PAIRING APACHE SHIRO AND JAVA EE 7

Nebrass Lamouchi

InfoQueue
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

Table of Contents

Pairing Apache Shiro and Java EE 7 . . . 1
Dedication . . . 2
Acknowledgements . . . 3
Preface . . . 4
- What is in an InfoQ mini-book? . . . 4
- Who this book is for . . . 4
- What you need for this book . . . 4
- Conventions . . . 4
- Reader feedback . . . 6

Introduction . . . 7
- Personal case . . . 7
- Professional experience . . . 8
- Motivation for writing this tutorial . . . 8

The Shiro Philosophy . . . 9
- What is Shiro? . . . 10
- Plan of the castle . . . 10
- Why not JAAS or Spring Security ? . . . 11

Sample Technology Stack . . . 12
- Technologies . . . 13
- Apache Shiro . . . 13
- Java EE 7 . . . 13
- Payara Server . . . 14
- NetBeans IDE . . . 15

The Tutorial . . . 16
- Step 1: The project . . . 17
- Step 2: JPA entities . . . 22
- Step 3: Apache Shiro prime view . . . 27
- Step 4: Shiro: Getting serious . . . 29
- Step 5: Exposing Shiro operations as REST services . . . 47

What's Next? . . . 67
- How to consume Shiro's web services . . . 68
- What can you add to the implementation? . . . 68
- Recommendations . . . 69

Do It Now! . . . 70
- Additional reading . . . 70

The End ; . . . 71

Pairing Apache Shiro and Java EE 7

Published by C4Media, publisher of InfoQ.com.

Production Editor: Ana Ciobotaru
Copy Editor: Lawrence Nyveen
Cover and Interior Design: Dragos Balasoiu

Library of Congress Cataloguing-in-Publication Data: ISBN: 978-1-365-12404-4

Dedication

To Mom, it’s impossible to thank you adequately for everything you’ve done.

To the soul of Dad, I miss you every day…

To Firass, you are the best, may God bless you…

To my sweetheart, since you’ve come into my life, there are so many new emotions and feelings begging to come out..

Acknowledgements

I would like to express my gratitude to the many people who saw me through this book. To all those who provided support, read, wrote, offered comments and assisted in the editing and design.

To my friends thank you for always being there when I need you.

To Jérome Salles, thank you for teaching me the Java EE Art and for your support and guidance.

To Paul Bernardi and Geertjan Wielenga, thank you for believing in my skills and for the chances that you are offering to me.

To Victor Grazi and Laurie Nyveen, the great technical reviewers who worked on this book. Thank you for the great job that you did.

To Ana Ciobotaru and Charles Humble, thank you for making the idea of this book come true. Thank you for all the help & support that you have provided me.

Last and not least, I beg forgiveness of all those who have been with me over the course of the years and whose names I have failed to mention.

Preface

Over the last few years, I've been working on many projects and companies that used Java to develop their applications. On those applications, I tried to implement the authentication & authorization modules and sometimes using the Spring Security or JBoss Picketlink.

My M.Sc studies of the IT Security and my experience with the OWASP Foundation gave me more motivation to explore the security frameworks such as JAAS, JASPIC, Spring Security and Picketlink.

When I heard about Apache Shiro, I was motivated to use it right away. It combined my most-often-used frameworks into an easy-to-use package. I wanted to write this book because I saw that it will be very helpful for junior Java developers to find some quick introduction for starting securing their apps. I wanted to further the knowledge of this wonderful project. I wanted to show them how security isn't scary, it's just another framework that can improve your Application Security eXperience.

What is in an InfoQ mini-book?

InfoQ mini-books are designed to be concise, intending to serve technical architects looking to get a firm conceptual understanding of a new technology or technique in a quick yet in-depth fashion. You can think of these books as covering a topic strategically or essentially. After reading a mini-book, the reader should have a fundamental understanding of a technology, including when and where to apply it, how it relates to other technologies, and an overall feeling that they have assimilated the combined knowledge of other professionals who have already figured out what this technology is about. The reader will then be able to make intelligent decisions about the technology once their projects require it, and can delve into sources of more detailed information (such as larger books or tutorials) at that time.

Who this book is for

This book is aimed specifically at Java developers who want a rapid introduction to Apache Shiro by learning how to couple it with the Java EE 7.

What you need for this book

To try code samples in this book, you will need a computer running an up-to-date operating system (Windows, Linux, or Mac OS X). You will need Node.js and Java installed. The book code was tested against Node.js v0.12 and JDK 8, but newer versions should also work.

Conventions

We use a number of typographical conventions within this book that distinguish between different kinds of information.

Code in the text, including commands, variables, file names, XML configuration files, and property names are shown as follows: "The ShiroFilterActivator extends the `ShiroFilter` class. Its annotation `@WebFilter("/*")` enables the filter for all the requests."

A block of code is set out as follows. It may be colored, depending on the format in which you're reading this book.

src/main/webapp/WEB-INF/beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                      http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
        bean-discovery-mode="all">
</beans>
```

src/main/java/demo/DemoApplication.java

```
@ApplicationPath("api")
public class JaxRsConfiguration extends Application {
    // JAX-RS Configuration Class
}
```

When we want to draw your attention to a particular part of the code, it's called out with numbers.

