Session Management

Related Content

Getting Started

Resources, guides and tutorials for new Shiro users.
Read More >>

10-Minute Shiro Tutorial

Try Apache Shiro for yourself in under 10 minutes.
Read More >>

Web App Tutorial

Step-by-step tutorial for securing a web application with Shiro.
Read More >>

Java Authentication Guide

Learn how Authentication in Java is performed in Shiro.
Read More >>

Java Authorization Guide

Learn how Shiro handles access control in Java.
Read More >>

Apache Shiro offers something unique in the world of security frameworks: a complete enterprise-grade Session solution for any application, from the simplest command-line and smart phone applications to the largest clustered enterprise web applications.

This has large implications for many applications - until Shiro, if you required session support, you were required to deploy your application in a web container or use EJB Stateful Session Beans. Shiro's Session support is much simpler to use and manage than either of these two mechanisms, and it is available in any application, regardless of container.

And even if you deploy your application in a Servlet or EJB container, there are still compelling reasons to use Shiro's Session support instead of the container's. Here is a list of the most desirable features provided by Shiro's session support:

Features

- **POJO/J2SE based (IoC friendly)** - Everything in Shiro (including all aspects of Sessions and Session Management) is interface-based and implemented with POJOs. This allows you to easily configure all session components with any JavaBeans-compatible configuration format, like JSON, YAML, Spring XML or similar mechanisms. You can also easily extend Shiro's components or write your own as necessary to fully customize session management functionality.
- **Easy Custom Session Storage** - Because Shiro's Session objects are POJO-based, session data can be easily stored in any number of data sources. This allows you to customize exactly where your application's session data resides - for example, the file system, in memory, in a networked distributed cache, a relational database, or proprietary data store.
- **Container-Independent Clustering!** - Shiro's sessions can be easily clustered using any of the readily-available networked caching products, like Ehcache + Terracotta, Coherence, GigaSpaces, et. al. This means you can configure session clustering for Shiro once and only once, and no matter what container you deploy to, your sessions will be clustered the same way. No need for container-specific configuration!
- **Heterogeneous Client Access** - Unlike EJB or Web sessions, Shiro sessions can be 'shared' across various client technologies. For example, a desktop application could 'see' and 'share' the same physical session used by the same user in a web application. We are unaware of any framework other than Shiro that can support this.
- **Event Listeners** - Event listeners allow you to listen to lifecycle events during a session's lifetime. You can listen for these events and react to them for custom application behavior - for example, updating a user record when their session expires.
- **Host Address Retention** – Shiro Sessions retain the IP address or host name of the host from where the session was initiated. This allows you to determine where the user is located and react accordingly (often useful in intranet environments where IP association is deterministic).

- **Inactivity/Expiration Support** – Sessions expire due to inactivity as expected, but they can be prolonged via a `touch()` method to keep them 'alive' if desired. This is useful in Rich Internet Application (RIA) environments where the user might be using a desktop application, but may not be regularly communicating with the server, but the server session should not expire.
- **Transparent Web Use** - Shiro's web support fully implements and supports the Servlet 2.5 specification for Sessions (`HttpSession` interface and all of it's associated APIs). This means you can use Shiro sessions in existing web applications and you don't need to change any of your existing web code.
- **Can be used for SSO** - Because Shiro session's are POJO based, they are easily stored in any data source, and they can be 'shared' across applications if needed. We call this 'poor man's SSO', and it can be used to provide a simple sign-on experience since the shared session can retain authentication state.

Using Sessions

Like almost everything else in Shiro, you acquire a `Session` by interacting with the currently executing `Subject`:

```
Subject currentUser = SecurityUtils.getSubject();

Session session = currentUser.getSession();
session.setAttribute( "someKey", someValue);
```

The `subject.getSession()` method is a shortcut for calling `currentUser.getSubject(true)`.

For those familiar with `HttpServletRequest` API, the `Suject.getSession(boolean create)` method functions the same way as the `HttpServletRequest.getSession(boolean create)` method:

- If the `Subject` already has a `Session`, the boolean argument is ignored and the `Session` is returned immediately
- If the `Subject` does not yet have a `Session` and the `create` boolean argument is `true`, a new session will be created and returned.
- If the `Subject` does not yet have a `Session` and the `create` boolean argument is `false`, a new session will not be created and `null` is returned.

