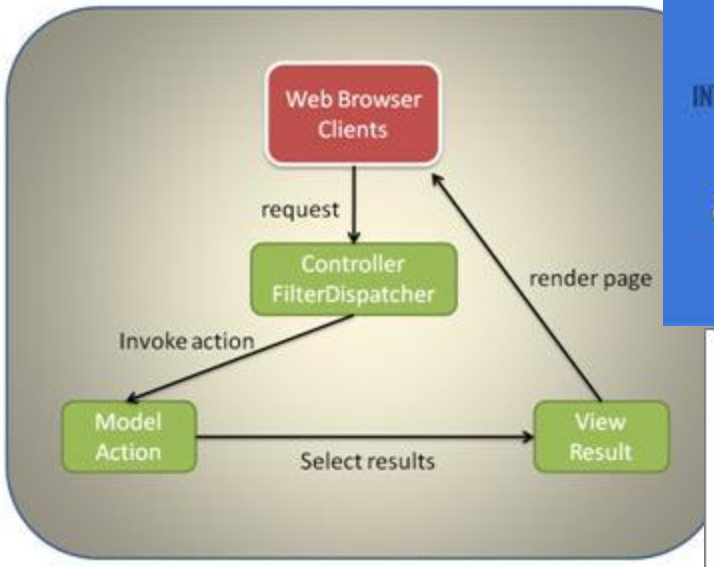
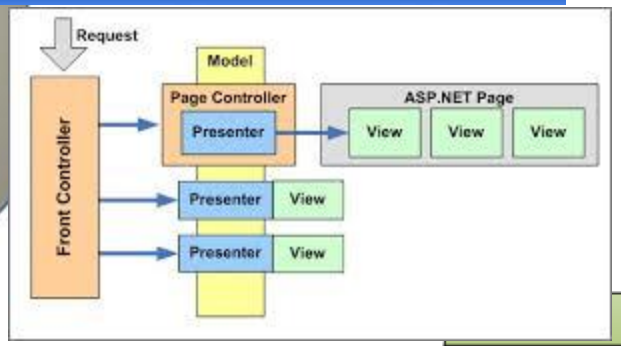
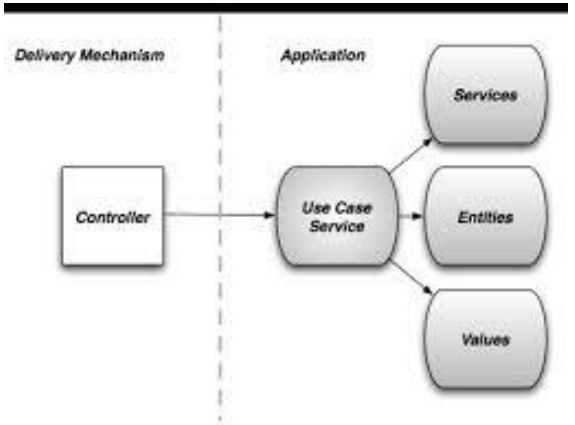
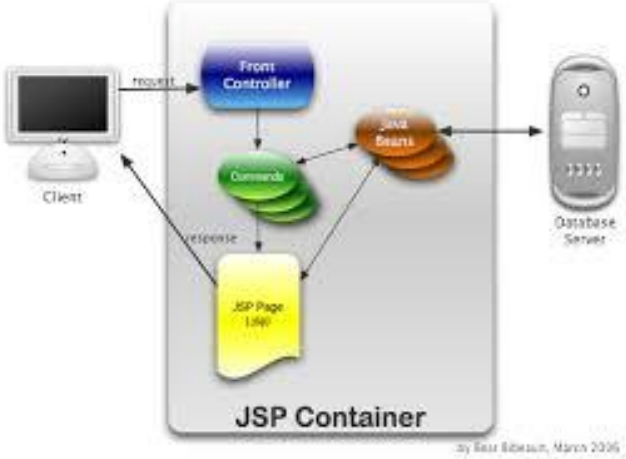
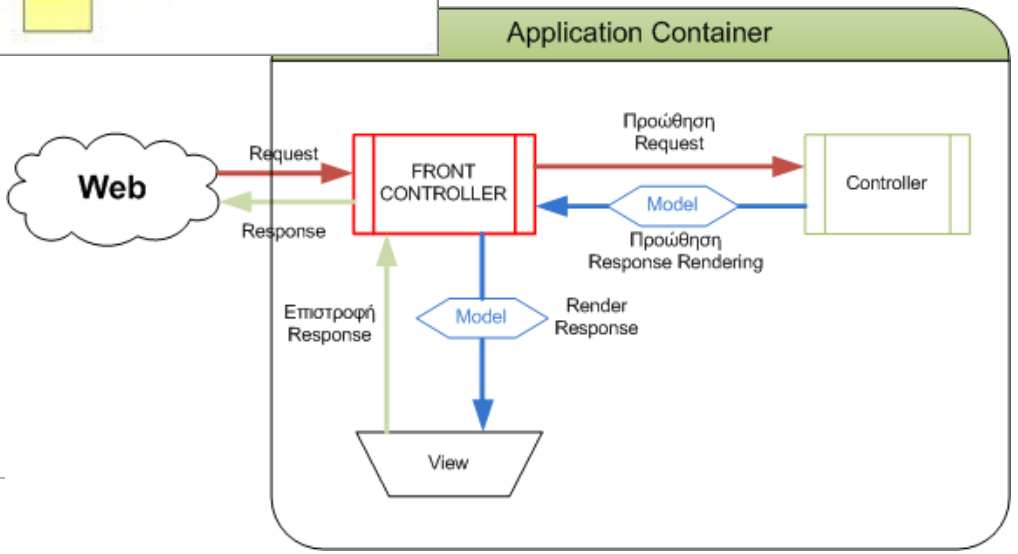
FRONT CONTROLLER







