Security SPI Implementation

Table of Contents

Domain Model . . . 3
Screenshots . . . 4
- Automatically Seeds Roles . . . 4
- Add permission at different scopes . . . 6
- Permissions can ALLOW or VETO access . . . 6
- Permissions can apply to VIEWING or CHANGING the feature . . . 7
- Specify package . . . 8
- Application users . . . 8
- User Sign-up (Self-Registration) . . . 9
- Application Tenancy . . . 13

How to configure/use . . . 16
- Classpath . . . 16
- Shiro configuration (shiro.ini) . . . 16
- Bootstrapping . . . 17
- Configuration Properties . . . 18
- Overriding the schema . . . 20

API and Implementation . . . 21
- PasswordEncryptionService . . . 21
- PermissionsEvaluationService . . . 21

Default Roles, Permissions and Users . . . 22
Known issues . . . 23
Dependencies . . . 24

This module (`isis-module-security`) provides the ability to manage users, roles, and permissions. Users have roles, roles have permissions, and permissions are associated with application features. These features are derived from the Isis metamodel and can be scoped at either a *package*, *class* or individual *class member*. Permissions themselves can either *allow* or *veto* the ability to *view* or *change* any application feature.

A key design objective of this module has been to limit the amount of permissioning data required. To support this objective:

- permissions are hierarchical: a class-level permission applies to all class members, while a package-level permission applies to all classes of all subpackages
- permissions can *allow* or *veto* access; thus a role can be granted access to most features, but excluded from selective others
- permissions are scoped: a member-level permission overrides a class-level permission, a class-level permission overrides a package-level permission; the lower-level package permission overrides a higher-level one (eg `com.mycompany.invoicing` overrides `com.mycompany`).
- if there are conflicting permissions at the same scope, then the *allow* takes precedence over the **veto**.

The module also supports multi-tenancy, whereby a user can be prevented from either viewing or modifying objects to which they don't have access ("don't belong to them").

- In the original design of this module, this was based on the concept of a hierarchical tenancy, each of which is identified by an *application tenancy path* ("atPath") and the interpretation of which was hard-coded into the module. For example, users would typically have the ability to view/edit data in "their" tenancy or sub-tenancies, could view data in parent tenancies (for example, global reference data/standing data), and would have no access to any peer entities.
- As of v1.13.3, this has been generalized; the interpretation of the "atPath" is entirely application-specific (with legacy support for the original design).
- As of v1.13.6, this has been generalized further; an `ApplicationUser`'s "atPath" is no longer a reference to a particular `ApplicationTenancy` entity; it is simply a string. That string could of course still be used to represent the identifier of

a single `ApplicationTenancy`, but it need not: it should be thought of instead as encoding all the characteristics of the user or domain object which pertain to determining visibility/editability. The `ApplicationTenancy` entity has been retained only to make it easier to migrate to this more generalized model.

The module also provides an implementation of Apache Shiro's AuthorizingRealm. This allows the users/permissions to be used for Isis' authentication and/or authorization.

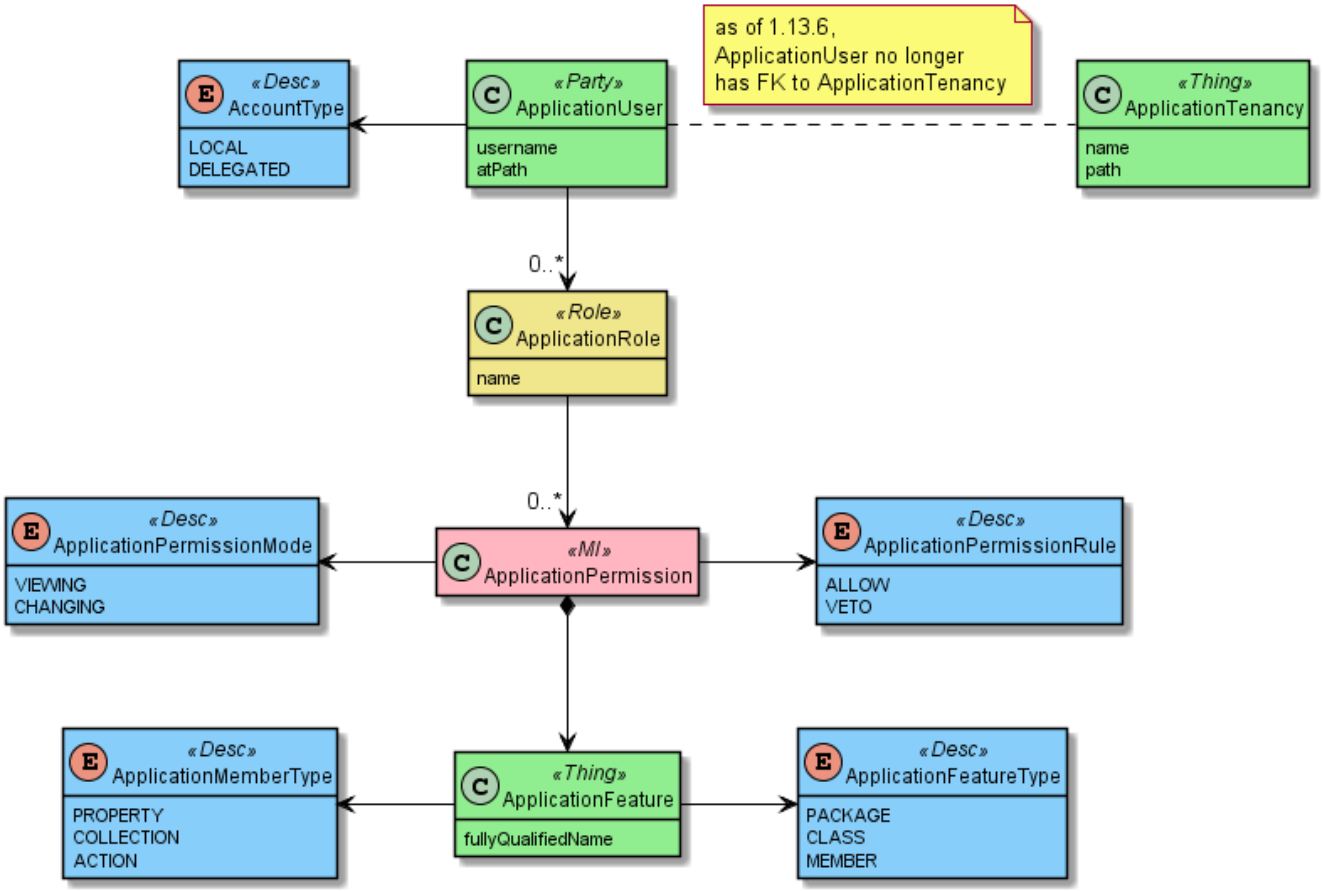
Authentication is optional; each user is either *local* or *delegated*:

- users with a *delegated* account type are authenticated through a (configured) *delegated authentication realm* (for example LDAP). Any other implementation of Shiro's `AuthenticatingRealm` can be used.
- users with a *local* account type are authenticated through a `PasswordEncryptionService`.

The module provides a default implementation based on jBCrypt, but other implementations can be plugged-in if required.

