OpenRefine's Technical Documentation for Extension's Writing

Giuliano Tortoreto
giulian.trt@gmail.com

September 2014

Contents

This documentation aims to make OpenRefine's extensions writing easier. To do this, we use a functional approach especially for the Back End, where there is a subsection for each kind of data operation.
In this introduction section some general information will be given, with carefully to the basis, installation set and an overview on the source code. In the next section we mind extension initialization and extension structure, recalling the official documentation. After, we discuss how to perform the interface and the client side in general. The last section takes care of Server Side.

Basis First of all, to be able to write an extension for OpenRefine what is needed is a good knowledge of Java for the server side and a good JavaScript knowledge for the client side.
Both languages are Object Oriented, so this documentation makes extensive use of Objects. An Object is a particular instance of a Class. A Class is a template that consist of variables and methods. Class and Class Interface are extensive used in Java, therefore we need also the Class Interface definition. A Java Class Interface is an abstract type that provides a set of method signature, it defines the Class behaviour.

Installation Set Writing OpenRefine's extensions requires the installation of Apache Ant, JDK and a Java IDE. Apache Ant is useful to compile your extension by command line and to run and test OpenRefine.
This web application requires the Java Development Kit 1.4, but the recommendation is to use always the latest official version.
Recommended Java IDE, Integrated Development Environment, are Netbeans and Eclipse because there are already ready settings and documentation. You can find more information to the official Documentation here: https://github.com/OpenRefine/OpenRefine/wiki/Developers-Guide. This link gives more information on building, setting, testing and running OpenRefine.

OpenRefine server side source code map In spite of we want to limit referring to the OpenRefine code, it could be useful to easily search into the code. Therefore we give a OpenRefine source code map to make developer's life easier.
In this map there is a description for every package, emphasizing the most important classes and class interfaces. All the java server side code it's available in the directory textsl/main/src/com/google/refine. Hereunder there are the main useful packages for OpenRefine extension's writing.

Package	Contents
Browsing	Facet classes and all the secondary classes needed for facet information processing. Furthermore there are the filter classes, the engine class and visitor interfaces.
Commands	All command implementations are here. There are also the Command and EngineDependentCommand class, essentials for extensions.
Exporters	This folder contains default OpenRefine exporters classes and interfaces.
Expr	All OpenRefine function classes and the EvalError class, this class allows to show error in cells.
Grel	Here you can find the Function interface and some other grel functions.
History	The Change Interface and some History classes
Importers	All default OpenRefine importers classes are in this folder
Importing	This folder contains classes which manage the importing phase
Model	The main cell, column, row, project classes and alternatives implementation. There is also the Overlay Model and the AbstractOperation interface.
Operations	All OpenRefine's default operation are in this package. Two classes are more interesting: EngineDependentOperation and EngineDependentMassCellOperation.
Process	The Process, LongRunningProcess and ProcessManager class are here.

1 Starting: Structure and Initialization

Simile Butterfly For the OpenRefine Extension structure and Initialization we have to recall that this software is based on Simile Butterfly framework. This framework has a specific module layout. You can see it below.

```
example-module /
   + (files)
   + MOD-INF /
      + module.properties
      + controller.js
      + src/ (optional)
      + lib/ (optional)
      + classes/ (optional)
```

OpenRefine's extensions are very similar to it and also use the same initialization file's names. The module.properties file is a configuration file and it has a key = value format.

Extension Structure The OpenRefine extension structure is similar to the standard Simile Butterfly module, but it's not the same. First of all it needs a build.xml file in the main folder and two main sub-folder:module and src. The src folder contains the back end files. Instead, the module folder includes all the folder for the extension interface and one folder containing the initialization and setup files, called MOD-INF. The MOD-INF folder comprehend two folders:lib and classes. The java libraries files (.jar) are within lib folder. Instead, the classes folder contains all the compiled java source files. You can see below the folder structure quoted from the official documentation.

```
build.xml
src/
    com/foo/bar/... *.java source files
module/
    *.html, *.vt files
    scripts/... *.js files
    styles/... *.css and *.less files
    images/... image files
    MOD-INF/
        lib/*.jar files
        classes/... java class files
        module.properties
        controller.js
```

1.1 Initialization Files

The main initialization files for an OpenRefine extension are: build.xml, module.properties and controller.js. Each one of them has particular characteristic. First of all, the build.xml file is important for the extension building. After, extension's metadata are within the *module.properties* file. The last is the controller.js file that contains the links from the extension to OpenRefine.

module.properties This file contains at least 3 information: name, dependency and description. The name field is important because it is used in many places, it could be different from the extension's directory name. The second field is usually description, here you can describe your extension. After the dependency field is important, it must ensure that the OpenRefine's core module is loaded before the extension.

```
name = custom-extension-name
description = Extension's description
requires = core
```

controller.js Within this file must be registered all the extension's elements. Other important information can be added, like a logger and the class mapping registration between an extension class and one already available in OpenRefine. For the server side, recalling the official documentation (https://github.com/OpenRefine/OpenRefine/wiki/Extension-Points), in the init method of this file must be registered all the:

- Commands:

```
var RServlet = Packages.com.google.refine.RefineServlet;
RServlet.registerCommand(module, "my-command",
new Packages.com.foo.bar.MyCommand());
```

- Operations:

```
var operations = Packages.com.google.refine.operations;
var OR = operations.OperationRegistry.registerOperation(
module, "operation-name",
Packages.com.foo.bar.MyOperation);
```

- Functions:

```
var grel = Packages.com.google.refine.grel;
grel.ControlFunctionRegistry.registerFunction(
   "functionName",
    new Packages.com.foo.bar.TheFunctionClass()
);
```

- Binders:

```
var expr = Packages.com.google.refine.expr;
expr.ExpressionUtils.registerBinder(
new Packages.com.foo.bar.MyBinder());
```

- functions concerning the **Importer** addition:

```
var importing = Packages.com.google.refine.importing;
var IM = importing.ImportingManager;
IM.registerFormat(
  "generic-type/more-specific-type",
  "User-friendly name",
  "name", new Packages.com.foo.bar.MyImporter());

IM.registerExtension(
".extension", "generic-type/more-specific-type");

IM.registerMimeType(
"Mime-type", "generic-type/more-specific-type")
```

```
IM.registerFormatGuesser(
   "generic-type/more-specific-type",
   new Packages.com.foo.bar.MyImporterGuesser());

IM.registerController(
module, "identifier-name",
new Packages.com.foo.bar.MyImportinController())
```

- Exporters:

```
var Exp = Packages.com.google.refine.exporters;
Exp.ExporterRegistry.registerExporter(
"exporter-name", new Packages.com.foo.bar.MyExporter());
```

- Overlay Models:

```
var project = Packages.com.google.refine.model.Project;
project.registerOverlayModel(
"model-name", Packages.com.foo.bar.MyOverlayModel);
```

- Scripting Languages:

```
var MP = Packages.com.google.refine.expr.MetaParser;
MP.registerLanguageParser("jython",
"Jython",
Packages.com....jython.JythonEvaluable.createParser(),
"return value");
```

For the client side, instead, you can add all your javascript files and css files, in this way:

```
ClientSideResourceManager.addPaths(
       "project/scripts",
       module,
       [
           "scripts/foo.js",
           "scripts/subdir/bar.js"
       ]
   );
ClientSideResourceManager.addPaths(
"project/styles",
       module,
       [
           "styles/foo.css",
           "styles/subdir/bar.less"
       ]
   );
```

In the controller.js can be added other information like a logger that gives information to the developer about the progress of the extension initialization.

```
var LF = Packages.org.slf4j.LoggerFactory;
var logger = LF.getLogger("extension-name");
logger.info("Log message to the developer");
```

Another interesting method that can be called by the controller.js is registerClassMapping. From the OpenRefine source code:"This allows to determine how old class names can be updated to newer class names". For example:

```
var refineServlet = Packages.com.google.refine.RefineServlet;
refineServlet.registerClassMapping(
"com.metaweb.*", "com.google.*"
)
```

build.xml file The required file to compile the OpenRefine extensions through ant is the build.xml. In this file there are many information about the project, among which: properties, conditions, paths and targets.
At the beginning of the file there are the XML version and the encoding type. The root element that contains all the other elements is **Project**. The Project element has 3 parameters: name, default and basedir. Conditions elements has two parameters: property and value, are needed for information like the version and some distribution details. Properties elements has two main attributes name and value. This kind of elements gives information to ant about OpenRefine folders location. Paths elements has only an id, but contains many fileset elements, these elements communicate where are the *.jar files. The last but not the least elements are Target which aim to easily compile and clean extensions. Below there is a default build.xml file. It contains all information to compile every OpenRefine extension that is placed in the OpenRefine's extensions folder, just change "extension-name". The most important point is that the java classes should be built into /module/MOD-INF/classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="extension-name" default="build" basedir=".">
   <property name="name" value="extension-name"/>
   <property environment="env"/>
   <condition property="version" value="trunk">
      <not><isset property="version"/></not>
   </condition>
   <condition property="revision" value="rXXXX">
      <not><isset property="revision"/></not>
   </condition>
   <condition property="full_version" value="0.0.0.1">
      <not><isset property="full_version"/></not>
```

```
</condition>
<condition property="dist.dir" value="dist">
    <not><isset property="dist.dir"/></not>
</condition>
<property name="fullname"
value="${name}-${version}-${revision}" />
<property name="refine.dir" value="${basedir}/../../main"/>
<property name="refine.webinf.dir"
          value="${refine.dir}/webapp/WEB-INF" />
<property name="refine.modinf.dir"
          value="${refine.dir}/webapp/modules/core/MOD-INF"/>
<property name="refine.classes.dir"
          value="${refine.webinf.dir}/classes" />
<property name="refine.lib.dir"
          value="${refine.webinf.dir}/lib" />
<property name="server.dir" value="${basedir}/../../server"/>
<property name="server.lib.dir" value="${server.dir}/lib"/>
<property name="src.dir" value="${basedir}/src" />
<property name="module.dir" value="${basedir}/module" />
<property name="modinf.dir" value="${module.dir}/MOD-INF"/>
<property name="lib.dir" value="${modinf.dir}/lib" />
<property name="classes.dir" value="${modinf.dir}/classes"/>
<path id="class.path">
    <fileset dir="${lib.dir}" erroronmissingdir="false">
        <include name="**/*.jar" />
    </fileset>
    <fileset dir="${refine.lib.dir}">
        <include name="**/*.jar" />
    </fileset>
    <fileset dir="${server.lib.dir}">
        <include name="**/*.jar" />
    </fileset>
    <pathelement path="${refine.classes.dir}"/>
</path>
<target name="build_java">
    <mkdir dir="${classes.dir}" />
    <javac encoding="utf-8" destdir="${classes.dir}"
    debug="true" includeAntRuntime="no">
        <src path="${src.dir}"/>
        <classpath refid="class.path" />
    </javac>
</target>
<target name="build" depends="build_java"/>
<target name="clean">
    <delete dir="${classes.dir}" />
</target>
```

