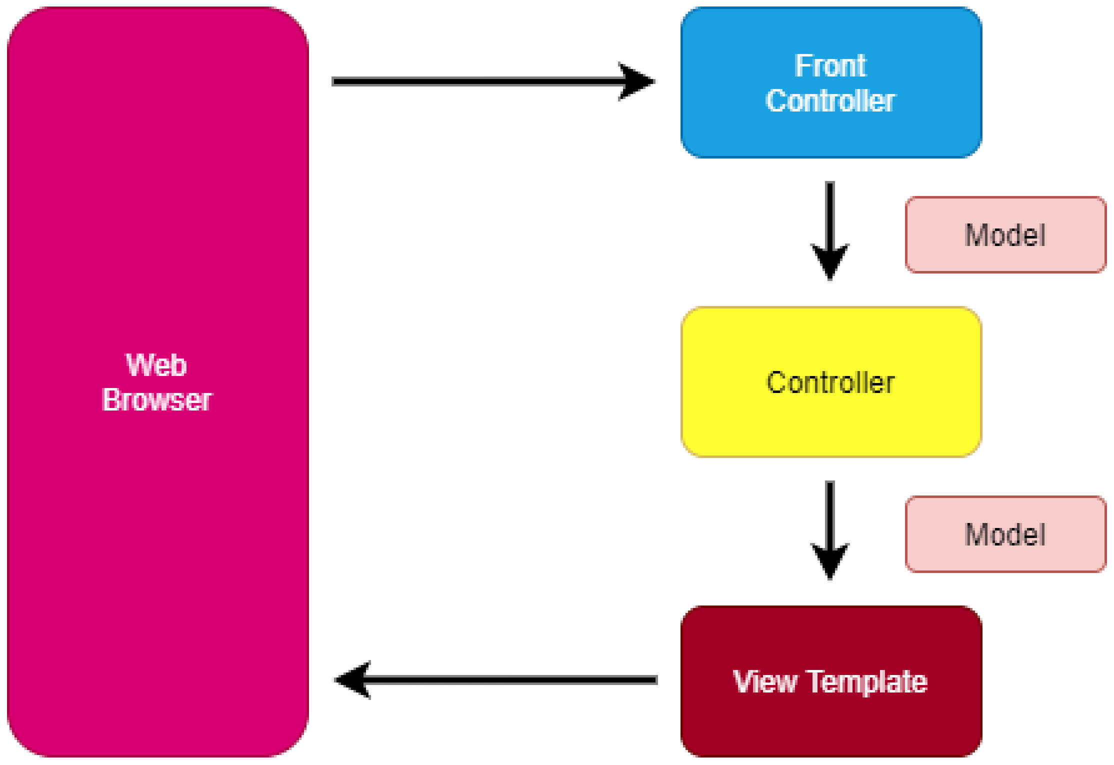
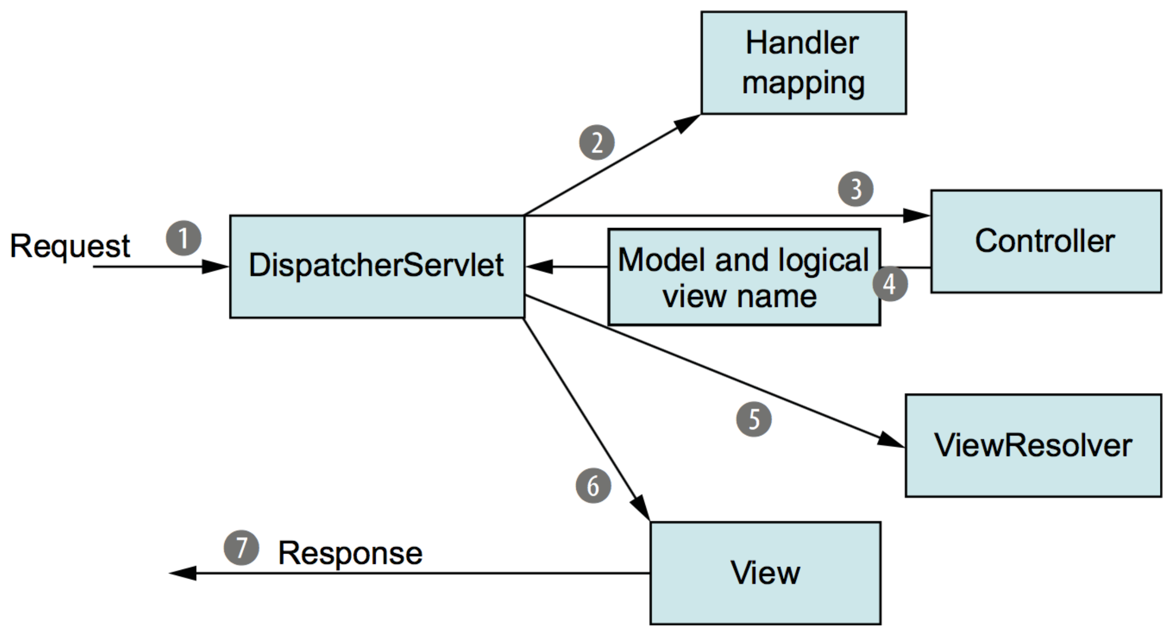
What is Spring MVC?

