Spring MVC Validation

Avoid the Danger that has not yet come

Spring Validation

- Validation:
  - should not be tied to the web tier
  - should be easy to localize
  - should be possible to plug in any validator available

- Spring Validation uses a Validator interface that is basic and usable in every layer of an application. Additionally an application can use the Spring Validator directly without the use of annotations.
- An application can choose to enable Bean Validation (JSR-303) and the corresponding annotations for all validation needs.

Form Validation Intro

- To do simple validation, use `javax.validation.constraints` annotations (also known as JSR-303 annotations).
- JSR-303 is also know as the Bean Validation API
- JSR-303 provider library, e.g., Hibernate-Validator.jar

Form Validation through Annotation

It's for Strings and collections.

Step 1: Annotate domain model properties

```java
public class Employee {
    private Long id;

    @NotBlank // any characters besides "space"
    @Size(min = 4, max = 50, message = "{Size.name.validation}")
    private String firstName;

    @NotBlank(message = "Enter the last name")
    private String lastName;

    @NotNull
    @Past
    @DateTimeFormat(pattern = "MM-dd-yyyy")
    private Date birthDate;

    @NotNull
    private Integer salaryLevel;

    @Valid
    private Address address;

    public void setFirstName(String firstName) {
        this.firstName = firstName.trim();
    }
}
```

use for Objects

```java
public class Address {

    @NotEmpty(message = "{String.empty}")
    private String street;
    private String city;

    @Size(min = 2, max = 2, message =
        "Size.state")
    private String state;
}
```

Note: Curly {} brackets ensure that the text will be used as a property file lookup

Validation Property Annotations [JSR-303]

Constraint	Description	Example
@AssertFalse	The value of the field or property must be false.	@AssertFalse boolean isUnsupported;
@AssertTrue	The value of the field or property must be true.	@AssertTrue boolean isActive;
@Email	The string has to be a well-formed email address	@Email String email;
@Digits	The value of the field or property must be a number within a specified range.	@Digits(integer=6, fraction=2) BigDecimal price;
@Future	The value of the field or property must be a date in the future.	@Future Date eventDate;
@Max	The value of the field or property must be an integer >= the value.	@Max(10) int quantity;
@Min	The value of the field or property must be an integer <= the value.	@Min(5) int quantity;
@NotNull	The value of the field or property must not be null.	@NotNull String username;
@Null	The value of the field or property must be null.	@Null String unusedString;
@Past	The value of the field or property must be a date in the past.	@Past Date birthday;
@Pattern	The value of the field or property must match the regular expression defined in the regexp element.	@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;
@Size	The size of the field or property is evaluated and must match the specified boundaries. Can pertain to String, Collection, Map…	@Size(min=2, max=240) String briefMessage;

Hibernate JSR 303 Annotations

Form Validation through Annotation (cont.)

- Step 2: Externalize error messages in properties file

```
typeMismatch.java.lang.Integer={0} must be an integer
typeMismatch.java.util.Date={0} is an invalid date. Use format
   MM-DD-YYYY.

NotNull={0} is a required field
NotEmpty={0} field must have a value
Size.name.validation =Size of the {0} must be between {2} and {1}
Pattern.zipcode={0} is incorrect. Use format nnnnn-nnnn

address.zipCode=Zip Code
```

- Spring organizes "placeholders" in alphabetical order. `@Size(min=1,max=5)`, field name as **`{0}`** , the max value as **`{1}`** , and the min value as **`{2}`**.

Form Validation through Annotation (cont.)

- Step 3: Annotate model to be validated in the Controller method signature with @Valid:

```
@RequestMapping(value = "/employee_save")
public String saveEmployee(@Valid @ModelAttribute("employee")
   Employee employee, BindingResult bindingResult,
Model model) {

   if (bindingResult.hasErrors()) {
     return "EmployeeForm";
   }

   // save product here
   model.addAttribute("employee", employee);

   return "EmployeeDetails";
}
```

BindingResult IMMEDIATELY after model attribute

From Validation through Annotation (cont.)

- Step 4: Display error in View

```
<form:form commandName="employee" method="post">
  <p>
    <form:errors path="*" cssStyle="color : red;" />
  </p>
  <p>
    <label for="id">ID: </label>
    <form:input path="id" id="id" />
    <div style="text-align: center;">
      <form:errors path="id" cssStyle="color : red;" />
    </div>
  </p>
</form:form>
```

Show ALL errors on Page

Show field level error

From Validation through Annotation (cont.)