```
</project>
```

2 Interface: Front End

About the client side, there are some information to give. First of all, all the OpenRefine's client side source file are at the path /main/webapp/-modules/core. In this place you can find all the external libraries, images, stylesheet files and javascript files.
OpenRefine also allows to add velocity templates and many extension's point to the interface. To make OpenRefine developer's life easier this software use two external library to perform the interface: JQUERY and JQUERY-UI. Both JQUERY and JQUERY-UI are JAVASCRIPT open source libraries. The first one provides many features of manipulation, event handling, animation and AJAX. The second one includes a set of interface:

- elements, like: button, dialog, slider, spinner, tabs, etc. ;
- interactions on elements, like: drag, drop, resize, select and sort;
- effects on elements, like: show, hide, color animation, style changes animation, etc.;

Client side source code overview The main directory of the client side contains many folders, so we give an overview of it. The external libraries introduced before are placed in /core/externals folder.
Another directory is about the initialization files: /core/MOD-INF folder, where the OpenRefine's controller.js and module.properties are placed.
A directory contains all JAVASCRIPT files: /core/scripts. In this folder you can understand how the client side manages the AJAX calls, looking at the project.js file. In this file is defined the main function to perform an AJAX call. Here is also defined theProject object that contains the information about the client visible data. Other useful functions are implemented in this file, like a function that given the column name returns the column.
The /core/scripts includes also other folders:

- util contains ready-made function that aims to: manage the column menu; process strings; manage the DOM; manage the dialog system; manage dates; manage menus.
- project contains JAVASCRIPT files for the extension-bar and history operation.
- index includes: files for the project creation; files for the import operation and the project open operation; a folder for the parser interfaces useful for ones who wants to add an importer.
- facets contains the files for all facet interfaces.

- dialogs, instead, contains dialog's implementations.

There are other folders, but they are not so interesting for writing an extension.

2.1 UI - adding elements

Recalling the official documentation, to the user interface can be added:

- Dialogs, using HTML templates and a specific function that loads the HTML file on the fly and then injects it into the page's DOM.
- Extension menu: it is a bar over the data model on the user interface, each extension can add buttons to it, using a specific function.
- Column header menu item: it is a drop-down menu that appear clicking on every column, can be extended using some functions.
- Action area: it is an area within the starting page. As default there is a menu with four entry on the left side of the page. The menu can be extended through a method. To each entry correspond a User Interface class, which is displayed, near the menu, when the user clicks on the entry.

All these elements don't need to be added in a specific file. We recommend to keep the menu adding functions in the same file and create a dialog dedicated folder. Each dialog can require more than one file, among which: an HTML template, a stylesheet and the JAVASCRIPT to inject the template into the user interface.

Extension menu The extension menu, also known as main menu, is constructed once at start-up time. It can be extended by using the addExtensionMenu of the ExtensionBar object. This function can be called from every file in the extension path. The addExtensionMenu function allows to add a new specific button for your extension, it needs a JSON object containing 3 "key-value" parameter: id, label and one between submenu or click. The `id` parameter should correspond to a relative url prefixed with the extension name. The `label` parameter is displayed on the user interface. About the submenu parameter, its value element must be a JSON array. Each object inside the array must have three parameters, id and label are mandatory the third could still be one between submenu or click. About the click parameter requires a function that can:

- show a dialog;
- perform an AJAX call;
- add a facet;

In this section we deal on show a dialog, leaving the other two points for the next section. Below an example of Extension bar:

```
ExtensionBar.addExtensionMenu({
   id: "foo-extension",
   label: "Foo",
   submenu :[
   {
      id: "foo-extension/about",
      label: "About...",
      click: function() {
         new AboutDialog(column.name, "value");
      }
   }]
})
```

Column header menu item First of all, we recommend to put the code concerning this kind of menu extension in the same file of the previous extension menu.
However, its construction mechanism is slightly different from the previous menu. Given that this menu can not be done once, but it must be constructed every time a user click on a column header. For this reason, the column header menu extension needs a registration that provides a function which can get called every time.

```
DataTableColumnHeaderUI.
       extendMenu(function(column,columnHeaderUI,menu){
       .... use a MenuSystem's function to add an item ....
}
```

OpenRefine provides some function for the column header menu extension. Each function allows to add a menu element in a different position through the MenuSystem object. This object provides 3 specific functions, that needs the same parameters root Items, path and what.
The first parameter must be the menu variable from the function signature. The path parameter requires an array of id strings, often an array of one element is enough. The what parameter could be an array of JSON objects or just a single JSON object. In more than one parameter case the first item stands for the menu item, whereas the second parameter is one of its submenus and so on. You can see the count of the elements in the array like the depth rate in the menu. The object is the menu item with 3 attributes: id, label, submenu or click. In this parameter could be also added a menu separator, it can be done adding an empty JSON object to the array.
Furthermore, you have to use a unique id. There are already used ids, among which:"core/facet", "core/edit-cells", "core/edit-column", "core/-transpose", "core/sort" and "core/view". These ids can be used like path parameters in some functions:

- appendTo, the path parameter is needed if and only if you want to append your menu item to another already existent core menu item. Otherwise it must be an empty string without blank spaces, just “”, and the menu item will be appended to the main column menu.

```
DataTableColumnHeaderUI.extendMenu(
   function(column, columnHeaderUI, menu){
      MenuSystem.appendTo(menu, ["core/edit-cells"], {
         id : "foo-extension/foo",
         label: "Click here, you'll find wonderful things",
         click: function(){new FooDialog();}
      });

   // ... you can call as many functions as you want ... //

      MenuSystem.appendTo(menu,"", {
         id : "foo-extension/try",
         label: "Just try it",
         submenu: [
            { ... an item .... }
         ]
      });
});
```

- insertBefore and insertAfter, these functions are both based on a certain element already existent in the menu, so the path parameter is mandatory. Otherwise you won't see your item on the menu. InsertBefore adds a an item before a specific item, on the other hand insertAfter adds an item after a specific item.

```
DataTableColumnHeaderUI.extendMenu(
   function(column, columnHeaderUI, menu){
      MenuSystem.insertBefore(menu,
      ["core/edit-column","core/rename-column"],
      [ { /* separator */ }, {
         id : "foo-extension/column-fancy-things",
         label: "Do fancy things on a column",
         submenu: [
            { ... an item .... }
         ]
      } ]);

   MenuSystem.insertAfter(menu, ["core/transpose"], {
      id : "foo-extension/the-foo-after",
      label: "This is a very beautiful item",
      click: function(){ new MagicDialog(); }
   });
```

```
});
```

Dialog A OpenRefine's Dialog is a specific feature based interface. Performing a dialog requires an html template, an object preferably within a JAVASCRIPT file and a stylesheet (a .css or a .less file).
To call a dialog it's needed an extension menu or a header column menu item with a click attribute. This attribute triggers a function that previously creates the dialog and after shows the just created dialog. The creation phase requires many operation, among which: the injection of the html template into the page's DOM, the binding of the HTML element and the showDialog method call.

```
....
   ....
   label: "About my extension"
   click: function () {new AboutDialog(); }
   });
})

 /* .... another file .... */

function AboutDialog(columnName, expression) {
   this.aboutColumn = columnName;
   this.aboutExpression = expression;

   this.createDialog();
   this.whetheverYouWant();
}

AboutDialog.prototype.createDialog = function() {
   var self = this;
       ....
}
```

First of all, to inject the html template in the DOM, we need to load the HTML file using a specific function that requires 2 parameters. The first parameter is the extension name used in the controller.js, the second one requires the file name prefixed with the relative path starting from the module folder (see below).

```
var dialogElement =
   $(DOM.loadHTML("extension-name",
                  "cool_directory/file.html")
    );
```

We choose to save the loaded HTML element in a variable, because the element is still not visible. After that it should be useful to bind "actions" to

HTML elements contained in the loaded file. To do this we need to add a bind attribute to them. For instance:

```
<button bind="hide-from-my-sight"> Hide </button>
```

To retrieve HTML "bind-tagged" elements, the bind method of the DOM object can be called. The bind function requires one parameter: the previous loaded file. Thus, we can store in a variable the list of returned elements.

```
var controls = DOM.bind(dialogElement);
```

After that it is possible to bind dialog elements with functions. So, we are ready to add an "action" to each control through adding a function to the click attribute of each control object.

```
controls.hide-from-my-sight.click(function() {
      self.hide();
}
   ...
   /* ............ other code ........... */
   ...
AboutDialog.prototype.hide = function() {
   DialogSystem.dismissUntil(this.dialogLevel -1);
}
```

When all controls are bound to a function the dialog can be shown. The DialogSystem object provides a method, named showDialog, that displays dialogs on the user interface.

```
this.dialogLevel = DialogSystem.showDialog(this.dialogElement);
```

Now the dialog creation process is complete. We recommend to store returned object from this function. So, in this way, it's possible to hide the dialog (see the previous piece of code).

Action area An action area is a dedicated user interface for features which don't concern already created project's data. For instance, the "Create project" feature has a dedicated action area, as does the "Open project" one.
To implement this type of features in a different way, a new Action area can be useful. Through it, either a customized project's meta-data visualization or a customized project creation can be implemented. To add a new Action area we need the push method of the Refine's actionAreas object. It requires a JSONobject containing three keys: an id, a label and a UI class. The id key requires a unique string as value. Whereas the label requires a string value, this string will be visible on the User Interface. On the other hand, the UI class key expects a JAVASCRIPT class. It must have an implemented constructor and an implemented resize method, also containing an empty body. Hereunder there is an example, from OpenRefine's source code "import-project-ui.js" file:

```
Refine.ImportProjectUI = function(elmt) {
  elmt.html(DOM.loadHTML("core",
                "scripts/index/import-project-ui.html"));

  this._elmt = elmt;
  this._elmts = DOM.bind(elmt);
  ...
};

Refine.actionAreas.push({
  id: "import-project",
  label: $.i18n._('core-index-import')["import-proj"],
  uiClass: Refine.ImportProjectUI
});

Refine.ImportProjectUI.prototype.resize = function() {
};
```

You can find all implemented Action Areas at the path main/webapp/modules/scripts/index.

2.2 Passing Arguments, adding Facet and other Information

Passing arguments Once the dialog is created, our user interface is ready to transmit information to the server side. The communication between client and server side is essential to retrieve information about all rows and to trigger changes on data. Therefore, it's necessary to have a function that allows the front end to communicate with the back end. The needed function is the postProcess() method of the Refine object.

```
Refine.postProcess("foo-extension", "foo-command",{},{},{},{});
```

This function requires 6 parameters:

- modulename is the name of the module defined in the module.properties within a string
- command, this is also a string which expects a registered command name. Depending on this string the client side communicates with a certain command rather then another.
- params, this is a JSONobject containing a pair for each parameter to pass.

```
var data = { };
data["column"] = this.column.name;
data["mode"] = this.mode;
```

- body, this is a JSON object that contains a pair for each parameter. You can use this instead or in addition of the previous parameter, it doesn't make difference. It can be useful to divide different parameters into two sets.

- updateOptions, in the case there are no changes on data or on the data model you can just put an empty JSON object. Otherwise, can be added two pairs: rowChanged setted as true or false and modelsChanged setted as true or false. It depends on the changes that you are going to do.

- callbacks, this parameter is very important in the case you want to show the dialog just after the server side has terminated to compute.

```
var self = this;
callback = {
   "onDone": function (response) {
      self.myResponse = response;
      if (initCallback)
         initCallback.apply(self);
   }
}
```

Information already available On the client side to retrieve the available information can be used the theProject JSON object. This object contains all information shown in your browser. The theProject.columnModel contains all information about columns, whereas the theProject.rowModel contains all the data displayed in the table, each cell is placed in this object. For example, to extract all the rows indexes displayed on your browser it is enough to iterate on the theProject.rowModel.rows array object and copying the "i" attribute.

