Application Security With Apache Shiro

Are you frustrated when you try to secure your applications? Do you feel existing Java security solutions are difficult to use and only confuse you further? This article introduces [Apache Shiro](http://shiro.apache.org/), a Java security framework that provides a simple but powerful approach to application security. It explains Apache Shiro’s project goals, architectural philosophies and how you might use Shiro to secure your own applications.

## What is Apache Shiro?

Apache Shiro (pronounced “shee-roh”, the Japanese word for ‘castle’) is a powerful and easy-to-use Java security framework that performs authentication, authorization, cryptography, and session management and can be used to secure any application - from the command line applications, mobile applications to the largest web and enterprise applications.

Shiro provides the application security API to perform the following aspects (I like to call these the 4 cornerstones of application security):

* Authentication - proving user identity, often called user ‘login’.
* Authorization - access control
* Cryptography - protecting or hiding data from prying eyes
* Session Management - per-user time-sensitive state

Shiro also supports some auxiliary features, such as web application security, unit testing, and multithreading support, but these exist to reinforce the above four primary concerns.

## Why was Apache Shiro created?

For a framework to really make a good case for its existence, and therefore a reason for you to use it, it should satisfy needs that aren’t met by other alternatives. To understand this, we need to look at Shiro’s history and the alternatives when it was created.

Before entering the Apache Software Foundation in 2008, Shiro was already 5 years old and previously known as the JSecurity project, which started in early 2003. In 2003, there weren’t many general-purpose security alternatives for Java application developers - we were pretty much stuck with the Java Authentication and Authorization Service, otherwise known as JAAS. There were a lot of shortcomings with JAAS - while its authentication capabilities were somewhat tolerable, the authorization aspects were obtuse and frustrating to use. Also, JAAS was heavily tied to Virtual Machine-level security concerns, for example, determining if a class should be allowed to be loaded in the JVM. As an application developer, I cared more about what an application end-user could do rather than what my code could do inside the JVM.

Due to the applications I was working with at the time, I also needed access to a clean, container-agnostic session mechanism. The only session choices in the game at the time were HttpSessions, which required a web container, or EBJ 2.1 Stateful Session Beans, which required an EJB container. I needed something that could be decoupled from the container, usable in any environment I chose.

Finally, there was the issue of cryptography. There are times when we all need to keep data secure, but the Java Cryptography Architecture was hard to understand unless you were a crypto expert. The API was full of checked exceptions and felt cumbersome to use. I was hoping for a cleaner out-of-the-box solution to easily encrypt and decrypt data as necessary.

So looking at the security landscape of early 2003, you can quickly realize that there was nothing that could satisfy all of those requirements in a single, cohesive framework. Because of that, JSecurity, and then later, Apache Shiro, was born.

## Why would you use Apache Shiro today?

The framework landscape has changed quite a bit since 2003, so there should still be a compelling reason to use Shiro today. There are quite a few reasons actually. Apache Shiro is:

* **Easy To Use** - Ease of use is the project’s ultimate goal. Application security can be extremely confusing and frustrating and thought of as a ‘necessary evil’. If you make it so easy to use that novice programmers can start using it, it doesn’t have to be painful anymore.
* **Comprehensive** - There is no other security framework with the breadth of scope that Apache Shiro claims, so it can likely be your ‘one stop shop’ for your security needs.
* **Flexible**- Apache Shiro can work in any application environment. While it works in web, EJB, and IoC environments it does not require them. Nor does Shiro mandate any specification or even have many dependencies.
* **Web Capable**- Apache Shiro has fantastic web application support, allowing you to create flexible security policies based on application URLs and web protocols (e.g. REST), while also providing a set of JSP libraries to control page output.
* **Pluggable**- Shiro’s clean API and design patterns make it easy to integrate with many other frameworks and applications. You’ll see Shiro integrated seamlessly with frameworks like Spring, Grails, Wicket, Tapestry, Mule, Apache Camel, Vaadin, and many others.
* **Supported**- Apache Shiro is part of the [Apache Software Foundation](http://www.apache.org/), an organization proven to act in the best interest of its community. The project development and user groups have friendly citizens ready to help. Commercial companies like [Katasoft](http://www.katasoft.com/) also provide professional support and services if desired.

