Community Experience Distilled

Spring Security Essentials

A fast-paced guide for securing your Spring applications effectively with the Spring Security framework

Nanda Nachimuthu

[PACKT] open source* PUBLISHING
community experience distilled

Spring Security Essentials

A fast-paced guide for securing your Spring applications effectively with the Spring Security framework

Nanda Nachimuthu

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Spring Security Essentials

First published: January 2016

Production reference: 1060116

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-262-1

www.packtpub.com

Credits

Author
Nanda Nachimuthu

Reviewer
Vinoth Kumar Purushothaman

Commissioning Editor
Dipika Gaonkar

Acquisition Editor
Kevin Colaco

Content Development Editor
Preeti Singh

Technical Editor
Pranil Pathare

Copy Editor
Vibha Shukla

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Nanda Nachimuthu works as a principal architect with Emirates Airlines, Dubai. He grew up in a joint family set up and holds an engineering degree from Tamil Nadu Agricultural University and an advanced Internet programming certification from IIT Kharagpur.

He has 18 years of experience in IT, which includes 12 years as an architect in various technologies such as J2EE, SOA, ESB, Cloud, big data, and mobility. He has designed, architected, and delivered many national and large-scale commercial projects. He is also involved in design and development of various products in the insurance, finance, logistics, and life sciences domains.

His hobbies include travelling, painting, and literature. He is also involved in various pro bono consulting activities, where he finds a way to utilize his extra time and innovative ideas in order to become practical and useful for the society. He is the founder of `JCOE.in`, a portal that deals with the Java Center of Excellence (CoE) activities, which is useful for the Java community and companies.

First, I would like to thank my wife Rathi for pushing me to man up and complete the book. Next, I would like to thank my mom Maruthayee for her blessings, encouragement, and moral support. I cannot simply forget the cooperation of my daughter Shravanthi and son Shashank, who have always played and fought with me since the inception of this book, which turned out to be a great help for me to reduce some stress.

About the Reviewer

Vinoth Kumar Purushothaman, a graduate from University of Madras, specializes in architecture design. He has 18 years of experience in design and development of large-scale applications in banking, telecommunication, automobile, e-commerce, and life sciences using Java, J2EE, service-oriented architecture framework components and big data.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Spring Security Essentials focuses on the Spring Security framework. There are three essential aspects to application security: authentication, authorization, and access control list (ACL). We will be concentrating on these three aspects in this book. This book will teach the readers the functionalities required to implement industry-standard authentication and authorization mechanisms to secure enterprise-level applications using the Spring Security framework. It will help the readers to explore the Spring Security framework as a Java model and develop advanced techniques, including custom user realms, custom authorization constraints, method-based authorization, and instance-based authorization. It will also teach up-to-date use cases, such as building a security layer for RESTful web services and applications.

Spring Security Essentials focuses on the need to master the security layer, which is an area that is not often explored by a Spring developer. The IDEs that are used and the security servers that are involved are briefly explained in the book, including the steps to install them. Many sample projects are provided in order to help you practice your newly developed skills. Step-by-step instructions are provided to help you master the security layer integration with the server, and then implement the experience gained from this book in your real-time application.

What this book covers

Chapter 1, *Getting Started with Spring Security*, explores the various flavors of Spring Security implementations that are available in the Spring 4.0.3 framework, along with the Spring 3.2.3 module. We dive into each of the options in detail with the help of practical examples. I recommend you have a good understanding of the application development environment (ADE) for various technologies that we will address, such as LDAP, SAML, Wicket, and so on.

Chapter 2, *Spring Security with SAML*, covers the basics of the Spring 4.0 Web MVC creation and build tools, such as Maven and Gradle, as a recap and practice session. We create a web-based MVC project and explore the open source implementations of SAML 2.0 that are available as Identity providers.

You will learn about Spring 4.0 SAML Extensions in order to implement single sign-on and sign-off by connecting to the SSOCircle web-based authentication mechanism.

Chapter 3, *Spring Security with LDAP*, covers the basics of LDAP and the different implementations available. It covers the features of Apache Directory Server and the steps involved in installing ApacheDS and Studio with Spring Tool Suite. We will create a directory and the values for different departments and users.

Chapter 4, *Spring Security with AOP*, explains the basic terminologies of Aspect-Oriented Programming. We go through a few simple examples of Spring AOP and AspectJ. The use of annotation is explained using samples and we will implement AOP security for method-level and UI Component creation. You can extend the features and implementations that are described in this chapter in your real-time applications in order to avoid the complexities that are involved in cross-cutting concerns.

Chapter 5, *Spring Security with ACL*, introduces the basics of access control lists and the available classes and interfaces in the Spring ACL package. We will see a few working examples of the basic ACL implementation with various access privileges for a given principal.

Chapter 6, *Spring Security with JSF*, covers the JSF basics and required Spring Security configurations. We create a sample project from scratch and explain each artifact.

Chapter 7, *Spring Security with Apache Wicket*, starts with basic the Apache Wicket application structure and a sample project. We cover the configurations that are required from the Spring perspective and dependencies required in the Maven POM file. We make the security credentials settings in the Spring Security file and execute the sample application by entering different security credentials for different types of user.

Chapter 8, *Integrating Spring Security with SOAP Web Services*, covers the basics of the Spring Web Services package and the different types of SOAP Web service creation. We execute and test the authentication of the SOAP message as well.

Chapter 9, *Building a Security Layer for RESTful Web Services*, starts with basics of RESTful web services and their advantages. We develop a basic Spring implementation to configure the Security credentials entry points and success handlers. We also execute RESTful web services through the cURL command-line utility to check Spring Security authentication in action.

Chapter 10, *Integrating Spring Security with JAAS*, covers JAAS basics, Spring JAAS Security package components and developing a Spring JAAS implementation project and executing it.

What you need for this book

You need to have fair knowledge of Java, and knowing the basics of Spring is recommended.

Who this book is for

If you are a developer who is familiar with Spring and are looking to explore its security features, then this book is for you. All beginners and experienced users will benefit from this book as it explores both the theory and practical use in detail.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In these scenarios, we will have to set the security authorization constraints in a secured way in the `web.xml` file."

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The user clicks on the **Logout** button and the instance executes the logout script."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/2621OS_ColouredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started with Spring Security

When we talk about enterprise security, three major areas of security—authentication, authorization, and **access control list** (**ACL**)—will play a major role. The Spring Framework 4.0.3 has a seven-layered architecture that includes a core container, context, **Aspect-Oriented Programming** (**AOP**), **Data Access Object** (**DAO**), **Object-relational mapping** (**ORM**), Web, and **Model-View-Controller** (**MVC**). To provide security features to all these layers, we have The Spring Security 3.2.3 module, which will provide security facilities such as user authentication and authorization, role-based authorization, database configuration, password encryption, and others.

In general, Spring developers focus on the seven layers to develop the web applications, and most of them will not be able to master the security mechanisms involved in different layers with different implementations as they might have to call the abstract programs in which the security implementations are built.

Spring 3.2.3 supports various authentication approaches for different industry standard connectivity for Java EE-based enterprise applications. Many people use Spring Security in the layers of Java EE's Servlet Specification and **Enterprise Java Beans** (**EJB**) Specification, which will limit the usage of proper Spring Security implementations. Due to this, many enterprise security scenarios are left unattended. Authentication is the process of creating a principal in the enterprise system for which a user needs to provide credentials. The role-based access privileges will be decided on a predefined role authorizer system from which the core system will read the access rights for the given principal. The advanced techniques of the Spring Security mechanisms are as follows:

- Custom user realms
- Custom authorization constraints

- Method-based authorization
- Instance-based authorization
- Building a security layer for RESTful web services

The following modules of Spring 3.2.3 support the implementation of enterprise security:

- Spring Security Core
- Spring Security remoting
- Spring Security Web
- Spring Security configuration
- Spring Security LDAP
- Spring Security ACL
- Spring Security CAS
- Spring Security OpenID

Additionally, we will cover specific techniques such as **JavaServer Faces** (**JSF**) 2.0, Wicket, and **Java Authentication and Authorization Service** (**JAAS**). The following are the new security features provided in Spring 4.0, which we will talk about later:

- Web socket support
- Test support
- Spring data integration
- **Cross-Site Request Forgery** (**CSRF**) token argument resolver
- Secure defaults

Most of these authentication levels are from third parties or developed by relevant standard bodies such as **Internet Engineering Task Force** (**IETF**). Spring Security has its own authentication features that will be useful to establish connections securely with third-party request headers, protocols, and single sign-on systems. We will have a detailed description of each system and mechanism in the following chapters.

Spring custom user realms

Custom security realms facilitate you to use an existing data store such as a directory server or database when authenticating and authorizing users to enterprise applications, which are deployed in a standard application server, such as WebSphere, JBoss, and so on. We will have to provide the attribute details to the server to create the user realms such as the name, realm class name, configuration data, and password. We can create a custom realm by providing a custom JAAS login module class and custom realm class. However, when we use the client-side JAAS login modules, this may not be suitable for use with the enterprise server.

There can be two different realms that cater to two different URL patterns. We can use the same authentication logic for both the realms. The standard Spring Security mechanism will invoke `j_spring_security_check` automatically when a login form is getting called, and we can define our own URLs that are to be intercepted. This approach is called browser-based client security realm. If the user has not been provided with a username and password and if the principal is not created to access this URL, then the user will be redirected to the login page by the Spring Security checker.

Spring custom authorization constraints

There are many types of security constraints. This consists of web resource collections such as URL patterns, HTTP methods, and authorization constraints by providing role names. User data constraints such as web requests are passed over an authenticated transport. A security constraint is used to define the access privileges for a collection of resources using their URL mapping. The security token will be given from an HTTPS request when it gets validated and will be given back to the enterprise application server. There may be possibilities that the security token does not return any valid roles for authorization.

In these scenarios, we will have to set the security authorization constraints in a secured way in the `web.xml` file. The web resources can have unchecked access, checked access, and no access. We can omit the authorization constraints so that any web resource can access the resource. We can specify the role name for the authorization constraint so that only these roles can access the web resource. We can also exclude a set of web resources from accessing any request by specifying no roles for these resources. We can also exclude particular URLs to access the specific secured web resources.

Spring method-based authorization

Method security is a bit more complicated than a simple allow or deny rule. Custom methods can be provided with specific security settings. In Spring, we can achieve this by providing the proper annotations for the methods to be secured. There are four annotations that support expression attributes to allow preinvocation and post-invocation authorization checks and also support the filtering of the submitted collection arguments or return values. They are `@PreAuthorize`, `@PreFilter`, `@PostAuthorize`, and `@PostFilter`. If you want to create a custom secured method called `customCheckUser()`, then you can annotate the method with the `@PreAuthorize` tag for a presecurity check before execution.

While the other security methods focus on servlets and controllers, security method-based authorization deals with the service layer components particularly. We can control various services to be accessed by specific principals. For example, an administrative principal can access only the database credential layer or the logging layer can be accessed by all the principals. The global method security tag or the `@EnableGlobalMethodSecurity` annotation will help developers in setting up the method level security.

Spring instance-based authorization

At the class level, we can check whether the intended principal is authorized to invoke the particular instance or not when we create an instance for a particular request. This can be achieved by providing annotations before instantiating the object in order to check the authenticity. This instance-based security is important in handling non-application server-related code or any other code related to the business logic that needs to be closely monitored to prevent non-privileged access.

The approach here is to define the information clearly so that the domain object-based security restrictions can be applied accurately. The Actor who is performing the use case action, the domain acted created internally to perform the action, and the intended action are the three pieces of information that we need to define clearly in order to achieve instance-based authorization. Here comes the usage of ACLs and **access control entries** (**ACEs**), which will be elaborated on in further chapters. The advantage of using Spring ACL and ACE here is that Spring has an internal mechanism to manage the ACE volume by implementing the ACE inheritance mechanism so that when a number of domain objects increases, the ACEs also will become manageable.

Apart from these techniques, Spring provides you with options to build a security layer for RESTful and SOAP web services, and we can create security layers for JAAS, JSF 2.0, and Wicket. Let's take a quick look at these four techniques now.

Spring Security with SOAP web services

Spring Web Services (**Spring-WS**) packages focus mainly on the creation of document-driven web services, where the data communication between web services is done through XML envelopes and web services can be accessed from any other technology application server. The features supported by Spring-WS are powerful XML mappings, support for various XML APIs, flexible XML marshalling, support for WS-Security, and others. WS-Security comprises of three areas—authentication, digital signatures, and encryption/decryption.

The security flow in Spring Web Services will be as follows. The system will generate a security token for a valid principal using a separate web service method. If the user wants to access other web services, he or she should pass this token along with the payload as a security key and these web services will validate this token for authenticity and then allow the users to access the resources. If the token has expired or is invalid, the user should go through the authentication web service once again. This entire mechanism is called message signing.

Spring Security with RESTful web services

To achieve **Representational State Transfer** (**REST**) services calls with basic security authentication, we will have to depend on the libraries provided by the Spring framework, such as the `core`, `configuration`, and `web`. We also need to make some entries in the Spring application context files.

In real-time scenarios, we will have to get the credentials from **Lightweight Directory Access Protocol** (**LDAP**), Database, and others.

Spring Security with JSF2.0

Coming to the JSF and Spring Security integration, the Spring web flow provides you with a JSF integration that simplifies the handshake between JSF and Spring. A dedicated Spring Security tag library is available for JSF Security integration. To achieve this, `springsecurity.taglib.xml` needs to be updated with facelet entries. These modifications must be reflected in `web.xml` as well. We can include nested contents based on security conditions using the authorize tags. During JSF rendering, many expression language-based functions can be used.

Spring Security with Wicket

Apache Wicket is designed based on a component-oriented structure and less HTML file handling. Wicket-related security settings must be handled first by modifying the `web.xml` file for the corresponding filter mapping. As a Wicket programmer, you will need to have a clear understanding about the pull and push concepts and form processing life cycle of the Apache Wicket framework. There are two unique issues to be handled from Wicket. Wicket does not manage the life cycle of its components, and the components and models of Wicket are often serialized, which may be an issue for Spring's dependency injection mechanism. The work around this will be some entries in the Web and ApplicationContext XML files, but this approach will have its own pros and cons, which we will discuss later.

Spring Security with JAAS

The Spring framework has a JAAS authentication provider, which must be configured in an `applicationcontext.xml` file. We need to create an array of entries for the URLs that need to be secured. We have to define the security policies for different URLs of the website. JAAS will expect a callback from the user—the username and password. Spring will have this information collected and populated on an authentication object, which will be passed to JAAS as an input.

Spring Security with SAML

Security Assertion Markup Language (**SAML**) is a popular open standard, which simplifies federated user logins. A user can provide credentials to a centralized enterprise registry, and using this principal, the user can access other independent applications that are mapped with the centralized registry.

This is called single sign-on implementation using the Spring and SAML integration. We can also create a common setup to make an enterprise an **single sign-on** (**SSO**)-enabled one with the following certain standards. This is based on how we set up Spring and SAML to pass the SAML tokens to the other applications that are using the SSO. We can create a shared cookie that will contain the authorized SAML token. Additionally, we can develop an internal SAML token verifier, which may frequently assess the validity of the token. The `securityContext` XML file needs to be updated with the IDP metadata. **IDP** is nothing but the centralized **Identity provider**.

Spring Security with LDAP

You must be aware of the LDAP basics, and you can refer to a popular open source LDAP implementation called **OpenLDAP** if you want to further explore. Spring has an LDAP package that is helpful in accessing many LDAP implementations without bothering much about their internals. This is developed based on the JdbcTemplate package design. Basic operations such as looking up, context initiation and closing, iterating through the results, and encoding/decoding the values are taken care of by this package. On top of this, Spring LDAP comes with various enhanced features such as LDAP template, LDAP context, LDAP filters, LDAP transaction management, and others.

Summary

We have seen the various flavors of the Spring Security implementations available in the Spring Framework 4.0.3 along with the Spring 3.2.3 module. We will explore each of these options in detail with practical examples in the coming chapters. We recommend that you have a good understanding of the application development environment for various technologies that we will address, such as LDAP, SAML, Wicket, and so on. In the next chapters, we will explain the security implementations that include the basics of the IDE setup, understanding a sample source code, building mechanisms, and so on.

2
Spring Security with SAML

In this chapter, we will explore the various security integration options with Spring and SAML. Many of us are aware of the basics of **Security Assertion Markup Language** (**SAML**), which is a standard way of providing authentication and authorization information from an Identity provider to a service provider. For Intranet, an application providing **single sign-on** (**SSO**) and **single logout** (**SLO**) is possible and easy using Local Cookies Information, whereas it is difficult to implement single sign-on for Internet-based applications. So, we need a sophisticated web browser-based SSO implementation using standard technologies such as the SAML open standard data format.

Spring comes with a standard extension for SAML that will facilitate the federated applications to integrate with existing SAML implementations. Refer to the popular SAML implementations such as Shiboleth, Kerboros, and many more, which have identity management capabilities, and some of them are available on the cloud as well. The Spring SAML extension is flexible in such a way that you can integrate SAML SSO and other authentication mechanisms in a single application without affecting each other.

We will cover the following topics in this chapter:

- The basics and structure of SAML 2.0
- A recap of the Maven build tool
- A recap of the Gradle build tool
- The basic project creation and execution of the Spring tool suite
- Open source SAML 2.0 implementations
- Identity provider configurations and registrations
- Spring SAML extensions usage

The basics and structure of SAML 2.0

SAML 2.0 is an XML-based protocol that facilitates the passing of the session information in the form of a security token. These tokens will be carrying the authentication and authorization information of the principal across the web servers involved. The cross-domain single sign-on is possible using an XML protocol such as SAML, which involves an Identity provider (SAML authority) and service web server (SAML consumer) that will get the security tokens from the SAML implementation. With this mechanism, we will be able to avoid maintaining principal credential information in many areas that in turn will make the security ecosystem a robust one.

The SAML 2.0 critical aspects are SAML conformance, SAML core module, SAML bindings definitions, and SAML profiles information. Let's take a quick look at these critical aspects:

- The **conformance program** specification ensures interoperability between cross-domains while exchanging authentication and authorization information. It also standardizes the conformance test development. On a basic level, it provides a common understanding of the conformance process and what is required to claim conformance. The SAML bindings and profiles, which are supported by the participating applications or implementations, must be expressed as a conformance.
- The **SAML core** provides you with the specifications for assertions and protocols. This module provides you with ways to use notations, schema organization, namespaces, and so on. SAML assertions and protocols are typically embedded in industry-standard protocols such as HTTP POST requests or XML-encoded SOAP messages. You can refer to the SAML assertion schema and SAML protocol schema documents to understand more about the keywords and conventional XML namespace prefixes.

- The **SAML binding** specifications provides you with details about protocol binding concepts, notations, guidelines to specify additional protocol bindings, and others. Bindings are important when using SAML assertions and request-response messages in communication protocols and frameworks. If we map SAML request-response message exchanges to the HTTP protocol, then this binding will be called HTTP SAML binding. This is to make sure that the SAML implementation software can interoperate with the applications built on top of a standard messaging or communication protocol.
- A **SAML profile** is nothing but a set of rules describing how to embed a SAML assertion in a framework or protocol and how to fetch the SAML assertion from the framework or protocol. Another type of profile defines a set of rules to use the specific SAML functionality such as attributes, conditions, and bindings. The SAML provider must ensure that the profiles are defined clearly so that the SAML consumer can interoperate with all the details required to exchange the authentication and authorization information.

Let's take a close look at some of the important SAML components such as assertions, bindings, and profiles.

SAML 2.0 assertions

An assertion is nothing but a collection or package of information that is bundled and distributed by the SAML authority to the SAML consumers. SAML 2.0 comes with three types of assertion statements called authentication, attribute, and authorization decision:

- **Authentication assertion** is the user that has proven his or her identity
- **Attribute assertion** carries specific information about the principal that will help the system to understand the limits or parameters of the users
- **Authorization decision assertion** has the authorization details such as resource access and role access

A SAML assertion XML file may have child elements, as shown in the following screenshot:

- `saml:Issuer` element: This is the unique identifier of the Identity provider
- `saml:Subject` element: This identifies the authenticated principal
- `saml:AuthnStatement` element: This is the authentication level of the Identity provider

```
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ID="b07b804c-7c29-ea16-7300-4f3d6f7928ac" Version="2.0" IssueInstant="2004-12-05T09:22:05">
  <saml:Issuer>https://idp.packt.org/SAML2example</saml:Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">...</ds:Signature>
  <saml:Subject>
    <saml:NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient">
      3f7b3dcf-1674-4ecd-92c8-1544f346baf8
    </saml:NameID>
    <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <saml:SubjectConfirmationData InResponseTo="aaf23196-1773-2113-474a-fe114412ab72"
      Recipient="https://serviceprovider.packtpub.com/SAML2/SSO/POST" NotOnOrAfter="2004-12-05T09:27:05"/>
    </saml:SubjectConfirmation>
  </saml:Subject>
  <saml:Conditions NotBefore="2004-12-05T09:17:05" NotOnOrAfter="2004-12-05T09:27:05">
    <saml:AudienceRestriction>
      <saml:Audience>https://serviceprovider.example.com/SAML2</saml:Audience>
    </saml:AudienceRestriction>
  </saml:Conditions>
  <saml:AuthnStatement AuthnInstant="2004-12-05T09:22:00" SessionIndex="b07b804c-7c29-ea16-7300-4f3d6f7928ac">
    <saml:AuthnContext>
      <saml:AuthnContextClassRef>
        urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
      </saml:AuthnContextClassRef>
    </saml:AuthnContext>
  </saml:AuthnStatement>
  <saml:AttributeStatement>
    <saml:Attribute xmlns:x500="urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500" x500:Encoding="LDAP"
      NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
      Name="urn:oid:1.3.6.1.4.1.5923.1.1.1.1" FriendlyName="eduPersonAffiliation">
      <saml:AttributeValue
        xsi:type="xs:string">member</saml:AttributeValue>
      <saml:AttributeValue
        xsi:type="xs:string">staff</saml:AttributeValue>
    </saml:Attribute>
  </saml:AttributeStatement>
</saml:Assertion>
```

SAML 2.0 protocols

In the SAML core package, assertion query and request protocol, authentication request protocol, artifact resolution protocol, name identifier management protocol, single logout protocol, and name identifier mapping protocol are specified. Out of these, the authentication request protocol and artifact resolution protocol are very important. Let's see the description of each of these:

- **Assertion query and request protocol**: We can query and request existing assertions by passing their subject and statement types.
- **Authentication request protocol**: An authenticated principal can fetch assertions by sending a message element to the SAML authority. With this protocol, the SAML consumer can establish a security context with one or more participating applications, as follows:

```
<samlp:AuthnRequest xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
  ID="aaf23196-1773-2113-474a-fe114412ab72"  Version="2.0" IssueInstant="2004-12-05T09:21:59"
  AssertionConsumerServiceIndex="0" AttributeConsumingServiceIndex="0">
  <saml:Issuer>https://serviceprovider.packt.com/SAML2</saml:Issuer>
  <samlp:NameIDPolicy AllowCreate="true" Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"/>
</samlp:AuthnRequest>
```

- **Artifact resolution protocol**: SAML protocol messages can be passed as a SAML binding, as follows:

```
<samlp:ArtifactResolve xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol" xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
  ID="_cce4ee769ed970b501d680f697989d14" Version="2.0" IssueInstant="2004-12-05T09:21:58">
  <saml:Issuer>https://identityprovider.packt.org/SAML2</saml:Issuer>
  <!-- an ArtifactResolve message SHOULD be signed -->
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">...</ds:Signature>
  <samlp:Artifact>AAQAAMh48/1oXIM+sDo7Dh2qMp1HM4IF5DaRNmDj6RdUm11wn9jJHyEgIi8=</samlp:Artifact>
</samlp:ArtifactResolve>
```

- **Single logout protocol**: A logout signal can be exchanged through a message so that all the sessions will be logged out or terminated by the SAML authority.
- **Name identifier management protocol**: The SAML service consumer will be notified if a subject or issuer is changed.
- **Name identifier mapping protocol**: This is used to map an identity of a user across different service providers with the consent of the issuing authority.

SAML 2.0 bindings

We have the following different types of bindings available in SAML 2.0, which come under the binding specifications of SAMLBind:

- **SAML SOAP binding**: The definitions of SAML request and response message exchanges are mapped to SOAP message exchanges
- **Reverse SOAP binding**: This is a mechanism to express the ability of an HTTP requestor to act as a SOAP responder to a SAML consumer
- **HTTP redirect binding**: This is suitable for short messages

- **HTTP POST binding**: This is suitable for long messages
- **HTTP artifact binding**: The Identity provider or consumer will issue an artifact
- **SAML URI binding**: A SAML URI reference will be provided to identify a specific SAML assertion

Maven Recap

Before we proceed with the development integration and coding part with Spring and SAML, let's take a quick recap of the Maven build tool. We will do some hands-on exercises that will be useful throughout this book. This is provided for readers who are at the novice level. The experienced ones can take a speedy overview!

As I mentioned earlier, Maven is a build tool with which the developers can perform builds, documentation, testing, reporting, release management, and so on.

The Maven project structure is described in an XML file called **Project Object Model** (**POM**). The POM will have details about project dependencies, plugins used, goals, build profiles, project version, and others. As a first step, we may have to decide on the `groupid` and `artifactid`. For our Spring and SAML integration, we can have these values as `com.packtpub.spring4.security` and `springsecurity`. I am avoiding the concepts of parent POM, plugins, repositories, and so on at this time as we want to focus more on Spring and SAML.

Steps involved in Mavenization are as follows:

- Make sure that you are exploring other concepts of Maven as well. It will come in handy as the sole purpose is to familiarize you with the basics.
- Install Java 8, complete the environment variable settings, and check the version by running the `java -version` command on the command prompt.
- Download `apache-maven-2.0.11-bin.*` and install as per the instructions. Complete the environmental variables settings and check the installation by running the `mvn -version` command.

- Create a POM in your project folder, as shown in the following image:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.packtpub.spring4.security</groupId>
   <artifactId>springsecurity</artifactId>
   <version>1.0</version>
</project>
```

- Run `mvn post-clean` from your project folder command prompt. You can see that maven is doing all the life cycle operations.
- Modify the POM file by adding the following entries. Run `mvn site` and Maven will start processing the entire site requirements. The result is given as a screenshot for your reference:

```
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
   <executions>
      <execution>
         <id>id.pre-site</id>
         <phase>pre-site</phase>
         <goals><goal>run</goal></goals>
         <configuration>
            <tasks><echo>pre-site phase</echo></tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.site</id>
         <phase>site</phase>
         <goals><goal>run</goal></goals>
         <configuration>
            <tasks><echo>site phase</echo></tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.post-site</id>
         <phase>post-site</phase>
         <goals><goal>run</goal></goals>
         <configuration>
            <tasks><echo>post-site phase</echo></tasks>
         </configuration>
      </execution>
      <execution>
         <id>id.site-deploy</id>
         <phase>site-deploy</phase>
         <goals><goal>run</goal></goals>
         <configuration>
            <tasks><echo>site-deploy phase</echo></tasks>
         </configuration>
      </execution>
   </executions>
</plugin>
</plugins>
```

- The life cycle behavior can be modified by mentioning goals in any phase. In the preceding screenshot, we added the `maven-antrun-plugin:run` goal to the preclean, clean, and post-clean phases. While running the build, we can see the logging echo messages for each phase of the clean life cycle. Refer to the resultant screenshot:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.packtpub.spring.security:project:jar:1.0
[INFO]    task-segment: [site]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-site}]
[INFO] Executing tasks
     [echo] pre-site phase
[INFO] Executed tasks
[INFO] [site:site {execution: default-site}]
[INFO] Generating "About" report.
[INFO] Generating "Issue Tracking" report.
[INFO] Generating "Project Team" report.
[INFO] Generating "Dependencies" report.
[INFO] Generating "Project Plugins" report.
[INFO] Generating "Continuous Integration" report.
[INFO] Generating "Source Repository" report.
[INFO] Generating "Project License" report.
[INFO] Generating "Mailing Lists" report.
[INFO] Generating "Plugin Management" report.
[INFO] Generating "Project Summary" report.
[INFO] [antrun:run {execution: id.site}]
[INFO] Executing tasks
     [echo] site phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Thu Aug 06 15:25:10 IST 2015
[INFO] Final Memory: 28M/189M
[INFO] ------------------------------------------------------------------------
```

- As a starter for Maven, we have seen some basics so far. You can explore the following points as well:
    - **Maven dependencies**: Libraries required with version numbers
    - **Maven profiles**: Which libraries are used for which environments
- Each life cycle is made of phases. Plugins are attached to phases. A phase can contain multiple plugins.
- Accessing a specific snapshot version of a dependency
- Library dependency explanation and installing custom libraries to the local repository

Gradle Recap

Gradle is a combination of Ant and Maven in terms of using the simplicity of Ant and handling multiple phases of the life cycle as Maven. As Gradle has been developed based on the Groovy **Domain Specific Language** (**DSL**), the amount of code required to be written to handle software movement through various life cycles, from compilation, analysis, testing, packaging, and deploying, will be reduced considerably. The typical Gradle build file is given here for your reference:

```
apply plugin: 'java'
apply plugin: 'checkstyle'
apply plugin: 'findbugs'
apply plugin: 'pmd'
apply plugin: 'surefire'

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    testCompile group: 'com.packtpub', name: 'packtpub-all', version: '1.23'
}
```

Some of the advantages of Gradle are as follows:

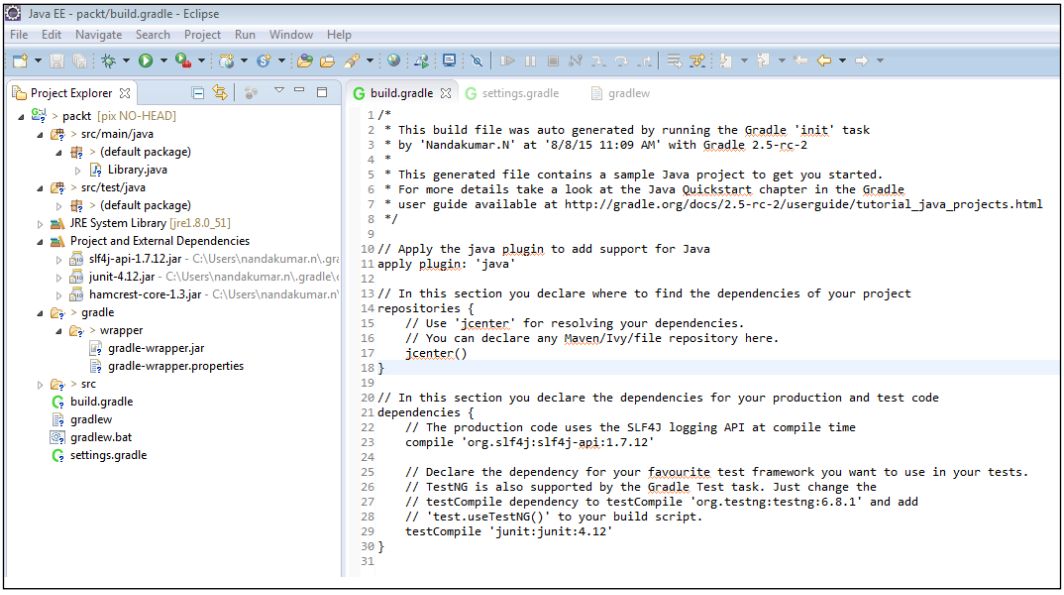
- Gradle is a programming language
- Lots of built-in tasks in the plugin code, for example, the apply plugin declaration in Gradle will do around 15 and more tasks for us
- Gradle is JVM-based, declarative modeling, expressive, and DSL-oriented
- You must be good in Java programming to handle Gradle
- The build script size is reduced and readability is increased in Gradle compared to Maven
- The time taken to clean, deploy, and identify the changed files is significantly reduced
- Gradle has lots of powerful plugins that can be adopted in projects very easily
- Mature libraries such as Spring, Hibernate, Grails, Groovy, and others already use Gradle to power their builds

Let's see the quick steps involved in Gradle building.

