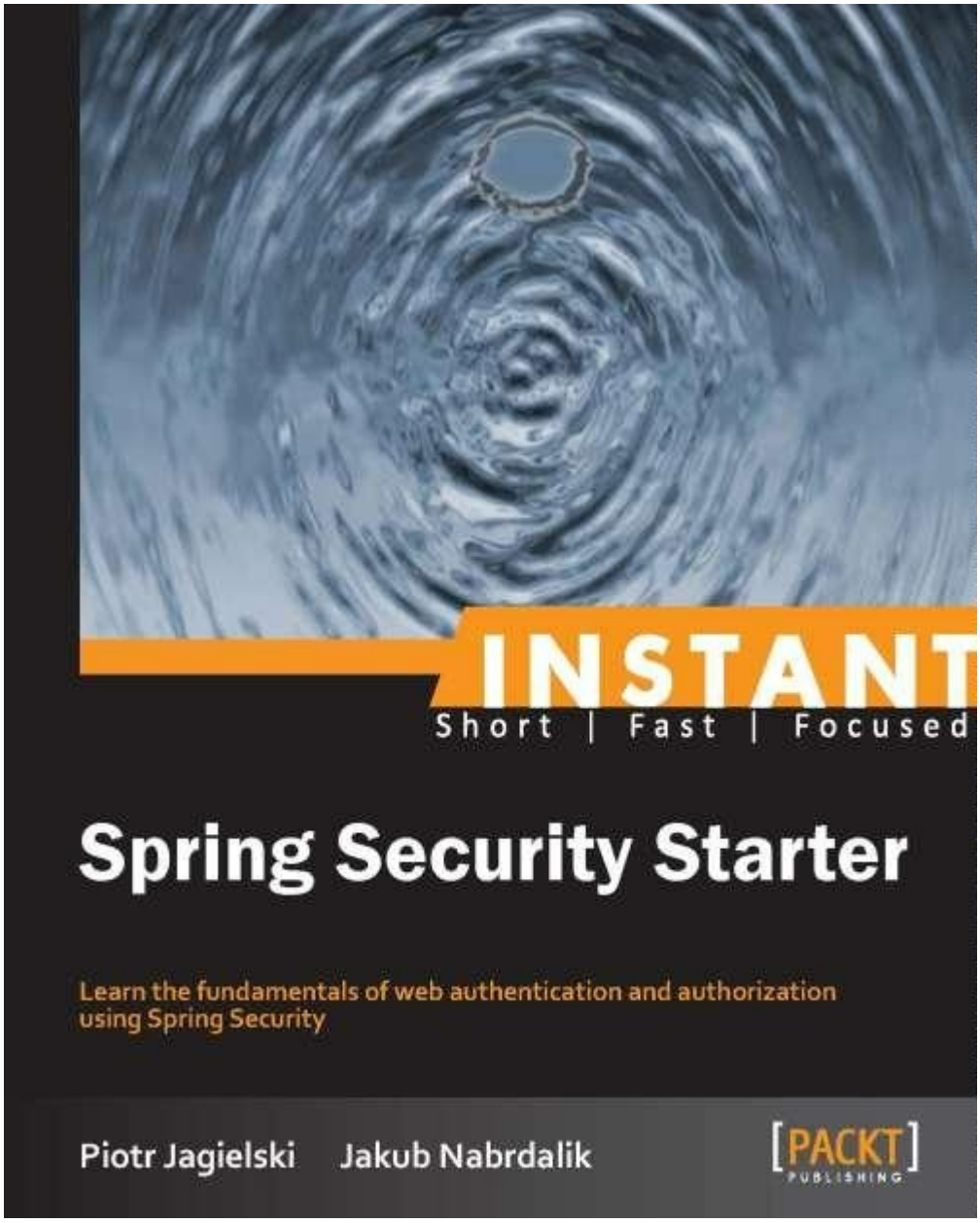

INSTANT
Short | Fast | Focused
Spring Security Starter
Learn the fundamentals of web authentication and authorization using Spring Security
Piotr Jagielski
Jakub Nabrdalik
[PACKT]
PUBLISHING

Table of Contents

Instant Spring Security Starter

Instant Spring Security Starter

First published: June 2013

Production Reference: 1190613

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-883-6

www.packtpub.com

Credits

Authors

Piotr Jagielski

Jakub Nabrdalik

Reviewer

Greg L. Turnquist

Acquisition Editor

Usha Iyer

Commissioning Editor

Sruthi Kutty

Technical Editors

Sumedh Patil

Dominic Pereira

Copy Editors

Insiya Morbiwala

Aditya Nair

Project Coordinator

Amigya Khurana

Proofreader

Maria Gould

Graphics

Abhinash Sahu

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Authors

Piotr Jagielski graduated in Computer Science from Warsaw University, where he encountered functional (SML) and logic (Prolog) programming 10 years before *Seven Languages* by *Bruce A. Tate* was published. He started coding for money with C++, and after two years, switched to Java, which he still uses to this day (with a little help from Groovy) in both integration middleware and frontends as a senior developer at TouK.

In his spare time, he discovers tech startups and trending open source frameworks, which has led him to create a framework rating system at DevRates.com. He spent the last couple of months hacking Raspberry Pi and Arduino, with some quite successful proofs-of-concept (he won second place in Hackwaw for a teddy bear baby monitor).

He is also a husband and the geek dad of a 2-year-old geek boy.

I would like to thank my dad for convincing me to study Computer Science, my wife for her support and patience, and Jakub for being such a great co-author.

Jakub Nabrdalik was born a year before Commodore 64 was introduced, and soon started programming text-based adventure computer games for that machine. He got his B.Sc. from the University of Warmia and Mazury, and his M.Sc. from Military University of Technology, both in Poland. Sucked up by the world's transition to the Web, he started building ERP, e-commerce, and enterprise systems on a thin client with PHP, C#, Java, and Groovy. In 2005, his life was changed dramatically by Test Driven Development. As a strong believer in the Agile Manifesto and Craftsmanship, he started sharing his passion by speaking at conferences (33rd Degree, TopConf, Confitura, and Javarsowia), Warsaw Java User Group meetings, Agile Warsaw meetings, and mentoring/coding at several workshops and Hackathons.

Since 2007, he has been working as a Solution Architect at TouK, a medium-sized Agile company, making dedicated software for telcos, banks, and smaller customers. His main goal is to build highly maintainable services, in a very time-and-money-constrained environment. He blogs a bit at blog.solidcraft.eu.

I'd like to thank the guys and gals at TouK, for being real nerds, Piotr Szarwas, for kickstarting me in the right direction, and Magda, you know what for.

About the Reviewer

Greg L. Turnquist has worked in the software industry since 1997. He is a test-bitten script junkie who is always looking for the right tool for the job. As an open source enthusiast, he has made contributions to several projects, including Tomcat, Homebrew, Spring Security, Spring XD, MythTV, and Mediawiki. In 2006, he created the Spring Python project.

He is a member of the Spring team at Pivotal where he has worked on several things, including JMS on Rabbit, tc Server, and various Spring projects. When he worked at Harris, he built a mission-critical, rules-based system to monitor a nationwide network while managing a team of engineers. He has written *Python Testing Cookbook* and *Spring Python 1.1*, both for Packt Publishing.

Greg has completed his Master's degree in Computer Engineering from Auburn University and lives in the United States with his family.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

packtlib.packtpub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Instant Spring Security Starter

Welcome to *Instant Spring Security Starter*. This book will walk you through the most common use cases that occur when creating web applications with Spring Security. The book includes some tricks and solutions that the authors have learned from their experiences. All this comes from real-life projects, which are well and healthy, and are online either on the Internet or on corporate intranets.

This book contains the following sections:

So, what is Spring Security? explains the basics of Spring Security and gives a quick overview of its history.

Quick start – getting the basics right shows you how to get started with Spring Security. Furthermore, it introduces you to the big picture and covers the steps involved in adding a Spring Security layer.

Top 11 features you need to know about explains all the essential features of Spring Security, such as password encoders, the power of SPEL, exposing secured RESTful services, and much more.

People and places you should get to know includes useful links that could help you fight your developer frustration when debugging Spring Security. Here you will discover various communities and forums dedicated to Spring Security.

So, what is Spring Security?

Spring Security is a Java library for authentication and authorization. You can use it in the JEE environment, servlet containers, and standalone applications. It doesn't make any assumptions about your application, though it has a solution ready for most authentication/authorization problems and a handy module or a plugin to get the boring part out of the way.

The Java platform ecosystem is extremely rich. Whatever you do, there are plenty of libraries to help you with it. Security is such an important and basic feature that no matter which web framework you use, somebody already has that covered and usually covered pretty well. Oracle has JEE security tutorials for both web and enterprise applications. It's a part of every JEE container.

Where does Spring Security fit into that picture?

In 2003, Rod Johnson released **Spring Framework**, a new way to create enterprise applications in Java. It was born from the pain of using JEE specifications. We probably are a bit biased, but nonetheless, in those days you could either use Spring Framework or be doomed with JEE. Because of this, Spring Framework was often called the de facto standard in enterprise Java programming.

In 2003, Ben Alex started **Acegi Security** and released it a year later. Acegi Security was good and soon many Spring projects used it. With the increasing popularity of both Spring and Acegi, it became a very popular mix. Both the projects were formally married in 2008 when Acegi was incorporated as an official Spring subproject and it's name changed to Spring Security.

So yes, we are talking about a technology that has been very popular on the market for the last 10 years. It has proved to work well for both big players and small teams. We were happy to use it for banking systems and telcos, as small side projects. All the hiccups are already fixed. While the hipster deep inside of you may die a little from going so mainstream, this is a very important advantage from the security perspective of your systems.

Unlike some other frameworks, Spring never stopped to evolve. With heavy backing from **Spring Source** (now a part of **VMware**) and a vivid community, it still has a fresh sticker on it; though going mainstream washed the colors out a bit and you may not get that Starbucks look anymore.

Today, the playground looks a bit different, and the situation is not that simple anymore. JEE specifications finally look sane. Starting with JEE6 in 2009, you can be very productive with it alone. Several web frameworks have their own ways of dealing with security, though many integrate with Spring either right out of the box or with a plugin. If you can't decide, here are a few reasons why you may want to use Spring Security:

- If you are using Spring Framework (core) anywhere in the project, Spring Security will be very natural for you to use
- If your framework security is based on a closed piece of software that you have no control, or only partial control of (JEE containers, we are looking at you), and you don't like it, Spring Security is completely open, designed so that you can change anything and everything; and it has the best documentation you can ever find for a software project (which also applies to to the Spring Framework core itself)
- If your framework security is complex and not to be touched by the hands of mere mortals, Spring Security is simple, easy to

understand, and written with a code quality that we wish all our projects had
- If your framework security has not been battle-tested and you are anxious about it, Spring Security has all the corners covered
- Finally, if you have to integrate with a lot of different protocols/solutions (OAuth, OpenID, LDAP, and so on), there are open source projects integrating Spring Security with most providers already

Whenever you are introducing a new library to the project, you should always ask yourself: do I really need it? Can I do without it? Finally, is learning that library worth it or can I write the solution myself in the same time?

Sometimes, those questions are hard to answer. But not in this case. Security is so important that you really do not want to get it wrong. Even though it's not really rocket science, and we have implemented authentication and authorization several times in non-JVM technologies, it took us less time to learn Spring Security and get it (almost) right the first time than it would have taken us to implement the basic authentication mechanism ourselves. The fact that we could use that knowledge over and over again in the last 5 years made it one of the best investments ever.

Quick start – getting the basics right

There are two ways to learn anything: by reading first and working later or by reading and working bit by bit at the same time. We would like to mix these two by giving you the big picture first and then moving to a working example.

Understanding the big picture

So we've got this thing for authentication and authorization. Let's see who is responsible and what for.

There is an `AccessDecisionManager`, which, as the name suggests, is responsible for deciding whether we can access something or not; if not, an `AccessDeniedException` or `InsufficientAuthenticationException` is thrown.

`AuthenticationManager` is another crucial interface. It is responsible for confirming who we are.

Both are just interfaces, so we can swap our own implementations if we like.

In a web application, the job of talking with these two components and the user is handled by a web filter called `DelegatingFilterProxy`, which is decomposed into several small filters. Each one is responsible for a different thing, so we can turn them on, off, or put our own filters in between and mess with them anyway we like. These are quite important, and we will dig into them later. For the big picture, all we need to know is that these filters take care of all the talking, redirect the user to the login page (or an access-denied page), and save the current user details in an `HTTPSession`.

Well, the last part, while true, is a bit misleading. User details are kept in a `SecurityContext` object, which we can get a hold of by calling `SecurityContextHolder.getContext()`, and which in the end is stored in `HTTPSession` by our filters. But we had promised a big picture, not the gory details, so here it is:

Quite simple, right?

If we have an authentication protocol without login and password, it works in a similar way. We just switch one of the filters, or the authentication manager, to a different implementation. If we don't have a web application, we just need to do the talking ourselves.

But this is all for web resources (URLs). What is much more interesting and useful is securing calls to methods. It looks, for example, like this:

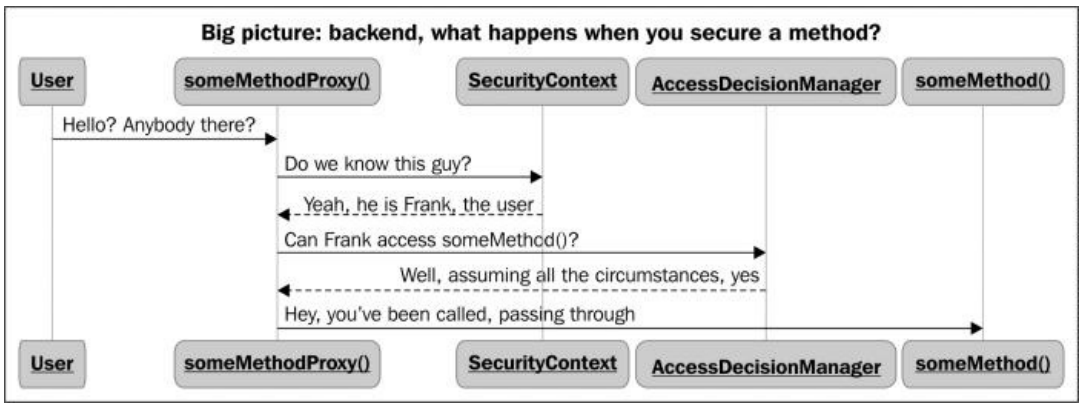
```
@PreAuthorize("isAuthenticated() and
hasRole('ROLE_ADMIN')")
public void somethingOnlyAdminCanDo() {
}
```

Here, we decided that `somethingOnlyAdminCanDo` will be protected by our `AccessDecisionManager` and that the user must be authenticated (not anonymous) and has to have an admin role. Can a user be anonymous and have an admin role at the same time? In theory, yes, but it would not make any sense. Because it's much cheaper to check if he is authenticated and stop right there. We see a bit of optimization here. We could drop the `isAuthenticated()` method and the behavior wouldn't change.

