# Apache Shiro Configuration



Shiro is designed to work in any environment, from simple command-line applications to the largest enterprise clustered applications. Because of this diversity of environments, there are a number of configuration mechanisms that are suitable for configuration. This section covers the configuration mechanisms that are supported by Shiro core only.

**Many Configuration Options**
Shiro's `SecurityManager` implementations and all supporting components are all JavaBeans compatible. This allows Shiro to be configured with practically any configuration format such as regular Java, XML (Spring, JBoss, Guice, etc), YAML, JSON, Groovy Builder markup, and more.

## Programmatic Configuration

The absolute simplest way to create a SecurityManager and make it available to the application is to create an `org.apache.shiro.mgt.DefaultSecurityManager` and wire it up in code. For example:

```
Realm realm = //instantiate or acquire a Realm instance.  We'll discuss Realms later.

SecurityManager securityManager = new DefaultSecurityManager(realm);

//Make the SecurityManager instance available to the entire application via static memory:
SecurityUtils.setSecurityManager(securityManager);
```

Surprisingly, after only 3 lines of code, you now have a fully functional Shiro environment suitable for many applications. How easy was that!?

### SecurityManager Object Graph

As discussed in the Architecture chapter, Shiro's `SecurityManager` implementations are essentially a modular object graph of nested security-specific components. Because they are also JavaBeans-compatible, you can call any of the nested components `getter` and `setter` methods to configure the `SecurityManager` and its internal object graph.

For example, if you wanted to configure the `SecurityManager` instance to use a custom `SessionDAO` to customize Session Management, you could set the `SessionDAO` directly with the nested SessionManager's `setSessionDAO` method:

```
...
```

```
DefaultSecurityManager securityManager = new DefaultSecurityManager(realm);

SessionDAO sessionDAO = new CustomSessionDAO();

((DefaultSessionManager)securityManager.getSessionManager()).setSessionDAO(sessionDAO);
...
```

Using direct method invocations, you can configure any part of the `SecurityManager`'s object graph.

But, as simple as programmatic customization is, it does not represent the ideal configuration for most real world applications. There are a few reasons why programmatic configuration may not be suitable for your application:

- It requires you to know about and instantiate a direct implementation. It would be nicer if you didn't have to know about concrete implementations and where to find them.
- Because of Java's type-safe nature, you're required to cast objects obtained via `get*` methods to their specific implementation. So much casting is ugly, verbose, and tightly-couples you to implementation classes.
- The `SecurityUtils.setSecurityManager` method call makes the instantiated `SecurityManager` instance a VM static singleton, which, while fine for many applications, would cause problems if more than one Shiro-enabled application was running on the same JVM. It could be better if the instance was an application singleton, but not a static memory reference.
- It requires you to recompile your application every time you want to make a Shiro configuration change.

However, even with these caveats, the direct programmatic manipulation approach could still be valuable in memory-constrained environments, like smart-phone applications. If your application does not run in a memory-constrained environment, you'll find text-based configuration to be easier to use and read.

## INI Configuration

Most applications instead benefit from text-based configuration that could be modified independently of source code and even make things easier to understand for those not intimately familiar with Shiro's APIs.

To ensure a common-denominator text-based configuration mechanism that can work in all environments with minimal 3rd party dependencies, Shiro supports the INI format to build the `SecurityManager` object graph and its supporting components. INI is easy to read, easy to configure, and is simple to set-up and suits most applications well.

### Creating a SecurityManager from INI

Here are two examples of how to build a SecurityManager based on INI configuration.

#### SecurityManager from an INI resource

We can create the SecurityManager instance from an INI resource path. Resources can be acquired from the file system, classpath, or URLs when prefixed with `file:`, `classpath:`, or `url:` respectively. This example uses a `Factory` to ingest a `shiro.ini` file from the root of the classpath and return the `SecurityManager` instance:

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.util.Factory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.config.IniSecurityManagerFactory;

...

Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.ini");
SecurityManager securityManager = factory.getInstance();
SecurityUtils.setSecurityManager(securityManager);
```

#### SecurityManager from an INI instance

The INI configuration can be constructed programmatically as well if desired via the `org.apache.shiro.config.Ini` class. The Ini class functions similarly to the JDK `java.util.Properties` class, but additionally supports segmentation by section name.

For example:

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.util.Factory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.config.Ini;
import org.apache.shiro.config.IniSecurityManagerFactory;

...

Ini ini = new Ini();
//populate the Ini instance as necessary
...
Factory<SecurityManager> factory = new IniSecurityManagerFactory(ini);
SecurityManager securityManager = factory.getInstance();
```

```
SecurityUtils.setSecurityManager(securityManager);
```

Now that we know how to construct a `SecurityManager` from INI configuration, let's take a look at exactly how to define a Shiro INI configuration.

## INI Sections

INI is basically a text configuration consisting of key/value pairs organized by uniquely-named sections. Keys are unique per section only, not over the entire configuration (unlike the JDK Properties). Each section may be viewed like a single `Properties` definition however.

Commented lines can start with either with an Octothorpe (# - aka the 'hash', 'pound' or 'number' sign) or a Semi-colon (';')

Here is an example of the sections understood by Shiro:

```
# =======================
# Shiro INI configuration
# =======================

[main]
# Objects and their properties are defined here,
# Such as the securityManager, Realms and anything
# else needed to build the SecurityManager

[users]
# The 'users' section is for simple deployments
# when you only need a small number of statically-defined
# set of User accounts.

[roles]
# The 'roles' section is for simple deployments
# when you only need a small number of statically-defined
# roles.

[urls]
# The 'urls' section is used for url-based security
# in web applications.  We'll discuss this section in the
# Web documentation
```

### [main]

The `[main]` section is where you configure the application's `SecurityManager` instance and any of its dependencies, such as Realms.

Configuring object instances like the SecurityManager or any of its dependencies sounds like a difficult thing to do with INI, where we can only use name/value pairs. But through a little bit of convention and understanding of object graphs, you'll find that you can do quite a lot. Shiro uses these assumptions to enable a simple yet fairly concise configuration mechanism.

We often like to refer to this approach as "poor man's" Dependency Injection, and although not as powerful as full-blown Spring/Guice/JBoss XML files, you'll find it gets quite a lot done without much complexity. Of course those other configuration mechanism are available as well, but they're not required to use Shiro.

Just to whet your appetite, here is an example of a valid `[main]` configuration. We'll cover it in detail below, but you might find that you understand quite a bit of what is going on already by intuition alone:

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher

myRealm = com.company.security.shiro.DatabaseRealm
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
myRealm.password = secret
myRealm.credentialsMatcher = $sha256Matcher

securityManager.sessionManager.globalSessionTimeout = 1800000
```

#### Defining an object

Consider the following `[main]` section snippet:

```
[main]
myRealm = com.company.shiro.realm.MyRealm
...
```

This line instantiates a new object instance of type `com.company.shiro.realm.MyRealm` and makes that object available under the **myRealm** name for further reference and configuration.

If the object instantiated implements the `org.apache.shiro.util.Nameable` interface, then the the `Nameable.setName` method will be invoked on the object with the name value (**myRealm** in this example).

#### Setting object properties

##### Primitive Values

Simple primitive properties can be assigned just by using the equals sign:

```
...
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
...
```

these lines of configuration translate into method calls:

```
...
myRealm.setConnectionTimeout(30000);
myRealm.setUsername("jsmith");
...
```

How is this possible? It assumes that all objects are Java Beans-compatible POJOs.

Under the covers, Shiro by default uses Apache Commons BeanUtils to do all the heavy lifting when setting these properties. So although INI values are text, BeanUtils knows how to convert the string values to the proper primitive types and then invoke the corresponding JavaBeans setter method.

##### Reference Values

What if the value you need to set is not a primitive, but another object? Well, you can use a dollar sign ($) to reference a previously-defined instance. For example:

```
...
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
myRealm.credentialsMatcher = $sha256Matcher
...
```

This simply locates the object defined by the name **sha256Matcher** and then uses BeanUtils to set that object on the **myRealm** instance (by calling the `myRealm.setCredentialsMatcher(sha256Matcher)` method).

##### Nested Properties

Using dotted notation on the left side of the INI line's equals sign, you can traverse an object graph to get to the final object/property that you want set. For example, this config line:

```
...
securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

Translates (by BeanUtils) into the following logic:

```
securityManager.getSessionManager().setGlobalSessionTimeout(1800000);
```

The graph traversal can be as deep as necessary: `object.property1.property2....propertyN.value = blah`

**BeanUtils Property Support**
Any property assignment operation supported by the BeanUtils.setProperty method will work in Shiro's [main] section, including set/list/map element assignments. See the Apache Commons BeanUtils Website and documentation for more information.