- Spring MVC is a framework for building web applications in Java
- It is based on the Model-View-Control design pattern
- It leverages features of the core Spring framework such as Inversion of control and dependency injection

Model-View-Controller

Below is the architectural flow diagram of Spring MVC





- Browser sends a request to the server. Server takes that request, process it and send it to the Front controller
- Front controller is a Dispatcher Servlet, its the job is the send the request to the appropriate controller
- Controller code is the one written by developers, which contains the business logic

Controller

- Controller classes are created by developers
- Contains business logic and it handles the request
- It store and retrieve the data from database and other services
- It will write the data to the model
- It will send the data to the appropriate view template

Model

- Model is just a java object contains the data
- It will help us the store and retrieve the data from database or web service

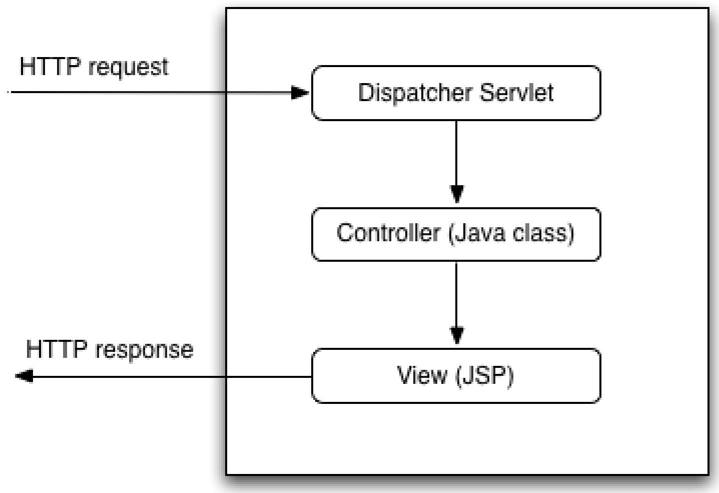
View Template

- Spring MVC is flexible, it supports many view templates
- Most commonly used view templates are JSP and Thymeleaf

Spring MVC Benifits

- It is the best way of building web applications in Java
- It leverages a set of reusable UI components
- It will help us to manage application state for web requests
- It process the form data by validating, converting etc,
- It is very flexible to configure the view layer with other view templates.

