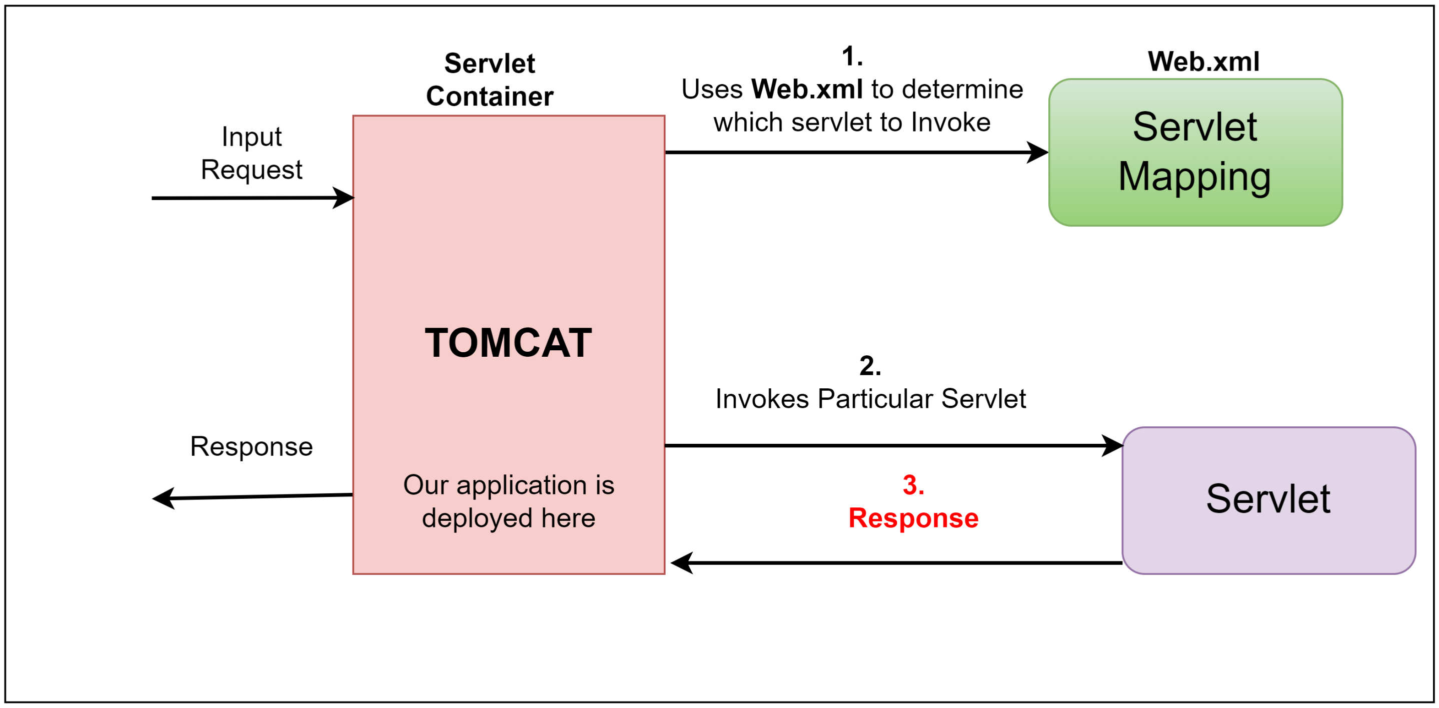
Introduction To Spring Boot Java

1. **Why Spring Boot exist ?**

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 java class, which handle client request, process it and return the response.
- Servlet container are the ones which manage the Servlets.