- Step 5: External error message and Validation configuration (XML version)

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSou
   rce">
   <property name="basename" value="classpath:errorMessages" />
</bean>

<bean id="validator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBea
   n">
   <property name="validationMessageSource" ref="messageSource" />
</bean>

<mvc:annotation-driven validator="validator" />
```

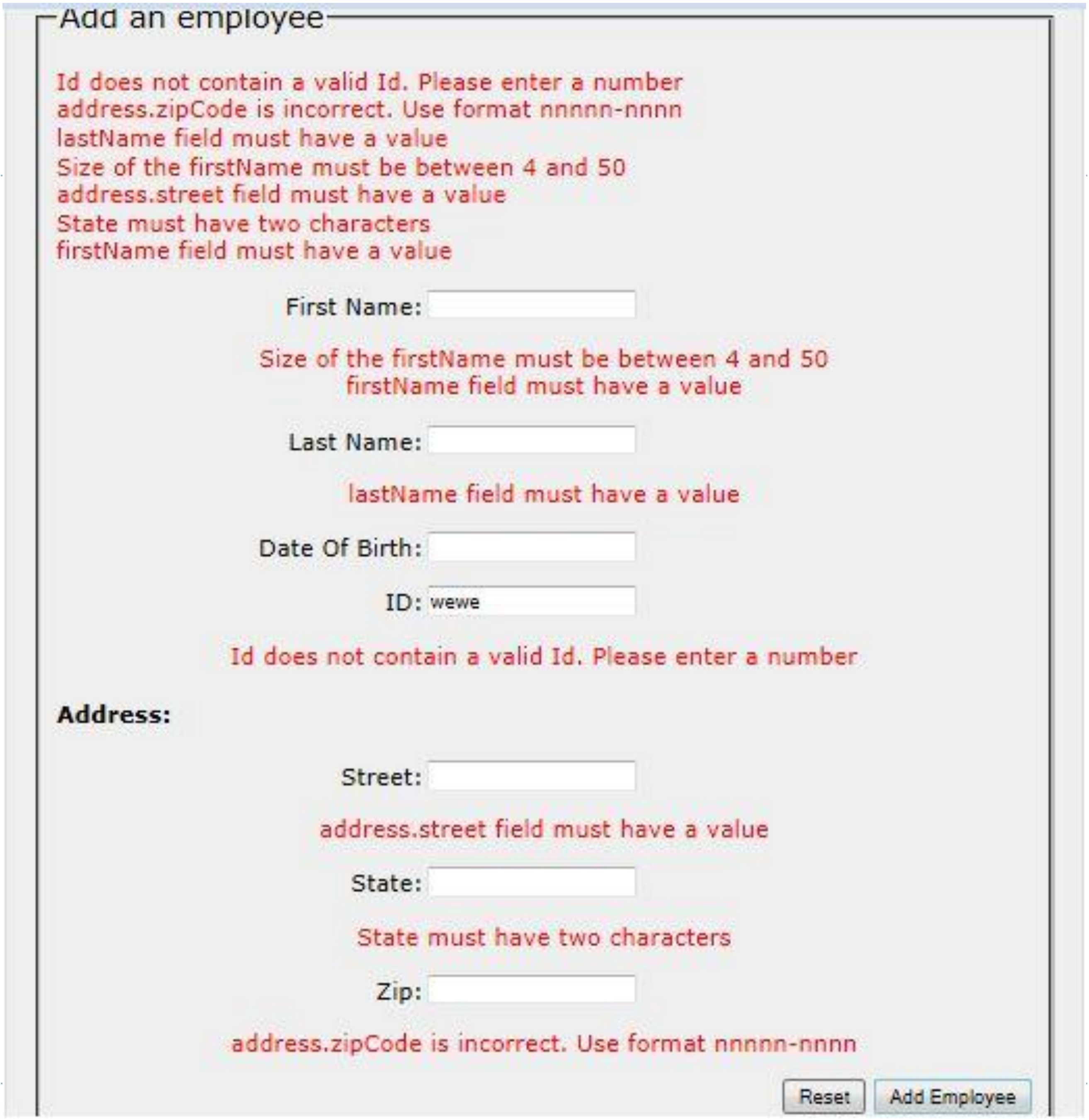
From Validation through Annotation (cont.)

- Step 5: External error message and Validation configuration (Java Config version) – WebApplicationContextConfig.java

