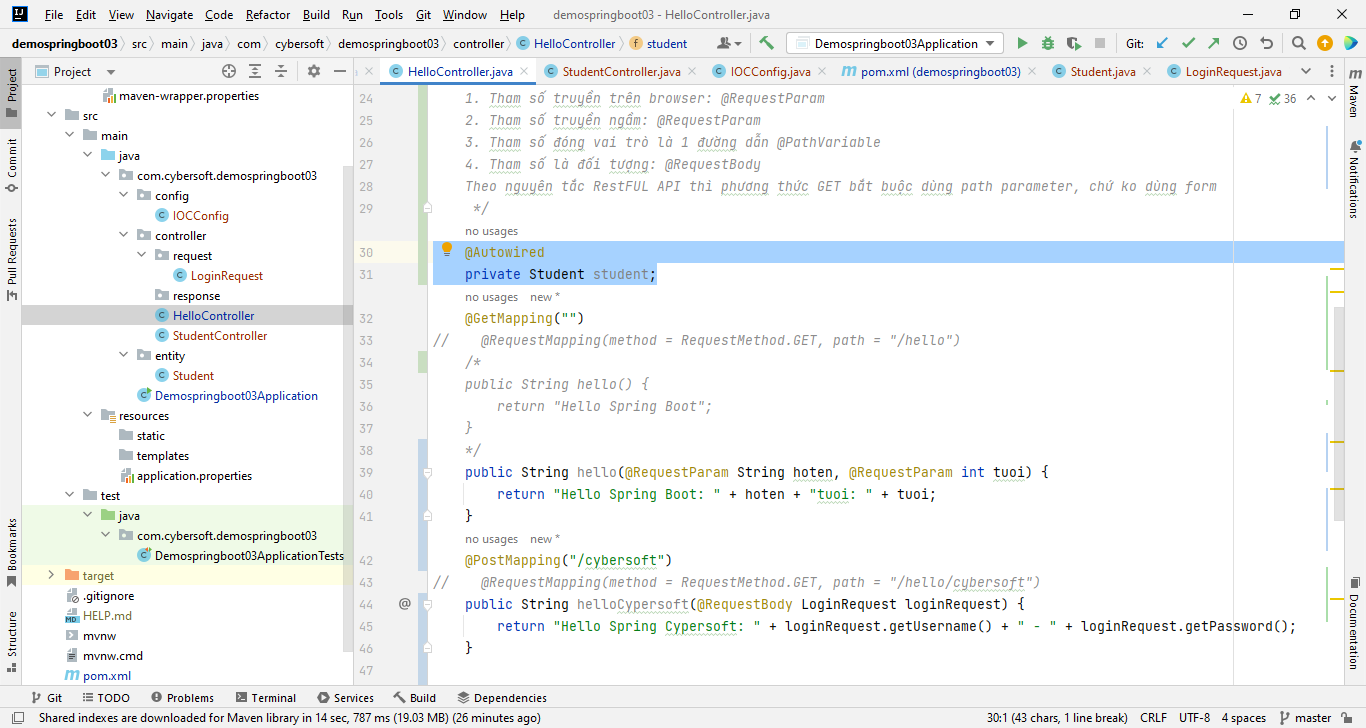


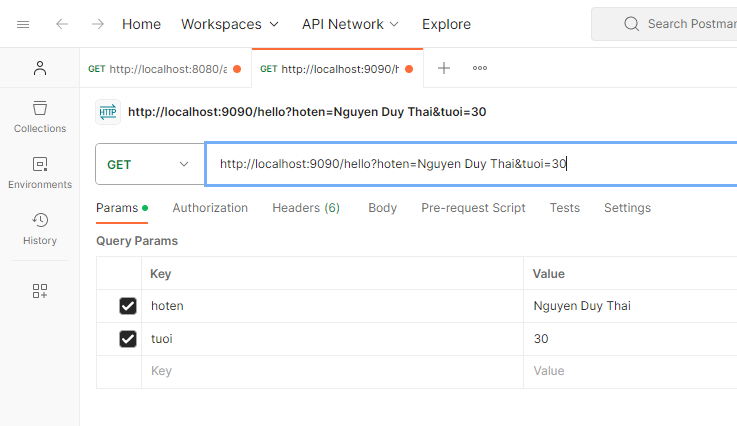
@Bean  
public Student createStudentIOC() {  
 Student student = new Student();  
 student.setName("Cypersoft");  
 student.setAge(21);  
 return student;  
}

@Bean: cho phép tạo ra 1 method của class để lưu trữ trên heap



Để lấy được object từ IOC thì dùng @Autowire





why do we need dependency injection in jpa spring boot

Ans1:

Dependency injection is an important concept in Spring Boot and JPA (Java Persistence API) for several reasons:

1. **Decoupling Components**: Dependency injection helps in decoupling components within your Spring Boot application. It allows you to define relationships between various components (such as controllers, services, repositories, etc.) without them needing to create instances of each other explicitly. This decoupling promotes modular, maintainable, and testable code.
2. **Reusability**: By injecting dependencies, you can reuse components across different parts of your application. For example, you can inject a service or repository into multiple controllers or services, promoting code reuse.
3. **Testability**: Dependency injection makes it easier to write unit tests for your components. You can easily create mock or stub implementations of dependencies and inject them for testing. This allows you to isolate the component you're testing and ensures that you are testing the component in isolation, which is a best practice for unit testing.
4. **Flexibility and Configurability**: With dependency injection, you can configure your application's behavior by changing the injected dependencies. For example, you can easily switch between different implementations of a repository (e.g., in-memory database vs. relational database) by changing the configuration without modifying the code that uses the repository.
5. **Encapsulation**: It helps in encapsulating the creation and management of dependencies within the Spring framework, which simplifies the code of your application. You don't need to worry about object creation, lifecycle management, or resource cleanup; Spring handles these aspects for you.
6. **AOP (Aspect-Oriented Programming)**: Spring Boot can leverage dependency injection to apply aspects (cross-cutting concerns like logging, security, and transactions) to specific components. AOP allows you to separate concerns and apply them consistently across your application.

