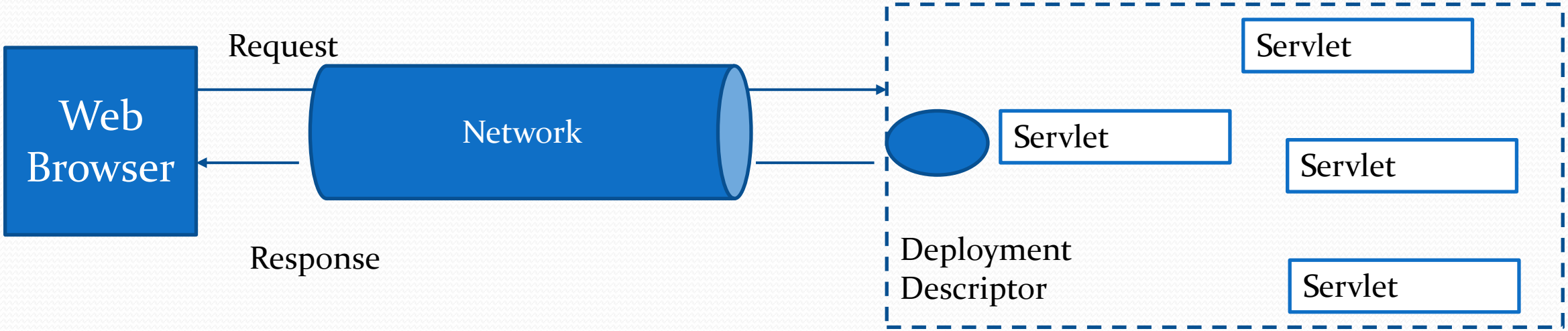
Web Frameworks: Spring

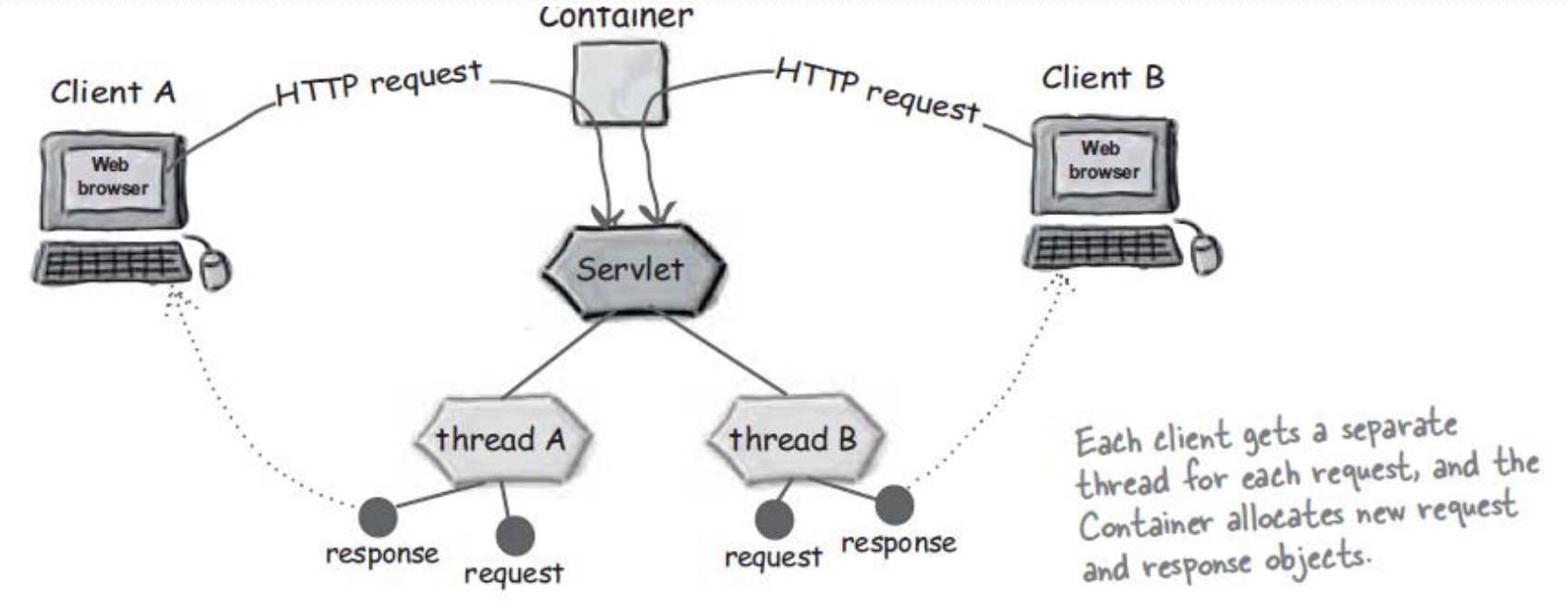
An Introduction

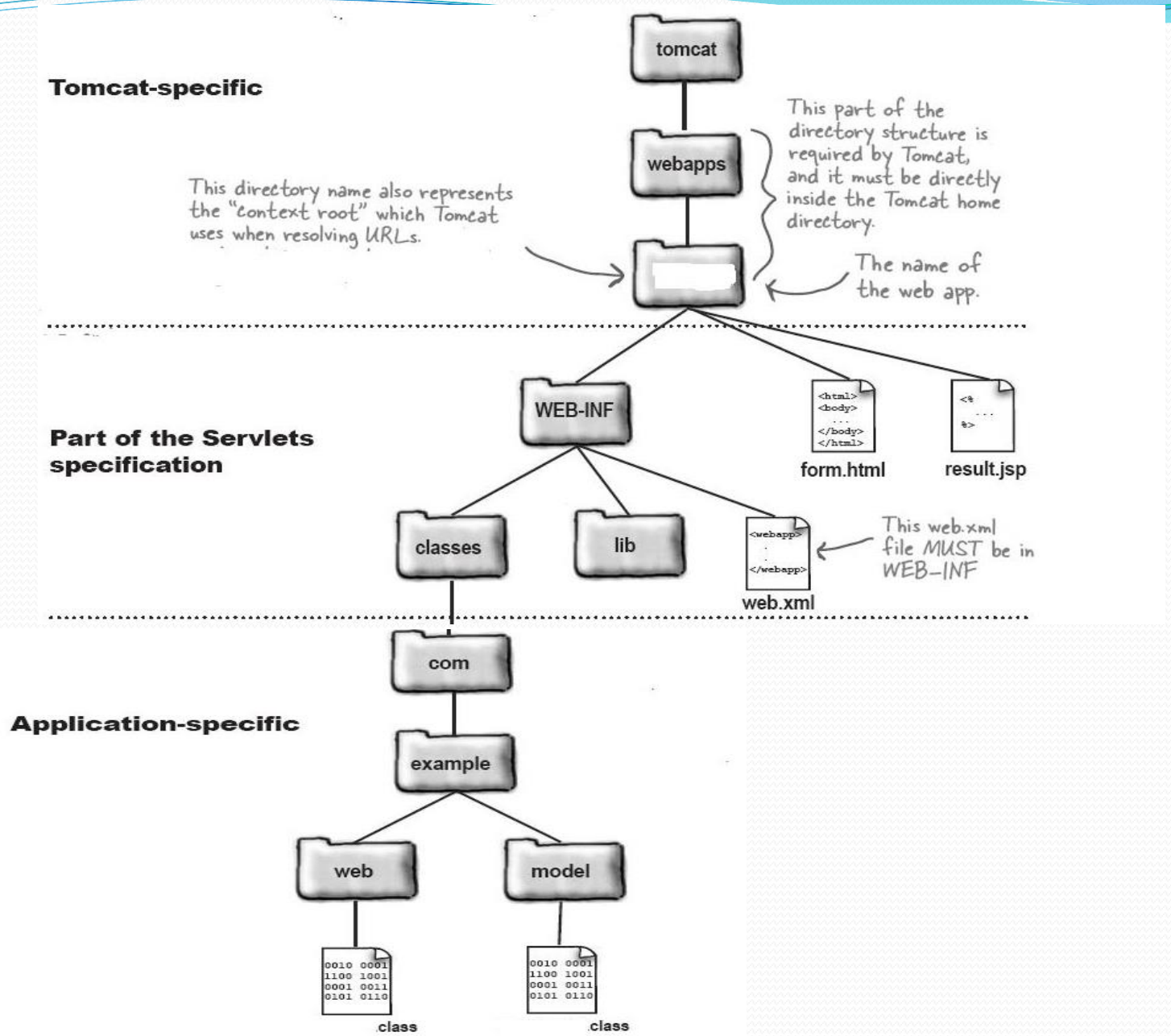
Web Container

```
@WebServlet("/SelectCoffeeMVC")
public class CoffeeSelectMVC extends HttpServlet {

        public void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws IOException, ServletException {
```



