

Willie Wheeler and John Wheeler

MEAP

Unedited Draft



Spring

IN PRACTICE

- GET GOING
- GET SAVVY
- 50+ PROBLEMS SOLVED
- COVERS SPRING 3

 MANNING

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>



MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

TABLE OF CONTENTS

Part 1: Getting Started

Chapter 1: Introducing Spring: the IoC container

Chapter 2: Working with databases and transactions

Chapter 3: Spring Web MVC

Part 2: General Application Recipes

Chapter 4: User registration

Chapter 5: Authentication

Chapter 6: User account redux and authorization

Chapter 7: Site navigation and themes

Chapter 8: Communicating with users and customers

Chapter 9: Community

Chapter 10: Application monitoring and diagnostics

Part 3: Domain-Specific Recipes

Chapter 11: Product catalogs

Chapter 12: Shopping carts

Chapter 13: Customer orders

Chapter 14: Lead generation and qualification

Chapter 15: Customer relationship management

Chapter 16: Technical support

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

4

User registration

User registrations are important. For many applications and websites, registration rates are closely tied to revenue goals: registrations generate revenues. They enable orders, they support targeted marketing campaigns and advertising, and in general they represent a user's or customer's desire for an ongoing relationship with your company or website. For these reasons alone it literally pays to take user registrations seriously.

User registrations are also a source of risk. User data is sensitive and we have to secure it to avoid PR, legal and operational problems. We want to reduce our exposure to legal risk by presenting our privacy policy and asking registrants to acknowledge our terms of use. Sometimes we have to take steps to protect children too.¹

Given this dual importance, we want to get our registration forms right. We don't want avoidable usability issues to cost us registrations. We don't want to compromise user data. And we don't want to expose our employers or ourselves to lawsuits.

Attention to detail helps. Consider the problem of maximizing user registrations. The form should be short, nonthreatening and easy to use. We can

- put users at ease by delivering the initial form over HTTPS even though it doesn't offer any security advantage;
- provide an explicit privacy policy (which has usability and presumably legal benefits);
- use a "Register securely" button instead of simply saying "Register" or "Submit;" and

¹ See <http://www.coppa.org/> for more information about protecting children. It's about now that I'll invoke IANAL ("I am not a lawyer"). Don't take anything in this chapter or in this book to be legal advice. It's not. If you need legal advice please contact a qualified legal expert.

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

- improve the form's usability by giving focus to the first form field when the form loads, providing descriptive form labels and error messages, highlighting fields containing errors and so forth.

While these gestures may seem small, collectively they can increase registration rates by making your site more trustworthy and convenient to use.

Our goal for this chapter is to learn how to design and implement user registration forms using Spring. We will of course consider standard topics such as displaying the form, validating user data and saving user data. But we'll also explore design issues that allow us to improve usability, improve security and address legal risk.

4.1 Display a registration form

PREREQUISITES

None

KEY TECHNOLOGIES

Spring Web MVC, Spring form tag library

Background

Users often establish a more in-depth relationship with a website or an organization by registering. The resulting account supports an ongoing relationship between the user and the website, allowing logins, order placement, community participation and so on.

Usability, security and legal risk are key concerns commonly associated with user registration forms: we want simple and trustworthy forms to encourage registrations, we want secure forms to prevent user data from being compromised, and we want to do whatever we need to do to keep our lawyers smiling. In this recipe we'll build out a registration form, paying particular attention to these and other issues.

Problem

You want to create a web-based user registration form. The form should accept standard inputs (first name, e-mail, etc.) and bind them to a user model for further processing. The form should be usable and secure. Finally it should adopt standard approaches for reducing legal exposure, such as requiring the user to acknowledge terms of use.

Solution

In this recipe, we'll use Spring Web MVC to display a user registration form. We'll build a user model, a registration form, a confirmation page and a controller to process form submissions. We'll bind user inputs to the user model so we can easily validate and store user data. (We'll validate the user model in recipe 4.2 and store the data in recipe 4.3.)

CREATE A USER MODEL

Model objects store information users provide in web-based forms. These are variously referred to as command objects, form-backing objects (a.k.a. form beans) or model objects, depending on the specific flavor. In this case we're using something that's halfway between a

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

form bean and a model object since we're modeling the form itself (witness the `passwordConfirmation` field) but later we'll persist our `User` instances (which makes `User` more like a model object). We'll just call ours a "user model." Here it is:

Listing 4.1 `User.java`, a user model for our registration form

```
package r4_1.model;

import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.commons.lang.builder.ToStringStyle;

public class User {
    private String firstName;           #1
    private String lastName;
    private String username;
    private String email;
    private String password;
    private String passwordConfirmation; #2
    private boolean marketingOk = true; #3
    private boolean acceptTerms = false; #4
    private boolean enabled = true;     #5

    public String getFirstName() {
        return firstName;
    }

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

    ...other getter/setter pairs...

    public String toString() {         #6
        return new ToStringBuilder(this, ToStringStyle.SHORT_PREFIX_STYLE)
            .append("firstName", firstName)
            .append("lastName", lastName)
            .append("userName", username)
            .append("email", email)
            .append("marketingOk", marketingOk)
            .append("acceptTerms", acceptTerms)
            .append("enabled", enabled)
            .toString();
    }
}
```

Cueballs in code and text

- #1 Standard persistent demographic data**
- #2 Nonpersistent password confirmation**
- #3 OK to market to user?**
- #4 Does user accept terms of use?**
- #5 Is account enabled?**
- #6 `toString()` implementation**

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

`User` is a user model with properties for personal [#1], security-related [#2], marketing [#3] and legal [#4] information, which are all common registration form concerns. Note that the `passwordConfirmation` field is a little out of place: in reality it's more a part of the form model rather than the user model, but it's convenient to include it here. We also have an administrative `enabled` property [#5], which has precisely the opposite characteristics: it's part of the user model but not part of the form. One of the design decisions you'll have to make when dealing with registration forms (and forms in general) is how "pure" you want your models to be. If you want to separate your form model and user model you can do that, but we're opting here to just combine the two into a single form.

We've defaulted our `marketingOk` property to true since we'd like to market to our users unless they explicitly opt out. On the other hand, we've defaulted `acceptTerms` to false because we want the user's acceptance of the terms of use to be active rather than passive. Presumably this gives us a stronger legal leg to stand on in the event of a disagreement with the user.²

`User` conforms to the JavaBeans component model, having getters and setters for each property. This allows Spring Web MVC to bind registration form data our `User` bean for further processing. Also, we've included a descriptive `toString()` method [#6], based on the Commons Lang library, so we can see the form binding later in the recipe.

Let's move on to the views.

CREATE A REGISTRATION FORM VIEW

Let's create a basic registration form that we'll enhance over the course of the chapter. All we need for now are labels and input fields. Figure 4.1 shows what we want to build:

² Again, I am not a lawyer! Consult a qualified legal expert if necessary. See footnote 1.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

Register

First name:

Last name:

E-mail address:
 We won't spam you

User name:

Password:

Confirm password:

You can read our [privacy policy](#) in a new window if you'd like.

