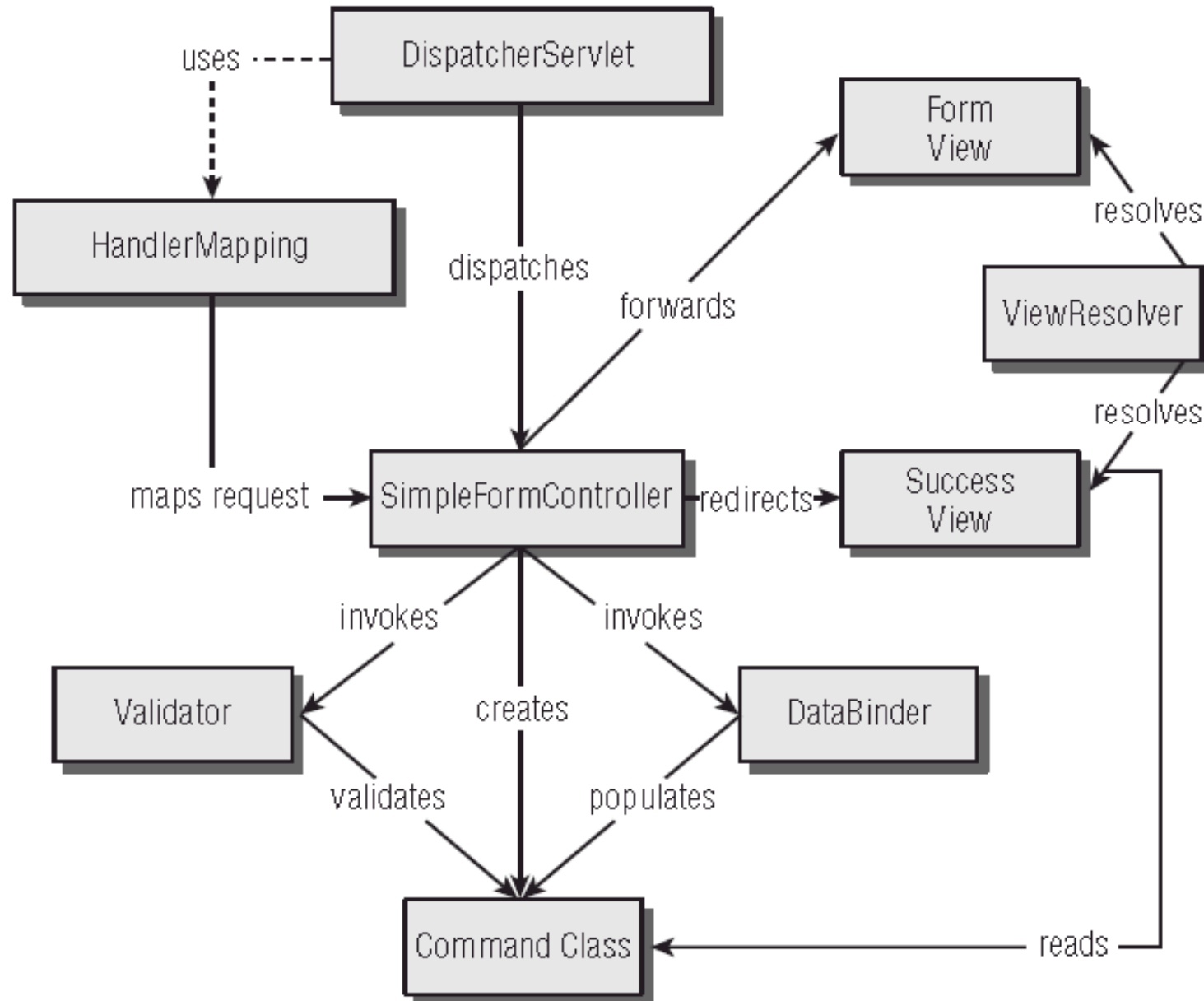# Getting Data from the User with Forms

- Providing the ability to capture user data is critical for most dynamic web applications.

