CS 416
Web Programming

Spring Framework

Dr. Williams
Central Connecticut State University

The Spring Framework

- Open source
- Core features designed to be used for developing any Java application
- Extensions specifically for making J2EE development easier
- Key role of framework is to promote good programming practices such as division of responsibility like MVC

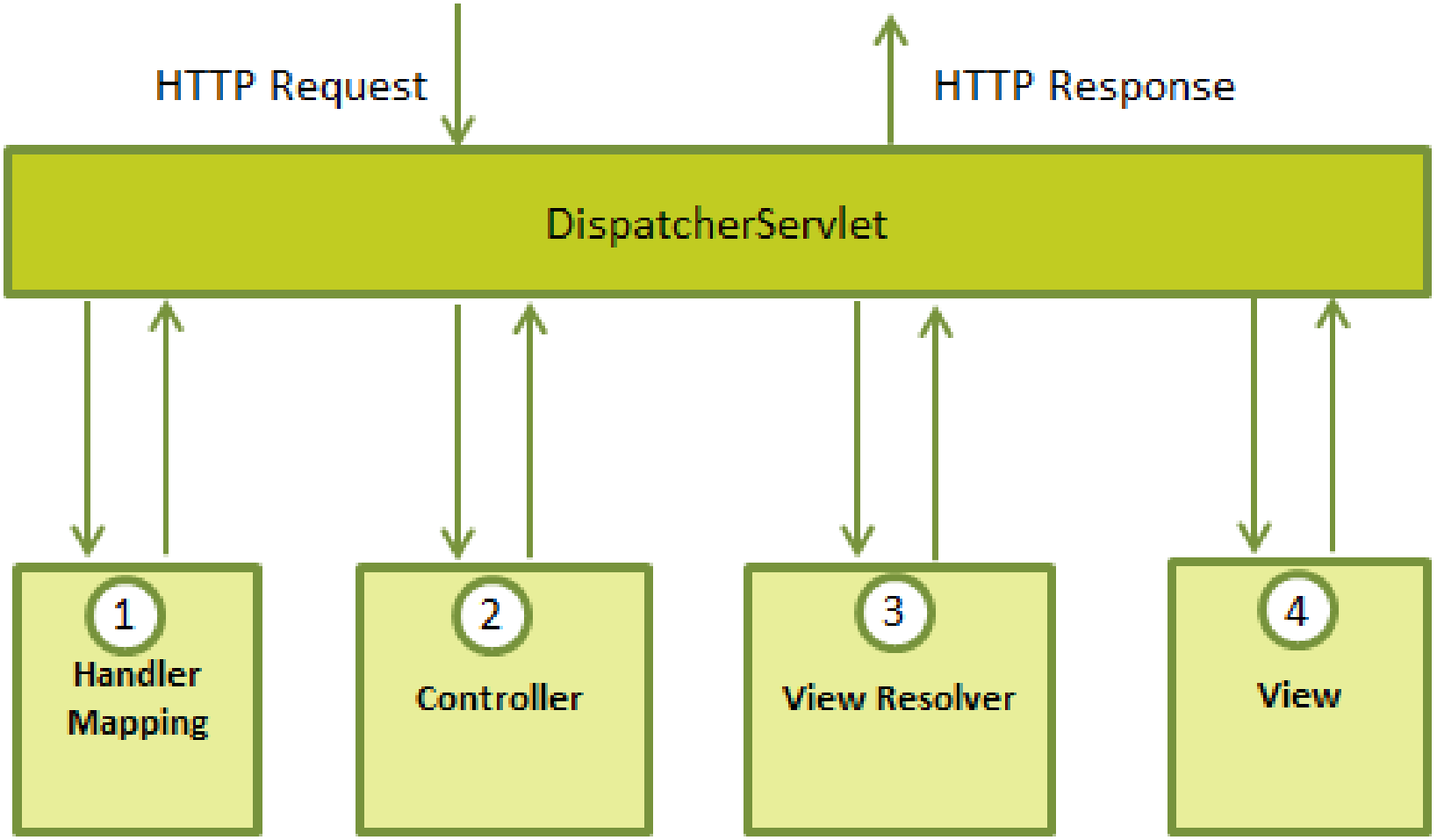
Benefits

- Designed to be inclusive such that you can bring in multiple different technologies rather than prescriptive restricting to specific implementation
- Intended to be simplified solution to avoid over engineering required of some frameworks for common simple scenarios (specifically Struts)
- Lighter weight than most frameworks in terms of demand on server

Spring web modules

- Web
  - File upload, listeners, context
- Web-MVC
  - Model-view-controller framework
- Web-socket
  - APIs for 2-way communication
- Web-portlet
  - MVC implementation for portlet environment

Spring MVC Framework



DispatcherServlet configured in XML (web.xml)

```
<servlet>
  <servlet-name>HelloWeb</servlet-name>
  <servlet-class>
org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>HelloWeb</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Servlet configuration

- With Spring it looks for config file to be in [servlet-name]-servlet.xml in the WEB-INF directory:
  - WEB-INF/HelloWeb-servlet.xml

HelloWeb-servlet.xml

```
<beans
xmlns="http://www.springframework.org/schema/beans"
... />

  <context:component-scan base-package="edu.ccsu.cs416" />

  <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
  </bean>
</beans>
```

Package to scan for relevant @Controller and @RequestMapping annotations

Specifies logical name of “hello” maps to /WEB-INF/jsp/hello.jsp

Defining a Controller

- DispatcherServlet delegates request to controller with matching RequestMapping
- Two options for registering controller:

1) Annotate class as controller and map request path to controller class itself then specific request method (GET/POST) to specific methods

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

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

Defining Controller cont.

2) Annotate class as controller and map request path to individual control methods can specify request method (GET/POST) to specific methods as well:

```
@Controller
public class StudentController {

@RequestMapping(value="/student",
                    method=RequestMethod.GET)
public ModelAndView student(){…}

@RequestMapping(value="/addStudent",
                   method=RequestMethod.POST)
public String addStudent(…){…}
```

Controller cont.

- Within controller method
  - Call relevant method(s) to perform business logic
  - Populate model
    - Beans
    - Spring Framework ModelMap (hashmap)
  - Can return a string which contains the name of the **view** to render

Views

- With Spring can use pretty much anything as your view, but most typically JSP. Within view model elements set in controller accessed as attributes

```
<h1>${message}</h1>
```

Working with forms

- On controller directing to form add bean to be bound to the returned model
  - Remember Model is essentially Map<String,Object>

```
@RequestMapping(value = "/users/add",
                method = RequestMethod.GET)
public String showAddUserForm(Model model)
{
  User user = new User();
  model.addAttribute("userForm", user);
  //...
}
```

On form use spring form tag to bind form to object

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


<form:form method="post" modelAttribute="userForm"
     action="${userActionUrl}">
  <form:input path="name" type="text" />
  <!-- binds to user.name-->
  <form:errors path="name" />
</form:form>
```

Getting bound form data

```
@RequestMapping(value = "/users",  method =
RequestMethod.POST)
public String saveOrUpdateUser(
     @ModelAttribute("userForm") User user,
     BindingResult result, Model model) {
   if (result.hasErrors()) {
     return "users/userform";
   } else {
     String myName = user.getName();
     ...
```

Spring form elements

- Text input box
  - `<form:input path="firstName" />`
- Checkbox
  - Expects bean:
    - `setInterests(String[] interests)`
    - `setInterests(Collection interests)`

  Two Options
  - `<form:checkbox path="preferences.interests" value="Quidditch"/>`
  - `<form:checkboxes path="preferences.interests" items="${interestsList}"/>`

Spring form elements cont.

- Radio button
  - Expects bean:
    - `setSex(String value)`

    Two Options
  - `<form:radiobutton path="user.sex" value="M"/>`
  - `<form:radiobuttons path="user.sex" items="${sexOptions}"/>`
- `Password`
  - `<form:password path="password" />`

Spring form elements cont.

- Select
  - Expects bean:
    - `setSkill(String value)`
    - `setSkill(String[] values)`

    Two Options
  - `<form:select path="user.skills" items="${skillOptions}/>`
  - `<form:select path="user.skills">`
  - `  <form:option value="-" label="—Please select" />`
  - `  <form:option value="dancing" selected="selected" />`
  - `  <form:options items="${skills}" />`

Spring form elements cont.

- Text area
  - Expects bean:
    - `setNotes(String value)`
  - `<form:textarea path="notes" rows="3" cols="20"/>`

Validation showing errors

```
<form:form>
  First Name:<form:input path="firstName" />
  <%-- Show errors for firstName field --%>
  <form:errors path="firstName" />

Or

 <%-- Show errors for all fields --%>
<form:errors path="*" />
```

Validation

```
public class UserValidator implements Validator {
  public boolean supports(Class candidate) {
    return User.class.isAssignableFrom(candidate);
  }

  public void validate(Object obj, Errors errors) {
    ValidationUtils.rejectIfEmptyOrWhitespace(errors,
      "firstName", "required", "Field is required.");
    if(user.getNumber()==null || user.getNumber()<=0){
      errors.rejectValue("number", "NotEmpty.userForm.number");
    }
  }
}
```

Validation messages

`validation.properties` file

```
NotEmpty.userForm.password = Password is required!
NotEmpty.userForm.sex = Sex is required!
NotEmpty.userForm.number = Number is required!
Pattern.userForm.email = Invalid Email format!
```

Running validation - option 1

- Add the validator to the controller via `@InitBinder` and annotate model with `@Validated`

```
@InitBinder
protected void initBinder(WebDataBinder binder)
{
  binder.setValidator(userFormValidator);
}

@RequestMapping(value = "/users", method =
RequestMethod.POST)
public String saveOrUpdateUser(... @Validated User user,
     ...) {
     //...
}
```

Running validation - option 2

- Add the validator manually

```
@RequestMapping(value = "/users", method =
RequestMethod.POST)
public String saveOrUpdateUser(... User user,
      ...) {
   userFormValidator.validate(user, result);
   //...
}
```

Services

- Spring's approach business logic and data access off of beans, off of controller **@Service**

```
@Service("userService")
public class UserServiceImpl implements UserService {
  CustomerRepository customerRepository;

  @Autowired
  public void setCustomerRepository(
              CustomerRepository customerRepository) {
      this.customerRepository = customerRepository;
  }
  @Override
  public List<Customer> findAll() {
    return customerRepository.findAll();
  }
```

```
@Controller
public class UserController {
  @Autowired
  private UserService userService;
}
```

Data access - Repository

- Creates majority of JPA code automatically simply by specifying interface
  - Automatically inherits methods for saving, deleting, finding
    - save()
    - findAll()
    - findOne()
  - Define queries on interface, code generated underneath

```
import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends
      CrudRepository<Customer, Long> {

  List<Customer> findByLastName(String lastName);
}
```

MVC design revisited - Spring(simple ex)

- Commerce site is developing some basic functionality. User accesses the site (only consider new user scenario), creates their profile, data saved. Presented with list of all products with checkboxes, user selects checkboxes submits. Next page presents read only version, place order. Order placed, saved in DB and presented confirmation page
  - Identify model
  - Identify business logic
  - Identify views
  - Identify control rules
  - Tie elements together