```
private String getCredentials(AuthenticationInfo info) { ①

    Object credentials = info.getCredentials(); ②
    return toString(credentials); ③
}
```

① We are passing the fundamental AuthenticationInfo object to the `getCredentials()` method.

② `info.getCredentials()` is a function called in the `getCredentials()` method, used to grab the `credentials` object from the `AuthenticationInfo` instance.

③ The textual format of the `credentials` instance is returned to the `getCredentials()` caller.

Tips are shown using callouts like this.

Warnings are shown using callouts like this.

Sidebar

Additional information about some topics may be displayed using a sidebar like this.

Finally, this text shows what a quote looks like.

> In the end, it’s not the years in your life that count. It’s the life in your years.
>
> — Abraham Lincoln

Reader feedback

We always welcome feedback from our readers. Let us know what you think about this book — what you liked or disliked. Reader feedback helps us develop titles that you get the most out of. To send us feedback, e-mail us at feedback@infoq.com.

If you have a topic that you have expertise in and you are interested in either writing or contributing to a book, please take a look at our mini-book guidelines at http://www.infoq.com/minibook-guidelines.

Introduction

Security is essential for every enterprise application. In many cases, securing the application costs more than any other part of the project.

To be able to protect the application functionalities and the data stored, the team has to divide tasks to fulfil the security duties. Developers have to write secure code and implement cryptographic layers, authentication and authorization mechanisms, and session-management strategies. Sysadmins have to implement firewalls, VPNs, and anti-virus, anti-adware, and all other anti-* solutions.

> If you know yourself but not your enemy, for every victory gained you will also suffer a defeat.
>
> — Sun Tzu, The Art of War

Every competent Java EE developer knows that you must have security APIs in the platform, either natively or offered by third-party frameworks. But since the birth of the Java EE platform on December 12, 1999, Java EE has lacked a full, dedicated security specification. JSR-375 has been created to provide a complete specification for Java EE security, one that covers all needs, but will only be available in the Java EE 8 release, estimated to arrive in 2017.

One of the most desired of security features are authentication and authorization. I will refer to authentication and authorization in this article as "Auth²". The platform hasn't provided a useful solution for these despite attempts like JAAS (2001-2003) and JASPIC (2009) to allow Java EE folks to do some "standardized" Auth².

As usual, when the community demands a feature that is unfortunately not provided by the platform, third parties try to satisfy the need. The most popular stars for Auth² in Java EE are Spring Security, JBoss's PicketLink, and Apache Shiro.

Many developers implement their own Auth² mechanisms rather than taking the time to understand and implement these weird third-party solutions. I write "weird" because every third-party solution is tightly linked to its creator, such as Spring Security and JBoss PicketLink or poorly documented and ambiguous, which is the case for Apache Shiro.

Personal case

I decided to share my personal experience with Auth² frameworks. I am trying to build my own up-to-date security framework for Java EE 7. My implementation will be handy until the release of a reference implementation for JSR-375. I will try to explain every choice made and I hope this it is useful to you.

I have tried many APIs but every one disappointed me. Either the solution was hard to implement (implementing Spring Security with EJB, using JAAS and JASPIC is not exactly terrific) or the solution

was vendor specific (PicketLink is usable only in a JBoss server) or the API was poorly documented (Apache Shiro).

I discovered Apache Shiro in December 2012. At that time, it wasn't popular. It was hard to use and suffered from poor documentation and references. I tried to use it but ended up dropping it.

Professional experience

Since starting my career as a Java developer, every client has asked me to help implementing their own Auth² solution, except for one who asked me to implement Spring Security on his web app. Every time I work on the security level, I keep asking myself "Why do I have to do this again? Why there is no efficient solution? Where is the reusability? Why I am repeating myself?"

The last time I implemented an Auth² mechanism, I used older Java EE 6 technologies: JSP 2.5, Servlets 3.0, JMS 1.1, EJB 3.1, and JPA 2.0. It was an amazing experience to implement the Open Web Application Security Project (OWASP) specifications using Java EE APIs.

Motivation for writing this tutorial

After finishing that last project, I decided to revisit Apache Shiro, that mysterious framework that I could not use three years earlier.

Oddly, I found that the Shiro website pages had not changed over the years. There were no new sections, no new tutorials, and no new documentation. Obviously, I looked on YouTube, Vimeo, and Dailymotion for Shiro video tutorials, with no success. I looked for any relevant books or articles, with no significant result. This was so strange for an Apache project. Usually, any Apache project rapidly gets well covered. But this wasn't the case for Shiro.

I tried to use Shiro in a sample project, and here's my initial feedback:

- There is the use for an INI file, which differs from Spring Security style, which uses XML files for configuration.
- There is not enough official documentation and resources. For example, the official tutorial on the website focuses only on Stormpath.
- The JavaDoc website of the latest Shiro version is almost always inaccessible (I reported that to the Apache Shiro Twitter account).

I decided to write this tutorial to help fill in the gap.

PART ONE

The Shiro Philosophy

What is Shiro?

Apache Shiro is an open-source software-security framework that performs authentication, authorization, cryptography, and session management. Originally known as JSecurity, Shiro has been designed to provide robust security features with intuitive and easy implementation.

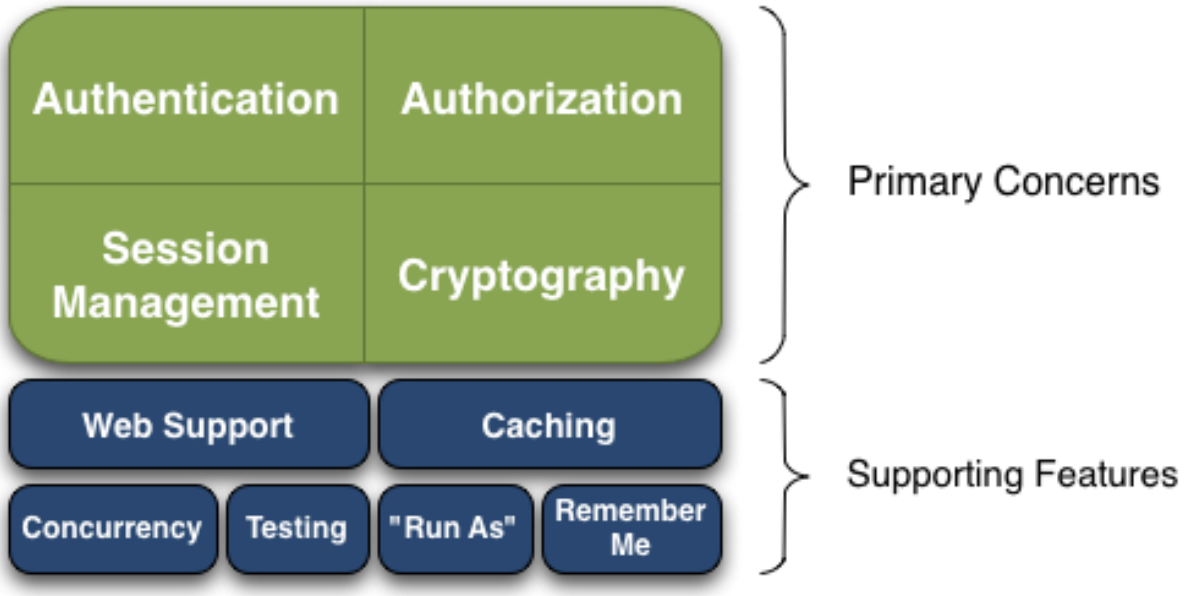
The tutorial detailed in this book is targeted at Java developers seeking an Auth² framework for use on contemporary technology stacks.

The project got its name ("shiro" is Japanese for castle") in 2008 when it was pushed to the Apache Community. The "castle" mandate aims to provide a standardized framework that satisfies the full gamut of security considerations for any Java project.

Plan of the castle

Apache Shiro covers the main four concerns of application security:

- Authentication, sometimes called "login", is the act of proving that a user is who they say they are.
- Authorization is the process of access control, i.e. determining who has access to what.
- Session management is managing user-specific sessions, even in non-web or Enterprise JavaBeans (EJB) applications.
- Cryptography is keeping data secure using cryptographic algorithms while still maintaining ease of use.



Supported environments include:

- Web support — Shiro's web support APIs help easily secure web applications.
- Caching — caching is a first-class citizen in Apache Shiro's API and ensures that security operations

remain fast and efficient.

- Concurrency — Apache Shiro concurrency features are designed to support multi-threaded applications.
- Testing — testing support simplifies the creation of unit and integration tests and ensures that your code will be secured as expected.
- "Run As" — Shiro allows users to assume the identity of another user (if entitled), a feature that can be useful in administrative scenarios.
- "Remember Me" — Shiro remembers users' identities across sessions so they only need to log in when mandatory.

Why not JAAS or Spring Security ?

Existing Java security mechanisms (like JAAS) are confusing and fall short in the area of application-level security. Some, for example PicketLink, are server-dependent. Others, such as Spring Security, require tedious XML configuration.

Apache Shiro aims to address these challenges by providing a user-friendly, standardized API for implementing security in your projects.

PART TWO

Sample Technology Stack

Technologies

This tutorial uses:

- Apache Shiro 1.2.4
- Java EE 7 back end with JPA 2.1, EJB 3.2, and JAX-RS 2.0
- Payara Server
- NetBeans 8.1 IDE

Apache Shiro

Apache Shiro is a powerful, easy-to-use Java security framework that allows your applications to perform authentication, authorization, cryptography, and session management. You can use it to secure any kind of application, from command-line scripts to mobile apps to the largest web and enterprise applications.

The framework landscape has changed quite a bit since JSecurity, Shiro's predecessor, was released in 2004, but there are still compelling reasons to use Shiro today. It's easy to use, comprehensive, web-capable, pluggable, and well supported.

For more information or for official documentation, see the Apache Shiro website. And be sure to follow the offical Twitter account.

Java EE 7

Java Platform Enterprise Edition (Java EE) is the standard in community-driven enterprise software. Java EE is ever expanding via the Java Community Process, with contributions from industry experts, commercial and open-source organizations, Java user groups, and countless individuals. Each release integrates new features that align with industry requirements, increases developer productivity, and improves application portability.

The latest version of Java EE is Java EE 7, released in June 2013. The Java EE 7 platform added first-class support for recent developments in web standards, including WebSockets and JSON, which provide the underpinnings for HTML 5 support in Java EE. Java EE 7 also adds a modern HTTP client API as defined by JAX-RS 2.0.

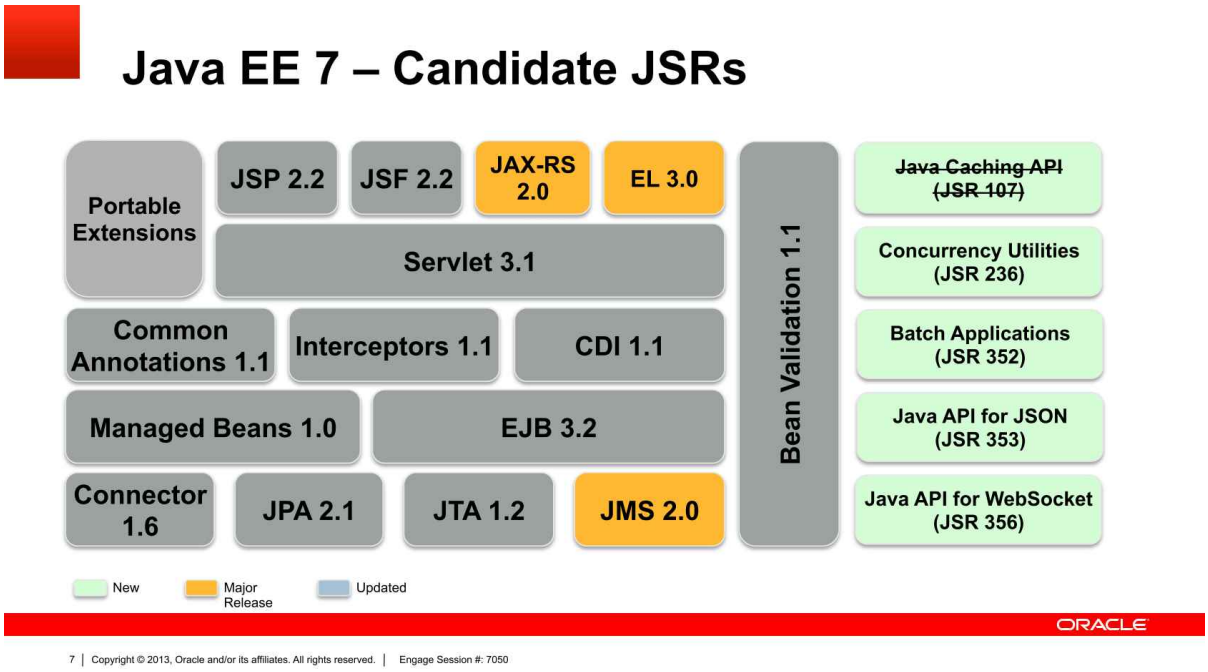
The Java EE 7 specifications include:

- Java API for WebSockets — JSR-356

- Java API for JSON Processing — JSR-353
- Batch Applications for Java Platform — JSR-352
- Concurrency Utilities for Java EE — JSR-236
- Java API for RESTful Web Services — JSR-311
- Java Message Service 2.0 — JSR-343
- Expression Language 3.0 — JSR-341
- Enterprise JavaBeans 3.2 — JSR-345
- Java Persistence 2.1 — JSR-338
- JavaServer Faces 2.2 — JSR-344

You can find all necessary information about the Java EE 7 specifications on the umbrella JSR-342 page.

Java EE 7 specifications.



Payara Server

Payara Server is a drop-in replacement for the GlassFish server with guaranteed quarterly releases. The Payara team provides enhancements, bug fixes, and patches to upstream GlassFish server and dependent libraries. This is what makes Payara Server one of the best application servers for production in the Java EE ecosystem. The Payara team also provides 24/7 dedicated incident and software support delivered by the best middleware engineers in the industry. The team is open to contributions and publicly reports its progress in writing features or fixing bugs. There also a responsive community that reacts quickly to any request. Payara Server tweaks GlassFish but keeps its ease of use and its beautiful administration features.

You can grab the latest version of Payara Server from the project website.

NetBeans IDE

NetBeans IDE is the free Java IDE from Oracle, and is designed to limit coding errors and optimize productivity without configuration or plugins. NetBeans can interact with all of your environment components such as databases, automation servers, application servers, quality tools, etc. without requiring a lot of your time to prepare the environment.

NetBeans IDE has awesome debugger functionalities to optimize the refactoring and debugging processes. It includes a powerful text editor with refactoring tools and code completion, snippets and analyzers, and even a multilingual spelling checker. NetBeans also offers powerful multisystem versioning using out-of-the-box integration with tools such as Git, SVN, CVS, and Mercurial. Beyond Java and JavaFX, NetBeans IDE supports C/C++, PHP, and HTML5/JavaScript. NetBeans IDE is available for all operating systems that support a compatible JVM.

NetBeans organizes many free conferences and meetings all over the world, in the US, France, UK, Netherlands, Brazil, and more countries. Attending these events will let you acquire skills not only with NetBeans but for general programming, too.

You can download the latest version of NetBeans IDE at the NetBeans website.

PART THREE

The Tutorial

The remainder of this book is a tutorial presented in steps designed to let you easily follow along.

First, install the tutorial tool set so that you can follow the screenshots:

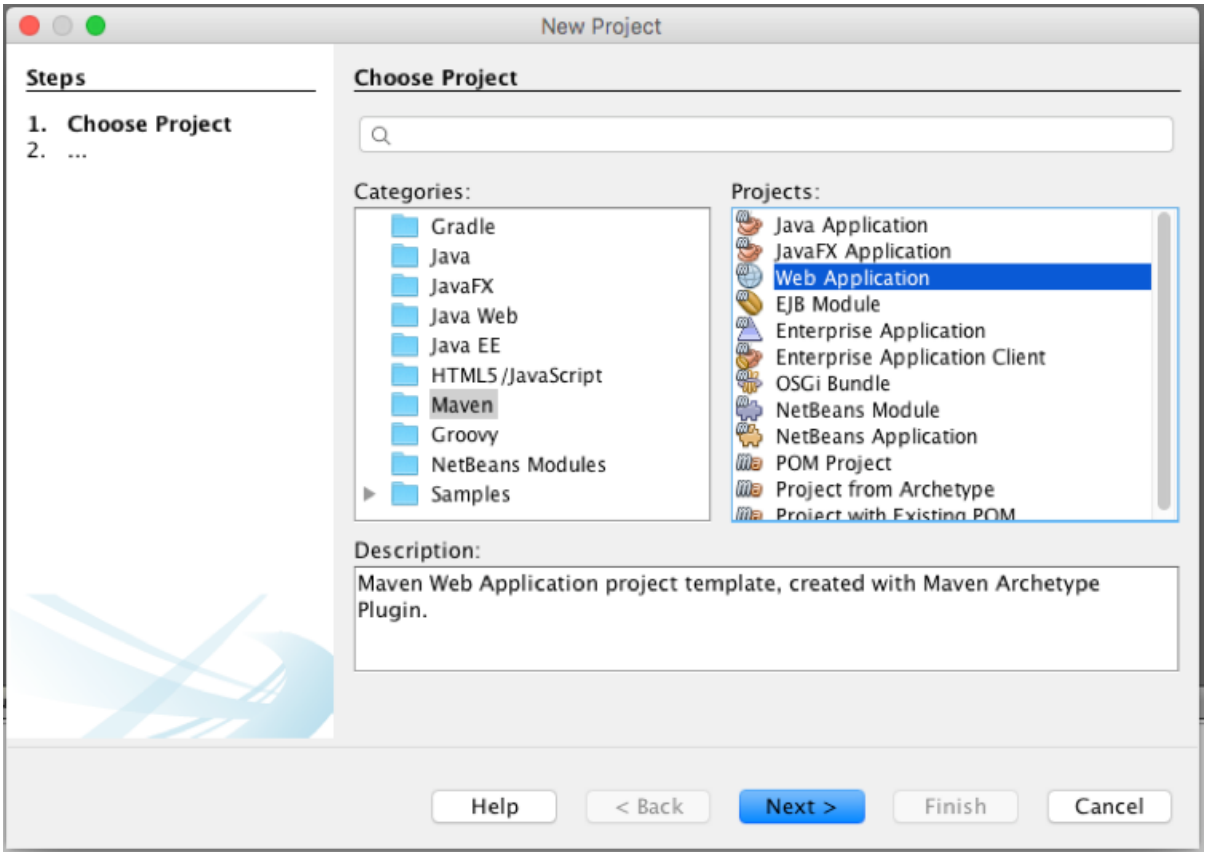
- JDK 8
- Maven 3
- a relational database such as MySQL or PostgreSQL (recommended)
- Payara Server (recommended), WildFly, or any Java EE 7 server
- an IDE, and I prefer NetBeans
- lots of hot coffee!

Before starting the project, configure your system. Install the JDK, database, etc., and create and configure the project environment and IDE

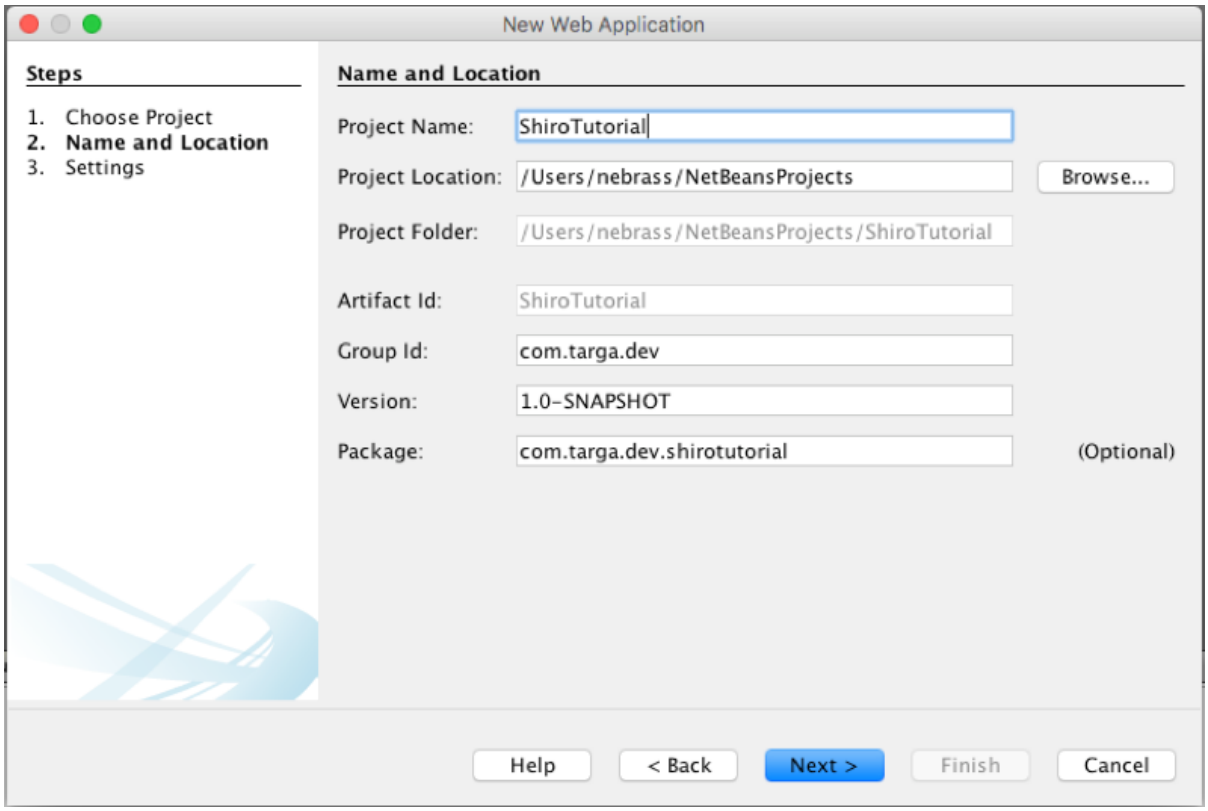
Step 1: The project

Step 1.1: Creating the project

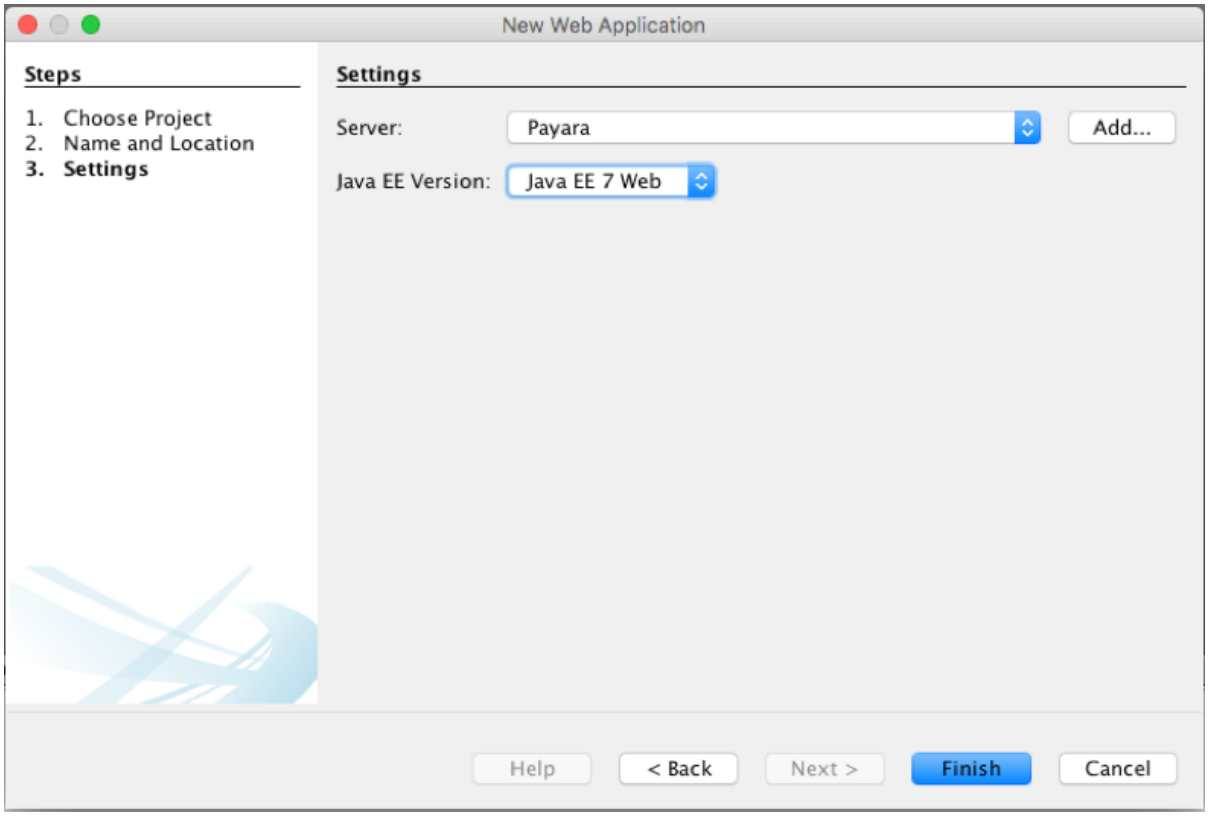
Start the tutorial by creating a new Maven project — a web application.



Next, complete the Name and Location pane.



And next, fill in the server and Java EE settings.



After creating the project, go to the pom.xml file and edit it.