Setting up Gradle with Eclipse

As I mentioned in the *Maven Recap* section, it is important for you to understand some of the basics of Gradle. Follow these steps to integrate the Gradle Buildship with the Eclipse IDE:

1. Download `gradle-2.5-bin.zip` of 42.6 MB size and unzip to your folder.
2. Set the class path settings.
3. Verify the installation by executing the `gradle -v` and `gradle tasks -q` commands.
4. Download `eclipse-jee-mars-R-win32-x86_64.zip` of 269 MB size.
5. Install Gradle Buildship from the Eclipse marketplace, and then create a Gradle project named `Packt-Gradle`.
6. The project structure and Gradle build file will look as follows:



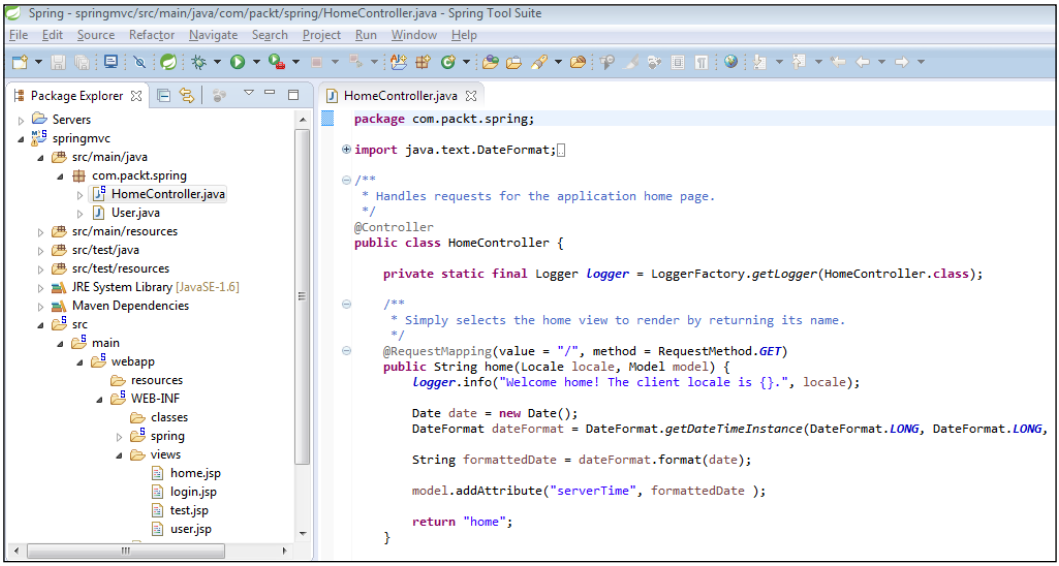
Downloading the example code

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

The Spring Tool Suite

After a quick introduction to the **Spring Tool Suite** (**STS**), we will develop a small application with 3-4 screens for a basic understanding. This sample can be carried out or reused throughout the book. STS is an Eclipse-based IDE that provides you with robust project templates for various Spring projects such as batch, integration, persistence, and so on. Download the **STS 3.7** release and open the IDE.

Create a Spring project, choose **Templates** as the **Spring MVC project**, specify a top-level package name such as `com.packt.spring.security`, and click on **finish**. The STS will have the following project structure:



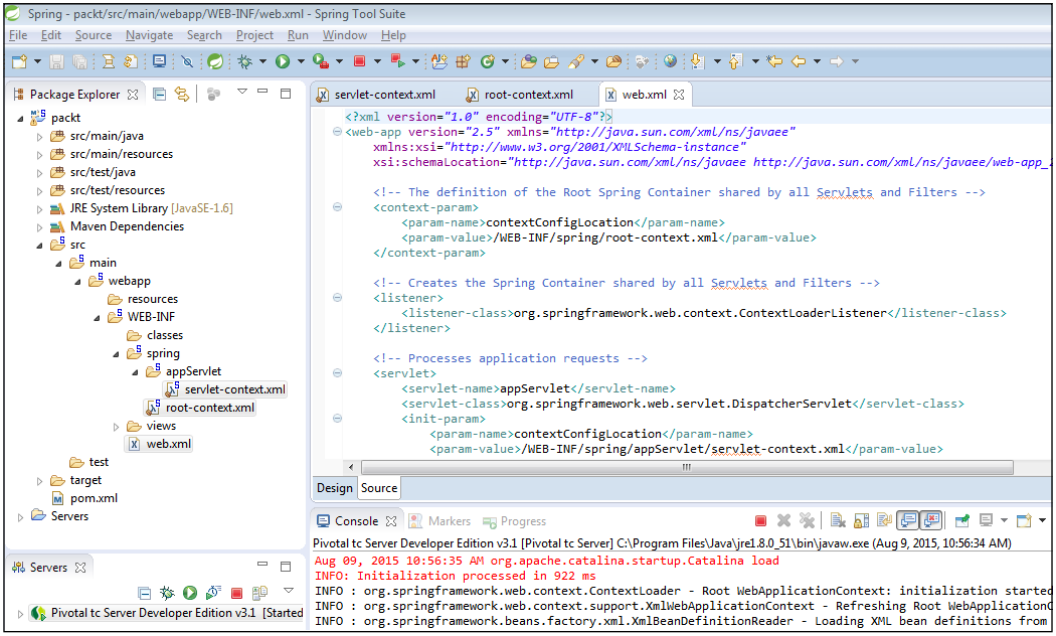
You can see the `springmvc` project tree and default `HomeController` Java class. In the `views` folder, you can see some default JSP files, and at the bottom, you can see the **Pivotal tc Server Developer Edition**, which comes built-in with STS 3.7.

Let's see the configuration mappings and how to run this application:

- `root-context.xml`: This file is empty by default. It is the configuration for root Spring containers, which are shared by all the servlets and filters.
- `servlet-context.xml`: This file is loaded by the Spring's `DispatcherServlet` that receives all the requests coming in the application and dispatches the processing for controllers, based on the configuration specified in the `servlet-context.xml` file.

- `web.xml`: This file contains declarations for Spring's `ContextLoaderListener` and `DispatcherServlet` along with the Spring configuration files, `root-context.xml` and `servlet-context.xml`. It also has the mapping for `DispatcherServlet`, which handles all the requests.

You can right-click on the `packt` root directory and run the application. Set the server settings accordingly:



Improving the samples

Let's modify the controller and add some more JSP files. Follow the following steps:

1. Add the following code snippet to the `controller` class:

```
@RequestMapping(value = "/login", method = RequestMethod.GET)
public String test(Model model) {
    String message = "Greetings, Spring MVC!";
    model.addAttribute("Welcome", message);
    return "login";
}
```

2. Create one more JSP file called `login.jsp`, and add the following content:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Login Page</title>
</head>
<body>
<form action="home" method="post">
<input type="text" name="userName"><br>
<input type="text" name="passWord"><br>
<input type="submit" value="Login">
</form>
</body>
</html>
```

3. In `home.jsp`, add the `${userName}` code and check for the values that are passed.

SAML open source implementations

There are several open source implementations available for SAML 2.0. Most of the implementations need a registered account. Let's see them one by one:

- **OX**: This uses the Shibboleth IDP but adds a GUI to make the configuration easier. It has been developed using the J2EE stack of software to enable domain authentication and authorization. It is available as an open source or managed service and is sponsored by Gluu at `http://www.gluu.org`.
- **Enterprise Sign On Engine** (**ESOE**): This is a pure Java implementation of SAML V2.0. Check the website for more details, `http://esoeproject.qut.edu.au/`.
- **OneLogin SAML Toolkits, SAML 2.0 SP**: This is available as Java, C#, Python, Ruby, and PHP implementations.
- **OpenSSO**: This is a Java implementation from Sun Microsystems and is currently in use at the SSOCircle. Refer to `http://opensso.org/`.
- **OpenSAML**: This has C++ and Java toolkits for SAML V1.1 and V2.0. Implementation of SAML assertions, protocols, and bindings (no profiles) are available at `http://www.opensaml.org/`.
- **Shibboleth**: This includes the Identity provider (Java) and service provider (C++ Apache module), and is a basic implementation built on top of OpenSAML. Refer to `http://shibboleth.internet2.edu/`.

The SAML 2.0 login flow

SAML 2.0 specifies a web browser SSO profile that involves exchanging information among an **Identity provider** (**IDP**), **service provider** (**SP**), and principal (user) on a web browser. The Identity provider can be any SSO service offering SAML authentication services (for example, SSOCircle). The service provider is always a `ServiceNow` instance. The message flow begins with a request for a secured resource at the service provider. The principal requests a target resource at the service provider, `https://instance.service-now.com/`.

The `ServiceNow` instance checks the request to see if the `SAMLRequest` and `RelayState` URL parameters are present. It constructs `AuthnRequest` to be sent to the IDP using the `SAMLRequest` value. The instance also constructs and sends a `RelayState` URL parameter value.

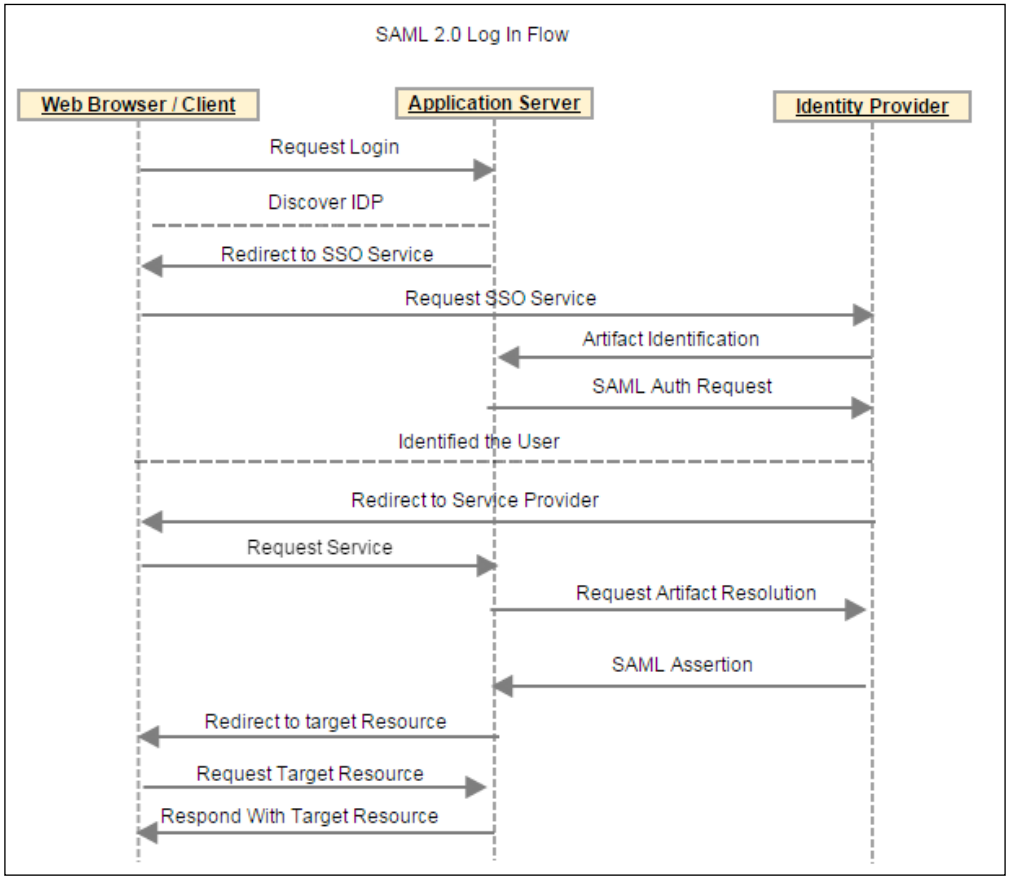
The `RelayState` token is an opaque reference to the state information maintained at the service provider. The value of the `SAMLRequest` parameter is the deflated and base64 encoded value of the `<samlp:AuthnRequest>` element:

```
<samlp:AuthnRequest
   xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
   xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
   ID="identifier_1"
   Version="2.0"
   IssueInstant="2004-12-05T09:21:59Z"
   AssertionConsumerServiceIndex="0">
   <saml:Issuer>https://sp.example.com/SAML2</saml:Issuer>
   <samlp:NameIDPolicy AllowCreate="true" Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"/>
</samlp:AuthnRequest>
```

The integration then URL-encodes the `<samlp:AuthnRequest>` element and sends it as the `SAMLRequest` URL parameter.

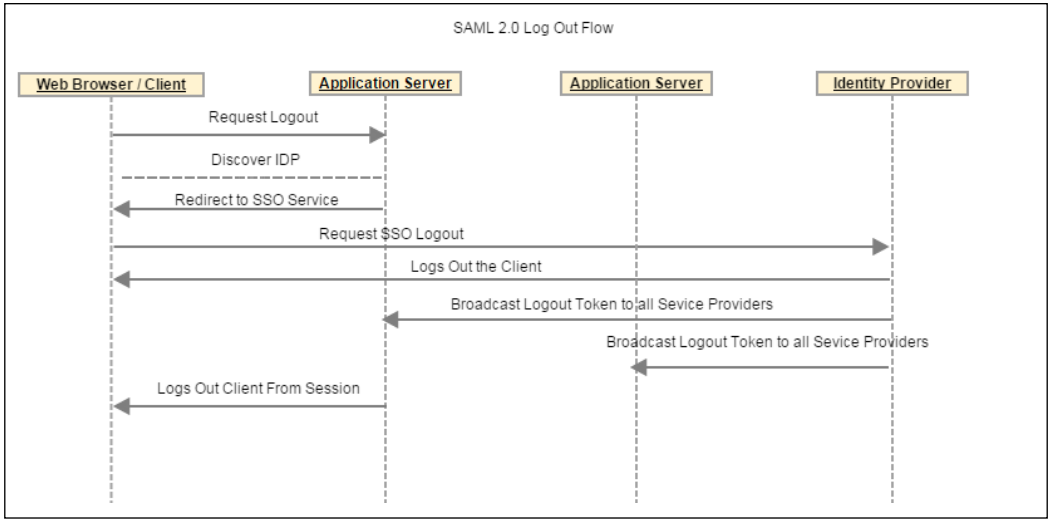
The SSO service processes the `<samlp:AuthnRequest>` element by URL-decoding, base64-decoding, and inflating the request, in that order. It then performs a security check. If the user does not have a valid security context, the IDP identifies the user by prompting for login credentials. If the user is already logged in, the IDP simply responds with the `SAMLResponse<tt>` and `<tt>RelayState` URL parameters.

The login script also extracts the session ID from the `//AuthnStatement/@SessionIndex` element and stores it for `LogoutRequest`:



The SAML 2.0 logout flow

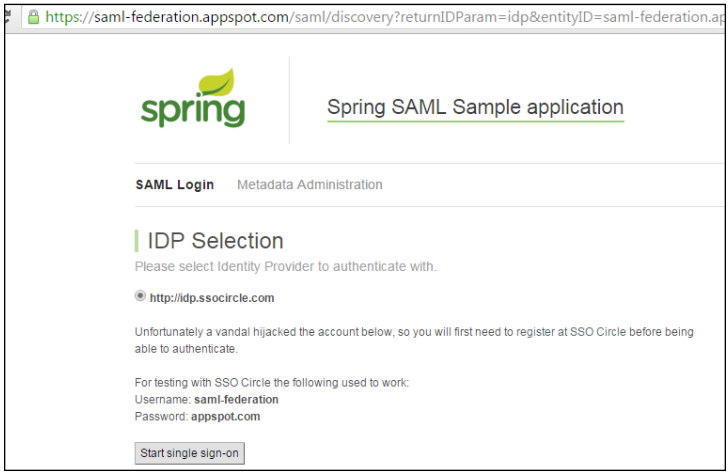
During logout, `ServiceNow` issues the SAML 2.0 LogoutRequest service call to the IDP. This service logs the user out and then redirects them to the specified logout URL. The user clicks on the **Logout** button and the instance executes the logout script. The logout script constructs SAML 2.0 LogoutRequest and posts it to the preconfigured SingleLogoutRequest SAML 2.0 service at the IDP. The IDP deflates the request and then base64-encodes it. An example `LogoutRequest` looks as follows:




```
<saml2p:LogoutRequest xmlns:saml2p="urn:oasis:names:tc:SAML:2.0:protocol"
  ID="21B78E9C6C8ECF16F01E4A0F15AB2D46" IssueInstant="2010-04-28T21:36:11.230Z" Version="2.0">
    <saml2:Issuer xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion">https://dloomac.service-now.com</saml2:Issuer>
    <saml2:NameID xmlns:saml2="urn:oasis:names:tc:SAML:2.0:assertion"
          Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress"
          NameQualifier="http://idp.ssocircle.com"
          SPNameQualifier="https://dloomac.service-now.com/navpage.do">david.loo@service-now.com</saml2:NameID>
    <saml2p:SessionIndex>s211b2f811485b2a1d2cc4db2b271933c286771104
    </saml2p:SessionIndex>
</saml2p:LogoutRequest>
```

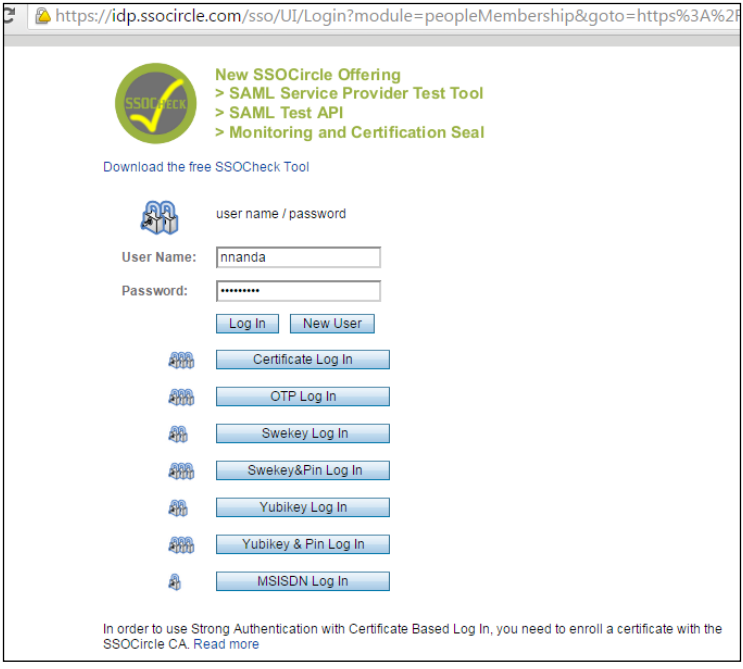
The user logs out of the IDP. The IDP redirects it to `ServiceNow`, which in turn redirects to the IDP as the user is not logged in.

IDP selection and testing

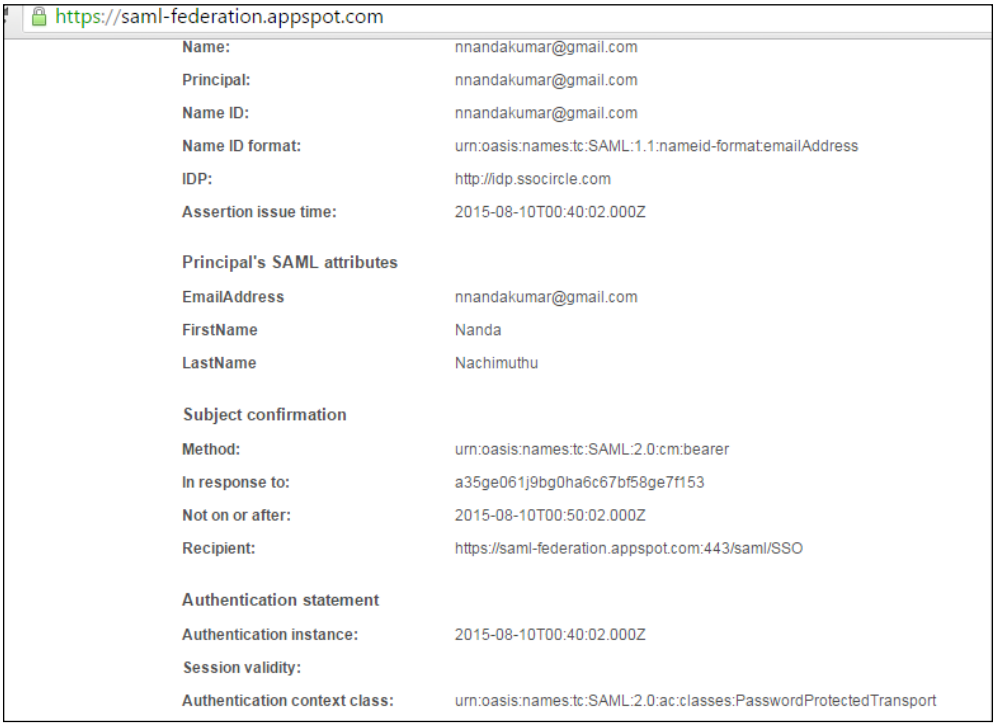


SSOCircle is a popular web-based SSO Identity provider.

You can go through the sample application, `saml-federation.appspot.com`, given by SSOCircle, where you may have to register to check the SSO behavior. The following screenshot shows you the steps involved:



The output of the preceding steps is as follows:



You may have to check out the other IDPs as well, such as Shiboleth and OpenSSO.

The Spring Security SAML dependency

Here is the dependency that Spring Security provides you with in order to use SAML. The Spring Security SAML extension makes both the existing new applications behave like service providers, thus making the application achieve single sign-on and single logout profiles as per the SAML protocol:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.security.extensions</groupId>
        <artifactId>spring-security-saml2-core</artifactId>
        <version>1.0.0.RELEASE</version>
    </dependency>
</dependencies>
```

Spring Security with SAML classes

In this section, we will look at the Spring Security SAML package. The classes in this package extend the Spring Security core classes that are responsible for SAML authentication, authorization, and logout:

- `SAMLAuthenticationProvider`: This is capable of verifying the validity of a `SAMLAuthenticationToken`, and in case the token is valid, creates an authenticated `UsernamePasswordAuthenticationToken`.
- `SAMLAuthenticationToken`: This is used to pass the `SAMLContext` object through to the SAML authentication provider.
- `SAMLBootstrap`: This is the initialization for the SAML library.
- `SAMLConstants`: These are the constant values for the SAML module.
- `SAMLCredential`: An object is a storage for entities parsed from the SAML 2.0 response during its authentication.
- `SAMLDiscovery`: A filter implements the Identity provider Discovery Service Protocol and Profile, as defined in `http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-idp-discovery.pdf`.
- `SAMLEntryPoint`: A class initializes the SAML WebSSOProfile, IDP discovery, or ECP profile from the SP side.
- `SAMLLogoutFilter`: This is the logout filter that leverages the SAML 2.0 single logout profile.
- `SAMLLogoutProcessingFilter`: This filter processes the arriving SAML single logout messages by delegating to `LogoutProfile`.
- `SAMLProcessingFilter`: This filter processes the arriving SAML messages by delegating to WebSSOProfile.
- `SAMLStatusException`: This SAML exception contains the status code that should be returned to the caller as part of the status message.
- `SAMLUserDetailsService`: The `SAMLUserDetailsService` interface is similar to `UserDetailsService` with a difference that SAML data is used in order to obtain information about the user. Implementers of the interface are supposed to locate the user in a arbitrary data store based on the information present in `SAMLCredential`, and return such a date in the form of an application-specific `UserDetails` object:
    - `Object loadUserBySAML (SAMLCredential credential)`: This method is supposed to identify the local account of a user referenced by the data in the SAML assertion and return the `UserDetails` object describing the user.

Spring Security SAML internals

Let's look at what specific configuration is done to support SAML.

SAML filters are defined in `springContext.xml`, which diverts the URLs of the application through SAML filters and URLs:

```
<!-- Secured pages -->
    <security:http entry-point-ref="samlEntryPoint">
        <security:intercept-url pattern="/**" access="IS_AUTHENTICATED_FULLY"/>
        <security:custom-filter before="FIRST" ref="metadataGeneratorFilter"/>
        <security:custom-filter after="BASIC_AUTH_FILTER" ref="samlFilter"/>
    </security:http>

    <bean id="samlFilter" class="org.springframework.security.web.FilterChainProxy">
        <security:filter-chain-map request-matcher="ant">
            <security:filter-chain pattern="/saml/login/**" filters="samlEntryPoint"/>
            <security:filter-chain pattern="/saml/logout/**" filters="samlLogoutFilter"/>
            <security:filter-chain pattern="/saml/metadata/**" filters="metadataDisplayFilter"/>
            <security:filter-chain pattern="/saml/SSO/**" filters="samlWebSSOProcessingFilter"/>
            <security:filter-chain pattern="/saml/SSOHoK/**" filters="samlWebSSOHoKProcessingFilter"/>
            <security:filter-chain pattern="/saml/SingleLogout/**" filters="samlLogoutProcessingFilter"/>
            <security:filter-chain pattern="/saml/discovery/**" filters="samlIDPDiscovery"/>
        </security:filter-chain-map>
    </bean>
```

The context file defines a SAML logger that will log the SAML messages. The most important thing that we do in the Spring Security configuration is to configure an authentication manager. The manager is usually configured explicitly as a database query with the database/data source information, LDAP information, or just a bean class that extends the UserDetails Service class.

For SAML, the authentication manager is configured as follows:

```
<!-- Register authentication manager with SAML provider -->
    <security:authentication-manager alias="authenticationManager">
        <security:authentication-provider ref="samlAuthenticationProvider"/>
    </security:authentication-manager>
```

You can give the `SAMLAuthenticationProvider` reference in your application, as shown in the following image:

```
!-- SAML Authentication Provider responsible for validating of received SAML messages -->
    <bean id="samlAuthenticationProvider" class="org.springframework.security.saml.SAMLAuthenticationProvider">
        <property name="userDetails" ref="samlUserDetailsService" />
    </bean>

    <!-- Custom user details service to attach app specific roles to federated identities -->
    <bean id="samlUserDetailsService" class="org.packt.springmvc.saml.SimpleSAMLUserDetailsService">
        <property name="roles">
            <list>
                <value>ROLE_VIEWER</value>
            </list>
        </property>
    </bean>
```

As discussed, the configuration of SAML in the context must mainly consist of Identity provider information's circle as an open Identity provider. The IDP provider information is configured as follows:

```
<!-- IDP Metadata configuration - paths to metadata of IDPs in circle of trust is here -->
    <!-- Do no forget to call iniitalize method on providers -->
    <bean id="metadata" class="org.springframework.security.saml.metadata.CachingMetadataManager">
        <constructor-arg>
            <list>
                <bean class="org.opensaml.saml2.metadata.provider.HTTPMetadataProvider">
                    <constructor-arg>
                        <value type="java.lang.String">http://idp.ssocircle.com/idp-meta.xml</value>
                    </constructor-arg>
                    <constructor-arg>
                        <value type="int">5000</value>
                    </constructor-arg>
                    <property name="parserPool" ref="parserPool"/>
                </bean>
            </list>
        </constructor-arg>
    </bean>
```

Let's now look at the SAML UserDetails Service class. This class also has a `loadUserBySAML()` method that needs to be implemented by the class that implements `SAMLUserDetailsService`. The implementing class tells the framework how to perform the authentication:

```
public class SimpleSAMLUserDetailsService implements SAMLUserDetailsService {
    public static final String DUMMY_PASSWORD = "DUMMY_PASSWORD";
    private List<String> roles;

    public void setRoles(List<String> roles) {
        this.roles = roles;
    }
    public Object loadUserBySAML(SAMLCredential credential) throws UsernameNotFoundException {
        String username = credential.getNameID().getValue();
        Collection<GrantedAuthority> gas = new ArrayList<GrantedAuthority>();
        for (String role : roles) {
            gas.add(new SimpleGrantedAuthority(role));
        }
        return new User(username, DUMMY_PASSWORD, gas);
    }
}
```

Spring Security with SAML logout

So far, we have seen how we can run the sample SAML application. Now, we will look at how Spring Security supports SAML logout. First, we will see how SAML logout works and then we will see the class that is supported by Spring Security for SAML logout.

