Developer Guide

Build 10.7.0.0202512011257_SNAPSHOT

Table of Contents

List of Tables

1. Preface

This chapter gives an introduction to RCE.

1.1. Abstract

RCE (Remote Component Environment) is an open source software that helps engineers, scientists and others to create, manage and execute complex calculation and simulation workflows. A workflow in RCE consists of components with predefined inputs and outputs connected to each other. A component can be a simulation tool, a tool for data access, or a user-defined script. Connections define which data flows from one component to another. There are predefined components with common functionalities, like an optimizer or a cluster component. Additionally, users can integrate their own tools. RCE instances can be connected with each other. Components can be executed locally or on remote instances of RCE (if the component is configured to allow this). Using these building blocks, use cases for complex distributed applications can be solved with RCE.

1.2. Intended audience

This document is intended for developers who would like to extend RCE according to their needs and/or contribute to RCE's development.

1.3. License Information

For downloads and further information, please visit https://rcenvironment.de/.

1.4. Compatible Operating Systems

RCE releases are provided for Windows and Linux. It is regularly tested on

- Windows 10
- Windows Server 2019
- CentOS 8
- Debian 11
- Ubuntu 20.04 LTS

and should also run on Mint 10.04 and SUSE Linux Enterprise Server 15 SP2.

1.4.1. Support of 32 Bit Operating Systems

Starting with release 8.0.0, RCE is only shipped for 64 bit systems. If you still require 32 bit packages, you can continue to use previous RCE releases, but there will be no standard feature or bugfix updates for them.

2. Introduction

2.1. Getting Started with RCE Development

This section covers setting up a development environment for running, modifying and extending RCE on your local machine. After completing this section you should be able to build and run a development version of RCE from your local development environment.

The development environment is built on top of

- OpenJDK 11 and
- Eclipse for RCP and RAP Developers, Version 2022-03
- Eclipse Checkstyle Plug-in

2.1.1. Install JDK, Eclipse, and the Checkstyle Plugin

- Make sure you have a Java Development Kit [https://www.oracle.com/java/technologies/downloads/] installed on your system in version 11 or higher. The canonical choice of JDK is Eclipse Temurin Version 11 [https://adoptium.net/de/temurin/releases/?version=11].

  Note

  If you are installing a JDK on a centrally administrated computer that already has Java installed, it is usually a good idea to *uncheck* the "Install public JRE" option during installation.

- Download and unpack "Eclipse for RCP and RAP Developers" in version "2022-03" from the Eclipse Foundation [https://www.eclipse.org/downloads/packages/release/2022-03/r/eclipse-ide-rcp-and-rap-developers]. In the following sections, this version is assumed.

  Note

  OpenJDK 11 and Eclipse 2022-03 are most widespread among developers and provides a relatively stable basis for development. However, newer versions of both have been reported to work anecdotally.

- Adapt the `eclipse.ini` file found in the base directory of your Eclipse installation. The following changes are required for productive development of RCE. For additional options, please refer to the Eclipse documentation at https://wiki.eclipse.org/Eclipse.ini.
  - Set the proper path to your JDK installation by adapting the `-vm` parameter: `<your path>/jdk11/bin` immediately below the `-vm` line.
  - Change the maximum heap size to at least 2 GiB by adapting the `-Xmx` parameter somewhere below `-vmargs`. Change this line to, e.g. `-Xmx2048m`. Add such a line if it does not exist.
- Start Eclipse and install the "Eclipse Checkstyle Plug-in" via Help -> Eclipse Marketplace -> Checkstyle. Restart Eclipse if you are prompted.

2.1.2. Importing and building RCE

- If you already have other projects in your Eclipse workspace, it is recommended to create a new workspace for RCE. There are some necessary global settings that may interfere with the other projects (e.g. the so-called "target platform").

- Disable "Project > Build Automatically" in the main menu to speed up the next steps.
- In the eclipse preferences (avaible from "Window"-> "Preferences"), open the page "Plugin-Development" -> "DS Annotations" and check the box for "Generate descriptors for annotated sources." Set the descriptor directory to "OSGI-INF/generated".
- There are currently three ways to properly import the complete RCE source code: from the SVN repository, from provided zip files, or from GitHub. At the moment, only the SVN approach provides access to the current development tree; for the time being, the zip files and the GitHub repository are only updated on release.
- Option 1 - Importing from the zip files provided with each release:
  - Browse to the "source" sub-folder of a release's download location, for example `https://download.rcenvironment.de/rce/10.x/products/standard/releases/10.6.0/source/` for the 10.6.0 release. Download the "source" zip file.
  - Extract the archive. You should see a list of more than 200 sub-folders, all except one beginning with "de.rcenvironment".
  - In Eclipse, select "File > Import > General > Existing Projects into Workspace" and choose the directory that you unpacked the archives into. You should see a long list of projects, once again, all except one beginning with "de.rcenvironment". Select all projects confirm the import.

    Note

    Make sure not to change the "Search for nested projects" option in the import dialog; it must *not* be selected/checked.
- Option 2 - Importing from GitHub:
  - For successfully importing the RCE project, you need a Git client with LFS support.
  - Clone `https://github.com/rcenvironment/rce-main` into a directory of your choice.

    Note

    The clone settings of certain Git clients (e.g. TortoiseGit), have an "LFS" option. Make sure that this option is enabled before you clone.
  - Check out an appropriate branch (e.g. `main` or `integration10`) or tag; by default, the `main` branch points to the source code of the latest release.
  - In Eclipse, select "File > Import > General > Existing Projects into Workspace" and choose the directory that you unpacked the archives into. You should see a long list of projects, once again, all except one beginning with "de.rcenvironment". Select all projects confirm the import.

    Note

    Make sure not to change the "Search for nested projects" option in the import dialog; it must *not* be selected/checked.
- After you have successfully imported the RCE projects using one of the above methods, the next step is to set the RCE target platform in your workspace. A target platform provides external artifacts like the Eclipse RCP framework and various libraries. To get started with RCE development, the easiest way is to use a precompiled target platform. For convenience, there is a Eclipse .target file inside the code base that always points at an appropriate precompiled target platform release. Follow these steps to apply it:
  - In the Project Explorer, navigate to the `de.rcenvironment/eclipse/tp` folder.

    - Open the `local_or_unpacked_thirdparty_build.target` file by double-clicking it.
    - Click "Set as active target platform" in the top right corner. You can close the .target file after this.
- If you previously changed the global Java compiler compliance level e.g. to 1.8 for previous RCE releases, it is recommended to revert this setting to default, or explicitly set it to 11. This setting can be accessed by opening "Window > Preferences" from the menu, and then navigating to the "Java > Compiler" tab. If you never actively changed this setting, no action is required.
- Enable "Project > Build Automatically". Eclipse will start building all projects against the new target platform, which provides all required libraries and OSGi bundles.
- At this point, most projects will have a red error marker. To fix this, open the "Problems" view ("Window > Show View > Problems"). You should see a lot of "Plugin execution not covered by lifecycle configuration" entries. Right-click one of them, select "Quick Fix" from the context menu, select "Discover new m2e connectors" and click "Finish". Eclipse should present one or more installation options with "Tycho" in their name. Confirm their installation and restart eclipse.

    Note

    You only need to do this once per Eclipse installation.

- After this, all RCE bundles should compile without errors (with the exceptions noted below), and you are ready to start developing. If this is not the case, try running "Project > Clean > Clean all projects" from the main menu.

    Note

    On Linux platforms, there will be compilation errors in some Windows-only Excel and TiGLViewer bundles (5 and 3 projects, respectively). We don't have an elegant solution for this problem yet. You can simply close these projects to get rid of the errors, as they won't be loaded at runtime anyway.

2.1.3. Running RCE from Eclipse

Before proceeding to the more detailled settings, try running RCE from Eclipse to verify your setup.

- There are several pre-defined launch configurations for RCE. To find them, navigate to `de.rcenvironment/eclipse/launch` in the "Project Explorer" on the left.
- A good starting point is the "default" configuration. Right-click the "rce.default.launch" file and choose "Run As > rce.default.launch" from the context menu.
- RCE should now start and prompt for an RCE workspace location. Confirm the default value or choose another empty folder.

2.1.4. Configuring Workspace Mechanic (optional)

Note

The Workspace Mechanic project in its original form is not being maintained anymore, and the original project site is gone. However, it has been forked and is being continued by a new maintainer at this location [https://github.com/alfsch/workspacemechanic/]. While this is not an "official" successor, it seems to be the de-facto location of this project now.

Workspace Mechanic (which can be installed via Eclipse Marketplace from this location [https://marketplace.eclipse.org/content/workspace-mechanic]) is an Eclipse plugin that automates common

settings in local workspaces. For RCE, the most important settings are the Java code formatting rules and templates. Other settings are provided for convenience, like disabling the console output limit, or showing line numbers in the editor.

Configuring Workspace Mechanic consists of copying a set of "rule" files to a location where the plugin can find them. There are two options for this:

- If you want to apply the rules to all Eclipse installations on your machine, use the `.eclipse/mechanic` subfolder in your home directory; by default, this is `/home/<user id>/.eclipse/mechanic`.
- To apply the rules to a single Eclipse installation only, use `<eclipse installation folder>/configuration/com.google.eclipse.mechanic/mechanic`.

Using your system's file browser, navigate to the folder of your choice. Using any SVN tool, check out `https://svn.dlr.de/rce/new/meta/eclipse/mechanic/` into a sub-folder called "`checkout`" within it. (Note that the actual name of the sub-folder is not relevant; adapt if you like.) This sub-folder now contains common rules on its top level, and optional or experimental rules in sub-folders. Copy all common rules to the parent folder (the one you started in), and add any optional rules that you want to apply as well. (TODO add and describe batch/shell files for this.)

