RAPTURE FEATURE LAYOUT

september 4, 2012

RESEARCH BY
INCAPTURE

1

FEATURE

A *feature* in **Rapture** is an installable *package* that encapsulates all of the changes necessary to make that feature exist in a **Rapture** environment. A feature is physically a folder structure in a file system with a very specific layout – named sub-folders define what partitions and types need to exist, what users and permissions need to be setup, what scripts need to be installed, the workflows that are to be created and what other features must exist before this feature will work. A feature can also define initial data sets that need to be updated.

The presence of a feature (and which version is installed) can be determined through the **Rapture** *Feature* api - but *normal* users will use the user interface provided for this purpose.

This document describes the layout of a feature, what is an absolute requirement, what is optional and the process a *Feature Loader* will go through to install a feature.

types of feature

A *feature* can be presented to the installer in one of three ways.

1. A folder whose contents follows a very strict format.
2. A jar file which has been created from the same structure as above.
3. A jar file in a remote location (e.g. available via `http`) whose URL is formed from the name of the feature and its version.

In all cases the Feature Installer application will download and extract a copy of the feature as needed and install that. The next section describes the strict format of the resulting folder.

top level

A feature's layout is a series of files and folders. When a feature is installed the process will start at a top level folder. Within this folder there *must* be a file, `feature.txt` that is a configuration reference for this feature. If this file is not present in the root folder of a putative feature the installer will not continue.

The contents of a typical `feature.txt` document is reproduced below:

```
{
  "feature" : "bootstrap",
  "version" : {
        "major" : 1,
        "minor" : 0,
        "release" : 0
      },
  "depends" : {
         "featureX" : [ 1, 0, 0 ]
      },
  "description" : "This is the bootstrap feature,
       it must be installed before anything else"
}
```

Table **??** describes the fields in this file.

Field Name	Description
feature	This is the unique name of this feature
version	This is the version number of this feature, and will be used to determine whether this feature could overwrite an existing installation
depends	Depends is a set of features, with the value against each feature being the minimum version number required for *this* feature to be installed
description	A useful description for a user interface

Table 1: Fields in the top level feature definition

structure

The required *structure* that must be in place for a feature to work is defined through a `structure.txt` file in the root folder. By structure we mean the partitions, types, fountains, queues and the like. These are entities that must exist, and if they do not exist they must be created. In a feature installation this is the first step.

The file `structure.txt` in the root folder of the feature is a json formatted document that contains a list of the partitions that must exist. If any partition does not exist it is created using **Rapture** 's `createPartition` api call. Furthermore the `partitions` file contains a list of the types that must exist in a given partition and the configuration needed to create that partition. Also defined are the queues and the fountains in a similar manner. The file is basically a structured input document to a series of `createXXX` calls through the **Rapture** API, with checks made to determine whether the structure exists or not.

A typical `structure.txt` file is reproduced below.

```
{
  "structure" : [
     {
       "name" : "mdrs",
       "types" : [
          { "name" : "bond" , "configuration" :  "REP {} USING MONGODB { pre
          { "name" : "future", "configuration" :  "REP {} USING MONGODB { pr
       ],
       "fountains" : [
          { "name" : "test" ,
               "configuration" : "FOUNTAIN { base='36', length='8',
              prefix='' } USING MONGODB { prefix='mdrs.testfount' }",
               "type" : "bond"
          }
       ],
```

```
        "queues" : [
           { "name" : "priceloader", "configuration" : "QUEUE USING
               MONGODB { prefix='mdrspriceloader' }"
           }
        ],
        "operations" : [
            { "name" : "claimOrder",
                "type" : "order",
                "script" : "claimOrder",
                "paramDef" : "trader"
           }
        ]
    }
 ]
}
```

The structure is self-describing but for the avoidance of doubt, the top level key is the name of a partition that must exist. Below that are the entities described in Table **??** with each entry containing enough information for the installer to call the appropriate `createXXX` call in **Rapture** .

Area Name	Description	API Call
types	**Rapture** types	`admin.createType(...)`
fountains	**Rapture** fountain with config and associated type	`fountain.createFountain(...)`
queues	**Rapture** queue	`queue.createQueue(...)`
operations	**Rapture** operations on types	`user.addOperationToType(...)`

Table 2: Configuration structure

scripts

An important aspect of a feature is the Reflex scripts that are the glue to providing the functionality of that feature. Scripts are used in workflows, operations, scheduled tasks and as needed through program calls through a user interface or application.

Scripts are defined in files in a `script` sub-folder of the root folder. Immediately below this folder are folders that match with partition names in **Rapture** and below those folders are the actual files that make up the scripts. **Reflex** scripts should be files ending in the extension "rfx". No other files are deemed to be valid scripts by the feature installer. This means that a developer can put commentary (`readme`) files in folders as descriptions of what is contained in a set of scripts.