Handling Multiple Clients



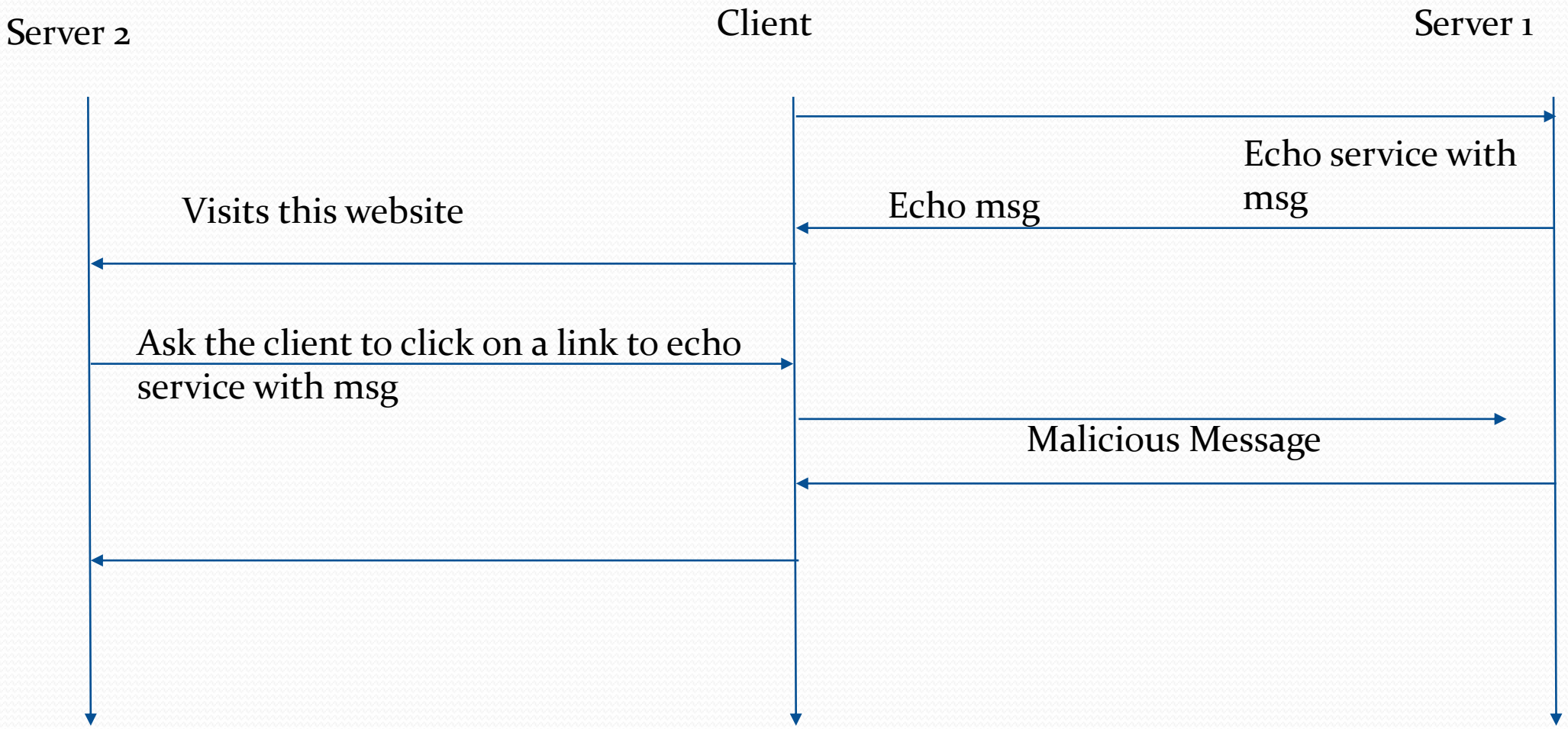

Tomcat-specific
tomcat
webapps
This part of the directory structure is required by Tomcat, and it must be directly inside the Tomcat home directory.
This directory name also represents the "context root" which Tomcat uses when resolving URLs.
The name of the web app.
Part of the Servlets specification
WEB-INF
form.html
result.jsp
classes
lib
web.xml
This web.xml file MUST be in WEB-INF
Application-specific
com
example
web
model
.class
.class

```
@WebServlet("/SelectCoffeeMVC.do")
public class CoffeeSelectMVC extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException {

            String color = request.getParameter("color");
            String addOn=request.getParameter("addOns");
            if(!color.equals("") && ! addOn.equals("")) {
                    Coffee c=new Coffee(color, addOn); }

            Cookie ck1;   HttpSession session=request.getSession();

            CoffeeExpert ce = new CoffeeExpert();
             String result="";
    try{
            Connection con=(Connection)getServletContext().getAttribute("key2");