```
var rows = theProject.rowModel.rows;
var indexes = "";
for (var index = 1; index <rows.length; index++)
   indexes = indexes + " " + rows[index].i;
```

Within each rows object there is an array object cells which contains every cell for that row. This implies that each cell is reachable by its index in the row. For this reason in the project.js there are some utility model functions, like: Refine.cellIndexToColumn, Refine.columnNameToColumn and Refine.columnNameToColumnIndex.

2.2.1 Adding a Facet

First of all, a facet is the way to filter rows in OpenRefine. Each type of facet is based on a single column, except for scatterplot facet type. The official documentation says that each browser's window can have a different

browsing state. This implies that adding a facet is a client side matter. It's possible by using the BrowsingEngine object that offers a specific method for this: `addFacet`. This function requires 3 parameters:

1. type, this parameter is mandatory and must be a string and OpenRefine accepts only 5 values: "list", "range", "timerange", "text", "scatterplot". Every previous value corresponds to a different facet type. In case OpenRefine receives a different value will instantiate a "list" facet, indeed "list" facet is the default case.

2. configurations parameter is mandatory. Contains all the information to perform the facet. Each facet requires different configurations.

3. options, this is a JSON object that allows to add more information, like the pair scroll and sort for the list facet. The first pair allow to set the result panel scrollable and resizeable or not, default case is scrollable. The sort parameter implies that results are sorted by the most returned result to the least.

```
var expression =
   "value.isInTheArea(\"" +
     + jsonToString.replace(/"/g, "\\\"") + "\")";
ui.browsingEngine.addFacet(
   "list",
   {
      name : self.column.name,
      columnName : self.column.name,
      expression : expression,
      selectError : false,
      invert : false
    },
    {
       scroll : true,
       sort : true
    }
);
```

Facet Configurations This parameter vary strongly according to the facet type. This JSON object can contain many pairs, each with a specific purpose. Each JSON pair is composed by a string and a value. Every string requires a different value type. We can divide the pairs by value:

- String values are mandatory configuration to compute the facet:
  - "name" Usually correspond to the column name on which the facet is performed. Except for the not based on column case. This is the only common pair to every facet type.

  - "columnName" contains the column name on which the facet is performed. Often this pair match the "name" pair. In case of not based on column facet this parameter must be empty.
  - "expression" contains a piece of program code, in a scripting language between GREL, JYTHON and CLOJURE, which returns a value. To make explicit what language in use prefix the string with "chosen-language:", default is GREL.
  - "mode" according to the facet type used can accept different values. For the range facets the accepted value is "range". On the other hand for text facets can be either "text", if you want to search by text, or "regex", if you want to search through a regular expression.
  - "query" this is the pair used by the text facet to filter rows. The value can be either a regular expression or a text part. How this query is interpreted depends on the "mode" value.

- Boolean value are usually configurations that affect the data visualization when the facet computation is terminated, except for the caseSensitive.
  - "selectBlank": if this parameter is setted as "true", rows, containing a blank value in the column on which facet is performed, are selected first.
  - "selectError": if this parameter is setted as "true", rows, that returned an error during facet computation, are selected first.
  - "selectNumeric": if this parameter is setted as "true", rows, which expression returned a numeric value, are selected first.
  - "selectNonNumeric": if this parameter is setted as "true", rows, which expression returned a non-numeric value, are selected first.
  - "selectTime": if this parameter is setted as "true", rows which expression returns a time-stamp value are selected first.
  - "selectNonTime": if this parameter is setted as "true", rows which expression returns a not time-stamp value are selected first.
  - "omitBlank": if this parameter is setted as "true", rows which expression return a blank result are not shown in the results list.
  - "omitError": if this parameter is setted as "true", rows that returned an error during facet computation are not shown in the results list.
  - "invert": when this value is setted as true if the user clicks on a list item on the results panel, the nonmatching result rows are the selected rows.
  - "caseSensitive", unlike the previous pair this string affects the computation phase, enabling the distinction between upper- and lower-case letters.

For scatterplot facet there are different pairs, indeed there is a detached paragraph for it.

Scatterplot configuration This facet type aims to display numeric data on a graph. User can select rows choosing an area of the graph. This facet requires many pairs:

- "project" value expects the project's id, easily reachable from theProject object.
- "engine" value expects the stringified browsing engine state. To extract the browsing engine state can be used a ui.BrowsingEngine's functions: getJSON()
- "plotter" value is a stringified JSON object that contains all the informations to display on the graph:
  1. "name" is the graph title;
  2. "cx" contains the column name that is displayed on the x axis, "cy" is the column name standing for y axis;
  3. "l", is the size of the graph when placed in the facet result panel;
  4. "ex" contains the expression on x, "ey" instead, the expression on y;
  5. "dot" stands for dot size;
  6. "dim-x" and "dim-y", replace the dash with underscore, stands for the x axis dimension and y axis dimension
  7. "r" stands for graph rotation.

Mapping Configurations on type

1. "list" facets gather together all the element that has the same expression result. Then it is possible to filter grounding on results, but if there are too much different results, OpenRefine doesn't display any of them. The required configurations are just three pairs: "name", "columnName", "expression". There are other configurations, but they are just discretionary: "invert", "selectBlank", "selectError", "omitBlank" and "omitError".
2. "range", this facet allows to display the numeric result allocation of a specific expression, in order to show dispersion and accumulation points. This facet requires 4 mandatory pairs: "name", "columnName", "expression" and "mode" within "range" value. There are also discretionary pairs: "selectBlank", "selectError", "selectNumeric" and "selectNonNumeric".

3. "text", this is a features that allows to filter text based on a regular expression or a same old find in text. This facet requires 4 mandatory pairs: "name", "columnName", "query" and "mode". Another pair can be added: "caseSensitive".

4. "timerange", this is the timestamp version of the previous described range facet. Requires the same pairs of the range facet, but the discretionary pairs are different: "selectTime" and "selectNonTime".

3 Data Processing: Back End

When a command is triggered, by the Refine.PostProcess method on the client side, it's possible to compute many kind of elaborations on data. Some elaborations can affect either the data model or the column model. Others aim just to compute operations in order to facilitate data comprehension. For this purpose to this software can be added Data Visualization features and additional High-level Models on top of project data. OpenRefine allows also to extend some already existent features, like: adding new importers, exporters, functions and a scripting languages.

Receive Input In case you want to add Data Visualization features or Data Processing features you have to pass through a command. The command usually is the object that receives communications from the client side. From the client side comes many parameters, like: the browsing engine state, the project's id and the parameters needed to satisfy the request and start the computation. Therefore the command object is useful for both Data Visualization and Data Processing. In first case the command is the main class, whereas in second case it is useful just to pass parameters and instantiate the abstract operation.

Rows, Columns and Cells Whichever feature we want to add, we have to deal with Rows, Columns and Cells. To access to a certain row we need to know its index in the project. The project object is needed in order to process the project data. This object can be retrieved by the request coming from the client side using the getProject function. For each request this function extract the whole project information from the project's id within the request. The project rows are reachable from the list attribute row of the Project object by the get function. This function, given the row index parameter, returns the row object.

```
int rowIndex = 0;
Project project = getProject(request);
Row row = project.rows.get(rowIndex);
```

Processing a certain cell requires to know the cell index in the row. The cell index can be obtained from the column object. Indeed the Column class provide the getCellIndex function that returns the index value in the

row of that certain column. In order to obtain the specific Column object we need at least the column name. Its name can be passed like argument to the command. Then by using the getColumnByName function of the columnModel attribute of the project object the column object is returned. The getColumnByName function requires one string parameter, the column name.

```
String columnName = "address";
Column column =
     project.columnModel.getColumnByName(columnName);
int cellIndex = column.getCellIndex();
```

When also the cell index in the row is known, the cell object is reachable by using the getCell function of the row object. Each cell has two attributes: "value" and "recon". The value attribute belongs to the Serializable class. In order to access to the value is required to call the toString method. The recon attribute contains the information concerning the reconciliation with an external service. For example, through the getField method of the recon attribute every information of the reconciliation can be obtained. The available informations are like "judgement" that tells us if the cell is already reconciled or not. Another useful field is "candidates" which is the list of the possible matching candidates.
The getField function requires 2 parameters, a string with the name of the desired field and the Properties object not used by Recon, so this parameter is always null. The String accepted are many: "best", "candidates", "judgement", "judgementAction", "judgementHistoryEntry", "judgementBatchSize", "matched", "new", "matchRank", "features", "service", "identifierSpace" and "schemaSpace".

FilteredRows and RowVisitor In case we need to evaluate more rows, OpenRefine provides a function that can visit filtered rows. This is a simple and clean way to iterate on the selected project rows. At this purpose OpenRefine provides the FilteredRows interface. The FilteredRows interface requires an accept function that goes through the rows of the given project. This function determines which row match and which don't. On those that match the implementation of the FilteredRows interface call the method visit of the RowVisitor passed as argument. RowVisitor is an interface for visiting rows one by one.
Depending on the need, we can use the RowVisitor and the FilteredRows inside an "operation" class or a "command" class. This entail different potential implementation according to the object chosen. The RowVisitor interface is shown hereunder (from the source code).

```
public interface RowVisitor{

   /**
    * Called before any visit() call
    */
```

```
   public void start(Project project);

   /**
    * Return true to abort visitation early
    * no further visit calls will be made
    */
   public boolean visit(Project project,
                        int rowIndex,
                        Row row
   );

   /**
    * Called after all visit() calls
    */
   public void end(Project project);
}
```

Records and Rows In OpenRefine there is difference between rows and records. Each record can contain more than one row. A row can start with an empty cell, whereas a record never starts with empty cells. A row having empty cells at the beginning is considered part of a record starting in a previous row. Due to the fact that cells belonging to a certain column can be strongly connected to the previous column. For example, assuming that we have a project containing data concerning films and there is a column "main-actors". Probably a film has more then one main actor and under this column there is more than one cell. For this reason there could be rows with just one or two non-empty cells. The meaning of these rows depends from the previous row that give context to it. That is way blank cells are very important to understand if a row depends from another or not. So blank cells allows to interpret table of rows as a list of tree-shaped data records. For this purpose, a while ago OpenRefine used group of columns. Now group of columns are used only for the XML imported files, letting the records to group the connected rows for the other file types. Indeed, in OpenRefine the user can choose between display rows or records.

3.1 Data Visualization

To the purpose to have a different representation of the project data is needed only a class that extends the Command class. From a Command the rows, columns and cells can't be changed. Nevertheless it can be used either to summarize the project's data on whole or to get a different representation of the project's data.
To recap there are two main way to visualize data:

1. computing calculus with the purpose to display to the users the results.

For example computing mean, standard deviation, median, mode, interval, min and max of one specific column of the project's data.

2. displaying data in a different way. E.g. display data on a chart, on a graph or on a map.

These features often require a pre-processing phase to prepare the data to be displayed. In this phase the client side make an AJAX call to the server side. When the back end receives the request from the client side data are prepared and then sent back to the client side. When the pre-processed data are received, the front end manages the visualization phase.

Command This is the main class in Data Visualization Extension case. It's the class that receives the input from the client side, that process the data and that send back the output to the front end. Command is an abstract class that has just 2 attribute: the logger and RefineServlet. In addition the abstract Class requires many methods. The logger attribute is useful to print on the console and to trace the execution of the code. The methods that receives the request are the doPost and doGet method, as does almost every servlet in JAVA. After that there are the doPut and doDelete which concerns the request made by the client to put or delete a file in the server. Then there are other protected useful function to get more information about the project or from the request, like:

- getEngineConfig and getEngine, these methods allows to retrieve the information about the client faceted browsing state. To the purpose to easily get the filtered rows.
- getProject and getProjectMetaData, these methods allows to easily retrieve information about the project. The project object allows to easily access to the project data.
- getIntegerParameter and getJsonParameter, these methods allow to easily access to the parameter within the request
- respond methods, these methods manage the respond phase when the data processing phase is terminated.
- performProcessAndRespond, this method is called as default in the EngineDependentCommand abastract class which extends the Command class. The EngineDependentCommand manage the features that make changes on project data.

RowVisitor To use the RowVisitor from a class that extends a Command we need an engine object. The engine can be instanced when the project object is retrieved with the getProject function. To instance an engine object we can use the constructor passing the project object as argument.
When the engine object is instanced, we can set it up by using a JSONobject

where the engine configuration will be placed. To retrieve the engine configuration, a.k.a. the client facet browsing state, there is the getEngineConfig function. This method, given the request object, returns the configuration. When the configuration object is retrieved, we can initialize the engine with the function initializeFromJSON with the JSONobject as parameter. Now we can get the filtered rows by calling the Engine's getAllFilteredRows function. The getAllFilteredRows method returns an object that implements the FilteredRows interface. The FilteredRows interface requires the accept function that goes through the rows of the given project. This function determines which rows match and witch don't. On matching rows the visit method is called.

