Resource Traceability SDK

Overview

Resource Traceability gives Polarion users the power to link source code assets semantically. (Link Work Items directly to the source code function, method or variable that implements them.) It can also be used to parse files of different formats not related to programming languages. The processing of resource files is done in two steps. In the first step, resource files that can be parsed are collected from the repository. Then collected files are parsed by the compatible parser. These two components of the Resource Traceability feature are called the Collector and Parser respectively. The Collector is responsible for connecting to a specific repository type like SVN or Git and for collecting the files that were updated since the last collection run. Parsers are responsible for parsing file syntax in a specific programming language or file format. They recognize semantic elements and related links and return them to the collection engine. Parsed links from the collected files are persisted by the collector engine in internal storage.

The Resource Traceability SDK provides a way to develop custom Parser and Collector components. Developers should implement the `com.siemens.polarion.rt.collectors.api.RtCollector` interface to define a custom Collector and the `com.siemens.polarion.rt.parsers.api.RtParser` interface for a custom Parser. Developed components are exposed to Polarion by using `extension` elements in the plug-in manifest file, `plugin.xml`. View the javadoc to get more details.

Custom Collector Example

Workspace Preparation

See the *Workspace Preparation* section in the main Polarion SDK Guide.

Create a Project Plugin

1. Select **File > New > Java Project**.
2. In the dialog that appears, enter your new plugin project name and press the **Next** button.
3. On the **Projects** tab add the required projects to the build path. Plugins `com.siemens.polarion.rt.api`, `com.polarion.platform.repository` and `com.polarion.subterra.base` are mandatory and required for projects to develop the Collector. You will also need to add a plugin that implements the `com.polarion.platform.repository.external.IExternalRepositoryProvider` interface to accept the repository connection configuration in your Collector. Other dependencies like `com.polarion.core.util` are optional. (NOTE: Do not create dependencies on the Polarion platform plugins.)
4. Click **Finish**.
5. Create a `plugin.xml` file and add an `extension` element for your Collector so it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension point="com.siemens.polarion.rt.collectors">
      <collector
            class="com.example.collectors.CustomCollector"
            repositoryProvider="exampleProvider">
      </collector>
   </extension>
</plugin>
```

Implementation

Implement the `com.siemens.polarion.rt.collectors.api.RtCollector` interface to define a custom Collector. The general algorithm for a custom Collector is the following:

1. Initialize the collector using the context. Use `com.siemens.polarion.rt.collectors.api.RtCollectionContext.getConfiguration()` to get the repository connection configuration and cast the object to a specific RT repository configuration interface. Open the connection to the repository.
2. For each branch that is specified in the configuration do the following:
    1. Determine the last revision for the current branch that was processed during the last collection run.
    2. Collect information about added, modified and removed files since the last processed revision. If the last processed revision is not defined, like when it is first ran, collect information about the files located in the repository.
    3. For each added or modified file invoke the `com.siemens.polarion.rt.collectors.api.RtCollectionContext.shouldBeCollected(RtFileProperties)` method to determine if the file *should* be processed by the parsers. If the file should be collected, notify the collection engine about it by invoking the `com.siemens.polarion.rt.collectors.api.RtCollectionContext.collect(RtFile)` method. For each removed file and folder, invoke the `com.siemens.polarion.rt.collectors.api.RtCollectionContext.locationRemoved(String, ILocation)` method.
3. Dispose used objects.

Deployment to Installed Polarion

See the *Deployment to Installed Polarion* section in the main Polarion SDK guide.

Execution from Workspace

See the *Execution from Workspace* section in the main Polarion SDK guide.

Configuration

After a successful deployment of the plug-in into Polarion, you can start using the Collector with repository providers that have an id equal to the id specified in the `plugin.xml`. Using the mentioned `plugin.xml` a Collector can be developed that will work with the repository provider that has the `exampleProvider` id.

Add such a repository to the `repositories.xml`:

```
<exampleProvider>
  <id>exampleProviderRepository</id>
  <!-- repository settings -->
</exampleProvider>
```

Reference it in `resource-traceability.xml`:

```
<repositoryConfiguration id="exampleRepositoryConfiguration" repository="exampleProviderRepository">
  <!-- configuration parametetrs -->
</repositoryConfiguration>
```

Custom Parser Example

Workspace Preparation

See the *Workspace preparation* section in the main Polarion SDK Guide.

Creating Project Plugin

1. Select **File > New > Java Project**.
2. In the dialog that appears, enter a new plugin project name and click the **Next** button.
3. On the **Projects** tab add the required projects to the build path. The `com.siemens.polarion.rt.api` plugin is mandatory and required for a project to develop a Parser. Other dependencies like, `com.polarion.core.util` are optional.
   (NOTE: Do not create dependencies on the Polarion platform plugins.)
4. Click **Finish**.
5. Create a `plugin.xml` file and add the extension element for the Parser so it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension
         point="com.siemens.polarion.rt.parsers">
      <parser
            class="com.example.parsers.CustomParser"
            descriptor="com.example.parsers.CustomParserDescriptor"
            name="com.example.parsers.custom">
      </parser>
   </extension>
</plugin>
```

Implementation

Implement the `com.siemens.polarion.rt.parsers.api.RtParser` interface to define a custom Parser. The general algorithm for a custom Parser is the following:

1. Receive the file input stream using `com.siemens.polarion.rt.collectors.api.model.RtFile.getContent()`.
2. Read the file content and recognise the semantic elements inside the file. Depending on the source file language, you can use third-party parser implementations to build an AST tree from the file content or make other transformations that create the file content model.
3. Find comments or other text fragments that contain links to Work Items and are related to the semantic elements of the content model.

4. Create link objects from the text using the `com.siemens.polarion.rt.parsers.api.RtParsingContext.parseLinks(String, RtPosition)` method.
5. Build and return element objects that will be persisted in internal storage.

> Info: The source code for the C parser example plugin is provided on demand. Please contact Polarion Support.

Deployment to Installed Polarion

See the *Deployment to Installed Polarion* section in the main Polarion SDK guide.

Execution from Workspace

See the *Execution from Workspace* section in the main Polarion SDK guide.

Configuration

After a successful deployment of the plug-in into Polarion, you can start using the Parser for files with a specific extension.

For example, add the follwing Parser configuration in the `resource-traceability.xml` to use a custom parser for `.sql` files:

```
<fileParser id="customParser" name="com.example.parsers.custom" extensions=".sql">
  <properties>
  </properties>
</fileParser>
```