Apache Shiro Architecture

Apache Shiro's design goals are to simplify application security by being intuitive and easy to use. Shiro's core design models how most people think about application security - in the context of someone (or something) interacting with an application.

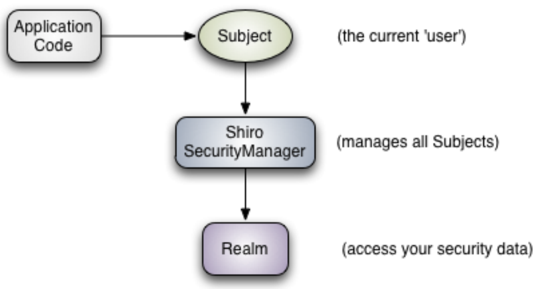
Software applications are usually designed based on user stories. That is, you'll often design user interfaces or service APIs based on how a user would (or should) interact with the software. For example, you might say, "If the user interacting with my application is logged in, I will show them a button they can click to view their account information. If they are not logged in, I will show a sign-up button."

This example statement indicates that applications are largely written to satisfy user requirements and needs. Even if the 'user' is another software system and not a human being, you still write code to reflect behavior based on who (or what) is currently interacting with your software.

Shiro reflects these concepts in its own design. By matching what is already intuitive for software developers, Apache Shiro remains intuitive and easy to use in practically any application.

High-Level Overview

At the highest conceptual level, Shiro's architecture has 3 primary concepts: the `Subject`, `SecurityManager` and `Realms`. The following diagram is a high-level overview of how these components interact, and we'll cover each concept below:



- **Subject**: As we've mentioned in our Tutorial, the `Subject` is essentially a security specific 'view' of the the currently executing user. Whereas the word 'User' often implies a human being, a `Subject` can be a person, but it could also represent a 3rd-party service, daemon account, cron job, or anything similar - basically anything that is currently interacting with the software.

  `Subject` instances are all bound to (and require) a `SecurityManager`. When you interact with a `Subject`, those interactions translate to subject-specific interactions with the `SecurityManager`.

- **SecurityManager**: The `SecurityManager` is the heart of Shiro's architecture and acts as a sort of 'umbrella' object that coordinates its internal security components that together form an object graph. However, once the SecurityManager and its internal object graph is configured for an application, it is usually left alone and application developers spend almost all of their time with the `Subject` API.

  We will talk about the `SecurityManager` in detail later on, but it is important to realize that when you interact with a `Subject`, it is really the `SecurityManager` behind the scenes that does all the heavy lifting for any `Subject` security operation. This is reflected in the basic flow diagram above.

- **Realms**: Realms act as the 'bridge' or 'connector' between Shiro and your application's security data. 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.

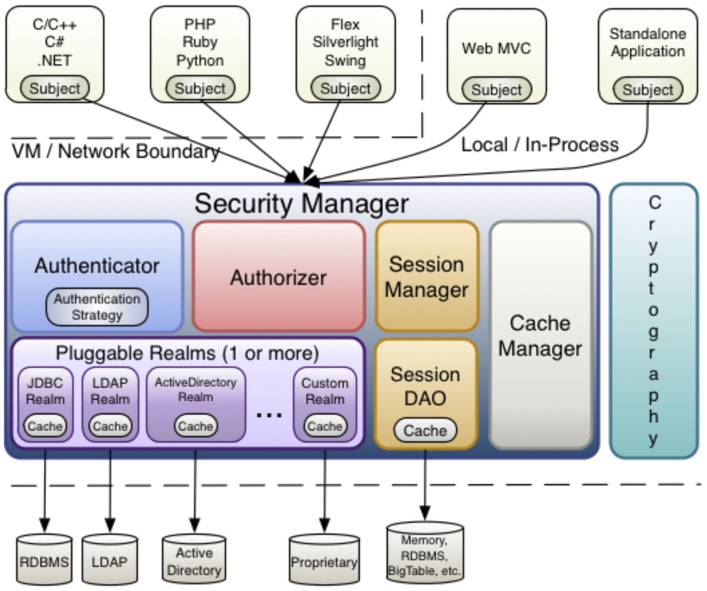
  In this sense a Realm is essentially a security-specific DAO: it encapsulates connection details for data sources and makes the associated data available to Shiro as needed. When configuring Shiro, you must specify at least one Realm to use for authentication and/or authorization. The `SecurityManager` may be configured with multiple Realms, 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.

  Like other internal components, the Shiro `SecurityManager` manages how Realms are used to acquire security and identity data to be represented as `Subject` instances.

Detailed Architecture

The following diagram shows Shiro's core architectural concepts followed by short summaries of each:



- **Subject** (`org.apache.shiro.subject.Subject`)
  A security-specific 'view' of the entity (user, 3rd-party service, cron job, etc) currently interacting with the software.

- **SecurityManager** (`org.apache.shiro.mgt.SecurityManager`)
  As mentioned above, the `SecurityManager` is the heart of Shiro's architecture. It is mostly an 'umbrella' object that coordinates its managed components to ensure they work smoothly together. It also manages Shiro's view of every application user, so it knows how to perform security operations per user.