## Who’s Using Shiro?

Shiro and its predecessor JSecurity has been in use for years in projects for companies of all sizes and across industries. Since becoming an Apache Software Foundation Top Level Project, site traffic and adoption have continued to grow significantly. Many open-source communities are using Shiro as well, for example, Spring, Grails, Wicket, Tapestry, Tynamo, Mule, and Vaadin, just to name a few.

Commercial companies like Katasoft, Sonatype, MuleSoft, one of the major social networks, and more than a few New York commercial banks use Shiro to secure their commercial software and websites.

## Core Concepts: Subject, SecurityManager, and Realms

Now that we’ve covered Shiro’s benefits, let’s jump right in to its API so you can get a feel for it. Shiro’s architecture has three main concepts - the Subject, the SecurityManager, and Realms.

### Subject

When you’re securing your application, probably the most relevant questions to ask yourself are, “Who is the current user?” or “Is the current user allowed to do X”? It is common for us to ask ourselves these questions as we're writing code or designing user interfaces: applications are usually built based on user stories, and you want functionality represented (and secured) based on a per-user basis. So, the most natural way for us to think about security in our application is based on the current user. Shiro’s API fundamentally represents this way of thinking in its Subject concept.

The word *Subject* is a security term that basically means "the currently executing user". It's just not called a 'User' because the word 'User' is usually associated with a human being. In the security world, the term 'Subject' can mean a human being, but also a 3rd party process, daemon account, or anything similar. It simply means 'the thing that is currently interacting with the software'. For most intents and purposes though, you can think of this as Shiro’s ‘User’ concept. You can easily acquire the Shiro Subject anywhere in your code as shown in Listing 1 below.

**Listing 1. Acquiring the Subject**

import org.apache.shiro.subject.Subject;

import org.apache.shiro.SecurityUtils;

...

Subject currentUser = SecurityUtils.getSubject();

Once you acquire the Subject, you immediately have access to 90% of everything you’d want to do with Shiro for the current user, such as login, logout, access their session, execute authorization checks, and more - but more on this later. The key point here is that Shiro’s API is largely intuitive because it reflects the natural tendency for developers to think in ‘per-user’ security control. It is also easy to access a Subject anywhere in code, allowing security operations to occur anywhere they are needed.

### SecurityManager

The Subject’s ‘behind the scenes’ counterpart is the SecurityManager. While the Subject represents security operations for the current user, the SecurityManager manages security operations for *all* users. It is the heart of Shiro’s architecture and acts as a sort of ‘umbrella’ object that references many internally nested security components that form an object graph. However, once the SecurityManager and its internal object graph is configured, it is usually left alone and application developers spend almost all of their time with the Subject API.

So how do you set up a SecurityManager? Well, that depends on your application environment. For example, a web application will usually specify a Shiro Servlet Filter in web.xml, and that will set up the SecurityManager instance. If you’re running a standalone application, you’ll need to configure it another way. But there are many of configuration options.

There is almost always a single SecurityManager instance per application. It is essentially an application singleton (although it does not need to be a *static* singleton). Like almost all things in Shiro, the default SecurityManager implementations are [POJO](http://en.wikipedia.org/wiki/Plain_Old_Java_Object)s and are configurable with any POJO-compatible configuration mechanism - normal Java code, Spring XML, YAML, .properties and .ini files, etc. Basically anything that is capable of instantiating classes and calling JavaBeans-compatible methods may be used.

To that end, Shiro provides a default ‘common denominator’ solution via text-based [INI](http://en.wikipedia.org/wiki/INI_file)configuration. INI is easy to read, simple to use, and requires very few dependencies. You’ll also see that with a simple understanding of object graph navigation, INI can be used effectively to configure simple object graph like the SecurityManager. Note that Shiro also supports Spring XML configuration and other alternatives, but we’ll cover INI here.

The simplest example of configuring Shiro based on INI is shown in the example in Listing 2 below.

**Listing 2. Configuring Shiro with INI**

[main]

cm = org.apache.shiro.authc.credential.HashedCredentialsMatcher

cm.hashAlgorithm = SHA-512

cm.hashIterations = 1024

# Base64 encoding (less text):

cm.storedCredentialsHexEncoded = false

iniRealm.credentialsMatcher = $cm

[users]  
jdoe = TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJpcyByZWFzb2  
asmith = IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbXNoZWQsIG5vdCB

In Listing 2, we see the example INI configuration we will use to configure the SecurityManager instance. There are two INI sections: [main] and [users].

