Quickstart

Table of Contents

The quickstart is a Maven archetype intended to be used as a template for new applications.

Like the Apache Isis framework's own simpleapp archetype, the application provides a `SimpleObject` to adapt and copy, along with supporting unit tests, integration tests and BDD (cucumber) specs.

The application generated from the archetype is also preconfigured with the dependencies for all of the modules available in the Incode Platform, with auditing, command profiling, security, flywaydb and (feature) togglz enabled by default. The application also configures the *Quartz scheduler* for background scheduling, and the *Jolokia* servlet, allowing consoles such as *hawt.io* to monitor the internal state of the webapp.

Prerequisites

The prerequisite software is:

- Java JDK 8
- maven 3 (3.5.0 or later is recommended).

We also strongly recommend an IDE; the Incode Platform developers use JetBrains IntelliJ Community Edition.

Running the archetype

Use Maven to generate your application from the `quickstart-archetype`. For example:

```
mvn archetype:generate  \
    -D archetypeGroupId=org.incode.platform.archetype \
    -D archetypeArtifactId=quickstart-archetype \
    -D archetypeVersion=1.16.2 \
    -D groupId=com.mycompany \
    -D artifactId=myapp \
    -D version=1.0-SNAPSHOT \
    -B
```

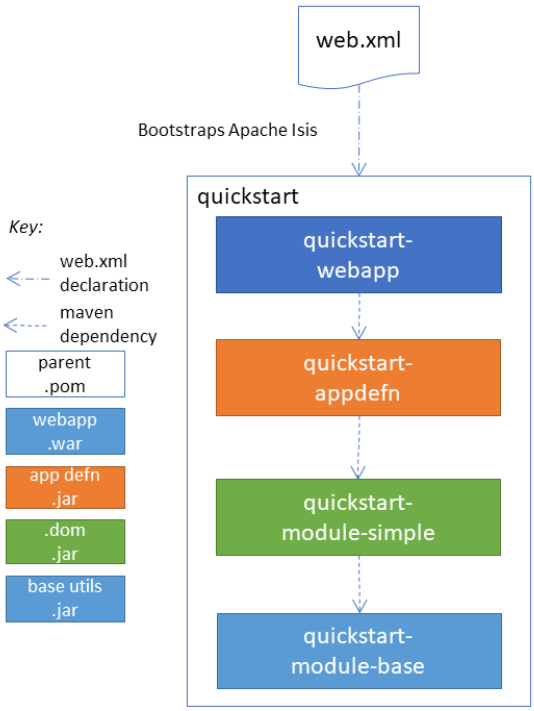
Change the `groupId` and `artifactId` as required.

The application will be generated in a directory named after the artifact (`myapp` in the example above):

```
cd myapp
```

Project Structure

The quickstart app consists of five Maven modules: a parent module and four submodules. The diagram below shows their dependencies:



where:

- `quickstart-module-simple` is where the business logic resides for an initial "simple" module.

  This consists of nothing more than a `SimpleObject` and supporting menu and repository (basically the same domain as generated by Apache Isis' simpleapp archetype).

  In your own application we recommend that you clone this module for each discrete module that makes up your business domain. Doing this from the very beginning will help ensure your app doesn't become the proverbial "big ball of mud".

- It's quite common to factor out base/abstract classes to remove boilerplate, or to hold utility classes. The `quickstart-module-base` module is an ideal place for this, intended as a base module that all business modules will depend upon. It is also where the example togglz feature toggle `enum` class resides.

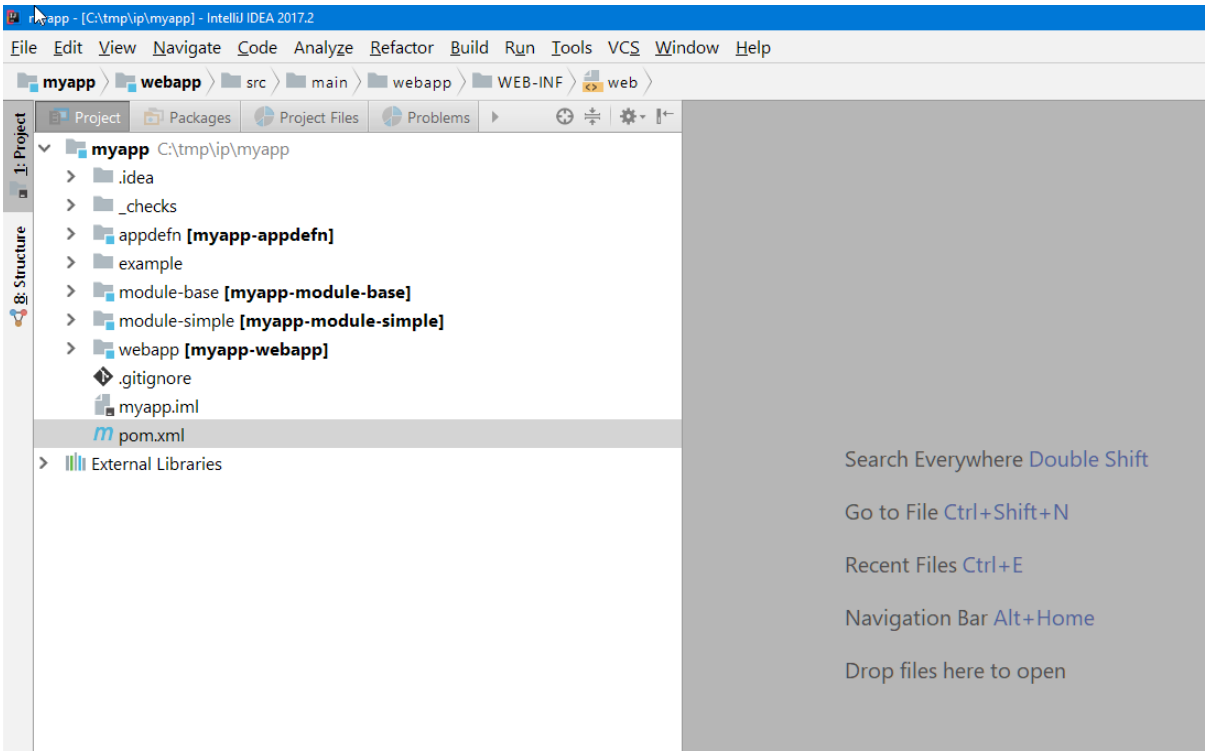
- The `quickstart-appdefn` module holds the application's implementation of the AppManifest interface, which defines the modules that make up the application. The quickstart's implementation is `domainapp.appdefn.DomainAppAppManifest`.

  It's also a useful place to define fixtures for demoing prototypes to the business, as well as supporting services that might be needed to help configure modules. The quickstart includes implementations of services for both the security spi module and the togglz (feature toggle) module.

- Finally, the `quickstart-webapp` module is what packages up the application as a `.war` file (the `WEB-`

`INF/web.xml` shown in the diagram actually lives in this module). There's very little Java code in this module: the most significant class is `DomainAppWicketApplication` (extends `IsisWicketApplication`), required by the Wicket viewer.

You'll want to load the generated app into an IDE; the figure below shows the app as loaded into IntelliJ:



by default the `example` modules are excluded from the project:

See the Apache Isis documentation for further guidance.

Adding/removing modules

By default the `DomainAppAppManifest` app manifest has dependencies on *all* of the SPI services, *all* of the wicket components, the (feature) togglz extension module, the settings subdomain (used by the togglz extension), and the paraname8 metamodel facet.

To exclude the modules that you don't require, delete or comment entries in the app manifest. To keep things tidy, you should probably also remove the corresponding `<dependency>`s from your application's `pom.xml` files.

Conversely, the application does *not* include any of the example subdomains nor libraries. To include these, take a look at the quickstart with example usage, or just read the individual documentation for the required module(s). In many cases all that's required is to update the `pom.xml` to reference the required module, and reference its `XxxModule` class in your app manifest.

Building (and tests)

To build the generated app:

```
mvn clean install
```

This will compile the code and automatically run all tests (discussed in more detail below), and it also package up the application as a WAR file (namely, `webapp/target/myapp.war`).

Tests

The generated application includes unit tests, integration tests and BDD cucumber specifications. All of these are executed using the surefire plugin, which is configured to follow a naming convention:

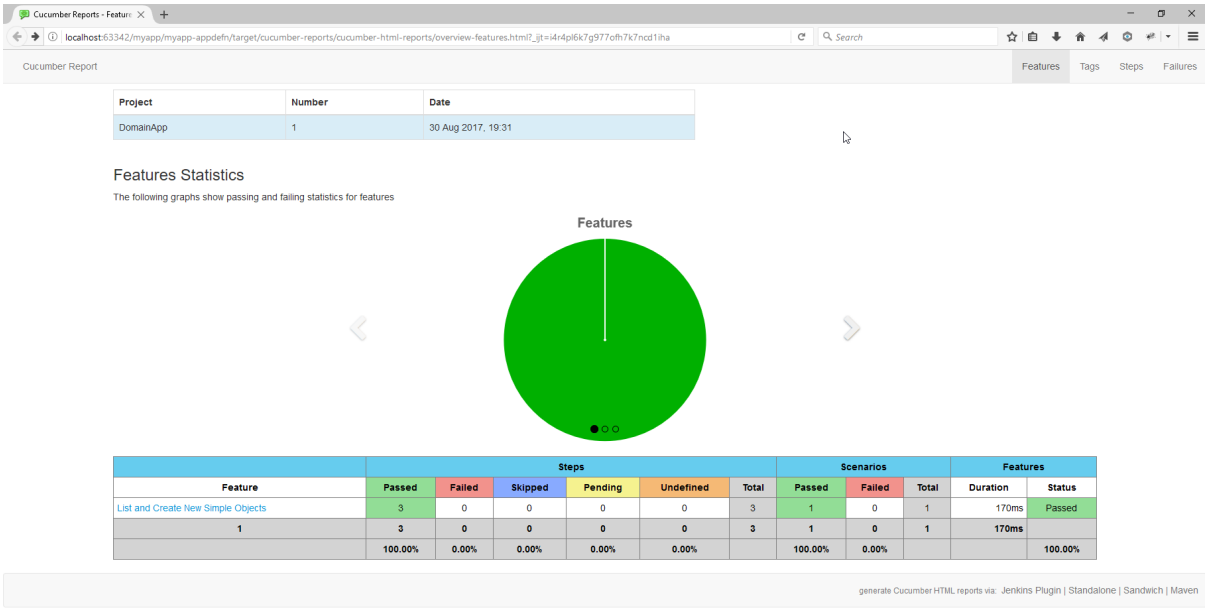
Table 1. Test configurations

Test type	Naming convention	Used in Module(s)	Disabled using	Generated Reports
Unit tests	`*_Test`	`module-simple`	`-DskipUT`	`target/surefire-unittest-reports`
Integration tests	`*_IntegTest`	`module-simple`; `appdefn`	`-DskipIT`	`target/surefire-integtest-reports`
BDD specs	`*_IntegBddSpec`	`appdefn`	`-DskipBS`	`target/surefire-integbddspecs-reports`

It's also possible to disable all tests using the usual `-DskipTests` flag.

The reports are generated relative to the module that contains the tests. Such reports can be picked up by your continuous integration server.

In addition, the BDD specs also generate a (very simple) website at `target/cucumber-html-report/index.html`, and a (much richer) website at `target/cucumber-reports/cucumber-html-reports/overview-features.html`.



One thing to be aware of: if a unit test or integration test fails, then the build will break. However, note that if a BDD spec "fails" (eg due to a missing binding), then the build continues; the failure is shown in the generated cucumber report instead.

The configuration of tests in the Maven `pom.xml` project files is done using the *surefire* and *cucumberreporting* maven mixins, discussed below.

Metamodel validation

The generated application also configures the Apache Isis maven plugin to validate the domain application. For example, if a supporting method for an action `findByName` is misspelt, then this will be reported and the build will fail. In the quickstart app the metamodel validation plugin is configured in the `module-simple` module.

Running metamodel validation does require bootstrapping the application, so will lengthen the overall build time. If required, the metamodel validation can be skipped using `-Dskip.isis-validate`.

The configuration in the Maven `pom.xml` project files is done using the *isis-validate mavenmixin*, discussed below.

Swagger spec

The generated application also configures the Apache Isis maven plugin to generate a Swagger spec file. This can, for example, be used to generate stubs in various programming languages.

In the quickstart app, the plugin is configured to run in the `appdefn` module (so providing a single spec for the entire app). The configuration is done using the *isis-swagger* mavenmixin, discussed below.

Generating the swagger spec does require bootstrapping the application, so will lengthen the overall build time. If required, the swagger generation can be skipped using `-Dskip.isis-swagger`.

When run, it generates swagger files in `target/generated-resources/isis-swagger`, relative to module that configures the plugin.

Running

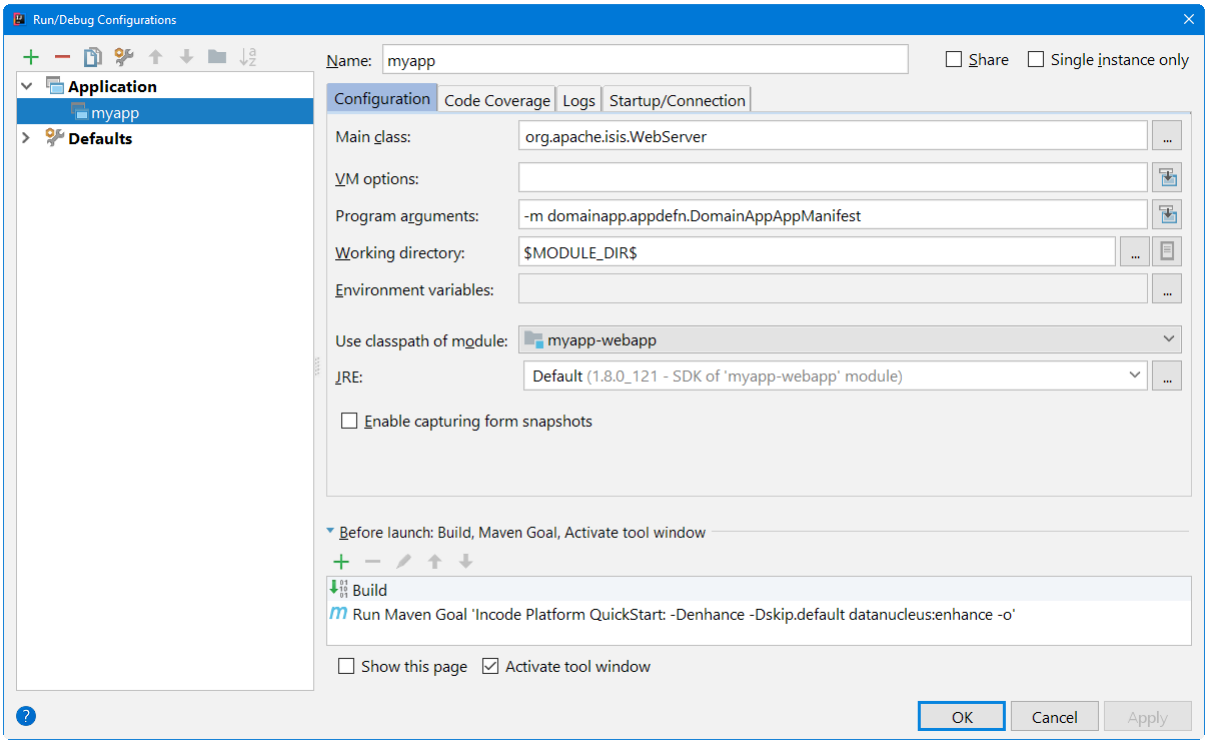
There are a number of ways that you can run your application.

IDE

When developing an Apache Isis application, you'll typically want to run the app from within the IDE. Not only does this reduce the feedback loop (no need to package and deploy, for example), you can also very easily set up debugging breakpoints and the like.

To run the app from the IDE, we use Apache Isis' `org.apache.isis.WebServer`, which runs the app as a command line application. This starts up an embedded Jetty instance configured to run the webapp.