```
<?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/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.targa.dev</groupId>
    <artifactId>ShiroTutorial</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>ShiroTutorial</name>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <failOnMissingWebXml>false</failOnMissingWebXml>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <finalName>ShiroTutorial</finalName>
    </build>

</project>
```

We need only these lines. The thinner your pom.xml is, the lighter your WAR file will be.

Step 1.2: Designing the sample application

Step 1.2.1: The BCE/ECB pattern

We will be using the entity-control-boundary (ECB) pattern, a variation of the ubiquitous model-view-

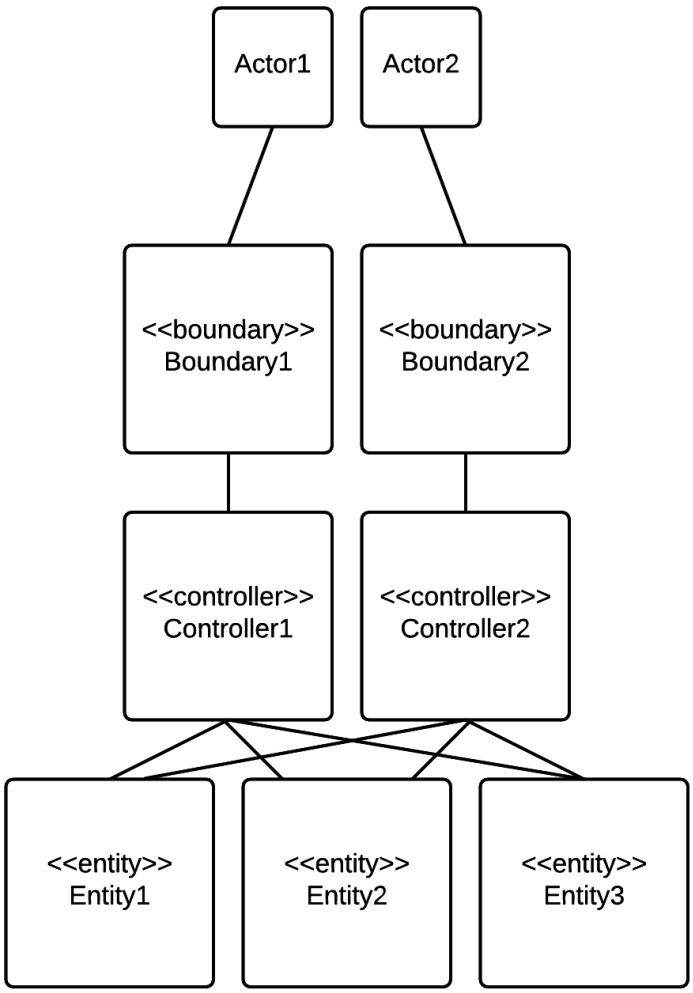
controller (MVC) pattern. When identifying the elements for an execution sequence in your system, you can classify each participating element into one of three categories: entity, control, or boundary.

An entity is a long-lived, passive element that is responsible for some meaningful chunk of information. This is not to say that entities are "data" while other design elements are "function", just that entities perform behavior that is organized around some cohesive sets of data.

A control element manages the flow of interaction within a scenario. A control element can manage the end-to-end behavior of a scenario or it can manage the interactions between a subset of the elements. You should classify as entities those behavior and business rules relevant to the scenario that relate to the information; control elements are responsible only for the flow of the scenario.

A boundary element lies just within the periphery of a system or subsystem. For any scenario being considered either across the whole system or within some subsystem, some boundary elements will be front-end elements that accept input from outside the area under design and other elements will be back end, managing communication to supporting elements outside the system or subsystem.

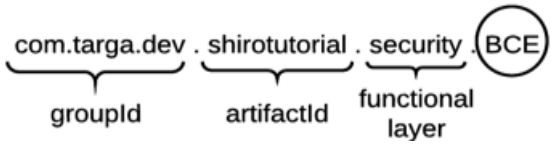
Boundary-control-entity pattern.



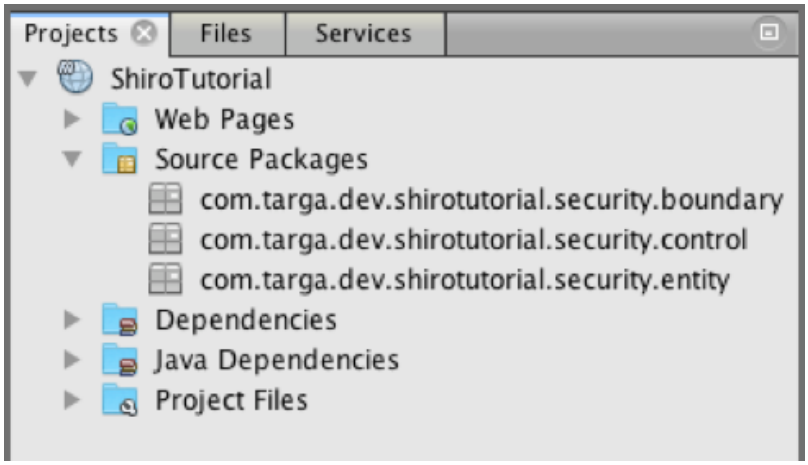
Step 1.2.2: BCE implementation

Create some empty Java packages just to highlight the BCE-pattern implementation in your application. The packages will take this format:

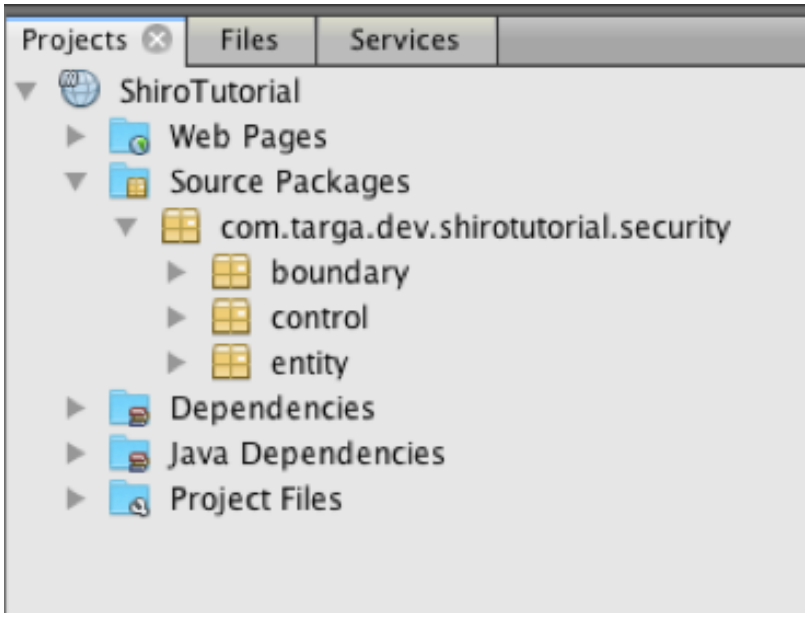
BCE-pattern package structure.



Let's call the functional layer for the authentication and authorization needs "security". This is the output:



In NetBeans, you can simplify the view of the packages by choosing "Reduced Tree" view for the packages. On the project view, right-click → View Java Packages as → Reduced Tree.



Now start making your entities.

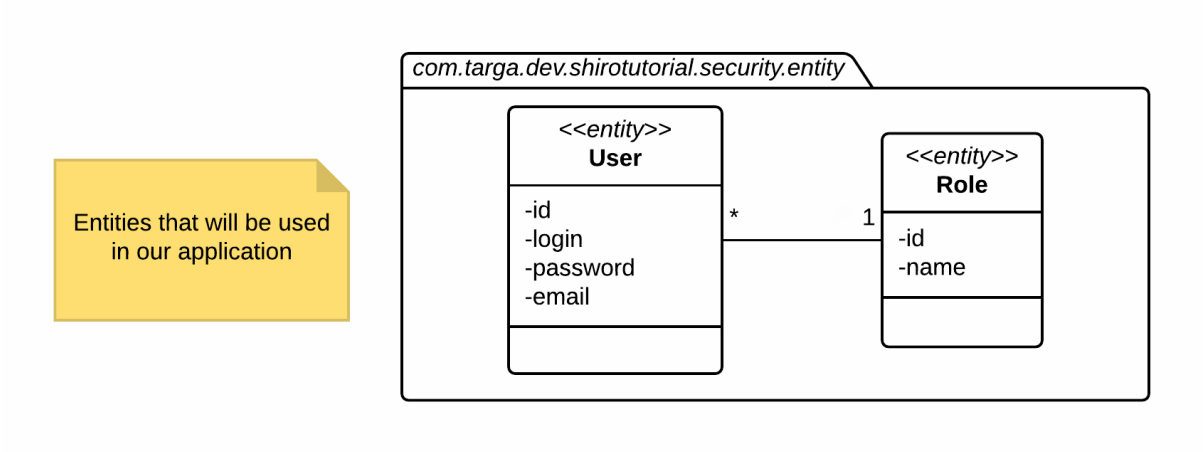
Step 2: JPA entities

Use JPA 2.1 for your persistent entities. Create two entities in the **com.targa.dev.shirotutorial.security.entity** package.

For our Auth² implementation, we will have two entities:

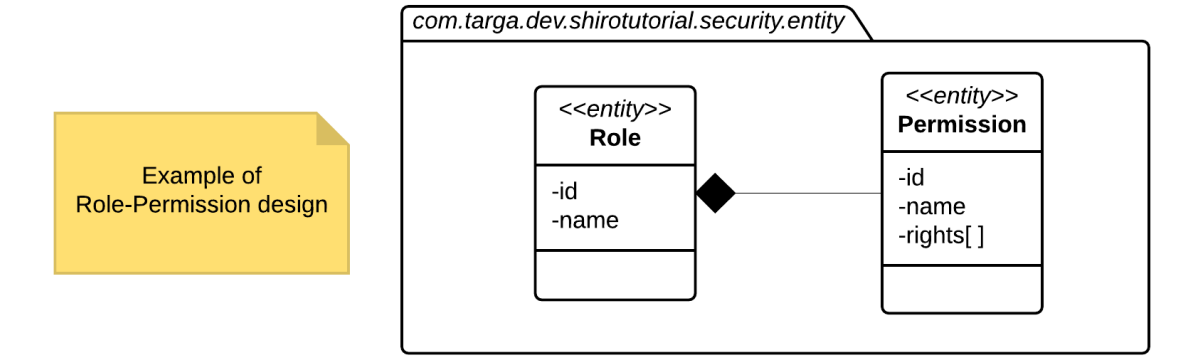
User, which refers to the client of the application, and **Role**, which refers to the authorization level of the user.

Class diagram of **User** *and* **Role** *entities.*



You can extend the "roles and rights" strategy (for permission-based authorization) by adding a **Permission** entity that contains the rights attributed for the associated **Role** — for example, "Read only" or "Control". The resulting class diagram is:

Class diagram of **Role** *and* **Permission** *entities.*



User:

```
@Entity ①
@NamedQueries({ ②
    @NamedQuery(name = "User.findAll", query = "select u from User u"),
    @NamedQuery(name = "User.findByUsername", query = "SELECT u FROM User u WHERE
u.username = :username") ③
})
public class User {
    private static final long serialVersionUID = 1L;
    @Id ④
    @GeneratedValue ⑤
    @Column(name = "id", nullable = false, updatable = false) ⑥
    private long id;
    @NotNull ⑦
    @Column(name = "username", length = 50, nullable = false)
    private String username;
    @NotNull
    @Column(name = "password", nullable = false)
    private String password;
    @OneToOne(cascade = CascadeType.PERSIST) ⑧
    private Role role;
    @Column(name = "enabled", nullable = false)
    private Boolean enabled;
    @Version ⑨
    @Column(name = "version", nullable = false)
    private Timestamp version; ⑩

//Getters & setters
}
```

① Declares the annotated class as a JPA entity.

② Declares a list of **NamedQuery** elements.

③ The minimal **NamedQuery** declaration = query name + JPQL query.

④ Defines the attribute as the entity's **id** field.

⑤ Used on the entity's **id** field to specify the **id** generation strategy.

⑥ Specifies the definition of the corresponding table's column.

⑦ Defines a validation contraint for the corresponding field value → username is not null.

⑧ Defines the relation between the entities.

⑨ Specifies the version field of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control.

⑩ The **Timestamp** is the version property type.

Role:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Role.findAll", query = "SELECT r FROM Role r")
})
public class Role {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue
    @Column(name = "id", nullable = false, updatable = false)
    private long id;
    @NotNull
    @Column(name = "name", length = 50, nullable = false)
    private String name;
    @NotNull
    @Past ①
    @Column(name = "creation", nullable = false, updatable = false)
    @Temporal(TemporalType.TIMESTAMP) ②
    private Date creation;
    @Column(name = "enabled", nullable = false)
    private Boolean enabled;

//Getters & setters
}
```

① Creation date must occur in the past.