> **Note**
>
> These rule files will most likely be integrated into the main project at some point, making this extra checkout step unnecessary. Please note that these rule files are currently not available as part of the GitHub source code mirror [https://github.com/rcenvironment/rce] or the released source zip files either, which makes them inaccessible unless you have access to the internal SVN server.

The next time you open a workspace, Workspace Mechanic should pick up these rule files and show a notice asking if it should apply them. See the plugin's web site [https://github.com/alfsch/workspacemechanic/] for further information.

2.1.5. Configuring Code Formatting and CheckStyle Rules (optional)

If you only plan to try out or modify RCE locally, you can safely skip this section. If you plan to commit your changes to the central code base, however, you need these settings to get your code accepted into the repository. Code that does not match the style guidelines will be refused on commit. The Checkstyle-CS plugin simplifies development by highlighting violations that need to be fixed.

To configure Checkstyle-CS for RCE:

- Open the Checkstyle preferences (Window > Preferences > Checkstyle).
- Click "New" on the right side. Enter "RCE" as the name of the configuration.
- Choose "Project Relative Configuration", click "Browse" and choose `de.rcenvironment/checkstyle/checks.xml`.
- Click "Ok" in the main dialog. The list of configurations should now have three entries; select "RCE" and click "Set as Default" on the right side.
- Close the preferences with "Apply and Close" and confirm the rebuild.

> **Note**
>
> We are currently using version 6.19 of the Eclipse CheckStyle plugin within the development team. Using a newer version (e.g. 8.0) works as well, but you may see error markers for constructs that do not actually violate the RCE

code guidelines. Versions > 8.0 do not work with current code guidelines. We will most likely adapt/migrate the CheckStyle settings in the near future.

To configure the Eclipse source code formatter:

- Open the code formatter preferences (Window > Preferences > Java > Code Style > Formatter).
- Click "Import", browse to your `de.rcenvironment/eclipse/` checkout folder and choose the "`eclipse-formatter.xml`" file.
- You should now see "RCE" as the "Active Profile".
- Click "Apply and Close" to activate the settings.

Note

There are rare cases where these code formatter settings lead to source files that are not being accepted by our CheckStyle rules. These inconsistencies are being collected and tracked in issue #0005898 [https://mantis.sc.dlr.de/view.php?id=5898], and will be fixed/addressed in a future update.

2.1.6. Building RCE

RCE's build system has undergone significant changes for the 10.6.0 release. Starting with that release, builds are always performed by using Bash scripts located in the "build" folder in the root of the repository. Building on Windows is still fully supported by calling these scripts within a "git bash" shell provided by the commonly used "Git for Windows" software package. Using these build scripts always generates builds for both Linux and Windows, regardless of the platform the scripts are run on.

For an introduction on how to execute the default ("product") build, refer to the `build/README.ad` file in the `rce-main` repository. Note that you will need to provide a matching build of the `rce-thirdparty` repository to build against. You typically do not need to build that repository locally, as builds matching each RCE release are conveniently available from the RCE download page. See the `de.rcenvironment/eclipse/tp/README.ad` file for more information.

Note that the transition to the new build system is not fully complete yet, so you may still find traces of the old setup in the repository and in the documentation. Also, while this documentation enables you to build fully functional distribution packages of RCE, equivalent to the official zip release files, its is currently not describing all features of the new build system.

2.2. Common Classes and Interfaces

This section lists classes and interfaces that every RCE developer should be familiar with.

Note that at this point, this list is probably incomplete. If you come across a class you wish you had known earlier, please let us know.

2.2.1. General

ResolvableNodeId, InstanceNodeId, InstanceNodeSessionId, LogicalNodeId, LogicalNodeSessionId (former: NodeIdentifier)	Explanation for NodeIdentifier: This interface represents the "identity" of a node, and is used whenever nodes are specified in API calls. In general, these node identifiers (or "node ids") are stored and reused by nodes, so they are persistent unless the node's operator deletes its settings folder. From a developer's perspective, the inner format of these ids is usually not relevant.

TODO: Replace with explanation referring to current node identifier approach

Package: de.rcenvironment.core.communication.common

2.2.2. Components and Workflows

TypedDatum — Represents a chunk of data that is passed between the components of a workflow. This is the central abstraction of all data passing, so you will encounter it when you start writing or modifying workflow components.

Package: de.rcenvironment.core.component.datamodel (Note: may be moved in 5.0.0)

2.2.3. Utilities

ThreadPool/SharedThreadPool — This is a central thread pool that should be used for all asynchronous operations (except for the SWT GUI thread itself, and GUI-embedded "background tasks"). Always use this instead of creating Thread or Executor/ExecutorService instances.

(TODO add code example(s), explain @TaskDescription, ...)

Package: de.rcenvironment.core.utils.common.concurrent

TempFileUtils — This utility class should be used whenever a temporary file or directory should be created. Its main benefit is that it allows for managed cleanup of leftover temporary files/directories (Note: This is not yet implemented!). Additional benefits are convenience functions (like generating a temporary file with a given filename or name pattern), central handling of cleanup issues (like undeletable files), and making sure that all temporary files are created in a consistent location.

Package: de.rcenvironment.commons (Note: will be moved in 5.0.0)

3. Build and Infrastructure

Note

The chapter previously present here was both highly outdated and incomplete, both regarding RCE's release process as well as the technical steps involved. To avoid confusion, we have removed it for the time being. We aim to review this in a future release and decide on which updated information to add back.

3.1. Build Process FAQ / Tips and Tricks

This section gives answers and hints to common build issues.

Q: After running a local product build using the provided script files, where do I find the generated product zips and files?

A: The generated distribution ("product") zip files are located in `build/target/products`, with a matching Eclipse "update site" provided in `build/target/products/updatesite`. A zipped version of the update site is also available in `build/target/products`.

Q: I created a new snapshot / RC / release build of the `rce-thirdparty` repository. What do I have to edit to make the product build use it?

A: Unlike before the RCE 10.6.0 release, local `rce-thirdparty` builds do not require any change in the Eclipse environment used to work on the `rce-main` code base. The `rce-thirdparty` builds are always created in a defined location, so simply updating your "target platform" definition should be sufficient.

Q: A package import in Eclipse fails with the error message "The import xxx cannot be resolved" although the imports and exports are set correctly in the respective Manifest files. What else can I try to solve the problem?

A: Go to the Project Explorer and select the affected bundle. Open the context menu and select "Plug-in Tools > Update Classpath...". Select all affected bundles (possibly including the Fragment-Host) from the dialog list provided and finish.

4. Coding Guidelines

4.1. Adding a new Bundle with Production Code

Here, we describe how to add a new bundle to RCE. This bundle will only contain production code, but no test code. In RCE, unit tests reside in "companion bundles" to those deployed in production. The process for how to add a "testing bundle" to RCE is undocumented as of yet.

4.1.1. RCE Structure

RCE is built on top of the Eclipse RCP framework. Speaking in terms of RCP, RCE is a *product*, which consists of a number of *features*, which in turn consist of a number of *plug-ins*. The product itself is defined in the file `de.rcenvironment/maven/modules/repository.mainProduct/rce_default.product`. Each feature and each plug-in corresponds to one top-level directory in the RCE-repository. There exist, however, top-level directories which do not correspond to either a feature or a plug-in, e.g., `de.rcenvironment`.

A plug-in consists of a number of Java packages and a manifest which defines how the feature interacts with other features. This interaction takes the form of, e.g. providing editors, views, or hooking into so-called *extension points* defined by other features. As with most RCP-based applications, some features of RCE are provided by Eclipse or some other third-party developer, while others are developed by the RCE developers.

When compiling RCE, Maven compiles each feature and each plug-in more or less individually before packaging all features and plug-ins into the resulting zip. At runtime, all plug-ins are connected to one another using OSGi, which mainly serves as our dependency injection framework. Speaking in terms of OSGi, each plug-in is an *OSGi bundle* and vice versa. We use the terms (Eclipse) Project, (OSGi) Bundle, and (Eclipse) Plug-in interchangeably in this guide.

In order to create a new bundle, you have to

1. Create a new Eclipse project in a top-level directory in the root directory of the RCE repository
2. Add the new bundle (i.e., the new plug-in) to the Maven build process so that it is compiled and packaged as a plugin when building via Maven
3. Add the new plug-in to an existing feature so that it is included in the RCE product

We adress each of these points individually in the following sections

4.1.2. Create a new Eclipse Project

There are two major ways to create a new Eclipse Project containing code for productive use: Either by copying an existing bundle and subsequently importing that bundle into Eclipse, or by creating a new bundle from scratch using the Eclipse IDE. Here, we only describe the former method.

First, choose some *bundle identifier* as well as a human-readable *display name* for your new bundle. For this guide, we pick the identifier `de.rcenvironment.core.newbundle` and the display name *RCE Core New Bundle*.

Then, create a copy of some top-level directory in the repository and rename it with the name of your bundle identifier. That directory should contain the subdirectories `META-INF`, `src`, as well as the files `.checkstyle`, `.classpath`, `.project`, `build.properties`, and `pom.xml`.

