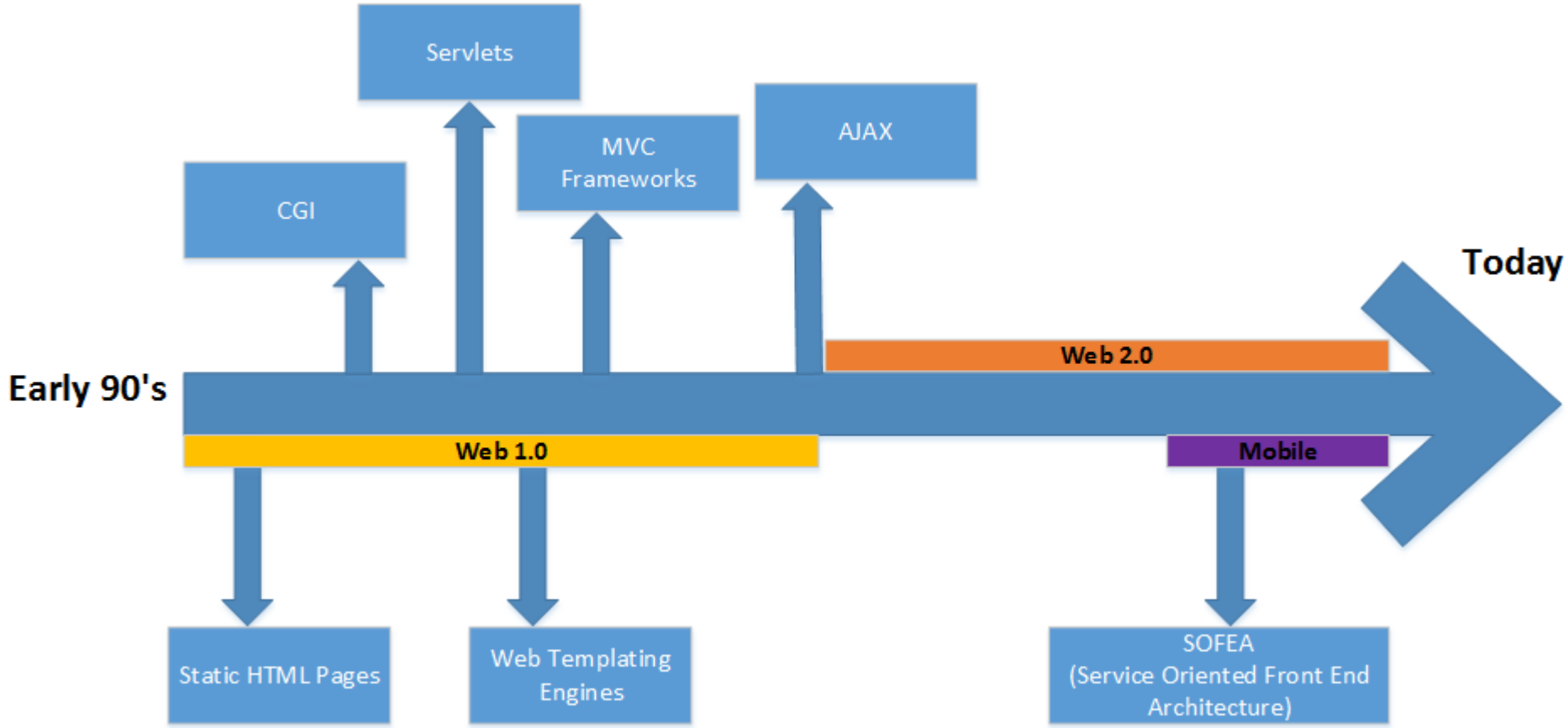
Spring MVC

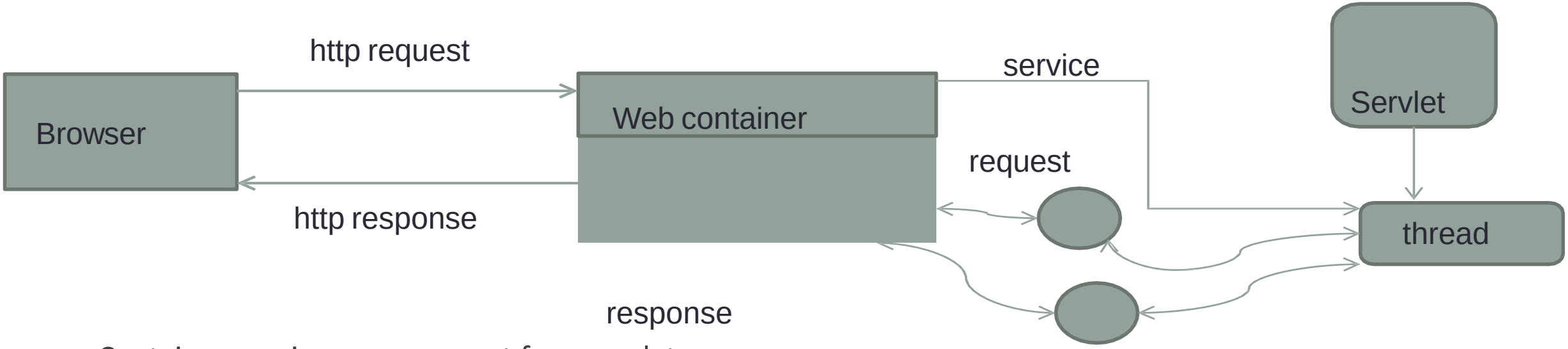
RUJUAN XING

History of Web Application



HTTP Servlets

- Java classes that run on Web Server to process HTTP Requests and Response



Container receives new request for a servlet

Creates HttpServletRequest and HttpServletResponse objects

Calls service method on HttpServlet object in thread

When thread completes, converts response object into HTTP response message

SimplestServlet

```
public class SimplestServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
       PrintWriter out = response.getWriter();
       out.print("<html><head><title>Test</title></head><body>");
       out.print("<p>Postback received</p>");
       out.print("</body></html>");

   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
       PrintWriter out = response.getWriter();
       out.print("<html><head><title>Test</title></head><body>");
       out.print("<form method='post'>");
       out.print("<p>Please click the button</p>");
       out.print("<input type='submit' value='Click me'/>");
       out.print("</form>");
       out.print("</body></html>");

   }
}
```

XML vs Annotations

```
<servlet>
   <servlet-name>HelloServlet</servlet-name>
   <servlet-class>edu.mum.cs.SimplestServlet</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>HelloServlet</servlet-name>
   <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

