EKumi Documentation

Emmanuel CHEBBI

Feb 08, 2020

Getting Started

EKumi is a workflow management system licensed under the Eclipse Public License 2.0.

EKumi allows to automate the execution of chains of tasks. Its bigger strength is its extensibility as it allows anyone to provide and share new:

- workflow editors,
- scripting languages,
- datatypes,
- execution hooks.

Please follow **Getting Started** chapters to learn how to create and execute a workflow using EKumi's built-in features.

CHAPTER 1

Table of Contents

1.1 Main Concepts

The following glossary sums up EKumi's main concepts and define the terms used throughout this documentation.

Task A single unit of work. A task takes inputs, performs some operation and then produces outputs. A task may neither take any input nor produce any output.

Activity A collection of tasks, which is also considered as a task. The terms "workflow" and "activity" are equivalent.

Input A typed value that is given to a task before it is executed.

Output A typed value that is produced by the execution of a task.

Datatype A type that determines the values that can be associated to an input or an output.

1.2 Install EKumi

Todo: No download available at the moment, please wait for the first release.

EKumi can be installed in to different ways:

- either on the top of an existing Eclipse IDE installation
- or as a standalone product.

Depending on your needs you may choose one or the other.

1.2.1 On top of an existing Eclipse IDE installation

EKumi can be installed as a set of plug-ins directly within Eclipse IDE.

To this end, open the IDE, click on `Help` > `Install new software...` then paste the following URL in the dialog:

- <URL not available yet>

Check the following features:

- EKumi IDE Integration
- EKumi IDE UI Integration
- EKumi Java Scripting Language

Click on `Finish`, accept the licenses then `Finish`.

Wait for the installation to end then restart the IDE.

1.2.2 As a standalone product

To use EKumi as a standalone product you have to download the archive corresponding to your OS at the following address:

- <URL not available yet>

You can then decompress it, and run the `ekumi` executable.

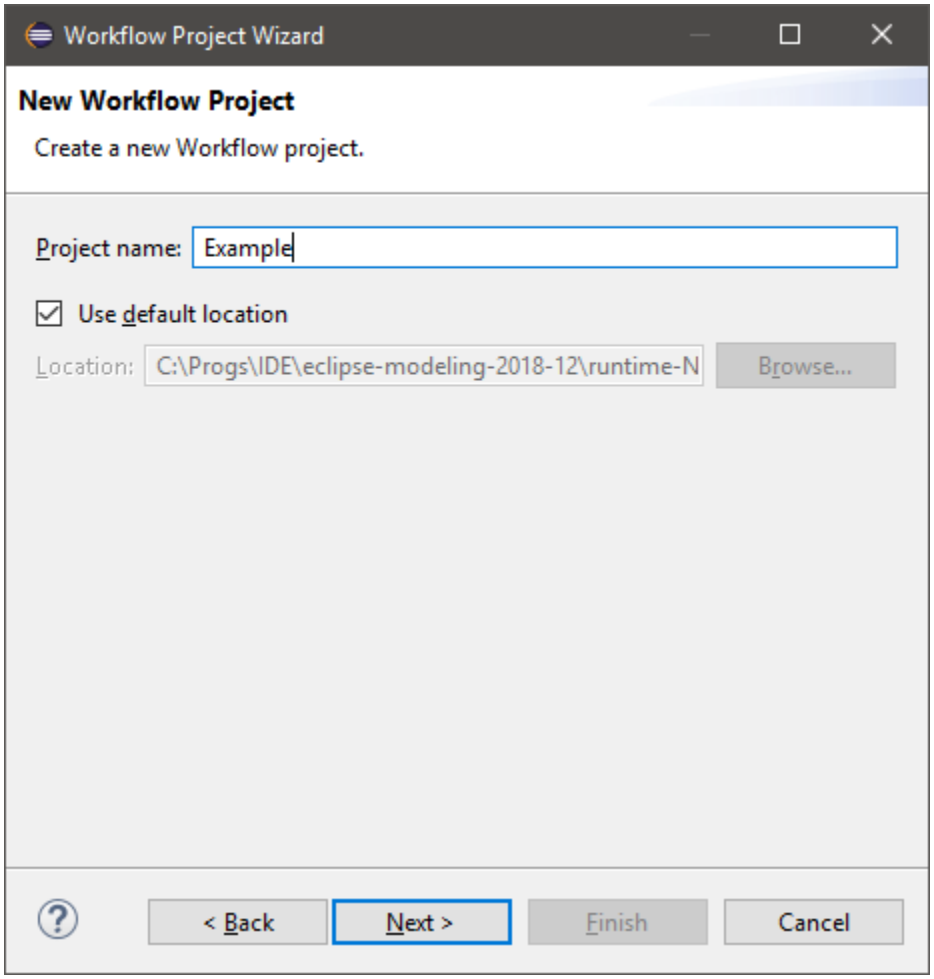
You are now ready to create a first workflow project.

