MỘT SỐ KHÁI NIỆM NÂNG CAO TRONG JAVA

Một số khái niệm nâng cao trong Java:

- Chú thích (Annotation)
- Phản xạ (Reflection)
- Tiêm nhiễm phụ thuộc
  (**Dependency Injection - DI**)
- IoC (Invert of Control)

I. Annotation

1. Annotation trong Java là gì?

- Annotation được hiểu là một dạng chú thích hoặc một dạng siêu dữ liệu (metadata) được dùng để cung cấp thông tin dữ liệu cho mã nguồn Java.
- Các chú thích không có ảnh hưởng trực tiếp đến hoạt động của mã mà chúng chú thích.
- Annotation được thêm vào Java từ Java 5.
- Các Annotation được sử dụng trong mã nguồn sẽ được biên dịch thành bytecode và sử dụng kỹ thuật phản chiếu (Reflection) để truy vấn thông tin siêu dữ liệu và đưa ra hành động thích hợp.
- Annotation có thể sử dụng chú thích các lớp (class), phương thức (method), các biến (variable), các gói (package) và các tham số (prameter) trong Java.
- Java Annotation có hai loại:
  - Các Annotation được tích hợp sẵn.
  - Annotation do người dùng tự định nghĩa.

2. Cấu trúc của một Annotation trong Java

- Một chú thích luôn bắt đầu với ký hiệu @ và sau đó là tên của chú thích. Ký hiệu @ chỉ ra cho trình biên dịch rằng đây là một chú thích.

Cú pháp:

@<chú thích>

Ví dụ:

@Deprecated

- Ký hiệu @ mô tả đây là một chú thích.
- Deprecated là tên của chú thích.

3. Chức năng của Annotation trong Java

Annotation được sử dụng trong Java cho 3 mục đích chính:

- ***Thứ nhất:*** chỉ dẫn cho trình biên dịch (Compiler)

Chú thích có thể được trình biên dịch sử dụng để phát hiện lỗi hoặc triệt tiêu các cảnh báo. Java có 3 Annotation có thể được sử dụng nhằm cung cấp chỉ dẫn cho trình biên dịch:

1. @Deprecated
2. @Override
3. @SuppressWarnings

- ***Thứ hai:*** chỉ dẫn trong thời điểm biên dịch (Build-time)

Các công cụ phần mềm có thể thông qua các chỉ dẫn của chú thích để tạo mã nguồn, tệp XML, nén mã biên dịch và các tâp tin vào một tập tin, v…v

- ***Thứ ba***: chỉ dẫn trong thời gian chạy (Runtime)

Thông thường, các Annotation không có mặt trong mã Java sau khi biên dịch. Tuy nhiên, có thể xác định trong thời gian chạy bằng cách sử dụng kỹ thuật Reflection và có thể sử dụng để đưa ra những hướng dẫn cho chương trình trong thời gian chạy

4. Các Annotation tích hợp sẵn trong Java

Annotation tích hợp sẵn có hai loại:

- Annotation được tích hợp để sử dụng trực tiếp trong code Java.
- Annotation được tích hợp sẵn được sử dụng trong Annotation khác

a) Annotation được tích hợp để sử dụng trực tiếp trong code Java.

@Override:

Được sử dụng cho các phương thức có nghĩa là ghi đè một phương thức trong lớp cha (supperclass). Nếu một phương thức đánh dấu @Override không ghi đè chính xác một phương thức trong lớp cha của nó hay hiểu đơn giản là phương thức đó không hợp lệ thì trình biên dịch sẽ báo lỗi. Chúng ta không nhất thiết phải sử dụng @Override khi ghi đè phương thức, nhưng Annotation này sẽ giúp chúng ta tránh lỗi dễ dàng hơn.

```
public class superExampleNMD {
    public void methodOverride() {
        System.out.println("SupperClass Nguyen Minh Duc");
    }
}
```

```
public class ExampleNMD extends superExampleNMD{
    @Override
    public void methodOverride() {
        System.out.println("SubClass Nguyen Minh Duc");
    }
}
```

- **@Deprecated**

Được sử dụng để dánh dấu một đối tượng (class, method hoặc field) và chỉ dẫn rằng nó tốt nhất không nên được sử dụng nữa. Trình biên dịch sẽ đưa các câu cảnh báo khi chương trình sử dụng các thuộc tính, lớp hoặc phương thức có gắn với @Deprecated.

- **@SuppressWarnings**

Thông báo cho trình biên dịch biết là không được in các câu cảnh báo nào đó.
Cú pháp sử dụng: @SuppressWarnings("...") hoặc @SuppressWarnings({"...","...",v...v...}) trong "..." được hiểu là tên các loại cảnh báo.

Chúng ta thường dùng:

@SuppressWarnings(“deprecation”) để thông báo trình biên dịch không cảnh báo việc sử dụng phương thức có sử dụng @Deprecated.

@SuppressWarnings(“unchecked”) để thông báo trình biên dịch không cảnh báo việc sử một ép kiểu không an toàn.

@SuppressWarnings(“rawtypes”) để thông báo trình biên dịch không cảnh báo lỗi trong khai báo kiểu dữ liệu.

```java
package Warning;
import java.util.Date;
public class Warning {
    public Date getDate() {
        Date date = new Date(2020, 05, 20);
    }
}
```

Surround with ...

May split declaration into a declaration and assignment

Explict type can be replaced with 'var'

(Alt-Enter shows hints)

Cảnh báo sử dụng hàm lỗi thời (@Deprecated)

```java
package Warning;
import java.util.Date;
public class Warning {
    @SuppressWarnings("deprecation")
    public Date getDate() {
        Date date = new Date(2020, 05, 20);
        return date;
    }
}
```

Sử dụng @SuppressWarnings tắt cảnh báo

b) Annotation được tích hợp sẵn được sử dụng trong Annotation khác

- **@Target:** Dùng để chú thích **phạm vi sử dụng** của một Annotation. Các chú thích này đã được định nghĩa trong enum **java.lang.annotation.ElementType**:

ElementType.TYPE	Chú thích trên Class, interface, enum, annotation
ElementType.FIELD	Chú thích trường (field), bao gồm cả các hằng số enum.
ElementType.METHOD	Chú thích trên method.
ElementType.PARAMETER	Chú thích trên parameter.
ElementType.CONSTRUCTOR	Chú thích trên constructor.
ElementType.LOCAL_VARIABLE	Chú thích trên biến địa phương.
ElementType.ANNOTATION_TYPE	Chú thích trên Annotation khác.
ElementType.PACKAGE	Chú thích trên package.

•**@Retention:** Dùng để chú thích **mức độ tồn tại** của một Annotation nào đó. Cụ thể có 3 mức nhận thức tồn tại của vật được chú thích, và được định nghĩa trong enum **java.lang.annotation.RetentionPolicy**:

RetentionPolicy.SOURCE	Tồn tại trên mã nguồn, và không được trình biên dịch nhận ra.
RetentionPolicy.CLASS	Mức tồn tại được trình biên dịch nhận ra, nhưng không được nhận biết bởi máy ảo tại thời điểm chạy (Runtime).
RetentionPolicy.RUNTIME	Mức tồn tại lớn nhất, được trình biên dịch nhận biết, và máy ảo (JVM) cũng nhận ra khi chạy chương trình.

- **@Inherited:** Chú thích này chỉ ra rằng chú thích mới nên được bao gồm trong tài liệu Java được tạo ra bởi các công cụ tạo tài liệu Java.
- **@Documented: C**hú thích chỉ ra rằng loại chú thích có thể được kế thừa từ lớp cha và có giá trị mặc định là **false**. Khi người dùng truy vấn kiểu Annotation của lớp con và lớp con không có chú thích cho kiểu này thì lớp cha của lớp được truy vấn cho loại chú thích sẽ được gọi. Chú thích này chỉ áp dụng cho các khai báo class.

5. Annotation tự định nghĩa

Annotation khá giống một *interface*, để khai báo một Annotation chúng ta sử dụng **@interface**. Annotation có thể có hoặc không có các phần tử (element) trong đó.

Một phần tử của Annotation có các đặc điểm như sau:

1. Không có thân hàm
2. Không có tham số hàm
3. Khai báo trả về phải là một kiểu dữ liệu cụ thể (Kiểu nguyên thủy, Enum, Annotation hoặc Class).
4. Có thể có giá trị mặc định.

Một Annotation sẽ đượng định nghĩa bởi các **Meta-Annotations**. Các Meta-Annotations gồm @Retention, @Target, @Documented, @Inherited.

Ví dụ:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(value = RetentionPolicy.SOURCE)
//Nó tồn tại trên mã nguồi và trình biên dịch không nhận ra
@Target(value = {ElementType.METHOD, ElementType.FIELD})
//Nó sẽ được dùng chú thích tren mot method
public @interface MyCustomAnnotation{
}
```

```
public class UsingAnno {
    @MyCustomAnnotation  //Gán trước một field
    private int myAge = 22;

    @MyCustomAnnotation   //Gán trước một method
    public void aMethod(){

    }
}
```

6. Annotation lồng nhau (lặp lại chú thích) và cách sử dụng

Trong một số trường hợp, nhu cầu sử dụng một Annotation nhiều lần. Khi đó chúng ta cần tạo một Wrapper chứa danh sách các Annotation có thể lặp lại.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
public class TestAnnotationMain {
   // 1. Koi tao mot chu thich can lap lai
   @Target(ElementType.TYPE)
   @Retention(RetentionPolicy.RUNTIME)
   public @interface ReAnno {
      String value();
   };
   // 2. Khoi tao mot chu thich long nhau
   @Target(ElementType.TYPE)
   @Retention(RetentionPolicy.RUNTIME)
   public @interface ReAnnos {
      ReAnno[] value();
   }
```

```
// 3. Use repeating annotations
    @ReAnnos({
        @ReAnno("Codelearn"),
        @ReAnno("NMD.SE")})
    public interface ReAnnotation {
    }
    public static void main(String[] args) {
        // 4. Retrieving Annotations via the Filters class
        ReAnnos rep =
ReAnnotation.class.getAnnotation(ReAnnos.class);
        for (ReAnno re : rep.value()) {
            System.out.println(re.value());
        }
    }
}
```