```
@WebServlet(name = "ServletDemo", urlPatterns = {"/", "/index", "welcome"})
public class SimplestServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    }
}
```

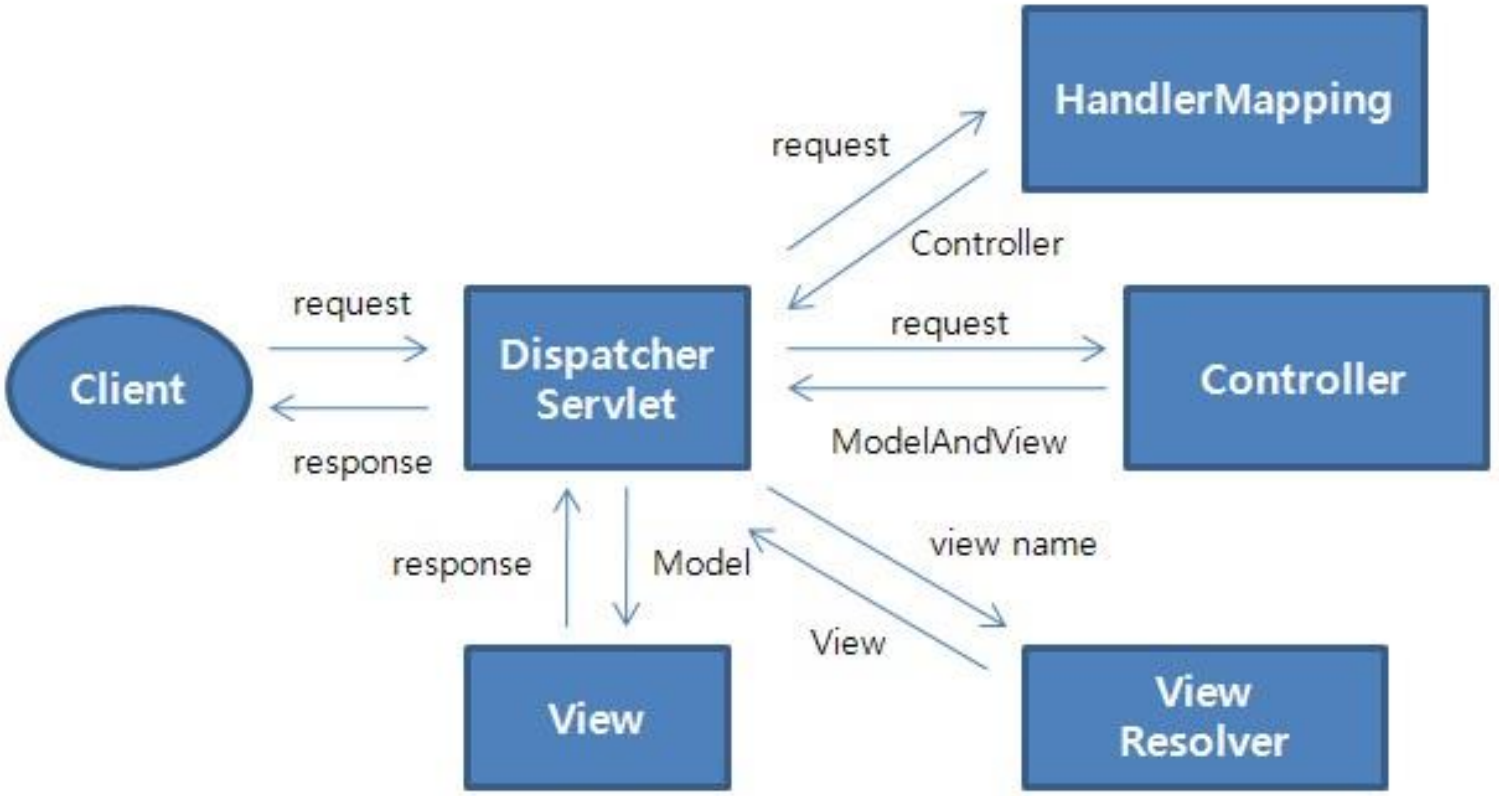
Cons of Servlet

- **Boilerplate Code:** Servlets often require writing significant amounts of boilerplate code for tasks like request parsing, response generation, and session management. This can lead to code duplication and decreased productivity.
- **Low-Level Abstraction**: Servlets provide a low-level API for handling HTTP requests and responses. Developers have to deal with raw servlet API constructs like `HttpServletRequest` and `HttpServletResponse`, which can be verbose and error-prone.
- **Lack of Convention Over Configuration**: Servlets do not offer convention over configuration (CoC) like some higher-level frameworks, such as Spring MVC. Developers need to configure many aspects explicitly, which can be time-consuming.
- **Testing Challenges**: Unit testing servlets can be challenging because you need to mock parts of the Servlet API, such as request and response objects, to simulate interactions. This makes it harder to write clean and reliable unit tests.
- **Lack of Dependency Injection**: Servlets do not provide built-in support for dependency injection (DI) and inversion of control (IoC). Managing dependencies and promoting modular, maintainable code can be more difficult.
- And more…

What is Spring MVC?

- A web framework built on Servlet API
- Cleary defined interfaces for role/responsibilities “beyond” Model-View-Controller
- Single Central Servlet - DispatcherServlet
  - Manages HTTP level request/response
  - delegates to defined interfaces
- Models integrate/communicate with views