Figure 4.1 The bare-bones registration form we will create

And here, in listing 4.2, is a JSP that does it:

Listing 4.2 form.jsp, a view to display our registration form

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Register</title>
    <style>
      .form-item { margin: 20px 0; }
      .form-label { font-weight: bold; }
    </style>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById("firstName").focus();
      }
    </script>
  </head>
  <body>
    <h1>Register</h1>
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

<p>All fields are required.</p>
<form:form modelAttribute="user"> #1
  <div class="form-item">
    <div class="form-label">First name:</div>
    <form:input path="firstName" size="40"/> #2
  </div>
  <div class="form-item">
    <div class="form-label">Last name:</div>
    <form:input path="lastName" size="40"/>
  </div>
  <div class="form-item">
    <div class="form-label">E-mail address:</div>
    <form:input path="email" size="40"/> We won't spam you
  </div>
  <div class="form-item">
    <div class="form-label">User name:</div>
    <form:input path="username" size="40"/>
  </div>
  <div class="form-item">
    <div class="form-label">Password:</div>
    <form:password path="password"
      showPassword="true" size="40"/> #3
  </div>
  <div class="form-item">
    <div class="form-label">Confirm password:</div>
    <form:password path="passwordConfirmation"
      showPassword="true" size="40"/>
  </div>
  <div class="form-item">
    <form:checkbox path="marketingOk"/> #4
    Please send me product updates by e-mail.<br />
    (You can unsubscribe at any time.)
  </div>
  <div class="form-item">
    <form:checkbox path="acceptTerms"/>
    I accept the
    <a href="#" target="_blank"> terms of use</a>.
  </div>

  Please see our <a href="#" target="_blank">privacy policy</a>.

  <div class="form-item">
    <input type="submit" value="Register"/> #5
  </div>
</form:form>
</body>
</html>

```

Cueballs in code and text

- #1 Renders an HTML <form>
- #2 Renders an HTML <input type="text">
- #3 Renders an HTML <input type="password">

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

#4 Renders an HTML checkbox
#5 No special submit tag

Our registration view is simple enough. It uses the `form` [#1], `input` [#2], `password` [#3] and `checkbox` [#4] tags from the Spring form tag library to render HTML form elements that are "bound" to their underlying properties. This means that the form elements automatically reflect the value contained in the underlying user model, and that changes to the form element will be pushed into the user model when the form is submitted. We'll see this in action in recipe 4.2; for now it's enough to understand that these are basically enhanced HTML form input widgets. The tag library doesn't provide anything for submit buttons, so we just use standard HTML [#5].

Let's pause to consider some usability concerns we've attempted to address with the design of the registration form in listing 4.2.

DESIGN ISSUES: USER PSYCHOLOGY AND USABILITY

In many cases it's an important business goal to maximize registration rates. Attention to user psychology and usability can pay dividends here; the general theme is to eliminate barriers to completing the form. Here are several ideas:

1. Obviously you don't always control the business requirements here, but try to minimize the number of fields that the user is required to complete. Clearly distinguish required from optional fields.
2. Even better than minimizing the number of required fields is to minimize the number of fields, period. Long forms are intimidating. Multipage registration forms are similarly intimidating, especially if it isn't clear how long the form actually is. Providing visual indications of progress can be helpful in multipage registration scenarios.
3. Understand that certain types of data are much more "personal" than other types, and that by requiring (or even asking for) highly personal data, your registration rates will drop. If you ask me for my first name, that is probably OK, but if you ask for my phone number, my home address or my SSN, I have to be pretty committed before I complete your registration form.
4. Do what you can to establish trust with the user. In listing 4.2, we've indicated that we won't spam the user and we've provided a link to a privacy policy. If you're asking for highly personal data of the sort described in item 3 above, then consider adding "Why?" links next to the field to explain why you're asking for the data in question. In chapter 6 we'll discuss the use of HTTPS to further establish trust with the user.
5. Dip into your bag of "stupid JavaScript tricks" if it helps you enhance usability. In listing 4.2 we use JavaScript to place initial focus on the first name field. It's small but it helps. Avoid using purely gratuitous effects. Effects that enhance usability (such as highlighting the currently selected form field) are great.

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

6. Normally we'd use the HTML `label` element for our checkboxes for improved usability, but it doesn't make to do that for the terms of use since the text contains a link. And it probably doesn't make sense to use the label for one checkbox but not the other, so we've just left them off.
7. Use analytics and site optimization tools to try different ideas out, such as "what happens to registration rates if we stop asking for home addresses?" Google Analytics and Google Optimizer can be of great value here (and they're free).

Those are some common usability design issues associated with registration forms. Let's look briefly at some other design issues.

DESIGN ISSUES: ADDRESSING SECURITY AND LEGAL CONCERNS

We've already discussed these a bit, but besides taking steps to improve usability, we also want the form to be secure, and to address legal risks.

Regarding security, we'll have much more to say in recipe 4.4 and in chapter 6, but even with the simple display of the form we're dealing with certain security issues. We use the HTML `password` element to keep passwords hidden from onlookers. We can use the `showPassword` attribute of the `form:password` tag to control whether the password is prepopulated in the case of a form validation error. (We'll cover validation in recipe 4.2.) Suppressing the password is probably a bit more secure, especially if you aren't using HTTPS, but showing it is more convenient for the end user. So take your pick.

We mentioned the privacy policy and terms of use in connection with legal issues. We're launching those two items in a new window so the user doesn't lose all of his registration data should he click on them.

One other piece that bears repeating is the Children's Online Privacy Protection Act, or COPPA. If your site targets children under 13 or if you know that you have users under 13, then you will want to find out more about COPPA.

Obviously the specific legal issues that apply to your site are going to depend on the nature of your business and site, so you should consult a qualified legal expert if you have any questions or doubts in this area.

With the design issues behind us, we're done with the registration form itself (at least as far as the current recipe is concerned). In the next recipe we'll see how to use the registration page we just created not only for initially delivering the form but also for handling validation failures. But for now let's examine the confirmation page that users see after they successfully register.

CREATE A BASIC CONFIRMATION PAGE

We need a page to send users to once they've successfully registered. Here's a minimalistic `confirm.jsp` file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

        <h1>Registration Confirmed</h1>
        <p>Thank you for registering.</p>
    </body>
</html>

```

Nothing fancy at all. It doesn't really even have to be a JSP though it's a little easier to set your Spring view resolution up if you make it one, as opposed to making it an HTML file.

We're all done with our user model and our two views. Now it's time for us to handle the third part of the MVC triumvirate: the controller.

COORDINATE THE MODEL AND VIEWS WITH A REGISTRATION FORM CONTROLLER

Our registration form controller, in listing 4.3, handles both GET and POST requests for the registration form.

Listing 4.3 RegFormController.java, a controller to serve and process registration forms

```

package r4_1.web;

import java.util.logging.Logger;

import org.apache.commons.lang.StringUtils;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.*;

import r4_1.model.User;

@Controller
@RequestMapping("/register/form")
public class RegFormController {
    private static final String MA_USER = "user";

    @RequestMapping(method = RequestMethod.GET) #1
    public void form(Model model) {
        model.addAttribute(MA_USER, new User());
    }

    @RequestMapping(method = RequestMethod.POST) #2
    public String form(
        @ModelAttribute(MA_USER) User user,
        BindingResult result) {

        verifyBinding(result);
        Logger.global.info("Processing user: " + user);
        return "redirect:confirm";
    }

    @InitBinder
    public void initBinder(WebDataBinder binder) { #3
        binder.setAllowedFields(new String[] {

