
Code.Hub
The first Hub for Developers

Spring MVC

Code.Hub

- ✓ Web Intro
- ✓ MVC Principles
- ✓ Exception Handling
- ✓ Views and Templating
- ✓ Forms and validation

Glykeria Katsari

Web Applications

What is a web application?

A typical web application consists of:

- A **web server** to manage requests from the client
- An **application server** to perform the tasks
- A **database** to store the information

Code.Hub

Web Applications

When you type a web address into your browser:

The browser goes to the DNS server, and finds the real address of the server that the website lives on

The browser sends an HTTP request message to the server, asking it to send a copy of the website to the client. This message, and all other data sent between the client and the server, is sent across your internet connection using **TCP/IP.**

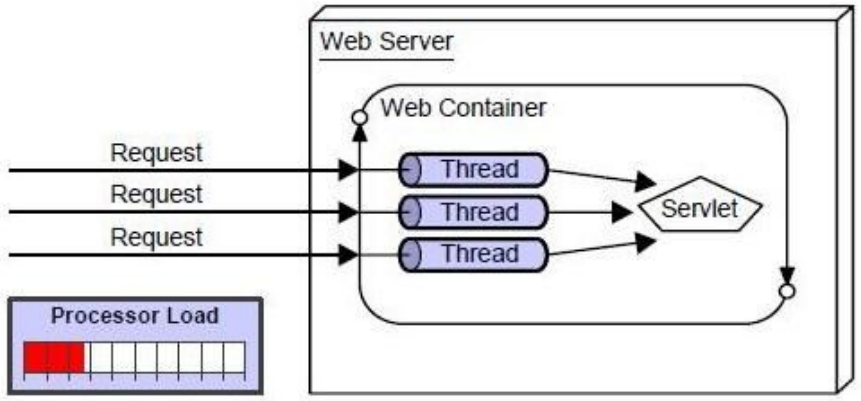
Provided the server approves the client's request, the server sends the client a "200 OK" message, and then starts sending the website's files to the browser as a series of small chunks called data packets.

Servlets

Most enterprise web applications are based on Servlets.

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP clients and applications running on the HTTP server.

Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.



The browser sends an HTTP request message to the server, asking it to send a copy of the website to the client. This message, and all other data sent between the client and the server, is sent across your internet connection using **TCP/IP.**

Provided the server approves the client's request, the server sends the client a "200 OK" message, and then starts sending the website's files to the browser as a series of small chunks called data packets.

Servlets

Servlets perform the following major tasks :

- Read the explicit data sent by the clients (browsers).
  This includes an HTML form on a Web page or a custom HTTP client program.

- Read the implicit HTTP request data sent by the clients (browsers).
  This includes cookies, media types and compression schemes the browser understands.

- Process the data and generate the results.
  This process m**ay require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.**

Servlets

Servlets perform the following major tasks :

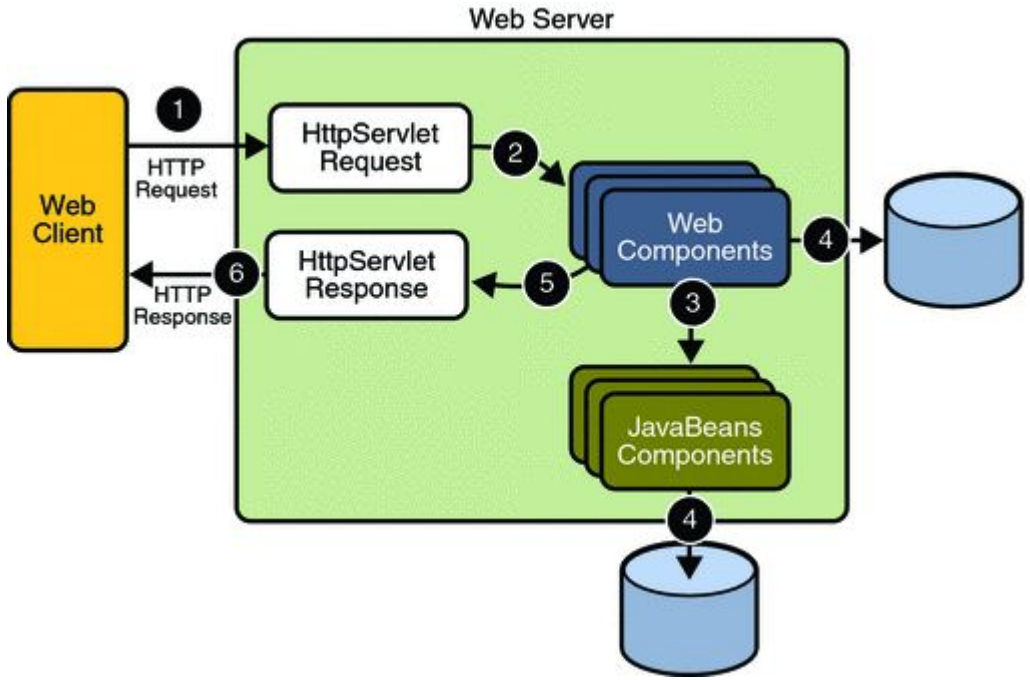
- Send the explicit data to the clients (browsers).
  The response can be sent in a variety of formats, including text (HTML or XML, JSON), binary (images), etc.

- Send the implicit HTTP response to the clients (browsers).
  This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Servlets

Each time a client requests a url on a web browser, the server will have to do the following:

1. Allocate a thread to handle the request
2. Intercept the request url and delegate it to the required Servlet
3. Retrieve the response from the Servlet class and return it to the client.



Code.Hub

MVC Principles

Spring MVC

Spring Web MVC is a web framework built on the Servlet API.

It is the leading framework in web enterprise applications and an extension to the core Spring Framework.

The Spring MVC framework's architecture is based on the the Model View Controller design pattern that it an established and flexible way to create large scale web apps with complex business logic.

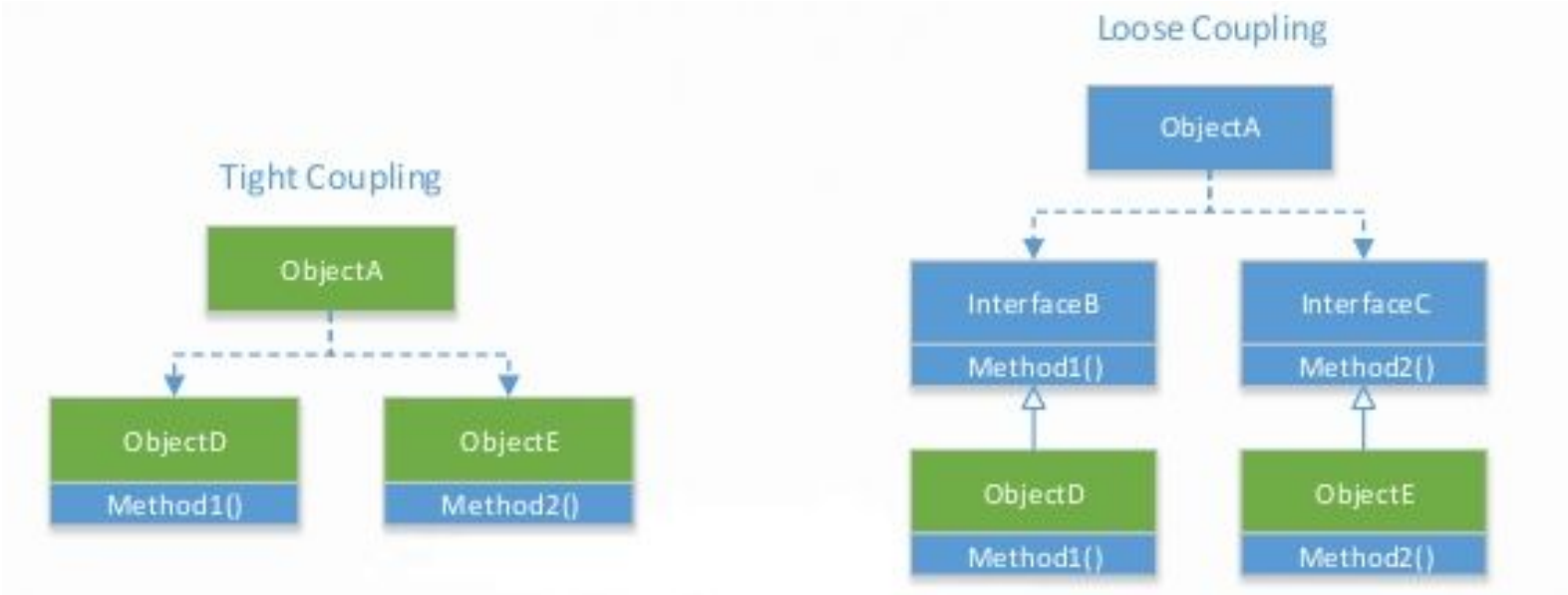
The main benefit of the Framework is that all web application components are **Loosely Coupled:**

- No strict dependencies between them
- Functionality and communication are kept abstract
- Can transparently change components with minimal changes to other layers

Tight Coupling vs Loose Coupling

We have tight coupling when classes are **interdependent** in such a way that a small change in one class will require massive changes to all those depending to it.