Servlet Container
Input Request
TOMCAT
Our application is deployed here
Response
1. Uses Web.xml to determine which servlet to Invoke
Web.xml
Servlet Mapping
2. Invokes Particular Servlet
3. Response
Servlet

```
package spring_boot.servlet_and_servlet_container;
@WebServlet("/servletOne")
public class Servlet1 extends HttpServlet{

    @Override
    protected void doGet(HttpServletRequest req,HttpServletResponse res){
        String requestPathInfo = req.getPathInfo();
        if(requestPathInfo.equals("/")){
            //do something
        }
        else  if (requestPathInfo.equals("/hi")){
            //do something
        }
        else  if (requestPathInfo.equals("/bye")){
            //do something
        }
    }

    @Override
    protected void doPost(HttpServletRequest req,HttpServletResponse res){
        String requestPathInfo = req.getPathInfo();

       //do something
    }

}
```

```
<servlet>
    <servlet-name>Servlet1</servlet-name>
    <servlet-class>Servlet1</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Servlet1</servlet-name>
    <url-pattern>/servletOne</url-pattern>
    <url-pattern>/servletOne/hi</url-pattern>
    <url-pattern>/servletOne/bye</url-pattern>
</servlet-mapping>



<servlet>
<servlet-name>Servlet2</servlet-name>
<servlet-class>Servlet2</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>Servlet2</servlet-name>
<url-pattern>/servletTwo</url-pattern>
<url-pattern>/servletTwo/hello</url-pattern>
<url-pattern>/servletTwo/gello</url-pattern>
</servlet-mapping>
```

Spring Framework, solve challenges which exists with Servlets.

- Removal of web.xml

  This web.xml over the time becomes too big and becomes too big and becomes very difficult to manage and understand.

   Spring framework introduced Annotations based configuration.

- Request filtering is done by web.xml

- **Inversion of Control [IOC]**

Dependency Injection is implementation Of IOC.

As the object creation depends on Servlet, Mocking is not easy. Which makes unit testing process harder.

```
public class User{

   public void getSenderDetails(String userId){
      //do something
   }
}
```

```
public class Payment{
   User sender = new User();
   void getSenderDetails(String userId){
      sender.getUserDetails(userId);
   }
}
```

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

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

How?

Suppose I want to write Unit test cases

for payment "**getUserDetails()**" method, but now i can not eaisly MOCK "User" object , as Payment class is creating new object of User, so it will invoke the method of user class too.

Only Invoke getUserDetails method Of Payment class.

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 Let's see an example with Dependency Injection.

- Spring dependency injection facility makes the Unit testing very easy.

```
@Component
public class User{

   public void getSenderDetails(String userId){
      //do something
   }
}
```

```
@Component
public class Payment{
   User sender = new User();
   @Autowired
   void getSenderDetails(String userId){
      sender.getUserDetails(userId);
   }
}
```

@Component: tell Spring that, you have class or bean.

@Autowired: tell Spring to resolve and add this object dependency.

Now Mocking is eaisly for Unit test cases.

This whole process is known as Dependency Injection.

- **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 it's easy to build RESTful APIs.

- There are many other areas where Spring framework makes developer life easy such as: integeration with other technology like hibernate, adding security etc..
- This allow Developers to choose different combination of technologies and frame work which best fits their requirements like:

Integeration with Unit testing framework like Junit or Mockito.

Integeration with Data access framework like Hibernate, JDBC, JPA etc.