Domain Model

The following class diagram highlights the main concepts:

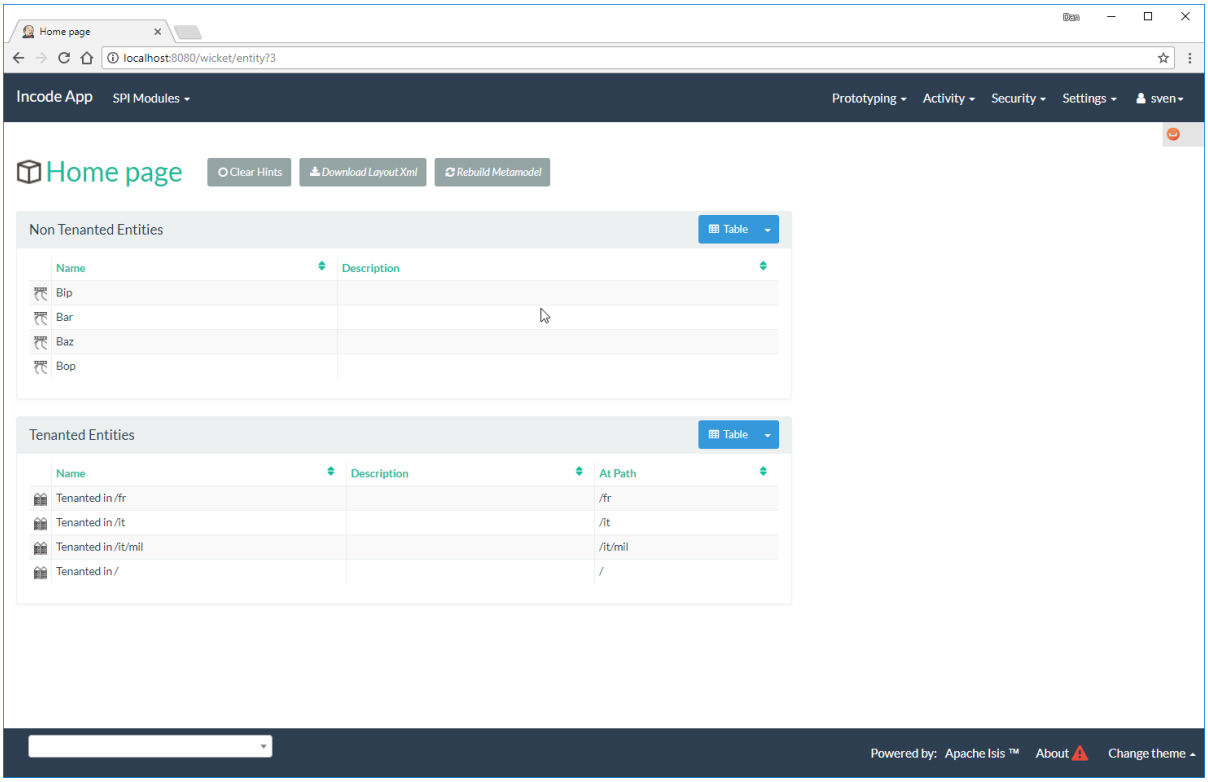


(The colours used in the diagram are - approximately - from Object Modeling in Color)

Screenshots

The module's functionality can be explored by running the quickstart with example usage using the `org.incode.domainapp.example.app.modules.ExampleDomSpiSecurityAppManifest`.

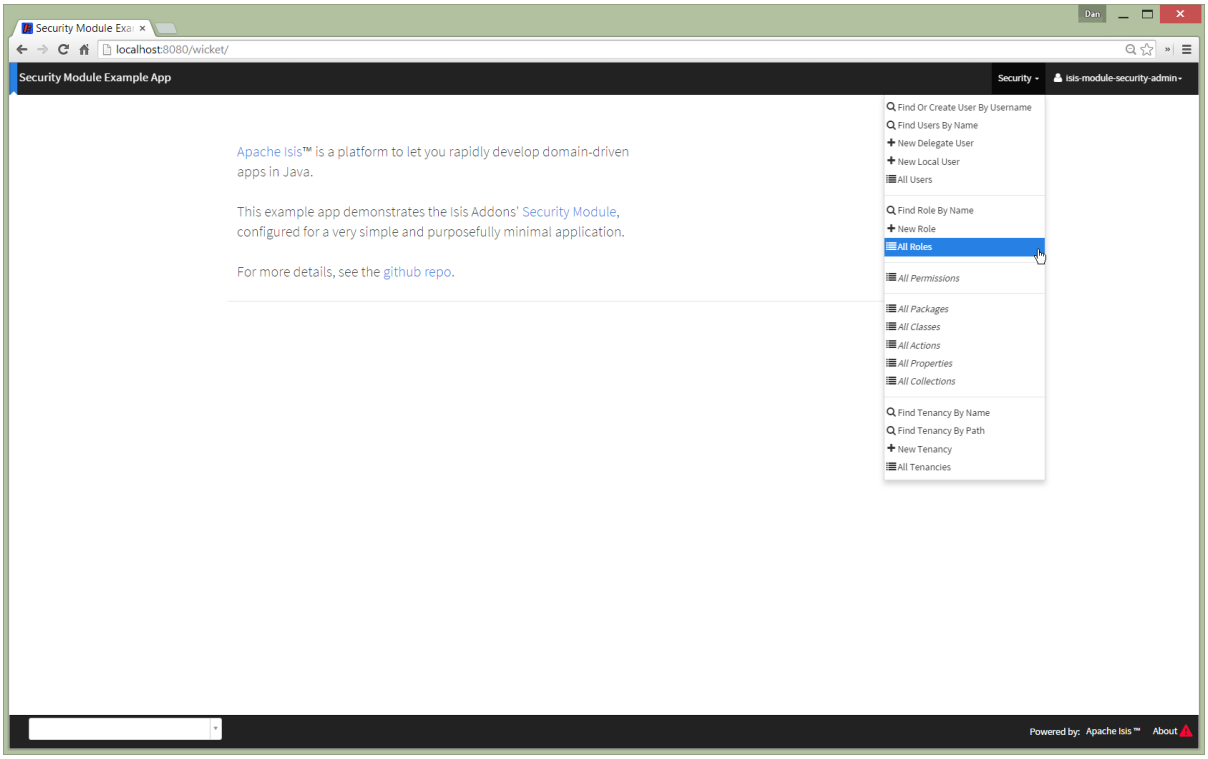
A home page is displayed when the app is run:



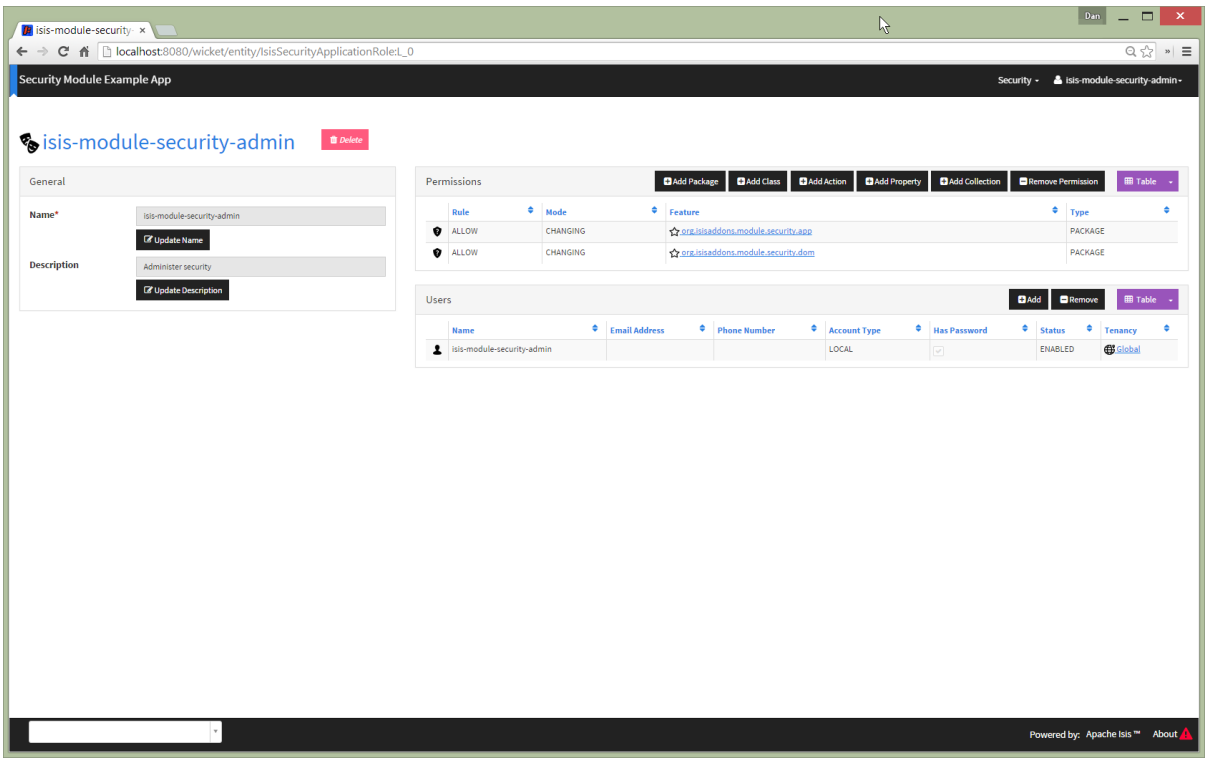
The remaining screenshots below **do** demonstrate the functionality of this module, but are out of date in that they are taken from the original isisaddons/incodehq module (prior to being amalgamated into the incode-platform).