When it comes to JPA and Spring Boot, dependency injection is crucial because JPA entities, repositories, and services often need to work together. For example, a service may use a repository to fetch or persist entities, and the entity classes themselves may have relationships with other entities. Dependency injection ensures that these components can be wired together seamlessly.

In summary, dependency injection in Spring Boot and JPA promotes modularity, testability, flexibility, and encapsulation in your application. It simplifies the management of dependencies, making your code cleaner, more maintainable, and easier to extend or modify.

Dependency injection (DI) in JPA (Java Persistence API) with Spring Boot is used to manage and inject the following components:

1. **EntityManager**: The EntityManager is responsible for managing the lifecycle of JPA entities, including persisting, merging, and removing them. It's typically injected into Spring services or repositories.

*@Autowired private EntityManager entityManager;*

1. **Repositories**: Spring Data JPA repositories are used to interact with the database. They can be injected into services or controllers to perform CRUD operations on entities.

*@Autowired private UserRepository userRepository;*

1. **Services**: Services encapsulate business logic and may interact with repositories. They can be injected into controllers or other services.

*@Autowired private UserService userService;*

1. **Custom Query Methods**: You can define custom query methods in your repositories, and Spring Data JPA will generate the query logic. These methods can be used within your services.

public interface UserRepository extends JpaRepository<User, Long> {

    List<User> findByFirstName(String firstName);

}

1. **Transaction Management**: Spring Boot uses DI for managing transactions. You can use annotations like **@Transactional** to specify the scope of a transaction.

@Transactional

public void performTransactionalOperation() {

    // Database operations here

}

1. **Entity Relationships**: You can define relationships between JPA entities (e.g., @ManyToOne, @OneToMany). These relationships are managed automatically by the JPA provider (e.g., Hibernate) and can be used in your service methods.

@Entity

public class Order {

    // ...

    @ManyToOne

    private Customer customer;

}

1. **Auditing**: Spring Boot can automatically populate audit fields like **createdBy**, **createdAt**, **lastModifiedBy**, and **lastModifiedAt** using DI. You can use annotations like **@CreatedBy** and **@LastModifiedBy** on fields.

javaCopy code

@EntityListeners(AuditingEntityListener.class) @Entity public class User { // ... @CreatedBy private String createdBy; }

1. **Validation**: Spring Boot can perform validation on entity objects before they are persisted. Validation annotations (e.g., **@NotNull**, **@Size**) can be used within entities.

javaCopy code

@Entity public class Product { // ... @NotNull @Size(min = 2, max = 255) private String name; }

1. **DTO (Data Transfer Object) Mapping**: You can use DI to map entities to DTOs and vice versa. Libraries like MapStruct can simplify this process.

javaCopy code

@Mapper(componentModel = "spring") public interface UserMapper { UserDto userToUserDto(User user); }

1. **Testing**: DI is essential for testing. You can inject mock repositories, services, or EntityManager for unit testing or integration testing.

javaCopy code

@MockBean private UserRepository userRepository;

In summary, dependency injection in JPA with Spring Boot simplifies the management of components, database interactions, transactions, and testing. It promotes modularity and separation of concerns, making your application more maintainable and testable.

Ans2:

Dependency injection (DI) is a design pattern that allows us to decouple our code and make it more maintainable and flexible. In JPA Spring Boot, DI is used to inject dependencies into our beans. This means that we can provide the dependencies that our beans need, rather than having them create their own dependencies.

There are several benefits to using DI in JPA Spring Boot:

* Testability: DI makes our code more testable, as we can easily mock or stub out dependencies in our tests. This makes it easier to test our code in isolation and ensure that it is working as expected.
* Reusability: DI makes our code more reusable, as we can easily reuse beans in different contexts. This can help us to reduce code duplication and make our code more maintainable.
* Modularity: DI makes our code more modular, as we can easily swap out dependencies without having to modify our code. This makes our code more flexible and adaptable to change.