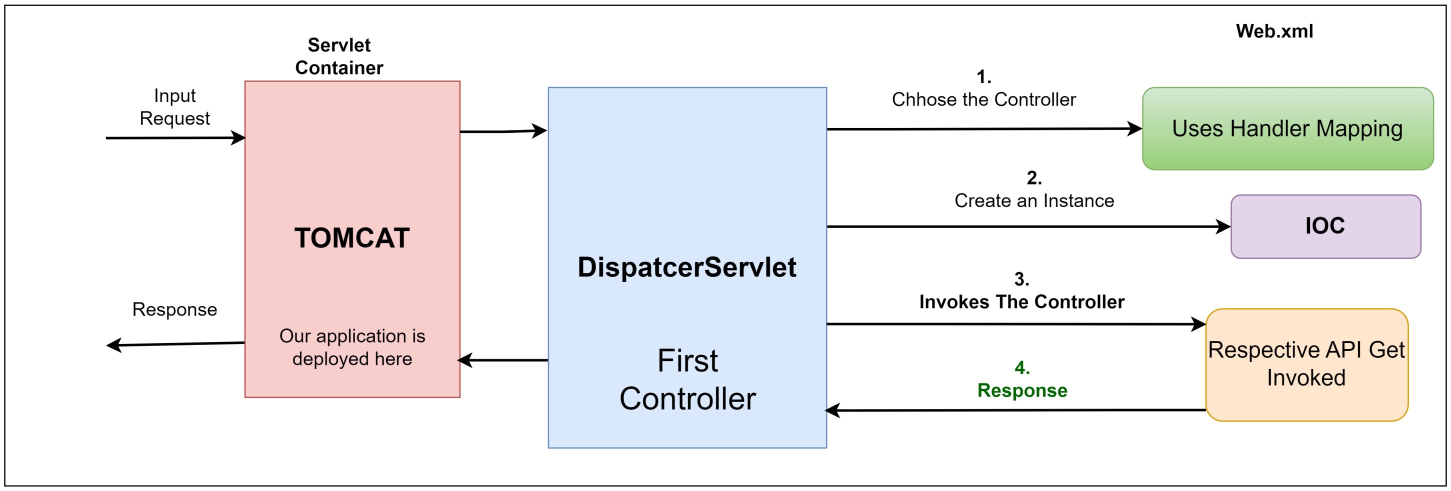
Integeration with Asynchronous programming.

Similar Way, it has different integeraton available for:

  Casching

  Messaging

  Security etc.



```
@Controller
@RequestMapping("/paymentapi")
public class PaymentController{
  @Autowired
  PaymentDAO paymentServices;

  @GetMapping("makePayment")
  public String getPaymentDetails()
  {
     return paymentServices.getPaymentDetails();
  }
}
```

- DispatcherServlet going to create object of First Controller which is PaymentController Using IOC and resolse other dependency injection for PaymentDAO creating object and inject into PaymentDAO object.

SPRING MVC EXAMPLE:

```
@Controller
@RequestMapping("/paymentapi")
public class PaymentController{
  @Autowired
  PaymentDAO paymentServices;

  @GetMapping("makePayment")
  public String getPaymentDetails()
  {
     return paymentServices.getPaymentDetails();
  }
}
```

//pom.xml

All dependency for running spring application like junit.

Config.class

```
@Configuration
@EnableWebMvc//load all the dependency which is required
@Component(basePackages = "com.codesake")
public class AppConfig{
  //add configuration here if required.
}
```

Dispatcher Servlet class

```
public class MyApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer{
  @Override
  protected Class<?> [] getRootConfigClasses(){
     return null;
  }

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

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

Spring Boot, solve challenges which exist with Spring MVC

- Dependency Managent: No need for adding different dependencies seperately and also compatible version headache.

 All version are loaded for different dependencies which is cmpatible to spring-boot-starter-parent version.

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.5</version>
        <relativePath/> <!-- lookup parent from repository -->
</parent>
```

- Auto Configuration: No need for seperately configuring "DispatcherServlet" , "AppConfig", "EnableWebMvc", "ComponentScan", Spring boot add internally by-default.
- Spring boot has opiniated nature.

 In spring boot @ComponentScan gves where ever your spring boot main method start i will take that start package

which is "package codesake.in.securecapita"

In case if you are not agree with default nature you can add mannully like this

@Component(basePackages = "com.codesake")

```
package codesake.in.securecapita;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
```

```
//securecapitabackendspringboot-production.up.railway.app
@SpringBootApplication(exclude = {SecurityAutoConfiguration.class})
@EnableAsync
public class SecurecapitaApplication {

    private static final int STRENGTH = 12;
    public static void main(String[] args) {
        SpringApplication.run(SecurecapitaApplication.class, args);
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder(STRENGTH);
    }

}
```

- 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 embede, we don't have to do all this stuff. just the application, that's all.

So, What is Spring Boot.

- It provides a quick way to create a production ready applicaton.
- it is based on spring frame work.
- It support "Convention over configuration"

Use default values for configuration, anf 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.