1.3 Create a new Workflow project

1.3.1 Use the *New Workflow Project* wizard

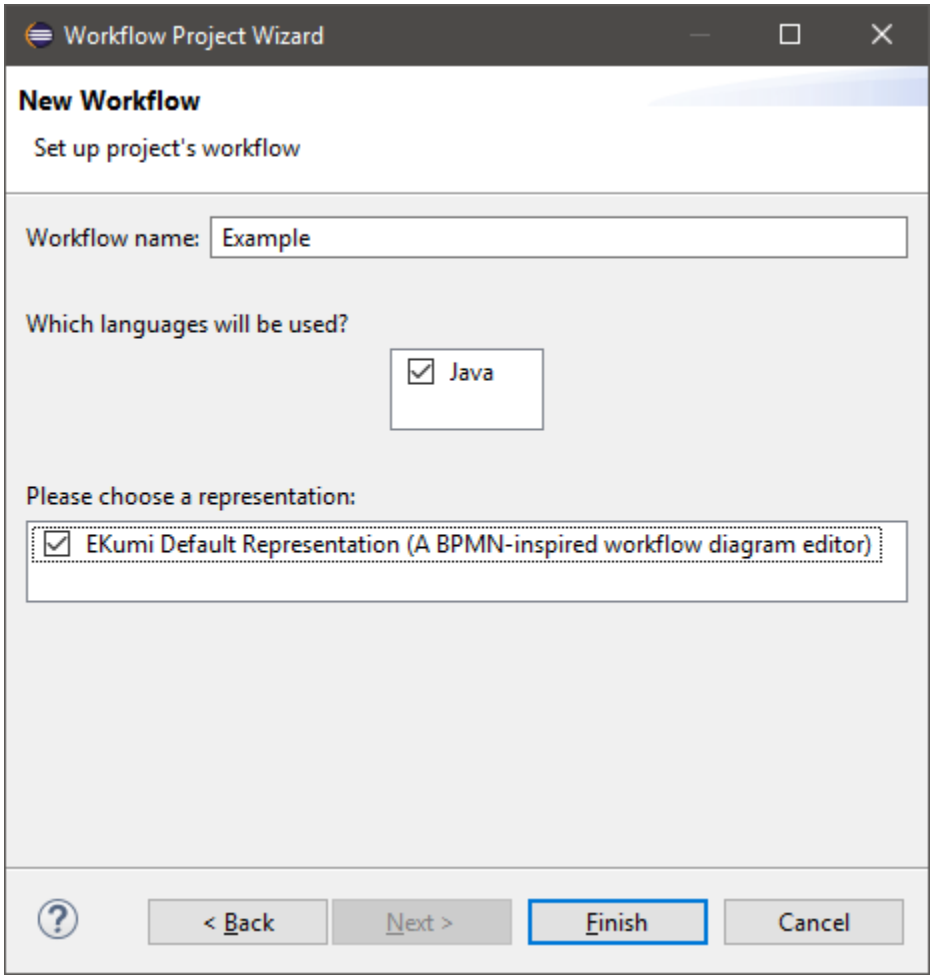
The simplest way to create a new Workflow project is to use the dedicated wizard. It can be opened from `File` > `New...` > `Workflow Project`.

The wizard first asks for the name of the new project. Fill in the text field then press `Next`.



The second page asks the user to choose:

- the name of the workflow (which, by default, is the same as the name of the project),
- the scripting languages enabled for this workflow,
- the representation.



1.3.2 Enable scripting languages

Scripting languages are used to specify the behaviour of a task at runtime. A scripting language is typically kind of an interpreter able to execute a script associated with a task.

Tip: See **Available Scripting Languages** for an overview of available scripting languages.

Several scripting languages can be selected at once for a project. In the context of this tutorial just select `Java`; if Java is not available, please see *Install EKumi*.

Important: It is currently impossible to enable new scripting languages in an existing project. Take care to the languages you select during the creation of the projects.

1.3.3 Choose a representation

A representation defines the way an activity can be seen. A representation is usually associated with an editor providing tools to modify the activity.

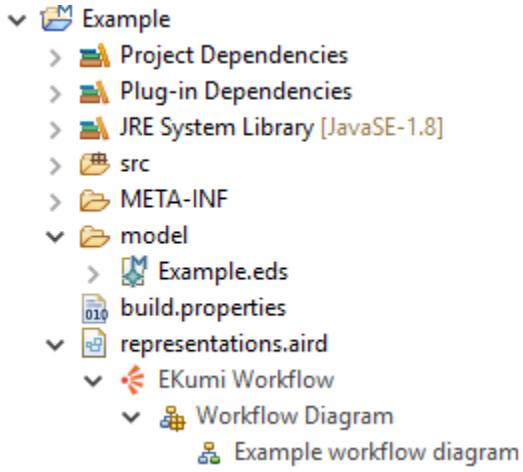
These editors can take any shape: it can be a diagram editor, a textual DSL or even a GUI with text fields and buttons. It is important to pick a representation which is relevant to your goal because it will define the way you design the activity.

Tip: See **Available Representations** for an overview of available representations.

Currently only one representation is allowed per project. Select `EKumi Default Representation`.

1.3.4 Create the project

When the setup is done, click on `Finish`. Wait a few seconds to see the project being added to the Explorer.



Tip: See the *Java* and *EKumi Default Representation* for a detailed presentation of the project's content.

You are now ready to design your first activity.