Here are some specific examples of how DI can be used in JPA Spring Boot:

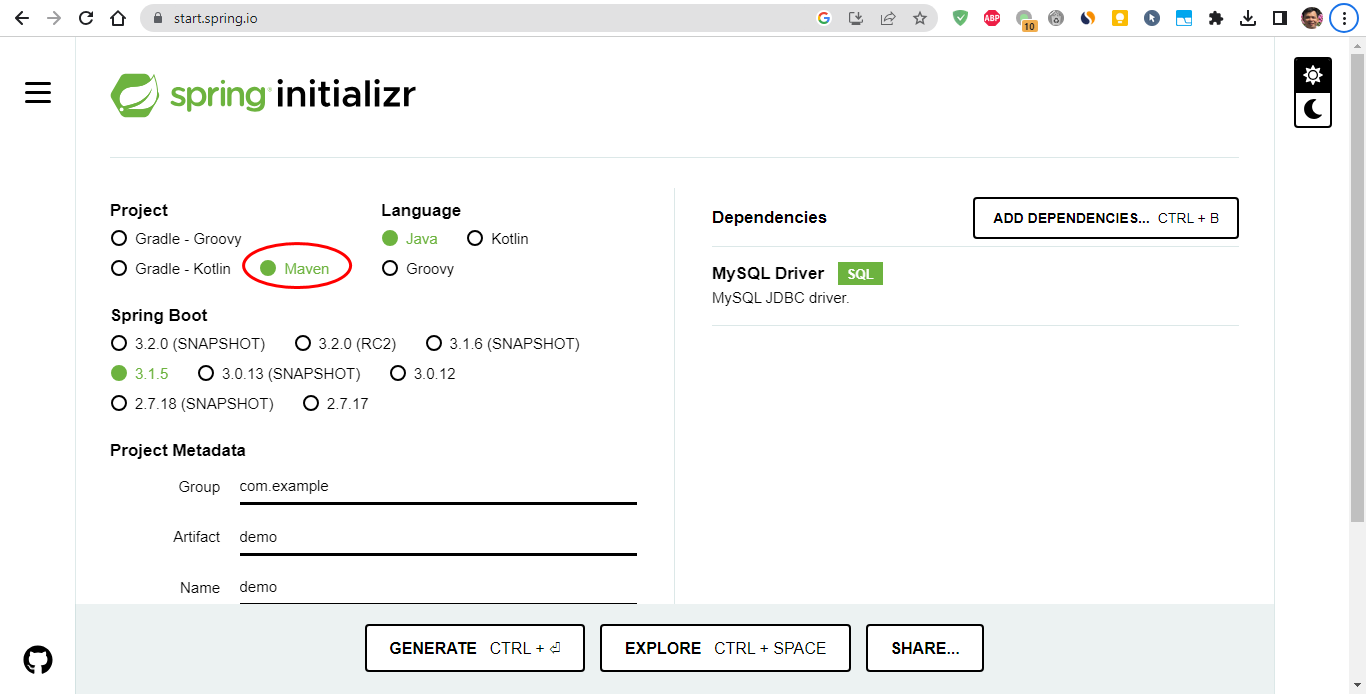
* We can inject a DataSource into our EntityManagerFactory bean. This allows us to easily change the database that we are using without having to modify our code.
* We can inject a Repository into our Service bean. This allows us to easily decouple our service layer from our database layer.
* We can inject a Service into our Controller bean. This allows us to easily decouple our controller layer from our service layer.

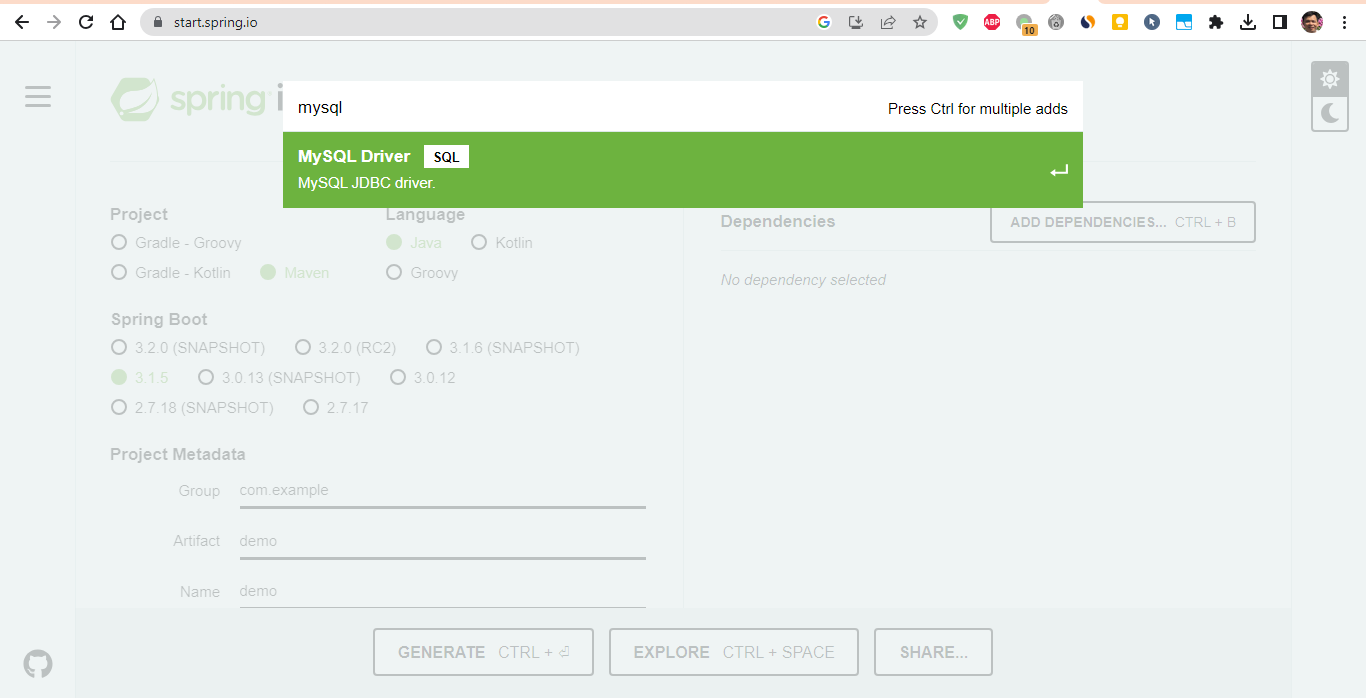
Overall, DI is a powerful design pattern that can help us to write more maintainable, flexible, and testable code in JPA Spring Boot.