Automatically Seeds Roles

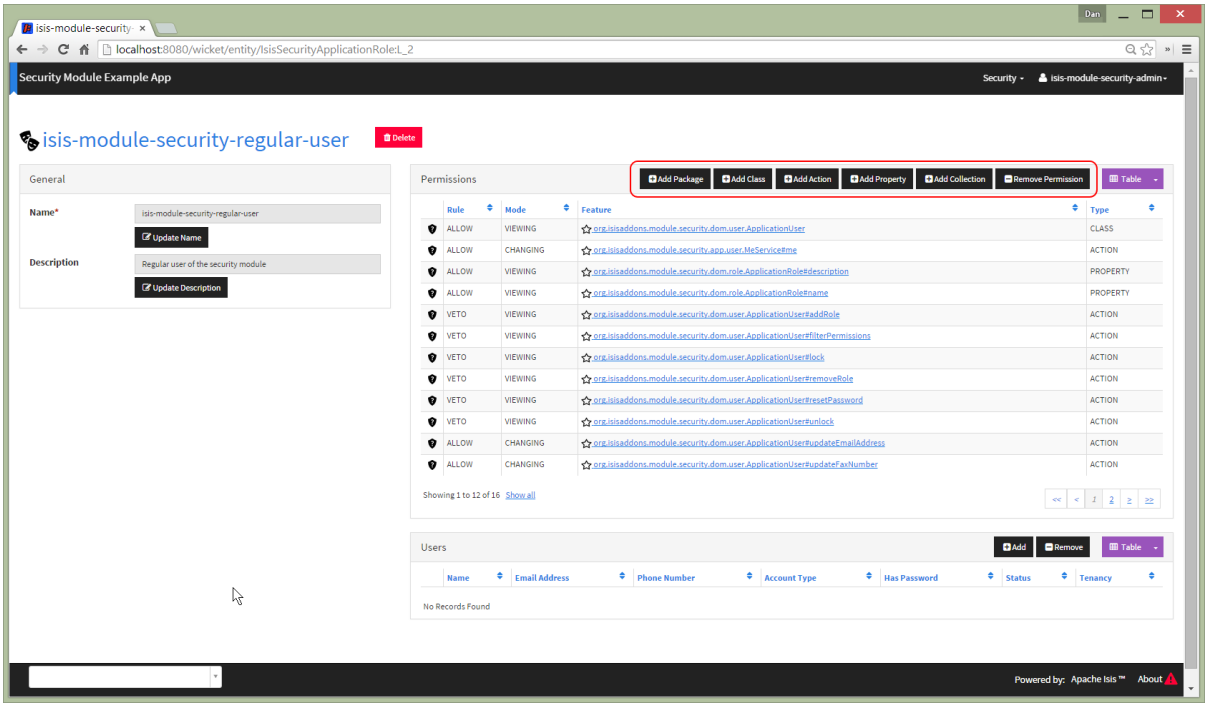
When the security module starts up, it will automatically (idempotently) seed a number of roles, corresponding permissions and a default `isis-module-security-admin` user. This user has access to the security menu:



One of the roles granted to the `isis-module-security-admin` user is the corresponding (similarly named) `isis-module-security-admin` role. It is this that grants all permissions to all classes in the security module itself:



The `isis-module-security-regular-user` role grants selected permissions to viewing/changing members of the `ApplicationUser` class (so that a user with this role can view/update their own record):



Add permission at different scopes

Permissions can be created at different scopes or levels (highlighted in the above screenshot).

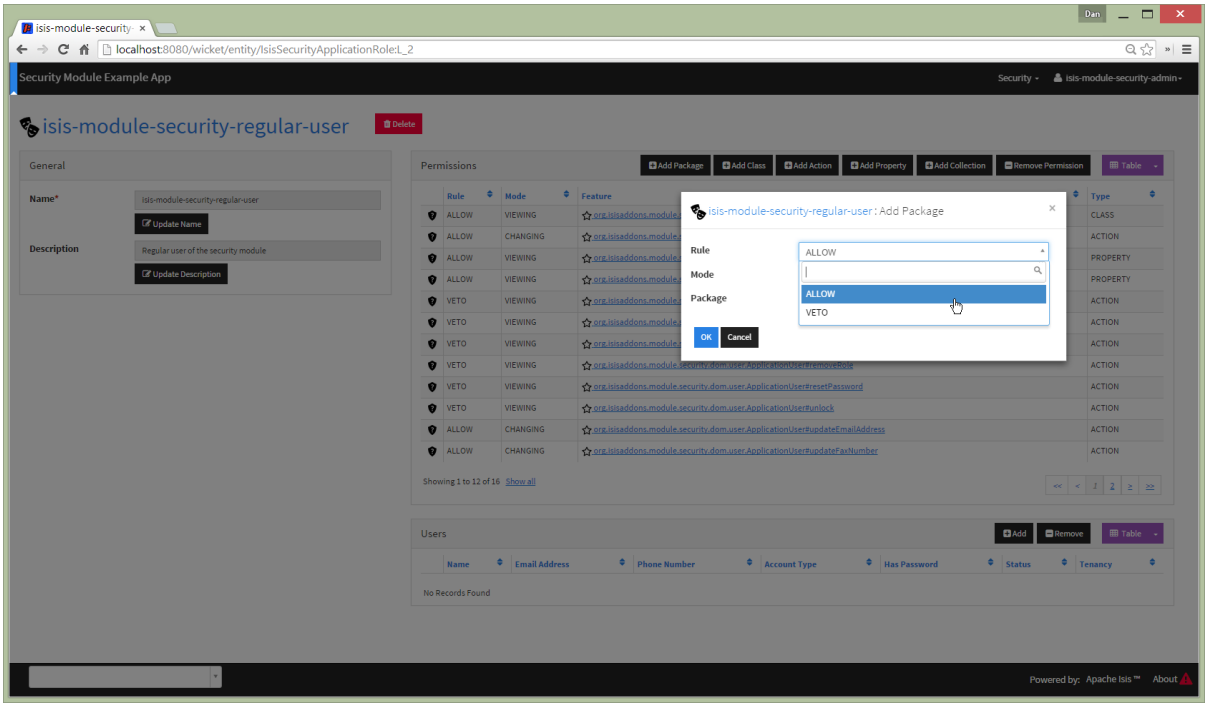
- Permissions created at the *package level* apply to all classes in all packages and subpackages (that is, recursively).
- Permissions defined at the *class level* take precedence to those defined at the package level.

For example, a user might have *allow/viewing* at a parent level, but have this escalated to *allow/changing* for a particular class. Conversely, the class-level permission might veto access.

Permissions can also be defined the *member level*: action, property or collection. These override permissions defined at either the class- or package-level.