```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

        "firstName", "lastName", "email",
        "username", "password", "passwordConfirmation"
        "marketingOk", "acceptTerms"
    });
}

private void verifyBinding(BindingResult result) { #4
    String[] suppressedFields = result.getSuppressedFields();
    if (suppressedFields.length > 0) {
        throw new RuntimeException("You've attempted to bind fields
            [CA]that haven't been allowed in initBinder(): "
            + StringUtils.join(suppressedFields, ", "));
    }
}
}

```

Cueballs in code and text

- #1 Show the form on GET requests**
- #2 Process the form on POST requests**
- #3 Define a binding "whitelist"**
- #4 Throw RuntimeException when fields aren't in the whitelist**

GET requests retrieve our form at [#1] and we process POST requests at [#2]. Currently, we don't do much when users click submit. We simply send them to our confirmation page, which is handled by a separate controller (not shown). Of course, in the recipes that follow we'll add functionality to the form submission handler method.

As noted earlier, Spring Web MVC takes care of binding HTML form fields to our model properties. Internally, it uses request parameters and reflection to do this, and that's something that makes Spring Web MVC easy to use from a developer's perspective.

Unfortunately, that convenience also makes Spring Web MVC easy to hack if you're not paying attention.

A POTENTIAL SECURITY ISSUE SURROUNDING FORM BINDING

Hackers (the bad kind) could potentially alter the state of our model objects by crafting malicious requests. For example, our `User` object has an `enabled` property that is initially set to `true`, but what if we needed to initially set it to `false` and enable users only after they clicked a special link in a confirmation e-mail we sent them? In that case, a hacker could craft a request with `enabled` set to `true` in our form, and he'd bypass our confirmation process.

Spring Web MVC makes it fairly straightforward to protect our application from this sort of data injection attack. To make sure our forms aren't altered, we're using the `@InitBinder` annotation to define an explicit whitelist containing all the fields the binder is allowed to bind at [#3]. Any submitted parameters not in the whitelist remain unbound.

In practice, the problem with doing this is that we have to remember to update our whitelist every time we add a new field to our class and form; otherwise we'll scratch our

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

heads wondering why our field isn't being bound. To solve this problem we've created a helper method at [#4] that throws a `RuntimeException` every time a new parameter is submitted but isn't explicitly allowed by the binder.

Discussion

User registrations are an important piece of application functionality. Whether or not our goal is to maximize registrations, we certainly want our registration forms to be usable, secure, legally helpful and well-engineered. In this recipe we've seen how a little analysis coupled with Spring Web MVC can help.

It's worth noting that much of what we've done so far isn't particularly tied to registration forms; you can use this recipe as a blueprint for displaying web-based forms in general. As we move forward in the chapter we'll continue to target registration forms, but most of the discussion and techniques will be broadly applicable.

Not all applications that require user logins need registration forms. Sometimes you will have an account provisioning process instead of a user registration process.

In the following recipes we'll build upon the work we've done to validate and save registration data. We'll also look at techniques for making registration forms more secure, both in recipe 4.4 and in chapter 6.

4.2 Validate user registrations

PREREQUISITE

4.1 Display a registration form

KEY TECHNOLOGIES

Spring Web MVC, Spring Validation, Bean Validation Framework, Spring form tag library

Background

People make mistakes. No matter how intuitive our registration form might seem, chances are good that people will accidentally or even intentionally fill it out with garbage information. We generally treat such errors as user errors rather than system or application exceptions, meaning that we usually want to explain the error to the user in non-technical language and help him overcome it.

Several frameworks exist to assist with handling user errors of this sort, including Commons Validator, Valang, Hibernate Validator and the Bean Validation Framework. Fortunately, Spring integrates with such frameworks.

Problem

When users submit registration form data, you want to validate it before saving it to a persistent store. If there are validation errors, you would like to help the user understand what went wrong, and you would like to make it as easy as possible for him to correct the issue so he can continue.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

Solution

We will use the Bean Validation Framework (BVF) to validate user registrations. The BVF, which is based on the Spring Validation framework, is part of the Spring Modules project. The BVF is nice because it supports annotation-based declarative validation.³ With the BVF, we can annotate our `User` properties with validation rules. The annotations work with the form binding we set up in the previous recipe to flag rule violations and generate error codes. Spring Validation maps those to localized error messages.

ANNOTATE THE USER DOMAIN OBJECT WITH VALIDATION RULES

BVF annotations are straightforward and easy to use. Revisiting our `User` class, let's see how it looks fortified with validation rules. Listing 4.4 shows the code:

Listing 4.4 Annotating User fields with validation rules

```
package r4_2.model;

import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.commons.lang.builder.ToStringStyle;
import org.springframework.validation.bean.conf.loader.annotation.handler.*;

public class User {

    @NotBlank                               #1
    @MaxLength(40)                           #2
    private String firstName;

    @NotBlank
    @MaxLength(40)
    private String lastName;

    @NotBlank
    @MaxLength(20)
    private String username;

    @NotBlank
    @MaxLength(80)
    @Email                                   #3
    private String email;

    @NotBlank
    @MaxLength(20)
    @Expression(value = "password equals passwordConfirmation", #4
                errorCode = "password.not.match") #5
    private String password;

    private String passwordConfirmation;

    private boolean marketingOk = true;
```

³ Hibernate Validator also supports annotation-based declarative validation. JSR-303 targets the development of a standard approach to annotation-based bean validation.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

    @Expression(value = "acceptTerms equals true",
                errorCode = "terms.not.accepted")
    private boolean acceptTerms = false;

    private boolean enabled = true;

    ...
}

```

Cueballs in code and text

- #1 First name can't be empty or null**
- #2 First name can't be over 40 characters**
- #3 E-mail must be formatted correctly**
- #4 Password must match confirmation**
- #5 Override error code 'expression' with 'password.not.match'**
- #6 User must accept terms**

In listing 4.4, `@NotBlank` [#1] makes sure fields aren't empty or null, `@MaxLength` [#2] constrains fields to specified lengths and `@Email` [#3] ensures the e-mail field contains a valid e-mail address. The `@Expression` [#4] annotation over `password` takes as its first argument a freeform expression; the one we've provided enforces password equality.

MAP ERROR CODES TO MESSAGES IN A PROPERTIES FILE

When one or more validation rules are broken, `BeanValidator` generates error codes that Spring then resolves to localized messages. The correspondence between the annotations themselves and their generated error codes is intuitive. For example, `@NotBlank` generates `not.blank` and `@Email` generates `email`. In our `@Expression` annotation, we've replaced the default error code, `expression`, with `password.not.match` [#5].

Error codes are mapped to messages in a standard `error.properties` file:

```

not.blank=Please enter a value.
max.length=Please enter no more than {1} characters.
email=Please enter a valid e-mail.
password.not.match=Passwords do not match.

```

It's also worth mentioning that we have fine-grained control over how error codes map to messages. We can specify error codes with `rule.bean.property` syntax, and they will take precedence, on a per-field basis, over error codes with only the rule specified like those above. So, `not.blank.user.firstName` would take precedence over `not.blank` and would only apply to the `firstName` field on `User`. You might want to do this, for instance, to provide a more descriptive error message, such as one that describes the expected format, or one that is just clearer. (Compare "Password cannot be blank" to "Please choose a password containing at least six characters.")

The user must accept the terms of use to register [#6].

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

SET UP BEANVALIDATOR WITH ANNOTATIONS SUPPORT IN THE SPRING CONFIGURATION

Now we need to wire up BeanValidator with a few dependencies. We also need to specify our message source. Listing 4.5 shows the configuration for this:

Listing 4.5 Configure your BeanValidator in your application context

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/
      [CA]spring-context-2.5.xsd">

  <context:component-scan base-package="r4_2.web"/>

  <bean class="org.springframework.web.servlet.view.
    [CA]InternalResourceViewResolver"
    p:viewClass="org.springframework.web.servlet.view.JstlView"
    p:prefix="/WEB-INF/jsp/"
    p:suffix=".jsp"/>

  <bean id="configurationLoader"
    class="org.springframework.validation.bean.conf.loader.annotation.
      [CA]AnnotationBeanValidationConfigurationLoader"/> #1

  <bean id="errorCodeConverter"
    class="org.springframework.validation.bean.converter.
      [CA]KeepAsIsErrorCodeConverter"/> #2

  <bean id="validator"
    class="org.springframework.validation.bean.BeanValidator"
    p:configurationLoader-ref="configurationLoader"
    p:errorCodeConverter-ref="errorCodeConverter"/> #3

  <bean id="messageSource"
    class="org.springframework.context.support.
      [CA]ReloadableResourceBundleMessageSource"
    p:basename="WEB-INF/errors"/>
</beans>
```

Cueballs in code and text

- #1 Extracts validation rules from annotations
- #2 Fine- and coarse-grained message codes
- #3 BeanValidator

AnnotationBeanValidationConfigurationLoader [#1] extracts validation rules from our annotations. Incidentally, BeanValidator [#3] can also work with XML files given the

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

appropriate `configurationLoader`, but we don't cover that in this book. `KeepAsIsErrorCodeConverter` [#2] lets us choose between fine- and coarse-grained message resolution as we discussed above. The default `ErrorCodeConverter` implementation, `ModelAwareErrorCodeConverter`, supports only fine-grained message codes, so there isn't a good reason to use it.

INJECT THE VALIDATOR IN THE CONTROLLER AND VALIDATE THE USER MODEL

With our `BeanValidator` configured, listing 4.6 shows how to autowire it into our controller and use it to validate our user model:

Listing 4.6 Wiring up `BeanValidator` in `RegFormController` and validating Users

```
package r4_2.web;

import org.springframework.validation.Validator;
...other imports...

