NHM IDE - how it works

richardt

October 12, 2016

Contents

This is the repository for the NHM IDE. It contains the code to make the application work, but not the code for the model or the API between the application and the model.

It's built using tycho, which is a set of maven plugins for building Eclipse applications. To explain what and why this is a thing, here is a quick rundown of some key Eclipse concepts and words;

- Eclipse / the Eclipse Rich Content Platform is a framework for making desktop applications
- It's made of various layers, like a lot of things
- In the middle is "Equinox" which is an OSGi runtime
- The basic unit for eclipse is the OSGi bundle (bundle). A bundle is just a jar file, but it has extra stuff in its META-INF/MANIFEST.MF file which does a few things. This extra stuff is:
  - A name and version number
  - A list of "exported" packages and their version numbers; an exported package contains types that are visible to other bundles in the runtime
  - Dependency specifications in the form of:
    - Require-Bundle: headers - these are an old-fashioned way to say that one bundle needs another bundle to work; they refer to the name and version (or range) of that bundle. When the runtime tries to install and load up the bundle, it will make sure it can find the dependency and it will stick its exported packages onto the classpath
    - Import-Package: headers - these are the newer and better way to specify dependencies; they give a package name and version (or range), and the runtime will make sure that an implementation of that package at that version is on the classpath. The difference here is that more than one bundle can provide a particular package.
  - Stuff to do with providing services to the platform, in the form either of a Bundle-Activator: header which names a class that will get instantiated and have some methods called on it by the framework when the bundle is loaded, or
  - OSGi declarative services stuff, which has some manifest headers that refer to some XML files usually in OSGI-INF/something.xml; these tell the framework to new up some class AND publish it to the platform as a service.

- As mentioned, OSGi bundles provide "services" to each other
- A service is just an implementation of some interface
- A bundle can ask the runtime to provide it with implementations of some service
- Bundles can also supply classes that provide some services
  - for example, you could publish HTTP servlets to the runtime under the servlet interface, and a web-serving part could consume those objects and wire them up to HTTP request handlers
- One more thing; in Eclipse, some bundles are also "plugins". A plugin is a bundle which has a "plugin.xml" file in it as well. This plugin.xml contains (suprisingly enough) some XML, which is used to wire stuff up to the eclipse platform's user interface in some declarative, lazy-loaded way. The word to google here is "extension point"; for example to stick a button in a toolbar from your plugin, you contribute to the relevant extension point from the plugin.xml. The XML says what toolbar to put the button in, what icon and text to give it, and what class to instantiate when the button is actually used. This is how most bits of Eclipse's UI are connected up

1 Key parts

There are several maven modules in this repository; each one builds part of the application

1.1 cse.nhm.ide.build

This is what brings all the stuff together into what is called a *product*; this is an installable zip file with a launcher icon and all the dependencies. Inside here are a very few files:

1. NHM.product is an eclipse product definition file; it is kind of a config file which says where the main method is and what features should go into the build (a feature is a collection of plugins or bundles)
2. plugin$_{\text{customization}}$.ini is a properties file which overrides default settings in the application when it's started up
3. the ubiquitous pom.xml, which just tells tycho to build an app

1.2 cse.nhm.ide.feature

This also just has a few parts, mainly the `feature.xml` file. This is an eclipse feature definition, which defines a group of plugins that all go together and do something. In this case our feature contains all the NHM specific plugins plus the eclipse plugins that we depend on to make it all work.

1.3 cse.nhm.ide.product

This is a plugin, but it only has one interesting thing, which is an extension point defining a product - for some reason this doesn't go in `.build`, but `.build` does refer to the product extension which is defined in there. You probably won't need to touch it

1.4 cse.nhm.ide.ui

This plugin is used by most of the other NHM related plugins. It defines what is called the "NHM Project Nature"; in eclipse a project may have several "natures". A nature adds some kind of behaviour to the project. The NHM nature adds:

- The project-specific preference about NHM versions
- A project "builder", in NHMBuilder; this is the code which runs the validation when you save. Builders are triggered when some files in the project are modified.

The main useful stuff in this is provided by NHMNature; other plugins which want to use the NHM for a project will typically use NHMNature.of(IProject) to get an instanceof NHMNature for some project, and then .getModel() on that to get an INationalHouseholdModel. This will be an instance of ServiceTrackingModel, which does the work of finding the right version from the many versions installed, and switching versions in response to preference changes on the project.