Any Application
`getSession` calls work in any application, even non-web applications.

`subject.getSession(false)` can be used to good effect when developing framework code to ensure a Session isn't created unnecessarily.

Once you acquire a Subject's `Session` you can do many things with it, like set or retrieve attributes, set its timeout, and more. See the Session JavaDoc to see what is possible with an individual session.

The SessionManager

The SessionManager, as its name might imply, manages Sessions for *all* subjects in an application - creation, deletion, inactivity and validation, etc. Like other core architectural components in Shiro, the `SessionManager` is a top-level component maintained by the `SecurityManager`.

The default `SecurityManager` implementation defaults to using a `DefaultSessionManager` out of the box. The `DefaultSessionManager` implementation provides all of the enterprise-grade session management features needed for an application, like Session validation, orphan cleanup, etc. This can be used in any application.

Web Applications
Web applications use different `SessionManager` implementations. Please see the Web documentation for web-specific Session Management information.

Like all other components managed by the `SecurityManager`, the `SessionManager` can be acquired or set via JavaBeans-style getter/setter methods on all of Shiro's default `SecurityManager` implementations (`getSessionManager()`/`setSessionManager()`). Or for example, if using `shiro.ini` Configuration:

Configuring a new SessionManager in shiro.ini

```
[main]
...
sessionManager = com.foo.my.SessionManagerImplementation
securityManager.sessionManager = $sessionManager
```

But creating a `SessionManager` from scratch is a complicated task and not something that most people will want to do themselves. Shiro's out-of-the-box `SessionManager` implementations are highly customizable and configurable and will suit most needs. Most of the rest of this documentation assumes that you will be using Shiro's default `SessionManager` implementations when covering configuration options, but note that you can essentially create or plug-in nearly anything you wish.

Session Timeout

By default, Shiro's `SessionManager` implementations default to a 30 minute session timeout. That is, if any `Session` created remains idle (unused, where its `lastAccessedTime` isn't updated) for 30 minutes or more, the `Session` is considered expired and will not be allowed to be used anymore.

You can set the default `SessionManager` implementation's `globalSessionTimeout` property to define the default timeout value for all sessions. For example, if you wanted the timeout to be an hour instead of 30 minutes:

Setting the Default Session Timeout in shiro.ini

```
[main]
...
# 3,600,000 milliseconds = 1 hour
securityManager.sessionManager.globalSessionTimeout = 3600000
```

Per-Session Timeout

The above `globalSessionTimeout` value is the default for all newly created `Sessions`. You can control session timeout on a per-Session basis by setting the individual Session's `timeout` value. Like the above `globalSessionTimeout`, the value is time in **milliseconds** (not seconds).

Session Listeners

Shiro supports the notion of a `SessionListener` to allow you to react to important session events as they occur. You can implement the `SessionListener` interface (or extend the convenience `SessionListenerAdapter`) and react to session operations accordingly.

As the default `SessionManager` `sessionListeners` property is a collection, you can configure the `SessionManager` with one or more of your listener implementations like any other collection in `shiro.ini`:

SessionListener Configuration in shiro.ini

```
[main]
...
aSessionListener = com.foo.my.SessionListener
anotherSessionListener = com.foo.my.OtherSessionListener

securityManager.sessionManager.sessionListeners = $aSessionListener, $anotherSessionListener, etc.
```

All Session Events
`SessionListeners` are notified when an event occurs for *any* session - not just for a particular session.

Session Storage

Whenever a session is created or updated, its data needs to persisted to a storage location so it is accessible by the application at a later time. Similarly, when a session is invalid and longer being used, it needs to be deleted from storage so the session data store space is not exhausted. The `SessionManager` implementations delegate these Create/Read/Update/Delete (CRUD) operations to an internal component, the `SessionDAO`, which reflects the Data Access Object (DAO) design pattern.

The power of the SessionDAO is that you can implement this interface to communicate with *any* data store you wish. This means your session data can reside in memory, on the file system, in a relational database or NoSQL data store, or any other location you need. You have control over persistence behavior.

You can configure any `SessionDAO` implementation as a property on the default `SessionManager` instance. For example, in shiro.ini:

Configuring a SessionDAO in shiro.ini

```
[main]
...
sessionDAO = com.foo.my.SessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO
```

However, as you might expect, Shiro already has some good `SessionDAO` implementations that you can use out of the box or subclass for your own needs.

