+Install tomcat

go the apache tomcat side and download
http://tomcat.apache.org/download-80.cgi
check ---->**Binary Distributions ---> core---> zip (donload)**

after installtion unzip folder

know setup the tomcat in eclipse

windows----> show view----->server-----> click the link create new server---->
choose the tomcat 8 --->and add the path of unzip tomacat ---> kill the process

```
kill $(ps -aef | grep java | grep apache | awk '{print $2}')
```

Downlaod Binary Distributions , common login jar,jstl and add in WEB-INF/lib

DispatcherServlet

Spring MVC also uses a front controller to receive all incoming request and delegates to other components for further processing e.g. Spring MVC controllers which are annotated using @Controller annotation and ViewResolvers e.g InternalResourceViewResolver class.

A Front Controller (see Patterns of Enterprise Application Architecture) is a common pattern in web application and used to receive request and delegate to other components in the application for actual processing.
The **DispatcherServlet** is a front controller e.g. it provides a single entry point for a client request to Spring MVC web application and forwards request to Spring MVC controllers for processing.

How does DispatcherServlet know that which request should be forwarded to which Controller?

DispatcherServlet is responsible for initialize the **WebApplicationContext** and it loads all configuration related to the web components like controllers, view resolver, interceptors etc., It will be loaded and initialized by calling init() method init() of DispatcherServlet will try to identify the Spring Configuration Document with naming conventions like “servlet_name-servlet.xml” then all beans can be identify.

How to configure DispatcherServlet in Spring?
The DispatcherServlet is like any other Servlet class and it has to be declared inside the deployment descriptor or web.xml file as shown below:

```
  <display-name>1Hello World</display-name>

  <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>
</web-app>
```