- Users often need to register for a user account in order to access certain sections of a site, or fill out forms so that an online purchase order can be processed.

- Whatever site you're building, chances are you will want to capture user data.

A Basic Form-Submission Workflow

In the previous lecture on Spring MVC development you saw how a request reaches a controller and how the controller can retrieve data from the request to interact with the model. The controller then decides which view to use to render the data. The workflow for processing web forms follows a similar path, with the addition of a couple of extra steps to make sure that the data is valid before it is submitted to the database. Form validation is key to processing web forms, as users can never be trusted to enter data the way you would like them to.
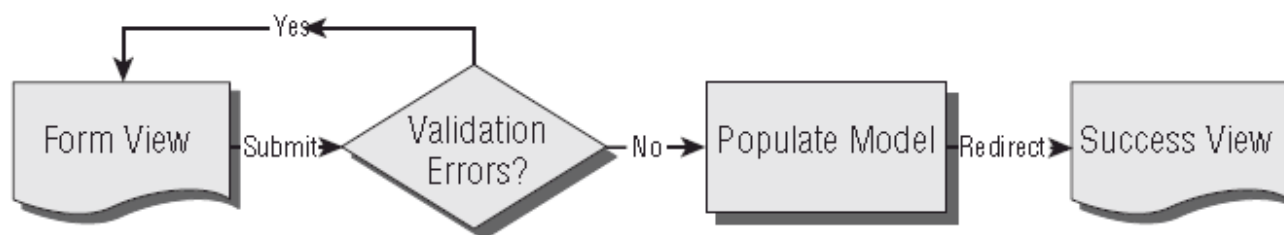
In theory you could start writing this workflow yourself by implementing the `Controller` interface. But obviously we wouldn't be talking about form submissions if there weren't an easier way to go about this. As you will see next, Spring MVC provides out-of-the-box implementations for handling form submissions, which will make your life a lot easier.

```
                              ┌──────────────────────┐
                              │   DispatcherServlet  │
         uses ┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┤                      │                          ┌───────────┐
              ┊                └──────────┬───────────┘                          │   Form    │
              ┊                           │                                      │   View    │
              ▼                           │                                      └───────────┘
     ┌──────────────────┐                 │                                        resolves
     │  HandlerMapping  │            dispatches                              ┌───────────────┐
     │                  │                 │          forwards                │  ViewResolver │
     └────────┬─────────┘                 │                                  └───────────────┘
              │                           ▼                                        resolves
              │              ┌─────────────────────────┐        ┌──────────────┐
 maps request ─────────────▶│  SimpleFormController   │─redirects─▶│   Success   │
              │              └─────────────────────────┘        │     View     │
                              invokes  │  invokes                └──────────────┘
                                 creates
   ┌──────────────┐                                    ┌──────────────┐
   │  Validator   │                                    │  DataBinder  │
   └──────────────┘                                    └──────────────┘
        validates                populates
              ┌──────────────────────────┐
              │      Command Class       │◀──── reads ────
              └──────────────────────────┘
```

FormHandling Components

- The DispatcherServlet and the HandlerMapping are used to map URLs to the controller, whereas the ViewResolver takes care of mapping a logical view name to a view implementation.

- The controller at the center of Spring MVC's web form processing is the SimpleFormController classs

- The SimpleFormController provides the boilerplate code to support the basics of a form-submission workflow.

- As an application developer, you would typically subclass the SimpleFormController and configure a number of auxiliary components invoked by the SimpleFormController to validate form data and pass it to the model.

- Validator implementations can be plugged in to validate the data posted to a command class.

- The command class is simply a JavaBean class that has fields used to store the data submitted by the user. These fields are accessible through getters and setters.

- Data from the web form is mapped to the command class's getters and setters with the help of a DataBinder.

- The DataBinder is responsible for converting form data to the command class's field types.

