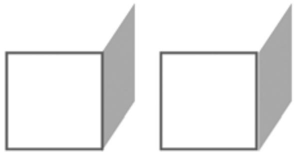
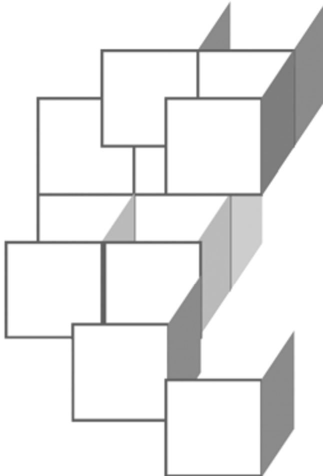
Chapter 3

URL Authorization





OBJECTIVES

After completing "URL Authorization," you will be able to:

- **Develop fine-grained authorization policies over your web application's request URLs.**
- **Use the Spring Security API to discover information about the current user, and to make programmatic authorization decisions.**
- **Show pieces of user interface conditionally, based on authority.**

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

URL Authorization

- **The first and most common sort of authorization policy for web applications requires that the user be in one of a list of roles in order to get a response for a certain request URL.**
- **In Spring Security we do this with the `<intercept-url>` element.**
  - The **pattern** attribute identifies one or more URLs to be protected. This is interpreted by a pluggable pattern-matching algorithm.
  - By default, patterns are **Ant-like**: they can be single URLs or masks using wildcards `*` (any token) and `**` (any path).
  - They are **context-relative**, so should use a leading `/` character.
  - We can further narrow the constraint by specifying an HTTP **method**, such as "GET" or "POST".
  - Then the **access** attribute identifies one or more roles: the user must be playing at least one of these, to avoid the dreaded 403.
  - The **filters** attribute allows one or more filters in the configured filter chain to be dropped out for this URL pattern. In practice this is generally either left out or it's "none" to open all access.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

URL Authorization

- **`<intercept-url>`s are processed in the order of declaration**
  - **First match** sets the rules, and the rest are ignored.
  - This is another difference from servlet-container authentication, where there are more complex rules about what constraints override what others, and the general idea is order doesn't matter and specific patterns should trump general ones.
  - In Spring Security, it's entirely up to you to put things in the proper order, as a general pattern will take precedence over a more specific one that follows – and this is usually not desired.
  - Then, two general-purpose patterns might overlap – consider the following – and it's a design question which one should go first.

```
/admin/**
/**/*.xml
```

- **By default, role names must all begin with "ROLE_".**
  - This is a common pitfall in working with Spring Security, as developers will often assume that any unique string is fine and define roles such as "user", "admin", "member", etc.
  - Then everyone gets a 403, and it can take some digging and trial-and-error to discover the root cause.
- **This can be configured, by a couple of strategies, and we'll consider this further in the following chapter.**

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Who Goes There?

EXAMPLE

- In **Examples/Login/Step5**, we have a few specific page constraints and a catch-all – see **docroot/WEB-INF/applicationContext.xml**.

```
<http>
  <intercept-url pattern="/login.jsp"
                 filters="none" />
  <intercept-url pattern="/index.jsp"
                 access="ROLE_ANONYMOUS" />
  <intercept-url pattern="/protected.jsp"
                 access="ROLE_ADMIN" />
  <intercept-url pattern="/**"
                 access="ROLE_USER"/>
  ...
```

- We've seen the results of this policy:
  - Any **user** can see the index page and the login page – though by two mechanisms, filter bypassing vs. anonymous authentication.
  - Any **authentic user** (ROLE_USER) can see those plus **home.jsp**.
  - An **administrator** (ROLE_ADMIN) can see everything.
- You can give this a quick try now, if you like:
  - Login as **friend/trustme** and try to visit the protected page ...

Authorization Failure

You are not authorized to view the requested page.