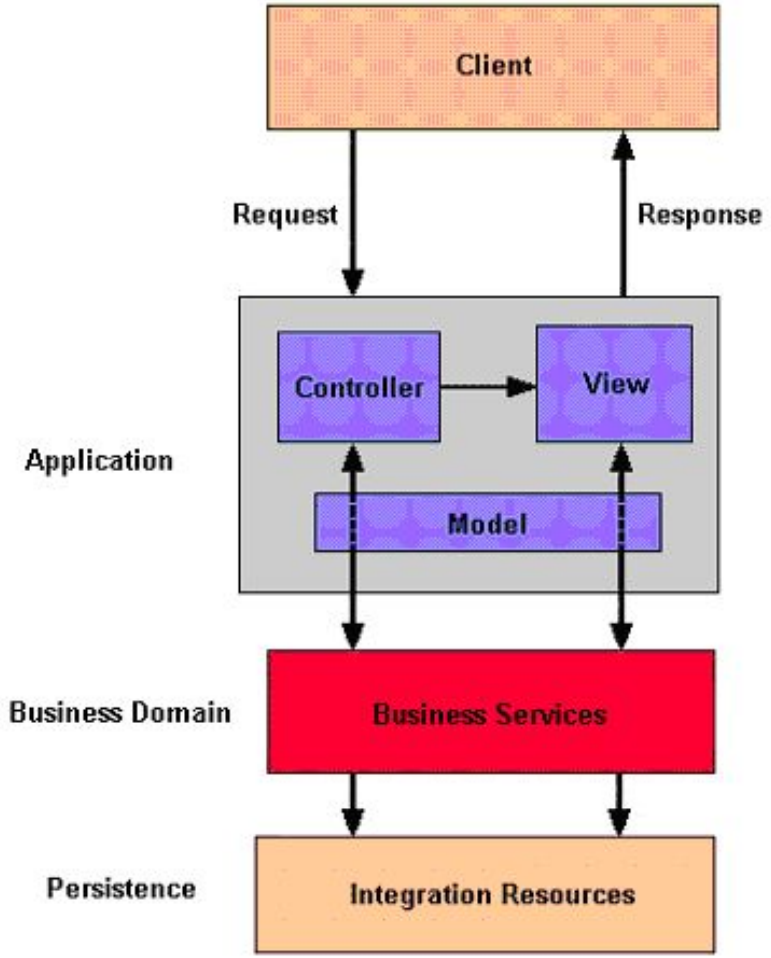
In the example below , Object A depends on ObjectD and ObjectE. If those are re-implemented or removed, we have to update ObjectA. In The Second Example, we are using interfaces so changes in Object E will not affect A.



Why is Loose Coupling important?

- **Testability**
  Component isolation results in easier testing

- Propagates the **Multi Layered** architecture
  The app is comprised of hierarchical layers called **Tiers**
  Changes to each tier MUST NOT affect other tiers.



Why is Loose Coupling important?

- **Reusability**
  Functionality can be re-used across multiple components

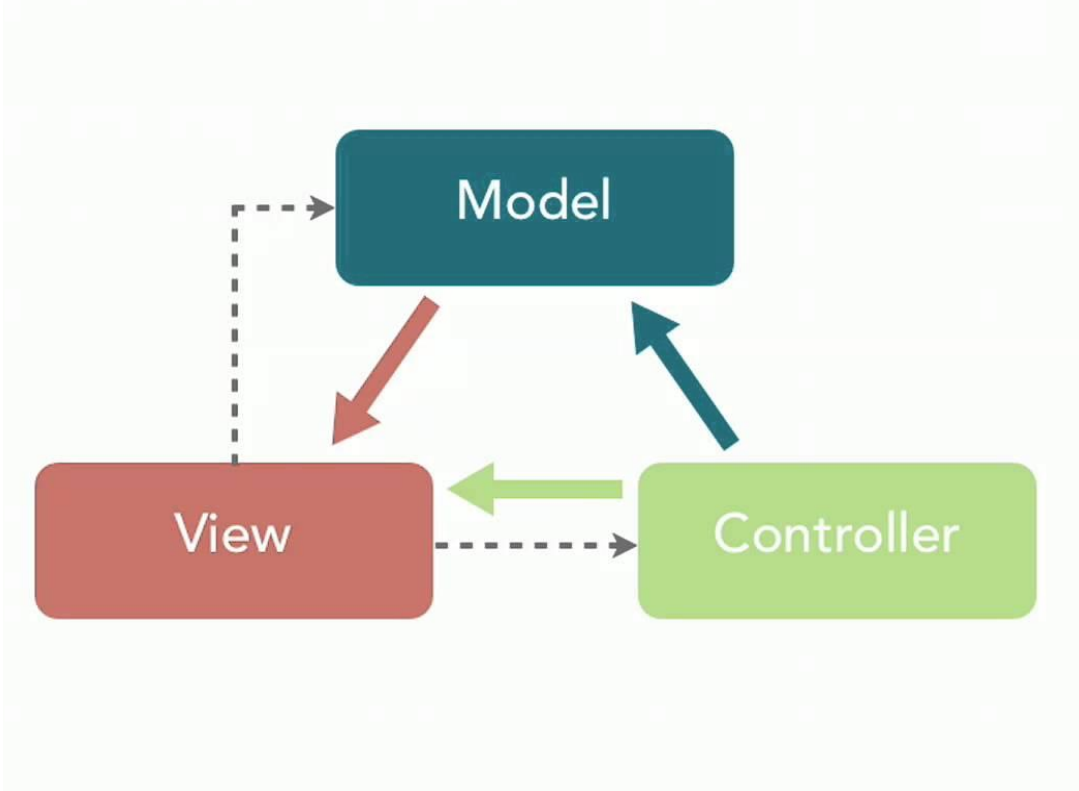
- **Open Closed Principle:**
  We can update a class' functionality without modifying the parent class or changing the original implementation of the function.

- **Facilitating work:**
  We can work in parallel between most of the app components and achieve higher throughput.

The MVC Pattern

Spring Web is built around the **MVC pattern**

The MVC pattern results in separating the different aspects of the application
(input logic, business logic, and UI logic),
while providing a loose coupling between these elements.