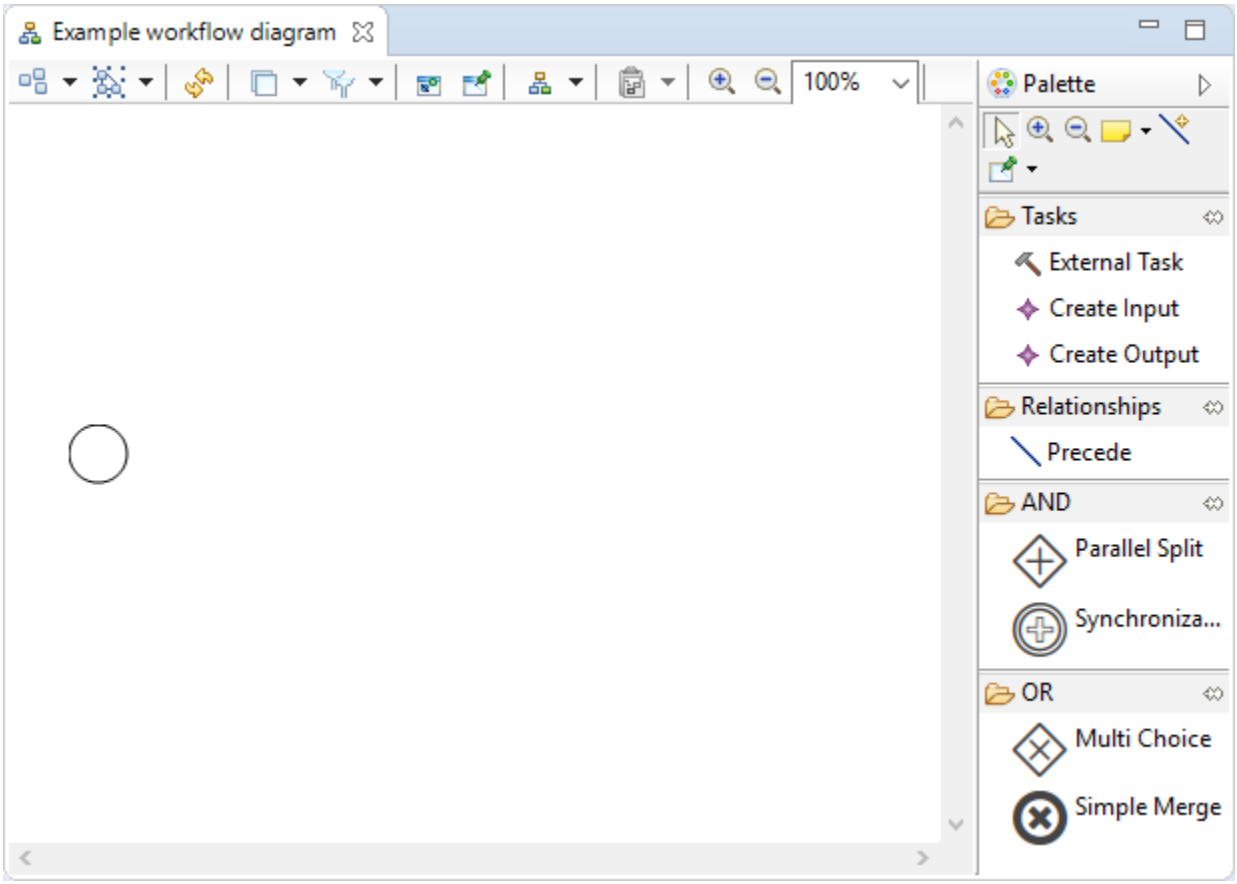
1.4 Design an activity

1.4.1 Open the diagram editor

In order to open the diagram editor:

1. Unfold the *representations.aird* file
2. Double-click on *Example workflow diagram*

The following view should open:



1.4.2 Understand the diagram editor

The workflow diagram editor is made of two parts:

- the edition area, which is the blank area on the left,
- the palette, which is the section on the right.

The edition area provides a visual representation of the workflow. Tools can be applied on it in order to modify the representation.

The circle represents the start node, which is the entry point of the workflow when it is executed.

The palette provides access to the different tools that can be used to modify the workflow. A tool can be used by:

1. Clicking on the tool in the palette
2. Clicking on the edition area

Tip: See *EKumi Default Representation* for an in-depth presentation of available tools.

1.4.3 Create a Greeting task

A new Task can be created thanks to the External Task tool:

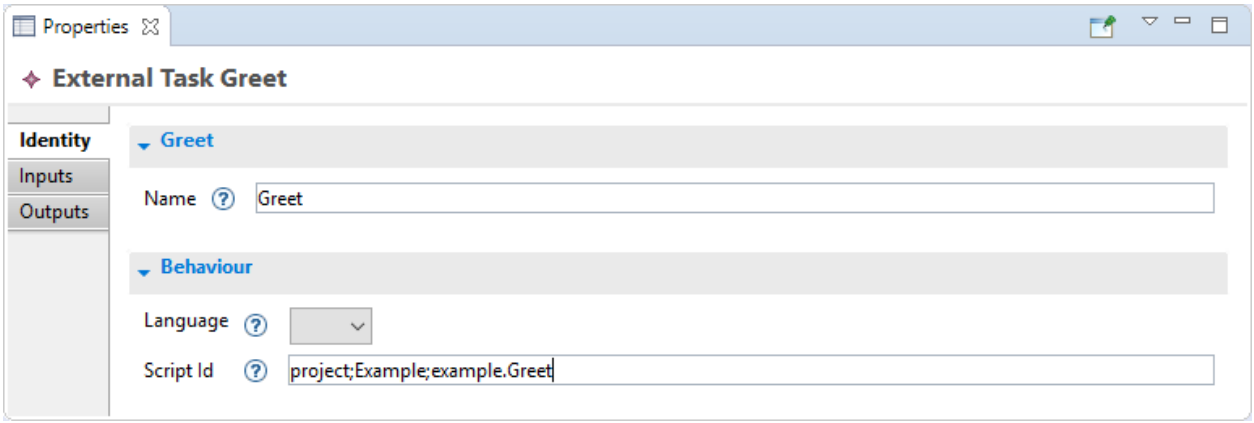
1. Click on the tool
2. Click somewhere in the edition area

A new box appears on the editor, representing the new task.

Open the `Properties` view, select the task, then type “Greet” in the *Name* text field.

In order to add behavior to this task we have to link it to a Java script. To this end:

1. Select `Java` in the *Languages* drop-down menu
2. Type “project;<the_name_of_your_project>;example.Greet” in the *Script Id* text field



Note: The *Script Id* fields allows EKumi to resolve the script to run. The UI will evolve in the future so that users won’t have to type it by hand anymore.