```
@Bean
public MessageSource messageSource() {
   ResourceBundleMessageSource resource = new ResourceBundleMessageSource();
   // resource.setBasenames("messages");
   resource.setBasenames("messages", "errorMessages");
   return resource;
}

@Bean(name="validator")
public LocalValidatorFactoryBean validator() {
   LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
   bean.setValidationMessageSource(messageSource());
   return bean;
}

@Override
public Validator getValidator() {
   return validator();
}
```



Add an employee

Id does not contain a valid Id. Please enter a number
address.zipCode is incorrect. Use format nnnnn-nnnn
lastName field must have a value
Size of the firstName must be between 4 and 50
address.street field must have a value
State must have two characters
firstName field must have a value

First Name:

Size of the firstName must be between 4 and 50
firstName field must have a value

Last Name:

lastName field must have a value

Date Of Birth:

ID: wewe

Id does not contain a valid Id. Please enter a number

Address:

Street:

address.street field must have a value

State:

State must have two characters

Zip:

address.zipCode is incorrect. Use format nnnnn-nnnn

Typemismatch

- Non-String – if value cannot be converted to the data-type then an Exception is thrown.
- Define the error message for type mismatch [e.g.]:

```
typeMismatch.java.lang.Integer="{0}" must be an
integer.
typeMismatch.java.lang.Double="{0}" must be a double.
typeMismatch.java.lang.Long="{0}" must be a long.
typeMismatch.java.util.Date="{0}" is not a date.
```

- Field Specific:

```
typeMismatch.id= Id is not valid. Please enter a number
```

Main Point

- Validation checks the correctness of data against business rules. This prevents problems in the business model from arising.
- *In Cosmic Consciousness, life is lived stress-free; problem-free.*

Manual Validation [W/O Annotations]

- ObjectValidator implements Validator interface.

```
public class MemberValidator implements Validator {
    @Override
    public boolean supports(Class<?> c) {
        return Member.class.isAssignableFrom(c);
    }

    @Override
    public void validate(Object command, Errors errors) {
        String[] errorArgs = { "First Name" };
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "NotEmpty", errorArgs);
        errorArgs = new String[] { "Last Name" };
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "LastName", "NotEmpty", errorArgs);
        Member member = (Member) command;

        if (member.getMemberNumber() == null || member.getMemberNumber() <= 0)
           errors.rejectValue("memberNumber", "Member.Number.lessthan");
        if (member.getAge() < 18)
           errors.rejectValue("age", "Member.age");
    }

}
```

Manual Validation (cont.)

- InitBinder setting of validator can be used with `@Valid`

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

- 100% Manual Does NOT use `@Valid`; Looks like this:

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String processAddNewMemberForm(@ModelAttribute("newMember") Member
   memberToBeAdded, BindingResult result) {

    MemberValidator memberValidator = new MemberValidator();
    memberValidator.validate(memberToBeAdded, result);

    if (result.hasErrors()) {
    return "addMember";
    }

    memberService.save(memberToBeAdded);
    return "redirect:/members";
}
```

Custom Validation Annotation

- The annotation implementation must conform to Bean Validation API [JSR 303]

- There are three steps that are required:
  1. Define a default error message
  2. Create a constraint annotation
  3. Implement a validator

Step 1: Define Default Error Message

- Put messages in errorMessages.properties file

```
com.packt.webstore.validator.ProductId.message = A
    product already exists with this product id.
```

Step 2: Create the annotation

- @Target Indicates the kinds of program element to which an annotation type is applicable.
- @Retention Indicates how long annotations with the annotated type are to be retained.
- @Constraint Specifies the validator to be used.

```
@Target({ ElementType.METHOD, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = ProductIdValidator.class)
public @interface ProductId {

   String message() default
      "{com.packt.webstore.validator.ProductId.message}";

   Class<?>[] groups() default {};

   public abstract Class<? extends Payload>[] payload() default {};
```

Identifies the default key for creating error messages

Groups are typically used to control the order in which constraints are evaluated, or to perform validation of the partial state of a JavaBean.

Payloads are typically used by validation clients to associate some metadata information with a given constraint declaration.