The [main] section is where you configure the SecurityManager object and/or any objects (like Realms) used by the SecurityManager. In this example, we see two objects being configured:

1. The cm object, which is an instance of Shiro’s HashedCredentialsMatcher class. As you can see, various properties of the cm instance are being configured via 'nested dot' syntax - a convention used by the IniSecurityManagerFactory shown in Listing 3, to represent object graph navigation and property setting.
2. The iniRealm object, which is a component used by the SecurityManager to represent user accounts defined in the INI format.

The [users] section is where you can specify a static list of user accounts - convenient for simple applications or when testing.

For the purposes of this introduction, it is not important to understand the intricacies of each section, but rather to see that INI configuration is one simple way of configuring Shiro. For more detailed information on INI configuration, please see [Shiro's documentation](http://shiro.apache.org/documentation.html).

**Listing 3. Loading shiro.ini Configuration File**

import org.apache.shiro.SecurityUtils;

import org.apache.shiro.config.IniSecurityManagerFactory;

import org.apache.shiro.mgt.SecurityManager;

import org.apache.shiro.util.Factory;

...

//1. Load the INI configuration  
Factory<SecurityManager> factory =  
new IniSecurityManagerFactory("classpath:shiro.ini");

//2. Create the SecurityManager  
SecurityManager securityManager = factory.getInstance();

//3. Make it accessible  
SecurityUtils.setSecurityManager(securityManager);

In Listing 3, we see a three-step process in this simple example:

1. Load the INI configuration that will configure the SecurityManager and its constituent components.
2. Create the SecurityManager instance based on the configuration (using Shiro’s Factory concept that represents the [Factory Method design pattern](http://en.wikipedia.org/wiki/Factory_method_pattern)).
3. Make the SecurityManager singleton accessible to the application. In this simple example, we set it as a VM-static singleton, but this is usually not necessary - your application configuration mechanism can determine if you need to use static memory or not.

### Realms

The third and final core concept in Shiro is that of a *Realm*. A Realm acts as the ‘bridge’ or ‘connector’ between Shiro and your application’s security data. That is, when it comes time to actually interact with security-related data like user accounts to perform authentication (login) and authorization (access control), Shiro looks up many of these things from one or more Realms configured for an application.

In this sense a Realm is essentially a security-specific [DAO](http://en.wikipedia.org/wiki/Data_access_object): it encapsulates connection details for data sources and makes the associated data available to Shiro as needed. When configuring Shiro, you must specify at least one Realm to use for authentication and/or authorization. More than one Realm may be configured, but at least one is required.

Shiro provides out-of-the-box Realms to connect to a number of security data sources (aka directories) such as LDAP, relational databases (JDBC), text configuration sources like INI and properties files, and more. You can plug-in your own Realm implementations to represent custom data sources if the default Realms do not meet your needs. Listing 4 below is an example of configuring Shiro (via INI) to use an LDAP directory as one of the application’s Realms.

**Listing 4. Example realm configuration snippet to connect to LDAP user data store**

[main]

ldapRealm = org.apache.shiro.realm.ldap.JndiLdapRealm

ldapRealm.userDnTemplate = uid={0},ou=users,dc=mycompany,dc=com

ldapRealm.contextFactory.url = ldap://ldapHost:389

ldapRealm.contextFactory.authenticationMechanism = DIGEST-MD5

Now that we’ve seen how to set up a basic Shiro environment, let’s discuss how you, as a developer, would go about using the framework.

## Authentication

Authentication is the process of verifying a user's identity. That is, when a user authenticates with an application, they are proving they actually are who they say they are. This is also sometimes referred to as 'login'. This is typically a three-step process.

1. Collect the user’s identifying information, called *principals*, and supporting proof of identity, called *credentials*.
2. Submit the principals and credentials to the system.
3. If the submitted credentials match what the system expects for that user identity (principal), the user is considered authenticated. If they don’t match, the user is not considered authenticated.

A common example of this process that everyone is familiar with is that of the username/password combination. When most users login to a software application, they usually provide their username (the principal) and their supporting password (the credential). If the password (or representation of it) stored in the system matches what the user specifies, they are considered authenticated.

Shiro supports this same workflow in a simple and intuitive way. As we’ve said, Shiro has a Subject-centric API - almost everything you care to do with Shiro at runtime is achieved by interacting with the currently executing Subject. So, to login a Subject, you simply call its login method, passing an AuthenticationToken instance that represents the submitted principals and credentials (in this case, a username and password). This example is shown in Listing 5 below.