  - Login again as **overlord/fearme** and get through.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

The Healthcare Case Study

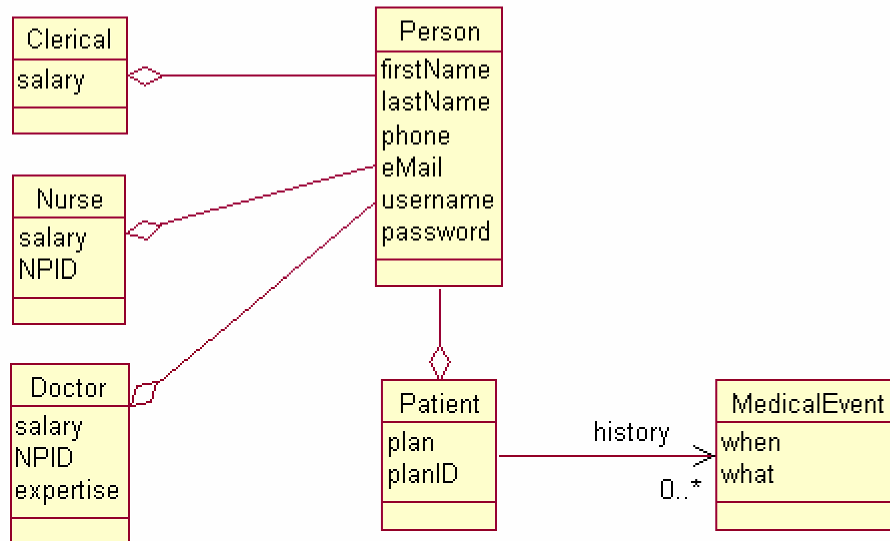
- **We've worked with the Health application a bit already, but as we get into implementing a full authorization policy (which you're about to do in the upcoming lab), it's worth a little time to get more familiar with its functionality.**
- **The Healthcare application implements a handful of functions for a healthcare system.**
- **It has a database of doctors, nurses, patients, and clerical staff – all of whom are potential users of the application as well.**
  - **Java Persistence API** (**JPA**) entities are mapped to this database and form the primary domain model.
- **Main use cases are as follows, and are implemented by servlets and JSPs:**
  - Users can **modify their profiles**, and this includes changing their usernames and passwords.
  - Users can **make appointments** between doctors and patients.
  - Doctors, nurses, and a patient himself or herself can view the patient's **medical history**.
  - Doctors can **prescribe medication**.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Domain Model

- Here is a UML class diagram representing the domain model:



  - Not shown is the **id** field for each entity, which maps to a generated primary key value in the database.

- At the database level, the schema also includes a SQL VIEW that presents each of the **Doctor**, **Nurse**, **Clerical**, and **Patient** types as role names to Spring Security.

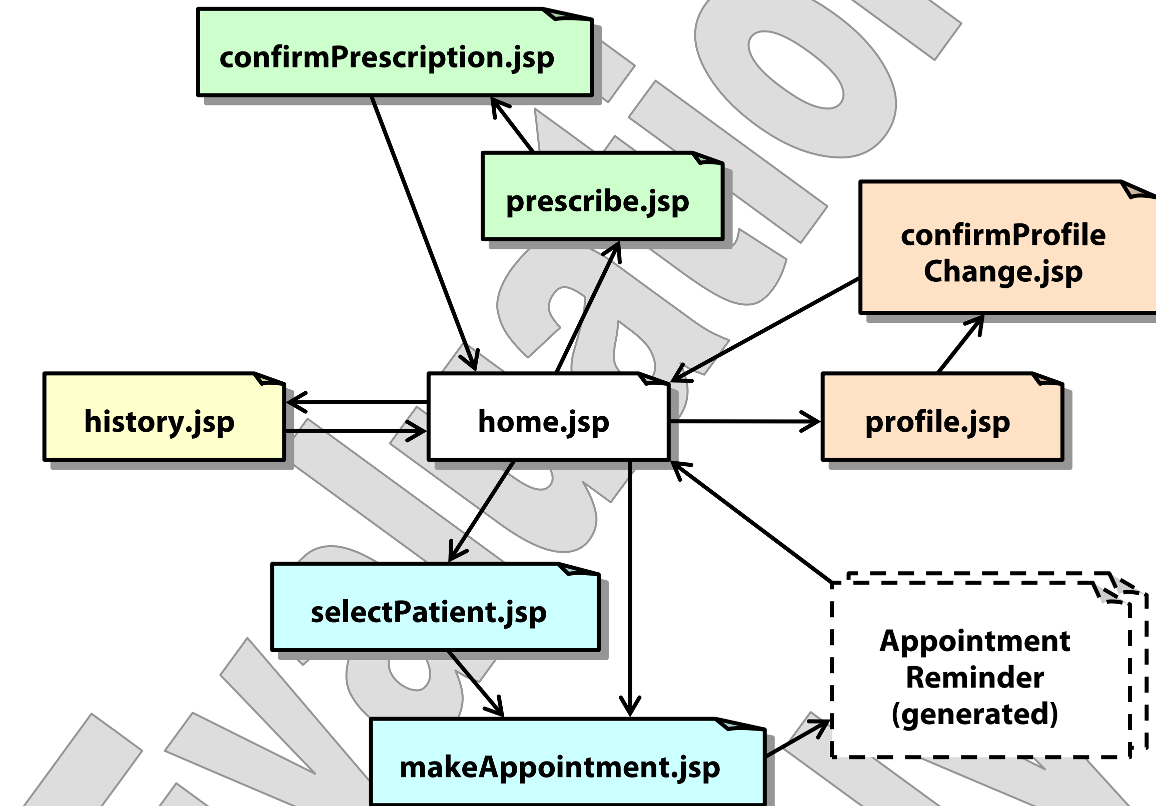
```
ROLE_PATIENT
ROLE_DOCTOR
ROLE_NURSE
ROLE_CLERICAL
```