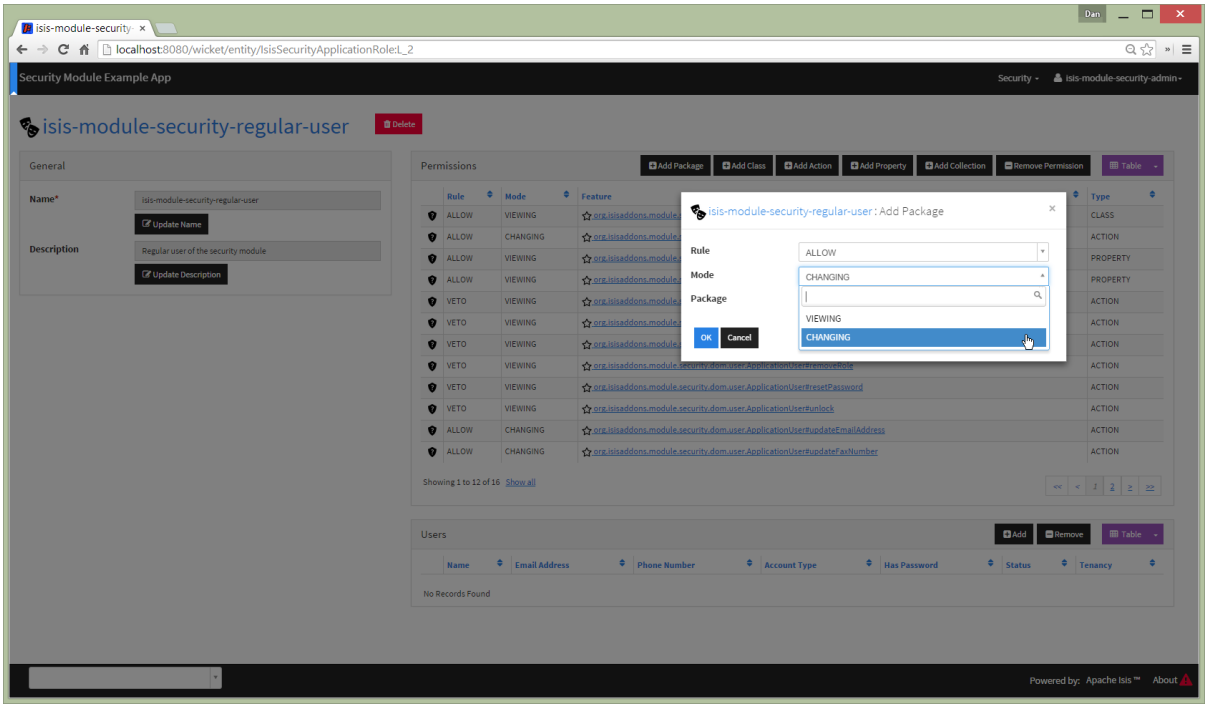
Permissions can ALLOW or VETO access

Permissions can either grant (allow) access or prevent (veto) access. If a user has permissions that contradict each other (for example, they are a member of "roleA" that allows the permission, but also of "roleB" that vetoes the permission) then by default the allow wins. However, this is strategy is pluggable, and the security module can be configured such that a veto would override an allow if required.



Permissions can apply to VIEWING or CHANGING the feature

For a property, "changing" means being able to edit it. For a collection, "changing" means being able to add or remove from it. For an action, "changing" means being able to invoke it.

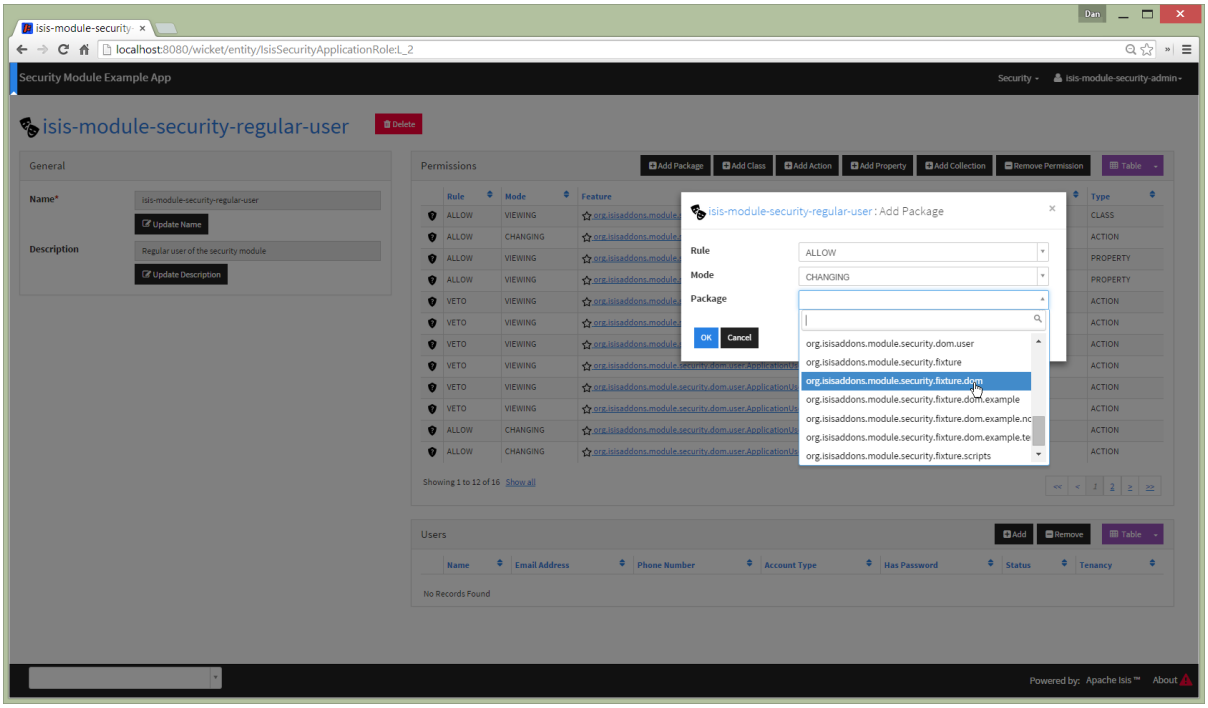


Note that Isis' Wicket viewer currently does not support the concept of "changing" collections; the work-around is instead create a pair of actions to add/remove instead. This level of control is usually needed anyway.

An *allow/changing* permission naturally enough implies *allow/viewing*, while conversely and symmetrically *veto/viewing* permission implies *veto/changing*.

Specify package

The list of packages (or classes, or class members) is derived from Isis' own metamodel.

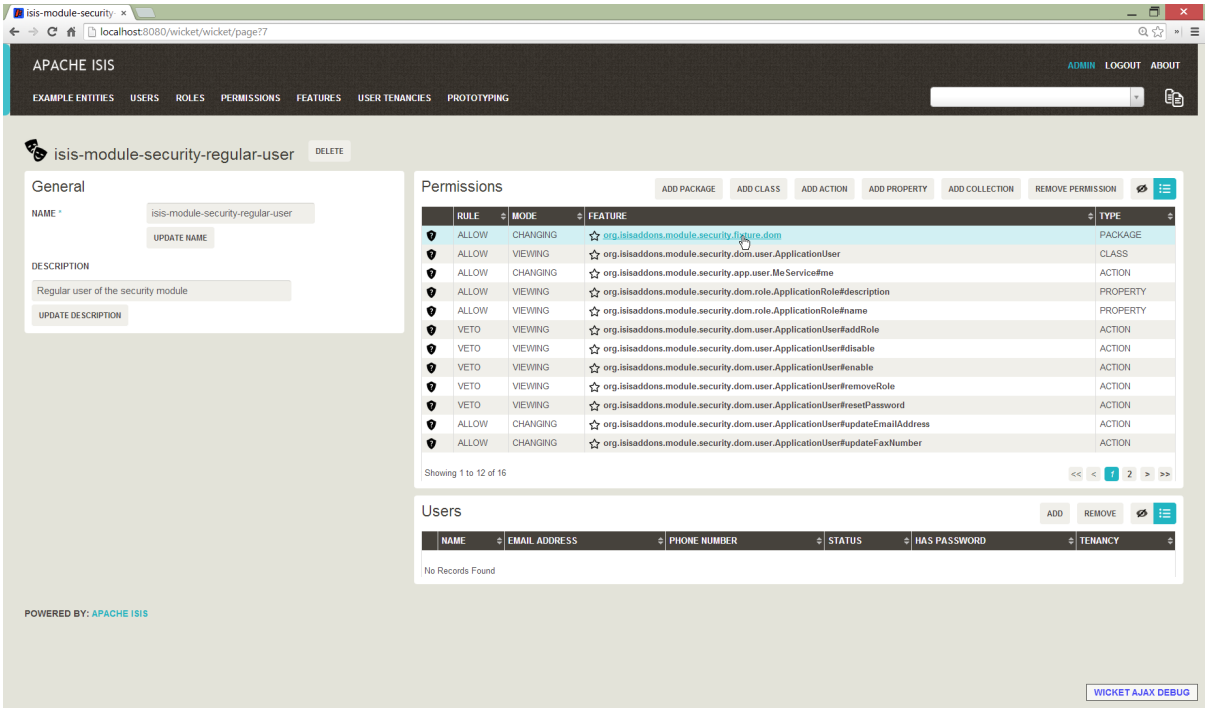


Application users

Application users can have either a *local* or a *delegated* account type.