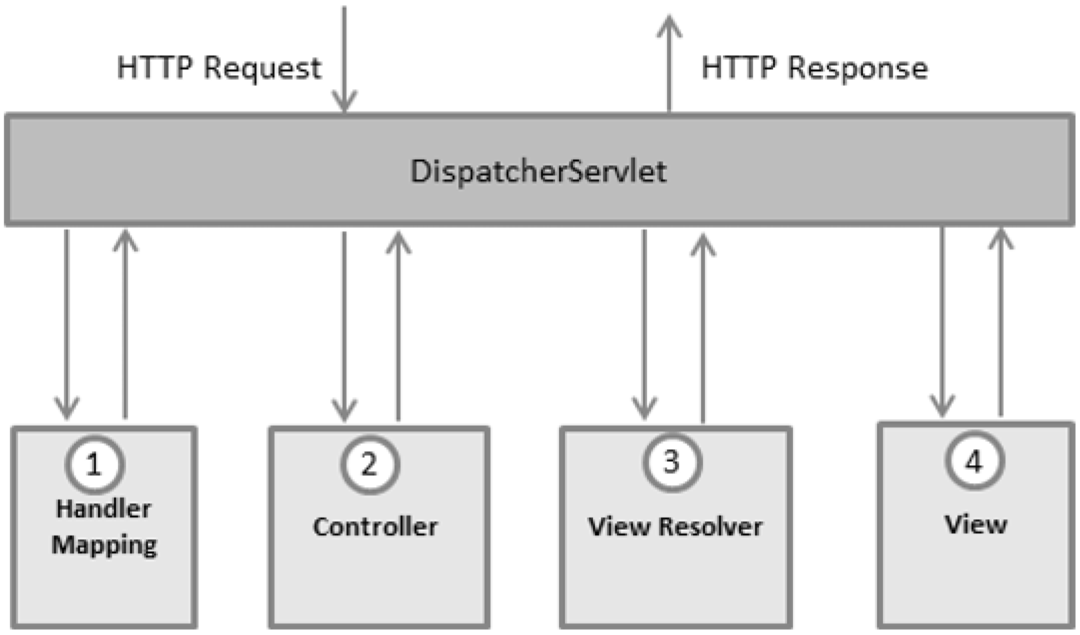
DispatcherServlet

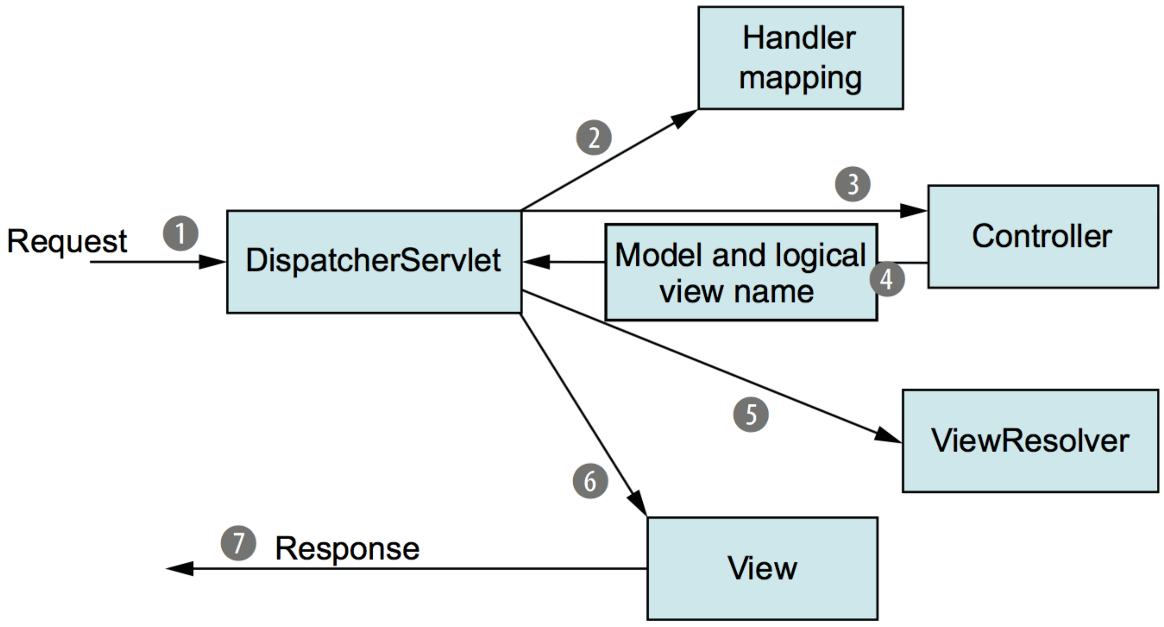


- We can say like **DispatcherServlet** taking care of everything in Spring MVC.
- At web container start up:
- **DispatcherServlet** will be loaded and initialized by calling **init()** method

- **init()** of **DispatcherServlet** will try to identify the Spring Configuration Document with naming conventions like "servlet_name-servlet.xml" then all beans can be identified.
- Example:
- public class DispatcherServlet extends HttpServlet {
-
-     ApplicationContext ctx = null;
-
-     public void init(ServletConfig cfg){
-         // 1. try to get the spring configuration document with default naming conventions
-         String xml = "servlet_name" + "-servlet.xml";
-
-         //if it was found then creates the ApplicationContext object
-         ctx = new XmlWebApplicationContext(xml);
-     }
-     ...
- }

- So, in generally **DispatcherServle**t capture request URI and hand over to **HandlerMapping**. **HandlerMapping** search mapping bean with method of controller, where controller returning **logical name(view).**
- Then this logical name is send to **DispatcherServlet** by **HandlerMapping**.
- Then **DispatcherServlet** tell **ViewResolver** to give full location of view by appending prefix and suffix, then **DispatcherServlet** give view to the client.





@ContextConfiguration

@ContextConfiguration defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests.

@ContextConfiguration

@ContextConfiguration loads an **ApplicationContext** for Spring integration test. **@ContextConfiguration** can load ApplicationContext using XML resource or the JavaConfig annotated with **@Configuration**. The **@ContextConfiguration** annotation can also load a component annotated with **@Component, @Service, @Repository** etc. We can also load classes annotated with **javax.inject**.