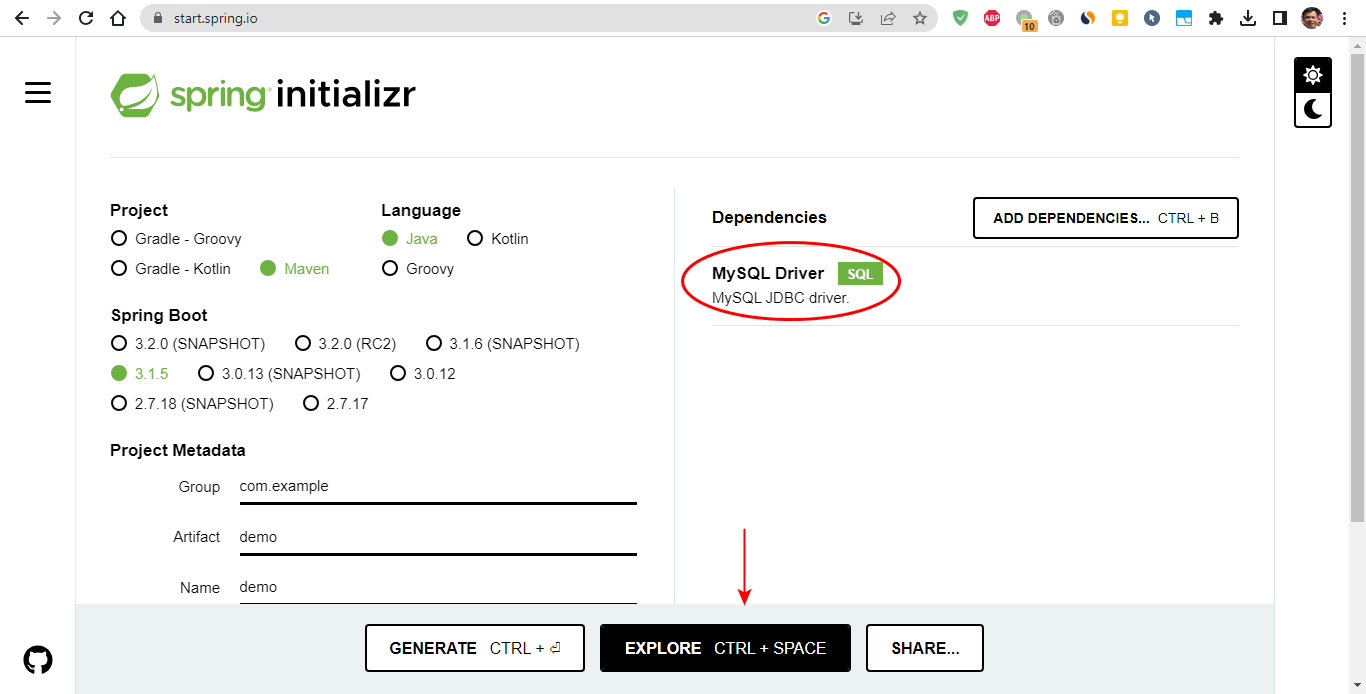
Safety considerations:

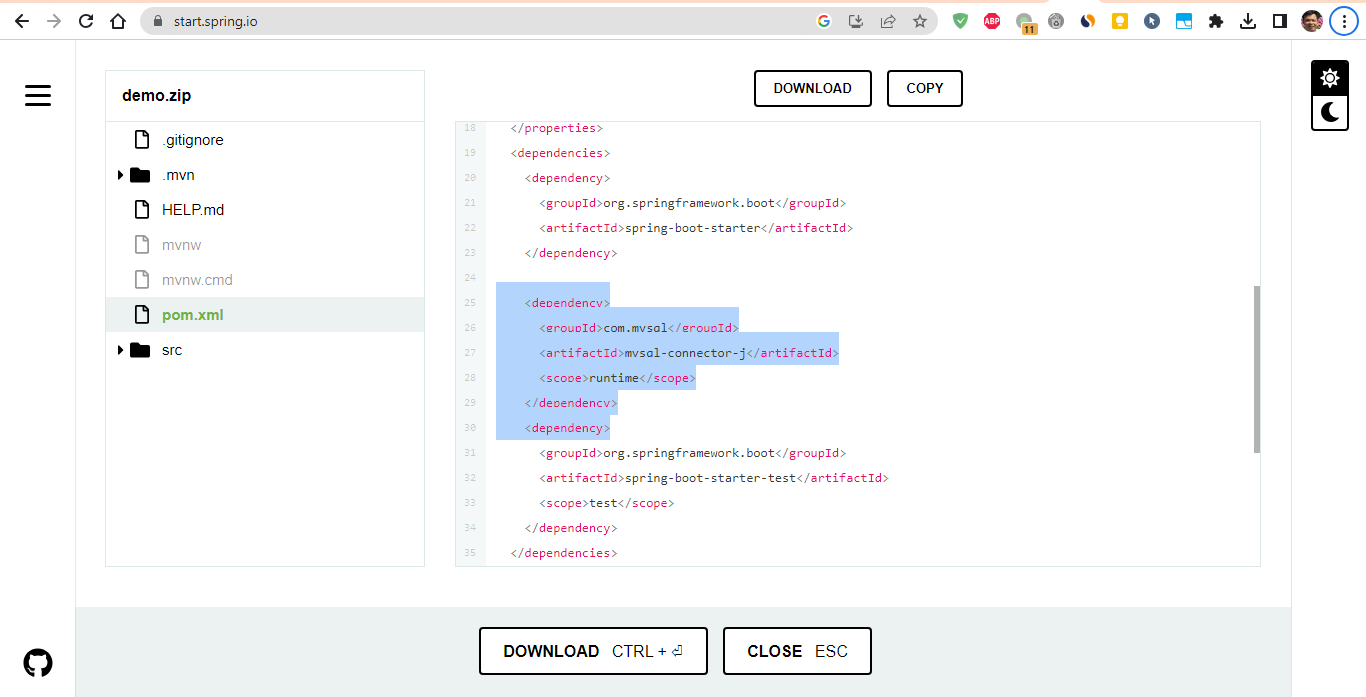
* It is important to use DI correctly to avoid introducing security vulnerabilities. For example, we should not inject dependencies that are not trusted.
* We should also be careful to avoid circular dependencies when using DI. Circular dependencies can lead to runtime errors.

