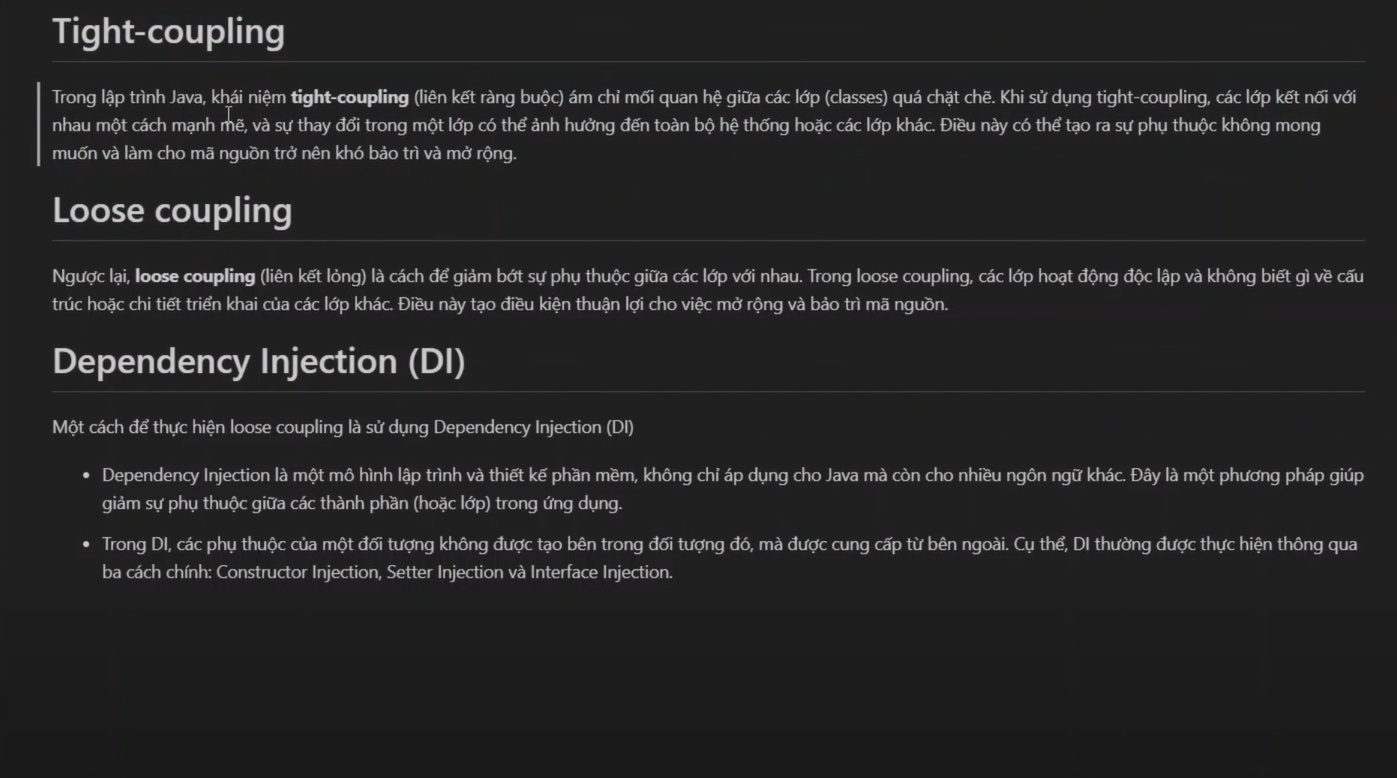
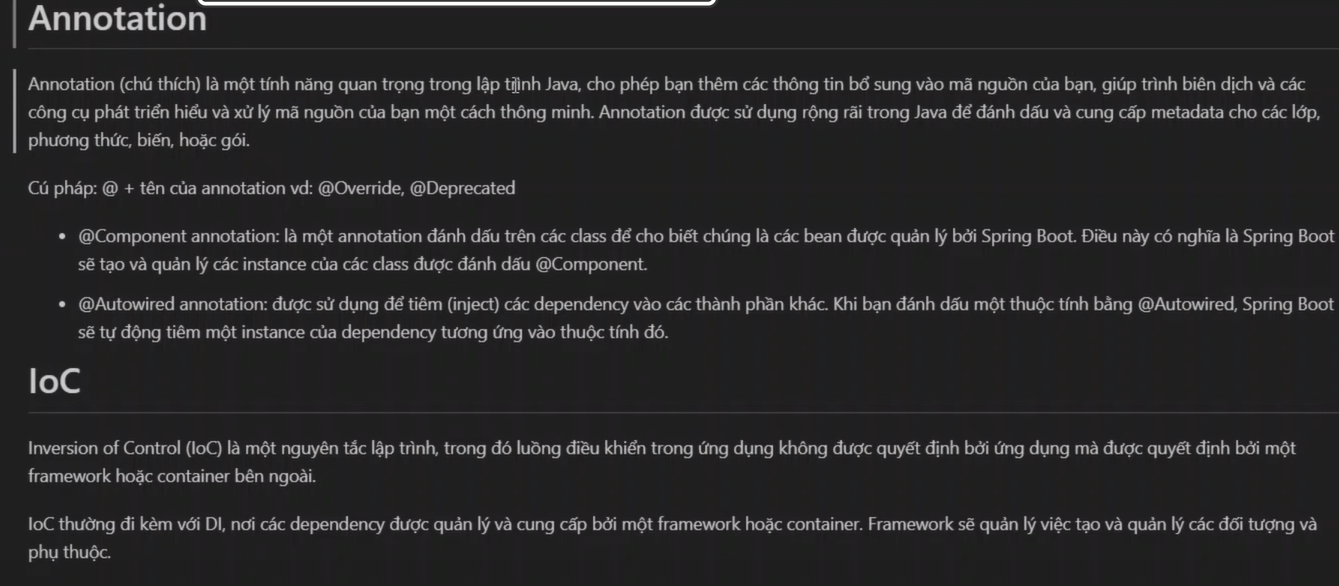
# **Sping boot**

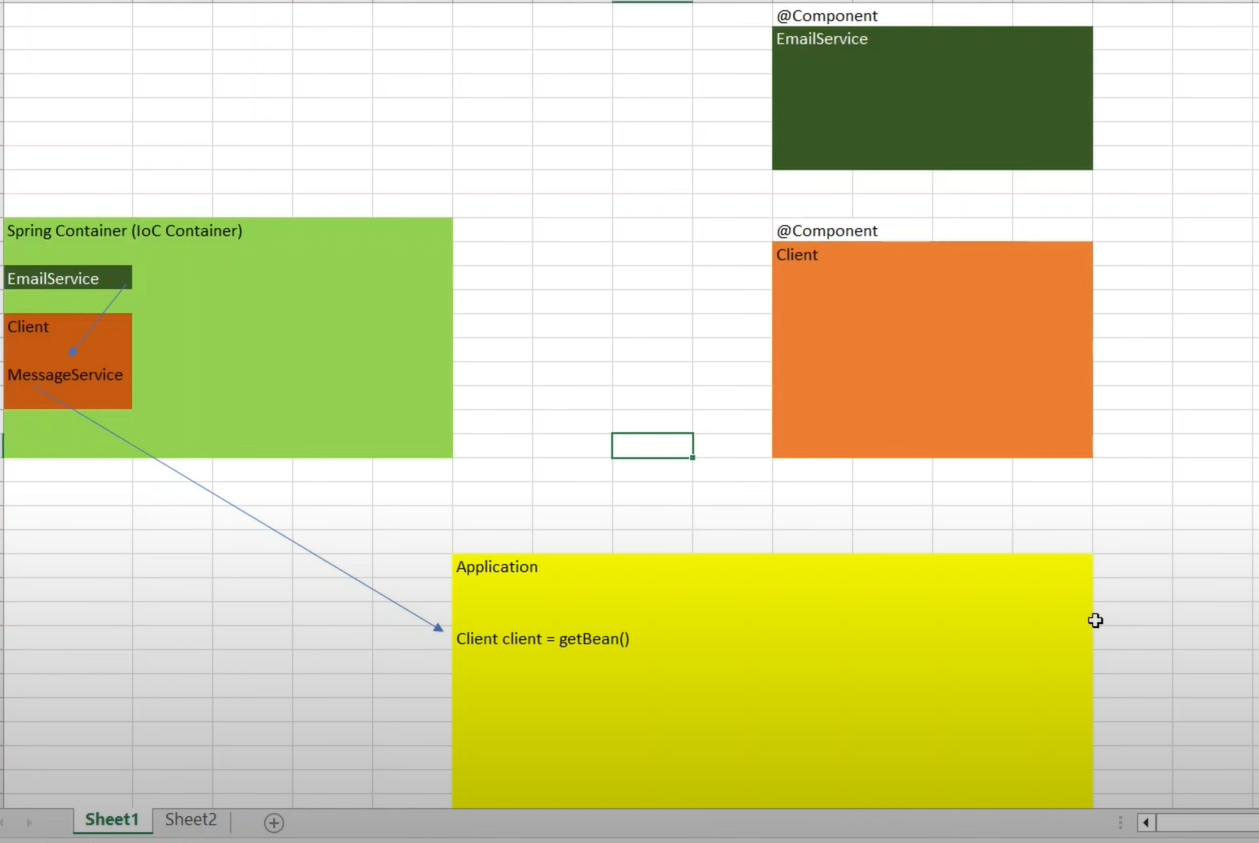
## DI( Dependentcy injection)