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Page-Flow Model

- The user can carry out those four use cases, each by clicking a link on the home page and following a short page flow that resolves back to home:



  – Not shown are various servlets and filters that manage information flow between pages.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Roles and Privileges

- **Only certain roles are appropriate to some functionality:**
  - Anyone can see and change their own **user profile**.
  - Anyone can **make an appointment**; a patient should only be able to make an appointment for himself or herself.
  - Doctors and nurses should be able to see **medical histories**; and patients should be able to see their own, but not others'.
  - Only doctors should be able to **prescribe medication**.
- **More subtly, the application should behave differently depending on the role(s) assigned to the interactive user.**
  - If the user is a patient, then the **choice of patient** for showing medical history or making an appointment should be implicit; if the user is a doctor, nurse, or clerical worker, he or she should be given a chance to select a patient.
  - This switch is implemented for medical history: a servlet checks the logged-in user and determines whether the **selectPatient.jsp** should be shown before moving on to **showHistory.jsp**.
  - It is not yet implemented for appointments; that will be one goal of an upcoming lab.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Healthcare Users: a Cheat Sheet

- As you work on the upcoming lab, you will need to log in variously as different patients, doctors, nurses, and staff.
- There are about 50 total users in the database, but below are a small set of logins that you can use for your testing:
  - Patients:

```
Lesa_Griffith/globalVILLAGEidiot
beard/password
burgess/CAVEman
```

  - Doctors:

```
dtravis/sadlyECSTATIC
feelgood/mrsmd
church/hEoTpEeRsNpArLings
```

  - Nurses:

```
Nola_Mullins/miffmufferedmoof
Edgar_Frank/OFTENTIMES889 (note: also a patient)
ratched/meanie
```

  - Clerical staff:

```
hebert/invisible^JUMP!SUIT
Larry_Best/if_you_cAn_guess+THIS
                           (note: also a patient)
martinez/CavesOfAltamira
```

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Healthcare URL Authorization

LAB 3A

Suggested time: 30 minutes

In this lab you will build a fine-grained authorization policy for the Health application, based on request URLs and the requirements stated earlier. You will write out `<intercept-url>`s to express specific constraints and allow doctors, nurses, clerical staff, and patients variously to access those URLs.