  - No need for separate form objects
- Views are plug and play
- Controllers allowed to be HTTP agnostic

Spring MVC Major Interfaces



Spring MVC Flow

The *DispatcherServlet* first receives the request.

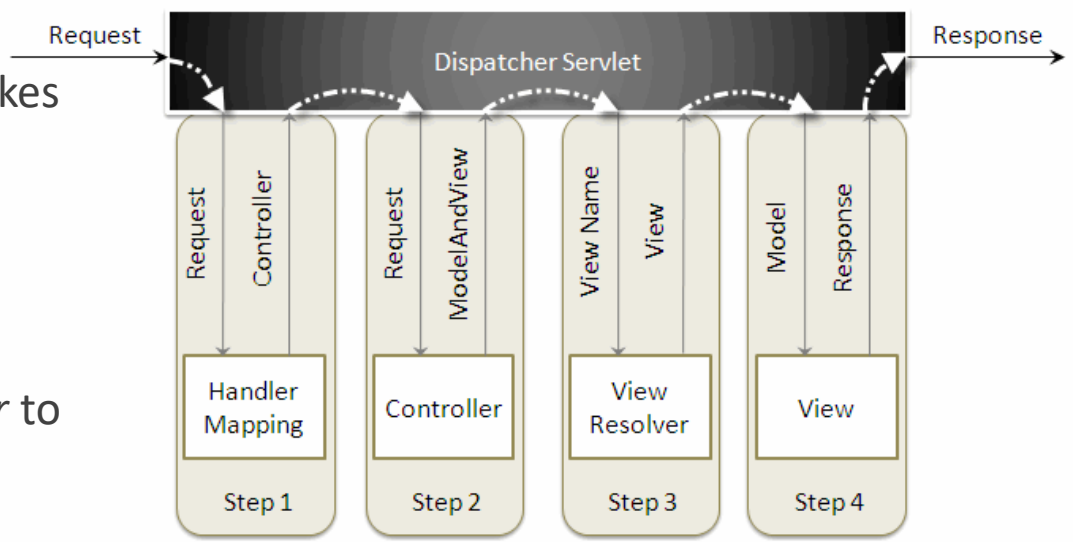
The *DispatcherServlet* consults the *HandlerMapping* and invokes the *Controller* associated with the request.

The *Controller* process the request by calling the appropriate service methods and returns a *ModeAndView* object to the *DispatcherServlet*. The *ModeAndView* object contains the model data and the view name.

The *DispatcherServlet* sends the view name to a *ViewResolver* to find the actual *View* to invoke.

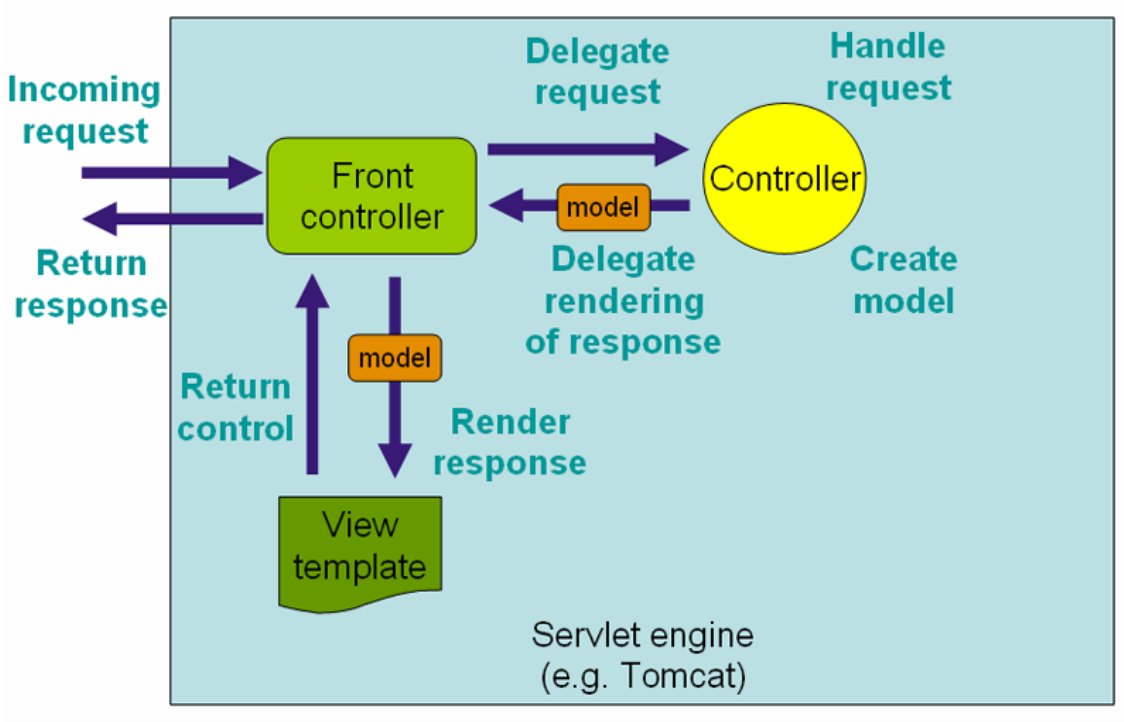
Now the *DispatcherServlet* will pass the model object to the *View* to render the result.

The *View* with the help of the model data will render the result back to the user.



Front Controller - DispatcherServlet

- Spring MVC is designed around a central servlet named DispatcherServlet
- DispatcherServlet receives all of the HTTP requests and delegates them to controller classes.



DispatcherServlet

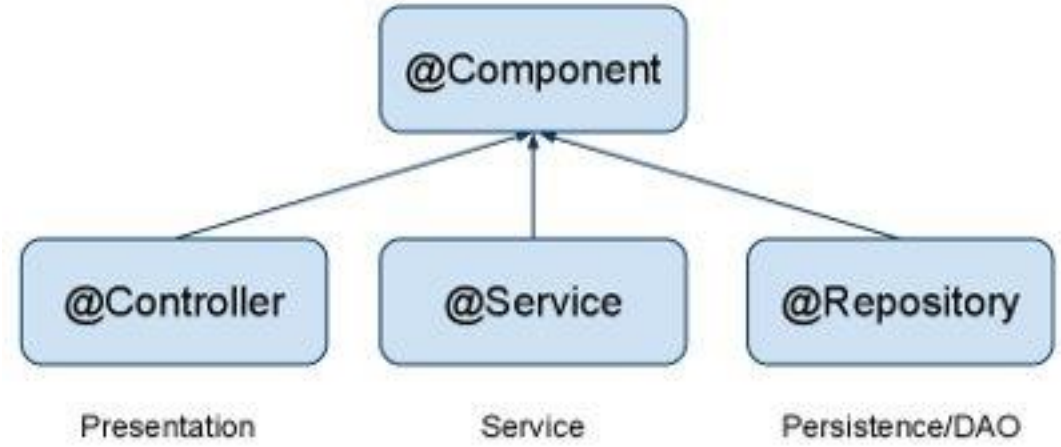
- The Spring Boot autoconfiguration registers and configures the DispatcherServlet automatically. Therefore, we don't need to register the DispatcherServlet manually.
- By default, the spring-boot-starter-web starter configures DispatcherServlet to the URL pattern "/". However, we can customize the URL pattern using `server.servlet.*` in the `application.properties` file:
  - `server.servlet.context-path=/demo`
  - `spring.mvc.servlet.path=/waa`