Step 3: Implement Validator

Annotation & Type to be validated

```
public class ProductIdValidator implements ConstraintValidator<ProductId, String> {
    @Autowired
    private ProductService productService;
    @Override
    public void initialize(ProductId arg0) {}

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        Product product = null;
        try {
            product = productService.getProductById(value);
        } catch (Exception e) {
            System.out.println("Couldn't find product...");
        }
        return product == null ? true : false;
    }
}
```

add additional error messages or completely disable the default error message

- **Usage:**

```
@Pattern(regexp = "P[1-9]+", message = "{Pattern.Product.productId.validation}")
@ProductId
private String productId;
```

Cross Field Validation

- NEED: validate the combination of two or more fields
- Similar to field level Validator BUT different
- Class Level…Validation against entire Class object

```
public class StockMaximumValidator implements ConstraintValidator<StockMaximum, Product>{
    BigDecimal maxValue = null;

    public void initialize(StockMaximum constraintAnnotation) {
        int maximum = constraintAnnotation.maximum();
        maxValue = new BigDecimal(maximum);
    }

    @Override
    public boolean isValid(Product product, final ConstraintValidatorContext context) {
        BigDecimal unitPrice = product.getUnitsInStock();
        Long unitsInStock = product.getUnitPrice();

        BigDecimal currentValue = new BigDecimal(0);
        if (unitsInStock > 0) {
            currentValue = unitPrice.multiply(new BigDecimal(unitsInStock));
        }
        if (currentValue.compareTo(maxValue) >= 0) return false;
        return true;
    }
}
```