Detailed instructions are found at the end of the chapter.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Programmatic Authorization: Servlets

- Everything we've seen so far is **declarative**, and this is a selling point: we state the security requirements for an application, and Spring Security enforces them.
- There is also a **programmatic** approach, which can be mixed and matched with the declarations we've seen already.
- For this purpose, **HttpServletRequest** offers a few methods that are easy to use and well-known to most servlet programmers:

```
public interface HttpServletRequest
{
  public Principal getUserPrincipal ();
  public String getRemoteUser ();
  public boolean isUserInRole (String roleName);
}
```

- Using these methods you can get
  - The principal record itself – which may be of any subtype of **Principal** and offer any amount of information specific to the authentication system, but will always provide paths to username and assigned roles
  - The primary name of the authenticated **user**
  - Whether that user is in a desired **role**
- It is <u>not</u> necessary to refactor existing servlets code to adapt to Spring Security once you've put it in place.
  - Rather, Spring Security adapts to the servlets standard: one filter in the chain wraps the container-built request object in one that will implements these methods to use the Spring Security API.

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Evaluation Only

Programmatic Authorization: Spring Security

- **New applications that don't mind developing some code-level dependencies on Spring Security can use the API directly.**
- **There are several avenues of approach – and now the drumbeat for Chapter 4 and a deeper study of the API is getting pretty loud! – but the simplest of these are as follows.**
- **To get the current user – this is equivalent to getPrincipal or getRemoteUser – use the following incantation:**

```
SecurityContextHolder.getContext ()
  .getAuthentication ().getPrincipal ()
```

  - This gives you an object which you can downcast to the type **Authentication** and then query for username, roles, etc.
  - The security context is **thread-affinite**, so you don't need a servlets-level request object, or anything else.

- **To check for roles/authorities more directly, try this:**

```
AuthorityUtils.userHasAuthority ("ROLE_SOMETHING")
```

  - This returns a boolean and is by far the easiest way to do programmatic security with the API.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

When to Use Programmatic Authorization

- **Declarative security is preferred whenever it can precisely express the authorization policy.**
- **Programmatic security steps in when more fine-grained decisions must be taken:**
  - If credentials are drawn from a larger body of persistent domain objects, the application may well want to **determine who is logged in**, so as to represent information about that user without having to ask twice.
  - **Different behavior** might occur based on user identity or role. For example, we might want to show certain UI content, or go to entirely different pages, based on role.
  - Sometimes an HTTP request will trigger a **complex business process**, only some of which would be carried out if the user were not in a sufficiently powerful role.
  - Different **error handling** might obtain for administrators than for ordinary users.
- **Many applications combine declarative and programmatic techniques:**
  - Declare security constraints that, by themselves, might allow unauthorized access or functionality.
  - Weed out those remaining cases programmatically.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Role-Based Presentation

- **It's rather rude to invite the user to click a button or a link that would request a URL they can't actually request, and then serve them up a 403 error message.**
  - Ideally, such errors should only appear in cases of forceful browsing, where the user explicitly requests a specific URL.
- **Better to guide the user only to choices that he or she is in fact authorized to make.**
- **This means showing certain pieces of a UI conditionally, based on the authenticated user's role assignments.**
- **This can be a little bit of a chore in Java EE.**
  - For one thing, the **isUserInRole** method isn't JSP-friendly: we can't write a JSP expression to test that. (If there were a map of the user's **roles**, we'd be in great shape ... but, no.)
  - Also, in order to solve the problem generally, one must have some enumeration of all the roles supported by the application – again, **isUserInRole** won't volunteer information.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

The Spring Security Tag Library

- **To the rescue comes a small tag library, distributed with Spring Security, that gives JSPs some programmatic security capabilities.**
- **Declare this tag library as follows (we use the prefix sec: but as with all tag libraries the prefix is up to you):**

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

- **Use the `<sec:authorize>` tag to wrap content that should be produced only if the user has a certain role or certain roles.**

```
<sec:authorize ifAnyGranted="ROLE_ADMIN" >
  <p>The secret monitoring data is ${mon.data}</p>
</sec:authorize>
```

  - The **ifAnyGranted** attribute lists one or more roles, any of which, if granted, will cause the tag to produce its body content.
  - You can also specify **ifAllGranted** to require multiple roles.
- **You can also show content in case the user is <u>not</u> authorized to a certain function – perhaps some static text in replacement for the dynamic content you would otherwise be showing.**
  - Use the **notGranted** attribute, which will cause content to be shown if <u>none</u> of the listed roles are in force.
- **There is also a tag that gives direct access to properties of the Authentication object representing the current user:**

```
<sec:authentication property="principal.username"/>
```

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Evaluation Only

Programmatic Security for Healthcare

LAB 3B

Suggested time: 30 minutes

In this lab you will add role-based presentation to the home page of the Health application, guiding the user only to the functions to which he's authorized, using the Spring Security tag library.