            result = ce.getBrands(c,con);
    }catch(Exception e){ System.out.println(e);}

             request.setAttribute("brands", result);
            RequestDispatcher view =    request.getRequestDispatcher("result.jsp");
            view.forward(request, response);
 }}
```

Introduction

- Web.xml routes requests to the individual servlet's doGet or doPost methods
- doGet(…)
  - //extract parameters from request

Injection Attack



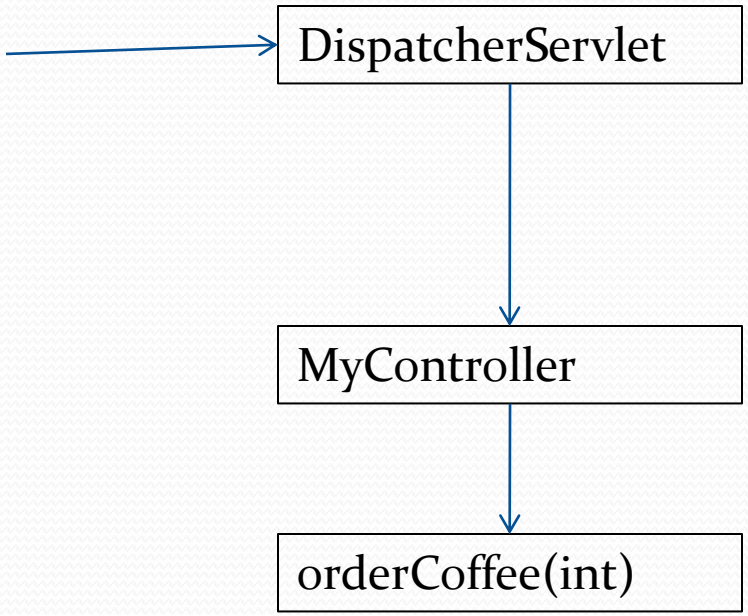
Introduction

- Web.xml routes requests to the individual servlet's doGet or doPost methods
- doGet(…)
  - //extract parameters from request
  - //validation
  - //Construct objects with parameters
  - //do the processing

Spring

- In Spring
  - A specialized servlet-DispatcherServlet
  - One or more controllers having simple methods to process HTTP requests
  - The DispatcherServlet routes requests to appropriate controller-individual methods of the controllers
  - DispatcherServlet extracts request parameters, performs data validation and marshalling
  - Provides an extra layer of routing over web.xml