## IOC (inversion of control), annotation, @component



IOC tìm tất cả các bean implement thằng interface đó

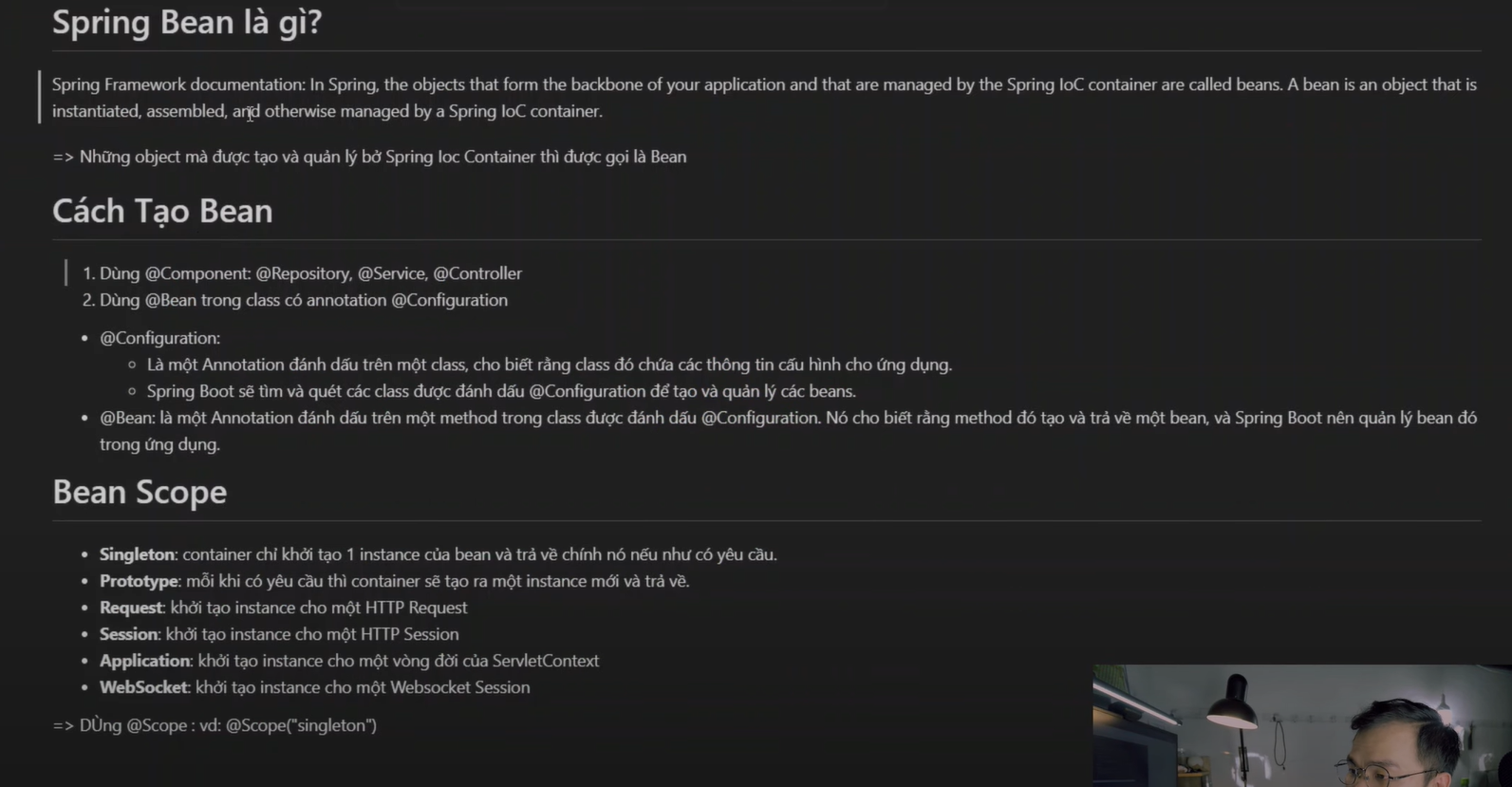


## **Giải quyết xung đột Bean, @Primary và @Qualifier**



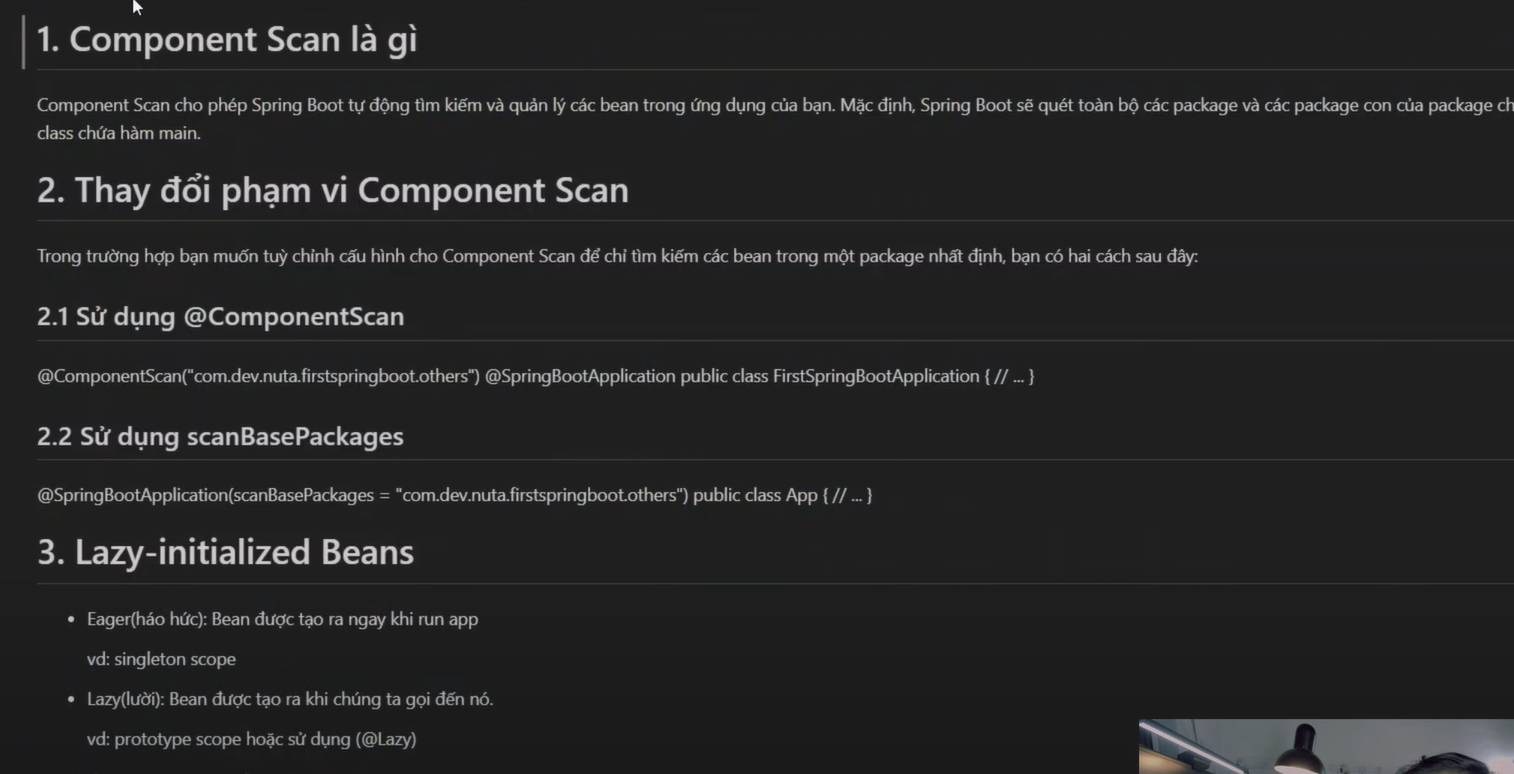
@Qualifer luôn được ưu tiên hơn @Primary

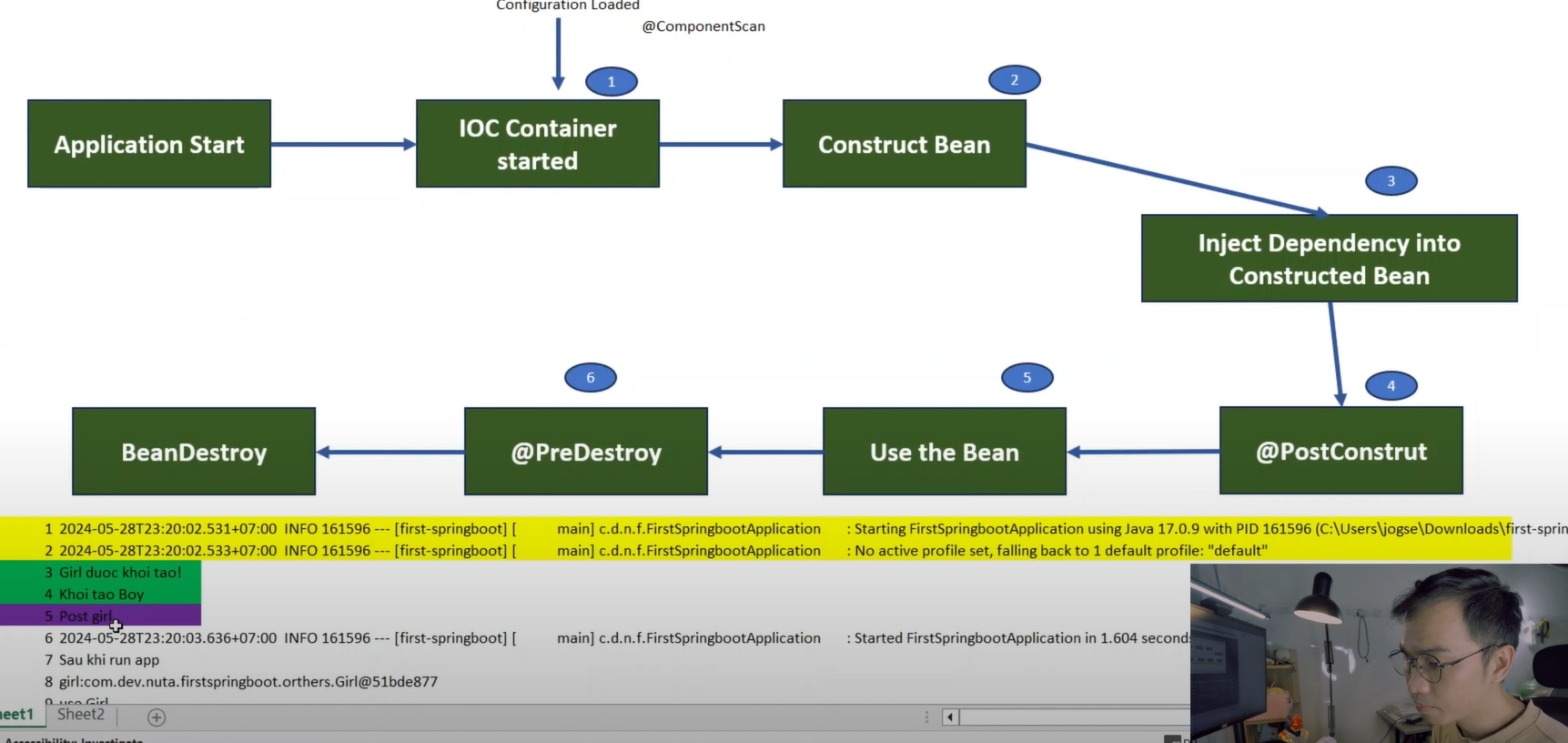
## Spring beans

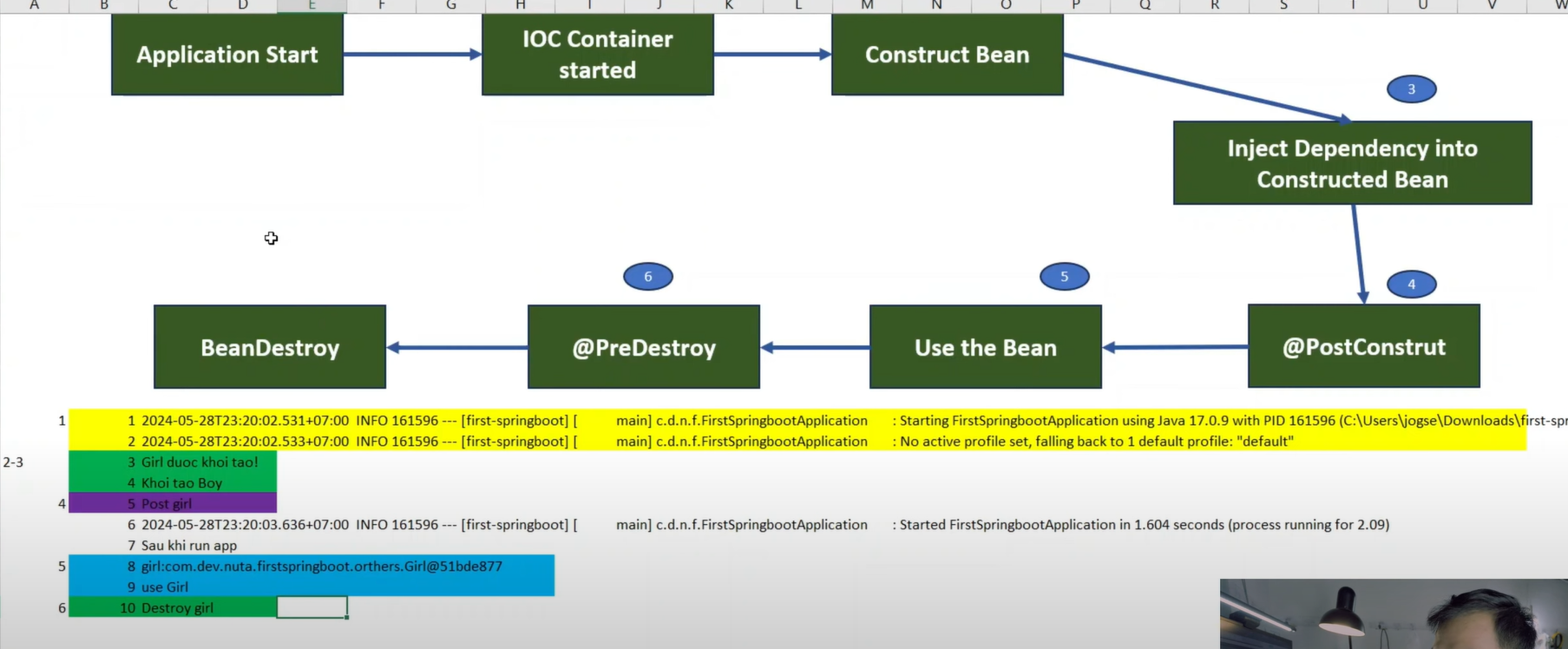


Tại sao phải dùng @Configuration @Bean ?

Khi ta cần dùng 1 thư viện bên ngoài và không thể thêm @Component cho nó nhưng @Configuration @Bean sẽ khắc phục điều đó. Nhưng spring boot cx tự động config khác nhiều lib mới.