Web Applications
The above `securityManager.sessionManager.sessionDAO = $sessionDAO` assignment only works when using a Shiro native session manager. Web applications by default do not use a native session manager and instead retain the Servlet Container's default session manager which does not support a SessionDAO. If you would like to enable a SessionDAO in a web-based application for custom session storage or session clustering, you will have to first configure a native web session manager. For example:

```
[main]
...
sessionManager = org.apache.shiro.web.session.mgt.DefaultWebSessionManager
securityManager.sessionManager = $sessionManager

# Configure a SessionDAO and then set it:
securityManager.sessionManager.sessionDAO = $sessionDAO
```

Configure a SessionDAO!
Shiro's default configuration native SessionManagers use ***in-memory-only*** Session storage. This is unsuitable for most production applications. Most production applications will want to either configure the provided EHCache support (see below) or provide their own `SessionDAO` implementation.

Note that web applications use a servlet-container-based SessionManager by default and do not have this issue. This is only an issue when using a Shiro native SessionManager.

EHCache SessionDAO

EHCache is not enabled by default, but if you do not plan on implementing your own `SessionDAO`, it is **highly** recommended that you enable the EHCache support for Shiro's SessionManagement. The EHCache SessionDAO will store sessions in memory and support

overflow to disk if memory becomes constrained. This is highly desirable for production applications to ensure that you don't randomly 'lose' sessions at runtime.

Use EHCache as your default
If you're not writing a custom `SessionDAO`, definitely enable EHCache in your Shiro configuration. EHCache can also be beneficial beyond Sessions, caching authentication and authorization data as well. See the Caching documentation for more information.

Container-Independent Session Clustering
EHCache is also a nice choice if you quickly need container-independent session clustering. You can transparently plug in TerraCotta behind EHCache and have a container-independent clustered session cache. No more worrying about Tomcat, JBoss, Jetty, WebSphere or WebLogic specific session clustering ever again!

Enabling EHCache for sessions is very easy. First, ensure that you have the `shiro-ehcache-<version>.jar` file in your classpath (see the Download page or use Maven or Ant+Ivy).

Once in the classpath, this first `shiro.ini` example shows you how to use EHCache for all of Shiro's caching needs (not just Session support):

Configuring EHCache for all of Shiro's caching needs in shiro.ini

```
[main]

sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO

cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
securityManager.cacheManager = $cacheManager
```

The final line, `securityManager.cacheManager = $cacheManager`, configures a `CacheManager` for all of Shiro's needs. This `CacheManager` instance will propagate down to the `SessionDAO` automatically (by nature of `EnterpriseCacheSessionDAO` implementing the `CacheManagerAware` interface).

Then, when the `SessionManager` asks the `EnterpriseCacheSessionDAO` to persist a `Session`, it will use an EHCache-backed `Cache` implementation to store the Session data.

Web Applications
Don't forget that assigning a `SessionDAO` is a feature when using Shiro native SessionManager implementations. Web applications by default use a Servlet container-based SessionManager which does not support a `SessionDAO`. Configure a native web SessionManager as explained above if you want to use Ehcache-based session storage in a web application.

EHCache Session Cache Configuration

By default, the `EhCacheManager` uses a Shiro-specific `ehcache.xml` file that sets up the Session cache region and the necessary settings to ensure Sessions are stored and retrieved properly.

However, if you wish to change the cache settings, or configure your own `ehcache.xml` or EHCache `net.sf.ehcache.CacheManager` instance, you will need to configure the cache region to ensure that Sessions are handled correctly.

If you look at the default `ehcache.xml` file, you will see the following `shiro-activeSessionCache` cache configuration:

```
<cache name="shiro-activeSessionCache"
       maxElementsInMemory="10000"
       overflowToDisk="true"
       eternal="true"
       timeToLiveSeconds="0"
       timeToIdleSeconds="0"
       diskPersistent="true"
       diskExpiryThreadIntervalSeconds="600"/>
```

If you wish to use your own `ehcache.xml` file, ensure that you have defined a similar cache entry for Shiro's needs. Most likely you might change the `maxElementsInMemory` attribute value to meet your needs. However, it is very important that at least the following two attributes exist (and are not changed) in your own configuration:

- `overflowToDisk="true"` - this ensures that if you run out of process memory, sessions won't be lost and can serialized to disk
- `eternal="true"` - ensures that cache entries (Session instances) are never expired or expunged automatically by the cache. This is necessary because Shiro does its own validation based on a scheduled process (see "Session Validation & Scheduling" below). If we turned this off, the cache would likely evict Sessions without Shiro knowing about it, which could cause problems.