Then create a new Java class called `Greet` in the *example* package that prints something to the console.

```
1  package example;
2
3  import fr.kazejiyu.ekumi.core.workflow.Context;
4  import fr.kazejiyu.ekumi.core.workflow.gen.impl.RunnerImpl;
5
6  public class Greet extends RunnerImpl {
7
8          @Override
9          public void run(Context context) {
10                 System.out.println("Hello!");
11         }
12
13 }
```

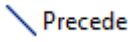
Todo: Build a more complex activity with two tasks, one producing outputs and another consuming them.

Now that the activity is ready, it can be executed.

1.5 Execute an activity

1.5.1 Link the task to the start node

First of all, the `Greet` task must be linked to the start node, otherwise it won't be considered as a runnable task.

To this end, use the 

tool:

1. Click on the tool
2. Click on the start node
3. Click on the task

1.5.2 Launch the execution

Then you can create a new Run configuration:

1. `Run` > `Run Configurations...`
2. Double-click on *Workflow*
3. Click on *Browse*
4. Select *model/Example.eds*
5. Click on *Run*

Hello! should be printed to the console.

1.6 Good Practices

1.6.1 Use meaningful names

When naming a task it is important to use a clear and easy to understand name.

Since tasks represent computations, it is relevant to use **verbs** to name them. For instance:

- Compute X
- Merge data
- Write to file

When a task is only used to extract data, naming it after the name of the data can make its purpose clearer. For instance, a task used to compute the length of a String may be called Length.

Exceptions are tasks representing mathematical operations. In such cases it may be clearer to use the mathematical expression as name:

- a + b
- y = f(x) + 2b

1.7 EKumi Default Representation

Important: Section under construction

1.7.1 A BPMN-inspired editor

This is the default built-in representation of an activity. It provides a BPMN-inspired diagram workflow editor.

1.7.2 Use Cases

This representation should be used when a graphical representation makes easier to design an activity.

1.7.3 Features

This representation allows anyone to:

- Design an activity made of multiple tasks,
- Add inputs and outputs to a task,
- Associate a script to a task,
- Specify that several tasks must be executed concurrently.

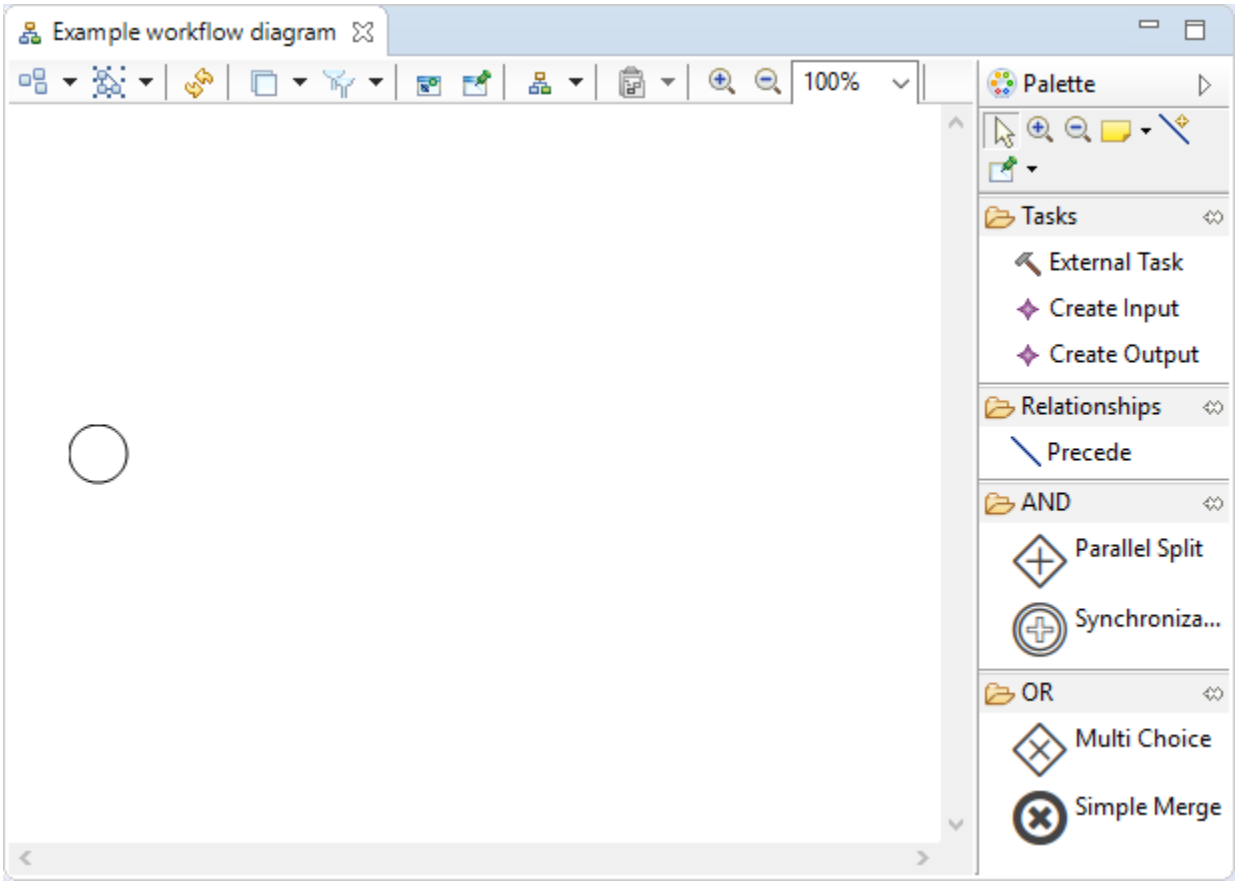
1.7.4 Impacts on project

When this representation is chosen for a project, it creates the following files:

File	Purpose
`representations.aird`	Describes the workflow diagram.
`model/<activity-name>.eds`	Describes the activity, defining the different tasks it is made of.