# **Factory Design Pattern**

Factory Design Pattern được sử dụng khi chúng ta có một superclass có nhiều class con và cần phải trả về một trong các class con đó dựa trên một input nhất định. Design Pattern này giúp chúng ta khởi tạo class từ factory class thay vì khởi tạo trên chương trình bên client.

Trước tiên, chúng ta hãy tìm hiểu cách implement design pattern này trong java, và sau đó nêu ra các ưu điểm của nó. Design pattern này cũng được sử dụng trong JDK. Design pattern này còn có tên gọi khác là **Factory Method Design Pattern**

**Factory Design Pattern Super Class**

Super class trong factory design pattern có thể là một interface, một abstract class hoặc một class bình thường. Ví dụ trong đoạn code dưới đây, chúng ta có một abstract super class với phương thức override toString():

package com.journaldev.design.model;

public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

**Factory Design Pattern Sub Classes**

Ví dụ chúng ta có 2 subclass là PC và Server được implement như sau:

package com.journaldev.design.model;

public class PC extends Computer {

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

Lưu ý là cả 2 class này đều extend superclass là Computer

package com.journaldev.design.model;

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**Factory Class**

Sau khi đã viết xong các super class và sub-class, chúng ta có thể tạo factory class. Implementation đơn giản sẽ như sau:

package com.journaldev.design.factory;

import com.journaldev.design.model.Computer;

import com.journaldev.design.model.PC;

import com.journaldev.design.model.Server;

public class ComputerFactory {

public static Computer getComputer(String type, String ram, String hdd, String cpu){

if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);

else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd, cpu);

return null;

}

}

Một số điểm quan trọng về design pattern này là:

1. Chúng ta có thể giữ Factory class dưới dạng Singleton, hoặc giữ phương thức trả về class con dưới dạng static.
2. Các class con khác nhau có thể được tạo và trả về, tùy vào phương thức đầu vào. getComputer() là factory method.

Dưới đây là một chương trình client đơn giản có sử dụng design pattern này

package com.journaldev.design.test;

import com.journaldev.design.factory.ComputerFactory;

import com.journaldev.design.model.Computer;

public class TestFactory {

public static void main(String[] args) {

Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4 GHz");

Computer server = ComputerFactory.getComputer("server","16 GB","1 TB","2.9 GHz");

System.out.println("Factory PC Config::"+pc);

System.out.println("Factory Server Config::"+server);

}

}

Output của chương trình trên:

Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

**Ưu điểm của factory design pattern**

1. Factory design pattern cung cấp cách làm theo nguyên tắc "Program to an interface, not an implementation"
2. Factory design pattern tác việc khởi tạo các implementation class thực tế ra khỏi máy client. Factory design pattern làm cho code mạnh hơn, ít bị coupled hơn và dễ dàng mở rộng hơn. Ví dụ, chúng ta có thể dễ dàng thay đổi implementation của class PC, vì nó không dính líu gì đến code bên client.
3. Factory design pattern cung cấp một lớp abstraction giữa implementation class và class bên client thông qua kế thừa.

**Ví dụ về việc sử dụng factory design pattern trong**

**JDK**

Các class java.util.Calendar, ResourceBundle and NumberFormat có method getInstance() sử dụng Factory pattern. Phương thức valueOf() trong các warper class như Boolean, Integer, v.v...

Spring boot

**BeanFactory , ObjectMapper ,**

# Lazy loading và eager loading

1. Eager Loading (Tải ngay lập tức)

**Nên dùng khi:**

* Bạn **luôn luôn cần** dữ liệu liên quan
* Quan hệ **@OneToOne** (thường nên eager)
* Dữ liệu nhỏ và ít thay đổi (danh mục, tham chiếu)
* Số lượng bản ghi ít, không gây vấn đề hiệu suất

java

@Entity

public class Order {

@Id

private Long id;

@ManyToOne(fetch = FetchType.EAGER) *// Tải ngay thông tin khách hàng*

private Customer customer;

}

**Ưu điểm:**

* Giảm số lần truy vấn database
* Đơn giản hóa logic ứng dụng

**Nhược điểm:**

* Có thể tải dữ liệu thừa không dùng đến
* Gây vấn đề hiệu suất với dữ liệu lớn

2. Lazy Loading (Tải khi cần)

**Nên dùng khi:**

* Dữ liệu liên quan **không phải lúc nào cũng cần**
* Quan hệ **@OneToMany**, **@ManyToMany** (thường nên lazy)
* Dữ liệu lớn, phức tạp
* Ứng dụng web (tránh tải dữ liệu không cần thiết cho mọi request)

java

@Entity

public class Customer {

@Id

private Long id;

@OneToMany(fetch = FetchType.LAZY) *// Chỉ tải đơn hàng khi cần*

private List<Order> orders;

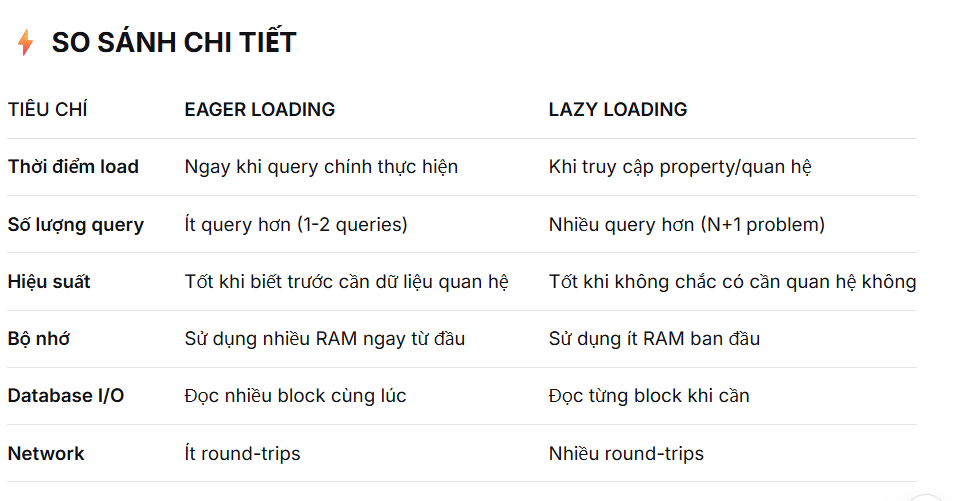
}

**Ưu điểm:**

* Tải nhanh entity chính
* Tiết kiệm bộ nhớ
* Linh hoạt trong các tình huống khác nhau

**Nhược điểm:**

* Có thể gặp **LazyInitializationException** nếu truy cập ngoài session
* Tăng số lần truy vấn database (N+1 problem)



Ví dụ:

Khi dùng eager loading:

* Khi ta load thông tin bảng user: thì sẽ load toàn bộ thông tin của bảng user và bảng liên quan tới nó, trong lần đầu query

Khi dùng lazy loading:

* Ở câu query đầu tiên: nó chỉ load mỗi thông tin bảng user
* Nhưng khi mình cần bất cứ thông tin gì của bảng Role thì nó sẽ query tiếp lần nữa cho đến hết (n lần querry)

Link tham khảo:

# Các annotation

@Entity

@Entity**là gì?**

* @Entity là một annotation đánh dấu một **class Java** là một Entity (thực thể).
* Một Entity đại diện cho một **bảng** (table) trong cơ sở dữ liệu.
* Mỗi **instance** (đối tượng) của class Entity đó đại diện cho một **hàng** (row) trong bảng.
* Mỗi **field** (trường) trong class Entity đại diện cho một **cột** (column) trong bảng.

**Tại sao phải sử dụng**@Entity**?**

Nó là cầu nối giữa thế giới Hướng Đối Tượng (OOP) trong Java và thế giới quan hệ (Relational) trong cơ sở dữ liệu. Khi bạn sử dụng @Entity, bạn đang nói với Spring Boot/JPA:

"Hãy quản lý class này và ánh xạ nó sang một bảng trong database. Tự động tạo các câu lệnh SQL (SELECT, INSERT, UPDATE, DELETE) để tương tác với đối tượng này mà tôi không cần phải viết chúng bằng tay."

**Cách sử dụng**@Entity

1. **Import annotation:** import jakarta.persistence.Entity; (với Spring Boot 3.x+) hoặc import javax.persistence.Entity; (với Spring Boot 2.x).
2. **Đánh dấu class:** Đặt annotation @Entity ngay trên khai báo class.
3. **Khai báo trường:** Khai báo các trường (fields) trong class. Mỗi trường sẽ tương ứng với một cột trong bảng.
4. **Khai báo khóa chính:** Bắt buộc phải có một trường được đánh dấu là khóa chính bằng @Id.

@Repository

@Repository**là gì?**

* @Repository là một annotation đánh dấu một class đảm nhận vai trò **tầng truy cập dữ liệu (Data Access Layer - DAL)** hoặc **tầng persistence**.
* Class được đánh dấu bằng @Repository chịu trách nhiệm **tương tác trực tiếp với cơ sở dữ liệu**: thực hiện các thao tác CRUD (Create, Read, Update, Delete), gọi stored procedures, thực thi các câu truy vấn, v.v.
* Về bản chất, @Repository là một @Component**đặc biệt**. Nó được kế thừa từ @Component, nghĩa là nó cũng sẽ được Spring tự động phát hiện và quản lý như một Spring Bean. Tuy nhiên, nó mang thêm ý nghĩa nghiệp vụ rõ ràng.

**Tại sao phải sử dụng**@Repository**?**

1. **Đánh dấu kiến trúc:** Nó giúp xác định rõ ràng kiến trúc của ứng dụng. Bạn biết ngay class nào chịu trách nhiệm cho việc truy cập dữ liệu.
2. **Xử lý ngoại lệ đặc biệt:** Đây là lợi ích QUAN TRỌNG NHẤT. Spring có một cơ chế xử lý ngoại lệ (exception translation) dành riêng cho các class được đánh dấu bằng @Repository. Nó tự động chuyển đổi các ngoại lệ cụ thể của cơ sở dữ liệu (như SQLException, PersistenceException) thành các ngoại lệ thuộc hệ thống phân cấp DataAccessException của Spring. Điều này giúp code của bạn:
   * **Độc lập với cơ sở dữ liệu:** Bạn có thể thay đổi từ H2 sang MySQL hay PostgreSQL mà không cần thay đổi code xử lý lỗi.
   * **Xử lý lỗi nhất quán:** Bạn chỉ cần bắt (catch) các DataAccessException thay vì phải biết từng loại ngoại lệ cụ thể của từng DB.