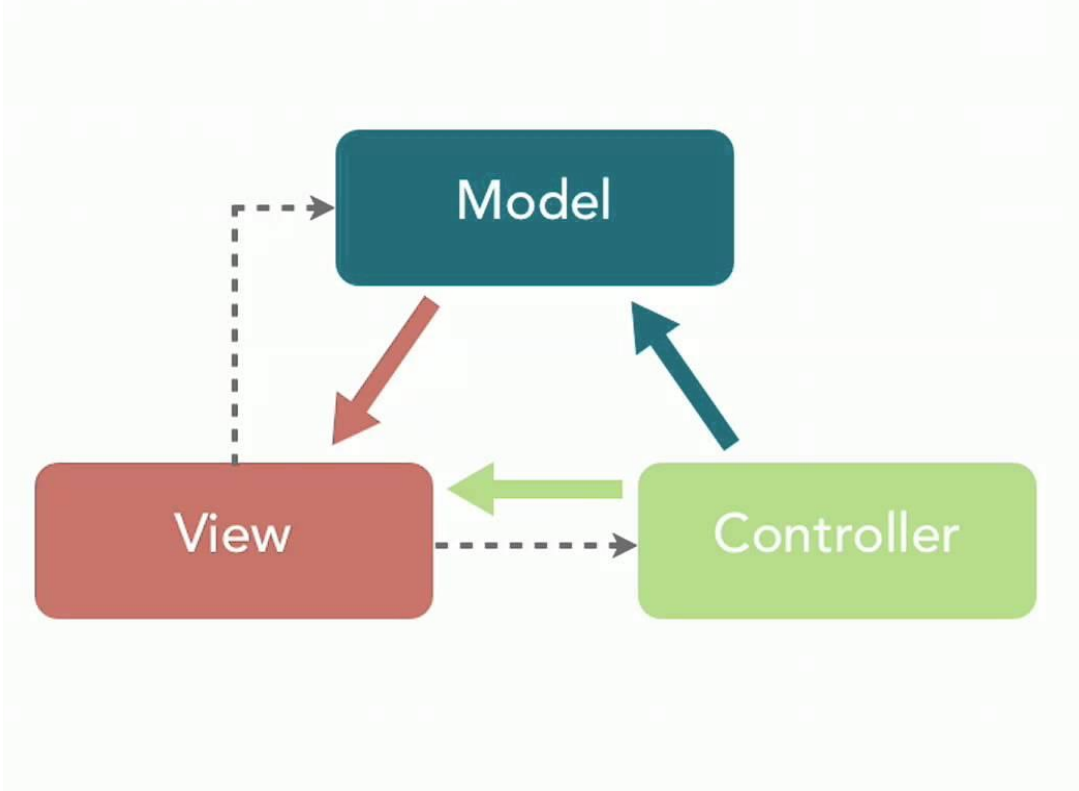
The MVC Pattern

Spring Web is built around the MVC pattern

The **Model** is where the application's data objects are stored.

The model doesn't know anything about views and controllers.

When a model changes, typically it will notify its observers that a change has occurred.

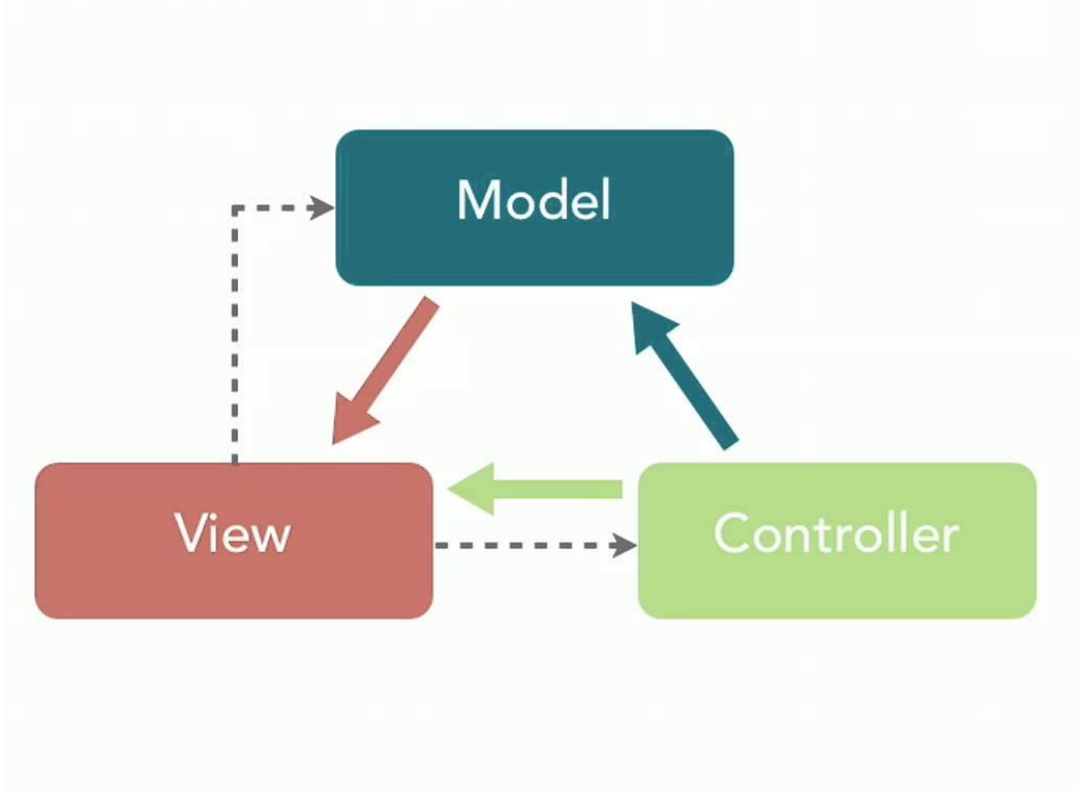


The MVC Pattern

Spring Web is built around the MVC pattern

The **View** is responsible for rendering the model data and generate HTML output that the client's browser can interpret.

Essentially, is what's presented to the users and how users interact with the app. The view is made with HTML, CSS, JavaScript and often templates.