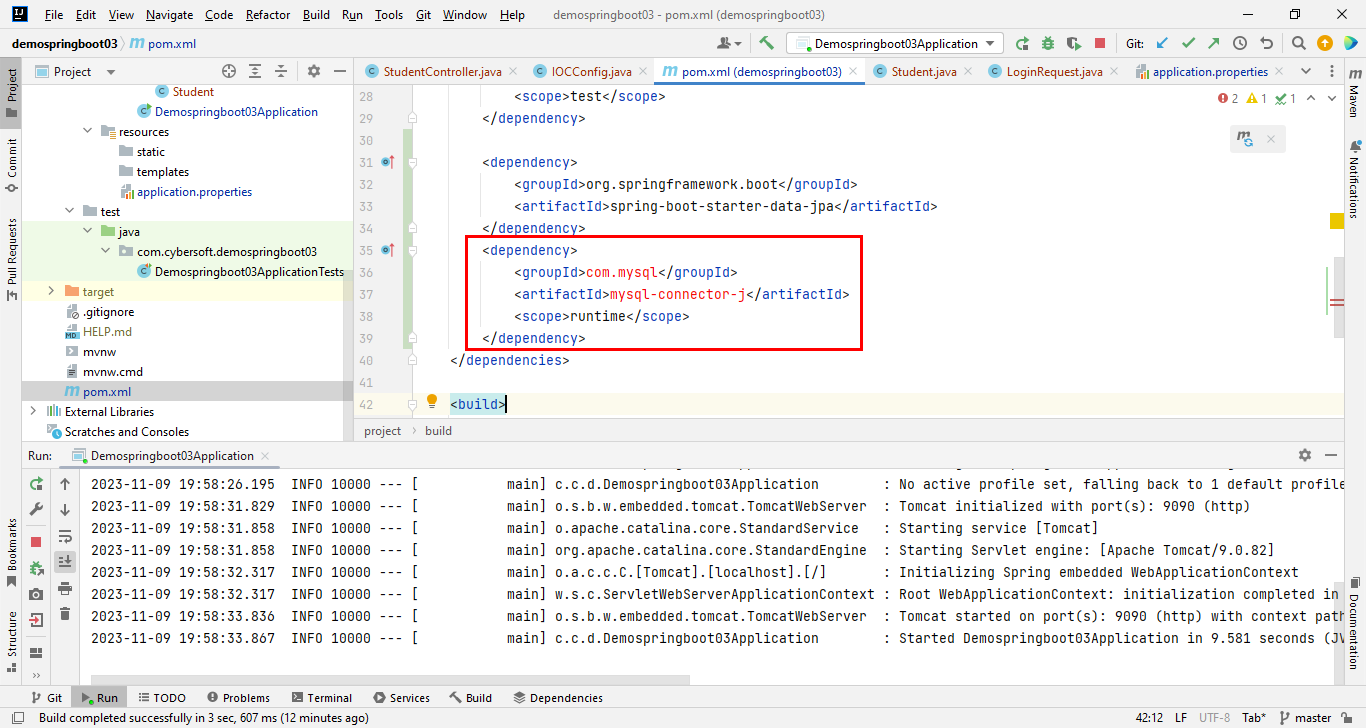
By following these safety considerations, we can use DI in JPA Spring Boot safely and effectively.











spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

<https://techmaster.vn/posts/36182/spring-boot-11-huong-dan-spring-boot-jpa-mysql>