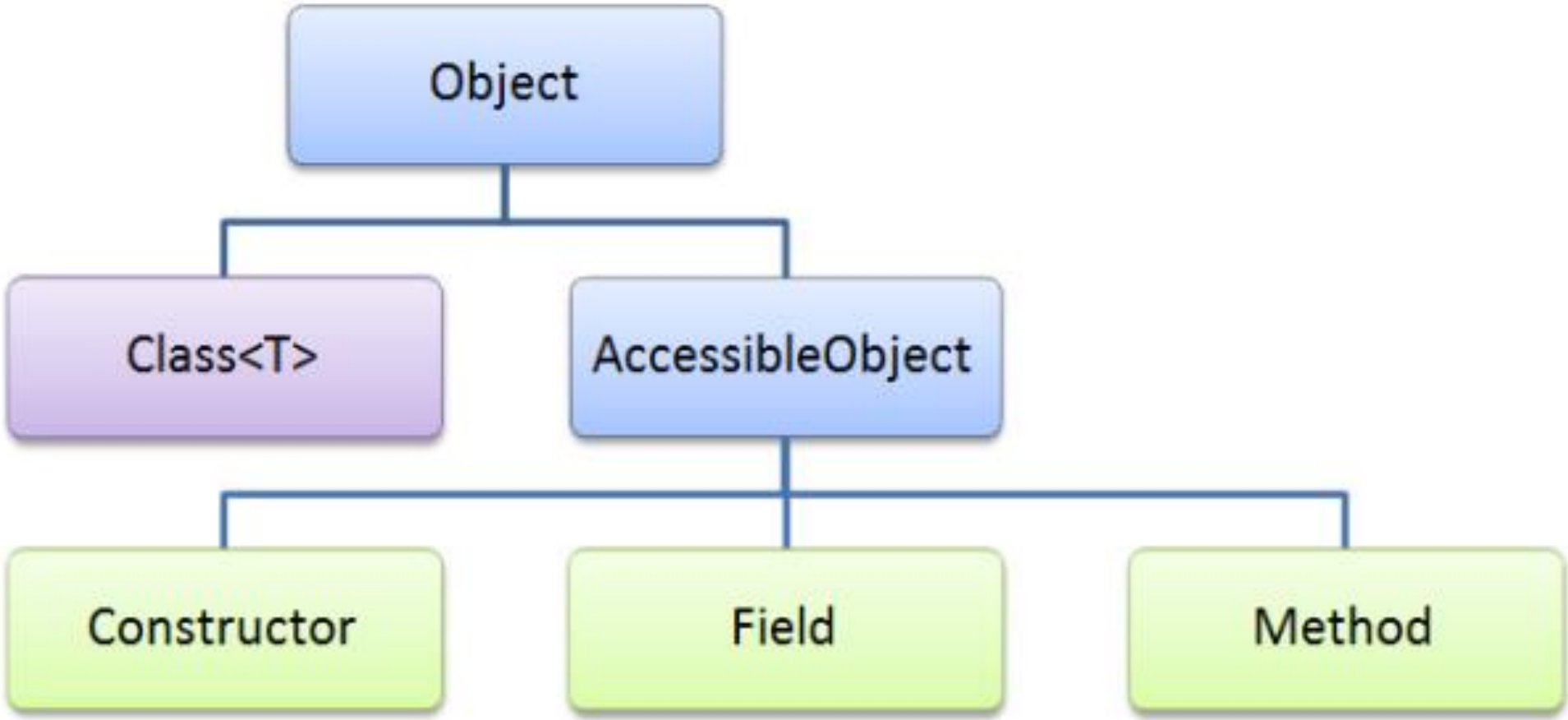
II. Reflection

1. Java Reflection là gì?

- Java là một ngôn ngữ hướng đối tượng (Object-oriented), thông thường phải tạo ra một đối tượng sau đó mới truy cập vào các trường (field), hoặc gọi phương thức (method) của đối tượng này thông qua toán tử dấu chấm (.).
- Java Reflection giới thiệu một cách tiếp cận khác, nó cho phép truy cập vào một trường của một đối tượng nếu biết tên của trường đó. Hoặc nó có thể gọi một phương thức của đối tượng nếu biết tên phương thức, các kiểu tham số của phương thức, và các giá trị tham số để truyền vào …
- Java Reflecion cho phép truy cập, sửa đổi cấu trúc và hành vi của một đối tượng tại thời gian chạy (runtime) của chương trình. Đồng thời nó cho phép truy cập vào các thành viên private (private member) tại mọi nơi trong ứng dụng, điều này không được phép với cách tiếp cận truyền thống.
- Java Reflection khá mạnh mẽ và rất hữu ích đối với những ai hiểu rõ về nó. Ví dụ, người dùng có thể ánh xạ (mapping) đối tượng (object) thành table dưới database tại thời điểm runtime. Kỹ thuật này có thể thấy rõ nhất ở JPA và Hibernate.

2. Kiến trúc của Java Reflection API

- Các lớp được dùng trong reflection nằm trong hai package là java.lang và java.lang.reflect. Gói java.lang.reflect bao gồm ba lớp chính là Constructor, Field và Method:
- Class<T>: lớp này đại diện cho các lớp, interface và chứa các phương thức dùng để lấy các đối tượng kiểu Constructor, Field, Method,…
- AccessibleObject: kiểm tra về phạm vi truy xuất (public, private, protected) của field, method, constructor sẽ được bỏ qua. Nhờ đó nó có thể dùng reflection để thay đổi, thực thi các thành phần này mà không cần quan tâm đến phạm vi truy xuất của nó.
- Constructor: chứa các thông tin về một constructor của lớp.
- Field: chứa các thông tin về một field của lớp, interface.
- Method: chứa các thông tin về một phương thức của lớp, interface.


Object
Class<T>
AccessibleObject
Constructor
Field
Method

Lớp Classes:

Khi sử dụng Java Reflection để duyệt qua một class thì việc đầu tiên thường phải làm là có một đối tượng kiểu Class, từ các đối tượng kiểu Class chúng ta có thể lấy được các thông tin về:

- Class Name
- Class Modifies (public, private, synchronized etc.)
- Package Info
- Superclass
- Implemented Interfaces
- Constructors
- Methods
- Fields
- Annotations

Tạo đối tượng Class<>

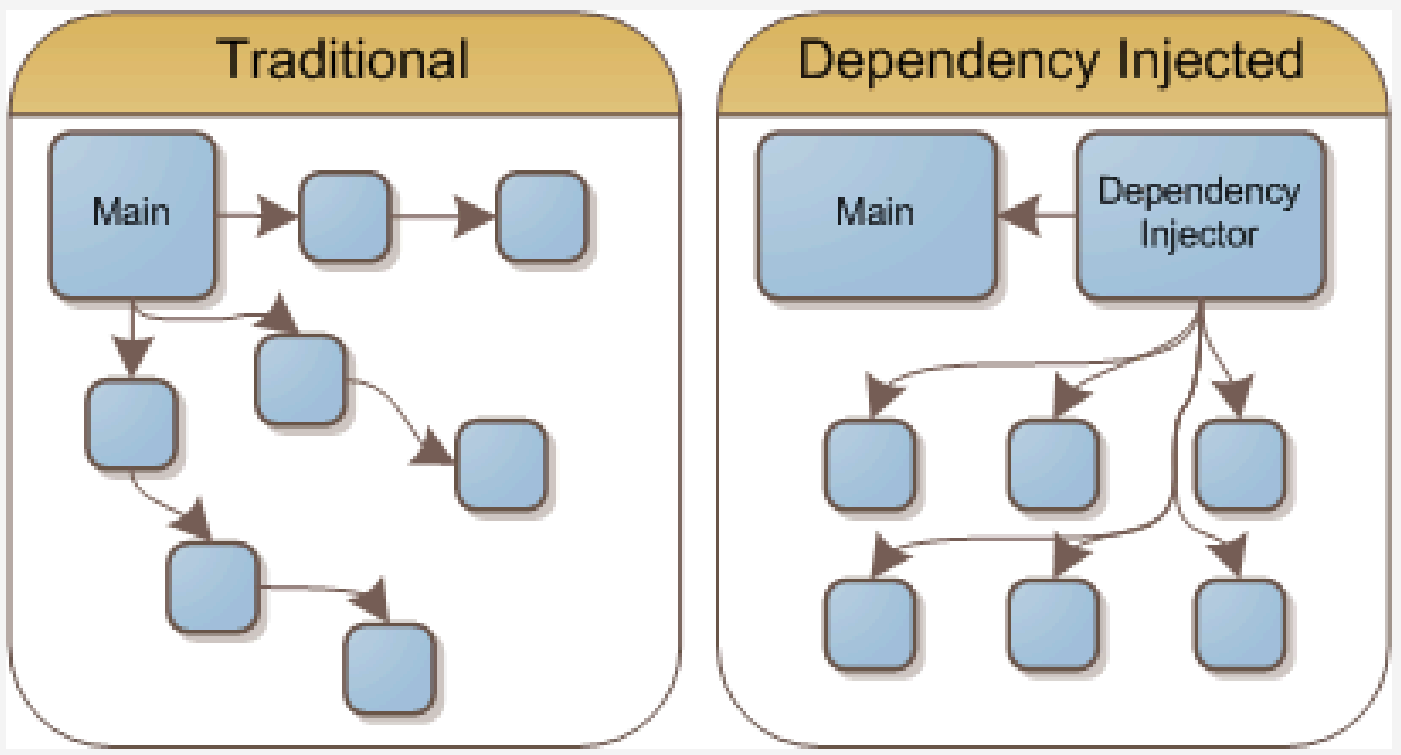
```
try {

    Class c =  Class.forName("com. abc.Cat");
    // ...
} catch (ClassNotFoundException e) {
    System.err.println(e);
}
```

Tham khảo chi tiết: https://gpcoder.com/2883-huong-dan-su-dung-java-reflection/

III. Dependency Injection - DI

Dependency Injection

- Với lập trình hướng đối tượng, chúng ta thường xuyên làm việc với rất nhiều class trong một chương trình, các class được liên kết với nhau theo một mối quan hệ nào đó. Dependency là một loại quan hệ giữa 2 class mà trong đó một class hoạt động độc lập và class còn lại phụ thuộc bởi class kia. Sự phụ thuộc chặt chẽ này gây rất nhiều khó khăn khi hệ thống cần thay đổi, nâng cấp. Để giải quyết vấn đề này chúng ta có thể sử dụng Dependency Injection (DI), một dạng design pattern được thiết kế nhằm ngăn chặn sự phụ thuộc nêu trên.



Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI) là gì

- Nguyên lý SOLID trong OOP:

Dependency Inversion Principle (nguyên lý đảo ngược sự phụ thuộc). Nội dung của nguyên lý này như sau :

- Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
- Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại, chi tiết nên phụ thuộc vào abstraction. Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.

SOLID:

Single Responsibility

Open/Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

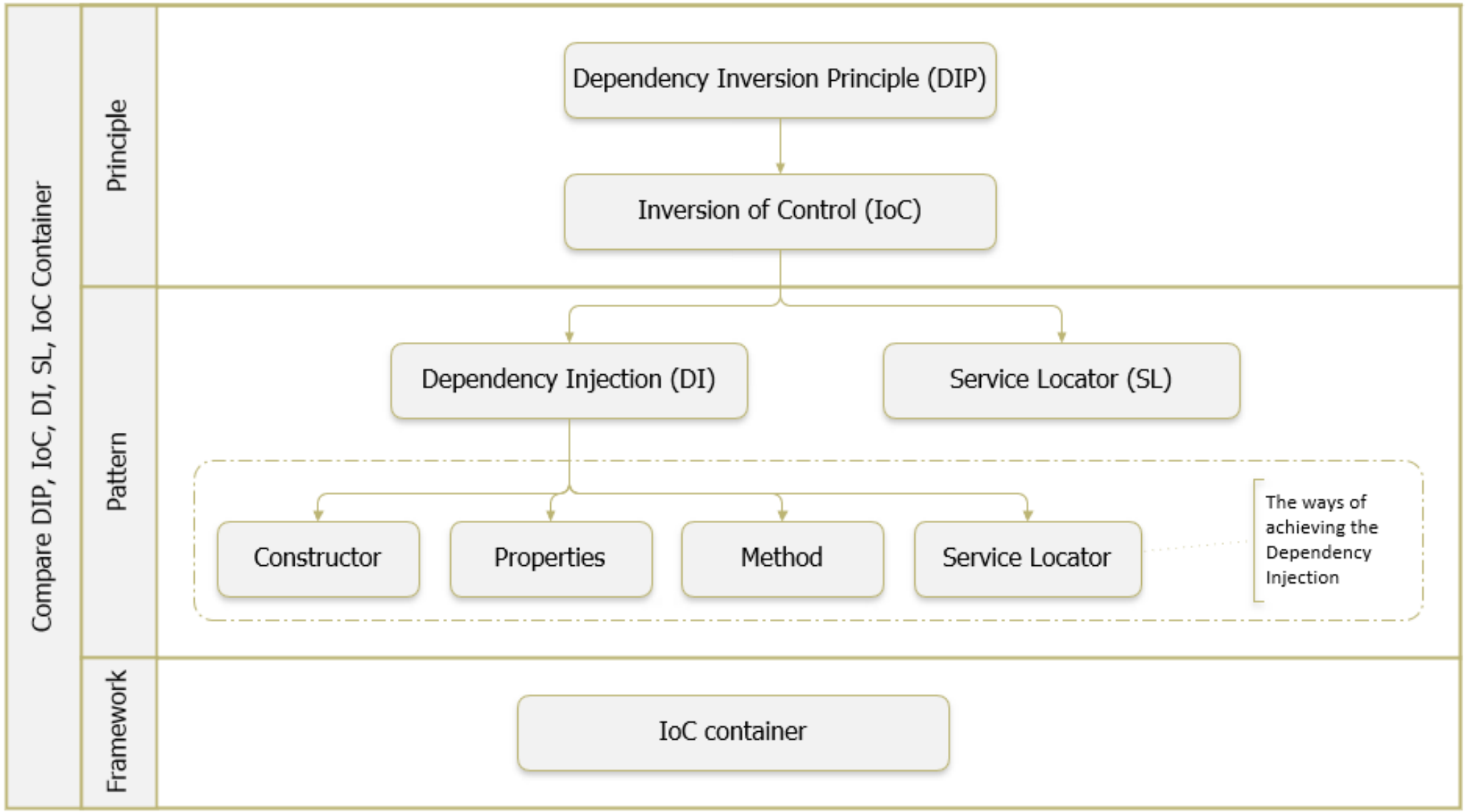
Tham khảo: https://viblo.asia/p/nguyen-ly-solid-trong-lap-trinh-huong-doi-tuong-Qpmlegr7Krd

Chúng ta thường hay lẫn lộn giữa các khái niệm Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI). Ba khái niệm này tương tự nhau, tất cả đều hướng đến một mục đích duy nhất là tạo ra ứng dụng ít kết dính (loosely coupling), dễ mở rộng (flexibility) cũng như giúp lập trình viên tập trung chủ yếu vào dòng công việc (business flow)

Sự khác biệt giữa 3 khái niệm trên như sau:

- Dependency Inversion Principle (DIP): *là một nguyên lý để thiết kế và viết code.*
- Inversion of Control (IoC): *là một design pattern được tạo ra để code có thể tuân thủ nguyên lý Dependency Inversion. Có nhiều cách hiện thực Pattern này như ServiceLocator, Event, Delegate, … và Dependency Injection là một trong các cách đó.*
- Dependency Injection (DI): *là một Design Pattern, một cách để hiện thực Inversion of Control Pattern. DI chính là khả năng liên kết giữa các thành phần lại với nhau, các module phụ thuộc (dependency) sẽ được inject vào module cấp cao.*

DI is about wiring, IoC is about direction, and DIP is about shape.
IoC là hướng đi, DIP là định hình cụ thể của hướng đi, còn DI là một hiện thực cụ thể.



Định nghĩa và khái niệm Dependency Injection

- Dependency Injection (DI) là một design pattern, một kỹ thuật cho phép xóa bỏ sự phụ thuộc giữa các module, làm cho ứng dụng dễ dàng hơn trong việc thay đổi module, bảo trì code và testing.
- DI cung cấp cho một đối tượng các thể hiện phụ thuộc (dependencies) của nó từ bên ngoài truyền vào mà không phải khởi tạo trực tiếp từ trong class sử dụng.
- Nhiệm vụ của dependency injection:
  - Tạo các đối tượng.
  - Quản lý sự phụ thuộc (dependencies) giữa các đối tượng.
  - Cung cấp (inject) các phụ thuộc được yêu cầu cho đối tượng (được truyền từ bên ngoài đối tượng).

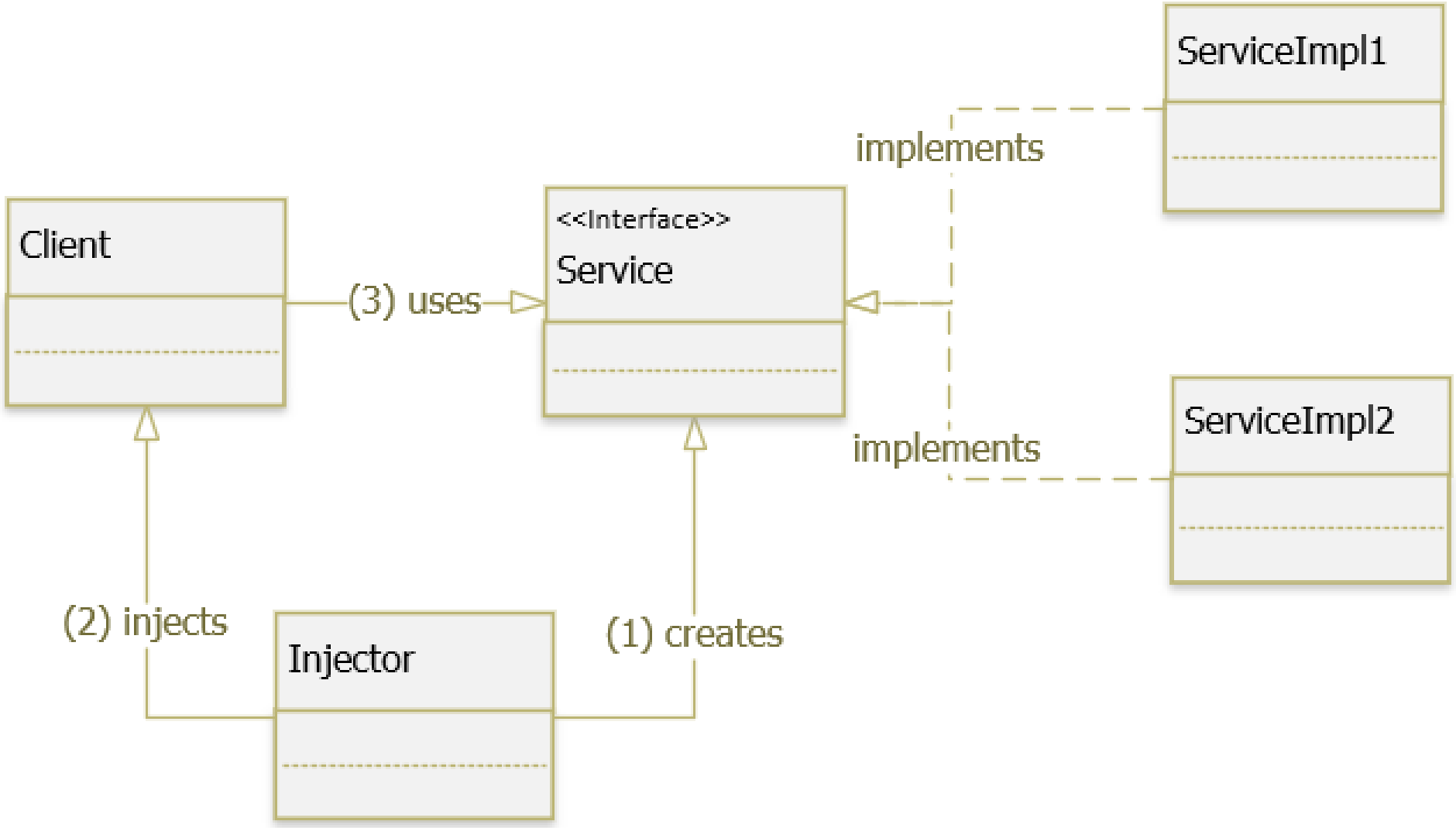
- Nguyên tắc hoạt động của DI:

➢ Các module không giao tiếp trực tiếp với nhau, mà thông qua interface. Module cấp thấp sẽ implement interface, module cấp cao sẽ gọi module cấp thấp thông qua interface.

➢ Việc khởi tạo các module cấp thấp sẽ do DI Container/ IoC Container thực hiện.