@Controller
public class RegFormController {
    private static final String MA_USER = "user";

    @Autowired
    private Validator validator;

    public void setValidator(Validator validator) {
        this.validator = validator;
    }

    @RequestMapping(method = RequestMethod.POST)
    public String form(
        @ModelAttribute(MA_USER) User user,
        BindingResult result) {

        verifyBinding(result);
        validator.validate(user, result); #1

        if (result.hasErrors()) {
            result.reject("form.problems"); #2
            return "register/form";
        }

        Logger.global.info("Processing user: " + user);
        return "redirect:confirm";
    }

    ...other methods...
}
```

Cueballs in code and text

#1 Validate the model

#2 Return to form if errors present

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

The `validate()` method [#1] takes in our user model and determines whether or not any validation rules have been violated. It reports any problems to the `result` object, which is then implicitly passed on to our view and rendered (as error messages) if it contains errors. If there are any validation errors, we create a global error [#2], which we'll display at the top of the form shortly.

USE THE SPRING FORM TAG LIBRARY TO DISPLAY VALIDATION ERRORS ON THE REGISTRATION FORM

We indicated earlier that when a user makes an error, one of our goals is to help him understand what went wrong and to fix it. We present user-friendly error messages that explain exactly those things. For that, we turn to the Spring form tag library.

Besides displaying helpful error messages, we also want to help the user quickly triangulate on the error by visually distinguishing the problematic fields from the others.

Figure 4.2 shows how our form looks with validation applied:

First name:

Please enter a value.

Last name:

Please enter no more than 40 characters.

E-mail address:
 We won't spam you
Please enter a valid e-mail.

Figure 4.2 Here's how our form will look when it is submitted with problems.

In listing 4.7, we display validation errors to the user using the `form:errors` tag:

Listing 4.7 Displaying validation errors in `form.jsp` with the Spring Form tag library

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

<html ...>
<head>
  <title>Register</title>
  <style>
    .form-item { margin: 20px 0; }
    .form-label { font-weight: bold; }
    .form-error-message { color: #900; } #A
    .form-error-field { background-color: #FFC; } #B
  </style>
  <script type="text/javascript">
    window.onload = function() {
      document.getElementById("firstName").focus();
    }
  </script>
</head>
<body>
  <form:form ...>
    <form:input type="text" name="firstName" />
    <form:input type="text" name="lastName" />
    <form:input type="text" name="email" />
  </form:form>
</body>
</html>
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

    }
  </script>
</head>
<body>
  <h1>Register</h1>
  <form:form>
    <div class="form-item">
      <div class="form-label">First name:</div>
      <form:input path="firstName" size="40"
        cssErrorClass="form-error-field"/> #1
      <div class="form-error-message"> #2
        <form:errors path="firstName"/>
      </div>
    </div>
    ..other form fields...
  </form:form>
  ...
</body>
</html>

```

Cueballs in code and text

#A Error messages are red to distinguish from other text

#B Inputs with errors have a light yellow background

#1 This CSS class is only selected for inputs with errors

#2 Empty until errors are present

The divs that contain `form:errors` tags [#1] are empty until validation errors are present. Then, our localized error messages display in red to distinguish them from the rest of the form labels. The `cssErrorClass` attribute [#2] renders as a standard HTML `class` attributes when its corresponding field has a validation error. We use it to apply a subtle yellow background to indicate that the field requires attention.

DISPLAY A BANNER OVER THE REGISTRATION FORM TO ALERT USERS OF PROBLEMS

When there are errors, let's display a large banner over the entire form that reads, "Please fix the problems below." This helps the user understand more immediately that the form data must be corrected. Figure 4.3 shows the form after we add the banner:

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

Please fix the problems below:

First name:

Please enter a value.

Last name:

Please enter no more than 40 characters.

E-mail address:

We won't spam you

Please enter a valid e-mail.

Figure 4.3 We want to display a banner over the form to grab our users' attention

First, let's add the error code and message to our `errors.properties` file:

```
form.problems=Please fix the problems below:
not.blank=Please enter a value.
max.length=Please enter no more than {1} characters.
email=Please enter a valid e-mail.
password.not.match=Passwords do not match.
```

Next, we add a line of code to our POST-handler `form()` method in our controller:

```
@RequestMapping(method = RequestMethod.POST)
public String form(
    @ModelAttribute(MA_USER) User user,
    BindingResult result) {

    verifyBinding(result);
    validator.validate(user, result);
    if (result.hasErrors()) {
        result.reject("form.problems");
        return "register/form";
    }
    return "redirect:confirm";
}
```

The `result` object tracks two different types of errors. When the annotation-based validation rules we set up earlier are broken, they are tracked as field errors. Additionally, `result` can track global errors, or errors related to an entire form. Here, we've manipulated `result` directly by calling `result.reject()` with a single error code, which will be interpreted as being global.

In listing 4.8 we add a bit of code to our registration form so it displays the global error:

Listing 4.8 Displaying a global error message with the Spring tag library

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %> #1
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

...

<head>
  <title>Register</title>
  <style>
    .form-item { margin: 20px 0; }
    .form-label { font-weight: bold; }
    .form-error-message { color: #900; }
    .form-error-field { background-color: #FFC; }
    .form-global-error-message {
      width: 500px;
      padding: 6px;
      background: #E2ABAB;
      color: #FFF;
      font-weight: bold;
    }
  </style>
</head>
<form:form>
  <spring:bind path="user">
    <spring:hasBindErrors name="user"> #2
      <div class="form-global-error-message"><form:errors/></div>
    </spring:hasBindErrors>
  </spring:bind>

  ... the fields and their errors ...

</form:form>

...

```

Cueballs in code and text

- #1 Import Spring tag library**
- #2 Check for global errors**

Here we've imported the Spring tag library [#1], which is a predecessor of the Spring form tag library. Among other things, it provides lower-level binding functionality than the Spring form tag library. It still supplies one bit of functionality that the Spring form tag library curiously does not: the ability to test whether or not a model object or its fields have errors associated with them [#2]. We rely on that functionality here because our global error message displays in a div with padding applied to it, and we don't want a large empty space to show up when the error isn't present.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

Discussion

Form validation helps us clean our data before passing it along to a persistent store. In this recipe we saw how to use annotations to attach validation rules to individual form fields in a convenient and intuitive way.

Our treatment handles validation in the web tier. That's a standard approach, but you should know that it's not the only one—indeed, validation frameworks nowadays (e.g., Hibernate Validator, Bean Validation Framework) tend to abandon the assumption that validation exclusively in the web tier. Some people advocate validating data in the business and data access tiers as well. This approach can provide extra assurance that bad data won't make its way into our business logic or our data stores, often with no significant performance impact.

In some cases, performing validation in the business tier (with the web tier perhaps delegating validation to the business tier) can also keep the business tier API clean. We'll discuss this further in a recipe 4.4 "brother vs. brother" sidebar.

With validation done, it's now time to look at submitting form data from the browser to the server in a secure manner.

4.3 Save user registrations

PREREQUISITES

4.1 Display a registration form

4.2 Validate user registrations

KEY TECHNOLOGIES

Spring, JPA, Hibernate 3, MySQL or other RDBMS

Background

So far we're accepting and validating user registrations as they come in, but we aren't saving them anywhere. In this recipe, we'll persist them using Spring, JPA and Hibernate.

Problem

You want to save users to a persistent store so you can load and work with them later, and you'd like to do this in a way that's compatible with Spring Security.

Solution

There are of course multiple ways in which we might store user data. Here we're going to take one perfectly standard and popular approach, which is to use Hibernate 3 ORM sitting in front of a database.

CREATE A DATABASE SCHEMA FOR STORING USERS

We'll start with a simple database schema that contains a table for users and another for their roles. We haven't discussed roles yet, but Spring Security logins require them, so we'll proceed with that in mind. Listing 4.11 shows our MySQL schema, but you should be able to get it working with your RDBMS of choice with some minor tweaking:

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

Listing 4.11 A schema that supports user registrations