**Demo**

Bây giờ chúng ta sẽ làm ứng dụng Demo các tính năng cơ bản với JpaRepository

Bước đầu tiên là config thông tin về MySQL trong application.properties

*application.properties*

spring.datasource.url=jdbc:mysql:*//localhost:3306/micro\_db?useSSL=false*

spring.datasource.username=root

spring.datasource.password=root

## Hibernate Properties

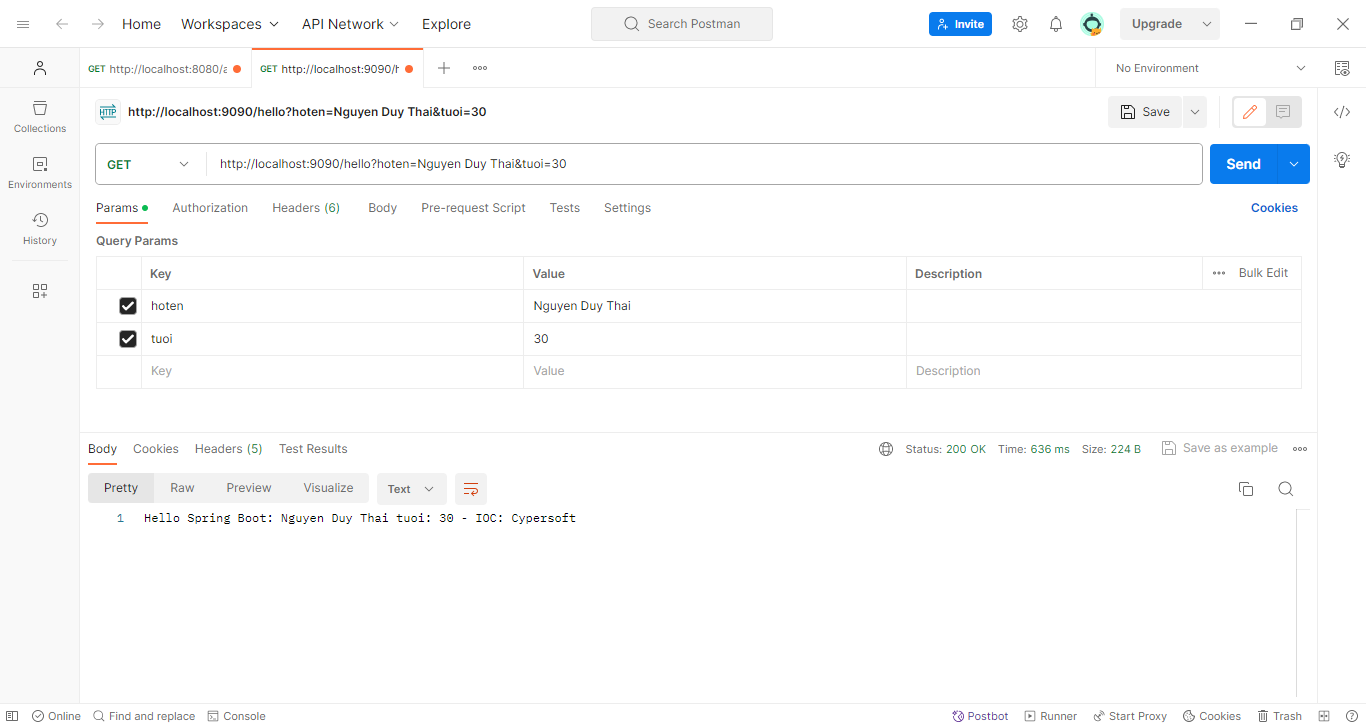
# The SQL dialect makes Hibernate generate better SQL for the chosen database

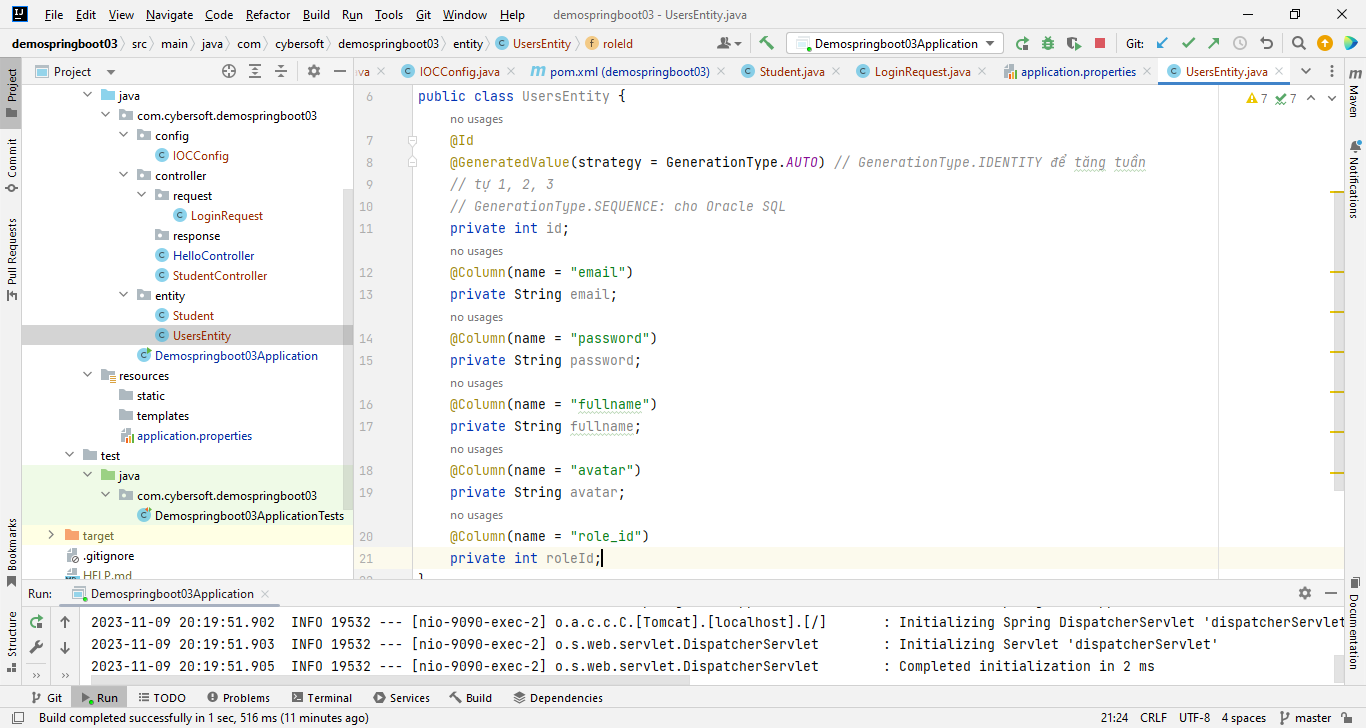
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