➢ Việc Module nào gắn với interface nào sẽ được config trong file properties, trong file XML hoặc thông qua Annotation. Annotation là một cách thường được sử dụng trong các Framework, chẳng hạn như @Inject với CDI, @Autowired với Spring hay @ManagedProperty với JSF.

Cài đặt Dependency Injection như thế nào?



Các thành phần tham gia Dependency Injection Pattern:

- *Client : là một class cần sử dụng Service.*
- *Service : là một class/ interface cung cấp service/ dependency cho Client.*
- *ServiceImpl: cài đặt các phương thực cụ thể của Service.*
- *Injector: là một lớp chịu trách nhiệm khởi tạo các service và inject các thể hiện này cho Client.*

Các dạng Dependency Injection:

- Constructor Injection: Các dependency sẽ được container truyền vào (inject vào) 1 class thông qua constructor của class đó. Đây là cách thông dụng nhất.
- Setter Injection: Các dependency sẽ được truyền vào 1 class thông qua các hàm Setter.
- Fields/ properties: Các dependency sẽ được truyền vào 1 class một cách trực tiếp vào các field.
- Interface Injection: Class cần inject sẽ implement 1 interface. Interface này chứa 1 hàm tên Inject. Container sẽ injection dependency vào 1 class thông qua việc gọi hàm Inject của interface đó. Đây là cách rườm rà và cũng ít được sử dụng.
- Service Locator: nó hoạt động như một mapper, cho phép thay đổi code tại thời điểm run-time mà không cần biên dịch lại ứng dụng hoặc phải khởi động lại.

Ví dụ sử dụng Dependency Injection

Ví dụ chúng ta có một class cho phép User có thể gửi message.

EmailService.java

```
public class EmailService {
    public void sendEmail(String message) {
        System.out.println("Message: " + message);
    }
}
```

UserController.java

```
public class UserController {
    private EmailService emailService = new EmailService();
    public void send() {
      emailService.sendEmail("Hello Dependency injection
pattern");
   }
}
```

Chương trình trên rất đơn giản, chỉ gồm có 2 class. Tuy nhiên, nó có một vài giới hạn như sau:

- Lớp UserController phụ thuộc trực tiếp vào class EmailService. Mỗi khi có thay đổi trong lớp EmailService, chẳng hạn thêm tham số cho constructor của class này lên sẽ ảnh hưởng trực tiếp đến class UserController.
- Một User khác không muốn sử dụng cách gửi message thông qua email, chẳng hạn qua sms, facebook, …
- Khó khăn khi viết Unit Test cho UserController do phụ thuộc trực tiếp vào EmailService.
- Bây giờ chúng ta sẽ áp dụng Dependency Injection để giải quyết các giới hạn trên.

MessageService.java

```
public interface MessageService {
    void sendMessage(String message);
}
```

EmailService.java

```
public class EmailService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Email message: " + message);
    }
}
```

MessageService.java

```
public class SmsService implements MessageService {
    @Override
    public void sendMessage(String message) {
        System.out.println("Sms message: " + message);
    }
}
```

UserController.java

```
public class UserController {
    private MessageService messageService;
    public UserController(MessageService messageService) {
        this.messageService = messageService;
    }
    public void send() {
        messageService.sendMessage("Hello Dependency injection pattern");
    }
}
```

DependencyInjectionPatternExample.java

```
public class DependencyInjectionPatternExample {
    public static void main(String[] args) {
        MessageService messageService = new EmailService();
        UserController userController = new
UserController(messageService);
        userController.send();
    }
}
```

- Thay vì sử dụng trực tiếp lớp EmailService chúng ta sử dụng interface. Lớp UserController không trực tiếp khởi tạo message service mà nó được truyền từ bên ngoài vào. Chính vì vậy mà các lớp không còn phụ thuộc vào nhau, chúng ta có thể dễ dàng sử đổi hay thêm bất kỳ service nào khác mà không ảnh hưởng đến UserController. Đây chính là dạng Constructor Injection của Dependency Injection Pattern.
- Tuy nhiên, chương trình trên vẫn còn một hạn chế là chúng ta phải khởi tạo một thể hiện cụ thể cho MessageService ở rất nhiều nơi, chương trình cũng rất khó khăn khi cần thay thế một service cho toàn bộ hệ thống. Để giải quyết vấn đề này, chúng ta có thể áp dụng Inversion of Control Container (IoC container), một nơi để quản lý các thành phần phụ thuộc này và cung cấp thể hiện cụ thể khi cần sử dụng.

Ưu điểm và khuyết điểm của Dependency Injection

- **Ưu điểm**:

➢Reduced dependencies: giảm sự kết dính giữa các module.

➢Reusable: code dễ bảo trì, dễ tái sử dụng, thay thế module. Giảm boiler-plate code do việc tạo các biến phụ thuộc đã được injector thực hiện.

➢Testable: rất dễ test và viết Unit Test.

➢Readable: dễ dàng thấy quan hệ giữa các module vì các dependecy đều được inject vào constructor.

- **Khuyết điểm:**

➢Khái niệm DI khá khó hiểu đối với người mới tìm hiểu.

➢Sử dụng interface nên đôi khi sẽ khó debug, do không biết chính xác module nào được gọi.

➢Các object được khởi tạo toàn bộ ngay từ đầu, có thể làm giảm performance.

➢Có thể gặp lỗi ở run-time thay vì compile-time.

Sử dụng Dependency Injection khi nào?

Một số trường hợp sử dụng DI:

- Khi cần inject các giá trị từ một cấu hình cho một hoặc nhiều module khác nhau.
- Khi cần inject một dependency cho nhiều module khác nhau.
- Khi cần một vài service được cung cấp bởi container.
- Khi cần tách biệt các dependency giữa các môi trường phát triển khác nhau. Chẳng hạn, với môi trường dev chỉ cần log việc gửi mail, trong môi trường product cần gởi mail thông qua một API thật sự.
- Một số thư viện thư viện và Framework triển khai DI: Spring, Inject, JSF, Google Guice, Dagger.

IV. IoC

1. Giới thiệu IoC – Inversion of Control

- Inversion of Control (IoC) là pattern tuân theo Dependency inversion principle.
- Inversion of Control (IoC) dịch là đảo ngược điều khiển. Ý của nó là làm thay đổi luồng điều khiển của ứng dụng, giúp tăng tính mở rộng của một hệ thống.
- IoC được chia thành 2 loại:

➢ **Dependency Lookup**

- ❖ Dependency Pull
- ❖ Contextualized Dependency Lookup (CDL)

➢ **Dependency Injection**

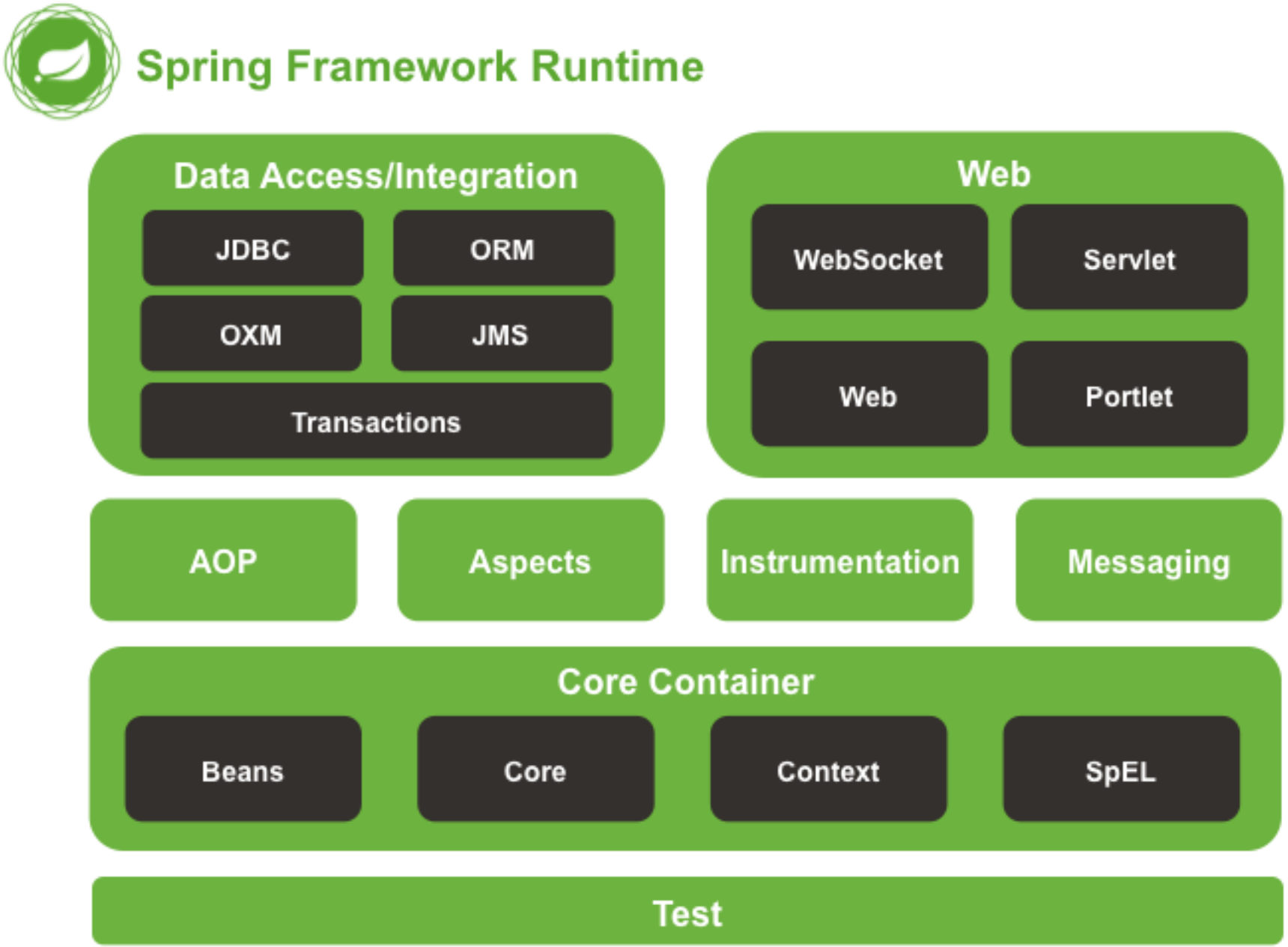
Inversion of Control Container (IoC container) là gì?