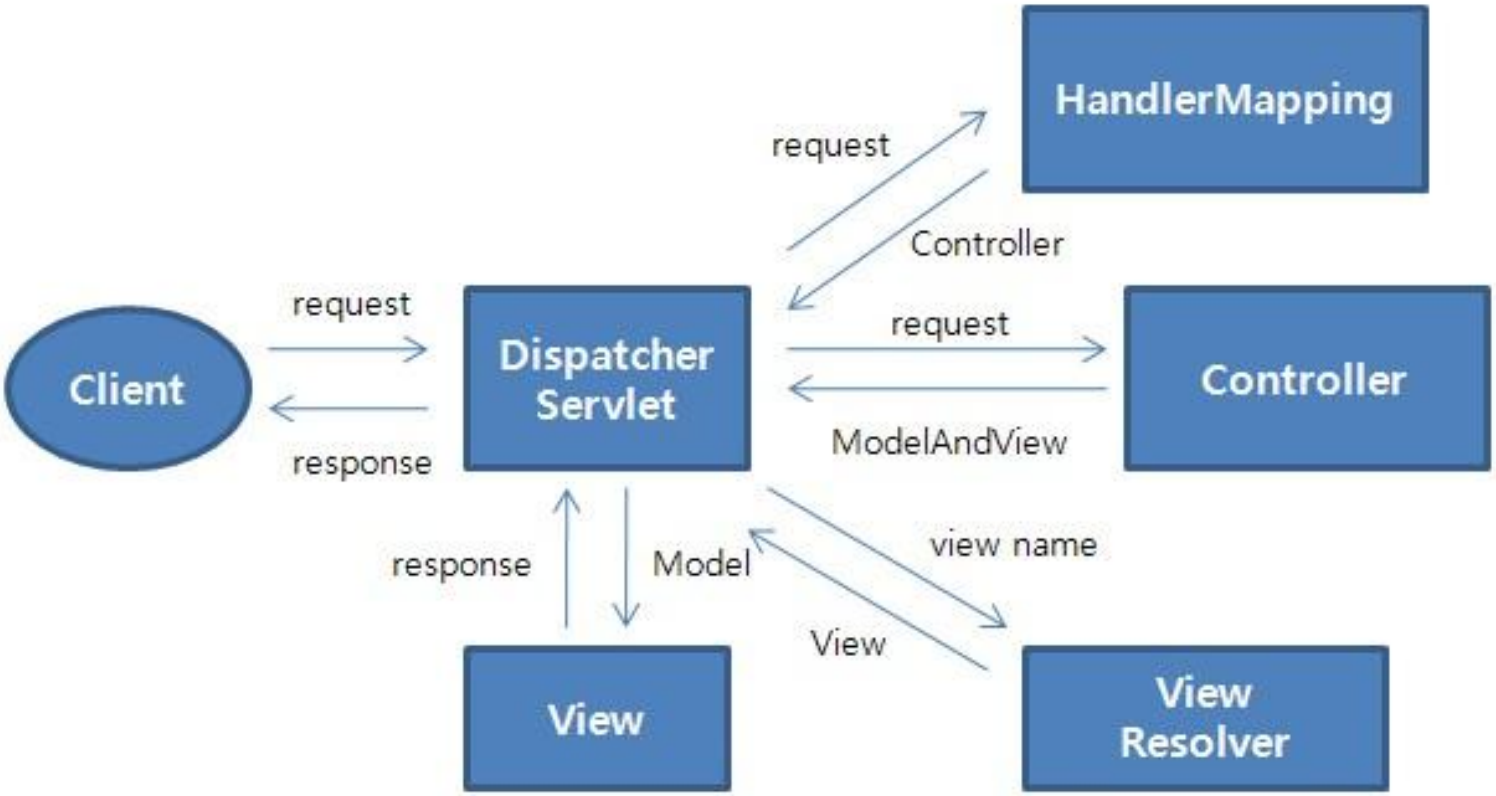




Spring MVC

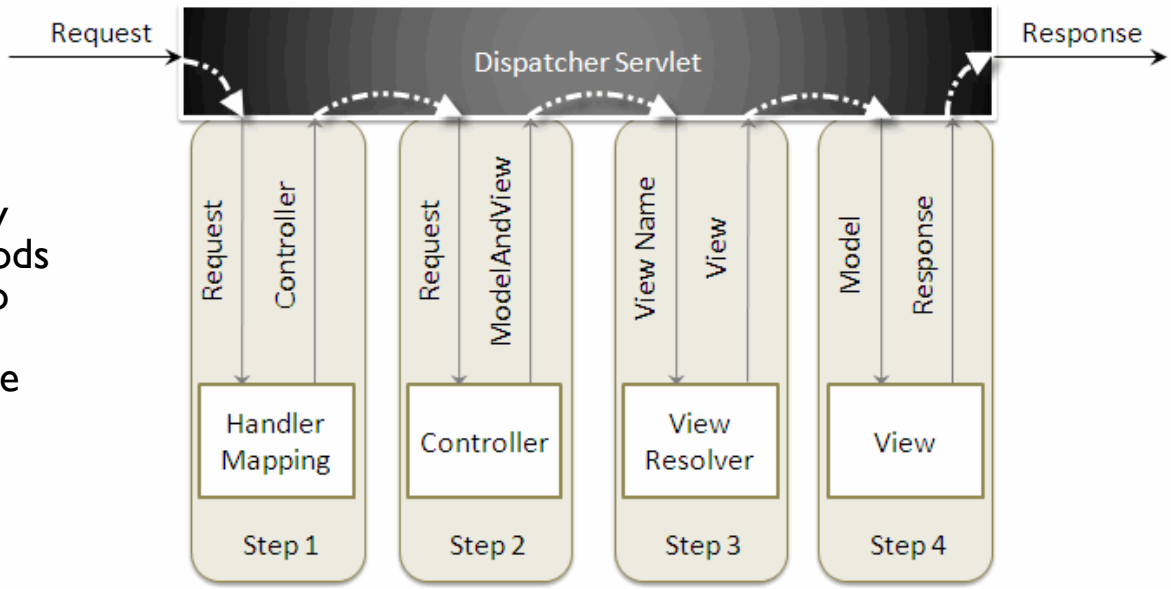
- Distinct Separation of Concerns
- Cleary defined interfaces for role/responsibilities "beyond" Model-View-Controller
- Single Central Servlet
  - Manages HTTP level request/response
  - delegates to defined interfaces
- Models integrate/communicate with views
  - No need for separate form objects
- 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.



@RequestParam

- Placed on Method argument

http://localhost:8080/webstore_SpringMVC/market/product?id=P1234

```
@RequestMapping("/product")
public String getProductById(@RequestParam("id") String productId, Model
   model) {
   model.addAttribute("product",
      productService.getProductById(productId));
   return "product";
}
```

- Handling multiple values [e.g., multiple selection list]

http://localhost:8080/webstore_SpringMVC/sizechoices?sizes=Small&sizes=Large&sizes=Medium

```
public String getSizes(@RequestParam("sizes")String[] sizeArray)
```

@PathVariable

- Facility to pass resource request as part of URL INSTEAD of as a @RequestParam
- Conforms to RESTful service syntax

http://localhost:8080/webstore/products/Laptop

```
@RequestMapping("/products/{category}")
public String getProductsByCategory(Model model, @PathVariable("category") String
   productCategory) {
   model.addAttribute("products",
      productService.getProductsByCategory(productCategory));
   return "products";
}
```

@PathVariable is used in conjunction with @RequestMapping URL template.
In this case it is a means to get the category string passed in the method signature.