@Service

@RestController

# **Exception Handling and Validation**

## **Exception Handling**

@ControllerAdvice

@ExceptionHandler

@ControllerAdvice  
public class GlobalExceptionHandler {  
  
// @ExceptionHandler(value = RuntimeException.class)  
// ResponseEntity<ApiReponse<String>> runtimeExceptionHandler(RuntimeException e) {  
// ApiReponse<String> apiReponse = new ApiReponse<>();  
// apiReponse.setMessage(e.getMessage());  
// apiReponse.setCode(HttpStatus.INTERNAL\_SERVER\_ERROR.toString());  
// apiReponse.setResult(e.getMessage());  
// return ResponseEntity.badRequest().body(apiReponse);  
// }  
  
 @ExceptionHandler(value = AppException.class)  
 ResponseEntity<ApiReponse> runtimeExceptionHandler(AppException e) {  
 ApiReponse apiReponse = new ApiReponse<>();  
 ErrorCode errorCode = e.getErrorCode();  
  
 apiReponse.setCode(errorCode.getCode());  
 apiReponse.setMessage(errorCode.getMessage());  
 return ResponseEntity.*badRequest*().body(apiReponse);  
 }  
  
 @ExceptionHandler(value = MethodArgumentNotValidException.class)  
 ResponseEntity<ApiReponse> validationExceptionHandler(MethodArgumentNotValidException e) {  
 ApiReponse apiReponse = new ApiReponse();  
 String enumKey = e.getFieldError().getField();  
 apiReponse.setCode(999);  
 apiReponse.setMessage(e.getFieldError().getDefaultMessage());  
 return ResponseEntity.*badRequest*().body(apiReponse);  
 }  
  
}

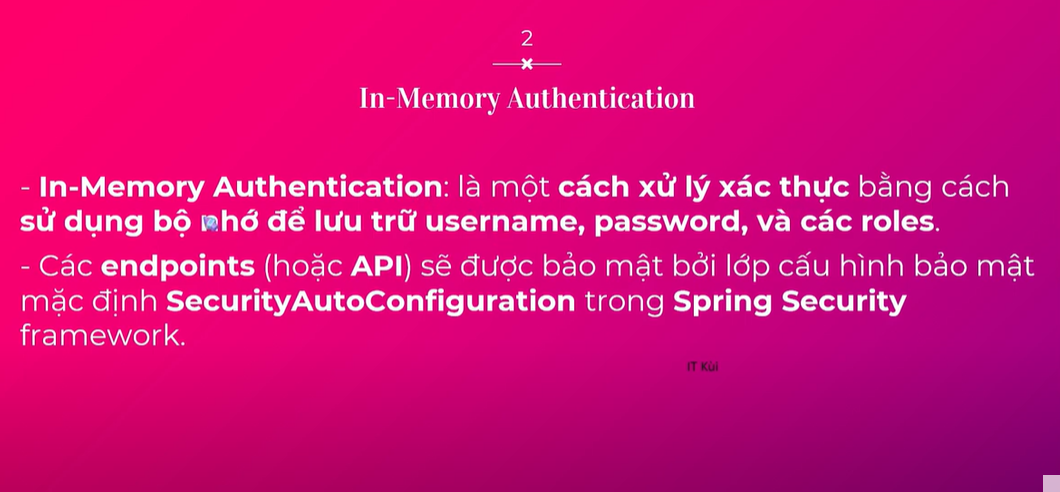
## **Validation**

@Valid

@Size

## **Chuẩn hóa response API - Advance Exception Handling - how to normalize api response**

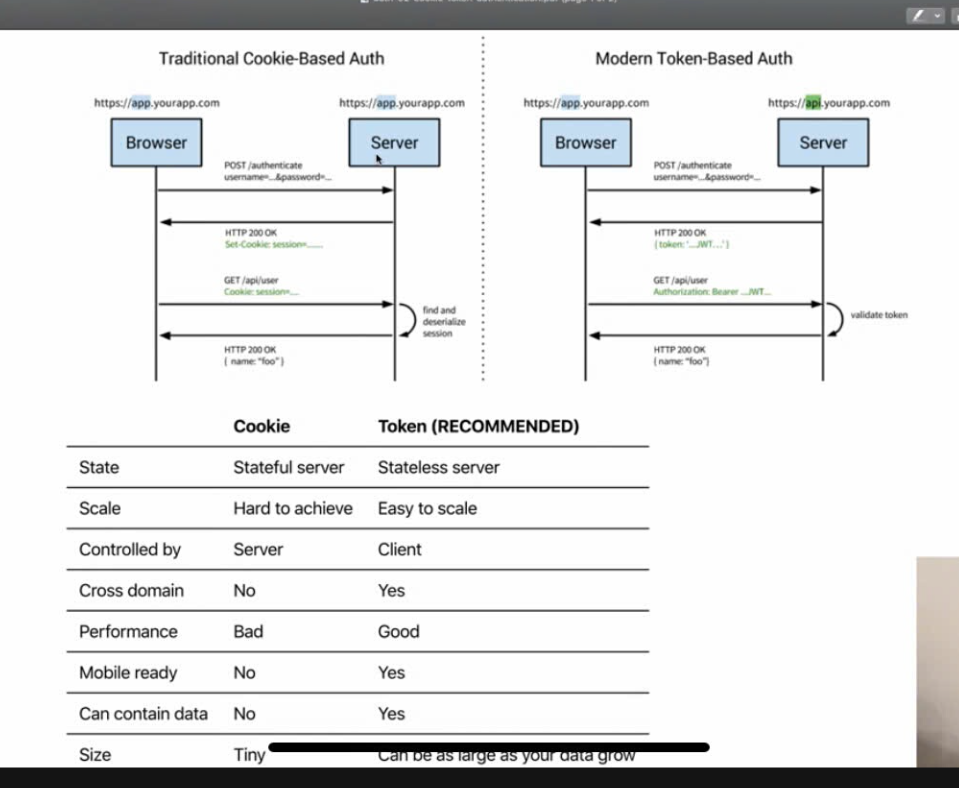
# Spring security

 Authentication: Who are you? Trả về lỗi 401

+ Phân biệt được Username nhưng ko có mật khẩu

+ Mật khẩu làm mấu chốt

Authoration: Are you allowed to do that? Trả về lỗi 403



Authentication:

Token:

Server:

+ Tạo token gửi về cho client

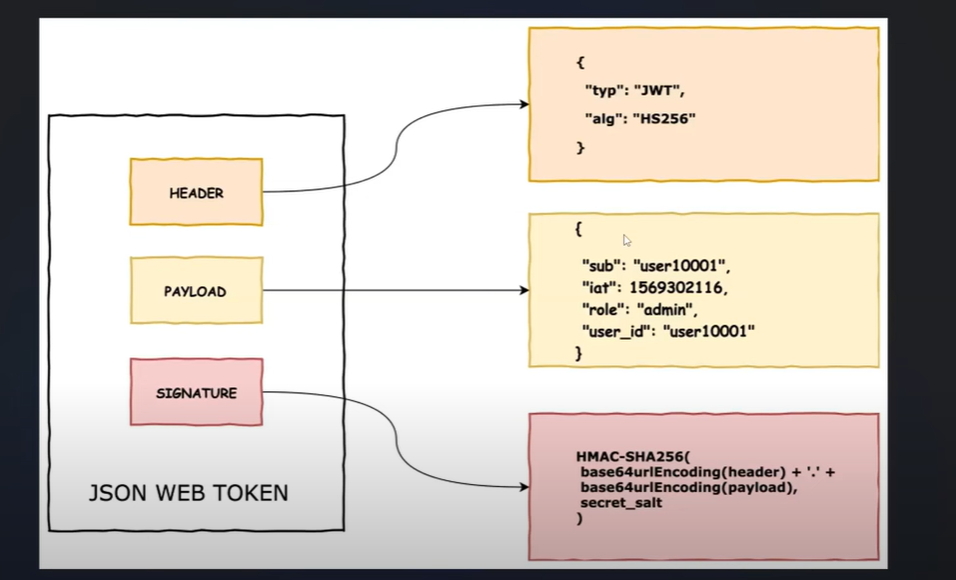
+ Phát hiện token có bị sửa từ phía client không

Client: Lưu token ở local Storage

Cockies:

Request: Post: Username, password tạo 1 session trong db

## JWT: JSON WEB TOKEN



Header:

Thuật toán mã hóa

Loại token: JWT

Payload:

Verify signature

Bao gồm các header, payload cùng với 1 secretKey

Server:

-Chi server có key để giải mã

Hạn chế việc tấn công khi bị đánh cắp token:

-Giảm exp

Ví dụ:

Thư viện

<dependency>  
 <groupId>com.nimbusds</groupId>  
 <artifactId>nimbus-jose-jwt</artifactId>  
 <version>9.30.1</version>  
</dependency>

Generate 1 token khi client đăng nhập:

Xử lý tạo Header,Payload,Signature

public String generateToken(String username) {  
 JWSHeader header = new JWSHeader(JWSAlgorithm.*HS256*);  
  
 JWTClaimsSet jwtClaimSet = new JWTClaimsSet.Builder()  
 .subject(username)  
 .issuer("localhost")  
 .issueTime(new Date())  
 .expirationTime(new Date(  
 Instant.*now*().plus(1, ChronoUnit.*MINUTES*).toEpochMilli()  
 ))  
 .build();  
 Payload payload = new Payload(jwtClaimSet.toJSONObject());  
  
 JWSObject jwsObject = new JWSObject(header,payload);  
 try {  
 jwsObject.sign(new MACSigner(key.getBytes()));  
 return jwsObject.serialize();  
 } catch (JOSEException e) {  
 logger.error("Cannot create JWT object", e.getMessage());  
 throw new RuntimeException(e);  
 }  
}

Tạo request xác thực token

public class IntrospectRequest {  
 String token;  
}

Xử lý logic xác thực ở tầng service

public IntrospectResponse introspect(IntrospectRequest introspectRequest) {  
 var token = introspectRequest.getToken();  
 try {  
 JWSVerifier verifier = new MACVerifier(key.getBytes());  
 SignedJWT signedJWT = SignedJWT.*parse*(token);  
 var verified = signedJWT.verify(verifier);  
  
 Date expiration = signedJWT.getJWTClaimsSet().getExpirationTime();  
  
 return IntrospectResponse.*builder*()  
 .valid(verified && expiration.after(new Date()))  
 .build();  
 } catch (JOSEException | ParseException e) {  
 logger.error("Cannot create JWT object", e.getMessage());  
 throw new RuntimeException(e);  
 }  
}

Tạo URL check token

@GetMapping("/introspect")  
 public ApiReponse<IntrospectResponse> checkValidToken(@RequestBody IntrospectRequest introspectRequest) {  
 IntrospectResponse introspectResponse = authenticationService.introspect(introspectRequest);  
 ApiReponse apiReponse = new ApiReponse();  
 apiReponse.setCode(99999);  
 apiReponse.setResult(introspectResponse);  
 return apiReponse;  
 }

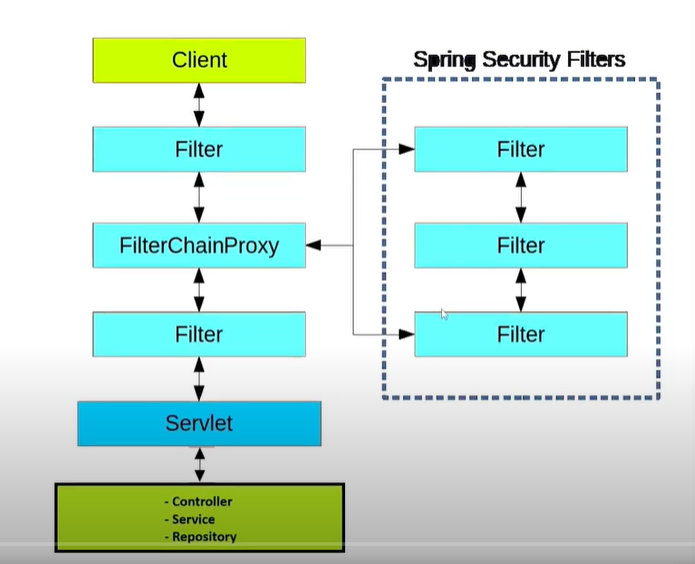
## So sánh khi sử dụng JSON và XML

|  |  |  |
| --- | --- | --- |
| Tiêu Chí | XML (eXtensible Markup Language) | JSON (JavaScript Object Notation) |
| **Định dạng** | Ngôn ngữ đánh dấu (**Markup Language**), sử dụng thẻ (**tags**). | Định dạng đối tượng dữ liệu (**Data Object Format**), sử dụng cặp **key-value**. |
| **Khả năng đọc** | Có thể đọc được, nhưng cồng kềnh và dài dòng hơn. | **Rất dễ đọc** và viết đối với con người. Gần giống với cấu trúc dữ liệu trong code. |
| **Kích thước** | **Nặng hơn** do sử dụng thẻ đóng/mở, các thuộc tính. Tệp có xu hướng lớn. | **Nhẹ hơn đáng kể** do cú pháp ngắn gọn. Giảm băng thông và thời gian tải. |
| **Phân tích cú pháp (Parsing)** | Phức tạp, yêu cầu **XML parser** (DOM, SAX). **Chậm hơn** và tốn nhiều tài nguyên CPU & bộ nhớ hơn. | **Cực kỳ dễ dàng và nhanh chóng**. Có thể parsed bằng JSON.parse() trong JavaScript hoặc thư viện đơn giản trong mọi ngôn ngữ. |
| **Hỗ trợ Kiểu Dữ Liệu** | **Hạn chế**. Mọi thứ đều là chuỗi, cần được chuyển đổi. | **Hỗ trợ nguyên thủy**: string, number, boolean, null, array, object. |
| **Cấu trúc Dữ Liệu** | **Rất linh hoạt và mạnh mẽ**. Hỗ trợ **schema** (XSD, DTD) để **xác thực (validate)** cấu trúc và kiểu dữ liệu một cách chặt chẽ. | Linh hoạt nhưng **không có schema chặt chẽ mặc định**. Validation phụ thuộc vào logic ứng dụng (JSON Schema là một tiêu chuẩn bổ sung). |
| **Khả năng Mở rộng** | **Tuyệt vời**. Được thiết kế cho dữ liệu phức tạp, có cấu trúc, và có thể mở rộng với namespaces. | Tốt, nhưng đơn giản hơn. Không có khái niệm namespace mặc định. |
| **Hỗ trợ Chú thích/Metadata** | **Có**. Thông qua attributes, comments, và processing instructions. | **Không**. Chỉ có dữ liệu thuần túy. |
| **Bảo mật** | Có lịch sử lỗ hổng liên quan đến **XML External Entity (XXE)** injection và các cuộc tấn công phức tạp khác. | **An toàn hơn** về mặt thiết kế. Ít lỗ hổng phức tạp liên quan đến parsing. |