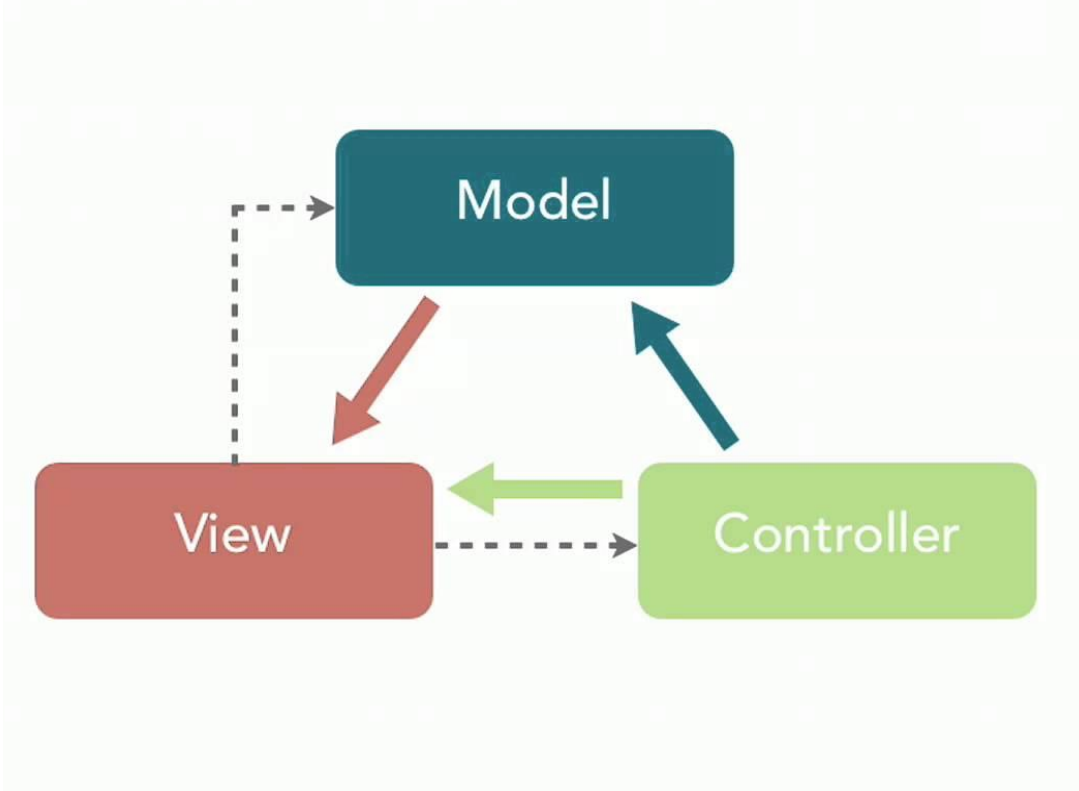


The MVC Pattern

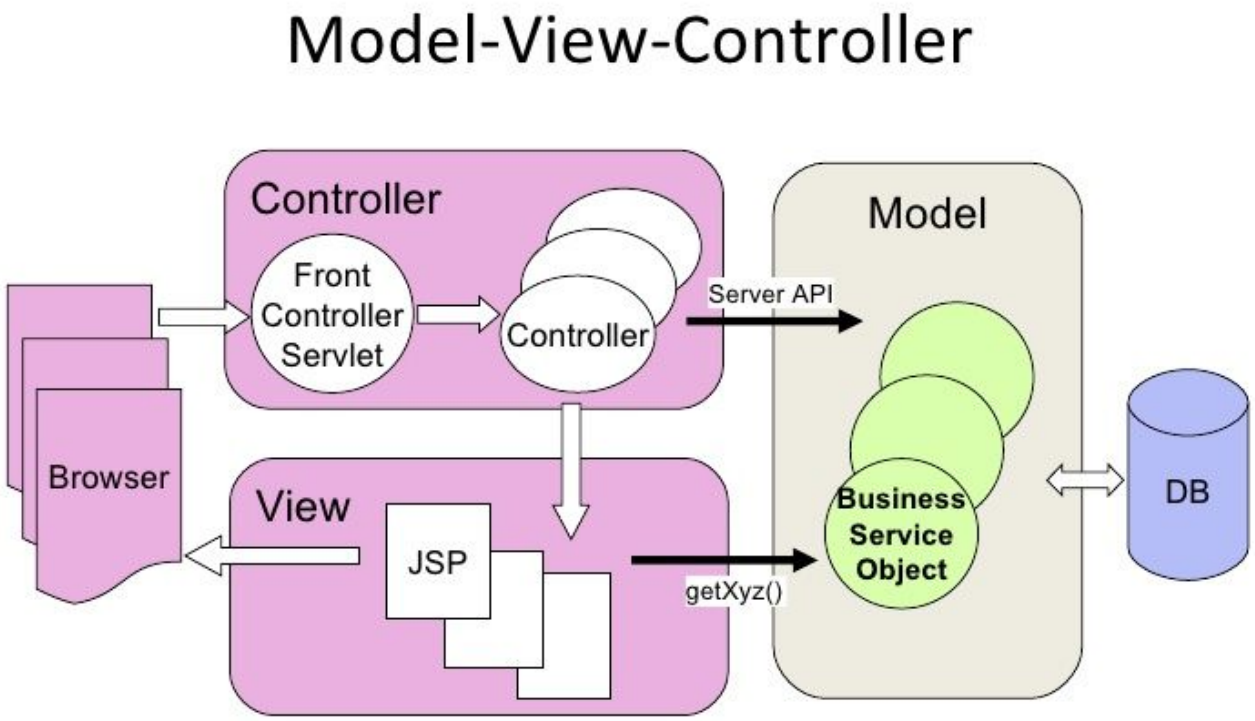
Spring Web is built around the MVC pattern

The **Controller** is responsible for processing user requests and building an appropriate model that will passes it the View for rendering.

It's the decision maker connects the Model with the View : it updates the view when the model changes and updates the model when the user manipulates the view.



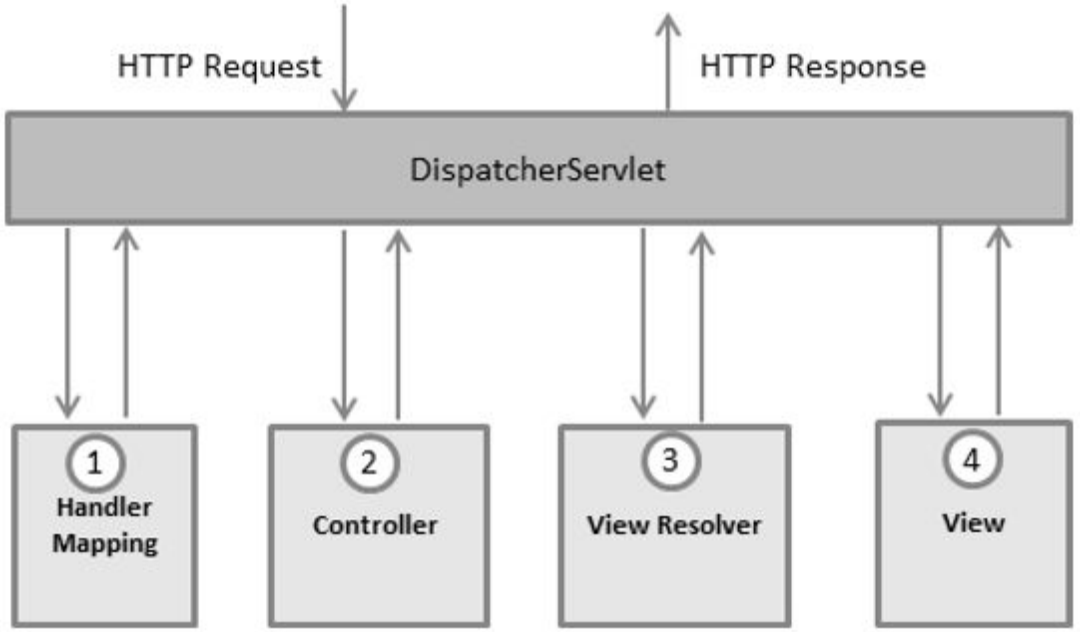
The MVC Pattern



The Dispatcher Servlet

Spring Web (MVC) is designed around a ***DispatcherServlet*** that handles all the HTTP requests and responses.

- After receiving an HTTP request, ***DispatcherServlet*** consults the ***HandlerMapping*** to call the appropriate ***Controller***
- The ***Controller*** takes the request and calls the appropriate service methods based on url name and HTTP method. The service method will set model data based on business logic and returns a view name to the *DispatcherServlet*.
- The ***DispatcherServlet*** will take help from ***ViewResolver*** to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.



The Dispatcher Servlet

The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class)

It classic Spring configuration, it is declared in the web.xml of your web application

You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in an ***web.xml*** file.

In Spring boot, this is configured automatically.

The Handler Mapper

Retains a map of all the declared URLs and their methods and the Controller method that executes them

When a Request with a given URL (/home) is called then the Handler Mapper:

- Scans the map for that url
- Searches for the available HTTP methods for that path and tries to match it with the one in the Request
    - If the request is a POST and the path is only declared as a GET, then Spring will throw a ‘method not supported’ exception.

If all the above perform correctly, then the Handler Mapper will delegate execution to the chosen method.

The mapping between urls and controllers is done when the app is loaded for the first time so it is immutable during execution.

The View Resolver

All MVC frameworks for web applications provide a way to address views.

Spring provides View resolvers, which enable you to render models in a browser without tying you to a specific view technology.

The two interfaces that are important to the way Spring handles views are **ViewResolver** and **View**.

The **ViewResolver** provides a mapping between view names and actual views.

The **View** interface addresses the preparation of the request and hands the request over to one of the view technologies.(More on that later!)

Controllers

Controllers provide access to the application behavior that you typically define through a service interface.

They interpret user input and transform it into a model that is represented to the user by the view.

You can **define** a Controller class by adding the **@Controller** annotation on it.

You also need to specify a **path** for the request that this controller will handle with the **@RequestMapping** annotation

A controller can return a **View** or **redirect to another controller** method or return a page with raw data.

Controllers

```
@Controller
public class HomeController {

    @GetMapping(value = "/")
    public String home() {
        return "home";
    }

}
```

Model

The **model** is an entity that Spring handles and is responsible **for carrying all the data you need to render your view.**

Spring defines the Model interface (and provides some implementations by default).
When you add it to a Controller Method as an parameter, Spring will create a model for you every time that methods is called.

Its most simple implementation is a map , called Model Map. Essentially, its a hash map that contains key-value pairs. The key is the name of the attribute that you will use to access the data in the view level and the value is the actual object.

Model

```
@Controller
public class HomeController {

    @GetMapping(value = "/")
    public String home(ModelMap model) {
        model.addAttribute("message","Hello World!");
        return "index";
    }

}
```

Model

How to choose the correct HTTP Verbs for your methods?

-- If the method is a **GET**, Spring checks if there are already data that should be added to this model, carried from redirects from other pages, or added during the initialization phase of the controller. If not, it returns an empty model.

-- If the method is a **POST**, spring will add to the model the form data that you are submitted and retain the values that are already inside it.

View

The **view** is essentially the **HTML page that we will be populating with data from the model and return to the client.**

Views are usually **HTML files, t**hat are written in a templating language (Freemarker, Thymeleaf, etc) that allows you to evaluate objects, perform comparisons, etc, and render the page data.

The HTML files that produce the view are stored by default in the **/views** folder of the WEB-INF directory in your web application.

When you return the name of a view in a Controller method (return “home”) then the View resolver searches that directory for a file with that name. It then proceeds with enriching it with the data from the Model and returns the rendered HTML to the client.

This is called server-side rendering, because the client receives the rendered HTML and does not have to do any extra data manipulation to enrich it. In contrast, technologies like Angular, do the rendering on the client side.

Model

```
@Controller
public class HomeController {

    @GetMapping(value = "/") -> URL and HTTP method
    public String home(ModelMap model) {
        model.addAttribute("message","Hello!");  -> Model population
        return "index";   -> View name
    }

}
```

URI and URL

A **Uniform Resource Identifier** (**URI**) is a path that uniquely identifies a particular **resource**.

/clients/id/1

A **Uniform Resource Locator** (**URL**), is a reference to a web resource that specifies its location on a network and a mechanism for retrieving it.