**Listing 5. Subject Login**

//1. Acquire submitted principals and credentials:

AuthenticationToken token =

new UsernamePasswordToken(username, password);

//2. Get the current Subject:  
Subject currentUser = SecurityUtils.getSubject();

//3. Login:  
currentUser.login(token);

As you can see, Shiro’s API easily reflects the common workflow. You’ll continue to see this simplicity as a theme for all of the Subject’s operations. When the login method is called, the SecurityManager will receive the AuthenticationToken and dispatch it to one or more configured Realms to allow each to perform authentication checks as required. Each Realm has the ability to react to submitted AuthenticationTokens as necessary. But what happens if the login attempt fails? What if the user specified an incorrect password? You can handle failures by reacting to Shiro’s runtime AuthenticationException as shown in Listing 6.

**Listing 6. Handle Failed Login**

//3. Login:

try {

    currentUser.login(token);

} catch (IncorrectCredentialsException ice) { …

} catch (LockedAccountException lae) { …

}

…

catch (AuthenticationException ae) {…

}

You can choose to catch one of the AuthenticationException subclasses and react specifically, or generically handle any AuthenticationException (for example, show the user a generic “Incorrect username or password” message). The choice is yours depending on your application requirements.

After a Subject logs-in successfully, they are considered authenticated and usually you allow them to use your application. But just because a user proved their identity doesn’t mean that they can do whatever they want in your application. That begs the next question, “how do I control what the user is allowed to do or not?” Deciding what users are allowed to do is called *authorization*. We’ll cover how Shiro enables authorization next.

## Authorization

Authorization is essentially access control - controlling what your users can access in your application, such as resources, web pages, etc. Most users perform access control by employing concepts such as roles and permissions. That is, a user is usually allowed to do something or not based on what roles and/or permissions are assigned to them. Your application can then control what functionality is exposed based on checks for these roles and permissions. As you might expect, the Subject API allows you to perform role and permission checks very easily. For example, the code snippet in Listing 7 shows how to check if a Subject has been assigned a certain role.

**Listing 7. Role Check**

if ( subject.hasRole(“administrator”) ) {

    //show the ‘Create User’ button

} else {

    //grey-out the button?

}

As you can see, your application can enable or disable functionality based on access control checks.

Permission checks are another way to perform authorization. Checking for roles as in the example above suffers from one significant flaw: you can’t add or delete roles at runtime. Your code is hard-coded with role names, so if you changed the role names and/or configuration, your code would be broken! If you need to be able to change a role’s meaning at runtime, or add or delete roles as desired, you have to rely on something else.

To that end, Shiro supports its notion of *permissions*. A permission is a raw statement of functionality, for example ‘open a door’, ‘create a blog entry’, ‘delete the ‘jsmith’ user’, etc. By having permissions reflect your application’s raw functionality, you only need to change permission checks when you change your application’s functionality. In turn, you can assign permissions to roles or to users as necessary at runtime.

As an example, shown in Listing 8 below, we can rewrite our previous role check and instead use a permission check.

**Listing 8. Permission Check**

if ( subject.isPermitted(“user:create”) ) {

    //show the ‘Create User’ button

} else {

    //grey-out the button?

}

This way, any role or user assigned the “user:create” permission can click the ‘Create User’ button, and those roles and assignments can even change at runtime, providing you with a very flexible security model.

The “user:create” string is an example of a permission string that adheres to certain parsing conventions. Shiro supports this convention out of the box with its WildcardPermission. Although out of scope for this introduction article, you’ll see that the WildcardPermission can be extremely flexible when creating security policies, and even supports things like *instance-level* access control.

**Listing 9. Instance-Level Permission Check**

if ( subject.isPermitted(“user:delete:jsmith”) ) {

    //delete the ‘jsmith’ user

} else {

    //don’t delete ‘jsmith’

}

This example shows that you can control, even down to a very fine-grained instance level, access to individual resources if you needed to. You could even invent your own permission syntax if you wanted to. See the [Shiro Permission](http://shiro.apache.org/permissions.html) documentation for more information. Finally, just as with authentication, the above calls eventually make their way to the SecurityManager, which will consult one or more Realms to make the access control decisions. This allows a Realm to respond to both authentication and authorization operations as necessary.

So that is a brief overview of Shiro’s authorization capabilities. And while most security frameworks stop at authentication and authorization, Shiro provides much more. Next we’ll talk about Shiro’s advanced Session Management capabilities.