See demo: webstore_validation

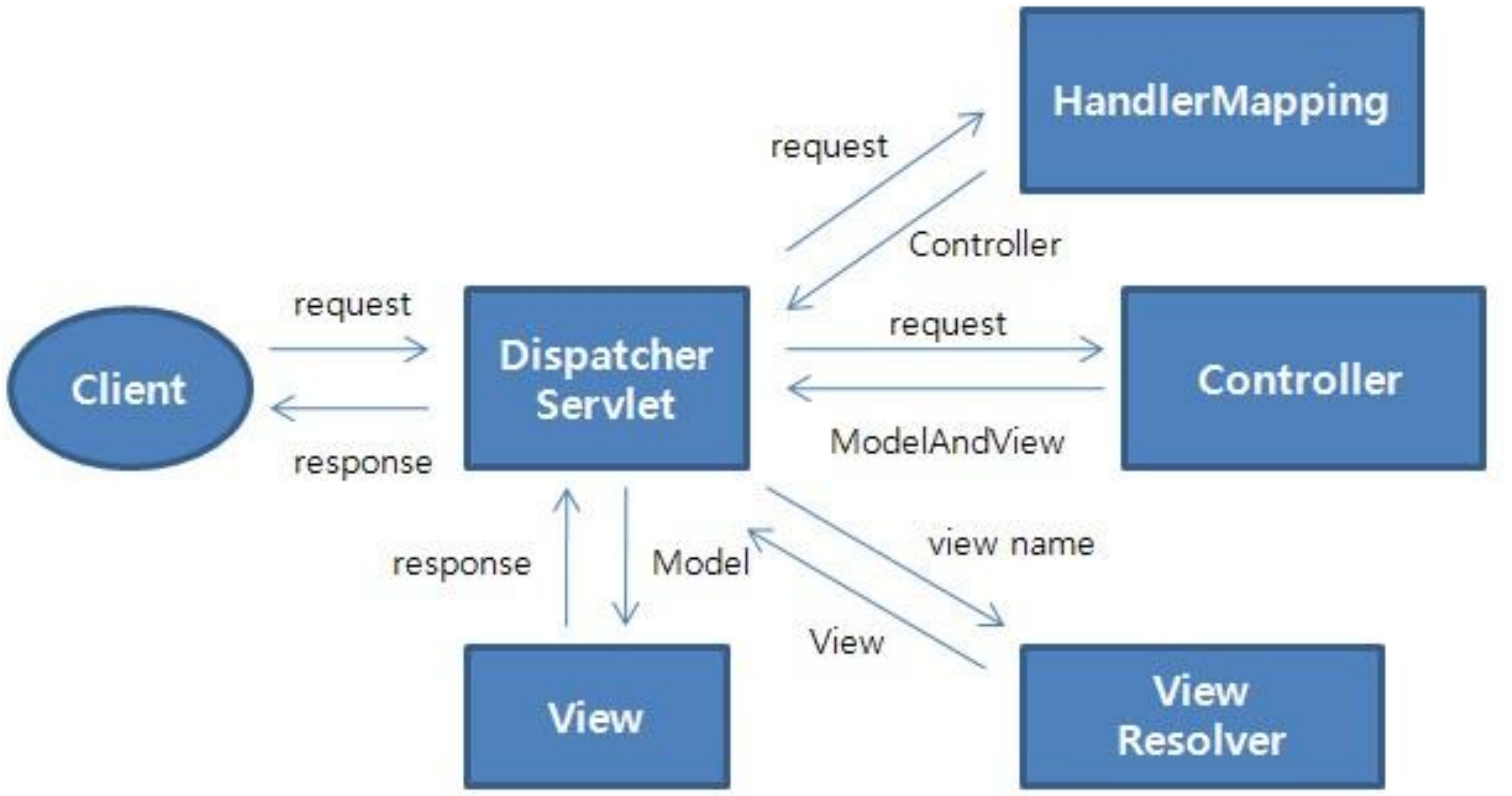
Main Point

- Custom validation allows for handling more complex, extraordinary verification issues.
- *A quality of Cosmic Consciousness is the ability to know what is true and right in every situation.*

Spring MVC Architecture & Annotations

- ***Handler Mapping***
- Spring Annotations
  - Spring Managed Components:
    - @Controller Indicates a Controller component in the presentation layer.
    - @Service Indicates a Service component in the business layer
    - @Repository Indicates DAO component in the persistence layer.
  - @RequestMapping
  - @RequestParam
  - @ModelAttribute
  - @PathVariable
- ViewResolvers
- Views

