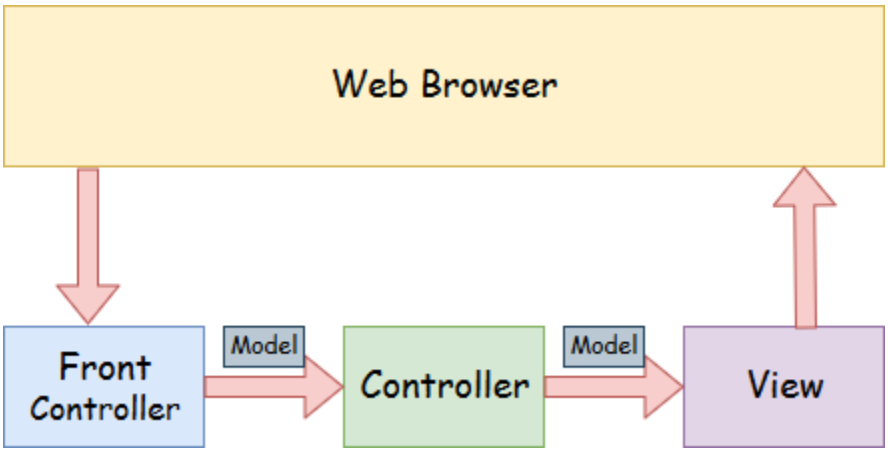
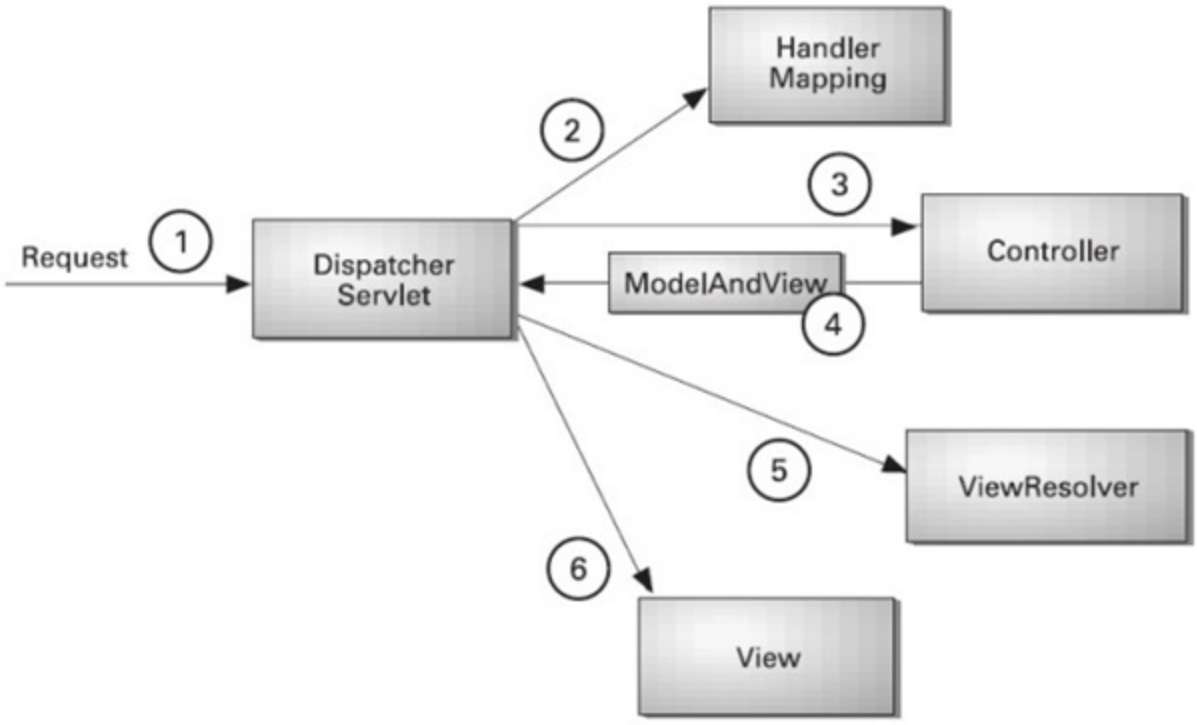
Spring MVC framework

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. **It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.**

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.