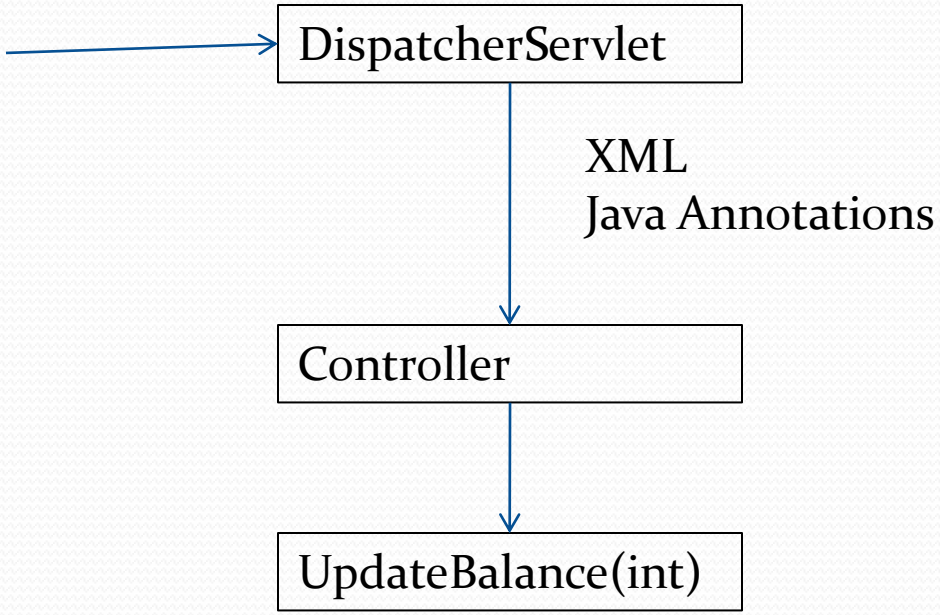
```
public class MyController {
    String orderCoffee(int) {
        ….
return …
}
```




```
@WebServlet("/SelectCoffee")
public class CoffeeSelect extends
HttpServlet {

p.v. doPost(HttpServletRequest
request, HttpServletResponse
response)
            throws IOException,
ServletException {
    //extract parameters from request
   //validation
   //Construct objects with
   parameters
   //do the processing
   }
```

Spring

Spring Controllers are plain java objects
No special interfaces to be implemented or classes to be inherited



- Routing is possible based on Path like servlets
- Request parameters using annotations
- Data validation is taken care of

Routing through DispatherServlet

```
public class ContactController {


  public Contacts getContacts() {
      //retrieve contacts
      Contacts c=…
              …
              return c;
    }
```

A Simple java class-no framework code

Routing through DispatherServlet

```
public class ContactController {

   public Contacts getContacts() {
        //retrieve contacts
        Contacts c=…
                …
                return c;
     }

public void friends(){
…
}
```

A Simple java class-no framework code

Introduction

- In EJB *public class HelloWorldBean implements SessionBean {*
- Spring avoids (as much as possible) littering your application code with its API
- Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class
- Instead, the classes in a Spring-based application often have no indication that they're being used by Spring
- Spring has enabled the return of the plain old Java object (POJO) to enterprise development

Mapping Request parameters to method parameters

```
@Controller
public class ContactController {

@RequestMapping("/search")
   public Contacts searchContacts(
      @RequestParam searchstr String SearchStr) {
         //retrieve contacts
         Contacts c=...
         ...
         return c;
      }
```

Retrieves request parameters and performs basic data validation so that value of *searchstr* can be mapped to *SearchStr*

Mapping Request parameters to method parameters

```
@Controller
public class ContactController {

@RequestMapping(value={"/search"}, method = RequestMethod.GET)
   public Contacts searchContacts(
      @RequestParam searchstr String SearchStr) {
         //retrieve contacts
         Contacts c=…
         …
         return c;
      }
```

Mapping Requests

```
@RestController
@RequestMapping(value = "/demo")
public class LoginController {

@RequestMapping(value = "/login")
public String sayHello1() {
 return "Hello World ";

}

@RequestMapping(value = "/dummy")
public String sayHello() {
 return "Hello World dummy";

}

}
```

- No need to worry about
  - how that request got to the server,
  - what format it got there in,
  - how all the data got extracted from it.
- It simplifies the methods and write cleaner, simpler methods, by using request parameters in the request mapping to extract that data and pass it into the method

```
@Controller
public class ContactController {

@RequestMapping("/search/{str}")
   public Contacts searchContacts(
                                  Search s) {
         //retrieve contacts
         Contacts c=...
          ...
          return c;
       }
```

Path variable provides a nicer way of parsing the request parameters rather than ?<key>=value