OR

```
  <servlet>

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

  <init-param>

  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/beans.xml,/WEB-INF/my-
servlet.xml</param-value>

  </init-param>

 </servlet>
<servlet-mapping>
<servlet-name>dispatcher</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

If your servlet does a lot of job on initialization e.g. DispatcherServlet which initializes all the beans declared in its web context e.g. controllers, view resolvers, and mapping handlers then it could slow down the response time.

How dispatcher servlet works internally?

It acts as a front controller and provides a single entry point for the application. It then uses handler mappings and handler adapters to map a request to the Spring MVC controllers. It uses @Controller and @RequestMapping annotation for that purpose.

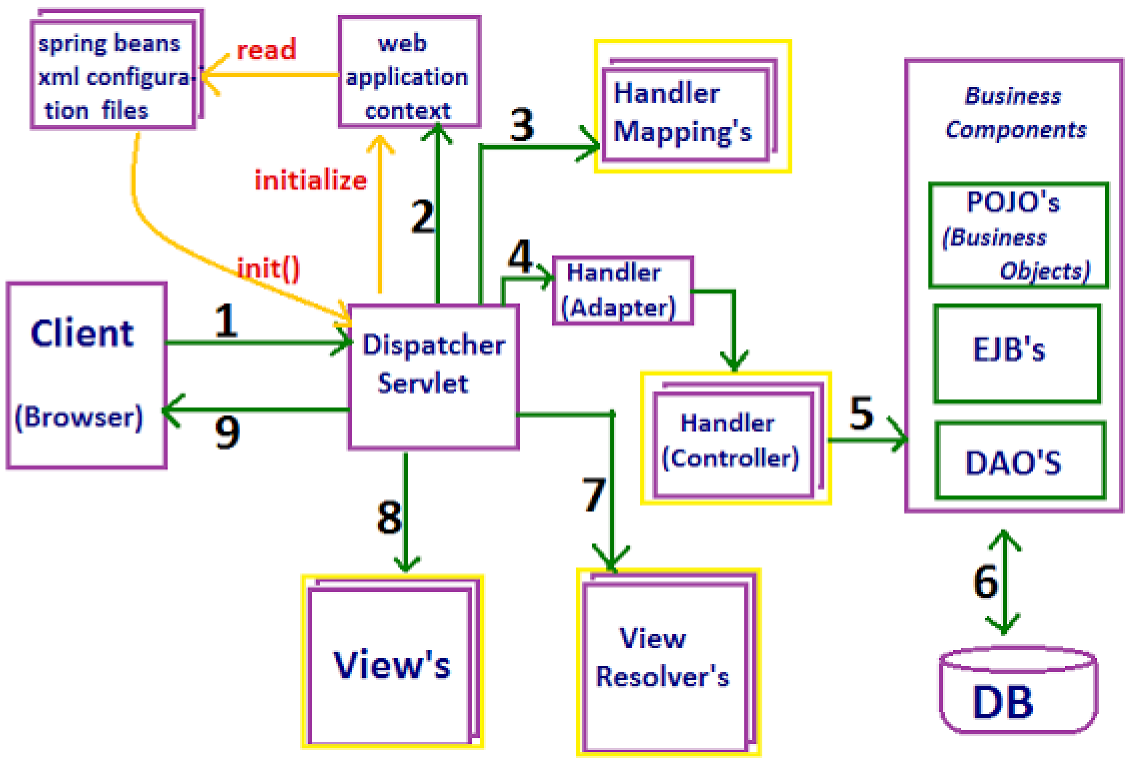
Once the request is processed by Spring MVC controller, it returns a logical view name instead of the view. Though, you can even configure Controler's handler methods to not return any View name by declaring return type as void. You can even use @ResponseBody annotation in the case of REST to directly write the output to the HTTP response body. See REST with Spring course by Eugen to learn more about developing RESTful web services using Spring MVC.

When DispatherServlet receives view name, it consults the ViewResolver to find the right view. There is a chain of ViewResolver is maintained at Spring MVC framework. They try to resolve the logical view name into a Physical resource e.g. a JSP page or a FreeMaker or Velocity template.

The **ViewResolver** are invoked in an order, if first in the chain not able to resolve the view then it returns null and next ViewResolver in the chain is consults. Once the right view is found, DispatcherServlet forwards the request along with Model data to the View for rendering e.g. a JSP page.

By default, DispatcherServlet uses **InternalResourceViewResolver** which uses prefix and suffix to convert a logical view name e.g. "home" to /WEB-INF/home.jsp. The View interface also has getContentType() method, which returns content type the view produces (JstlView has text/html). This is usually the default content type for requests handled by the dispatcher servlet in Spring.

Here is a nice diagram which explains how DispatcherServlet works internally in Spring MVC



In short, DispatcherServlet is used following things in Spring MVC
- receives all request as Front Controller and provides a single entry point to the application
- mapping requests to correct Spring MVC controller
- Consulting ViewResolvers to find correct View
- forwarding request to chosen View for rendering
- returning the response to the client
- creates web-context to initialize the web specific beans e.g. controllers, view resolvers and handler mapping

That's all about **what is the use of DispatcherServlet in Spring framework**. It's is one of the key components of Spring MVC which is used to receive all incoming request and forward them to right controllers for actual processing. It finds the right controllers by using handler mappings e.g. SimpleUrlHandlerMapping or BeanNameUrlHandlerMapping, which check if the bean name is same as view name and the bean implements the View interface.

If you are using annotations then it can also use @Controller and @RequestMapping annotations to find the right controller to process a particular request. Once the request is processed by controller it returns a logical view name to DispatcherServlet.

The DispatcherServlet then consults ViewResolver and LocalResolvers to find the right View to render the output. Once the correct View is chosen, it forwards the request to the View for rendering the response.

----------------- InternalResourceViewResolver------------------------------

The **InternalResourceViewResolver** is an implementation of ViewResolver in Spring MVC framework which resolves logical view name e.g. "hello" to internal physical resources e.g. Servlet and JSP files e.g. jsp files placed under WEB-INF folder. It is a subclass of UrlBasedViewResolver, which uses "prefix" and "suffix" to convert a logical view name returned from Spring controller to map to actual, physical views.

Configuring ViewResolver using XML in Spring

Here is some XML snippet to configure a view resolve in Spring, you can use this if you are working on a Spring project which is using XML based confirmation:

```
  <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">

        <property name="prefix">
        <value>/WEB-INF/views/</value>
        </property>

        <property name="suffix">
        <value>.jsp</value>
        </property>

        </bean>
```

Configuring ViewResolver using Java Configuration

From Spring 3.0 you can also configure view resolver using Java i.e. without XML. You can use following code to configure internal resource view resolver in your spring project:

```
@Bean
  public ViewResolver viewResolver() {
    InternalResourceViewResolver irv = new InternalResourceViewResolver();
    irv.setPrefix("/WEB-INF/");
    irv.setSuffix(".jsp");

    return irv;

  }
```

Important points about InteralResourceViewResolver in Spring MVC

1) When chaining ViewResolvers, an InternalResourceViewResolver always needs to be last, as it will attempt to resolve any view name, no matter whether the underlying resource actually exists.

2) The InternalResourceViewResolver is also the default view resolver of DispatcherServlet class, which acts as the front controller in Spring MVC

framework.

3) By default, InternalResourceViewResolver returns InternalResourceView (i.e. Servlets and JSP) but it can be configured to return JstlView by using the viewClass attribute as shown below:

4. The most important benefit of using ViewResolver in Spring MVC is that it **decouples request-handling logic in the controller from the view-rendering of a view**. In short, the controller doesn't know anything about which view technology is used to render the view.

It just returns a logical name which could resolve to a JSP, FreeMarker template, Apache tiles or any other view technology. It also means you can change the view layer without changing controller as long as logical view name is same.

Read more: http://javarevisited.blogspot.com/2017/08/what-does-internalresourceviewresolver-do-in-spring-mvc.html#ixzz4xe5igetu

There are three levels of request mapping can be defined in Spring controlle

1. Handler level mapping

The simplest strategy for using **@RequestMapping annotations is to decorate the handler methods directly**. In this method, you need to declare mapping for each handler method with the @RequestMapping annotation containing a URL pattern. If a handler’s @RequestMapping annotation matches the request URL, DispatcherServlet it dispatches the request to this handler for it to process the request.

Following are the different mapping types supported.

•By path
@RequestMapping(“path”)

•By HTTP method
@RequestMapping(“path”, method=RequestMethod.GET). Other Http methods such as POST, PUT, DELETE, OPTIONS, and TRACE are are also supported.

•By query parameter
@RequestMapping(“path”, method=RequestMethod.GET, params=”param1”)

•By presence of request header
@RequestMapping(“path”, header=”content-type=text/*”)

The request mapping for methods can be defined as follows:

```
@Controller
public class Employee {

        @RequestMapping("/employee/add")
        public ModelAndView add(
                        @RequestParam(value = "firstName") String firstName,
                        @RequestParam(value = "surName") String surName) {
                //....
                //....
                return null;
        }

        @RequestMapping(value={"/employee/remove","/employee/delete"})
        public ModelAndView delete(
                        @RequestParam(value = "uuid") String uuid) {
                //....
                //....
                return null;
        }
}
```

In the above code snippet, the controller `add()` method is declared with `@RequestMapping("/employee/add")`. If the incoming request path to matches is `/employee/add`, then the add() handler method will be invoked to process the request.

Handler mappings match URLs according to their paths relative to the context path (i.e., the path where the web application is deployed) and the servlet path (i.e., the path mapped to DispatcherServlet).

2. Mapping at Controller class level

The **@RequestMapping annotation can also be used to decorate a controller class.** This is helpful to take the control at top level and filter the incoming requests. If the incoming request matches the pattern defined in controller class, **then it search the controller methods mappings.**

The following code snippet describes how to define Spring @RequestMapping at controller class level.