```
create table users (  
  id int(10) unsigned not null auto_increment primary key,  
  first_name varchar(40) not null,  
  last_name varchar(40) not null,  
  email varchar(80) unique not null,  
  username varchar(20) unique not null,  
  password varchar(40) not null,  
  marketing_ok tinyint(1) not null,  
  accepts_terms tinyint(1) not null,  
  enabled tinyint(1) not null,  
  date_created timestamp default current_timestamp  
);  
  
create table roles (  
  user_id int(10) unsigned not null,  
  role varchar(20) not null,  
  foreign key (user_id) references users (id)  
);
```

In the `users` table, all of the columns, except `id` and `date_created`, represent our familiar `User` properties. `id` is our standard primary key, and `date_created` is a timestamp that MySQL will automatically fill in when a row is created (we don't happen to use `date_created` in our model, but it's a nice-to-have for reporting and auditing).

We're using a pretty large length for the `password` field because we're eventually going to store an SHA-1 hash in there. (We'll do this in chapter 6.)

We've also put unique constraints on the `username` and `email` columns. It's standard practice to disallow duplicate usernames in a system since they essentially function as user-friendly identifiers. For some applications duplicate e-mails are acceptable and for others they're not.

SECURITY TIP

If your system requires unique e-mail addresses, it's a good idea to require users to confirm their e-mail address (for example, by sending a confirmation e-mail to the address in question) before activating the account. Otherwise it's possible for a user to hijack somebody else's e-mail address. (I sign up as you, I'm awarded the account, now you're out of luck if you want to sign up for the same site.)

ANNOTATE THE USER MODEL FOR PERSISTENCE

As described in chapter 1, Hibernate 3 provides annotations to support the Java Persistence API (JPA) standard. Listing 4.12 below shows our `User` object annotated for persistence.

Listing 4.12 `User.java` with persistence semantics

```
package r4_3.model;
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

import org.hibernate.annotations.CollectionOfElements;
... JPA and other imports ...

@Entity
@Table(name = "users")
public class User {
    private static final String DEFAULT_ROLE = "ROLE_USER";

    private Long id;

    ... other private fields, with validation annotations ...

    public User() {
        addRole(DEFAULT_ROLE);
    }

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    @SuppressWarnings("unused")
    private void setId(Long id) {
        this.id = id;
    }

    @Column(name = "first_name") #1
    public String getFirstName() {
        return firstName;
    }

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

    ... other getter/setter pairs ...

    @Transient #2
    public String getPasswordConfirmation() {
        return passwordConfirmation;
    }

    @CollectionOfElements(targetElement = java.lang.String.class) #3
    @JoinTable(name = "roles", joinColumns = @JoinColumn(name = "user_id"))
    @Column(name = "role")
    public Set<String> getRoles() {
        return roles;
    }

    public void setRoles(Set<String> roles) {
        this.roles = roles;
    }

    public void addRole(String role) {
        roles.add(role);
    }
}

```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

    }

    public String toString() { ... }
}

```

Cueballs in code and text

- #1 Maps first name to database column
- #2 Not persistent
- #3 Hibernate annotation; collection of value types

JPA assumes all properties are persistent unless they are marked as `@Transient`, so `username`, `email`, and `password` are ready as-is. We use the `@Column` annotation to map properties to database columns [#1]. `passwordConfirmation` is marked `@Transient` [#2] because we don't want to save it to the database; we were simply using it for form validation. We've set `roles` up as collection of simple Hibernate value types, and JPA doesn't support relationships for those yet; therefore, we've dropped back to native Hibernate `@CollectionOfElements` [#3] annotation to make them persistent.

We add a default role by calling `addRole(DEFAULT_ROLE)` in the constructor. Spring Security logins require at least one role.

CREATE THE USER DATA ACCESS OBJECT

In listing 4.13 we extend the `HbnAbstractDao` we introduced in chapter 1 by adding finder methods we can use to check for existing usernames and e-mails. We've suppressed the `UserDao` interface but it's there.

Listing 4.13 HbnUserDao.java with finders to support dupe checks

```

package r4_3.dao;

import org.hibernate.Query;
import org.springframework.stereotype.Repository;

import r4_3.model.User;

@Repository("userDao")
public class HbnUserDao extends HbnAbstractDao<User> implements UserDao {

    @SuppressWarnings("unchecked")
    public User findByUsername(String username) {
        Query q = getSession().createQuery("from User where username = ?");
        q.setParameter(0, username);
        return (User) q.uniqueResult();
    }

    @SuppressWarnings("unchecked")
    public User findByEmail(String email) {
        Query q = getSession().createQuery("from User where email = ?");
        q.setParameter(0, email);
        return (User) q.uniqueResult();
    }
}

```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```
}  
}
```

To detect duplicate usernames and e-mails, it's not sufficient to rely on the unique constraints we set up for the `username` and `email` table columns. While Hibernate will throw a `ConstraintViolationException` if we attempt to insert duplicates, we wouldn't easily be able to tell which constraint was violated. `ConstraintViolationException` provides a `getConstraintName()` method, but it is unreliable across different RDBMSes and always returns `null` for the standard MySQL driver. Additionally, Hibernate would throw the exception upon the first constraint violation that occurred, so if both the supplied username and e-mail address were duplicates, we'd only be able to report one at a time.

Now let's create a user service around the user DAO.

CREATE THE USER SERVICE

We are going to create a service with a method that contains the business logic we need to register users. It will check for duplicate usernames and e-mail addresses and then persist User objects to the database. In this service's design, we assume that most validation is handled externally. The exceptions are cases where validation is "costly;" we handle those in the service bean.

Recall from chapter 1 that we don't need to explicitly flush and close our Hibernate sessions so long as they execute within the context of a transaction. Instead of specifying transaction boundaries at the DAO level, it makes more sense to do so at the service level since service methods often rely on multiple database operations. Listing 4.14 shows our service implementation (we've suppressed the `UserService` interface):

Listing 4.14 `UserServiceImpl.java`, a service bean for registrations and more

```
package r4_3.service;  
  
... imports ...  
  