It also adds the `Modeling` nature to the project.

1.7.5 Understand the diagram editor



The workflow diagram editor is made of two parts:

- the edition area, which is the blank area on the left,
- the palette, which is the section on the right.

The edition area provides a visual representation of the workflow. Tools can be applied on it in order to modify the representation. The circle represents the start node, which is the entry point of the workflow when it is executed.

The palette provides access to the different tools that can be used to modify the workflow. A tool can be used by:

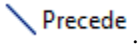
1. Clicking on the tool in the palette
2. Clicking on the edition area

Available tools are described in the following chapters.

1.7.6 Create a new task

A new Task can be created thanks to the External Task tool.

1.7.7 Link two tasks

Two task can be linked in order to specify which one should be executed first. This can be achieved thanks to the 

.

1.7.8 Launch an activity

Once the activity is ready, it can be executed. An execution can be launched in two ways.

Create a dedicated launch configuration

1. `Run` > `Run Configurations...`
2. Double-click on *Workflow*
3. Click on *Browse*
4. Select the *.eds* file located under the *model/* folder
5. Click on *Run*

Use the context menu shortcut

In the file explorer:

1. Right-click on the *.eds* file located under the *model/* folder
2. Select `Run As` > `EKumi Activity`

1.8 Java

Important: Section under construction

Currently, Java is the only available scripting language. It allows to specify the behaviour of tasks by writting Java scripts. Each script is a class that extends the `RunnerImpl` class.

1.8.1 Impacts on project

When Java is enabled on a Workflow Project, it creates the following files:

File	Purpose
`src/`	Directory containing Java source files.
`META-INF/MANIFEST.MF`	Defines project's dependencies, including EKumi's API.
`build.properties`	Defines the files to include when the project is packaged as a binary.

The `Java` and `Plugin` natures are also added to the project. That enables the Java builder to compile the sources and allows the dependencies toward EKumi API to be resolved.

1.8.2 Script implementation

A new script can be added to a task by specifying the class' canonical name as script id. The class must extend the `RunnerImpl` class as in the example below:

```
/**
 * A script that prints 'Hello' when the corresponding task is executed.
 */
public class SayHello extends RunnerImpl {

    @Override
    public void run(Context context) {
        System.out.println("Hello!");
    }

}
```

1.8.3 Dependency Injection

Java scripts can be injected with some environment objects. Currently two objects can be injected:

- `Events`: allows to send specific events and to register new listener
- `ExecutionStatus`: allows to check the current status of the execution (failed, cancelled, etc.)

```
/**
 * A script that waits until the execution is cancelled.
 */
public class WaitCancellation extends RunnerImpl {

    @Inject
    private final ExecutionStatus execution;

    @Override
    public void run(Context context) {
        while (! execution.isCancelled()) {
            // wait
        }
    }

}
```

1.9 Monitor Executions

Important: This section requires some knowledge about Eclipse Extension Points.

1.9.1 Why monitoring executions?

Monitoring executions allows developers to enhance the execution engine or to connect it to other tools. For instance, one could monitor executions in order to:

- show ongoing executions in an online dashboard,

- notify user by e-mail or Slack when an execution fails,
- update a database with the result of the execution.

Executions can be monitored by registering **execution hooks**.

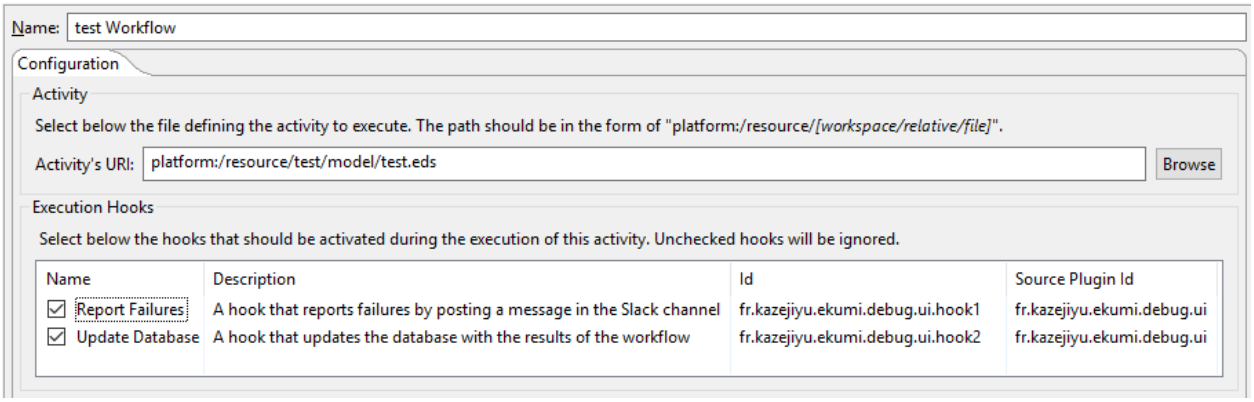
1.9.2 How to register a new execution hook?

An execution hook can be registered by contributing a new `hook` element to the `fr.kazejiyu.ekumi.core.execution` extension point.

This element needs the following attributes:

Attributes	Purpose
id	Uniquely identifies the hook
class	An instance of the `ExecutionHook` interface
name	Human-readable name displayed in UI
description	Details of what the hook does, displayed in UI
activated by default	(Optional) Indicates whether the hook should be notified by default

When the user executes an activity, he can select the hooks that should be notified. The `activated by default` attribute indicates whether the hook is selected by default. The following screenshot shows the Run Configuration tab that allows the user to select available hooks:



1.9.3 How to implement a new execution hook?

A new hook is implemented by creating a class that implements the `ExecutionHook` interface. This interface provides a lot of methods to react on specific events but their default behavior is to do nothing so you can only override the ones you are interested in.