- Local users are authenticated and authorized through the module's Shiro realm implementation. The users are created explicitly by the administrator.
- Optionally a delegate authentication realm can be configured; if so then delegated users can be created and their credentials will be authenticated by the delegate authentication realm. By default, users are created *automatically* when that user attempts to log in (though this feature can be disabled, see below). However, for safety their `ApplicationUser` accounts are created in a disabled state and with no roles, so the administrator is still required to update them.

Once the user is created, then additional information about that user can be captured, including their name and contact details. This information is not otherwise used by the security module, but may be of use to other parts of the application. The users' roles and effective permissions are also shown.



A user can maintain their own details, but may not alter other users' details. An administrator can alter all details, as well as reset a users' password.

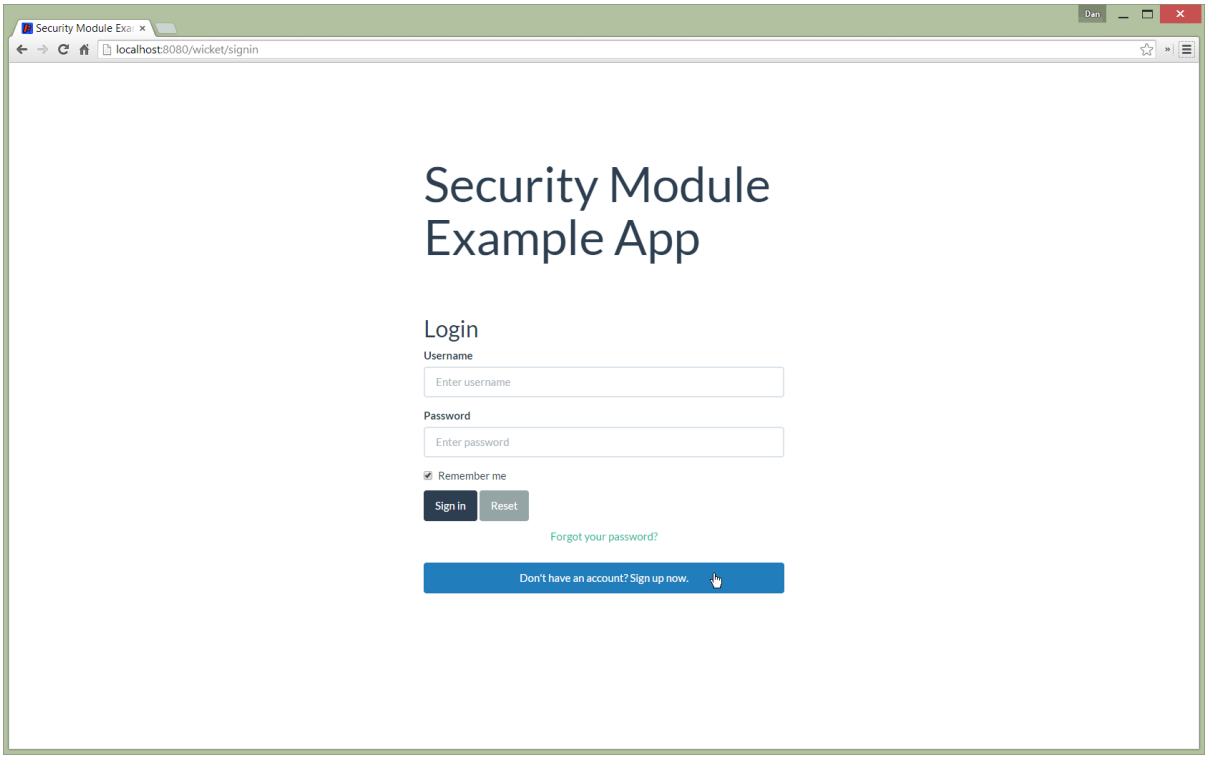
If a user is disabled, then they may not log in. This is useful for temporarily barring access to users without having to change all their roles, for example if they leave the company or go on maternity leave.

User Sign-up (Self-Registration)

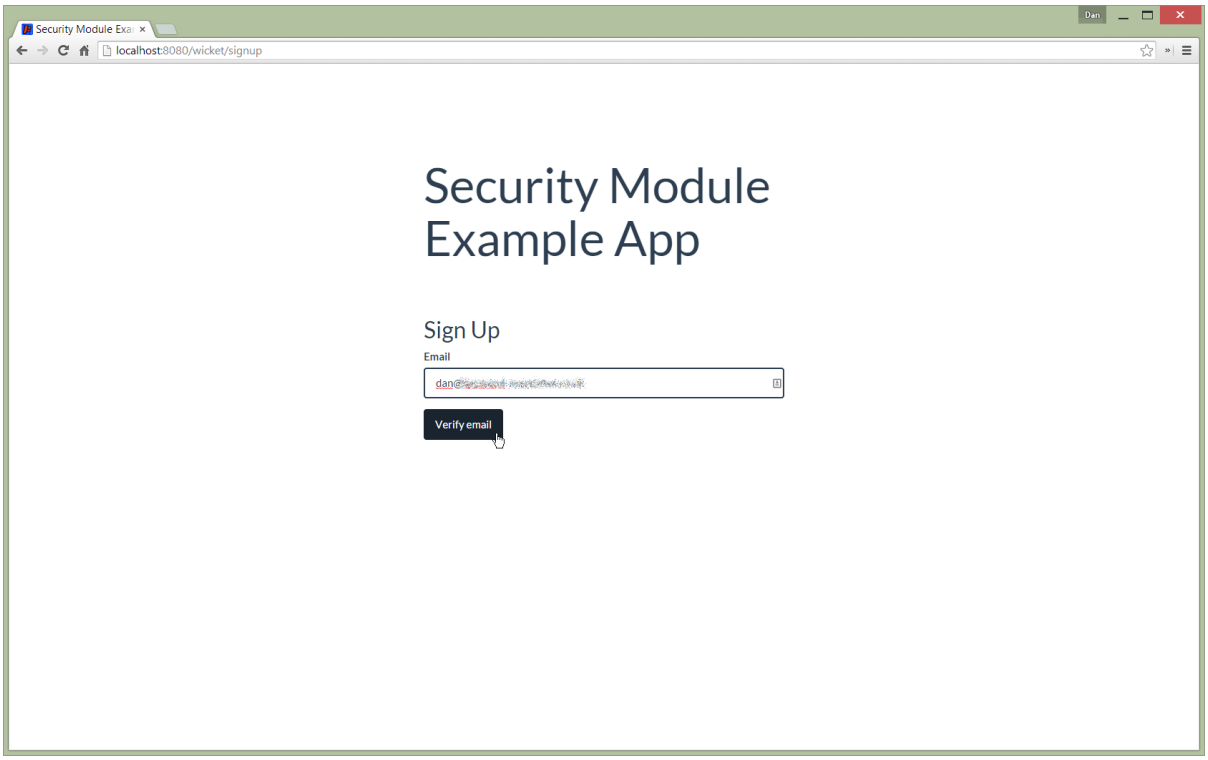
Apache Isis allows users to sign-up (self-register) with an application provided that:

- the application is correctly configured for the `EmailNotificationService`, by specifying `isis.service.email.sender.address` and `isis.service.email.sender.password` configuration properties; and
- the application provides an implementation of the `UserRegistrationService` (more on this below).