SAML supports single sign-on, so we can also say that it supports single logout as well.

LogoutRequest issued by SP to IDP

The IDP determines authenticated SPs for a given user session. If there are no SPs, other than the SP who sends logout request, the profile proceeds with issuing a LogoutResponse to SP who sends logout request. Otherwise, LogoutRequest issued by the IDP to the SP and the SP-issued LogoutResponse to the IDP are repeated for each SP. The IDP issues LogoutResponse to the SP who sends the logout request.

Let's see what is in these request and response messages:

- **LogoutRequest** is extended from `RequestAbstractType`. There are some attributes that must be in the `RequestAbstractType` element.
- **LogoutResponse** is extended from `StatusResponseType`. There are some attributes that must be in the `StatusResponseType` element, that is, `ID`, `version`, and `IssueInstant`, which is the same as in `RequestAbstractType`. There is an element called the `status` element that is required. The `status` element contains the status code corresponding to the request. These attributes are explained as follows:
    - `ID`: This is an identifier for the request. This must be unique; basically, a random number.
    - `Version`: This indicates the SAML version.
    - `IssueInstant`: This is the time instant of the issue of the request. The time value is encoded in UTC.

  Apart from this, one of the following is a required attribute for a LogoutRequest request:

    - `BaseID`, `NameID`, or `EncryptedID`: This indicates the principal (user identifier). Basically, this is a name that is known to both the IDP and SP.
    - `NotOnOrAfter`: This is the time when the request expires in UTC.
    - `Reason`: This is the reason for the logout in the form of a URI reference.

      There are two standard reasons:

    - `urn:oasis:names:tc:SAML:2.0:logout:user`: The user terminates the session and initiates the logout
    - `urn:oasis:names:tc:SAML:2.0:logout:admin`: The administrator terminates the session and initiates the logout

- `SessionIndex`: This is the session identifier that is used to identify the user session with both the IDP and SP for a given user.

You can see that a simple SAML response consists of a URL that it needs in order to redirect the applications on logout:

```
<samlp:LogoutResponse
    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    ID="_cbb63e9741259e3f1c98a1ae38ac5ac25889720b32" Version="2.0"
    IssueInstant="2008-06-03T12:59:57Z"
    Destination="https://myapplication.feide.no/simplesaml/saml2/sp/SingleLogoutService.jsp"
    InResponseTo="_72424ea37e28763e351189529639b9c2b150ff37e5">
    <saml:Issuer>https://openidp.feide.no</saml:Issuer>
    <samlp:Status>
        <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"> </samlp:StatusCode>
        <samlp:StatusMessage>Successfully logged out from service https://openidp.feide.no</samlp:StatusMessage>
    </samlp:Status>
</samlp:LogoutResponse>
```

The `org.springframework.security.saml` package contains all the classes that support SAML.

The filters intercept the request, terminate the session with respect to the users, and send out the logout response, which looks like the preceding response to the IDP and SP providers, so that they get intimated about the logout request that has been initiated.

Let's look at the Spring Security SAML filters that are available to process the single logout feature.

We have two classes that are filters, as follows:

```
public class SAMLLogoutFilter extends org.springframework.security.web.authentication.logout.LogoutFilter
```

Logout filter leveraging SAML 2.0 Single Logout profile. On the invocation of the filter's URL, it is determined whether global (termination of all the participating sessions) or local (termination of only the session running in Spring Security) logout is requested based on the request attribute.

In case global logout is in question, LogoutRequest is sent to the IDP.

The default constructors of the filter show complete information about the class. The constructor signature implies that the Spring framework's logout handlers are called to log out and successHandler is also called to build the SAML logout response as follows:

```
SAMLLogoutFilter(org.springframework.security.web.authentication.logout.LogoutSuccessHandler logoutSuccessHandler,
org.springframework.security.web.authentication.logout.LogoutHandler[] localHandler,
org.springframework.security.web.authentication.logout.LogoutHandler[] globalHandlers)
SAMLLogoutFilter(String successUrl, org.springframework.security.web.authentication.logout.LogoutHandler[] localHandler,
org.springframework.security.web.authentication.logout.LogoutHandler[] globalHandlers)
```

The filter processes the arriving SAML single logout messages by delegating to LogoutProfile:

```
public class SAMLLogoutProcessingFilter
extends org.springframework.security.web.authentication.logout.LogoutFilter
```

This is the class that actually does the logout processing; this also has the logout handlers. Let's look at the constructor signature of the class:

```
SAMLLogoutProcessingFilter(org.springframework.security.web.authentication.logout.LogoutSuccessHandler logoutSuccessHandler,
org.springframework.security.web.authentication.logout.LogoutHandler... handlers)
Constructor uses custom implementation for determining URL to redirect after successful logout.
SAMLLogoutProcessingFilter(String logoutSuccessUrl, org.springframework.security.web.authentication.logout.LogoutHandler... handlers)
Constructor defines URL to redirect to after successful logout and handlers.
```

Summary

In this chapter, you have seen the basics of Spring 4.0 Web MVC creation and build tools such as Maven and Gradle as a recap and practice session. You have seen the usage of the Spring tool suite where we created a web-based MVC project and executed and modified the programs to implement the login and logout features.

Then, we explored the open source implementations of SAML 2.0 available as Identity providers and how to register with the web-based IDP SSOCircle. The other IDPs such as Shiboleth and OpenSSO were also introduced for further experiments with SAML 2.0.

Finally, you learned how to use the Spring4.0 SAML extensions to implement single sign-on and sign off by connecting to the SSOCircle web-based authentication mechanism. At this point, feel free to explore the other SSO providers by registering with them, or you can install some of them in your local system as well.

You can refer to my GitHub account for many working SSO programs. The link is `https://github.com/nnanda` and you can see a separate directory named **Sample Codes for Spring Security Essentials**, which will have all the working projects that are specified in this book.

3
Spring Security with LDAP

Spring 4.0 comes with an LDAP Template package that makes the integration between the Spring applications and LDAP implementations such as **OpenLDAP**, **ApacheDS**, and so on.

In this chapter, we will cover following topics:

- Various LDAP implementations available
- Apache Directory Server and Studio for **Spring Tools Suite** (**STS**): Overview, installation, and usage
- Basic **Java Naming and Directory Interface** (**JNDI**) LDAP programs
- Spring LDAP Template: **spring-ldap-core** 2.0.3 overview
- Spring LDAP Template: Directory structure operations, **LDAP Data Interchange Format** (**LDIF**) handling, and queries

A quick overview of LDAP

Let's us have a quick overview of what LDAP is all about. **Lightweight Directory Access Protocol** (**LDAP**) came up during the 1980s for easier accessibility and maintainability of the distributed directory services, which are used over the Internet Protocol network. The frequent usage of LDAP is to implement single sign-on across various applications in different networks. LDAP has standard bodies such as network protocols, directory structure and services provided by the LDAP server. It originated in the University of Michigan and was endorsed by more than 40 companies.

Some industry standard usage of LDAP is as follows:

- User authentication
- User/system groups
- Address book
- Organization representation
- User resource management
- E-mail address look-ups
- Application configuration store
- **Private branch exchange** (**PBX**) configuration store

LDAP is a software network protocol that will provide options for Internet applications to identify organizations, usernames, passwords, and other resources by searching based on the key and value pair. **X.500** is a standard for directory services that holds a small amount of data that is to be used in a network and this standard is the basic for **Directory Access Protocol** (**DAP**). LDAP is formed based on various entries, it is nothing but a collection of attributes that is identified by a globally-unique **Distinguished Name** (**DN**).The LDAP directory structure follows a simple tree structure that contains information about organizations, usernames, password, and so on. Directories will contain a descriptive, attribute-based information and support sophisticated filtering capabilities.

Directories generally do not support complicated transaction or rollback schemes that are found in the database management systems designed to handle high-volume complex updates. Directory updates are typically simple all-or-nothing changes, if they are allowed at all. They are generally tuned to give quick response to the high-volume look-up or search operations. The directory can be distributed among many servers and applications. Each server can have their own copy of the directory information, which can be refreshed periodically. This will be taken care by LDAP **Directory System Agent** (**DSA**), to which all the application servers need to be subscribed.

The LDAP server will be responsible to maintain, manage, and provide the directory information to the participating applications. The client starts a session with the server and will call DSA by default on the TCP and UDP 389 port or on the 636 port for **LDAP over SSL** (**LDAPS**). Global catalog is available by default on the 3268 and 3269 ports for LDAPS. Directory Server will facilitate adding, modifying, and deleting the directory structure information. The LDAP servers are designed to store general purpose data so that it is not restricting itself to hold only a particular type of data. We can clearly define the type of data and structure that is to be stored and maintained for each type of industry usage; therefore, different types of entities can be stored in the directory structure as the LDAP's general architecture provides the capabilities that are needed to manage large amounts of diverse directory entries.

LDAP implementations

The following software programs communicate with built-in directory services that are developed on LDAP. The LDAP implementations are designed on the following framework components:

- **Pluggable authentication module** (**PAM**): This allows the integration of various authentication ways
- **Name Service Switch** (**NSS**): This converts the information available in the text file to a C library
- **Name service caching daemon** (**nscd**): This provides a cache for the name services and makes them available as a lookup
- **Lightweight Directory Access Protocol** (**LDAP**): This provides the clients with information about user accounts and groups

The efficiency of LDAP servers usually depends on their compatibility with clients, installation overhead, scalability (multi-master replication), and integration with other framework components such as console, password manager, and so on. We may have to consider other factors such as ranging attributes, password updates and object class mapping. Most LDAP directories use the `inetOrgPerson` and `groupOfUniqueNames` object classes for users and groups. Let's see some LDAP implementations such as ApacheDS, OpenLDAP, and so on.

ApacheDS

Apache Directory Server (**ApacheDS**) is an open-source implementation of the X.500 directory server. It's an open source project of Apache Software Foundation that comes with a plug-in for STS, where we can configure various directory structures and run the ApacheDS in the STS. ApacheDS is written in Java, which is compliant to the LDAP 3.0 standards and certified by The Open Group. ApacheDS also supports Kerberos 5 and the Kerberos Change Password Protocol. Using ApacheDS, we can create triggers, stored procedures, queues, and views for the directory values and the usage of directory data can be made simpler.

OpenLDAP 2.4.42

OpenLDAP 2.4.42 is another LDAP implementation that is commonly available with all Linux bundles. This is a command line tool and **phpLDAPAdmin** will be used as a frontend for this server. OpenLDAP comes with **slapd** (the standalone command line LDAP server). It listens for the directory connections on any number of ports (default: 389), responding to the directory services operations that it receives over these connections), libraries that have the LDAP implementation, and other utilities and sample clients. With OpenLDAP, you can secure the communication and define privileges for your users. This suite also provides role-based identity access management Java SDK called **Fortress**, specific Java libraries for LDAP access called JLDAP and JDBC-LDAP Bridge Driver.

OpenDJ

Like OpenLDAP, this server also provides good attention and commercial support. This is developed on top of Sun Microsystems Sun Directory Server. An LDAP SDK, directory server, and client tools are developed by the OpenDJ community. This implementation comes with various APIs for synchronous and asynchronous communications. As it is providing an asynchronous LDAP access, we have the options of choosing from various access methods such as REST, **System for Cross-domain Identity Management** (**SCIM**), LDAP, and web services. OpenDJ is developed using pure Java architecture so that it supports the most demanding **service-level agreement** (**SLA**) environments with high throughput and low response times.

When it comes to security, OpenDS secures all data including passwords through a wide variety of encryption mechanisms. Also, it supports multiple levels of authentication and authorization policies including SSL, **STARTTLS**, and certificate.

The 389 Directory Server (previously Fedora Directory Server)

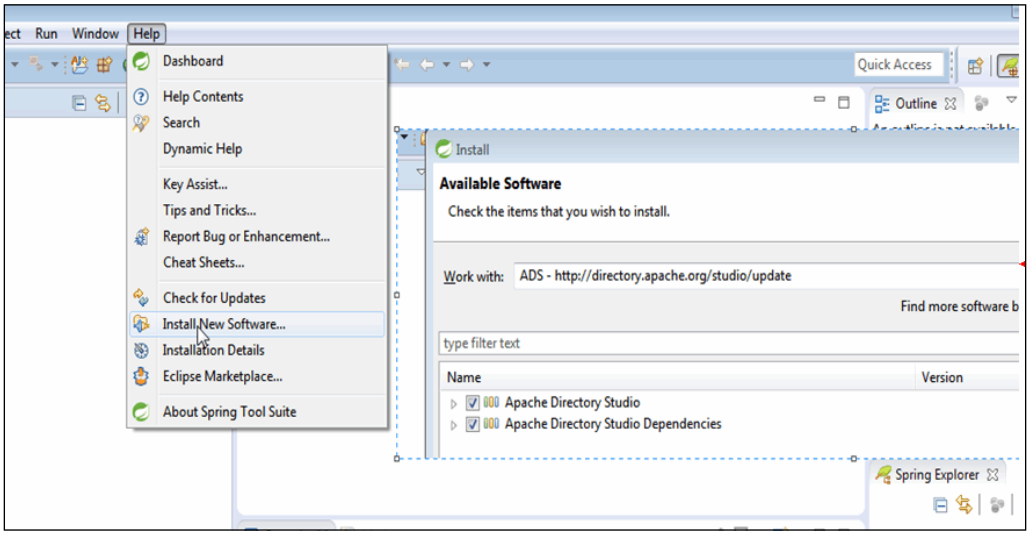
This is developed by Red Hat, as part of Red Hat's community-supported Fedora Project. The name 389 is derived from the port number of LDAP. The key features of this server include multi-master replication, secure authentication and transport, online, zero downtime, LDAP-based update of schema, configuration, management, and in-tree **Access Control Information** (**ACI**) graphical console for all facets of user, group, and server management. The 389 Directory Server offers more features and the Admin console makes it easier to manage the directory server compared to OpenLDAP. **Fedora Directory Server** (**FDS**) provides a flexible mechanism for grouping and sharing attributes among entries in a dynamic fashion.

FDS has secure communications across networks with 168-bit encryption ciphers. The major components of FDS consist of an LDAP server, **Directory Server Console**, **Simple Network Management Protocol** (**SNMP**) agent, and online backup and restore. We will choose ApacheDS as it is very simple to use and comes with a plug-in support for STS. **Apache Directory Studio** is a powerful frontend to create organizational user and password tree structure. Let's understand how to work with STS and Apache DS.

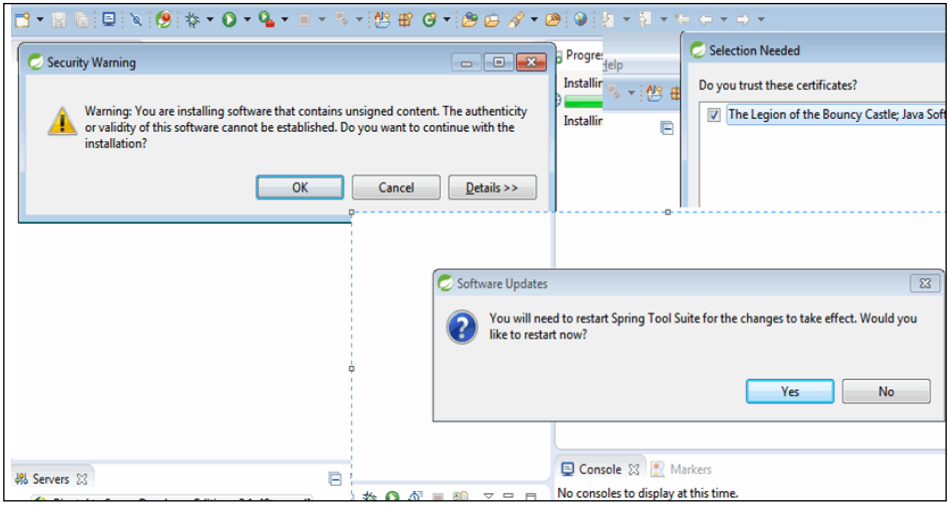
Apache Directory Server and Studio installation

Let's see how to install and run STS and update the site with the ApacheDS:

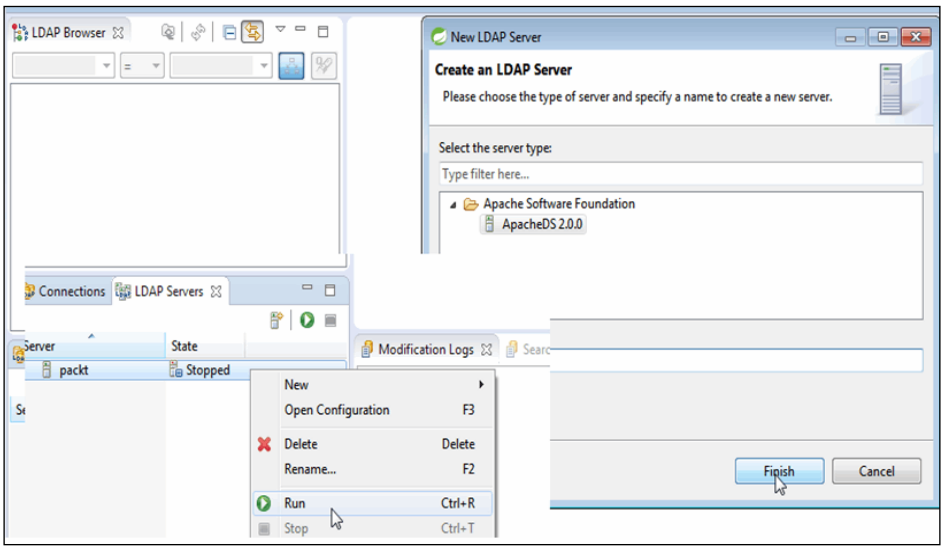
1. Download and install `spring-tool-suite-3.7.0.RELEASE-e4.5-win32-x86_64.zip` 411 MB. Open the **Install New Software** dialog box.
2. Enter `http://directory.apache.org/studio/update` in the work field and check the `Apache Directory Studio` and `Apache Directory Studio Dependencies` options and proceed till completion, as shown in the following screenshot. This operation may take several minutes, depending on your system and Internet configurations:



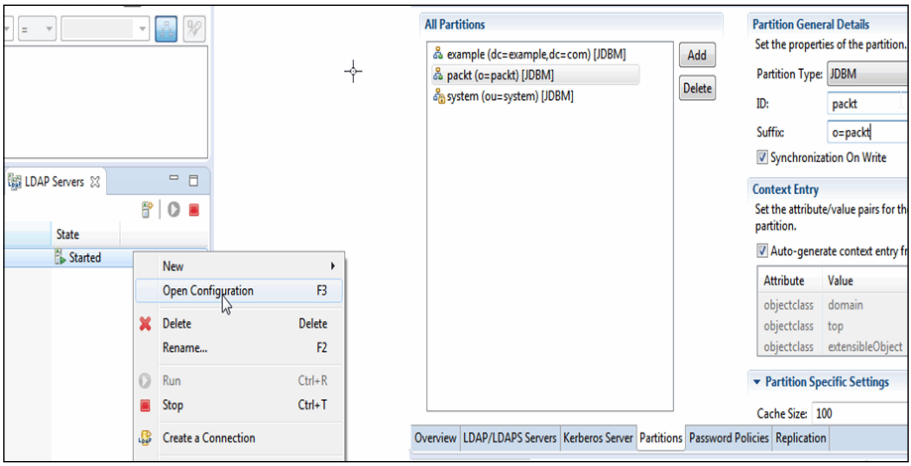
3. After some time, you will see the **Security Warning** dialog. Click on **OK** to proceed. Then, you will see a **Selection Needed** dialog, check the trust certificate checkbox and proceed. On completion, you will be asked to restart the STS. Click on **Yes** as follows:



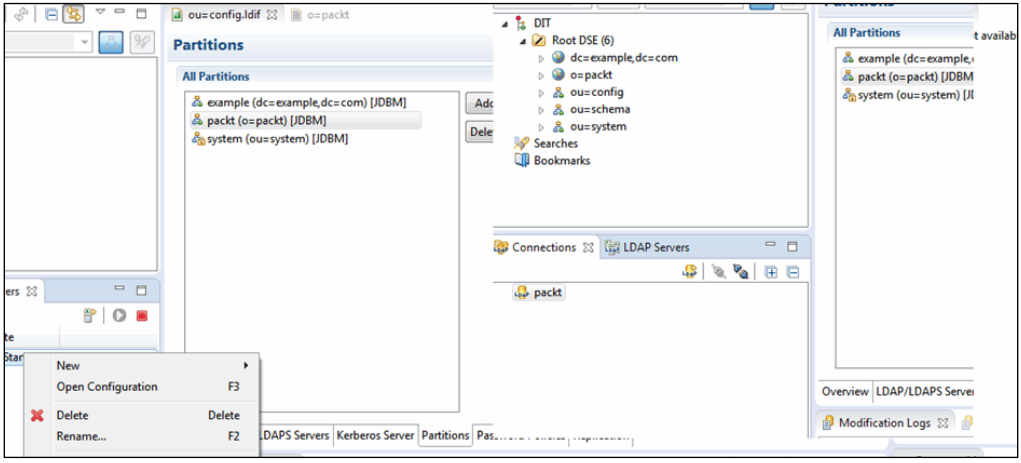
4. Now, navigate to **Windows** | **Open Perspective** | **Other** and select **LDAP** perspective. In the server panel, right-click and select **New** | **Server**.
5. Give a name of your choice and you can see the server is created. Now, right-click and **Run**. You can see the server status changed to **Run**, as follows:



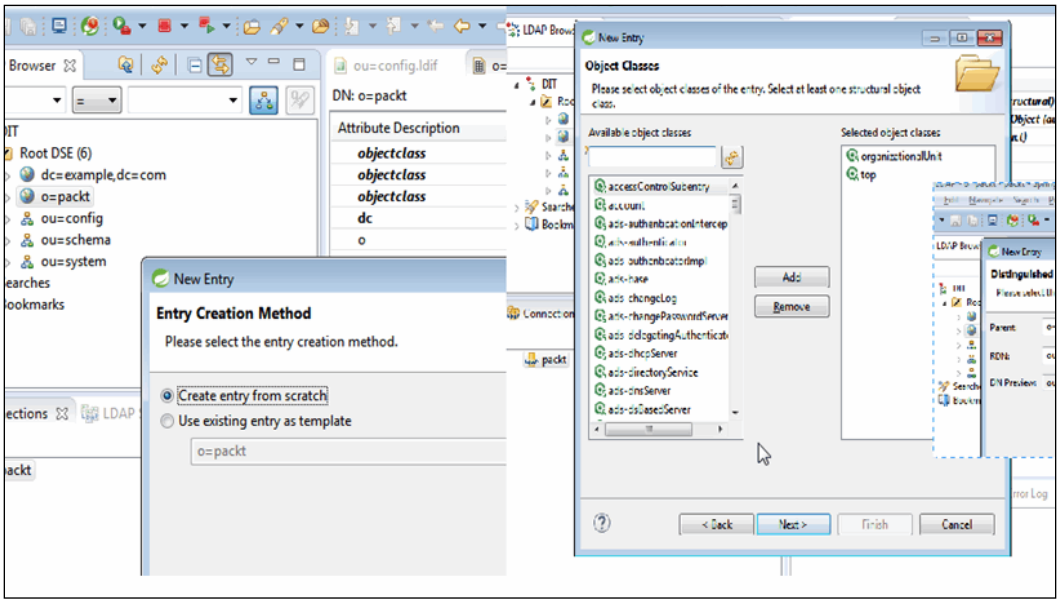
6. Right-click on the server and open the configurations window.
7. Go to **Partitions** and click on **Add** to create a new partition.
8. Provide **ID** and **Suffix** as per your preference and use *Ctrl + S* every time you want to save the modifications. Restart the server once you have created a partition:



9. Right-click on the server and select **Create a Connection**. You will see that the dialog box with a connection named `packt` is created.
10. Double-click on the `packt` connection to view the `Root DSE` details. You can see the connection browser with entries for newly created partition called `packt`. As we are already given a suffix name, `o=packt`, you can see the `o=packt` entry under `Root DSE`, as follows:

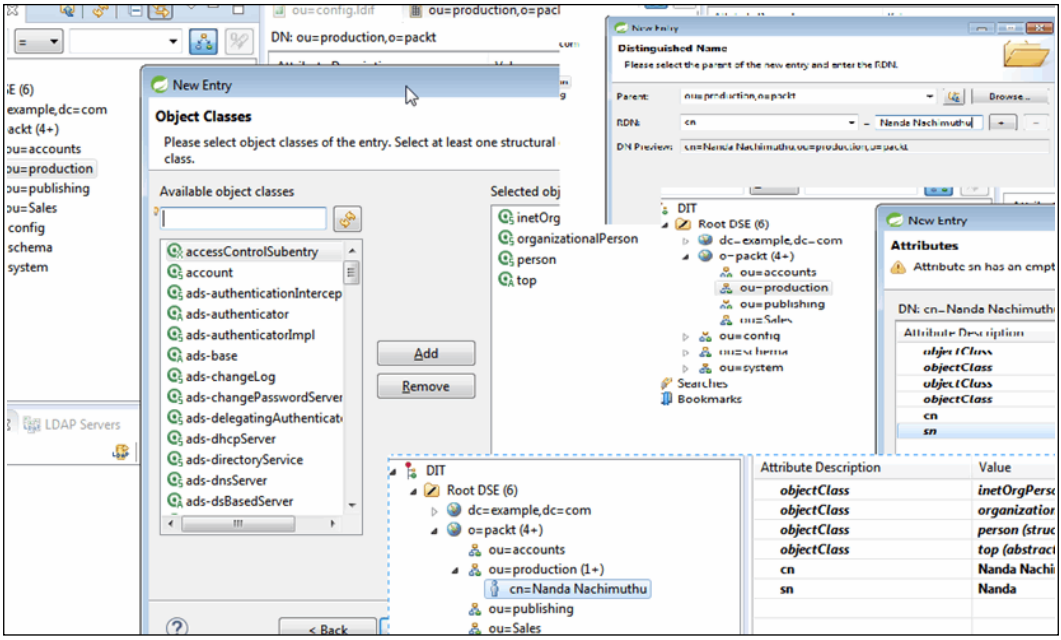


11. Right-click on the `o=packt` node and select the **New Entry** dialog.
12. Leave the `Create entry from scratch` selection as it is and click on **Next**.
13. Select the **organizationalUnit** object class and click on **Next**.
14. Give values for **RDN** as `ou` and value as `production` and click on **Next** and **Finish**. You can see an organization unit called production created under organization packt:

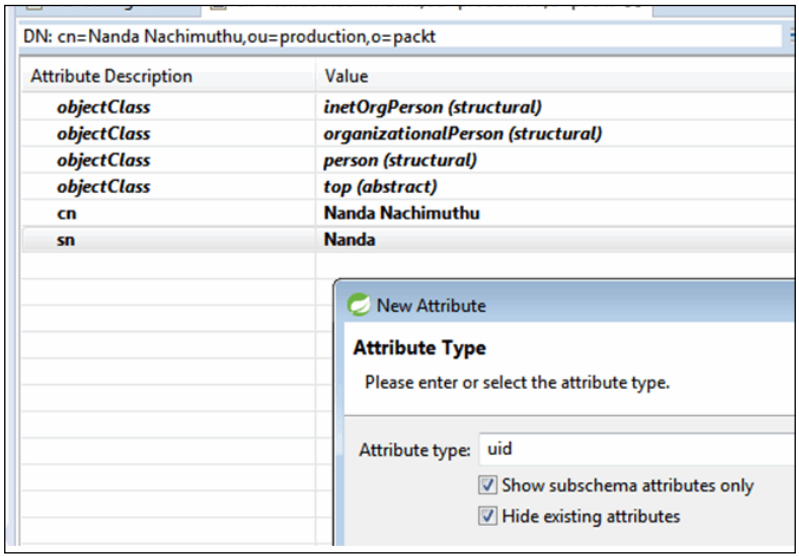


Now, let's try to add a user and password under the production unit. First, try the same steps for the sales, publishing, and accounts departments.

Now, right-click on production unit and add new entry. In the **Object Classes** dialog, select **inetOrgPerson** and click on **Next**. The **Parent** field must read `ou=production,o=packt` and enter `cn` in the RDN field and enter a first name and last name in the value field. The next dialog will be the **Attributes** dialog; enter your surname in the **sn** field:



Now, we need to provide a username to this user. Right-click on the same window and choose **New Attribute**. In the **Attribute type** field, enter `uid` and click on **Next** and **Finish** and provide a username of your choice:



Do this one more time to set the password by entering `userPassword` in the **Attribute type** field. Make sure that the `SHA hashed password` is selected. You can add some more usernames and passwords for various departments for practice.