We can put this kind of annotation on any Java method, but our configuration and mechanism to fire up the security will depend on the type of objects we are trying to protect.

For objects declared as **Spring beans** (which is a short name for anything defined in our Inversion of Control (IoC) configuration, either via XML or annotations), we don't need to do much. Spring will just create proxies (dynamic classes) that take over calls to our secured methods and fire up `AccessDecisionManager` before passing the call to the object we really wanted to call. For objects outside of the IoC container (anything created

with `new` or just code not defined in Spring context), we can use the power of **Aspect Oriented Programming** (**AOP**) to get the same effect. If you don't know what AOP is, don't worry. It's just a bit of magic at the `classloader` and `bytecode` level. For now, the only important thing is that it works basically in the same way. This is depicted as follows:



We can do much more than this, as we'll see next, but these are the basics.

So, how does the `AccessDecisionManager` decide whether we can access something or not?

Imagine a council of very old Jedi masters sitting around a fire. They decide whether or not you are permitted to call a secured method or access a web resource. Each of these masters makes a decision or abstains. Each of them can consult additional information (not only who you are and what you want to do, but every aspect of the situation). In Spring Security, those smart people are called `AccessDecisionVoters`, and each of them has one vote.

The council can be organized in many different ways. It has one voice, and so it may make the decision based on a majority of votes. It may be veto-based, where everything is allowed unless someone disagrees. Or it may need everyone to agree to grant access, otherwise access is denied. The council is the `AccessDecisionManager`, and we have three implementations previously mentioned out of the box. We can also decide who's in the council and who is not. This is probably the most important decision we can make, because this will decide the security model that we will use in our application.

Let's talk about the most popular counselors (implementations of `AccessDecisionVoter`).

- **Model based on roles** (`RoleVoter`): This guy makes his decision based on the role of the user and the required role for the resource/method. So if we write `@PreAuthorize("hasRole('ROLE_ADMIN')")`, you better be a damn admin or you'll get a no-no from this guy.

- **Model based on entity access control permissions** (`AclEntryVoter`): This guy doesn't worry about roles. He is much more than that. `Acl` stands for **Access Control List**, which represents a list of permissions. Every user has a list of permissions, possibly for every domain object (usually an object in the database), that you want to secure.

  So, for example, if we have a bank application, the supervisor can give Frank access to a single specific customer (say, **ACME—A Company that Makes Everything**), which is represented as an entity in the database and as an object in our system. No other employee will be able to do anything to that customer unless the supervisor grants that person the same permission as Frank.

  This is probably the most scrutinous voter we would ever use. Our customer can have a very detailed configuration with him/her. On the other hand, this is also the most cumbersome, as we need to create a usable graphical interface to set permissions for every user and every domain object. While we have done this a few times, most of our customers wanted a simpler approach, and even those who started with a graphical user interface to configure everything asked for a simplified version based on business rules, at the end of the project.

  If your customer describes his security needs in terms of rules such as "Frank can edit every customer he has created but he cannot do anything other than view other customers", it means it's time for `PreInvocationAuthorizationAdviceVoter`.

- **Business rules model** (`PreInvocationAuthorizationAdviceVoter`): This is usually used when you want to implement static business rules in the application. This goes like "if I've written a blog post, I can change it later, but others can only comment" and "if a friend asked me to help him write the blog post, I can do that, because I'm his friend". Most of these things are also possible to implement with ACLs, but would be very cumbersome.

  This is our favorite voter. With it, it's very easy to write, test, and change the security restrictions, because instead of writing every possible relation in the database (as with ACL voter) or having only dumb roles, we write our security logic in plain old Java classes. Great stuff and most useful, once you see how it works.

  Did we mention that this is a council? Yes we did. The result of this is that we can mix any voters we want and choose any council organization we like. We can have all three voters previously mentioned and allow access if any of them says "yes". There are

even more voters. And we can write new ones ourselves. Do you feel the power of the Jedi council already?

Adding the Spring Security layer

In this section we'll find out how to add the Spring Security layer to an existing project. Our example is a simple to-do list web application. It's available on GitHub (https://github.com/jakubnabrdalik/spring-security-starter), so you can check it out if you want a working sample.

To add Spring Security, we only need to follow the five steps, detailed in the next section.

Step 1 – adding the correct dependencies to your project

Spring Security is split into several submodules so that we can grab only those things that we need. The base is `spring-security-core`. We can't get anything without it, but everything else is optional. `spring-security-web` provides us with everything related to the Web, `spring-security-config` gives us the XML configuration namespace, and `spring-security-taglibs` is for our JSP tags (don't add it if you are not using JSP in the view layer).

After we are done with adding these dependencies, we must make sure that we have no unexpected dependencies, such as commons-logging or a different version of Spring Framework on the `classpath`. Exclude everything doubled or unnecessary. A common mistake is having two versions of Spring Security in your dependencies. God save us all from dependency hell.

Step 2 – firing up Spring Security using a filter in web.xml

Now we need something to start Spring Security, right? Remember web filters in the big picture? That's exactly what sets Spring Security spinning in a web application. And it's called `DelegatingFilterProxy`.

It's a proxy because it delegates the real work to a chain of filters, each responsible for one thing and one thing only. This allows us to take some functionality out by removing a filter or adding something new by introducing a new filter into the chain.

To make things crystal clear, we are going to name that filter `springSecurityFilterChain`.

```
<filter>

 <filter-name>springSecurityFilterChain</filter-
name>

 <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
```

We also have to set the URL path that is going to be protected. You are usually going to set all requests to go through the filter we mentioned before, so this is what we'll do as well.

```
<filter-mapping>

 <filter-name>springSecurityFilterChain</filter-
name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

If you aren't using the Spring IoC container already, don't forget to also add the Spring IoC listener and point it to a configuration file.

```
<listener>

 <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>

 <param-value>classpath:ioc/applicationContext-
security.xml</param-value>
</context-param>
```

Tip

Downloading the example code

You can download the example code fi les for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the fi les e-mailed directly to you.

We are telling the web application to look for our configuration file on the `classpath` parameter in `ioc/applicationContext-security.xml`. But we don't have anything there yet. This is a perfect time for the next step.

Step 3 – setting up the security context

Now we need to configure Spring Security's internal behavior. This is done by defining beans in a security namespace—a supplementary XML schema designed just for configuring security in a Spring context. So we begin with the correct declaration of the namespace in the `applicationContext-security.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beansxmlns="http://www.springframework.org/schema/security"

 xmlns:beans="http://www.springframework.org/schema/beans"

 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-3.1.xsd

 http://www.springframework.org/schema/securityhttp://www.springframework
security-3.1.xsd">
<!-- Here there be dragons... err... all our
configuration mentioned later -->
</beans:beans>
```

All the configurations we mention from now on should go between `<beans:beans>` and `</beans:beans>`.

Step 4 – getting the basic web security configuration

The first thing usually defined in the security namespace is the web layer security. We should start with basic form authentication (good ol' login/password style).

```
<http auto-config="true">
  <form-login
login-processing-
url="/j_spring_security_check"login-
page="/login"
authentication-failure-url="/login?
login_error=t" requires-channel="https"/>
  <logout
logout-url="/j_spring_security_logout" />
```

```
  <intercept-url
pattern="/secure/**"access="IS_AUTHENTICATED_FULLY" />
  <intercept-url
pattern="/**"access="IS_AUTHENTICATED_ANONYMOUSLY" />
</http>
```

In these few lines, we've told Spring Security that:

- The application allows users to log in via a login form
- The login page (the one presenting login/password form) is hosted at the `/login` URL
- The request from the login form should go to the `/j_spring_security_check` URL (this is the default and is provided by Spring Security), where it will be validated, and you should send it via a secure channel (HTTPS)
- Invoking `/j_spring_security_logout` will log out the user (this is also the default and is provided by Spring Security)
- Requests pointing to URLs starting with `/secure` should be allowed only for authenticated users
- All other requests should be permitted to anyone

By default, `springSecurityFilterChain` expects to get two properties in the request, `j_username` and `j_password`, which are going to be validated.

Our login page can be using any technology we want. The three important things to remember are:

- It should point to `/j_spring_security_check`
- It should `POST` `j_username` and `j_password`
- It should `POST` over `https`

The last part is very important. There is little point in having a secure application when passwords fly over the wire in plain text. Anyone with a sniffer will be able to compromise the system, and since WiFi is so common today, that really means anyone.

We now need to specify how the user will be authenticated. That's the job for `AuthenticationManager`. To configure one in the security context, we will use an `authentication-manager` element and an `authentication-provider` element.

```
<authentication-
manager alias="authenticationManager">
  <authentication-provider
user-service-ref="userAccountDetailsService"/>
</authentication-manager>
```

The default authentication manager iterates through the list of authentication providers and asks each one to authenticate the user. If one

of them succeeds while checking login and password, it will create an `Authentication` object and stop. Otherwise, it moves on to the next authentication provider. By default, we have just one provider – an instance of the `DaoAuthenticationProvider` class. It is responsible for delegating user retrieval to some service as in the Data Access Object pattern. But we can add more authentication providers later. Can you think of a reason we would want to do that?

One reason could be a future upgrade of our password-hashing algorithm from, let's say, SHA256 to SHA512. Since we cannot recalculate old passwords (we keep only the hash of it), we will not be able to tell which user uses which encoding. All we have to do is add the new authentication provider to the list and the password validation will go through both of them.

We can also add other providers to handle LDAP, JAAS, OpenID, or our own custom authentication protocol. Another reason to add multiple authentication providers would be if we had more than one LDAP server. We could add an entry for each server to increase our application's availability.

Let's go back to our single `DaoAuthenticationProvider` class. We can see one bean set, `userAccountDetailsService`. The `userAccountDetailsService` is an implementation of `UserDetailsService`, responsible for finding the user by login (also called username) or throwing `UsernameNotFoundException` if it can't. We have to write this implementation ourselves since Spring has no way of knowing where or how we keep our users. The interface is pretty simple.

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String username)
throwsUsernameNotFoundException;
}
```

Our implementation should be visible in the Spring application context, hence we'll use the `@Service` annotation. If our application doesn't use Spring-component scanning, we need to register the service by declaring it in the configuration XML.

Here is a simple implementation:

```
import
org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
```

```
@Service("userAccountDetailsService")
public class UserAccountDetailsService
implementsUserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String
username)throws UsernameNotFoundException {
        UserAccount account =
getUserAccount(username);
        return new
User(account.getUsername(),account.getPassword(),

 AuthorityUtils.createAuthorityList(account.getAuthority()));
    }
    public UserAccount getUserAccount(String
username) {
      try {
        TypedQuery<UserAccount> query
=UserAccount.findUserAccountsByUsernameEquals(username);
          return query.getSingleResult();
      }catch (EmptyResultDataAccessException ex) {
      throw new UsernameNotFoundException("Could
not find user " + username, ex);
    }
  }
}
```

After calling our user details service, Spring Security will perform several checks—does the password match, is the user disabled or expired, and does the user have the rights to access the requested resource. Our service doesn't need to handle any of this; it just needs to load the user by a given login.

Since we want this guide to be as technology-agnostic as possible, let's just assume that `UserAccount` somehow finds the user account or throws an `EmptyResultDataAccessException` when no user with the given username is found. It can use JPA, JDBI (http://jdbi.org), JdbcTemplate, or whatever, underneath—it's out of the scope of this book. We are using **Spring Roo** and the active record pattern for this example.

The user class is an out-of-the-box implementation of the `UserDetails` interface, included in the Spring Security core module. We only need to call a handy helper (`AuthorityUtils.createAuthorityList`) to convert String-based authorities to `GrantedAuthority` objects, as required by the `UserDetails` interface.

Perhaps you've noticed that the method `getUserAccount` could be private, but since we will have to get the user account in other parts of this book, we have made it public.

We can configure many other things in here, such as password encoding and salt, but to get things going the only thing really needed is a login page.

Step 5 – login page

The login page should consist of a login form and a notification area for exposing invalid authentication messages. As previously described, the inputs must map to the `j_username` and `j_password` parameters and the action URL must match that configured in the form-login element in `applicationContext-security.xml` (the default is `/j_spring_security_check`).