The @PathVariable param needs to be the same as the param in the @RequestMapping

See demo: webstore_SpringMVC

Data Binding

- Automatically maps request parameters to domain objects
- Simplifies code by removing repetitive tasks
- Built-in Data Binding handles simple String to data type conversions
- HTTP request parameters [String types] are converted to model object properties of varying data types.
- Does NOT handle COMPLEX data types; that requires custom formatters
- Does handle complex nested relationships

Out-of-the-box Spring Formatters

- The Number Package:
  NumberFormatter,
  CurrencyFormatter
  PercentFormatter

- The DateTime Package:
  DateFormatter API

- Custom Examples [NOT out-of-the-box]:
  ISBN Number
  Multiple Select List – collection of Objects

ISBN Formatter Example

```
public class ISBNFormatter implements Formatter<ISBNumber> {

  public String print(ISBNumber isbn, Locale locale) {
     return isbn.getStart() + "-" +
                          isbn.getMiddle() + "-" + isbn.getEnd();
  }

  public ISBNumber parse(String source, Locale locale)
      throws ParseException {
    int start = Integer.parseInt(source.substring(0, 3));
    int middle = Integer.parseInt(source.substring(4, 7));
    int end = Integer.parseInt(source.substring(8, 11));
    return new ISBNumber(start, middle, end);
  }
}
```

See demo: Book_Tag_Formatter

@ModelAttribute

- **Can be placed on a method parameter:**

```
@RequestMapping(value = "/addBook", method = RequestMethod.POST)
public String saveBook(@ModelAttribute("newBook") Book book) {}
```

- The Object should be retrieved from the model or instantiated if doesn't exist. The Object fields should be populated from all request parameters that have matching names.

- **Can be placed on method. Method invoked before methods annotated with @RequestMapping**

```
@ModelAttribute("books")
List<Book> addBookList(Model model) {
     return  bookService.getAllBooks();
}
```

- Object is added to Model – in this example the List of books is added

Form examples with HTML output

```
<form:input id="title" path="title"/>
```

- Generated HTML:

```
<input id="title" name="title" type="text" value=""/>
```

If category.id already has a value,
Then it will be market as selected

- When *categories* is a LIST

```
<form:select id="category" path="category.id"
items="${categories}" itemValue="id" itemLabel="name" />
```

- Generated HTML:

```
<select id="category" name="category.id">
        <option value="1" selected="selected">Computing</option>
        <option value="2">Travel</option >
        <option value="3">Health</option >
</select>
```

- **NOTE:** *path is the "binding Path" defined previously*

Form example with HTML output [Cont.]

```
<form:select id="category" path="category.id">
 <form:option value="0" label="--Select Category"/>
 <form:options items="${categories}" itemLabel="name" itemValue= "id"/>
</form:select>
```

- Generated HTML:

```
<select id="category" name="category.id">
        <option value="0" selected="selected">--Select Category</ option >
        <option value="1">Computing</ option >
        <option value="2">Travel</ option >
        <option value="3">Health</option>
</select>
```

- **When categories is a MAP**

```
<form:select path="category.id" items="${categoriesMap}" />
```

- **Automatically Yields:**

```
<option value="1">Computing</ option >
```

Where "1" is the Map key

Model Scoped Attributes

- JSP page scope
  - The page scope restricts the scope and lifetime of attributes to the same page where it was created.
- Request scope
  - only be available for that request
  - Thread Safe
- Session Scope
  - Session is defined by set of session scoped attributes
  - Lifetime is a browser session
  - **Sessions are a critical state management service provided by the web container.**
- Context scope
  - Application level state
  - Lifetime is "usually" defined by deployment of application
  - Attributes available to every controller and request in the application

Managing state information

How to handle the different scopes of model information:

- Request scope: short term computed results to pass from one servlet to another (i.e., "forward")

  request.setAtttribute(key,value)

  model.addAttribute(key,value)

- Session scope: conversational state info across a series of sequential requests from a particular user

  HttpSession session = request.getSession(); session.setAttribute(key,value);

  @SessionAttributes - model.addAttribute(key,value)

- Application/context scope: global info available to all controllers in this application

  `request.getServletContext().getAttribute("appName")`

  - **OR**

```
@Autowired
ServletContext servletContext;

servletContext.getAttribute("appName")
```

