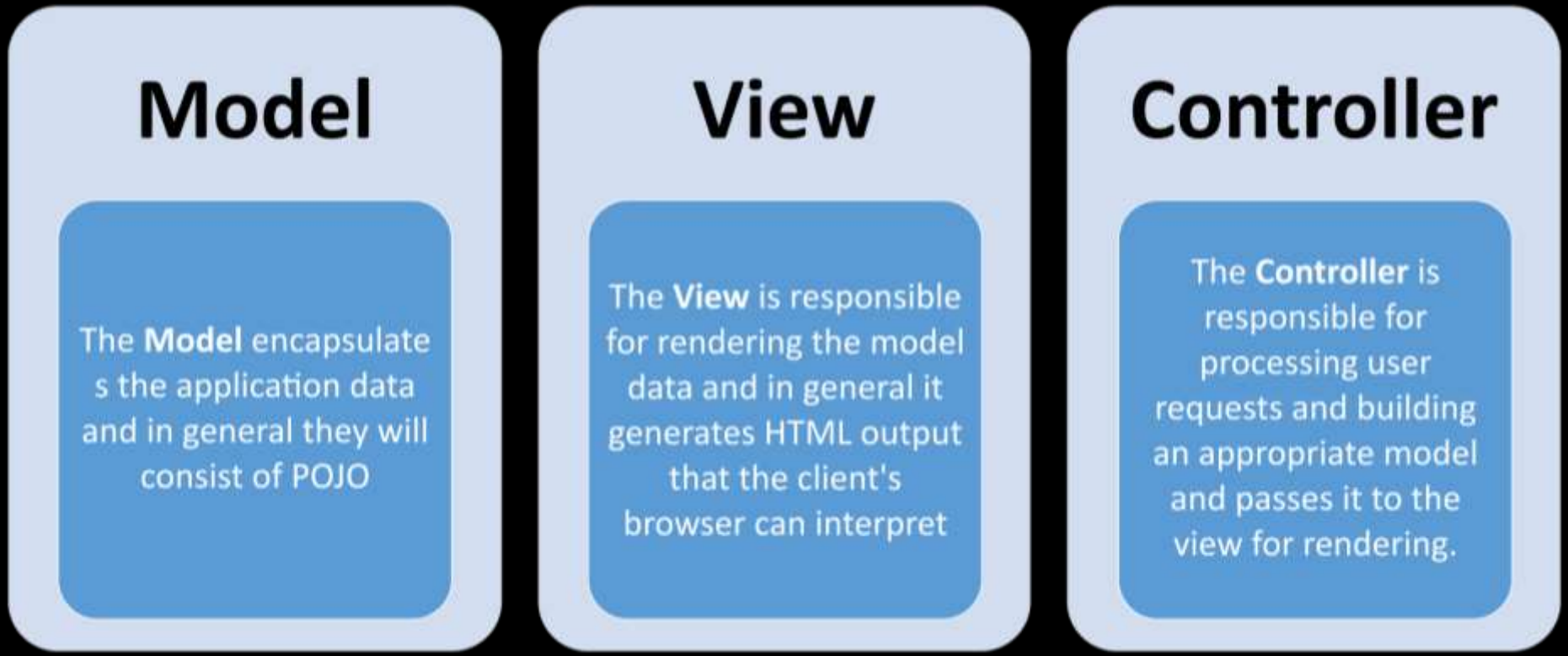
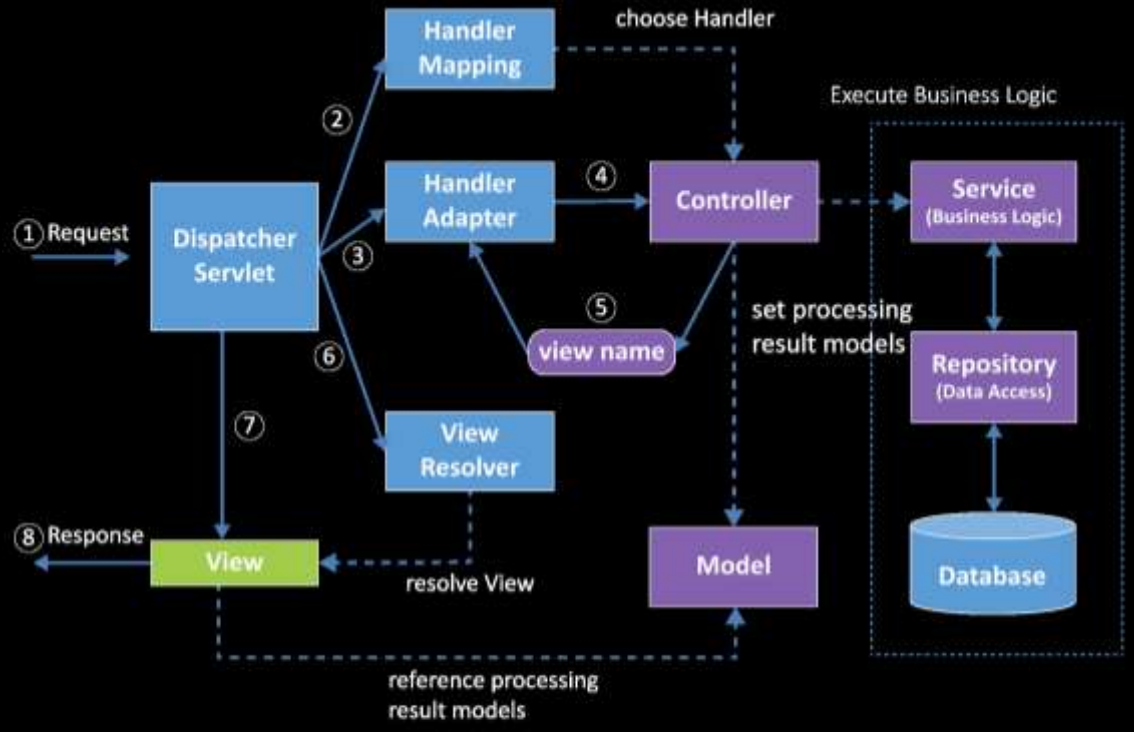
Spring MVC
(notes)