- **Authenticator** (`org.apache.shiro.authc.Authenticator`)
  The `Authenticator` is the component that is responsible for executing and reacting to authentication (log-in) attempts by users. When a user tries to log-in, that logic is executed by the `Authenticator`. The `Authenticator` knows how to coordinate with one or more `Realms` that store relevant user/account information. The data obtained from these `Realms` is used to verify the user's identity to guarantee the user really is who they say they are.

  - **Authentication Strategy** (`org.apache.shiro.authc.pam.AuthenticationStrategy`)
    If more than one `Realm` is configured, the `AuthenticationStrategy` will coordinate the Realms to determine the conditions under which an authentication attempt succeeds or fails (for example, if one realm succeeds but others fail, is the attempt successful? Must all realms succeed? Only the first?).

- **Authorizer** (`org.apache.shiro.authz.Authorizer`)
  The `Authorizer` is the component responsible determining users' access control in the application. It is the mechanism that ultimately says if a user is allowed to do something or not. Like the `Authenticator`, the `Authorizer` also knows how to coordinate with multiple back-end data sources to access role and permission information. The `Authorizer` uses this information to determine exactly if a user is allowed to perform a given action.

- **SessionManager** (`org.apache.shiro.session.mgt.SessionManager`)
  The `SessionManager` knows how to create and manage user `Session` lifecycles to provide a robust Session experience for users in all environments. This is a unique feature in the world of security frameworks - Shiro has the ability to natively manage user Sessions in any environment, even if there is no Web/Servlet or EJB container available. By default, Shiro will use an existing session mechanism if available, (e.g. Servlet Container), but if there isn't one, such as in a standalone application or non-web environment, it will use its built-in enterprise session management to offer the same programming experience. The `SessionDAO` exists to allow any datasource to be used to persist sessions.

  - **SessionDAO** (`org.apache.shiro.session.mgt.eis.SessionDAO`)
    The `SessionDAO` performs `Session` persistence (CRUD) operations on behalf of the `SessionManager`. This allows any data store to be plugged in to the Session Management infrastructure.

- **CacheManager** (`org.apache.shiro.cache.CacheManager`)
  The `CacheManager` creates and manages `Cache` instance lifecycles used by other Shiro components. Because Shiro can access many back-end data sources for authentication, authorization and session management, caching has always been a first-class architectural feature in the framework to improve performance while using these data sources. Any of the modern open-source and/or enterprise caching products can be plugged in to Shiro to provide a fast and efficient user-experience.

- **Cryptography** (`org.apache.shiro.crypto.*`)
  Cryptography is a natural addition to an enterprise security framework. Shiro's `crypto` package contains easy-to-use and understand representations of crytographic Ciphers, Hashes (aka digests) and different codec implementations. All of the classes in this package are carefully designed to be very easy to use and easy to understand. Anyone who has used Java's native cryptography support knows it can be a challenging animal to tame. Shiro's crypto APIs simplify the complicated Java mechanisms and make cryptography easy to use for normal mortal human beings.

- **Realms** (`org.apache.shiro.realm.Realm`)
  As mentioned above, Realms act as the ‘bridge’ or ‘connector’ between Shiro and your application’s security data. 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. You can configure as many `Realms` as you need (usually one per data source) and Shiro will coordinate with them as necessary for both authentication and authorization.

The `SecurityManager`

Because Shiro's API encourages a `Subject`-centric programming approach, most application developers will rarely, if ever, interact with the `SecurityManager` directly (framework developers however might sometimes find it useful). Even so, it is still important to know how the `SecurityManager` functions, especially when configuring one for an application.

Design

As stated previously, the application's `SecurityManager` performs security operations and manages state for *all* application users. In Shiro's default `SecurityManager` implementations, this includes:

- Authentication
- Authorization
- Session Management
- Cache Management
- Realm coordination
- Event propagation
- "Remember Me" Services
- Subject creation
- Logout
  and more.

But this is a lot of functionality to try to manage in a single component. And, making these things flexible and customizable would be very difficult if everything were lumped into a single implementation class.

To simplify configuration and enable flexible configuration/pluggability, Shiro's implementations are all highly modular in design - so modular in fact, that the SecurityManager implementation (and its class-hierarchy) does not do much at all. Instead, the `SecurityManager` implementations mostly act as a lightweight 'container' component, delegating almost all behavior to nested/wrapped components. This 'wrapper' design is reflected in the detailed architecture diagram above.

While the components actually execute the logic, the `SecurityManager` implementation knows how and when to coordinate the components for the correct behavior.

The `SecurityManager` implementations and are also JavaBeans compatible, which allows you (or a configuration mechanism) to easily customize the pluggable components via standard JavaBeans accessor/mutator methods (get*/set*). This means the Shiro's architectural modularity can translate into very easy configuration for custom behavior.

Easy Configuration
Because of JavaBeans compatibility, it is very easy to configure the `SecurityManager` with custom components via any mechanism that supports JavaBeans-style configuration, such as Spring, Guice, JBoss, etc.

We will cover Configuration next.

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.