@Service("userService")  
@Transactional(  
    propagation = Propagation.REQUIRED,  
    isolation = Isolation.DEFAULT,  
    readOnly = true)  
public class UserServiceImpl implements UserService {  
  
    @Autowired  
    private UserDao userDao;  
  
    public void setUserDao(UserDao userDao) {  
        this.userDao = userDao;  
    }  
  
    @Transactional(readOnly = false)  
    public void registerUserIfValid(User user, boolean userIsValid)  
        throws DuplicateRegistrationException {  
        boolean usernameTaken =
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

        userDao.findByUsername(user.getUsername()) != null;           #A
    boolean emailTaken =
        userDao.findByEmail(user.getEmail()) != null;               #B

    if (usernameTaken || emailTaken) {
        throw new DuplicateRegistrationException(
            usernameTaken, emailTaken);                             #1
    }

    if (userIsValid) {                                             #2
        userDao.save(user);
    }
}
}

```

Cueballs in code and text

#A Check whether username is taken
#B Check for duplicate e-mail
#1 Throw exception if either is taken
#2 Save only if validation succeeds

In `registerUserIfValid()` we throw a `DuplicateRegistrationException` if we detect either a duplicate username, e-mail address or both [#1]. Then only if our external validation passed do we persist the `User` object to the database [#2]. You can download the code for `DuplicateRegistrationException` from the book's website.

The fact that `registerUserIfValid()` relies on an external flag makes it sort of an odd bird. Developers using our API in different contexts, perhaps independently of Spring Web MVC, wouldn't necessarily understand the circumstances we faced when we designed it the way we did. Therefore, if we were designing this API for public consumption, it would make sense to Javadoc it well and add an alternative method, `registerUser()`, that saved `Users` unconditionally. We'll leave all that out for our purposes.

Brother vs. brother: Validation in the business tier

In recipe 4.2 we noted that while we're doing our validation in the web tier, it's also possible to perform validation in the business and data access tiers. (It is even routinely done in the persistence tier via database constraints.) We also said that validation in the business tier can simplify the business tier API. Let's consider that in the context of our `UserService` interface.

We have the `userIsValid` flag and `DuplicateRegistrationException` because `UserService` is not itself validation-aware. The `userIsValid` flag essentially allows us to pass in contextual information that tells the service bean whether the registration request is "real," and the exception allows the service bean to signal validation errors that aren't captured by BVF validation rules. (We *could* externalize the explicit dupe checks to allow the client to call

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

them directly before calling a simpler register() method, but it's nice to keep the checks and registration in the same transaction without requiring extra work of the client.)

John prefers to keep validation out of the business tier because he thinks we should ensure the data is clean before passing it to the service, and he doesn't think the service API should expose the Spring-specific Errors interface.

Willie thinks that this situation is ideal for business tier validation, even if it means exposing Errors, since it eliminates the flag and exception, and it presents "lightweight" (BVF) and "heavyweight" (database-calling) rule violations uniformly to the client.

VOKE THE USER SERVICE FROM REGFORMCONTROLLER

Now we are ready to wire UserService into our aptly-named registration form controller, RegFormController, and register users. Listing 4.15 shows how this we do this:

Listing 4.15 RegFormController.java with persistence code

```
package r4_3.web;

... imports ...

@Controller
@RequestMapping("/register/form")
public class RegFormController {
    private static final String MA_USER = "user";

    private Validator validator;
    private UserService userService;

    @Autowired
    public void setValidator(Validator validator) {
        this.validator = validator;
    }

    @Autowired
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public void form(Model model) {
        Logger.global.info("Serving initial form");
        model.addAttribute(MA_USER, new User());
    }

    @RequestMapping(method = RequestMethod.POST)
    public String form(
        @ModelAttribute(MA_USER) User user,
        BindingResult result) {

        verifyBinding(result);
        validator.validate(user, result);

        try {
```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

        userService.registerUserIfValid(user, !result.hasErrors()); #2
    } catch (DuplicateRegistrationException dre) {
        if (dre.isUsernameTaken()) {
            result.rejectValue("username", "not.unique",
                new String[] { "Username" }, null); #3
        }
        if (dre.isEmailTaken()) {
            result.rejectValue("email", "not.unique");
        }
    }

    if (result.hasErrors()) {
        result.reject("form.problems");
        return "register/form"; #4
    }

    return "redirect:confirm"; #5
}

... initBinder() and verifyBinder() from recipe 4.2 ...
}

```

Cueballs in code and text

- #1 Validate with BeanValidator**
- #2 Pass in external validation results**
- #3 Signal a duplicate username**
- #4 If invalid, redisplay form with errors**
- #5 Otherwise redirect to confirmation page**

After we validate users with `BeanValidator` [#1], we call `registerUserIfValid()` on our service while passing the flag indicating whether or not validation was successful [#2]. If a `DuplicateRegistrationException` is thrown during the service call, we directly add error codes to the `BindingResult` [#3] then we either redisplay the form with errors [#4] or forward to our confirmation page [#5]. We handle the confirmation page with a separate controller that isn't shown here.

We also need to add the `not.unique` error code to our `errors.properties` message bundle:

```
not.unique={0} has already been taken.
```

APPLICATION CONTEXT CONFIGURATION

In listing 4.16 we present `spring-servlet.xml`, which is our application context file. We won't go into the details of the configuration here since this will to a large extent be a repetitive element throughout the book. We'll show it here, however, since this is our first recipe using ORM persistence. For more information on the configuration please see part 1.

Listing 4.16 `spring-servlet.xml`, our application context

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context
    [CA]/spring-context-2.5.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

  <context:property-placeholder
    location="file:${user.home}/springbook.properties"/>

  <context:component-scan base-package="r4_3.dao"/>
  <context:component-scan base-package="r4_3.service"/>
  <context:component-scan base-package="r4_3.web"/>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${dataSource.driverClassName}"
    p:url="${dataSource.url}"
    p:username="${dataSource.username}"
    p:password="${dataSource.password}"/>

  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.
      [CA]AnnotationSessionFactoryBean"
    p:dataSource-ref="dataSource">

    <property name="annotatedClasses">
      <list>
        <value>r4_3.model.User</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.
          [CA]MySQL5InnoDBDialect</prop>
        <prop key="hibernate.show_sql">>true</prop>
      </props>
    </property>
  </bean>

  <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.
      [CA]HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory"/>

```

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

<tx:annotation-driven/>

... internal resource view resolver (see recipe 4.2) ...
... configuration loader (see recipe 4.2) ...
... error code converter (see recipe 4.2) ...
... validator (see recipe 4.2) ...
... message source (see recipe 4.2) ...
</beans>

```

The configuration above uses Hibernate 3 and MySQL 5. If you're not using those, you'll need to consult a reference manual to wire it up with your ORM (assuming you're using one) and DBMS of choice.

PROVIDE A LINK TO AN ACCOUNT RECOVERY PAGE WHEN DUPLICATE E-MAILS ARE DETECTED

Let's display a link to an account recovery page if users submit a duplicate e-mail, as it probably means they registered with us a long time ago and forgot about it. We can take advantage of the fine-grained validation message we set up during the validation section by adding the following line to `errors.properties`.

```

not.unique.user.email=We have this e-mail on file. Have you
<a href="#">lost your username or password</a>?

```

The link is just a placeholder, and Spring form tag's `form:errors` won't display it properly unless we turn off HTML escaping for e-mail in `form.jsp`.

```

<div class="form-error-message">
  <form:errors path="email" htmlEscape="false"/>
</div>

```

In figure 4.4 we see the end result:

E-mail address:

We won't spam you
 We have this e-mail on file. Have you [lost your username or password?](#)

User name:

Username has already been taken.

Figure 4.4 We allow users to recover their account information if they enter a duplicate e-mail address.

With that, we have a fully-functional registration form. Congratulations! From an end user perspective, everything works as expected: he can register and (once logins are in place) he'll be able to use the registration to log in.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

We're not completely done yet, though. So far we're transmitting and storing passwords in the clear, and that's usually a bad idea for production applications. In chapter 6 we'll continue with the security theme by showing how to handle passwords more securely. There are other security measures we might take as well, as we'll see in recipe 4.4.

Discussion

In this recipe, we've checked for duplicate usernames and e-mail addresses and saved user registrations to a database. Sometimes, however, it is preferable to nix the `username` field altogether in favor of using `email` and `password` as security credentials. People are generally more likely to forget their usernames than their e-mail addresses, especially if the user was forced to choose a second-choice username because his usual was already taken.