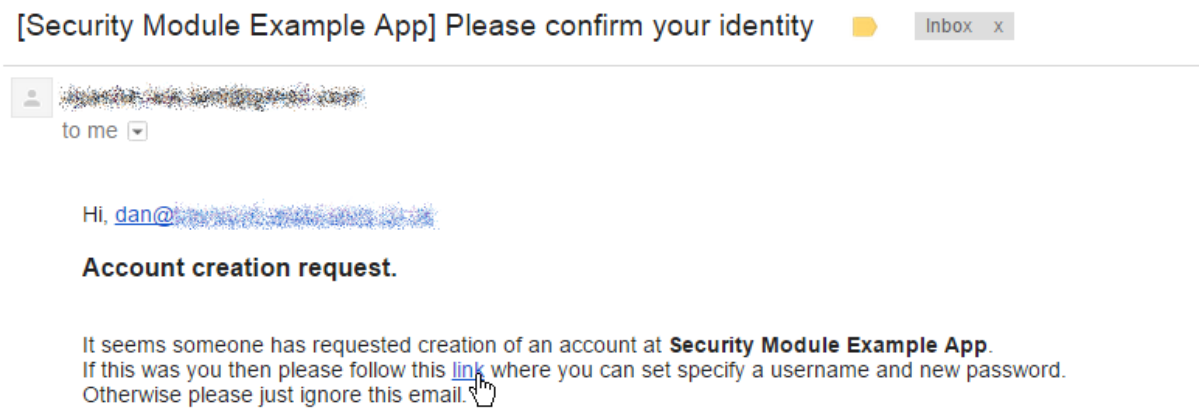
The sign-up link is shown on the initial login page:



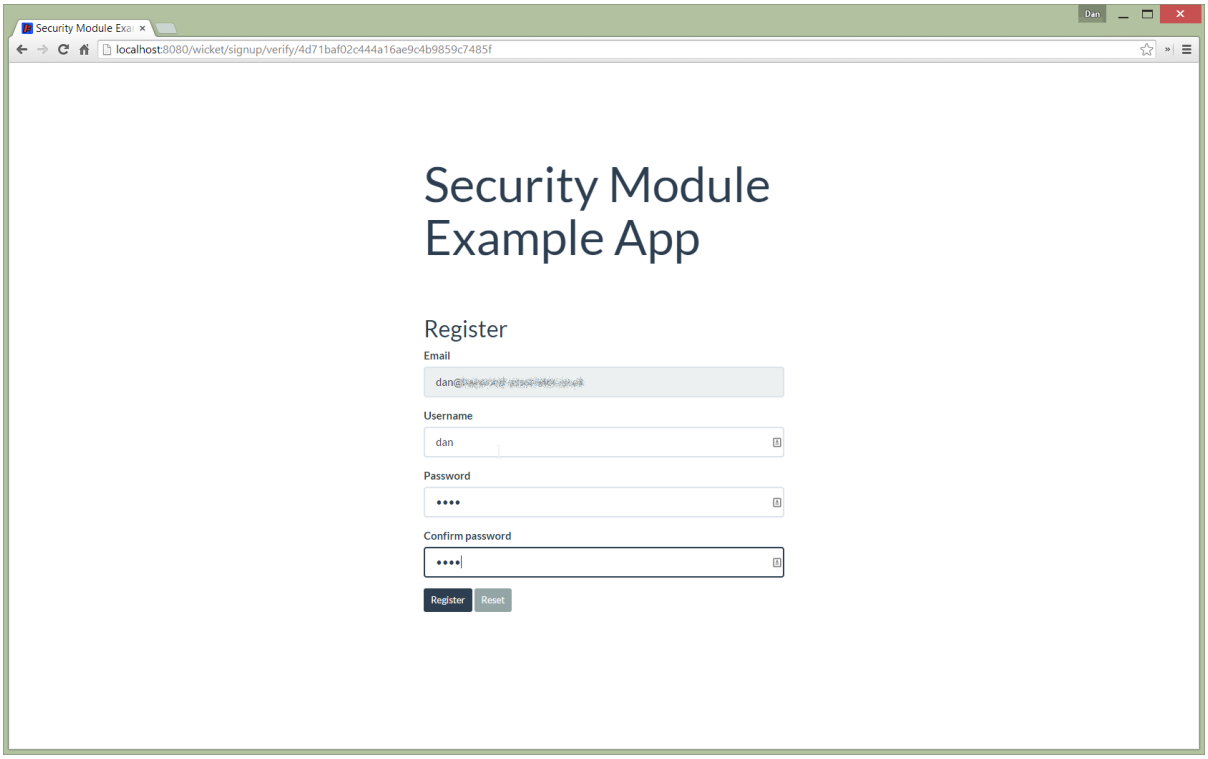
Following the link prompts for an email:



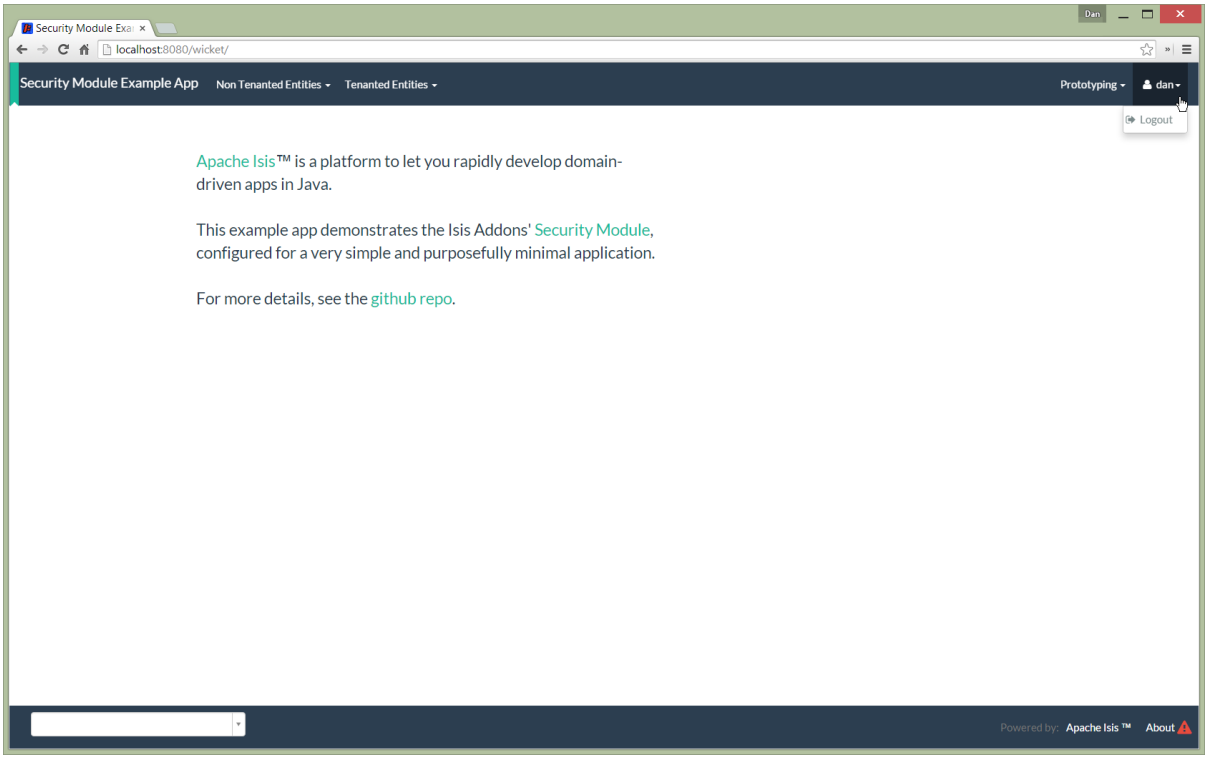
An email is sent to the specified address, with a link to complete the registration:



Completing registration consists of selecting a username and password:



The user can then login:



In the screenshot above note that the user has a default set of permissions. These are set up by the `UserRegistrationService` implementation. The security module provides `SecurityModuleAppUserRegistrationServiceAbstract` which provides most of the implementation of this service; the demo app's `AppUserRegistrationService` service completes the implementation by specifying the role(s) to assign any new users:

```
@DomainService
public class AppUserRegistrationService extends
SecurityModuleAppUserRegistrationServiceAbstract {
    protected ApplicationRole getInitialRole() {
    return findRole(ExampleFixtureScriptsRoleAndPermissions.ROLE_NAME);
    }
    protected Set<ApplicationRole> getAdditionalInitialRoles() {
        return Collections.singleton(findRole(ExampleRegularRoleAndPermissions
.ROLE_NAME));
    }
    private ApplicationRole findRole(final String roleName) {
        return applicationRoles.findRoleByName(roleName);
    }
    @Inject
    private ApplicationRoles applicationRoles;
}
```

So, for the demo app at least, any new user has access to the "example-fixture-scripts" role (= the *Prototyping* menu) and to the "example-regular-role" (= the *Tenanted Entities* and the *Non-Tenanties Entities* menus).

Speaking of which...

Application Tenancy

Application tenancy is the concept of determining which application users can see/modify domain objects. For this to work, there needs to be a application-specific mechanism for making the decision. This is defined by the (optional) `ApplicationTenancyEvaluator` SPI service:

```
public interface ApplicationTenancyEvaluator {
    boolean handles(Class<?> cls);                                          ①
    String hides(Object domainObject, ApplicationUser applicationUser);     ②
    String disables(Object domainObject, ApplicationUser applicationUser);  ③
}
```

① Whether this evaluator can determine the tenancy of the specified domain entity being interacted with (the "what").

② Whether this instance of the domain object can be viewed by the user. Any non-null string is interpreted as meaning that the object should be hidden from view

③ Whether this instance of the domain object can be modifed by the user; a non-null return value is interpreted as the reason it is read-only.

> This demo app demonstrates how this can work for an application where the name of each object is correlated to the roles of the user.

ApplicationTenancy using Paths

The security module also provides its own built-in mechanism for determining application tenancy, through the concept of the application tenancy path ("atPath").

> This is this design is designed for hierarchical tenancies, eg where a tenancy corresponds to a country or region and a user cannot reside in multiple regions concurrently. If this does not suit your requirements, then use the more general purpose `ApplicationTenancyEvaluator` SPI service described above.

Both application users and domain objects can have an "at path", this is expected to identify a single `ApplicationTenancy` entity. For application user's this is simply a property of the object, for domain object's this is performed by implementing the `HasAtPath` interface:

```
public interface HasAtPath {
    String getAtPath();
}
```

The application can then be configured so that access to domain objects can be restricted based on the respective tenancies of the user accessing the object and of the object itself. The table below summarizes the rules:

object's tenancy	user's tenancy	visible?	editable?
null	null	Y	Y
null	non-null	Y	Y
/	/	Y	Y
/	/it	Y	
/	/it/car	Y	
/	/it/igl	Y	
/	/fr	Y	
/	null		
/it	/	Y	Y
/it	/it	Y	Y
/it	/it/car	Y	
/it	/it/igl	Y	
/it	/fr		
/it	null		
/it/car	/	Y	Y
/it/car	/it	Y	Y
/it/car	/it/car	Y	Y
/it/car	/it/igl		
/it/car	/fr		
/it/car	null		

To enable this requires a single configuration property to be set, see below.

`ApplicationTenancyPathEvaluator`

You may not wish to have your domain objects implement the `WithApplicationTenancy`. As all that is required is to determine the application "path" of a domain object, an alternative is to provide an implementation of the `ApplicationTenancyPathEvaluator` SPI service.

This is defined as:

```
public interface ApplicationTenancyPathEvaluator {
    boolean handles(Class<?> cls);                                  ①
    String applicationTenancyPathFor(final Object domainObject);    ②
}
```

① indicates if the domain object's class has multi-tenancy

② the method that actually returns the path.

For example, the todoapp provides an implementation for its `ToDoItem`:

```
@DomainService(
        nature = NatureOfService.DOMAIN
)
public class ApplicationTenancyPathEvaluatorForToDoApp implements
ApplicationTenancyPathEvaluator {
    @Override
    public boolean handles(final Class<?> cls) {
        return ToDoItem.class == cls;
    }
    @Override
    public String applicationTenancyPathFor(final Object domainObject) {
        // always safe to do, per the handles(...) method earlier
        final ToDoItem toDoItem = (ToDoItem) domainObject;
        return toDoItem.getAtPath();
    }
}
```

The evaluator can also optionally handle and return a path for the security domain module's own `ApplicationUser` entity; but if it does not, then the user's own tenancy (`ApplicationUser#getTenancy()`) is used instead.

How to configure/use

Classpath

Update your classpath by adding this dependency in your dom project's `pom.xml`:

```
<dependency>
    <groupId>org.isisaddons.module.security</groupId>
    <artifactId>isis-module-security-dom</artifactId>
    <version>1.14.0</version>
</dependency>
```

If using the `PasswordEncryptionServiceUsingJBcrypt` service (discussed below), also add a dependency on the underlying `jbcrypt` library:

```
<dependency>
    <groupId>org.mindrot</groupId>
    <artifactId>jbcrypt</artifactId>
    <version>0.3m</version>
</dependency>
```

Check for later releases by searching [Maven Central Repo](http://search.maven.org/#search|ga|1|isis-module-security-dom).

For instructions on how to use the latest `-SNAPSHOT`, see the contributors guide.

Shiro configuration (shiro.ini)

The module includes `org.isisaddons.module.security.shiro.IsisModuleSecurityRealm`, an implementation of Apache Shiro's `org.apache.shiro.realm.AuthorizingRealm` class. This realm is intended to be configured as the single realm for Shiro, but it can optionally have a delegateAuthenticationRealm injected into it.

- if configured without a delegate realm then `IsisModuleSecurityRealm` deals only with *local* users and performs both authentication and authorization for them. Authentication is performed against encrypted password. Users with *delegate* account type will be unable to log in.
- if configured with a delegate realm then `IsisModuleSecurityRealm` deals with both *delegated* and *local* users. Authentication of *delegated* users is performed by the delegate authentication realm, while *local* users continue to be authenticated in the same way as before, against their encrypted password. Authorization is performed the same way for either account type, by reference to their user roles and those roles' permissions.

For both *local* and *delegated* users the realm will prevent a disabled user from logging in.

To configure, update your `WEB-INF/shiro.ini's` `[main]` section:

```
[main]

isisModuleSecurityRealm=org.isisaddons.module.security.shiro.IsisModuleSecurityRealm

authenticationStrategy=org.isisaddons.module.security.shiro.AuthenticationStrategyForI
sisModuleSecurityRealm
securityManager.authenticator.authenticationStrategy = $authenticationStrategy

securityManager.realms = $isisModuleSecurityRealm
```

If a delegate authentication realm is used, then define it and inject (again, in the `[main]` section):

```
someOtherRealm=...                                                    ①

isisModuleSecurityRealm.delegateAuthenticationRealm=$someOtherRealm
```

① the `someOtherRealm` variable defines some other realm to perform authentication.