- In case any data validation errors occur, the SimpleFormController returns the user to the original form view so that error messages can be presented to the user.

- When the form is successfully processed, the user is taken to the next page.

Sample Form

Understanding Form Submissions in Spring MVC

□ Spring's SimpleFormController is used, as its name implies, to process simple web forms consisting of a single page.

□ The form shown in the previous slide can be used to create and populate a new photo album.

□ It has four fields on it, which are used to provide some basic information about a photo album.

Understanding Form Submissions in Spring MVC

- The album name is required.

- The description, album labels, and creation date are optional.

- The user can associate multiple labels with a photo album by checking the box next to a label.

- The creation date must be entered as *dd/mm/yyyy; 12/29/2007 is valid but 29/12/2007 is not.*

- *The labels on the form are* retrieved from the database.

- When the form is successfully validated, the user is redirected to the albums page.

- If any validation error occurs, an error message is written at the top of the form.

Command Class

- Typically the first step in creating a web form is to define the command class that will be used to store the data.

- A command class is a fancy name for an object that has fields that can be accessed via getter and setter methods.

- The JSP's field names follow a particular naming convention so that a DataBinder can automatically figure out where data from the form goes in the command class.

- Before you get into the details of mapping form fields to the command class, you should take a look at command class used by the sample application to create a new photo album.

Following are the relevant fields of the Album domain object that are populated by the form:

```java
public class Album implements Serializable
{
    private integer id;
    private String name;
    private String description;
    private Date creationDate;
    private String[] labels;
    // Additional fields.

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    // Additional accessors.
}
```

Using the Form View

As mentioned in the previous section, to populate the model with form data you need to provide a mapping between the form's field names and the model's properties. Spring uses the standard JavaBean naming convention for accessing JavaBean properties. The following table shows an example of the mapping between form field name and JavaBean properties.

Form Field Name	JavaBean Properties
Name	getName(), setName(..)
Date	getDate(), setDate(..)
person.age	getPerson(),.getAge(), getPerson(),.setAge(..)

```jsp
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
    <title><spring:message code="title.create" /></title>
</head>
<body>
<jsp:include flush="true" page="header.jsp"></jsp:include>
<div align="left"><spring:message code="${message}" text="" /></div>
<form:form>
<form:errors path="*" />
<table>
<tr>
    <th>Album Name*</th>
    <td><form:input path="name" /></td>
</tr>
<tr>
    <th>Description</th>
    <td><form:textarea path="description" /></td>
. . . . . . .
```

Why not use ordinary HTML page that follows the naming conventions described in the previous table?

□There is one caveat: it is not clear how you can redisplay the form's values after the data has been submitted to confirm the input values with the user.

□When using static HTML, you should use another part of the Spring MVC's tag library —the form tags.

□Spring MVC's form tags take care of rendering the appropriate HTML along with the field values.

□As you can see, the tags are similar to standard HTML form tags.

- Each references the model with the path attribute.
- The value of path is used to render the field's name and references the corresponding JavaBean property.

Populating the Model

□You may be wondering how Spring manages to convert a form's date input field to a Date object.

□Remember that dates are represented differently depending on where you live.

□PropertyEditor is a standard JDK interface, found in the java.beans package.

□This class provides the capability to take a String representation of an object and convert it to the appropriate object type.

□Spring provides implementations of the PropertyEditor interface for common object types such as URL, File, String and Locale, as well as others.
(See the org.springframework.beans.propertyeditors package for a complete list of property editors provided by Spring.)

□Property editors are not used only for type conversion with dependency injection but also for converting data submitted by a web form to an appropriate type in the model.

□The Album model shown in the earlier listing accepts a Date object to set the album's creation date.

□As you know, dates come in many shapes and forms and take different formats depending on the user's region. Because there is no common date format, it is necessary to tell the application how dates are entered by the user.

□The web form presented in the following listing accepts the *dd/mm/yyyy format.*

□*Spring's CustomDateEditor is used to provide support* for this date format.