A URL is a specific type of Uniform Resource Identifier (URI), although many people use the two terms interchangeably.

http://192.168.1.1:8081/myservice/clients/id/1

Mapping URIs

http://bookstore/books/{bookId} contains the variable *bookId*.

When we provide a bookId=15 the URI becomes *http://bookstore/books/15*

This type of mapping is very useful for pages where we need to search for items or catalogue lists.

Spring lets us map this into a Controller using a **@PathVariable**:

```
@GetMapping(value = "/search/{id}")
public String search(ModelMap model, @PathVariable String id) {
    model.addAttribute("message", "My Path Variable is " + id);
  return "index";
}
```

Mapping URIs

We can have as many path variables we want within a Controller:

```
@GetMapping(value = "/search/{type}/{id}")
public String searchMore(ModelMap model, @PathVariable String type,
@PathVariable String id) {
    model.addAttribute("message", "My Path Variables are " + type + " "
+ id);
    return "index";
}
```

Request Params

Passing arguments to the Controllers is done through Request Parameters:

```
@GetMapping(value = "/search")
public String searchMore(Model model, @RequestParam(name = "type",
required = false, defaultValue = "all") String type) {
    model.addAttribute("message", "My Request Parameter is " + type);
    return "index";
}
```

Request Parameters can be omitted by changing the required flag and they can have a default value.

You can set their values in the request url by adding them as URL parameters:
/books?type=fiction&author=James

Initializing controllers

There are cases when we want to perform some basic initialization for our controller- for example, load a list of all counties supported in the Registration, or load a map with default values that we want to show in our page.

Stuff like that need to happen just after the controller class is created and then be available until the web app terminates.

Spring permits this kind of initialization with the @PostConstruct annotation:

```
@PostConstruct
void initBooksMap() {
   books = new HashMap<>();
   books.put("The Scarlet Letter", new Book("The Scarlet Letter", "1900", "Nathaniel
Hawthorne"));
}
```

This function will run just after the Controller is created and it will fill the map called books with the required data.

Session

A HTTP session is a '*temporary storage'* that the application server establishes for each client.

Session is a way to keep necessary data for the client stored during his visit on the website. Essentially, session is what we can call "server-side cookies" : a storage layer that is persistent as long as the user is active on a page.

Attributes stored in Session are persistent.

This means that they are not removed until a) The session is destroyed when the user logs out, leaves the site OR a session expiration event occurs or b) the programmer explicitly removes them. Unlike parameters stored in Model , which is temporary and tied to each view, attributes stored in session can be accessed from all pages.

The server keeps a unique id (called SID- session ID) that identifies each session for a given user.

Session

Session storage is a way to retain data between redirects or chained requests.
A session is unique for each user and it has a limited TTL, but data stored there cannot be lost, unless the session has expired or you remove them manually

In Spring you can mark data that you want to be persisted in the Session using the **@SessionAttribute** annotation:

```
@Controller
@SessionAttributes("userForm")
public class HomeController {...}
```

Once we call a method where we add a value for `userForm` to the model, data will be persisted. From now on we can fetch them from the session in subsequent calls.

Important note:
Using the session a easy way to transfer data, but it's a bit tricky since you need to clear it manually and you might end up with leftover data that you no longer need. Remember to clear out anything you don't need anymore!

Redirects

Besides returning a view, a Controller can also redirect to another controller method:

```
@GetMapping(value = "/redirectExample")
public String redirectExample(ModelMap model) {
    return "redirect:/";
}
```

This technique is used mostly in POST methods, where we usually redirect to a GET after success.

However, redirecting to a page has one major drawback:
You cannot pass data through the model attributes.
Spring offers to handy solutions for this issue: **Session Storage** and **RedirectAttributes.**

Redirects

A more elegant way to transfer data between redirects , rather than using Session Storage, is RedirectAttributes
The RedirectAttributes are “Flash attributes” that are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and removed immediately.

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String foo(ModelMap model, RedirectAttributes
redirectAttributes) {
   redirectAttributes.addFlashAttribute("message", "test");
   return "redirect:/";
}
```

Redirect attributes can only store primitive values , since they serialize your data, store it in the session and pass it to the next view as url parameters.

Redirects

There are two types of redirect attributes that we can use:

Redirect Attributes

```
RedirectAttributes addAttribute(String var1, Object var2)
```

When we use that method, the objects we add are displayed in the url as request parameters.
Only simple (serializable) values can be added.

Flash Attributes

```
RedirectAttributes addFlashAttribute(String var1,Object var2)
```

Values added this way are not added in the url. Its the recommended method.

Cookies

Cookies are small text files that are stored on client computers browser directory.

Cookies can be used for any of the following

1. Remember the information about the user who has visited a website, so that when the user returns to the website in future they can show information useful to the user,

2. Track internet users web browsing.

3. Used in Session management.

Cookies

Adding cookies

Cookies are written in the response of the HTTP entity.
We can customize their name, value, and max age.

```
@GetMapping("/")
public String addCookies(HttpServletResponse response) {
    Cookie cookie = new Cookie("name", "value");
    cookie.setMaxAge(24 * 60 * 60);
    response.addCookie(cookie);
    return "index";
}
```

Cookies

The max age feature is pretty important.

If it is set to **-1** ,then the cookie is marked as a **“session cookie”**, that is, its TTL will be until the user logs out or closes the browser.

If it is set it to **0**, then the cookie is **deleted immediately** after the browser process the request.

Finally, any other positive value set the lifetime of the cookie to X seconds.

Cookies

In order to read a cookie, we need to parse it from the request headers.

```
@GetMapping
public String readCookies (HttpServletRequest request){
    Cookie[] cookies = request.getCookies();

    if (cookies != null) {
        Arrays.stream(cookies).forEach(c -> System.out.println(c.getName() + "=" +
c.getValue()));

}

    return "index";
}
```

Interceptors

Sometimes we need to add some business logic before or after a Controller Method is invoked.

For example, we might want to add a warning message to users that have logged in multiple times without success. This means that we have to check the failed logged in attempts and then add the warning to whatever page the user might be redirected to after successful login.

So we need something - a method- that will be invoked after the login process, and then add some extra functionality before the user is redirected to his normal view.

Interceptors

For this functionality, Spring offers a special class called Interceptor.

This class is used to "**intercept" client requests** and **process** them. Essentially**, it is injected before or after the Controller you specify,** and it can modify the contents of the model, redirect to another page, or modify the response.

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.

Interceptors

Interceptors must implement the ***HandlerInterceptor*** interface.

This interface contains three main methods:

- **prehandle()** – called before the actual handler is executed, but the view is not generated yet
- **postHandle()** – called after the handler is executed
- **afterCompletion()** – called after the complete request has finished and view was generated

These three methods provide flexibility to do all kinds of pre- and post-processing.

And a quick note – the main difference between *HandlerInterceptor* and *HandlerInterceptorAdapter* is that in the first one we need to override all three methods: *preHandle()*, *postHandle()* and *afterCompletion()*, whereas in the second we may implement only required methods.

Interceptors

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

This method is used to intercept the request before it's handed over to the handler method. This method should return 'true' to let Spring know to process the request through another spring interceptor or to send it to handler method if there are no further spring interceptors.

If this method returns 'false' , Spring assumes that request has been handled by the spring interceptor itself and no further processing is needed. We should use response object to send response to the client request in this case.

Object *handler* is the chosen handler object to handle the request.

Interceptors

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.

Interceptors

```
@Component

   public class HelloInterceptor  extends HandlerInterceptorAdapter {

      @Override
      public void postHandle(HttpServletRequest request,
                             HttpServletResponse response, Object handler,
                             ModelAndView modelAndView)  throws Exception {

          modelAndView.getModelMap().addAttribute( "message", "This is an intecepted
   message");

      }

   }
```

Interceptors

```
@Configuration
public class AppConfiguration implements WebMvcConfigurer {

    @Autowired
    HelloInterceptor helloInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(helloInterceptor).addPathPatterns("/");
    }