```
  <form method="post"
action="/j_spring_security_check">
    <label for="j_username">login:</label>
    <input type="text" name="j_username"/>
    <label for="j_password">password:</label>
    <input type="password" name="j_password"/>
    <input type="submit" value="Login"/>
  </form>
```

And don't forget to send this form via HTTPS. The easiest way to do this is to let Spring Security force the page to redirect the user from `http` to `https` by adding `requires-channel` to the `intercept-url` element between the `http` tags as follows:

```
<intercept-
url pattern="/login" requires-channel="https"/>
```

Following these five steps should wire Spring Security into our web application. In case of any problem, please refer to our to-do list application on GitHub.

In the next section we'll dig in and find out a bit more about what Spring Security offers us and what to pay attention to when securing commercial applications.

Top 11 features you need to know about

At this point, we have a working application with a login page, but we have no way to get through since we don't have a valid username and password in our database yet. Before we figure out how to save our users (or how to create a registration page), we need to talk about passwords.

Password encoders

Passwords are the scariest way of protecting information. There are several technical and psychological reasons for this:

- Most people don't want to remember anything more complex than, say, `pony_123`.
- For years and years, we were taught that a secure password is a word with letters, numbers, and special characters. Hence, `pony_123` looks good and secure to us. In theory this is true, but that's only part of the truth. When most people use 123 as a suffix, this is true no more. The truth is that the longer and more unpredictable our password is, the harder it is to break, So, `pony eating chair from inside while singing in green` is a much better password than `io18!%_` (and much easier to remember too).
- Most people use just a few passwords, if not one, for all services. Hence, if one service stores them as plain text, the entire security becomes just a myth.

Connect these three and you may start having trouble sleeping tonight. Have you heard the news about a popular service that got hacked lately? This is quite common, unfortunately. If we can afford it, it's good practice to keep logins and passwords on a separate login server, behind additional firewalls, and never give them to anyone. The idea is that the login server will be accessible only from our web servers and will only respond with **yes, the password is correct** or **no, it's incorrect** to authentication requests. The rest of the system would never be able to get those passwords. With such a simple API for a server, it would be much easier to properly configure its security on the network level thus making it much more difficult to crack open. That would be nice. But, it would take a lot of effort. So instead, we'll do something simple in a few minutes.

Whether you require your users to have at least 16-character passwords is up to you. Let us deal with the third problem now.

The first thing we'll do is encode our passwords with a one-way only algorithm such as SHA-256. It's one-way, so in theory, we cannot get the

original password back even if we know the encoded version. But that's in theory. In practice, there is something called **rainbow-tables** that is a precomputed table for reversing cryptographic hash functions. So as long as your original password is in the table, we can get it back. And because of the first problem previously described, `pony_123` will be in the table.

So the next thing we can do is add something to the password before hashing it so that the hash is not found in any rainbow-tables. We can save this hash next to the password so that the user doesn't have to remember it. This is called **salt** in cryptography and is most secure when it's random.

But since we have to store the salt together with the password, it will be visible to the hacker if he/she gets his/her hands on our database. Perhaps with enough computing power or smartness, he/she can use our salt to break our hashing algorithm again.

So the last thing we can do is add another magic word to the salt, not random this time, but hidden well enough that the hacker will have problems obtaining it. While this may not be important, it doesn't cost much and is another lock on our gates.

OK, let's do all of this. Does it sound like a lot of work? We have good news then.

Poof! It's done. Almost.

By default, with the configuration you already have, Spring Security applies 1024 iterations of SHA-256 with an 8-byte random salt. Now, let's add our magic word to it.

```
  <authentication-manager
alias="authenticationManager">

 <authentication-provideruser-service-
ref="userAccountDetailsService">
      <password-encoder ref="passwordEncoder"/>
    </authentication-provider>
  </authentication-manager>

  <beans:bean
name="passwordEncoder"class="org.springframework.security.crypto.password
    <beans:constructor-arg
name="secret"value="myVerySecratWordThatShouldBeSomewhereHidden"/>
  </beans:bean>
```

We have explicitly set `password-encoder` to the `authentication-manager` and declared the encoder further down in our application context. `StandardPasswordEncoder` has several constructors. The one used here

just needs the word. The default algorithm, as already mentioned, is SHA-256. Where you want to store the secret word is up to you.

So now that we know how our password is encoded and who is responsible for it, we can finally add some users.

Registration

Saving a user in the database so that he/she can log in later, is as simple as encoding the password and using our favorite database access mechanism. In our sample application, we are using **Roo** and **JPA** through the active record pattern (all thanks to the AspectJ magic), so our code may be as simple as this:

```
@Autowired
//injecting my password encoder
StandardPasswordEncoder passwordEncoder;
[…]
UserAccount user = new UserAccount();
user.setUsername(username);
user.setPassword(passwordEncoder.encode(password));
user.setAuthority("ROLE_USER");
user.persist();
```

How we gather a login and password (and probably, additional information) from a web page depends on our frontend technology. There are a few things to remember though:

- The target page for the registration form should be secured (HTTPS), otherwise passwords fly over the Internet in plain text and bad things happen
- It is very nice if the registration form is also displayed with HTTPS
- The more information you require, the less people will register

We can use a user's e-mail ID as a login (also called username). Whether or not we should do this depends on if we need to show a unique username to other users. If we do, an e-mail ID should not be used. An e-mail ID is a piece of very private information, and we should keep it secured from others. Otherwise, spams occur.

If we need a user's e-mail ID in our application, make sure to verify it by sending a single e-mail with an activation link (including a random activation token) to the user and not letting him/her log in before the e-mail is confirmed.

Remember-me

Remember-me is a persistent login method, based on cookies in the web browser, such that you can enter your password once and stay logged in even when the session dies or when you turn off your browser.

Remember-me is a bit insecure. First, because the cookie might get stolen. Second, the user might forget to log out (invalidating the cookie) on a public computer. However, unless you are creating a bank system, the convenience of a persistent login is usually a good trade-off for such risk.

Spring provides two different implementations.

The default implementation creates a cookie with an MD5 hash of the username, password, cookie expiration time, and a secret key. This way, we don't have to store anything about the cookie on the server.

To start using it, we just have to declare it in our configuration:

```
<http>
  …
  <remember-me
key="mySecondSecretWordThatShouldBeHidden"user-
service-ref="userAccountDetailsService"/>
</http>
```

We can also set other options, such as how long the cookie should be valid for (token-validity-seconds) or whether it should be sent via https (use-secure-cookie). If the guidelines for securing both the form and its destination with HTTPS have been followed, the defaults should work fine.

Now let's add a `_spring_security_remember_me` checkbox to the login form:

```
<div>
  <label
for="_spring_security_remember_me">Remember
me</label>
  <input type='checkbox'
name='_spring_security_remember_me'/>
</div>
```

Whenever a user checks this checkbox, a filter is created and a `SPRING_SECURITY_REMEMBER_ME` cookie is sent to the browser.

The second implementation creates a completely random cookie and stores it in a database. This is a bit safer but requires that we create a `persistent_login` table. It's still simple (you can read the documentation at http://static.springsource.org/spring-security/site/docs/3.2.x/reference/springsecurity-single.html#remember-me-

persistent-token) since there is little we can add to it, and we would need to have some assumptions about your database layer if we were to dig into it.

Logging out

Since we have a persistent login with a remember-me cookie, we need to be able to clear the cookie. In fact, we should be able to log out even without remember-me because session cookies may not be deleted by the browser when the user thinks it should be (like closing a tab).

As mentioned briefly in the *Step 4 – getting the basic web security configuration* section, the easiest way to clear all cookies and get a user out of a session is to create an HTTP link to `/j_spring_security_logout`. Spring Security handles this URL by default, by processing it in the `LogoutFilter`.

We can change that URL as follows:

```
<http >
  …
  <logout logout-url="/j_spring_security_logout"/>
</http>
```

In the section about backend security, we will figure out how to log the user out in code.

Securing web resources

In the *Step 4 – getting the basic web security configuration* section, we already requested Spring Security to give access to some URLs (starting with /`secure`/) to only authenticated users:

```
<intercept-
url
pattern="/secure/**"access="IS_AUTHENTICATED_FULLY"/>
<intercept-
url
pattern="/**"access="IS_AUTHENTICATED_ANONYMOUSLY"/>
```

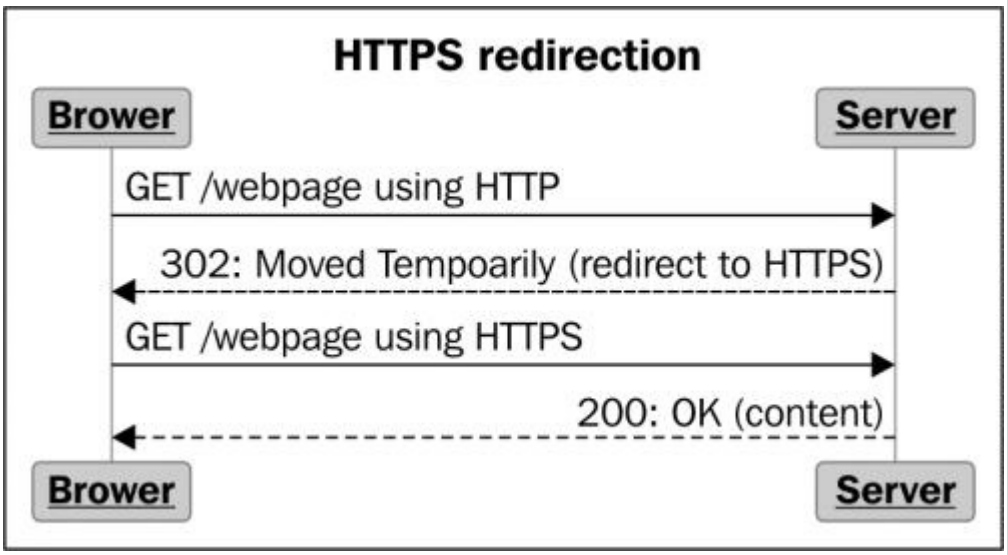
As you can see, it's pretty simple. Let's get through all the options that we have for securing web resources (URLs).

HTTPS versus HTTP

First, we can choose if we want the URL to be accessible only with HTTPS, HTTP, or if it doesn't matter:

```
<intercept-
url
pattern="/httpsOnly/**"requires-channel="https"/>
<intercept-
url
pattern="/httpOnly/**"requires-channel="http"/>
<intercept-
url
pattern="/doesntMatter/**"requires-channel="any"/>
```

The thing you should note is that this will not stop the browser from posting to the wrong channel, but after the first post, the browser will be redirected to a proper channel. Take a look at how it works in the following figure:



Why is this important? If you're sending sensitive information (for example, passwords) from an HTTP to an HTTPS-only page, this data will first go as plain text over the wire, hit the server, the server will then redirect it to HTTPS, and the data will go there over the SSL/TLS channel again. The rule of thumb is to put all forms with sensitive information on an HTTPS-only page to make sure it stays in SSL/TLS. Then, inform the user that it's secure (most browsers show HTTPS as a *safe* connection).

We can also set which ports are going to be used for HTTPS and HTTP with `<port-mappings>` inside `<http>`:

```
<http>
  …
  <port-mappings>
    <port-mapping http="9080" https="9443"/>
  </port-mappings>
</http>
```

Basic access control

We can configure access to web resources using the `access` keyword in `<intercept-url>`:

```
<intercept-
url
pattern="/somePathAndEverythingBelow/**"access="..."/>
```

What we can put into the `access` attribute depends on whether we are using basic access control or expression-based access control. Here is what we can do by default in the former mode:

- `IS_AUTHENTICATED_ANONYMOUSLY`:

  ```
  <intercept-
  url
  pattern="/**"access="IS_AUTHENTICATED_ANONYMOUSLY"/>
  ```

  This gives access to anyone (also users who are not logged in).

- `IS_AUTHENTICATED_REMEMBERED`:

  ```
  <intercept-
  url
  pattern="/secured"access="IS_AUTHENTICATED_REMEMBERED"/>
  ```

  This gives access to authenticated users, including users who were logged in via remember-me cookies.

- `IS_AUTHENTICATED_FULLY`:

  ```
  <intercept-
  url
  pattern="/secured"access="IS_AUTHENTICATED_FULLY"/>
  ```

  This gives access only to fully authenticated users. If the user has logged in via a remember-me cookie, they will be redirected to the login page.

  The thing to remember here is to not secure the login page by accident. If we want the users to log in before seeing any of the content of your application, we need to exclude the login page as follows:

  ```
  <intercept-
  url
  pattern="/login"access="IS_AUTHENTICATED_ANONYMOUSLY"/>
  ```

```
<intercept-
url
pattern="/**"access="IS_AUTHENTICATED_FULLY"/>
```

Otherwise, we will end up in a never-ending loop; accessing any web page will redirect you to the login, which will again redirect you to the login, and so on, till the end of time (though most browsers will give up after a certain number of repeated redirects).

- `ROLE_SOMETHING`:

```
<intercept-
url
pattern="/secured"access="ROLE_SOMETHING"/>
```

This gives access to users who have `ROLE_SOMETHING` in their granted authorities. The name doesn't matter, except it has to start with `ROLE_` to be considered a role. It's possible to change the `ROLE_` prefix to something else, but that's probably not needed in most cases.

How can we set up granted authorities on a user? Roles are basically just strings, so usually you just save them in the database together with the user (or in a `roles` table and join table between `UserAccount` and `roles`) and add them to the `UserDetails` object returned from your `UserAccountDetailsService` class.

There is a helper class, `AuthorityUtils`, that will take these strings and convert them into a collection of `GrantedAuthority` objects with the `AuthorityUtils.createAuthorityList(String... roles)` method.

Take a look again at the method from our `UserAccountDetailsService` class described in the *Step 4 – getting the basic web security configuration* section. In this example, our `UserAccount` can have only one authority (role), but it's quite common to have several:

```
public UserDetails
loadUserByUsername(String username)throws
UsernameNotFoundException {
  UserAccount account =
getUserAccount(username);
  return new
User(account.getUsername(),account.getPassword(),

 AuthorityUtils.createAuthorityList(account.getAuthority()));
```

```
}
```

Roles can have a hierarchical structure, so a user with `ROLE_ADMIN` can, for example, have access to all the other roles as well.