EHCache Session Cache Name

By default, the `EnterpriseCacheSessionDAO` asks the `CacheManager` for a `Cache` named "`shiro-activeSessionCache`". This cache name/region is expected to be configured in `ehcache.xml`, as mentioned above.

If you want to use a different name instead of this default, you can configure that name on the `EnterpriseCacheSessionDAO`, for example:

Configuring the cache name for Shiro's active session cache in shiro.ini

```
[main]
...
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionDAO.activeSessionsCacheName = myname
```

```
...
```

Just ensure that a corresponding entry in `ehcache.xml` matches that name and you've configured `overflowToDisk="true"` and `eternal="true"` as mentioned above.

Custom Session IDs

Shiro's `SessionDAO` implementations use an internal `SessionIdGenerator` component to generate a new Session ID every time a new session is created. The ID is generated, assigned to the newly created `Session` instance, and then the `Session` is saved via the `SessionDAO`.

The default `SessionIdGenerator` is a `JavaUuidSessionIdGenerator`, which generates `String` IDs based on Java `UUIDs`. This implementation is suitable for all production environments.

If this does not meet your needs, you can implement the `SessionIdGenerator` interface and configure the implementation on Shiro's `SessionDAO` instance. For example, in `shiro.ini`:

Configuring a SessionIdGenerator in shiro.ini

```
[main]
...
sessionIdGenerator = com.my.session.SessionIdGenerator
securityManager.sessionManager.sessionDAO.sessionIdGenerator = $sessionIdGenerator
```

Session Validation & Scheduling

Sessions must be validated so any invalid (expired or stopped) sessions can be deleted from the session data store. This ensures that the data store does not fill up over time with sessions that will never be used again.

For performance reasons, `Sessions` are only validated to see if they have been stopped or expired at the time they are accessed (i.e. `subject.getSession()`). This means that without additional regular periodic validation, `Session` orphans would begin to fill up the session data store.

A common example illustrating orphans is the web browser scenario: Let's say a user logs in to a web application and a session is created to retain data (authentication state, shopping cart, etc). If the user does not log out and closes their browser without the application knowing about it, their session is essentially just 'lying around' (orphaned) in the session data store. The `SessionManager` has no way of detecting that the user was no longer using their browser, and the session is never accessed again (it is orphaned).

Session orphans, if they are not regularly purged, will fill up the session data store (which would be bad). So, to prevent orphans from piling up, the `SessionManager` implementations support the notion of a `SessionValidationScheduler`. A `SessionValidationScheduler` is responsible for validating sessions at a periodic rate to ensure they are cleaned up as necessary.

Default SessionValidationScheduler

The default `SessionValidationScheduler` usable in all environments is the `ExecutorServiceSessionValidationScheduler` which uses a JDK `ScheduledExecutorService` to control how often the validation should occur.

By default, this implementation will perform validation once per hour. You can change the rate at which validation occurs by specifying a **new** instance of `ExecutorServiceSessionValidationScheduler` and specifying a different interval (in milliseconds):

ExecutorServiceSessionValidationScheduler interval in shiro.ini

```
[main]
...
sessionValidationScheduler = org.apache.shiro.session.mgt.ExecutorServiceSessionValidationScheduler
# Default is 3,600,000 millis = 1 hour:
sessionValidationScheduler.interval = 3600000

securityManager.sessionManager.sessionValidationScheduler = $sessionValidationScheduler
```

Custom SessionValidationScheduler

If you wish to provide a custom `SessionValidationScheduler` implementation, you can specify it as a property of the default `SessionManager` instance. For example, in `shiro.ini`:

Configuring a custom SessionValidationScheduler in shiro.ini

```
[main]
...
sessionValidationScheduler = com.foo.my.SessionValidationScheduler
securityManager.sessionManager.sessionValidationScheduler = $sessionValidationScheduler
```

Disabling Session Validation

In some cases, you might wish to disable session validation entirely because you have set up a process outside of Shiro's control to perform the validation for you. For example, maybe you are using an enterprise Cache and rely on the cache's Time To Live setting to automatically expunge old sessions. Or maybe you've set up a cron job to auto-purge a custom data store. In these cases you can turn off session validation scheduling:

Disabling Session Validation Scheduling in shiro.ini

```
[main]
...
securityManager.sessionManager.sessionValidationSchedulerEnabled = false
```

Sessions will still be validated when they are retrieved from the session data store, but this will disable Shiro's periodic validation.