}
```

A few things about MVC architecture

The MVC architecture is built strictly upon the multi-layer architecture layer.
A good structural design implies that each layer of the system will have **as few responsibilities as possible.**
This is called the Single Responsibility Principle.

Essentially, it means that :
Your class should do as few things as possible.

For example, the **Controller** class needs to ONLY have the business logic and error handling for our page. It should not have concerns about connection to the database , sorting data, or calling external services.
So who will do that?
-> Seperate classes to which the controller will delegate the responsibility for those actions.

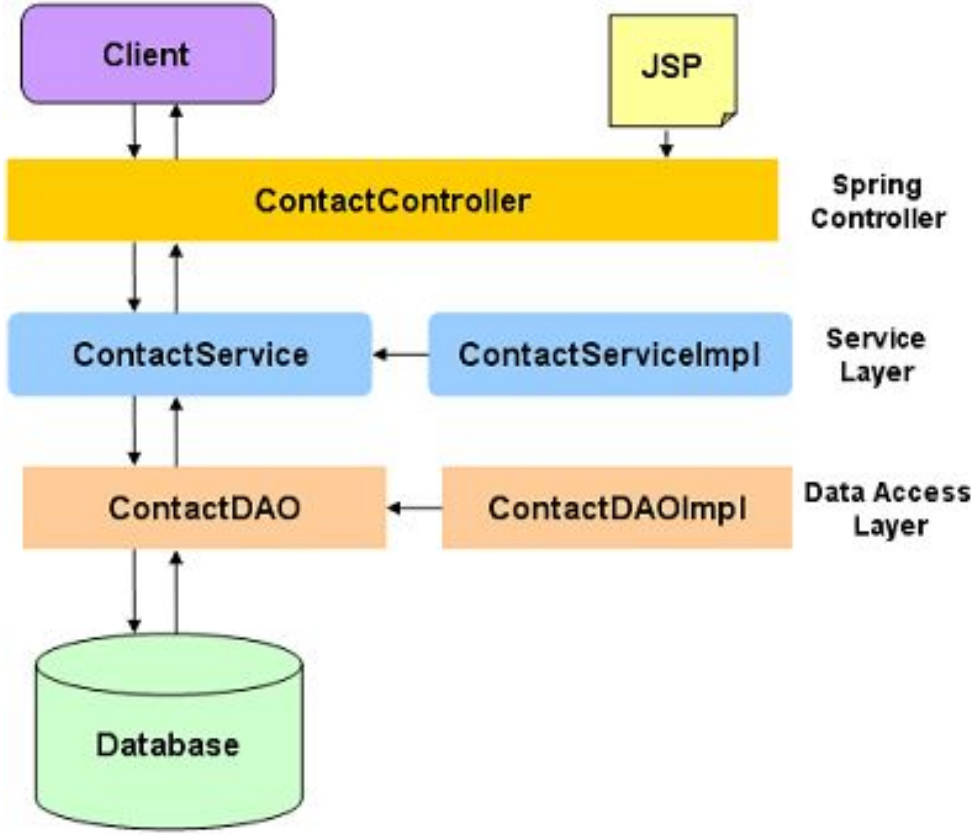
Multi Layered Architecture

In order to solve the previous problem, the following architecture has been proposed for MVC applications:

Controller Layer
Should handle just url mappings and business logic.

Service Layer
Should provide the controller with useful functionality (reach external services, fetch data, caching)

Data Access Layer
Should handle all data related operations



Spring Bean Annotations

@Component
Generic stereotype annotation for any Spring-managed component.

@Controller
Stereotypes a component as a Spring MVC controller.

@Repository
Stereotypes a component as a (Database) store. Delegates SQL exceptions to DataAccessExceptions.

@Service
Stereotypes a component as a service.

Exception Handling

But what do we do in cases that things go wrong?
We need to handle the exceptions.

There **two kinds of exceptions in Java:**

Unchecked Exceptions:

Those are the exceptions that are thrown when something goes wrong unexpectedly.
They all inherit the RuntimeException class and do not need to be declared in the signature of a method.

Checked Exceptions:

Those are more specific, 'controlled' exceptions that a method throws. We anticipate them (for example, a method that opens a file expects the possibility of the given file not being there). They are declared in the method signature.
Generally, the best practice is to handle checked exceptions, but always have a fallback for the generic runtime exception so that we never end up with a broken app.

Exception Handling

The basic, most intuitive way of handling an exception si by inserting a try -catch block in your code and then controlling the business logic about what should happen next.

```
try {
   clientService.fetchClients();
} catch (Exception ex) {
   log.error("An exception occured : ", ex);
   model.addAttribute("errorMessage", "Clients could not be fetched");
}
```

All exceptions that inherit the Exception class will be caught in that try/catch.

Exception Handling

Sometimes, however, we might need to have **different error handling logic for each error case.**

For example, if there are no clients in the DB, we might need to throw a specific error, but it the system is down (no access to the DB) then we need to show a 500 error page to the user until the system is up and running again.

In order to do that, we need to seperate the exception handling into concrete exceptions. This means, **that for every error we anticipate from our services, we need to define a custom exception class and throw it when necessary.**

Exception Handling

For example , we can:

1) Define a **ClientsNotFoundException** is there are no clients in the system and throw it in our service if there are no clients to fetch.

```
public class ClientsNotFoundException extends RuntimeException {}
```

2) Throw that in our service layer:

```
public List<Client> getClients() {

    List<Client> clients = helloRepository.getClients();
    if (clients.isEmpty()) {
        throw new ClientsNotFoundException("No clients Found!");
    }
    return clients;
}
```

Exception Handling

3) Catch that specific exception in our try-catch block and handle it:

```
try {
   clientService.fetchClients();
} catch (ClientsNotFoundException ex) {
   log.error("An exception occured : ", ex);
   model.addAttribute("errorMessage", "Clients could not be fetched");
} catch (Exception ex) {
   log.error("A Generic exception occurred : ", ex);
    // do other stuff here
}
```

All exceptions that are not `ClientsNotFoundException` will be caught in the second try/catch.

Exception Handling

This logic applies a lot to the MVC architecture, where we are dealing with the presentational model of our webapp as well. It's very likely that we need to follow this approach and separate our exceptions and their handling , so that we create different user journeys.

Do not be afraid of declaring a lot of exceptions.
This way , it will be easier for you to know what's going on and let the Java inheritance mechanism hande the different cases of errors:

```
try {
   clientService.fetchClients();
} catch (ClientsNotFoundException ex) {
//do stuff
} catch (TimeoutException ex) {
//do stuff
} catch (Exception ex) {
//fallback to unknown }
```

Exception Handling

As good as try catches are, when we have many cases, they tend to clutter our code.
Also, what happens when we call the same method multiple times from many controllers and we need to have the same error handling in our code?

Spring lets us centralize our error handling mechanism by using two major features:

1. Declaring specific methods for handling errors in our Controllers, using the **@ExceptionHandler** annotation
2. Using a **HandlerExceptionResolver** class that will intercept all exceptions
3. Using a **@ControllerAdvice** class that will run after/before our controller methods and deal with exceptions

All of these do have one thing in common – they deal with the separation of concerns very well.

@ExceptionHandler

We can define exception- handling methods in our Controller classes.

All we need is to **annotate** these **methods** with **@ExceptionHandler annotation**.

```
@ExceptionHandler(ClientsNotFoundException.class)
public String handleError(HttpServletRequest request, RedirectAttributes redirectAttrs,
RuntimeException e) {
    redirectAttrs.addFlashAttribute( "message","Couldn't fetch clients");
    return "redirect:/";
}
```

Those methods will be caught as soon as an exception is thrown.

Execution of the main code block will be stopped and the error handling method will be invoked.

Those annotations work ONLY in controller level. They do not apply to services or other components

@ControllerAdvice

The second pattern is using the @ExceptionHandler annotation within an Advice Controller

Spring supports a special case of Controllers called Advice Controllers.

Those Controllers work like Interceptors: they intercept requests to specific controllers and are executed before - and sometimes, after- a Controller.

This way, we can add custom logic for:

1. Handling cases globally: add logic that is common to all controllers
2. Initializing the state before an actual controller is called
3. Catch runtime exceptions that occur within a controller and after the execution in interrupted, handle what must happen

@ControllerAdvice

We can create an Advice Controller using the **@ControllerAdvice** annotation and specify the package or the classes it should monitor.