## **Cấu trúc của spring security - Spring Security Architecture and config to authenticate with JWT**

### **Authentication**

**Cấu trúc:**



**Luồng xử lý request trong Spring Security**

**1. Client → Filter → FilterChainProxy → Servlet**

* **Client**: Gửi HTTP request
* **Filter**: Các filter thông thường (không phải Spring Security)
* **FilterChainProxy**: Là trái tim của Spring Security, quản lý tất cả các security filter

**2. Spring Security Filters**

Đây là chuỗi các filter chuyên biệt của Spring Security:

* **Authentication filters**: Xác thực người dùng
* **Authorization filters**: Kiểm tra quyền truy cập
* **Security context filters**: Duy trì thông tin security
* **CSRF protection**: Bảo vệ chống tấn công CSRF
* **CORS handling**: Xử lý Cross-Origin Resource Sharing

**3. Servlet → Spring MVC Layers**

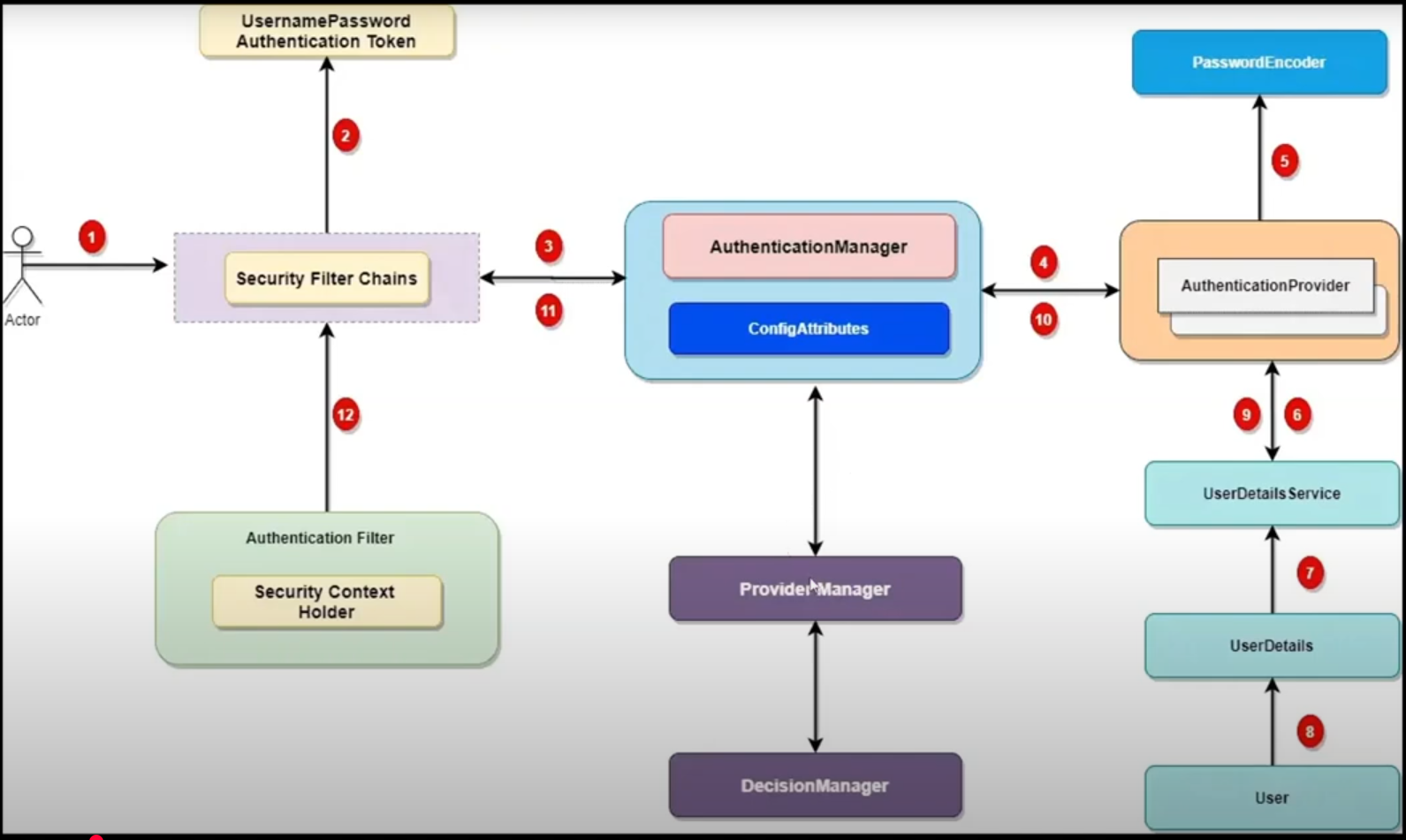
Sau khi qua security filters, request đến:

* **Controller**: Xử lý business logic cấp cao
* **Service**: Thực hiện nghiệp vụ
* **Repository**: Tương tác với database

**Đặc điểm quan trọng:**

* **FilterChainProxy** điều phối toàn bộ security filter chain
* Các security filter được thực thi **trước** khi request đến Servlet
* Mỗi filter có nhiệm vụ cụ thể trong quá trình bảo mật
* Sau khi qua tất cả security filters, request mới được chuyển đến Spring MVC

### Filters



**1. Security Filter Chains - Authentication Filter**

text

Authentication Filter → Security Context Holder

* **Authentication Filter**: Intercept request và tạo UsernamePasswordAuthenticationToken
* **Security Context Holder**: Lưu trữ thông tin authentication cho current thread

**2. AuthenticationManager**

text

AuthenticationManager → Provider Manager → AuthenticationProvider

* **AuthenticationManager**: Interface chính quản lý xác thực
* **Provider Manager**: Implementation chính, delegate cho các AuthenticationProvider
* **AuthenticationProvider**: Xử lý cụ thể loại xác thực

**3. AuthenticationProvider & UserDetails**

text

AuthenticationProvider → UserDetails Service → UserDetails

* **AuthenticationProvider**: So sánh credentials từ request với database
* **UserDetails Service**: Load user từ database theo username
* **UserDetails**: Interface chứa thông tin user (username, password, authorities)

**4. PasswordEncoder & Authorization**

text

PasswordEncoder ←→ AuthenticationProvider

ConfigAttributes → DecisionManager

* **PasswordEncoder**: Mã hóa và so sánh password
* **ConfigAttributes**: Cấu hình security rules (@PreAuthorize, URL patterns)
* **DecisionManager**: Quyết định authorization dựa trên ConfigAttributes

**Luồng xử lý chi tiết:**

1. **Authentication Filter** chặn login request
2. Tạo **UsernamePasswordAuthenticationToken** (chưa authenticated)
3. Gọi **AuthenticationManager**
4. **ProviderManager** tìm **AuthenticationProvider** phù hợp
5. **AuthenticationProvider** gọi **UserDetailsService** để load user
6. **PasswordEncoder** verify password
7. Nếu thành công, tạo **AuthenticationToken** đã authenticated
8. Lưu vào **SecurityContextHolder**
9. **Decision Manager** kiểm tra authorization dựa trên **ConfigAttributes**

Thư viện: oauth2

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>  
</dependency>

@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
  
 @NonFinal  
 @Value("${signerKey}")  
 protected String key;  
  
 @Bean  
 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
  
 http.authorizeHttpRequests(request ->  
 request.requestMatchers(HttpMethod.*POST*, "/users").permitAll()  
 .requestMatchers(HttpMethod.*GET*, "/auth/\*\*").permitAll()  
 .anyRequest().authenticated());  
  
 // Đăng ký 1 provider manager : Lấy token đăng ký được để xác thực các endpoint khác  
 http.oauth2ResourceServer(oauth ->  
 oauth.jwt(jwtConfigurer -> jwtConfigurer.decoder(jwtDecoder()))  
 );  
  
 // Tắt cow chế csrf  
 http.csrf(httpSecurityCsrfConfigurer -> httpSecurityCsrfConfigurer.disable());  
 return http.build();  
 }  
  
  
 //Decode token xác thực tính hợp lệ  
 @Bean  
 JwtDecoder jwtDecoder() {  
 SecretKeySpec secretKeySpec = new SecretKeySpec(key.getBytes(), "HS256");  
 return NimbusJwtDecoder.*withSecretKey*(secretKeySpec)  
 .macAlgorithm(MacAlgorithm.*HS256*).build();  
 };  
  
}

### Authorization

// Tạo token gửi về cho client trong lần đăng nhập ầu tiên  
public String generateToken(User user) {  
 JWSHeader header = new JWSHeader(JWSAlgorithm.*HS256*);  
  
 JWTClaimsSet jwtClaimSet = new JWTClaimsSet.Builder()  
 .subject(user.getUsername())  
 .issuer("localhost")  
 .issueTime(new Date())  
 .expirationTime(new Date(  
 Instant.*now*().plus(1, ChronoUnit.*MINUTES*).toEpochMilli()  
 ))  
 // Bổ sung scope\_roles trong token  
 .claim("scope", buildScope(user))  
 .build();  
 Payload payload = new Payload(jwtClaimSet.toJSONObject());  
  
 JWSObject jwsObject = new JWSObject(header,payload);  
 try {  
 jwsObject.sign(new MACSigner(key.getBytes()));  
 return jwsObject.serialize();  
 } catch (JOSEException e) {  
 logger.error("Cannot create JWT object", e.getMessage());  
 throw new RuntimeException(e);  
 }  
}  
  
// Tạo scope trong token để chứa list thông tin roles  
public String buildScope(User user){  
 StringJoiner stringJoiner = new StringJoiner(" ");  
 if(!CollectionUtils.*isEmpty*(user.getRoles())){  
 user.getRoles().forEach(role -> stringJoiner.add(role));  
 }  
 return stringJoiner.toString();  
}

Sau khi tạo token bao gồm các thông tin chính và roles SecurityContextHolder sẽ tự map data của token với data của spring security

http.authorizeHttpRequests(request ->  
 request.requestMatchers(HttpMethod.*POST*, "/users").permitAll()  
 // Lúc này ta có thể dùng hasAuthority() để lấy roles của User và xác định endpoint  
 .requestMatchers(HttpMethod.*GET*, "/users").hasAuthority("SCOPE\_USER")  
 .requestMatchers(HttpMethod.*GET*, "/auth/\*\*").permitAll()  
 .anyRequest().authenticated());

### SecurityContextHolder

Tài liệu tham khảo : [Method Security :: Spring Security](https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html#authorization-expressions)

**@PreAuthorize - Kiểm tra trước khi thực thi method**

**Cơ bản:**

**java**

**@PreAuthorize("hasRole('ADMIN')")**

**public void deleteUser(Long userId) {**

**userRepository.deleteById(userId);**

**}**

**Với biểu thức SpEL:**

**java**

***// Check role***

**@PreAuthorize("hasAuthority('ROLE\_ADMIN')")**

**public List<User> getAllUsers() {**

**return userRepository.findAll();**

**}**

***// Check multiple roles***

**@PreAuthorize("hasAnyRole('ADMIN', 'MANAGER')")**

**public void updateUser(User user) {**

**userRepository.save(user);**

**}**

***// Check permission***

**@PreAuthorize("hasPermission(#user, 'WRITE')")**

**public void saveUser(User user) {**

**userRepository.save(user);**

**}**

**Truy cập tham số:**

**java**

***// #username tham chiếu đến tham số method***

**@PreAuthorize("#username == authentication.principal.username")**

**public User findUserByUsername(String username) {**

**return userRepository.findByUsername(username);**

**}**

***// Multiple parameters***

**@PreAuthorize("#user.id == authentication.principal.id or hasRole('ADMIN')")**

**public void updateUserProfile(User user, boolean isAdmin) {**

**userRepository.save(user);**

**}**

**@PostAuthorize - Kiểm tra sau khi thực thi method**

**Kiểm tra return value:**

java

@PostAuthorize("returnObject.owner == authentication.name")

public User findUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

@PostAuthorize("returnObject.status != 'DELETED'")

public Document getDocument(Long id) {

return documentRepository.findById(id).orElse(null);

}

## **Lưu Access Token ở đâu? Được nhưng chưa đủ**

Access Token là một phần quan trọng trong bảo mật API, giúp xác thực và phân quyền người dùng. Tuy nhiên, nếu lưu trữ Access Token không đúng cách, ứng dụng có thể bị tấn công, gây rò rỉ dữ liệu. Trong bài viết này, chúng ta sẽ tìm hiểu các cách phổ biến để lưu trữ Access Token, phân tích ưu nhược điểm của từng phương pháp và đề xuất giải pháp tốt nhất để bảo vệ thông tin người dùng.