Enable Session Validation *somewhere*
If you turn off Shiro's session validation scheduler, you *MUST* perform periodic session validation via some other mechanism (cron job, etc.). This is the only way to guarantee Session orphans do not fill up the data store.

Invalid Session Deletion

As we've stated above, the purpose of periodic session validation is mainly to delete any invalid (expired or stopped) sessions to ensure they do not fill up the session data store.

By default, whenever Shiro detects an invalid session, it attempts to delete it from the underlying session data store via the `SessionDAO.delete(session)` method. This is good practice for most applications to ensure the session data storage space is not exhausted.

However, some applications may not wish for Shiro to automatically delete sessions. For example, if an application has provided a `SessionDAO` that backs a queryable data store, perhaps the application team wishes old or invalid sessions to be available for a certain period of time. This would allow the team to run queries against the data store to see, for example, how many sessions a user has created over the last week, or the average duration of a user's sessions, or similar reporting-type queries.

In these scenarios, you can turn off invalid session deletion entirely. For example, in `shiro.ini`:

Disabling Invalid Session Deletion in shiro.ini

```
[main]
...
securityManager.sessionManager.deleteInvalidSessions = false
```

But be careful! If you turn this off, you are responsible for ensuring that your session data store doesn't exhaust its space. You must delete invalid sessions from you data store yourself!

Note also that even if you prevent Shiro from deleting invalid sessions, you still should enable session validation somehow - either via Shiro's existing validation mechanisms or via a custom mechanism you provide yourself (see the above "Disabling Session Validation" section above for more). The validation mechanism will update your session records to reflect the invalid state (e.g. when it was invalidated, when it was last accessed, etc), even if you will delete them manually yourself at some other time.

If you configure Shiro so it does not delete invalid sessions, you are responsible for ensuring that your session data store doesn't exhaust its space. You must delete invalid sessions from you data store yourself!

Also note that disabling session deletion is **not** the same as disabling session validation scheduling. You should almost always use a session validation scheduling mechanism - either one supported by Shiro directly or your own.

Session Clustering

One of the very exciting things about Apache Shiro's session capabilities is that you can cluster Subject sessions natively and never need to worry again about how to cluster sessions based on your container environment. That is, if you use Shiro's native sessions and configure a session cluster, you can, say, deploy to Jetty or Tomcat in development, JBoss or Geronimo in production, or any other environment - all the while never worrying about container/environment-specific clustering setup or configuration. Configure session clustering once in Shiro and it works no matter your deployment environment.

So how does it work?

Because of Shiro's POJO-based N-tiered architecture, enabling Session clustering is as simple as enabling a clustering mechanism at the Session persistence level. That is, if you configure a cluster-capable `SessionDAO`, the DAO can interact with a clustering mechanism and Shiro's `SessionManager` never needs to know about clustering concerns.

Distributed Caches

Distributed Caches such as Ehcache+TerraCotta, GigaSpaces Oracle Coherence, and Memcached (and many others) already solve the distributed-data-at-the-persistence-level problem. Therefore enabling Session clustering in Shiro is as simple as configuring Shiro to use a distributed cache.

This gives you the flexibility of choosing the exact clustering mechanism that is suitable for *your* environment.

Cache Memory
Note that when enabling a distributed/enterprise cache to be your session clustering data store, one of the following two cases must be true:

- The distributed cache has enough cluster-wide memory to retain *all* active/current sessions
- If the distributed cache does not have enough cluster-wide memory to retain all active sessions, it must support disk overflow so sessions are not lost.
  Failure for the cache to support either of the two cases will result in sessions being randomly lost, which would likely be frustrating to end-users.