- Khi áp dụng kỹ thuật Dependency Injection, thì một vấn đề khác nảy sinh là làm thế nào chúng ta biết được một lớp sẽ phụ thuộc vào những lớp nào để khởi tạo nó. Để giải quyết điều này, người ta nghĩ ra Dependency Injection Container hay còn gọi là Inversion of Control Container (IoC container).
- IoC Container được xây dựng dựa trên ý tưởng của IoC, nó có nhiệm vụ quản lý các thành phần khác nhau, cung cấp tài nguyên cho các thành phần khi chúng đòi hỏi tài nguyên dựa vào thông tin từ các file cấu hình. Nhờ đó việc quản ý sẽ dễ dàng hơn, tập trung hơn và đơn giản, hiệu quả hơn rất nhiều so với việc phải phân tán tài nguyên cho từng thành phần tự xử lý.

- Về bản chất thì IoC Conainter như một tấm bản đồ, nó cho ta biết một lớp phụ thuộc vào những lớp nào khác bằng kỹ thuật Reflection, hoặc từ danh sách đã được đăng ký trước.
- Sức mạnh của IoC Container được thể hiện thông qua việc nó được áp dụng trong các Framework. Một đặc điểm quan trọng của Framework là các phương thức được định nghĩa bởi người dùng thông thường được gọi từ trong bản thân Framework chứ không phải từ code ứng dụng của người dùng. Framework đóng vai trò của chương trình chính trong việc điều phối và sắp xếp hoạt động ứng dụng. Sự đảo ngược điều khiển này tạo ra cho Framework sức mạnh thông qua việc mở rộng. Chính IoC mang đến sự khác biệt giữa một Framework và một thư viện.

- Thư viện là tập hợp các tính năng mà chúng ta có thể sử dụng, nó được tổ chức thành các class. Sau mỗi lần gọi một phương thức, thư viện sẽ làm một số việc và sau đó trả quyền điều khiển về cho người dùng.
- Framework là một biểu hiện của thiết kế trừu tượng với nhiều hành vi được xây dựng sẵn bên trong, để sử dụng nó chúng ta cần chèn các hành vi của mình vào các nơi khác nhau trong Framework bằng các class hoặc plugin. Code của Framework sẽ gọi đến code của chúng ta tại những điểm cần thiết.

SPRING FRAMEWORK


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Instrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

SPRING CORE

- IoC Container là gì?
- Bean pring là gì

- IoC Container trong Spring chính là lõi của Spring Framework. IoC Container sẽ tạo ra các đối tượng, nối chúng lại với nhau, cấu hình chúng, và quản lý vòng đời của chúng từ khi tạo ra đến khi bị hủy. IoC Container sử dụng DI (Dependency Injection) để quản lý các thành phần tạo nên một ứng dụng. Những đối tượng này được gọi là Spring Bean. IoC Container được cung cấp thông tin từ các tập tin XML.
- Có hai loại IoC Container, đó là:
  - BeanFactory
  - ApplicationContext