② Indicates this is a date or time column.

Next, create a service for every entity. Your services will be created in the **com.targa.dev.shirotutorial.security.control** package. The **EntityManager** is an object that packages the CRUD operations needed to handle the entity:

- Save/Edit
- Delete
- Find

UserService:

```
@Stateless
public class UserService {
    @PersistenceContext
    EntityManager em;

    public User save(User entity) {
        return this.em.merge(entity);
    }

    public void delete(long id) {
        try {
            User reference = this.em.getReference(User.class, id);
            this.em.remove(reference);
        } catch (EntityNotFoundException e) {
            //It doesn't exist already
        }
    }

    public User findById(long id) {
        return this.em.find(User.class, id);
    }

    public List<User> findAll() {
        return this.em.createNamedQuery("User.findAll", User.class)
                .getResultList();
    }
}
```

RoleService:

```
@Stateless
public class RoleService {
    @PersistenceContext
    EntityManager em;

    public Role save(Role entity) {
        return this.em.merge(entity);
    }

    public void delete(long id) {
        try {
            Role reference = this.em.getReference(Role.class, id);
            this.em.remove(reference);
        } catch (EntityNotFoundException e) {
            // It doesn't exist already
        }
    }

    public Role findById(long id) {
        return this.em.find(Role.class, id);
    }

    public List<Role> findAll() {
        return this.em.createNamedQuery("Role.findAll", Role.class)
                .getResultList();
    }
}
```

Your services are now ready to use. For simpler and more flexible context dependency injection (CDI), create a beans.xml file, and change the **bean-discovery-mode** from **annotated** to **all**.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
    bean-discovery-mode="all">

</beans>
```

This recommended step lets you inject **all** the components, not only the **annotated** object.

Step 3: Apache Shiro prime view

First, add the Apache Shiro dependency to your project pom:

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-web</artifactId>
    <version>1.2.4</version>
</dependency>
```

Apache Shiro uses an INI configuration file to define parameters. Here is an example of a shiro.ini file:

```
[main]
authc.loginUrl = /login.jsp ①
authc.successUrl = /home.jsp

passwordMatcher = org.apache.shiro.authc.credential.TempFixPasswordMatcher ②
passwordService = org.apache.shiro.authc.credential.DefaultPasswordService
passwordMatcher.passwordService = $passwordService

ds = com.jolbox.bonecp.BoneCPDataSource ③
ds.driverClass=com.mysql.jdbc.Driver
ds.jdbcUrl=jdbc:mysql://localhost:3306/simple_shiro_web_app
ds.username = root
ds.password = 123qwe

jdbcRealm = org.apache.shiro.realm.jdbc.JdbcRealm ④
jdbcRealm.permissionsLookupEnabled = true
jdbcRealm.authenticationQuery = SELECT password FROM USERS WHERE username = ?
jdbcRealm.userRolesQuery = SELECT role_name FROM USERS_ROLES WHERE username = ?
jdbcRealm.permissionsQuery = SELECT permission_name FROM ROLES_PERMISSIONS WHERE
role_name = ?
jdbcRealm.credentialsMatcher = $passwordMatcher
jdbcRealm.dataSource=$ds

securityManager.realms = $jdbcRealm ⑤

[urls] ⑥
# The /login.jsp is not restricted to authenticated users (otherwise no one could log
in!), but
# the 'authc' filter must still be specified for it so it can process that url's
# login submissions. It is 'smart' enough to allow those requests through as specified by
the
# shiro.loginUrl above.
/login.jsp = authc
/home.jsp = anon, authc
/logout = logout
/account/** = authc
```

① Configuration for the **authc** authentication filer.

② Configuration for the **passwordMatcher** and **passwordService**, which are components for password verification and matching.

③ The definition of the **Datasource** in this example is a JDBC **DataSource**.

④ Configuration for the **Realm** in this example is a JDBC **Realm**.

⑤ This assigns the configured **Realm** to the Siro **SecurityManager**.

⑥ This assigns URL patterns to the appropriate filters.

This shiro.ini file has to be placed in **src/main/webapp/WEB-INF** or **src/main/resources**. To enable the Shiro framework, you have to configure your web.xml file.

```
<listener>
    <listener-class>
        org.apache.shiro.web.env.EnvironmentLoaderListener
    </listener-class>
</listener>

<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

I hear your protests! These configuration files totally conflict with the Java EE "convention over configuration" approach!

In every new Java EE release, you should strive to minimize configuration files. One focus of this tutorial will be to build our Shiro framework application without messy configuration files. You can try to circumvent the problem with some innovative solutions. For example, you can use the Servlet 3.0 features instead of the web.xml file to define the filter in the code:

- You can increase pluggability with methods to add servlets, filters, web fragments, etc.
- You can improve ease of development with annotations for servlets, servlet filters, and servlet-context listeners.

Step 4: Shiro: Getting serious

Step 4.1: Eliminating the web.xml configuration

Step 4.1.1: From the filter to the @WebFilter

A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be attached to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can

be composed with more than one type of web resource.

The main tasks that a filter can perform are:

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

The Shiro filter is the primary filter for web applications implementing the Shiro framework.

To do so, introduce the class **ShiroFilterActivator** that extends **org.apache.shiro.web.servlet.ShiroFilter**, and apply the filter to all requests ("/*").

```
@WebFilter("/*")
public class ShiroFilterActivator extends ShiroFilter {
    private ShiroFilterActivator() {
    }
}
```

Step 4.1.2: The listener to the @WebListener

Create a **ShiroListener** class that extends **org.apache.shiro.web.env.EnvironmentLoaderListener**. In the **contextInitialized()** method, initialize **ENVIRONMENT_CLASS_PARAM** (**shiroEnvironmentClass**) with the **DefaultWebEnvironment** class's name. It is used to define the **WebEnvironment** implementation class to use in the servlet context.

```
@WebListener
public class ShiroListener extends EnvironmentLoaderListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        sce.getServletContext()
           .setInitParameter(ENVIRONMENT_CLASS_PARAM, DefaultWebEnvironment.class.
getName());
        super.contextInitialized(sce);
    }
}
```

Step 4.2: Eliminating the INI configuration file

To eliminate the use of the INI configuration file, use the **createEnvironment(ServletContext sc)** method to define the **WebEnvironment**.

```
@Override
protected WebEnvironment createEnvironment(ServletContext sc)
    DefaultWebEnvironment webEnvironment = (DefaultWebEnvironment) super
.createEnvironment(sc);
    ...
    return webEnvironment;
}
```

Within the **WebEnvironment** component, you must define two parameters: the **FilterChainResolver** and the **SecurityManager**. But before defining the **FilterChainResolver**, you first must understand the concept of a **FilterChain**.

To paraphrase the Java EE 5 JavaDoc, the servlet container provides the **FilterChain** object to allow the developer to view the invocation chain of a filtered request for a resource. Filters use the **FilterChain** to invoke the next filter in the chain or, if the calling filter is the last filter in the chain, to invoke the resource at the end of the chain.

The official Shiro JavaDoc defines some **WebEnvironment** components.

It states that a **FilterChainResolver** can resolve an appropriate **FilterChain** to execute during a **ServletRequest**. It allows resolution of arbitrary filter chains which can be executed for any given request or URI/URL.

This mechanism allows for more flexible **FilterChain** resolution than normal web.xml servlet filter definitions. It allows you to define arbitrary filter chains per URL in a more concise and easy-to-read manner, and even allows filter chains to be dynamically resolved or constructed at run time if the underlying implementation supports it.

The **SecurityManager** executes all security operations for all **Subjects** (i.e. users) across a single application. The interface itself primarily exists as a convenience —t extends the **Authenticator** , **Authorizer**, and **SessionManager** interfaces, thereby consolidating these behaviors into a single point of reference. In most cases, this simplifies configuration by allowing developers to interact with a single **SecurityManager** instead of having to reference **Authenticator**, **Authorizer**, and **SessionManager** instances separately.

Obviously, you have to create these components: **FilterChainResolver** & **SecurityManager**.

Start by creating a new class to hold the configuration for your **ShiroConfiguration** class. The class will have two public getters.

```
public class ShiroConfiguration {
    private ShiroConfiguration() {
    }

    public WebSecurityManager getSecurityManager() {
    }

    public FilterChainResolver getFilterChainResolver() {
    }
}
```

But this solution is still incomplete; you want to use dependency injection to set the **FilterChainResolver** and the **SecurityManager** instances. To enable injection for the two configuration components, bind the instance to your session by annotating the two getter methods with the **@Produces** annotation.

```
public class ShiroConfiguration {
    private ShiroConfiguration() {
    }

    @Produces
    public WebSecurityManager getSecurityManager() {
    }

    @Produces
    public FilterChainResolver getFilterChainResolver() {
    }
}
```

You can create a more elegant **ShiroListener** that sports **@Inject** for both parameters.

```
@WebListener
public class ShiroListener extends EnvironmentLoaderListener {
    @Inject
    WebSecurityManager webSecurityManager;
    @Inject
    FilterChainResolver filterChainResolver;

    @Override
    protected WebEnvironment createEnvironment(ServletContext sc) {
        DefaultWebEnvironment webEnvironment = (DefaultWebEnvironment) super
.createEnvironment(sc);

        webEnvironment.setSecurityManager(securityManager);
        webEnvironment.setFilterChainResolver(filterChainResolver);

        return webEnvironment;
    }

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        sce.getServletContext()
            .setInitParameter(ENVIRONMENT_CLASS_PARAM, DefaultWebEnvironment.class
.getName());
        super.contextInitialized(sce);
    }
}
```

The **ShiroListener** class is now ready, and it's time to move on to defining your **ShiroWebEnvironmentConfiguration** class.

Step 4.2.1: Writing the SecurityManager producer

The Shiro **SecurityManager** has many features and options for tweaking your security experience. For web-based applications, Shiro provides a default **SecurityManager** implementation called **DefaultWebSecurityManager**, which features many preconfigured options. This choice is especially useful for newbies as well as for applications with minor functional requirements. I recommend that you start with the default implementation and modify each option as the need arises.

The following are the most important core elements of the Apache Shiro framework:

- **SessionManager**
- **Realm**
- **Authenticator**

- **Authorizer**
- **CacheManager**
- **SubjectDAO**
- **CredentialsMatcher**

Before we get into the code, let's review the definitions as specificed in the official Shiro Javadoc:

- **SessionManager** manages the creation, maintenance, and clean-up of all application sessions.
- **Realm** is a security component that can access application-specific security entities such as users, roles, and permissions to determine authentication and authorization operations. A **Realm** usually has a one-to-one correspondence with a datasource such as a relational database, file system, or similar resource. As such, implementations of this interface use datasource-specific APIs to determine authorization data (roles, permissions, etc.), such as JDBC, File IO, Hibernate or JPA, or any other data-access API. A **Realm** is essentially a security-specific DAO.
- **Authenticator** is responsible for authenticating accounts in an application. It is one of the primary entry points into the Shiro API.
- **Authorizer** performs authorization (access control) operations for any given **Subject** (i.e. application user). Each method requires a subject principal to perform the action for the corresponding user.
- **CacheManager** provides and maintains the lifecycles of cache instances. Shiro doesn't implement a full cache mechanism itself, since that falls outside the core competency of a security framework. Instead, this interface provides an abstraction (wrapper) API on top of an underlying cache framework's main manager component (e.g. JCache, Ehcache, JCS, OSCache, JBoss Cache, Terracotta, Coherence, GigaSpaces, etc.), allowing Shiro users to configure any cache mechanism they choose.
- **SubjectDAO** is a data-access-object design-pattern specification to enable session access to an enterprise information system (EIS). It provides the usual CRUD methods:
  - create (**org.apache.shiro.session.Session**)
  - readSession (**java.io.Serializable**)
  - update (**org.apache.shiro.session.Session**)
  - delete (**org.apache.shiro.session.Session**)
- **CredentialsMatcher** is an interface implemented by classes that can determine if an **AuthenticationToken**'s provided credentials match a corresponding account's credentials stored in the system.
- **PasswordMatcher** is a **CredentialsMatcher** that employs best-practices comparisons for hashed text passwords.
- **RememberMeManager** is the Shiro framework's default concrete implementation of the **SecurityManager** interface, based around a collection of **Realms**. This implementation delegates its authentication, authorization, and session operations to wrapped **Authenticator**, **Authorizer**,

and **SessionManager** instances respectively via superclass implementation.

The Shiro JavaDoc leaves something to be desired, but hopefully our explanations will improve that.

For your **SecurityManager**, you will:

- Use **DefaultWebSecurityManager** as implementation.
- Create your own customized **Realm** (we will see why later).
- Use **PasswordMatcher** with your own **PasswordService** as the **CredentialsMatcher**.
- Use **CookieRememberMeManager** as the **RememberMeManager**, because your implementation will use cookies to remember and track users (**Subjects**). You will add a **CipherKey** to the **RememberMeManager** to improve cookie naming.
- And for the big surprise, you will use a cache manager, namely **EhCacheManager**! (Shiro promises to provide native support for caching providers such as Hazelcast in future releases).

Step 4.2.1.1: Writing the Realm

To create the **Realm**, you must create a class that extends the **org.apache.shiro.realm.AuthorizingRealm** abstract class and implement the critical methods:

- The **AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)** method is the "getter" of the authentication information (username and password in our case) sitting in the storage system (DB, LDAP, etc.).
- The **AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)** method is the "getter" of the authorization information (roles only in our case, but can be roles and permissions or only permissions, based on the authorization strategy selected for the project).

As this project uses Java EE features with the Shiro framework, you will be using EJB to deal with data. So we will use the injection (or try to) to call your EJBs. As the context dependency injection (CDI) is not enabled for the Shiro components, the Shiro **Realm** is not a CDI-aware component, because of some final methods incoming from the inheritance tree.

Let's take full advantage of the Java EE and use EJB to deal with data, and try to use dependency injection to call your EJB. Note that context dependency injection (CDI) is not enabled for Shiro components because the Shiro **Realm** is not a CDI-aware component, because some final methods are inherited.

Stack Overflow has some posts that shed light on the problem:

- http://stackoverflow.com/questions/15605038
- http://stackoverflow.com/questions/18507629

But not to worry — there are some tricks to make this work! There are at least two options to inject our EJB: **UserService** and **RoleService**.

Method 1) Inject the EJB into the **ShiroConfiguration** *class and pass it to the* **Realm***'s constructor.*