`EnterpriseCacheSessionDAO`

As you might expect, Shiro already provides a `SessionDAO` implementation that will persist data to an enterprise/distributed Cache. The EnterpriseCacheSessionDAO expects a Shiro `Cache` or `CacheManager` to be configured on it so it can leverage the caching mechanism.

For example, in `shiro.ini`:

```
#This implementation would use your preferred distributed caching product's APIs:
activeSessionsCache = my.org.apache.shiro.cache.CacheImplementation

sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionDAO.activeSessionsCache = $activeSessionsCache

securityManager.sessionManager.sessionDAO = $sessionDAO
```

Although you could inject a `Cache` instance directly to the `SessionDAO` as shown above, it is usually far more common to configure a general `CacheManager` to use for all of Shiro's caching needs (sessions as well as authentication and authorization data). In this case, instead of configuring a `Cache` instance directly, you would tell the `EnterpriseCacheSessionDAO` the name of the cache in the `CacheManager` that should be used for storing active sessions.

For example:

```
# This implementation would use your caching product's APIs:
cacheManager = my.org.apache.shiro.cache.CacheManagerImplementation

# Now configure the EnterpriseCacheSessionDAO and tell it what
# cache in the CacheManager should be used to store active sessions:
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
# This is the default value.  Change it if your CacheManager configured a different name:
sessionDAO.activeSessionsCacheName = shiro-activeSessionsCache
# Now have the native SessionManager use that DAO:
securityManager.sessionManager.sessionDAO = $sessionDAO

# Configure the above CacheManager on Shiro's SecurityManager
# to use it for all of Shiro's caching needs:
securityManager.cacheManager = $cacheManager
```

But there's something a bit strange about the above configuration. Did you notice it?

The interesting thing about this config is that nowhere in the config did we actually tell the `sessionDAO` instance to use a `Cache` or `CacheManager`! So how does the `sessionDAO` use the distributed cache?

When Shiro initializes the `SecurityManager`, it will check to see if the `SessionDAO` implements the `CacheManagerAware` interface. If it does, it will automatically be supplied with any available globally configured `CacheManager`.

So when Shiro evaluates the `securityManager.cacheManager = $cacheManager` line, it will discover that the `EnterpriseCacheSessionDAO` implements the `CacheManagerAware` interface and call the `setCacheManager` method with your configured `CacheManager` as the method argument.

Then at runtime, when the `EnterpriseCacheSessionDAO` needs the `activeSessionsCache` it will ask the `CacheManager` instance to return it it, using the `activeSessionsCacheName` as the lookup key to get a `Cache` instance. That `Cache` instance (backed by your distributed/enterprise caching product's API) will be used to store and retrieve sessions for all of the `SessionDAO` CRUD operations.

Ehcache + Terracotta

One such distributed caching solution that people have had success with while using Shiro is the Ehcache + Terracotta pairing. See the Ehcache-hosted Distributed Caching With Terracotta documentation for full details of how to enable distributed caching with Ehcache.

Once you've got Terracotta clustering working with Ehcache, the Shiro-specific parts are very simple. Read and follow the Ehcache SessionDAO documentation, but we'll need to make a few changes

The Ehcache Session Cache Configuration referenced previously will not work - a Terracotta-specific configuration is needed. Here is an example configuration that has been tested to work correctly. Save its contents in a file and save it in an `ehcache.xml` file:

TerraCotta Session Clustering

```
<ehcache>
    <terracottaConfig url="localhost:9510"/>
    <diskStore path="java.io.tmpdir/shiro-ehcache"/>
    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="false"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120">
        <terracotta/>
   </defaultCache>
   <cache name="shiro-activeSessionCache"
       maxElementsInMemory="10000"
       eternal="true"
       timeToLiveSeconds="0"
       timeToIdleSeconds="0"
```

```
      diskPersistent="false"
      overflowToDisk="false"
      diskExpiryThreadIntervalSeconds="600">
      <terracotta/>
   </cache>
   <!-- Add more cache entries as desired, for example,
        Realm authc/authz caching: -->
</ehcache>
```

Of course you will want to change your `<terracottaConfig url="localhost:9510"/>` entry to reference the appropriate host/port of your Terracotta server array. Also notice that, unlike the previous configuration, the `ehcache-activeSessionCache` element ***DOES NOT*** set `diskPersistent` or `overflowToDisk` attributes to `true`. They should both be `false` as true values are not supported in clustered configuration.