- Model - A model contains the data of the application. A data can be a single object or a collection of objects.
- Controller - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
- View - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
- Front Controller - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

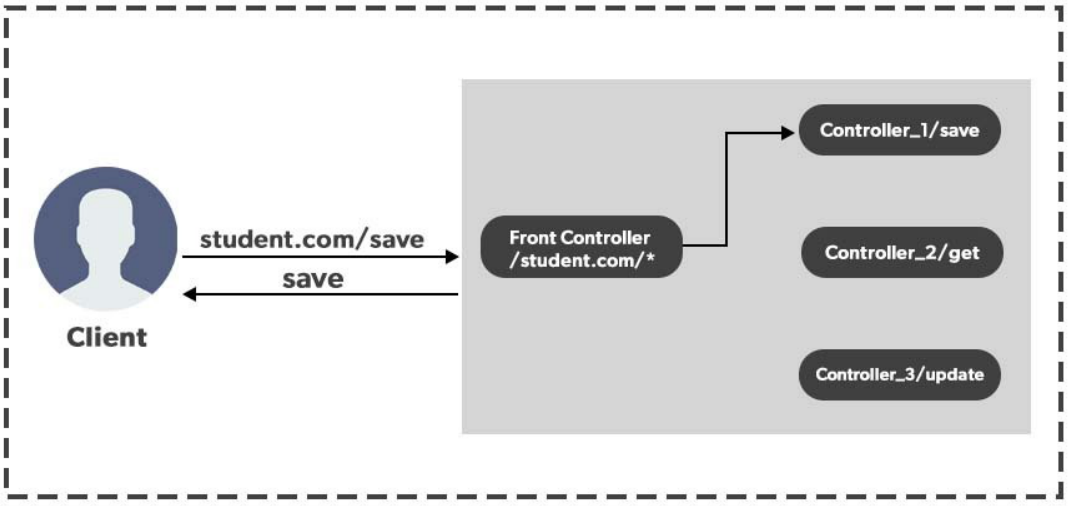


- all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
- The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
- The controller returns an object of ModelAndView.
- The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

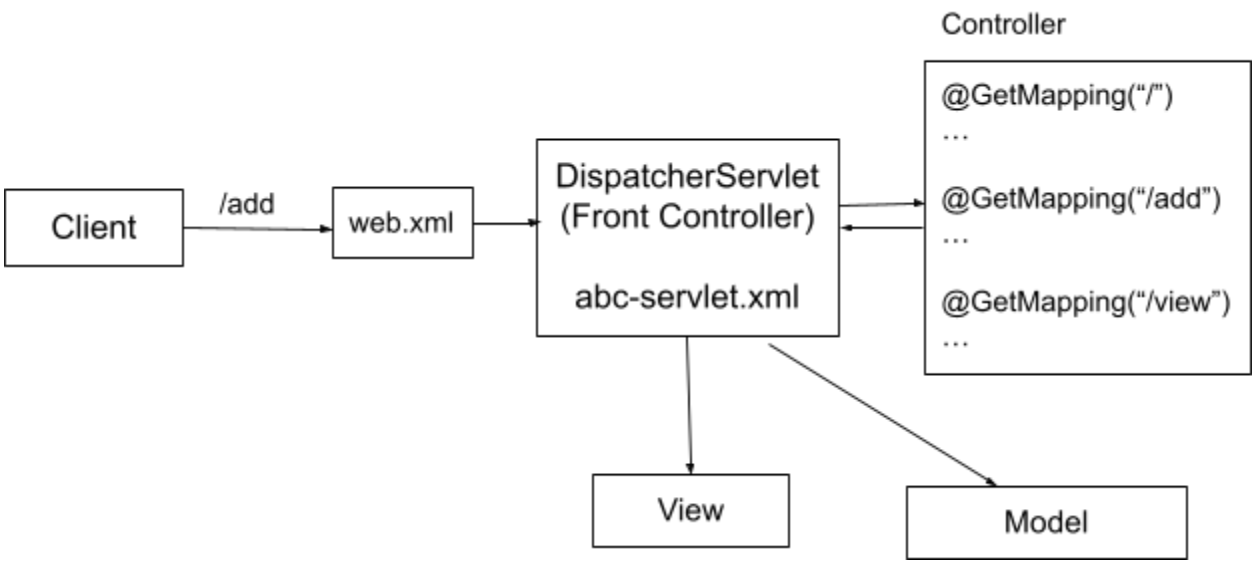
DispatcherServlet

DispatcherServlet acts as the **Front Controller** for Spring-based web applications. So now what is Front Controller? So it is pretty simple. Any request is going to come into our website the front controller is going to stand in front and is going to accept all the requests and once the front controller accepts that request then this is the job of the front controller that it will make a decision that who is the right controller to handle that