```
@Controller
@RequestMapping("/employee/*")
public class Employee {

        @RequestMapping("add")
        public ModelAndView add(
                        @RequestParam(value = "firstName") String firstName,
                        @RequestParam(value = "surName") String surName) {
                //....
                //....
                return null;
        }

        @RequestMapping(value={"remove","delete"})
        public ModelAndView delete(
                        @RequestParam(value = "uuid") String uuid) {
                //....
                //....

                return null;
        }
}
```

Notice that, we have used the wildcard (*) for the @RequestMapping annotation for broader URL matching.

3. Mapping requests by request type

The @RequestMapping annotation handles all types of incoming HTTP request including GET, POST, PUT, DELETE, PATCH etc. By default, it's assumed all incoming requests to URLs are of the HTTP GET kind. To differentiate the mapping by HTTP request type, we need to explicitly specify the HTTP request method in the @RequestMapping declaration.

The following code snippet describes how to declare Spring mappings by HTTP request methods.

```
@Controller
@RequestMapping("/employee/*")
public class Employee {

          @RequestMapping(value = { "remove", "delete" }, method = {
RequestMethod.POST,  RequestMethod.DELETE })
                    public ModelAndView delete(@RequestParam(value = "uuid") String
uuid) {
                              // ....
                              // ....

                              return null;
                    }
}
```

-----------------------------Annotation---

@Controller-

In spring-context first we need to declare a bean. This is just a stand spring bean definition in dispatcher's context.

Latest way of doing this is, enabling autodetection. For the @Controller annotation spring gives a feature of autodetection. We can add "component-scan" in spring-context and provide the base-package.

```
 <context:component-scan base-
package="com.javapapers.spring.mvc" />
  <mvc:annotation-driven />
```

Then add @Controller annotation to controllers. The dispatcher will start from the base-package and scan for beans that are annotated with @Controller annotation and look for @RequestMapping. @Controller annotation just tells the container that this bean is a designated controller class.

@RequestParam(query string)/employees?dept=IT :

`@RequestParam` annotation is used to bind parameter values of query string to the controller method parameters

@RequestParam is used to specify URL query param name. Following handler method will be mapped with the request **/employees?dept=IT** :

```
public String handleEmployeeRequestByDept (@RequestParam("dept") String
deptName) {

        return "my-page";

    }
```

@RequestParam without 'value' element

`@RequestParam` annotation can be skipped if the target variable name is same as param name.
Following handler will be mapped with **/employees?sate=NC**

```
    @RequestMapping

    public String handleEmployeeRequestByArea (@RequestParam String state) {

        return "my-page";



    }
```

Using multiple @RequestParam annotations

@RequestParam annotations. Following will be mapped with **/employees?dept=IT&state=NC** :

```
 public String handleEmployeeRequestByDept (@RequestParam("dept")
String deptName,@RequestParam("state") String stateCode) {
        return "my-page";
    }
```

Using Map with @RequestParam for multiple params

If the method parameter is Map<String, String> or MultiValueMap<String, String> then the map is populated with all query string names and values. Following will be mapped with **/employees/234/messages?sendBy=mgr&date=20160210**

```
  @RequestMapping("{id}/messages")
public String handleEmployeeMessagesRequest (@PathVariable("id") String
employeeId, @RequestParam Map<String, String> queryMap) {
        return "my-page";
    }
```

@Pathvariable with dot . Tuncate

When we are using domain name as request in the spring mvc then it will truncate or remove suffix .xxx or ./ to overcome these prbolem we can use

before confiugration in xml file output will be

www.google   www.yahoo

```
 <bean
class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerM
apping">
       <property name="useDefaultSuffixPattern" value="false"></property>
       </bean>
```

set property value false then we are able to display full domian name like

google.com

yahoo.com

@Pathvariable with Map

if we are sending multiple path variable in url then we can use map to handle the multipe path varibale

like = "/api/login/raja/deva/123

Note: you have to enbable in dispetcher-servele.xml

```
<mvc:annotation-driven></mvc:annotation-driven>

@RequestMapping(value="/login/{userName}/{sirname}/{age}",
method=RequestMethod.GET)
      public ModelAndView getGreet(@PathVariable Map<String,String>
pathVariable, @PathVariable String age) {

            return new ModelAndView("home","message","Name
"+pathVariable.get("userName")+pathVariable.get("sirname")+" Age
"+pathVariable.get("age"));

      }
```

Specifying URL Mapping for String MVC Handler Method

@RequestMapping() handler handle [] (array) type of pattern , so we can pass array type url patter any one of match of request url then that method will be execute

example

url ---> /api/user/save

url----> /api/user/insert

url----> /api/user/update