The following class defines a hook that prints messages when an execution starts, ends, and when an activity fails:

```
1  import fr.kazejiyu.ekumi.core.execution.listeners.ExecutionHook;
2  import fr.kazejiyu.ekumi.core.workflow.Activity;
3  import fr.kazejiyu.ekumi.core.workflow.Execution;
4
5  public class PrintsExecutionStepsToConsole implements ExecutionHook {
6
7      @Override
8      public String id() {
9          // Must be equal to the id specified in the plugin.xml file
```

(continues on next page)

(continued from previous page)

```
        return "fr.kazejiyu.ekumi.ide.hooks.example1";
    }

    @Override
    public void onExecutionStarted(Execution execution) {
        System.out.println("The execution " + execution.name() + " has started on "
↪+ execution.startDate());
    }

    @Override
    public void onExecutionSucceeded(Execution succeeded) {
        System.out.println("The execution " + succeeded.name() + " has finished
↪successfully on " + succeeded.endDate());
    }

    @Override
    public void onActivityFailed(Activity failed) {
        System.out.println("The activity " + failed.name() + " has failed");
    }

}
```

1.10 Add a new Scripting Language

Important: This section requires some knowledge about Eclipse Extension Points.

1.10.1 What is a scripting language?

A scripting language is a language that can be used to specify the behaviour of a task. Concretely, a language is a parser that can:

1. Instantiate an `Activity` from a given String
2. Create a String from an existing `Activity`.

It is hence responsible of serializing and deserializing Activities so that they can both be persisted and executed.

1.10.2 How to add a new scripting language?

A new one can be defined by contributing to the `fr.kazejiyu.ekumi.core.languages` extension point.

It requires one class that implements the `ScriptingLanguage` interface.

The interface is defined as follows:

```
public interface ScriptingLanguage {

    /**
    * Returns a unique id identifying the language.
    * @return a unique id identifying the language
    */
```

(continues on next page)

(continued from previous page)

```
    String id();

    /**
    * Returns a human-readable name of the language.
    * @return a human-readable name of the language
    */
    String name();

    /**
    * Turns a runner written with the language into a EKumi {@link Runner}.
    *
    * @param identifier
    *                       Uniquely identifies the runner to resolve.
    *                       Must not be {@code null}.
    * @param context
    *                       The context of the {@link Execution}. Can be null if the execution
    *                       does not provide any context, or if the Runner is not resolved in
    *                       the context of an execution.
    *
    * @return a runner that can be handled by EKumi.
    *
    * @throws ScriptLoadingFailureException if the script cannot be loaded.
    * @throws IllegalScriptIdentifierException if the given identifier is not properly formatted.
    */
    Runner resolveRunner(String identifier, Context context);
}
```

1.10.3 How to use the new scripting language within the workflow diagram editor?

Important: Feature not implemented yet.

1.11 Add a new Data Type

Important: This section requires some knowledge about Eclipse Extension Points.

1.11.1 What is a data type?

A data type is a language that can be used to specify the format of a data. The term data represents both inputs and outputs. Concretely, a datatype is a parser that can:

1. Instantiate an `Object` from a given String
2. Create a String from an existing `Object`.

It is hence responsible of serializing and deserializing data so that they can both be persisted and used during the execution.

1.11.2 How to add a new datatype?

A new one can be defined by contributing to the `fr.kazejiyu.ekumi.core.datatypes` extension point.

It requires one class that implements the `DataType<T>` interface.

The interface is defined as follows:

```
public interface DataType<T> {

    /**
    * Returns an identifier for this type.
    * @return an identifier for this type.
    */
    String getId();

    /**
    * Returns the name of the type.
    * @return the name of the type.
    */
    String getName();

    /**
    * Returns the Java class corresponding to this type.
    * @return the Java class corresponding to this type.
    */
    Class<T> getJavaClass();

    /**
    * Returns the default value of a new instance of this type.
    * @return the default value of a new instance of this type.
    */
    T getDefaultValue();

    /**
    * Returns a String representation of the type.<br>
    * <br>
    * For any type {@code type}, the following assertion must be {@code true}:
    * <pre>{@code instance.equals( type.unserialize(type.serialize(instance)) );}</
↪pre>
    *
    * @return a String representation of the type.
    *
    * @throws DataTypeSerializationException if T cannot be turned into a String
    *
    * @see #unserialize(String)
    */
    String serialize(T instance);

    /**
    * Returns a new instance of the type from a given representation.<br>
    * <br>
    * For any type {@code type}, the following assertion must be {@code true}:
    * <pre>{@code instance.equals( type.unserialize(type.serialize(instance)) );}</
↪pre>
    *
    * @param representation
    *                        The string representation of the type.
    *
```

(continues on next page)

(continued from previous page)

```
50      * @throws DataTypeUnserializationException if representation cannot be turned␣
↪into an instance of T
51      *
52      * @see #serialize()
53      */
54     T unserialize(String representation);
55 }
```

1.11.3 How to use the new datatype within the workflow diagram editor?

Important: Feature not implemented yet.

1.12 Add a new Specification

Important: This section requires some knowledge about Eclipse Extension Points.

1.12.1 What is a specification?

A specification is a definition of how a workflow is structured; as such, it can be affiliated to a concrete grammar.

A specification can be used to customize the way workflows are persisted, but are mainly aimed at supporting new editors (see *Add a new Representation*).

1.12.2 How to add a new specification?

A new one can be defined by contributing to the `fr.kazejiyu.ekumi.core.specs` extension point which requires one class that implements the `ActivityAdapter` interface.