You'll also review some programmatic security already implemented in the medical history feature, and fix one remaining piece of code so that the page flow for staff vs. non-staff users is different when making appointments.

Detailed instructions are found at the end of the chapter.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

SUMMARY

- **We've done quite a lot with Spring Security at this point – and notice that, except for the most recent lab, we've done no Java coding, just XML configuration and some work in JSPs.**
- **There is more to Spring Security, and with many of the features below we do cross that line and need to start building Java classes that extend or plug into the framework.**
  - Custom authentication providers
  - Adding your own filters to the chain
  - Instance-level authorization
- **Other features can still be done purely with declaration, but require more familiarity with the underlying mechanism of Java classes, and that the developer work at a lower level with plain old `<bean>` definitions of those classes.**
  - Channel security
  - HTTP DIGEST
  - Localization
  - Using JAAS
  - Run-as identities

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Healthcare URL Authorization

LAB 3A

In this lab you will build a fine-grained authorization policy for the Health application, based on request URLs and the requirements stated earlier. You will write out **`<intercept-url>`**s to express specific constraints and allow doctors, nurses, clerical staff, and patients variously to access those URLs.

Lab workspace: **Labs/Lab3A**

Backup of starter code: **Examples/Health/Step3**

Answer folder(s): **Examples/Health/Step4**

Files: **docroot/WEB-INF/applicationContext.xml**
docroot/WEB-INF/web.xml

Instructions:

1. Build and test the starter application, which is just the final answer from a lab in the previous chapter. Log in as **burgess/CAVEman** and see the home page:

   Healthcare 'R' Us

   Medical History

   Make Appointment

   Prescribe Medication

   View/Modify User Profile

   Notice that you can click any of these four links, and follow through the processes – even prescribing medication, though no doctor will be listed in the final prescription because none is logged in! And you'll be able to make appointments – even for other patients.

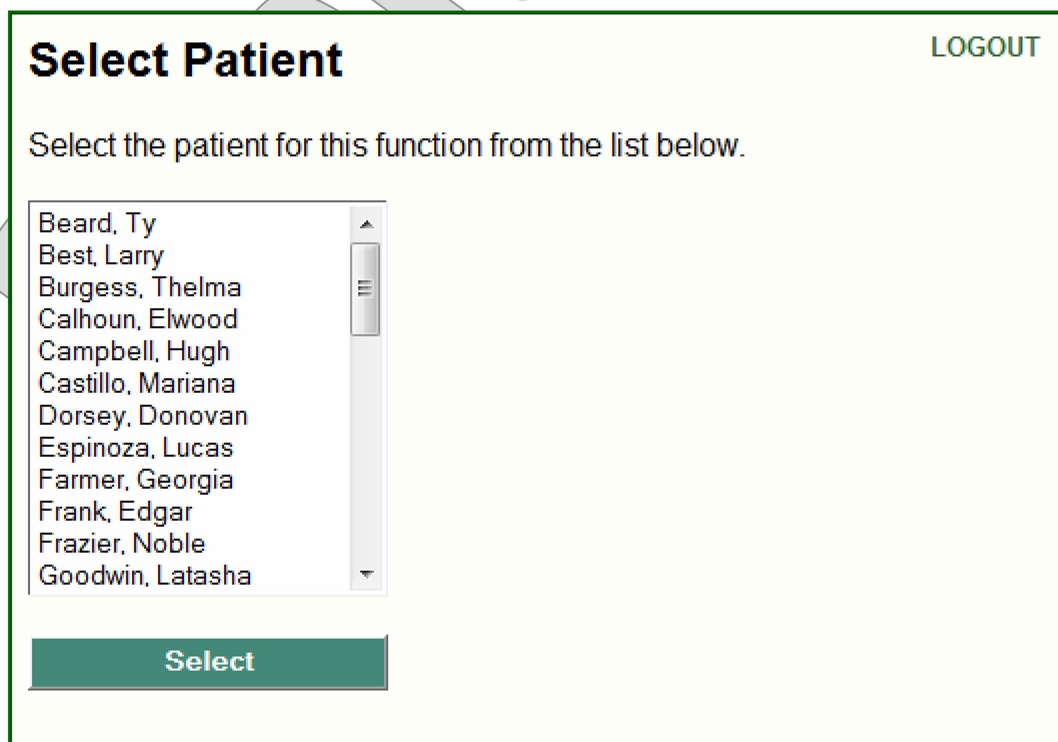
Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Healthcare URL Authorization LAB 3A

