NetBeans Platform Tutorial

for

Metamodel tool

Version 2.0 approved

kupcimat

MMN group

14. 12. 2010

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
kupcimat	14.10.10		1.0
kupcimat	14.12.10	Added new parts	2.0

1 NetBeans Platform Guide

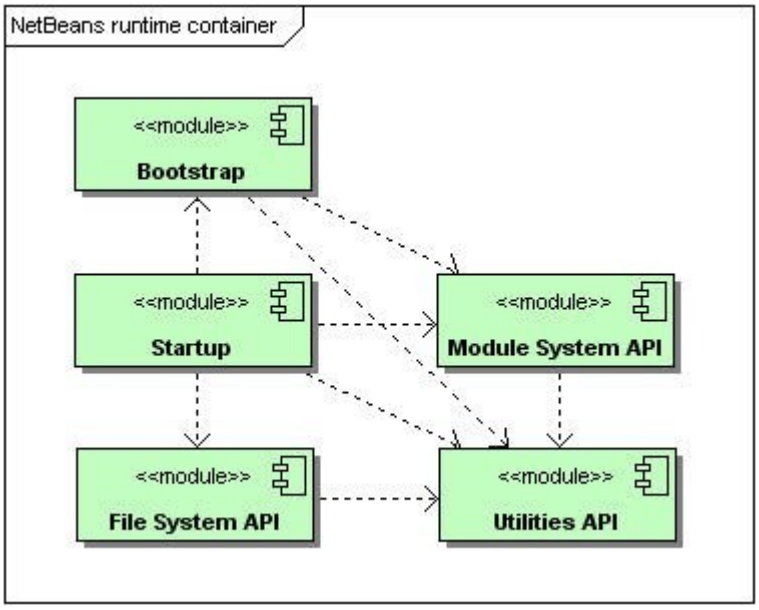
First of all this document is not intended as a complete tutorial for NetBeans Platform or a replacement for online NetBeans documentation. However it should provide you a quick start for understanding its key features and capabilities. For more comprehensive help go to netbeans.org.

2 What is NetBeans Platform

Netbeans.org has this to say on NetBeans Platform: “The NetBeans Platform is a broad Swing-based framework on which you can base large desktop applications”. NetBeans IDE itself is also built on this platform. It contains many useful APIs to manage window components, files, actions etc. Encourages developers to create applications with strict modular architecture. This approach enables to easily add new modules even at runtime. Following chapters will introduce and demonstrate the most common APIs.

3 Runtime Container

Before proceeding to APIs itself it is good to know the basic structure and essential modules of every NetBeans Platform application. The runtime container is the execution environment for the modules that define a NetBeans Platform application. It consists of 5 modules as shown below.



- ***Bootstrap*** – enables to recognize modules and how to load and compose them into application
- ***Startup*** – provides the main method for application and all necessary code for starting up
- ***Module System API*** – influences the lifecycle of the modules in application
- ***File System API*** – gives access to filesystem

- ***Utilities API*** – includes many utility classes, in older versions includes also “Lookup class”, from Platform version 6.9 it is in separate module – ***Lookup API***

4 XML Layers

Every module can define xml “layer” file which represents virtual files and directories that are merged into System Filesystem. It is a sort of virtual file filsystem where files are not on actual disk but in a xml file. In this way modules can add their configuration an data to the system. This mechanism is very helpful in modular applications because it simplifies dynamic configuration loading. Each module can have at most 1 xml layer file. This concept is used ie. in NetBeans main menu where each menu item is represented as special “file” in a folder structure.

4.1 Example layer.xml file

This is only a simple demonstration of System Filesystem which points to a single file on a disk. It has many more features and for complete reference you should go to netbeans.org.

First you need to add this line to the jar manifest

```
OpenIDE-Module-Layer: org/netbeans/modules/mymodule/layer.xml
```

Second you need to create layer.xml file at location mentioned earlier

```
<filesystem>
  <folder name="myFolder">
    <file name="myFile.txt" url="resources/TextFile.txt"/>
  </folder>
</filesystem>
```

- ***url*** – specifies the location of actual file, it is relative to the path of layer.xml file

Finally you can access specified file in code

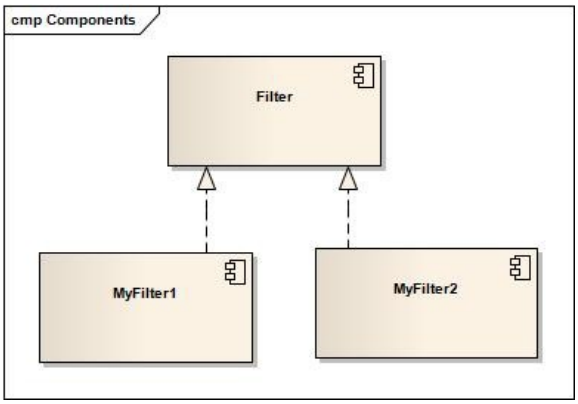
```
FileObject myFile =
FileUtil.getConfigFile("myFolder").getFileObject("myFile.txt");
InputStream in = myFile.getInputStream();
```

Of course NetBeans IDE can do all the necessary steps for you, but it is good to know the basics.