```
@ControllerAdvice(basePackageClasses = ClientsController.class)
public class GlobalAdviceController {
    @ExceptionHandler(ClientsNotFoundException.class)
    public String handleError(HttpServletRequest request, RedirectAttributes redirectAttrs,
    RuntimeException e) {
        redirectAttrs.addFlashAttribute("message", "Couldn't fetch clients");
        return "redirect:/";
    }
}
```

Views and Templates

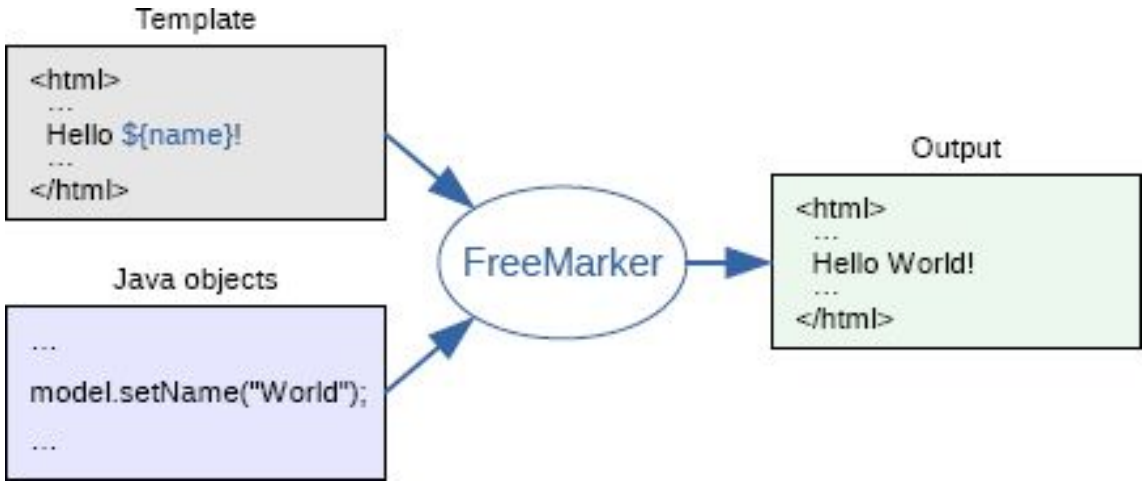
Code.Hub

Templates

As discussed before, the Model layer is responsible for preparing a page's data (issue database queries and do business calculations)

The View layer is responsible that already prepared data.

In the template you are focusing on how to present the data, and outside the template you are focusing on what data to present.



Freemarker

FreeMarker is a Java based template engine from the Apache Software Foundation.

Like other template engines, FreeMarker is designed to support HTML web pages in applications following the MVC pattern.

Essentially,it’s an enriched HTML based language, where you can perform direct data manipulation and access Spring’s Model attributes.

Conditional blocks, iterations, assignments, string and arithmetic operations and formatting, macros and functions are all supported

Freemarker

Now let’s try creating a simple index page. We’ll create a page called index.ftl under the /web-inf/views/ directory.

```
<html>
<head>
    <title>Welcome!</title>
</head>
<body>
    <h1>Welcome John Doe!</h1>
    <h2> Popular books for the week:</h2>
    <p> Patterns of Enterprise Application Architecture </p>
    <p> Martin Fowler </p>
    <img src= "https://www.martinfowler.com/books/eaa.jpg"/>
</body>
</html>
```

Adding data

Now let's create a method in our controller in order to be able to see it:

```
@Controller
public class IndexController {
    @GetMapping(value = "/")
    public String home(Model model) {
      model.addAttribute("message", "Hello World!");
    return "index";
}
```

Adding data

Now let's create a method in our controller in order to be able to see it:

- Notice that in the previous page , we had the user's name as well the book's info hardcoded
- Those are pieces of data that would be fetched dynamically from a data store in our Spring controller and added to the model , so we need to being able to access them in our page
- In order to do this, we'll create a *template* and use Freemarker's instructions to handle data.
- Suppose our controller has the user's name stored in a variable called ***name*** and the book data stored in an object called ***book.*** We will add them in our model in the `home(ModelMap model)` method .

Displaying data

And now we can access directly our variables using the ${variable} command:

```
<html>
  <head>
     <title>Welcome!</title>
  </head>
    <body>
        <h2>Popular books for the week:</h2>
        <p> ${book.title}</p>
        <p> ${book.author}</p>
            <img src="${book.image}"/>
  </body>
</html>
```

Data types

- The name variable which holds a simple constant value is called a ***scalar***. Scalars can be accessed directly using the ${var} command. Their value can be a string , number , date or a boolean.

- The book variable is a complex object with fields which we access through the ${object.field} notation. Those are called ***hashes.*** Hashes store other variables (the so called *sub variables*) which we can look up by their name. Maps are a good example of hashes.

- Suppose that in the previous example, we had a list of books instead of one called books:{book1, book}. The subvariables in books are just items in a list- they dont have a names. This data type is called a ***sequence*** . To access a subvariable of a sequence we use a numerical index in square brackets. So to get the first book we need to write books[0].

  Let’s try that in the previous example.

Data types

```
@Controller
public class BooksController {
private static final String BOOK_DATA= "books";

 @GetMapping(value = "/books")
   public String home(ModelMap model) {
    List<Books> books = getBooksList();
    model.addAttribute(BOOK_DATA,books);
    return "books";
}
}
```

Data types

```
<html>
  <head>
     <title>Welcome!</title>
  </head>
    <body>
        <h2>Popular books for the week:</h2>
        <p> ${books[0].title}<p>
        <p> ${books[0].author}<p>
            <img src="${books[0].image}"/>

        <p> ${books[1].title}</p>
        <p> ${books[1].author}</p>
            <img src="${books[1].image}"/>
  </body>
</html>
```

Data types

Now, the previous example just showcases how to access the list items one by one, which is not really useful. Let's try adding a loop to iterate through all list items instead.
Looping through a **List** in Freemarker is similar to the foreach loop in Java:

```
<#list  myList  as myItem>
     // do stuff with myItem
      ${myItem.name}
</#list>
```

Iterating through Maps is done in a similar way , where we need to iterate over the keys and get values associated with each of them:

```
<#list myMap?keys as key>
    ${value} = ${myMap.get(key)}
</#list>
```

Data types

```
<head>
    <title>Welcome!</title>
</head>
  <body>
    <h2>Popular books for the week:</h2>
  <ul>
      <#list  books  as book>
      <li>
          <p> ${book.title}</p>
          <p> ${book.author}</p>
          <img src="${book.image}"/>
      </li>
      </#list>
  </ul>
  </body>
```

Conditionals

Now let's suppose that in the previous example, we need to display the same page for both registered and guest users.
In case of guests, we cannot display the user's name, so we need to display a generic greeting instead.
For this, we would need to check if the **${name}** variable exists in our model and if not, display the greeting

The **if-else** directive in freemarker is the following:

```
  <#if (condition)>
    //do stuff
<#else>
    //do other stuff
</#if>
```

Conditionals

Comparison operators: **< , > , <=, >=**
Logical operators : **|| , && , !**
Equality operators : **!= , ==**

A few notes on comparisons:

- You can use comparison operators on Primitive types (numerical/boolean), but **NOT** Strings.
- You can test for equality with == and != for **ALL** types, including Strings.
- However, the expressions on both sides of the == or != must evaluate to a scalar (not a sequence or hash - we cannot compare complex objects).
- Furthermore, the two scalars must have the same type (i.e. strings can only be compared to strings and numbers can only be compared to numbers)

Forms

Now let's try creating a Registration Page. For this we will need to create a form in register.ftl:

```
<div class="container">
  <form action="/register" name="registerForm"  method="post">
      <label for="email">Email</label>
      <input type="email" name="email" id="email" placeholder="email">

    <label for="username">Username</label>
      <input type="text" name="username" id="username" placeholder="username">

      <label for="password">Password</label>
      <input type="password" name="password" id="password" placeholder="password">

      <button type="submit">Register</button>
  </form>
</div>
```

Forms

Create a corresponding RegisterForm class in our data model:

```
public class RegisterForm {

   private String username;

   private String password;

   private String email;

//Field accessors go here - omitted for space
 }
```

Forms

And here we would create our controller:

```
@Controller
public class RegistrationController {

 private static final String REGISTER_FORM = "registerForm";

   @GetMapping(value = "/register")
   public String register(Model model) {
     model.addAttribute(REGISTER_FORM, new RegisterForm();
      return "register";
   }