##### Byte Array Values

Because raw byte arrays can't be specified natively in a text format, we must use a text encoding of the byte array. The values can be specified either as a Base64 encoded string (the default) or as a Hex encoded string. The default is Base64 because Base64 encoding requires less actual text to represent values - it has a larger encoding alphabet, meaning your tokens are shorter (a bit nicer for text config).

```
# The 'cipherKey' attribute is a byte array.    By default, text values
# for all byte array properties are expected to be Base64 encoded:

securityManager.rememberMeManager.cipherKey = kPH+bIxk5D2deZiIxcaaaA==
...
```

However, if you prefer to use Hex encoding instead, you must prefix the String token with `0x` ('zero' 'x'):

```
securityManager.rememberMeManager.cipherKey = 0x3707344A4093822299F31D008
```

#### Collection Properties

Lists, Sets and Maps can be set like any other property - either directly or as a nested property. For sets and lists, just specify a comma-delimited set of values or object references.

For example, some SessionListeners:

```
sessionListener1 = com.company.my.SessionListenerImplementation
...
sessionListener2 = com.company.my.other.SessionListenerImplementation
...
securityManager.sessionManager.sessionListeners = $sessionListener1, $sessionListener2
```

For Maps, you specify a comma-delimited list of key-value pairs, where each key-value pair is delimited by a colon ':'

```
object1 = com.company.some.Class
object2 = com.company.another.Class
...
anObject = some.class.with.a.Map.property

anObject.mapProperty = key1:$object1, key2:$object2
```

In the above example, the object referenced by `$object1` will be in the map under the String key `key1`, i.e. `map.get("key1")` returns `object1`. You can also use other objects as the keys:

```
anObject.map = $objectKey1:$objectValue1, $objectKey2:$objectValue2
...
```

#### Considerations

##### Order Matters

The INI format and conventions above are very convenient and easy to understand, but it is not as powerful as other text/XML-based configuration mechanisms. The most important thing to understand when using the above mechanism is that **Order Matters!**

**Be Careful**
Each object instantiation and each value assignment is executed *in the order they occur in the [main] section*. These lines ultimately translate to a JavaBeans getter/setter method invocation, and so those methods are invoked in the same order!

Keep this in mind when writing your configuration.

##### Overriding Instances

Any object can be overridden by a new instance defined later in the configuration. So for example, the 2nd `myRealm` definition would overwrite the first:

```
...
myRealm = com.company.security.MyRealm
...
myRealm = com.company.security.DatabaseRealm
...
```

This would result in `myRealm` being a `com.company.security.DatabaseRealm` instance and the previous instance will never be used (and garbage collected).

##### Default SecurityManager

You may have noticed in the complete example above that the SecurityManager instance's class isn't defined, and we jumped right in to just setting a nested property:

```
myRealm = ...

securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

This is because the `securityManager` instance is a special one - it is already instantiated for you and ready to go so you don't need to know the specific `SecurityManager` implementation class to instantiate.

Of course, if you actually *want* to specify your own implementation, you can, just define your implementation as specified in the

"Overriding Instances" section above:

```
...
securityManager = com.company.security.shiro.MyCustomSecurityManager
...
```

Of course, this is rarely needed - Shiro's SecurityManager implementations are very customizable and can typically be configured with anything necessary. You might want to ask yourself (or the user list) if you really need to do this.
users

**[users]**

The `[users]` section allows you to define a static set of user accounts. This is mostly useful in environments with a very small number of user accounts or where user accounts don't need to be created dynamically at runtime. Here's an example:

```
[users]
admin = secret
lonestarr = vespa, goodguy, schwartz
darkhelmet = ludicrousspeed, badguy, schwartz
```

**Automatic IniRealm**
Just defining non-empty [users] or [roles] sections will automatically trigger the creation of an `org.apache.shiro.realm.text.IniRealm` instance and make it available in the [main] section under the name `iniRealm`. You can configure it like any other object as described above.

#### Line Format

Each line in the [users] section must conform to the following format:

`username` = `password`, *roleName1*, *roleName2*, ..., *roleNameN*

- The value on the left of the equals sign is the username
- The first value on the right of the equals sign is the user's password. A password is required.
- Any comma-delimited values after the password are the names of roles assigned to that user. Role names are optional.

#### Encrypting Passwords

If you don't want the [users] section passwords to be in plain-text, you can encrypt them using your favorite hash algorithm (MD5, Sha1, Sha256, etc) however you like and use the resulting string as the password value. By default, the password string is expected to be Hex encoded, but can be configured to be Base64 encoded instead (see below).

**Easy Secure Passwords**
To save time and use best-practices, you might want to use Shiro's Command Line Hasher, which will hash passwords as well as any other type of resource. It is especially convenient for encrypting INI `[users]` passwords.

Once you've specified the hashed text password values, you have to tell Shiro that these are encrypted. You do that by configuring the implicitly created `iniRealm` in the [main] section to use an appropriate `CredentialsMatcher` implementation corresponding to the hash algorithm you've specified:

```
[main]
...
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
iniRealm.credentialsMatcher = $sha256Matcher
...

[users]
# user1 = sha256-hashed-hex-encoded password, role1, role2, ...
user1 = 2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b, role1, role2, ...
```

You can configure any properties on the `CredentialsMatcher` like any other object to reflect your hashing strategy, for example, to specify if salting is used or how many hash iterations to execute. See the `org.apache.shiro.authc.credential.HashedCredentialsMatcher` JavaDoc to better understand hashing strategies and if they might be useful to you.

For example, if your users' password strings were Base64 encoded instead of the default Hex, you could specify:

```
[main]
...
# true = hex, false = base64:
sha256Matcher.storedCredentialsHexEncoded = false
```

**[roles]**

The `[roles]` section allows you to associate Permissions with the roles defined in the [users] section. Again, this is useful in environments with a small number of roles or where roles don't need to be created dynamically at runtime. Here's an example:

```
[roles]
# 'admin' role has all permissions, indicated by the wildcard '*'
admin = *
# The 'schwartz' role can do anything (*) with any lightsaber:
schwartz = lightsaber:*
# The 'goodguy' role is allowed to 'drive' (action) the winnebago (type) with
# license plate 'eagle5' (instance specific id)
goodguy = winnebago:drive:eagle5
```

#### Line Format

Each line in the [roles] section must must define a role-to-permission(s) key/value mapping with in the following format:

`rolename` = *permissionDefinition1*, *permissionDefinition2*, ..., *permissionDefinitionN*

where *permissionDefinition* is an arbitrary String, but most people will want to use strings that conform to the `org.apache.shiro.authz.permission.WildcardPermission` format for ease of use and flexibility. See the Permissions documentation for more information on Permissions and how you can benefit from them.

**Internal commas**
Note that if an individual *permissionDefinition* needs to be internally comma-delimited (e.g. `printer:5thFloor:print,info`), you will need to surround that definition with double quotes (") to avoid parsing errors:
`"printer:5thFloor:print,info"`

**Roles without Permissions**
If you have roles that don't require permission associations, you don't need to list them in the [roles] section if you don't want to. Just defining the role names in the [users] section is enough to create the role if it does not exist yet.

#### [urls]

This section and its options is described in the Web chapter.

## Lend a hand with documentation

While we hope this documentation helps you with the work you're doing with Apache Shiro, the community is improving and expanding the documentation all the time. If you'd like to help the Shiro project, please consider corrected, expanding, or adding documentation where you see a need. Every little bit of help you provide expands the community and in turn improves Shiro.

The easiest way to contribute your documentation is to send it to the User Forum or the User Mailing List.