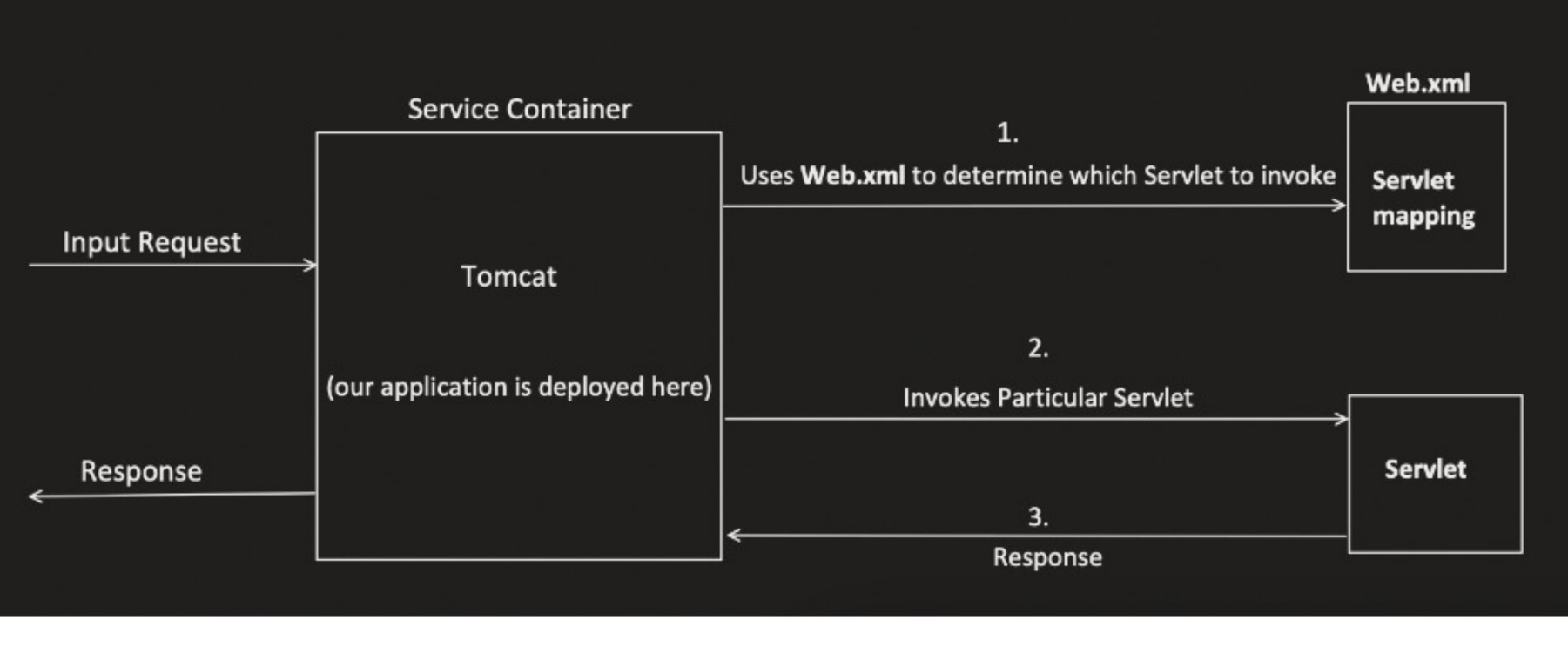
Spring boot : Introduction

"Concept && Coding" YT Video Notes

Before we talk about Spring or Spring Boot, first we need to understand about "**SERVLET**" and "**Servlet Container**"

- Provide foundation for building web applications.
- Servlet is a Java Class, which handles client request, process it and return the response.
- And Servlet Container are the ones which manages the Servlets.