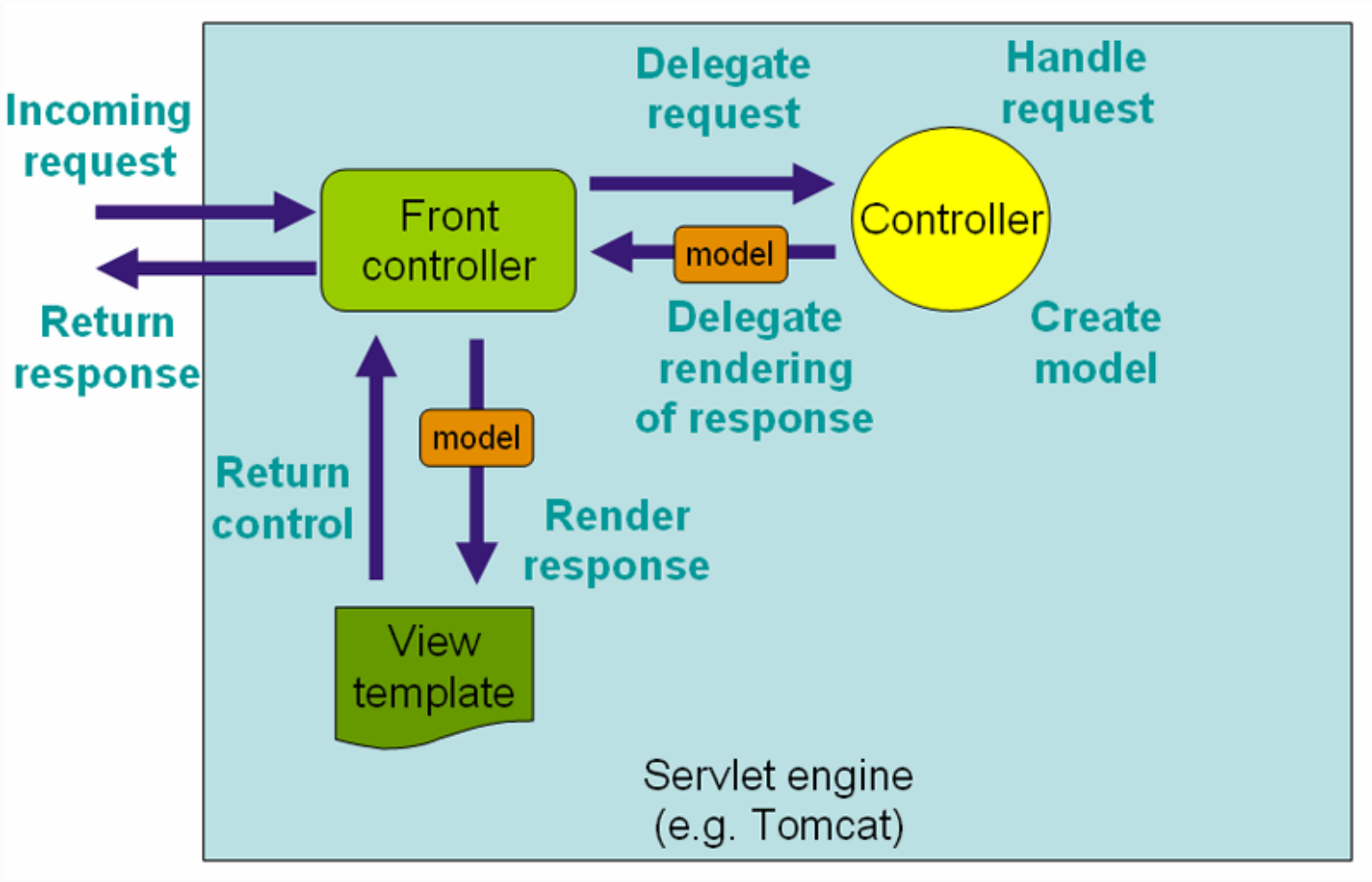
https://spring.io/guides/gs/serving-web-content/

```
public class ContactController {

@RequestMapping("/search/")
   public Contacts searchContacts(
                          Search s) {
        //retrieve contacts
        Contacts c=...
         ...
         return c;
      }
```

```
public class Search  {

 private string fname;
Private string lname;

public String
getFname() {..}
public setFname(String
name) {..}

...}
```