For example, here's the launch configuration to run the generated application from IntelliJ:



That is to say:

- main-class: `org.apache.isis.WebServer`
- program args: `-m domainapp.appdefn.DomainAppAppManifest`

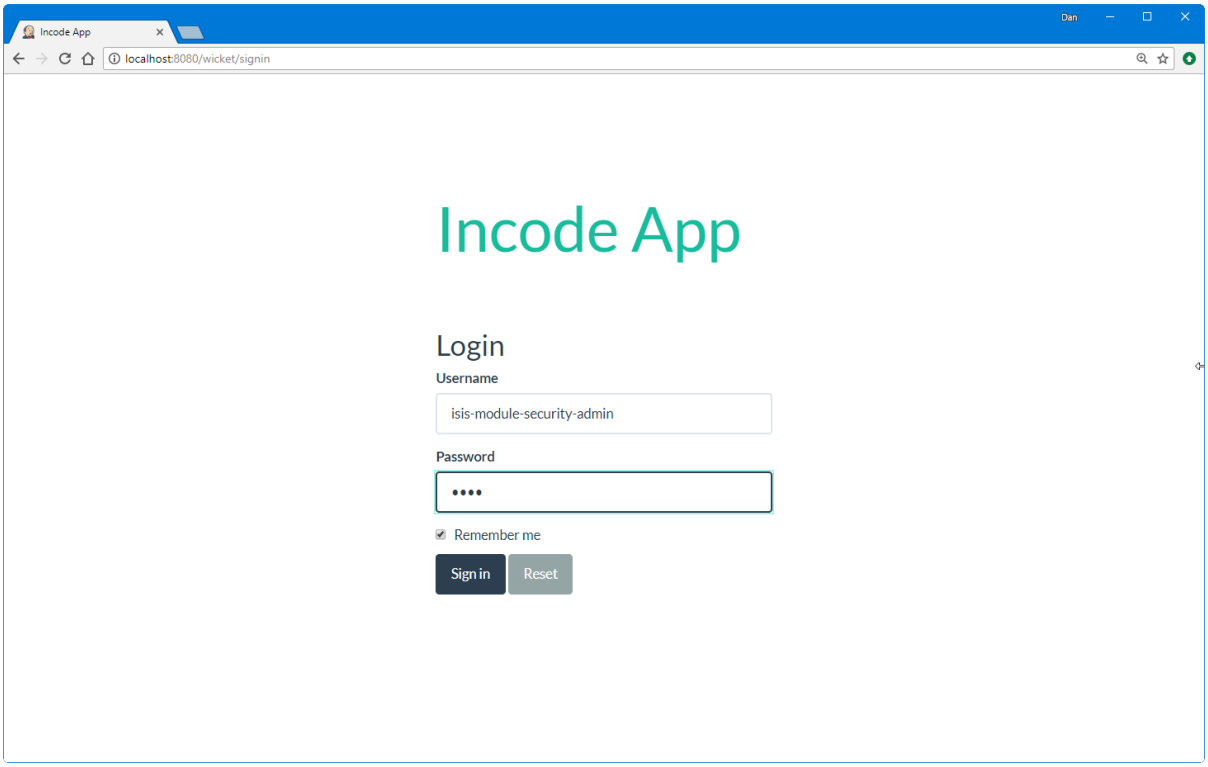
using:

```
mvn -Denhance -Dskip.default datanucleus:enhance -T1C -o
```

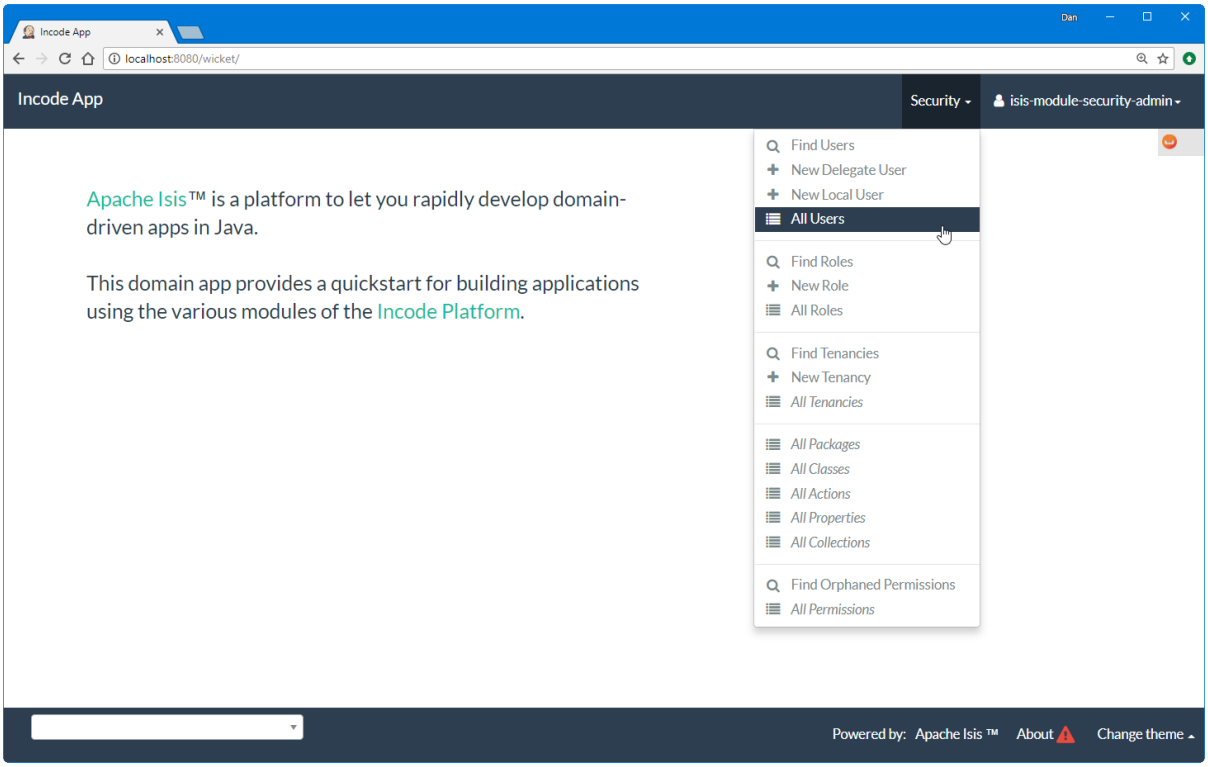
run against the parent pom to enhance the JDO domain entities. See the Apache Isis documentation for further guidance.

When the application is started this way, it runs without any fixtures, that is, using

`domainapp.appdefn.DomainAppAppManifest` configured in `webapp/WEB-INF/isis.properties`. The only user account that is available therefore is the default superuser provided by the security module, namely `isis-module-security-admin`/`pass`.

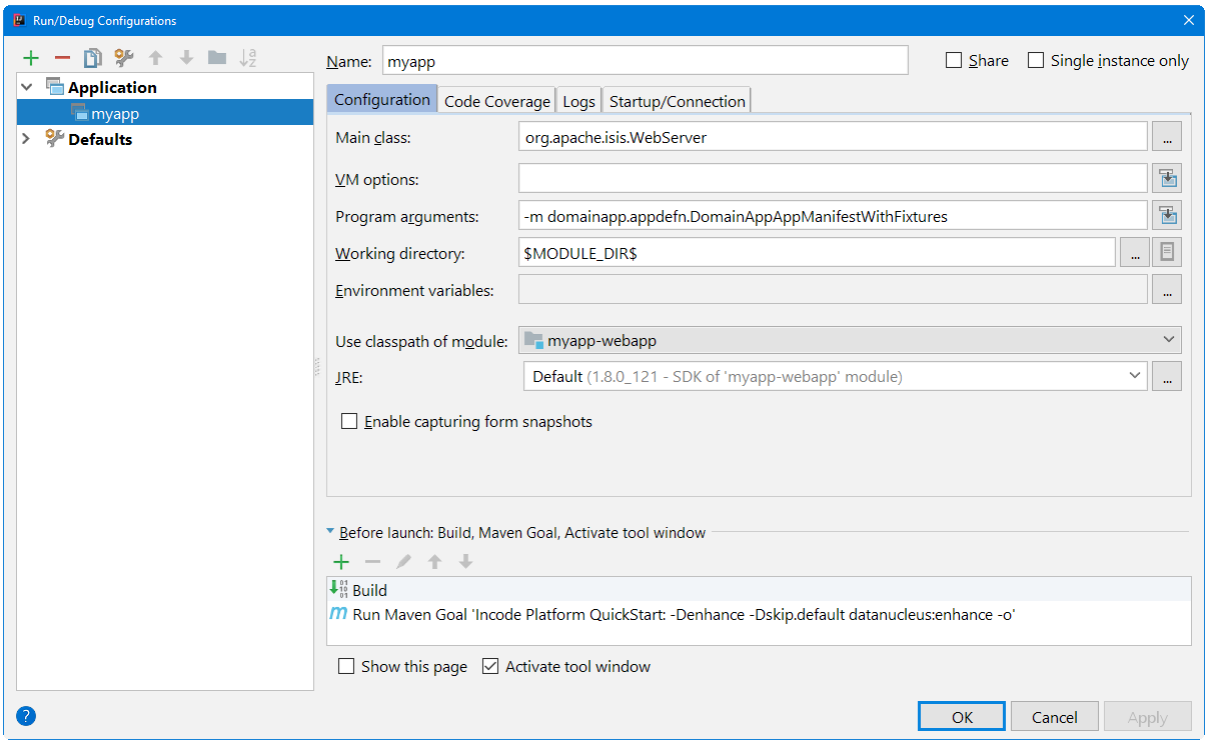


The home page shows only the security module's menu, not the domain app's entities (`SimpleObject's menu etc):



Manually setting up a user and permissions to be able to access the business domain would be tedious to say the least. So instead we can use an extended version of the app manifest which will also run some fixtures. These set up a user account and also some dummy data.

For example, here's the updated launch configuration using the app manifest:



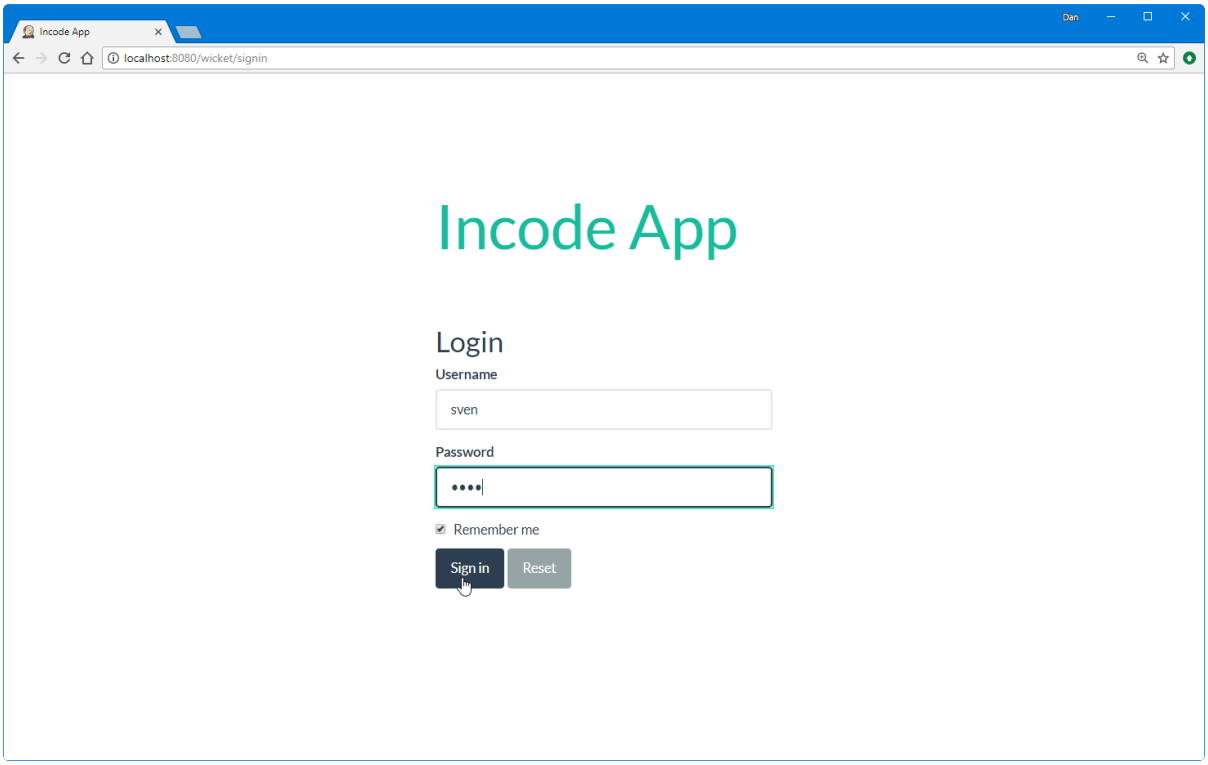
That is to say:

- program args: `-m domainapp.appdefn.DomainAppAppManifestWithFixtures`

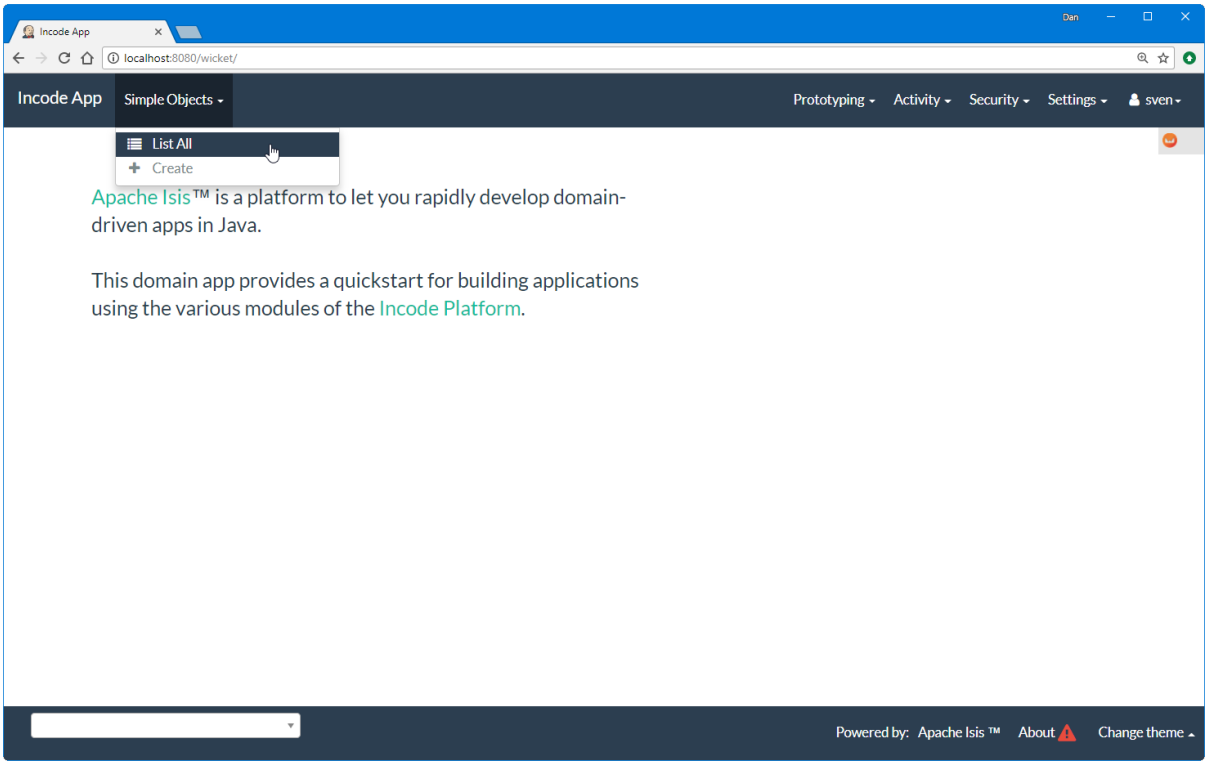
which supplies additional fixtures using:

```
@Override
protected void overrideFixtures(final List<Class<? extends FixtureScript>>
fixtureScripts) {
    fixtureScripts.add(SimpleObject_data.PersistScript.class);
    fixtureScripts.add(SeedSuperAdministratorRoleAndSvenSuperUser.class);
}
```

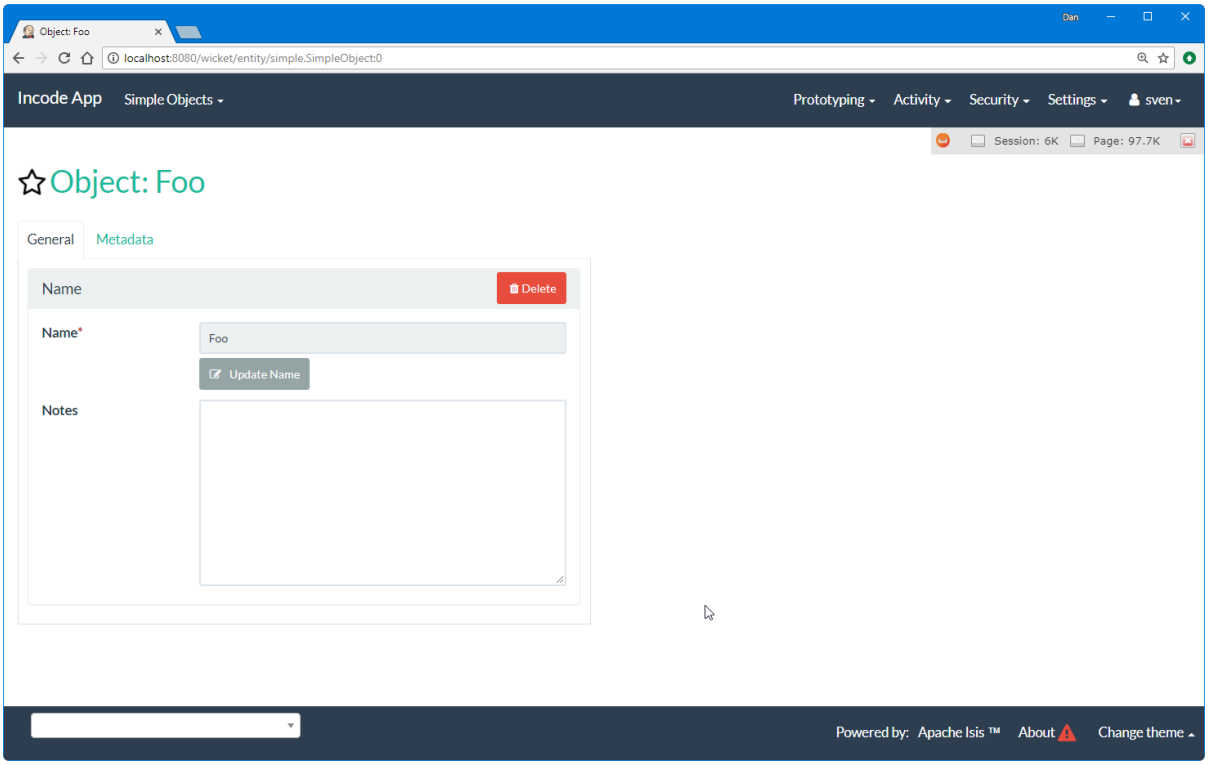
When we run the app again, we can now log in using a new `sven/pass` user account:



The home page now shows a `Simple Objects` menu:



from whence we can access the `SimpleObject`s also created by the fixture:



Note also the additional menus on the right-hand side to access other modules set up and accessible to the `sven` superuser. These are discussed in more detail below.

mvn (jetty:run)

If you want to run the application without even loading it into the IDE, then you can do this using the maven Jetty plugin:

```
mvn -pl webapp jetty:run
```

The "-pl webapp" argument indicates that the command is run in the "webapp" subdirectory/submodule.

However, note that this will run with the standard `DomainAppAppManifest`, as configured in `WEB-INF/isis.properties`; the only user account is `isis-module-security-admin`.