Apache DS Studio features

Let's go through the features of Apache DS Studio quickly in order to understand the LDAP activities better:

- The LDAP browser:
    - This is used to create connections, browse directory, and search directory
    - This has many wizards to edit: **New Entry**, **New Context**, and **Attributes**
    - The LDIF and DSML export facility is available
    - Many value editors are available such as password, image, date and time, Distinguished Name, **Object Identifier** (**OID**), certificate, and so on
    - Property viewer and editor are available for connection, entry, attribute, value, and search and bookmark values are available
- Apache Directory Studio LDIF Editor:
    - New and existing files can be created and modified
    - The LDIF editor has an option to directly connect with a server and its schema
- Apache Directory Studio Schema Editor
    - Many views, such as **Hierarchy**, **Problems**, **Projects**, and so on, are available
    - This can create a new project, new schema, and export of the same
    - This also helps in creating new attributes and object classes

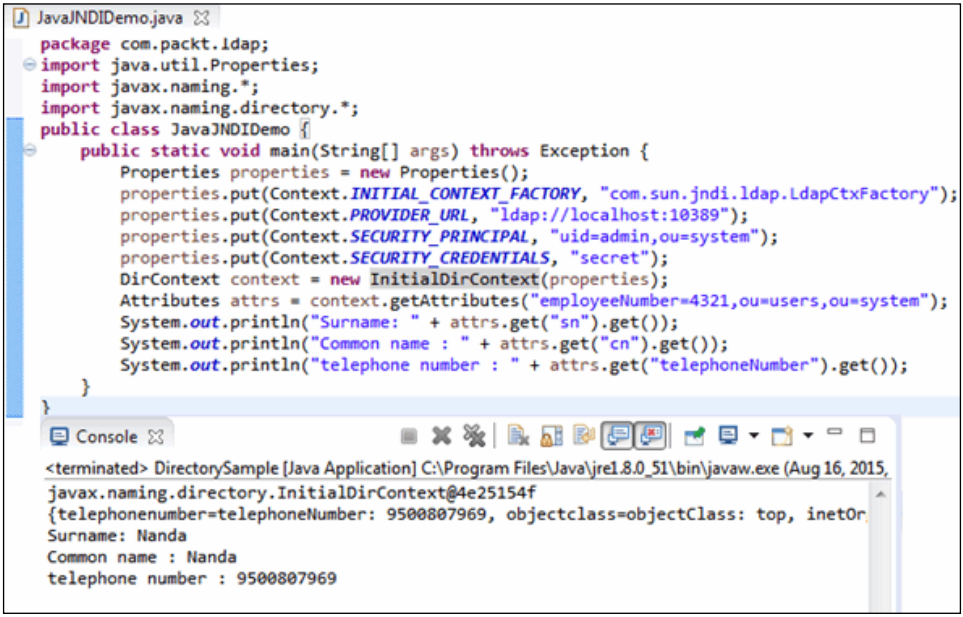
These features are mentioned here for additional knowledge about LDAP and future practices. So far, we have seen the basics of LDAP, industry standard LDAP implementations, how to install a popular Apache DS LDAP server and studio and how to integrate the same with the well-known STS.

We have configured the server, established the connection, browsed the directory, and added a few department, user, and password details.

Now, let's see how to access the LDAP server and values using the basic Java JNDI program and advanced Spring LDAP Template package.

Simple Java JNDI program to access LDAP

Like creating the username and password entries, go to the schema browser, right-click on `ou=system`, and then choose `ou=user`. Create one entry under the **inetOrgPerson** object class. In the **Distinguished Name** dialog, choose `employeeNumber` in the RDN field and provide the value as `4321`. The next dialog will be **Attributes** and give your name as `cn` and `sn`. Add one more attribute type (you can add multiple of these) as telephone number and provide a value for the purpose of testing. Create a basic Java project in STS and run the program. You can see the result in the console, as follows:



In the same set up, create another Java class and run the following code and you can see the search for the LDAP directory by providing the search criteria:

```
DirContext context = new InitialDirContext(initilaProperties);
String searchFilter = "(objectClass=inetOrgPerson)";
String[] requiredAttributes = { "employeeNumber", "cn","telephoneNumber" };
SearchControls controls = new SearchControls();
controls.setSearchScope(SearchControls.SUBTREE_SCOPE);
controls.setReturningAttributes(requiredAttributes);
NamingEnumeration users = context.search("ou=users,ou=system",searchFilter, controls);
SearchResult searchResult = null;            String commonName = null;            String empNumber = null;
while (users.hasMore()) {
    searchResult = (SearchResult) users.next();
    Attributes attr = searchResult.getAttributes();
    commonName = attr.get("cn").get(0).toString();
    empNumber = attr.get("employeeNumber").get(0).toString();
    telephoneNumber = attr.get("telephoneNumber").get(0).toString();
    System.out.println("Name = " + commonName);
    System.out.println("Employee Number = " + empNumber);
    System.out.println("Phone Number = " + telephoneNumber);
}
```

These two JNDI programs will give you the idea to create contexts by providing the LDAP parameters and browsing the LDAP directory. Let's start exploring the Spring LDAP Template now.

Spring LDAP Template – step by step

To make the LDAP calls easier, Spring has come up with an LDAP Template package, which is designed in a similar manner to `JdbcTemplate`. The LDAP Template eliminates the problem of creating and closing `LdapContext` and looping through `NamingEnumeration`. This also has a comprehensive unchecked exception hierarchy built on Spring's `DataAccessException`. The LDAP Template contains classes to dynamically build LDAP filters and **Distinguished Names** (**DNs**). The client-side LDAP transaction management is taken care by this extension. Let 's see some of the classes that are involved in this package:

- `org.springframework.ldap.core.DistinguishedName`: This is useful to build and modify the LDAP path dynamically

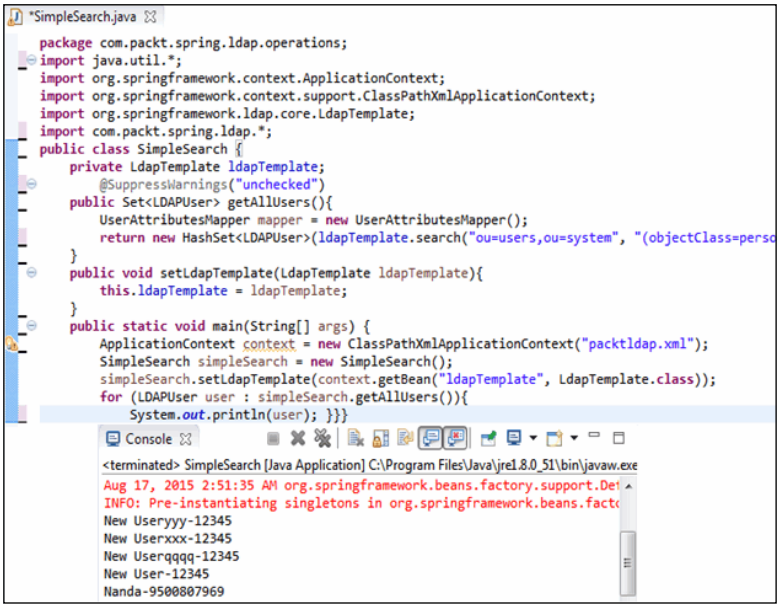
- `org.springframework.ldap.core.LdapTemplate`: This executes the core LDAP functionality and helps to avoid common errors, relieving the user of the burden of looking up contexts, looping through `NamingEnumerations`, and closing contexts
- `org.springframework.ldap.filter.AndFilter`: This adds a query to the AND expression
- `org.springframework.ldap.filter.EqualsFilter`: This is a filter for the equals operation
- `org.springframework.ldap.ldif.parser.LdifParser`: This is the base class for the Spring LDAPs LDIF parser implementation
- `org.springframework.ldap.ldif.parser.Parser`: This represents the required methods that are to be implemented by the parser utilities
- `org.springframework.ldap.support.LdapNameBuilder`: This is the helper class to build the `LdapName` instances
- `org.springframework.ldap.odm.core.OdmManager`: This is the interface for interaction with an LDAP directory

Apart from these classes, we have XML configuration files to mention the server URL and credentials details, `LDAPTemplate` bean ID, and `LdapContextSource`. The configuration file will also have the entries for the `contextSourceTarget`, `dirContextValidator`, `contextSource`, and `odmManager` factory bean. We are going to use the same setup and values as `url=ldap://localhost:10389` and `userDn = uid=admin,ou=system`. Let's see how to create the entries, search, and modify using LDAPTemplate, ApacheDS and STS.

Simple LDAP search

Create a Spring web project and create packages named `com.packt.spring.ldap`, `com.packt.spring.ldap.ldif`, `com.packt.spring.ldap.odm`, and `com.packt.spring.ldap.operations`.

As shown in the following screenshot, create the classes and the `packtldap.xml` file and keep them in the proper folders. The `UserAttributesMapper` program is also given for your reference. The `SimpleSearch` program calls the `simpleSearch.getAllUsers()` method to print out all users available in this system. Take a look at the `packtldap.xml` file, where we have provided the `url`, `userDn`, and `password`. You can see the results printed on the right-hand side console:

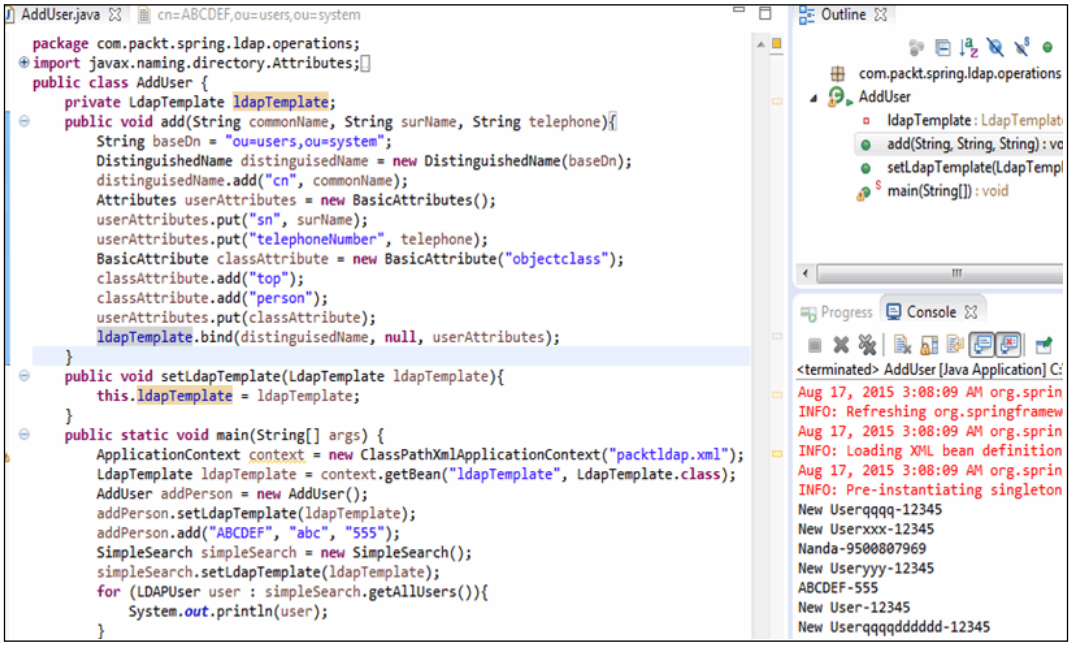


You can see the contents of `AttributeMapper`. This is used to map the LDAP attributes to the custom **LDAPUsers** object:

```
package com.packt.spring.ldap;
import org.springframework.ldap.core.AttributesMapper;
public class UserAttributesMapper implements AttributesMapper {
        public LDAPUser mapFromAttributes(Attributes attributes) throws NamingException {
                LDAPUser userObject = new LDAPUser();
                String commonName = (String)attributes.get("cn").get();
                userObject.setCommonName(commonName);
                if (attributes.get("telephoneNumber") == null){
                        System.out.println("Telephone is null for " + commonName);
                }else{
                        String telephone = attributes.get("telephoneNumber").get().toString();
                        userObject.setTelephone(telephone);
                }
                return userObject;
        }
}
```

Add, modify, and delete LDAP user

In the same package, add a class to add the LDAP user. The code and the executed result is as follows. The creation of `LDAPTemplate` and usage is given in the program. The added user details are shown in the schema browser, as follows:



The code snippet for dynamic search, modifying user, and deleting user is given in the following for your reference:

```
DynamicSearch dynamicSearch = new DynamicSearch();
dynamicSearch.setLdapTemplate(context.getBean("ldapTemplate", LdapTemplate.class));
for (LDAPUser user : dynamicSearch.getAllUsers("username")){
        System.out.println(user);
}

ModifyUser modifyPerson = new ModifyUser();
modifyPerson.setLdapTemplate(ldapTemplate);
modifyPerson.modify("some filter", "1234");

RemoveUser removePerson = new RemoveUser();
removePerson.setLdapTemplate(ldapTemplate);
removePerson.remove("some user");

SimpleSearch simpleSearch = new SimpleSearch();
simpleSearch.setLdapTemplate(ldapTemplate);
for (LDAPUser user : simpleSearch.getAllUsers()){
        System.out.println(user);
}
```

LDAP 1.3.1 features – Object Directory Mapping and LDIF parsing

First, let's see a quick example of the LDIF parser. Create `sample.ldif`, as given in the following screenshot, and store it in a readable folder:

```
dn: ou=users,ou=system
sn: SN-1
telephoneNumber: 12345
objectClass: person
objectClass: top
cn: CN-1

dn: ou=users,ou=system
sn: SN-2
telephoneNumber: 67890
objectClass: person
objectClass: top
cn: CN-2
```

You can call the `users.ldif` from the following program and parse the user details. In this program, the `Attribute` and `Attributes` classes come from the Java naming directory package. We used the `Parser` and `LDIFParser` classes from the Spring LDAP package to read and parse the LDIF file:

```
public class LDIFParserDemo {
	public static void main(String[] args) throws Exception{
		Parser parser = new LdifParser();
		parser.setResource(new FileSystemResource("users.ldif"));
		parser.open();
		while (parser.hasMoreRecords()){
			Attributes attributes = parser.getRecord();
			String personDetails = getPersonDetails(attributes);
			System.out.println(personDetails);
		}
	}
	private static String getPersonDetails(Attributes attributes) throws Exception{
		StringBuilder personDetails = new StringBuilder();
		personDetails.append("{");
		NamingEnumeration<? extends Attribute> attributeNames = attributes.getAll();
		while (attributeNames.hasMoreElements()) {
			Attribute attribute = attributeNames.next();
			personDetails.append("[" + attribute.getID());
			@SuppressWarnings("unchecked")
			NamingEnumeration<String> attributeValues = (NamingEnumeration<String>) attribute.getAll();
			while (attributeValues.hasMoreElements()){

				String attributeValue = attributeValues.next();
				personDetails.append("(").append(attributeValue).append(")");
			}
			personDetails.append("]");
		}
		personDetails.append("}");
		return personDetails.toString();
```

Object Directory Mapping (**ODM**) is used to persist and retrieve a user domain object from an LDAP directory. We need to create a user class and annotate using the `Attribute`, entry, and `Id` annotations as shown in this example:

```
@Entry(objectClasses = {"applicationEntity", "top"})
public class ApplicationEntity {

    @Id
    private Name distinguisedName;

    @Attribute(name="cn")
    private String cn;

    @Attribute(name="description")
    private String description;

    @Attribute(name="presentationAddress")
    private String presentationAddress;

    @Attribute(name="objectClass")
    private List<String> objectClassNames;

    public ApplicationEntity(){
        objectClassNames = new ArrayList<String>();
    }

    public Name getDistinguisedName() {
        return distinguisedName;
    }

    public void setDistinguisedName(Name distinguisedName) {
        this.distinguisedName = distinguisedName;
    }

    public String getCn() {
        return cn;
    }

    public void setCn(String cn) {
        this.cn = cn;
    }
```

Then, you can run the following program to see how to instantiate the ODM and set the `DistinguishedName` to get the application entity to read:

```
package com.packt.spring.ldap.odm;
import java.util.List;
import javax.naming.directory.SearchControls;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.ldap.core.DistinguishedName;
import org.springframework.ldap.odm.core.OdmManager;
public class ODMDemo{
        public static void main(String[] args) {
                ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
                OdmManager odmManager = context.getBean("odmManager", OdmManager.class);
                String baseDn = "ou=services,ou=configuration,ou=system";
                DistinguishedName distinguisedName = new DistinguishedName(baseDn);
                distinguisedName.add("cn", "Book");
                ApplicationEntity applicationEntity = odmManager.read(ApplicationEntity.class, distinguisedName);
                System.out.println(applicationEntity);
        }
}
```

In `Config.xml`, along with the `bean id` definitions for `fromStringConverter`, `toStringConverter`, and `converterManager`, we will have bean ID declarations for `LdapContextSource` and `OdmManagerImplFactoryBean` as follows:

```
<bean id="contextSourceTarget" class="org.springframework.ldap.core.support.LdapContextSource">
    <property name="url" value="ldap://127.0.0.1:10389" />
    <property name="userDn" value="uid=admin,ou=system" />
    <property name="password" value="secret" />
    <property name="pooled" value="false" />
</bean>
<bean id="odmManager" class="org.springframework.ldap.odm.core.impl.OdmManagerImplFactoryBean">
    <property name="converterManager" ref="converterManager" />
    <property name="contextSource" ref="contextSource" />
    <property name="managedClasses">
        <set>
            <value>net.javabeat.artices.spring.odm.ApplicationEntity</value>
        </set>
    </property>
</bean>
```

Summary

In this chapter, we have seen the basics of LDAP and different implementations that are available. We have gone through the features of ApacheDS, which is available as an open source. The steps involved in installing ApacheDS and Studio with STS have been discussed in detail. We were able to create the directory and values for different departments and users.

We tried many programs to call the LDAP server values from the plain Java JNDI method and we used the same steps for the Spring LDAP template extensions. The Spring LDAP template's features such as search, create, and modify have been demonstrated along with the advanced features such as the ODM and LDIF handling.

I request the readers to go through the other LDAP implementations and get the source code for this chapter from the Packt website.

4
Spring Security with AOP

To address the cross-cutting concerns, such as logging, exceptions, and security, **Aspect-Oriented Programming** (**AOP**) is used as a programming approach. By introducing AOP as a programming practice, modularization of complex coding is made possible. Usually, the program code consists of various components that will deal with many aspects, such as logging and security. With conventional programming approach, these components are written in a single program that leads to a complex and non-maintainable bundle of code, which may be a threat in the future. AOP makes programmers' lives easy as it is possible for us to compartmentalize the various cross-cutting concerns in multiple aspects. AOP is considered as a complement of OOPS rather its replacement.

The following topics will be covered in this chapter:

- AOP basics
- AOP terminologies
- Simple AOP examples
- Spring AOP using AspectJ annotations
- Securing UI invocation using Aspects

AOP basics

AOP is designed to handle modularization of concerns at source code level, which is called concerns. The concerns play a major role in AOP as this is used by multiple programs to address a specific reusable requirement; therefore, they are called cross-cutting concerns or horizontal concerns. In larger projects, a single line of code change may lead to many dependencies and impact the entire software development life cycle by undergoing processes such as code check-in, build, and testing. To make it easier, reducing the dependencies of larger code is important. AOP provides a solution to separate business code and cross-cutting concerns so that simple code changes do not affect the business functionality or behavior of the software bundle.

AOP creates a big impact on simplifying the system-level coding by implementing cross-cutting concerns. We can address each aspect separately in a modular fashion without tightly coupling the business logic and concerns. Avoiding duplication of code is also possible and we can introduce new aspects to address functionalities, which are introduced in middle of the **systems development life cycle** (**SDLC**). By implementing AOP, overdesigning and complex coding styles are avoided. Designers are made to focus only on the business logic and prototypes without bothering much about the cross-cutting concerns. To put it simply, we can say that AOP helps the developers to write small code snippets and keep them as an aspect that facilitates better maintainability and reusability, which can be applied across the entire project.

AOP terminologies

The AOP terminologies are as follows:

- **Aspect**: This is a module or package that has a set of APIs that are used to address the cross-cutting requirements. For example, an exception module can be treated as an AOP aspect for Exception Handling.
- **Join point**: The plugging point to introduce the aspect in the application is called join point. This is where the AOP code will start its activities.
- **Advice**: This is where the actual actions such as Logging or Exception are handled from the Aspect code, which is not a part of the application code. The Action can be initiated before or after the application method execution. Advice is of four types, as follows:
    - **Around advice**: This will have the custom code that is to be executed before and after the method invocation. We can decide on the proceedings to the join point, or we can return some values, or we can decide on throwing some exceptions using Around advice.

- **Before advice**: Before executing join points, this advice will be called; however, this advice will not affect the flow of the join point.
- **Throws advice**: If a method throws an exception, this advice will be invoked.
- **After returning advice**: If a join point flow ends normally, then this advice will be called.

- **Pointcut**: This is the collection of join points, where an advice should be executed. We can specify the Pointcut using expressions or patterns.
- **Introduction**: An Advice class can be added with new methods and attributes using the Introduction feature.
- **Target object**: This is also called advised object. We can advise an object using one or more Aspects. This object is always instantiated as a proxy object.
- **Weaving**: This is the process of linking many aspects with application types or objects in order to create an advised object. This can be done at compile time, load time, or run time.

Simple AOP examples

Here are the steps to run basic AOP samples that will explain the AfterAdvice, BeforeAdvice, AroundAdvice, and ExceptionAdvice concepts.

Create a Spring Maven project in STS and add the `BookService` class:

```
package com.packt.spring;

public class BookService {
    private String name;
    private String url;

    public void setName(String name) {
        this.name = name;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public void printName() {
        System.out.println("Book name : " + this.name);
    }

    public void printURL() {
        System.out.println("Book website : " + this.url);
    }

    public void printThrowException() {
        throw new IllegalArgumentException();
    }

}
```

Create `AOPBeforeMethod.java` and attach BeforeAdvice to `BookService.java` as configured in the `Spring-Book.xml`:

```
package com.packt.spring.aop;

import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class AOPBeforeMethod implements MethodBeforeAdvice
{
    @Override
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
            System.out.println("AOPBeforeMethod : Before method Captured!");
    }
}
```

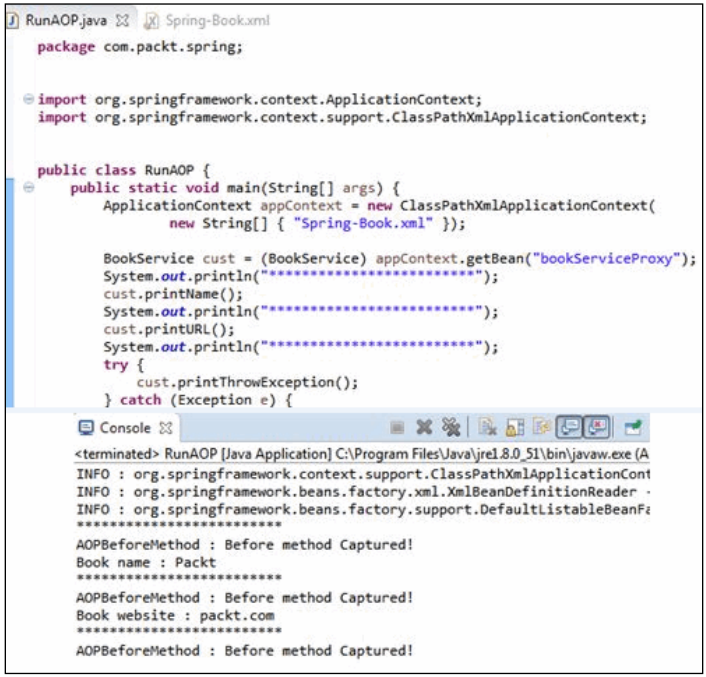
From the `config` file, you can understand that we are creating a bean for the BeforeMethod class and proxy bean for BookService. The `target` tag specifies the bean that needs to be applied with the Aspect. The `interceptorNames` denotes the Aspect class that will be attached to the proxy bean:

```
<beans xmlns="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-2.5.xsd">
    <bean id="bookService" class="com.packt.spring.BookService">
        <property name="name" value="Packt" />
        <property name="url" value="packt.com" />
    </bean>
    <bean id="AOPBeforeMethodBean" class="com.packt.spring.aop.AOPBeforeMethod" />
    <bean id="bookServiceProxy"
                class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="bookService" />
        <property name="interceptorNames">
            <list>
                <value>AOPBeforeMethodBean</value>
            </list>
        </property>
    </bean>
</beans>
```

Add the `CGLIB2` library in the **Project Object Model** (**POM**) file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packt</groupId>
    <artifactId>spring</artifactId>
    <name>AOP</name>
    <packaging>war</packaging>
    <version>1.0.0-BUILD-SNAPSHOT</version>
    <properties>
        <java-version>1.6</java-version>
        <org.springframework-version>3.1.1.RELEASE</org.springframework-version>
        <org.aspectj-version>1.6.10</org.aspectj-version>
        <org.slf4j-version>1.6.6</org.slf4j-version>
    </properties>
    <dependencies>
        <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2.2</version>
    </dependency>

```

Create `RunAOP.java` and run it as a Java application and you can see the BeforeAdvice code is executed before the actual bean method is invoked:



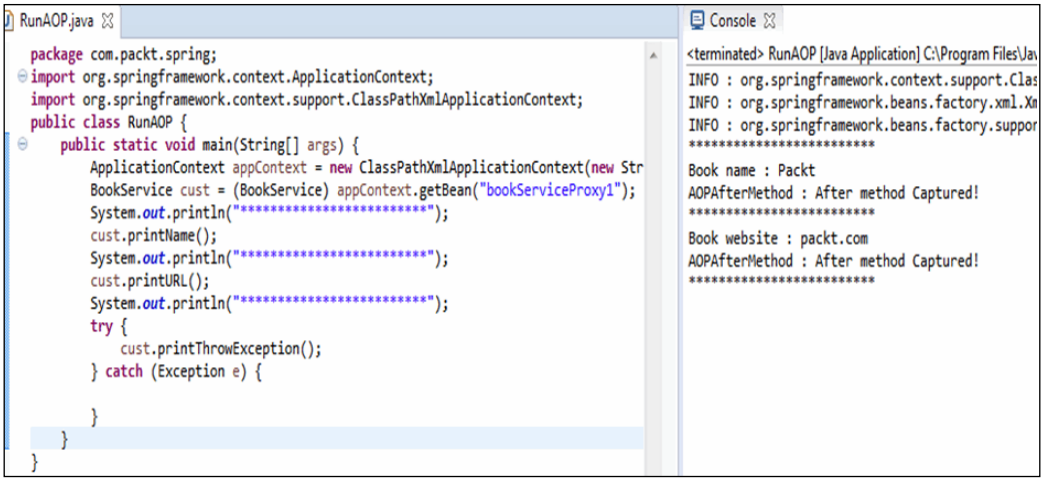
Add another AfterAdvice to the project:

```
package com.packt.spring.aop;

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

public class AOPAfterMethod implements AfterReturningAdvice
{
    @Override
    public void afterReturning(Object returnValue, Method method,
        Object[] args, Object target) throws Throwable {
            System.out.println("AOPAfterMethod : After method Captured!");
    }
}
```

Modify the configurations as shown in the following:

```
<beans xmlns="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-2.5.xsd">
    <bean id="bookService" class="com.packt.spring.BookService">
        <property name="name" value="Packt" />
        <property name="url" value="packt.com" />
    </bean>
    <bean id="AOPAfterMethodBean" class="com.packt.spring.aop.AOPAfterMethod" />
    <bean id="bookServiceProxy1"
                class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="bookService" />
        <property name="interceptorNames">
            <list>
                <value>AOPAfterMethodBean</value>
            </list>
        </property>
    </bean>
</beans>
```

Run the Java program to see After Advice in action:



You can see that After Advice is running and printing the statements after executing the method.

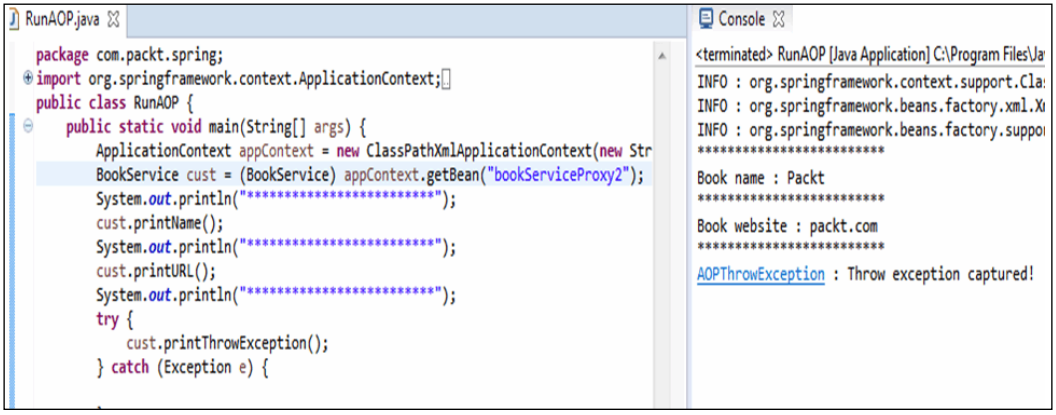
The following screenshot explains the creation of Exception Advice Java program and its usage:

```
1  package com.packt.spring.aop;
2
3  import org.springframework.aop.ThrowsAdvice;
4
5  public class AOPThrowException implements ThrowsAdvice {
6      public void afterThrowing(IllegalArgumentException e) throws Throwable {
7          System.out.println("AOPThrowException : Throw exception captured!");
8      }
9  }
10
```

Make the necessary changes in the `config` file:

```
<beans xmlns="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-2.5.xsd">
    <bean id="bookService" class="com.packt.spring.BookService">
        <property name="name" value="Packt" />
        <property name="url" value="packt.com" />
    </bean>
    <bean id="AOPThrowExceptionBean"
    class="com.packt.spring.aop.AOPThrowException" />
    <bean id="bookServiceProxy2"
    class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="bookService" />
        <property name="interceptorNames">
            <list>
                <value>AOPThrowExceptionBean</value>
            </list>
        </property>
    </bean>
</beans>
```

Modify the previous `RunAOP` program to call `bookserviceProxy2` and you will see the Exception is captured by using `ThrowsAdvice`:



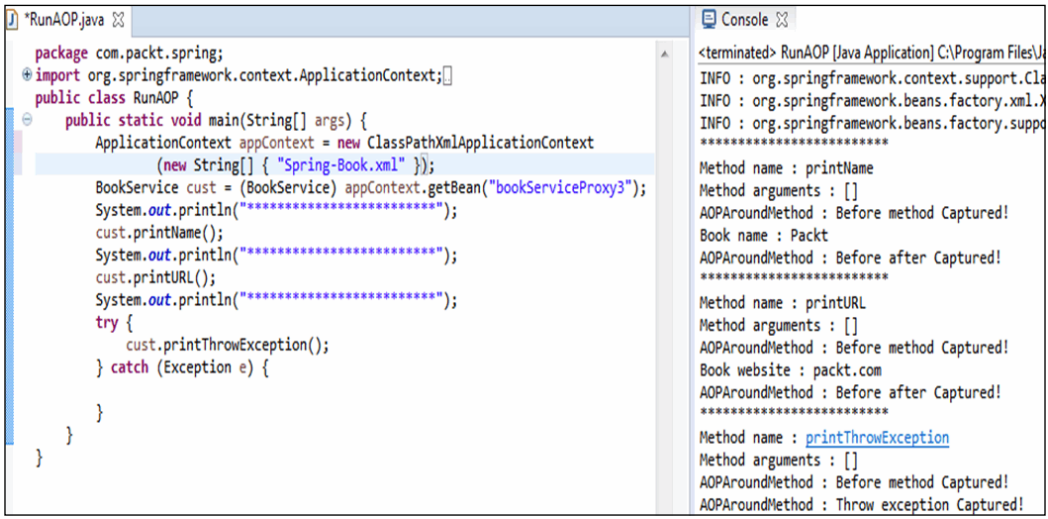
The Around Advice is important as it is a combination of all three advices that we have seen before. Please try to add a AroundAdvice as given in the following screenshot and see the results:

```
package com.packt.spring.aop;
import java.util.Arrays;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class AOPAroundMethod implements MethodInterceptor {
        @Override
        public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        System.out.println("Method name : "     + methodInvocation.getMethod().getName());
        System.out.println("Method arguments : "+ Arrays.toString(methodInvocation.getArguments()));
        System.out.println("AOPAroundMethod : Before method Captured!");
                try {
                        Object result = methodInvocation.proceed();
                        System.out.println("AOPAroundMethod : Before after Captured!");
                        return result;
                } catch (IllegalArgumentException e) {
                        System.out.println("AOPAroundMethod : Throw exception Captured!");
                        throw e;
```

Here is the `config` file changes:

```
<beans xmlns="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-2.5.xsd">
    <bean id="bookService" class="com.packt.spring.BookService">
        <property name="name" value="Packt" />
        <property name="url" value="packt.com" />
    </bean>
        <bean id="AOPAroundMethodBean"
        class="com.packt.spring.aop.AOPAroundMethod" />
    <bean id="bookServiceProxy3"
    class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="bookService" />
        <property name="interceptorNames">
            <list>
                <value>AOPAroundMethodBean</value>
            </list>
        </property>
    </bean>
</beans>
```

We will notice that in the following image, the After, Before, and Throw advices are handled using a single `Around Advice` class:



AOP Alliance

AOP Alliance is a joint open source venture having the participation of many active AOP communities, including Spring. The aim of AOP Alliance is to avoid the duplicate implementations of the same AOP features among AOP engineering groups. We can avoid rebuilding the existing AOP Alliance components by reusing them. AOP Alliance also ensures interoperability between other AOP implementations by providing a root AOP. We should use the reusable features of AOP Alliance in order to build powerful **aspect-oriented environment** (**AOE**) implementations.

Spring AOP using AspectJ Annotations

As shown in the following, create a simple Spring Maven project and classes as given in the project explorer and configure the XML file for the AspectJ, Bean Creation, and Aspect Mappings. We have added the `aop:aspectj-autoproxy` element to the config XML file in order to enable the AspectJ support:

```
<aop:aspectj-autoproxy />
<bean name="Book" class="com.packt.spring.model.Book">
    <property name="name" value="Book Name"></property>
</bean>
<bean name="BookService" class="com.packt.spring.service.BookService">
    <property name="Book" ref="Book"></property>
</bean>
<bean name="BookAspect" class="com.packt.spring.aspect.BookAspect" />
<bean name="BookAspectPointcut" class="com.packt.spring.aspect.BookAspectPointcut" />
<bean name="BookAspectJoinPoint" class="com.packt.spring.aspect.BookAspectJoinPoint" />
<bean name="BookAfterAspect" class="com.packt.spring.aspect.BookAfterAspect" />
<bean name="BookAroundAspect" class="com.packt.spring.aspect.BookAroundAspect" />
<bean name="BookAnnotationAspect" class="com.packt.spring.aspect.BookAnnotationAspect" />
 <bean name="BookXMLConfigAspect" class="com.packt.spring.aspect.BookXMLConfigAspect" />
 <!-- Spring AOP XML Configuration -->
<aop:config>
    <aop:aspect ref="BookXMLConfigAspect" id="BookXMLConfigAspectID" order="1">
        <aop:pointcut expression="execution(* com.packt.spring.model.Book.getName())" id="getNamePointcut"/>
        <aop:around method="BookAroundAdvice" pointcut-ref="getNamePointcut" arg-names="proceedingJoinPoint"/
    </aop:aspect>
</aop:config>
```

Create the `BookAspect`, `BookAnnotationAspect`, `BookAfterAspect`, and `BookAroundAspect` classes. Take a look at the AspectJ annotations that are used all over the programs for various purposes

Create the `BookAspect` class as shown in the following:

```
package com.packt.spring.aspect;
import org.aspectj.lang.annotation.*;
 @Aspect
public class BookAspect {
     @Before("execution(public String getName())")
    public void getNameAdvice(){
        System.out.println("<<BookAspect>>Executing Advice on getName()");
    }
    @Before("execution(* com.packt.spring.aspect.*.get*())")
    public void getAllAdvice(){
        System.out.println("<<BookAspect>>Service method getter called");
    }
}
```

Also, create the `BookAnnotationAspect` class as shown in the following image:

```
package com.packt.spring.aspect;
import org.aspectj.lang.annotation.*;
@Aspect
public class BookAnnotationAspect {
   @Before("@annotation(com.packt.spring.aspect.Loggable)")
   public void myAdvice(){
       System.out.println("<<BookAnnotationAspect>>Executing myAdvice!!");
   }
}
```

Create the `BookAfterAspect` class as follows:

```
package com.packt.spring.aspect;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
@Aspect
public class BookAfterAspect {
     @After("args(name)")
    public void logStringArguments(String name){
        System.out.println("<<BookAfterAspect>>Running After Advice. "+
        "String argument passed="+name);
    }
     @AfterThrowing("within(com.packt.spring.model.Book)")
    public void logExceptions(JoinPoint joinPoint){
        System.out.println("<<BookAfterAspect>>Exception thrown in Book Method="
        +joinPoint.toString());
    }
     @AfterReturning(pointcut="execution(* getName())", returning="returnString")
    public void getNameReturningAdvice(String returnString){
        System.out.println("<<BookAfterAspect>>getNameReturningAdvice executed. "
        + "Returned String="+returnString);
    }
}
```

Create the `BookAroundAspect` class as shown in the following:

```
package com.packt.spring.aspect;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
@Aspect
public class BookAroundAspect {
     @Around("execution(* com.packt.spring.model.Book.getName())")
    public Object BookAroundAdvice(ProceedingJoinPoint proceedingJoinPoint){
        System.out.println("<<BookAroundAspect>>Before invoking getName() "
                        + "method");
        Object value = null;
        try {
            value = proceedingJoinPoint.proceed();
        } catch (Throwable e) {
            e.printStackTrace();
        }
        System.out.println("<<BookAroundAspect>>After invoking getName() method. "
                        + "Return value="+value);
        return value;
    }
}
```

Then, create the `Loggable` interface:

```
package com.packt.spring.aspect;

public @interface Loggable {

}

```

Create the `BookAspectJoinPoint` class as shown in the following image:

```
package com.packt.spring.aspect;
import java.util.Arrays;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
@Aspect
public class BookAspectJoinPoint {
    @Before("execution(public void com.journaldev.spring.model..set*(*))")
    public void loggingAdvice(JoinPoint joinPoint){
        System.out.println("<<BookAspectJoinPoint>>Before running "
                        + "loggingAdvice on method="+joinPoint.toString());
        System.out.println("<<BookAspectJoinPoint>>Agruments Passed=" +
                        Arrays.toString(joinPoint.getArgs()));
    }
    @Before("args(name)")
    public void logStringArguments(String name){
        System.out.println("<<BookAspectJoinPoint>>String argument "
                        + "passed="+name);
    }
}
```

The following image shows how to create the `BookXMLConfigAspect` class:

```
package com.packt.spring.aspect;
import org.aspectj.lang.ProceedingJoinPoint;
public class BookXMLConfigAspect {
   public Object BookAroundAdvice(ProceedingJoinPoint proceedingJoinPoint){
       System.out.println("<<BookXMLConfigAspect>>Before "
                + "invoking getName() method");
       Object value = null;
       try {
           value = proceedingJoinPoint.proceed();
       } catch (Throwable e) {           e.printStackTrace();       }
       System.out.println("<<BookXMLConfigAspect>> After "
                + "invoking getName() method. Return value="+value);
       return value;
   }
}
```

Also, create the `BookAspectPointcut` class:

```
package com.packt.spring.aspect;
import org.aspectj.lang.annotation.*;
@Aspect
public class BookAspectPointcut {
     @Before("getNamePointcut()")
    public void loggingAdvice(){
        System.out.println("<<BookAspectPointcut>>Executing loggingAdvice
        on getName()");
    }
     @Before("getNamePointcut()")
    public void secondAdvice(){
        System.out.println("<<BookAspectPointcut>>Executing secondAdvice
        on getName()");
    }
     @Pointcut("execution(public String getName())")
    public void getNamePointcut(){}
      @Before("allMethodsPointcut()")
    public void allServiceMethodsAdvice(){
        System.out.println("<<BookAspectPointcut>>Before executing
        service method");
    }
     @Pointcut("within(com.packt.spring.aspect.*)")
    public void allMethodsPointcut(){}

}
```

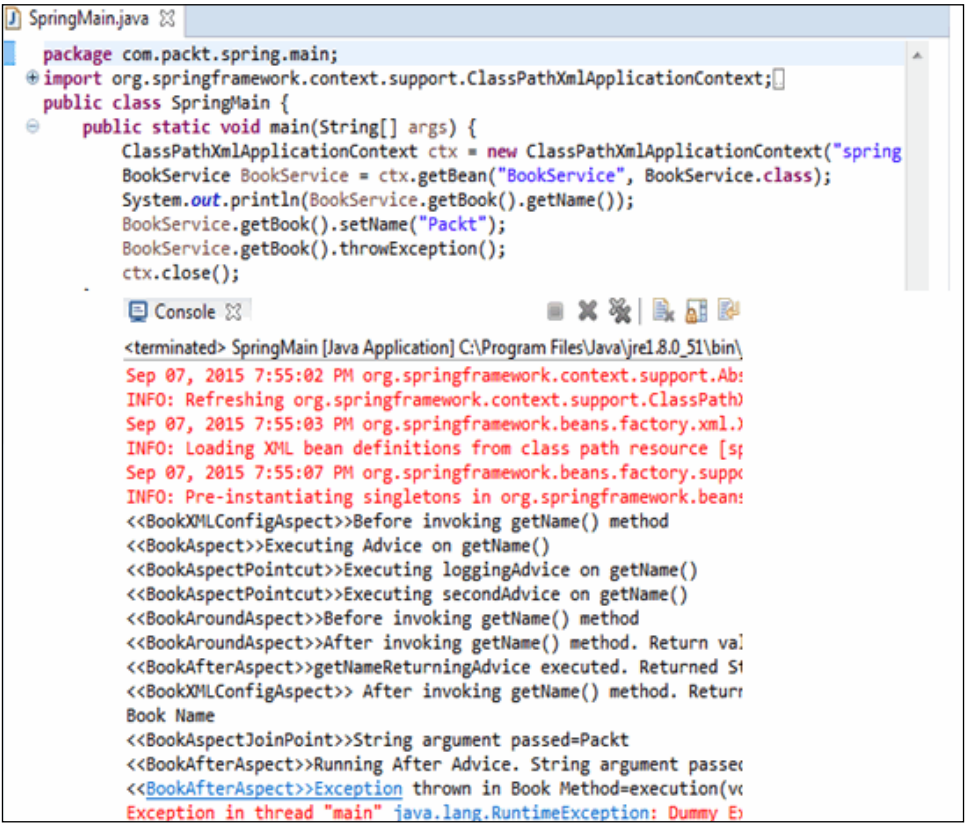
Create the `BookService`, `Book`, and `SpringMain` classes and run them as a Java application to see all Aspects in action. Please refer to the console output for Aspects that are executed in a particular order. You will notice that `BookAspectPointcut` is applied twice to the `getName()` method and executed before invoking the `getName()` method:

```
1  package com.packt.spring.model;
2  public class Book {
3     private String name;
4     public String getName() {
5         return name;
6     }
7     public void setName(String nm) {
8         this.name=nm;
9     }
10    public void throwException(){
11        throw new RuntimeException("Dummy "
12                + "Exception");
13    }
14 }
```

Finally, create the `BookService` class:

```
1  package com.packt.spring.service;
2  import com.packt.spring.model.Book;
3  public class BookService {
4      private Book Book;
5      public Book getBook(){
6          return this.Book;
7      }
8      public void setBook(Book e){
9          this.Book=e;
10     }
11 }
12
```

Run the `SpringMain` program to see the `BookAspect` invocation:



Securing UI invocation using Aspects

In this section, we are going to see how to secure method-level and object-level invocations using annotations, Aspects, and Pointcuts. Create the `UserHolder`, `UIFactory`, `UIFactoryImpl`, and `UIComponent` Java classes.

The following screenshot explains how to create the `UIFactory` and `UIFactoryImpl` classes:

```
package com.packt.spring.aop.ui;
public interface UIFactory {
        UIComponent createComponent(Class<? extends UIComponent>
        componentClass) throws Exception;
}
package com.packt.spring.aop.ui;

import org.apache.commons.lang.NullArgumentException;
import org.springframework.stereotype.Component;
@Component
public class UIFactoryImpl implements UIFactory {
        @Override
        public UIComponent createComponent(Class<? extends UIComponent>
        componentClass) throws Exception {
                if (componentClass == null) {
                        throw new NullArgumentException("Provide
                        class for the component");
                }
                return (UIComponent) Class.forName(componentClass.
                getName()).newInstance();
        }
}
```

Also, create the `UIComponent` and `UserHolder` classes:

```
package com.packt.spring.aop.ui;
public abstract class UIComponent {
        protected String componentName;

        protected String getComponentName() {
                return componentName;
        }

}
package com.packt.spring.aop.user;
import com.packt.spring.aop.type.Role;
public class UserHolder {
        private Role userRole;
        public UserHolder(Role userRole) {
                this.userRole = userRole;
        }
        public Role getUserRole() {
                return userRole;
        }
        public void setUserRole(Role userRole) {
                this.userRole = userRole;
        }
}
```

Then, create the `SecurityAnnotation` and `UserService` interfaces as given in the following:

```
package com.packt.spring.aop.annotation;
import java.lang.annotation.*;
import com.packt.spring.aop.type.Role;
@Retention(RetentionPolicy.RUNTIME)
public @interface SecurityAnnotation {
        Role[] allowedRole();
}

package com.packt.spring.aop.service;
import com.packt.spring.aop.type.Role;
import com.packt.spring.aop.user.UserHolder;

public interface UserService {
        UserHolder getCurrentUser();

        void setCurrentUser(UserHolder userHolder);

        Role getUserRole();
}
```

Create the `UserServiceImpl` and `Role` classes as follows:

```
package com.packt.spring.aop.service;
import org.springframework.stereotype.Service;
import com.packt.spring.aop.type.Role;
import com.packt.spring.aop.user.UserHolder;
@Service
public class UserServiceImpl implements UserService {
        private UserHolder userHolder;
        @Override
        public UserHolder getCurrentUser() {
                return userHolder;
        }
        @Override
        public void setCurrentUser(UserHolder userHolder) {
                this.userHolder = userHolder;
        }
        @Override
        public Role getUserRole() {
                if (userHolder == null) {
                        return null;
                }
                return userHolder.getUserRole();
        }
}
package com.packt.spring.aop.type;
public enum Role {
        ADMIN("ADM"), WRITER("WRT"), GUEST("GST"),USER("USR"),READER("RDR");
        private String name;
        private Role(String name) {
                this.name = name;
        }
        public static Role getRoleByName(String name) {
                for (Role role : Role.values()) {
                        if (role.name.equals(name)) {
                                return role;
                        }
                }
                throw new IllegalArgumentException("NO ROLES [" + name + "]");
        }
        public String getName() {
                return this.name;
        }}
```

You can see the role definitions in the `Role` class such as admin, writer, reader, and so on. The next step is to create the component classes that will be invoked, based on the security privileges.

Create `SomeComponentForGuest` and `SomeComponentForWriter` classes as shown in the following image:

```
1  package com.packt.spring.aop.component;
2
3  import com.packt.spring.aop.annotation.SecurityAnnotation;
4  import com.packt.spring.aop.type.Role;
5  import com.packt.spring.aop.ui.UIComponent;
6  @SecurityAnnotation(allowedRole = { Role.WRITER,Role.READER })
7  public class SomeComponentForWriter extends UIComponent {
8          public SomeComponentForWriter() {
9                  this.componentName = "SomeComponentForWriter";
10         }
11         public static UIComponent getComponent() {
12                 return new SomeComponentForWriter();
13         }
14 }
15 package com.packt.spring.aop.component;
16 import com.packt.spring.aop.annotation.SecurityAnnotation;
17 import com.packt.spring.aop.type.Role;
18 import com.packt.spring.aop.ui.UIComponent;
19 @SecurityAnnotation(allowedRole = { Role.GUEST })
20 public class SomeComponentForGuest extends UIComponent {
21         public SomeComponentForGuest() {
22                 this.componentName = "SomeComponentForGuest";
23         }
24         public static UIComponent getComponent() {
25                 return new SomeComponentForGuest();
26         }
27 }
```

Also, create `SomeComponentForAdmin` and `SomeComponentForAdminAndGuest` classes as shown in the following image. You can modify the allowed role by adding new roles in `Role` class and calling them here:

```
1  package com.packt.spring.aop.component;
2  import com.packt.spring.aop.annotation.SecurityAnnotation;
3  import com.packt.spring.aop.type.Role;
4  import com.packt.spring.aop.ui.UIComponent;
5  @SecurityAnnotation(allowedRole = { Role.ADMIN })
6  public class SomeComponentForAdmin extends UIComponent {
7          public SomeComponentForAdmin() {
8                  this.componentName = "SomeComponentForAdmin";
9          }
10         public static UIComponent getComponent() {
11                 return new SomeComponentForAdmin();
12         }
13 }
14 package com.packt.spring.aop.component;
15 import com.packt.spring.aop.annotation.SecurityAnnotation;
16 import com.packt.spring.aop.type.Role;
17 import com.packt.spring.aop.ui.UIComponent;
18 @SecurityAnnotation(allowedRole = { Role.ADMIN, Role.GUEST })
19 public class SomeComponentForAdminAndGuest extends UIComponent {
20         public SomeComponentForAdminAndGuest() {
21                 this.componentName = "SomeComponentForAdmin";
22         }
23         public static UIComponent getComponent() {
24                 return new SomeComponentForAdminAndGuest();
25         }
26 }
```

Here is the important class called `SecurityInterceptor` that has all pointcut and advice-related codes. These implementations will be invoked before calling the component creation code and will throw exceptions if the role is does not have the permission to invoke the component.

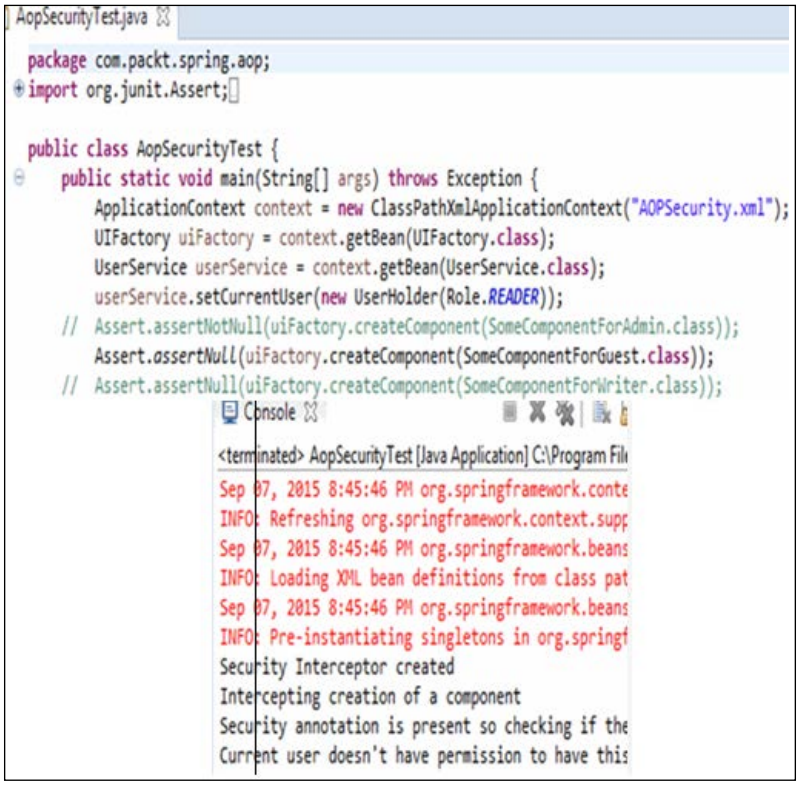
The `SecurityInterceptor` class will have the implementation to check the security access privileges. You can see that the pointcut annotation is invoked before the creation of the components:

```
@Aspect
public class SecurityInterceptor {
public SecurityInterceptor() {  System.out.println("Security Interceptor created"); }
@Autowired
private UserService userService;
@Pointcut("execution(com.packt.spring.aop.ui.UIComponent
com.packt.spring.aop.ui.UIFactory.createComponent(..))")
private void getComponent(ProceedingJoinPoint thisJoinPoint) {}
@Around("getComponent(thisJoinPoint)")
public UIComponent checkSecurity(ProceedingJoinPoint thisJoinPoint) throws Throwable {
System.out.println("Intercepting creation of a component");
Object[] arguments = thisJoinPoint.getArgs();
if (arguments.length == 0) {            return null;    }
Annotation annotation = checkTheAnnotation(arguments);
boolean atrAccessSecurityAnnotationPresent = (annotation != null);
if (atrAccessSecurityAnnotationPresent) {
boolean userHasRole = verifyRole(annotation);
if (!userHasRole) {
System.out.println("Current user doesn't have permission to have this component created");
return null;}}
System.out.println("Current user has required permissions for creating a component");
return (UIComponent) thisJoinPoint.proceed();}
private Annotation checkTheAnnotation(Object[] arguments) {
Object concreteClass = arguments[0];
AnnotatedElement annotatedElement = (AnnotatedElement) concreteClass;
Annotation annotation = annotatedElement.getAnnotation(SecurityAnnotation.class);
return annotation;  }
private boolean verifyRole(Annotation annotation) {
System.out.println("Security annotation is present so checking if the user can use it");
SecurityAnnotation annotationRule = (SecurityAnnotation) annotation;
List<Role> requiredRolesList = Arrays.asList(annotationRule.allowedRole());
Role userRole = userService.getUserRole();
boolean userHasRole = requiredRolesList.contains(userRole);     return userHasRole; }}
```

Let's see the configuration file and the testing class of the component creation. Create an XML configuration, as shown in the following screenshot. Create a testing class and call the corresponding component creation code by passing some role definitions. The console will show you the pointcut and advice messages about the security checking. You can change the role definition and component calling to test this implementation in different ways:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.1.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
        <context:annotation-config />
        <context:component-scan base-package="com.packt.spring.aop">
                <context:exclude-filter type="annotation"
                expression="org.aspectj.lang.annotation.Aspect"/>
        </context:component-scan>
        <aop:aspectj-autoproxy/>
        <bean class="com.packt.spring.aop.interceptor.SecurityInterceptor"
        factory-method="aspectOf"/>
</beans>
```

Run the JUnit test cases by invoking the `createComponent()` methods by setting different usernames using the `UserHolder` class:



Summary

In this chapter, we have covered the basic terminologies of AOP. We have gone through some simple examples of Spring AOP and AspectJ. The uses of annotations have been explained using samples and we have implemented the AOP security for method-level and UI component creation. You can extend the features and implementations that are described in this chapter in your real-time applications in order to avoid the complexities involved in cross-cutting concerns and the code simple and maintainable. In our next chapter, we will cover the access control list implementations in the Spring Framework in detail.

5
Spring Security with ACL

Access control list (**ACL**) is used to map the permissions of the objects against the users of an application. ACL will have the access grants for users and system process in order to access and perform operations on particular objects. Typically, ACL stores the operation against an object by a user. In Java programming, you can assume an entry in ACL, such as `Admin: Create` or `User: Read`, for a given screen or entity that would give permission to the `Admin` to create an entity and the `User` to read the same. The implementation of ACL using SQL and File System may also vary for different technologies.

Advanced SQL-based ACL implementations follow **role-based access control** (**RBAC**) models. The RBAC model is widely used in security applications that have complex security requirements, such as role-based data segregation. In SQL implementation, ACLs are used to manage groups, subgroups, and hierarchy of groups. The flexibility, in terms of creating and managing the Access Control Policy, is quite high in advanced SQL implementations. Different ACL algorithms can be defined using advanced SQL systems. Many modern systems such as **Enterprise resource planning** (**ERP**), **Supply chain management** (**SCM**), and **Customer relationship management** (**CRM**) use ACLs in their access control policies.

The complex applications require authentication and authorization at end user login level and role access level, where the mapping information will be retrieved from the database. They also need to be implemented at every domain object level and method instantiation level in order to ensure that the object instantiation is based on actual domain objects, which are specific to the user and the method. To achieve this, we need to combine the role, permissions, and business objects. By doing this, we can assign the permission to a set of users in order to access the specific domain objects that are intended only for them. In Spring Security, we have the Spring Security ACL package that has main classes to perform various ACL activities.

In this chapter, we are going to explore the following topics:

- The Spring ACL package and infrastructure classes
- The ACL implementation example and XML configuration for ACL

Spring ACL package and infrastructure classes

The key interfaces of Spring ACL packages are as follows:

- **Acl**: Each domain object is associated with only one ACL object. The AccessControlEntries are held by this ACL and these ACLs do not refer directly to the domain objects. Instead, they refer the Object Identity.
- **AccessControlEntry**: The **access control entry** (**ACE**) is a combination of Permission, Sid, and ACL.
- **Permission**: This is a bit masking information to specify the operation.
- **Sid**: Security Identifier is a common class that represents the principal in an authentication object.
- **Object Identity**: This is used to internally hold the domain object in the ACL module.
- **AclService**: This is used to retrieve the ACL of the given Object Identity.
- **MutableAclService**: Persisting the modified ACL.