```
<intercept-
url
pattern="/secured"access="ROLE_SOMETHING,
ROLE_ANOTHERTHING"/>
```

This shows that you can mix these keywords previously mentioned. The comma here should be read as *or*.

Please note that all these keywords work thanks to the default council of Jedi masters, that is, `AccessDecisionVoters` from the `org.springframework.security.access.vote` package. If you are building your own council and don't include these voters, the corresponding access control keywords/methods will not work anymore.

Expression-based access control

Basic access control, while very useful, is just the beginning. With **Spring Expression Language** (**SPEL**), we can have more control by using Boolean expressions, regular expressions, accessing object properties, verifying method parameters or returned objects, and most of all, calling methods on the objects registered in our Spring context.

Most of these are usually used in backend security (securing methods), so we will show most of the useful options there. For now, let's see how to configure it for securing web pages. All you have to do is add `use-expressions="true"` to `<http>`:

```
<http auto-config="true" use-expressions="true">
```

And now, you can use expressions in the `access` attribute:

```
<intercept-
url
pattern="/backend/**"access="hasRole('ROLE_ADMIN')"/>
```

All the basic access control keywords have a corresponding built-in expression:

- `isAnonymous()`: `IS_AUTHENTICATED_ANONYMOUSLY`
- `isRememberMe()`: `REMEMBERED`
- `isFullyAuthenticated()`: `IS_AUTHENTICATED_FULLY`
- `isAuthenticated()`: It doesn't matter how, it's just not anonymous

- `hasRole(String role)`, `hasAuthority(String authority)`: `ROLE_`
- `hasAnyRole(String... roles)`, `hasAnyAuthority(String... authorities)`: `ROLE_1`, `ROLE_2`

While there are two equivalents of `ROLE_`, both work in the same way (in fact, one of them calls another). The difference between a role and an authority is purely cosmetic.

Apart from these, we have some special statements:

- `principal`: This allows direct access to the principal object representing the current user
- `authentication`: This allows direct access to the current `Authentication` object obtained from the `SecurityContext` class
- `permitAll`: This evaluates to `true`
- `denyAll`: This evaluates to `false`

One special method, `hasIpAddress`, specific to the Web, allows us to check for an IP or range (using netmask). This makes it trivial to secure some resource to only a localhost or a trusted network:

```
access="hasIpAddress('192.168.0.1/24')"
```

For more about what expressions can do for you, go to the *The power of SPEL* section.

Web filters

So far, we've been using Spring as it is, out of the box. The best thing about this framework though is that it makes it very easy to do all the things that are not present out of the box, such as sending a one-time password via SMS or integrating with a closed, third-party single sign-on system.

To do anything advanced, we need to understand the glue that holds it all together. Take a look again at the *Understanding the big picture* section, and you will instantly see that the parts doing all the talking are Spring web filters.

In the *Step 2 – firing up Spring Security using a filter in web.xml* section, we configured `ContextLoaderListener` in `web.xml`, which starts up Spring and `DelegatingFilterProxy`, which then delegates all the talking to `FilterChainProxy`. By adding `<http>` to our IoC configuration, we get a working `FilterChainProxy`.

As the name suggests, `FilterChainProxy` is just a proxy containing an ordered list of smaller filters, each responsible for a single task only. The

ordering is crucial if we want to get them working, so let us have a look at the most important filters in the correct order:

1. `ChannelProcessingFilter:` This one is responsible for making sure the request is in HTTPS or HTTP, as desired. If not, it does the redirection.

   It may make sense to remove this filter if all your requests are trusted requests over the local network and you don't care about the channel but care a lot about performance.

2. `ConcurrentSessionFilter`: This one saves the time of the last update in the session and checks if the current session has expired. If so, it calls the logout handlers to clean up the session.
3. `SecurityContextPersistenceFilter`: This one takes care of `SecurityContext` in `SecurityContextHolder`. You always need this one, and as a matter of fact, you cannot get rid of it. It has a `forceEagerSessionCreation` property that you may want to set to `false` if all the requests you are handling are REST/SOAP requests from other applications that do not need or want to keep a session. The default is `true`, so you will always get a new session on a request without `jsessionid`. We will cover the REST subject later in the book.
4. `LogoutFilter`: On a request to `/j_spring_security_logout` (which is configurable), this takes care of cleaning everything up after logout and calling the same previously mentioned logout handlers.
5. Any preauthorization filter: This is where any filter based on `AbstractPreAuthenticatedProcessingFilter` will be invoked. By default, nothing is there, but Spring Security comes with several implementations, such as `J2eePreAuthenticatedProcessingFilter` for using container-provided authentication. The main idea behind these filters is that the request has already been authenticated by some other mechanism, and you just need to extract the user credentials.

   It's also a perfect place to implement integration with your own single sign-on system (if you really have to).

6. `UsernamePasswordAuthenticationFilter`: Here the user is authenticated. Since we have configured `<form-login>`, we have a `UsernamePasswordAuthenticationFilter` class, but we can also swap it with anything that takes care of the authentication, such as a `DigestAuthenticationFilter` class or our own third-party single sign-on authentication filter.

If we want to have an application with several authentication methods, we usually add additional authentication filters right after or before this one.

If you have no need for username/password authentication, you can, of course, get rid of it.

7. `OpenIDAuthenticationFilter`: This one takes care of OpenID integration. We'll dig more into this subject later on.
8. `BasicAuthenticationFilter` and `DigestAuthenticationFilter`: If you want to use HTTP Basic authentication headers, this one will parse them. Where you would like to use it is up to you, but we strongly recommend leaving it only for the intranet and server-to-server communication over private network communication, because with HTTP basic authentication, all passwords fly in plain text over the wire, and it's a joke to hack into a public network. On the plus side, it's quite popular.

   A safer approach for server-to-server communication is using Digest authorization (RFC 2617), and as expected, we have a filter for that as well.

9. `SecurityContextHolderAwareRequestFilter`: Thanks to this, you'll have several additional methods in `HttpServletRequest`, such as `getRemoteUser()`, `isGranted(String role)`, `isUserInRole()`, and `getPrincipal()`. Since you don't need the naked `HttpServletRequest` class for web security, and using a request object in the backend would be strange, you can safely get rid of this.
10. `JaasApiIntegrationFilter`: If you have to integrate with **Java Authentication and Authorization Service** (**JAAS**), this will come in handy.
11. `RememberMeAuthenticationFilter`: We have remember-me tokens thanks to this guy.
12. `AnonymousAuthenticationFilter`: If it's not known who you are, you are anonymous. This filter makes sure that we have something better than `NullPointerExceptions`. If you are using `IS_AUTHENTICATED_ANONYMOUSLY` or something similar, you will need it, and it's on by default.
13. `SessionManagementFilter`: This filter, if set up, fires up a corresponding `SessionAuthenticationStrategy` on a successful login. This, in effect, can prevent session fixation attacks by migrating the whole session or stopping the user from using two sessions at once.

    Actually, it can prevent the user from using any number of sessions at once, but the most common approach is to limit the

user to just one. Don't ask us why. We would hate not being able to log in with two browsers at the same time just as much as you do; fortunately it's possible, so someone has to have some reason for that.

14. `ExceptionTranslationFilter`: If any of your filters throw `AccessDeniedException` or `AuthenticationException`, this filter will take action. It takes care of redirection to the login page and sending neat 403 HTTP headers (or 401 for HTTP authentication).

    As expected, you can decide what should happen in case of denial of access by implementing an `AccessDeniedHandler` class and injecting it into this filter.

    You also always need this one and cannot get rid of it.

15. `FilterSecurityInterceptor`: This one secures web resources. You also cannot get rid of it.

Was the list long? Well it gets even longer, but you will probably never have to hear about other filters. The takeaway from this section is that filters do *a lot*, and you can configure what exactly you want to have and when. The only required filters are `SecurityContextPersistenceFilter`, `ExceptionTranslationFilter`, and `FilterSecurityInterceptor`.

The other thing is that you should stick with the order or funny things start happening. You can add your own filters to this chain wherever you like, either before, after, or instead of other filters. For a complete and correct order, please go to the `org.springframework.security.config.http.SecurityFilters` enum.

To add your custom filter, just add it to the Spring context and declare it in `<http>`:

```
<custom-
filter
ref="mySpecialAuthenticationFilter"position="FORM_LOGIN_FILTER"/>
```

Instead of position, you can use the `before` and `after` attributes. Since there can only be one filter at any given position, if you want to add your own filter at `FORM_LOGIN_FILTER`, you need to take the existing one away by removing `<form-login>` and `auto-config="true"`.

Now that we know who does the talking, let's mess with it a little.

One-time password and two-phase authentication

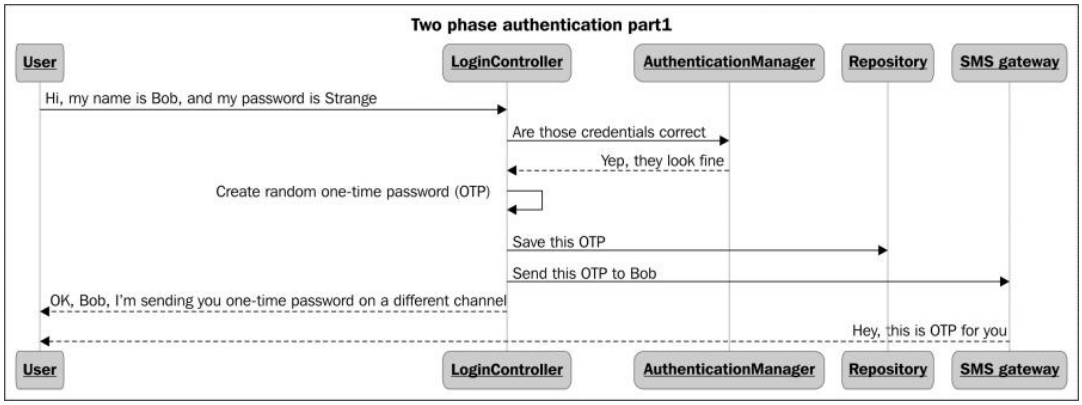
What is better than password protection? Well actually, a lot. From the perspective of an Internet service, single sign-on providers take away most of the pain and trouble accompanying login/password authentication. But that may not always be an option; and even if it is, you usually can't just say *no* to all the users that don't yet have their Facebook, Twitter, or Google account. And logging in to your bank via your Facebook account would look just silly.

A very effective way to improve the security of your form logging is to provide two-factor authentication.

The idea is simple. Knowledge is not enough; you need more. You not only need to know the password, but you also need to possess something, such as an e-mail or a cell phone, to which we can send you a second password. The second password is random and one time only. Only after providing both passwords will you be authenticated successfully. If the second password is sent via SMS, it makes the job of a hacker very difficult.

Because people tend to use the same password over and over, and we cannot do much about it, this is a very popular method for services that could get you into real trouble if hacked, such as Internet banking or e-mail. Google added support for two-factor authentication for all their services not long ago.

Ok, so let's implement this. The first step is to verify the first password, then to generate the second password, and the last step is to send it to the user:



As you can see, this part doesn't need much from Spring Security context. You can do this in your favorite web framework, and the only Spring Security object you need is the `AuthenticationManager` class.

An example implementation using the Spring MVC controller is as follows:

```
@Controller
public class LoginController {
```

```
  private final Log logger =
LogFactory.getLog(getClass());

  @Autowired
  private AuthenticationManager
authenticationManager;

  @Autowired
  private UserAccountDetailsService
userAccountDetailsService;

  @RequestMapping(method =
RequestMethod.POST,value =
"/j_spring_security_check")
  public String
handleLogin(@RequestParam("j_username") String
username, @RequestParam("j_password") String
password, ModelMap modelMap) {
    try {
      authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(username,
password));
    }
    catch (DisabledException | LockedException
|BadCredentialsException e) {
        modelMap.put("login_error", "Bad login or
password");
        return "redirect:/login";
     }

    UserAccount userAccount
=userAccountDetailsService.getUserAccount(username);
    String oneTimePassword =
generateRandomOneTimePassword();

 userAccount.setOneTimePassword(oneTimePassword);
    userAccount.persist();
    sendViaDifferentProtocol(oneTimePassword);
    modelMap.put("username", username);
    return "redirect:/OTPlogin";
  }

  private void sendViaDifferentProtocol(String
oneTimePassword) {
    logger.debug("One time password sent: " +
oneTimePassword);
  }

  private String generateRandomOneTimePassword() {
    return "this depends very much onthe protocol
you use for sending";
  }
```

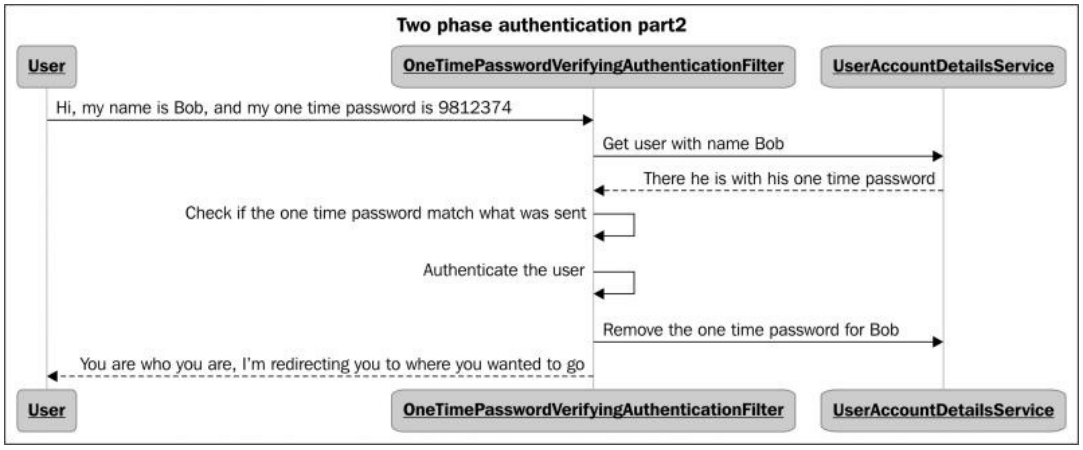
```
  public void
setAuthenticationManager(AuthenticationManager
authenticationManager) {
      this.authenticationManager =
authenticationManager;
  }

  public void
setUserAccountDetailsService(UserAccountDetailsService
userAccountDetailsService) {
    this.userAccountDetailsService =
userAccountDetailsService;
  }
}
```

We have registered `handleLogin` method under the `/j_spring_security_check` URL, the default URL at which the `UsernamePasswordAuthenticationFilter` works, because we are going to replace this filter and we don't have to change the login form this way.

Since our `UserAccount` class implements the `Active Record` pattern, we call `persist` to save it after updating the one-time password. You may want to call a `User` repository or something similar instead. Whether you want to save the one-time password in the database or not is actually not obvious. You can decide to save it together with the username in an HTTP session instead, or in the Flash scope, which is a storage in the session that is only valid for the next request. Which option is better depends a lot on your business cases, your web framework, and even your infrastructure.

Both `generateRandomOneTimePassword` and `sendViaDifferentProtocol` methods are, of course, just stubs. Both depend on the protocol you are going to use. If you are sending the second password via SMS, remember that the password should be easy to read on all devices. Letters like `l` and numbers like `1` look very similar. Also the password cannot be too long because the user will probably not be able to copy and paste it. If on the other hand you are sending the password via email, you can be much more verbose.

It all looks good. Our controller redirects the user, passing the username, to the `/OTPlogin` URL, where we have another form to accept the one-time password. Before we go any further, let's look at the flow of the second part of our authentication:



`OneTimePasswordVerifyingAuthenticationFilter` is the filter that we will create and that will replace the `UsernamePasswordAuthenticationFilter` class.

Let's have a look at the form:

```
<form
action="/j_spring_security_one_time_password_check"
method="POST">
   <input id="username" type="text"
name="username" style="visibility: hidden"
autocomplete="on">
   <label for="oneTimePassword">One time
password</label>
   <input id="oneTimePassword" type='password'
name='oneTimePassword'/>
   <input id="proceed" type="submit"
value="Submit">
</form>
```

The form has two fields: one for the username (hidden) and one for the password. Here, we are filling in the username from our previous request. The form is submitted to `j_spring_security_one_time_password_check` from where the form will be picked up by our filter. Talking about which, let us finally write it.

We are going to extend `AbstractAuthenticationProcessingFilter` that will provide us with some handy features out of the box:

```
public class
OneTimePasswordVerifyingAuthenticationFilter
extends AbstractAuthenticationProcessingFilter {
   public static final String
ONE_TIME_PASSWORD_VERIFICATION_URL =
"/j_spring_security_one_time_password_check";
   private String oneTimePasswordRequestKey =
"oneTimePassword";
   private String usernameRequestKey = "username";
```

```
   @Autowired
   private UserAccountDetailsService
userAccountDetailsService;

   public
OneTimePasswordVerifyingAuthenticationFilter() {
       super(ONE_TIME_PASSWORD_VERIFICATION_URL);
   }

   @Override
   public Authentication
attemptAuthentication(HttpServletRequest request,
HttpServletResponse response) throws
AuthenticationException, IOException,
ServletException {
       String username =
request.getParameter(usernameRequestKey);
       String receivedOneTimePassword =
request.getParameter(oneTimePasswordRequestKey);
       UserAccount userAccount =
userAccountDetailsService.getUserAccount(username);
       if(userAccount == null) {
           throw new
UsernameNotFoundException("User with username " +
username + " not found");
       }

       String expectedOneTimePassword =
userAccount.getOneTimePassword();
       if(expectedOneTimePassword == null ||
expectedOneTimePassword.isEmpty() ||
!
expectedOneTimePassword.equals(receivedOneTimePassword))
{
           throw new BadCredentialsException("One
Time Password did not match");
       }

       PreAuthenticatedAuthenticationToken
authenticationToken = authorizeUser(username);

 userAccountDetailsService.clearOneTimePasswordFor(username);
       return authenticationToken;
   }

  private PreAuthenticatedAuthenticationToken
authorizeUser(String username) {
       UserDetails userDetails =
userAccountDetailsService.loadUserByUsername(username);
       PreAuthenticatedAuthenticationToken
authenticationToken = new
PreAuthenticatedAuthenticationToken(userDetails,
userDetails.getAuthorities());
       authenticationToken.setAuthenticated(true);
```

```
        SecurityContextHolder.getContext().setAuthentication(authenticationToken
              return authenticationToken;
           }

           public void
        setUserAccountDetailsService(UserAccountDetailsService
        userAccountDetailsService) {
              this.userAccountDetailsService =
        userAccountDetailsService;
           }
        }
```

The `attemptAuthentication` method is the key. In case it is successful, it should return an `Authentication` object. Otherwise, it can throw an `AuthenticationException` exception or just return null if we think that some other filter should do the work for us.

First, we take the username and one-time password from the request. We search for the corresponding user account. Then, we get the one-time password and verify it. If all is well, we authorize the user, clear the one-time password, and return an authentication token.

Since we do not have the original password at the moment (and this is a conscious decision: for safety, we do not want to store the original, plain-text password anywhere, not even in the session), we cannot use an `AuthenticationManager` interface to fill the token for us, like the `UsernamePasswordAuthenticationFilter` class does; so we have to fill it ourselves. We use `PreAuthenticatedAuthenticationToken` because it does not require the original password. We could also write our own token. We also mark the token as authenticated so that it is not passed to the `AuthenticationManager` interface during consequent requests anymore. The token will be valid either till the user logs out or the session times out.

Our login filter also sets the `/j_spring_security_one_time_password_check` URL in the constructor as the URL on which it will work, and this is exactly where our one-time password form is leading us.

So now that we have our filter, let's configure it and replace the former one. We need to make the following changes to our security configuration:

1. Remove `form-login` from the configuration, taking away the possibility to get through with only the first password.

   This is done by removing the `<form-login>` tag and setting `auto-config` to `false` (or just by removing it completely as the default value is false).

Setting `auto-config` to `true` is equivalent to configuring the `<form-login />`, `<http-basic />`, and `<logout />` tags. We don't need `http-basic` in here, but logout would be nice; so, let's add that back.

2. Let's add our own filter in place of the previous one. This can be done by adding a `<custom-filter>` tag.
3. Set an entry point for the security in our application.

   This is where the user will be redirected to start the authentication process if our security decides he needs to do so. This is done by adding the `entry-point-ref` attribute to the `<http>` tag and pointing it to `LoginUrlAuthenticationEntryPoint` with our login URL.

4. Add the filter itself and set an authentication manager for it.

   While we are not going to use the authentication manager, the class we inherit from requires it.

5. In the case of a failure, we want to send the user back to the beginning (the login form).

   This can be done by defining `authenticationFailureHandler` in our filter and pointing it to `SimpleUrlAuthenticationFailureHandler` with our login URL.

   We could also change the `authenticationSuccessHandler` interface in our filter if we wanted to, but the default one (`SavedRequestAwareAuthenticationSuccessHandler`) is quite good. It will redirect the user either to a default page or to the last page he/she was trying to visit when he/she got kicked out back to the login form.

That is a lot of configuration, so let's see what our final results should look like:

```
<http use-expressions="true"
entry-point-
ref="loginUrlAuthenticationEntryPoint"
authentication-manager-
ref="authenticationManager"  >
       <logout />
       <intercept-url pattern="/login"
requires-channel="https" access="permitAll"/>
       <intercept-url pattern="/OTPlogin"
requires-channel="https" access="permitAll"/>
       <intercept-url pattern="/**"
access="permitAll" />
```

```
        <custom-filter
ref="oneTimePasswordVerifyingAuthenticationFilter"
position="FORM_LOGIN_FILTER" />
    </http>

    <beans:bean
id="loginUrlAuthenticationEntryPoint"
class="org.springframework.security.web.authentication.LoginUrlAuthentica
        <beans:property name="loginFormUrl"
value="/login" />
    </beans:bean>

    <beans:bean
id="oneTimePasswordVerifyingAuthenticationFilter"
class="spring.security.starter.web.filters.OneTimePasswordVerifyingAuthen
        <beans:property
name="authenticationManager"
ref="authenticationManager"/>
        <beans:property
name="authenticationFailureHandler">
            <beans:bean
class="org.springframework.security.web.authentication.SimpleUrlAuthentic
                <beans:constructor-arg
value="/login"/>
            </beans:bean>
        </beans:property>
    </beans:bean>
```

Here in the preceding code, we aren't showing the `passwordEncoder` and `authentication-manager` objects that we set up earlier because that part of our configuration hasn't changed.

And we are done. Our two-factor authentication is ready.

Logged-in user in the backend

Security in the frontend is all nice and cool, but let's move to the backend. Suppose that the user has managed to log in to your application and Spring has performed all the necessary magic necessary. Now how can we get the logged-in user's details from the code? The bridge between our application and the Spring Security context is the `SecurityContextHolder` class. By calling the `getContext()` method, we can obtain the `SecurityContext` interface associated with the current thread. And, the `SecurityContext` interface gives us access to the `Authentication` object; just call `getAuthentication()`:

```
Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
```

The `Authentication` class exposes all that we need to deal with the user:

- `login`: The username
- `credentials`: Usually a password
- `authorities`: Roles granted to the users
- `principal`: Either the username or the `UserDetails` implementation (depends on `AuthenticationProvider`)
- `details`: Additional details about the authentication request (also depends on `AuthenticationProvider`)

Note that the only properties that we can make any assumptions about are the `login` and `authorities` properties. The `Authority` interface is designed to handle many different use cases; so the `getPrincipal()`, `getDetails()`, and `getCredentials()` methods return an `Object` type, and we cannot assume anything about what we'll get from them. Of course, we'll get what we've put in there, but this typically ends with a sequence of `instanceof` statements to make sure that the returned `Authentication` object is the one we expected.

Consider the case when no user is logged in. To make the security chain work fine, without nasty not-null checks on each layer, the `SecurityContext` will hold the `AnonymousAuthenticationToken` interface in that case. The `principal` property of `AnonymousAuthenticationToken` contains the `anonymousUser` string; `authorities` consists of a single entry named `ROLE_ANONYMOUS`, and `getCredentials()` returns an empty string. So we have to pay attention to `AnonymousAuthenticationToken` when writing our own authentication validation logic.

But what about `Authentication.getDetails()`? This is usually created by a call from security filters to `AuthenticationDetailsSource.buildDetails()`. One of the common implementations is the `WebAuthenticationDetails` class that holds additional data provided by the `HttpServletRequest` interface: the IP address of the authenticated user's computer (`getRemoteAddr()`) and the ID of the session that contains the authentication data (`getSessionId()`).

Ok, so we have gotten familiar with `Authentication` and `AuthenticationDetails`. Now let's access a logged-in user's object from the application domain and not from some useless `Authentication` object.

Recall `UserAccountDetailsService` from the *Step 3 – setting up the security context* section? We exposed a method that retrieves a `UserAccount` object by the given username. As we have access to `UserDetails` via `Authentication`, we can easily get the username and pass it to the `getUserAccount` method. This leads us to the `LoggedUserGetter` service implementation:

```
@Service("loggedUserGetter")
public class LoggedUserGetter {
```

```
    @Autowired
    private UserAccountDetailsService
userAccountDetailsService;

    @Autowired
    private BackendAuthenticator
backendAuthenticator;

    public UserAccount getLoggedUser() {
        return
userAccountDetailsService.getUserAccount(getLoggedUserName());
    }

    public UserDetails getLoggedUserDetails() {
        UserDetails loggedUserDetails = null;
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        if
(backendAuthenticator.isAuthenticated(authentication))
{
            Object principal =
authentication.getPrincipal();
            if (principal instanceof UserDetails) {
                loggedUserDetails = ((UserDetails)
principal);
            } else {
                throw new
RuntimeException("Expected class of
authentication principal is UserDetails. Given: "
+ principal.getClass());
            }
        }
        return loggedUserDetails;
    }

    public String getLoggedUserName() {
        return
getLoggedUserDetails().getUsername();
    }

    public void
setUserAccountDetailsService(UserAccountDetailsService
userAccountDetailsService) {
        this.userAccountDetailsService =
userAccountDetailsService;
    }
}
```

In the preceding code, we implemented the following three methods:

- `getLoggedUserDetails()`: This method returns the logged-in `UserDetails` object from `Authentication.principal`. This method doesn't touch the database and thus is very fast/cheap.

We should always use it when the `UserDetails` object contains all that we need.

- `getLoggedUserName()`: This method returns the `login` property of the logged-in user. We create a separate method for it because it's usually called very often.
- `getLoggedUser()`: This method finds the `UserAccount` object in the database. This is heavyweight, so if all you need is to show the username, consider any one of the preceding two methods.

Apart from accessing the current authentication, `SecurityContext` allows you to set the `Authentication` object. This is invoked internally by Spring Security filters when the servlet request is processed by the filter chain.

But why would we do that in our code? This can be useful when we want to log the user in right after he/she has registered an account in our application. It is nice if the login form is not shown again in that case.

We just need to think of a valid authentication implementation to pass a given username and password. This leads us to the simple `BackendAuthenticator` implementation:

```
@Service("backendAuthenticator")
public class BackendAuthenticator {
   @Autowired
   private AuthenticationManager
authenticationManager;

   public boolean login(String login, String
password) {
       Authentication authentication =
authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(login,
password));
       boolean isAuthenticated =
isAuthenticated(authentication);
       if (isAuthenticated) {

 SecurityContextHolder.getContext().setAuthentication(authentication);
       }
       return isAuthenticated;
   }

   public boolean isAuthenticated(Authentication
authentication) {
       return authentication != null
&& !
(authentication instanceof
AnonymousAuthenticationToken) &&
authentication.isAuthenticated();
   }
```

```
   public void
setAuthenticationManager(AuthenticationManager
authenticationManager) {
       this.authenticationManager =
authenticationManager;
   }
}
```

The `login` method takes `login` and `password` as parameters. It builds `UsernamePasswordAuthenticationToken` and delegates the authentication logic to `AuthenticationManager`. We'll perform some checks to see whether or not the returned `Authentication` object is valid: that it is not null, not `AnonymousAuthenticationToken`, and that `isAuthenticated()` returns `true`. Of course, you can provide your own logic for checking if the returned `Authentication` object is valid.

Once we've implemented the `login` method, we should think of a method to log the user out. In the simplest case, we just have to set the authentication to `null`:

```
public void logout() {

 SecurityContextHolder.getContext().setAuthentication(null);
}
```

Note that this should be enough in most cases, but when using a `RememberMeAuthenticationProvider` class, you should also clear the browser's cookies so the log out action can be performed; for example, by calling a proper URL from the user's browser.

Securing methods

So far we've used URLs matching the `<intercept-url>` elements in the Spring Security configuration file. It works at a controller (URL) level—each URL requested by the browser is checked by our filter chain. This is fine if you want to secure the web layer.

But often, we need something more powerful: a fine-grained declaration of secured methods in our services. This typically means that Spring Security will reject a method call when some security conditions are not met. These conditions are specified by marking methods with annotations.

There are two different types of annotation-based security:

- Spring 2.x and JSR-250 annotations (`@Secured`) that allow us to specify the roles needed to execute methods

- Spring 3.x annotations (`@Pre/PostAuthorize`) that support complex expressions (**Spring Expression Language—SPEL**)

To show the differences between these two types, let's introduce a simple `BackendService` implementation:

```
@Service("backendService")
public class BackendService {
   protected static String VERY_IMPORTANT_DATA =
"Very important data";
}
```

As a first try, let's use the `@Secured` annotation. In the simplest case, you just need to specify the role that the logged-in user must be granted to execute the method:

```
@Secured("ROLE_ADMIN")
public String securedForAdmin() {
    return VERY_IMPORTANT_DATA;
}
```

And in the `applicationContext-security.xml` file, we can turn on method-level security by adding the following line:

```
<global-method-security
secured-annotations="enabled" />
```

Accessing the service (for example, from the MVC controller) by the user without the `ROLE_ADMIN` authority will end up with an `AccessDeniedException` exception:

```
org.springframework.security.access.AccessDeniedException:
Access is denied.
```

The classes involved in the method security validation are:

- `MethodSecurityInterceptor`: This is the Spring AOP interceptor that validates method invocation requests.
- `AbstractSecurityInterceptor`: This is the base class for validating security requests against `Authentication` stored in the current security context. It is used by the methods-of-service invocations and for the web processing of secured URLs.
- `AffirmativeBased`: This is the implementation of `AccessDecisionManager` that delegates access decisions to the collection of `AccessDecisionVoters`. In the case of `@Secured`, it is `RoleVoters`.

The `@Secured` annotation allows you to specify a list of "authorities" as well. In that case, a user that is granted any of the mentioned roles is permitted to execute the method:

```
@Secured({"ROLE_ADMIN", "ROLE_USER"})
 public String securedForAdminOrUser() {…}
```

Spring Security 3.x annotations offer many more features. There are four different types of annotations: `@PreAuthorize`, `@PreFilter`, `@PostAuthorize`, and `@PostFilter`. To turn on the 3.x annotations, we have to add the `pre-post-annotations` attribute to the `global-method-security` element:

```
<global-method-
security pre-post-annotations="enabled"/>
```

The simplest `@Secured` equivalent for 3.x annotations is `@PreAuthorize`:

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
public String preAuthorizeForAdmin() {
    return VERY_IMPORTANT_DATA;
}
```

`@PreAuthorize` is the most common of the 3.x annotations. It decides whether or not the user should be allowed to call a method. Like `@Secured`, it is evaluated before the actual invocation—if access is denied, the `AccessDenied` exception is thrown and a real call to the method never happens.