1.5 cse.nhm.ide.runner.api, .local, .remote and .ui

These guys actually do something, and are an example of how OSGi services often work; the `.api` bundle contains interface definitions for things that run scenarios. `.local` and `.remote` publish implementations of those services, and `.ui` consumes those implementations and wires up user-interface stuff to make a view, buttons for triggering a run, and so on.

The broad structure is:

- runner.api
  - Interfaces
    - IScenarioRunner represents a thing which runs jobs and has a list of jobs, each job being an
    - IScenarioRun, which provides the methods to cancel / get results etc.
  - Utility classes
    - ScenarioRunner which is an abstract base class for IScenarioRunner - it is where the basic stuff to do with doing a run happens, like collecting up the inputs and hashing it all down to an ID. Base classes implement doSubmit, and invoke the updated() method to keep the state current.
- runner.local Extends ScenarioRunner to LocalScenarioRunner and implements LocalScenarioRun; these classes do scenario running by storing the run metadata and stocks in the local filesystem in a plugin-specific metadata folder in the workspace. There is a single execution thread which goes through jobs and runs them one at a time.
- runner.remote Also extends ScenarioRunner to give RemoteRunner and RemoteRun; it also adds a preference page extension to let you manage a list of servers. Each server defined in the preferences produces a RemoteRunner instance which is provided as a service to the rest of the runtime. These talk HTTP using the basic HttpUrlConnection; all the code for that is in the RemoteRunner, so that defines the API used really.
- runner.ui This is the consumer for runner.remote and runner.local, and has a few things in it:
  - Commands - a command is what backs buttons, menus etc in the eclipse UI. Each command may have one or more handlers, where a handler is the class that does the work. Different handlers are activated in different situations; mostly this is handled by some stuff in plugin.xml. One of the commands does actual interesting work, namely GetResultsCommandHandler - this is where batch outputs are stitched together.
  - Views - this is where the list of jobs and so on lives. There is an interesting thing here which is the ScenarioRunnerTracker; this is

the connection to OSGi which gets told when you add or remove an IScenarioRunner service from the runtime.

1.6 cse.nhm.ide.ui.editor

Contains the scenario editor, and is responsible for:

- hover help
- suggestions (control space)
- syntax highlighting
- snippets / templates in eclipse speak
- scenario editor bracket and motion commands
- jump-to-thing (control shift T) and outline (control o) in scenarios

1.7 cse.nhm.ide.stock.ui

Contains the stock browser and the stock import wizard.

1.8 cse.nhm.ide.ui.results

Contains the tab file viewer

1.9 cse.nhm.models.feature

Contains the single file `feature.xml`; this needs to be updated when we do an nhm model release; the stanzas in the <requires> block are what cause the different model versions to get included into the build. Each bundle impl has a version like x.y.z.R1234566; the dependencies for those are specified with match="equivalent", which instructs P2 to find the highest version whose first two version digits match. For example, if the repository contains

- 6.4.1.D123456
- 6.3.0.R123465
- 6.3.1.R567675
- 6.3.1.R787878
- 6.3.1.D999999

6.3.0 equivalent will pick 6.3.1.R787878. 6.4.0 equivalent will pick 6.4.1.D123456. Each part of the version is compared numerically except the qualifier (the last bit), which is compared lexically.

1.10 cse.nhm.jre.win64.feature

Contains a copy of the JRE, which is built into the windows version.

2 Running in eclipse vs building with tycho

Eclipse programs are built (or in eclipse speak, "provisioned") using a complicated nightmare called P2 (Provisioning Protocol I think, or something like that). A thing called the P2 director takes a set of P2 repositories and a definition of a thing to build (this is the .product file mentioned before), and solves the dependency problem to find a set of bundles that can make the product work.

A P2 repository is a bit like a maven repo, except it has an index that knows much more about the things in the repo. Specifically, it knows the MANIFEST.MF metadata for all the jars it contains, including the package and bundle dependencies & versions for everything. The P2 director reads in all this metadata for all the repositories which are being used, which comprises what is called the *target platform.* Then it does a big solving job to find a set of bundles which meets all the requirements; this is harder than it sounds.

Remember that a bundle depends on particular version ranges of packages, which may be exported by many other bundles, so it's not just doing some transitive resolution by name. P2 can solve all kinds of horrible diamond problems between your dependencies, which is nice, but unfortunately if it gets stuck you may have a bit of a problem. For P2 to be able to do its job, everyone has to spell out their dependencies correctly, and sometimes if someone is a bit overzealous the situation can get unsatisfiable. In this case it can be a matter of bodging the maven configuration to filter certain plugins from the target platform, so they can't jam everything up.