A downside of using `email` as an identifier is that doing so might violate a user's privacy. What if someone who knew the e-mail address tried to use it to register with our website, just to see whether an account already exists? We'd have to report its unavailability. In many cases this disclosure would be unacceptable; examples include adult websites, credit repair websites, or websites where there are legal reasons (FERPA, HIPAA, etc.) why such information must be tightly held. In cases like that, you might want to stick with usernames, and you might even advise the user to avoid usernames that create privacy concerns.

4.4 Use CAPTCHAs to prevent automated user account creation

PREREQUISITES

- 4.1 Display a registration form
- 4.2 Validate user registrations

KEY TECHNOLOGIES

Spring Web MVC, Spring Validation, reCAPTCHA, reCAPTCHA4j

Background

It's well-understood that computers and people are good at solving different—and largely complementary—problems. While that's probably a statement with a shelf life, at the time of this writing (2008) Skynet⁴ appears to be at least four or five years away.⁵ It turns out that we can use this difference in capability to prevent automated agents, or "bots," from performing actions that we want only real people to do, such as filling out user registration forms. Enter the "Completely Automated Public Turing test to tell Computers and Humans Apart," or CAPTCHA.

Here's how it works. Suppose that we have a registration form and we want to ensure that only real people complete the form. We add to the form a "problem"—or a CAPTCHA—

⁴ [http://en.wikipedia.org/wiki/Skynet_\(Terminator\)](http://en.wikipedia.org/wiki/Skynet_(Terminator))

⁵ Don't laugh. It's already possible to use the Amazon Mechanical Turk to orchestrate armies of human components to solve problems that computers can't directly solve themselves. While I don't necessarily expect MTurk to evolve into Skynet, it's hard not to be impressed with how well computers are competing against the human race.

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

that all (or most) and only real human beings can easily solve, and we don't accept the form submission unless the problem is actually solved. That allows us to be sure that a real human being completed the form.

The most common type of CAPTCHA asks the user to look at some visually distorted text and enter that text into a text field in the form. You've probably run into those many times on the web.

In this recipe we're going to look at an interesting, socially-conscious variant to that approach from Carnegie Mellon University called reCAPTCHA. ReCAPTCHA⁶ was conceived as a way to use CAPTCHAs to digitize books. When optical character recognition (OCR) systems scan books, they often come across words that they can't read. ReCAPTCHA collects OCR-unrecognizable words and presents them to human beings as two-part CAPTCHAs: one known word and one OCR-unrecognizable word. The known word gives reCAPTCHA a way to determine whether there's really a human on the other end since by definition the other word can't be directly validated. To solve the CAPTCHA (which is the end user's concern), the human only has to get the known word right. To solve the unknown word (which is CMU's concern), enough humans have to agree on its proper translation. Of course, the user isn't told which word is the known word. And most users are blissfully unaware the distinction between the known and unknown words anyway.

Problem

You have a user registration form, and you want to make sure that only real human beings can use it to create accounts.

Solution

We are going to add a reCAPTCHA widget to our registration form. Figure 4.5 shows how it looks:



Figure 4.5 This is a reCAPTCHA widget. ReCAPTCHA allows you to load a new pair of words if you can't read the current pair, and it also provides audio CAPTCHAs for blind users.

⁶ <http://recaptcha.net/>

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

As you can see, the user interface is nice and polished. It's also possible to select different themes for the widget though we won't go into that here.

We will continue to use Spring Web MVC here, along with the Spring form tag library.

CREATE AN ACCOUNT AND KEY PAIR

The first step is to log into <http://recaptcha.net/> and sign up for a reCAPTCHA account. You'll be asked to specify the domain your reCAPTCHAs will service. After providing that, the reCAPTCHA website will provide you with a key pair (which they call a "reCAPTCHA key" even though it's really a key pair) that you can use for requesting reCAPTCHA challenges and validating user responses. The key pair includes a public key you use to request the widget, and a private key you use when forwarding user responses to the reCAPTCHA server.⁷

Now let's look at how to add reCAPTCHA to a Spring application.

PLACE A RECAPTCHAIMPLEMENT BEAN ON THE APPLICATION CONTEXT

The `ReCaptchaImpl` class, which comes from the reCAPTCHA4j client library, allows us to request challenges from a remote reCAPTCHA server, and to check with the same whether the user's response to the reCAPTCHA challenge was correct. We need to inject it with our public key so we can request challenges from the reCAPTCHA server, and with our private key we can validate user responses. Because we might want to use `ReCaptchaImpl` in multiple places (not just the registration form), and because we don't want to duplicate our keys (especially our private key) everywhere, we're going to place a `ReCaptchaImpl` bean on the application context.

```
<bean id="reCaptcha"
      class="net.tanisha.recaptcha.ReCaptchaImpl"
      p:publicKey="YOUR_PUBLIC_KEY"
      p:privateKey="YOUR_PRIVATE_KEY"/>
```

`ReCaptchaImpl` is threadsafe by design, so it's OK to use a shared single instance in this fashion. As is obvious, replace `YOUR_PUBLIC_KEY` and `YOUR_PRIVATE_KEY` with the keys you received when you created your reCAPTCHA account.

That's it for the Spring application context. Now let's go to the controller.

UPDATE THE CONTROLLER

We'll need to modify the controller a little to make reCAPTCHA work. First we'll have to add a setter for the reCAPTCHA bean from reCAPTCHA4j to support dependency injection. I won't even show the code here; I'll just recommend that you follow good API design practice and code the setter to the `ReCaptcha` interface rather than to the `ReCaptchaImpl` implementation.

Next let's update the form-serving method. We show the reCAPTCHA widget by embedding some boilerplate HTML and JavaScript into our web form. The only non-

⁷ From the reCAPTCHA website: "This key pair helps to prevent an attack where somebody hosts a reCAPTCHA on their website, collects answers from their visitors and submits the answers to your site." An attacker would need your private key to check responses for a reCAPTCHA generated for your public key.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

boilerplate pieces would be the public key you use to identify your reCAPTCHA account and any customization options you want to include. Conveniently, we can use the reCAPTCHA bean to generate the HTML and JavaScript for us. We make both the user account and the reCAPTCHA HTML available to the form JSP by placing them on the model:

```
private static final String MA_USER = "user";
private static final String MA_RECAPTCHA_HTML = "reCaptchaHtml";

...

@RequestMapping(method = RequestMethod.GET)
public void form(Model model) {
    model.addAttribute(MA_USER, new User());
    String html = reCaptcha.createRecaptchaHtml(null, null);
    model.addAttribute(MA_RECAPTCHA_HTML, html);
}
```

Though you can't see it from the Java code (you can see it if you view the HTML page source though), the reCAPTCHA bean generates HTML and JavaScript to pull a reCAPTCHA widget down from the reCAPTCHA servers using your public key. We just put that HTML/JavaScript on the model so the JSP can incorporate it. The first argument to `createRecaptchaHtml()` allows us to indicate whether a validation error happened (the answer is no when we're first serving the form, so we pass in `null`). The second argument allows us to specify some properties to customize the widget, but we aren't doing that here. Consult the reCAPTCHA Javadocs for more information.

Third, update the `initBinder()` method to allow parameters named `recaptcha_challenge_field` and `recaptcha_response_field`. We're going to bind the corresponding incoming data not to the user model but to some request parameters with those names.

The fourth and final controller modification is to update the form-processing method. This one's actually fairly interesting so let's take a good look.