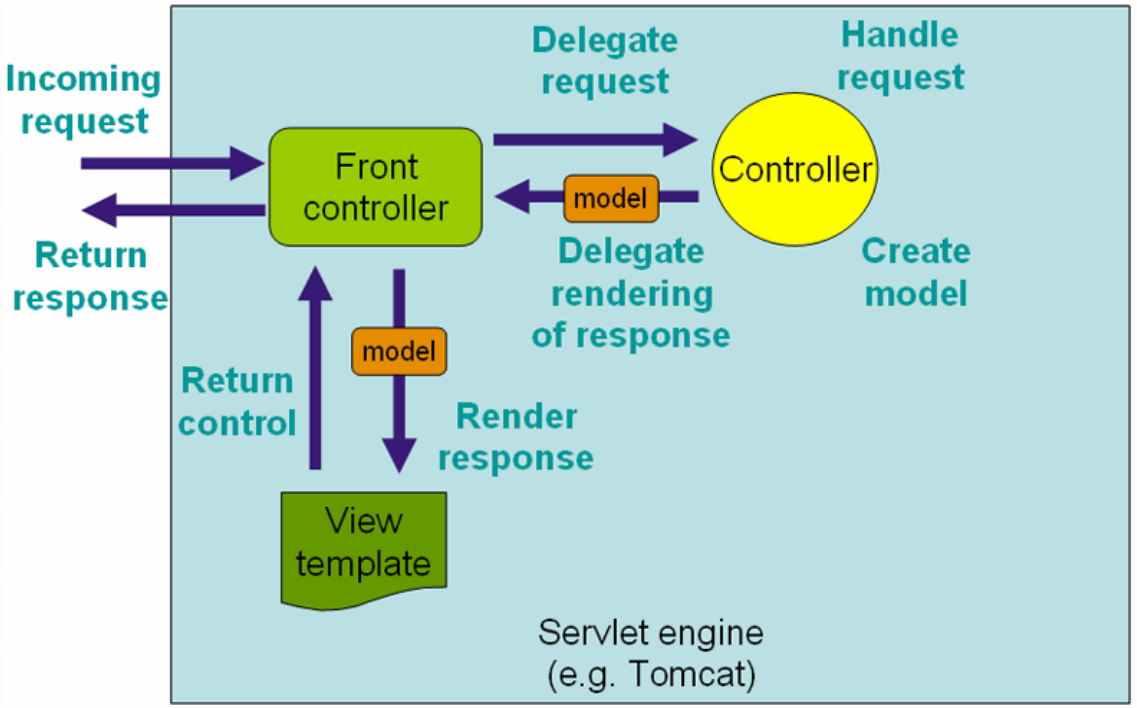
Automatic data marshalling through HTTP message converters

MVC Workflow



DispatcherServlet

- It gets its name from the fact that it dispatches the request to many different components, each an abstraction of the processing pipeline



1. Discover the request's Locale; expose for later usage.
2. Locate which request handler is responsible for this request (e.g., a Controller).
3. Locate any request interceptors for this request. Interceptors are like filters, but customized for Spring MVC.
4. Invoke the Controller.
5. Call postHandle() methods on any interceptors.
6. If there is any exception, handle it with a HandlerExceptionResolver.
7. If no exceptions were thrown, and the Controller returned a ModelAndView, then render the view. When rendering the view, first resolve the view name to a View instance.

DI Example

- The IoC container is in charge of creating objects, connecting them, configuring them and managing their entire life cycle from creation to destruction
- Spring Container uses DI to manage the components that make the application.
- These objects are named *Spring beans*.
- In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.
- Spring IoC Container defines the rules by which beans work.
- Bean is pre-initialized through its dependencies.
- After that, bean enters the state of readiness to perform its own functions.
- Finally, the IoC Container destroys bean

DI in Spring

- Beans are defined to be deployed in one of two modes:
  - singleton or
  - non-singleton.
- When a bean is a singleton, which is a default mode for bean's deployment, only one shared instance of the bean will be managed
  - all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned.
- The non-singleton, prototype mode of a bean deployment, results in the creation of a new bean instance every time a request for that specific bean occurs.