BeanFactory

- BeanFactory là container đơn giản nhất cung cấp hỗ trợ cơ bản cho DI và được xác định bởi giao diện org.springframework.beans.factory.BeanFactory. BeanFactory và các giao diện có liên quan, như BeanFactoryAware, InitializingBean, DisposableBean vẫn còn tồn tại trong Spring vì mục đích tích hợp các framework bên thứ ba với Spring.
- XmlBeanFactory là lớp thực hiện cho giao diện BeanFactory. Để sử dụng BeanFactory, chúng ta cần phải tạo ra thể hiện của lớp XmlBeanFactory như dưới đây:là container đơn giản nhất cung cấp hỗ trợ cơ bản cho DI và được xác định bởi giao diện org.springframework.beans.factory.BeanFactory.
- BeanFactory và các giao diện có liên quan, như BeanFactoryAware, InitializingBean, DisposableBean vẫn còn tồn tại trong Spring vì mục đích tích hợp các framework bên thứ ba với Spring.
- XmlBeanFactory là lớp thực hiện cho giao diện BeanFactory. Để sử dụng BeanFactory, chúng ta cần phải tạo ra thể hiện của lớp XmlBeanFactory như dưới đây:

```
Resource resource = new ClassPathResource("applicationContext.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

ApplicationContext

- ClassPathXmlApplicationContext là lớp thực hiện của giao diện ApplicationContext. Chúng ta cần phải tạo ra thể hiện của lớp ClassPathXmlApplicationContext để sử dụng ApplicationContext như dưới đây:

ApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

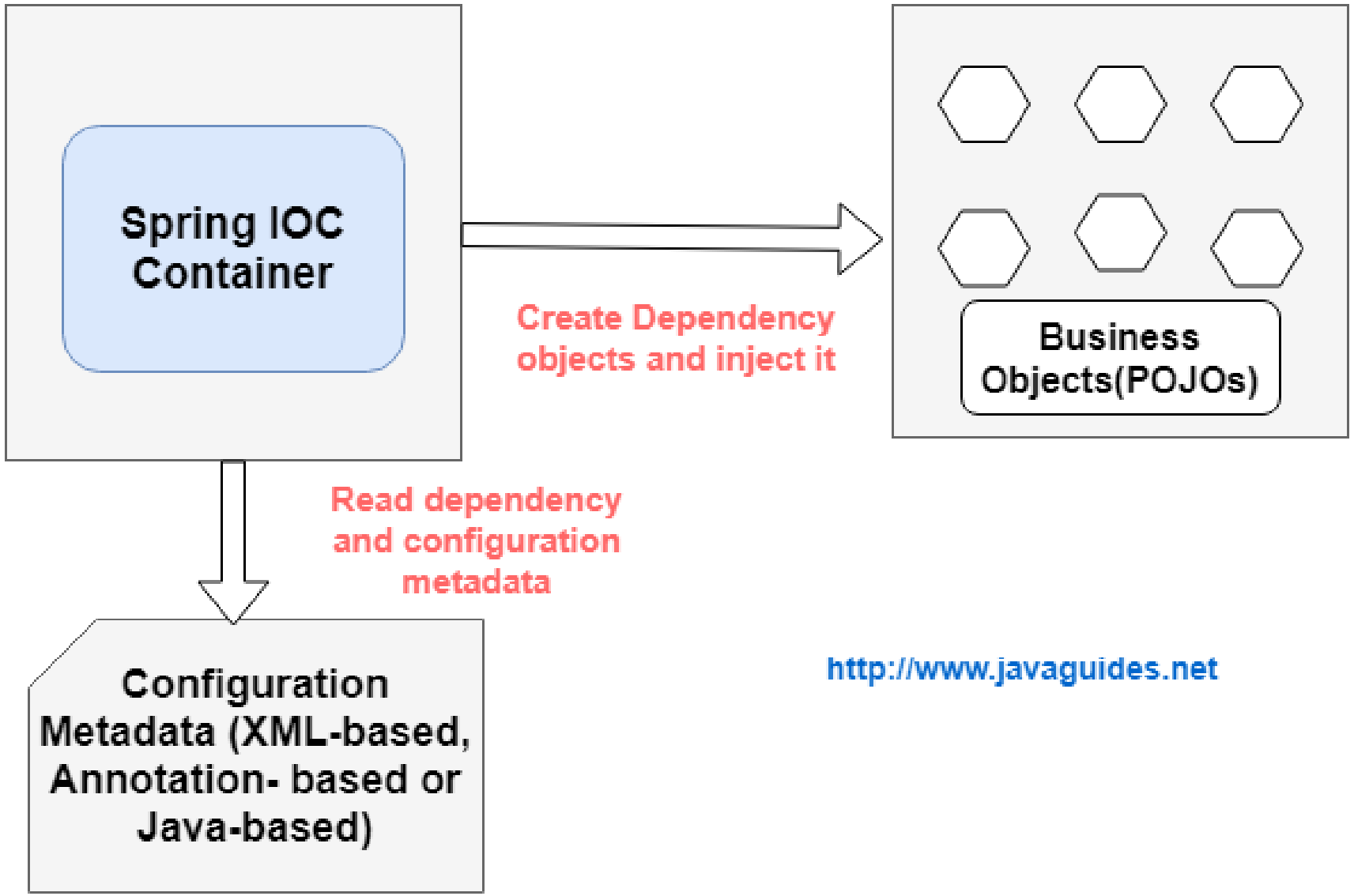
- Constructor của lớp ClassPathXmlApplicationContext nhận tham số truyền vào là một chuỗi, vì vậy chúng ta truyền tên của tập tin xml để tạo ra thể hiện của ApplicationContext.

Bean Spring

- Các đối tượng tạo thành xương sống của ứng dụng và được quản lý bởi Spring IoC container được gọi là Bean. Một bean là một đối tượng được khởi tạo, lắp ráp, và được quản lý bởi một Spring IoC container. Các bean này được tạo ra bằng siêu dữ liệu cấu hình mà bạn cung cấp cho container, ví dụ dưới dạng định nghĩa XML <bean/>.
- Có ba phương pháp quan trọng để cung cấp siêu dữ liệu cấu hình cho Spring IoC Container:
  - Tệp tin cấu hình dựa trên XML.
  - Cấu hình dựa trên Annotation.
  - Cấu hình dựa trên Java.
- Vòng đời của Bean trong Spring bao gồm khởi tạo, sử dụng và kết thúc. Mặc dù, có một danh sách các hoạt động xảy ra đằng sau thời điểm bắt đầu và trước khi bean bị hủy.

Phạm vi của Bean:

No.	Phạm vi & Mô tả
1	**singleton** Đây là phạm vi mặc định, phạm vi này nói với Spring IoC container rằng chỉ tạo ra một thể hiện duy nhất của bean này trên toàn bộ dứng dụng.
2	**prototype** Phạm vi này nói với Spring IoC container rằng tạo ra một thể hiện mới của bean mỗi khi cần thiết.