## Session Management

Apache Shiro provides something unique in the world of security frameworks: a consistent Session API usable in any application and any architectural tier. That is, Shiro enables a Session programming paradigm for any application - from small daemon standalone applications to the largest clustered web applications. This means that application developers who wish to use sessions are no longer forced to use Servlet or EJB containers if they don’t need them otherwise. Or, if using these containers, developers now have the option of using a unified and consistent session API in any tier, instead of servlet or EJB-specific mechanisms.

But perhaps one of the most important benefits of Shiro’s sessions is that they are container-independent. This has subtle but extremely powerful implications. For example, let’s consider session clustering. How many container-specific ways are there to cluster sessions for fault-tolerance and failover? Tomcat does it differently than Jetty, which does it differently than Websphere, etc. But with Shiro sessions, you obtain a container-independent clustering solution. Shiro’s architecture allows for pluggable Session data stores, such as enterprise caches, relational databases, NoSQL systems and more. This means that you can configure session clustering once and it will work the same way regardless of your deployment environment - Tomcat, Jetty, JEE Server or standalone application. There is no need to reconfigure your app based on how you deploy your application.

Another benefit of Shiro’s sessions is session data can be shared across client technologies if desired. For example, a Swing desktop client can participate in the same web application session if desired - useful if the end-user is using both simultaneously. So how do you access a Subject’s session in any environment? There are two Subject methods as shown in the example below.

**Listing 10. Subject’s Session**

Session session = subject.getSession();

Session session = subject.getSession(boolean create);

As you can see, the methods are identical in concept to the HttpServletRequest API. The first method will return the Subject’s existing Session, or if there isn’t one yet, it will create a new one and return it. The second method accepts a boolean argument that determines whether or not a new Session will be created if it does not yet exist. Once you acquire the Subject’s Session, you can use it almost identically to an HttpSession. The Shiro team felt that the HttpSession API was most comfortable to Java developers, so we retained much of its feel. The big difference, of course, is that you can use Shiro Sessions in any application, not just web applications. Listing 11 shows this familiarity.

**Listing 11. Session methods**

Session session = subject.getSession();

session.getAttribute(“key”, someValue);  
Date start = session.getStartTimestamp();  
Date timestamp = session.getLastAccessTime();  
session.setTimeout(millis);  
...

## Cryptography

Cryptography is the process of hiding or obfuscating data so prying eyes can’t understand it. Shiro’s goal in cryptography is to simplify and make usable the JDK’s cryptography support.

It is important to note that cryptography is not specific to Subjects in general, so it is one area of Shiro’s API that is not Subject-specific. You can use Shiro’s cryptography support anywhere, even if a Subject is not being used. The two areas where Shiro really focuses its cryptography support is in the areas of cryptographic hashes (aka message digests) and cryptographic ciphers. Let's take a look at these two in more detail.

### Hashing

If you have used the JDK’s [MessageDigest](http://download.oracle.com/javase/6/docs/api/java/security/MessageDigest.html) class, you quickly realize that it is a bit cumbersome to work with. It has an awkward static method factory-based API instead of an object-oriented one, and you are forced to catch checked exceptions that may never need to be caught. If you need to hex-encode or Base64-encode message digest output, you’re on your own - there’s no standard JDK support for either. Shiro addresses these issues in a clean and intuitive hashing API.

For example, let’s consider the relatively common case of MD5-hashing a file and determining the hex value of that hash. Called a ‘checksum’, this is used regularly when providing file downloads - users can perform their own MD5 hash on the downloaded file and assert that their checksum matches the one on the download site. If they match, the user can sufficiently assume that the file hasn’t been tampered with in transit.

Here is how you might try to do this **without** Shiro:

1. Convert the file to a byte array. There is nothing in the JDK to assist with this, so you’ll need to create a helper method that opens a [FileInputStream](http://download.oracle.com/javase/6/docs/api/java/io/FileInputStream.html), uses a byte buffer, and throws the appropriate IOExceptions, etc.
2. Use the [MessageDigest](http://download.oracle.com/javase/6/docs/api/java/security/MessageDigest.html) class to hash the byte array, dealing with the appropriate exceptions, as shown in Listing 12 below.
3. Encode the hashed byte array into hex characters. There is nothing in the JDK to assist with this either, so you’ll need to create another helper method and probably use bitwise operations and bitshifting in your implementation.