Anyway, the point is that you have a target platform which is full of plugins, and the build system's job is to assemble a set of plugins which let your product work properly.

2.1 Building with tycho

The build itself is simple: in the top level, mvn clean package

The application builds are stored in cse.nhm.ide.build/target/products/ as zip files; you can unzip one of these and run it directly.

Tycho knows the P2 repositories which define the target platform from the pom.xml; in our build, the target platform consists of:

- the submodules in the ide
- a few remote P2 repositories
  - eclipse mars, which contains the base eclipse stuff
  - eclipse orbit, which contains some other dependencies not in eclipse needed by the results view ("glazed lists")
  - eclipse nebula's NatTable, which contains the grid control used by the results plugin
  - the NHM p2 repository `http://p2.research.cse.org.uk/` (username tom.hinton@cse.org.uk password 'password')
- The POM dependencies which are OSGi bundles; this is used for one dependency (the JSON parser) which is not in any of the P2 repositories. Using maven repos for bundles is not really a good thing in general, as maven's dependency information is really limited.

2.2 Running in eclipse

First off, you need the right kind of eclipse; go to `http://www.eclipse.org/downloads` and get the "Eclipse for RCP and RAP developers".

Eclipse has quite good integration with Tycho these days, but it cannot yet understand the target platform from the pom in the same way, or run a product that's built by the pom directly. Instead, we need to set up a few things:

2.2.1 Target platform

First, eclipse needs to know the target platform; for this, eclipse uses .target files, which contain the same information as the pom does, i.e. a list of repositories that should contain stuff to be considered for the build.

The repository contains a target definition in the `cse.nhm.ide.target` folder; this isn't part of the maven build, and is only there to make it possible to run stuff from within eclipse. So, to make this work you need to:

1. Import the `cse.nhm.ide.target` project into eclipse (import existing maven projects)

2. Open the target platform file `flip.target` in that project - this gets you the target platform editor, which is pretty flaky I'm afraid
3. Hit the "reload" button on the right, which is usually a good idea - this will think for a while, as it downloads all the metadata and does some hard maths
4. If necessary, save the target file (hopefully not necessary)
5. Click "set as target platform" in the top right bit of the editor

Eclipse will have a think, and build errors should go away. If the target definition shows some errors, panic and wave your arms in the air. The first entry in the list (\${$\text{project}_{\text{loc}}$}/bundles/) sometimes doesn't work, you can sometimes fix that by editing it (Edit button) and making no change (Finish button) to give eclipse a kick.

Once your target platform is set, eclipse will be able to build everything, hopefully.

2.2.2 Run configuration from product

To run a thing in eclipse, you need a run configuration; the easiest way to get this sorted to start with is to open `NHM.product` (C-S-R NHM.product, RET), and in the "Overview" tab click "Launch an Eclipse application". You only need to do this once, after which you can forget the product file; generally you want to avoid fiddling with it as it defines the dependencies and multi-platform build stuff. Anyway, you should now get the run configuration window come up, and have a run configuration in the run dropdown called `nhm`. This will launch the NHM app.

If this has problems, you may need to make sure the run configuration has all the plugins it needs in it; this can end up wrong when a new build of the implementation plugin happens.

To do this, open the run configurations window, and look at the `nhm` run configuration (under Eclipse Application). You will see on that a "Plug-ins" tab. Mostly if something is not working, you can click "Add required plugins" and all will be well. Otherwise, you may need to manually fiddle with the auto-start column for a particular plugin.

2.3 Deployment & updates

There is a deploy job on the build server; this just uses scp to copy some files onto the deccnhm server, in a folder that's served up over http. It copies:

1. the product zip files, which you can get from `http://deccnhm.org.uk/standalone/`
2. the p2 repository for the product which is at `http://deccnhm.org.uk/p2/`

The second is what is used for updates.

3 External dependencies

3.1 API & impl bundles

All the actual nhm code is built into versions of the implementation bundle; this connects to the GUI by publishing an instance of INationalHouseholdModel, which is an interface defined in the API bundle. Both of these are built by gradle and published to the CSE internal p2 repository at `http://p2.research.cse.org.uk/`

3.2 Documentation

The nhm-documentation project contains a submodule which builds eclipse documentation; this is also published to the p2 repository

3.3 p2 repository

The aforementioned p2 repository is accessible using my top-secret credentials:

- tom.hinton@cse.org.uk / password

It's a program called package-drone, which seems to work quite well. It does one particularly clever thing for us, which is to automatically create a "feature" that helps the target platform file work in eclipse.