The interface to implement is defined as follows:

```
1  public interface ActivityAdapter {
2
3      /**
4       * Returns whether the adapter can turn the given specification into an Activity.
5       *
6       * @param specification
7       *                       The specification to adapt, may be null.
8       *
9       * @return whether the adapter can turn the given specification into an Activity
10      */
11     boolean canAdapt(Object specification);
12
13     /**
14      * Creates an Activity from the given specification.
15      *
16      * @param specification
```

(continues on next page)

(continued from previous page)

```
17        *                        The specification to adapt.
18        * @param datatypes
19        *                        The factory used to instantiate available datatypes.
20        * @param languages
21        *                        The factory used to instantiate available scripting␣
↪languages.
22        *
23        * @return a new Activity
24        */
25       Optional<Activity> adapt(Object specification, DataTypeFactory datatypes,␣
↪ScriptingLanguageFactory languages);
26
27    }
```

An `ActivityAdapter` is responsible of turning your own specification model into an `Activity` so that the framework can execute it.

1.12.3 How to integrate the new specification within the IDE?

Important: Feature not implemented yet.

1.13 Add a new Representation

Important: This section requires some knowledge about Eclipse Extension Points.

1.13.1 What is a representation?

A specification describes a possible representation of a workflow. The main purposes of represenations is allowing new workflow editors (which can visual, textual or even in-memory).

1.13.2 How to add a new representation?

A new one can be defined by contributing to the `fr.kazejiyu.ekumi.ide.project_customization` extension point.

It requires a contribution to the *representations* attribute.

1.13.3 How to integrate the new representation within the IDE?

The representation is automatically proposed to the user in the *New Workflow Project* wizard as soon as the new extension is completed.

1.14 Share an Activity

Important: This section requires some knowledge about Eclipse Extension Points.

Todo: Explain usage of the `categories` extension point.

1.15 Build a workflow

1.15.1 Create an activity

An activity is a single unit of process that takes inputs and produces outputs. The `Activity` interface defines the exact behavior of an activity.

In order to create a new activity, the simplest way is to create a class that extends `AbstractActivityWithStateManagement`. This abstract class hides all the complexity related to state, inputs and outputs management as well as error handling. It only requires the sub-class to implement the `doRun(Context)` method, which is then called each time the activity is executed.

The following example shows a simple `HelloWorld` activity:

```
1  import fr.kazejiyu.ekumi.core.workflow.Context;
2  import fr.kazejiyu.ekumi.core.workflow.impl.AbstractActivityWithStateManagement;
3
4  /**
5   * An activity that prints "Hello World!" when run.
6   */
7  public class HelloWorld extends AbstractActivityWithStateManagement {
8
9      @Override
10     public void doRun(Context context) {
11         System.out.println("Hello World!");
12     }
13
14 }
```

About the execution context

As shown in the example above, the `doRun` method takes a `Context` instance as parameter. This instance represents the context of the execution and can be used, among other things, to:

- set/get global variables
- cancel the execution

The following activity prints different sentence depending on whether the execution has been cancelled:

```
1  import fr.kazejiyu.ekumi.core.workflow.Context;
2  import fr.kazejiyu.ekumi.core.workflow.impl.AbstractActivityWithStateManagement;
3
4  /**
```

(continues on next page)

(continued from previous page)

```
 * An activity that, when run, prints:
 *  - "Cancelled" if the execution has been cancelled
 *  - "Succeeded" otherwise
 */
public class HelloWorld extends AbstractActivityWithStateManagement {

    @Override
    public void doRun(Context context) {
        if (context.execution().isCancelled()) {
            System.out.println("Cancelled");
        }
        else {
            System.out.pritnln("Succeeded");
        }
    }

}
```

Provide inputs and outputs

An activity's inputs and outputs can be accessed through the `inputs` and `outputs` methods.

The following example shows how to create an `AplusB` activity that:

- has two inputs called *a* and *b* of type `double`
- has one output called *c* of type `double`
- set *c* to the value of *a* + *b*

```
import fr.kazejiyu.ekumi.core.workflow.Context;
import fr.kazejiyu.ekumi.core.workflow.impl.AbstractActivityWithStateManagement;
import fr.kazejiyu.ekumi.datatypes.DoubleType;

/**
 * An activity that computes its outputs 'c' from the addition of its two inputs 'a'␣
↪and 'b'.
 */
public class AplusB extends AbstractActivityWithStateManagement {

    public AplusB(String id) {
        super(id, "a + b = c");

        inputs().create("a", new DoubleType());
        inputs().create("b", new DoubleType());
        outputs().create("c", new DoubleType());
    }

    @Override
    protected void doRun(Context context) throws Exception {
        double a = (double) input("a").value();
        double b = (double) input("b").value();

        output("c").set(a + b);
    }

}
```

Inputs and outputs can also be assigned to fields for more convenience:

```java
import fr.kazejiyu.ekumi.core.workflow.Context;
import fr.kazejiyu.ekumi.core.workflow.Input;
import fr.kazejiyu.ekumi.core.workflow.Output;
import fr.kazejiyu.ekumi.core.workflow.impl.AbstractActivityWithStateManagement;
import fr.kazejiyu.ekumi.datatypes.DoubleType;

public class AplusB extends AbstractActivityWithStateManagement {

    private Input a;
    private Input b;
    private Output c;

    public AplusB(String id) {
        super(id, "AplusB");

        a = inputs().create("A", new DoubleType());
        b = inputs().create("B", new DoubleType());
        c = outputs().create("C", new DoubleType());
    }

    @Override
    protected void doRun(Context context) throws Exception {
        double a = (double) this.a.value();
        double b = (double) this.b.value();

        c.set(a + b);
    }

}
```

From lambda expressions

Alternatively, the `Activity` interface provides factory methods allowing to creates activities from lambda methods:

```java
import fr.kazejiyu.ekumi.core.workflow.Activity;

public class Main {

    public static void main(String[] args) {
        Activity sayHello = Activity.of(() -> System.out.println("Hello!");
        Activity sayIfCancelled = Activity.of(context -> System.out.println(context.
↪execution().isCancelled());
    }

}
```

1.15.2 Bind activities

Configure execution order

The execution order of two activities can be specified through a predecessor/successor relationship. A predecessor is always executed **before** its successor.