Spring Web MVC



request lifecycle diagram



Let's take a look at the request lifecycle diagram.

First, the ***DispatcherServlet*** (servlet manager) receives a request, then it looks at its settings to understand which controller to use (in the figure, Handler Mapping).

After receiving the name of the controller, the request is sent to it (***Controller*** in the figure). In the controller, the request is processed and the ***ModelAndView*** is sent back (the model is the data itself; the view is how to display this data).

DispatcherServlet, based on the received ***ModelAndView***, looks for which view to use (***View Resolver***) and receives the name of the view Viev in the response
The data (model) is passed to the view and back, if necessary, the response from the view is sent.

Spring MVC defines the ***ViewResolver*** and ***View*** interfaces that let you render models in a browser without tying you to a specific view technology. ViewResolver provides a mapping between view names and actual views. View addresses the preparation of data before handing over to a specific view technology.

@Controller annotation

```
@Controller
@RequestMapping("/hello")
public class HelloController{

   @RequestMapping(method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }

}
```

The *@Controller* annotation defines a class as a Spring MVC controller. Here, the first use of @RequestMapping indicates that all processing methods on this controller refer to the path */hello*.

The following *@RequestMapping (method = RequestMethod.GET)* annotation is used to declare the printHello () method as the default controller service method for handling an HTTP GET request. We can define another method to handle any POST request for the same URL.

@Controller annotation

```
@Controller
public class HelloController{

   @RequestMapping(value = "/hello", method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }

}
```

There are some important points regarding the controller defined above:

1) You will define the necessary business logic inside the service method. You can call another method within this method as per requirement.

2) Based on the defined business logic, you will create a model in this method. You can set various attributes on the model and these attributes will be available to the view to represent the result. This example creates a model with the attribute "message".