@ContextConfiguration annotation has following elements.

classes: The classes annotated with @Configuration are assigned to load ApplicationContext.

inheritInitializers: A Boolean value to decide whether context initializers from test super classes should be inherited or not. Default is true.

inheritLocations: A Boolean value to decide whether resource locations or annotated classes from test super classes should be inherited or not. Default value is true.

initializers: We specify application context initializer classes that initialize ConfigurableApplicationContext.

loader: We specify our ContextLoader or SmartContextLoader class to load ApplicationContext.

locations: We specify resource locations to load ApplicationContext.

name: Name of context hierarchy level represented by this configuration.

value: It is the alias for locations element.

Why We Care about Spring Annotations

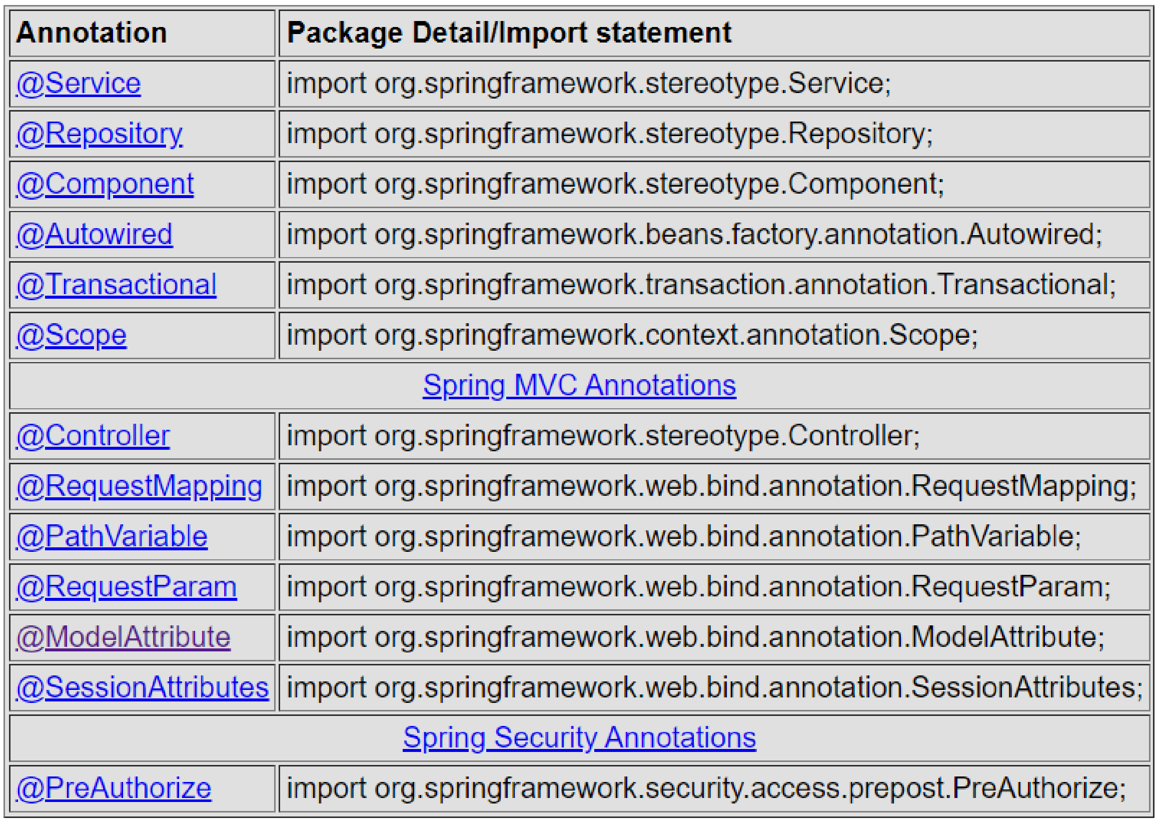
Spring has some powerful concepts like dependency injection (DI) which is made possible thanks to Spring-managed components.

Here, all Classes annotated with **@Component** are candidates for DI and can be used by other Classes without explicit instantiation.

Spring offers a feature called **Inversion of Control** Container which will do all necessary injections in the background, all magic happens without any work from the developer

This is a great example for learning Spring by studying annotations: The Java code of a given Class may look completely normal, but when we consider this metadata, much richer functionality is possible (like DI).

There are many of those annotations that significantly add functionality unique to Spring.