5 Lookup API

Lookup API is NetBeans' form of dependency injection. It is similar to JDK 6 ServiceLoader mechanism. In modular architecture like NetBeans Platform it is important to have strict contracts between isolated modules, which means to explicitly declare module dependencies and public packages.

5.1 Example of modular architecture using Lookup



- ***Filter*** – defines an interface, which is explicitly declared public (NetBeans → *Filter* → Project Properties → API Versioning → Public Packages)

```
public interface Filter {
     String process(String s);
}
```

- ***MyFilter1*** and ***MyFilter2*** – implement public interface from *Filter*, must declare explicit dependency on *Filter* module (NetBeans → *MyFilter1* → Project Properties → Libraries → Add Dependency → *Filter*)

```
@ServiceProvider(service = Filter.class)
public class MyFilter1 implements Filter {
     public String process(String s) {
          return s.toLowerCase();
     }
}
```

`@ServiceProvider` annotation at compile time will create a META-INF/services folder under /src folder of *MyFilter1* and *MyFilter2* module with a file that registers the implementation of the *Filter* interface, following the JDK 6 ServiceLoader mechanism.

This way Lookup API can take care of modules interaction. To access *Filter* implementations you can use:

```
Filter f = (Filter) Lookup.getDefault().lookup(Filter.class);
if (f != null)
{
     f.process("Hello World!");
}
```

Lookup has two common usages:

- ***Local lookup*** – asking an object for an instance of some interface (i.e. Node)

- ***Global lookup*** – services, often singleton instances of some class - can be registered into the default lookup (i.e. Filter example)

6 Project Type

A *project type* is a NetBeans Platform term for a grouping of folders and files that is treated as a single unit. Treating related folders and files as a single unit makes working with them easier for the end user. One way in which a project type simplifies life for the user is that you are able to fill the Projects window only with those folders and files that the end user is most likely to work. For example, the Java project type in NetBeans IDE helps the end user to work with the folders and files belonging to a single Java application.

6.1 Module dependencies

- Lookup API
- Datasystems API
- Dialogs API
- File System API
- Nodes API
- Project API
- Project UI API
- UI Utilities API
- Utilities API

6.2 Create your own project type

We start by implementing the `ProjectFactory` class.

```
//Specifies when a project is a project, i.e.,
//if the project directory "MyProject" is present:
@Override
public boolean isProject(FileObject projectDirectory) {
      return projectDirectory.getFileObject("MyProject") != null;
}

//Specifies when the project will be opened, i.e.,
//if the project exists:
@Override
public Project loadProject(FileObject dir, ProjectState state) throws
IOException {
      return isProject(dir) ? new MyProject(dir, state) : null;
}

@Override
public void saveProject(final Project project) throws Exception {
      FileObject projectRoot = project.getProjectDirectory();
      if (projectRoot.getFileObject("MyProject") == null) {
            throw new IOException("Project dir deleted");
      }
      //save MyProject }
```

Next, we implement the `Project` class.

```
class MyProject implements Project {

    private final FileObject projectDir;
    private final ProjectState state;
    private Lookup lkp;

    public MyProject(FileObject projectDir, ProjectState state) {
        this.projectDir = projectDir;
        this.state = state;
    }

    //The project type's capabilities are registered in the project'
    //lookup:
    @Override
    public Lookup getLookup() {
        if (lkp == null) {
            lkp = Lookups.fixed(new Object[]{
                state, //Allow outside code to mark the project as
                       //needing saving
                new ActionProviderImpl(), //Provides standard actions
                                          //like Build and Clean
                new MyProjectDeleteOperation(), //Delete operation
                new MyProjectCopyOperation(this), //Copy operation
                new Info(), //Project information implementation
                new MyProjectLogicalView(this),//Logical view of
                                               //project implementation
                });
        }
        return lkp;
    }
}
```

Finally we implement `ActionProviderImpl`, `MyProjectDeleteOperation`, `MyProjectCopyOperation`, `Info` and `MyProjectLogicalView` class. You can find more info here – http://platform.netbeans.org/tutorials/nbm-projecttype.html.