ACL implementation example and XML configuration for ACL

We will implement the ACL functionality in a Spring Service class now. We can provide the access privileges using ACL classes, as follows:

1. Create a Spring Service class, as shown in the following:

```
package com.packt.spring.acl;

import java.util.List;

public interface BookService {

    public enum Permission {
        READ, WRITE
    }

    public void createBook(Book book);

    public Book findBookById(long id);

    public List<Book> findAllBooks();

    public void updateBook(Book book);

    public void grantPermission(String principal, Book book,
        Permission[] permissions);
}

```

2. Create the `Book.java` model class as shown in the following figure:

```
package com.packt.spring.acl;

public class Book {

    private Long id;

    private String text;

    @Override
    public String toString() {
        return id + " " + text;
    }

    public Book(long id, String text) {
        this.id = id;
        this.text = text;
    }

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

    public void setText(String text) {
        this.text = text;
    }

    public Long getId() {        return id;    }

    public String getText() {        return text;    }
}

```

3. Create the `BookServiceImpl.java` service implementation class. In this block, we will create the `grantPermission()` method that has the ACL implementation to grant permission to the given principal. We also have to provide the read and write permission entries, as follows:

```
@Repository
public class BookServiceImpl implements BookService {

    private Logger LOGGER = LoggerFactory.getLogger(BookServiceImpl.class);

    private Map<Long, Book> Books = new HashMap<>();

    @Resource
    private MutableAclService aclService;

    @Override
    @Transactional
    public void grantPermission(String principal, Book Book,
        Permission[] permissions) {
        LOGGER.info("Grant {} permission to principal {} on Book {}",
            permissions, principal, Book);
        ObjectIdentity oi = new ObjectIdentityImpl(Book.class,
            Book.getId());
        Sid sid = new PrincipalSid(principal);

        MutableAcl acl;
        try {
            acl = (MutableAcl) aclService.readAclById(oi);
        } catch (NotFoundException nfe) {
            acl = aclService.createAcl(oi);
        }

        for (Permission permission : permissions) {
            switch (permission) {
                case READ:
                    acl.insertAce(acl.getEntries().size(), BasePermission.READ,
                    sid, true);
                    break;
                case WRITE:
                    acl.insertAce(acl.getEntries().size(), BasePermission.WRITE,
                    sid, true);
                    break;
            }
        }
        aclService.updateAcl(acl);
    }

```

4. In the following block, we are adding the implementations to create, find, and update all books:

```
65      @Override
66      public void createBook(Book Book) {
67          LOGGER.info("Create Book: {}", Book);
68          Books.put(Book.getId(), Book);
69      }
70
71      @Override
72      @PreAuthorize("hasPermission(#id, 'com.packt.spring.acl.Book', 'read')
73      or hasRole('ADMIN')")
74      public Book findBookById(long id) {
75          return Books.get(id);
76      }
77
78      @Override
79      @PreAuthorize("hasPermission(#Book, 'write') or hasRole('ADMIN')")
80      public void updateBook(Book Book) {
81          Books.put(Book.getId(), Book);
82      }
83
84      @Override
85      @PostFilter("hasPermission(filterObject, 'read') or hasRole('ADMIN')")
86      public List<Book> findAllBooks() {
87          return new ArrayList<>(Books.values());
88      }
```

5. After running the first test, the following result will be printed:

```
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.243 sec
Sep 21, 2015 10:24:07 AM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.support.GenericApplicationContext@5d012c: startup date [Mon Sep 21 10:24:04
Sep 21, 2015 10:24:07 AM org.springframework.beans.factory.support.DefaultSingletonBeanRegistry destroySingletons
INFO: Destroying singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@1695df3: defining
[BookService,org.springframework.security.config.method.GlobalMethodSecurityBeanDefinitionParser$LazyInitBeanDefinitio
rk.ProxyFactoryBean#0,org.springframework.security.access.method.DelegatingMethodSecurityMetadataSource#0,org.springfr
ngframework.security.access.intercept.aopalliance.MethodSecurityInterceptor#0,org.springframework.security.methodSecur
internalAutoProxyCreator,expressionHandler,aclPermissionEvaluator,aclCache,lookupStrategy,aclService,org.springframewo
Source#0,org.springframework.transaction.interceptor.TransactionInterceptor#0,org.springframework.transaction.config.i
org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.context.annotation
k.context.annotation.internalRequiredAnnotationProcessor,org.springframework.context.annotation.internalCommonAnnotati
gurationClassPostProcessor.importAwareProcessor]; root of factory hierarchy
Sep 21, 2015 10:24:07 AM org.springframework.cache.ehcache.EhCacheManagerFactoryBean destroy
INFO: Shutting down EhCache CacheManager

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

------------------------------------------------------------------------
BUILD SUCCESS
```

6. To test the ACL implementation, we need to create a `BookServiceTest.java` JUnit test class, where we can include various test cases to access the domain object, as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/applicationContext.xml", "/applicationContext-test.xml"})
public class BookServiceTest {

    @Resource
    private BookService bookService;

    @Resource
    private MutableAclService aclService;

    private JdbcTemplate jdbcTemplate;

    @Before
    public void init() {
        SecurityContextHolder.getContext().setAuthentication(
            new UsernamePasswordAuthenticationToken("admin1", "pass1", Collections.
                singletonList(new SimpleGrantedAuthority("ADMIN"))));

        aclService.deleteAcl(new ObjectIdentityImpl(Book.class, 1), true);
        aclService.deleteAcl(new ObjectIdentityImpl(Book.class, 2), true);

        Book Book1 = new Book(1, "test");
        bookService.createBook(Book1);

        bookService.grantPermission("user1", Book1, new BookService.Permission[]
            {BookService.Permission.READ, BookService.Permission.WRITE});
        bookService.grantPermission("user2", Book1, new BookService.Permission[]
            {BookService.Permission.READ});

        Book Book2 = new Book(2, "test");
        bookService.createBook(Book2);
        bookService.grantPermission("user1", Book2, new BookService.Permission[]
            {BookService.Permission.READ, BookService.Permission.WRITE});
    }

    @Test(expected = NotFoundException.class)
    public void testGrantPermissionAuthenticationRequired() {
        SecurityContextHolder.getContext().setAuthentication(
            new UsernamePasswordAuthenticationToken("user1", "pass1", Collections.
                singletonList(new SimpleGrantedAuthority("USER"))));
        bookService.grantPermission("user1", new Book(1, "test"),
            new BookService.Permission[]{BookService.Permission.READ, BookService.Permission.WRITE});
    }
```

7. We can test and find all the books services by adding the following test cases to the existing program and execute it:

```
111
112     @Test
113     public void testFilterAdmin() {
114         List<Book> Books = bookService.findAllBooks();
115         assertEquals(2, Books.size());
116     }
117
118     @Test
119     public void testFilterUser1() {
120         SecurityContextHolder.getContext().setAuthentication(
121             new UsernamePasswordAuthenticationToken("user1", "pass1", Collections.
122                 singletonList(new SimpleGrantedAuthority("USER"))));
123         List<Book> Books = bookService.findAllBooks();
124         assertEquals(2, Books.size());
125     }
126
127     @Test
128     public void testFilterUser2() {
129         SecurityContextHolder.getContext().setAuthentication(
130             new UsernamePasswordAuthenticationToken("user2", "pass2", Collections.
131                 singletonList(new SimpleGrantedAuthority("USER"))));
132         List<Book> Books = bookService.findAllBooks();
133         assertEquals(1, Books.size());
134     }
135
136     @Test
137     public void testFilterUser3() {
138         SecurityContextHolder.getContext().setAuthentication(
139             new UsernamePasswordAuthenticationToken("user3", "pass3", Collections.
140                 singletonList(new SimpleGrantedAuthority("USER"))));
141         List<Book> Books = bookService.findAllBooks();
142         assertEquals(0, Books.size());
143     }
144
145     @Resource
146     public void setDataSource(DataSource dataSource) {
147         jdbcTemplate = new JdbcTemplate(dataSource);
148     }
149
150
151     @Resource
152     public void setDataSource(DataSource dataSource) {
153         jdbcTemplate = new JdbcTemplate(dataSource);
154     }
```

8. Take a look at the results that are generated on running the additional test cases:

```
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 1 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ] permission to principal user2 on Book 1 test
[main] INFO com.packt.spring.acl.BookServiceImpl - Create Book: 2 test
[main] ERROR com.packt.spring.acl.BookServiceImpl - Grant [READ, WRITE] permission to principal user1 on Book 2 test
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.336 sec

Results :

Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
```

9. We will explore the `config` file entries as well:

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:security="http://www.springframework.org/schema/security"
       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-3.2.xsd
       http://www.springframework.org/schema/security
       http://www.springframework.org/schema/security/spring-security-3.2.xsd">

    <bean id="BookService" class="com.packt.spring.acl.BookServiceImpl" />

    <security:global-method-security pre-post-annotations="enabled">
        <security:expression-handler ref="expressionHandler"/>
    </security:global-method-security>

    <bean id="expressionHandler"
          class="org.springframework.security.access.expression.method.
          DefaultMethodSecurityExpressionHandler">
        <property name="permissionEvaluator" ref="aclPermissionEvaluator"/>
    </bean>

    <bean id="aclPermissionEvaluator" class="org.springframework.security.acls.
    AclPermissionEvaluator">
        <constructor-arg ref="aclService" />
    </bean>

    <bean id="aclCache"
          class="org.springframework.security.acls.domain.EhCacheBasedAclCache">
        <constructor-arg>
            <bean class="org.springframework.cache.ehcache.EhCacheFactoryBean">
                <property name="cacheManager">
                    <bean class="org.springframework.cache.ehcache.
                    EhCacheManagerFactoryBean" />
                </property>
                <property name="cacheName" value="aclCache" />
            </bean>
        </constructor-arg>
    </bean>
```

In this file, we added the entries for permission evaluator that will be handled by the `DefaultMethodSecurityExpressionHandler` Spring package. The `pre` and `post` annotations of Spring `global-method-security` are enabled and the `bean id` expression handler has been configured in the previous XML file.

10. In the following block, we can see the authority settings for ownership, auditing, and modification. The `JdbcMutableACLService` Spring package is configured as the `aclService` bean and passed to `aclPermissionEvaluator`, as follows:

```
40      <bean id="lookupStrategy"
41            class="org.springframework.security.acls.jdbc.BasicLookupStrategy">
42          <constructor-arg ref="dataSource" />
43          <constructor-arg ref="aclCache" />
44          <constructor-arg>
45              <bean
46                      class="org.springframework.security.acls.domain.
47                      AclAuthorizationStrategyImpl">
48                  <constructor-arg>
49                      <list>
50                          <!-- authority for taking ownership -->
51                          <bean class="org.springframework.security.core.authority.
52                          SimpleGrantedAuthority">
53                              <constructor-arg value="ROLE_ADMIN" />
54                          </bean>
55                          <!-- authority to modify auditing -->
56                          <bean class="org.springframework.security.core.authority.
57                          SimpleGrantedAuthority">
58                              <constructor-arg value="ROLE_ADMIN" />
59                          </bean>
60                          <!-- authority to make general changes -->
61                          <bean class="org.springframework.security.core.authority.
62                          SimpleGrantedAuthority">
63                              <constructor-arg value="ROLE_ADMIN" />
64                          </bean>
65                      </list>
66                  </constructor-arg>
67              </bean>
68          </constructor-arg>
69          <constructor-arg>
70              <bean class="org.springframework.security.acls.domain.
71              ConsoleAuditLogger" />
72          </constructor-arg>
73      </bean>
74
75      <bean id="aclService"
76            class="org.springframework.security.acls.jdbc.JdbcMutableAclService">
77          <constructor-arg ref="dataSource" />
78          <constructor-arg ref="lookupStrategy" />
79          <constructor-arg ref="aclCache" />
80      </bean>
```

Summary

In this chapter, we saw the basics of ACL, available classes, and interfaces in the Spring ACL package. We have seen a working example of the basic ACL implementation with various access privileges for a given principal. Please modify the grants and principals in order to practice the ACL implementation for better understanding. In our next chapter, we will explore the JavaServer Faces security integrations with the Spring Framework.

6
Spring Security with JSF

JavaServer Faces (**JSF**) provides Java specification for various components in order to build web applications. The underlying concept of JSF is Facelets, which is a templating mechanism. The Facelet Servlet integrates the request, templates, component tree, events, and responses. The state of each component is also saved at end of each request. The advanced JSF also includes Java 5 annotations, such as `@ManagedBean`, `@ManagedProperty`, and `@FacesComponent`, which simplifies the configurations. Page transitions and rendering can be done by simply passing the name of views or facelets. The JSF MVC framework has more than 100 ready-to-use UI tags using which the reusable UI component can be built easily. JSF comes with many concepts such as Managed Beans, Navigations, Resource Bundles, Tag Libraries, Convertors, Validators, Event Handlers, and so on.

The Spring Framework is designed based on dependency injection, where as JSF is a component-based framework. Therefore, it is easy to integrate both seamlessly in user interface design and backend server-side logic. We need to configure Spring applications in order to integrate them with the JSF framework using multiple XML files.

In this chapter, we are going to explore the following topics:

- Maven dependencies
- Configuration files and entries
- JSF form creation and integration
- Spring Security implementation and execution

Maven dependencies

The following screenshot shows the required JSF and Spring dependencies with their versions:

```
<dependencies>
<dependency><groupId>javax.faces</groupId><artifactId>jsf-api</artifactId>
<version>2.0</version>
<scope>compile</scope>  </dependency>
<dependency><groupId>com.sun.faces</groupId><artifactId>jsf-impl</artifactId>
<version>2.0.2-b10</version>
<scope>compile</scope></dependency>
<dependency><groupId>javax.servlet</groupId><artifactId>jstl</artifactId>
<version>1.1.2</version>
</dependency>
<dependency><groupId>javax.servlet</groupId><artifactId>servlet-api</artifactId>
<version>2.5</version></dependency>
<dependency><groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>${spring.version}</version></dependency>
<dependency><groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>${spring.version}</version></dependency>
<dependency><groupId>org.springframework.security</groupId>
<artifactId>spring-security-core</artifactId>
<version>${spring-security.version}</version></dependency>
<dependency><groupId>org.springframework.security</groupId>
<artifactId>spring-security-config</artifactId>
<version>${spring-security.version}</version></dependency>
<dependency><groupId>org.springframework.security</groupId>
<artifactId>spring-security-web</artifactId>
<version>${spring-security.version}</version></dependency>
<dependency><groupId>org.springframework.security</groupId>
<artifactId>spring-security-taglibs</artifactId>
<version>${spring-security.version}</version></dependency></dependencies>
<repositories><repository><id>org.springframework.maven.milestone</id>
<name>Spring Maven Milestone Repository</name>
<url>http://maven.springframework.org/milestone</url>
<snapshots><enabled>false</enabled></snapshots>
</repository>
<repository><id>maven2-repository.dev.java.net</id>
<name>Java.net Repository for Maven</name>
<url>http://download.java.net/maven/2</url></repository></repositories>
```

Configuration files and entries

The configuration files that are involved in JSF and Spring Security integration are as follows:

- `web.xml`: Here, we will specify `springSecurityFilterChain` as `org.springframework.web.filter.DelegatingFilterProxy` and assign it to the filter mapping. We need to specify the listeners as `ContextLoaderListener` and `RequestContextListener` in the same file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
    <context-param> <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/root-context.xml/WEB-INF/spring/security.xml
        </param-value>
    </context-param>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>REQUEST</dispatcher>
    </filter-mapping>
    <listener>
        <listener-class>
        org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>
        org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
```

The following portion will have entries for the `appServlet`, `Faces Servlet`, and `*.jsf` pattern:

```
    <servlet>
        <servlet-name>appServlet</servlet-name>
        <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
            /WEB-INF/spring/servlet-context.xml
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>appServlet</servlet-name>
        <url-pattern>/spring/</url-pattern>
    </servlet-mapping>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>
        javax.faces.webapp.FacesServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.jsf</url-pattern>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
```

- `root-context.xml`: The context component base package will be specified in this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.packt.spring.jsfsecurity" />
</beans>
```

- `servlet-context.xml`: The `resources` mapping and `InternalResourceViewResolver` will be mentioned in this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
<resources mapping="/resources/**" location="/resources/" />
<beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>
</beans:beans>
```

- `security.xml`: This is the main file of JSF and Spring integration. We can see the authentication manager, authentication provider, and URL interceptors entries in this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="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.xsd
                        http://www.springframework.org/schema/security
            http://www.springframework.org/schema/security/spring-security.xsd">
    <http use-expressions="true">
        <intercept-url pattern="/landingpage.jsf" access="isAuthenticated()" />
        <intercept-url pattern="/**" access="permitAll()" />
        <form-login login-page="/checkuser.jsf" />
        <logout />
    </http>
    <authentication-manager>
        <authentication-provider>
            <user-service>
                <user name="admin" password="packt"
        authorities="ROLE_SUPERVISOR, ROLE_USER, ROLE_TELLER" />
            </user-service>
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

JSF form creation and integration

Let's see the files that are required for the JSF Form creation and how the integration works with the Spring Framework. Create `home.xhtml` as given in the following. This file will redirect to the `landingpage` screen:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    template="/WEB-INF/templates/template.xhtml">
    <ui:define name="content">
        <h1><h:outputText value="Welcome Page"/></h1>
        <br />
        <p>
            Enter your credentials here <h:link value="JSF Login Page"
            outcome="landingpage" />
        </p>
        </ui:define>
</ui:composition>
```

The following is the `landingpage.xhtml` page:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    template="/WEB-INF/templates/template.xhtml">
    <ui:define name="content">
        <h1><h:outputText value="Secured Login Success" /></h1>
        <br />
        <h2>Landing Page Accessed ! </h2>
    </ui:define>
</ui:composition>
```

In the `security.xml` file, we had mentioned that while accessing `landingpage`, the `checkuser` security login page will be invoked. Take a look at the `checkuser.xhtml` file as given in the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
template="/WEB-INF/templates/template.xhtml">
<ui:define name="content">
<h:body>
<h2>JSF LOGIN PAGE</h2>
<h:form id="loginForm" prependId="false">
<h:messages globalOnly="true"/>
<h:panelGrid columns="3">
    <h:outputLabel for="j_username" value="User: * " />
    <h:inputText id="j_username"  required="true" label="username"/>
    <h:message for="j_username" display="text" style="color:red"/>
    <h:outputLabel for="j_password" value="Password: * " />
    <h:inputSecret id="j_password" label="password" required="true"/>
    <h:message for="j_password" display="text" style="color:red"/>
    <h:outputLabel for="_spring_security_remember_me" value="Remember me: " />
    <h:selectBooleanCheckbox id="_spring_security_remember_me" />
</h:panelGrid>
<h:commandButton type="submit" id="login" value="Login"
action="#{loginController.doLogin}" />
</h:form></h:body></ui:define></ui:composition>
```

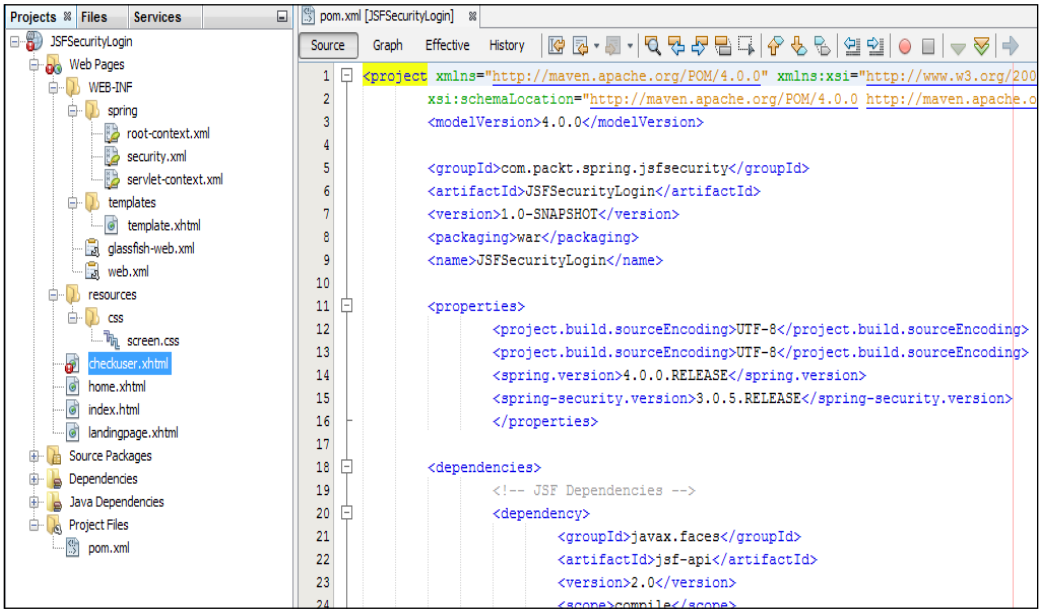
In this form, we have added the `username` and `password` fields that will be checking the values configured in user service tags of the `security.xml` file. We have configured the username as `admin` and password as `packt`.

Spring Security implementation and execution

The following `loginController` will take care of dispatcher mapping, passing inputs to controller, and performing the Spring Security check Let's see the completed package structure and execution now:

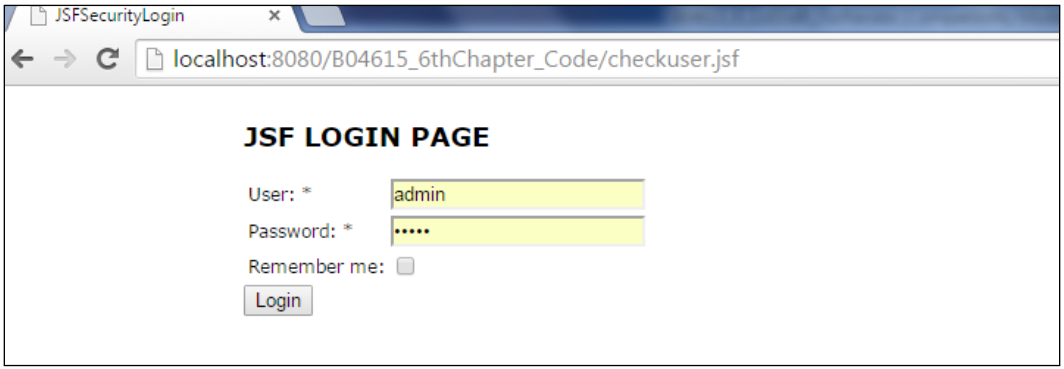
```
package com.packt.spring.jsfsecurity;
import java.io.IOException;
import javax.faces.application.FacesMessage;
import javax.faces.bean.*;
import javax.faces.context.*;
import javax.faces.event.*;
import javax.servlet.*;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.web.WebAttributes;
@ManagedBean(name="loginController")
@RequestScoped
public class LoginController implements PhaseListener {
    public String doLogin() throws ServletException, IOException {
        ExternalContext context = FacesContext.getCurrentInstance().
            getExternalContext();
        RequestDispatcher dispatcher = ((ServletRequest) context.getRequest())
                .getRequestDispatcher("/j_spring_security_check");
        dispatcher.forward((ServletRequest) context.getRequest(),
            (ServletResponse) context.getResponse());
        FacesContext.getCurrentInstance().responseComplete();
        return null;
    }
    public void afterPhase(PhaseEvent event) {
    }
    public void beforePhase(PhaseEvent event) {
        Exception e = (Exception) FacesContext.getCurrentInstance().
            getExternalContext().getSessionMap().get(
                WebAttributes.AUTHENTICATION_EXCEPTION);
         if (e instanceof BadCredentialsException) {
            FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put(WebAttributes.AUTHENTICATION_EXCEPTION, null);
            FacesContext.getCurrentInstance().addMessage(null,
                new FacesMessage(FacesMessage.SEVERITY_ERROR,
                "Input not valid.", "Input not valid"));
        }
    }
    public PhaseId getPhaseId() {
        return PhaseId.RENDER_RESPONSE;
    }
}
```

The completed package structure will look as shown in the following:

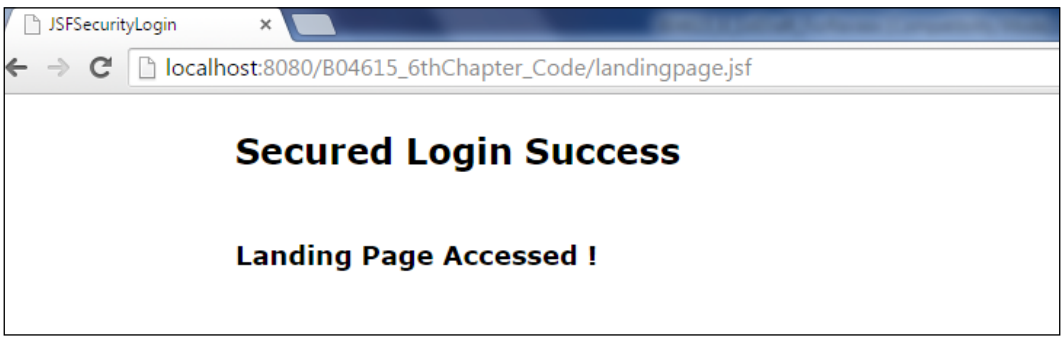


When we run the project, the `index.html` file will be invoked and `home.xhtml` will be executed, as shown in the following screenshot. You will be asked to proceed to the login page:

On clicking the login page link, you will be directed to the login form screen, where you can provide the username as `admin` and password as `packt` and submit it:



On successful login, you can reach the landing page, as shown in the following:



Summary

We have covered the JSF basics and the required Spring Security configurations in this chapter. We also tried to create a sample project from scratch and explained each artifact. You can try adding some files to the same project for practice and apply some features of JSF in order to make the project close to your real-time applications. In the next chapter, we will cover the **Apache Wicket** project creation and integration of the same with the help of Spring Security framework.

7
Spring Security with Apache Wicket

Apache Wicket is a well-designed web framework to create faster websites and web applications. Wicket's development is based on the component-oriented Java web framework concept, therefore, the reusability is high when dealing with Apache Wicket. We can feel the clear separation between markup and logic in this approach. Here, markup means pure HTML code and logic means pure Java program. The HTML developer can keep creating markups without messing up the complex templating language or input parameters. In Wicket, each page component is a real Java object and the object can persist state information, and any UI or business function can be attached to the Java object easily. The MVC pattern-based frameworks will work with whole requests and complete the set of pages, whereas in Wicket, instead of dealing with the complete set of pages, we can closely deal with each components of the pages individually as it is designed based on the component framework. Like in Swing Framework, each GUI component is a stateful component in Wicket. Each component is associated with a listener that can react to the HTML requests using events. This feature of Apache Wicket makes the application flow of control simple, manageable, and reusable.

Wicket uses the plain XHTML approach for the purpose of templating, where each component is attached to a name element and responsible for rendering of that element in the final response. The components can be further grouped into panels and dealing with multiple panels will be possible for future purpose. The components are automatically serialized and persisted while requested. The developers need not bother about how the components are interacting with their models as the objects are not exposed to the developers. By default, the server-side state is managed by Wicket; therefore, the developers don't need to deal with the `HttpSession` object directly or use a wrapper to handle or store the state. Instead, each component will associated with a **Plain Old Java Object** (**POJO**) model, which is nothing but a nested hierarchy of all the stateful components. Apache Wicket is simple in such a way that the developers don't need to handle anything in the configuration files.

In this chapter, we are going to cover the following topics:

- The Apache Wicket project with Spring Integration
- The `spring-security.xml` setup
- Executing the project

Apache Wicket project with Spring Integration

The basic structure of Apache Wicket consists of a `WebApplication` extension subclass, a `WebPage` component class, and an associated HTML file, which will be mounted in the `WebApplication` class. We can start exploring the **Project Object Model** (**POM**) file and other basic Wicket programs now:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.packt.wicket.spring.security</groupId>
<artifactId>Spring-Wicket-Security-packt</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
<name>Spring-Wicket-Security-packt</name>
<dependencies>
<dependency>
    <groupId>org.apache.wicket</groupId><artifactId>wicket-core</artifactId>
    <version>${wicket.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId><version>3.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId><version>3.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.apache.wicket</groupId><artifactId>wicket-auth-roles</artifactId>
    <version>6.7.0</version>
</dependency>
<dependency>
    <groupId>org.apache.wicket</groupId><artifactId>wicket-spring</artifactId>
    <version>6.7.0</version>
</dependency>
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId><version>1.1.2</version>
</dependency>
<dependency>
 <groupId>javax.servlet</groupId>
 <artifactId>javax.servlet-api</artifactId> <version>3.1.0</version>
 <type>jar</type>
</dependency>
```

The preceding POM file showed you the dependencies that are required for the basic Wicket and Spring Integration project.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
    <display-name>Spring-Wicket-Security-packt</display-name>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:spring-security.xml
        </param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter>
        <filter-name>wicket.Spring-Wicket-Security-packt
        </filter-name>
        <filter-class>org.apache.wicket.protocol.http.WicketFilter
        </filter-class>
        <init-param>
            <param-name>applicationClassName</param-name>
            <param-value>com.packt.wicket.spring.security.WicketApplication
            </param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>wicket.Spring-Wicket-Security-packt</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Once you are done with the POM file, we can start modifying the `web.xml` file in order to have the entries for the default `WicketApplication` program to be called in the `init-param` tags.

The spring-security.xml setup

The `spring-security.xml` file is as follows:

```
<beans:beans xmlns="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-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <http use-expressions="true" create-session="never" auto-config="true">
        <remember-me />
        <intercept-url pattern="/" access="permitAll" />
        <intercept-url pattern="/home" access="permitAll" />
        <intercept-url pattern="/login" access="permitAll" />
        <intercept-url pattern="/publish/**" access="hasRole('publisher')" />
        <intercept-url pattern="/author/**" access="hasRole('author')" />
        <intercept-url pattern="/read/**" access="hasRole('reader')" />
        <intercept-url pattern="/**" access="denyAll" />
        <form-login login-page="/login" />
    </http>
    <authentication-manager alias="authenticationManager">
        <authentication-provider>
        <user-service>
            <user name="packt" password="test"
            authorities="publisher,author,reader" />
            <user name="nanda" password="test" authorities="reader,author" />
            <user name="developer" password="test" authorities="reader" />
        </user-service>
        </authentication-provider>
    </authentication-manager>
<beans:bean id="securityContextPersistenceFilter"
class="org.springframework.security.web.context.SecurityContextPersistenceFilter" />
</beans:beans>
```

The next step is to configure the security settings in the `spring-security.xml` file. You will notice the three different user types: Publishers, Authors, and Readers. This file is important file as the main configurations for security settings are available here. Please take a look at the `intercept-url` tag, where each age is associated with different access privileges:

```
<p><a href="/Spring-Wicket-Security-packt/login">login</a>
Default Login page </p>
<p><a href="/Spring-Wicket-Security-packt/publish">publish Page</a>
Publishers can access</p>
<p><a href="/Spring-Wicket-Security-packt/author">author Page</a>
Authors and Publishers can access</p>
<p><a href="/Spring-Wicket-Security-packt/read">read Page</a>
Readers, Authors and Publishers can access</p>


package com.packt.wicket.spring.security;

import org.apache.wicket.markup.html.WebPage;
import org.apache.wicket.request.mapper.parameter.PageParameters;

public class HomePage extends WebPage {

    private static final long serialVersionUID = 1L;

    public HomePage(final PageParameters parameters) {
        super(parameters);
    }
}
```

This is our starting page called `HomePage.html`, which is associated with the subclass of `WebPage` named `HomePage.java`:

```
package com.packt.wicket.spring.security;
public enum Role {
     PUBLISHER("publish"),AUTHOR("author"),READER("read");
    private final String springSecurityRoleName;
    private Role(String springSecurityRoleName) {
        this.springSecurityRoleName = springSecurityRoleName;
    }
    public String getSpringSecurityRoleName() {
        return springSecurityRoleName;
    }
}
```

We can store the roles in a `Role.java` enumeration file that will be referred all over the application.

The following image shows `LoginPage.html` and `LoginPage.java`. You can see the login form implementation and form data handling in the `onSubmit()` method:

```
<body>
        <h2>Login</h2>
        <form wicket:id="loginForm">
            <p> Username: <input type="text" wicket:id="username"/></p>
            <p>Password: <input type="password" wicket:id="password"/></p>
            <input type="submit" value="Login"/>
        </form>
</body>
public class LoginPage extends WebPage {
    public LoginPage(PageParameters parameters) {
        add(new LoginForm("loginForm"));
    }
    private class LoginForm extends Form<Void> {
    private transient RequestCache requestCache = new HttpSessionRequestCache();
    private String username;private String password;
    public LoginForm(String id) {
            super(id);
            setModel(new CompoundPropertyModel(this));
            add(new RequiredTextField<String>("username"));
            add(new PasswordTextField("password"));
        }
        @Override
        protected void onSubmit() {
            HttpServletRequest servletRequest = (HttpServletRequest)
                RequestCycle.get().getRequest().getContainerRequest();
            String originalUrl = getOriginalUrl(servletRequest.getSession());
            AuthenticatedWebSession session = AuthenticatedWebSession.get();
            if (session.signIn(username, password)) {
                if (originalUrl != null) {
                    throw new RedirectToUrlException(originalUrl);
                } else {

                    setResponsePage(getApplication().getHomePage());
                }
            } else {error("Login failed due to invalid credentials");}
        }
        private String getOriginalUrl(HttpSession session) {
            SavedRequest savedRequest = (SavedRequest) session.
                getAttribute("SPRING_SECURITY_SAVED_REQUEST");
            if (savedRequest != null) {
                return savedRequest.getRedirectUrl();
            } else {
                return null;    }
```

The following image shows `WicketApplication.java`:

```
1  package com.packt.wicket.spring.security;
2
3  import org.apache.wicket.authroles.authentication.AbstractAuthenticatedWebSession;
4  import org.apache.wicket.authroles.authentication.AuthenticatedWebApplication;
5  import org.apache.wicket.markup.html.WebPage;
6  import org.apache.wicket.spring.injection.annot.SpringComponentInjector;
7
8  public class WicketApplication extends AuthenticatedWebApplication {
9
10     @Override
11     public Class<? extends WebPage> getHomePage() {
12         return HomePage.class;
13     }
14
15     @Override
16     public void init() {
17         super.init();
18
19         getComponentInstantiationListeners().add(new SpringComponentInjector(this));
20
21        mountPage("login", LoginPage.class);
22        mountPage("home", HomePage.class);
23        mountPage("publish", PublishPage.class);
24        mountPage("author", AuthorPage.class);
25        mountPage("read", ReaderPage.class);
26     }
27
28     @Override
29     protected Class<? extends WebPage> getSignInPageClass() {
30         return LoginPage.class;
31     }
32
33     @Override
34     protected Class<? extends AbstractAuthenticatedWebSession> getWebSessionClass() {
35         return SecureWebSession.class;
36     }
37 }
```

This is our main Wicket program that will be configured as the starting point of the project. Here, you can see the `init()` method that is registering all the pages using the `mountPage()` method:

```
public class SecureWebSession extends AuthenticatedWebSession {
    private HttpSession httpSession;
    @SpringBean(name = "authenticationManager")
    private AuthenticationManager authenticationManager;
    public SecureWebSession(Request request) {
        super(request);
        this.httpSession = ((HttpServletRequest) request.
            getContainerRequest()).getSession();
        Injector.get().inject(this);    }
    @Override
    public boolean authenticate(String username, String password) {
        try {
            Authentication auth = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(username, password));
            if (auth.isAuthenticated()) {
                SecurityContextHolder.getContext().setAuthentication(auth);
                httpSession.setAttribute(HttpSessionSecurityContextRepository.
                    SPRING_SECURITY_CONTEXT_KEY,
                        SecurityContextHolder.getContext());
                return true;
            } else {                return false;           }
        } catch (AuthenticationException e) {
            return false;       }
    }
    @Override
    public Roles getRoles() {
        Roles roles = new Roles();
        if (isSignedIn()) {
            Authentication authentication = SecurityContextHolder.
                getContext().getAuthentication();
            addRolesFromAuthentication(roles, authentication);
        }
        return roles;  }
    private void addRolesFromAuthentication(Roles roles,
        Authentication authentication) {
        for (GrantedAuthority authority : authentication.getAuthorities()) {
            roles.add(authority.getAuthority());
        }   }
    public boolean hasRole(Role role) {
        return getRoles().hasRole(role.getSpringSecurityRoleName());
    }
}
```

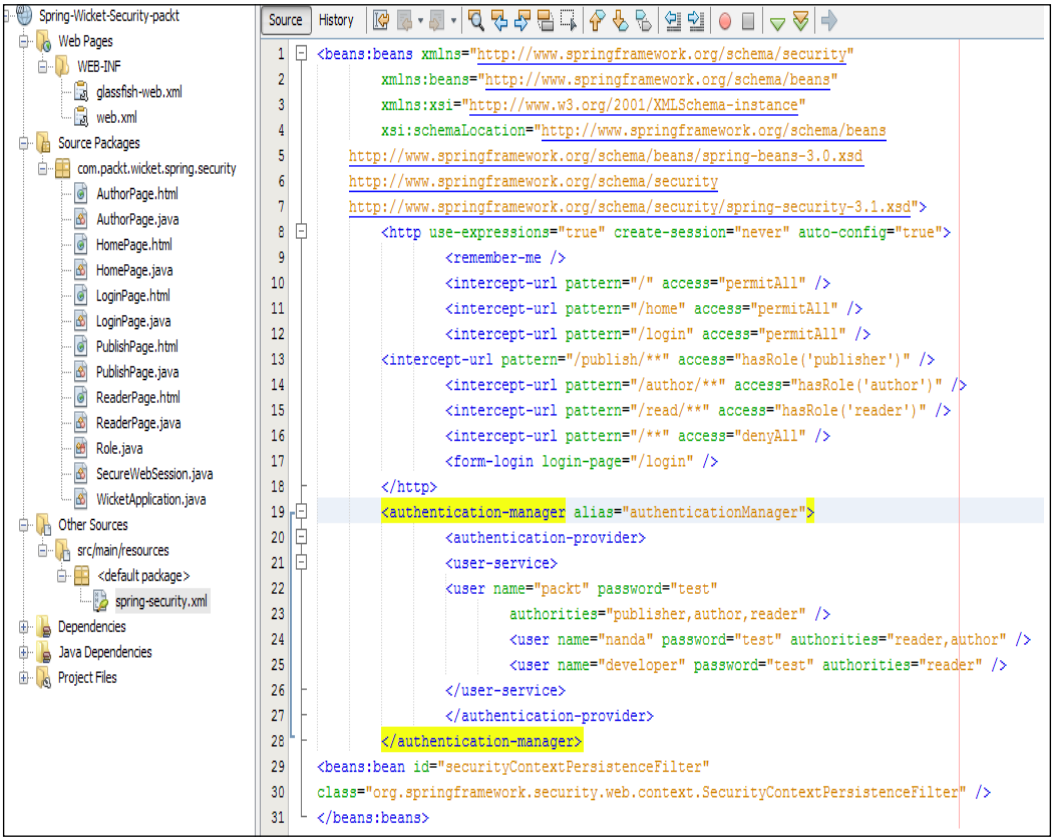
The `SecureWebSession` program will authenticate the username and password combination by creating a new username and password authentication token.

The request object will be passed to this program and the roles will be accessed through the `getRoles()` method:

```
import org.apache.wicket.markup.html.WebPage;
public class PublishPage extends WebPage {}
PublishPage.html : <h2>Publisher</h2>

import org.apache.wicket.markup.html.WebPage;
public class AuthorPage extends WebPage {}
AuthorPage.html : <h2>Author</h2>

import org.apache.wicket.markup.html.WebPage;
public class ReaderPage extends WebPage {}
ReaderPage.html : <h2>Reader</h2>
```

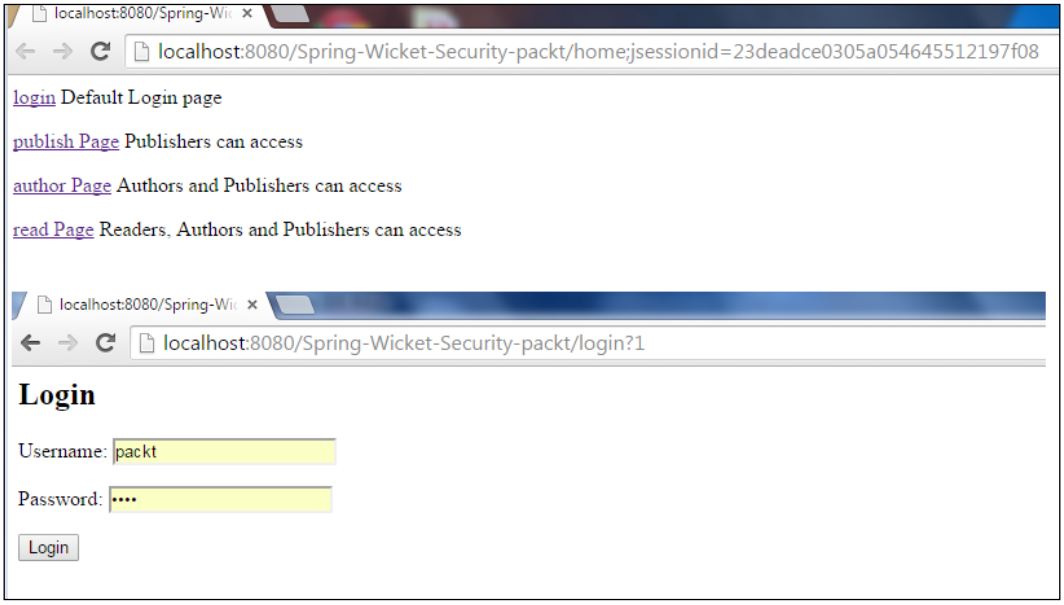
Here is our `PublishPage.Html`, `AuthorPage.Html`, and `ReaderPage.Html` and the corresponding `PublishPage.java`, `AuthorPage.java`, and `ReaderPage.java` programs.

The final project will look similar to the following screenshot:



Execution of the Project

On execution, you will be directed to the login page and other publisher, author, and reader pages, where you can apply the security credentials that are set in the Spring Security `config` file. You can see that the Publishers can access all pages, while Authors and Readers cannot access the restricted pages:



Summary

In this chapter, we went through the basic Apache Wicket application structure and sample project. We have seen the configurations that are required from the Spring perspective and dependencies required in the Maven POM file. We made the security credentials settings in the Spring Security file and executed the sample application by entering different security credentials for different types of users. In our next chapter, we will explore the Spring Security concepts to handle the SOAP web service security.

8
Integrating Spring Security with SOAP Web Services

Spring Framework comes with a web service package that helps the developers to build new SOAP web services. Spring-WS project includes a substantial list of APIs for various web services framework such as JAX-WS, Axis, JBossWS, and so on. **Simple Object Access Protocol** (**SOAP**) is a protocol to exchange structured information across applications. The application can initiate a SOAP request to a web service provider with the required parameters and the response will be provided in the XML format. The SOAP specification consists of SOAP processing model, extensibility model, protocol binding framework, and message construction. SOAP can be used with any transport protocols such as SMTP, JMS, and Message Queues. Compared to the RESTful services, the SOAP web services are more verbose in nature. To discover the SOAP service, we have to depend only on **Web Services Description Language** (**WSDL**) mechanism.

In this chapter, we will learn the following:

- Creating SOAP web service with security
- Client creation for consuming the web service
- Executing the project

Creating SOAP web service with security

We have two popular ways to create a web service called code first and contract first. Code first is the popular method and it is a bottom-up approach and is simpler to implement. We need to create services as classes and interfaces and use tools to generate corresponding WSDLs. We have three ways of dynamically generating WSDLs, that is, manually generating WSDL using a tool, automatic generation using Spring factory, or automatic generation from the Java 5 annotations. In this approach, developers need not worry about the knowledge of SOAP, WSDL, or even XML. The deployment is also faster using few annotations. The other method is contract-first method that is a top-down approach, it means that we need to create the WSDL first with all types, messages, port, binding, and services and then create the implementation from the tools such as Ant script and IDE plugin. To make the development simple, Spring-WS does not support contract-first method. Also, Spring supports different forms of messages such as standard XML APIs (SAX, dom4j, and JDom) and Serialized Java object (JAXB, Castor, XMLBeans, and XStream).

Create a Spring-WS project in your favorite IDE and define the contract data to be transferred in the form of the XML schema. The **XML Schema Definition** (**XSD**) nomenclature is important as the relationship between the request and response binding is tightly coupled with this. Once the message is constructed, the next step is to create the endpoint that will interpret the incoming messages and return the response. Spring provides many base classes to retrieve messages in different forms such as SAX, JDOM, DOM4J, Stax, JAXB, Castor, and so on.

The following screenshot shows `pom.xml` that consists of the dependencies required to create a Spring-WS project:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packtpub.spring.soap.security</groupId>
    <artifactId>spring_soap_security</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>spring_soap_security</name>
    <repositories>
        <repository>
            <id>jboss</id>
            <name>jboss Repository</name>
            <url>http://repository.jboss.org/maven2</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>org.springframework.ws</groupId>
            <artifactId>spring-ws-security</artifactId>
            <version>2.0.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>3.0.5.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
          <version>1.2.9</version>
        </dependency>
    </dependencies>
</project>
```

The following is the screenshot of the `web.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
    <display-name>spring_soap_security</display-name>
    <context-param>
        <param-name>log4jConfigLocation</param-name>
        <param-value>/WEB-INF/log4j.properties</param-value>
    </context-param>
    <context-param>
        <param-name>log4jRefreshInterval</param-name>
        <param-value>1000</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.util.Log4jConfigListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>transformWsdlLocations</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The `Web.xml` file will have the configurations such as `MessageDispatcherServlet` and `Servlet` mappings:

```
    <context:component-scan base-package="com.packtpub.spring_soap_security.service" />
    <sws:annotation-driven />
    <sws:dynamic-wsdl id="BookService" portTypeName="BookService"
        locationUri="http://localhost:8080/spring_soap_security/spring-ws/BookService"
        targetNamespace="http://www.packtpub.com/spring_soap_security/BookService/schema">
        <sws:xsd location="/WEB-INF/BookService.xsd" />
    </sws:dynamic-wsdl>
    <sws:interceptors>
        <bean
            class="org.springframework.ws.soap.server.endpoint.interceptor.
            PayloadValidatingInterceptor">
            <property name="schema" value="/WEB-INF/BookService.xsd" />
            <property name="validateRequest" value="true" />
            <property name="validateResponse" value="true" />
        </bean>
                <bean class="org.springframework.ws.soap.security.xwss.XwsSecurityInterceptor">
                <property name="policyConfiguration" value="/WEB-INF/securityPolicy.xml"/>
                <property name="callbackHandlers">
                    <list>
                          <ref bean="springSecurityHandler"/>
                    </list>
                </property>
            </bean>
    </sws:interceptors>
  <bean id="springSecurityHandler"
      class="org.springframework.ws.soap.security.xwss.callback.
      SpringDigestPasswordValidationCallbackHandler">
       <property name="userDetailsService" ref="userDetailsService"/>
  </bean>
  <bean id="userDetailsService" class="com.packtpub.spring_soap_security.service.dao.
  MyUserDetailService" />
</beans>
```

We need to provide the `spring-ws-servlet` xml configurations to mention `dynamic-wsdl`, `PayloadValidatingInterceptor`, and `XwsSecurityInterceptor`. Here, we also have to mention `SpringDigestPasswordValidationCallbackHandler` that is `springSecurityHandler`. The `spring-ws-servlet.xml` file contains Spring Web Services specific beans information and it is used to create a new container for WS Beans:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:RequireTimestamp maxClockSkew="60" timestampFreshnessLimit="300"/>
    <xwss:RequireUsernameToken passwordDigestRequired="true" nonceRequired="true"/>
</xwss:SecurityConfiguration>
```

This is the `securityPolicy.xml` file, where we will specify the `SecurityConfiguration` details.

Here comes the XSD file, which has the elements declaration. Here, we specified the `Book`, `BookResponse`, and `BookRequest` object:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.packtpub.com/spring_soap_security/BookService/schema"
    xmlns:tns="http://www.packtpub.com/spring_soap_security/BookService/schema"
    elementFormDefault="qualified"
    xmlns:QBook="http://www.packtpub.com/spring_soap_security/BookService/schema">
    <element name="getBookRequest">
        <complexType>
            <sequence>
                <element name="Book" type="QBook:Book"></element>
            </sequence>
        </complexType>
    </element>
    <element name="getBookResponse">
        <complexType>
            <sequence>
                <element name="refNumber" type="string"></element>
            </sequence>
        </complexType>
    </element>
   <complexType name="Book">
      <sequence>
         <element name="refNumber" type="string"></element>
         <element name="customer" type="QBook:Customer"></element>
         <element name="dateSubmitted" type="dateTime"></element>
         <element name="BookDate" type="dateTime"></element>
      </sequence>
  </complexType>
</schema>
```

Let's start creating the `UserDetailsService` as shown in the following. This class contains the methods to load the user details by name and gets the user data from DAO class:

```
package com.packtpub.spring_soap_security.service.dao;

import org.springframework.dao.DataAccessException;
import org.springframework.security.core.authority.GrantedAuthorityImpl;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;

public class MyUserDetailService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException, DataAccessException {

        return getUserDataFromDao(username);
    }

    private MyUserDetail getUserDataFromDao(String username) {

        MyUserDetail mydetail=new MyUserDetail(username,"pass",true,true,true,true);
        mydetail.getAuthorities().add(new GrantedAuthorityImpl("ROLE_GENERAL_OPERATOR"));

        return mydetail;

    }
}
```

The next step is to create the `UserDetails` class that is used to store the user information:

```
public class MyUserDetail implements UserDetails {
    private String password;
    private String userName;
    private boolean isAccountNonExpired;
    private boolean isAccountNonLocked;
    private boolean isCredentialsNonExpired;
    private boolean isEnabled;
    public static  Collection<GrantedAuthority> authority =
        new ArrayList<GrantedAuthority>();
      public MyUserDetail( String userName, String password,boolean isAccountNonExpired,
          boolean isAccountNonlocked,boolean isCredentialsNonExpired, boolean isEnabled){
          this.userName=userName;
          this.password=password;
          this.isAccountNonExpired=isAccountNonExpired;
          this.isAccountNonLocked=isAccountNonlocked;
          this.isCredentialsNonExpired=isCredentialsNonExpired;
          this.isEnabled=isEnabled;
      }
    @Override
    public Collection<GrantedAuthority> getAuthorities() {
        return authority;
    }
    @Override
    public String getPassword() {
        return password;
    }
    @Override
    public String getUsername() {
        return userName;
    }
```

Now, we can write the `Service Implementation` class that will implement the `getBook` and `cancelBook` method, as shown in the following screenshot:

```
package com.packtpub.spring_soap_security.service;
public interface BookService {
    String getBook( String fName,String lName,String refNumber);
    boolean cancelBook( String refNumber );
}


package com.packtpub.spring_soap_security.service;
import org.springframework.stereotype.Service;
@Service
public class BookServiceImpl implements BookService {
    public String getBook( String fName,String lName,String refNumber){
        return "Book-"+fName+"_"+lName+"_"+refNumber;
    }
    public boolean cancelBook( String refNumber ){
        return true;
    }
}
```

The next step is to create the `Endpoint` class in order to implement the `handlegetBookRequest` and `handleCancelBookRequest` methods. We need to use `@ResponsePayload` and `@RequestPayload` to specify the local parts and name spaces:

```
@Endpoint
public class BookServiceEndpoint {
    private final String SERVICE_NS =
        "http://www.packtpub.com/spring_soap_security/BookService/schema";
    private BookService BookService;
    @Autowired
    public BookServiceEndpoint(BookService BookService) {
        this.BookService = BookService;
    }
    @PayloadRoot(localPart = "getBookRequest", namespace = SERVICE_NS)
    public @ResponsePayload
    Source handlegetBookRequest(@RequestPayload Source source) throws Exception {
        String fName="Packt";
        String lName="Pub";
        String refNumber="567";
        return new StringSource(
        "<tns:getBookResponse xmlns:tns=\"http://www.packtpub.com/spring_soap_security/
        BookService/schema\"><tns:refNumber>"+BookService.getBook(fName, lName, refNumber)+
            "</tns:refNumber></tns:getBookResponse>");
    }
    @PayloadRoot(localPart = "cancelBookRequest", namespace = SERVICE_NS)
    public @ResponsePayload
    Source handleCancelBookRequest(@RequestPayload Source source) throws Exception {
        //extract data from input parameter
        String refNumber="1234";
            return new StringSource(
                    "<tns:cancelBookResponse xmlns:tns=\"http://www.packtpub.com/
            spring_soap_security/BookService/schema\"><tns:cancelled>"+
                BookService.cancelBook(refNumber)+"</tns:cancelled></tns:cancelBookResponse>");
    }
}
```

Client creation to consume the web service

The following screenshot shows the POM file for the client project. We need to specify the Spring Web Services package and repository information in this file:

```
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.packtpub.spring.soap.security.test</groupId>
    <artifactId>spring_soap_security_Client</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>spring_soap_security_Client</name
    <repositories>
         <repository>
            <id>jboss</id>
            <name>jboss Repository</name>
            <url>http://repository.jboss.org/maven2</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>org.springframework.ws</groupId>
            <artifactId>spring-ws-security</artifactId>
            <version>2.0.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>3.0.5.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.9</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>3.0.5.RELEASE</version>
        </dependency>
    </dependencies>
```

Next, we see the following implementation:

```
<xwss:SecurityConfiguration dumpMessages="true"
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
    <xwss:Timestamp />
        <xwss:UsernameToken name="PacktUser"
        password="password"
        digestPassword="true" useNonce="true"/>
</xwss:SecurityConfiguration>
```

This is the security policy file, where the username and password information are being stored. Here, you can see `digestPassword` and `useNonce` is set to `true`.

The next step is to create the application the `context.xml` file, as shown in the following screenshot. Here, we need to specify the `SecurityInterceptor` and Spring callback handler called `SimplePasswordValidationCallbackHandler`:

```
<bean id="messageFactory"
class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory" />
<bean id="webServiceTemplate"
class="org.springframework.ws.client.core.WebServiceTemplate">
<constructor-arg ref="messageFactory" />
<property name="defaultUri"
    value="http://localhost:8080/spring_soap_security/spring-ws/BookService" />
<property name="interceptors">

    <list>
        <ref local="xwsSecurityInterceptor" />
    </list>
</property>
</bean>
<bean id="xwsSecurityInterceptor"
    class="org.springframework.ws.soap.security.xwss.XwsSecurityInterceptor">
    <property name="policyConfiguration" value="/securityPolicy.xml"/>
    <property name="callbackHandlers">
            <list>
                <ref bean="callbackHandler"/>
            </list>
    </property>
</bean>
<bean id="callbackHandler" class="org.springframework.ws.soap.security.xwss.
callback.SimplePasswordValidationCallbackHandler"/>
    <bean id="log4jInitialization"
    class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetClass" value="org.springframework.util.Log4jConfigurer" />
    <property name="targetMethod" value="initLogging" />
    <property name="arguments">
        <list>
            <value>src/test/resources/log4j.properties</value>
        </list>
    </property>
```

The following screenshot shows the response and request `xml` objects that will be used to transport the data:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:placeBookRequest xmlns:tns="http://www.packtpub.com/
spring_soap_security/BookService/schema">
  <tns:order>
    <tns:refNumber>1234</tns:refNumber>
    <tns:customer>
      <tns:addressPrimary>
        <tns:doorNo>22</tns:doorNo>
        <tns:building>12</tns:building>
        <tns:street>ABC</tns:street>
        <tns:city>Mumbai</tns:city>
        <tns:phoneLandLine>12345678</tns:phoneLandLine>
        <tns:email>packt@packt.com</tns:email>
      </tns:addressPrimary>
      <tns:name>
        <tns:fName>Packt</tns:fName>
        <tns:mName>Pub</tns:mName>
        <tns:lName>Pact</tns:lName>
      </tns:name>
    </tns:customer>
    <tns:dateSubmitted>2008-09-29T05:49:45</tns:dateSubmitted>
    <tns:orderDate>2014-09-19T03:18:33</tns:orderDate>
    <tns:items>
      <tns:type>Book1</tns:type>
      <tns:name>Book2</tns:name>
      <tns:quantity>8</tns:quantity>
    </tns:items>
  </tns:order>
</tns:placeBookRequest>

<tns:placeBookResponse
xmlns:tns="http://www.packtpub.com/spring_soap_security/
BookService/schema">
<tns:refNumber>Packt_pub_1234</tns:refNumber>
</tns:placeBookResponse>
```

Executing the project

To execute the project, we need to create a `BookServiceClientTest` class as follows. You can see the `InputStream` declarations for request and response XMLs:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("/applicationContext.xml")
public class BookServiceClientTest {
    @Autowired
    private  WebServiceTemplate webServiceTemplate;
    private static InputStream placeBookRequest;
    private static InputStream placeBookResponse;
    @Autowired
    private GenericApplicationContext applicationContext;
    @Before
    public   void setUpBefore() {
        placeBookRequest = new BookServiceClientTest().getClass().
            getResourceAsStream("placeBookRequest.xml");
        placeBookResponse = new BookServiceClientTest().getClass().
            getResourceAsStream("placeBookResponse.xml");

    }
    @Test
    public  final void testPlaceBookRequest() throws Exception {
        Result result = invokeWS(placeBookRequest);
         XMLAssert.assertXMLEqual("Invalid content received",
             getStringFromInputStream(placeBookResponse), result.toString());
    }
    private    Result invokeWS(InputStream is) {
        StreamSource source = new StreamSource(is);
        StringResult result = new StringResult();
        webServiceTemplate.sendSourceAndReceiveToResult(source, result);
        return result;
    }

```

```
INFO: ==== Sending Message Start ====
...
<SOAP-ENV:Envelope ...>
<SOAP-ENV:Header>
<wsse:Security ...>
<wsu:Timestamp ...>
<wsu:Created>2015-11-06T10:06:36.</wsu:Created>
<wsu:Expires>2015-11-06T10:02:36.</wsu:Expires>
</wsu:Timestamp>
<wsse:UsernameToken...>
<wsse:Username>PacktUser</wsse:Username>
<wsse:Password #PasswordDigest">****</wsse:Password>
<wsse:Nonce #Base64Binary">...</wsse:Nonce>
<wsu:Created>2011-11-06T10:04:36.683Z</wsu:Created>
</wsse:UsernameToken>
</wsse:Security>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<tns:placeOrderRequest xmlns:tns="http://www.packtpub.com/
liverestaurant/OrderService/schema">
......
Securing SOAP Web-Services using XWSS Library
200
</tns:placeOrderRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
==== Sending Message End ====

INFO: ==== Received Message Start ====
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
<tns:placeOrderResponse...">
<tns:refNumber>order-Packt-Pub-1234</tns:refNumber>
</tns:placeOrderResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
==== Received Message End ====
```

On execution, you can see the username and password is sent along with the SOAP request, and the response is printed in the console in a SOAP envelope.

Summary

In this chapter, we covered the basics of Spring Web Services package and the different types of SOAP web services creation. We have seen the Spring web service project creation and we also created the client project. We have executed and tested the authentication of SOAP message as well. In the next chapter, we will see how to handle the security for RESTful web services.

9
Building a Security Layer for RESTful Web Services

Representational State Transfer (**REST**) is an architectural style with which other web services can be designed. It serves the resources based on the request from the client. Web service is nothing but a unit of managed code that can be invoked using HTTP requests. We can develop the core functionality of any application and expose the same by deploying it in a server. The exposed web services can be accessed using URIs through HTTP requests from a wide range of client applications. With this method, duplication of business services for different implementations, such as desktop, mobile, and so on, will be avoided.

RESTful web services are quick in responding as there is no strict specification such as SOAP web services. REST requires much less bandwidth and resources, as it is a lightweight design. It is also language- and platform-independent. RESTful web services are applicable for any programming language and platforms. We can combine SOAP web services with RESTful as REST is only a concept and allows building other web services on top of it. Different data formats such as plain text, HTML, XML, and JSON RESTful are allowed in REST web services. Also, RESTful web services inherit security measures of the underlying transport protocol.

The Spring framework supports two ways of creating RESTful services using MVC with ModelAndView and HTTP message converters. The MVC approach is better; however, it is older, requires more verbose, and is heavyweight. Due to this approach, the entire application may become unmanageable. Therefore, from Spring 3.0 onwards, we have a robust annotation-based, less complex implementation of RESTful web services. In the new approach, the configuration is minimal and addresses most of the required defaults of the standard RESTful service.

The major difference between a traditional MVC Controller and RESTful web service controller is the way the HTTP response body is created. In RESTful, instead of depending on View Technique to perform server-side rendering of the response data, the controller simply returns a JSON format response object.

In this chapter, we will explore the following:

- Creating a RESTful web service
- Spring Security configurations
- Executing the project

Creating a RESTful web service

The following screenshot will explain the POM file creation. We need to add the following dependencies in the POM file in order to create a Spring RESTful web service security project:

- `spring-security-web`
- `spring-security-config`
- `spring-core`
- `spring-context`
- `spring-web`
- `spring-webmvc`

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>${org.springframework.security.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>${org.springframework.security.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-core</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-context</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-beans</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-tx</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-expression</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-web</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>          <artifactId>spring-webmvc</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>           <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>            <scope>provided</scope>
        </dependency>
```

The next step is to configure the XML files for Spring set up. The following is the screenshot of `web.xml` where `AnnotationConfigWebApplicationContext`, `ContextLoaderListener`, `DispatcherServlet`, and `DelegatingFilterProxy` are configured. Here, servlet-mapping and URL pattern are also specified:

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    xsi:schemaLocation="       http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">
    <display-name>Spring MVC Application</display-name>
    <session-config>
        <session-timeout>1</session-timeout>
    </session-config>
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
         org.springframework.web.context.support.AnnotationConfigWebApplicationContext
      </param-value>
    </context-param>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>spring.security.rest.packt</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>packt</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>packt</servlet-name><url-pattern>/packt/*</url-pattern>
    </servlet-mapping>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

We can configure the `Security Configurations` as shown in the following `webSecurityConfig.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:sec="http://www.springframework.org/schema/security"
    xsi:schemaLocation="
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security-3.2.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

    <http use-expressions="true" entry-point-ref="restAuthenticationEntryPoint">
        <intercept-url pattern="/packt/**" />
        <sec:form-login authentication-success-handler-ref="mySuccessHandler"
            authentication-failure-handler-ref="myFailureHandler" />
        <logout />
    </http>

    <beans:bean id="mySuccessHandler"
        class="spring.security.rest.packt.security.RestAuthenticationSuccessHandler" />

    <beans:bean id="myFailureHandler"
    class="org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler" />

    <authentication-manager alias="authenticationManager">
        <authentication-provider>
            <user-service>
                <user name="admin" password="admin" authorities="ROLE_ADMIN" />
                <user name="user" password="user" authorities="ROLE_USER" />
            </user-service>
        </authentication-provider>
    </authentication-manager>

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

</beans:beans>
```

The entry point references are given in this security `config` file, along with the interceptor URL pattern. We can also see the entries for `RestAuthenticationSuccessHandler` and `SimpleUrlAuthenticationFailureHandler`. Here, we have mentioned the credentials for two different roles: admin and user with different username and password combinations.