Spring MVC Flow



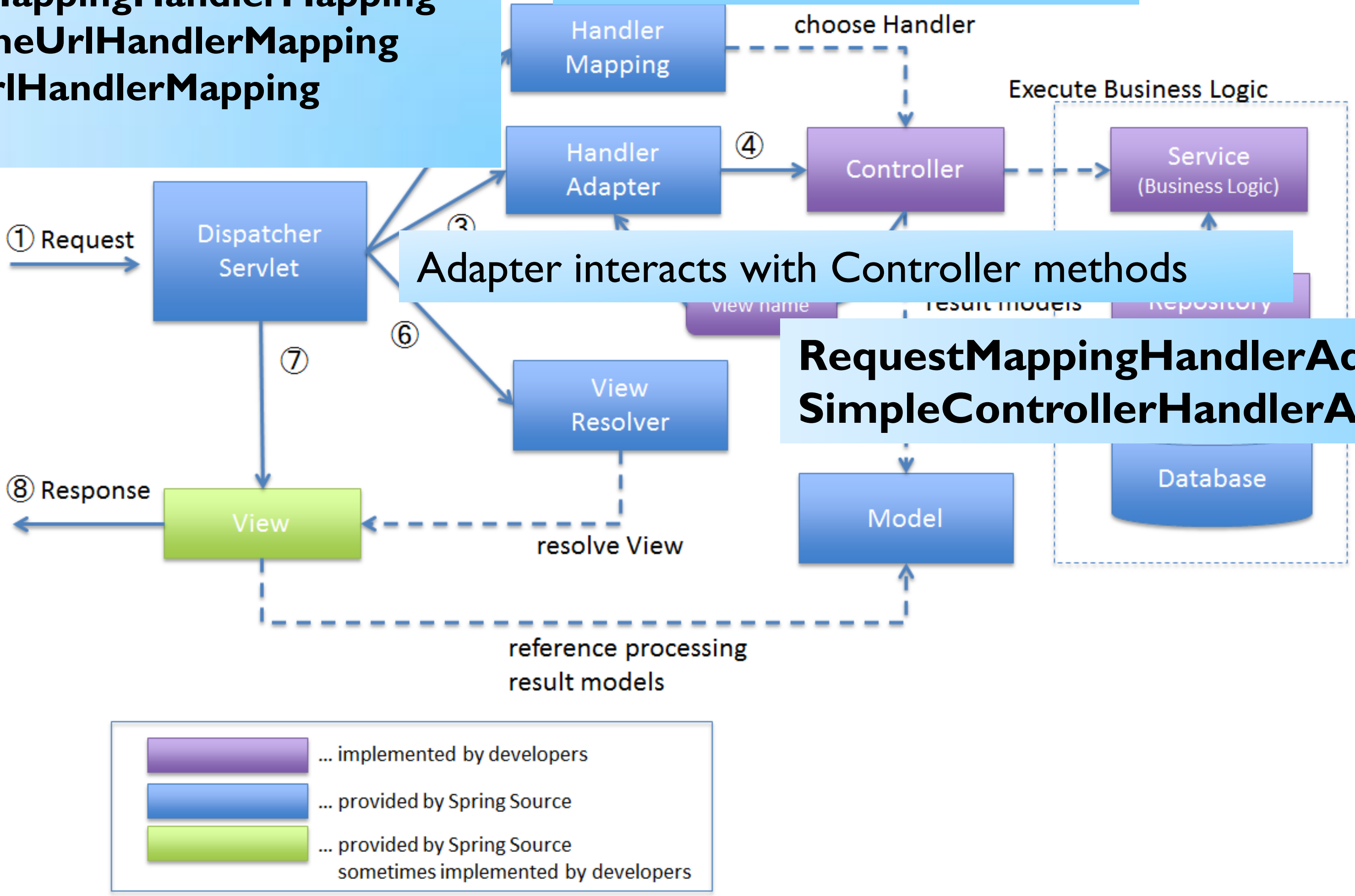
Spring MVC Flow More Details

RequestMappingHandlerMapping
BeanNameUrlHandlerMapping
SimpleUrlHandlerMapping
…

Mapping chooses Controller

Adapter interacts with Controller methods

RequestMappingHandlerAdapter
SimpleControllerHandlerAdapter



Handler Mapping

- Using a handler mapping you can map incoming web requests to appropriate handlers.
- When a request comes in, the DispatcherServlet will hand it over to the handler mapping to let it inspect the request and come up with an appropriate HandlerExecutionChain.

HandlerMapping

- The Handler Mapping is used to map a request from the Client to its Controller object by searching through the various Controllers.

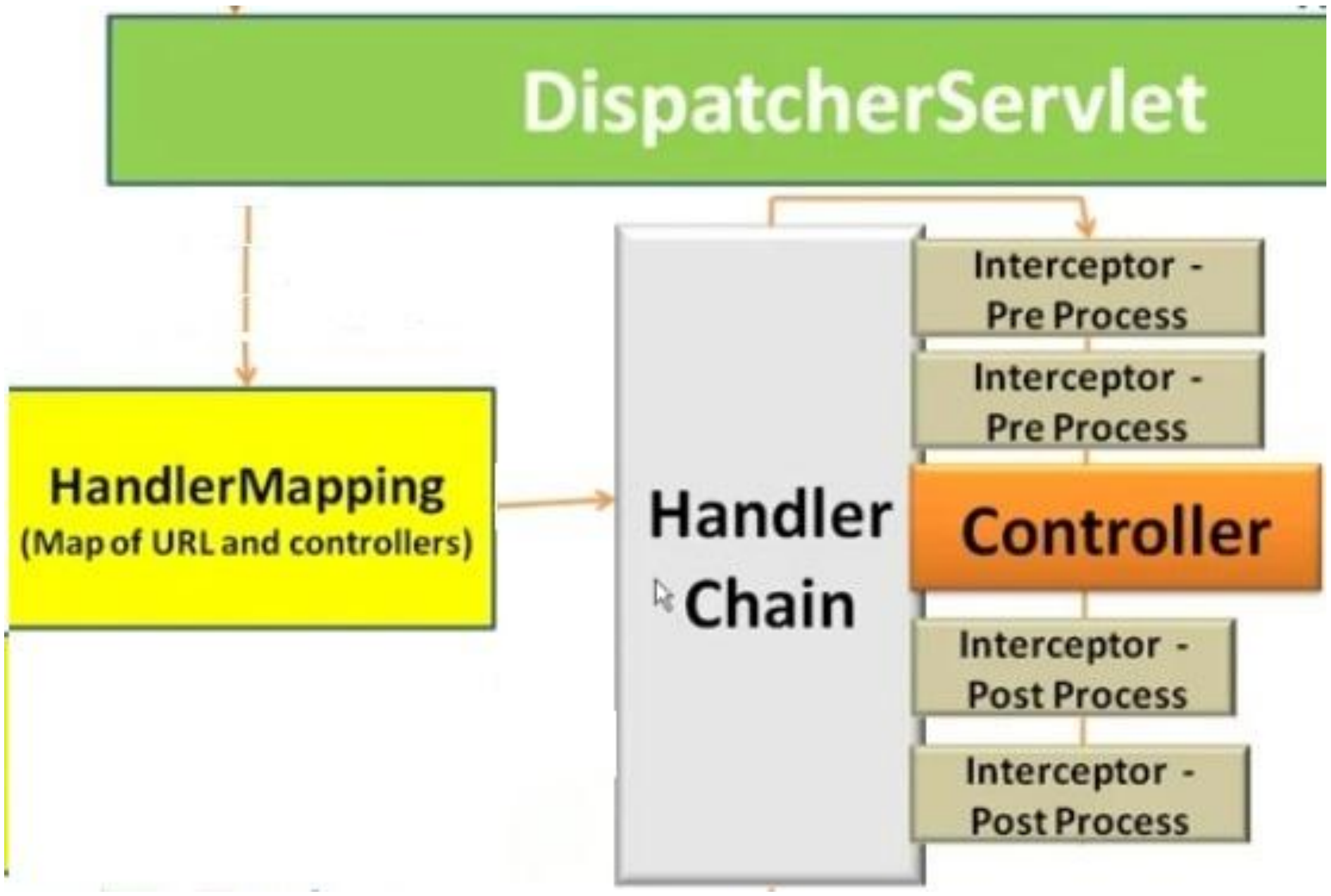
- **BeanNameUrlHandlerMapping** *******default******
  - The URL of the Client is directly mapped to the Controller
  - <bean name="/ProductForm.do" class="edu.mum.controller.InputProductController"/>