Adapt the file `META-INF/MANIFEST.MF` by changing the `Bundle-Name` to the display name of your bundle and the `Bundle-SymbolicName` as well as the `Automatic-ModuleName` to the identifier of your bundle. Moreover, since we are creating an empty bundle that does not yet export any Java packages or imports any dependencies, remove the entries `Export-Package` and `Import-Package`. After these changes, the file `META-INF/MANIFEST.MF` should look as follows:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: RCE Core New Bundle
Bundle-SymbolicName: de.rcenvironment.core.newbundle
Bundle-Version: 10.2.2.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-11
Bundle-Vendor: DLR
Automatic-Module-Name: de.rcenvironment.core.newbundle
```

Remove the file `.checkstyle`. We will direct Eclipse to regenerate that file towards the end of this section.

Adapt the file `pom.xml` by setting the `artifactId` and the `name` to the id and the display name of your bundle, respectively. After these changes, the file pom.xml should look similar to this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <artifactId>de.rcenvironment.core.newbundle</artifactId>
 <name>RCE Core New Bundle</name>
 <version>10.2.2-SNAPSHOT</version>
 <packaging>eclipse-plugin</packaging>

 <parent>
  <groupId>de.rcenvironment</groupId>
  <artifactId>de.rcenvironment.componentgroups.standard.parent</artifactId>
  <version>1.0.0</version>
  <relativePath>../de.rcenvironment.componentgroups.standard/parent.pom</relativePath>
 </parent>
</project>
```

After finishing these adaptations, import the new bundle into RCE via `File -> Import -> General -> Existing Projects into Workspace`. Since you have copied an existing Eclipse project, Eclipse should recognize the new project and import it.

4.1.3. Add the new Bundle to Maven

Recall that RCE can be built either via Eclipse (during development) or via Maven. Since you have already imported your new project into Eclipse in the previous step, it will automatically be built during development. Thus, it remains to include the new project / bundle / plug-in into the Maven build.

There exist multiple *build scopes*, each of which consists of a list of projects that Maven should build. Each build scope is defined in an individial `pom.xml` file in a subdirectory of `de.rcenvironment/maven/modules`. Pick one or more of these build scopes in which you want to include your new bundle

> **Note**
>
> In most cases either `components.all` or `core.combined` are a reasonable choice of build scope.

Once you have picked a build scope, open the `pom.xml` contained in that directory and add the path to your new bundle to the list of modules already contained in the `pom.xml`. In our example, we pick the build scope `core.combined` and obtain a file `de.rcenvironment/maven/modules/core.combined/pom.xml` similar to the following:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
```

```
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<artifactId>de.rcenvironment.modules.core.combined</artifactId>
<name>RCE Module ${project.artifactId}</name>
<packaging>pom</packaging>

<parent>
 <groupId>de.rcenvironment</groupId>
 <artifactId>de.rcenvironment.maven.parent.module</artifactId>
 <version>1.0.0</version>
 <relativePath>../../parent/module</relativePath>
</parent>

<modules>
            ... truncated for brevity ...
 <module>${projects-root}/de.rcenvironment.core.newbundle</module>
            ... truncated for brevity ...
     </modules>
</projects>
```

Now your bundle will be compiled and packaged when building a snapshot. The resulting plug-in, however, will not be included in the final snapshot: Recall that, speaking in terms of Eclipse RCP, RCE is a product, which consists of features, which in turn consist of plug-ins. At this point, your new plug-in is not yet part of a feature, thus it does not get deployed into the product.

4.1.4. Add the new Plug-in to the RCE product

Pick some existing feature to which you want to add your new plug-in. Features are contained in top-level directories ending in `.feature`. In this example, we choose the feature `de.rcenvironment.core.feature`. Among others, this directory contains the file `feature.xml`, which defines the plug-ins that constitute this feature. You can either edit this file via Eclipse or manually via a text editor of choice.

When opening the file in Eclipse, Eclipse will provide a graphical editor for the file. Here, you can add your new bundle via `Included Plug-Ins -> Add...`. If you are using a plain text editor, add an entry similar to `<plugin id="de.rcenvironment.core.newbundle" download-size="0" install-size="0" version="0.0.0" unpack="false"/>` as a child of the top-level element `<feature>` in the file `feature.xml`.

Independently of the used method, your plug-in is now part of the feature you have chosen and will be included in the RCE product.

4.2. Developing a new Component

4.2.1. Component Bundle Setup

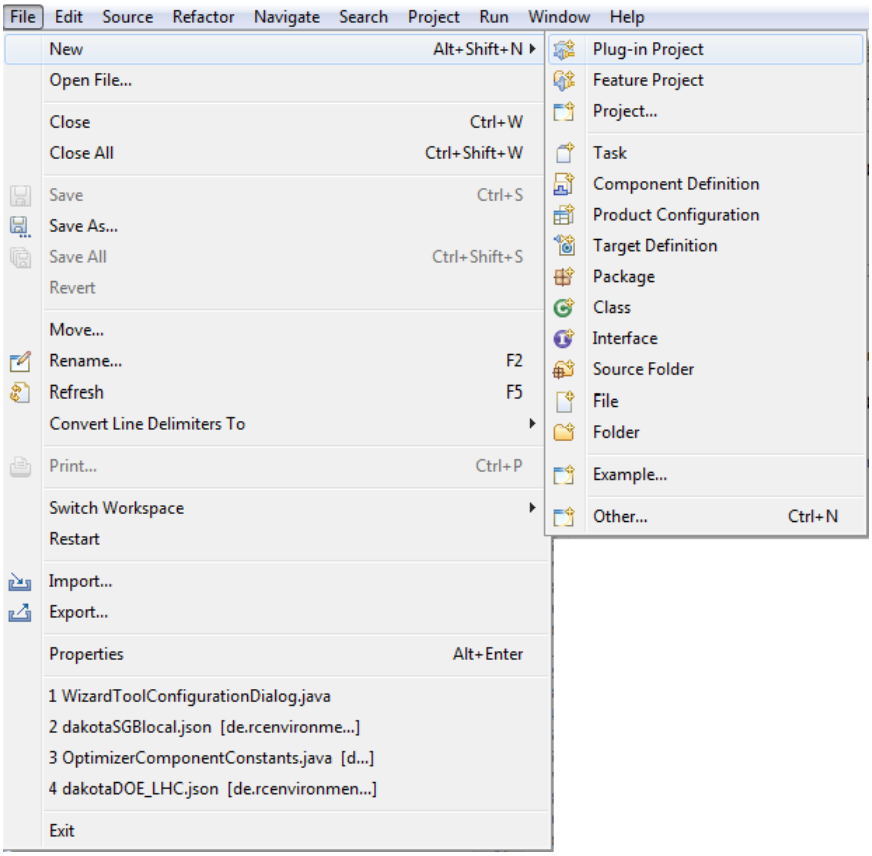
An RCE component usually consists of up to three OSGi bundles: named the *execution*, *gui*, and *common* bundles:

- Execution: Contains the component's lifecycle.
- GUI: Contains the user interface to configure the component. If a component does not require a user interface this bundle can be ommitted.
- Common: Contains code resources which are used by both the execution and the GUI bundle. This bundle is optional, too.

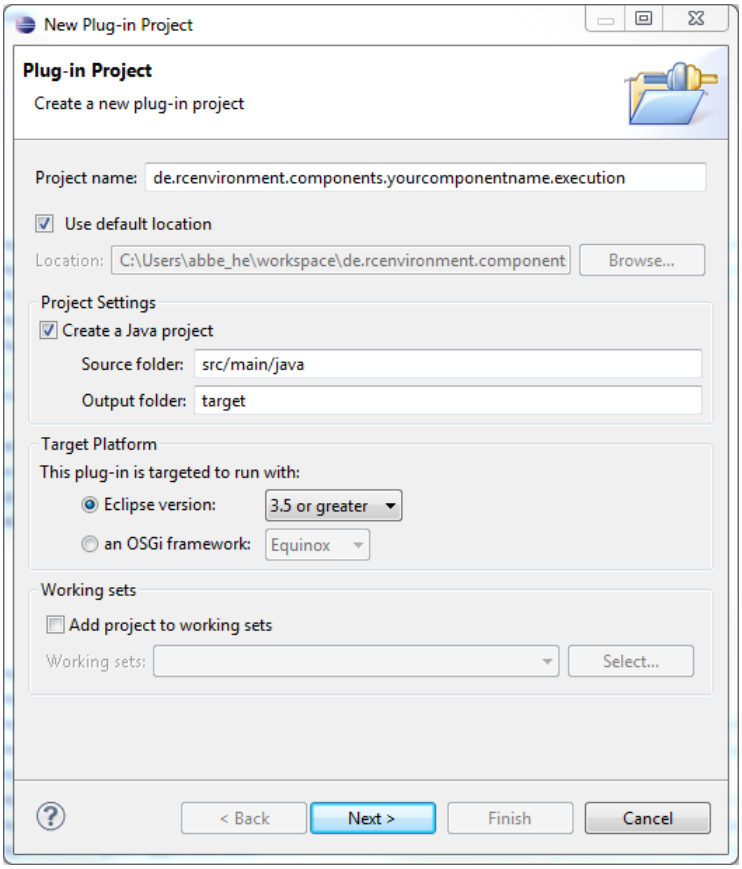
Note that in the context of Eclipse, OSGi bundles are called *Plug-In Projects*, and can be extended with Eclipse-specific features. In practice, the terms *bundle* and *plugin* are often used interchangeably.

4.2.2. Creating the *execution* Bundle

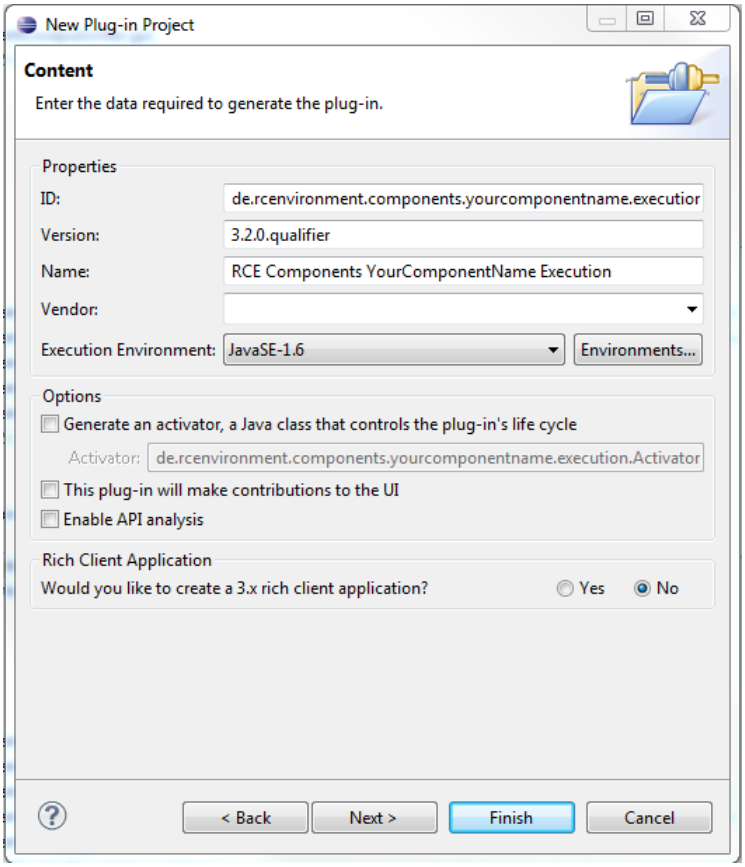
To start developing your first component, generate a new Eclipse Plug-In Project by selecting the corresponding item in the menu bar:



Fill in the dialog properties to configure the plug-in project. The name of the new project should match the RCE naming conventions. That means it should start with "de.rcenvironment.components." and end with ".execution"; so the full name of the execution bundle should be "de.rcenvironment.components.*<yourcomponentid>*.execution", where *<yourcomponentid>* is the ID of your new component. Change the property "Source folder" to "src/main/java" and "Output folder" to "target". Proceed by clicking "Next >".



Now specify the version of your component; you can choose this freely. The name of this plugin should always be "RCE Components `<YourComponentName>` Execution" for the execution bundle, where `<YourComponentName>` again is your component's display name. Press the "Finish" button to complete the general configuration of this plugin.



Eclipse now creates the configured structure of folders, but the plugin is not ready for being used as an RCE component yet.

Create a folder called "resources" in the project you just created. If you have icon files for your component, put them into this folder. Supported formats are PNG, JPG, BMP and GIF. We recommend a resolution of 16×16 and 32×32 pixels. Conventionally these icons are namend "<*yourcomponentname*>16.png" and "<*yourcomponentname*>32.png". Also create an "inputs.json", an "outputs.json" and a "configuration.json" file in the resource folder you just created, where you later define the inputs, outputs and configuration of your component.

As a start, the files can contain an empty JSON object as content. So it suffices to enter the following text in these three files:

```
{ }
```

Note that the files must be present and must not be empty.

Now create a Java class in your source folder by right-clicking on "src/main/java" in the Eclipse Project Explorer. As this will be the central class of your component, give it a name like "<*YourComponentName*>Component.java" (<*YourComponentName*> is your component's display name in camel case, and then add "Component.java"). This Java class must extend `de.rcenvironment.core.component.model.spi.DefaultComponent`.

To implement the functionality of your component override the according methods. The most basic methods to be overriden are:

- processInput: Is called whenever the component receives a new input. In loops this method is called multiple times.
- start: Is called at component start once. Initializations can be placed here.
- dispose: Is called when the component disposes. Clean up methods can be placed here to release resources.

Now create a folder called "`OSGI-INF`" in your project folder, by using the standard Eclipse function (File->New->Folder). This folder will contain all OSGi service definitions for your project. One way to create an OSGi service definition is creating a file "*<yourComponentName>*.xml" ("*<yourComponentName>*" is your own component's display name again) and copying the following source code into it. Replace all occurences of "*<yourComponentName>*" with the display name of your component, and all occurences of "*<yourcomponentname>*" with its id:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
 factory="de.rcenvironment.rce.component" name="<yourcomponentname>">
 <implementation class="de.rcenvironment.components.<yourcomponentname>.
 execution.<yourComponentName>Component" />
 <service>
  <provide interface="de.rcenvironment.core.component.registration.api.Registerable" />
 </service>
 <property name="rce.component.class" type="String" value="de.rcenvironment.components.
 <yourcomponentname>.execution.<yourComponentName>Component" />
 <property name="rce.component.version" type="String" value="1.0" />
 <property name="rce.component.name" type="String" value="<yourcomponentname>" />
 <property name="rce.component.group" type="String" value="Test" />
 <property name="rce.component.icon-16" type="String" value="/resources/yourcomponentname16.png" />
 <property name="rce.component.icon-32" type="String" value="/resources/yourcomponentname32.png" />
 <property name="rce.component.inputs" type="String" value="/resources/inputs.json" />
 <property name="rce.component.outputs" type="String" value="/resources/outputs.json" />
 <property name="rce.component.configuration" type="String" value="/resources/configuration.json" />
</scr:component>
```

Some of these definitions are optional or refer to elements that do not exist yet. The first lines provide the general XML header, followed by the OSGi root element. The `factory` attribute is the part that links this definition into the RCE framework. The `name` attribute defines your component's name when inspected with OSGi tools and should be the same as the `rce.component.name` property below. The `implementation class` entry defines the main Java class of the component. The `rce.component.group` property sets the GUI group in which your component will appear. (Since the component is still under development, something like "Test" is a good choice for now.) The `rce.component.icon-16` and `rce-component.icon-32` properties are optional and define the icon for your component. The `rce.component.inputs`, `rce.component.outputs` and `rce.component.configuration` attributes define the locations of configuration files that will be described later.

The created plugin project automatically contains a folder named "META-INF". Edit the "MANIFEST.INF" file in this directory by double-clicking it and selecting the "MANIFEST.MF" tab. Add the following lines to it and save:

```
Service-Component: OSGI-INF/*.xml
RCE-Component: true
Export-Package: de.rcenvironment.components.<yourcomponentname>.execution
```

Note that "Export Package: " and the first name have to be in the same line. A second element would be added in the next line with a leading space. Moreover the file must end with a linebreak.

Also note that errors containing "inconsistent hierarchies" can be solved by adding the required referenced packages in the Import-Package property analogous to the Export-Package property.

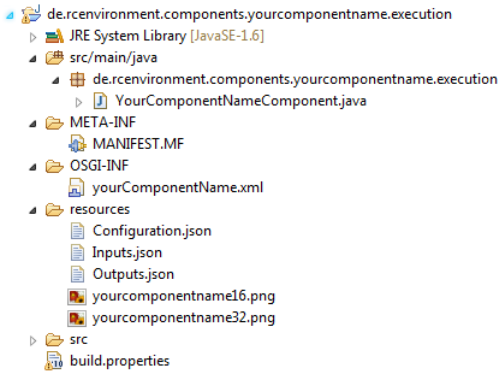
The first line tells the OSGi framework where to look for the component declaration we created before. The second line declares this plug-in as an RCE component. The last line makes the package of your main class visible to other bundles, which is required so RCE can load and initialized the component.

Next, edit the "build.properties" file in the main folder of your plugin project. This file should always look the same for the execution bundles and can be copied from the sample component. It should look like this:

```
source.. = src/main/java
bin.includes = META-INF/,\
OSGI-INF/,\
resources/,\
```

```
.
```

After this step, the plugin configuration is complete. It should look like this in the Eclipse Package Explorer:



When you start RCE from Eclipse and open a workflow file, your component should be shown in the component palette on the right, in the group that you have set in the "OSGI-INF/..." XML file.

4.2.3. Creating a *common* Bundle

Generate a new Eclipse Plug-In Project by selecting the same option as in the "execution" bundle section above. The project name should follow the RCE naming conventions, similar to the execution bundle but ending with ".common" instead of ".execution".

Add the Java classes to the source folder, within a package with the same name as the bundle's name (or sub-packages of it). For holding shared constants, the usual convention is a Java class named "*<YourComponentName>*ComponentConstants.java". Create this class, and add your first constant for defining the component id and adapt the placeholders in the usual manner:

```
public static final String COMPONENT_ID =
    "de.rcenvironment.components.<yourcomponentname>.
        execution.<YourComponentName>Component";
```

In order to keep track of your constants, it's advisable to give all constants a short comment.

To complete the setup, add the names of the all Java packages you created to the "Export-Package" attribute in the source code of the META-INF/MANIFEST.MF file. Example:

```
Export-Package: de.rcenvironment.components.<yourcomponentname>.common
```

4.2.4. Creating a *gui* Bundle

To add a graphical user interface for your component, create a "gui" bundle. Start by generating a new blank Eclipse Plug-In Project, as you did for the "execution" and "common" bundle.

The graphical user interface is shown in the properties tab which is usually shown at the bottom of RCE when the component is focused in the workflow editor.

Custom sections can be defined. Therefore create a package called like the project containing it. Inside the package create a class called "*<YourComponentName>*Section.java" which extends `de.rcenvironment.core.gui.workflow.editor.properties.WorkflowNodePropertySectio`
Override the method `createCompositeContent` to fill the GUI.

Each component GUI must provide a *Component Filter*. It is used to determine which sections the GUI for the respective component consists of. Inside the package you just created add a class called "<*YourComponentName*>ComponentFilter.java". extending `de.rcenvironment.core.gui.workflow.editor.properties.ComponentFilter`. To define the component filter, override the following method:

```
@Override
public boolean filterComponentName(String componentId) {
return componentId.startsWith(YourComponentNameComponentConstants.COMPONENT_ID);
}
```

As usual, change *`YourComponentName`* to the name of your own component. This method will only return `true` for the component id used in the "execution" bundle, so it will only be shown for that component.