```
@RequestMapping(value={"/user/save","/user/insert",”/user/update”}
method=RequestMethod.POST)
public ModelAndView saveUSer(@RequestBody UserBean bean) {
      //save logic of user

      }
```

Http methods

GET--->The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.

HEAD-->Same as GET, but transfers the status line and header section only.

POST-->A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.

PUT--->Replaces all current representations of the target resource with the uploaded content.

DELETE-->Removes all current representations of the target resource given by a URI.

TRACE-->Performs a message loop-back test along the path to the target resource.

PATCH-->the HTTP methods PATCH can be used to update partial resources

OPTIONS-->Describes the communication options for the target resource.

-------------------------Spring handle form data as request---------------------

@ModelAttribute--->We want all the form parameters to bind with prescribing properties of model object. This is applied at a argument level.

 on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, **the argument's fields should be populated from all request parameters that have matching names**

@RequestBody In case of POST or PUT request and we have incoming request data in form JSON and want to capture it on server side to initialize the model object.

@RequestParam---> used for accessing the values of the query parameters

automatically binds the request parameters to the arguments of your handler method. It also provides auto type conversion for some standard type like int, long, float, string, date etc.

example:/api/users?name=jagasan&age=25

@pathVariable-->used for accessing the values from the URI template.
It is used to pass parameter along with the url, sometimes we need to pass parameters along with the url to get the data. Spring MVC provides support for customizing the URL in order to get data. To achieving this purpose @PathVariable annotation is used in Spring mvc framework
example:/api/users/jagasan/dansena/25

-------------------------Spring MVC ModelAndView---------------------

Holder for both Model and View in the web MVC framework. Note that these are entirely distinct. This class merely holds both to make it possible for a controller to return both model and view in a single return value.
Represents a model and view returned by a handler, to be resolved by a DispatcherServlet. The view can take the form of a String view name which will need to be resolved by a ViewResolver object; alternatively a View object can be specified directly. The model is a Map, allowing the use of multiple objects keyed by name.

Downlaod jstl jar for jsp page and add WEB-INF/lib

different approach to access ModelAndView

```
@RequestMapping(value="/get-user",method=RequestMethod.GET)
public String getForm(Map<String,String> map) {
      System.out.println("call the method-------");
      map.put("firstName", "raja");
      map.put("lastName", "kumar");
      map.put("email","raja@thrymr.net");
      return "user";
}

@RequestMapping(value="/get-user-info",method=RequestMethod.GET)
```

```java
	public String getUerInfo(Model model) {

		model.addAttribute("firstName", "Mohan");
		model.addAttribute("lastName", "chinka");
		model.addAttribute("email", "Mohan@thrymr.net");

		return "user";
	}
```

------------Handling HttpRequest and HttpResponse by handler-----------------

```java
@RequestMapping(value="user/save-user",method=RequestMethod.POST)
	public ModelAndView saveUserList(HttpServletRequest 
request,HttpServletResponse response) {

		String firstName =request.getParameter("firstName");
		String lastName =request.getParameter("lastName");
		String email =request.getParameter("email");
		String password =request.getParameter("password");
		int id=1245;

		User user = new User(id,firstName,lastName,email,password);


		userDao.addUser(user);

		ModelAndView modal = new ModelAndView("userList");
		modal.addObject("users", userDao.getAllUser());

		return modal;

	}
```

-----------------------------------Page Redirect------------------------------------

```java
	@RequestMapping(value="user/save-user",method=RequestMethod.POST)
	public ModelAndView saveUserList(@ModelAttribute User user) {
		userDao.addUser(user);
		return new ModelAndView("redirect:/user/get-users");

	}
	@RequestMapping(value="/user/get-users",method=RequestMethod.GET)
	public ModelAndView getUserList() {

		System.out.println("call get methods");
		ModelAndView modal = new ModelAndView();
		System.out.println("---size---"+userDao.getAllUser().size());
		return new ModelAndView("userList","users",userDao.getAllUser());