request. For example, refer to the below image. Suppose we have a website called student.com and the client is make a request to save student data by hitting the following URL student.com/save and its first come to the front controller and once the front controller accepts that request it is going to assign to the Controller_1 as this controller handle the request for /save operation. Then it is going to return back the response to the Client.



DispatcherServlet handles an incoming HttpRequest, delegates the request, and processes that request according to the configured HandlerAdapter interfaces that have been implemented within the Spring application along with accompanying annotations specifying handlers, controller endpoints, and response objects.



Pom.xml

```
<plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-war-plugin</artifactId>
                                <version>3.3.1</version>
                        </plugin>
</plugins>


<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webmvc</artifactId>
                        <version>5.3.24</version>
```

```
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.24</version>
        </dependency>
```

Web.xml

```
 <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

Springmvc-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:ctx="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd ">


        <ctx:annotation-config></ctx:annotation-config>
        <ctx:component-scan base-package=""></ctx:component-scan>
</beans>
```

Annotation based configuration

SpringMvcConfig.java

```
@Configuration
```

```
@ComponentScan({"com.pmm"})
public class SpringMvcConfig {



	@Bean
	public InternalResourceViewResolver viewResolver()
	{
		InternalResourceViewResolver viewR = new InternalResourceViewResolver();
		viewR.setPrefix("/WEB-INF/views/");
		viewR.setSuffix(".jsp");
		return viewR;
	}

}
```