```
Project project = getProject(request);
Engine engine = new Engine(project);
JSONObject engineConfig = getEngineConfig(request);
engine.initializeFromJSON(engineConfig);
FilteredRows filteredRows = engine.getAllFilteredRows();
```

To use the accept function we need the project object that we already have retrieved and a row visitor that can perform operation on the matching rows. So we need to create a dedicated row visitor for our purpose. Our advice is to add a specific function that returns the RowVisitor. Inside the function can be defined the RowVisitor. A method that create a RowVisitor requires at least 2 parameter:

- project object parameter is mandatory. Due to the rowVisitor methods;
- cellIndex or columnName parameter can be useful in case the command requires a specific column;
- result object should contain the result of the computation.

```
protected RowVisitor createRowVisitor(Project project,
                                      String result) {
   return new RowVisitor() {
      String result;

      public RowVisitor init(String result) {
         this.result = result;
         return this
      }

      @Override
      public void start(Project project){ }

      @Override
      public boolean visit(Project project,
                           int rowIndex,
```

```
                        Row row) {
       .... do whatever you want...
              this.result =
                  this.result + ((Integer)rowIndex).toString();
              return false;
         }

         @Override
     public void end(Project project){ }
   }.init(project, result);
}
```

Now we can use the accept function of the filteredRows object. By this function the result of the computation on all rows in the "result" string can be stored.

```
String waitResult="";
filteredRows.accept(project,
                 createRowVisitor(project,
                                  waitResult));
```

In this example at the end of the computation in the "waitForResult" string there will be the concatenation of all row indexes.

Output Send back the result of the computation to the client side is quite easy. First of all, we have to construct a new JSONWriter. The constructor method of the JSONWriter requires a PrintWriter that can be obtained from the response. Indeed, the response provides the getWriter method that returns a PrintWriter object. So we can use it to construct our JSONWriter object. After that we can write in the response. To write with the JSONWriter there are some method to use:

- object, this method requires no parameter. It's important to start always with this function that creates an object. In the JSONWriter can be added many JSONobjects, but must all be nested to the first.
- key, this method requires a string parameter. It's the name that identifies a specific a value inside the object. Each object must have at least one key.
- value, this method requires just one parameter. The parameter could be a boolean, a double, a long and an Object. For each key in the JSONobject there must be either a value or an array.
- array, this method requires no parameter. The array function initialize an array within the object. Inside the array can be added either JSONobjects or values.

- endArray, this method requires no parameter. This function specify that the array is ended. If there are not ended arrays there will be an error on runtime.
- endObject,this method requires no parameter. This function specify that the object is ended. If there are not ended objects there will be an error on runtime.

For example:

```
JSONWriter writer =
         new JSONWriter(response.getWriter());
writer.object();
writer.key("manyObjects");
writer.array();
   for(int i=0; i<10;i++) {
      writer.object();
      writer.key("index");
      writer.value((Integer)i);
      writer.endObject();
   }
writer.endArray();
writer.endObject();
```

Example We want to create an extension which compute the mean of a certain column of all selected rows. We have to create a folder with the required structure and the module.properties ready.
After that we have to use the extension points and add our command and our JAVASCRIPT files in the controller.js file, like earlier explained (section 3.1). The name of our extension is "mean-extension", what a big imagination.
After that we suppose that the AJAX call to the command is triggered by clicking on a column header menu item.

```
DataTableColumnHeaderUI.extendMenu(
   function(column, columnHeaderUI, menu){
      MenuSystem.appendTo(menu,"",
      [ { /* separator */ }, {
         id : "mean-extension/mean",
         label: "Compute the mean",
         click: function () {
             new MeanDialog(column);
         }

      } ]);
});
```

In the MeanDialog will be shown the result of the row processing, but we have to ensure that javascript awaits for the server side response. For this

reason we have to deal with callbacks. Hereunder there is an example on how to manage the client side.

```
function MeanDialog(columnName){
    this.column = columnName;
    this.start();
}

MeanDialog.prototype.start = function() {
   var self = this;
   this.init(function() {
      this.dialogLevel =
          DialogSystem.showDialog(this.dialogElement);

      /* Set the label with the computed mean */
      $('.result', self.dialogElement).text(
                                      self.responseMean.Mean);
   })
}

MeanDialog.prototype.init = function(callback) {
   /* Inject the HTML page and store it in a global variable */
   ...

   /* Set labels */
   $('.column-name', dialogElement).text(this.column.name);

   /* Bind controls to actions */
   ...

   this.compute(callback);
}

MeanDialog.prototype.compute = function(initCallback) {
    var self = this;
    var data = { }; data["column"] = this.column.name;

    callback = {
         "onDone": function (response) {
             self.responseMean = response;
             if (initCallback)
                 initCallback.apply(self);
         }
     };

     Refine.postProcess('geo-extension', 'mean',
                        data, {}, {}, callback);
```

```
}
```

As you can see we have already supposed that the result is stored in the response in a attribute, called "Mean". So, on the server side, we must compute the mean of the column chosen by the user. So we have to implement a Command class that receives the request made by the client containing the column parameter, then compute the mean and finally send back the result.

```
public class ComputeMeanCommand extends Command {

   public void doPost(HttpServletRequest request,
                      HttpServletResponse response) {
      // Get the parameter within the request
      ...

      // Compute the mean by using the filteredRows
      // and the RowVisitor as explained before
      ...

      // Write in the response like explained in
      // the previous parameter
      ...
      }

   protected RowVisitor createRowVisitor(
               Project project, String result) {
      // As explained in the previous paragraphs
      ...
   }
}
```

3.2 Changing Project's Data

To the purpose to add features that allows to change from 0 to more rows we have to introduce Abstract Operations. These are the abstraction of processes that can be applied to different similar projects. To do this we still need a Command class. The command class must be the extension of the EngineDependentCommand Abstract Class. Each of this Command Class calls the createProcess method of the dedicated AbstractOperation object. The default createProcess method returns a QuickHistoryEntryProcess object, where data are processed. Inside the Process object the changes on project's data are stored in a dedicated object. When all rows processing is ended, the dedicated object containing the changes on project's data are passed to the Change Class. The Change Class concretely change the project's data by saving the changes computed within the process. After the changes are saved, the server side send to the client a message with the information that the computation is completed. So the client side calls many server side

command, in chronological order: getHistory command, the get-project-metadata, the get-models and the get-row. These functions allow to display on the client side the new saved project's data.

Command The EngineDependentCommand Abstract Class is the extension of the Command Abstract Class. Unlike the Command Abastract Class described in the previous subsection, there is an implementation of the doPost method. Furthermore the EngineDependentCommand provides the createOperation abstract method that must return an AbstractOperation. The "new" implementation of the doPost method does 4 things:

- getting the Project object and the engine configuration JSONobject from the request by using the already implemented functions of the command Abstract Class;
- create and construct a new Abstract Operation;
- create and construct a new Process, defined by the Abstract Operation;
- add the process to the processes queue. When the process is performed a history entry is added to the response object and sent to the client side;

So, the only method that must be implemented is the createOperation. In this method we have to do 2 things: take the parameters from the request, construct a new abstract operation by using the given arguments and return the just constructed Abstract Operation.

```
public class AbstractCommand extends EngineDependentCommand {
   @Override
   protected AbstractOperation createOperation(
         Project project,
         HttpServletRequest request,
         JSONObject engineConfig) {

      // Take parameters from the request
      ...

      // Construct a new AbstractOperation
      // with parameters taken from the request
      ...

      // Return the just constructed Operation
      ...
      }
}
```

The AbstractOperation The AbstractOperation Abstract class has no attribute and three default methods. The most important method is the createProcess one. This method returns a process given a Project object and a Properties object. The createProcess method is important because is called after the constructor from the EngineDependentCommand. This method returns a QuickHistoryEntryProcess object with an overridden method. The overridden method is the createHistoryEntry one. The overridden createHistoryEntry returns an HistoryEntry object by calling the createHistoryEntry of the Abstract Operation that implements the createProcess method.
There are three extra methods to implement in case we want to extends this class. One is due to the implementation of Jsonizable interface that requires the write method.
The write method is important in order to create an Abstract Operation that can be extracted from the history and reapplied to another similar project. Inside this method is important to add to the writer, passed as an argument, all the information to reconstruct the Abstract Operation. In case the AbstractOperation depends on the engine, also its configuration must be stored in the writer object.
There is one more method to add to every extension of the AbstractOperation: reconstruct method. This method is a static method and is called every time we want to apply an AbstractOperation extracted from the History. For this reason this method is strongly connected to the write method. The way method stores information affects deeply how the reconstruct method is implemented.
The second method is the getBriefDescription that given the project object returns the string that will be write in the History.

```
abstract public class AbstractOperation implements Jsonizable {

  public Process createProcess( Project project,
                    Properties options) throws Exception {

     return new QuickHistoryEntry(
               project, getBriefDescription(null)
               ) throws Exception {

        @Override
        protected HistoryEntry createHistoryEntry (
                  long historyEntryID) throws Exception {

           return AbstractOperation.this.
                createHistoryEntry(_project, historyEntryID);
        }
     }
  }

  protected HistoryEntry createHistoryEntry(Project project,
```

```
                        long historyEntryID) throws Exception{
      // to implement, the default method throws an Exception
   }

   protected String getBriefDescription(Project project) {
      // to implement, the default method throws an Exception
   }

   @Override //from the Jsonizable interface
   public void write(final JSONWriter writer,
               final Properties options) throws JSONException{
      // to implement
   }
}

   // This method is not required from the
   // AbstractOperation Abstract Class, but
   // each AbstractOperation extension needs
   // this method
   static public AbstractOperation reconstruct(
          Project project, JSONObject obj) throws Exception {
      // to implement
   }
```

This Abstract Class can be extended to perform operation on data, but it doesn't consider the Facet Browsing State of the client. In other words the operation overlooks if only a subset of the project's data are selected on the client Side. So, the operation will be performed on all project rows. If we want a operation that consider the Client Side Facet Browsing State we can extend the EngineDependentOperation Abstract Class.

Other AbstractOperation extensions There are two abstract extensions of the AbstractOperation: the EngineDependentOperation and the EngineDependentMassCellOperation.

- EngineDependentOperation: this Abstract Class is the extension of the AbstractOperation Abstract class. This class provides two methods that allows to manage the Client Side Facet Browsing State. Indeed, this Abstract Class aims to perform operation only on rows selected from the client side. Extending this class requires to call in the constructor method the super method passing the engineConfig JSONobject. This object must be passed as an argument from the command class calling the getEngineConfig() method.
  The two method provided from this class are the createEngine method and getEngineConfig(). The first method, given the project object, returns an initialized engine object. Whereas the second method returns

a JSONobject containing the engine configuration passed to the super method at the beginning.

- EngineDependentMassCellOperation: this Abstract Class is an extension of the EngineDependentOperation Abstract Class. This class has a ready made createHistoryEntry method. This abstract class can be useful in case we want to transform the content of cells in a specific column without changing rows and columns. This class provides two extra method signature to implement: createRowVisitor and createDescription. The createRowVisitor is very important because returns the RowVisitor which implements the visit method that will be called on each selected row. Indeed the createHistoryEntry method iterates on all filtered rows calling the method visit of the RowVisitor on the selected rows. After all rows are visited and a MassCellChange object is instanced by the createChange method. This method is based on the List of CellChange objects computed during the visit on rows and required from the RowVisitor.
  The createDescription method returns the string that will be shown to the user when the operation is terminated. The signature of the method are shown hereunder.