- With these customizations, DispatcherServlet is configured to handle the URL pattern `/waa` and the root contextPath will be `/demo`. Thus, DispatcherServlet listens at `http://localhost:8080/demo/waa/`

Annotate based on Function

OPTION - annotate all your component classes with `@Component`

Using `@Repository`, `@Service`, and `@Controller` is:

- Better suited for processing by tools
  - `@Controller` – rich set of framework functionality
  - `@Repository` - automatic translation of exceptions
  - `@Service` – "home" of `@Transactional`



Spring MVC Annoations

- `@Controller`
  - applied to classes only
  - used to mark a class as a web request handler
  - used in combination with annotated handler methods based on the @RequestMapping annotation, etc
- `@RequestMapping`
  - with Class: We can use it with class definition to create the base URI
  - with Method: We can use it with method to provide the URI pattern for which handler method will be used.
- `@GetMapping`: `@RequestMapping(value = "", method = RequestMethod.GET)`
- `@PostMapping`: `@RequestMapping(value = "", method = RequestMethod.POST)`
- `@PutMapping`: `@RequestMapping(value = "", method = RequestMethod.PUT)`
- `@DeleteMapping`: `@RequestMapping(value = "", method = RequestMethod.DELETE)`
- `@RequestParam`: extract query parameters, form parameters, and even files from the request
- `@PathVariable`: used to handle template variables in the request URI mapping

Spring MVC `@Controller`

Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

Spring Controllers do not extend specific base classes or implement specific interfaces

They do not have direct dependencies on Servlet APIs, although you can easily configure access to Servlet facilities. [actual request, response objects, etc.]

@Controller Annotation Example