```
public class ShiroConfiguration {
    @Inject
    UserService userService;

    @Produces
    public WebSecurityManager getSecurityManager() {
        DefaultWebSecurityManager securityManager = null;
        if (securityManager == null) {
            AuthorizingRealm realm = new SecurityRealm(userService);
            ...
            securityManager = new DefaultWebSecurityManager(realm);
        }
        return securityManager;
    }
}
```

Unfortunately, this strategy violates proper separation of concerns, so let's look at another approach.

Method 2) Use a classic JNDI lookup in the **Realm** *to grab an access to the EJB:*

The Java EE 6 specification provides for EJB JNDI lookup names of the general form **java:global[/appName]/moduleName/beanName**.

The **appName** component of the lookup name is shown as optional because it does not apply to beans deployed in standalone modules. Only beans packaged in .ear files contain the **appName** component in the **java:global** lookup name. As your application is packaged as a WAR, you will not have the **appName** value in the JNDI names of your EJB. Your **moduleName** is "ShiroTutorial" (see the pom.xml).

So, your EJB's JNDI names are:

```
java:global/ShiroTutorial/UserService
java:global/ShiroTutorial/RoleService
```

Now, you can write your **SecurityRealm** as:

```
public class SecurityRealm extends AuthorizingRealm {
    private Logger logger;
    private UserService userService;

    public SecurityRealm() {
        super();
        this.logger = Logger.getLogger(SecurityRealm.class.getName());

        try {
            InitialContext ctx = new InitialContext();
            this.userService = (UserService) ctx.lookup(
"java:global/ShiroTutorial/UserService");
        } catch (NamingException ex) {
            logger
                .warning("Cannot do the JNDI Lookup to instantiate the User service : " +
ex);
        }
    }
}
```

The specification has fixed predefined JNDI names for the **appName** and the **moduleName**, so you can grab them dynamically.

```
java:app/AppName
java:module/ModuleName
```

Using that, you can write your JNDI lookup code as:

```
try {
    InitialContext ctx = new InitialContext();
    String moduleName = (String) ctx.lookup("java:module/ModuleName");
    this.userService = (UserService) ctx.lookup(String.format("
java:global/%s/UserService", moduleName));
} catch (NamingException ex) {
    logger
        .warning("Cannot do the JNDI Lookup to instantiate the User service : " + ex);
}
```

I only used the **UserService** in this **SecurityRealm**. There's no need for the **RoleService** in this context.

Now, use your EJB in the **doGetAuthenticationInfo()** and **doGetAuthorizationInfo()** methods.

```
@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
    // Identify account to log to
    UsernamePasswordToken userPassToken = (UsernamePasswordToken) token;
    String username = userPassToken.getUsername();
    if (username == null) {
        logger.warning("Username is null.");
        return null;
    }

    // Read the user from DB
    User user = this.userService.findByUsername(username);
    if (user == null) {
        logger.warning("No account found for user [" + username + "]");
        throw new IncorrectCredentialsException();
    }

    return new SimpleAuthenticationInfo(username, user.getPassword(), getName());
}

@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
    //null usernames are invalid
    if (principals == null) {
        throw new AuthorizationException("PrincipalCollection method argument cannot be
null.");
    }

    String username = (String) getAvailablePrincipal(principals);
    Set<String> roleNames = new HashSet<>();
    roleNames.add(this.userService.findByUsername(username).getRole().getName());
    AuthorizationInfo info = new SimpleAuthorizationInfo(roleNames);

    /**
    * If you want to do Permission Based authorization, you can grab the Permissions List
associated to your user:
    * Set<String> permissions = new HashSet<>();
    *
permissions.add(this.userService.findByUsername(username).getRole().getPermissions());
    * ((SimpleAuthorizationInfo)info).setStringPermissions(permissions);
    */
    return info;
}
```

The code is straightforward and the components names tell the whole story.

Step 4.2.1.2: Writing the CredentialsMatcher

CredentialsMatcher is the component that validates credentials and checks if access can be granted. The **CredentialsMatcher** interface has many implementations:

- AllowAllCredentialsMatcher
- HashedCredentialsMatcher
- Md2CredentialsMatcher
- Md5CredentialsMatcher
- PasswordMatcher
- Sha1CredentialsMatcher
- Sha256CredentialsMatcher
- Sha384CredentialsMatcher
- Sha512CredentialsMatcher
- SimpleCredentialsMatcher

Each implementation of **CredentialsMatcher** has specific properties and functionality. For example, **SimpleCredentialsMatcher** is used for direct, plain comparison. **PasswordMatcher** uses an internal **PasswordService** to compare the passwords. The Shiro JavaDoc describes the **PasswordMatcher** as "a **CredentialsMatcher** that employs best-practices comparisons for hashed text passwords."

Choose **PasswordMatcher** as the implementation of **CredentialsMatcher** so that you can define a customized **PasswordService** for password verification.

Let's talk a little bit about about the password itself. We have to choose a hash algorithm for storing and comparing passwords. After some research, I chose bcrypt as a hash algorithm. Based on the Blowfish symmetric-key block-cipher cryptographic algorithm, bcrypt is an adaptive hash function that introduces a work factor (also known as security factor) that allows you to determine the cost of the hash function.

Let's use the **jBCrypt** implementation of bcrypt, and you must add that dependency to the pom.xml file.

```
<dependency>
    <groupId>de.svenkubiak</groupId>
    <artifactId>jBCrypt</artifactId>
    <version>0.4</version>
</dependency>
```

Next, write your customized **PasswordService** class. Call the class **BCryptPasswordService** and implement the **PasswordService** interface and its two methods, **encryptPassword()** and **passwordsMatch()**.

```
public class BCryptPasswordService implements PasswordService {
    public static final int DEFAULT_BCRYPT_ROUND = 10;
    private int logRounds;

    public BCryptPasswordService() {
        this.logRounds = DEFAULT_BCRYPT_ROUND;
    }

    public BCryptPasswordService(int logRounds) {
        this.logRounds = logRounds;
    }

    @Override
    public String encryptPassword(Object plaintextPassword) {
        if (plaintextPassword instanceof String) {
            String password = (String) plaintextPassword;
            return BCrypt.hashpw(password, BCrypt.gensalt(logRounds));
        }
        throw new IllegalArgumentException(
                "BCryptPasswordService encryptPassword only support java.lang.String
credential.");
    }

    @Override
    public boolean passwordsMatch(Object submittedPlaintext, String encrypted) {
        if (submittedPlaintext instanceof char[]) {
            String password = String.valueOf((char[]) submittedPlaintext);
            return BCrypt.checkpw(password, encrypted);
        }
        throw new IllegalArgumentException(
                "BCryptPasswordService passwordsMatch only support char[] credential.");
    }

    public void setLogRounds(int logRounds) {
        this.logRounds = logRounds;
    }

    public int getLogRounds() {
        return logRounds;
    }
}
```

This class will be used in the **CredentialsMatcher** by its **passwordMatch()** method. But it can also be used in the registration process of a new user, before invoking the **UserService**'s save method.

So, update your **ShiroConfiguration** class.

```
public class ShiroConfiguration {
    @Inject
    UserService userService;

    @Produces
    public WebSecurityManager getSecurityManager() {
        DefaultWebSecurityManager securityManager = null;
        if (securityManager == null) {
            AuthorizingRealm realm = new SecurityRealm(userService);
            CredentialsMatcher credentialsMatcher = new PasswordMatcher();
            ((PasswordMatcher) credentialsMatcher).setPasswordService(new
BCryptPasswordService());
            realm.setCredentialsMatcher(credentialsMatcher);
            ...
            securityManager = new DefaultWebSecurityManager(realm);
        }
        return securityManager;
    }
}
```

Step 4.2.1.3: Writing the RememberMeManager

Next, add to **SecurityManager** a **RememberMeManager** that will use cookies to remember security sessions. Use the **CookieRememberMeManager** implementation for this.

The cookie value is a Base64-encoded representation of an AES-encrypted representation of a serialized representation of a collection of principals. Basically, it contains the necessary user information, which can be decrypted with the right key. Using the default key will never do, since a hacker could easily figure that out by looking at Shiro's source code, so specify a custom AES cipher key.

Your customized **RememberMeManager** will look like this:

```
byte[] cypherKey = String.format("0x%s", Hex.encodeToString(new AesCipherService()
.generateNewKey().getEncoded())).getBytes();

RememberMeManager rememberMeManager = new CookieRememberMeManager();
((CookieRememberMeManager) rememberMeManager).setCipherKey(cypherKey);

securityManager.setRememberMeManager(rememberMeManager);
```

Your customized **RememberMeManager** is ready for use.

Step 4.2.1.4: Writing the CacheManager

Shiro has started supporting caching systems, and although the support currently is somewhat lacking, it should become more robust in upcoming releases. But Shiro supports EhCache, and that works nicely for this project's needs. Let's add that dependency to the pom.xml file.

```
<!-- EhCache Dependencies -->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-ehcache</artifactId>
    <version>1.2.4</version>
</dependency>

<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.10.1</version>
</dependency>
```

Next, create the EhCache configuration file in the **src/main/resources** folder. You can call it, for example, ehcache.xml.

```
<ehcache name="shiro" updateCheck="false">
    <diskStore path="/Users/nebrass/shiro-ehcache"/>

    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="false"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"/>

</ehcache>
```

I suggest that you avoid the defaults and always specify the **diskStore** value as the path to the cache storage.

Now, you can define the **CacheManager** for your **SecurityManager** in the **ShiroConfiguration** class.

```
CacheManager cacheManager = new EhCacheManager();
((EhCacheManager) cacheManager).setCacheManagerConfigFile("classpath:ehcache.xml");
securityManager.setCacheManager(cacheManager);
```

The **classpath:ehcache.xml** snippet indicates that the ehcache.xml file is already available in the classpath. You've just set the **EhCacheManager** to be the **CacheManager** for your **SecurityManager**. The cache will be handled according the settings defined in the specified configuration file.

Step 4.2.1.5: Assembling the pieces

Your **SecurityManager** is just about ready. Your **SecurityManager** getter method will be:

```
@Produces
public WebSecurityManager getSecurityManager() {
    DefaultWebSecurityManager securityManager = null;
    if (securityManager == null) {
        AuthorizingRealm realm = new SecurityRealm();

        CredentialsMatcher credentialsMatcher = new PasswordMatcher();
        ((PasswordMatcher) credentialsMatcher).setPasswordService(new
BCryptPasswordService());
        realm.setCredentialsMatcher(credentialsMatcher);

        securityManager = new DefaultWebSecurityManager(realm);

        CacheManager cacheManager = new EhCacheManager();
        ((EhCacheManager) cacheManager).setCacheManagerConfigFile("classpath:ehcache.xml
");
        securityManager.setCacheManager(cacheManager);

        byte[] cypherKey = String.format("0x%s",
Hex.encodeToString(new AesCipherService().generateNewKey().getEncoded()))
                    .getBytes();

        RememberMeManager rememberMeManager = new CookieRememberMeManager();
        ((CookieRememberMeManager) rememberMeManager).setCipherKey(cypherKey);
        securityManager.setRememberMeManager(rememberMeManager);
    }
    return securityManager;
}
```

The CDI **@Produces** annotation identifies a "producer" method or field, which means that it will identify the **getSecurityManager()** method as a producer of **WebSecurityManager** instances in the

CDI context.

The **WebSecurityManager** producer is ready for use. The next step is to define your **FilterChainResolver**.

Step 4.2.2: Writing the FilterChainResolver producer

The **FilterChainResolver** producer method provides the ability to define ad hoc filter chains for any matching URL path in your application. This approach is far more flexible, powerful, and concise than the familiar strategy for defining filter chains in web.xml; even if you never use any other Shiro feature, this alone would justify its use. The <**URL path expression, filterName**> maps each filter to its applied URL path expression.

The URL path expressions are evaluated against an incoming request in the order in which they are defined, and the first match wins.

Shiro provides a large selection of predefined filters.

Filter Name	Class
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
logout	org.apache.shiro.web.filter.authc.LogoutFilter
noSessionCreation	org.apache.shiro.web.filter.session.NoSessionCreationFilter
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter
user	org.apache.shiro.web.filter.authc.UserFilter

This project mainly uses **anon**, **authc**, and **user**:

- **anon** allows access to a path immediately, without performing security checks of any kind.
- **authc** requires the requesting user to be authenticated for the request to continue, forcing the user to log in by redirecting them to the **loginUrl** you've configured.
- **user** allows access to resources if the accessor is a known user, which is defined as having a known principal. This means that any user authenticated or remembered via a "remember me" feature will be allowed access via this filter.

After selecting your filters, specify your selections in the **FilterChainManager**.