- **RequestMappingHandlerMapping** *******default******
  - Maps handlers through the RequestMapping annotation at the type or method level.

- **ControllerClassNameHandlerMapping**
  - <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />
  - <bean class="edu.mum.controller.WelcomeController" />
  - WelcomeController maps to the '/**welcome***' URL based on naming

- **SimpleUrlHandlerMapping**
  - Keys defined on bean definition:
  - <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    - <property name="mappings"> <props> <prop key="/welcome.htm">welcomeController</prop> <</props> </property>
  - </bean>
  - <bean id="welcomeController" class="com.mkyong.common.controller.WelcomeController" />

Handler Chaining


DispatcherServlet
HandlerMapping
(Map of URL and controllers)
Handler Chain
Interceptor - Pre Process
Interceptor - Pre Process
Controller
Interceptor - Post Process
Interceptor - Post Process

Interceptor Configuration

- XML Version – Inside springmvc-config.xml

```
<mvc:interceptors>
   <mvc:interceptor>
     <mvc:mapping path="/*" />
     <bean
     class="edu.mum.interceptor.ProcessingTimeLogInterceptor"
     />
   </mvc:interceptor>
</mvc:interceptors>
```

- Java Config Version – Inside `WebApplicationContextConfig` class

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
  registry.addInterceptor(new
  ProcessingTimeLogInterceptor());
}
```

Interceptor Implementation

```
public class ProcessingTimeLogInterceptor implements HandlerInterceptor {
   private static final Logger LOGGER = Logger.getLogger(ProcessingTimeLogInterceptor.class);
   @Override
   public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
   throws Exception {
       long startTime = System.currentTimeMillis();
       request.setAttribute("startTime", startTime);
       return true;
   }
   @Override
   public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
   ModelAndView modelAndView) throws Exception {
       String queryString = request.getQueryString() == null ? "" : "?" + request.getQueryString();
       String path = request.getRequestURL() + queryString;
       long startTime = (Long) request.getAttribute("startTime");
       long endTime = System.currentTimeMillis();
       LOGGER.info(String.format("%s millisecond taken to process the request %s.", (endTime -
          startTime), path));
   }
   @Override
   public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
      Exception ex)
   throws Exception {
   //Callback after rendering the view.
   }}
```

Main Point

- Handler Mapping & Chaining aids in organizing functionality in layers. As a result the design is simpler & more consistent.
- *Life is structured in layers. This orderliness within us and around us allows us to enjoy more efficiency in our life.*