After you've saved this `ehcache.xml` file, we'll need to reference it in Shiro's configuration. Assuming you've made the terracotta-specific `ehcache.xml` file accessible at the root of the classpath, here is the final Shiro configuration that enables Terracotta+Ehcache clustering for all of Shiro's needs (including Sessions):

shiro.ini for Session Clustering with Ehcache and Terracotta

```
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
# This name matches a cache name in ehcache.xml:
sessionDAO.activeSessionsCacheName = shiro-activeSessionsCache
securityManager.sessionManager.sessionDAO = $sessionDAO

# Configure The EhCacheManager:
cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
cacheManager.cacheManagerConfigFile = classpath:ehcache.xml

# Configure the above CacheManager on Shiro's SecurityManager
# to use it for all of Shiro's caching needs:
securityManager.cacheManager = $cacheManager
```

And remember, **ORDER MATTERS**. By configuring the `cacheManager` on the `securityManager` last, we ensure that the CacheManager can be propagated to all previously-configured `CacheManagerAware` components (such as the `EnterpriseCachingSessionDAO`).

Zookeeper

Users have reported using Apache Zookeeper for managing/coordinating distributed sessions as well. If you have any documentation/comments about how this would work, please post them to the Shiro Mailing Lists

Sessions and Subject State

Stateful Applications (Sessions allowed)

By default, Shiro's SecurityManager implementations will use a Subject's Session as a strategy to store the Subject's identity (`PrincipalCollection`) and authentication state (`subject.isAuthenticated()`) for continued reference. This typically occurs after a Subject logs-in or when a Subject's identity is discovered via RememberMe services.

There are a few benefits to this default approach:

- Any applications that service requests, invocations or messages can associate the session ID with the request/invocation/message payload and that is all that is necessary for Shiro to associate a user with the inbound request. For example, if using the `Subject.Builder`, this is all that is needed to acquire the associated Subject:

  ```
  Serializable sessionId = //get from the inbound request or remote method invocation payload
  Subject requestSubject = new Subject.Builder().sessionId(sessionId).buildSubject();
  ```

  This is incredibly convenient for most web applications as well as anyone writing remoting or messaging frameworks. (This is in fact how Shiro's web support associates Subjects with ServletRequests in its own framework code).

- Any 'RememberMe' identity found on an initial request can be persisted to the session upon first access. This ensures that the Subject's remembered identity can be saved across requests without needing to deserialize and decrypt it on *every* request. For example, in a web application, there is no need to read an encrypted RememberMe cookie on every request if the identity is already known in the session. This can be a good performance enhancement.

Stateless Applications (Sessionless)

While the above default strategy is fine (and often desirable) for most applications, this would not be desirable in applications that try to be stateless whenever possible. Many stateless architectures mandate that no persistent state can exist between requests, in which case Sessions would not be allowed (a Session by its very nature represents durable state).

But this requirement comes at a convenience cost - Subject state cannot be retained across requests. This means that applications with this requirement must ensure Subject state can be represented in some other way for *every* request.

This is almost always achieved by authenticating every request/invocation/message handled by the application. For example, most stateless web applications typically support this by enforcing HTTP Basic authentication, allowing the browser to authenticate every request on behalf of an end user. Remoting or Messaging frameworks must ensure that Subject principals and credentials are attached to every Invocation or Message payload, typically performed by framework code.

Disabling Subject State Session Storage

Beginning in Shiro 1.2 and later, applications that wish to disable Shiro's internal implementation strategy of persisting Subject state to sessions may disable this entirely across *all* Subjects by doing the following:

In `shiro.ini`, configure the following property on the `securityManager`:

shiro.ini

```
[main]
...
securityManager.subjectDAO.sessionStorageEvaluator.sessionStorageEnabled = false
...
```

This will prevent Shiro from using a Subject's session to store that Subject's state across requests/invocations/messages *for all Subjects*. Just be sure that you authenticate on every request so Shiro will know who the Subject is for any given request/invocation/message.

Shiro's Needs vs. Your Needs
This will disable Shiro's own implementations from using Sessions as a storage strategy. It **DOES NOT** disable Sessions entirely. A session will still be created if any of your own code explicitly calls `subject.getSession()` or `subject.getSession(true)`.