```
FormAuthenticationFilter authc = new FormAuthenticationFilter();
AnonymousFilter anon = new AnonymousFilter();
UserFilter user = new UserFilter();

FilterChainManager fcMan = new DefaultFilterChainManager();
fcMan.addFilter("authc", authc);
fcMan.addFilter("anon", anon);
fcMan.addFilter("user", user);
```

Next, define the properties needed for the filters. For example, your **authc** and **user** filters need to have the **loginUrl** defined.

```
authc.setLoginUrl("/login.html");
user.setLoginUrl("/login.html");
```

Next, map the URL path expressions with the associated filter name.

```
fcMan.createChain("/index.html", "anon");
fcMan.createChain("/css/**", "anon");
fcMan.createChain("/api/auth", "anon");
fcMan.createChain("/login.html", "authc");
fcMan.createChain("/**", "user");
```

Set your **fcMan (FilterChainManager)** to the **FilterChainResolver** that you are producing.

```
PathMatchingFilterChainResolver resolver = new PathMatchingFilterChainResolver();
resolver.setFilterChainManager(fcMan);
filterChainResolver = resolver;

return filterChainResolver;
```

Finally, here's the **FilterChainResolver** producer method.

```
@Produces
public FilterChainResolver getFilterChainResolver() {
    FilterChainResolver filterChainResolver = null;
    if (filterChainResolver == null) {
        FormAuthenticationFilter authc = new FormAuthenticationFilter();
        AnonymousFilter anon = new AnonymousFilter();
        UserFilter user = new UserFilter();

        authc.setLoginUrl(WebPages.LOGIN_URL);
        user.setLoginUrl(WebPages.LOGIN_URL);

        FilterChainManager fcMan = new DefaultFilterChainManager();
        fcMan.addFilter("authc", authc);
        fcMan.addFilter("anon", anon);
        fcMan.addFilter("user", user);

        fcMan.createChain("/index.html", "anon");
        fcMan.createChain("/css/**", "anon");
        fcMan.createChain("/api/auth", "anon");
        fcMan.createChain(WebPages.LOGIN_URL, "authc");
        fcMan.createChain("/**", "user");

        PathMatchingFilterChainResolver resolver = new PathMatchingFilterChainResolver();
        resolver.setFilterChainManager(fcMan);
        filterChainResolver = resolver;
    }
    return filterChainResolver;
}
```

At this point, your **WebSecurityManager** and **FilterChainResolver** CDI producers are ready for use and you can update your **ShiroListener** class.

```
@WebListener
public class ShiroListener extends EnvironmentLoaderListener {
    @Inject
    WebSecurityManager securityManager;

    @Inject
    FilterChainResolver filterChainResolver;

    @Override
    protected WebEnvironment createEnvironment(ServletContext sc) {
        DefaultWebEnvironment webEnvironment = (DefaultWebEnvironment) super
.createEnvironment(sc);

        webEnvironment.setSecurityManager(securityManager);
        webEnvironment.setFilterChainResolver(filterChainResolver);

        return webEnvironment;
    }

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        sce.getServletContext().setInitParameter(ENVIRONMENT_CLASS_PARAM,
DefaultWebEnvironment.class.getName());
        super.contextInitialized(sce);
    }
}
```

As you see, CDI awareness is enabled in your **ShiroListener**.

Step 5: Exposing Shiro operations as REST services

Java EE 7 introduced strong support for REST web services, HTML5, WebSockets, and much more. To interact with our application from the client side, let's use JAX-RX to create some RESTful web services to expose server operations such as:

- login
- logout
- list connected users
- get information for a connected user
- etc.

Step 5.1: Enable JAX-RS

To enable JAX-RS services, you must define a JAX-RS application class. Create a class called **JaxRsConfiguration** in the root of the **com.targa.dev.shirotutorial.security** package. But why in the root?

In proper conformity with the BCE pattern, this class is configuration; it is not boundary, controller, or entity. Since it concerns the configuration of the application, it belongs in the package root.

```
@ApplicationPath("api")
public class JaxRsConfiguration extends Application {
}
```

The root URL of the RESTful web services will be **http://[ApplicationPath]/api/**.

You want to have two services: the **Authentication** web service for login, logout, listing connected users, and getting connected user information and the **User Management** web service for CRUD users.

The web services will be created in the **com.targa.dev.shirotutorial.security.boundary** as they are boundaries of your application and will help in interacting with external clients.

Start by creating the **Authentication** web service with the following functionalities:

- **login()**
- **logout()**
- **getSubjectInfo()**
- **getConnectedUsers()**

```
@Path("auth")
public class AuthenticationResource {
    public Response login(){}
    public Response logout(){}
    public Response getSubjectInfo(){}
    public Response getConnectedUsers(){}
}
```

This web service will be accessed at **http://[ApplicationPath]/api/auth**.

To log in, use this:

```
try {
    SecurityUtils.getSubject().login(new UsernamePasswordToken(username, password,
rememberMe));
} catch (AuthenticationException e) {
    throw new IncorrectCredentialsException("Unknown user, please try again");
}
```

In the code above, **username** and **password** are strings and **rememberMe** is a boolean. The user supplies these values through an HTML form, for example. So the **login()** method is:

```
@POST
@Path("login")
public Response login(@NotNull @FormParam("username") String username, ①
                      @NotNull @FormParam("password") String password, ②
                      @NotNull @FormParam("rememberMe") boolean rememberMe,
                      @Context HttpServletRequest request) { ③

    boolean justLogged = false;
    if (!SecurityUtils.getSubject().isAuthenticated() ) { ④
        justLogged = true;
    }

    try {
        SecurityUtils.getSubject().login(new UsernamePasswordToken(username, password,
rememberMe));
    } catch (AuthenticationException e) {
        throw new IncorrectCredentialsException("Unknown user, please try again");
    }

   SavedRequest savedRequest = WebUtils.getAndClearSavedRequest(request); ⑤

    if (savedRequest != null) {
        return this.getRedirectResponse(savedRequest.getRequestUrl(), request);
    } else {
        if (justLogged) {
            return this.getRedirectResponse(WebPages.DASHBOARD_URL, request); ⑦
        }
        return this.getRedirectResponse(WebPages.HOME_URL, request); ⑥
    }
}
```

① **@NotNull** is used for bean validation; validation is defined here to be performed at the earliest possible moment.

② **@FormParam("paramName")** grabs the parameter for the HTTP request.

③ **@Context** injects an instance of a supported resource. Here, it injects the **HttpServletRequest**.

④ This checks whether the user is authenticated.

⑤ This gets the request URL from a previously saved request. If there is no saved request or fallback URL, this method throws an **IllegalStateException**. This method is primarily used to support a common login scenario — if an unauthenticated user accesses a page that requires authentication, it is expected that the request is saved while the user gets directed to the login page. After a successful log in, this method can be called to redirect the user to their originally requested URL, a handy usability feature.

⑥ We will create a **WebPages** class that will hold useful page URLs such as **HOME_URL** (home page of the application), **DASHBOARD_URL** (a welcome page for connected users), and **LOGIN_URL** (the login page).

⑦ I created this method to perform the redirection to the specified URL:

```
private Response getRedirectResponse(String requestedPath, HttpServletRequest request) {

    String appName = request.getContextPath();
    String baseUrl = request.getRequestURL().toString().split(request.getRequestURI())[0]
+ appName;

    try {
        if (requestedPath.split(appName).length > 1) {
            baseUrl += requestedPath.split(appName)[1];
        } else {
            baseUrl += requestedPath;
        }
        return Response.seeOther(new URI(baseUrl)).build();
    } catch (URISyntaxException ex) {
        logger.warning("URL redirection failed for request[" + request + "] : " + ex);
        return Response.serverError()
                       .status(404)
                       .type(MediaType.TEXT_HTML)
                       .build();
    }
}
```

The **logout()** method is simpler.

```
@GET
@Path("logout")
public Response logout(@Context HttpServletRequest request) {
    SecurityUtils.getSubject().logout();
    return this.getRedirectResponse(WebPages.HOME_URL, request);
}
```

Here's the **getSubjectInfo()**.

```
@GET
@Path("me")
public Response getSubjectInfo() {
    Subject subject = SecurityUtils.getSubject();
    if (subject != null && subject.isAuthenticated()) {
        User connectedUser = this.userService.findByUsername(subject.getPrincipal()
.toString());
        return Response.ok(connectedUser).build();
    } else {
        return Response.serverError()
                       .type(MediaType.TEXT_HTML)
                       .build();
    }
}
```

The idea behind the **getConnectedUsers()** is to create a collection that holds all connected users. When a user is logged in, the user is added to the collection. When the user logs out, the user is removed from the collection. We will leverage a nice benefit of CDI: the CDI events mechanism.

Start by injecting **Event<T>** to listen for fired events of type T.

```
@Inject
private Event<AuthenticationEvent> monitoring;

@Inject
private AuthenticationEventMonitor authenticationEventMonitor;
```

The **AuthenticationEventMonitor** class is a holder class for the event producer and monitoring.

Create the **AuthenticationEvent** event class as follows.

```
public class AuthenticationEvent {
    public enum Type { LOGIN, LOGOUT }

    private String username;
    private Date creation;
    private Type type;

    public AuthenticationEvent(String username, Type type) {
        this.username = username;
        this.type = type;
        this.creation = new Date();
    }

    ...

    @Override
    public String toString() {
        return "AuthenticationEvent{" +
                "username='" + username + '\'' +
                ", creation=" + creation +
                ", type=" + type + '}';
    }
}
```

Next, create a holder for the events producer and manager.

```
public class AuthenticationEventMonitor {
    @Inject
    private Logger logger;

    private CopyOnWriteArrayList<String> connectedUsers;

    @PostConstruct
    public void init() {
        this.connectedUsers = new CopyOnWriteArrayList<>();
    }

    public void onAuthenticationEvent(@Observes AuthenticationEvent event) {
        if (event != null &&
                ((event.getType() == AuthenticationEvent.Type.LOGIN) ||
                        (event.getType() == AuthenticationEvent.Type.LOGOUT))) {
            if (event.getType() == AuthenticationEvent.Type.LOGIN) {
                if (!connectedUsers.contains(event.getUsername())) {
                    this.connectedUsers.add(event.getUsername());
                }
            } else {
                if (connectedUsers.contains(event.getUsername())) {
                    this.connectedUsers.remove(event.getUsername());
                }
            }
        } else {
            logger.warning("Unrecognized AuthenticationEvent : [" + event + "]");
        }
    }

    public boolean isUserAlreadyConnected(String username) {
        return this.connectedUsers.contains(username);
    }

    public List<String> getConnectedUsers() {
        return connectedUsers;
    }
}
```

The **AuthenticationResource** class becomes the following.

```
@Path("auth")
@Produces(MediaType.APPLICATION_JSON)
public class AuthenticationResource {
    @Inject
    private Logger logger;

    @Inject
    private UserService userService;

    @Inject
    private Event<AuthenticationEvent> monitoring;

    @Inject
    private AuthenticationEventMonitor authenticationEventMonitor;

    @POST
    @Path("login")
    public Response login(@NotNull @FormParam("username") String username,
                          @NotNull @FormParam("password") String password,
                          @NotNull @FormParam("rememberMe") boolean rememberMe,
                          @Context HttpServletRequest request) {
        boolean justLogged = false;
        if (!SecurityUtils.getSubject().isAuthenticated()) {
            justLogged = true;
        }

        try {
            SecurityUtils.getSubject().login(new UsernamePasswordToken(username,
password, rememberMe));
        } catch (Exception e) {
            throw new IncorrectCredentialsException("Unknown user, please try again");
        }

        SavedRequest savedRequest = WebUtils.getAndClearSavedRequest(request);
        if (savedRequest != null) {
            return this.getRedirectResponse(savedRequest.getRequestUrl(), request);
        } else {
            if (justLogged) {
                return this.getRedirectResponse(WebPages.DASHBOARD_URL, request);
            }
            return this.getRedirectResponse(WebPages.HOME_URL, request);
        }
    }
```

```
    @GET
    @Path("logout")
    public Response logout(@Context HttpServletRequest request) {
        SecurityUtils.getSubject().logout();
        return this.getRedirectResponse(WebPages.HOME_URL, request);
    }

    @GET
    @Path("me")
    public Response getSubjectInfo() {
        Subject subject = SecurityUtils.getSubject();
        if (subject != null && subject.isAuthenticated()) {
            User connectedUser = this.userService.findByUsername(subject.getPrincipal()
.toString());
            return Response.ok(connectedUser).build();
        } else {
            return Response.serverError().type(MediaType.TEXT_HTML).build();
        }
    }

    @GET
    @Path("users")
    public Response getConnectedUsers() {
        List<String> connectedUsers = this.authenticationEventMonitor.getConnectedUsers(
);
        return Response.ok(connectedUsers).build();
    }

    private Response getRedirectResponse(String requestedPath, HttpServletRequest
request) {
        String appName = request.getContextPath();
        String baseUrl = request.getRequestURL().toString().split(request.getRequestURI(
))[0] + appName;

        try {
            if (requestedPath.split(appName).length > 1) {
                baseUrl += requestedPath.split(appName)[1];
            } else {
                baseUrl += requestedPath;
            }
            return Response.seeOther(new URI(baseUrl)).build();
        } catch (URISyntaxException ex) {
            return Response.serverError().status(404).type(MediaType.TEXT_HTML).build();
        }
    }
}
```

Then, create the **UserResource** class.