**Listing 12. JDK’s MessageDigest**

try {

    MessageDigest md = MessageDigest.getInstance("MD5");

    md.digest(bytes);

    byte[] hashed = md.digest();

} catch (NoSuchAlgorithmException e) {

    e.printStackTrace();

}

That’s a significant amount of work for something so simple and relatively common. Now here’s how to do the exact same thing **with** Shiro.

String hex = new Md5Hash(myFile).toHex();

It is remarkably simpler and easier to understand what is going on when you use Shiro to simplify all of that work. SHA-512 hashing and Base64-encoding of passwords is just as easy.

String encodedPassword =

    new Sha512Hash(password, salt, count).toBase64();

You can see how much Shiro simplifies hashing and encoding, saving you a bit of sanity in the process.

### Ciphers

Ciphers are cryptographic algorithms that can reversibly transform data using a key. We use them to keep data safe, especially when transferring or storing data, times when data is particularly susceptible to prying eyes.

If you’ve ever used the JDK Cryptography APIs, and in particular the javax.crypto.Cipher class, you know that it can be an incredibly complex beast to tame. For starters, every possible Cipher configuration is always represented by an instance of javax.crypto.Cipher. Need to do public/private key cryptography? You use the Cipher. Need to use a Block Cipher for streaming operations? You use the Cipher. Need to create an AES 256-bit Cipher to secure data? You use the Cipher. You get the idea.

And how do you create the Cipher instance you need? You create a complex, unintuitive token-delimited String of cipher options, called a “transformation string”, and you pass this string to a Cipher.getInstance static factory method. With this cipher option String approach, there is no type-safety to ensure you’re using valid options. This also implicitly means that there is no JavaDoc to help you understand relevant options. And you are further required to deal with checked exceptions in case your String is incorrectly formulated, even if you know that the configuration is correct. As you can see, working with JDK Ciphers is quite a cumbersome task. These techniques were once standard for Java APIs a long time ago, but times have changed, and we want a much easier approach.

Shiro tries to simplify the entire concept of cryptographic ciphers by introducing its CipherService API. A CipherService is what most developers want when securing data: a simple, stateless, thread-safe API that can encrypt or decrypt data in its entirety in one method call. All you need to do is provide your key, and you can encrypt or decrypt as necessary. For example, 256-bit AES encryption can be used as shown in the Listing 13 below.

**Listing 13. Apache Shiro’s Encryption API**

AesCipherService cipherService = new AesCipherService();

cipherService.setKeySize(256);

//create a test key:  
byte[] testKey = cipherService.generateNewKey();

//encrypt a file’s bytes:  
byte[] encrypted =  
    cipherService.encrypt(fileBytes, testKey);

The Shiro example is simpler compared to the JDK’s Cipher API:

* You can instantiate a CipherService directly - no strange or confusing factory methods.
* Cipher configuration options are represented as JavaBeans-compatible getters and setters - there is no strange and difficult to understand “transformation string”.
* Encryption and decryption is executed in a single method call.
* No forced checked exceptions. Catch Shiro’s CryptoException if you want.

There are other benefits to Shiro’s CipherService API, such as the ability to support both byte array-based encryption/decryption (called ‘block’ operations) as well as stream-based encryption/decryption (for example, encrypting audio or video).

Java Cryptography doesn’t need to be painful. Shiro’s Cryptography support is meant to simplify your efforts to keep your data secure.

## Web Support

Last, but not least, we’ll briefly introduce Shiro’s web support. Shiro ships with a robust web support module to help secure web applications. It is simple to set up Shiro for a web application. The only thing necessary is to define a Shiro Servlet Filter in web.xml. Listing 14 shows this code.

**Listing 14. ShiroFilter in web.xml**

<filter>

    <filter-name>ShiroFilter</filter-name>

    <filter-class>

        org.apache.shiro.web.servlet.IniShiroFilter

    </filter-class>

    <!-- no init-param means load the INI config

        from classpath:shiro.ini -->

</filter>

<filter-mapping>  
     <filter-name>ShiroFilter</filter-name>  
     <url-pattern>/\*</url-pattern>  
</filter-mapping>

This filter can read the aforementioned shiro.ini config so you have a consistent configuration experience regardless of deployment environment. Once configured, the Shiro Filter will filter every request and ensure the request-specific Subject is accessible during the request. And because it filters every request, you can perform security-specific logic to ensure only requests that meet certain criteria are allowed through.