A Hybrid Approach

The above `shiro.ini` configuration line (`securityManager.subjectDAO.sessionStorageEvaluator.sessionStorageEnabled = false`) will disable Shiro from using the Session as an implementation strategy for *all* Subjects.

But what if you wanted a hybrid approach? What if some Subjects should have sessions and others should not? This hybrid approach can be beneficial for many applications. For example:

- Maybe human Subjects (e.g. web browser users) should be able to use Sessions for the benefits provided above.
- Maybe non-human Subjects (e.g. API clients or 3rd-party applications) should *not* create sessions since their interaction with the software may be intermittent and/or erratic.
- Maybe all Subjects of a certain type or those accessing the system from a certain location should have state persisted in sessions, but all others should not.

If you need this hybrid approach, you can implement a `SessionStorageEvaluator`.

SessionStorageEvaluator

In cases where you want to control exactly which Subjects may have their state persisted in their Session or not, you can implement the `org.apache.shiro.mgt.SessionStorageEvaluator` interface and tell Shiro exactly which Subjects should support session storage.

This interface has a single method:

SessionStorageEvaluator

```
public interface SessionStorageEvaluator {

    public boolean isSessionStorageEnabled(Subject subject);

}
```

For a more detailed API explanation, please see the SessionStorageEvaluator JavaDoc.

You can implement this interface and inspect the Subject for any information that you might need to make this decision.

Subject Inspection

When implementing the `isSessionStorageEnabled(subject)` interface method, you can always look at the `Subject` and get access to whatever you need to make your decision. Of course all of the expected Subject methods are available to use (`getPrincipals()`, etc), but environment-specific `Subject` instances are valuable as well.

For example, in web applications, if that decision must be made based on data in the current `ServletRequest`, you can get the request or the response because the runtime `Subject` instance is actually a `WebSubject` instance:

```
    ...
    public boolean isSessionStorageEnabled(Subject subject) {
        boolean enabled = false;
        if (WebUtils.isWeb(Subject)) {
            HttpServletRequest request = WebUtils.getHttpRequest(subject);
            //set 'enabled' based on the current request.
        } else {
            //not a web request - maybe a RMI or daemon invocation?
            //set 'enabled' another way...
        }

        return enabled;
    }
```

N.B. Framework developers should keep this type of access in mind and ensure that any request/invocation/message context objects are available via environment-specific `Subject` implementations. Contact the Shiro user mailing list if you'd like some help setting this

up for your framework/environment.

Configuration

After you've implemented the `SessionStorageEvaluator` interface, you can configure it in `shiro.ini`:

shiro.ini SessionStorageEvaluator configuration

```
[main]
...
sessionStorageEvaluator = com.mycompany.shiro.subject.mgt.MySessionStorageEvaluator
securityManager.subjectDAO.sessionStorageEvaluator = $sessionStorageEvaluator

...
```

Web Applications

Often web applications wish to simply enable or disable session creation on a per request basis, regardless of which Subject is executing a request. This is often used to good effect in supporting REST and Messaging/RMI architectures. For example, perhaps normal end-users (humans using a browser) are allowed to create and use sessions, but remote API clients use REST or SOAP and shouldn't have sessions at all (because they authenticate on every request, as is common in REST/SOAP architectures).

To support this hybrid/per-request capability, a `noSessionCreation` filter has been added to Shiro's 'pool' of default filters enabled for web applications. This filter will prevent new sessions from being created during a request to guarantee a stateless experience. In `shiro.ini [urls]` section, you typically define this filter in front of all others to ensure a session will never be used.

For example:

shiro.ini - Disable Session Creation per request

```
[urls]
...
/rest/** = noSessionCreation, authcBasic, ...
```

This filter allows session usage for any *existing* session, but will not allow new sessions to be created during the filtered request. That is, any of the four following method calls on a request or subject *that do not already have an existing session* will automatically trigger a `DisabledSessionException`:

- `httpServletRequest.getSession()`
- `httpServletRequest.getSession(true)`
- `subject.getSession()`
- `subject.getSession(true)`

If a `Subject` already has a session prior to visiting the noSessionCreation-protected-URL, the above 4 calls will still work as expected.

Finally, the following calls will always be allowed in all cases:

- `httpServletRequest.getSession(false)`
- `subject.getSession(false)`