More than likely you'll want to override the manifest; you can do this using a system property. For example:

```
mvn -pl webapp jetty:run \
       -Disis.appManifest=domainapp.appdefn.DomainAppAppManifestWithFixtures
```

The configuration in the Maven `pom.xml` project files for the jetty plugin that provides this feature is done using the *jettywar* mavenmixin, discussed below.

Webapp

The application can also be run in a servlet container, using the `.war` file generated from the usual build. For example:

```
mvn install \
     -DskipTests -Dskip.isis-validate -Dskip.isis-swagger
```

will result in a `xxx.war` (where xxx is the parent project's `artifactId`), generated in `webapp/target`. This can then be deployed to the servlet container in the normal way. For example, if deploying to Apache Tomcat, just copy to the `$TOMCAT_HOME/webapps/` directory.

If you want to change any of the configuration properties (eg the app manifest to run on start up), note that it is possible to override the configuration externally. See the Apache Isis docs for further guidance.

The configuration in the Maven `pom.xml` project files for maven's war plugin is done using the *jettywar* mavenmixin, discussed below.

Standalone

Yet another alternative is to build the webapp to run standalone as a single "uber-jar", once again using Jetty as an embedded instance. This could be useful for example to distribute standalone prototype of your application for review.

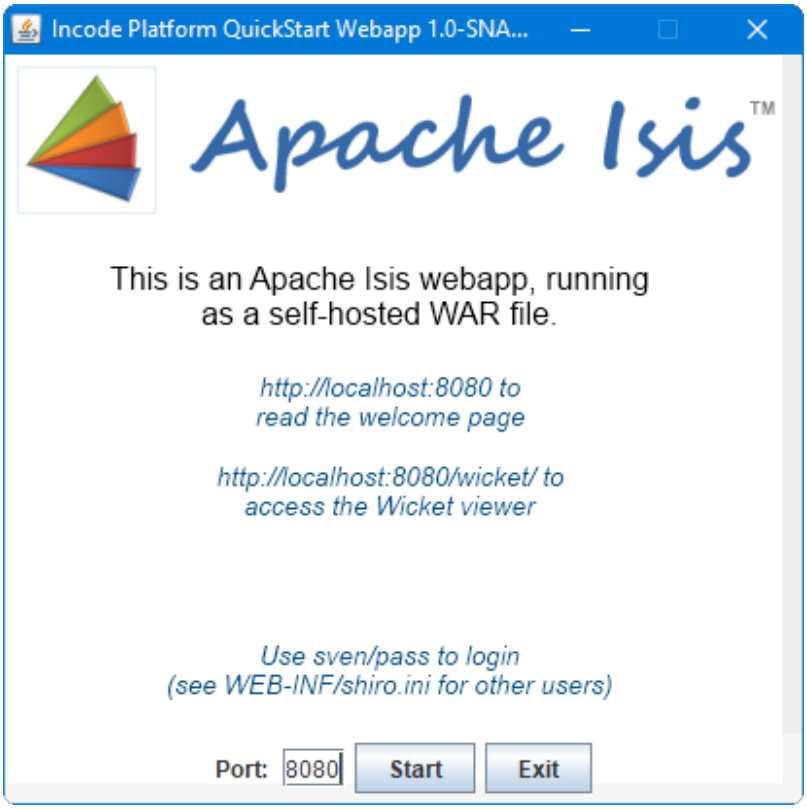
Building the standalone jar is done by setting the `-Dmavenmixin-jettyconsole` property. For example:

```
mvn install \
    -Dmavenmixin-jettyconsole \
    -DskipTests -Dskip.isis-validate -Dskip.isis-swagger
```

will result in a `xxx-webapp-1.0-SNAPSHOT-jetty-console.war` (where xxx is the project's `artifactId`) generated in `webapp/target`. This can then be running using java, for example:

```
java -jar webapp/target/myapp-webapp-1.0-SNAPSHOT-jetty-console.war
```

This displays a Swing UI from which the webapp can be started.



It's also possible to run headless, or to change the port. And again, the manifest can be changed using a system property. For example:

```
java -Disis.appManifest=domainapp.appdefn.DomainAppAppManifestWithFixtures \
     -jar webapp/target/myapp-webapp-1.0-SNAPSHOT-jetty-console.war \
          --headless \
          --port 9090
```

The configuration in the Maven `pom.xml` project files for the jettyconsole plugin that provides this feature is done using the *jettyconsole* mavenmixin, discussed below.

Docker

Finally, it's also possible to package up and run the webapp as a Docker container.

- to package the webapp as a Docker image:

  ```
  mvn -pl webapp \
      -o \
      -Dmavenmixin-docker \
      -Ddocker-plugin.imageName=mycompany/myapp \
      docker:build
  ```

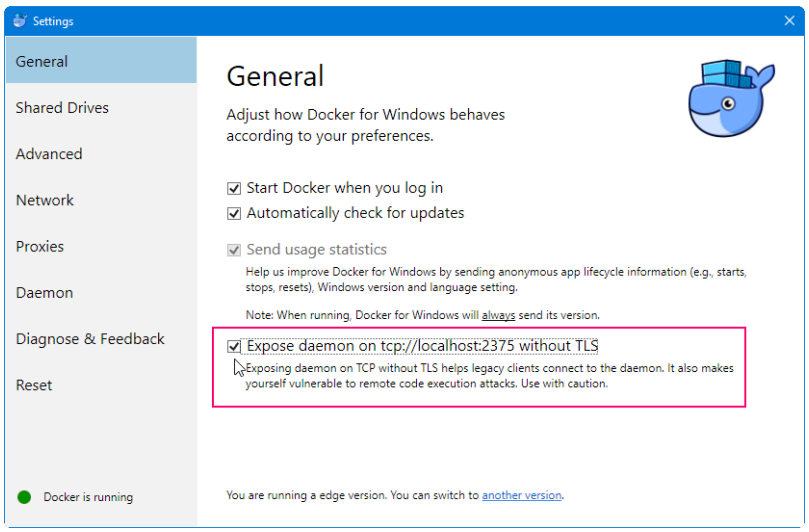
  where "imageName" is anything you want.

  The configuration for the docker plugin is done using the *docker* mavenmixin, discussed below.

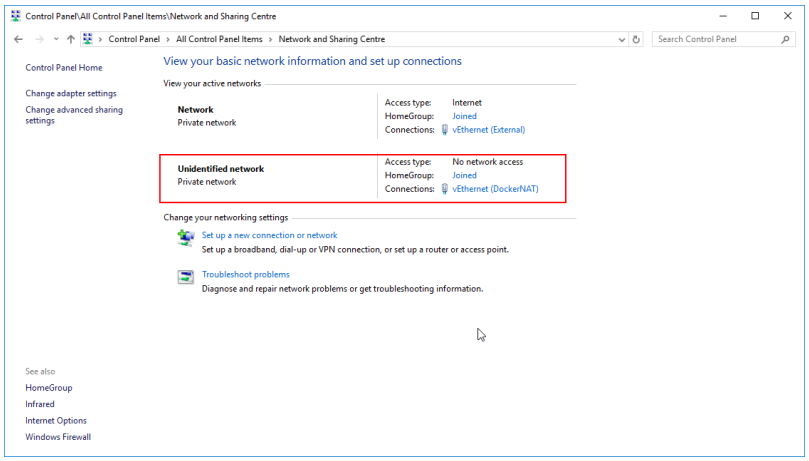
- and to run it:

```
docker container run -p 8080:8080 -d mycompany/myapp
```


On Windows, you may need to enable the Docker daemon first, otherwise the "mvn install" command above will fail:

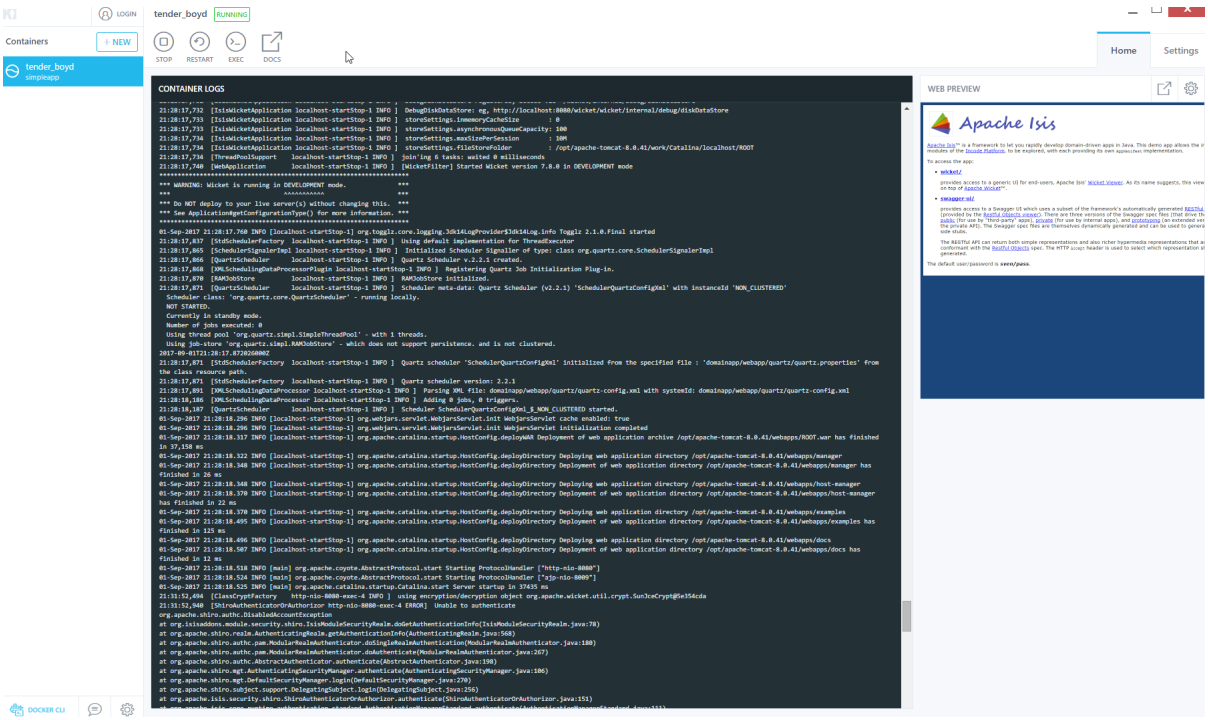


You might also need to ensure that the "Unidentified networks" are configured to be private:



This can be done using the "Local Security Policy" application.

This will bootstrap the application; `docker container ps` will show all running containers. Alternatively, Kitematic provides a simple UI to view and interact with running containers:



When the application is started this way, it runs without any fixtures. There are a variety of ways to override this but these easiest is probably to use the `$ISIS_OPT` environment variable, a set of key:value pairs concatenated together.

The `scripts/rundocker.sh` script is a simple wrapper that reads all properties from an externally specified configuration file and starts the docker container correctly. For example:

```
pushd scripts
rundocker.sh mycompany/myapp isis-overrides.properties -p 8080:8080
popd
```

where

- `mycompany/myapp` is the name of the image to be run
- `-p 8080:8080` is passed through to the `docker run` command

See the Apache Isis docs for further guidance on deploying with Docker.

Using an external database

All of the examples listed above run the application against an in-memory HSQLDB database. Obviously though at some point you'll want to persist your data against an external database.

To do so just requires that overriding four configuration properties that specify the JDBC driver, JDBC URL, user and password. It also (of course) requires that the JDBC driver is configured as a `<dependency>` in the webapp's `pom.xml`.

For example, to run the quickstart application against SQL Server:

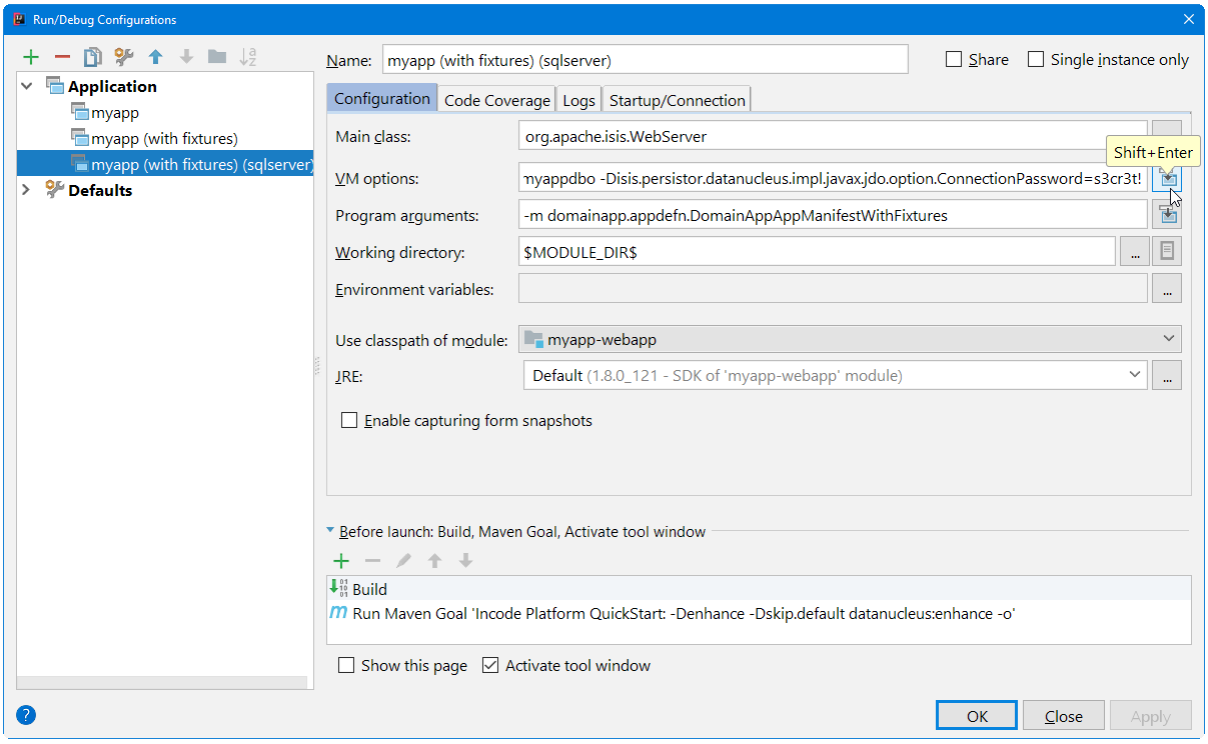
- create a new and empty database, eg `myappdb`, with corresponding user and password, `myappdbo/s3cr3t!`, say.
- edit the `webapp/pom.xml` to include the sqljdbc4 driver:

```
<profile>
    <id>jdbc-mssql</id>
    <activation>
        <property>
            <name>!skip.jdbc-mssql</name>
        </property>
    </activation>
    <dependencies>
        <dependency>
            <groupId>com.microsoft.sqlserver</groupId>
            <artifactId>mssql-jdbc</artifactId>
            <version>6.4.0.jre8</version>
        </dependency>
    </dependencies>
</profile>
```

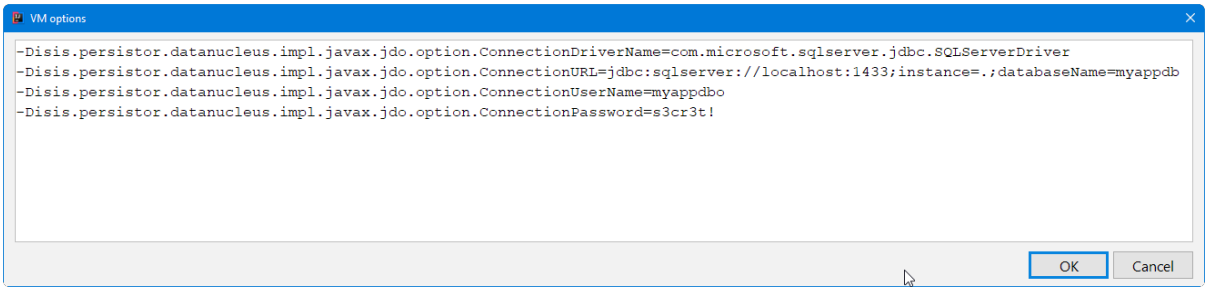
- change (by simply editing in `isis.properties`) or override (eg by passing in as `-D` system properties) the following configuration properties:

```
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionDriverName=com.microsoft
.sqlserver.jdbc.SQLServerDriver
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionURL=jdbc:sqlserver://loc
alhost:1433;instance=.;databaseName=myappdb
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionUserName=myappdbo
isis.persistor.datanucleus.impl.javax.jdo.option.ConnectionPassword=s3cr3t!
```

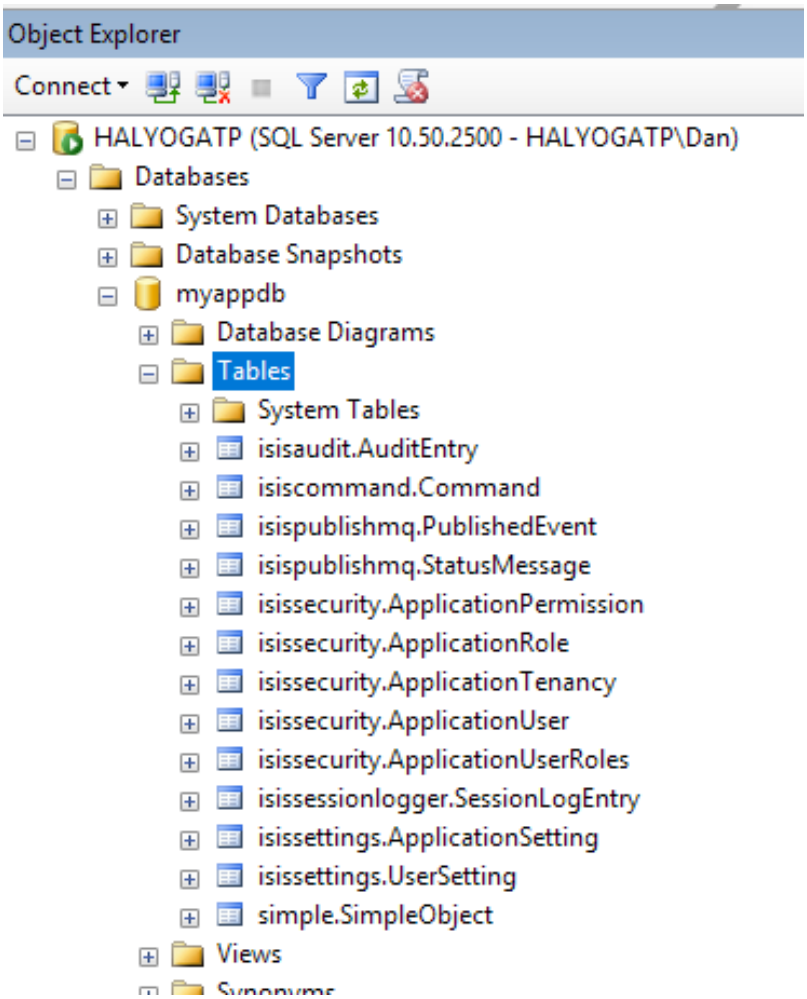
For example, an IntelliJ launch configuration can specify system properties:



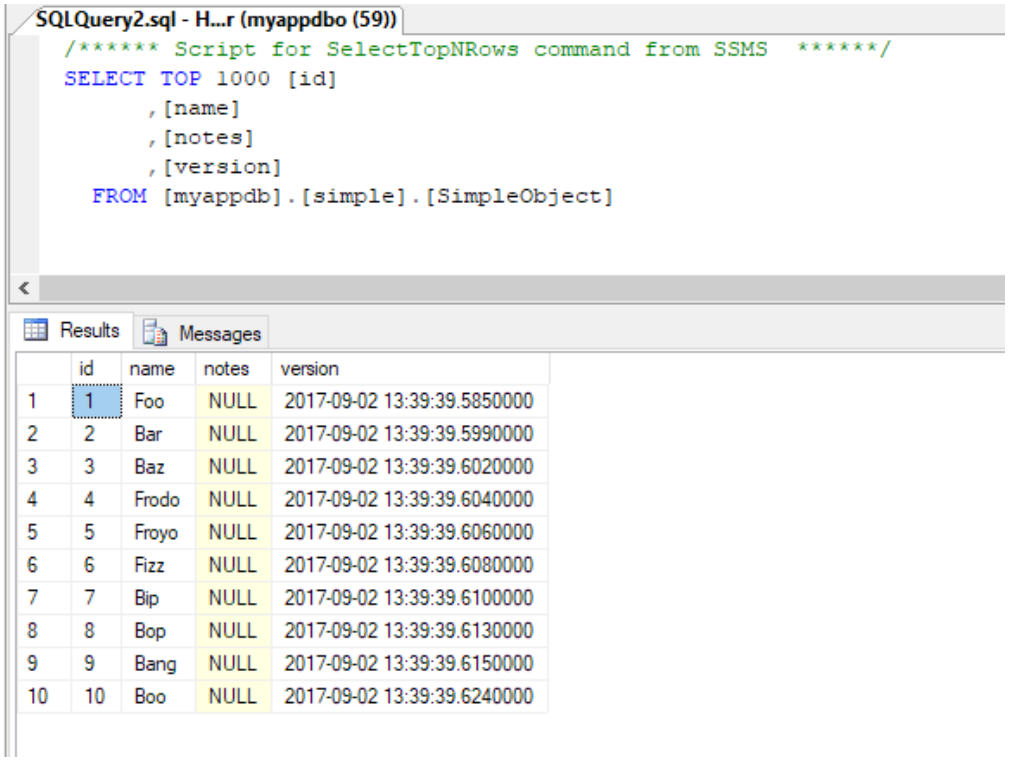
where the system properties dialog is:



When the application has started the tables should have been automatically created (by virtue of the `isis.persistor.datanucleus.impl.datanucleus.schema.autoCreateAll=true` configuration property in `isis.properties`):



with 10 `SimpleObject` instances created through the fixture:



If running against a persistent datastore, then remember that the fixture script should only be run the very first time you run up the application. Thereafter, switch to the regular app manifest (`domainapp.appdefn.DomainAppAppManifest`); otherwise you'll likely get INSERT errors on start up (trying to re-insert the same dummy data).

Modules

Now we've seen how to run the application, let's explore some of the features already configured in the quickstart.

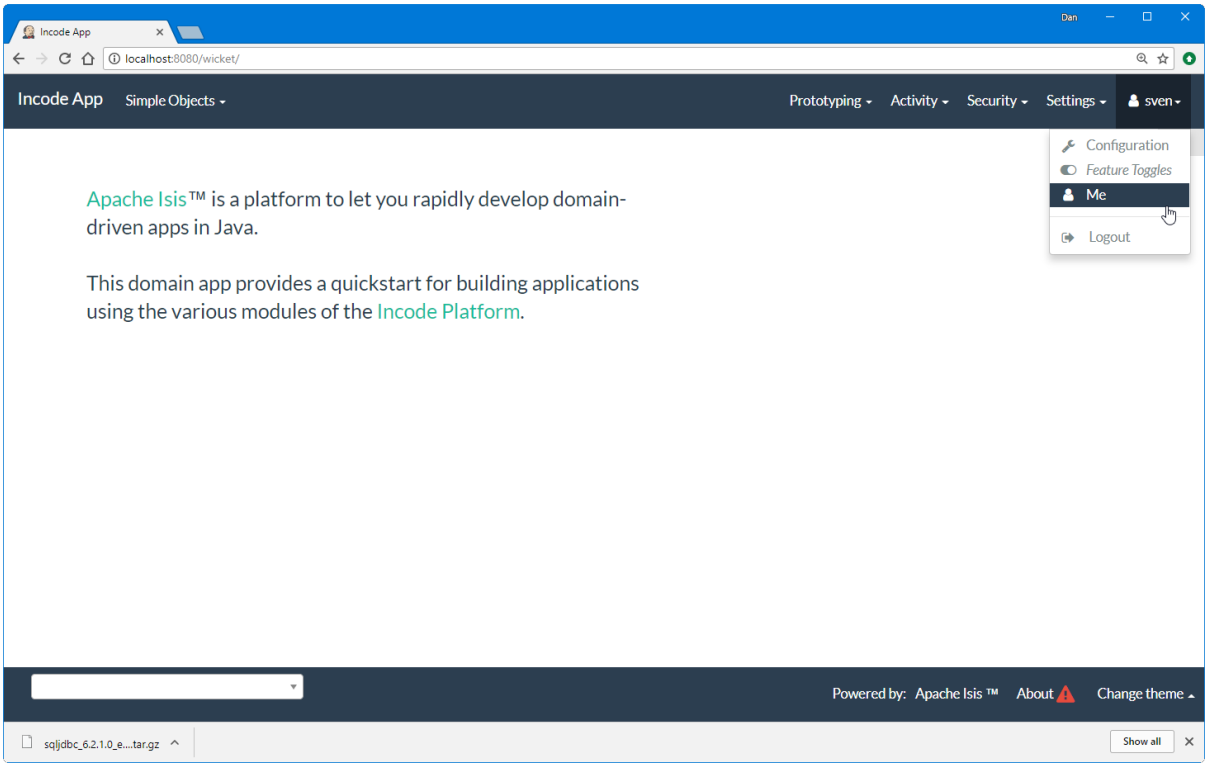
Security

The security module provides authentication and authorisation, implemented by an Apache Shiro Realm:

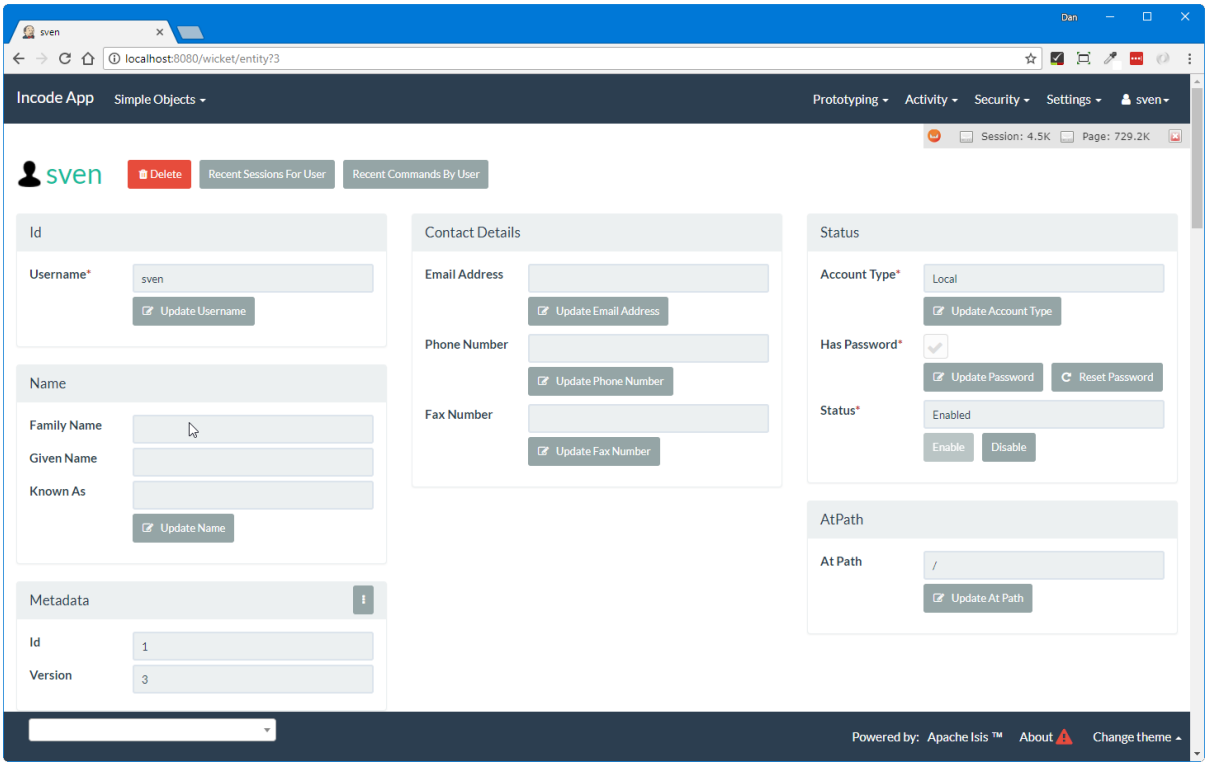
WEB-INF/shiro.ini file

```
incodePlatformSpiSecurityRealm = \
    org.isisaddons.module.security.shiro.IsisModuleSecurityRealm
securityManager.realms = $incodePlatformSpiSecurityRealm

authenticationStrategy = \

org.isisaddons.module.security.shiro.AuthenticationStrategyForIsisModuleSecurityRealm
securityManager.authenticator.authenticationStrategy = $authenticationStrategy
```

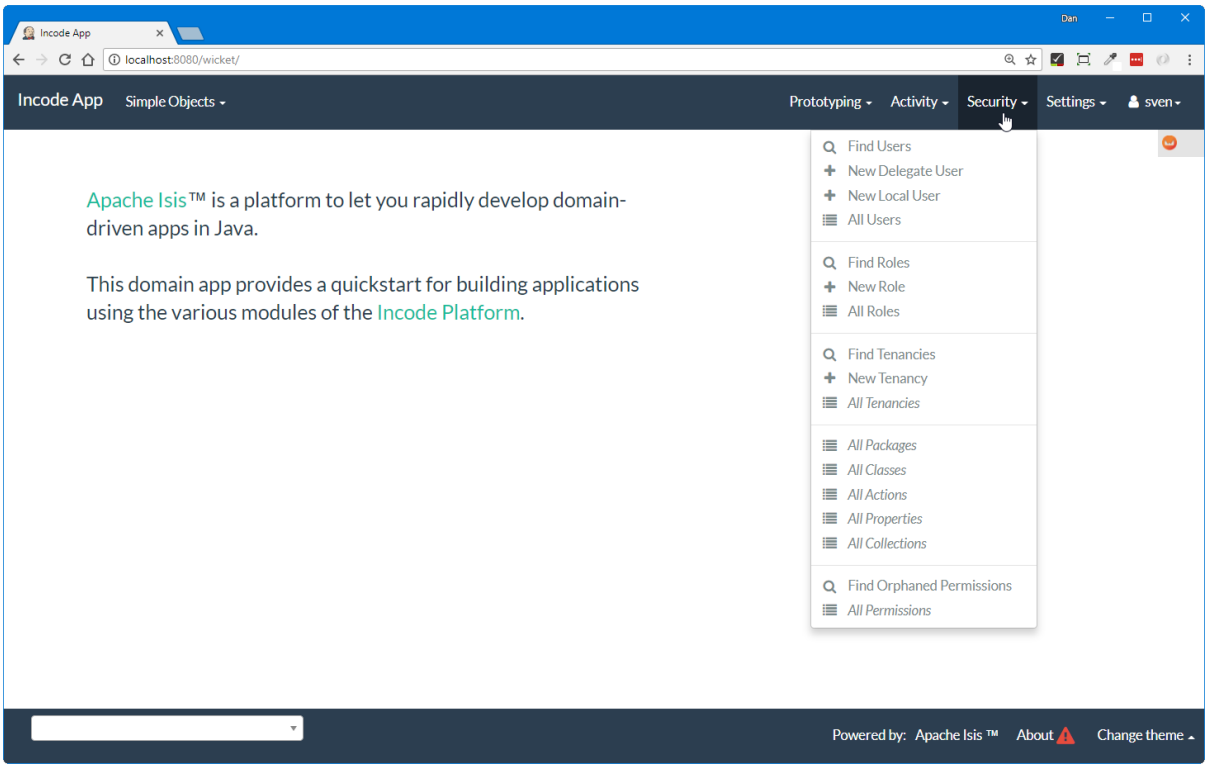
Each logged-in user has a corresponding `ApplicationUser` account, accessible from the "me" menu item:



which shows the current user:



Other functionality is available from the security menu:

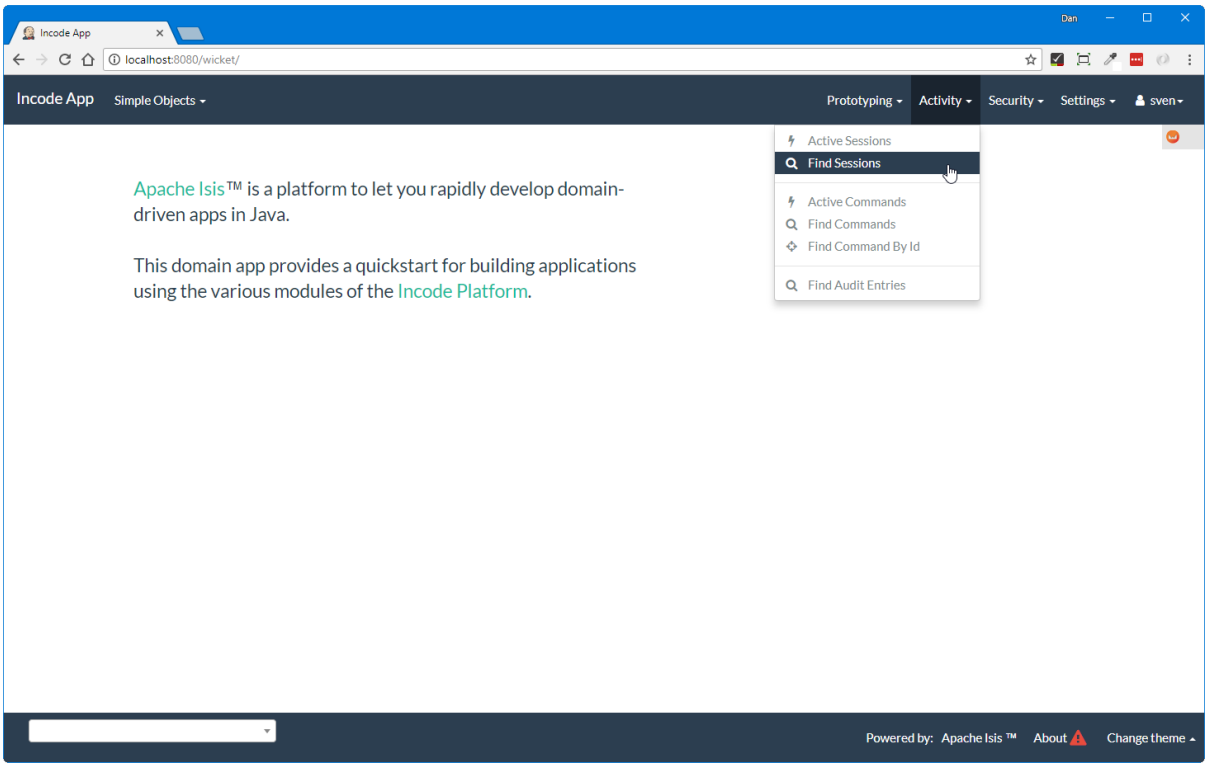


Note that the security seed data (defined in the `SeedSuperAdministratorRoleAndSvenSuperUser` fixture) also sets up a role required by the togglz module, also see below.