### URL-Specific Filter Chains

Shiro supports security-specific filter rules through its innovative URL filter chaining capability. It allows you to specify ad-hoc filter chains for any matching URL pattern. This means you have a great deal of flexibility in enforcing security rules (or combinations of rules) using Shiro’s filtering mechanisms - much more so than you could achieve defining filters in web.xml alone. Listing 15 shows the configuration snippet in Shiro INI.

**Listing 15. Path-specific Filter Chains**

[urls]

/assets/\*\* = anon

/user/signup = anon

/user/\*\* = user

/rpc/rest/\*\* = perms[rpc:invoke], authc

/\*\* = authc

As you can see, there is a [urls] INI section available to web applications. For each line, the values on the left of the equals sign represent a context-relative web application path. The values on the right define a Filter chain - an ordered, comma delimited list of Servlet filters to execute for the given path. Each filter is a normal Servlet Filter, but the filter names you see above (anon, user, perms, authc) are special security-related filters that Shiro provides out-of-the-box. You can mix and match these security filters to create a very custom security experience. You can also specify any other existing Servlet Filter you may have.

How much nicer is this compared to using web.xml, where you define a block of filters and then a separate disconnected block of filter patterns? Using Shiro’s approach, it is much easier to see exactly the filter chain that is executed for a given matching path. If you wanted to, you could define only the Shiro Filter in web.xml and define all of your other filters and filter chains in shiro.ini for a much more succinct and easy to understand filter chain definition mechanism than web.xml. Even if you didn’t use any of Shiro’s security features, this one small convenience alone can make Shiro worth using.

### JSP Tag Library

Shiro also provides a JSP tag library that allows you to control the output of your JSP pages based on the current Subject’s state. One common example of where this is useful is in displaying the ‘Hello <username>’ text after a user is logged-in. But if they are anonymous, you might want to show something else, like “Hello! Register Today!” instead. Listing 16 shows how you might support this with Shiro’s JSP tags.

**Listing 16. JSP Taglib Example**

<%@ taglib prefix="shiro"

    uri="http://shiro.apache.org/tags" %>

...

<p>Hello

<shiro:user>

    <!-- shiro:principal prints out the Subject’s main

        principal - in this case, a username: -->

    <shiro:principal/>!

</shiro:user>

<shiro:guest>

    <!-- not logged in - considered a guest. Show

        the register link: -->

    ! <a href=”register.jsp”>Register today!</a>

</shiro:guest>

</p>

There are other tags that allow you to include output based on what roles they have (or don’t have), what permissions are assigned (or not assigned), and whether they are authenticated, remembered from “Remember Me” services, or an anonymous guest.

There are many other web-specific features that Shiro supports, like simple “Remember Me” services, REST and BASIC authentication, and of course transparent HttpSession support if you want to use Shiro’s native enterprise sessions. Please see the [Apache Shiro web documentation](http://shiro.apache.org/web.html)for more.

### Web Session Management

Finally, it is interesting to point out Shiro’s support for sessions in a web environment.

#### Default Http Sessions

For web applications, Shiro defaults its session infrastructure to use the existing Servlet Container sessions that we’re all used to. That is, when you call the methods subject.getSession() and subject.getSession(boolean) Shiro will return Session instances backed by the Servlet Container’s HttpSession instance. The beauty of this approach is that business-tier code that calls subject.getSession() interacts with a Shiro Session instance - it has no ‘knowledge’ that it is working with a web-based HttpSession object. This is a very good thing when maintaining clean separation across architectural tiers.

#### Shiro’s Native Sessions in the Web Tier

If you’ve enabled Shiro’s native session management in a web application because you need Shiro’s enterprise session features (like container-independent clustering), you of course want the HttpServletRequest.getSession() and HttpSession API to work with the ‘native’ sessions and not the servlet container sessions. It would be very frustrating if you had to refactor any code that uses the HttpServletRequest and HttpSession API to instead use Shiro’s Session API. Shiro of course would never expect you to do this. Instead, Shiro fully implements the Session part of the Servlet specification to support native sessions in web applications. This means that whenever you call a corresponding HttpServletRequest or HttpSession method call, Shiro will delegate these calls to its internal native Session API. The end result is that you don’t have to change your web code even if you’re using Shiro’s ‘native’ enterprise session management - a very convenient (and necessary) feature indeed.