**Các phương pháp lưu trữ Access Token phổ biến**

## **1. Lưu trong Local Storage**

* Access Token được lưu trữ trong localStorage, có thể truy cập từ JavaScript trên trình duyệt.
* Khi cần sử dụng, ứng dụng sẽ lấy token từ localStorage và gửi trong các yêu cầu HTTP. **Ưu điểm:**
* Dễ triển khai và truy xuất từ bất kỳ trang nào trong ứng dụng.
* Token tồn tại sau khi reload trang hoặc đóng/mở lại trình duyệt. **Nhược điểm:**
* Dễ bị tấn công XSS (Cross-Site Scripting): Nếu một kẻ tấn công chèn được mã độc vào ứng dụng (ví dụ qua form nhập liệu hoặc tập lệnh bên ngoài), họ có thể truy cập localStorage và đánh cắp token.
* Không thể tự động gửi token trong mỗi request như Cookie.
* Không bảo vệ khỏi các cuộc tấn công CSRF. **Khi nào nên dùng?** Không khuyến nghị sử dụng Local Storage để lưu Access Token trong các ứng dụng yêu cầu bảo mật cao.

*Nguồn: Internet*

## **2. Lưu trong Session Storage**

Token được lưu trong sessionStorage, chỉ tồn tại trong phiên làm việc của trình duyệt. **Ưu điểm:**

* Ít rủi ro hơn so với localStorage vì token bị xóa khi người dùng đóng trình duyệt.
* Hạn chế được phần nào nguy cơ XSS nếu kết hợp với cơ chế bảo vệ tốt. **Nhược điểm:**
* Vẫn có thể bị tấn công XSS nếu ứng dụng không bảo vệ đúng cách.
* Token bị mất khi người dùng reload trang. **Khi nào nên dùng?** Có thể phù hợp với các ứng dụng có yêu cầu bảo mật trung bình nhưng không phải là giải pháp tốt nhất.

## **3. Lưu trong Cookie (HttpOnly, Secure)**

* Access Token được lưu trong cookie với flag HttpOnly, Secure, SameSite để bảo vệ khỏi các cuộc tấn công phổ biến.
* Trình duyệt sẽ tự động gửi token trong mỗi request đến server.

**Ưu điểm:**

* Bảo vệ tốt hơn khỏi XSS vì JavaScript không thể truy cập cookie HttpOnly.
* Khi kết hợp với SameSite=Strict hoặc Lax, có thể giảm thiểu nguy cơ CSRF.
* Cookie có thể được cấu hình để chỉ gửi qua HTTPS, tăng cường bảo mật.

**Nhược điểm:**

* Có thể bị tấn công CSRF nếu không thiết lập SameSite đúng cách.
* Phải quản lý thời gian hết hạn của cookie một cách hợp lý.

**Khi nào nên dùng?**

Rất phù hợp với các ứng dụng yêu cầu bảo mật cao.

*Nguồn: Internet*

## **4. Lưu trong Memory (RAM)**

Lưu trữ token trong biến JavaScript, thường là trong state của ứng dụng (ví dụ: React Context, Vuex, Redux, Pinia).

**Ưu điểm:**

* Token chỉ tồn tại trong phiên làm việc, không thể bị truy cập bởi XSS sau khi trang được reload.
* Giảm nguy cơ rò rỉ dữ liệu so với lưu trong localStorage hoặc sessionStorage.

**Nhược điểm:**

* Token sẽ bị mất nếu người dùng reload trang hoặc đóng trình duyệt.
* Cần kết hợp với Refresh Token để đảm bảo trải nghiệm người dùng.

**Khi nào nên dùng?**

* Khi yêu cầu bảo mật cao và kết hợp với Refresh Token.

**Cách Làm Đúng Và Khuyến Nghị**

* Sử dụng HttpOnly Cookie kết hợp với SameSite & Secure flag: Đây là cách tốt nhất để bảo vệ khỏi XSS và CSRF. Cookies nên được thiết lập với các flag HttpOnly, Secure, và SameSite để tăng cường bảo mật.
* Cân nhắc lưu Access Token trong Memory: Nếu cần bảo mật cao, lưu trữ trong Memory có thể là lựa chọn tốt vì dữ liệu không được lưu trữ lâu dài.
* Sử dụng Refresh Token: Để giảm thiểu rủi ro khi Access Token bị đánh cắp, sử dụng Refresh Token để có thể lấy lại Access Token mới mà không cần người dùng đăng nhập lại.

# **Using Lombok and Mapstruct**

## Lombok

@Getter

@Setter

@Data

@NoArgsConstructor

@AllArgsConstructor

@FieldDefaults

## **Mapstruct**

MapStruct là một code generator giúp tự động tạo mapping code giữa các Java beans, rất phổ biến trong Spring Boot applications.

Ví dụ như: DTO với Entity

@Mapper

Tự map các trường từ DTO vào Entity

@Mapping(source= “”,target=””)

Map từ đâu tới đâu

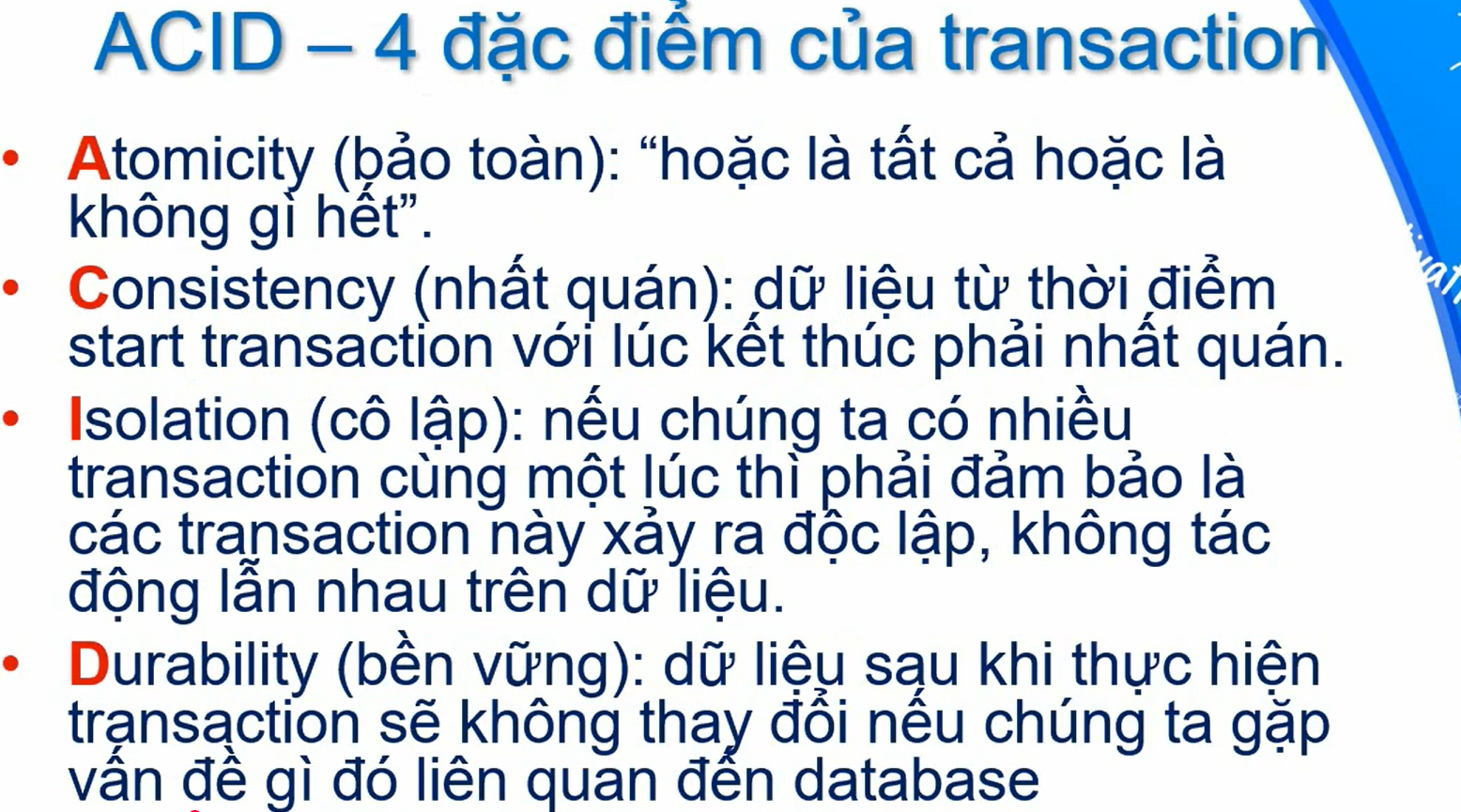
@Mapping(target = "A",ignore = true)

Tự map tay trường A

1. Transaction

Transaction là gì?

Transaction quản lý những thay đổi mà bạn thực hiện trong một hoặc nhiều hệ thống, nó có thể database, message brokers, hoặc bất kỳ loại hệ thống phần mềm nào khác. Mục tiêu chính của giao dịch là cung cấp các đặc điểm ACID để đảm bảo tính nhất quán và hợp lệ của dữ liệu của bạn.



1. Các sự cố xảy ra trong production và cách khắc phục

## Chống Duplicate request ?

Xử lý request:

* Nếu không trùng key -> Xử lý request và lưu request + response + key vào cache (Redis).
* Nếu trùng key và request body giống nhau -> Trả response đã lưu về client.
* Nếu trùng key và request body khác nhau -> Trả lỗi về người dùng.

## Xử lý vấn đề N+1 trong spring jpa ?

### Bài toán thực tế

Để hiểu bài toán N+1 queries là gì thì mình lấy ví dụ như thế này.

Hồi đi học, có môn lập trình web, thầy giáo thường yêu cầu cả lớp mình code chay, không sử dụng framework gì hết. Lần đó nhóm mình được yêu cầu làm một trang tin tức có hai đối tượng chính là Post và Comment. Cả nhóm hí hửng làm, giao diện đẹp, chạy mượt nên chắc mẩm là sẽ A hoặc A+ nhưng không...

Thầy chả thèm ngó ứng dụng mà vô coi source code xem chúng mài code thế nào. Và câu chuyện là...

Tính năng kiểm duyệt comment của bài viết được một bạn trong nhóm code như sau:

1. Lấy toàn bộ comments đang chờ duyệt:

SELECT c.id, c.content, c.post\_id as postId FROM tbl\_comments c WHERE c.status = 'PENDING';

1. Sau khi có danh sách comments thì lấy thông tin về bài viết chứa comment đó:

SELECT p.title, p.slug FROM tbl\_posts p WHERE p.id = :postId;

* Thầy hỏi, bây giờ thầy lấy 10 comments ra để duyệt. Vậy thì ứng dụng của các em sẽ thực thi bao nhiêu câu lệnh SQL?
* Nghe câu hỏi là mình biết lần này xong rồi, nhưng vẫn trả lời: "11 ạ!". Thầy hỏi tiếp:"Sao lại là 11?".
* Mình trả lời:"Gồm có 1 câu lệnh lấy ra danh sách các comments và mỗi comment trong vòng lặp sẽ thêm một câu lệnh select lấy thông tin bài viết tương ứng, 10 comments thì sẽ thực thi 10 + 1 = 11 câu lệnh ạ.".
* Thầy hỏi:"Thế làm vậy có tối ưu không?"
* Bạn trong nhóm (làm tính năng đó) trả lời:"Do ban đầu bọn em chỉ định lấy thông tin comment nên mới viết 1 câu sql. Nhưng sau đó muốn lấy thêm thông tin bài viết nhưng không muốn sửa câu sql nên em mới sử dụng vòng lặp ạ".
* Thôi xong...! Bạn tôi báo quá rồi. Lúc đó mình đã ra hiệu đừng giải thích nữa mà ông bạn vẫn nói.
* Thầy chốt:"Em biết làm thế này là không tối ưu mà vẫn làm! Chỗ này chỉ cần sử dụng câu lệnh join hai bảng vậy mà lại lặp thế này. Nhỡ có hàng trăm comments thì sẽ thực thi bao nhiêu câu lệnh sql? Fix như thế này là em chỉ fix phần ngọn mà không quan tâm đến hiệu năng hệ thống."