Annotation	**Package Detail/Import statement**
@Service	import org.springframework.stereotype.Service;
@Repository	import org.springframework.stereotype.Repository;
@Component	import org.springframework.stereotype.Component;
@Autowired	import org.springframework.beans.factory.annotation.Autowired;
@Transactional	import org.springframework.transaction.annotation.Transactional;
@Scope	import org.springframework.context.annotation.Scope;
Spring MVC Annotations	
@Controller	import org.springframework.stereotype.Controller;
@RequestMapping	import org.springframework.web.bind.annotation.RequestMapping;
@PathVariable	import org.springframework.web.bind.annotation.PathVariable;
@RequestParam	import org.springframework.web.bind.annotation.RequestParam;
@ModelAttribute	import org.springframework.web.bind.annotation.ModelAttribute;
@SessionAttributes	import org.springframework.web.bind.annotation.SessionAttributes;
Spring Security Annotations	
@PreAuthorize	import org.springframework.security.access.prepost.PreAuthorize;

@Component

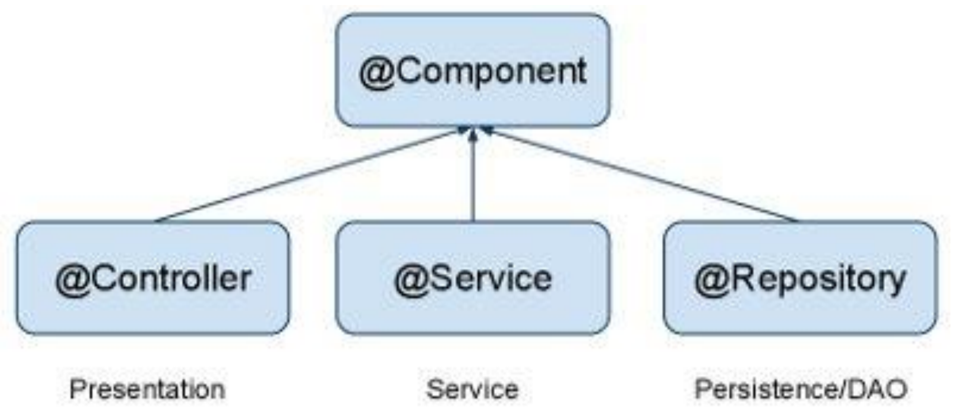
Annotate your other components (for example REST resource classes) with @Component.

```
@Component
```

```
public class ContactResource {
...
}
```

@Component is a generic stereotype for any Spring-managed component.

@Repository, **@Service**, and **@Controller** are specializations of **@Component** for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.



- **@Component:** This Class is registered in the Spring-ApplicationContext and becomes a candidate for dependency injection.
- **@Service:** has the same meaning as @Component, but is used for the **service layer** of an app (with business logic).
- **@Repository:** same as @Component, but for Classes performing **database access**. The standard database exceptions will automatically be caught by Spring and dealt with appropriately.

Note that all of these annotations are to be applied at class-level.

@Autowired

Let Spring auto-wire other beans into your classes using @Autowired annotation.

```
@Service
```

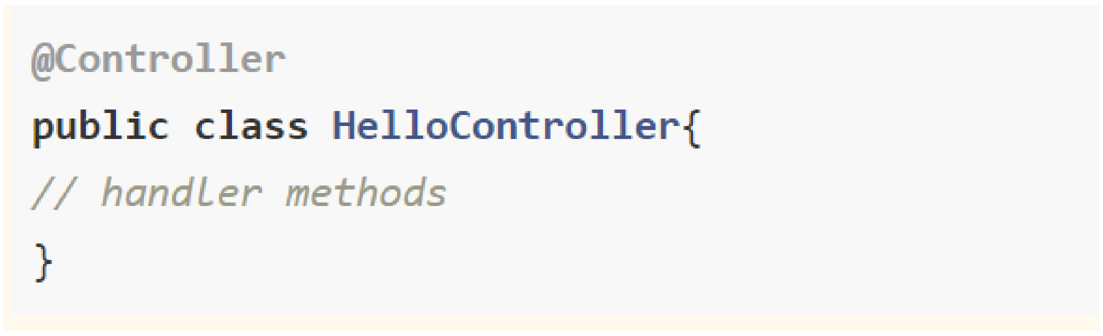
```
public class CompanyServiceImpl implements
CompanyService {

   @Autowired
   private CompanyDAO companyDAO;

   ...
}
```

Commonly used annotations in Spring MVC

1. @Controller

This annotation is used to make a class as a web controller, which can handle client requests and send a response back to the client. This is a class-level annotation, which is put on top of your controller class. Similar to @Service and @Repository it is also a stereotype annotation.

Here is an example of @Controller annotation in Spring MVC:



```
@Controller
public class HelloController{
// handler methods
}
```

This is a simple controller class that contains handler methods to handle HTTP requests for different URLs. You don't need to extend a class or implement an interface to create the controller anymore.