To disable the automatic creation of delegate users, use:

```
isisModuleSecurityRealm.autoCreateUser=false
```

Bootstrapping

If implementing `AppManifest` interface directly, then update its `getModules()` method and `getAdditionalServices()` method, eg:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
            ...
            org.isisaddons.module.security.SecurityModule.class,
    );
}
@Override
public List<Class<?>> getAdditionalServices() {
    return Arrays.asList(
            org.isisaddons.module.security.dom.password
.PasswordEncryptionServiceUsingJBcrypt.class         ①
           ,org.isisaddons.module.security.dom.permission
.PermissionsEvaluationServiceAllowBeatsVeto.class  ②
    );
}
```

① is an implementation of the `PasswordEncryptionService`. This is mandatory; local users (including the default `isis-module-security-admin` administrator user) must be authenticated using the

password service. If required, any other implementation can be supplied.

② is an implementation of the `PermissionsEvaluationService` that determines how to resolve conflicting permissions at the same scope. This service is optional; if not presentthen the module will default to an allow-beats-veto strategy. An alternative implementation of `PermissionsEvaluationServiceVetoBeatsAllow` is also available for use if required; or any other implementation of this interface can be supplied.

Or, if using 1.15.x or later, you can subclass from `AppManifestAbstract` and use the `AppManifestAbstract.Builder`, by calling `withModules()` and `withAdditionalServices()`, eg:

```
public class MyAppManifest extends AppManifestAbstract {

    public static final Builder BUILDER = Builder.forModules(
            ...
            org.isisaddons.module.security.SecurityModule.class,
    )
    .withAdditionalServices(
            org.isisaddons.module.security.dom.password
.PasswordEncryptionServiceUsingJBcrypt.class
           ,org.isisaddons.module.security.dom.permission
.PermissionsEvaluationServiceAllowBeatsVeto.class
    )

    ...
}
```

There is further discussion of the `PasswordEncryptionService` and `PermissionsEvaluationService` below.

> The security module automatically seeds users and roles, using fixture scripts. As of `1.9.0` and later it is no longer necessary to register an implementation of `FixtureScripts` domain service; the core Apache Isis framework provides a default implementation.

Configuration Properties

Tenancy Checking

To enable tenancy checking (as described above, to restrict a user's access to tenanted objects), then a configuration property must be added. This can either be specified in the `AppManifest` or in `WEB-INF/isis.properties`.

If using an `AppManifest`, then update its `getConfigurationProperties()` method:

```
@Override
public Map<String, String> getConfigurationProperties() {
    return ImmutableMap.of(
        "isis.reflector.facets.include",
        "org.isisaddons.module.security.facets.TenantedAuthorizationFacetFactory");

}
```

Alternatively, if using `isis.properties`, then define:

```
isis.reflector.facets.include=org.isisaddons.module.security.facets.TenantedAuthorizat
ionFacetFactory
```

Font awesome icons

The actions for the security module do *not* include font-awesome icons by default; you will most likely want to choose your own icons.

The easiest way to do this is using the `isis.reflector.facet.cssClassFa.patterns` configuration property which uses the name of the action methods to associate an appropriate font-awesome icon.

The action names defined by the domain objects within the security module use the following naming conventions:

- `newXxx` - create a new persisted object
- `findXxx` - find an existing object
- `updateXxx` - update an existing object
- `deleteXxx` - delete an existing object
- `addXxx` - add an existing object to a collection of another
- `removeXxx` - remove an object from a collection
- `allXxx` - for prototyping actions

There are also some other miscellaneous action names, eg:

- `lock` - lock a user (to prevent that user from logging in)
- `unlock` - unlock a user so that they can login
- `resetPassword` - allow an administrator to reset the password for a user
- `me` - to lookup the `ApplicationUser` entity for the currently logged-in user

For example, define the following configuration property:

```
isis.reflector.facet.cssClassFa.patterns=\
                         new.*:fa-plus,\
                         add.*:fa-plus-square,\
                         create.*:fa-plus,\
                         update.*:fa-edit,\
                         remove.*:fa-minus-square,\
                         find.*:fa-search,\
                         all.*:fa-list
```

Overriding the schema

By default the module's entities will be installed in the `isissecurity` schema. This is hard-coded in their annotations.

This can be overridden by creating a `.orm` file. For example, to change the `ApplicationUser` table to reside in the "dbo" schema (to run on SQL Server, say), create an `ApplicationUser-sqlserver.orm` file. This should reside in the `org.isisaddons.module.security.dom.user` package:

```
<?xml version="1.0" encoding="UTF-8" ?>
<orm xmlns="http://xmlns.jcp.org/xml/ns/jdo/orm"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/jdo/orm
        http://xmlns.jcp.org/xml/ns/jdo/orm_3_0.xsd">

    <package name="org.isisaddons.module.security.dom.user">
        <class name="ApplicationUser"
               schema="dbo">
        </class>
    </package>
```

Then, in `persistor.properties`, add:

```
isis.persistor.datanucleus.impl.datanucleus.Mapping=sqlserver
```

This will cause DataNucleus to also search for `ApplicationUser-sqlserver.orm` files and use them if found.

See here for details on how to run the domain app (with example module) against SQL Server.

API and Implementation

The module defines a number of services and default implementations. The behaviour of the module can be adjusted by implementing and registerng alternative implementations.

PasswordEncryptionService

The `PasswordEncryptionService` (responsible for authenticating *local* user accounts) is responsible for performing a one-way encryption of password to encrypted form. This encrypted version is then stored in the `ApplicationUser` entity's `encryptedPassword` property.

The service defines the following API:

```
public interface PasswordEncryptionService {
    public String encrypt(final String password);
    public boolean matches(final String candidate, final String encrypted);
}
```

The `PasswordEncryptionServiceUsingJbcrypt` provides an implementation of this service based on Blowfish algorithm. It depends in turn on `org.mindrot:jbcrypt` library; see above for details of updating the classpath to reference this library.

PermissionsEvaluationService

The `PermissionsEvaluationService` is responsible for determining which of a number of possibly conflicting permissions apply to a target member. It defines the following API:

```
public interface PermissionsEvaluationService {
    public ApplicationPermissionValueSet.Evaluation evaluate(
                final ApplicationFeatureId targetMemberId,
                final ApplicationPermissionMode mode,
                final Collection<ApplicationPermissionValue> permissionValues);
```

It is *not* necessary to register any implementation of this service in `isis.properties`; by default a strategy of allow-beats-veto is applied. However this strategy can be explicitly specified by registering the (provided) `PermissionsEvaluationServiceAllowBeatsVeto` implementation, or alternatively it can be reversed by registering `PermissionsEvaluationServiceVetoBeatsAllow`. Of course some other implementation with a different algorithm may instead be registered.

Default Roles, Permissions and Users

Whenever the application starts the security module checks for (and creates if missing) the following roles, permissions and users:

- `isis-module-security-admin` role
  - *allow changing* of all classes (recursively) under the `org.isisaddons.module.security.app` package
  - *allow changing* of all classes (recursively) under the `org.isisaddons.module.security.dom` package
- `isis-module-security-regular-user` role
  - *allow changing* (ie invocation) of the `org.isisaddons.module.security.app.user.MeService#me` action
  - *allow viewing* of the `org.isisaddons.module.security.app.dom.ApplicationUser` class
  - *allow changing* of the selected "self-service" actions of the `org.isisaddons.module.security.app.dom.ApplicationUser` class
- `isis-module-security-fixture` role
  - *allow changing* of `org.isisaddons.module.security.fixture` package (run example fixtures if prototyping)
- `isis-module-security-admin` user
  - granted `isis-module-security-admin` role
- `isis-applib-fixtureresults` role
  - *allow changing* of `org.apache.isis.applib.fixturescripts.FixtureResult` class

This work is performed by the `SeedSecurityModuleService`.

Known issues

There are no known issues at this time, however there are a few recognised limitations in the current implementation:

- It is not possible to set permissions on the root package. The workaround is to specify for `org` or `com` top-level package instead.
- The concept of "app tenancy" might be pulled out into its own concept (since other modules also use it).

We have the following ideas for future features:

- enhance the auto-creation of delegated user accounts, so that an initial role can be assigned and the user left as enabled
- users could possibly be extended to include user settings, refactored out from settings subdomain
- hierarchical roles

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/spi/security/impl -D excludeTransitive=true
```

which, excluding Apache Isis itself, returns these compile/runtime dependencies:

```
org.apache.shiro:shiro-core:jar:1.2.3
org.mindrot:jbcrypt:jar:0.3m
```

For further details on 3rd-party dependencies, see:

- Apache Shiro
- jBCrypt

  (Apache-like license); only required if the `PasswordEncryptionServiceUsingJBcrypt` service is configured.