	}
```

------------------------------------OR---

```
public void myController(HttpServletResponse response){
response.sendRedirect("/myURL");
}
```

--------------------Spring interceptor ------------------------

Spring Interceptor are used to intercept client requests and process them. Sometimes we want to intercept the HTTP Request and do some processing before handing it over to the controller handler methods. That’s where Spring MVC Interceptor come handy.

boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler):

Called before the handler execution, returns a boolean value, “true” : continue the handler execution chain; “false”, stop the execution chain and return it.

Object *handler* is the chosen handler object to handle the request. This method can throw Exception also, in that case Spring MVC Exception Handling should be useful to send error page as response.

void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView):

This HandlerInterceptor interceptor method is called when HandlerAdapter has invoked the handler but DispatcherServlet is yet to render the view. This method can be used to add additional attribute to the ModelAndView object to be used in the view pages. We can use this spring interceptor method to determine the time taken by handler method to process the client request.

void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex):

This is a HandlerInterceptor callback method that is called once the handler is executed and view is rendered.

If there are multiple spring interceptors configured, *preHandle()* method is executed in the order of configuration
whereas *postHandle()* and *afterCompletion()* methods are invoked in the reverse order.

Example

------------------------in servlet-decpetcher------------------------

```
 <mvc:interceptors>
        <bean class="com.infotech.Inteceptator"></bean>
        </mvc:interceptors>
```

-----------------interceptor controller---

```
public class Inteceptator implements
org.springframework.web.servlet.HandlerInterceptor{

      @Override
      public void afterCompletion(HttpServletRequest arg0, HttpServletResponse
arg1, Object arg2, Exception arg3)
                  throws Exception {
            // TODO Auto-generated method stub
            System.out.println("-------call afterCompletion---------------");

      }

      @Override
      public void postHandle(HttpServletRequest arg0, HttpServletResponse arg1,
Object arg2, ModelAndView arg3)
                  throws Exception {
            System.out.println("-------call postHandler---------------");

      }

      @Override
      public boolean preHandle(HttpServletRequest arg0, HttpServletResponse
arg1, Object arg2) throws Exception {
            // TODO Auto-generated method stub
            System.out.println("-------call preHandle---------------");
            return true;
      }
}
```

----------------Spring Global Exception Handing---------------

```
@ControllerAdvice

public class GlobalExceptionHandler {


      @ExceptionHandler(value=NullPointerException.class)
      public ModelAndView handleNullPointerException(HttpServletRequest request,
Exception e) {


            return new
ModelAndView("golobalException","exception",e.getMessage());
      }

      @ExceptionHandler(value=ArrayIndexOutOfBoundsException.class)
      public ModelAndView handleArrayIndexOutOfBoundException(HttpServletRequest
request, Exception e) {
```

```
            return new
ModelAndView("golobalException","exception",e.getMessage());
      }

      @ExceptionHandler(value=IllegalArgumentException.class)
      public ModelAndView handleIllegalArgumentException(HttpServletRequest
request, Exception e) {


            return new
ModelAndView("golobalException","exception",e.getMessage());
      }

      @ExceptionHandler(value=RuntimeException.class)
      public ModelAndView handleRuntimeException(HttpServletRequest request,
Exception e) {

            System.out.println("global exception handler is not excecuting");

            return new
ModelAndView("golobalException","exception",e.getMessage());
      }


      @ExceptionHandler(value=Exception.class)
      public ModelAndView handleException(HttpServletRequest request, Exception
e) {

            return new
ModelAndView("golobalException","exception",e.getMessage());
      }

}
```

```
      @RequestMapping(value="user/save-user",method=RequestMethod.POST)
      public ModelAndView saveUserList( @ModelAttribute User user)throws
Exception {


            if(user.firstName.isEmpty()) {

                  Assert.isNull(user.firstName, "first name can not be null");
            }
            if(user.lastName.isEmpty()) {

                  Assert.isNull(user.lastName, "Last name can not be null");

            }
            userDao.addUser(user);
            return new ModelAndView("redirect:/user/get-users");

      }
```