To provide sections as GUI elements, add a new file called "plugin.xml" to the root of the project folder. It is common to have a section where you can manage inputs and outputs and another section where component specific GUI elements are located. The following code demonstrates this and can be pasted into the plugin.xml you just created. As usual, adapt the component names and IDs accordingly. It is explained below:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

   <!-- Property Sections -->
   <extension point="org.eclipse.ui.views.properties.tabbed.propertySections">
      <propertySections contributorId=
      "de.rcenvironment.rce.gui.workflow.editor.WorkflowEditor">

         <propertySection
               tab=
               "workflow.editor.tab.<yourcomponentid>.General"
               class=
               "de.rcenvironment.components.<yourcomponentid>.
               gui.<yourComponentName>Section"
               id=
               "workflow.editor.tab.Properties.Section.General"
               filter=
               "de.rcenvironment.components.<yourcomponentid>.
               gui.<yourComponentName>ComponentFilter"
               enablesFor="1">
         </propertySection>

         <propertySection
               tab=
               "workflow.editor.tab.<yourcomponentid>.InputsOutputs"
               class=
               "de.rcenvironment.core.gui.workflow.editor.
               properties.DefaultEndpointPropertySection"
               id=
               "workflow.editor.tab.Properties.Section.InputsOutputs"
               filter=
               "de.rcenvironment.components.<yourcomponentid>.
               gui.<yourComponentName>ComponentFilter"
               enablesFor=
               "1">
         </propertySection>

      </propertySections>
   </extension>

   <!-- Register Property Sections -->
   <extension point="org.eclipse.ui.views.properties.tabbed.propertyTabs">
      <propertyTabs contributorId=
      "de.rcenvironment.rce.gui.workflow.editor.WorkflowEditor">

         <propertyTab
             label="General"
               category="default"
               id="workflow.editor.tab.<yourcomponentid>.General">
         </propertyTab>

         <propertyTab
               label="Inputs/Outputs"
               category="default"
               id="workflow.editor.tab.<yourcomponentid>.InputsOutputs">
         </propertyTab>
```

```
    </propertyTabs>
  </extension>

</plugin>
```

For each section that you would like to add to the "Properties" view, add the entries like you can see in the propertySection and the propertyTab parts. Make sure that the "filter" attribute contains the correct path of the component filter class created above. The first entry in property section and property tab adds a general custom section to the GUI while the second entry adds a section to manage inputs and outputs.

Remember to keep the file valid by closing the definition blocks with the corresponding end tags:

After assigning the property tabs to their classes, you have to register each tab as a `propertyTab`.

Note that the ID of the propertyTab should be the same as the "tab" in the propertySection above. Customise the section title by editing the `label` attribute.

TODO: Add explanation how to use Messages file.

4.2.5. Adding your new component to SVN

It is recommended to develop a new component in a separate development branch. Therefore create a new branch. One of the several ways to do so is using TortoiseSVN. Check out the trunk. In the context menu of this folder select "TortoiseSVN > Branch/Tag". Navigate to the destination path which should be located in the branches folder. Call the new branch "dev_*`YourComponentName`*" where *`YourComponentName`* is the display name of your component. Then add the folders of the projects you created above to the structure in your file system. Select them and right click to open the context menu. Then select "TortoiseSVN > Add...". To commit them into the branch, open the context menu again and select "Commit... ".

4.3. Logging

4.3.1. General Configuration

(TODO)

4.3.2. Verbose Logging

Some log messages are disabled by default as they produce large amounts of output, and are only needed in special circumstances (typically, for debugging). This is called "verbose logging", and is controlled by the `DebugSettings` utility class. Typical usage is to initialize a final (static or non-static) field in the logging class with the returned setting, to only incur the configuration checking overhead once:

- non-static field (preferred in most cases, as there is no risk of copying/pasting with the wrong class parameter): `private final boolean verboseLogging = DebugSettings.getVerboseLoggingEnabled(getClass());`
- static field / constant (preferred for classes that are instantiated very frequently): `private static final boolean VERBOSE_LOGGING = DebugSettings.getVerboseLoggingEnabled(TheClassName.class);`

To control the verbose logging, set the `rce.verboseLogging` system property. Example: `rce -p myProfile -vmargs -Drce.verboseLogging=*.NodePropertiesServiceImpl,*Workflow*`. The syntax of the pattern is a comma-separated list of identifiers. A "*" wildcard matches any part of the class name, including the dot ("."). An empty string disables all verbose logging.

> **Note**
>
> As with any other JVM property, this parameter must be placed behind the `-vmargs` delimiter, which separates it from the "direct" RCE command-line arguments (like `--headless` or `-p <profile>`). This rule also applies when adding this parameter to an `rce.ini` file.

If this system property is not set, the `DebugSettings.DEFAULT_VERBOSE_LOGGING_PATTERN` constant's value is used. While developing in Eclipse, it can be useful to enter a verbose logging pattern there, as this affects all local RCE instances at once without editing multiple launch configurations. (As usual, it is your responsibility to make sure these local debug values are not commited into version control.)

Identifiers created before 8.1.0 were fully qualified Java class names; since 8.1.0, identifiers are arbitrary strings. For existing identifiers, these strings were set to the pre-8.1.0 FQNs. Over time, all existing identifiers are planned to be migrated to more intuitive strings.

The following table lists some identifiers that may be useful for debugging:

Table 4.1. Useful Verbose Logging Identifiers

Identifier	Description
NetworkRequests	outgoing and received network requests and responses, and possibly other related operations (e.g. conversions)

5. Debugging

5.1. Tips, Tricks, and Good Practices

Here, we document tips, tricks, and good practices for debugging RCE. Since RCE is written in Java, many of the standard good practices for debugging Java code apply to debugging RCE as well, such as getting comfortable with the debugger integrated in Eclipse. As there are good general-purpose tutorials out there, our aim in this section is not to repeat already existing general advice. Instead, we discuss tools and techniques that experience shows are not as well known.

5.1.1. Finding Resource Leaks in the UI using S-Leak

Recall that SWT resources (such as `Composite`, `Button`, or `Label`) are not cleaned up completely by the garbage collector. Instead, they must be manually cleaned up by the developer via their `dispose` method (cf. this article [https://www.eclipse.org/articles/swt-design-2/swt-design-2.html] for more information on managing SWT objects). Since this behavior differs from virtually all other Java objects, it is easy for developers to overlook this cleanup. This leads to "leaked" SWT resources, i.e., to resources that are created, but never disposed of afterwards.

S-Leak is a tool that aids in finding leaked SWT resources. It allows developers to take snapshots of the currently allocated SWT resources and to create the diff between two such snapshots. Thus, the developer can determine which resources have been allocated during certain user actions and compare these allocations with their expectations.

In order to use S-Leak, you have to start RCE with the system property `rce.debug.sleak` defined. When starting RCE via Eclipse, you can do so by editing your preferred run configuration (Run `->` Run Configurations `->` Arguments `->` VM Arguments `->` Add `-Drce.debug.sleak`). When starting RCE as a standalone application, edit the file `rce.ini` to include the line `-Drce.debug.sleak` at some point after `-vmargs`. The class `de.rcenvironment.core.start.gui.GUIInstanceRunner` interprets this system property and starts S-Leak during startup of RCE.

5.1.2. Profiling RCE using VisualVM

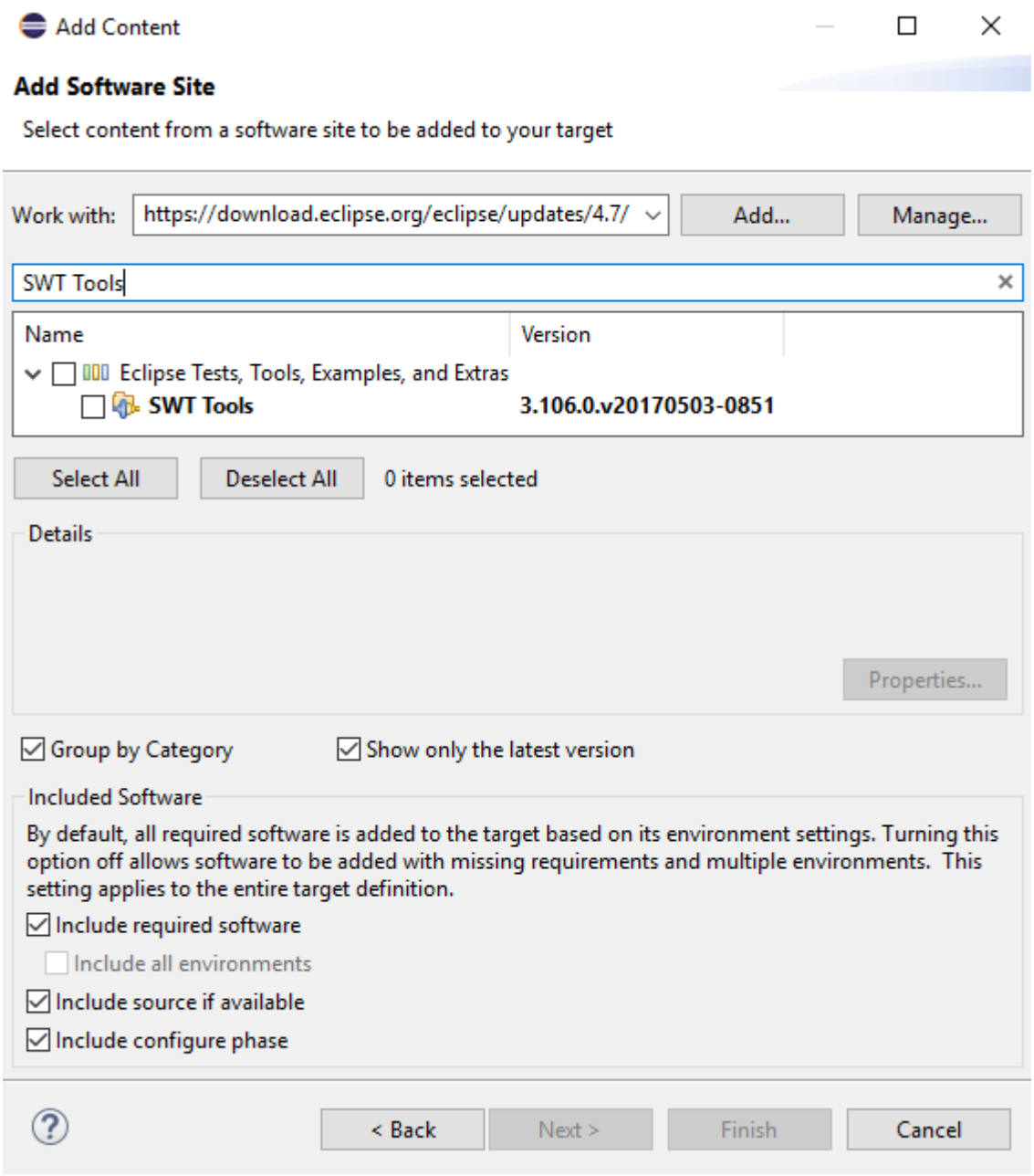
5.1.3. Debugging GUI layouts using SWT Spy

When developing GUIs using Eclipse SWT, one often wants to have more information about how the GUI of the running software is composed and what properties are set. One of the tools giving such an overview is SWT spy (https://eclipsesource.com/de/blogs/2018/01/10/swt-spy-debugging-swt-layouts/). In order to use this tool to debug the GUI of RCE started from Eclipse, follow these steps:

1. Add the plug-in "SWT Tools" available from https://download.eclipse.org/eclipse/updates/4.7/ to the target platform (`de.rcenvironment/eclipse/tp/remote/default_release_or_snapshot.target` `->` Add `->` Software Site `->` Work with https://download.eclipse.org/eclipse/updates/4.7/ `->` Check the plug-in "SWT Tools", located in the folder "Eclipse Tests, Tools, Examples and Extras" `->` Finish). We show the wizard for adding this plug-in below.

2. Reload the target platform.

3. Add the newly available feature `org.eclipse.swt.tools.feature` to the product definition of RCE (`de.rcenvironment/maven/modules/repository.mainProduct/rce_default.product` -> Contents -> Add).



The wizard for adding new plug-ins to the target platform. Select the plug-in "SWT Tools" in order to add SWT spy to RCE.

After rebuilding and restarting RCE from Eclipse, you should now have the view "SWT Spy" available in RCE. When clicking the spy-icon in its upper menu bar, this view will show information about the SWT construct that the cursor is currently pointing at. Using the shortcut `Ctrl+Alt+Shift+.` you can toggle the SWT spy to keep the information about the currently selected element in the view of the SWT spy to allow for further investigation about the information.

6. Quality Assurance and Testing

6.1. Automated GUI Testing

6.1.1. Getting started

This section describes the required steps to run existing or create new automated GUI tests for RCE using the RCP Testing Tools (RCPTT):

- Download RCPTT from https://eclipse.org/rcptt/download/
- Download RCE from https://rcenvironment.de/pages/download.html
- Start RCPTT
- In the view "Applications" add RCE as Application under test (AUT) via "New... ". Use RCE's main folder as location.
- Configure AUT (Rightclick on entry in "Applications" view -> Configure... -> Advanced...):

1. **Set a profile** different from the default one to make sure your productive RCEs won't interfere with RCPTT's RCE and vice versa. Therefore go to "Arguments" tab and add to Program arguments e.g "-p my_rcptt_profile".
2. **Set dev_config:** Download from dev_config.ini from https://svn.sistec.dlr.de/svn/rce/new/rce/trunk/de.rcenvironment/eclipse/launch/installation_data/ [https://svn.sistec.dlr.de/svn/rce/new/rce/trunk/de.rcenvironment/eclipse/launch/installation_data/dev_config.ini]

   In "Configuration" tab select "Use an existing config.ini file as a template" and navigate to the file you just downloaded.
3. **Set launcher:** In "Arguments" tab add the following VM argument: "-Dde.rcenvironment.launcher=de.rcenvironment.launcher"
4. **Set allocate console for StdIn and StdOut:** To access StdIn and StdOut directly from console view in RCPTT go to "Common" tab and check "Allocate console (necessary for input)

- Check if RCE can be started from RCPTT by double clicking on the entry in the "Applications" view
- Now you can either run existing test cases (A) or create your own test cases (B):

A: Run existing test cases:

- Create an **RCP Testing Tool Project** in the Test Explorer on the lefthand side
- Checkout the following folder and add it to the project: https://svn.sistec.dlr.de/svn/rce/new/rce/trunk/de.rcenvironment/eclipse/ui-testing/RCPTT
- In the folder "Testsuites" navigate to "AllPlatforms" and execute the testsuite (Rightclick -> Run As -> Test Cases)
- Find the Execution View on the bottom left which shows the progress of the testcases
- Do the same for the testsuite that matches your platform.

B: Create your own test cases:

- For the next steps also refer to RCPTT's getting started guide: https://www.eclipse.org/rcptt/documentation/userguide/getstarted/
- Create an **RCP Testing Tool Project** in the Test Explorer on the lefthand side
- Create a **Test Case** within this project
- **Capture script** via "Record" button in the upper right corner and then clicking around in RCE
- **Capture verification** by switching to Assertion Mode in the menu bar of the Control Panel and then selecting some element in RCE
- Click save, stopp and then the Return to RCPTT/Home button
- Create a **Context** within the project (New -> Context). Contexts are proconditions for tests, e.g. to make sure there is a clean setup. For instance, choose a context of type "Workspace" an leave the default settings so it will clear the workspace upon being applied.
- Add the context to the test case by selecting the "Contexts" tab in the test case, click "+" and choose the one you just created.
- **Replay script** by clicking "Replay" in the upper right corner.

6.2. Integrated Test Script Runner

The integrated "Test Script Runner" (TSR) was created to automate test sequences on one or more RCE standalone installations. These installations are automatically set up using an RCE feature called "Instance Management" (IM), which is still under development, and therefore not fully documented yet. However, the instructions below should be sufficient to configure this feature as needed for the TSR.

As of RCE 9.0.0, the Test Script Runner is included in the standard application release, as well as the standard Eclipse checkout. Therefore, very little configuration is required to use it.

6.2.1. Configuration

The only configuration that is required for using the TSR is adding an Instance Management configuration to the RCE instance that will execute the test scripts. This defines the root directory where work files and directories of managed RCE test installations will be stored. Locate the profile directory that is being launched and edit its `configuration.json` file. In this file, add this configuration block on the root JSON level, and adjust the settings as necessary:

```
"instanceManagement": {
  "dataRootDirectory": "/tmp/rce-im-dat",
  "installationsRootDirectory": "/tmp/rce-im-inst"
}
```

There are additional Instance Management configuration parameters available. These are, however not usually needed for using the TSR.

Note

It is *strongly* recommended to use a short full filesystem path as the `dataRootDirectory` and the `installationsRootDirectory`, as RCE installations will be placed inside of it, and long filesystem paths

are known to cause problems with these. The maximum lenght will be determined, documented, and maybe also automatically checked in the future.

6.2.2. Test Definitions

We use the language Gherkin [https://cucumber.io/docs/gherkin/], a structured natural language, to define our tests and execute them using Cucumber [https://cucumber.io/]. In the nomenclature of Gherkin, a single test is called a *scenario*, while multiple scenarios make up a *feature*. The tests are defined in `*.feature` files in the directory `/de.rcenvironment.supplemental.testscriptrunner.scripts/resources/scripts`.

Note

If you would like to edit `.feature`-files directly from Eclipse, we recommend using the *Cucumber Eclipse Plugin*, which can be found in the Eclipse Marketplace.

We show one test (or scenario) below:

```
@Start01
@DefaultTestSuite
Scenario: Concurrent headless instance starting and stopping

  Given instances "Node1, Node2, Node3, Node4, Node5" using the default build
  When  starting all instances concurrently
  Then  instances "Node1, Node2, Node3, Node4, Node5" should be running

  When  stopping all instances concurrently
  Then  the log output of all instances should indicate a clean shutdown with no warnings or errors
```

Each test has one or more *test IDs* denoted by the annotation-like tags above the indivual test scenarios, e.g. `@Start01` or `@DefaultTestSuite`. These IDs serve to later refer to that test for execution. The same ID may also be assigned to multiple tests. Thus, the test script runner can, e.g., be asked to execute all tests with a test ID of `DefaultTestSuite`. The values of the test IDs can be chosen almost arbitrarily. The only reserved IDs are `@Disabled` and `@disabled`, which prevent the test from being executed at all.

In order to execute a test written in Gherkin, Cucumber requires an implementation of each individual line in the test. Each line is called a *test step*, while the code implementing the desired behavior is called a *test step definition*. In RCE, the test step definitions are location in the bundle `de.rcenvironment.supplemental.testscriptrunner` in the package `de.rcenvironment.extras.testscriptrunner.definitions.impl`. Please refer to those implementations for the canonical overview over available test steps. In the following, we list some of the more commonly used test steps. We denote placeholders using angular brackets.

Given running instance <instance id> using the default build	Starts an instance of RCE without a GUI. The instance id is used to refer to this instance in later test steps.
When executing workflow <workflow id> on node <instance id>	Executes the given workflow on the given instance and waits for the termination of that workflow. The workflow id must correspond to the basename of a workflow file in the directory `de.rcenvironment.supplemental.testscriptrunner.scripts resources/scripts/workflows`, i.e., the file name without the suffix `.wf`.
Then that workflow run should be identical to <golden master id>	Can only be used after using the test step "When executing workflow <workflow id> on node <instance id>". Asserts that the workflow execution is identical to that stored in the directory `de.rcenvironment.supplemental.testscriptrunner.scripts resources/scripts/golden_masters`. The golden

master id must correspond to the basename of a file in that directory, i.e., to the file name without the suffix `.json`. You can export a workflow execution to serve as a golden master via the command `tc export_wf_run` in a running RCE instance.

6.2.3. Executing Tests

The TSR is invoked by a single RCE console command (`run-test`), with an alias for readability (`run-tests`). The general syntax is:

```
run-test[s] [--format pretty|json] <comma-separated list of test ids>|--all <build id>
```

By default, the result of the test is printed in human-readable format. If you would like output in the JSON-format instead, you may use the option `--format`, which requires either `pretty` or `json` as its only parameter.

There are three typical scenarios for calling this command:

- from within an RCE instance launched from Eclipse during development, usually using the GUI workflow console
- from within a standalone RCE instance, also usually using the GUI workflow console
- as a CLI batch run (`rce --batch "..."`) using a standalone instance.

The RCE installation to be tested is defined by the *<build id>* parameter in the above command. One important aspect to understand is that this installation is generally independent of the installation being used to execute the TSR command. The latter is, in a sense, only the "host" of test scripts. There are three ways of specifying the build to test:

1. A build download id, which corresponds to a certain part of the standard download URL, for example `snapshots/trunk` or `releases/9.0.0`. The structure should be self-explanatory. (The major release tree to use for snapshot builds is one of the optional Instance Management settings mentioned above; the default is to use the current major version, ie 9.x.)
2. A path to an *unpacked* local standalone (product) build, which can, for example, result from a local build run or from unpacking a downloaded product zip file. The syntax for this is `local:`*<local installation path>*. This directory can be either writable or read-only. For example, it is also possible to test a (read-only) .deb or .rpm package installation this way.

   Note that this path must point to an already-unpacked RCE build, unlike the first approach, which downloads zipped release packages and unpacks them automatically.
3. As it is a frequent use case when testing standalone builds to execute the test command the installation itself, there is a convenient shortcut for this. By specifying `:self` as the build id, the test scripts are executed on the installation of the instance used to run the test command.

   Note that due to technical limitations, however, this shortcut is not possible when launching RCE from Eclipse, as the test scripts require a standard product build to execute.

Recall that each test has one or more test ids, denoted by annotation-like tags. These test ids can be specified in the command `run-test[s]` with or without the `@` character. "`--all`" executes all available test scenarios.

6.2.4. Examples

- `run-test Test02,Test04 snapshots/trunk` - runs two specific tests on the latest snapshot build
- `run-test --format pretty Test02,Test04 snapshots/trunk` - equivalent to the command above
- `run-test --format json Test02,Test04 snapshots/trunk` - runs two specific tests on the latest snapshot build and outputs the result in JSON format
- `run-test DefaultTestSuite :self` - runs the default collection of tests on the current installation
- `run-test --all local:/tmp/local-rce-build` - runs all available tests on a local build
- `rce --batch "run-test DefaultTestSuite :self"` - the full command line for the standard self-test of an installation

7. Command Console

7.1. RCE’s Command Console

RCE features a command console.
This command console is accessible via the command console view in RCE’s desktop mode or via SSH. Commands consist of one or two tokens e.g. "im" for one token and "im list" for two tokens. Beside the user commands that are documented in RCE’s User Guide, there are developer commands supporting additional developer features. A list of all developer commands is shown in the following section.

7.1.1. Commands

Commands are structured in groups. A command group is defined by the first token of a command. This means, that commands like "im info" and "im list" are in the same command group. Some command groups like "stats" consist only of a single command. For each group there is a table explaining all the related commands.

The 'dev' command

Command	Description
dev <command group> [--details|-d] [--asciidoc]	Alias of "help --dev". *<command group>*: (optional) the command group of which the commands should be shown *[--details|-d]*: show details of the commands *[--asciidoc]*: output in asciidoc format

The 'dm' command

Command	Description
dm create-test-data [--prefix <prefix>] [--workflows <workflow number>] [--iterations <iterations number>] [--allowedDeviation <deviation>] [--smallfiles|-s] [--bigfiles|-b]	Creates test data in the database. *[--prefix <prefix>]*: prefix for created workflows *[--workflows <workflow number>]*: name for created workflows *[--iterations <iterations number>]*: number of iterations *[--allowedDeviation <deviation>]*: allowed deviation in %, standard value is 20% *[--smallfiles|-s]*: small input files for components *[--bigfiles|-b]*: big input files for components

The 'dummy' command

Command	Description
dummy	Prints a test message.

The 'force-crash' command

Command	Description
force-crash <delay>	"kills" the instance without proper shutdown at <delay> milliseconds after the command is executed. *<delay>*: delay in milliseconds

The 'im' command

Command	Description
im configure <instances…> [--set-rce-version <version>] [--apply-template <template name>] [--set-name <name>] [--set-comment <comment>] [--set-relay-option <(is relay)>] [--set-workflow-host-option <(is worflow host)>] [--set-custom-node-id <node id>] [--set-tempdir-path <tempdir>] [--add-server-port <id> <host> <port>] [--add-connection <id> <host> <port> <auto-connect>] [--remove-connection <id>] [--configure-ssh-server <ip> <port>] [--add-ssh-account <username> <role> <enable ssh-account> <password>] [--remove-ssh-account <username>] [--set-ip-filter-option <(has ip-filter)>] [--enable-im-ssh-access <port>] [--set-request-timeout <r timeout>] [--set-forwarding-timeout <f timeout>] [--add-allowed-inbound-ip <ip>] [--remove-allowed-inbound-ip <ip>] [--add-ssh-connection <name> <display name> <host> <port> <login name>] [--remove-ssh-connection <name>] [--add-uplink-connection <ip> <hostname> <port> <client id> <is gateway> <connect on startup> <auto-retry> <username> <passwrod keyword> <password>] [--remove-uplink-connection <id>] [--publish-component <name>] [--unpublish-component <name>] [--set-background-monitoring <id> <interval>] [--reset] [--wipe] [--disable-ssh-server]	Configures the configuration.json file of the specified RCE instance(s). *<instances…>*: list of instances to manage *[--set-rce-version <version>]*: sets the rce version of the instances. (Does not work on existing instances.) *[--apply-template <template name>]*: applies (i.e. copies) the given template as the new configuration *[--set-name <name>]*: sets the name of the instance *[--set-comment <comment>]*: sets a general comment *[--set-relay-option <(is relay)>]*: sets or clears the relay flag *[--set-workflow-host-option <(is worflow host)>]*: sets or clears the workflow host flag *[--set-custom-node-id <node id>]*: adds an override value for the node's network id; use with caution! *[--set-tempdir-path <tempdir>]*: sets the root path for RCE's temporary files directory *[--add-server-port <id> <host> <port>]*: adds a new server port and sets the ip and port number to bind to *[--add-connection <id> <host> <port> <auto-connect>]*: adds new connection to the given ip/hostname and port, and whether it should auto-connect *[--remove-connection <id>]*: removes a connection *[--configure-ssh-server <ip> <port>]*: enables the ssh server and sets the ip and port to bind to *[--add-ssh-account <username> <role> <enable ssh-account> <password>]*: adds an SSH account *[--remove-ssh-account <username>]*: removes an SSH account *[--set-ip-filter-option <(has ip-filter)>]*: enables or disables the ip filter; default: true *[--enable-im-ssh-access <port>]*: enables and configures SSH forwarding of RCE console commands by the IM "master" instance *[--set-request-timeout <r timeout>]*: sets the request timeout in msec *[--set-forwarding-timeout <f timeout>]*: sets the forwarding timeout in msec

Command	**Description**
	[--add-allowed-inbound-ip <ip>]: adds/allows an inbound IP address to the filter *[--remove-allowed-inbound-ip <ip>]*: removes/ disallows an inbound IP address from the filter *[--add-ssh-connection <name> <display name> <host> <port> <login name>]*: adds a new ssh connection *[--remove-ssh-connection <name>]*: removes a ssh connection *[--add-uplink-connection <ip> <hostname> <port> <client id> <is gateway> <connect on startup> <auto-retry> <username> <passwrod keyword> <password>]*: adds a new uplink connection *[--remove-uplink-connection <id>]*: removes an uplink connection *[--publish-component <name>]*: publishes a new component *[--unpublish-component <name>]*: unpublishes a component *[--set-background-monitoring <id> <interval>]*: enables background monitoring with the given interval (in seconds) *[--reset]*: resets the instance to an empty configuration *[--wipe]*: wipes the instance *[--disable-ssh-server]*: disables the ssh server
im dispose <instances…>	Disposes the specified instance meaning deletion of the profile directory. <*instances…*>: list of instances to manage
im info	Shows additional information.
im install <install policy: if-missing|force-download|force-reinstall> <major version> <installation id> [--timeout <timeout>]	Downloads and installs a new RCE installation. <*install policy: if-missing|force-download|force-reinstall*>: specify install policy <*major version*>: syntax format: [<major version>]/<url version id/part> <*installation id*>: id for the installation *[--timeout <timeout>]*: specifies the maximum length of time this command is allowed to run (in seconds) - default = 60s
im list <affected items: all|instances|installations| templates>	Lists information about instances, installations or templates. <*affected items: all|instances|installations| templates*>: specify group of affected items
im reinstall <install policy: if-missing|force-download|force-reinstall> <major version> <installation id>	Stops all instances running the given installation id, downloads and installs the new RCEinstallation, and starts the instances again with the new installation. <*install policy: if-missing|force-download|force-reinstall*>: specify install policy

Command	Description
	<*major version*>: syntax format: [<major version>]/<url version id/part> <*installation id*>: id for the installation