Servlet1:

```java
@WebServlet("/demoservletone/*")
public class DemoServlet1 extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {

        String requestPathInfo = request.getPathInfo();

        if(requestPathInfo.equals("/")) {
            //do something
        }
        else if(requestPathInfo.equals("/firstendpoint")) {
            //do something
        }
        else if(requestPathInfo.equals("/secondendpoint")) {
            //do something
        }
    }

    @Override
    protected void doPut(HttpServletRequest request,
                         HttpServletResponse response) {
        //do something
    }
}
```

Web.xml

```xml
<!-- my first servlet configuration below-->
<servlet>
    <servlet-name>DemoServlet1</servlet-name>
    <servlet-class>DemoServlet1</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>DemoServlet1</servlet-name>
    <url-pattern>/demoservletone</url-pattern>
    <url-pattern>/demoservletone/firstendpoint</url-pattern>
    <url-pattern>/demoservletone/secondendpoint</url-pattern>
</servlet-mapping>

<!-- my second servlet configuration below-->
<servlet>
    <servlet-name>DemoServlet2</servlet-name>
    <servlet-class>DemoServlet2</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>DemoServlet2</servlet-name>
    <url-pattern>/demoservlettwo</url-pattern>
</servlet-mapping>
```

Servlet2:

```java
@WebServlet("/demoservlettwo/*")
public class DemoServlet2 extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) {

        //do something
    }

    @Override
    protected void doPut(HttpServletRequest request,
                         HttpServletResponse response) {
        //do something
    }
}
```

Spring Framework, solve challenges which exists with Servlets.

- Removal of web.xml
  - this web.xml over the time becomes too big and becomes very difficult to manage and understand.
  - Spring framework introduced Annotations based configuration.
- Inversion of Control (IoC)
  - Servlets depends on Servlet container to create object and maintain its lifecycle.
  - IoC is more flexible way to manage object dependencies and its lifecycle (though Dependency injection)
- Unit Testing is much harder
  - As the object creation depends on Servlet container, mocking is not easy. Which makes Unit testing process harder.
  - Spring dependency injection facility makes the Unit testing very easy.
- Difficult to manage REST APIs
  - Handling different HTTP methods, request parameters, path mapping make code little difficult to understand.
  - Spring MVC provides an organised approach to handle the requests and its easy to build RESTful APIs.

There are many other areas where Spring framework makes developer life easy such as : integration with other technology like hibernate, adding security etc...

The most important feature of Spring framework is **DEPENDENCY INJECTION** or **Inversion of Control (IoC)**

Lets see an example **without** Dependency Injection:

```java
public class Payment {

    User sender = new User();

    void getSenderDetails(String userID){
        sender.getUserDetails(userID);
    }
}
```

```java
public class User {

    public void getUserDetails(String id) {
        //do something
    }
}
```

Payment class is creating an instance of User class, and there is one Major problems with this and i.e.

Tight coupling: Now payment class is tightly coupled with User class.

How?
-> Suppose I want to write Unit test cases for Payment "getSenderDetails()" method, but now I can not easily MOCK "User" object, as Payment class is creating new object of User, so it will invoke the method of User class too.

-> Suppose in future, we have different types of User like "admin", "Member" etc., then with this logic, I can not change the user dynamically.

Now, Lets see an example **with** Dependency Injection:

```java
@Component
public class Payment {

    @Autowired
    User sender;

    void getSenderDetails(String userID){
        sender.getUserDetails(userID);
    }
}
```