2. Look around the **docroot** tree and see that the pages of this application have been organized into directories by function: **history**, **appt**, **prescription**, **profile**, and one special area called **select** through which certain special patient-selection processes will flow.
3. Notice in **web.xml** that all the servlets have been mapped to URLs by the same policy: e.g. the **ChangeProfile** servlet is addressed as **/profile/changeProfile**.
4. Now, open **applicationContext.xml** and add an **access-denied-page** to the **<html>** configuration. The value can be "/accessDenied.jsp", which page is already in place.
5. Change the **pattern** attribute for the general-purpose constraint at the bottom of the list from "/**" to "/*". The former meant that all URLs in the application were subject to this constraint; now we're saying any URL with a single token – that is, in the root directory – is subject. For example, the former applies to **/profile/changeProfile**, while the new pattern does not.
6. Build and test. You should still have no trouble as far as the home page, but pretty much everything outside of that is unprotected. For example, try navigating directly to the following URL:

```
http://localhost:8080/Health/select/selectPatient
```



This serves as a useful reminder that, if you don't set a constraint for a given URL or pattern ... it is not secure!

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Healthcare URL Authorization LAB 3A

7. Now build a policy that matches the requirements stated earlier. More specifically, you want to grant access to the following roles for the following features:

Page	Doctor	Nurse	Clerical	Patient
Home page, CSS, etc.	X	X	X	X
Medical history	X	X		X
User profile	X	X	X	X
Appointments	X	X	X	X
Prescription	X			

   And you already have the first of these, with your "/*" constraint.

8. So, add an **`<intercept-url>`** with the **pattern** "/prescription/**" which grants access only to ROLE_DOCTOR.
9. Do the same for the other features, applying roles as shown above.
10. One internal function not shown in the above table is patient selection: when staff users (all roles but ROLE_PATIENT) need to carry out tasks that relate to patients, they are directed to URLs in the area **/select/****. Define one more constraint, with this as the pattern and all three of those staff roles granted access.
11. Build and deploy, and test out various users to see that you have the right positive and negative results for functions such as prescribing medication and seeing medical history. Work this until you're satisfied with it, and you can compare your implementation to the answer version in **Step4**.

   One caveat: for purposes of a later exercise, we've left the appointment-reminder feature in a somewhat hobbled state. It will work normally if you log in as any staff user: doctor, nurse, or clerical staff. But if you log in as a patient (who doesn't have any of the staff roles), when you try to make an appointment you'll hit a roadblock:

   Access Denied

   You do not have required permissions for access to the requested resource.

   This is some incomplete page-flow logic asking the patient to choose a patient for whom to make an appointment – unnecessary, certainly, and also not secure, since this next step would expose a list of all patients in the system and potentially allow the user to make appointments for other patients. Fortunately your authorization policy stops this before it can go any further! We'll fix this more directly in a later lab.

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Programmatic Security for Healthcare

LAB 3B

In this lab you will add role-based presentation to the home page of the Health application, guiding the user only to the functions to which he's authorized, using the Spring Security tag library.

You'll also review some programmatic security already implemented in the medical history feature, and fix one remaining piece of code so that the page flow for staff vs. non-staff users is different when making appointments.

Lab workspace: **Labs/Lab3B**

Backup of starter code: **Examples/Health/Step4**

Answer folder(s): **Examples/Health/Step5** (intermediate)
Examples/Health/Step6 (final)

Files: **docroot/WEB-INF/applicationContext.xml**
docroot/home.jsp
src/cc/health/web/ShowHistory.java
src/cc/health/web/SetUpAppointment.java

Instructions:

1. Build and test the starter application, which is the answer from the previous lab. Let's turn our attention to role-based presentation. Now that you've put your policy in place, Thelma Burgess can no longer prescribe medication – which is good! But the home page doesn't look any different:

   Healthcare 'R' Us

   Medical History

   Make Appointment

   Prescribe Medication

   View/Modify User Profile

   So why is a patient still invited to try to do something she won't be allowed to do?