3) A specific service method can return a string that contains the name of the **view** that will be used to render the model. This example returns "hello" as the name of the logical view.

Returning Model and View

```
@RequestMapping(value = "/login", method = RequestMethod.GET)
public String viewLogin() {
    return "LoginForm";
}
```

That's the simplest way of returning a view name. But if you want to send additional data to the view, you must return a **ModelAndView** object. Consider the following handler method:

```
@RequestMapping("/listUsers")
public ModelAndView listUsers() {

   List<User> listUser = new ArrayList<>();
   // get user list from DAO...

   ModelAndView modelView = new ModelAndView("UserList");
   modelView.addObject("listUser", listUser);

   return modelView;
}
```

Spring is also very flexible, as you can declare the **ModelAndView** object as a parameter of the handler method instead of creating a new one. Thus, the above example can be re-written as follows:

```
@RequestMapping("/listUsers")
public ModelAndView listUsers(ModelAndView modelView) {

   List<User> listUser = new ArrayList<>();
   // get user list from DAO...

   modelView.setViewName("UserList");
   modelView.addObject("listUser", listUser);

   return modelView;
}
```

Putting Objects Into the Model

```
modelView.addObject("listUser", listUser);
modelView.addObject("siteName", new String("CodeJava.net"));
modelView.addObject("users", 1200000);
```

You can declare a parameter of type **Map** in the handler method; Spring uses this map to store objects for the model. Let's see another example:

```
@RequestMapping(method = RequestMethod.GET)
public String viewStats(Map<String, Object> model) {
   model.put("siteName", "CodeJava.net");
   model.put("pageviews", 320000);

   return "Stats";
}
```

Redirection in Handler Method

In case you want to redirect the user to another URL if a condition is met, just append redirect:/ before the URL. In the above code, the user will be redirected to the /login URL if it is not logged in.

```
// check login status....

if (!isLogin) {
    return new ModelAndView("redirect:/login");
}

// return a list of Users
```

Handling Form Submission and Form Validation

Spring makes it easy to handle form submission by providing the @ModelAttribute annotation for binding form fields to a form backing object, and the BindingResult interface for validating form fields. The following code snippet shows a typical handler method that is responsible for handling and validating form data

```
@Controller
public class RegistrationController {

    @RequestMapping(value = "/doRegister", method = RequestMethod.POST)
    public String doRegister(
        @ModelAttribute("userForm") User user, BindingResult bindingResult) {

        if (bindingResult.hasErrors()) {
            // form validation error

        } else {
            // form input is OK
        }

        // process registration...

        return "Success";
    }
}
```

View Technologies

JSP is one of the most popular view technologies for Java applications, and it is supported by Spring out-of-the-box. For rendering JSP files, a commonly used type of *ViewResolver* bean is *InternalResourceViewResolver*:

Thymeleaf is a Java template engine which can process HTML, XML, text, JavaScript or CSS files. Unlike other template engines, *Thymeleaf* allows using templates as prototypes, meaning they can be viewed as static files.

<#FREEMARKER>

FreeMarker is a Java-based template engine built by the *Apache Software Foundation*. It can be used to generate web pages, but also source code, XML files, configuration files, emails and other text-based formats.
The generation is done based on template files written using the *FreeMarker Template Language*.

View Technologies

Spring MVC views can also be generated using the ***Groovy Markup Template Engine***. This engine is based on a builder syntax and can be used for generating any text format.

Jade4j is the Java implementation of the *Pug* template engine (originally known as *Jade*) for Javascript. *Jade4j* templates can be used for generating HTML files.

View Technologies

JMustache

- is a template engine which can be easily integrated into a Spring Boot application by using the *spring-boot-starter-mustache* dependency.

Pebble

- contains support for Spring and *Spring Boot* within its libraries.

JavaScript templates (on Nashhorn):

- Handlebars
- Mustache
- ReactJS
- EJS