## Additional Features

There are other features in the Apache Shiro framework that are useful to securing Java applications, such as:

* Threading and Concurrency support for maintaining Subjects across threads (Executor and ExecutorService support)
* Callable and Runnable support for executing logic as a specific Subject
* “Run As” support for assuming the identity of another Subject (e.g. useful in administrative applications)
* Test harness support, making it very easy to have full testing of Shiro secured-code in unit and integration tests

## Framework Limitations

As much as we might like it to be, Apache Shiro is not a ‘silver bullet’ - it won’t solve every security problem effortlessly. There are things that Shiro does not address that might be worth knowing:

* **Virtual Machine-level concerns**: Apache Shiro does not currently deal with Virtual Machine level security, such as the ability to prevent certain classes from loading in a class loader based on an access control policy. However, it is not unconceivable that Shiro could integrate with the existing JVM security operations - it is just that no one has contributed such work to the project.
* **Multi-Stage Authentication**: Shiro does not currently natively support ‘multi-stage’ authentication, where a user might login via one mechanism, only to be asked to then log-in again with a different mechanism. This has been accomplished in Shiro-based applications however by the application collecting all required information up front and then interacting with Shiro. It is a real possibility that this feature will be supported in a future Shiro version.
* **Realm Write Operations**: Currently all Realm implementations support ‘read’ operations for acquiring authentication and authorization data to perform logins and access control. ‘Write’ operations, like creating user accounts, groups and roles, or associating users with roles groups and permissions, are not supported. This is because the data model to support these operations varies dramatically across applications and it would be difficult to enforce a ‘write’ API on all Shiro users.

## Upcoming Features

Apache Shiro’s community continues to grow every day, and with it, so does Shiro’s features. In upcoming versions, you are likely to see:

* Cleaner Web Filter mechanism that allows more pluggable filtering support without subclassing.
* More pluggable default Realm implementations favoring composition over inheritance. You will be able to plug in components that look up authentication and authorization data instead of requiring that you subclass a Shiro Realm implementation
* Robust [OpenID](http://openid.net/) and [OAuth](http://oauth.net/) (and possibly Hybrid) client support
* Captcha support
* Easier configuration for 100% stateless applications (e.g. many REST environments).
* Multi-stage authentication via a request/response protocol.
* Coarse-grained authorization via an AuthorizationRequest.
* [ANTLR](http://www.antlr.org/) grammar for security assertion queries (e.g. (‘role(admin) && (guest || !group(developer))’)

## Summary

Apache Shiro is a full-featured, robust, and general-purpose Java security framework that you can use to secure your applications. By simplifying four areas of application security, namely, Authentication, Authorization, Session Management, and Cryptography, application security is much easier to understand and implement in real applications. Shiro’s simple architecture and JavaBeans compatibility allow it to be configured and used in almost any environment. Additional web support and auxiliary features like multithreading and test support round out the framework to provide what could be your ‘one stop shop’ for application security. Apache Shiro’s development team continues to move forward, refining the codebase and supporting the community. With continued open-source and commercial adoption, Shiro is only expected to grow stronger.

## Resources

* Apache Shiro’s [homepage](http://shiro.apache.org/).
* Shiro’s [Download Page](http://shiro.apache.org/download.html), with additional information for Maven and Ant+Ivy users.
* Apache Shiro’s [Documentation Page](http://shiro.apache.org/documentation.html), with Guides and a Reference Manual
* An [Apache Shiro presentation video and slides](http://shiro.apache.org/2011/01/06/apache-shiro-video-and-slide-from-sf-jug-presentation.html), presented by the project’s PMC Chair, Les Hazlewood.
* Other Apache Shiro [articles and presentations](http://shiro.apache.org/articles.html)
* Apache Shiro [mailing lists](http://shiro.apache.org/mailing-lists.html) and [forums](http://shiro.apache.org/forums.html).
* [Katasoft](http://www.katasoft.com/) - a company offering Apache Shiro professional support and application security products.

## About the Author

Les Hazlewood is the Apache Shiro PMC Chair and co-founder and CTO of [Katasoft](http://www.katasoft.com/), a start-up focusing on application security products and Apache Shiro professional support. Les has 10 years of experience as a professional Java developer and enterprise architect, with previous senior roles at Bloomberg, Delta Airlines, and JBoss. Les has been actively involved in Open Source development for more than 9 years, committing or contributing to projects like the Spring Framework, Hibernate, JBoss, OpenSpaces, and of course JSecurity, Apache Shiro's predecessor. Les currently lives in San Mateo, CA and practices Kendo and studies Japanese when he is not programming.