3	**request** Phạm vi này định nghĩa một bean cho một HTTP request. Chỉ có giá trị trong ngữ cảnh của một ứng dụng Spring Web.
4	**session** Phạm vi này định nghĩa một bean cho một HTTP session. Chỉ có giá trị trong ngữ cảnh của một ứng dụng Spring Web.
5	**global-session** Phạm vi này định nghĩa một bean cho một global HTTP session. Chỉ có giá trị trong ngữ


Spring IOC Container
Create Dependency objects and inject it
Business Objects(POJOs)
Read dependency and configuration metadata
Configuration Metadata (XML-based, Annotation- based or Java-based)
http://www.javaguides.net

Bean Spring là gi?

- Spring Bean là các đối tượng (object) trong Spring Framework, được khởi tạo thông qua Spring Container. Bất kỳ class Java POJO nào cũng có thể là Spring Bean nếu nó được cấu hình và khởi tạo thông qua container bằng việc cung cấp các thông tin cấu hình (các file config .xml, .properties..)
- Có 5 scope được định nghĩa cho Spring Bean:

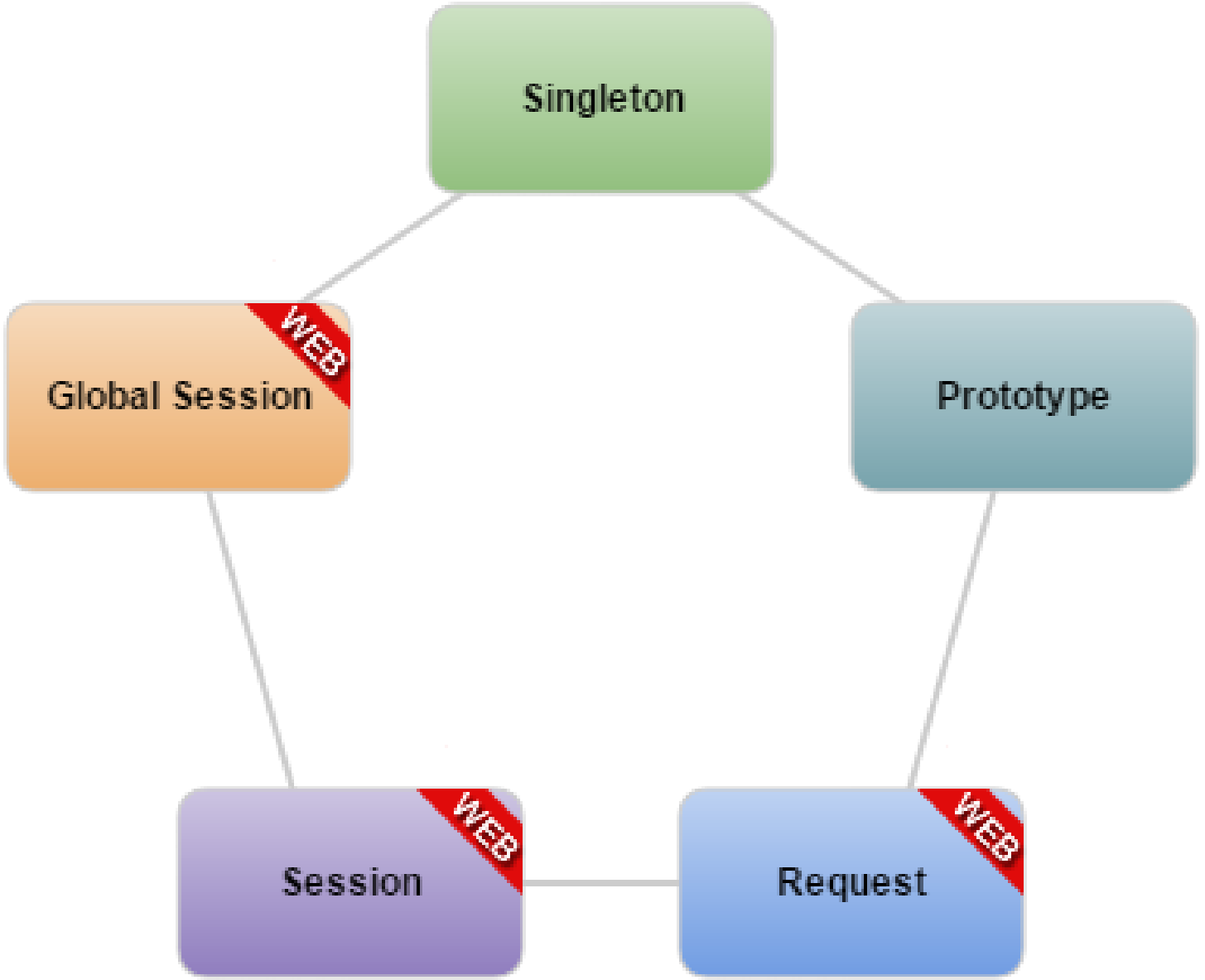
✓Singleton: Chỉ duy nhất một thể hiện của bean sẽ được tạo cho mỗi container. Đây là scope mặc định cho spring bean. Khi sử dụng scope này cần chắc chắn rằng các bean không có các biến/thuộc tính được share.

✓Prototype: Một thể hiện của bean sẽ được tạo cho mỗi lần được yêu cầu(request)

✓Request: giống với prototype scope, tuy nhiên nó dùng cho ứng dụng web, một thể hiện của bean sẽ được tạo cho mỗi HTTP request.

✓Session: Mỗi thể hiện của bean sẽ được tạo cho mỗi HTTP Session

✓Global-Session: Được sử dụng để tạo global sesion bean cho các ứng dụng Portlet.


Singleton
Global Session
WEB
Prototype
Session
WEB
Request
WEB

Development Steps

Follow these five steps to develop a spring application:

- Create a simple Maven Project
- Project Structure
- Add Maven Dependencies
- Configure HelloWorld Spring Beans
- Create a Spring Container
- Retrieve Beans from Spring Container

https://www.javaguides.net/2018/10/spring-ioc-container-java-config-example.html

Tài liệu:

- https://spring.io/projects/spring-framework
- https://docs.spring.io/spring-framework/reference/overview.html
- https://techmaster.vn/posts/36083/spring-core-phan-1-spring-ioc-inversion-of-control-trong-spring
- https://viblo.asia/p/spring-core-Ljy5VpnVZra
- https://www.digitalocean.com/community/tutorials/spring-tutorial-spring-core-tutorial
- https://www.javatpoint.com/spring-tutorial
- https://www.tutorialspoint.com/spring/index.htm

Ví dụ laapk trình Spring:

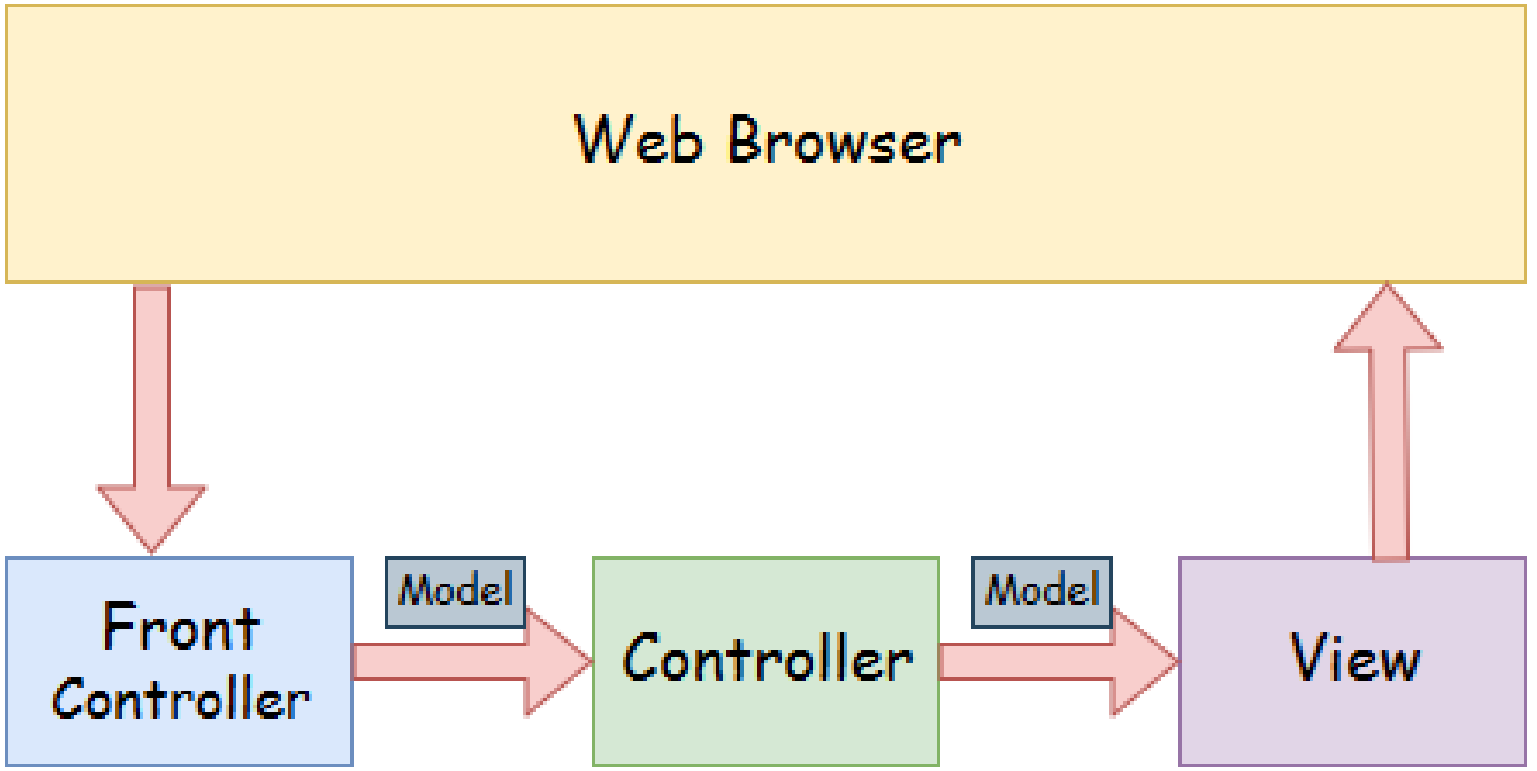
https://github.com/RameshMF/spring-core-tutorial