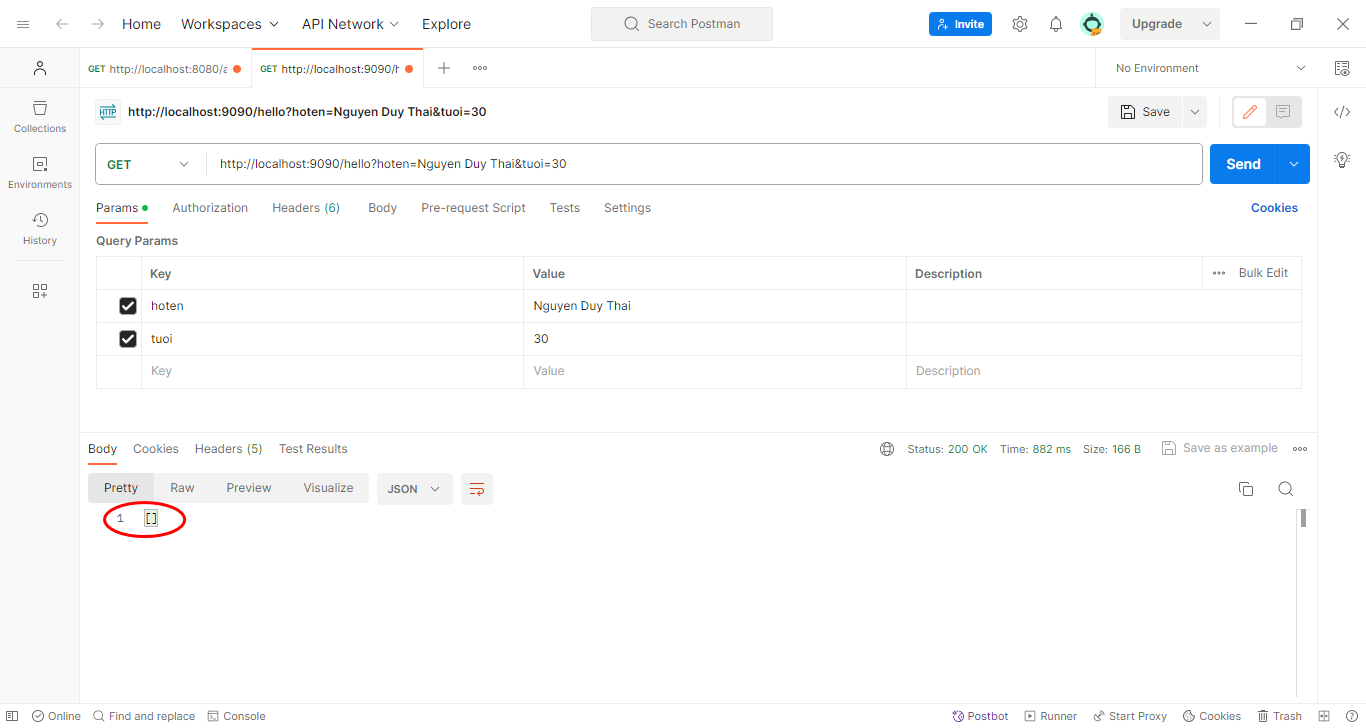
# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