The reCAPTCHA widget is itself able to display CAPTCHA validation errors when they occur, so first, listing 4.17 shows how to perform form validation (the whole form, including the CAPTCHA) and display CAPTCHA validation errors in the widget itself:

Listing 4.17 The first version of our reCAPTCHA validation code

```
@RequestMapping(method = RequestMethod.POST)
public String form(
    HttpServletRequest req,
    @RequestParam("recaptcha_challenge_field") String challenge,           #1
    @RequestParam("recaptcha_response_field") String response,           #2
    @ModelAttribute(MA_USER) User user,
    BindingResult result,
    Model model) {

    verifyBinding(result);
```

Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=503>

```

        validator.validate(user, result); #3
        ReCaptchaResponse reCaptchaResponse =
            reCaptcha.checkAnswer(req.getRemoteAddr(), challenge, response); #4
        if (result.hasErrors() || !reCaptchaResponse.isValid()) {
            String errMsg = reCaptchaResponse.getErrorMessage(); #5
            String html = reCaptcha.createRecaptchaHtml(errMsg, null); #6
            model.addAttribute(MA_RECAPTCHA_HTML, html);
            result.reject("form.problems");
            return "register/form";
        }

        ... save user ...

        return "redirect:confirm";
    }

```

Cueballs in code and text

- #1 Code representing challenge delivered to user**
- #2 User's response to challenge**
- #3 Validate non-CAPTCHA form data**
- #4 Validate CAPTCHA response**
- #5 Get error message, if any**
- #6 Generate widget HTML/script**

The method accepts, among other things, a couple of reCAPTCHA-specific parameters: `recaptcha_challenge_field` [#1] and `recaptcha_response_field` [#2]. The widget sends values for these, and we use them to validate the user's response.

The code begins with a call to `validate()` [#3] to perform a general form validation. This validates everything except the CAPTCHA itself. The next piece uses the reCAPTCHA bean to submit the user's response to the reCAPTCHA servers [#4]. Recall that the reCAPTCHA bean knows the site's private key, and it submits that as part of the CAPTCHA validation call.

If either the account data or else the CAPTCHA is invalid, we grab an error code from the response (it's really more code-like than message-like, despite the name `getErrorMessage()`) [#5] and pass it back to the reCAPTCHA bean when we generate the HTML and JavaScript we use for getting the widget [#6]. That way the widget knows what error message to display.

Otherwise we save the user account and redirect to a confirmation page using the `redirect-after-post` pattern to prevent double submissions.

The approach just described works, but there's a usability issue to deal with. The reCAPTCHA error message is small, embedded in the widget, and just doesn't look like the error messages for the other fields. It's easy to miss, and it's inconsistent.

USABILITY TIP

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

Jakob Nielsen, a well-known usability expert, notes that being consistent is a general principle of usable user interface design. Of consistency and standards he says: "Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions."⁸

To improve the usability of the form, let's make the error message for the CAPTCHA field look like the other ones. Here's listing 4.18:

Listing 4.18 The second and improved version of our reCAPTCHA validation code

```
@RequestMapping(method = RequestMethod.POST)
public String form(
    HttpServletRequest req,
    @RequestParam("recaptcha_challenge_field") String challenge,
    @RequestParam("recaptcha_response_field") String response,
    @ModelAttribute(MA_USER) User user,
    BindingResult result,
    Model model) {

    verifyBinding(result);
    validator.validate(user, result);
    RecaptchaResponse reCaptchaResponse =
        reCaptcha.checkAnswer(req.getRemoteAddr(), challenge, response);

    if (!reCaptchaResponse.isValid()) {
        result.rejectValue("captcha", "errors.badCaptcha");      #1
    }

    String html = reCaptcha.createRecaptchaHtml(null, null);      #2
    model.addAttribute(MA_RECAPTCHA_HTML, html);

    if (result.hasErrors()) {
        result.reject("form.problems");
        return "register/form";
    }

    return "redirect:confirm";
}
```

Cueballs in code and text

- #1 Flag captcha property as having error
- #2 Generate widget HTML/script, no error

From a Spring perspective this code is noteworthy because it shows how to combine Validator-based validation with manual validation (in this case, manual reCAPTCHA validation). Here, instead of displaying an error message through the reCAPTCHA, we call

⁸ http://www.useit.com/papers/heuristic/heuristic_list.html

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

`binding.rejectValue()`, passing in both the name of the offending property and the error code [#1]. Then we generate the reCAPTCHA HTML, but we pass in `null` for the first argument [#2] since we don't want the widget itself to display an error message; the registration form itself will do that.

You might have noticed that we don't actually have a property called `captcha`; we'll turn to that issue right now.

ADD A DUMMY GETTER TO THE FORM BEAN

To display error messages for a given property, there must be an associated getter method in the model. Hence we just add a dummy `getCaptcha()` method to `User`:

```
public String getCaptcha() { return null; }
```

This is of course nothing more than a hack to make it work, but it's probably the most straightforward way to display the error message in HTML.

UPDATE YOUR MESSAGE RESOURCE BUNDLE

To define an error message to display, open your `errors.properties` file and add a single line:

```
errors.badCaptcha=Please try again. Hover over the red buttons for  
[CA]additional options.
```

We're almost there. Now all we need to do is update the JSP and we're set.

UPDATE YOUR REGISTRATION FORM

Just add the reCAPTCHA widget and the error message to your JSP:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  
  
...  
  
<div class="form-item">  
  <div class="form-label">Please prove you're a person:</div>  
  <div>${reCaptchaHtml}</div>  
  <div class="form-error-message"><form:errors path="captcha"/></div>  
</div>
```

We're using the Spring form tag library to display the error message.

USABILITY TIP

Many users won't have any idea why they're being asked to respond to a CAPTCHA, and the widget doesn't itself make it obvious what's going on. It is helpful to include a short explanation, such as "Please prove you're a real person" as we've done above.

FIRE IT UP

Point your browser at the form and try it out. You should see your reCAPTCHA widget. Try submitting a blank form. You should see error messages for your form fields, including an

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>

error message for the reCAPTCHA widget rendered in the same style as your other error messages. You might view the page source to see what it is that the reCAPTCHA bean actually generates, in terms of HTML and JavaScript.

TIP

Note that even though reCAPTCHA normally expects your reCAPTCHA requests to originate from the domain you specified when creating your account, it won't complain if your requests come from `localhost` or `127.0.0.1`. (If however somebody tries to request a reCAPTCHA widget using your public key from some alien domain, the reCAPTCHA service will not respond with the widget.)

Good job! Score: human race 1, Skynet 0.

Discussion

CAPTCHAs aren't of course inherently tied to registration forms. They are useful any time you want to keep bots from submitting web forms. Examples beyond registration forms would include "contact us" forms, anonymous discussion forum postings, user comments, product reviews and so forth.

From a usability perspective, reCAPTCHA is something of a mixed bag. For end users, completing CAPTCHAs is a necessary evil; they don't derive any direct benefit from it. So the fact that reCAPTCHA requires that the user translate *two* words instead of just one means that we've doubled the nuisance factor. On the other hand, as noted above, reCAPTCHA does allow users to request a new challenge if the existing challenge is too difficult, and it does support accessibility by allowing audio CAPTCHAs as an alternate.

While reCAPTCHA is one of the nicer CAPTCHA solutions available (and arguably the one with most important social impact), it isn't the only one out there. You might look at Jcaptcha as well, which like reCAPTCHA integrates with Spring:

- <http://jcaptcha.sourceforge.net/>

Summary

At the beginning of the chapter we claimed that registration isn't just a matter of collecting form data and saving it. We hope you can see what we mean. We've highlighted functional, usability, security, legal and engineering concerns that registration forms can address either more or less successfully, and we've shown how to use Spring and other tools to tackle those issues effectively. At the end of the day, we want forms that are easy to use and that protect the private data that they capture, and Spring certainly provides a nice framework for accomplishing those ends.

In the next chapter we'll turn our attention to registration's partner in crime, which would be authentication. As you might expect, Spring Security will play an even more prominent role in that discussion.

Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=503>