`@PostAuthorize` is evaluated after the actual invocation of the method. This could be helpful for using the return results from our secured method in the security expression (we will discuss the expressions in the next section).

When a secured method returns a collection, the `@PostFilter` annotation can be used. It will filter the collection and keep only the elements that the user has access to (which pass a given expression). `@PreFilter` works the same on collections that are arguments of the secured method.

The stack trace of the `AccessDeniedException` exception thrown by `@PreAuthorize` is the same as when using `@Secured`. The only difference is that the voter is not `RoleVoter` but is `WebExpressionVoter` instead.

Let's see what happens when `@PostAuthorize` fails:

```
org.springframework.security.access.AccessDeniedException:
Access is denied at
org.springframework.security.access.expression.method.ExpressionBasedPost
```

Here is a list of classes involved in the `@PostAuthorize` security chain:

- `AfterInvocationProviderManager` is similar to `ProviderManager` that we've met earlier but delegates the decision to a list of `AfterInvocationProviders`, passing additional parameters with the object returned by an invocation of the secured method
- `PostInvocationAdviceProvider` implements `AfterInvocationProvider` but only delegates to `PostInvocationAuthorizationAdvice`
- `ExpressionBasedPostInvocationAdvice` implements `PostInvocationAuthorizationAdvice` and evaluatesboth the `@PostAuthorize` and `@PostFilter` expressions.

Spring 3.x annotations are based on Spring expressions, so instead of providing just a list of required role(s) in a string, we have to provide full expressions (SPEL) with authorization rules. Let's take a closer look at the Spring Expression Language.

The power of SPEL

SPEL stands for Spring Expression Language. This is a very simple language, interpreted at runtime and often used in Spring annotations. It allows us to call methods or write complex logical statements.

We can use logical operators to connect individual expressions. For example, to allow the execution of a method only for users having both `role1` and `role2`, we can write:

```
  @PreAuthorize("hasRole(ROLE_role1) and
hasRole(ROLE_role2)")
   public String secured() {...}
```

Another cool feature of SPEL is that we can write expressions referring to the parameters of an annotated method. Consider the following:

```
   @PreAuthorize("#todo.assignee ==
authentication.name")
   public void markAsCompleted(Todo todo) {...}
```

This means that calling `markAsCompleted` on `Todo` is only allowed by its assignee. In the list of built-in Spring 3.x EL annotation expressions, you can see variables such as `authentication` and `principal`. So, we are able to compare the value of the objects passed as a method argument property

(in this case, it is `Todo`) with the login of the currently authenticated user (`authentication.name`).

When using `@PostAuthorize/Filter` annotations, additional variables are visible in the expression context:

- `returnedObject` (`@PostAuthorize`): This provides a reference to an object just returned by a secured method
- `filterObject` (`@PostFilter`): This refers to the element of the collection just returned by a secured method

So to keep only `Todo` objects assigned to the current logged user, we can write something like the following:

```
   @PostFilter("filterObject.assignee ==
authentication.name")
   public List<Todo> listTodoAssignedToUser() {…}
```

The method itself could return all the `Todo` objects retrieved from the database, but the `@PostFilter` annotation will take out those that do not match the expression. Of course, this should be used with small amounts of data—it would be more reasonable to filter lots of records directly in the database query.

Another powerful SPEL feature is that of referring to beans defined in the application context. You can write:

```
@PreAuthorize("@permissionChecker.isPermitedToCloseTodo(#todo,
authentication.name)
public void close(Todo todo) {...}
```

The object registered in the Spring context under the `permissionChecker` ID will be called to decide if the user can access a given `Todo` object. Which objects and methods you call is completely up to you. Spring Security's only expectation is that the outcome of the expression be converted to Boolean.

Note that the SPEL evaluator is quite verbose; this makes tracking down expression errors easy. For example, when you make a typo in the bean method name, you'll get something like the following:

```
Caused by:
org.springframework.expression.spel.SpelEvaluationException:
EL1004E:
(pos 19): Method call: Method
isPermitedToCloseTodoasd(spring.security.starter.model.Todo,java.lang.Str
cannot be found on
spring.security.starter.service.PermissionChecker
type
```

This is quite important, as you are programming in strings and static code checking is not an option. Write integration tests instead.

Writing tests

Now let's try to validate that our authorization rules defined in the annotations are correct by writing some tests. While we may view this at the unit level, since we are focused on a single unit of code, they are really integration tests because we need to wire all the beans defined in the `applicationContext-security.xml` file. But the important point to note is that it's automated testing that will verify our security solution:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations =
{"classpath:META-
INF/spring/applicationContext.xml",
"classpath:META-INF/spring/applicationContext-
security.xml"})
public class BackendServiceTest extends
AbstractJUnit4SpringContextTests {
   @Autowired
   BackendService backendService;
}
```

In most cases, we'll have the `context:component-scan` element for autowiring in the `applicationContext.xml` file, so we need to read it in our test too.

Ok, so let's try to invoke some secured method in our test as follows:

```
   @Test
   public void shouldAllowCallingSecureForAdmin()
{
        //when
        String message =
backendService.securedForAdmin();
        //then
        assertEquals(VERY_IMPORTANT_DATA, message);
   }
```

But this will end up with an ugly exception as follows:

```
org.springframework.security.authentication.AuthenticationCredentialsNotF
An Authentication object was not found in the
SecurityContext
```

Spring is complaining about `Authentication` missing in the `SecurityContext` interface. That's fine, because we didn't perform any

authentication; so this scenario reflects an anonymous user calling a secured method. Let's try to fix the test by performing authentication. The thing we are bypassing in the test setup (in comparison to the full-blown configuration from *Step 3 – setting up the security context*) is `DelegatingFilterProxy`. This takes care of setting `Authentication` even when the user in not logged in (`AnonymousAuthenticationToken`). We need to somehow insert a valid `Authentication` object into `SecurityContext` to get rid of this `AuthenticationCredentialsNotFoundException` exception.

It looks like we can use `SecurityContext.setAuthentication()`, as mentioned at the beginning of this section. But what kind of authentication should we use in the context of a unit test? Spring Security comes with a class made especially for this case—`TestingAuthenticationToken`:

```
public TestingAuthenticationToken(Object
principal, Object credentials,
List<GrantedAuthority> authorities)
```

So we just need to pass a principal object (for example, `User` provided by `spring-security-core`), its credentials (let's leave those out for now), and a list of authorities that have been granted. Note that authorities passed to the `User` object are not needed at all in this case:

```
   private static final String USERNAME = "user";

   private void
loginDefaultUserWithRoles(String... roles) {
       loginUserWithRoles(USERNAME, roles);
   }

   private void loginUserWithRoles(String
username, String... roles) {

 SecurityContextHolder.getContext().setAuthentication(
              new TestingAuthenticationToken(
                  new User(username, "pass",
Collections.<GrantedAuthority>emptyList()),
                  "DUMMY_CREDENTIALS",

 AuthorityUtils.createAuthorityList(roles)));
   }
```

We also need to somehow allow our `TestingAuthenticationToken` to be authenticated automatically. We cannot use `DaoAuthenticationProvider` because the `TestingAuthenticationToken` class has nothing to do with the database (and we are in the unit test; no DB is involved here). The corresponding authentication provider is `TestingAuthenticationProvider`. So, we need to inject one in the `setup` method for our test case:

```
    @Autowired
    ProviderManager authenticationManager;

    @Before
    public void setup() {

 authenticationManager.getProviders().add(new
TestingAuthenticationProvider());
    }
```

And here is the final working test case:

```
    @Test
    public void shouldAllowCallingSecureForAdmin()
{
        //given
        initializeAuthentication("ROLE_ADMIN");
        //when
        String message =
backendService.securedForAdmin();
        //then
        assertEquals(VERY_IMPORTANT_DATA, message);
    }
```

What about negative scenarios? Anonymous authentication from the beginning of this section could be one. We can also authenticate with insufficient roles (for example, `ROLE_USER`) and expect `AccessDeniedException` to be thrown:

```
    @Test(expected = AccessDeniedException.class)
    public void shouldDisallowSecuredForUser() {
        //given
        initializeAuthentication("ROLE_USER");
        //when
        backendService.securedForAdmin();
    }
```

Exposing secured RESTful services

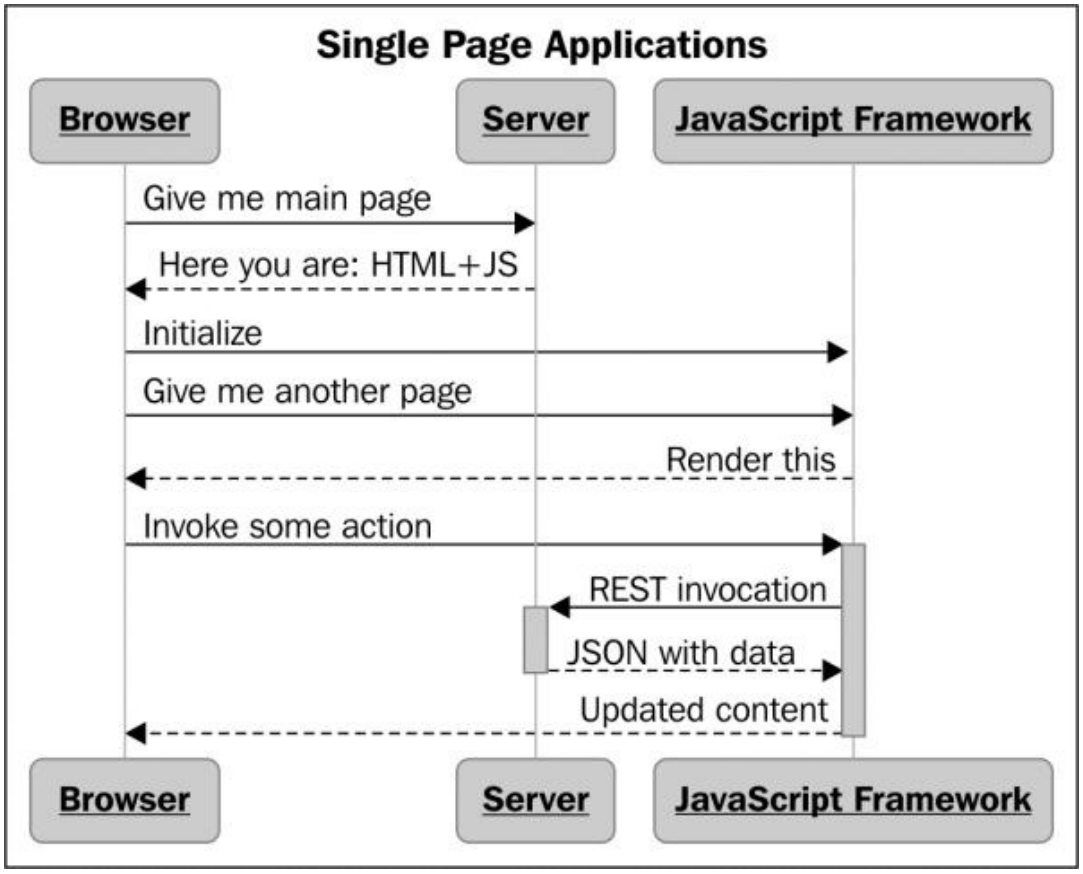
The term "RESTful" is gaining more and more momentum nowadays. If you google "SOAP vs REST", you will find tons of articles written by enthusiasts of both solutions. But is REST a cure for all of the problems of application integration? Saying that you should use REST instead of SOAP is more or less the same as saying that you should use a hammer instead of a screwdriver. When you integrate your application with some external systems maintained by another software vendor, it will be great to have a precise contract defined by SOAP's WSDL that can easily be used to generate some mock services. REST is great when you have full control over both services and their client or when you just want to get it done as

fast and simple as possible. In REST scenarios, changes are visible to the client much faster; this is great when you are developing a service layer for a single-page application, for example.

Single-page applications

Even if you haven't written one yourself, you've probably visited websites designed according to this pattern in your everyday coffee-break surfing.

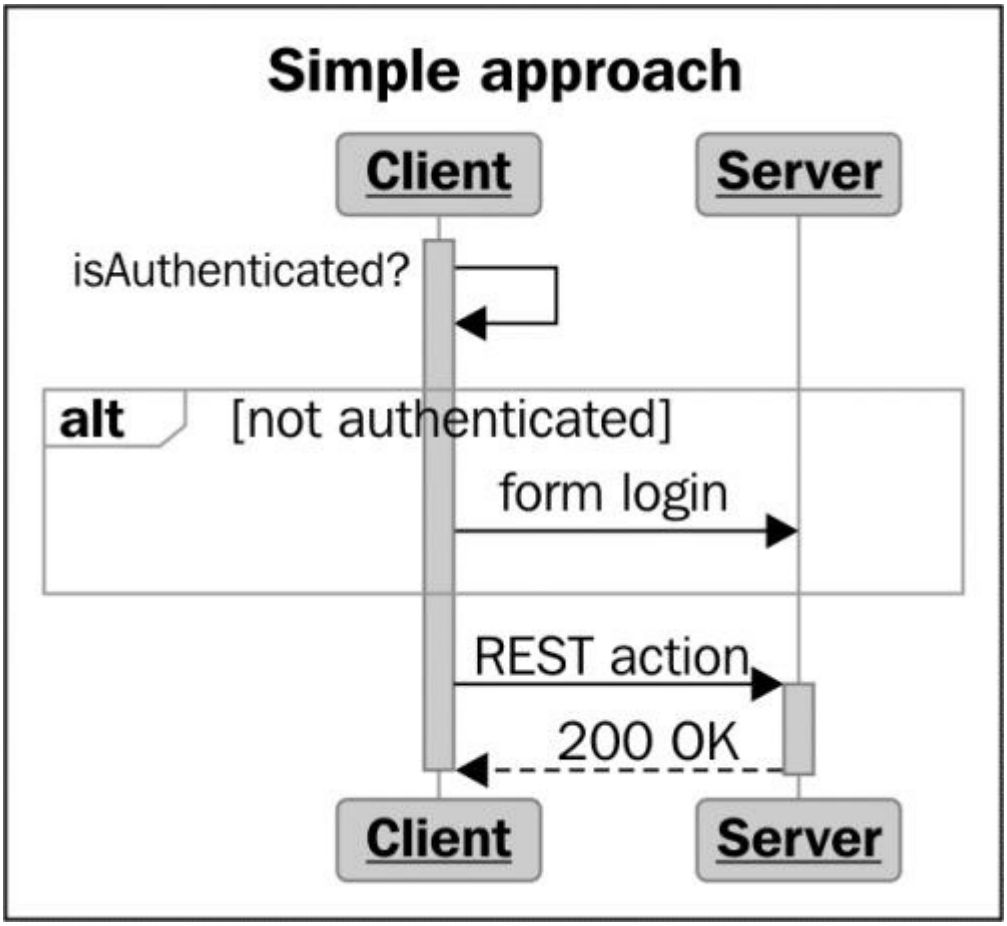
The concept is really simple: the browser opens a single-page application, it receives mostly what is needed to display the content in a single hit, and all the other communication is done via AJAX, with some REST backend returning JSON or XML with data necessary to handle an action. It often works via the **XMLHttpRequest** (**XHR**) API. After the initial hit, no HTML is rendered on the server side at all. In most cases, some clever JavaScript framework is initialized by the initial hit that renders the resulting views, typically by doing some templating with the updated data in the JSON format. This entire process is shown in the following diagram:



Let's try to implement an authentication scenario with a RESTful backend.

Straight approach

Knowing what we've learned up to this point, we can try to simulate the form login flow in the XHR invocations:



When the client wants to interact with a secured REST resource, it first checks whether or not the user is already logged in. If not, it must invoke the form login authentication by providing user credentials to start a Spring Security session. And then the real invocation occurs, which should be successful, as the session was properly initialized by the login call.

But one of the key paradigms of REST is that the server should not maintain a state. As *Roy Thomas Fielding* writes in *Architectural Styles and the Design of Network-based Software Architectures* (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm):

> *"Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server."*

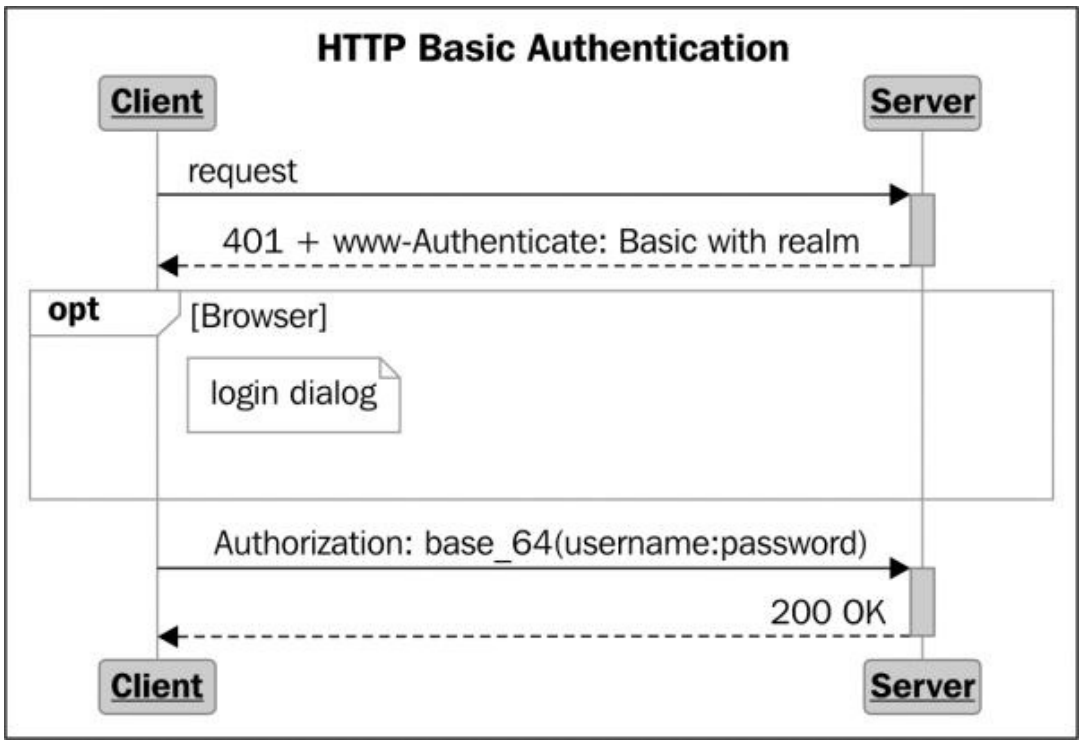
Our approach is not entirely consistent with REST—the server maintains a session that was started by the initial form login from the client and that

holds `SecurityContext` with the `Authentication` object used in subsequent calls from the client.

The HTTP protocol comes with two approaches that are fully compatible with the REST architectural style: **Basic** and **Digest Authentication**.

Basic Authentication

The Basic HTTP Authentication is the simplest possible method of passing a user's credentials to the server with each request:



When a client requests a resource protected with HTTP Basic, the server responds with a 401 Unauthorized HTTP code with a `WWW-Authenticate` header type Basic and a realm name of the protected resource. Most web browsers then show a default login dialog for the user to provide the needed credentials. When the user fills the form in, the browser resends the previous requests with a special `Authorization` HTTP header added to the request. The content of this header is just a base64-encoded value of the `username:password` string preceded by `Basic` (for example, `Authorization: Basic cGlvdHI6cGlvdHIxMjM=`). The server then decodes the credentials (which is easy since base64 is a two-way algorithm) and responds with either **200 OK** or **403 Access Denied**.

The Spring Security namespace contains a special element named `<http-basic/>` that is used to enable HTTP Basic Authentication:

```
<http use-expressions="true"
create-session="stateless">
```

```
<intercept-
url pattern="/backend/**"
access="isAuthenticated()" />
      <http-basic/>
 </http>
```

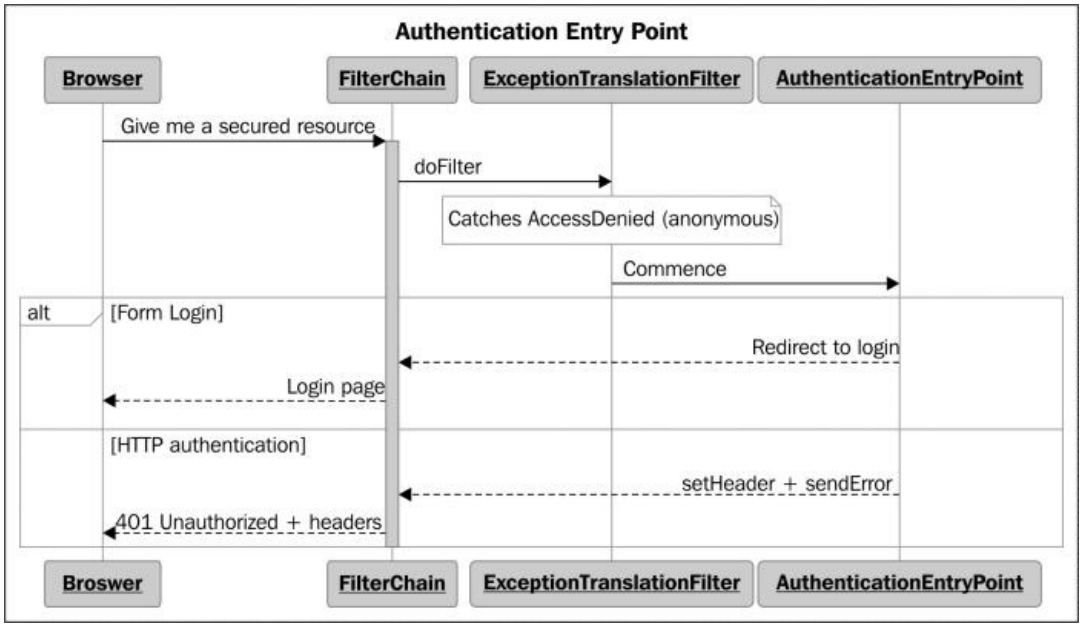
> **Tip**
>
> In fact, using `<http auto-config="true">` automatically wires HTTP Basic too.

To make crystal clear what is going on under the hood, we can use a more verbose configuration syntax and wire the required beans manually:

```
 <http use-expressions="true"
entry-point-
ref="basicEntryPoint" create-session="stateless">
   <intercept-url pattern="/backend/**"
access="isAuthenticated()" />
       <custom-filter ref="basicFilter"
position="BASIC_AUTH_FILTER"/>
 </http>
 <beans:bean id="basicEntryPoint"
class="org.springframework.security.web.authentication.www.BasicAuthentic
       <beans:property name="realmName"
value="Spring Security BasicRealm"/>
 </beans:bean>
 <beans:bean id="basicFilter"
class="org.springframework.security.web.authentication.www.BasicAuthentic
       <beans:constructor-arg
ref="authenticationManager"/>
       <beans:constructor-arg
ref="basicEntryPoint"/>
 </beans:bean>
```

As you can see, there are two beans needed to set up HTTP Basic: `BasicAuthenticationEntryPoint` and `BasicAuthenticationFilter`.

An entry point, as the name suggests, is a class that sets up the authentication scheme after access to the secured resource is denied. In a typical password-login scenario, when the user clicks on a secured link, a `LoginUrlAuthenticationEntryPoint` entry point is called. Then, it sends an HTTP redirect, pointing to a login page defined in the `<form-login>` tag. In the case of HTTP Basic, `BasicAuthenticationEntryPoint` sets a `WWW-Authenticate` HTTP header and sends back the **401 Unauthorized status** code as shown in the following diagram:



When the browser resends the request with an `Authorization` header, the `BasicAuthenticationFilter` class comes into play. It decodes the login and password from the header and invokes `AuthenticationManager` (injected as a property) to validate the credentials. When authentication fails, the filter calls `BasicAuthenticationEntryPoint` to reinitialize the HTTP Basic Authentication scheme. That's why when you provide invalid credentials, the browser displays the login form over and over again, unless you've closed it by pressing the *Esc* key.

You may also have noticed the `create-session` parameter in the preceding config. This tells Spring Security to not store authentication objects in the sessions—each request is authenticated separately. RESTful purists should be happy now.

Ok, our application should handle HTTP Basic Authorization, so let's try to test the configuration. Assuming that the `/backend/password` URL returns some data, you can open the URL in the browser and the standard login form will appear. After valid credentials have been provided, the data should be displayed. Great! We've set up HTTP Basic Authentication.

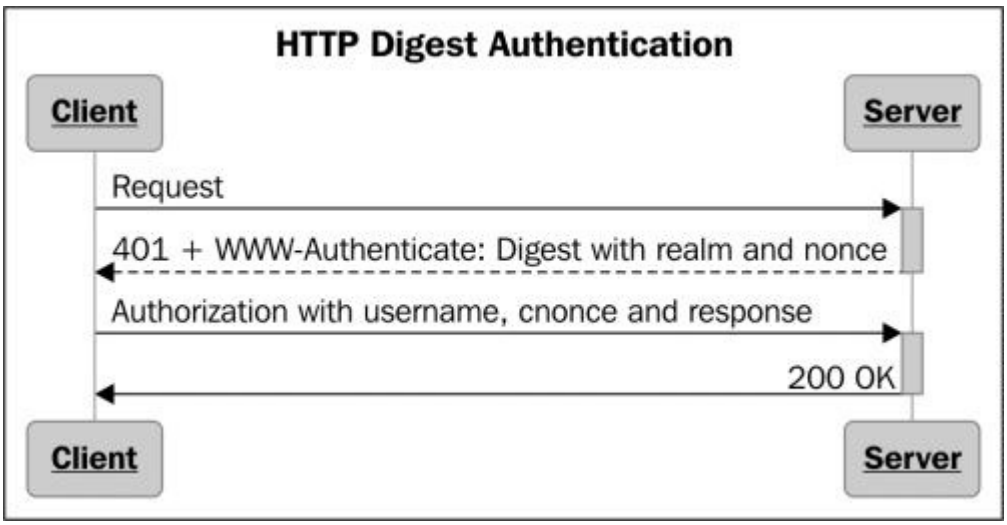
If you're a *nix user, you can test the configuration without even opening the browser—just use the `curl` command. Open the terminal and run:

```
curl --basic -u piotr:piotr123
http://localhost:8080/spring-security-
starter/backend/password
```

You should see the result of the call. In case of any problems, you could use the `-v` switch to turn up the verbosity and see the request and response exchanges with the server.

Digest

The problem with HTTP Basic is that the credentials are passed in plain text, making it quite unusable in a non-HTTPS environment. Here shown is HTTP Digest Authentication that is slightly more secure:



The scenario begins the same as in HTTP Basic: the server responds with a `401` status and sets the `WWW-Authentication` header. But, the content of the header is a bit different:

```
WWW-
Authenticate: Digest realm="Spring Security
Digest Realm", qop="auth",
nonce="MTM2NDIxNjgwOTQyODpiYTljZmE3MTIwYzhlODNhODc5YjVlOWI3MmRkYTI4NA=="
```

In addition to switching the keyword from `Basic` to `Digest`, two more values —`qop` and `nonce`—are being returned. The `qop` (**Quality of Protection**) value is one of `auth`, `auth-int`, or some other token. The `nonce` (**Number Once**) string is an arbitrary server-generated string that can only be used once (every nonauthenticated call should return a new `nounce` value). These two values are required to apply the following algorithm that computes the second call from the client to the server.

Here is how the algorithm looks for `qop=auth`, which is a default for Spring Security:

```
HA1 = MD5(username:realm:password)
HA2 = MD5(method:URI)
```

```
response = MD5(HA1:nonce:nc:cnonce:qop:HA2)
```

Where `nc` is the `nounce` counter, `cnonce` is the `nonce` string generated by the client and `MD5` is the MD5 Message-Digest Algorithm. Here is the final content of the `Authorization` header of the client's second invocation:

```
Authorization: Digest username="piotr",
realm="Spring Security Digest Realm",
nonce="MTM2NDIxNjgwOTQyODpiYTljZmE3MTIwYzhlODNhODc5YjVlOWI3MmRkYTI4NA==",
uri="/spring-security-
starter/backend/secured", cnonce="MDIyNjIy",
nc=00000001, qop="auth",
response="a7b4dce58c24a6f0a1202752043b1ba7"
```

To provide HTTP Digest Authentication, you must wire appropriate filters and an entry point into your Spring Security context just as in HTTP Basic:

```
 <http use-expressions="true"
entry-point-
ref="digestEntryPoint" create-session="stateless">
   <intercept-url pattern="/backend/**"
access="isAuthenticated()" />
       <custom-filter ref="digestFilter"
after="BASIC_AUTH_FILTER"/>
 </http>
 <beans:bean id="digestEntryPoint"
class="org.springframework.security.web.authentication.www.DigestAuthenti
       <beans:property name="realmName"
value="Spring Security Digest Realm"/>
       <beans:property name="key"
value="springSecurityNonce"/>
       <beans:property
name="nonceValiditySeconds" value="120"/>
 </beans:bean>

 <beans:bean id="digestFilter"
class="org.springframework.security.web.authentication.www.DigestAuthenti
       <beans:property name="userDetailsService"
ref="userAccountDetailsService"/>
       <beans:property
name="authenticationEntryPoint"
ref="digestEntryPoint"/>
 </beans:bean>
```

The `DigestAuthenticationEntryPoint` class is configured with `realmName`, `key`, and `nonceValiditySeconds` (nonce validity time). Nonce must be used within the configured time from when it has been generated. And the key is used by the nonce computation algorithm:

```
base64(expirationTime + ":" +
md5Hex(expirationTime + ":" + key))
```

`DigestAuthenticationFilter` validates and decodes values passed by the client in the `Authorization` header and then computes the digest and compares it with that received in the "response" value.

The important thing here is that the digest is being computed with a plain-text password, and the official Spring Security docs state the following:

> *"The configured `UserDetailsService` is needed because `DigestAuthenticationFilter` must have direct access to the clear text password of a user. Digest Authentication will NOT work if you are using encoded passwords in your DAO."*

This is not good news for us since we are using a standard password encoder to encode passwords before storing them in the database. However, looking deeper into the `DigestAuthenticationFilter` code, we might find a better solution. After retrieving the user with `UserDetailsService`, the filter executes the following code:

```
if (passwordAlreadyEncoded) {
  a1Md5 = password;
} else {
  a1Md5 =
DigestAuthUtils.encodePasswordInA1Format(username,
realm, password);
}
```

Setting `passwordAlreadyEncoded` should force Spring Security to just use thepassword from the database as `HA1`, which is a value of `MD5(username:realm:password)`. Remember what we've said about salt in the *Password encoders* section? It is a value that is added in front of the password before it is encoded with the password encoder. To make this work we need to:

- Use a salt of `username:realm`
- Use a password encoder that adds `salt:` in front of the plain-text password and `MD5` as the hashing algorithm
- Set `passwordAlreadyEncoded` to `true` in `DigestAuthenticationFilter`

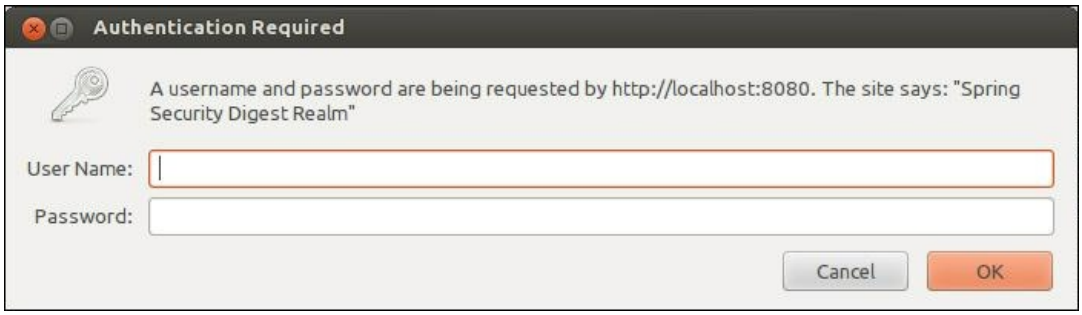
It would still be bad if someone stole our database because with all the `HA1` values, it would be possible to compute the whole response and therefore authorize it in our application. But since salt has started being used, it is not possible to use rainbow tables to guess the real user's password. And using

a realm as a part of salt makes other applications with different realms safe. As the original RFC 2617 states:

"The realm string should be unique among all realms which any single user is likely to use. In particular a realm string should include the name of the host doing the authentication."

Dealing with the ugly login dialog

The preceding solution should work fine with backend-to-backend integration. But at the beginning of the section, we mentioned single-page applications; they typically look much better than this:



The problem is that every **HTTP 401 Unauthorized** response ends with this dialog shown, irrespective of whether it is a real user clicking on a link or just an AJAX REST call to the server. To deal with this, we can make a little hack that will make our authentication scheme a bit different from the original HTTP Basic/Digest: instead of `401`, we can return a `403` status code for the first request returning the `WWW-Authenticate` header. To implement this, we just need to subclass `DigestAuthenticationEntryPoint` with a custom implementation:

```
public class
SinglePageAppAuthenticationEntryPoint extends
DigestAuthenticationEntryPoint {
  @Override
  public void commence(HttpServletRequest request,
   HttpServletResponse response,
AuthenticationException authException) throws
IOException, ServletException {
   super.commence(request, new
StatusCodeMappingHttpResponse(response),
authException);
  }

  private static class
StatusCodeMappingHttpResponse extends
HttpServletResponseWrapper {
```

```
    public
StatusCodeMappingHttpResponse(HttpServletResponse
response) {
     super(response);
    }

  @Override
  public void sendError(int sc, String msg)
throws IOException {
  if (sc == HttpServletResponse.SC_UNAUTHORIZED) {
  sc = HttpServletResponse.SC_FORBIDDEN;
      }
    super.sendError(sc, msg);
    }
  }
}
```

We've wrapped the `HttpServletResponse.sendError` method that has been called by the `commence` method and then mapped `401` to `403 Forbidden`. The dialog no longer appears, so our AJAX client only has to handle the `403` status code in the JavaScript code and retry the request with the appropriate value of the `Authorization` header. In the example GitHub project we refer to through this book, we've used AngularJS and the HTTP response interceptors pattern. You can find similar approaches in any JavaScript framework, for example, JQuery.

What else you may want to know

Hopefully, we've gotten a basic grasp of Spring Security and also seen some interesting features we can add. A book for one evening, like this one, has to be short, so we can't describe every aspect or every trick we've learned. And we can't go too deep into the details. That is a job for the documentation and for other books. Spring has an extensive collection of books and reference documentation. As we come to an end, we will try to broaden your view on a few more things that you may find useful.

Internet authentication – because login/password is so 80s

Unless you have a two-factor authentication, form authentication will always be vulnerable because people are going to use the same password for multiple services. Two-factor authentication with a one-time password is just so good that everybody should be using it, but it's very annoying to the user.

If the application is sending the second password via SMS, I'll need to have my phone with me. And while it's not a big problem, my phone may be laying on a night table, which is out of reach at the moment, so if I don't really have to use that service; I'm going to skip authentication and search

somewhere else for what I need. And if the application is sending the second password via e-mail, I need to either fire up my e-mail client or go to my web e-mail to get it. And if I'm using a phone that is not so good at multitasking... well, that's one more reason to skip authentication.

People are lazy. The more they have to do to pass a security check, the fewer of them are going to use the service. This usually means you are not going to earn all the money you could. Your clients are turning away at your doors because those doors are too cumbersome to open.

After all, this is the same reason why people have a single password (or a few of them) for all websites. People are lazy, and they know it.

People also know that any security can be broken. What they do not know is how good your security is. So from their perspective, registering to your service by creating a new account and using their single password puts them at risk if your system is hacked.

This, by the way, is what CVC2/CVV2 codes for credit cards look like. The little three or four digits, at the back of your card? These are plain text, and you often have to give them away while making a purchase on the Internet. Merchants are required to not store them in their databases. Unfortunately, they do keep them in their databases, and so 77 million PlayStation Network users had to block their credit/debit cards in 2011.

So is there a way to have the benefits of a strong, two-factor authentication mechanism and still be lazy, not requiring to remember more than one password for all the services?

There is. It's called **OpenID**.

OpenID 2.0

The concept is really simple. Let's say there is a service you trust, such as Gmail for example. It has a two-factor authentication, and you are quite sure security is taken seriously. You are also almost always logged in to this favorite service of yours. Now you want to log in to another Internet service. You don't want to create a new login and password, so you may just say to this new service: "hey, I'm already logged in to GMail. Ask Google who I am."

The service asks, Google responds, and you are in, even though you never had to enter your password on any other site other than Google.

OpenID is a Single Sign-On standard that works on the Internet.

In theory, everyone can create their own OpenID identity provider (the service that confirms who you are). In practice, most people already use a service that is also an identity provider, such as Google, Facebook, Twitter,

and Yahoo! Even Microsoft announced that it would support OpenID for their Windows Live services. This is why you see so many **login with …** buttons out there.

Now back to Spring Security. Our favorite library has built-in support for OpenID 2.0. We won't describe how to use it here. Instead, we would like to point you to two excellent options that do the job quite well, in our opinion.

First, we recommend you read *Spring Security 3* by *Peter Mularien*. This book is great, but you can read the following article for free at http://www.packtpub.com/article/opening-up-to-openid-with-spring-security.

And the second one, written by yours truly, is here:

http://blog.solidcraft.eu/2011/04/spring-security-by-example-openid-login.html

Both take you through the process step by step, but the whole thing is quite simple.

So is OpenID the solution to everything? Can we skip other authentication protocols? Can we get rid of our old form authentication?

While you may be tempted, do not skip form logging for your own services because you will lose some customers who just do not want to allow you to connect their public information with the services you provide. Respect their privacy and allow them to separate their life where they want to.

OpenID 2.0, unfortunately, is not the solution to everything. However, OAuth 2.0 may be.

OAuth 2.0

OpenID is all about authentication and thus doesn't cover much more than saying "I know who this guy is." Since you may need some additional information about the user, apart from "this guy is well known", such as an e-mail or a name, the OpenID standard comes with something called **Attribute Exchange** that allows your service to ask the OpenID provider about some data of the user. And that's it.

OAuth 2.0, on the other hand, is about authorization. It allows you to ask not only "who this user is" but also "can I do something on behalf of this user?", such as access his information, write on his wall, add something to his favorites, or save his documents in his Google Drive, or... anything. You can give another application whatever access you think it should have, and while this may sound scary, OAuth providers are quite good at describing what exactly you are agreeing to before you make the choice.

OAuth 2.0 is complementary to OpenID, and you may often see those two protocols together even though OAuth 2.0 does the authentication quite fine itself.

OAuth 2.0, on the client side, is simple. You can implement it yourself in no time. But as with everything in the Java world, you don't have to. It's already done. To use OAuth 2.0 with Spring Security, there is an extension called **Spring Social**. You can find it at http://www.springsource.org/spring-social.

People and places you need to know about

As mentioned a few times before, this starter guide is just an introduction to the world of Spring Security. It shows you how to inject a security context into your existing application with a basic setup. However, it is obvious that while developing complex web applications, you'll face some problems requiring a more complex configuration. Here are some useful links that could help you fight with your developer frustration while debugging Spring Security.

Official sites

- **Home page**: http://static.springsource.org/spring-security/site/index.html
- **Reference and API documentation**: http://static.springsource.org/spring-security/site/docs/3.1.x
- **Recommended by Spring Source (300+ pages book)**: http://www.springsecuritybook.com/

Articles, tutorials, and blogs

- **Mkyong – Spring Security tutorial**: http://www.mkyong.com/tutorials/spring-security-tutorials/
- **Jakub Nabrdalik – Spring Security by example**: http://blog.solidcraft.eu/2011/03/spring-security-by-example-set-up-and.html
- **Behind the Spring Security namespace**: http://blog.springsource.org/2010/03/06/behind-the-spring-security-namespace/
- **Eugen Paraschiv – securing a REST service with Spring Security**: http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/

Community

- **Official forums**: http://forum.springsource.org/forumdisplay.php?33-Security
- **Official issue tracker**: https://jira.springsource.org/browse/SEC
- **StackOverflow questions**: http://stackoverflow.com/questions/tagged/spring-security
- **Official Twitter handle**: https://twitter.com/SpringSecurity