Spring Security configurations

We can start building the security implementations in the Spring classes in the `spring.security.rest.packt.security` package as follows:

```
1  package spring.security.rest.packt.security;
2
3  import java.io.IOException;
4
5  import javax.servlet.ServletException;
6  import javax.servlet.http.HttpServletRequest;
7  import javax.servlet.http.HttpServletResponse;
8
9  import org.springframework.security.core.AuthenticationException;
10 import org.springframework.security.web.AuthenticationEntryPoint;
11 import org.springframework.stereotype.Component;
12
13 @Component("restAuthenticationEntryPoint")
14 public class RestAuthenticationEntryPoint implements AuthenticationEntryPoint {
15
16     @Override
17     public void commence(HttpServletRequest arg0, HttpServletResponse arg1,
18             AuthenticationException arg2) throws IOException, ServletException {
19         arg1.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
20
21     }
22 }
```

`RestAuthenticationEntryPoint` will be invoked once the request is missing the authentication. The **Authentication Failed** response will be sent if the request doesn't have a valid cookie.

In the following `RestAuthenticationSuccessHandler`, we have extended `SimpleUrlAuthenticationSuccessHandler` and implemented `onAuthenticationSuccess` method. This `RestAuthenticationSuccessHandler` will be called once the request is authenticated. If not authorized, the authenticate entry point will be called:

```
package spring.security.rest.packt.security;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.Authentication;
import org.springframework.security.web.authentication.SimpleUrlAuthenticationSuccessHandler;
import org.springframework.security.web.savedrequest.HttpSessionRequestCache;
import org.springframework.security.web.savedrequest.RequestCache;
import org.springframework.security.web.savedrequest.SavedRequest;
import org.springframework.util.StringUtils;

public class RestAuthenticationSuccessHandler extends SimpleUrlAuthenticationSuccessHandler {

    private RequestCache requestCache = new HttpSessionRequestCache();

    @Override
    public void onAuthenticationSuccess(final HttpServletRequest request,
            final HttpServletResponse response, final Authentication authentication)
            throws ServletException, IOException {
        final SavedRequest savedRequest = requestCache.getRequest(request, response);

        if (savedRequest == null) {
            clearAuthenticationAttributes(request);
            return;
        }
        final String targetUrlParameter = getTargetUrlParameter();
        if (isAlwaysUseDefaultTargetUrl()
                || (targetUrlParameter != null && StringUtils.hasText(request
                        .getParameter(targetUrlParameter)))) {
            requestCache.removeRequest(request, response);
            clearAuthenticationAttributes(request);
            return;
        }
        clearAuthenticationAttributes(request);
    }
    public void setRequestCache(final RequestCache requestCache) {
        this.requestCache = requestCache;
    }
}
```

Now, we have to create the basic `spring.security.rest.api REST` package in order to build two Java config classes to mention the `webSecurityConfig.xml` classpath and `spring.security.rest.api.security ComponentScan` classes. The `SpringSecurityConfig` class will have the security settings, as shown in the following screenshot:

```
1  package spring.security.rest.packt;
2
3  import org.springframework.context.annotation.ComponentScan;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.context.annotation.ImportResource;
6
7
8  @Configuration
9  @ImportResource({ "classpath:webSecurityConfig.xml" })
10 @ComponentScan("spring.security.rest.packt.security")
11 public class SpringSecurityConfig {
12
13     public SpringSecurityConfig() {
14         super();
15     }
16 }
```

Also, the `WebConfig.java` class will have the `spring.security.rest.api.service` components scan declarations, which is an extension of the `WebMvcConfigurerAdapter` class:

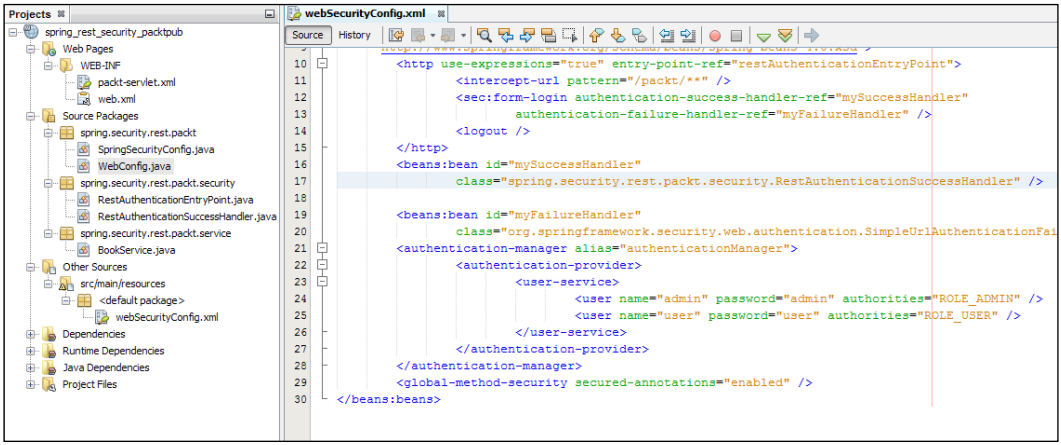
```
1  package spring.security.rest.packt;
2
3  import org.springframework.context.annotation.ComponentScan;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.web.servlet.config.annotation.EnableWebMvc;
6  import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
7
8
9  @Configuration
10 @ComponentScan("spring.security.rest.packt.service")
11 @EnableWebMvc
12 public class WebConfig extends WebMvcConfigurerAdapter {
13
14     public WebConfig() {
15         super();
16     }
17
18 }
```

The next step is to create the `BookService` RESTful web service as shown in the following screenshot. This has two RESTful calls: `userAccess` and `adminAccess`. This class uses Spring annotations to represent the controller and request mapping:

```
1  package spring.security.rest.packt.service;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.context.ApplicationEventPublisher;
5  import org.springframework.http.MediaType;
6  import org.springframework.security.access.annotation.Secured;
7  import org.springframework.stereotype.Controller;
8  import org.springframework.web.bind.annotation.RequestMapping;
9  import org.springframework.web.bind.annotation.RequestMethod;
10 import org.springframework.web.bind.annotation.ResponseBody;
11
12 @Controller
13 @RequestMapping(value = "/books")
14
15 public class BookService {
16
17     @Autowired
18     private ApplicationEventPublisher eventPublisher;
19
20     public BookService() {
21         super();
22     }
23
24     @RequestMapping(value = "/user", method = RequestMethod.GET, consumes =
25         {MediaType.APPLICATION_JSON_VALUE})
26     @ResponseBody
27     @Secured("ROLE_USER")
28
29     public String userAccess() {
30         return "<<<<<<<<<<USER>>>>>>>>>>";
31     }
32
33     @RequestMapping(value = "/admin", method = RequestMethod.GET, consumes =
34         {MediaType.APPLICATION_JSON_VALUE})
35     @ResponseBody
36     @Secured("ROLE_ADMIN")
37     public String adminAccess() {
38         return "<<<<<<<<<<ADMIN>>>>>>>>>>";
39     }
40 }
```

Executing the project

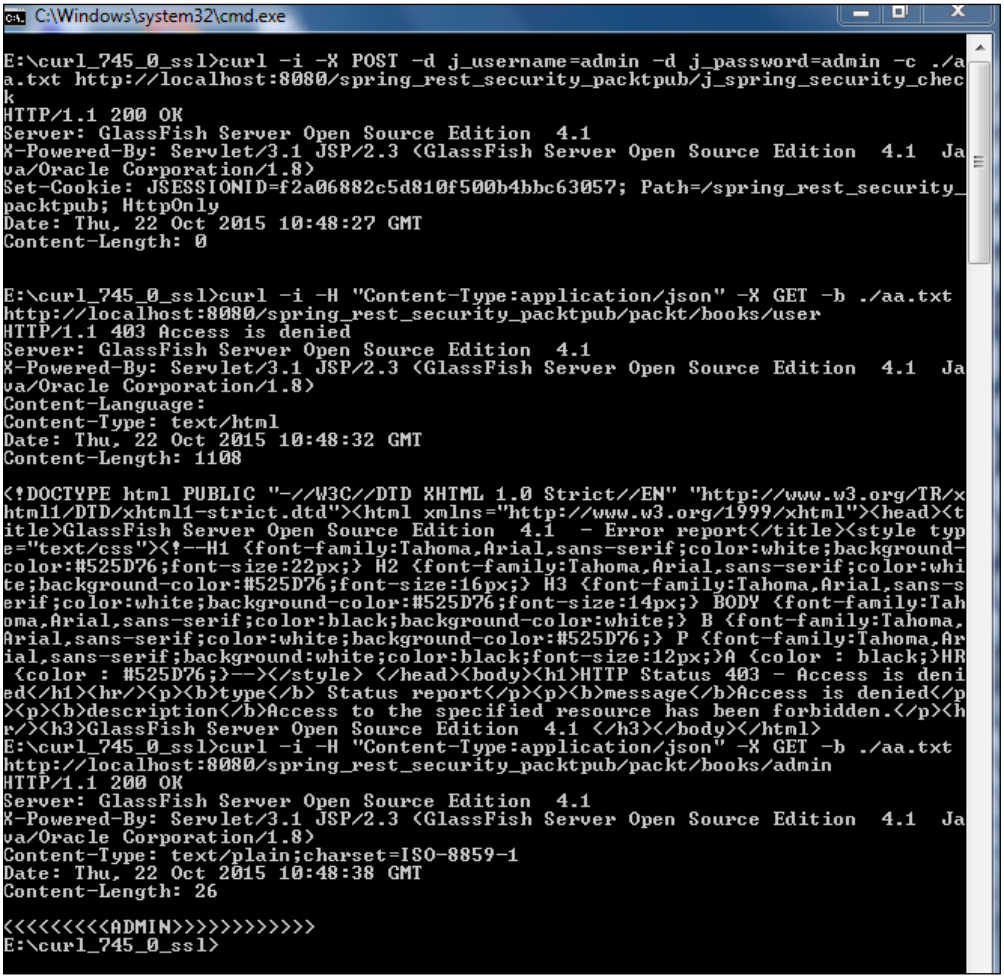
The final project structure will look as follows:



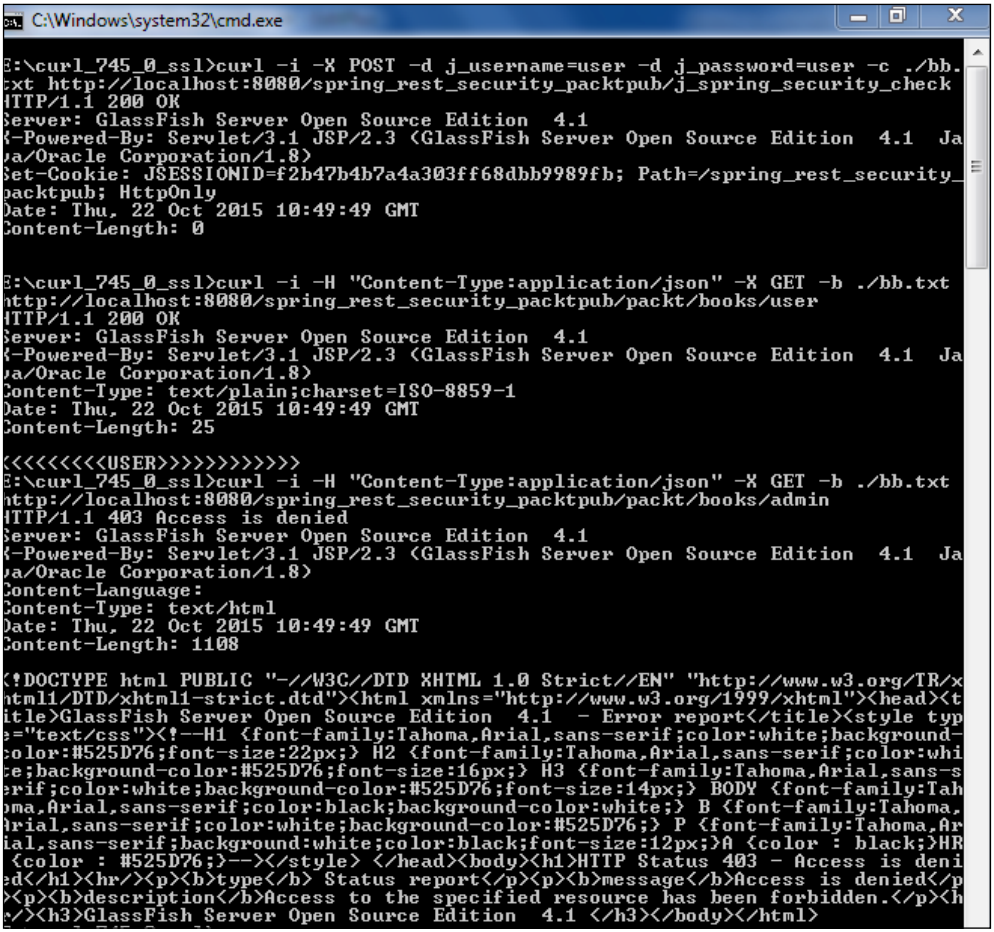
The following screenshot has the execution URLs that will be called through a tiny cURL command line executor. The `curl_745_0_ssl 2.47MB` can be downloaded from `http://curl.haxx.se/download.html`:

```
curl -i -X POST -d j_username=admin -d j_password=admin -c ./aa.txt
   http://localhost:8080/spring_rest_security_packtpub/j_spring_security_check
curl -i -H "Content-Type:application/json" -X GET -b ./aa.txt
   http://localhost:8080/spring_rest_security_packtpub/packt/books/user
curl -i -H "Content-Type:application/json" -X GET -b ./aa.txt
   http://localhost:8080/spring_rest_security_packtpub/packt/books/admin

curl -i -X POST -d j_username=user -d j_password=user -c ./bb.txt
   http://localhost:8080/spring_rest_security_packtpub/j_spring_security_check
curl -i -H "Content-Type:application/json" -X GET -b ./bb.txt
   http://localhost:8080/spring_rest_security_packtpub/packt/books/user
curl -i -H "Content-Type:application/json" -X GET -b ./bb.txt
   http://localhost:8080/spring_rest_security_packtpub/packt/books/admin

curl -i -X POST -d j_username=user -d j_password=wrongPass -c ./bb.txt
   http://localhost:8080/spring_rest_security_packtpub/j_spring_security_check
```

On executing the first three commands, you can see the authentication approval for admin and access granted only for the `adminAccess` method:



On executing the second set of commands, you can see the authentication approval for the user and access granted only for the `userAccess` method:



Summary

We have seen the basics of RESTful web services and their advantages. We have developed a basic Spring implementation to configure the security credentials, entry points, and success handlers. We also executed the RESTful web services through the cURL command line utility to check the Spring Security authentication in action. In the next chapter, we will study about the JAAS security aspects using Spring Integrations.

10
Integrating Spring Security with JAAS

Java Authentication and Authorization Service (**JAAS**) is the Java implementation that is based on the standard **Pluggable Authentication Module** (**PAM**) information security framework that is available as an extension library in Java 1.3. The aim of JAAS is to separate the user authentication layer from core applications so that the security-related features can be managed independently. JAAS is a combination of representation of identity called **principal** and a set of credentials called **subject**. The login service invokes the application callbacks to get the user inputs such as username and password. The login module of JAAS is primarily concerned with authentication and has methods such as `init`, `login`, `commit`, `abort`, and `logout`.

Spring Security provides a package that is able to delegate authentication requests to JAAS. Spring Security's authentication mechanism is responsible for populating the username and password that is taken from the user in the authentication object. As JAAS works with principals, the roles are also represented as a principal in JAAS. Each authentication object contains a single principal. To mediate between these different objects and value population, Spring has many interfaces.

In this chapter, we are going to cover the following:

- JAAS package basics
- Spring Security JAAS package components
- Spring JAAS configurations
- Spring JAAS implementation
- Executing the project

JAAS package basics

The basic components of JAAS are as follows:

- `javax.security.auth.spi.LoginModule`: This contains the actual code for authentication. Developers need to implement their own code in order to handle various mechanisms to authenticate user credentials.
- `javax.security.auth.login.LoginContext`: This is the core of the JAAS framework that kicks off the authentication process by creating a subject.
- `javax.security.auth.Subject`: This is the client that is requesting the authentication.
- `java.security.Principal`: This encapsulates features or properties of a client information.

Spring Security JAAS package components

Spring Security core package includes the following components to handle the JAAS implementation:

- `Authentication`: This is populated with the username and password.
- `AuthenticationProvider`: This creates `LoginContext` with a constructor that will have the callback handler information.
- `LoginContext`: This will be created by the provider. When the `login` method is called, this will invoke the `initialize` method that, in turn, creates new `JaasNameCallbackHandler` and `JaasPasswordCallbackHandler` for `JaasAuthenticationCallbackHandler`.
- `AuthorityGranter`: This returns the roles for the logged in username.
- `JaasAuthenticationToken`: This `Authentication` object will be created and returned.

Spring JAAS configurations

For the Spring configurations for JAAS implementation, we have to start from the POM file setting. Then, `web.xml` and `Servlet.xml` need to be configured. Finally, for the application context settings, we need to configure the application context XML file. Let's see these configuration settings one by one, as follows:

```
    <dependency>
        <artifactId>spring-asm</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-aop</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-beans</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-context</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-core</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-expression</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-orm</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-security-acl</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-security-config</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-security-core</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-security-crypto</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
        <artifactId>spring-security-taglibs</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
    <dependency>
            <artifactId>spring-security-web</artifactId>
        <version>3.1.0.RELEASE</version>    </dependency>
```

The preceding POM file has the required dependencies to create the basic JAAS-based Spring Security project. The Spring Security Core JAR file has all the JAAS components, as shown in the following:

```
<display-name>JAAS Sample Application</display-name>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<filter>
    <filter-name>localizationFilter</filter-name>
    <filter-class>org.springframework.web.filter.RequestContextFilter
    </filter-class>
</filter>
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>localizationFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<servlet>
    <servlet-name>springjaaspackt</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springjaaspackt</servlet-name>
    <url-pattern>/private/*</url-pattern>
</servlet-mapping>
</web-app>
```

The preceding `web.xml` file has configurations for the `springSecurityFilterChain`, `DispatcherServlet`, `ContextLoaderListener`, and `url` patterns for servlet mapping:

```
<sec:http auto-config="true" use-expressions="true">
<sec:intercept-url pattern="/private/admin/**" access="hasRole('ADMIN')" />
<sec:intercept-url pattern="/private/enduser/**" access="hasRole('ENDUSER')" />
<sec:form-login login-page="/login.jsp"
authentication-failure-url="/login.jsp?error=1" />
<sec:logout logout-success-url="/home.jsp"
logout-url="/j_spring_security_logout" />
</sec:http>
<sec:authentication-manager>
    <sec:authentication-provider ref="jaasAuthProvider" />
</sec:authentication-manager>
<bean id="jaasAuthProvider"
    class="org.springframework.security.authentication.
    jaas.DefaultJaasAuthenticationProvider">
    <property name="authorityGranters">
    <list>
        <bean class="com.packt.spring.jaas.security.RoleGranter" />
    </list>
    </property>
    <property name="configuration">
        <bean
            class="org.springframework.security.authentication.
            jaas.memory.InMemoryConfiguration">
            <constructor-arg>
            <map>
            <entry key="SPRINGSECURITY">
            <array>
            <bean class="javax.security.auth.login.AppConfigurationEntry">
            <constructor-arg value="com.packt.spring.jaas.security.Login" />
            <constructor-arg>
            <util:constant
            static-field="javax.security.auth.login.
            AppConfigurationEntry$LoginModuleControlFlag.REQUIRED" />
            </constructor-arg>
            <constructor-arg>
            <map></map>
```

The preceding `applicationContext.xml` file has all the entries for the JAAS security settings. The `http` security tag specifies the interceptor `url` patterns and access roles. The authentication manager is specified as `jaasAuthProvider` and you can see the configurations of `DefaultJaasAuthenticationProvider` and `AppConfigurationEntry`. Also, the `AuthorityGranter` implementation is configured as `authorityGranters`. The following screenshot shows the servlet configurations and basic package settings:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.packt.spring.jaas.security" />

    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix">
            <value>/WEB-INF/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
</beans>
```

Spring JAAS implementation

As the first step, we have to implement the `AuthorityGranter` interface as follows:

```
package com.packt.spring.jaas.security;

import java.security.Principal;
import java.util.Collections;
import java.util.Set;

import org.springframework.security.authentication.jaas.AuthorityGranter;

public class RoleGranter implements AuthorityGranter {
    public Set<String> grant(Principal principal) {
    if (principal.getName().equals("admin"))
            return Collections.singleton("ADMIN");
        else
            return Collections.singleton("ENDUSER");
    }
}
```

This screenshot shows the main Spring `SecureController` class that will return the success pages for two different roles called `admin` and `enduser`. You can see the implementations of `JaasGrantedAuthority` and `UserPrincipal`:

```
1  package com.packt.spring.jaas.security;
2  import org.springframework.stereotype.Controller;
3  import org.springframework.ui.ModelMap;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.security.authentication.jaas.JaasGrantedAuthority;
6  import org.springframework.security.core.Authentication;
7  import org.springframework.security.core.context.SecurityContextHolder;
8
9  @Controller
10
11 public class SecureController {
12
13     @RequestMapping(value="/admin/index")
14     public String getAdmin(ModelMap model) {
15         Authentication auth = SecurityContextHolder.
16             getContext().getAuthentication();
17         JaasGrantedAuthority jaasGrantedAuthority =
18             (JaasGrantedAuthority)(auth.getAuthorities().toArray()[0]);
19         UserPrincipal userPrincipal = (UserPrincipal)jaasGrantedAuthority.
20             getPrincipal();
21         userPrincipal.setRole(jaasGrantedAuthority.getAuthority());
22         model.addAttribute("userPrincipal", userPrincipal);
23         return "admin/index";
24     }
25
26     @RequestMapping(value="/enduser/index")
27     public String getEnduser(ModelMap model) {
28         Authentication auth = SecurityContextHolder.
29             getContext().getAuthentication();
30         JaasGrantedAuthority jaasGrantedAuthority = (JaasGrantedAuthority)
31             (auth.getAuthorities().toArray()[0]);
32         UserPrincipal userPrincipal = (UserPrincipal)jaasGrantedAuthority.
33             getPrincipal();
34         userPrincipal.setRole(jaasGrantedAuthority.getAuthority());
35         model.addAttribute("userPrincipal", userPrincipal);
36         return "enduser/index";
37     }
38 }
```

The following `Login` class implements the `LoginModule` interface and you can see the `login()` and `initialize()` methods being implemented:

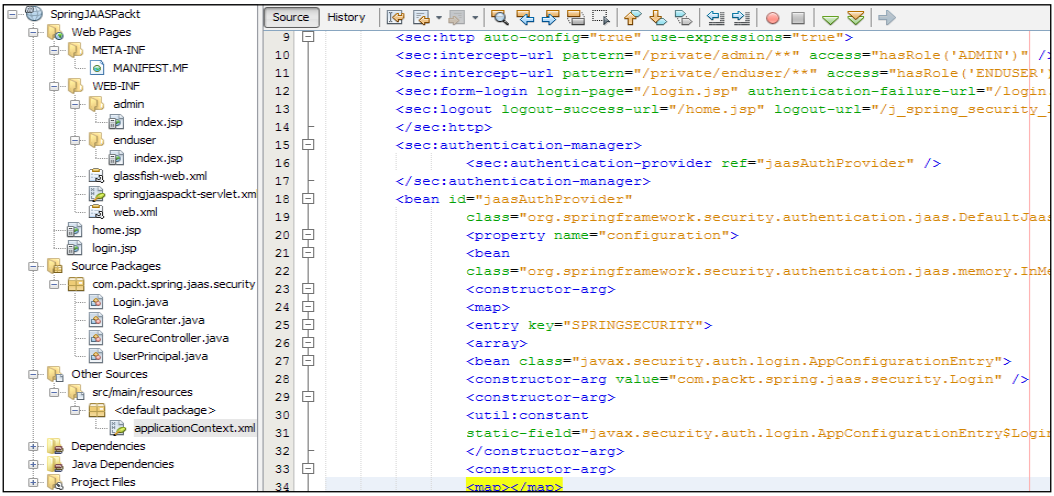
```
import javax.security.auth.*;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
public class Login implements LoginModule {
    private String password;
    private String username;
    private Subject subject;
    public boolean login() throws LoginException {
        if ((username.equals("admin") && password.equals("adminpass"))
            || (username.equals("enduser")&&password.equals("enduserpass"))) {
                subject.getPrincipals().add(new UserPrincipal(username));
        }
        return true;
    }
    public void initialize(Subject subject, CallbackHandler callbackHandler,
            Map<String, ?> state, Map<String, ?> options) {
        this.subject = subject;

        try {
            NameCallback nameCallback = new NameCallback("prompt");
            PasswordCallback passwordCallback = new PasswordCallback("prompt",
                    false);

            callbackHandler.handle(new Callback[] { nameCallback,
                    passwordCallback });

            password = new String(passwordCallback.getPassword());
            username = nameCallback.getName();

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Executing the project

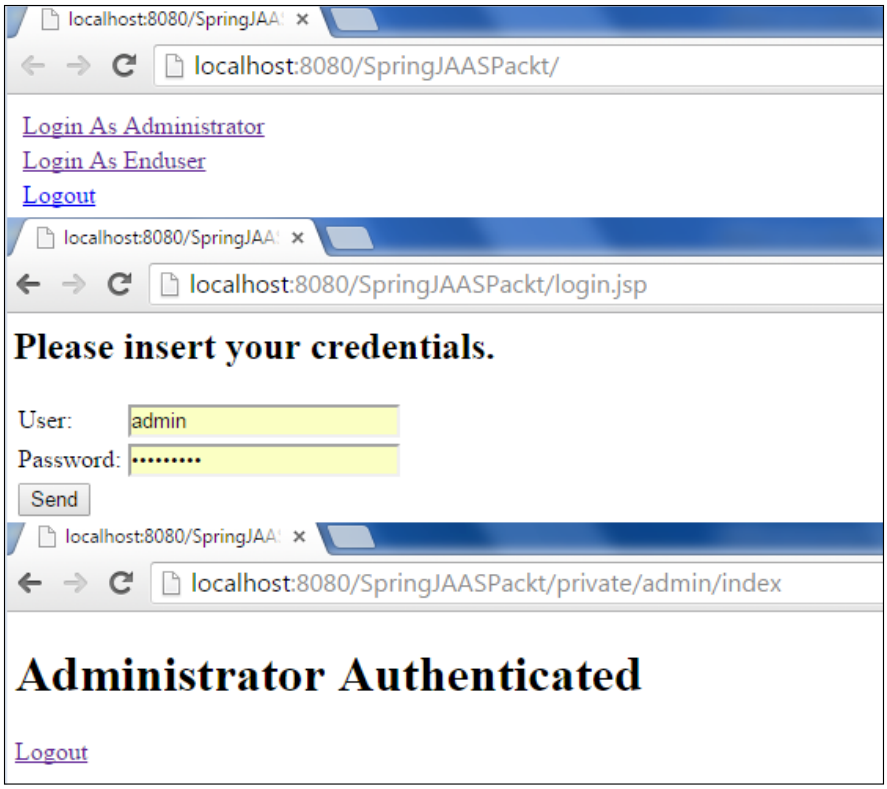
Let's create the following JSP pages: `home.jsp`, `login.jsp`, `admin/index.jsp`, and `enduser/index.jsp` as given in the following:

```
Home JSP Page
<table>
    <tr><td><a href="private/admin/index">Login As Administrator</a></td></tr>
    <tr><td><a href="private/enduser/index">Login As Enduser</a></td></tr>
    <tr><td><a href="<c:url value="/j_spring_security_logout" />" >
    Logout</a></td></tr>
</table>
Login JSP Page
<form name='f' action="<c:url value='j_spring_security_check' />" method='POST'>
<table>
    <tr>    <td>User:</td>
        <td><input type='text' name='j_username' value=''></td>
    </tr>
    <tr><td>Password:</td>
        <td><input type='password' name='j_password' /></td>
    </tr>
    <tr><td colspan='2'>
    <input name="submit" type="submit" value="Send" /></td></tr>
</table>
</form>
Administrator JSP Page
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
    <h1>Administrator Authenticated</h1>
    <a href="<c:url value="/j_spring_security_logout" />" > Logout</a>
Enduser JSP Page
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
    <h1>Enduser  Authenticated</h1>
    <a href="<c:url value="/j_spring_security_logout" />" > Logout</a>
```

The completed project structure is shown in the following screenshot:



On deployment and execution, you can see that the JAAS authentication is in action for different username and password combinations as specified in the `login` class: `admin/adminpass` and `enduser/enduserpass`, as shown in the following screenshot:



Summary

This is our last chapter in the *Spring Security Essentials* series and we covered the JAAS basic, Spring JAAS Security package components, and developing and executing a Spring JAAS implementation project.

I request and recommend the readers to try out the various combinations of Spring Security implementations in different layers of your real-time Spring applications using the working projects given throughout this book.

Index

Symbol

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

W

X