This relationship can be created via the `precede` and `succeed` methods:

```java
import fr.kazejiyu.ekumi.core.workflow.Activity;

public class Main {

    public static void main(String[] args) {
        Activity first = Activity.of(() -> System.out.println("First");
        Activity second = Activity.of(() -> System.out.println("second");

        // 'first' will be executed before 'second'
        first.precede(second);

        // Can also be written this way:
        second.succeed(first);
    }

}
```

Bind outputs to inputs

Outputs and inputs must be connected in order to share data between activities. An input can be connected to another data by calling the *Input.bind(Data)* method.

```java
import fr.kazejiyu.ekumi.core.workflow.Activity;

public class Main {

    public static void main(String[] args) {
        Activity add1 = new AplusB("add1");
        Activity add2 = new AplusB("add2");

        // At runtime, the value of add2's 'b' input
        // will be set to the value of add1's 'c' output
        add2.input("b").bind(add1.output("c"));
    }

}
```

1.15.3 Create a sequence of activities

Usually successors and predecessors must be executed sequentially. A `Sequence` is a composite activity that owns a root activity and which, when run, executes its root activity then all its successors and the successors of the successors and so on.

One can be created as follows:

```java
import fr.kazejiyu.ekumi.core.workflow.Activity;
import fr.kazejiyu.ekumi.core.workflow.Sequence;
import fr.kazejiyu.ekumi.core.workflow.impl.BasicSequence;

public class Main {

    public static void main(String[] args) {
        Activity print1 = new Print("print1", "1");
        Activity print2 = new Print("print2", "2");
```

(continues on next page)

(continued from previous page)

```
        // Create a predecessor/successor relationship,
        // Thus ensuring the sequence will execute print1 then print2
        print1.precede(print2);

        Sequence print1Then2 = new BasicSequence("id", "Print 1 then 2", print1);
    }

}
```

1.15.4 Create parallel activities

A `Parallel Split` is a composite activity that executes its own activities concurrently.

Once can be created as follows:

```
import fr.kazejiyu.ekumi.core.workflow.Activity;
import fr.kazejiyu.ekumi.core.workflow.ParallelSplit;
import fr.kazejiyu.ekumi.core.workflow.impl.BasicParallelSplit;

public class Main {

    public static void main(String[] args) {
        List<Activity> concurrentActivities = new ArrayList<>();
        for (int i = 0 ; i < 4; ++i) {
            Activity concurrentActivity = Activity.of(() -> System.out.println("I run concurrently");
            concurrentActivities.add(concurrentActivity);
        }
        ParallelSplit split = new BasicParallelSplit("id", "Run branches in parallel", concurrentActivities);
    }

}
```

1.15.5 Create loop of activities

A `Structured Loop` is a composite activity that executes another activity until a specified pre or post condition is verified.

One can be created as follows:

```
import fr.kazejiyu.ekumi.core.workflow.Activity;
import fr.kazejiyu.ekumi.core.workflow.Condition;
import fr.kazejiyu.ekumi.core.workflow.StructuredLoop;
import fr.kazejiyu.ekumi.core.workflow.impl.BasicStructuredLoop;

public class Main {

    public static void main(String[] args) {
        Condition preCondition = null; // the loop has no pre-condition
        Condition postCondition = Condition.of(() -> true); // the loop will end after the first execution

```

(continues on next page)

(continued from previous page)

```
        Activity greet = Activity.of(() -> System.out.println("Hello"));

        StructuredLoop loop = new BasicStructuredLoop("id", "name", greet,
↪preCondition, postCondition);
    }

}
```

1.16 Execute a Workflow

1.16.1 Launch a background execution

An `Execution` instance is required to execute properly an activity.

EKumi provides a `JobsExecution` class that can be used to execute activities in background, relying on Eclipse Jobs API.

The following code shows how to use it:

```
import fr.kazejiyu.ekumi.core.workflow.Activity;
import fr.kazejiyu.ekumi.core.workflow.Execution;
import fr.kazejiyu.ekumi.core.workflow.impl.JobsExecution;

public class Main {

    public static void main(String[] args) {
        Activity print1 = new Print("print1", "1");
        Activity print2 = new Print("print2", "2");

        // Create a predecessor/successor relationship,
        // Thus ensuring the sequence will execute print1 then print2
        print1.precede(print2);

        Sequence print1Then2 = new BasicSequence("id", "Print 1 then 2", print1);

        Execution execution = new JobsExecution(print1Then2);

        // Launches the execution in background
        execution.start();

        // Wait for the execution to end
        execution.join();
    }

}
```

1.16.2 Listen for events

Callbacks can be add to the execution in order to react when the workflow changes. This can for instance used to persist the workflow or to update an UI.

The following code shows how to print a message each time a new activity starts:

```java
import fr.kazejiyu.ekumi.core.workflow.Activity;
import fr.kazejiyu.ekumi.core.workflow.Execution;
import fr.kazejiyu.ekumi.core.workflow.impl.JobsExecution;

public class Main {

    public static void main(String[] args) {
        Activity workflow = ...
        Execution execution = new JobsExecution(workflow);

        // Register a callback
        execution.context()
                 .events()
                 .onActivityStarted(activity -> System.out.println(activity.name() +
↪" starts"));

        // Launches the execution in background
        execution.start();

        // Wait for the execution to end
        execution.join();
    }

}
```