```
@RequestMapping("/product")
@Controller
public class ProductController {

    @RequestMapping("/index")
    public String index(){
        return "index";
    }

    @GetMapping("/index2")
    public String index2(){
        return "index";
    }

    @RequestMapping(value= {"/index3", "/hello"})
    public String index3() {
        return "index";
    }

}
```

What is Thymeleaf?

A template engine for Java

- Open Source
- First Version 2011

It can process six kinds of templates: HTML, XML, TEXT, etc.

Usually view layer in Spring MVC

What does it look like?

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price,1,2}">0.99</td>
    </tr>
  </tbody>
</table>
```

Why Thymeleaf?

The design-development roundtrip issue

Natural Templates: templates can be documents as valid as the final result, the engine syntax doesn’t break the document structure. – Comparison of web template engines [Wikipedia]

```
<form:inputText name="userName" value="${user.name}" />
```

```
<input type="text" name="userName" value="James Carrot" th:value="${user.name}" />
```

The trick

- Use non-standard attributes

```
<ul>
  <li th:each="f : ${fruits}" th:text="${f.name}">Apricot</li>
</ul>
```

- Browsers ignore them
- HTML5 allows custom attributes

```
<ul>
  <li data-th-each="f : ${fruits}" data-th-text="${f.name}">Apricot</li>
</ul>
```

- They don’t have influence on the DOM!
- XML namspace
  - which has no influence at all in template processing, but works as an incantation that prevents our IDE from complaining about the lack of a namespace definition for all those th:* attributes.

```
<html xmlns:th="http://www.thymeleaf.org">
```

@RequestParam

- extract query parameters, form parameters, and even files from the request

```
@GetMapping("/getProduct")
public String getProduct(@RequestParam String id, @RequestParam(value = "title", required = false) String productTitle){
    System.out.print("id: " + id + ", title: " + productTitle);
    return "index";
}
```

- Request URLs
  - http://localhost:8080/product/getProduct?id=p1111 - `id: p1111, title: null`
  - http://localhost:8080/product/getProduct?id=p1111&title=laptop - `id: p1111, title: laptop`

@PathVariable

- used to handle template variables in the request URI mapping

```
@RequestMapping("/posts")
@Controller
public class PostController {

    @GetMapping("/{postId}/comments/{commentId}")
    public String getComment(@PathVariable("postId") String pid, @PathVariable String
commentId){
        System.out.println("Post Id: " + pid + ", Comment Id: " + commentId);
        return "index";
    }
}
```

- Request URL
  - http://localhost:8080/posts/1234/comments/c999 - `Post Id: 1234, Comment Id: c999`

Spring Container

Using IOC a container (like spring) can provide:

- Dependency Injection
- Aspect Oriented Programming

In essence, spring is a fancy factory that:

- reads a config file, creates objects, and connects them

Understanding Spring & DI

4 ways connect objects together

- Instantiate Objects Directly
- P2I for more flexibility, but still instantiate
- Use a factory object to instantiate
- Use Spring and DI