The feature installer converts the folder structure below a partition into a fully qualified name of the script, less the extension. As an example, the file:

`feature/script/mdrs/priceloaders/bondpriceload.rfx`

will be imported as the script named `priceloads/bondpriceload` in the partition `mdrs`.

data

Initial data values (or complete imports) can also be added through a feature install. Content is defined in a similar way to scripts, except that the initial folder is `data`. The extension of a file is removed and any files beginning with a 'dot' are ignored.

The content for files in the `data` area will be json formatted.

workflow

Workflows in **Rapture** are an important aspect of a feature. Workflows are defined in a similar way to scripts and data in that they are embodied within files in the file system below a `workflow` sub-folder of the root folder. Workflows are not aligned to any partition so the workflow sub-folder simply contains files and folders, with the same naming technique (by collapsing the folder structure into the workflow name) used to determine the name of a given workflow. The content of the file is actually the serialized contents of the **Rapture** API `WorkflowConfig` structure, and an example is reproduced below:

```
{
  "name" : "test",
  "description: "A test workflow",
  "defaultParameters" : {
            "param1" : 42,
            "param2" : "a parameter"
  }
  "steps" : [
        {
          "name" : "step1",
          "description" : "A step in a workflow",
          "partition" : "test",
          "scriptName" : "step1Test",
          "innerWorkflow" : "",
          "area" : "core",
          "failOnError" : true,
          "repeatCount" : 0,
```

```
          "delayInterval" : 0
       }
  ]
}
```

The feature loader will use this configuration verbatim to define a workflow configuration document and create the workflow.

hooks

The Feature Installer application can optionally run *hook* scripts at the start and end of the installation process. These scripts need to be written in the **Reflex** language and are located in the hook folder.

Table **??** describes the available hooks.

Location	Description
preHook.rfx	The script that will be run at the start of the installation process.
postHook.rfx	The script that will be run at the end of the installation process.

Table 3: Hook scripts

3

SAMPLE FOLDER STRUCTURE

Reproduced below is a directory listing of a sample feature.

```
\feature
  feature.txt
  structure.txt
  \hook
    preHook.rfx
    postHook.rfx
  \script
    \mdrs
       \test
          mysimpletest.rfx
          myothertest.rfx
    \smrs
       \loader
          assetLoader.rfx
  \data
     \mdrs
        \config
           priceLoader.txt
  \workflow
     \test
        testWorkflow.txt
```

The result of running the feature loader on this structure would be to:

1. Ensure that the feature defined in `feature.txt` does not already exist
2. Ensure that the dependencies referenced in `feature.txt` are available and installed
3. Run the **Reflex** script `preHook.rfx`
4. Create the entities in `structure.txt`
5. Create the three scripts defined in the `script` folder
6. Upload the single data document defined in the `data` folder
7. Create the workflow defined in the `workflow` folder
8. Run the **Reflex** script `postHook.rfx`
9. Register that this feature has been installed

4

FEATURE INSTALLER

Features described in this document are installed into **Rapture** using an application known as the `FeatureInstaller`.

The command line to `FeatureInstaller` has the parameters defined in the Table **??**.

Parameter	Sample	Description
-r	http://rapture	The URL for the Rapture environment to apply these features to
-u	admin	The user name to use for the installation procedures
-p	*****	The password to use. (If not supplied, the password will be asked for on the command line)
-f	feature,feature2.jar	The features to install
-m	file:///features	The place to look for remote features

Table 4: Feature Installer parameters

Other possible protocols for the *remote* parameter are defined in Table **??**.

Protocol	Description
file://	Local (or network reachable) file system
http://	Web based resources
ftp://	FTP based resource

Table 5: Feature Remote Protocol

feature resolution

As the Feature Installer application runs it takes each of the features provided on the command line and determines their dependencies. If a dependency is not provided as part of the command line the remote location is looked at to see whether the dependent feature can be loaded from there. When constructing the location of the remote feature the following process is followed:

```
[remotePrefix]/[featureName]/[versionNumber]/[featureName].jar
```

So given a dependency on the feature *sampleOne* at version *1.0.0* and when passed `file:///myfeatures` as the `-m` parameter to the application, the installer will look in the following location for the remote feature:

```
file:///myfeatures/sampleOne/1.0.0/sampleOne.jar
```

Once a remote feature is downloaded *its* dependencies are examined and resolved, and so on until all features needed to install the original feature have been determined and resolved.

creating a feature jar

The following command will create a feature jar given a folder structure that is a valid feature specification.

```
jar cvf ../feature.jar .
```

when executed in the actual folder containing the file `feature.txt`. The resultant file can then be copied to a central repository or passed around as needed.

DISCLAIMER

INCAPTURE TECHNOLOGIES LLC
183 Madison Avenue, Suite 801
New York, NY 10016

RESEARCH BY
INCAPTURE