   @PostMapping(value = "/register")
   public String register(Model model,  @ModelAttribute(REGISTER_FORM) RegisterForm
registerForm) {
     //here we would have the logic for sending the registration request  to our service
       return "register";
   }
```

Validation

Code.Hub

Data Validation

When we accept user input in any web application, it becomes necessary to **validate** them.

Validation

Assert that data are:

1. correct
   ex: Email address should contain a valid domain name like @gmail.com
2. clean
   ex: username should only consist of letters and numbers and no special characters (#$%..)
3. complete
   ex: the user must submit both an email and a username during registration
4. used as intended
   protect against code injection and data manipulation

Data Validation

Validation can be performed either **client side** or **server side**

Client side validation is used for validating user input on the fly :

- Programmer adds custom rules for each form input
- User is able to see the validation result before submitting the form
- Usually done through jquery's Validation plugin or similar libraries
- HTML-5 has also some built-in support for some basic validation rules

Data Validation

Server side validation is performed **after the form is submitted**

- Used for more complex validation rules or when we need to validate data against a data store (ex: check if a username already exists and prompt the user to pick another one during registration)
- Can be used in combination with client side validation as a second layer of security in case a user has javascript disabled or to cover cases that the client side validation cannot
- Protect against injection attacks: It's possible for a malicious user to manipulate or bypass the client side validation rules using the browser's dev tools
- Spring supports Server side validation and has a dedicated library for it

Data Validation

Example use case:

- Registration Page
- User tries to register with a new account
- Firstname, Lastname, email and password are mandatory
- Email should be valid
- Password should be > 5 characters
- We'll start with Client side validation fist and then proceed to server side with Spring

Server Side Validation

```
<div class="container">
    <form action="/register" name="registerForm"  method="POST">

         <label for="email">Email</label>
        <input type="email" name="email" id="email" placeholder="email">

        <label for="username">Username</label>
        <input type="text" name="username" id="username" placeholder="username">

        <label for="password">Password</label>
        <input type="password" name="password" id="password" placeholder="password">

        <button type="submit">Register</button>
    </form>
```

Server Side Validation

```
public class RegistrationValidator implements Validator {

    @Override
    public boolean supports(Class<?> aClass) {
        return RegisterForm.class.isAssignableFrom(aClass);
    }

    @Override
    public void validate(Object target, Errors errors) {

        RegisterForm registerForm= (RegisterForm) target;

           //here perform your checks
        if(registerForm.getEmail()==null)
           {
                // add an error for the field called 'email'
                errors.rejectValue("email", "empty");
           }


           ValidationUtils.rejectIfEmpty(errors, "username", "empty");
      }
```

Server Side Validation

A Validator works just like any other bean

- In order to use it we need to create an instance of our custom Validator and inject it to our Controller
- Moreover, we need to let Spring know that which object in our model needs validation
- For this we'll need to add it to the DataBinder instance

```
@Controller
public class RegistrationController {

private static final String REGISTER_FORM = "registerForm";

@Autowired
private RegistrationValidator registrationValidator;


@InitBinder(REGISTER_FORM)
protected void initBinder(final WebDataBinder binder) {
     binder.addValidators(registrationValidator);
}
```

Server Side Validation

Spring has the ability to automatically validate **@Controller** inputs.

- The configured Validator instance will validate the model object annotated by the **@Valid** annotation
- Any constraint violations will be exposed as errors in the **BindingResult** object

IMPORTANT:

The object annotated @Valid should **always** be followed with the BindingResult object in your Controller method declaration or validation won't work

```
public String register(Model model, @Valid RegisterForm registerForm,BindingResult bindingResult)
```

Server Side Validation

```
@Controller
public class RegistrationController {

     private static final String REGISTER_FORM = "registerForm";

     @Autowired
     private RegistrationValidator registrationValidator;

     @InitBinder(REGISTER_FORM)
     protected void initBinder(final WebDataBinder binder) {
          binder.addValidators(registrationValidator);
     }
   @PostMapping(value = "/register")
   public String register(Model model,
          @Valid @ModelAttribute(REGISTER_FORM) RegisterForm registerForm,
           BindingResult bindingResult) {

       if (bindingResult.hasErrors()) {
          //have some error handling here, perhaps add extra error messages to the model
          return "register";
       }

     //here we would have the logic for sending the registration request  to our service
       return "register";
```

Binding the form

In order for the validation to work, we need to ‘bind’ the form the the backend validator.
This essentially means that we have to tell spring how to match the form we have created in the FE with the one we have in our controller.

This is achieved by using some commands that are inside the built in templating library that spring has for Freemarker.

```
<#import "/spring.ftl" as spring />
```

Then we can use the binding command:

```
<form action="/register" name="registerForm" method="POST">
    <@spring.bind "registerForm.email"/>
        <input type="email" name="email" id="email" placeholder="email">
    </div>
```

Binding the form

After we have done this, we need to add another command that will evaluate the erros that the validation finds and then show them under the form inputs:

```
<@spring.bind "registerForm.email"/>
        <input type="email" name="email" id="email" placeholder="email">
            <#list spring.status.errorMessages as error>
                <span>${error}</span>
            </#list>
```

Annotations

Annotation based Validation

- **Java Bean Validation (JSR-303)** is a Java specification that allows us to express validation constraints on objects via annotations.

```
@NotNull(message = "Field cannot be empty")
private String firstName;
```

- Fields within a class are annotated with a constraint and a corresponding error message
- Constraints can be built in or user defined. Several built-in constraints are available in the *javax.validation.constraints* package.
  https://docs.oracle.com/javaee/7/tutorial/bean-validation001.htm
- The most popular implementation of this standard is Hibernate Validator.

Annotations

Add the necessary dependencies to your pom.xml:

```
    <dependency>
      <groupId>javax.validation</groupId>
      <artifactId>validation-api</artifactId>
      <version>1.1.0.Final</version>
</dependency>

<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>4.1.0.Final</version>
</dependency>
```

Annotations

```
public class RegisterForm {

  private static final String USERNAME_PATTERN = "^[a-zA-Z0-9]*$";

 private static final String PASSWORD_PATTERN = "^[a-zA-Z0-9@#$%^&]*$";

 private static final String MAIL_PATTERN = "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{1,63}$";

 private static final int  PASSWORD_MINSIZE = 6;


  @Pattern(regexp = USERNAME_PATTERN, message = "Please provide a valid input")
  private String username;

  @NotNull(message = "Please provide a valid input")
  @Pattern(regexp = PASSWORD_PATTERN, message = "Please provide a valid input")
  @Size(min =PASSWORD_MINSIZE, message = "Please provide a valid input")
  private String password;

  @Pattern(regexp = MAIL_PATTERN, message = "Please provide a valid input")
  private String email;

 //Field accessors go here - omitted for space
..
```

Code.Hub

Annotations

```
@Controller
public class RegistrationController {
private static final String REGISTER_FORM = "registrationForm";

    @PostMapping(value = "/register")
    public String register(Model model,
            @Valid @ModelAttribute(REGISTER_FORM) RegisterForm registerForm,BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            //have some error handling here, perhaps add extra error messages to the model
            return "register";
        }
         //here we would have the logic for sending the registration request  to our service
        return "register";
    }
```

Annotation based Validation is an alternative to using a custom validator class. It covers most common validation rules so it will save you a lot of time

Adding annotations to the form will help maintainability

You can use a mix of both methods if you need to perform more complex checks

Adding messages dynamically

In the previous example, we had all the error messages hardcoded

- This means that every time we change the message value, we need to change the code,recompile and redeploy the webapp
- This is a very bad practice : We must not have to make changes in classes for altering text or numeric values that could be easily parameterized

Spring gives us the option of using property files containing key - value pairs with our messages:

```
register.email.invalid=Please provide a valid mail.
```

We can then reference the error message in our code using the property key instead the actual value:

```
@Pattern(regexp = MAIL_PATTERN, message = "{register.email.invalid}"
private String email;
```

By default when using annotations , Spring looks for a *ValidationMessages.properties* in your resources folder.

We just have to create this file and then move all our messages there as key-value pairs.

Adding messages dynamically

```
public class RegisterForm {

   private static final String USERNAME_PATTERN = "^[a-zA-Z0-9]*$";

  private static final String PASSWORD_PATTERN = "^[a-zA-Z0-9@#$%^&]*$";

  private static final String MAIL_PATTERN = "[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{1,63}$";

  private static final int  PASSWORD_MINSIZE = 6;

   @Pattern(regexp = USERNAME_PATTERN, message = "register.username.invalid")
   private String username;

   @NotNull(message = "register.password.null")
   @Pattern(regexp = PASSWORD_PATTERN, message = "register.password.invalid")
   @Size(min =PASSWORD_MINSIZE, message = "register.password.size")
   private String password;

   @Pattern(regexp = MAIL_PATTERN, message = "register.mail.invalid"
   private String email;

  //Field accessors go here
```

Thank you for your time!

Code.Hub