2. @RequestMapping

The Controller class contains several handler methods to handle different HTTP requests but how does Spring map a particular request to a particular handler method? Well, that's done with the help of the **@RequestMapping** annotation. It's a method-level annotation that is specified over a handler method.

It provides the mapping between the request path and the handler method. It also supports some advanced options that can be used to specify separate handler methods for different types of requests on the same URI like you can specify a method to handle GET requests and another to handle POST requests on the same URI.

Here is an example of **@RequestMapping** annotation in Spring MVC:

```
@Controller
public class HelloControler{

  @RequestMapping("/")
  public String hello(){
    return "Hello Spring MVC";
  }
}
```

In this example, the home page will map to this handler method. So any request that comes to the **localhost:8080** will go to this method which will return "Hello Spring MVC".

The value attribute of **@RequestMapping** annotation is used to specify the URL pattern but if there are no more arguments then you also omit that.

You can also specify the HTTP method using the RequestMethod attribute

3. @RequestParam

This is another useful Spring MVC annotation that is used to bind HTTP parameters into method arguments of handler methods. For example, if you send query parameters along with URLlikie for paging or just to supply some key data then you can get them as method arguments in your handler methods.

Here is an example of @RequestParam annotation in Spring MVC from my earlier article about the difference between RequestParam and PathVariable annotation:

```
@RequestMapping("/book")
public String showBookDetails(

@RequestParam("ISBN") String ISBN, Model model){
  model.addAttribute("ISBN", ISBN);
  return "bookDetails";
}
```

If you access your web application which provides book details with a query parameter like below:

http://localhost:8080/book?ISBN=900848893

then the above handler method will be called because it is bound to the "/book" URL and the query parameter ISBN will be used to populate the method argument with the same name "ISBN" inside showBookDetails() method.

You can even use a different name of your choice.

4. @PathVariable

This is another annotation that is used to retrieve data from the URL. Unlike **@RequestParam** annotation which is used to extract query parameters, this annotation enables the controller to handle a request for parameterized URLs like URLs that have variable input as part of their path like:

http://localhost:8080/books/900083838

If you want to retrieve the ISBN number "900083838" from the URL as a method argument then you can use **@PathVariable** annotation in Spring MVC as shown below:

```
@RequestMapping(value="/books/{ISBN}",
                        method= RequestMethod.GET)
public String showBookDetails(@PathVariable("ISBN") String id,
Model model){
   model.addAttribute("ISBN", id);
   return "bookDetails";
}
```

The Path variable is represented inside curly braces like {ISBN}, which means the part after /books is extracted and populated on method argument id, which is annotated with **@PathVaraible**.

In short, this annotation binds placeholder from the URI to a method parameter inside the handler method.

5. @RequestBody

This annotation can convert inbound HTTP data into Java objects passed into the controller's handler method. Just as @ResponseBody tells the Spring MVC to use a message converter when sending a response to the client, the @RequestBody annotations tell the Spring to find a suitable message converter to convert a resource representation coming from a client into an object.

Here is an example:

```
@RequestMapping(method=RequestMethod.POST, consumers= "application/json")
public @ResponseBody Course saveCourse(@RequestBody Course aCourse){
   return courseRepository.save(aCourse);
}
```

This is again a very useful annotation while developing **RESTful** web service in Java using the Spring framework

6. @ResponseBody

The **@ResponseBody** annotation is one of the most useful annotations for developing RESTful web service using Spring MVC. This annotation is used to transform a Java object returned from he a controller to a resource representation requested by a REST client. It can completely bypass the view resolution part.

Here is an example of @ResponseBody annotation in Spring MVC:

```
@RequestMapping(method=RequestMethod.POST,consumers= "application/json")
public @ResponseBody Course saveCourse(@RequestBody Course aCourse){
  return courseRepository.save(aCourse);
}
```