```
abstract protected RowVisitor createRowVisitor (
     Project project,
     List<CellChange> cellChanges,
     long historyEntryID ) throws Exception;

abstract protected String createDescription (
     Column column,
     List<CellChange> cellChanges );
```

The createHistoryEntry method The purpose of the createHistoryEntry method is to process data, construct a Change Object and return an HistoryEntry. To the purpose to process data there are two ways either use the filteredRows and a RowVisitor object or iterate on the project rows. These two methods are described in previous sections. However there is a matter to take in consideration. If processing a certain cell requires to make a request to another web service then we need to use a different type of process. In this case what is needed is to override the createProcess method instead of the createHistoryEntry method. To deal with this kind of cell processing see the process sub-sub-section.
During the data processing each result obtained must be stored in a certain object. In addition to the usual objects available in JAVA, OpenRefine provides some additional dedicated objects:

- Cell object requires two parameter to be created: a string value and a Recon. The value should be a string, but is possible to show an error in the cell. This is made possible passing as value an EvalError object.

To construct an EvalError object is needed a String which contains the message that will be shown in the cell. The Recon can be null if we are not reconciling our data.

- CellAtRow object contains its own a cell and its own row index. Indeed the constructor of a CellAtRow object requires the row index value as first parameter and a Cell object as second.
- CellChange object is an implementation of the Change Interface, but in case we are performing an EngineDependentMassCellOperation can be used to store the processing results. This object is described in the next sub-sub-section.

The construction of a Change Object is very important because is the object that concretely perform the changes, received from the Abstract Operation, on the project's data. There are many different ready-made change classes to use, but we can also implement our own custom Change Class.
Regarding the object that must be returned from the createHistoryEntry method, the object must be an instance of the HistoryEntry class. The returned HistoryEntry object should contain many object, among which: the AbstractOperation which has instanced the HistoryEntry itself, the project object containing the project information on which the Operation is performed, the operation description string, the HistoryEntryID and the change object. Hereunder there is an example of createHistoryEntry method:

```
 @Override
protected HistoryEntry createHistoryEntry(
           Project project, long historyEntryID
           ) throws Exception {
  Engine engine = createEngine(project);
  Column column =
     project.columnModel.getColumnByName(columnName);

  // ... check if the new Column name already exist ...
  ...

  Vector<CellAtRow> cellsAtRows =
     new Vector<CellAtRow>(project.rows.size());

  FilteredRows filteredRows = engine.getAllFilteredRows();
  filteredRows.accept(project,
                    createRowVisitor(project, cellsAtRows));
  String description = createDescription(columnName,
                                         cellsAtRows);

  Change change = new ColumnAdditionChange(newColumnName,
                        column.getCellIndex()+1,
                        CellsAtRows);
```

```
   return new HistoryEntry(
        historyEntryID, project, description, this, change);
}
```

The createRowVisitor method returns an initialized RowVisitor. The initialized RowVisitor in the visit method adds the result of the computation on a row within a CellAtRow. The Change object will be described in the Change classes sub-sub-section.

Adding Rows and Columns Adding rows or/and columns are needed two phases. First we have to create new cells and their values, then we have to add the cells concretely to the project's data. This distinction between the computation of the changes and the application of the changes is due to the history feature of OpenRefine.
The first phase can be performed either inside an AbstractOperation or inside a Process. Whereas, the second phase requires a Change Class. However these two phase are strongly connected. The way the new cells are stored in a variable affects deeply how the change can be implemented and vice-versa.

3.2.1 Change classes

The Interface In order to construct a change Object we need a Change Class. Every Change Class must implement the Change Interface. This interface requires 3 methods: apply, revert and save. The apply method is called two times:

- when the values computation is terminated and the operation has returned the history entry.
- when History Entry is reselected from the user into the project history displayed menu.

The aim of the apply method is to add to the project data the changes. On the other hand, the save method is called only when the computation is terminated. The save method is called if and only if that operation is extracted and reapplied to another project. Essentially the save method stores the change object.
The revert method is called when the user clicking on the History menu clicks an older History Entry.
The Change interface does not require other methods, but another method is mandatory: load method. When the change object is not in the main memory, the load method is called from the revert method to load the stored Change object. For this reason, the load and the save method are strongly connected. The way the the load method is implemented depending on how the Change object is saved. Furthermore, the aim of the load method is to initialize the change class with the values stored in the change object and

then applying it.

```
public interface Change {

   public void apply(Project project);

   public void revert(Project project);

   public void save(Project project);
}

// This method is not in the Change interface,
// because it is a static method.
// However each Change class must implement this method
static public Change load(LineNumberReader reader,
                          Pool pool) throws Exception;
```

Ready-made Change Implementation There are a lot of ready-made change object. You can find them all in the changes package within the model package. For this reason we group together the similar ones in order to rough out all change classes:

- CellChange, this is the simpler Change implementation. The constructor requires only the row index, the cell column index, the old Cell object and the new Cell object.
- Changes on columns, every Change column class extends the ColumnChange Abstract Class. This Abstract Class is an implementation of the Change interface, but it does not define any of the Change required methods. The ColumnChange Abstract Class has two extra implemented methods:
  - writeOldColumnGroups, this method requires 3 parameters: a Writer object, a Properties object and a list of ColumnGroup objects. Given parameters this method writes the old columns groups into the given Writer object.
  - readOldColumnGroups, this method requires 2 parameters: a LineNumberReader object and an int value containing the old Column Group count. This method returns a list of column group objects.

  There are many different change classes because of each one of it is made for a specific feature. For this reason not all are useful for extension writing. Hereunder there is a brief description for each class:

- ColumSplit, this change is based on a specific column from which more columns are created. The first needed parameter is the column name string on which the operation is based. The second parameter should contains a list of new column names string objects. The third parameter contains the list of the extended row indexes. The fourth parameter is a matrix containing a list of cells for each extended row. The fifth parameter is a boolean value that specifies if the original column must be removed or not.
  There is another constructor method that requires the previous introduced parameter and many other parameters in addition, this method is useful in case the fifth parameter is setted as true.
- ColumnAdditionChange, from one cell adds another column. For each row can be added only one cell. If are added more cells for rows only the last added cell is stored. This change class requires some parameters: the column name String object of the new Column, the column index integer value that represent the position of the new column and a list of CellAtRow objects that are the new cells to place in the new column.
- ColumnRemovalChange, ColumnRenameChange, ColumnMoveChange and ColumnReoderChange do not add cells. The ColumnRemovalChange requires only the column index value of the column to remove. Whereas, the ColumnRenameChange expects only the old column name within a String object and the new column name within a String object. On the other hand the ColumnMoveChange needs two parameter the column name String object to move and the new column index integer value. Last but not least, the ColumnReorderChange requires only the list of column names string following the new order.

- Massive changes this Change class perform changes on a great number of cells. With some of this change classes is possible to rewrite all rows and/or columns so pay attention when using them.
  - MassCellChange, requires three parameters: the first expected parameter is an array/list of cell changed, to store the change is used the CellChange class; the second expected parameter is the common Column name (optional) on which the change is performed if exists; the last parameter expects a boolean value, if setted as true then the project is immediately reinitialized with the new cells. This class applies changes on cells.
  - MassChange needs just two parameter: the first parameter expects a list of Change Object; the second expected parameter is a boolean value, if setted as true then the project is immediately reinitialized with the new cells. This class applies more change object that can be of different types.

- MassReconChange requires two map that each consists of a Long and a Recon object. The first map parameter is considered the map of new Recons, the second is considered the old Recon map. This change simply applies new Recons to the project.
- MassRowChange, this change class needs only one parameter: the new rows list. All the rows that aren't in this list are dropped.
- MassRowColumnChange requires two parameters: the new rows list and the new column list. All the rows and columns that aren't in this list will be removed
- ReconChange, this class extends the MassCellChange. This class as default calls the super constructor passing three parameters: the Cell changes list, the common column name and sets as false the third parameter. Indeed this class requires four parameters: the first two parameters as the MassCellChange class, the others needs Recon information. To be more precise require a ReconConfig object and a ReconStat object.

- Changes on rows this kind of changes do not add rows to the project. These classes are not useful for extension writing. The row change class available are four:
  - RowStarChange, this class requires just two parameters value: an integer and a boolean. The first stands for the row index and the second, if setted as true, stars the row.
  - RowFlagChange requires the same parameters as the previous change, but instead of starring the row the class flags the row.
  - RowReorderChange works exactly like the ColumnReorderChange. The only difference is that the list expected must contain Integer objects instead String objects.
  - RowRemovalChange, this class is similar to the ColumnRemovalChange, but it can remove a list of rows given a list of Integer objects.

If none of the ready-made Change Class that suits you, you can implement a new Change Class.

Creating a custom Change Class First of all, to use a custom Change Class we have to add it to the controller.js file by using the RefineServlet object. The RefineServlet object provides two methods:

- registerClassMapping method requires two string parameters: the first should be the name of the old class and the second the new class name. This method is usually used to update the old class names to newer class names, but we have to use it to make our custom change class name accepted. So the old class name that we pass as parameter don't really exist, because we are adding a new Change Class not updating an old one. Other extensions like freebase pass as first parameter

"com.google.refine.model.changes.DataExtensionChange", but also "foo" works. The second parameter requires the real packages and name of our custom class. For example:

```
refineServlet = Packages.com.google.refine.RefineServlet;
refineServlet.registerClassMapping(
   "foo", "org.extension.change.CustomChange");
```

- cacheClass method requires the new change class. This method allow OpenRefine to reach our custom Change Class. Indeed, our example continues like this:

```
refineServlet.cacheClass(
Packages.org.extension.change.CustomChange
);
```

Now we can use our custom Change Class. So now we can go through the change implementation. Imagine that we want a Chang Class that adds a column firstly. Then in the just created columns can be added one or more cells for each row depending on the matrix received. So one of the parameters will be a list of rows containing for each row a list of values.

- constructor. In this method the results of data processing are passed as arguments and stored in the class attributes. For example:

```
   private final int columnIndex;
   private final String newColumnName;
   private final String[][] results;
   private final Vector<Integer> rowAddedIds;

public CustomChange(final int columnIndex,
                    final String newColumnName,
                    final String[][] results) {
   this.columnIndex = columnIndex;
   this.newColumnName = newColumnName;
   this.results = results;
   this.rowAddedIds = new Vector<Integer>();

}
```

- apply method. First of all, everything we do in this method must be inside the JAVA synchronized statement. This statement ensure that threads do not interlope. Inside this method we have to add a column by using a ready-made Change Class: the ColumnAdditionChange. When applied this class, given some parameter that will be explained in the next paragraphs, adds a column. After the column addition is applied we need the column index, a.k.a. cell index. To obtain the

column index we recommend to use the getColumnByName method of the columnModel attribute of the project object. The getIndexByName of the columnModel attribute of the project object returns the position index instead of the column index identifier.
Now we have the cell index and we can add the rows where needed and put the results in cells. It's important that when rows are added to store in such attribute the added row ids. Otherwise when the revert method will be called the added rows will not be deleted.