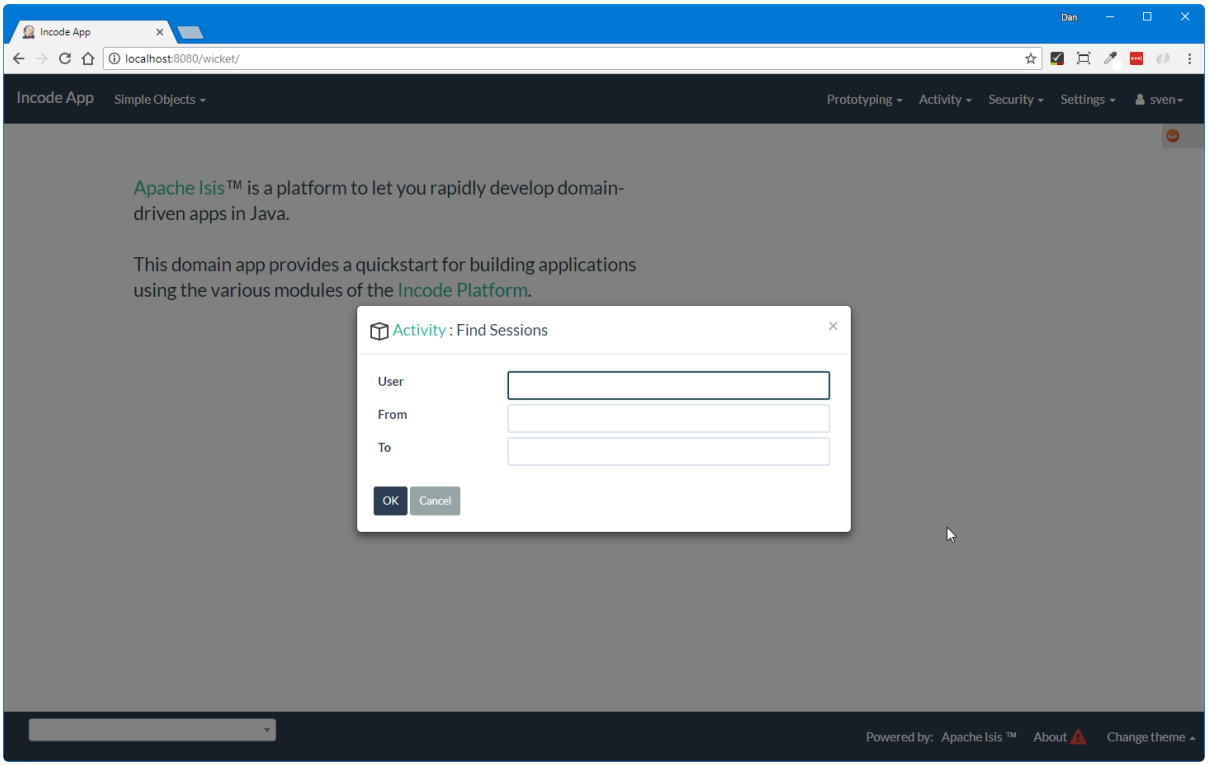
Session Logger

The session logger module records each user session as the user logs in or logs out (or is timed out automatically).

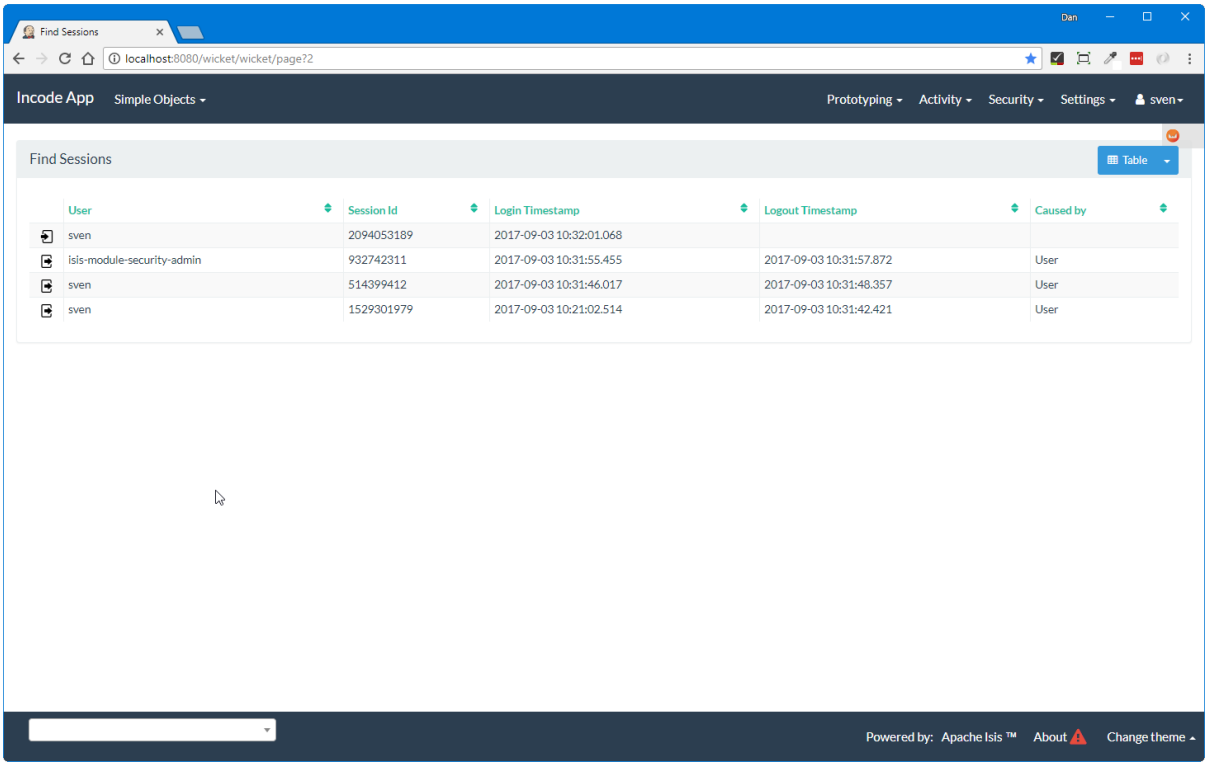
The session logger menu items are on the *Activity* menu:



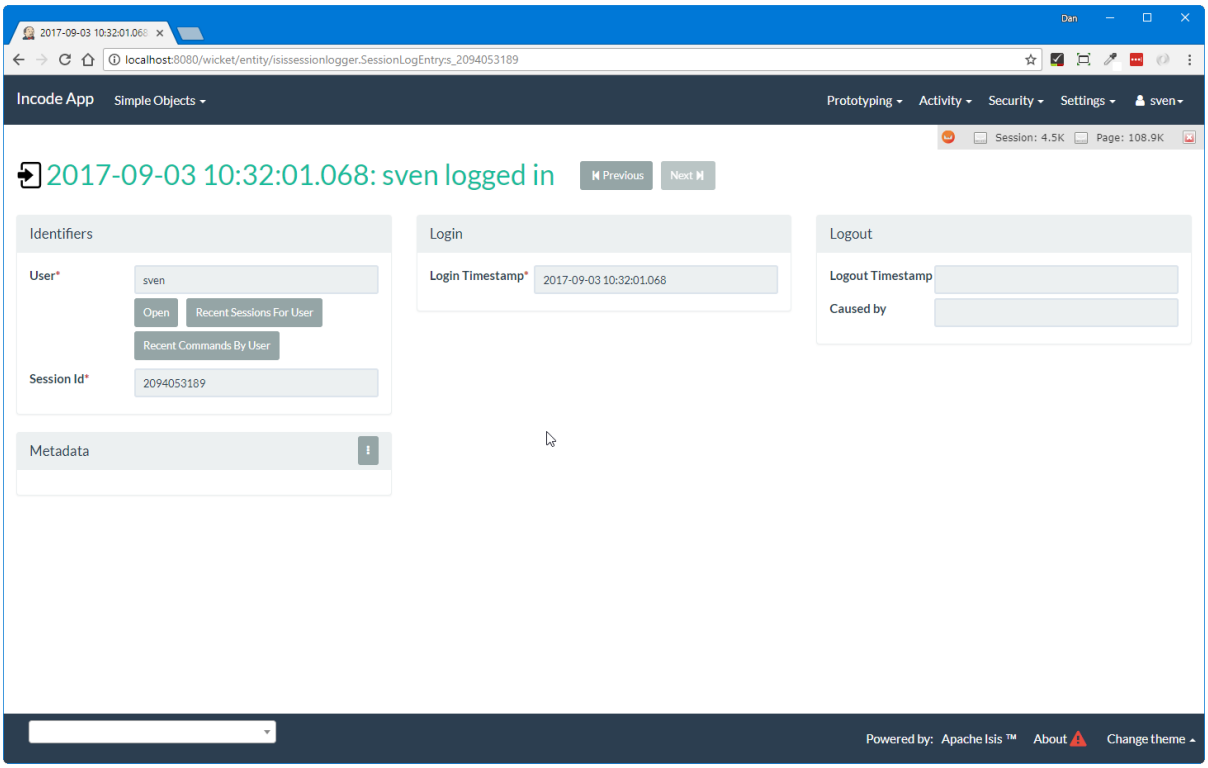
Sessions can be searched by date/time and user:



returning matching sessions:



which can be drilled into:



Commands

The session logger module captures actions and property edits as persistent `Command` objects. These commands represent the *intention* to invoke the action/edit the property, rather than the action invocation/property edit itself.

The quickstart app (when using the fixtures) disables commands "globally" in the app manifest:

DomainAppAppManifestWithFixtures

```
protected void disableAuditingAndCommandAndPublishGlobally(Map<String, String>
configurationProperties) {
    ...
    configurationProperties.put("isis.services.command.actions","none");
    configurationProperties.put("isis.services.command.properties","none");
    ...
}
```

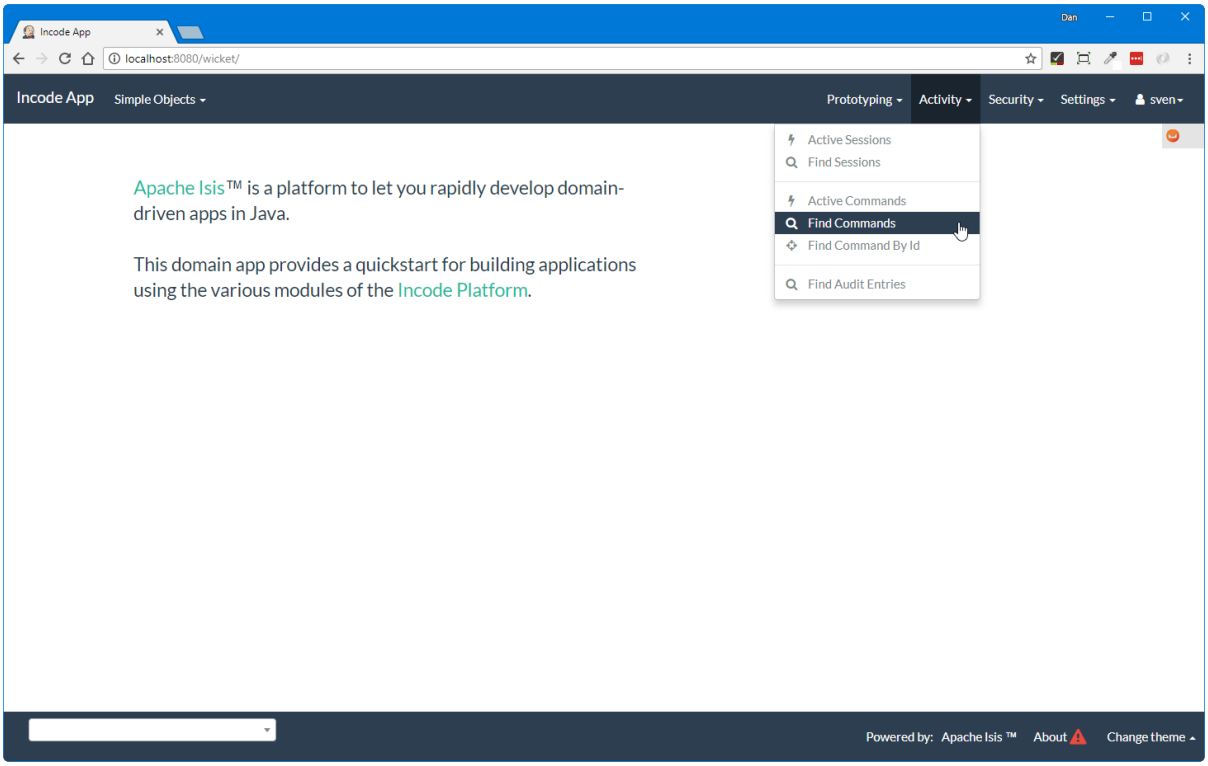
but the `SimpleObject` domain object re-enables command reification for both editing of the `notes` property;

```
    @Property(
            ...
            command = CommandReification.ENABLED,
            ...
    )
    private String notes;
```

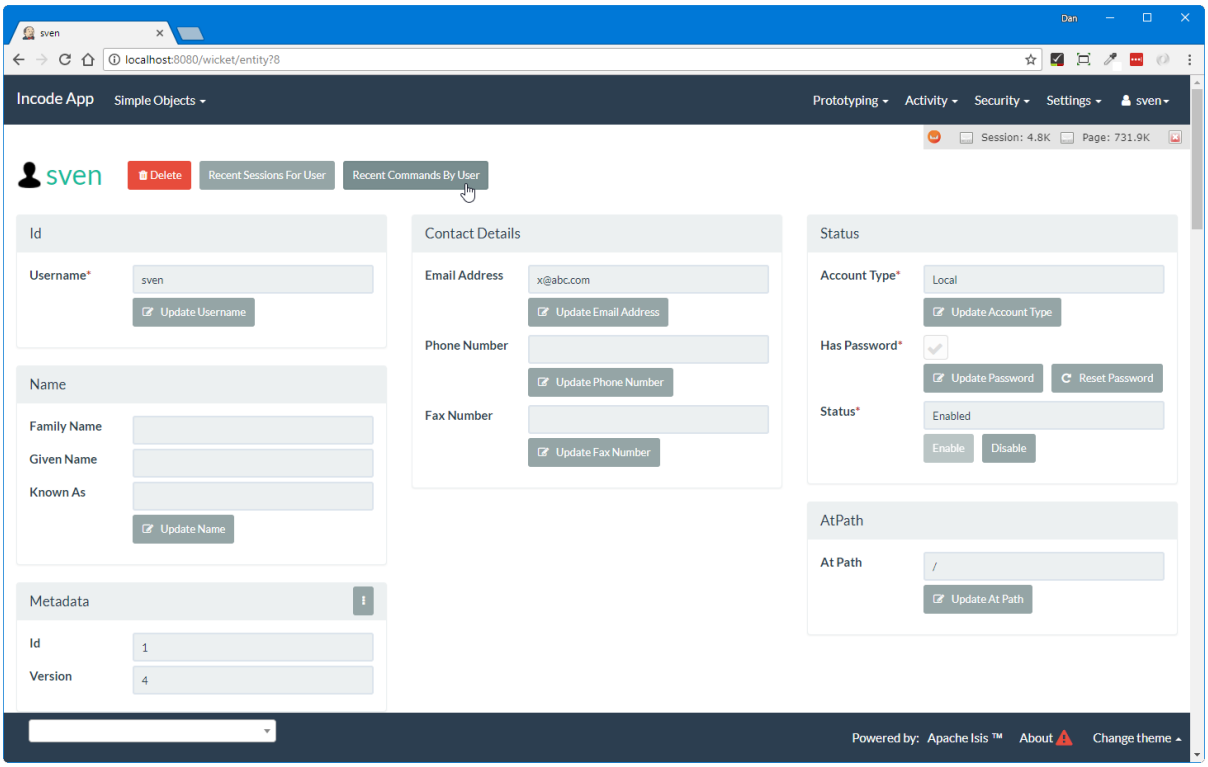
and for invoking the `updateName` action:

```
@Action(
        ...
        command = CommandReification.ENABLED,
        ...
)
public SimpleObject updateName( ... ) {  ... }
```

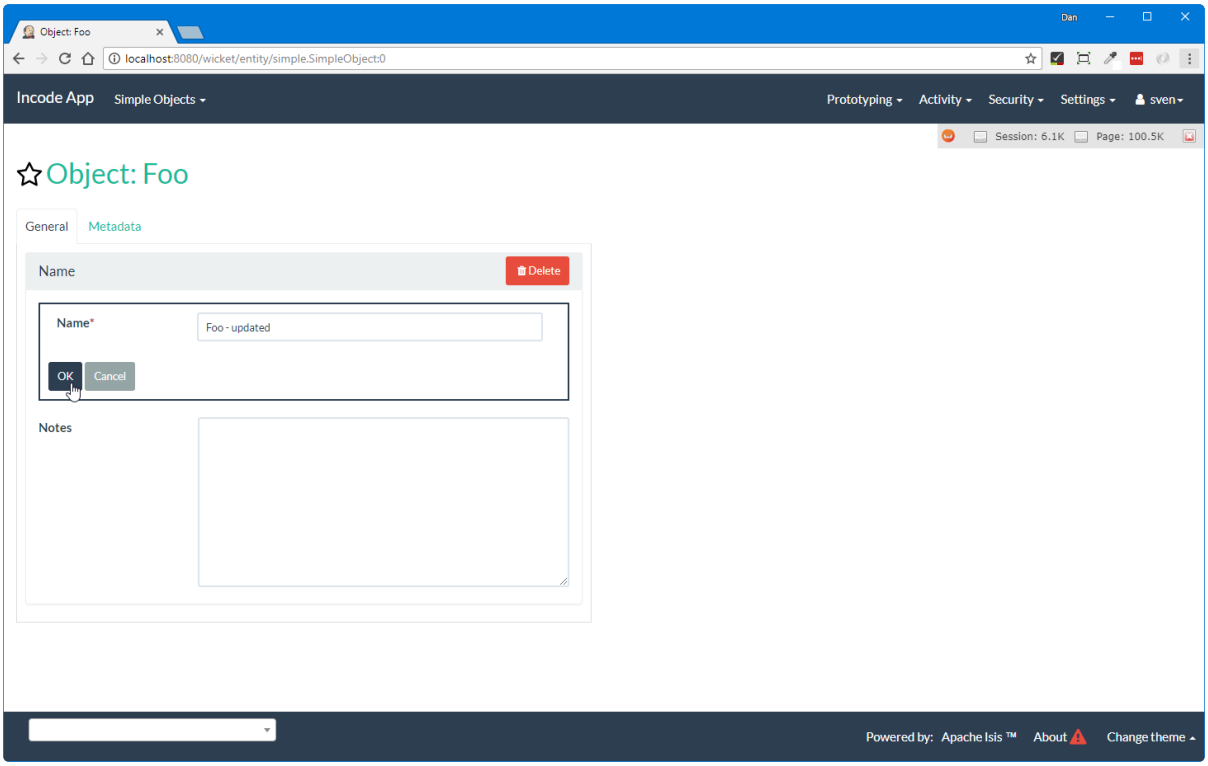
Persisted `Command` objects can be accessed in several ways. One option is to use the *Activity* menu:



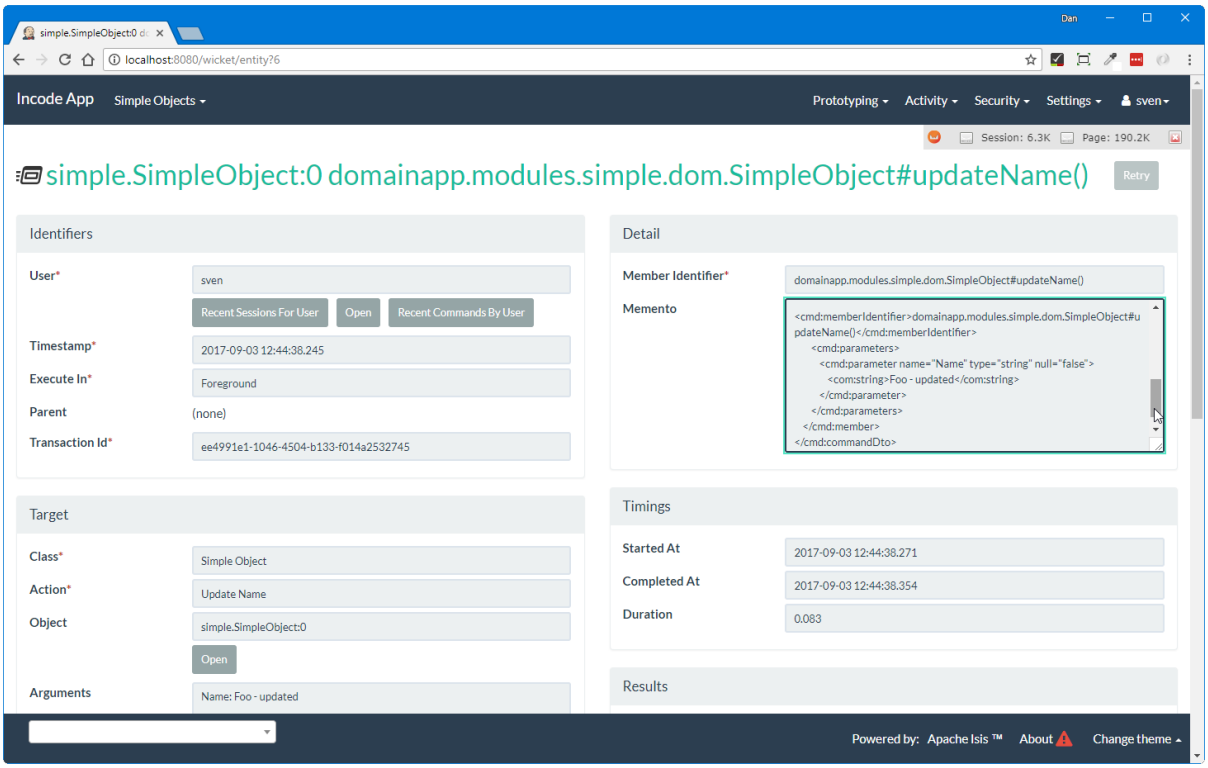
Alternatively, recent commands executed by an end-user can be found via their coresponding `ApplicationUser` object:



For example, invoking the `SimpleObject#updateName()` action:



results in this `Command`:



Using commands it's also possible to defer their invocation to be run in the background, either using `@Action#commandExecuteIn()` or using the `BackgroundService`. The quartz module is configured in the webapp to run such background commands, see below for details.

Auditing

Where commands capture the intention of a user to invoke an action/edit a property, audit records (as provided by the audit module) capture the effect of performing such an interaction. With the

quickstart app configuring both commands and audits, this provides excellent traceability of cause-and-effect.

The quickstart app (when using the fixtures) disables auditing "globally" in the app manifest:

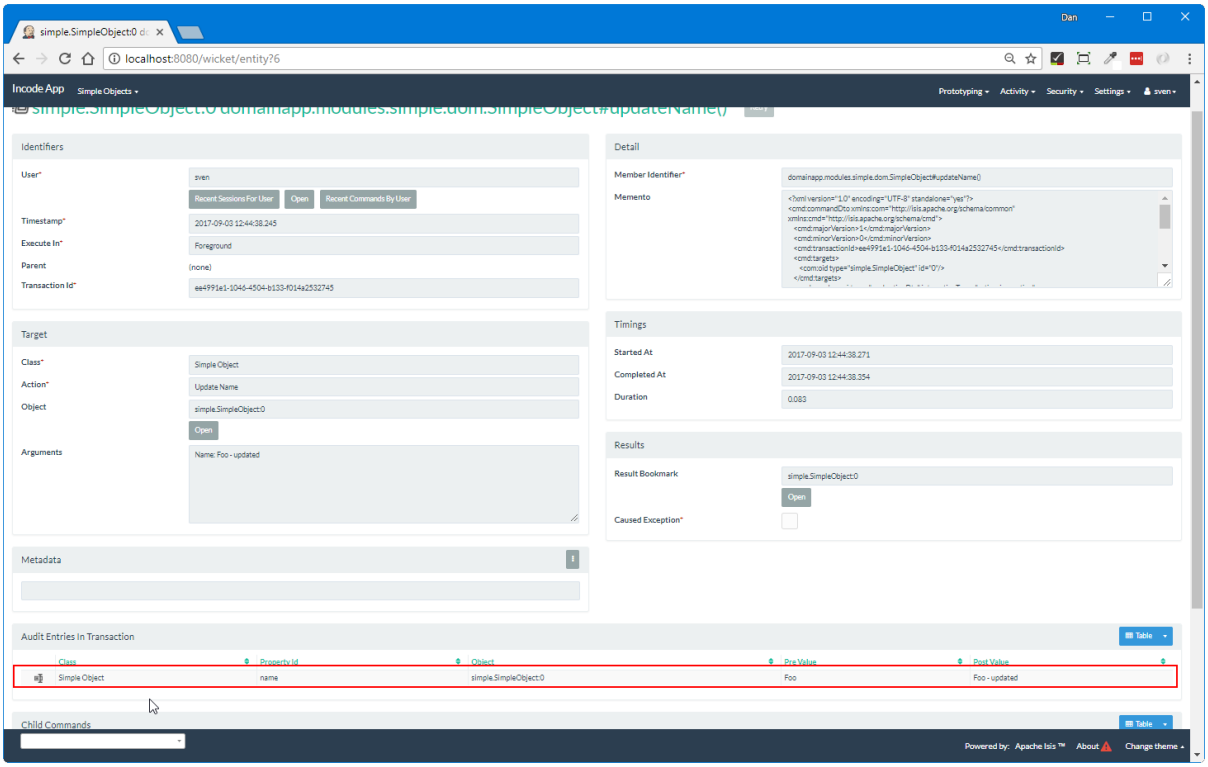
DomainAppAppManifestWithFixtures

```
protected void disableAuditingAndCommandAndPublishGlobally(final Map<String, String>
configurationProperties) {
    configurationProperties.put("isis.services.audit.objects","none");
    ...
}
```

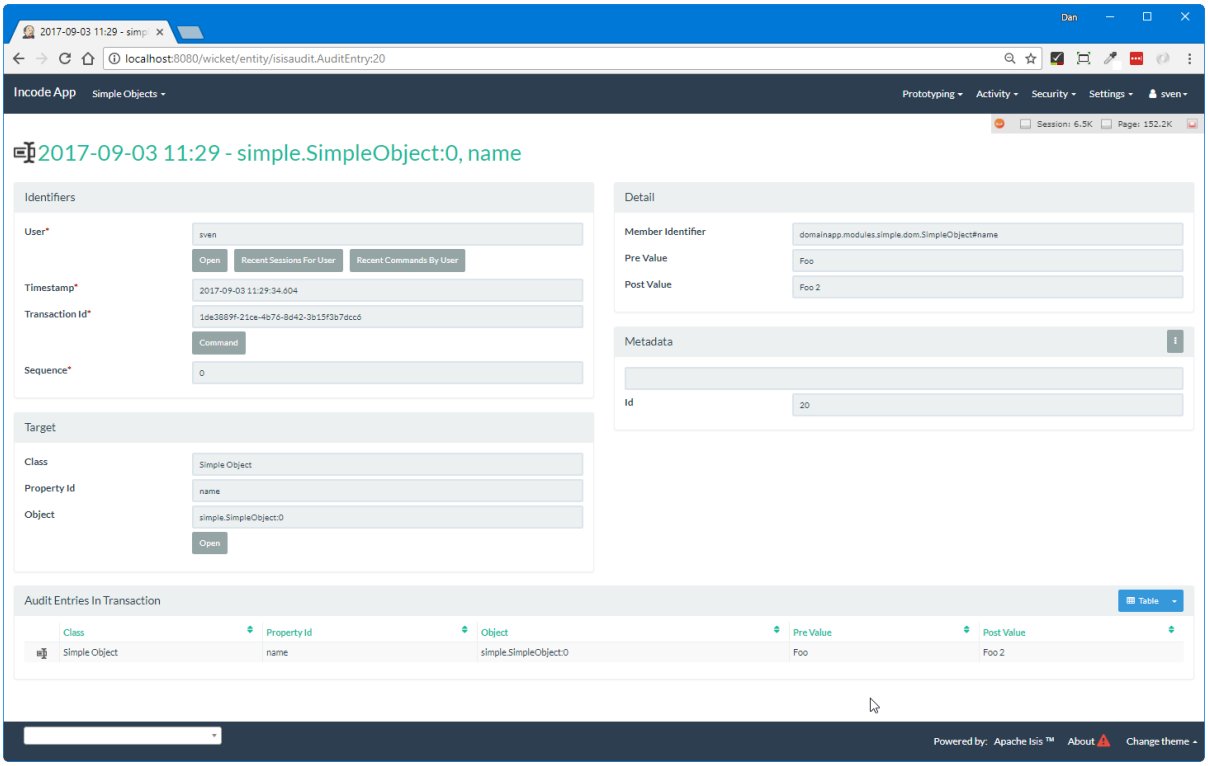
but the `SimpleObject` re-enables auditing using:

```
@DomainObject(
        auditing = Auditing.ENABLED
)
public class SimpleObject ...  { ... }
```

For example, invoking the `SimpleObject#updateName()` action (the example given above while discussing commands) persists a corresponding a `Command`, and which also shows the corresponding audit entries:



In this case there is just one `AuditEntry` object:



Publishing

Publishing and commands share some similarities: both create an XML representation of an action invocation or property edit. However, whereas a Command represents only the *intention* to invoke the action, with publishing the serialized XML also captures the result of the invocation and some basic metrics.

The main use case of publishing is to be able facilitate message exchange between external systems. The quickstart app uses the publishmq module which bootstraps an in-memory ActiveMQ instance and publishes to it.

In fact, the publishmq module allows the location of the ActiveMQ queue to be overridden. The quickstart app configures these properties, but leaves them as their defaults (that is, connecting to an in-memory ActiveMQ instance):

isis.properties (in webapp module)

```
isis.services.PublisherServiceUsingActiveMq.vmTransportUri=vm://broker
isis.services.PublisherServiceUsingActiveMq.memberInteractionsQueue=\
                                                    memberInteractionsQueue
```

The quickstart app (when using the fixtures) disables publishing "globally" in the app manifest:

DomainAppAppManifestWithFixtures (in the appdefn module)

```
protected void disableAuditingAndCommandAndPublishGlobally(final Map<String, String>
configurationProperties) {
    ...
    configurationProperties.put("isis.services.publish.objects","none");
    configurationProperties.put("isis.services.publish.actions","none");
    configurationProperties.put("isis.services.publish.properties","none");
}
```

but the `SimpleObject` domain object re-enables publishing for both editing of the `notes` property:

```
@Property(
        ...
        publishing = Publishing.ENABLED
)
private String notes;
```

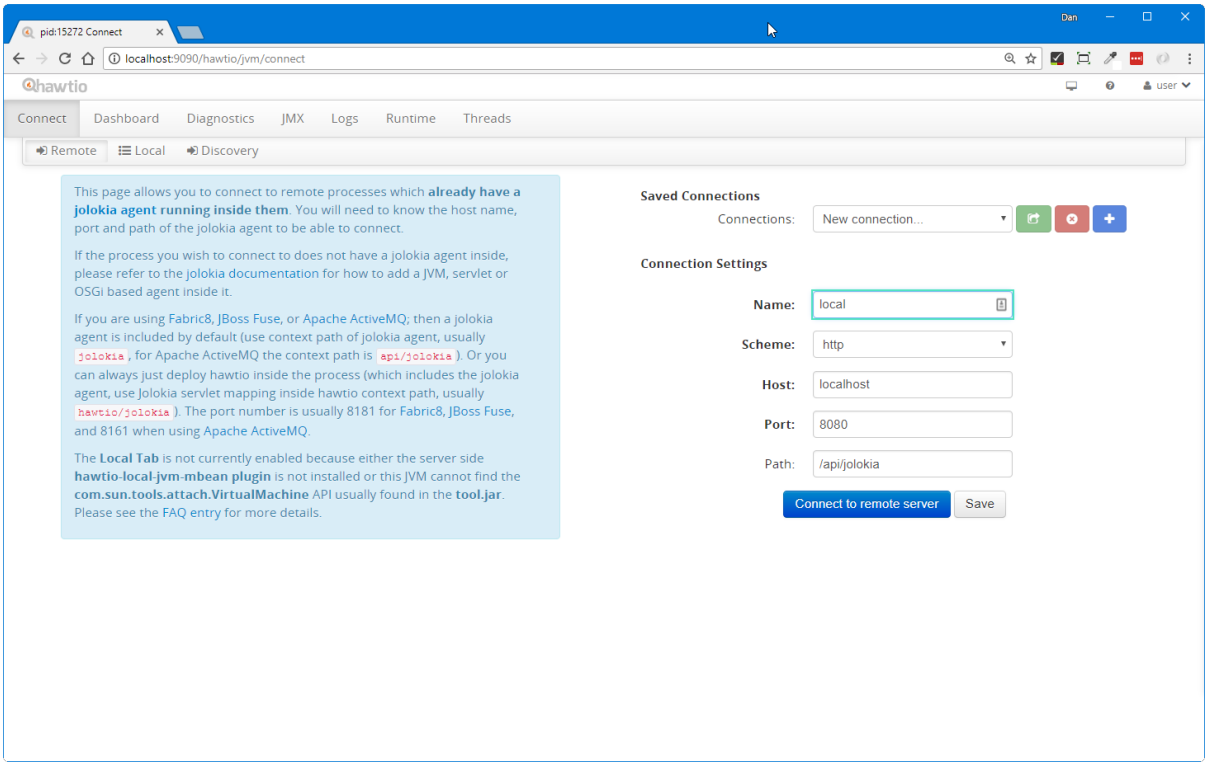
and for invoking the `updateName` action:

```
@Action(
        ...
        publishing = Publishing.ENABLED
)
public SimpleObject updateName( ... ) {  ... }
```

Using the hawt.io console we can monitor the messages sent to the ActiveMQ message. Download the hawt.io JAR file and start using:

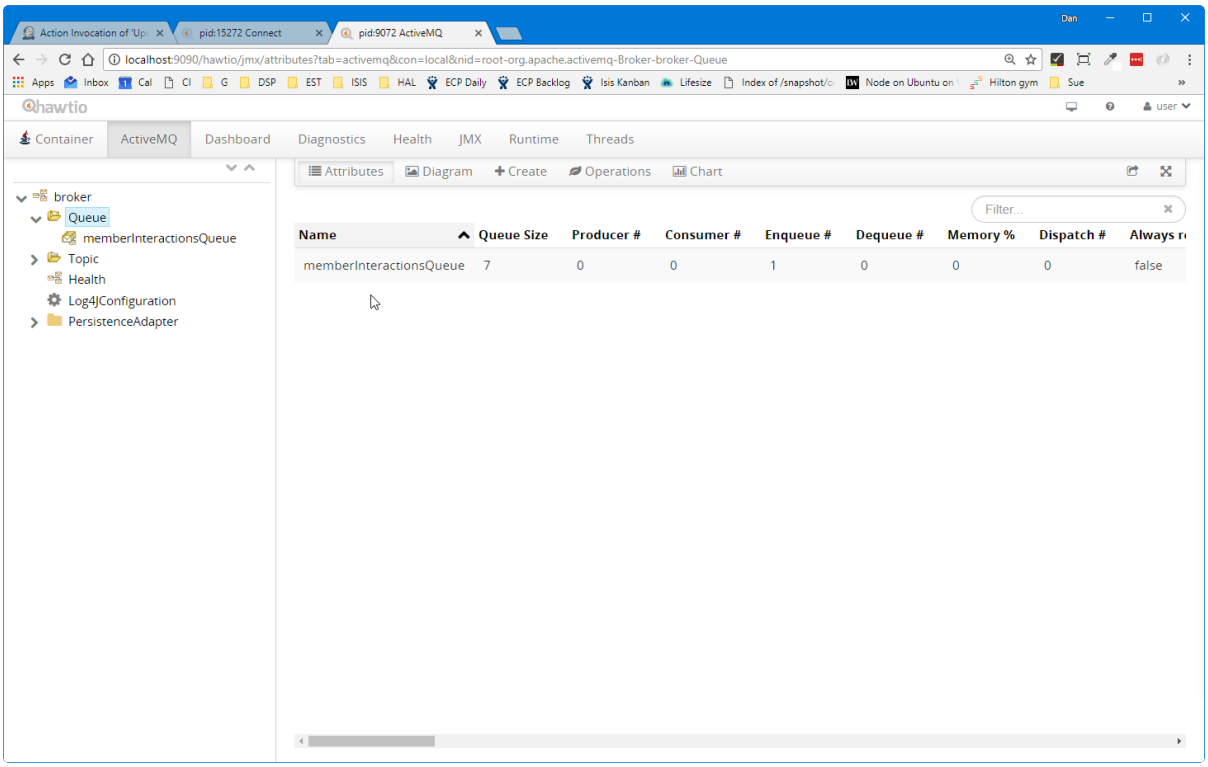
```
java -jar hawtio-app-1.5.3.jar --port 9090
```