□The following listing shows the implementation and registration in CreateAlbumController:

```
public class CreateAlbumController extends SimpleFormController
{
  //...
  protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) throws Exception
  {
    CustomDateEditor dateEditor = new CustomDateEditor(new SimpleDateFormat("dd/MM/yyyy"), true);
    binder.registerCustomEditor(Date.class, dateEditor);
  }
  //...
}
```

Registering the Album class with the controller

□**Next, you need to register the Album class with the controller by passing Album.class to the controller's setCommandClass method.**

□**By default, a new instance of Album is created each time the form is loaded. You can override the SimpleFormController's formBackingObject method if you want to customize this behavior so that you can, for example, reload an existing album in the form for editing.**

□**The following code shows an example of how this method is overloaded to retrieve an existing album from the database in case an ID parameter is passed to the form. This listing also shows how the Album class is registered as a command class in the constructor.**

```
public class CreateAlbumController extends SimpleFormController {
          private AlbumRepository albumRepo;

          public CreateAlbumController() {
                    setCommandClass(Album.class);
          }
          //...
          protected Object formBackingObject(HttpServletRequest request) throws Exception {
                    int id = ServletRequestUtils.getIntParameter(request, ALBUM_ID);
                    if (id != null && !"".equals(id)) {
                              return albumRepo.retrieveAlbumById(id);
                    } else {

                              return super.formBackingObject(request);
                    }
          }
          //...
          public void setAlbumRepo(AlbumRepository albumRepo) {
          this.albumRepo = albumRepo;
          }
}
```

□The request data is saved to the Album instance returned by the formBackingObject method and then passed to the doSubmitAction method.

□You need to override doSubmitAction to retrieve a populated Album object and store it in the database.

```
public class CreateAlbumController extends SimpleFormController
{
   private AlbumRepository albumRepo;
   //...
   protected void doSubmitAction(Object album) throws Exception
   {
      albumRepo.persistAlbum((Album) album);
   }

    //...
    public void setAlbumRepo(AlbumRepository albumRepo)
    {
       this.albumRepo = albumRepo;
    }
}
```

□Your CreateAlbumController is almost ready for deployment.

□The last missing piece is telling the controller where to point the user after the form has successfully been saved to the database. In your application you want to redirect the user to the albums page.

□To do so, you need to add the following code to the constructor:

```
public CreateAlbumController()
{
    setCommandClass(Album.class);
    setSuccessView("redirect:albums.htm");
}
```

□At this point your form is ready and you should be able to create new albums - that is, if you enter data correctly. Since you cannot depend on your users to do this, you need to learn how to add validation rules to your form.

Implementing Form Validation

☐ To enforce the form's data-entry rules you cannot rely on user discipline alone.

☐ Users often forget to fill out required fields or do not respect valid e-mail or date formats.

☐ To filter out these human errors before the data is submitted to the rest of the application, you need to enforce validation rules.

☐ Spring provides a flexible validation mechanism that can be used to preserve data integrity.

Data validation is provided by implementations of the `Validator` interface. Following is a `Validator` implementation for the `Album` class that checks to determine whether the name field is filled out:

```
public class AlbumValidator implements Validator {

    public boolean supports(Class clazz) {
        return Album.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
            "error.required.name");
    }
}
```

`Validator` implementations are fairly straightforward to write. A validator implements the `supports` method to provide information about the classes it knows to validate. The `validate` method receives an instance of this class along with an `Errors` collection, which can be used to pass validation errors to the application.

An `Errors` collection can be accessed in a JSP with the `form:errors` tag. `<spring:errors path="*"/>` prints out the entire collection of errors. It is also possible to print errors for a particular form field. For example, `<spring:errors path="name"/>` prints only validation errors that occur on the `Name` field. The path syntax is similar to the one used for form-field binding.

Validation can be used to enter user input errors that can be anticipated in advance. However, there are often user errors, coding errors, and system errors that cannot be easily anticipated.

In-class exercise