Chắc các bạn cũng biết kết cục như nào rồi! Và theo như thầy nói thì chỉ cần sử dụng JOIN là xong:

SELECT

c.id,

c.content,

c.post\_id as postId ,

p.title,

p.slug

FROM tbl\_comments c JOIN tbl\_posts p ON c.post\_id = p.id

WHERE c.status = 'PENDING';

Đây cũng chính là bài toán N+1 queries mà mình muốn đề cập. Đó là trường hợp chúng ta thực hiện thêm N câu queries để lấy cùng một dữ liệu mà lẽ ra có thể lấy chỉ bởi một câu query.

**Framework xử lý bài toán N+1 queries như thế nào?**

* Giả sử chúng ta đang phát triển một trang tin tức và có hai đối tượng chính:
  + Post : mô tả một bài viết.
  + Comment : mô tả một bình luận.
* Quan hệ giữa Post và Comment là 1-n (một bài viết có thể có nhiều bình luận).
* Mình sẽ tạo hai entities cho hai đối tượng Post và Comment đơn giản như sau:

@Setter

@Getter

@Entity(name = "Post")

@Table(name = "tbl\_posts")

public class Post {

@Id

private Long id;

private String title;

private String slug;

@OneToMany(mappedBy = "post")

private List<Comment> comments;

}

@Setter

@Getter

@Entity(name = "Comment")

@Table(name = "tbl\_comments")

public class Comment {

@Id

private Long id;

private String content;

private String status;

@ManyToOne

private Post post;

}

* @ManyToOne và @OneToMany được sử dụng để mô tả mối quan hệ 1-n giữa Post và Comment.
* Trong Hibernate, FetchType là một thuộc tính trong các annotation @OneToOne, @OneToMany, @ManyToOne, @ManyToMany được dùng để định nghĩa cách thức lấy thông tin các đối tượng liên quan. FetchType có hai loại đó là FetchType.EAGER và FetchType.EAGER.
* Tại sao lại nói về hai loại fetch type này? Chúng có liên quan gì đến bài toán N+1 queries trong Hibernate nói riêng?

**2.1 - FetchType.EAGER**

* Trong Hibernate, mặc định @OneToOne và @ManyToOne sẽ sử dụng fetch type là FetchType.EAGER. Tức là nếu các bạn khai báo @ManyToOne thì ngầm định sẽ là @ManyToOne(fetch = FetchType.EAGER).
* FetchType.EAGER có tác dụng gì?
* Đó là thực hiện load toàn bộ dữ liệu liên quan khi thực hiện truy vấn câu truy vấn. Nhưng điều quan trọng là việc load dữ liệu này được thực hiện "âm thầm" bởi Hibernate. Ví dụ, mình thực hiện truy vấn lấy toàn bộ danh sách comment như sau:

@Repository

public class CommentRepository {

@Autowired

EntityManager em;

public List<Comment> getComments() {

return em.createQuery("""

select c from Comment c

""", Comment.class)

.getResultList();

}

}

* Và đây là những câu lệnh Hibernate đã thực thi.

Hibernate: select c1\_0.id,c1\_0.content,c1\_0.post\_id,c1\_0.status from tbl\_comments c1\_0

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

=> Rõ ràng mình chỉ thực thi một câu sql nhưng bên dưới có tới 5 câu sql được thực thi, đó chính là N+1 queries.

Ví dụ khác về quan hệ 1-1 giữa User và Address sử dụng annotation @OneToOne:

@Setter

@Getter

@Entity(name = "User")

@Table(name = "tbl\_users")

public class User {

@Id

private Long id;

private String username;

@OneToOne

private Address address;

}

@Setter

@Getter

@Entity(name = "Address")

@Table(name = "tbl\_address")

public class Address {

@Id

private Long id;

private String detail;

}

@Repository

public class UserRepository {

@Autowired

EntityManager em;

public List<User> getUserInfo() {

return em.createQuery("""

select u from User u

""", User.class)

.getResultList();

}

}

* Những câu lệnh Hibernate đã thực thi.

Hibernate: select u1\_0.id,u1\_0.address\_id,u1\_0.username from tbl\_users u1\_0

Hibernate: select a1\_0.id,a1\_0.detail from tbl\_address a1\_0 where a1\_0.id=?

Hibernate: select a1\_0.id,a1\_0.detail from tbl\_address a1\_0 where a1\_0.id=?

**=> Kết luận 1:** Với hai annotation @OneToOne , @ManyToOne và FetchType.EAGER mặc định sẽ gặp phải bài toán N+1 queries.

**2.2 - FetchType.LAZY**

* Nếu sửa thành @ManyToOne(fetch = FetchType.LAZY) thì sao? FetchType.LAZY có tác dụng gì?
* FetchType.LAZY nghe cái tên chắc các bạn cũng đoán được vai trò của fetch type này đó là đảm bảo khi các dữ liệu liên quan chưa được sử dụng tới thì sẽ chưa query vào database để lấy.
* Cụ thể, nếu mình sửa thành @ManyToOne(fetch = FetchType.LAZY) và chạy lại chương trình thì chỉ một câu lệnh được thực thi.

Hibernate: select c1\_0.id,c1\_0.content,c1\_0.post\_id,c1\_0.status from tbl\_comments c1\_0

* Nhưng nếu mình có sử dụng tới dữ liệu liên quan thì sao? Trong trường hợp này là dữ liệu bảng Post, ví dụ như sau:

@Service

public class CommentService {

private final Logger LOGGER = LoggerFactory.getLogger(CommentService.class);

@Autowired

EntityManager em;

@Autowired

CommentRepository repo;

@Transactional

public void getComments() {

List<Comment> comments = repo.getComments();

for (Comment comment : comments) {

LOGGER.info(

"The Post '{}' got this comment '{}'",

comment.getPost().getTitle(),

comment.getContent()

);

}

}

}

* Ở đây, sau khi lấy được danh sách comments, mình thực hiện logs ra thông tin của bài viết và comment tương ứng. Lúc này Hibernate sẽ thực hiện thêm N câu query như sau:

Hibernate: select c1\_0.id,c1\_0.content,c1\_0.post\_id,c1\_0.status from tbl\_comments c1\_0

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

2024-07-11T23:39:58.269+07:00 INFO 38404 --- [ main] c.d.n.service.CommentService : The post 'This is a post 01' has this comment 'comment on post 1'

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

2024-07-11T23:39:58.270+07:00 INFO 38404 --- [ main] c.d.n.service.CommentService : The post 'This is a post 02' has this comment 'comment on post 2'

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0 where p1\_0.id=?

2024-07-11T23:39:58.271+07:00 INFO 38404 --- [ main] c.d.n.service.CommentService : The post 'This is a post 03' has this comment 'comment on post 3'

* Các bạn để ý logs chỗ này, đầu tiên vẫn là câu query lấy danh sách comment. Nhưng các câu query bên dưới được thực hiện lần lượt khi lặp qua danh sách comments.
* Đối với hai annotation @OneToMany và @ManyToMany thì fetch type mặc định là FetchType.LAZY.

**=> Kết luận 2:** Cho dù có sửa thành FetchType.LAZY hoặc sử dụng @OneToOne hoăck @ManyToMany vẫn không tránh được N+1 queries nếu chúng ta có sử dụng đến các dữ liệu liên quan.

### **Giải pháp**

* Ở phần 1, mình lấy ví dụ về bài toán N+1 sử dụng câu lệnh native sql. Khi đó giải pháp cho bài toán N+1 queries chính là sử dụng mệnh đề JOIN. Vậy nếu sử dụng framework thì sao?

1. **JOIN FETCH**

* Bản chất JOIN FETCH là kỹ thuật giúp chúng ta đạt được mệnh đề JOIN khi thao tác với các entities thay vì phải viết câu lệnh native sql. Với cách viết này thì khi dùng cả FetchType.EAGER và FetchType.LAZY đều sẽ tránh được N+1 queries.

@Repository

public class CommentRepository {

@Autowired

EntityManager em;

public List<Comment> getCommentsWithJoinFetch() {

return em.createQuery("""

select c

from Comment c

join fetch c.post p

""", Comment.class)

.getResultList();

}

}

* Lúc này, dù cấu hình fetch type là gì hay có sử dụng thông tin bảng Post không thì cũng chỉ có 1 câu query được thực hiện:

Hibernate: select c1\_0.id,c1\_0.content,p1\_0.id,p1\_0.slug,p1\_0.title,c1\_0.status from tbl\_comments c1\_0 join tbl\_posts p1\_0 on p1\_0.id=c1\_0.post\_id

Lưu ý: Nếu bảng Comment và Post có quá nhiều dữ liệu, việc join 2 bảng này có thể dẫn đến vấn đề hiệu năng. Khi đó, các bạn có thể sử dụng JOIN FETCH kết hợp với Pagination như sau:

public List<Comment> getCommentsWithJoinFetchAndPagination(int pageSize, int pageNumber) {

return em.createQuery("""

select c

from Comment c

join fetch c.post p

""", Comment.class)

.setFirstResult(pageNumber \* pageSize)

.setMaxResults(pageSize)

.getResultList();

}

**=> Kết luận 3:** Phương pháp đầu tiên để tránh N+1 queries đó là sử dụng JOIN FETCH trong các framework như Hibernate.

#### **b) Secondary Queries**

* Trong trường hợp các bạn muốn lấy dữ liệu Post và Comment độc lập thì có thể sử dụng kỹ thuật secondary queries thay vì sử dụng vòng lặp for. Lúc này chỉ có hai câu queries được thực thi nhưng có thể các bạn sẽ phải xử lý nhiều hơn ở tầng Service.
* Ví dụ mình muốn lấy danh sách bài viết và toàn bộ các comments của từng bài viết:

public Map<String, Object> getPostAndCommentsWithSecondaryQueries() {

Map<String, Object> result = new HashMap<>();

List<Post> posts = em.createQuery("""

select p from Post p

""", Post.class)

.getResultList();

List<Comment> comments = em.createQuery("""

select c from Comment c where c.post in :posts

""", Comment.class)

.setParameter("posts", posts)

.getResultList();

result.put("comments", comments);

result.put("posts", posts);

return result;

}

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0

Hibernate: select c1\_0.id,c1\_0.content,c1\_0.post\_id,c1\_0.status from tbl\_comments c1\_0 where c1\_0.post\_id in (?,?,?)

**=> Kết luận 4**: Sử dụng secondary queries cũng có thể tránh được N+1 queries nhưng các bạn sẽ thực thi hai câu sql độc lập dựa trên mệnh đề IN trong SQL.

#### **c)** [**@BatchSize**](https://viblo.asia/u/BatchSize)

* @BatchSize trên các annotation có fetch type là FetchType.LAZY cũng là một phương pháp để tránh bài toán N+1 queries trong Hibernate. Bản chất việc sử dụng annotation này là thực hiện một subquery sử dụng mệnh đề IN. Các bạn cập nhật entity Post như sau:

@Setter

@Getter

@Entity(name = "Post")

@Table(name = "tbl\_posts")

public class Post {

@Id

private Long id;

private String title;

private String slug;

@OneToMany(mappedBy = "post")

@BatchSize(size = 5)

private List<Comment> comments;

}

Ví dụ mình lấy danh sách 5 bài viết và comments tương ứng của mỗi bài viết thì mình sẽ cấu hình như bên trên và dưới đây là câu lệnh mà Hibernate thực thi.