```
public void apply(final Project project) {
   synchronized(project) {
     // create a CellAtRow List with the
     // size of the number of project rows
     int size = project.rows.size();
     Vector<CellAtRow> cells = new Vector<CellAtRow>(size);

     // add to the CellAtRow List an empty
     // cell for each row
     for(int i=0; i < size; i++)
        cells.add(new CellAtRow(i,null)) ;

     // construct the columnAdditionChange
     ColumnAdditionChange columnChange;
     columnChange = new ColumnAdditionChange(
            newColumnName, columnIndex, cells);

     // apply the change
     columnChange.apply(project);

     // get the index of the new column
     Column newColumn;
     newColumn =
       project.columnModel.getColumnByName(newColumnName);


     // check if the project rows are not empty
     if (project.rows.isEmpty())
        return;

     int rowCounter = 0;
     int cellInd = newColumn.getCellIndex();
     rowAddedIds.clear();

     // iterate on all row results
     for( String[] rowResults : results){
        int rowsToAdd;
```

```
        if (rowResults != null)
           rowsToAdd = Math.max(1, rowResults.length());

        // if there are more than one result for row
        // add empty rows and store the added row ids
        for (int i = 1; i < rowsToAdd;i++) {
           final Row resultRow =
                   new Row(cellInd + 1);
           final int rowId = rowCounter + 1;
           for (int j = 0; j < minRowSize; j++)
              resultRow.cells.add(null);
           project.rows.add(rowId, resultRow);
           rowAddedIds.add(rowId);
        }

        int i = 0;
        // sets the cells with the result
        for(String result : rowResults) {
           Row rowToSet = rows.get(rowCounter + i);
           rowToSet.cells.set(cellInd, rowResult.toCell(i));
        }

        rowCounter += rowsToAdd;
     }

     //when finished
     project.update();
   }
}
```

- save method is called after the apply method. In this method we have to write with the JSONWriter all the arguments received by the constructor. The JSONWriter as usual can be instanced by the writer parameter. In addition to the initial received arguments we have to write also the added row ids because when the revert method is called we have to know which rows have to be deleted. Otherwise when the revert method is called some empty rows will remain.

```
public void save(final Writer writer,
                 final Properties options)
                 throws IOException {
   // instance the JSONWriter
   JSONWriter json = new JSONWriter(writer);

   // create the Change Object
   json.object();
```

```
    // add all change attributes
    ...

    json.endObject();
}
```

- revert method requires the synchronized statement like the apply method. So all the operation must be inside the synchronized statement. First of all, we have to remove the added rows. To this purpose we can use the saved added row ids, iterating on them. After the rows are deleted we have to remove the column. Removing a column is quite easy, it's sufficient to call the ColumnRemovalChange.

```
public void revert(Project project){
    synchronized (project) {
    // iterate on the added rows
    // in order to delete each of them
    ...

    // delete the added column by
    // using the ColumnRemovalChange
    Change change = new ColumnRemovalChange(columnIndex);
    change.apply(project);

    project.update();
    }
```

- load method is called before the revert method when the Change object is not in the main memory. Firstly, this method must read the values saved in the Change object. Values can be read using a JSONTokener instanced with the reader passed as parameter. These parameters have to be used to construct the Change class and then return it.

```
static public Change load(LineNumberReader reader,
                          Pool pool) throws Exception {
    // instance the JSONTokener
    final JSONTokener tokener =
                    new JSONTokener(reader.readLine());
    // take the JSONObject which contains all saved values
    final JSONObject changeObject =
                    (JSONObject) tokener.nextValue();

    // take all values from the changeObject and
    // instance a new change Object
    final CustomChange thisChange;
    thisChange =
      new CustomChange(columnIndex, newColumnName, results);
```

```
// iterate on the added rows ids stored
// in the Change object and then add it to
// the just constructed Change
...

// return our constructed change.
return thisChange;
```

3.2.2 Processes

Inside OpenRefine there are two type of process: the QuickHistoryEntryProcess and the LongRunningProcess. The first type is the most used and it is dedicated to the operations that don't need to deal with request to external services. The second type is expected to make request to external services. This last process type requires to override the createProcess method because of the default returned object is the first type process. The LongRunningProcess let also to manage more than one thread. Both the LongRunningProcess and the QuickHistoryEntryProcess Abstract Class extends the Process Abstract Class. This Process Abstract Class defines 6 methods:

```
public abstract class Process implements Jsonizable {

   abstract public boolean isImmediate();

   abstract public boolean isRunning();

   abstract public boolean isDone();

   abstract public HistoryEntry performImmediate()
                throws Exception;

   abstract public void startPerforming(
                ProcessManager manager);

   abstract public void cancel();
```

QuickHistoryEntryProcess This is an extension of the Process AbstractClass. This class is also an Abstract Class and provides many methods:

- the constructor method of this class requires two parameter: project object on which the operation is performed and a string which should contain the description of the operation which is creating the process.
- the cancel method that returns a RuntimeException. This method is specific for the LongRunning processes.

- isImmediate method that returns true.
- isRunning method returns the same error of the cancel method. Also this method is specific for the LongRunning processes.
- startPerforming method same as isRunning.
- performImmediate method that calls the createHistoryEntry and returns an HistoryEntry. Before returning the HistoryEntry object this method sets the "done" attribute as true.
- write method writes a JSONObject that describes the process.
- isDone method returns the content of "done" attribute.
- createHistoryEntry method is an abstract method. Indeed as default each Abstract Operation overrides this method with its own createHistoryEntry() method. For this reason when we use a standard Abstract Operation we have to implement a new createHistoryEntry() method.

LongRunningProcess This type of process is used as mentioned above when we have to make request to other web services. This class is also an extension of the Process Abstract Class. For this reason implements the same methods with an addition:

- constructor method requires only a String value. This parameter should contain a description of the Abstract Operation.
- cancel method requires no parameters. This method sets the "canceled" attribute as true and stops the alive thread.
- isImmediate returns false.
- isRunning returns yes if there is a running thread.
- isDone returns true if there are threads, but neither is running.
- performImmediate returns a Runtime error. This method is specific for the QuickHistoryEntry processes.
- startPerforming calls the getRunnable() method in order to retrieve an object that implements the Runnable interface. After the thread is initialized the method starts the retrieved thread.
- write method writes a JSONObject that describes the process state.
- getRunnable is an abstract method. Inside OpenRefine all the implementation of this method are very simple just "return this;". Because of the classes that extend this Abstract Class usually implements also the Runnable interface. For this reason we introduce the method required by the Runnable interface: run method.

- run method is required from the Runnable interface. After the thread is started, this method is called . Inside this method data are processed and the result of computation is stored. Also, inside this method we can create one or more thread for each cell to compute.

For further information check the ColumnAdditionByFetchingURLsOperation class which is a good example. This class is in the column package within the operations package.

3.3 Adding Information to the project

OverlayModels In every project can be information that don't belong to a specific row. There are information which concern a certain column or a different representation of the project as a whole. For example the Freebase extension is based on the Protograph model. This kind of information can be stored inside an OverlayModel and can be used by the client side and by the server side. Each OverlayModel must be added into the controller.js file by using the registerOverlayModel method of the project object. This method requires two parameter: the identifier name String and the class that implements the OverlayModel interface. This interface extends the Jsonizable interface that requires the write method. In addition to the Jsonizable required method there are three more methods:

- onBeforeSave method is called just before the project is saved.
- onAfterSave method is called just after the project is saved.
- dispose method is triggered when the project is deleted.

In addition to the method required from the Jsonizable interface, each overlayModel requires two static methods. These methods are called when the project is loaded from the disk. The first method is the load method. The load method must return an instanced object which implements the OverlayModel interface. The OverlayModel object can be instanced by using the informations stored by the write method. To the load method is passed a JSONObject which contains the OverlayModel written information and the project object.
The second static method is the reconstruct method that given a JSONObject which contains the OverlayModel written information returns a Protograph instanced object.
Whereas, the write method is called by OpenRefine when the project is saved.
OverlayModels and AbstractOperation should be used together. Usually the OverlayModel is useful in order to compute some operation. On the other hand saving a change on an OverlayModel requires an AbstractOperation.

3.3.1 An example: Protograph

When a overlayModel is added to OpenRefine the overlayModel can be retrieved from the Project object. On the client side we can retrieve the

overlayModel by the overlaModels attribute of the theProject object. In Protograph case:

```
JSONObject Protograph =
   theProject.overlayModels.freebaseProtograph;
```

The Protograph overlay model is used by the FREEBASE extension. This overlay model is used by two features of this extension: the PreviewProtographCommand that allows to see a graph representation of data and by the ProtographTransposeExporter that allows to export the graph. There is another feature based on the Protograph overlay model: the SaveProtographCommand that allows to save the Protograph itself and to manage the changes on the Protograph.
If we look the client side, we notice that the Protograph is created before the save command is called. Although the Protograph is created on the client side, the Protograph is added to the overlayModels only after the save command call. The explanation is that firstly we have to apply the change and store it within a change object. In order to use the new overlay model only when it is really stored in the project.

```
SchemaAlignmentDialog.prototype._save = function(onDone) {
  var self = this;
  var protograph = this.getJSON();

  Refine.postProcess(
    "freebase",
    "save-protograph",
    {},
    { protograph: JSON.stringify(protograph) },
    {},
    {
      onDone: function() {
        theProject.overlayModels.freebaseProtograph
                              = protograph;
        ...

        if (onDone) onDone();
      }
    }
  );
};
SchemaAlignmentDialog.prototype.getJSON =
  function() {
     var rootNodes = [];
     ....
     return {
        rootNodes: rootNodes
     };
```

```
};
```

The save command instance an Abstract Operation object. This class provides some methods, all very simple. There are three main implemented methods in this class:

- the reconstruct method is just a call to the static reconstruct method of the Protograph class.
- the createHistoryEntry method creates just the change object using the protograph object, instanced in the Command class, and then return the HistoryEntry object.
- the write method that calls the write method of the protograph class and writes all the operation information within a JSONObject.

This abstract operation class contains a custom change class which allows to manage the Protograph updates:

- save method of the custom change class stores the old Protograph and the new Protograph using the write method of the Protograph object.
- apply and revert method just keep up to date the overlayModels attribute of the project object.
- load is a static method. This method, given a JSONObject, returns a Protograph object. This method is called from OpenRefine when the change object is not in the main memory.

Concerning the Protograph class is very simple. Each method required from the OverlayModel interface has an empty body. There are three important methods:

- reconstruct is an abstract method that returns a constructed Protograph.
- load, given a JSONobject and the project object, calls the reconstruct method.
- write, given a writer, writes a JSONobject that contains all information to construct a Protograph object.

3.4 Facets, Functions and Binders

First of all, an OpenRefine extension can't add customized facet, due to the way the software is implemented. For this reason the best that we can do is to use the ready-made facet with customized functions. Otherwise, if we add another facet, the facet will be never constructed from OpenRefine. OpenRefine provides two extension points: GREL functions and binders. Each function returns a value. The returned value can be a String, a number or a date. According to what is returned a different facet can be used. If we

return a number, we can use a range numeric facet. A scatterplot facet can be used, if we have at least two number returned (one for the x axis and one for the y axis). In order to return two numbers we can return two values converted to string and separated by something. A new variable type can be added by using binders. On the other hand if we return a date we can use a range date facet. Also a String can be returned and used with a list or text facet.
A binder is a new type of variable that can be used by a function in addition to the already existent type. The already existent types are: "value", "row", "cell", "cells", "recon" and "record. These type are really good described at this link https://github.com/OpenRefine/OpenRefine/wiki/Variables. Adding a new function type can be useful if in our extension we add information to the project.
So we can use these two extension point in order to create our custom expression. Then we can pass our custom expression to one of the available facet in order to have a selection based on custom criteria.

Functions Each function must be added in the controller.js file by the registerFunction method of the ControlFunctionRegistry class.