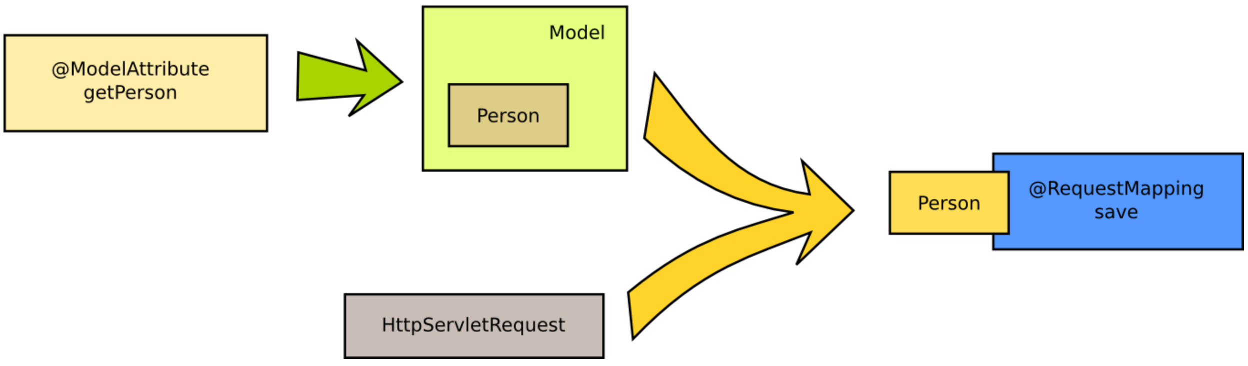
This is the same as the previous example but the **@ResponseBody** is used to indicate that the response returned by this method will be converted into a resource that the client can consume

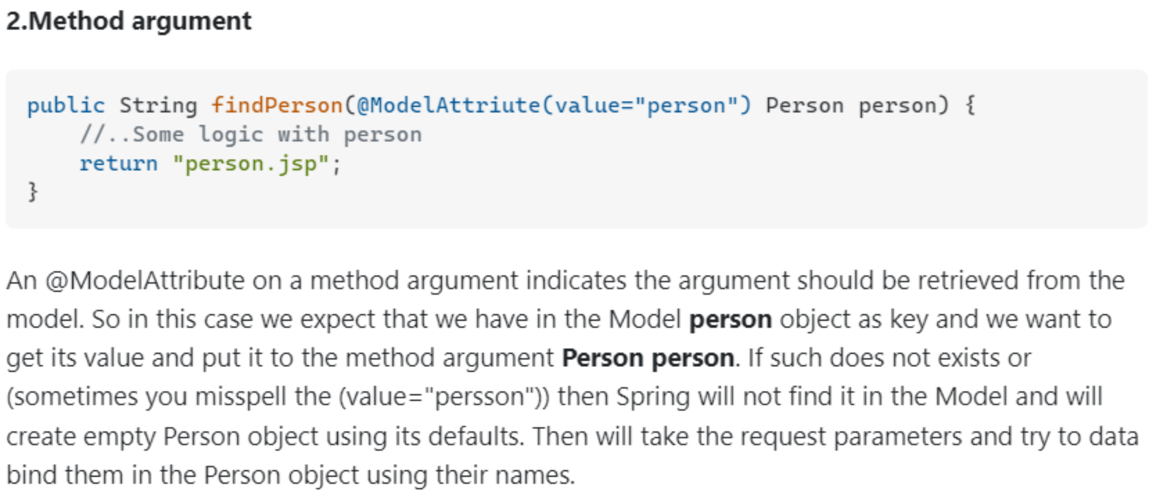
7. @ModelAttribute

An **@ModelAttribute** 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. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

What happened

Spring is going to call your **@ModelAttribute** method and merge the model and all objects that are in it with whatever comes from the request. The following is an image that shows that:





2.Method argument

```
public String findPerson(@ModelAttriute(value="person") Person person) {
    //..Some logic with person
    return "person.jsp";
}
```

An @ModelAttribute on a method argument indicates the argument should be retrieved from the model. So in this case we expect that we have in the Model **person** object as key and we want to get its value and put it to the method argument **Person person**. If such does not exists or (sometimes you misspell the (value="persson")) then Spring will not find it in the Model and will create empty Person object using its defaults. Then will take the request parameters and try to data bind them in the Person object using their names.

On the other hand the annotation is used to define objects which should be part of a Model. So if you want to have a Person object referenced in the Model you can use the following method:

```
public class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(final String name) {
        this.name = name;
    }
}
```

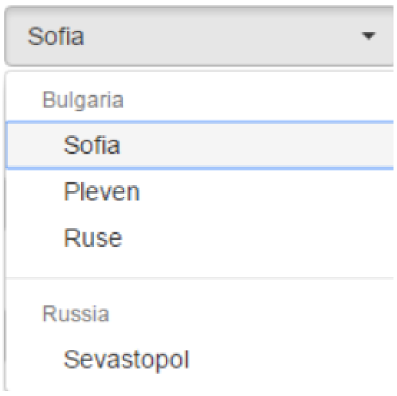
As described in the Spring MVC documentation - the **@ModelAttribute** annotation can be used on **methods** or on **method arguments**. And of course we can have both use at the same time in one controller.