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Programmatic Security for Healthcare LAB 3B

2. Open **home.jsp** and declare the use of the Spring Security tag library:

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

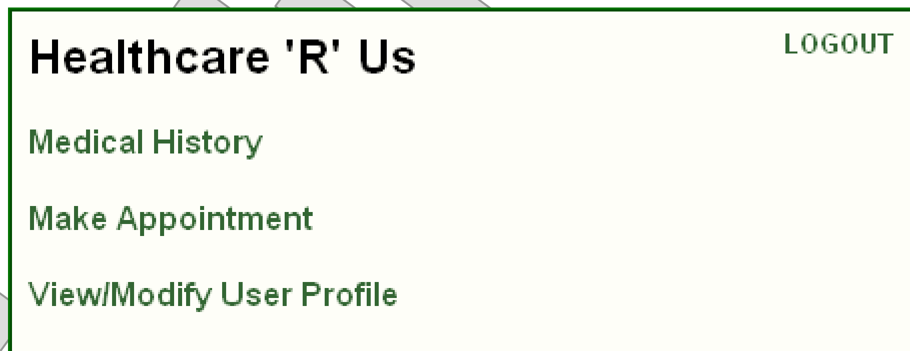
1. Wrap the **Medical History** link in a tag that will show it only to doctors, nurses, and patients:

```
<sec:authorize ifAnyGranted="ROLE_DOCTOR,ROLE_NURSE,ROLE_PATIENT" >
  <p><a href="history/showHistory" >Medical History</a></p>
</sec:authorize>
```

2. Do the same for prescription, now showing only to doctors:

```
<sec:authorize ifAnyGranted="ROLE_DOCTOR" >
  <p><a href="prescription/setUpPrescription" >
    Prescribe Medication</a></p>
</sec:authorize>
```

3. Build and test again, and now the page should be free of links that will just give 403s; for instance here's the page as shown to any patient:



This is the intermediate answer in **Step5**.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Programmatic Security for Healthcare LAB 3B

4. Now, logging in as **burgess/CAVEman**, look at medical history:

Patient Medical History

Patient: Thelma Burgess

Date	Event
09/24/01	Fractured wrist
04/04/02	Appendectomy
01/21/03	Episiotomy
01/21/03	Gave birth: girl, 6 lbs. 8 oz., 18.5"
08/11/04	Tendonitis

HOME

Notice that you see data for this specific patient (although our history data is not, strictly speaking, unique – there's a lot of repetition there). This is because a filter in the application code uses **getUserPrincipal** to derive the name of the user who has logged in, and then hits the database to pull up the patient record. See **IDFilter.java** – the JPA DAO piece won't be entirely clear, but you see how the **HttpRequestServlet** API is used:

```
if (session.getAttribute (Attributes.USER) == null &&
    realRequest.getUserPrincipal () != null)
{
    String username = realRequest.getUserPrincipal ().getName ();

    DAO dao = new DAO ();
    ServletsID user = new ServletsID ();
    user.setPerson (dao.getPersonByUsername (username));
    user.setPatient (dao.getPatientByPerson (user.getPerson ()));
    session.setAttribute (Attributes.USER, user);
    dao.close ();
}
```

The **ServletsID** instance carries its **person** and **patient** properties at session scope and is used throughout the application to derive data about the logged-in user.

Evaluation Only

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Programmatic Security for Healthcare LAB 3B

Most of the application tunes in to this information and manages workflow cleanly. One especially good test is to log in as one of the users who are assigned to multiple roles – see the "cheat sheet" for two of these. Can a patient/nurse do the things that only a nurse is supposed to be able to do? How about a clerical worker who is also a patient – can he see another patient's medical history, or only his own?