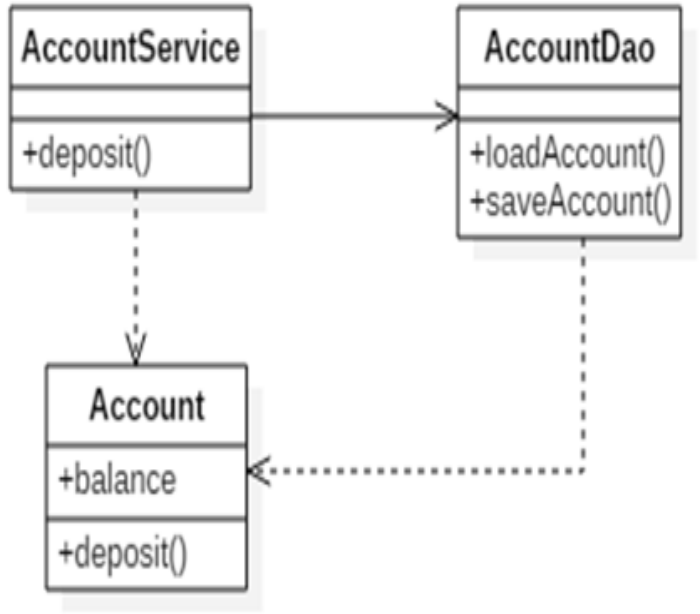
Instantiate Objects Directly

```
public class AccountService {
  private AccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Hardcoded



Relation between AccountService and AccountDao is hardcoded. To Change AccountDao implementation have to change the code

Use an Interface

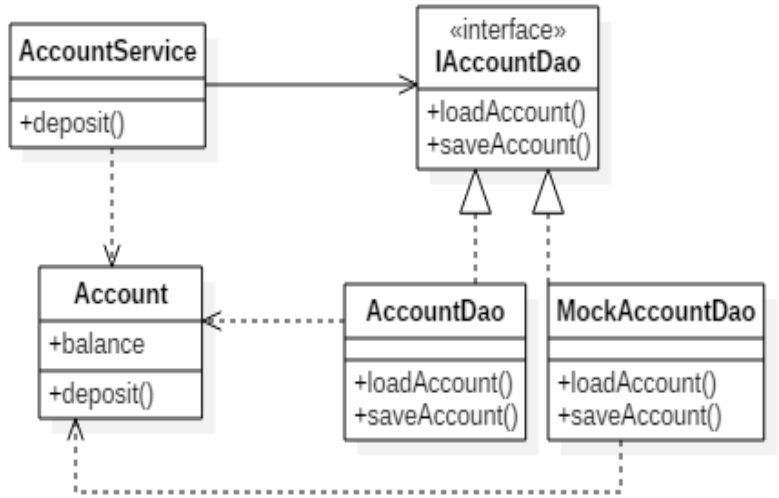
Flexibility

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Hardcoded



- By using an interface (P2I) we've gained the flexibility (two implementations)
  - But the relationship is still hard coded
  - To switch, we still have to change code

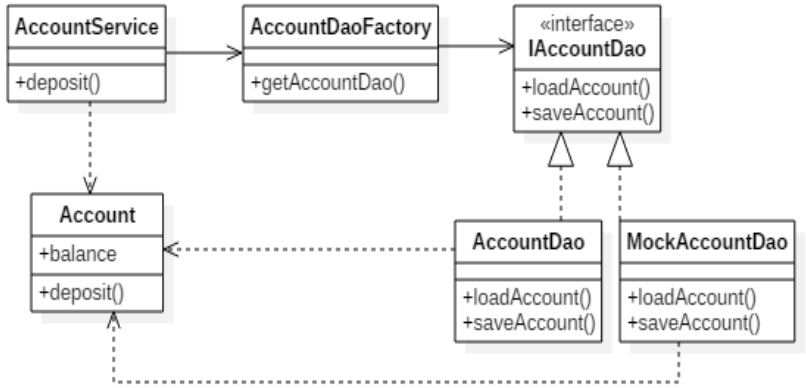
Use a Factory

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    AccountDAOFactory daoFactory = new AccountDAOFactory();
    accountDAO = daoFactory.getAccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Hardcoded



- The relation between AccountService and AccountDao is still hardcoded
  - We have more flexibility, but if you want to change the AccountDao implementation you have to change the code in the AccountDaoFactory

Spring Dependency Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Setter for Injection

What if the Factory created both Account Service and the AccountDao?

What if the Factory could read a config file?

That's in essence what Spring does.

Config for creating and Injecting

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

Dependency Injection

Dependency Injection is used to gain greater flexibility, loose coupling. Possible to change the connections without chancing the code.

- Code is clean, associations not hard-coded (no `new`)
- Unit testing becomes easier (inject mock)
- AOP / Interceptors become possible (inject proxy)

Dependency Injection [DI]

DI exists in three major variants

Dependencies defined through:

- Property-based dependency injection.
- Setter-based dependency injection.
- Constructor-based dependency injection

Container *injects* dependencies when it creates the bean.

Dependency Injection examples

- **Property based[byType]:**

```
@Autowired
ProductService productService;
```

- **Setter based[byName]:**

```
ProductService productService;
@Autowired
public void setProductService(ProductService productService){
        this.productService = productService;
}
```

- **Constructor based:**

```
ProductService productService;
@Autowired
public ProductController(ProductService productService) {
        this.productService = productService;
```

Main Points

The basic ingredients of a Spring MVC application include web pages for the view (the known), the back end domain logic and data (knower or underlying intelligence), and the Spring framework and Dispatcher Servlet and managed beans as the controller to connect the view and model.

Knowledge is the wholeness of knower, known, and process of knowing.