```
@Path("users")
@Produces(MediaType.APPLICATION_JSON)
public class UserResource {
    @Inject
    private UserService userService;

    @POST
    public Response addUser(@Valid User user, @Context UriInfo info) {
        User saved = this.userService.save(user);
        long id = saved.getId();
        URI uri = info.getAbsolutePathBuilder().path("/" + id).build();
        return Response.created(uri).build();
    }

    @DELETE
    @Path("{id}")
    public Response deleteUser(@PathParam("id") long id) {
        this.userService.delete(id);
        return Response.ok().build();
    }

    @PUT
    public Response editUser(@Valid User user, @Context UriInfo info) {
        User searched = this.userService.save(user);
        return Response.ok(searched).build();
    }

    @GET
    @Path("{id}")
    public Response findUser(@PathParam("id") long id) {
        User searched = this.userService.findById(id);
        return Response.ok(searched).build();
    }

    @GET
    public Response listUsers() {
        List<User> all = this.userService.findAll();
        return Response.ok(all).build();
    }
}
```

The last item in the JAX-RS layer is the **ExceptionMapper** class.

```
@Provider
public class ShiroExceptionMapper implements ExceptionMapper<Exception> {

    private static final String CAUSE = "cause";

    @Override
    public Response toResponse(Exception ex) {
        if (ex instanceof UnknownAccountException) {
            return Response.status(Response.Status.FORBIDDEN)
            .header(ShiroExceptionMapper.CAUSE, "Your username wrong")
                    .type(MediaType.TEXT_HTML)
                    .build();
        }
        if (ex instanceof IncorrectCredentialsException) {
            return Response.status(Response.Status.UNAUTHORIZED)
                    .header(ShiroExceptionMapper.CAUSE, "Password is incorrect")
                    .type(MediaType.TEXT_HTML)
                    .build();
        }
        if (ex instanceof LockedAccountException) {
            return Response.status(Response.Status.CONFLICT)
                    .header(ShiroExceptionMapper.CAUSE, "This username is locked")
                    .type(MediaType.TEXT_HTML)
                    .build();
        }
        if (ex instanceof AuthenticationException) {
            return Response.status(Response.Status.BAD_REQUEST)
                    .header(ShiroExceptionMapper.CAUSE, ex.getMessage())
                    .type(MediaType.TEXT_HTML)
                    .build();
        }

        return Response.serverError().
                header(ShiroExceptionMapper.CAUSE, ex.toString()).build();
    }
}
```

At this point, your JAX-RS services are ready for use, and all of your Shiro services are exposed via RESTful services.

When applying this configuration, you will add a **;jsessionid** param to the HTTP URL for every webpage request in the application. This parameter is used for tracking the session using cookies.

JSESSIONID is a cookie generated by a servlet container, such as Tomcat or Jetty, and is used for session management in Java EE web apps for HTTP. Since HTTP is a stateless protocol, a webserver must work to relate separate requests coming from a single client. Session management is the process of tracking a user session using standard techniques such as cookies and URL rewriting.

Once a session is created, the container sends the **JSESSIONID** cookie in the client response. In the case of HTML access, no user session is created. If the client has disabled cookies, then the container uses URL rewriting to manage the session, where the **JSESSIONID** is appended to the URL like this: https://localhost:8080/ShiroTutorial/login.htm;**jsessionid=9e934f9330d5081e34ce607b478a8e32dd9b0297dbb8**

To solve this, just ask the container to handle the session using **COOKIES** by adding this node to the web.xml file:

```
<session-config>
    <session-timeout>
        30
    </session-timeout>
    <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

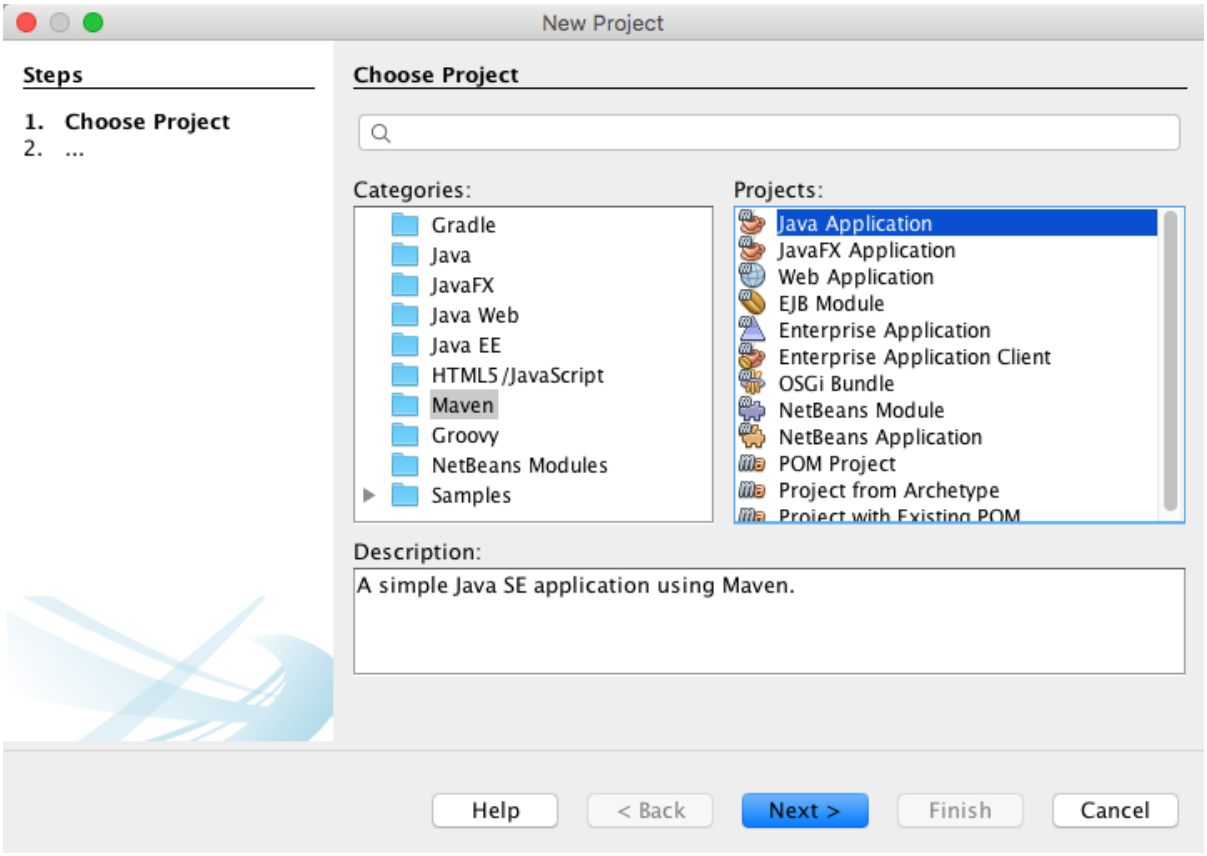
Step 5.2: Test your RESTful services

Let's write a small client in the form of a small Java SE app to test the web services. The client app will send requests to our web services and you will use JUnit to check that all goes well.

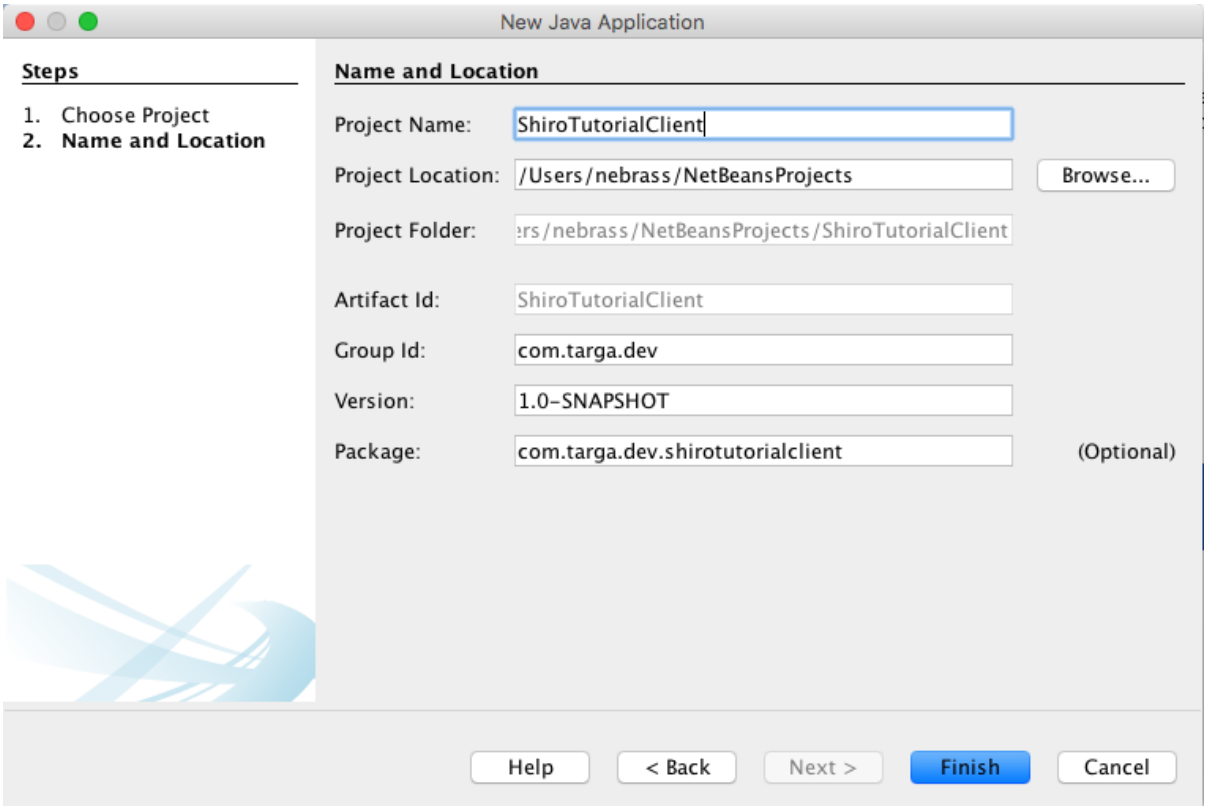
For testing the web services we are going to write a small client in the form of a small Java SE app. Our client app, will make some requests to our webservices to check if it is ok. This check will be done using **JUnit**.

Step 5.2.1: Create the client project

Now that everything is explained, assemble the pieces into a working project:



And next...



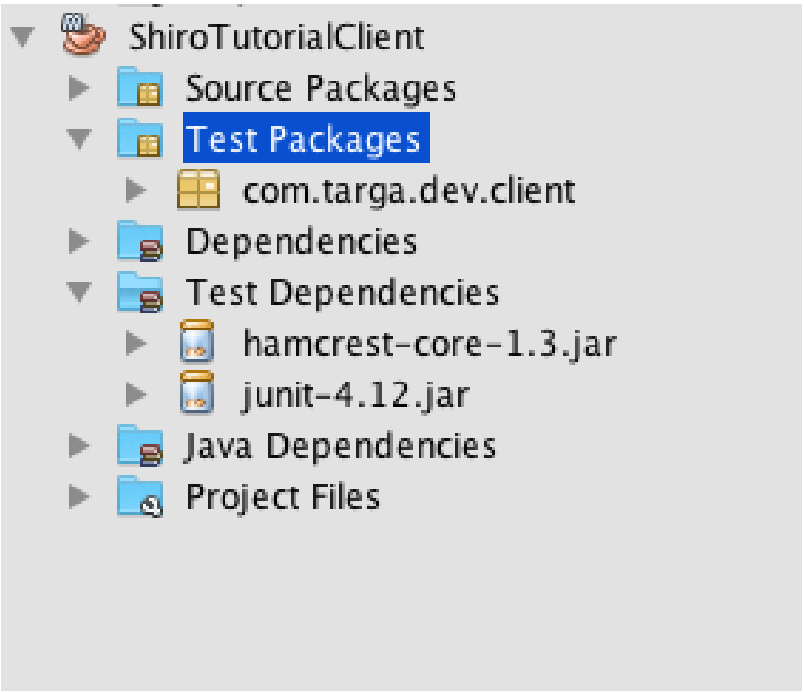
After creating the project, go to the pom.xml and change it as follows.

```
<?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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.targa.dev</groupId>
  <artifactId>ShiroTutorialClient</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <build>
    <finalName>ShiroTutorialClient</finalName>
  </build>
</project>
```

Add the JUnit dependency to your pom.xml.

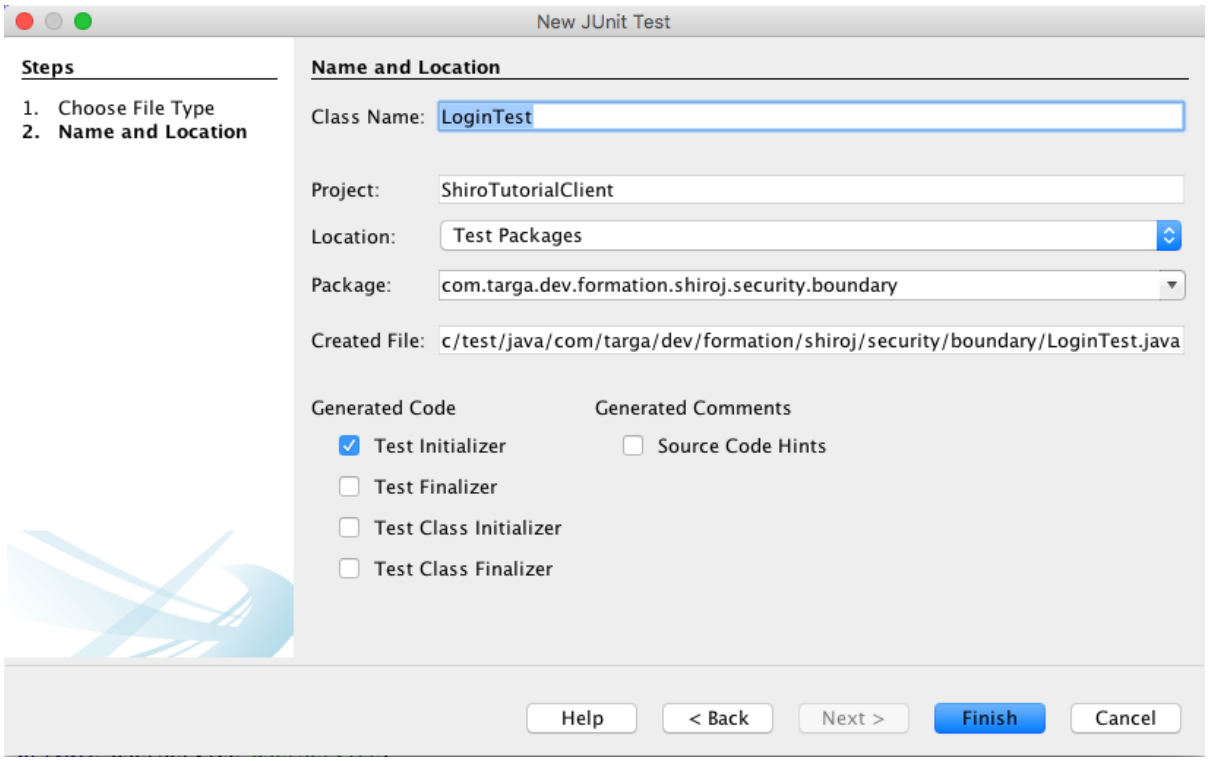
```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
      <version>1.3</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

After adding these dependencies with the "Test" scope, NetBeans will add a new section called "Test Packages" to the project.



Create rour first JUnit class. Right-click on Test Packages and choose New → JUnit Test.

Since you'll test the login feature in the authentication web service, you must place your **LoginTest** class in the same package as the **Authentication** class, the **com.targa.dev.shirotutorial.security.boundary** package.



To make a RESTful client, you have to add the dependencies. The following assumes you are using Payara Server and uses its bundled Jersey module as the JAX-RS implementation:

```
<dependency>
  <groupId>org.glassfish.jersey.core</groupId>
  <artifactId>jersey-client</artifactId>
  <version>2.22.1</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-processing</artifactId>
  <version>2.22.1</version>
</dependency>
```

Define the first version of your **LoginTest** class.