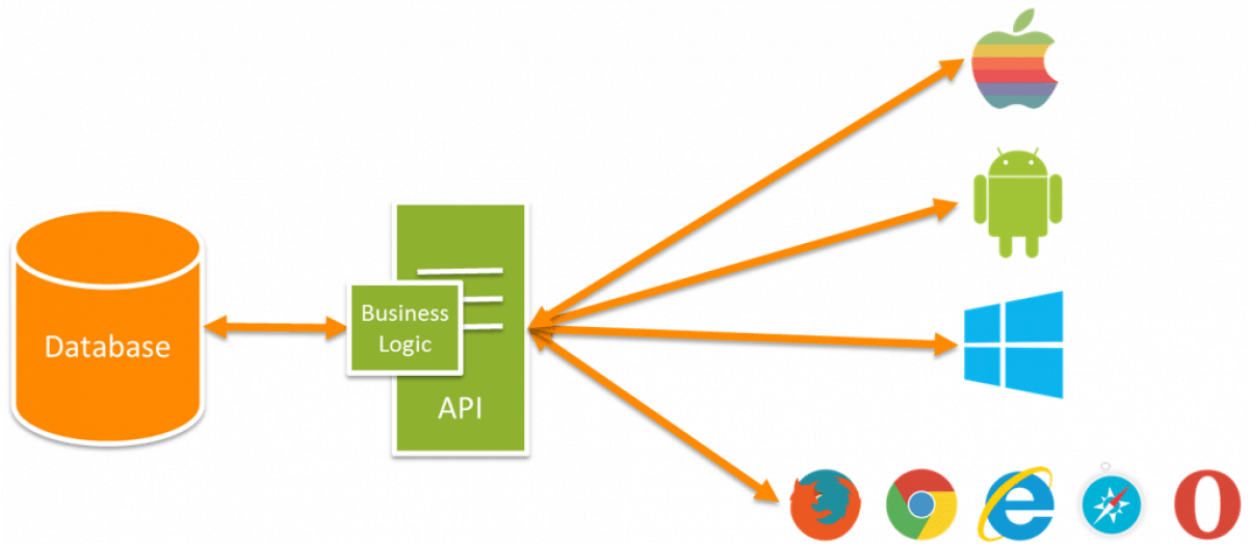
WebInitializer.java

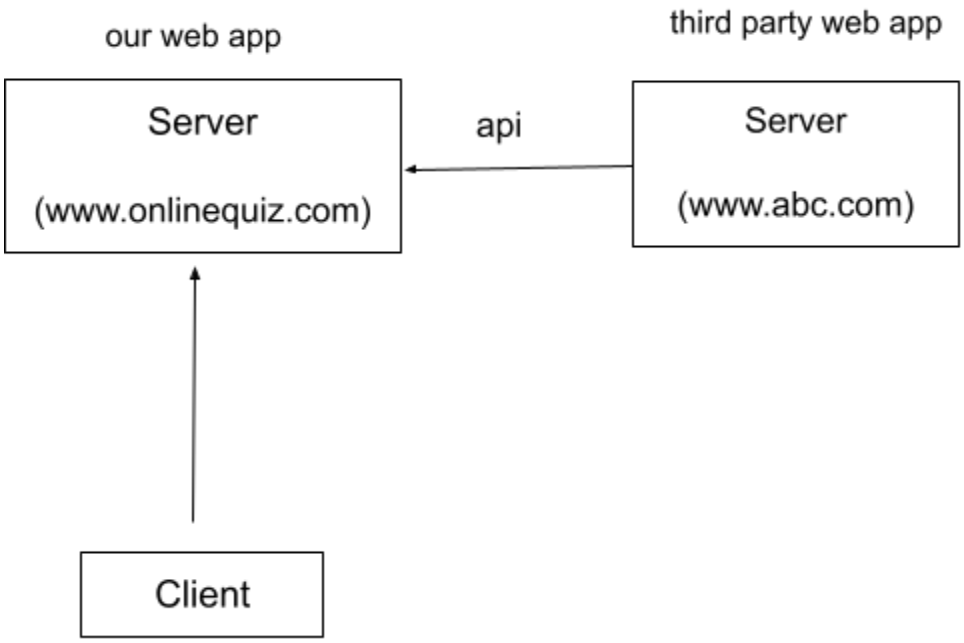
```
public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {


	@Override
	protected Class<?>[] getRootConfigClasses() {
		// TODO Auto-generated method stub
		return null;
	}


	@Override
```

```
	protected Class<?>[] getServletConfigClasses() {
		// TODO Auto-generated method stub
		return new Class[] {SpringMvcConfig.class};
	}

	@Override
	protected String[] getServletMappings() {
		// TODO Auto-generated method stub
		return new String[] {"/"};
	}

}
```

Application Programming Interface (API)

API is a software interface that allows two applications to interact with each other without any user intervention. API is a collection of software functions and procedures. In simple terms, API means a software code that can be accessed or executed. API is defined as a code that helps two different software's to communicate and exchange data with each other.





api -> www.abc.com/gk/currentaffairs/

REST Api

An API, or *application programming interface*, is a set of rules that define how applications or devices can connect to and communicate with each other. A REST API is an API that conforms to the design principles of the REST, or *representational state transfer* architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs.

JSON

JSON stands for JavaScript Object Notation. JSON is a text format for storing and transporting data. JSON is "self-describing" and easy to understand.

```
const student1 = {
                    name:"John",
                   age:16,
                   city:"Chennai"
                };
```

Postman installation

DevTools

DevTools stands for **Developer Tool.** The aim of the module is to try and improve the development time while working with the Spring Boot application. Spring Boot DevTools pick up the changes and restart the application.

We can implement the DevTools in our project by adding the following dependency in the pom.xml file.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime<scope >
</dependency>
```

Spring Boot DevTools provides the following features:

- **Property Defaults**
- **Automatic Restart**
- **LiveReload**
- **Remote Debug Tunneling**
- **Remote Update and Restart**

Actuator

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the S**pring Boot actuator.**

Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.
You can enable or disable each individual endpoint and expose them (make them remotely accessible) over HTTP or JMX. An endpoint is considered to be available when it is both enabled and exposed. The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of /actuator is mapped to a URL. For example, by default, the health endpoint is mapped to /actuator/health.

Most commonly used endpoints:

/actuator/health
/actuator/env
/actuator/beans
/actuator/mappings

Exposing Endpoints:

To enable all endpoints we have to add following configuration in application.properties file

```
management.endpoints.web.exposure.include=*
```

Applying Security to Actuator Endpoints

To enable spring security by adding security dependency

pom.xml

```
<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Add Username and Password to your application

application.properties

```
spring.security.user.name=admin
spring.security.user.password=123
```