im restart <instances…> <installation id> [--timeout <timeout>] [--gui <start with gui>] [--command-arguments <command arguments>]	Restarts a list of RCE instances with the given instance IDs and the given installation. <*instances…*>: list of instances to manage <*installation id*>: id for the installation *[--timeout <timeout>]*: specifies the maximum length of time this command is allowed to run (in seconds) - default = 60s *[--gui <start with gui>]*: option to start with gui - standard <false> *[--command-arguments <command arguments>]*: additional command arguments
im start <instances…> <installation id> [--timeout <timeout>] [--gui <start with gui>] [--command-arguments <command arguments>]	Starts a list of new RCE instances with the desired instance IDs and the desired installation. <*instances…*>: list of instances to manage <*installation id*>: id for the installation *[--timeout <timeout>]*: specifies the maximum length of time this command is allowed to run (in seconds) - default = 60s *[--gui <start with gui>]*: option to start with gui - standard <false> *[--command-arguments <command arguments>]*: additional command arguments
im start-all <installation id> [--timeout <timeout>] [--command-arguments <command arguments>]	Starts all available instances. Uses the given installation. <*installation id*>: id for the installation *[--timeout <timeout>]*: specifies the maximum length of time this command is allowed to run (in seconds) - default = 60s *[--command-arguments <command arguments>]*: additional command arguments
im stop <instances…> [--timeout <timeout>]	Stops a list of running RCE instances. <*instances…*>: list of instances to manage *[--timeout <timeout>]*: specifies the maximum length of time this command is allowed to run (in seconds) - default = 60s
im stop-all <installation id> [--timeout <timeout>]	Stops all running instances. <*installation id*>: id for the installation *[--timeout <timeout>]*: specifies the maximum length of time this command is allowed to run (in seconds) - default = 60s

The 'net' command

Command	Description
net bench <taskdef>	Run communication benchmark.

Command	Description
	<*taskdef*>: <targetNode|*>([<numMessages>], [<requestSize>],[<responseSize>], [<responseDelay(msec)>],[<threadsPerTarget>])
net graph <base name> [--all|-a]	Generates a Graphviz file of the current network topology. <*base name*>: base name parameter *[--all|-a]*: include unreachable nodes
net np	Show known RCE node properties.

The 'osgi' command

Command	Description
osgi <command> [-o <filename>]	Executes an OSGi/Equinox console command; use -o to write text output to a file. <*command*>: osgi command *[-o <filename>]*: text output to a file

The 'ra' command

Command	Description
ra cancel <session token>	Cancels a session. <*session token*>: token of the session
ra describe-tool <tool id> <tool version> [-n <tool node id>] [--template|-t]	Prints names and data types of the tool's or workflow's intputs and outputs. <*tool id*>: id of the tool <*tool version*>: version of the tool *[-n <tool node id>]*: set tool node id *[--template|-t]*: make template
ra describe-wf <workflow id> <workflow version>	Prints names and data types of the tool's or workflow's intputs and outputs. <*workflow id*>: id of the workflow <*workflow version*>: version of the workflow
ra dispose <session token>	Releases resources used by a remote access session. <*session token*>: token of the session
ra get-doc-list <tool id> <tool version>	Get tool documentation list. <*tool id*>: id of the tool <*tool version*>: version of the tool
ra get-tool-doc <session token> <tool id> <tool version> <tool node id> <hash value>	Download documentation. <*session token*>: token of the session <*tool id*>: id of the tool <*tool version*>: version of the tool <*tool node id*>: id of the tool node <*hash value*>: hash value
ra init [--compact|-c]	Initializes a remote access session, and returns a session token.

Command	Description
	[--compact|-c]: compact output
ra list-tools [-f <format>] [--with-load-data <time span> <time limit>]	Lists all available tool ids and versions for the "run-tool"command. *[-f <format>]*: output format *[--with-load-data <time span> <time limit>]*: load data
ra list-wfs [-f <format>]	Lists all available workflow ids and versions for the "list-wfs"command. *[-f <format>]*: output format
ra protocol-version	Prints the protocol version of this interface.
ra run-tool <session token> <tool id> <tool version> [-n <tool node id>] [-p <parameters…>] [--dynInputs <dyn inputs>] [--dynOutputs <dyn outputs>] [--nonReqInputs <non req inputs>] [--show-output|-o] [uncompressed upload|-u] [simple description format|--simple]	Invokes a tool by its id and version. <*session token*>: token of the session <*tool id*>: id of the tool <*tool version*>: version of the tool *[-n <tool node id>]*: set tool node id *[-p <parameters…>]*: additional parameters *[--dynInputs <dyn inputs>]*: dynamic inputs *[--dynOutputs <dyn outputs>]*: dynamic outputs *[--nonReqInputs <non req inputs>]*: non required inputs *[--show-output|-o]*: shows output *[uncompressed upload|-u]*: This is a command flag *[simple description format|--simple]*: This is a command flag
ra run-wf <session token> <workflow id> <workflow version> [-p <parameters…>] [--show-output|-o]	Invokes a published workflow by its id. <*session token*>: token of the session <*workflow id*>: id of the workflow <*workflow version*>: version of the workflow *[-p <parameters…>]*: additional parameters *[--show-output|-o]*: shows output

The 'run-test' command

Command	Description
run-test <tag name filters…> <build under test id> [--format <pretty|json>]	Run a test. <*tag name filters…*>: filter for tag names <*build under test id*>: build version to be used *[--format <pretty|json>]*: output format

The 'run-tests' command

Command	Description
run-tests <tag name filters…> <build under test id> [--format <pretty|json>]	(alias of "run-test"). <*tag name filters…*>: filter for tag names <*build under test id*>: build version to be used *[--format <pretty|json>]*: output format

The 'stats' command

Command	Description
stats	Show internal statistics.

The 'tasks' command

Command	Description
tasks [--all|-a] [--unique|-i]	Show information about internal tasks. *[--all|-a]*: Show all tasks, including inactive ones *[--unique|-i]*: Extended information: list tasks with a unique id

The 'tc' command

Command	Description
tc close_view <close view parameter>	Closes a GUI view. <*close view parameter*>: one of the following view keys is a valid input: [Workflow_List, Log, Optimizer, Data_Management_Browser, CPACS_Writer, Properties, Component_Publishing, Cluster_Job_Monitor, TIGL_Viewer, Workflow_Console, Timeline, Excel, Parametric_Study, Command_Console, Network]
tc close_welcome	Closes the welcome screen if present.
tc compare_wf_runs <abs export workflow path> <abs export workflow path>	Compares the two given workflowruns and indicates if they are identical or wether their are differences. <*abs export workflow path*>: absolute path to exported workflowrun <*abs export workflow path*>: absolute path to exported workflowrun
tc export_all_wf_runs <abs path to export dir>	Exports all workflow runs into the given export directory. <*abs path to export dir*>: absolute path to export directory
tc export_wf_run <absolute directory path> <workflow title>	Exports the run corresponding to the workflowtitle to the given directory. <*absolute directory path*>: absolute path to directory <*workflow title*>: title of the workflow
tc open_view <open view parameter>	Opens a GUI view and sets focus to it. <*open view parameter*>: one of the following view keys is a valid input: [Workflow_List, Log, Optimizer, Data_Management_Browser, CPACS_Writer, Properties, Component_Publishing, Cluster_Job_Monitor, TIGL_Viewer, Workflow_Console, Timeline, Excel,

Command	Description
	Parametric_Study, Command_Console, Network]

The 'wf' command

Command	Description
wf check-self-test-cases	Check if all test workflows are part of at least one test case.
wf graph <workflow file>	Prints .dot string representation of a workflow (can be used to create graph visualization with Graphviz). *<workflow file>*: path to the workflow file
wf list-self-test-cases	List available test cases for wf self-test.
wf self-test [--dispose <onfinished|never|always>] [--delete <onfinished|never|always>] [--pr <parallel runs>] [--sr <sequential runs>] [--python <python exe path>] [--cases <cases…>]	Batch test workflow files of the test workflow files bundle. *[--dispose <onfinished|never|always>]*: dispose behaviour *[--delete <onfinished|never|always>]*: deletion behaviour *[--pr <parallel runs>]*: number of parallel runs *[--sr <sequential runs>]*: number of sequential runs *[--python <python exe path>]*: path to python.exe *[--cases <cases…>]*: parameter for the cases

7.1.2. Parameters

RCE commands can use three different types of parameters. These are positional parameters, named parameters and flags. Positional and named parameters take at least one value. However, some of them also take a list parameters. List parameters are denoted by three dots after their name in the tables above, while parameters that take only a single value do not have it, e.g. **<parameter…>**. Multiple values for list parameters are entered using comma separation, e.g. **value0, value1, value2**.

Positional

Positional parameters must be entered immediately after a command. The order of the parameters must be followed for the command to process the parameters correctly. After all positional parameters have been entered, other types of parameters can be entered.

Named

Named parameters are optional parameters. The command input of these parameters always start with a double dash, e.g. **--name**. The value must be specified after the parameter name.

Flag

Command flags are used to modify the operation of a command. They are always optional. Flags start with a dash or a double dash. A single dash is used for flags consisting of only a single character, e.g. **-a**. Double dash is used for flags composed of more than one letter, e.g. **--all**. This long form is more descriptive and is therefore easier to understand. Command flags can have both long and short form, e.g. **-a** and **--all**, which do not differ in execution.

7.1.3. Documentation generation

Part of this documentation is automatically generated. This is done by executing the **help --asciidoc** command in RCE. The output of this command has to be copied to the file **de.rcenvironment.documentation.core/src/asciidoc/generated/user-commands.ad**. This can be done by using the **saveto** command.

8. Licensing and Copyright

8.1. Copyright Statements

8.1.1. Current Year Definition

In copyright statements, the current year is defined. Each year, it must be updated at the following places (files):

- Header in Java source files
- feature.xml files
- About dialog (de.rcenvironment.core.start/about.mappings)
- Splash images ([..].gui.branding.[..]/splash.bmp)