Then connect to the jolokia servlet (configured in the quickstart's `web.xml`):

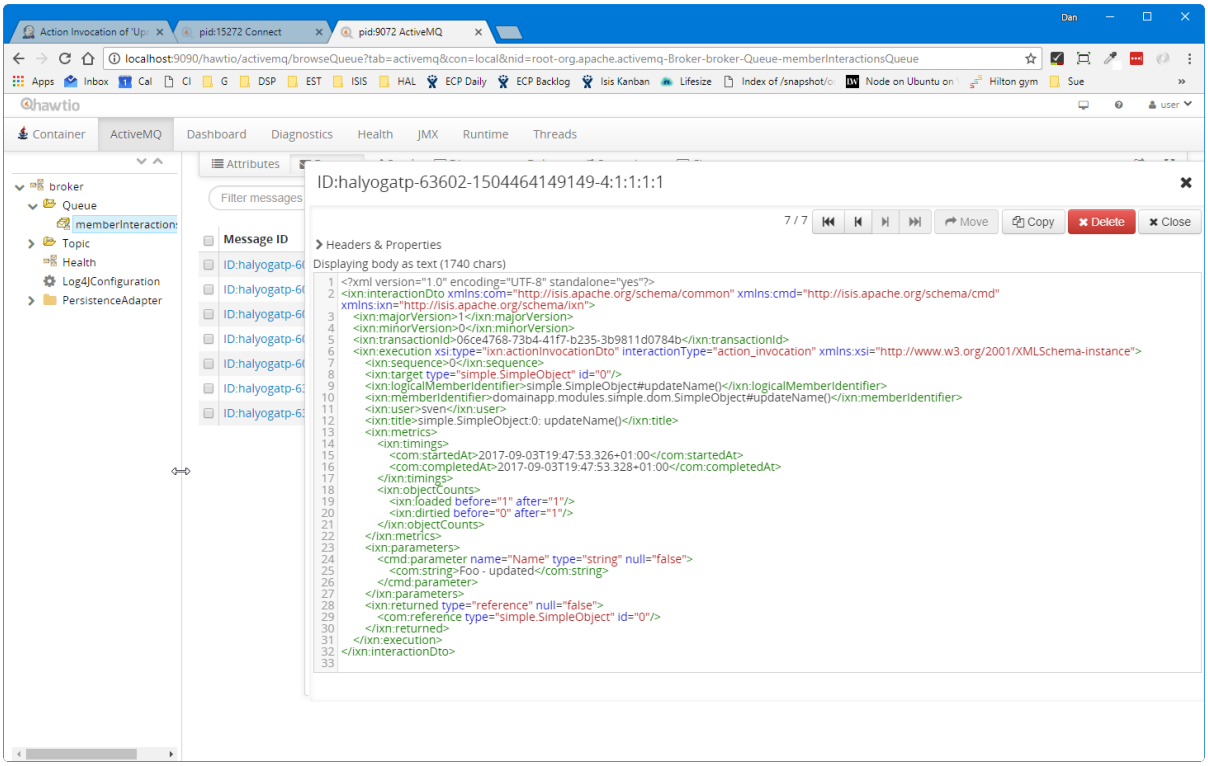


with the port set to `8080` and the path set to `/api/jolokia`.

Invoking the `updateName` action on `SimpleObject` will result in a message being sent to the ActiveMQ queue:

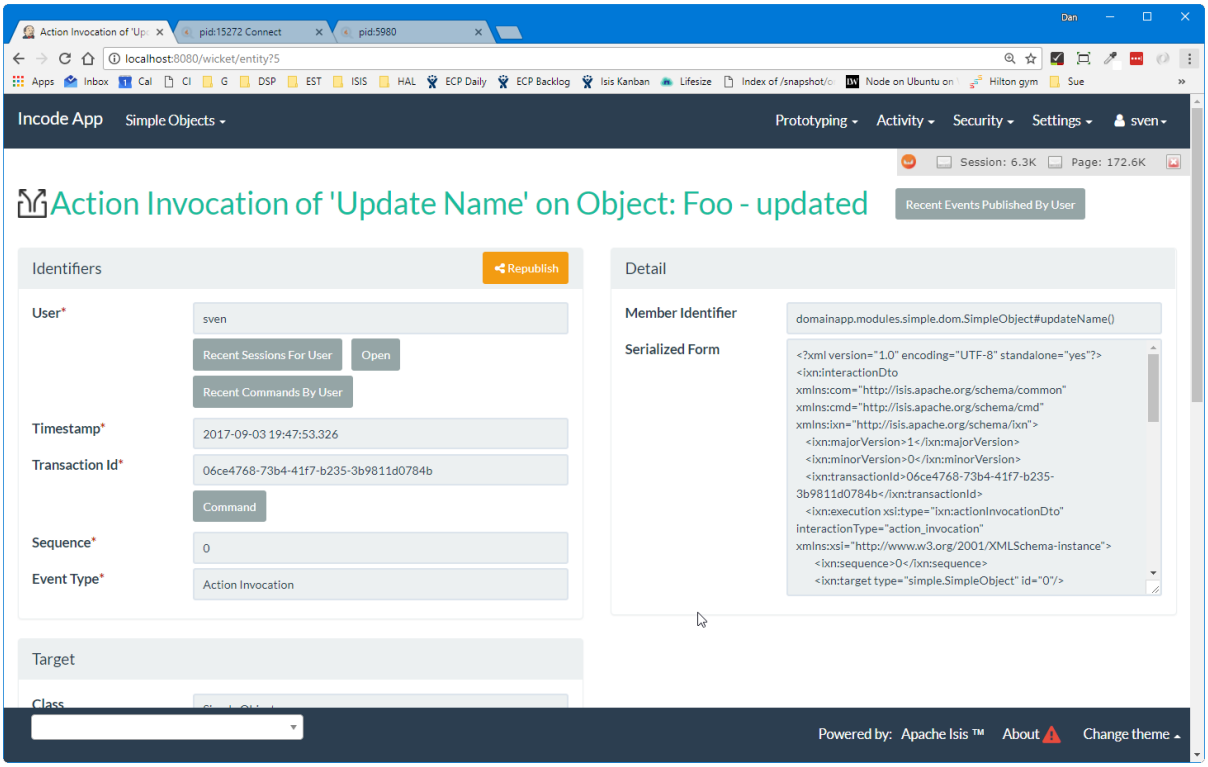


and indeed the details of the message can be inspected:



Note that these are only visible via hawt.io because there is nothing consuming the messages. For that, you need to configure a consumer, The embedded camel extended example does precisely this.

The publishmq module also persists all interactions to a database table; this is to allow messages to be resent if required. The message can be seen persisted as a `PublishedEvent` entity (searchable from the *Activity* menu):



Togglz

Feature toggles are a technique with various use cases, one of which is to allow functionality to be rolled out partially, eg for feedback. The togglz module provides support for this (leveraging the settings module for feature persistence.

The quickstart integrates this module, and demonstrates its usage in the `SimpleObjectMenu`:

SimpleObjectMenu (in module-simple)

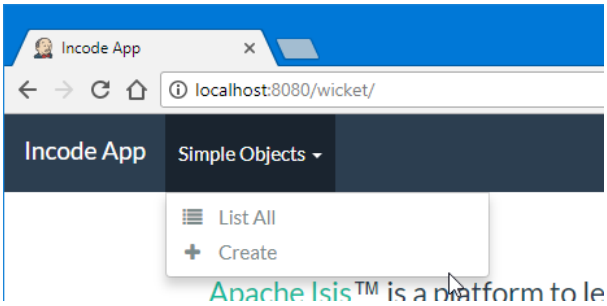
```
public class SimpleObjectMenu {

    public List<SimpleObject> listAll() { ... }

    public List<SimpleObject> findByName( ... ) { ... }
    public boolean hideFindByName() {
        return ! TogglzFeature.findByName.isActive();
    }

    public SimpleObject create( ... ) { ... }
    public boolean hideCreate() {
        return ! TogglzFeature.SimpleObject_create.isActive();
    }

    ...
}
```

where `TogglzFeature` is this enum:

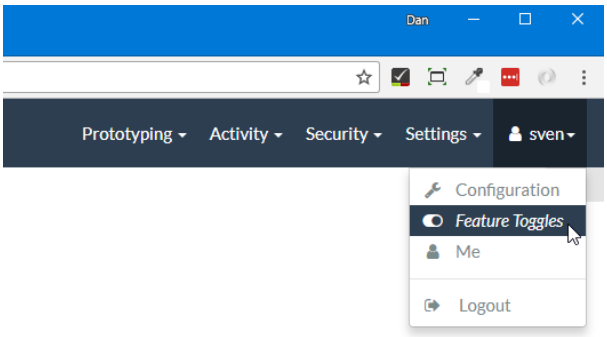
TogglzFeature (in module-base)

```
public enum TogglzFeature implements org.togglz.core.Feature {

    @Label("Enable SimpleObject#create")
    @EnabledByDefault
    SimpleObject_create,

    @Label("Enable SimpleObject#findByName")
    SimpleObject_findByName;

    public boolean isActive() {
        return FeatureContext.getFeatureManager().isActive(this);
    }
}
```

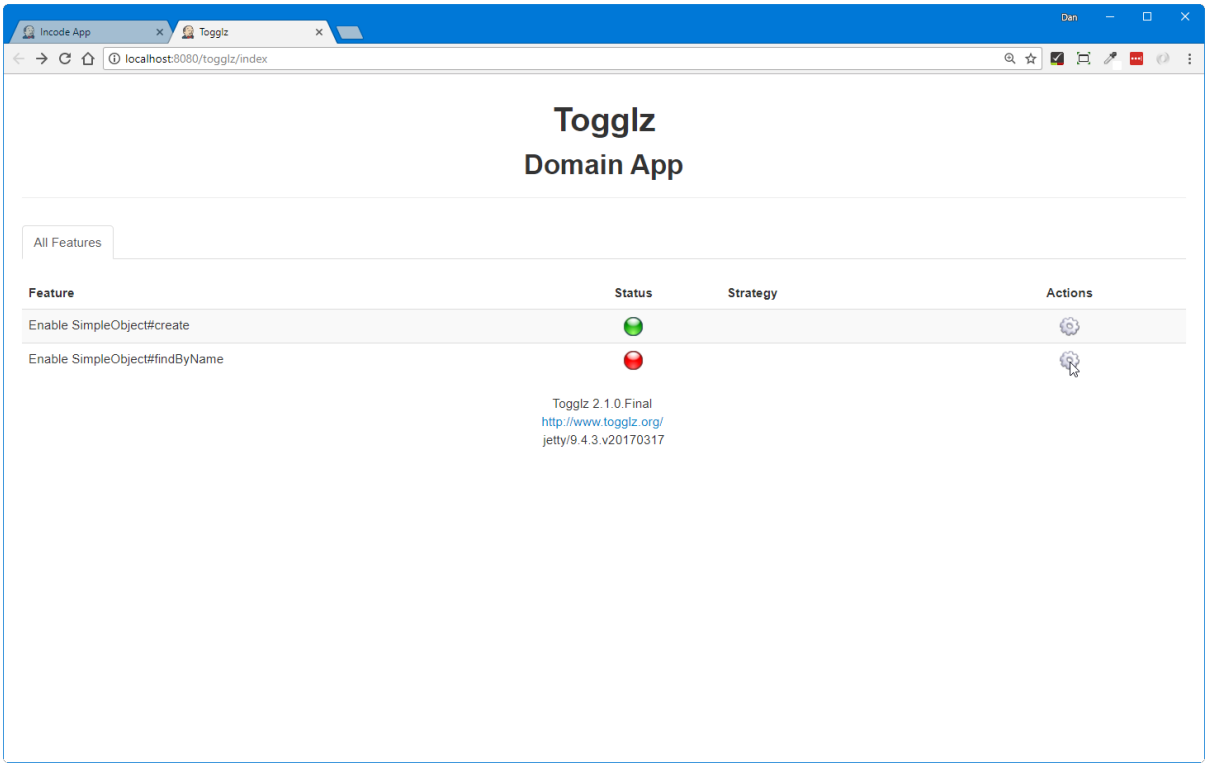
Because the `SimpleObject_findByName` feature toggle is disabled, the corresponding action is hidden:



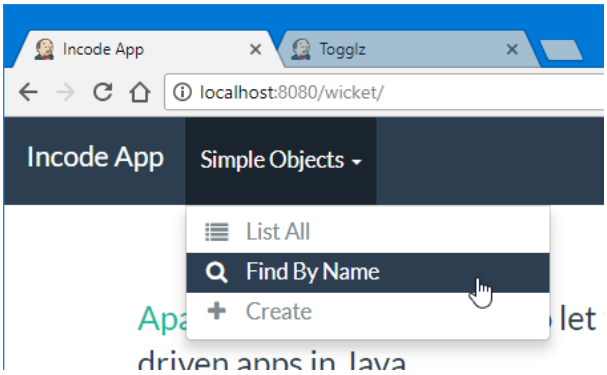
Users with the `isis-module-togglz-admin` role can change feature toggles dynamically:



which brings up the togglz console:



With the feature enabled, the "findByName" action become visible:



Quartz

The quartz extension module provides an in-memory scheduler for running jobs according to a schedule. The quickstart app uses this to schedule background commands every 10 seconds. Its configuration could of course be extended to perform other tasks.

The configuration is contained in `quartz-config.xml`:

quartz-config.xml (in the webapp module)

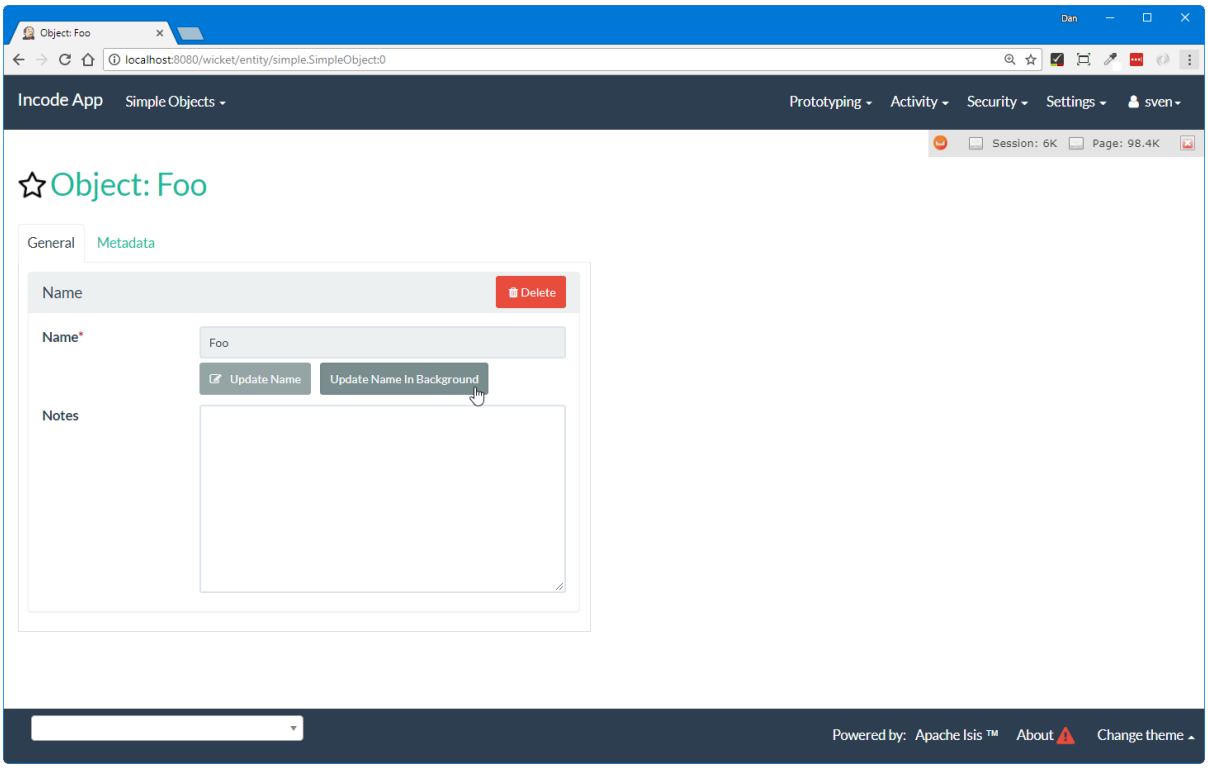
```
<?xml version="1.0" encoding="UTF-8"?>
<job-scheduling-data xmlns="http://www.quartz-scheduler.org/xml/JobSchedulingData"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://www.quartz-scheduler.org/xml/JobSchedulingData
http://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd" version="1.8">

    <schedule>

        <job>
            <name>RunBackgroundJobs</name>
            <group>DomainApp</group>
            <description>Runs all background jobs</description>
            <job-class>
org.isisaddons.module.quartz.dom.jobs.RunBackgroundCommandsJob</job-class>
            <job-data-map>
                <entry>
                    <key>user</key>
                    <value>scheduler_user</value>
                </entry>
                <entry>
                    <key>roles</key>
                    <value>admin_role</value>
                </entry>
            </job-data-map>
        </job>

        <trigger>
            <cron>
                <name>RunBackgroundJobsEvery10Seconds</name>
                <job-name>RunBackgroundJobs</job-name>
                <job-group>DomainApp</job-group>
                <cron-expression>0/10 * * * * ?</cron-expression>
            </cron>
        </trigger>

    </schedule>
</job-scheduling-data>
```

where `RunBackgroundCommandsJob` is provided by the quartz module.