```
public class LoginTest {

  private Client client;
  private WebTarget target;

  @Before
  public void setUp() {
    client = ClientBuilder.newClient();
    target = client.target("http://localhost:8080/ShiroTutorial/")
                   .path("api/auth/login"); ①
  }
  ...
}
```

① The target URL of your client is the login RESTful API URL http://localhost:8080/ShiroTutorial/api/auth/login .

Use this class to test the login operation. Verify that a successful login and a failed login work as expected.

For a successful login attempt:

- Send the login params (username, password, and **rememberMe**) to the login RESTful service
- → get the "OK" HTTP response code (200)

For a failing login attempt:

- Send the login params (username, password, and **rememberMe**) to the login RESTful service
- → get the "Unauthorized" HTTP response code (401)

Before testing the login operations, we create a new **User** record. There are many ways to do that but let's build an initializer EJB that inserts a new user when the application starts.

```
@Singleton
@Startup
public class InsertUser {
    @Inject
    UserService us; ①
    @Inject
    BCryptPasswordService passwordService; ②

    @PostConstruct ③
    public void insert() {
        User craftsman  = new User("shiro", passwordService.encryptPassword("netbeans"));
        Role role = new Role();
        role.setName("ADMIN");
        craftsman.setRole(role);

        this.us.save(craftsman);
    }
}
```

① The CRUD service for the user entities.

② The **BCryptPassword** utility class created for matching the passwords in the **PasswordMatcher**.

③ **@PostConstruct** executes the annotated method (**public void insert(){}**) after the EJB is contructed.

The EJB is a **@Singleton** marked with the **@Startup** annotation, which means it starts when the application starts — every time our application starts. To avoid duplicating records on every restart, we can ask the ORM to **drop-and-create** tables each time the application starts. To do so, just add the following line to the properties section of your persistence.xml file.

```
<property name="javax.persistence.schema-generation.database.action" value="drop-and-
create"/>
```

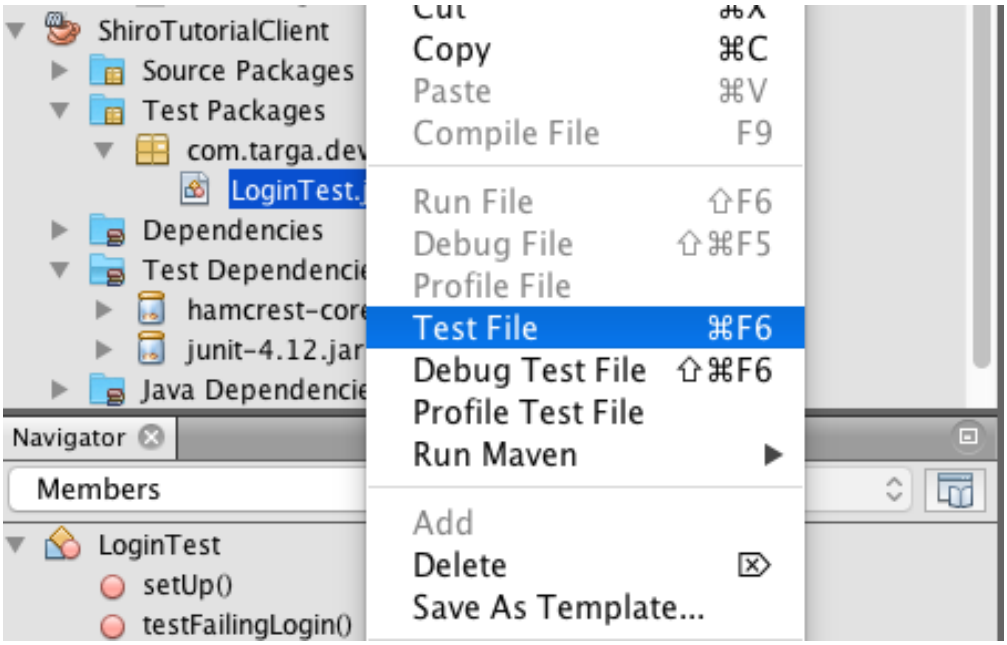
You must remove this configuration from the production environment or you will lose all your data every time you restart your app/server.

The test methods will be :

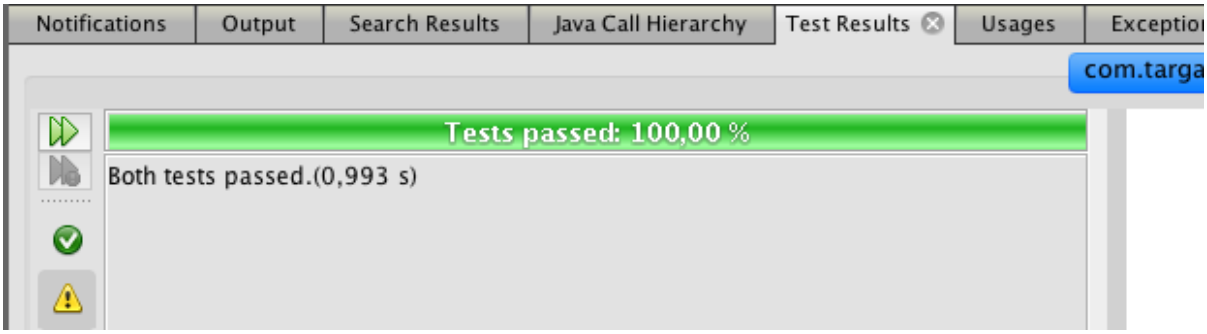
```
@Test
public void testSuccessfulLogin() {
  Form form = new Form();
  form.param("username", "shiro");
  form.param("password", "netbeans");
  form.param("rememberMe", "false");

  Response response = target.request(MediaType.APPLICATION_JSON_TYPE)
          .post(Entity.entity(form, MediaType.APPLICATION_FORM_URLENCODED));

  assertEquals(response.getStatus(), Response.Status.OK.getStatusCode());
}

@Test
public void testFailingLogin() {
  Form form = new Form();
  form.param("username", "shiro");
  form.param("password", "netbeans2");
  form.param("rememberMe", "false");

  Response response = target.request(MediaType.APPLICATION_JSON_TYPE)
          .post(Entity.entity(form, MediaType.APPLICATION_FORM_URLENCODED));

  assertEquals(response.getStatus(), Response.Status.UNAUTHORIZED.getStatusCode());
}
```

Sending the **rememberMe** parameter is not mandatory for false values, as the null value will be mapped to false at the RESTful boundary.

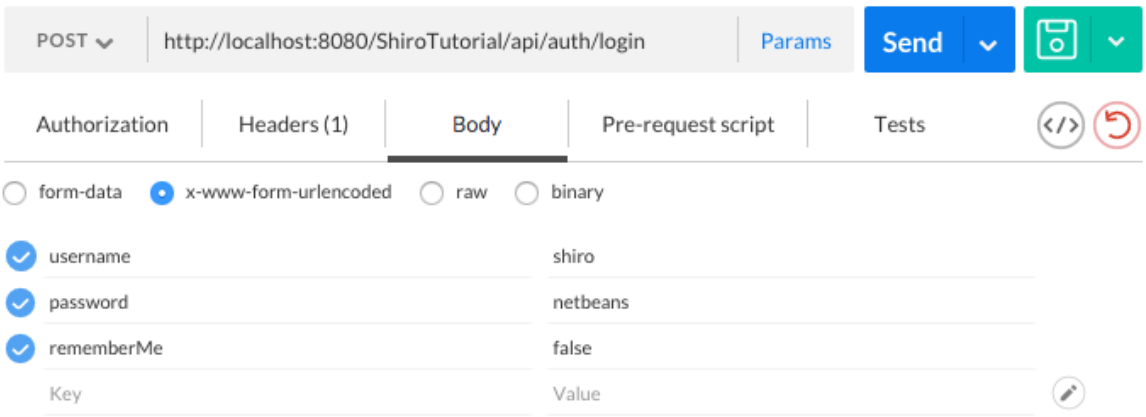
Execute the test.



You will get the following result.



You can keep making JUnit tests to check your RESTful services, which I recommend because you can easily debug them. In contrast, browser plugins such as Postman can make debugging more difficult.



PART FOUR

What's Next?

How to consume Shiro's web services

Our RESTful web services can be consumed using every client that can make HTTP requests:

- HTML5 pages
- Swing applications
- mobile applications on iOS, Android, etc.
- etc.

For example, we can make an AngularJS single-page application (SPA) as the view layer of our JavaEE7 application:

- Make the REST calls in the services layer.
- Manage your Auth[2] operations depending on the HTTP response codes.
- For the authorization, you can use AngularJS authorization plugins (angular-route-authorisation, angular-http-auth, angular-permission, etc.)

You may have to tweak your code to make it coherent with your authorization mechanism.

What can you add to the implementation?

Our implementation of Apache Shiro framework is basic. We did not try concepts such as LDAP, ActiveDirectory, CAS SSO, OAuth, or others.

The community offers many integrations:

- Pairing Stormpath User Management with Shiro provides a full application-security and user-management system with little to no coding.
- Grails offers up-to-date Grails/Shiro integration, including Grails 2.0 and Shiro 1.2. The plugin adds easy authentication and access control to Grails applications.
- OAuth provides source code for an OAuth module for Apache Shiro, based on Scribe.
- A Google App Engine plugin demonstrates one way to integrate Shiro with Google App Engine and Google Guice, and comes with front-end user registration and password management.
- Play Shiro Integration simply integrates Apache Shiro and Play 2.0. If you want to play with it, this project could use an update handling statelessness since the Shiro 1.2 release.
- The 55 Minutes Wicket project is a nifty set of tools and libraries for enhancing productivity with the Apache Wicket Java web framework, including Shiro integration.
- Lift Shiro integrates Shiro and the Lift Web framework, using Lift's sitemap locs instead of Shiro's built-in web.xml resource filters to control access to URLs.

Recommendations

1. Handle every HTTP error code (404, 400, 500, etc.) and try to have an explicit error message in the HTTP headers of every response.
2. Ensure that your design covers the maximum use cases and be sure that all resources that get anonymous access are neither vulnerable nor sensitive.
3. Activate and configure SSL when possible.
4. Avoid transmitting the **JSESSIONID** parameter over plain old HTTP.
5. Avoid using URL parameters for session tracking.
6. Be sure to define a session-timeout value that is rational for your needs.
7. Try to build your knowledge of Auth[2] mechanisms and strategies. Take a look at OWASP's useful resources:
    a. OWASP Authentication Cheat Sheet
    b. OWASP Guide to Authorization
    c. OWASP REST Security Cheat Sheet
    d. OWASP Session Management Cheat Sheet
8. Penetration-test your web application to be sure that your security mechanisms are powerful. You can use the OWASP Pentesting Guide.

Do It Now!

Additional reading

I suggest that you start diving into the official documentation of the Shiro framework. It's true that its documentation is poor in comparison to other frameworks such as Spring Security or PicketLink, which have more resources and more support:

- https://shiro.apache.org/static/latest/apidocs/
- http://shiro.apache.org/tutorial.html

There are some useful tutorials, including older Shiro stuff, like for INI and XML configuration:

- http://meri-stuff.blogspot.fr/2011/03/apache-shiro-part-1-basics.html
- http://en.kodcu.com/2013/10/java-security-framework-apache-shiro-2/
- http://www.jjoe64.com/2014/01/apache-shiro-with-hibernatesql-full.html
- http://balusc.omnifaces.org/2013/01/apache-shiro-is-it-ready-for-java-ee-6.html

You can learn tips and tricks from these tutorials, and you can adopt these modifications in any JavaEE7 project.

The End ;

This was my first mini-book. I have tried to explain a subject I find interesting. I hope you find some useful information and tips.

> Greatness is a road leading towards the unknown.
>
> — Charles de Gaulle

> Far better it is to dare mighty things, to win glorious triumphs, even though checkered by failure, than to take rank with those poor spirits who neither enjoy much nor suffer much, because they live in the gray twilight that knows not victory nor defeat.
>
> — Theodore Roosevelt

Table 1. Book Editions & Releases

Version	Date
1.0	January 2016
This book was written using AsciiDoctor	

I am available for any suggestion or corrections at lnibrass@gmail.com.

Best regards,

Nebrass