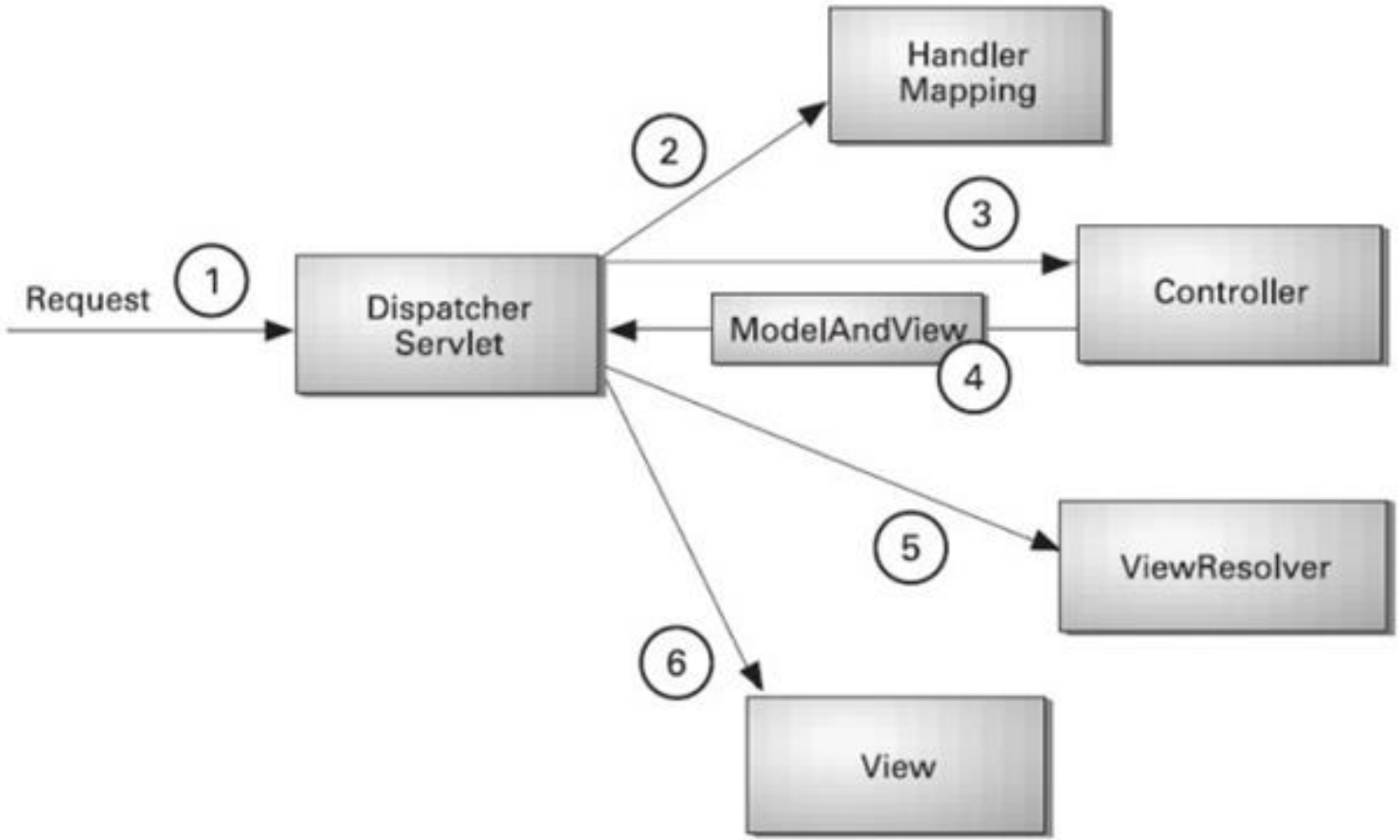
Thực hành:

- https://viettuts.vn/springhttps://viettuts.vn/spring/tao-spring-project-bang-spring-tool-suite-trong-eclipse
- https://viettuts.vn/spring/spring-hello-world-annotation-example
- https://viettuts.vn/spring/vi-du-login-trong-spring-4-web-mvc-hibernate-4-xml-mapping
- https://viettuts.vn/spring/vi-du-login-trong-spring-4-web-mvc-hibernate-4-annotation

SPRING MVC

Mô hình MVC:




Handler Mapping
2
3
Request
1
Dispatcher Servlet
ModelAndView
Controller
4
5
ViewResolver
6
View

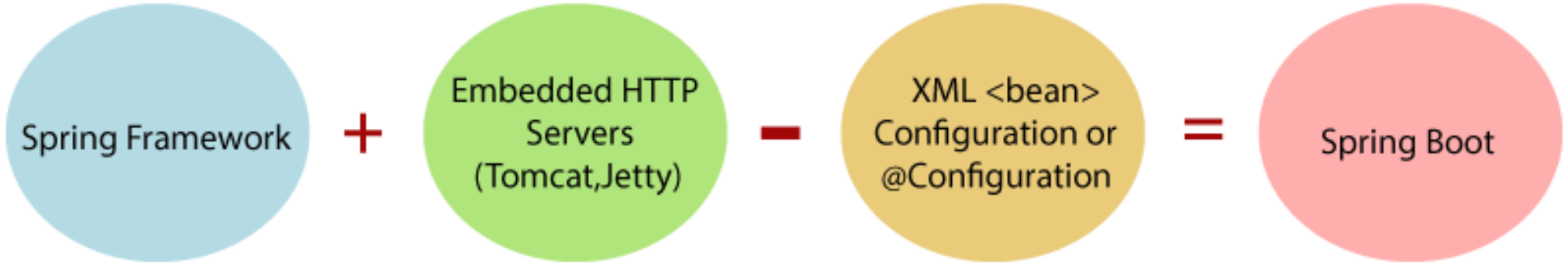
Thực hành:

- https://www.javatpoint.com/spring-mvc-crud-example
- https://www.javatpoint.com/spring-mvc-pagination-example
- https://www.javatpoint.com/spring-mvc-file-upload

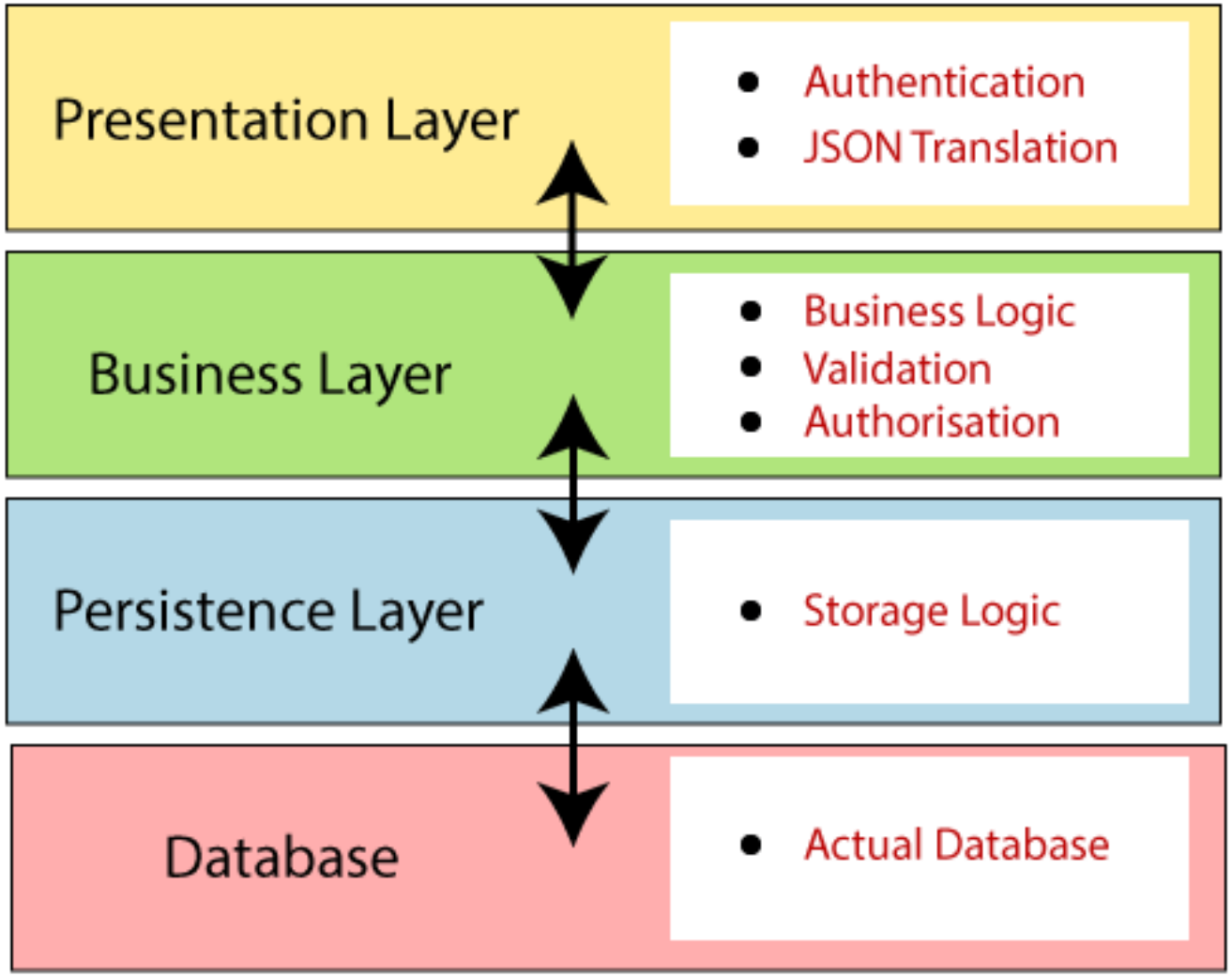
SPRING BOOT

Giới thiệu SB:

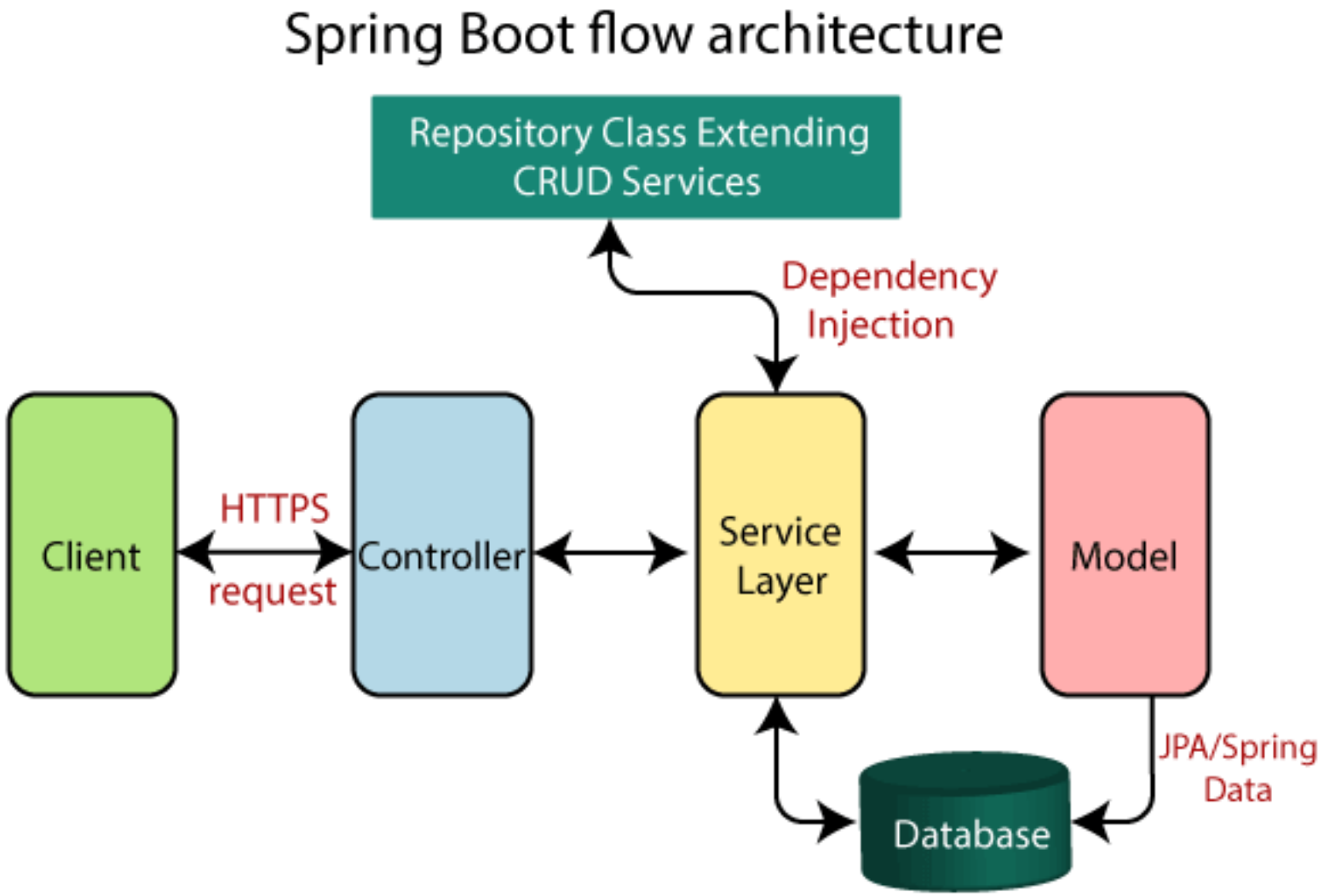
- Mô-đun Spring cung cấp tính năng RAD (Phát triển ứng dụng nhanh) cho Spring Framework.
- Nó được sử dụng để tạo một ứng dụng dựa trên Spring độc lập và có thể chạy với cấu hình Spring tối thiểu.
- Spring Boot là sự kết hợp giữa Spring Framework và Embedded Servers.
- Trong Spring Boot, không có yêu cầu đối với cấu hình XML (bộ mô tả triển khai).
- Để phát triển ứng dụng Java SB có thể dùng: Spring STS IDE hoặc Spring Initializr



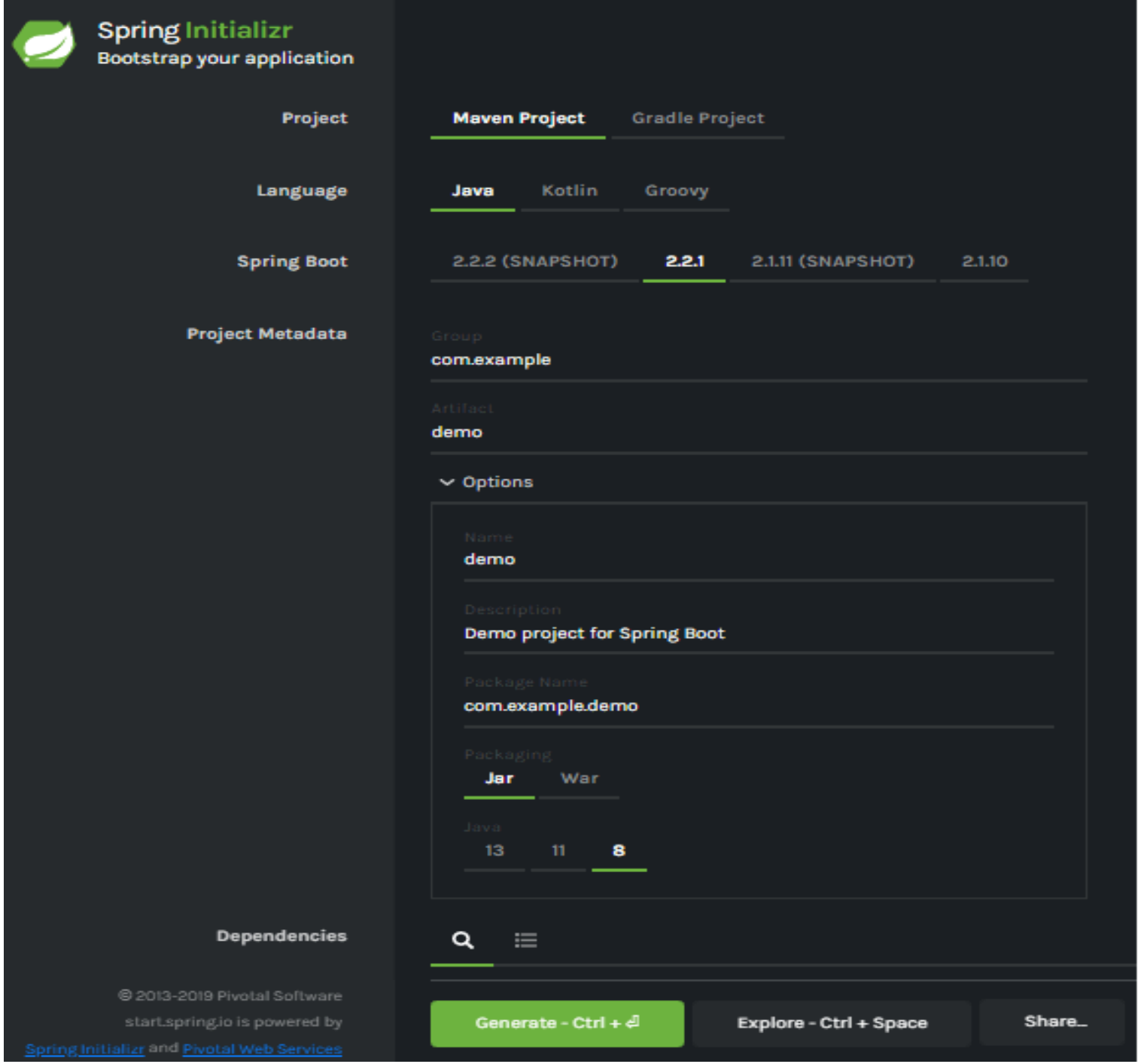
Kiến trúc của SB:



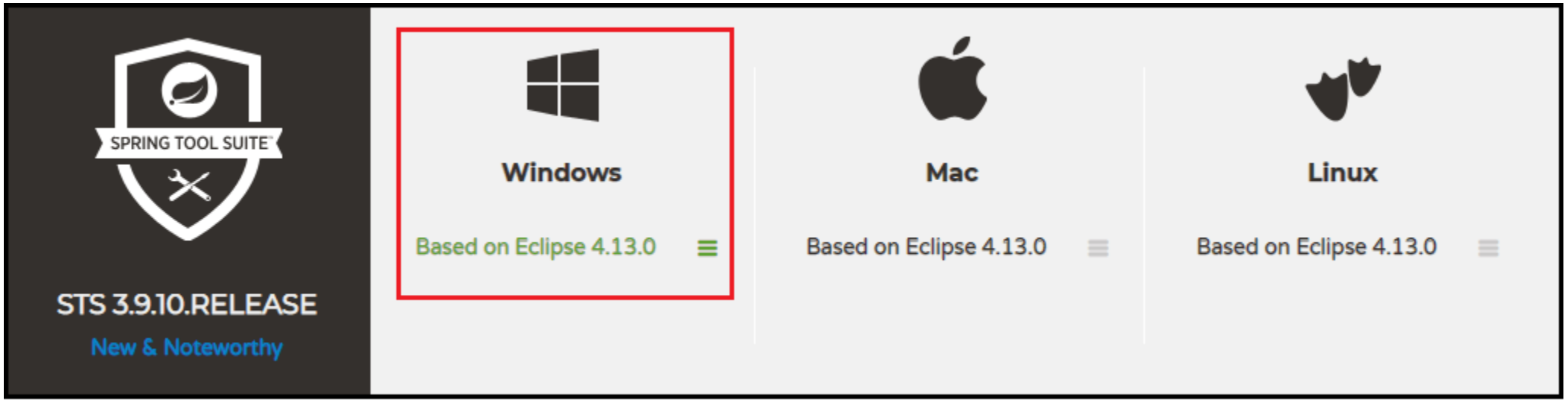
Kiến trúc luồng SB:



Tạo ứng dụng SB với Spring Initializr



Tạo ứng dụng với STS IDE:



Cài đặt STS trong Eclipse: https://viettuts.vn/spring/cai-dat-spring-tool-suite-sts-trong-eclipse

Ví dụ: Hello world:

- https://www.javatpoint.com/spring-boot-hello-world-example

Triển khai dự án với Tomcat:

- https://www.javatpoint.com/spring-boot-project-deployment-using-tomcat

Triển khai một dự án REST- SB:

- https://www.javatpoint.com/restful-web-services-spring-boot
- https://www.javatpoint.com/spring-boot-rest-example
- Các ví dụ khác:

https://viettuts.vn/spring/spring-hello-world-example

- Chạy lại service Edge-service và xem kết quả trả về là một list rỗng.
- Chạy lại service Item-catalog-service, truy cập lại Edge-service và ta lại có được list item như trước.

https://spring.io/blog/2015/07/14/microservices-with-spring

https://www.edureka.co/blog/microservices-with-spring-boot

Bài tập:

Thiết kế chi tiết hệ thống booking bằng micro service hỗ trợ chịu tải lớn bằng Java và Spring boot, MySQL

https://viblo.asia/p/thiet-ke-chi-tiet-he-thong-booking-bang-micro-service-ho-tro-chiu-tai-lon-bang-java-va-spring-boot-mysql-AZoJjXeyVY7