```
var grel = Packages.com.google.refine.grel;
var funcRegistry = grel.ControlFunctionRegistry;
funcRegistry.registerFunction("computeExamples",
    new Packages.example.functions.computeExamples()):
```

The first parameter is the name that must be used to call the function and the second is the class that implements the Function interface.
Each added function can be used from the Transform feature, from the Custom Facet features, from the Column Based Addition feature and more. Every function must implement the Function interface which implements the Jsonizable interface, these interfaces require two methods:

- call method. This method receives arguments, computes and returns a value. Passed arguments are stored inside an array of objects called "args". Every check must be done inside this method. When we are sure that the parameters are correct we can start computing. When the computation is terminated we can return the value, if there are no errors. In case of errors we can return a new EvalError. Hereunder there is the ToNumber function as example:

```
@Override
public Object call(Properties bindings, Object[] args) {
    if (args.length == 1 && args[0] != null) {
        if (args[0] instanceof Number) {
            return args[0];
        } else {
            String s = args[0].toString().trim();
            if (s.length() > 0) {
```

```
                try {
                   return Long.parseLong(s);
                } catch (NumberFormatException e) { }
                try {
                   return Double.parseDouble(s);
                } catch (NumberFormatException e) {
                   return
                    new EvalError("Cannot parse to number");
                }
            }
        }
    }
    return null;
}
```

- write method. This method returns a description of the function. Within this function we can write the information by a JSONWriter. The keys are: description, params and returns. Hereunder example is the continuation of the previous one.

```
@Override
public void write(JSONWriter writer, Properties options)
    throws JSONException {
    writer.object();
    writer.key("description");
    writer.value("Returns o converted to a number");
    writer.key("params"); writer.value("o");
    writer.key("returns"); writer.value("number");
    writer.endObject();
}
```

Binder In order to add a new Binder we must add it in the controller.js as usual. To register a Binder requires the registerBinder method of the ExpressionUtils class. Hereunder there is an example (recalling the official documentation).

```
Packages.com.google.refine.expr.ExpressionUtils.registerBinder(
      new Packages.com.foo.bar.MyBinder());
```

Each Binder we want to add must implement the Binder interface. This interface requires two methods:

- the initializeBindings method requires two parameters a Properties object and the project object. This method is called always before the bind method. The project object contains all the project data.
- the bind method expects 5 parameters: a Properties object, a Row object, the row index, the column name and the cell object. All this

parameters, except the Properties object, concern the specific value on which we are performing the computation.

The main differences between the two methods are the available parameters. For example if you want to add a variable that needs the attributes of the project class, you can use the first method. Whereas if your new variable is based on the current row, cell or column, the second method is better.
Regardless the method chosen, the new variables have to be added to the Properties object. Indeed the properties object is an hashmap, so each addition to it requires a key String and an object. The key String is the name to use to retrieve the object.

```
public class tryBinder implements Binder, HasFields{
Project project;
    @Override
    public void initializeBindings(Properties bindings,
                                   Project project) {
        this.project = project;
        bindings.put("projectInfo", this);
    }
    @Override
    public void bind(Properties bindings, Row row,
             int rowIndex, String columnName, Cell cell) {
    }
}
```

We recommend to pass an object that implements the HasFields interface. This is due to the fact that in GREL every attribute of the available variables can be retrieved either with a dot between the variable and its attribute or adding square brackets which contains the name of attribute between “” after the variable. E.g. “row.cells” or row[“cells“]. To make sure that also our object has the same behaviour we have to implement the HasFields interface.
The HasFields interface requires two methods:

- getField which is called when we write something after a dot attached to a variable name or within square brackets attached to a variable name. The continuation of the previous example:

```
public Object getField(String name, Properties bindings) {
   if (name.equals("projectName""))
      return project.getMetaData().getName();
   else if (name.equals("overlayModels"))
      return project.overlayModels.keySet().toArray();
   else if (name.equals("rowSize"))
      return project.rows.size();
   else
      return "not available field";
}
```

- fieldAlsoHasFields that given a String name returns a boolean value. This method is triggered when we call the hasField method on a certain variable and on a certain string. Passing as arguments the variable and the string name of the required field the method return true if the required field has also fields. To complete the example:

```
public boolean fieldAlsoHasFields(String name) {
   return false;}
```

3.5 A New Scripting Language

To add a new scripting language we have to call the registerLanguageParser method of the MetaParser class. Recalling the official documentation, this method requires four parameters: the first parameter is the identifier name, the second parameter needs the name of the scripting language for the user, the third requires an instanced object of the scripting language and the fourth is the default string. The instanced object of the create language must return an instance of the LanguageSpecificParser interface.

```
var MP = Packages.com.google.refine.expr.MetaParser;
MP.registerLanguageParser(
   "unique-name",
   "User friendly name",
   Packages.org.foo.exten.FooEvaluable.createParser(),
   "default string");
```

LanguageSpecificParser and Evaluable interface The LanguageSpecificParser interface requires just one method. The evaluate method that given a String must return an object that implements the Evaluable interface. The Evaluable interface needs the evaluate method that must return an object, given the Properties object.

```
public class FooEvaluable implements Evaluable{
   static public LanguageSpecificParser creatParser() {
      return new LanguageSpecificParser() {
         @Override
         public Evaluable parse(String s)
                        throws ParsingException {
            return new FooEvaluable(s);
         }
      }
   }
   public Object evaluate(Properties bindings) {
      // compute or retrieve the result
      ...
          // return the result
          ...
```

```
}
```

Example: Jython An example of an added scripting language is the Jython extension. The main class of this extension is the JythonEvaluable class. Indeed is this class that returns an instance of the LanguageSpecificParser. The method that does it is the createParser method, inside this function the parse method of the LanguageSpecificParser is implemented. The implementation of the method consist of returning an instanced JythonEvaluable class with the string received from the parse method. Please note that the string received from the parse method should be a piece of python code.
Inside, the constructor of the JythonEvaluable class the received string is adapted to be executed by the python interpreter. After that the modified string is passed as argument to the python interpreter which executes the string.
The evaluate method that must be implemented by the JythonEvaluable class gets the result from the python interpreter engine and then return a JAVA fitted result.

3.6 New file types - Importers and Exporters

Inside OpenRefine every time we want to work on a specific file or on a set of file we have to create a project. For this reason the project creation assumes that a file or a set of files are loaded and parsed. Each file type requires a different parsing technique. The parsing technique for a certain file, firstly, is based on the file extension and then try to guess better format based on the content. When the parsing technique is chosen, a preview of the will be project is shown to the user. If the user likes the preview, the user can create the project.
When the project is ready, the user will operate operations on it and data will be changed. If the user wants to export some changed data can use an exporter. This is a simple class that allows the user to extract the changed data with the same type as beginning or with another type. OpenRefine allows to add new exporters and importers, in order to create projects from different types.

3.6.1 Importers

How project creation works In order to create a project what is needed is to import a file. The project creation feature requires more than one step. The main commands used by this feature are basically two: the createImportingJobCommand and the ImportingControllerCommand. The first command creates a unique id for the importing job, adds the importing job to the queue managed by the Importing Manager and puts the job id into the response. When the client side receives the just created job id calls

the ImportingControllerCommand with a specific subcommand. This command will be called at least four times before the computation terminates and the Software display to the user a preview of the importation job. When the preview is ready the user is able to choose between creating the whole project, starting over to select a different file and changing the automatic selected parser. After that the user can create the project by clicking on the dedicated button, so the ImportingControllerCommand will be called and the last creation step processed. While the server side creates a preview of the imported file the client side calls the GetImportingJobStatus command in order to know if the importing job is running or not. Firstly, the ImportingControllerCommand gets from the request the name of the chosen ImportingController that will manage the whole importation feature. Then the ImportingController object is got by the ImportingManager. When the ImportingController is retrieved, the command calls the doPost method of the ImportingController object. Inside the default ImportingController's doPost method there is a check on the subcommand within the request, according to the subcommand a different operation will be performed. The possible subcommand are five:

1. "load-raw-data": in this phase, information about file or about the set of files are loaded and a first format is determined for the file. To determine which format suits better the file there are some checks. The first check is based on the file extension, the ImportingController calls the ImportingManager object. First of all, it looks for the registered extension. Then the ImportingManager searches for the registered Mime type. The last is preferred because is more specific. After that the ImportingController try to figure out the most common format for the set of file. If the set of file are lesser then two, this check is trivial. Afterwards the ImportingController try to guess a better format based on the file content. This operation is performed trying to run each registered Format guesser. If a Format guesser recognize a file type, this is the format parser chosen. When a Format parser is chosen, it is stored in a JSON containing all the information about the file or the set of files. If none of the checks returns a specific format, by default the format parser is a "text/line-based" parser.

2. "update-file-selection": this phase is specific for set of files, e.g. import a zip or a tar archive. When the user chooses the files within the archive to import this method is called in order to choose the new most common format.

3. "initialize-parser-ui": in this phase the parser's UI is initialized by calling a parser's method: createParserUIInitializationData. Inside this method the custom parser options are setted and returned within a JSONobject. Then the JSONobject is added to the response. In this way the client side can show specific options for each parser type. Please note that each parser requires a dedicated interface. This phase could

more complicated in case of parser based on importing trees, like JSON or XML.

4. "update-format-and-options": in this phase an import preview to show to the user is created. So the ImportingController object through the ImportingUtilities object calls the Format parser's parse method with a limit to the first 100 lines. After this operation is performed the results are displayed.

5. "create-project": this subcommand can be triggered from the user, after the preview is displayed. The ImportingController by the ImportingUtilities object call the parse method of the Format parser with limit equal to "-1", this means that the Parser is called on all records.

Importer Extension Points So, if we want to import a new file type we need a new parser that probably will require dedicated options. To add a new importer OpenRefine provides many methods of the ImportingManager class. This object stores the registered format parser, file extension, Mimetype, format guesser and importing controller.
ImportingManager is the class that allows to add importers. The ImportingManager provides 5 methods:

- registerFormat method, there are three implementation of this method. With this method a new type parser can be added.
  The first implementation of this method requires just 2 parameters, but allows only to add the format name and a label. It is made just to add generic format. Those which remain are different for a boolean parameter. In non-boolean parameter case is setted as default as true. In the other implementation the boolean parameter value can be chosen. In case we set it up as "false", this value can affect the "load-raw-data" and the "update-file-selection" phases for a set of files. To be more precise the rankFormats method of the ImportingUtilities class use this attribute. Anyhow the most used method is the one with "true" as default.
  The other required parameters are a format name, a label to display to users, a UI class name and an instanced Importing Parser. Concerning the UI class name you can use the ready-made UI JAVASCRIPT classes or implement a new custom one. About the instanced Importing Parser is important that implements the ImportingParser interface.

```
var importing = Packages.com.google.refine.importing;
var IM = importing.ImportingManager;
IM.registerFormat(
      "generic-form/foo", //or whatever unique name
      "A fancy User-friendly label",
      "FooParserUI",
      new Packages.org.foo.parser.FooImportingParser()
      );
```

- registerExtension method that link your parser to a specific file extension. This method requires two parameters: extension string and format name:

```
IM.registerExtension(".pdf", "generic-form/foo");
```

- registerMimeType method that maps the importer with the mime type. This method needs two parameters the string of the declared MimeType and the format name:

```
IM.registerMimeType("application/pdf", "generic-form/foo");
```

- registerFormatGuesser method that allows to add a specific class that will check the content of a file to determine the content file fitted Format Parser. This method requires two parameters: the format name derived either from the MimeType or from the extension and an instanced object that implements the FormatGuesser interface.