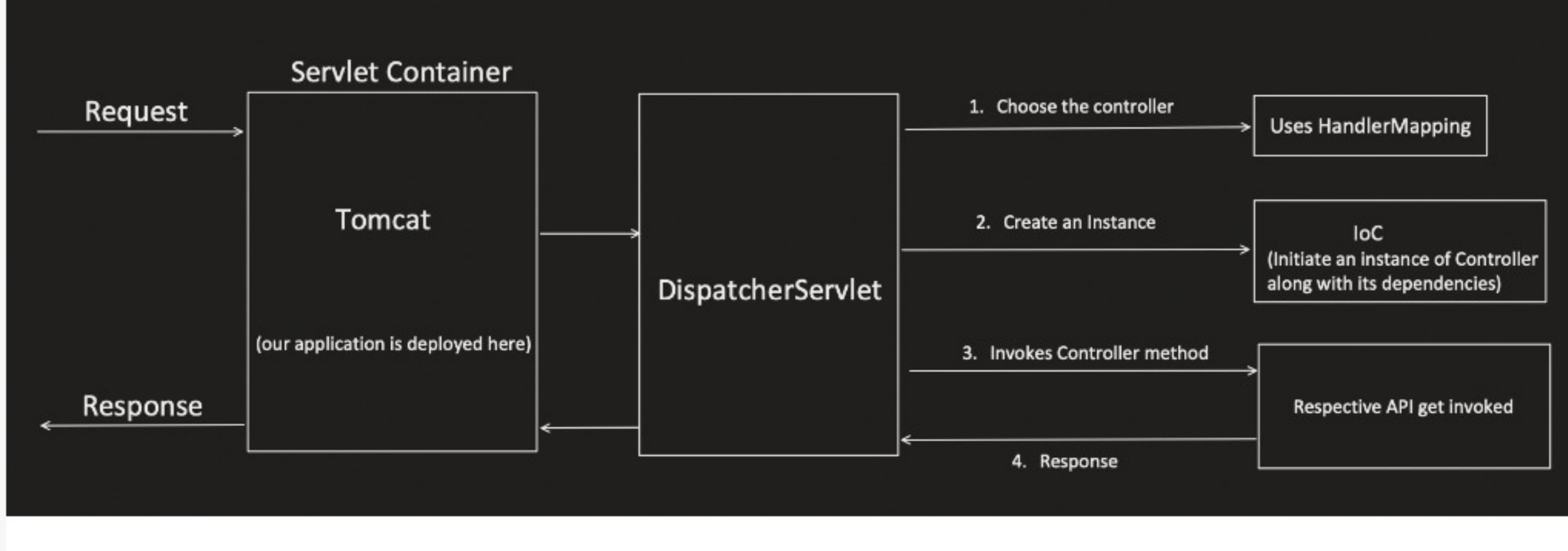
```java
@Component
public class User {

    public void getUserDetails(String id) {
        //do something
    }
}
```

@Component: tells Spring that, you have to manage this class or bean.
@Autowired: tells Spring to resolve and add this object dependency.

The another important feature of Spring framework is lot of **INTEGRATION** available with other frameworks.

This allow Developers to choose different combination of technologies and framework which best fits their requirements like:

- Integration with Unit testing framework like Junit or Mockito.
- Integration with Data Access framework like Hibernate, JDBC, JPA etc.
- Integration with Asynchronous programming.
- Similar way, it has different integration available for:
  - Caching
  - Messaging
  - Security etc.



pom.xml

```xml
<modelVersion>4.0.0</modelVersion>
<groupId>com.conceptandcoding</groupId>
<artifactId>learningspringboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot application</name>
<description>project for learning springboot</description>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>6.1.4</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.5</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

Controller class

```java
@Controller
@RequestMapping("/paymentapi")
public class PaymentController {

    @Autowired
    PaymentDAO paymentService;

    @GetMapping("/payment")
    public String getPaymentDetails() {
        return paymentService.getDetails();
    }
}
```

Config class

```java
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.conceptandcoding")
public class AppConfig {
    // add configuration here if required
}
```

Dispatcher Servlet class

```java
public class MyApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{AppConfig.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }
}
```

Spring Boot, solve challenges which exists with Spring MVC.

1. Dependency Management: No need for adding different dependencies separately and also their compatible version headache.

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.conceptandcoding</groupId>
<artifactId>learningspringboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot application</name>
<description>project for learning springboot</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2. Auto Configuration : No need for separately configuring "DispatcherServlet", "AppConfig" , "EnableWebMvc", "ComponentScan". Spring boot add internally by-default.

```java
@SpringBootApplication
public class SpringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

3. Embedded Server:

In traditional Spring MVC application, we need to build a WAR file, which is a packaged file containing your application's classes, JSP pages, configuration files, and dependencies. Then we need to deploy this WAR file to a servlet container like Tomcat.

But in Spring boot, Servlet container is already embedded, we don't have to do all this stuff. Just run the application, that's all.

So, what is Spring boot?

- It provides a quick way to create a production ready application.
- It is based on Spring framework.
- It support **"Convention over Configuration".**
  Use default values for configuration, and if developer don't want to go with convention(the way something is done), they can override it.
- It also help to run an application as quick as possible.

pom.xml

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.3</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.conceptandcoding</groupId>
<artifactId>learningspringboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springboot application</name>
<description>project for learning springboot</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

```java
@SpringBootApplication
public class SpringbootApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

```java
@RestController
@RequestMapping("/myapi")
public class MyController {

    @GetMapping("/firstapi")
    public String getData() {
        return "Hello from concept and coding";
    }
}
```