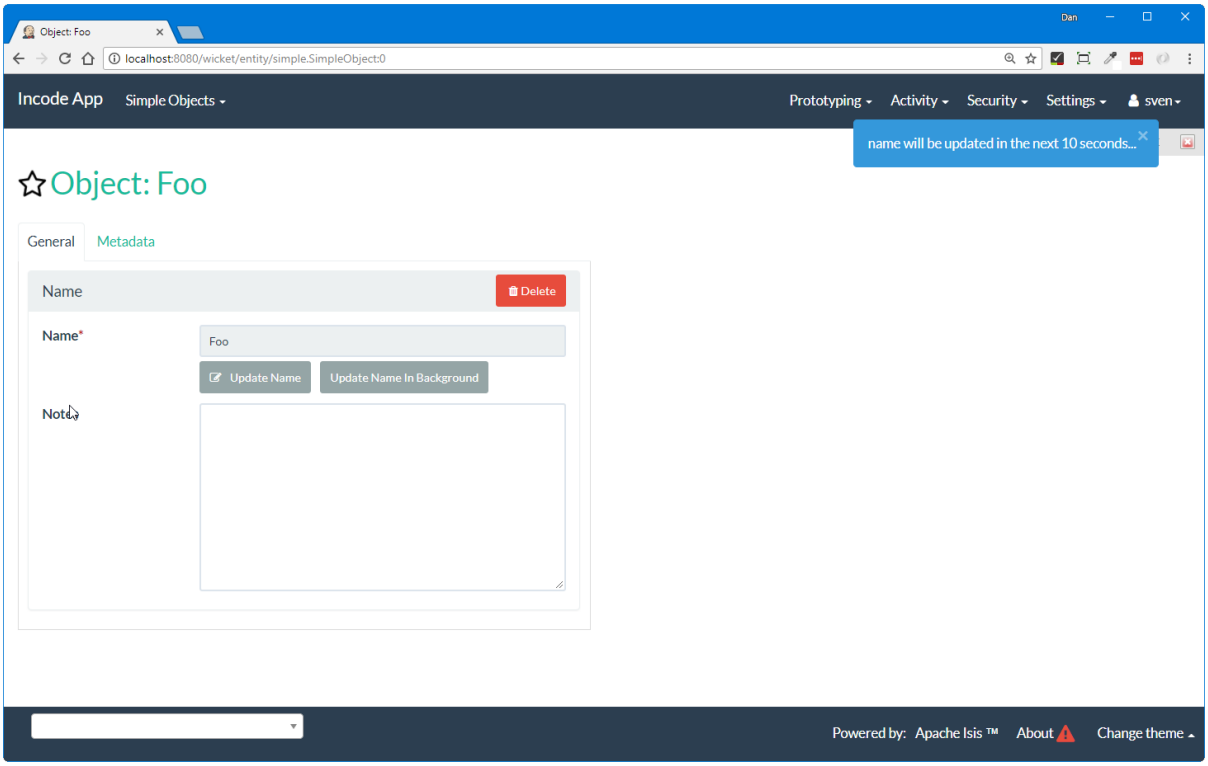
To see this in use, add the follow mixin:

```
@Mixin(method="act")
public class SimpleObject_updateNameInBackground {
    private final SimpleObject simpleObject;
    public SimpleObject_updateNameInBackground(final SimpleObject simpleObject) {
        this.simpleObject = simpleObject;
    }

    @MemberOrder(name = "name", sequence = "3")
    public SimpleObject act(final String name) {
        messageService.informUser("name will be updated in the next 10 seconds...");
        backgroundService2.execute(simpleObject).updateName(name);
        return simpleObject;
    }

    public String default0Act() {
        return simpleObject.default0UpdateName();
    }

    @Inject
    MessageService messageService;

    @Inject
    BackgroundService2 backgroundService2;
}
```

which provides a new "updateNameInBackground" action:

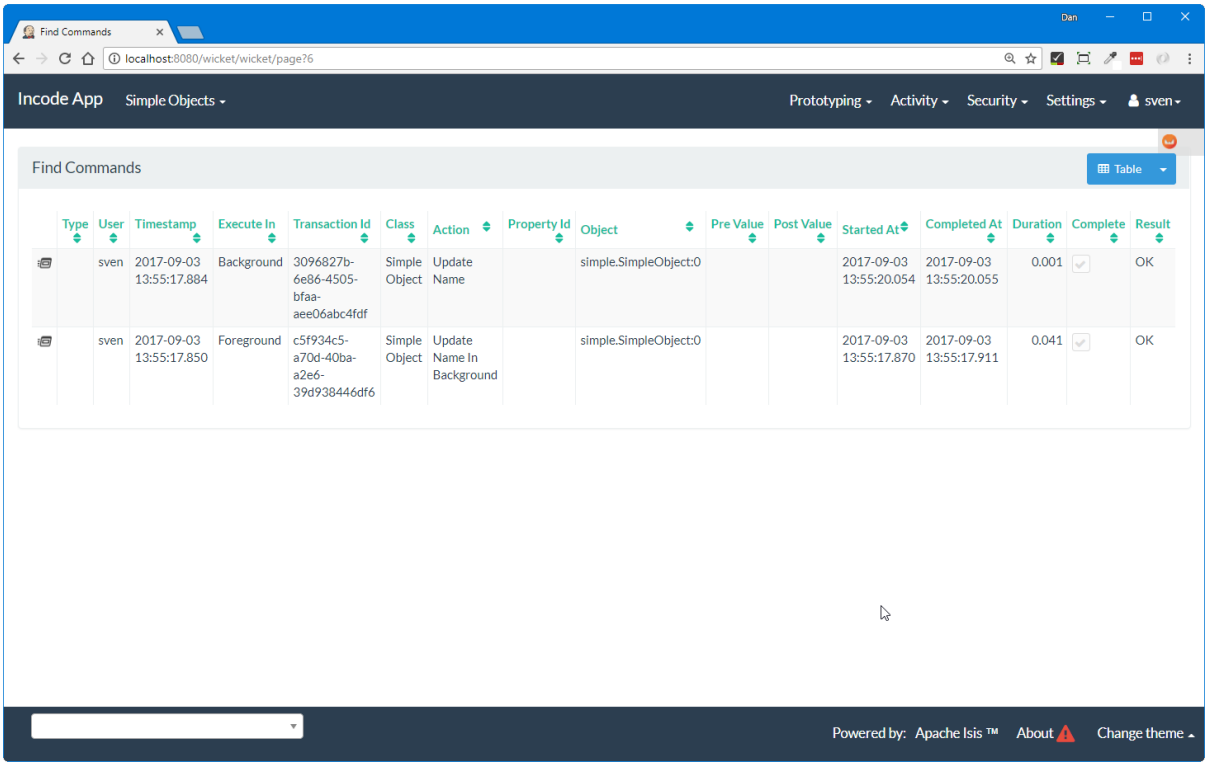


which when invoked does not immediately change the object's name but instead displays a message:



Keep clicking on the object's title to reload; within 10 seconds the name should change.

Listing all commands shows how the "updateNameInBackground" action is executed immediately (in the foreground), and as a side-effect creates a command for "updateName", executed in the background by the Quartz scheduler:



FlywayDB

The flywaydb extension integrates FlywayDB to automatically handle database migrations. These ensure that the schema of the (production) database is in sync with that required by the domain

entities.

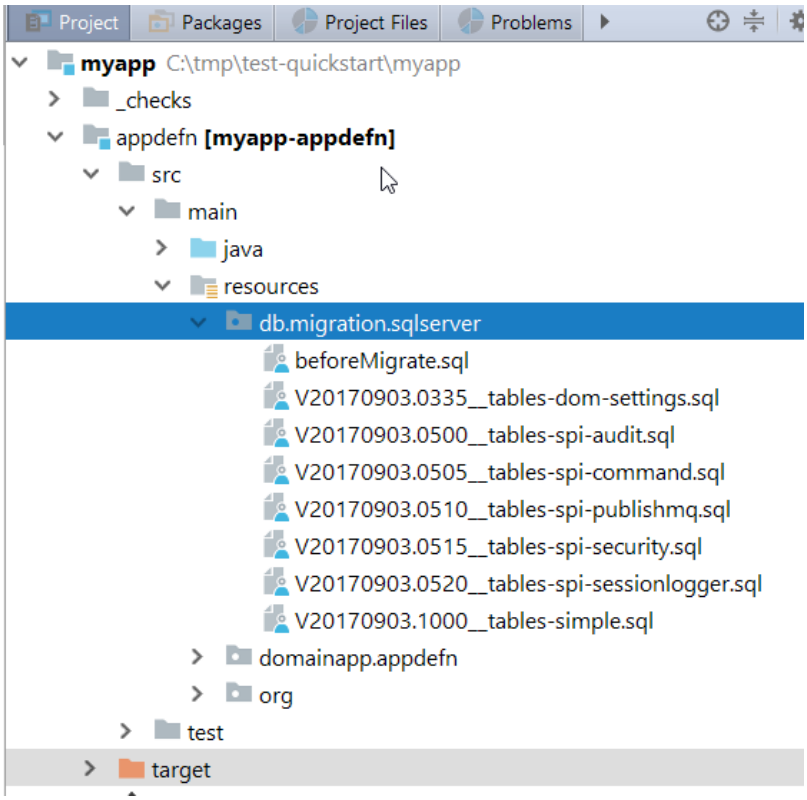
To demonstrate its usage, the quickstart app includes the `DomainAppAppManifestWithFlywayEnabledForSqlServer` manifest. This adds a number of configuration properties, the most important of which is:

```
configurationProperties.put(
    "isis.persistor.datanucleus.impl.datanucleus.schema.autoCreateAll", "false");
```

This tells DataNucleus to *not* automatically create the required database schema, so that it is instead created "manually" by Flyway running the provided SQL scripts.

There are a number of other configuration properties also set in that manifest; these indicate which schemas FlywayDB should track, and where to locate the migration scripts on the classpath.

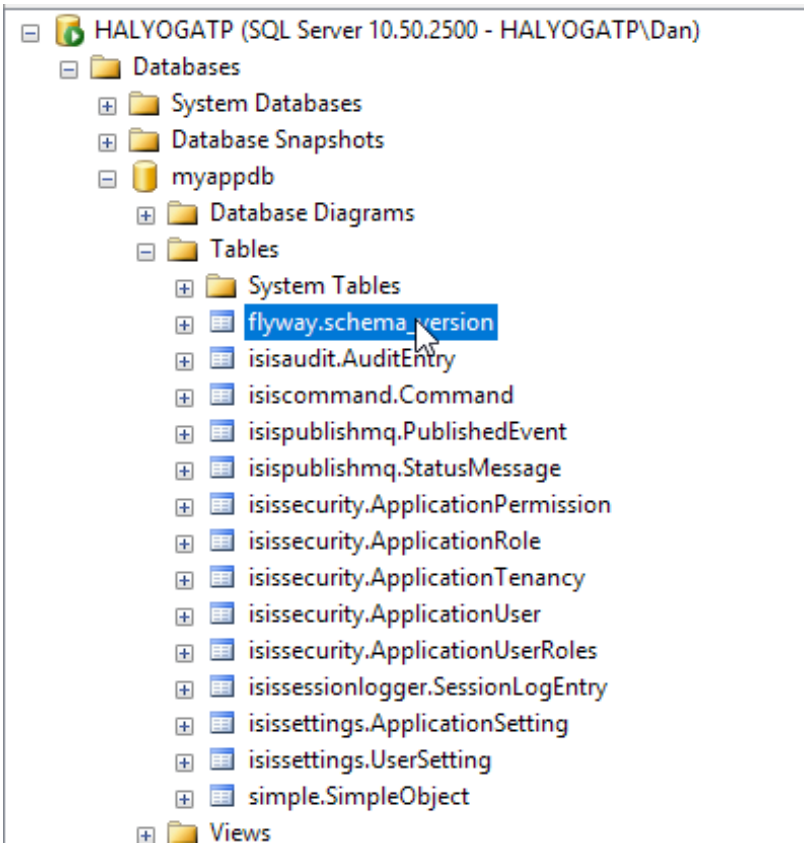
The migration scripts themselves are in the `appdefn` module, in a subpackage of `db.migration`:



Before running the app, it may be necessary to manually create the schema for FlywayDB's own `schema_version` table:

```
CREATE TABLE flyway
go
```

With this done, the app can be run against an (otherwise) empty SQL Server database. This results in FlywayDB automatically creating the database tables:



The `flyway.schema_version` table keeps track of the scripts that have been applied:

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [installed_rank]
      ,[version]
      ,[description]
      ,[type]
      ,[script]
      ,[checksum]
      ,[installed_by]
      ,[installed_on]
      ,[execution_time]
      ,[success]
  FROM [myappdb].[flyway].[schema_version]
```

Results | Messages

	installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	20170903.0335	tables-dom-settings	SQL	V20170903.0335__tables-dom-settings.sql	485211916	myappdbo	2017-09-18 20:16:32.760	12	1
2	2	20170903.0500	tables-spi-audit	SQL	V20170903.0500__tables-spi-audit.sql	-116202447	myappdbo	2017-09-18 20:16:32.810	29	1
3	3	20170903.0505	tables-spi-command	SQL	V20170903.0505__tables-spi-command.sql	1708688119	myappdbo	2017-09-18 20:16:32.857	24	1
4	4	20170903.0510	tables-spi-publishmq	SQL	V20170903.0510__tables-spi-publishmq.sql	1128635301	myappdbo	2017-09-18 20:16:32.897	17	1
5	5	20170903.0515	tables-spi-security	SQL	V20170903.0515__tables-spi-security.sql	274295441	myappdbo	2017-09-18 20:16:32.957	48	1
6	6	20170903.0520	tables-spi-sessionlogger	SQL	V20170903.0520__tables-spi-sessionlogger.sql	1339595970	myappdbo	2017-09-18 20:16:32.987	13	1
7	7	20170903.1000	tables-simple	SQL	V20170903.1000__tables-simple.sql	-29412561	myappdbo	2017-09-18 20:16:33.013	9	1

There are a couple of other points worth making.

- First, the manifest also uses the following configuration property:

  ```
  configurationProperties.put(
      "isis.persistor.datanucleus.impl.datanucleus.Mapping", "sqlserver");
  ```

  to instruct DataNucleus to pick up any `-sqlserver.orm` files from the classpath. There are two such: `DocumentAbstract-sqlserver.orm` and `CommandJdo-sqlserver.orm`.

- Second, the quickstart app also provides a `beforeMigrate.sql` script to drop all index/constraints, and uses the `autoCreateConstraints` property so that DataNucleus will automatically re-create any indices afterwards.

  For large databases this would not be practicable, in which case the `beforeMigrate.sql` script should be removed. Any changes to indices would therefore need to be provided in migration SQL scripts.

Dynamic Reloading

The `module-simple` module includes `layout.gradle` and `liveReload.gradle` scripts, and the Wicket viewer is configured for live reloading:

viewer_wicket.properties (in appdefn module)

```
isis.viewer.wicket.liveReloadUrl=http://localhost:35729/livereload.js?snipver=1
```

The procedure described in the Apache Isis documentation explains that the two scripts should be run together:

```
gradle -t --offline -b layouts.gradle &
gradle -t --offline -b liveReload.gradle &
```

Maven Mixins

To minimize boilerplate, the Maven `pom.xml` files use the `com.github.odavid.maven.plugins:mixin-maven-plugin`. This allows the `<build>` definitions of other `pom.xml` files to be "mixed in to" (that is, included in) the consuming `pom.xml`.

The maven mixins themselves used by the generated application are listed in the table below:

`groupId:artifactId` **github repo**	**Description**
`com.danhaywood.mavenmixin:cucumberreporting` java-mavenmixin-cucumberreporting	Configures the `net.masterthought:maven-cucumber-reporting` plugin, to generate HTML reports based on outputs of BDD specification tests.
`com.danhaywood.mavenmixin:datanucleusenhance` java-mavenmixin-datanucleusenhance	Configures the `org.datanucleus:datanucleus-maven-plugin` to post-process (enhance) persistent entities according to the JDO spec.
`com.danhaywood.mavenmixin:docker` java-mavenmixin-docker	Configures the `com.spotify:docker-maven-plugin` plugin to create Docker images and to upload these to a specified registry.
`com.danhaywood.mavenmixin:enforcerrelaxed` java-mavenmixin-enforcerrelaxed	Configures the `maven-enforcer-plugin` plugin with a number of pre-defined rules (though *not* dependency convergence checking).
`com.danhaywood.mavenmixin:jettyconsole` java-mavenmixin-jettyconsole	Configures the `org.simplericity.jettyconsole:jetty-console-maven-plugin` to create a console app (with optional Swing UI) to bootstrap the application from the command line using an embedded Jetty instance.
`com.danhaywood.mavenmixin:jettywar` java-mavenmixin-jettywar	Configures the `maven-war-plugin` to build a war (webapp archive), and the ` `org.eclipse.jetty:jetty-maven-plugin` to be able to run this from maven (using `mvn jetty:war`).
`com.danhaywood.mavenmixin:sourceandjavadoc` java-mavenmixin-sourceandjavadoc	Configures the `maven-javadoc-plugin` plugin to create Javadoc website and the `maven-jxr-plugin` to create a similar website of the source code (cross-referencing the Javadoc).
`com.danhaywood.mavenmixin:standard` java-mavenmixin-standard	Configures the standard `maven-clean-plugin`, `maven-resources-plugin` `maven-compiler-plugin`, maven-jar-plugin`, `maven-install-plugin`, `maven-deploy-plugin` and maven-site-plugin` plugins (mostly just setting their version).
`com.danhaywood.mavenmixin:staticanalysis` java-mavenmixin-staticanalysis	Configures the `maven-checkstyle-plugin`, maven-pmd-plugin`, `javancss-maven-plugin` and `jdepend-maven-plugin` plugins. The configuration files driving these plugins are specified as properties.
`com.danhaywood.mavenmixin:surefire` java-mavenmixin-surefire	Configures the `maven-surefire-plugin` with multiple executions to run unit tests, integration tests and BDD specifications. testing support is discussed further below.