5. You'll probably see that these edge cases work out well generally with your new policy, and that there's even some different page flow for different users over certain functions. For example, if a doctor goes to look at medical history, she has to start by choosing a patient; if a patient who is not also a staff member clicks the same link, her name is filled in and she goes right to the appointment form. This is thanks to programmatic authorization found in the existing application code:

The **ShowHistory** servlet makes a choice of page flows, depending on user role – see **ShowHistory.java**:

```
if (request.isUserInRole ("ROLE_DOCTOR") ||
    request.isUserInRole ("ROLE_NURSE"))
{
    request.getSession ().setAttribute
        (Attributes.DESTINATION_AREA, "history");
    request.getSession ().setAttribute
        (Attributes.DESTINATION_PAGE, "history.jsp");
    request.getRequestDispatcher ("../select/preparePatientList")
        .forward (request, response);
}
else
    request.getRequestDispatcher ("history.jsp")
        .forward (request, response);
```

Patient selection is a generic "sub-flow" in this application, and so **selectPatient.jsp** has a dynamic form **action** that allows it to return to the right spot once a patient is chosen, and the **SelectPatient** servlet manages the flow from there.

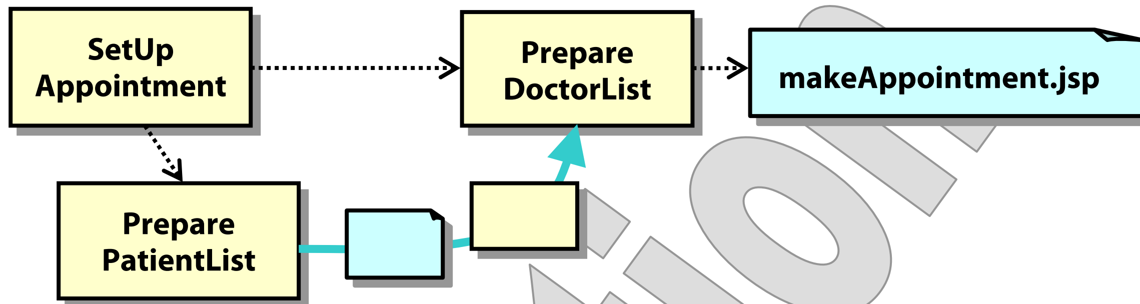
If you test out the appointments function, you'll see that we're not quite there yet: you see a page asking you to select a patient, no matter how you're logged in. This is both annoying for the patient, who naturally would want to make an appointment only for herself, and badly non-secure for everyone, since a patient can refer another patient!

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Evaluation Only

Programmatic Security for Healthcare LAB 3B

6. What we want is a switched flow just like the one for history:



7. Open **SetUpAppointment.java** and see that it has no conditional logic. Wrap the whole implementation of **doGet** in an **if** statement. For the test expression, instead of the servlets-centric code we saw in **ShowHistory.java**, we're going to use the Spring Security API directly to test if the user is in any of the staff roles. So:

```
if (AuthorityUtils.userHasAuthority ("ROLE_DOCTOR") ||
    AuthorityUtils.userHasAuthority ("ROLE_NURSE") ||
    AuthorityUtils.userHasAuthority ("ROLE_CLERICAL"))
{
    request.getSession ().setAttribute
        (Attributes.DESTINATION_AREA, "appt");
    request.getSession ().setAttribute
        (Attributes.DESTINATION_PAGE, "prepareDoctorList");
    request.getRequestDispatcher ("../select/preparePatientList")
        .forward (request, response);
}
```

8. Now add an **else** clause that forwards directly to the servlet that sets up the list of available doctors; this servlet forwards to **makeAppointment.jsp** in all cases. (You can lift most of this code verbatim from **ShowHistory.java** and just change the URL.)

```
else
    request.getRequestDispatcher ("prepareDoctorList")
        .forward (request, response);
```

9. Build and test, and see that now a non-staff user will automatically be selected as the patient for whom we're making an appointment, while any staff user will see the old functionality and be able to choose a patient (including the staff user himself, in the case of multiple roles).

   This is the final answer in **Step6**; it also includes a refactoring of the **ID** implementation to use the Spring Security API, and we'll discuss this in the following chapter.

© 2006-2010 Will Provost.
All rights reserved by Capstone Courseware, LLC.

Evaluation Only