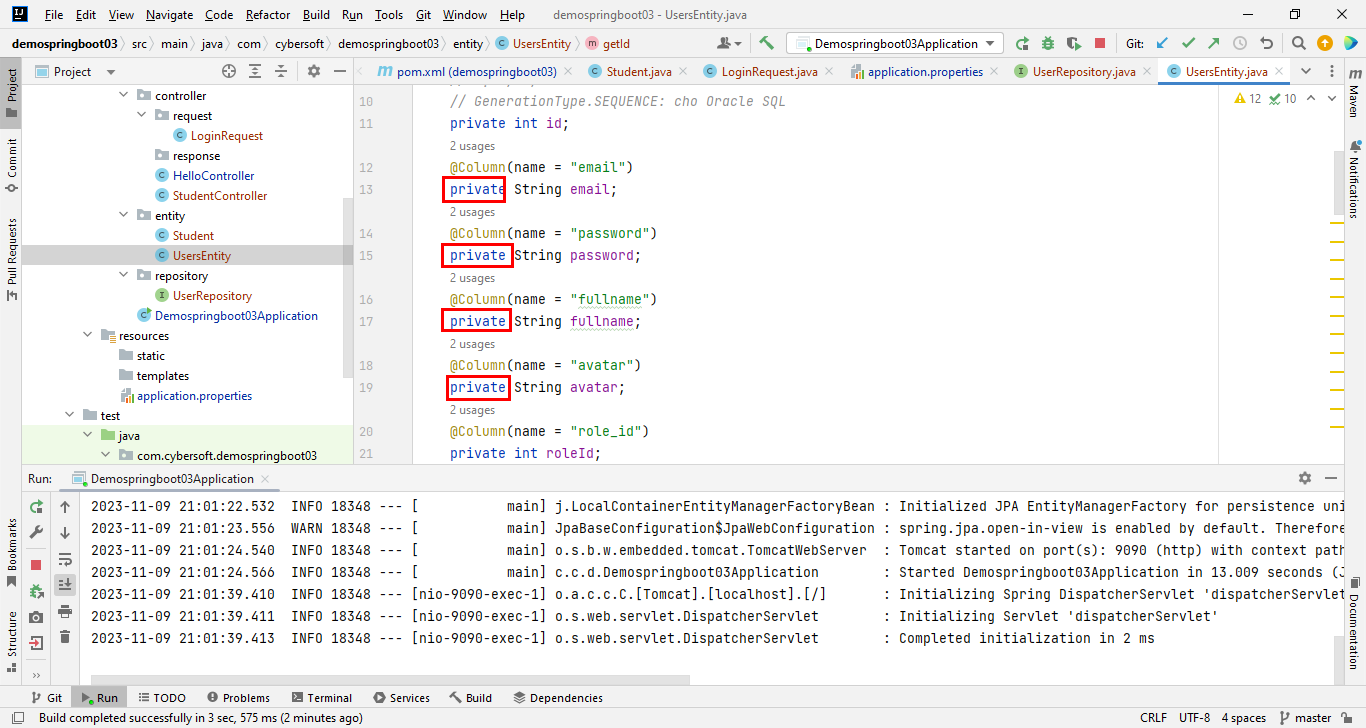
logging.level.org.hibernate = ERROR

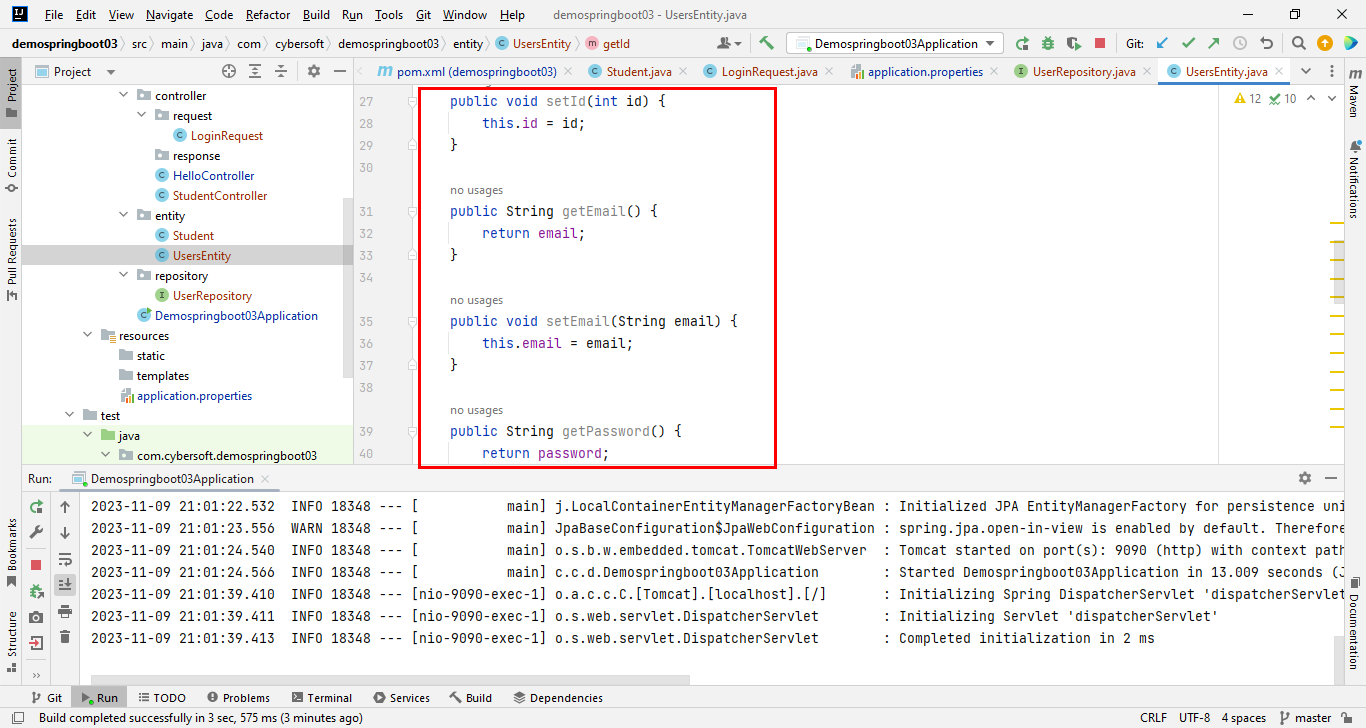






Cause: do biến là private nên không thể access được từ bên ngoài -> hoặc là dùng public hoặc là dùng getter/setter





<https://codelearn.io/sharing/hieu-ro-ve-dependency-injection>