Request Scope Attribute

```
@RequestMapping(value = "/forward")
public String forward(Product product, Model model) {
   product.setDescription("Request Attribute Exists!!");
   model.addAttribute("requestAttribute", product);
   model.addAttribute("redirectParamTest", "Request Parameter EXISTS!");
   return "forward:/get_forward";
}

@RequestMapping(value = "/get_forward")
public String getForward(Model model) {
   return "ForwardRedirect";
}
```

- ForwardRedirect.jsp

```
<h4>${redirectParamTest}</h4>
<h4>${requestAttribute.description}</h4>
```

- Demo: ProductSessionExample – Forward

@SessionAttributes

- Class level annotation that indicates an object is to be **added/retrieved** from Session.

```
@Controller
@SessionAttributes({ "Leonardo", "Splinter" })
public class SessionController {

    @RequestMapping(value = { "/getSession" }, method = RequestMethod.GET)
    public String inputProduct(Model model, HttpSession session) {
        Product product = new Product();
        product.setName("Leonardo Turtle");
        model.addAttribute("Leonardo", product);
        model.addAttribute("Splinter", "Splinter");
        // add Regular attribute
        session.setAttribute("Donatello", "Donatello Turtle");
        return "SessionForm";
    }
}
```

- Retrieve from Model

```
Product product = (Product) model.asMap().get("Leonardo");
```

- Used to mark a session attribute as not needed *after* the request has been processed by the controller

```
status.setComplete();
```


See demo: ProductSessionExample - Session

Application level Attributes

- ServletContext contains Application level state information

- XML configuration:

```
<bean
   class="org.springframework.web.context.support.ServletContextAttributeExporter
   ">
    <property name="attributes">
        <map>
            <entry key="appName" value="State Management Demo" />
        </map>
    </property>
</bean>
```

- Programmatic access:

```
@Autowired
ServletContext servletContext;

servletContext.getAttribute("appName");
```

Static Resources

- Want to handle static content, e.g., image file, js, css, etc.
- Need to identify them to the DispatcherServlet - since no Controller exists for serving static resources.
- Using Spring:
  - Declare resources folder[s]
  - Serve static content from there
  - Use mvc:resources – A Spring help element to map "url path" to a physical file path location.

- All references to `/resource/` will be mapped to the context root (webapp): `/css/` folder.

```
<mvc:resources mapping="/resource/**" location="/css/"/>
```

- Alternative: serves content from servlet containers
  - If we are using DefaultServletHttpRequestHandler, then we can replace :

```
<mvc:resources mapping="/js/**" location="/js/"/>
<mvc:resources mapping="/css/**" location="/css/"/>
<mvc:resources mapping="/images/**" location="/images/"/>
```

  - with :

```
<mvc:default-servlet-handler />
```

- path pattern - Apache ant
  - ('*') matches zero or more characters, up to the occurrence of a '/'.
  - ('**') matches zero or more characters. This could include the path separator '/'.

Flash Attributes

- Efficient solution for the *Post/Redirect/Get* pattern.
- Attributes are saved [in Session] temporarily before the redirect
- Attributes are added to the Model of the target controller and are deleted [from Session] immediately.

```
@RequestMapping(value = "/product", method = RequestMethod.POST)
public String saveProduct(Product newProduct, Model model,
   RedirectAttributes redirectAttributes,
    HttpServletRequest request) {

   redirectAttributes.addFlashAttribute(newProduct);
   // Returned as a parameter on GET URL
   redirectAttributes.addAttribute("name", "Kemosabe");
   return "redirect:/details";
}
```

- String & primitive types are added to URL [e.g., GET]

```
redirectAttributes.addAttribute(newProduct.name);
```

Use Case: Ensure Non-Empty Collection Elements

```
private List<String> names;

@NotEmpty
private List<String> names;

private List<@NotEmpty
@Pattern(regexp="[a-zA-Z]*") String> names;

@NotEmpty
private List<@NotEmpty String> names;
```

Cascaded Validation

```
@Valid
private List<Address> addresses;

private List<@Valid Address> addresses;

private Map<@Valid Address, Integer>
addressMap;

private Map<@Valid AddressType,
         List<@Valid Address>>
addressesByType;
```

Form Validation through Annotation

Step 1: Annotate domain model properties

It's for Strings and collections.

```
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 LocalDate birthDate;

    @NotNull
    private Integer salaryLevel;

    @Valid
    private Address address;

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

use for Objects

```
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

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}

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

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

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

- 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.*