```
IM.registerFormatGuesser("found-format",
   new Packages.org.foo.FooFormatGuesser() );
```

- registerController method that allows to add a new Importing Controller. By default the Refine client side calls the default Importing Controller. This implies that each Importing Controller requires a dedicated UI. This method requires three parameters: the module (available as default), the name of the new importing controller and an instanced object that implements the ImportingController interface.

```
var controller = new Packages.org.FooImportingController();
IM.registerController(module,
                    "foo-importing-controller",
                    controller)
```

Parser UI First of all, to implement a new custom UI what is needed is to add the javascript file which contains the UI class to the index.

```
var rM = Packages.com.google.refine.ClientSideResourceManager;
rM.addPaths("index/scripts", module,
           [ "module-relative-path/foo-Parser-UI.js" ] );
```

After that the UI class must implement some methods to interplay with the Server side. This implies that how the Server Side stores the output is important for the Client Side. OpenRefine calls four methods of UI classes:

1. the constructor method: this method manage the interface initialization and the preview table creation. During the interface initialization the HTML elements are setted up. After that the _updatePreview method is called.

```
Refine.FooParserUI = function(controller, jobID, job,
      format, config, dataContainerElmt,
      progressContainerElmt, optionContainerElmt) {
   // initialize interface and variables
   ...

   this._updatePreview();
```

2. the _updatePreview method is called when the "Update Preview" button is clicked. It can be also implemented a function that triggers this method when there are changes on options. Most of UI ready-implemented classes call the _updatePreview method when there are changes on options.
   First of all, creates the progress bar. After that the UI calls the ImportingController's updateFormatAndOption method. This method make a request to the server side with subcommand equal to "update-format-and-option" with the options setted by the user on the client side. Afterwards, if this operation returns a "ok" status, this means that preview data are accurately created. So, the UI displays the Preview Table.

3. the dispose method: triggered when the user clicks on the "Start Over" button. In most UI class implementation this method calls the clearTimeout method of the window object.

4. the confirmReadyToCreateProject method that triggers the creation of the project. This method must return a boolean value. If the returned value is true, the creation project phase on the Server Side is triggered.

```
Refine.FooParserUI.prototype._updatePreview = function() {
...
}

Refine.FooParserUI.prototype.dispose = function() {
...
}

Refine.FooParserUI.prototype.confirmReadyToCreateProject =
                                          function() {
...
return true
}
```

When all these methods are defined by the UI class we can add it to Refine.

```
Refine.DefaultImportingController.parserUIs.FooParserUI =
                                     Refine.FooParserUI;
```

ImportingParser Interface This interface requires two methods: createParserUIInitializationData and parse. The first method is called by the DefaultImportingController soon after the parsing technique is determined. This method must return a JSON object containing the options for the UI on the client side. Whereas, the parse method must parse each uploaded file and store rows and columns into the project object passed as parameter. From the OpenRefine's source code, hereunder there is the ImportingParser interface:

```
public interface ImportingParser {
    /**
     * Create data sufficient for the parser UI on the
     * client side to do its work.
     * @return JSONObject options
     */
    public JSONObject createParserUIInitializationData(
        ImportingJob job,
        List<JSONObject> fileRecords,
        String format
    );

    /**
     * @param project
     * store the row and column model inside this object
     * @param limit
     * maximum number of rows to create
     * @param options
     * custom options put together by the UI
     * corresponding to this
     * parser, which the parser should understand
     * @param exceptions
     * list of exceptions thrown during the parse.
     * Expects an empty List as input to which
     * it can append new Exceptions thrown
     */
    public void parse(
        Project project,
        ProjectMetadata metadata,
        ImportingJob job,
        List<JSONObject> fileRecords,
        String format,
        int limit,
        JSONObject options,
        List<Exception> exceptions
    );
}
```

In order to make the developer's life easier, OpenRefine provides an AbstractClass that implements the ImportingParser interface: ImportingParserBase. This AbstractClass implements the parse method. The implementation of the parse method manages the iteration on all uploaded file. This implies that the developer has to implement a function to parse each file of the same format separately. To this purpose the ImportingParserBase during the iteration calls one of parseOneFile method implementation. Inside this Abstract Class there are two functionsparseOneFile methods. These methods differ on how the file are read. The first uses a Reader and the second uses an InputStream. An InputStream reads binary data, whereas a Reader reads characters. The argument passed to the constructor of the ImportingParserBase determines which method is called. If the argument is true, the parse method uses the InputStream and vice-versa.

Ready-implemented ImportingParserBase OpenRefine contains three implementation of the ImportingParserBase: the RdfTripleImporter, the TabularImportingParserBase and the TreeImportingParserBase. All the other parser within OpenRefine are based on this three ImportingParserBase:

- the RdfTripleImporter. This importer is very specific and implements the parseOneFile method. Its constructor accepts only one parameter: mode. This parameter is an enum value and calls a different parsing method depending on the passed argument.

- the TabularImportingParserBase. This is the most used extension of the ImportingParserBase class. In spite of this is an Abstract Class since this implementation does not implements the parseOneFile methods. This class provides a readTable method that given some parameters and an implemented interface fills the project object.
  The readTable method requires almost all parameters required by the parseOneFile method. First of all, this method checks if there are more than one file, in this case the first column is the "file" column. After that starts a loop on the rows returned from the implemented getNextRowOfCells method until it returns null. Inside the cicle, the method creates the column model, if there is an header. Afterwards the method starts creating rows. At the end of the loop the project object will be filled with the rows returned from the getNextRowOfCells method. Hereunder there is the readTable method signature.

```
static public void readTable(
       Project project,
       ProjectMetadata metadata,
       ImportingJob job,
       TableDataReader reader,
       String fileSource,
       int limit,
       JSONObject options,
```

```
        List<Exception> exceptions)
```

Before using the readTable method is needed to implement the TableDataReader interface that contains the getNextRowOfCells method. Hereunder an example of interface implementation.

```
TableDataReader dataReader = new TableDataReader() {
   @Override
   public List<Object> getNextRowOfCells()
                          throws IOException {
      String line = lnReader.readLine();
         if (line == null) {
            return null;
         } else {
            return getCells(line, parser, lnReader);
         }
   }
};
```

Moreover this Abstract Class provides an implementation of the createParserUIInitializationData method. That calls the super method and sets some options with default values.

- the TreeImportingParserBase. This importer is used from the XMLImporter and from the JSONImporter. This Abstract Class provides a different implementation of the parse method and an implementation of the createParserUIInitializationData method. The TreeImportingParserBase use a different implementation of the parse class to open the possibility of using a TreeReader instead of a normal Reader. Furthermore when files are imported by extending this Abstract Class the user could see in the project page groups of columns. This Abstract Class does not implement the parseOneFile method with an InputStream as parameter instead of the TreeReader. Whereas implement the parseOneFile method with the TreeReader. The TreeReader is an Interface that requires many methods to scan a file. This interface is used by the importers in OpenRefine to construct a TreeReader from an InputStream(e.g. JsonImporter and XmlImporter).

Format guesser A Format Guesser is always called after that the format of a file is determined either from the extension or the Mime type to ensure that the content follow the format declared. A Format Guesser must implement the guess method that requires three parameters: the file, the encoding and the seedFormat, i.e. the format to check. When a Format Guesser is registered is placed at the first place in list of the Guessers, in order to give major priority to the new Guessers. Along these lines if the new Guesser find a better matching format, the older Guesser are not even called. The guess

method must return a string that identifies a specific registered Format in case a Format is guessed, otherwise null.

```
public class LineBasedFormatGuesser implements FormatGuesser {

    @Override
    public String guess(File file,
                        String encoding,
                        String seedFormat) {
    String result = null;

    // try to guess the format
    ...

    return result;
```

Importing Controller UI There are no specific rules to follow. This interface implementation is strongly connected to the implementation of the ImportingController interface. Seeing as the UI has to communicate with the ImportingController on the server side. Adding a new User Interface for a feature that does not concern project data requires the addition of an Action Area. Thus, first of all we have to add a file to the path in the index.

```
var refine = Packages.com.google.refine
var rM = refine.ClientSideResourceManager;
rM.addPaths("index/scripts", module,
        [ "module-relative-path/foo-Controller-UI.js" ] );
```

Inside the added file, our custom UI class has to be implemented. Writing a new UI class requires the two method described in the Interface section: the constructor and the resize method one.
Thereafter we can add our UI to the main page menu. To do this we need to add a new Action area, as described in the Front End section. You can find the default UI class for projects creation at the path main/webapp/modules/scripts/index/create-project-ui.js.

Importing Controller Interface First of all, adding a new Importing-Controller requires a command that triggers the controller. We always need a Command to trigger the Server Side from our UI on the client side. For this reason, OpenRefine provides an ImportingControllerCommand that parse the request parameter looking for a "controller" name property. If the request contains a controller's name, the ImportingControllerCommand gets the controller from the ImportingManager object. When the controller object is retrieved calls its doPost method.
The Importing Controller Interface requires three methods:

- init method which is called soon after the registerController method of the ImportingManager object. For example the DefaultImportingController use this method to initialize its servlet attribute.

- doGet method which is called by requests made in get. If you want to use your ImportingController through the ImportingControllerCommand this method will not be used. Anyhow, in this method the file could be uploaded and then used to create the project. Depending on how the UI is implemented could be useful to communicate to the Client Side when finished or middle passages.

- doPost method which is called by requests made in post. If you use the ImportingControllerCommand, this method is called. Inside this method the file can be uploaded. When the data is loaded the appropriate format parser can be setted up and sent to the client side. The client side can make another post request either to have a preview or create the project. After that the Server side can create a response containing the request data.
  Whatever operation you decide to perform its important to send always a response to the Client Side.

3.6.2 Exporter

In OpenRefine projects can be exported either in its entirety or a part of it. First of all, we have to register the Exporter. To register an Exporter we have to use the registerExporter method of the ExporterRegistry object. This method requires two parameters: the format name and an instanced object that implements the Exporter interface. The format name must be unique and it will be used both on server side and on client side. Hereunder there is an example.

```
var exporters = Packages.com.google.refine.exporters;
var ER = exporters.ExporterRegistry;
ER.registerExporter("format-name",
     new Packages.org.foo.MyExporter());
```

When our exporter is added to the ExporterRegistry we have to add it to the Exporter's menu. To do this we have to add a new JAVASCRIPT file to the path. Afterwards we have to push our JSONObject containing the menu item.

```
ExporterManager.MenuItems.push(
{
   "id" : "foo-extension/export",
   "label" : "Fancy user friendly Exporter",
   "click": function() {
      ExporterManager.handlers.exportRows("format-name",
                                          "extension");
   }
```

```
}
```

Please notice that for the "click" value we have used a method of an object within the ExporterManager. This object provides many methods to export project. The exportRows method exports the project in its entirety. If there are no arguments to pass to the Server Side, the exportRows method does its duty. However we can create our dedicated custom exporter's UI through a dialog. The ExporterManager class is available at the path webapp/modules/core/scripts/project/exporter.js. An example of dedicated exporter's UI is reachable at the path webapp/modules/core/scripts/dialogs/custom-tabular-exporter-dialog.js.

Exporter Interface The Exporter Interface requires only one method: getContentType. This method, without parameters, must return the declared Mime type managed by the Exporter. There are three extension of this interface within OpenRefine: StreamExporter Interface, UrlExporter Interface and WriterExporter Interface. Each extension of the Exporter Interface requires an export method. The first three parameters are the same for each extension. The main difference is in the fourth parameter that represent the way the output file is written. The StreamExporter is based on an OutputStream class, whereas the WriterExporter is based on a Writer class and so on.