1.Method annotation

```
@ModelAttribute(“cities”)
 public List<String> checkOptions(){
 return new Arrays.asList(new[]{“Sofia”,”Pleven","Ruse”});//and so on
}
```

Purpose of such method is to add attribute in the model. So in our case **cities** key will have the list `new Arras.asList(new[]{“Sofia”,”Pleven","Ruse”})` as value in the Model (you can think of Model as map(key:value)). **@ModelAttribute** methods in a controller are invoked before **@RequestMapping** methods, within the same controller.

Here we want to add to the Model common information which will be used in the form to display to the user. For example it can be used to fill a HTML select:



8. @SessionAttributes

@SessionAttributes spring annotation declares session attributes. This will typically list the names of model attributes which should be transparently stored in the session, serving as form-backing beans between subsequent requests.

```
@Controller
@RequestMapping("/company")
@SessionAttributes("company")
public class CompanyController {

  @Autowired
  private CompanyService companyService;
...
}
```

modelAttribute

```
<form:form action="processForm" modelAttribute="student">
  First Name : <form:input path="firstName" />
  <br/><br/>
  Last Name : <form:input path="lastName" />
  <br/><br/>
  <input type="submit" value="submit"/>
</form:form>
```

`<form:input path="firstName" />` `<form:input path="lastName" />` These are the fields/properties in the Student class. When the form is called/initialized their getters are invoked. On form submit their setters are invoked, and their values are transferred in the bean that was indicated with `modelAttribute="student"` in the form tag.

We have `StudentController` that includes the following methods:

```
@RequestMapping("/showForm")
// `Model` is used to pass data between controllers and views
public String showForm(Model theModel) {
    // attribute name, value
    theModel.addAttribute("student", new Student());
    return "form";
}

@RequestMapping("/processForm")
public String processForm(@ModelAttribute("student") Student theStudent) {
    System.out.println("theStudent :"+ theStudent.getLastName());
    return "form-details";
}

//@ModelAttribute("student") Student theStudent
//Spring automatically populates the object data with form data
//all behind the scenes
```

Now finally we have a `form-details.jsp`:

```
<b>Student Information</b>
${student.firstName}
${student.lastName}
```

@SessionAttribute works as follows:

- **@SessionAttribute** is initialized when you put the corresponding attribute into model (either explicitly or using @ModelAttribute-annotated method).
- **@SessionAttribute** is updated by the data from HTTP parameters when controller method with the corresponding model attribute in its signature is invoked.
- **@SessionAttributes** are cleared when you call **setComplete()** on SessionStatus object passed into controller method as an argument.

The following listing illustrate these concepts. It is also an example for pre-populating Model objects.

```
@Controller
@RequestMapping("/owners/{ownerId}/pets/{petId}/edit")
@SessionAttributes("pet")
public class EditPetForm {

	@ModelAttribute("types")

	public Collection<PetType> populatePetTypes() {
		return this.clinic.getPetTypes();
	}

	@RequestMapping(method = RequestMethod.POST)
	public String processSubmit(@ModelAttribute("pet") Pet
pet,
			BindingResult result, SessionStatus status)
{
		new PetValidator().validate(pet, result);
		if (result.hasErrors()) {
			return "petForm";
		}else {
			this.clinic.storePet(pet);
			status.setComplete();
			return "redirect:owner.do?ownerId="
				+ pet.getOwner().getId();
		}
	}
}
```

Spring MVC Form Tag Library

The Spring MVC form tags are the configurable and reusable building blocks for a web page. These tags provide JSP, an easy way to develop, read and maintain.

The Spring MVC form tags can be seen as data binding-aware tags that can automatically set data to Java object/bean and also retrieve from it. Here, each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and easy to use.

Configuration of Spring MVC Form Tag

The form tag library comes under the spring-webmvc.jar. To enable the support for form tag library, it is required to reference some configuration. So, add the following directive at the beginning of the JSP page.

```
1. <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

List of Spring MVC Form Tags

Form Tag	Description
form:form	It is a container tag that contains all other form tags.
form:input	This tag is used to generate the text field.
form:radiobutton	This tag is used to generate the radio buttons.
form:checkbox	This tag is used to generate the checkboxes.
form:password	This tag is used to generate the password input field.
form:select	This tag is used to generate the drop-down list.
form:textarea	This tag is used to generate the multi-line text field.
form:hidden	This tag is used to generate the hidden input field.

The form tag

The Spring MVC form tag is a container tag. It is a parent tag that contains all the other tags of the tag library. This tag generates an HTML form tag and exposes a binding path to the inner tags for binding.

Syntax

```
1. <form:form action="nextFormPath" modelAttribute=?abc?>
```