Hibernate: select p1\_0.id,p1\_0.slug,p1\_0.title from tbl\_posts p1\_0

Hibernate: select c1\_0.post\_id,c1\_0.id,c1\_0.content,c1\_0.status from tbl\_comments c1\_0 where c1\_0.post\_id in (?,?,?,?,?)

2024-07-13T10:21:32.351+07:00 INFO 71168 --- [ main] c.d.n.service.CommentService : The Post 'This is a post 01' has '1' comments

2024-07-13T10:21:32.351+07:00 INFO 71168 --- [ main] c.d.n.service.CommentService : The Post 'This is a post 02' has '1' comments

2024-07-13T10:21:32.351+07:00 INFO 71168 --- [ main] c.d.n.service.CommentService : The Post 'This is a post 03' has '2' comments

Lưu ý: Các bạn có thể cấu hình batch fetch size mặc định trong file application.properties hoặc application.yml như sau:

spring:

jpa:

properties:

hibernate:

default\_batch\_fetch\_size: 5

spring.jpa.properties.hibernate.default\_batch\_fetch\_size = 5

**=> Kết luận 5**: Bản chất @BatchSize cũng sử dụng mệnh đề IN trong SQL nhưng khác với secondary queries ở chỗ chúng ta có thể cấu hình và không cần viết nhiều câu sql.

**3. Điểm D có đáng không?**

Nếu các bạn để ý, cả 3 cách mà Hibernate sử dụng để giải quyết bài toán N+1 vẫn là cố đẩy về việc sử dụng JOIN và IN trong SQL. Vậy JOIN và IN khác nhau như thế nào?

**JOIN**

* JOIN được sử dụng để kết hợp các bản ghi từ hai hoặc nhiều bảng bằng cách sử dụng các giá trị chung của mỗi bảng.
* Cơ chế hoạt động của join là dựa trên điều kiện => tìm ra những bản ghi thoả mãn điều kiện ở các bảng => tạo ra danh sách các bản ghi kết hợp từ dữ liệu các bảng => trả về kết quả.
* Như vậy trước khi điều kiện WHERE được kiểm tra thì dữ liệu được duyệt đã có sẵn rồi.

**IN**

* IN thì dễ hiểu rồi, so với JOIN thì SELECT IN chỉ đơn giản lọc các bản ghi thoả điều kiện trong mệnh đề IN và mệnh đề WHERE (nếu có) => trả về kết quả.

=> Đến đây chắc các bạn cũng hình dung ra sử dụng IN sẽ tối ưu hơn việc join hai hay nhiều bảng lớn vì JOIN phải quét các bảng, kết hợp dữ liệu rồi, kiểm tra điều kiện rồi mới trả kết quả trong khi IN thì chỉ cần quét bảng và trả về kết quả theo điều kiện.

Quay lại câu chuyện ở đầu bài viết, đôi khi "giả ngu" lại hay hơn là cố cãi cho đúng, team mình đã sai khi "cố cãi" cho cái "ngu" của mình. Nhưng thầy nói dùng JOIN là xong?

* Vậy nếu ứng dụng của mình có hàng trăm ngàn bài viết, hàng triệu bình luận mà cứ miễn cưỡng JOIN thì liệu có tối ưu không?
* Nếu mình chỉ cần lấy mỗi lần 10 comments và sử dụng IN để khử N+1queries và lấy ra bài viết của mỗi comment thì có nhanh hơn việc JOIN trên hàng triệu bản ghi không?

1. So sánh các khái niệm trong spring

## Spring vs. Spring Boot vs. Spring MVC

### Tổng quan lý thuyết

**🌱 SPRING FRAMEWORK - NỀN TẢNG**

**Là gì?**

* **Framework tổng hợp** cho phát triển ứng dụng Java
* **Hạt nhân**: Dependency Injection (DI) và Inversion of Control (IoC)

**Các module chính:**

java

*// Core Container*

- spring-core: DI, IoC

- spring-beans: Bean factory

- spring-context: ApplicationContext

*// Data Access*

- spring-jdbc: JDBC abstraction

- spring-orm: Hibernate, JPA support

- spring-tx: Transaction management

*// Web*

- spring-web: Web integration

- spring-webmvc: MVC framework ← CHÍNH LÀ SPRING MVC

*// Others*

- spring-aop: Aspect-oriented programming

- spring-test: Testing support

**Ví dụ cấu hình Spring:**

java

*// Cấu hình Java-based*

@Configuration

public class AppConfig {

@Bean

public UserService userService() {

return new UserServiceImpl();

}

}

*// Sử dụng*

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

UserService userService = context.getBean(UserService.class);

**🌐 SPRING MVC - MODULE WEB**

**Là gì?**

* **Một module** của Spring Framework
* **Mô hình MVC** (Model-View-Controller) cho web applications

**Kiến trúc:**

text

Client Request → DispatcherServlet → Controller → Service → DAO

↑ ↓

└─────── View (JSP/Thymeleaf) ← Model ────┘

**Ví dụ Spring MVC:**

java

*// Controller*

@Controller

@RequestMapping("/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping

public String listUsers(Model model) {

model.addAttribute("users", userService.findAll());

return "users/list"; *// View name*

}

@PostMapping

public String createUser(@ModelAttribute User user) {

userService.save(user);

return "redirect:/users";

}

}

*// Cấu hình web.xml và dispatcher-servlet.xml required!*

**Cấu hình thủ công:**

xml

*<!-- web.xml -->*

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

**🚀 SPRING BOOT - "OPINIONATED" SPRING**

**Là gì?**

* **Framework trên Spring Framework**
* **"Opinionated"**: Đưa ra quyết định cấu hình mặc định
* **Giảm boilerplate code**

**Tính năng chính:**

java

@SpringBootApplication *// = @Configuration + @EnableAutoConfiguration + @ComponentScan*

public class MyApplication {

public static void main(String[] args) {

SpringApplication.run(MyApplication.class, args);

*// Tự động: Embedded Tomcat + Auto-configuration*

}

}

@RestController

public class UserController {

@GetMapping("/users")

public List<User> getUsers() {

return userService.findAll();

}

*// Không cần cấu hình XML!*

}

**Auto-configuration magic:**

* **Tự động phát hiện dependencies** và cấu hình
* **Embedded servers** (Tomcat, Jetty, Undertow)
* **Default application properties**
* **Health checks, metrics** (Actuator)

### **📊 SO SÁNH CHI TIẾT**

| Tiêu chí | Spring Framework | Spring MVC | Spring Boot |
| --- | --- | --- | --- |
| **Vai trò** | Tổng hợp, nền tảng | Web MVC framework | "Hộp công cụ" hoàn chỉnh |
| **Cấu hình** | Thủ công, phức tạp | Thủ công + XML | Auto-configuration |
| **Deployment** | WAR + App Server | WAR + App Server | **Standalone JAR** |
| **Server** | Cần server riêng | Cần server riêng | **Embedded server** |
| **Khởi tạo** | ApplicationContext | DispatcherServlet | @SpringBootApplication |
| **Dependencies** | Chọn từng module | Phụ thuộc Spring | Starter dependencies |

**🔧 VÍ DỤ THỰC TẾ**

**1. Spring Framework Project:**

xml

*<!-- pom.xml -->*

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.0</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.0</version>

</dependency>

*<!-- Phải config thủ công các dependencies khác -->*

</dependencies>

**2. Spring Boot Project:**

xml

*<!-- pom.xml -->*

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

*<!-- TỰ ĐỘNG: spring-mvc, tomcat, jackson, validation, ... -->*

</dependencies>

**🎯 KHI NÀO DÙNG CÁI GÌ?**

**Dùng Spring Framework khi:**

* Ứng dụng enterprise phức tạp
* Cần kiểm soát chi tiết cấu hình
* Legacy systems
* Team có kinh nghiệm Spring

**Dùng Spring MVC khi:**

* Chỉ cần web layer
* Đã có Spring Framework
* Web applications truyền thống

**Dùng Spring Boot khi:**

* Microservices
* Prototyping nhanh
* Standalone applications
* Cloud-native development
* Beginners với Spring

**📈 QUAN HỆ & TIẾN HÓA**

text

SPRING FRAMEWORK (Nền tảng)

├── SPRING MVC (Web module)

├── Spring Data

├── Spring Security

└── ...

↓

SPRING BOOT (Wrapper + Auto-configuration)

├── Spring Boot Starter Web

│ ├── Spring MVC

│ └── Embedded Tomcat

├── Spring Boot Starter Data JPA

└── ...

**TÓM TẮT**

* **Spring Framework**: Ông tổ - cung cấp DI/IoC và các module
* **Spring MVC**: Con - module web dựa trên Spring
* **Spring Boot**: Cháu thông minh - wrapper tự động hóa mọi thứ

**Câu thần chú:**

* **Spring** = Dependency Injection + Modules
* **Spring MVC** = Spring + Web MVC Pattern
* **Spring Boot** = Spring + Auto-configuration + Embedded Server

1. **Save() và saveAndFlush() trong JPA**

Trong Spring Data JPA, phương thức saveAndFlush() được sử dụng để lưu một thực thể (entity) vào cơ sở dữ liệu và ngay lập tức thực hiện thao tác **flush**. Dưới đây là một số điểm quan trọng về saveAndFlush():

**1. Ý nghĩa của saveAndFlush()**

* **saveAndFlush()**:
  + Lưu thực thể vào cơ sở dữ liệu.
  + Gửi ngay lập tức các thay đổi từ bộ nhớ đệm (persistence context) xuống cơ sở dữ liệu bằng cách gọi lệnh **flush**.
  + Đảm bảo rằng các thay đổi được phản ánh ngay lập tức trong cơ sở dữ liệu, ngay cả khi transaction chưa kết thúc.

**2. Sự khác biệt giữa save() và saveAndFlush()**

* **save()**:
  + Lưu thực thể vào bộ nhớ đệm (persistence context).
  + Thay đổi chỉ được gửi xuống cơ sở dữ liệu khi transaction được commit hoặc khi gọi thủ công flush().
* **saveAndFlush()**:
  + Lưu thực thể và ngay lập tức thực hiện thao tác **flush** để đồng bộ hóa dữ liệu với cơ sở dữ liệu.

**3. Khi nào nên sử dụng saveAndFlush()?**

* Khi bạn cần đảm bảo rằng dữ liệu được lưu ngay lập tức vào cơ sở dữ liệu, ví dụ:
  + Trong các trường hợp cần kiểm tra dữ liệu ngay sau khi lưu.
  + Khi bạn cần thực hiện các thao tác khác phụ thuộc vào dữ liệu vừa được lưu.

**4. Ví dụ minh họa**

**Java**

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

// Getters and Setters

}

@Repository

public interface UserRepository extends JpaRepository<User, Long> {

}

// Sử dụng saveAndFlush()

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public void saveUser() {

User user = new User();

user.setName("John Doe");

// Lưu và flush ngay lập tức

userRepository.saveAndFlush(user);

// Dữ liệu đã được lưu vào cơ sở dữ liệu tại thời điểm này

}

}

**Lưu ý**

* **Flush** không đồng nghĩa với **commit**: saveAndFlush() chỉ đẩy các thay đổi xuống cơ sở dữ liệu, nhưng transaction vẫn chưa được commit cho đến khi kết thúc.
* Sử dụng saveAndFlush() không cần thiết trong mọi trường hợp, chỉ nên dùng khi thực sự cần đồng bộ hóa dữ liệu ngay lập tức.

@Transactional

public void flushVsCommit() {

*// 🔹 OPERATION 1: Save user*

User user1 = new User("User1", "user1@email.com");

userRepository.save(user1);

*// ✅ FLUSH: Đồng bộ xuống database*

entityManager.flush();

System.out.println("✅ FLUSH completed - SQL executed");

*// Database: Có bản ghi User1, nhưng trong transaction*

*// 🔹 OPERATION 2: Save another user*

User user2 = new User("User2", "user2@email.com");

userRepository.save(user2);

*// ❌ CHƯA flush - chỉ trong persistence context*

*// ❌ Nếu exception ở đây:*

*// throw new RuntimeException();*

*// → User1 sẽ bị rollback (dù đã flush), User2 chưa flush*

}

*// ✅ COMMIT: Nếu không exception, cả User1 và User2 được commit*

