



Installing OpenRefine

System requirements

OpenRefine does not require internet access to run its basic functions. Once you download and install it, it runs as a small web server on your own computer, and you access that local web server by using your browser. It only requires an internet connection to import data from the web, reconcile data using a web service, or export data to the web.

OpenRefine requires three things on your computer in order to function:

Compatible operating system

OpenRefine is designed to work with **Windows**, **Mac**, and **Linux** operating systems. [Our team releases packages for each.](#)

Java

Java must be installed and configured on your computer to run OpenRefine. The Mac version of OpenRefine includes Java; new in OpenRefine 3.4, there is also a Windows package with Java included.

If you want to install Java yourself, you can install a pre-built Java Runtime Environment (JRE) from [Adoptium.net](#). Please note that OpenRefine works with Java 11 to Java 17 for OpenRefine 3.7.

If you install and start OpenRefine on a Windows computer without Java, it will automatically open up a browser window to this page.

Compatible browser

OpenRefine works best on browsers based on WebKit, such as:

- [Google Chrome](#)
- [Chromium](#)
- [Opera](#)
- [Microsoft Edge](#)
- [Safari](#)

We are aware of some minor rendering and performance issues on other browsers such as Firefox. We don't support Internet Explorer. If you are having issues running OpenRefine, see the [section on Running](#).

Release versions

OpenRefine always has a [latest stable release](#), as well as some more recent developments available in beta, release candidate, or [snapshot releases](#). If you are installing for the first time, we recommend [the latest stable release](#).

If you wish to use an extension that is only compatible with an earlier version of OpenRefine, and do not require the latest features, you may find that [an older stable version is best for you](#) in our list of releases. Look at later releases to see which security vulnerabilities are being fixed, in order to assess your own risk tolerance for using earlier versions. Look for “final release” versions instead of “beta” or “release candidate” versions.

Unstable versions

If you need a recently developed function, and are willing to risk some untested code, you can look at [the most recent items in the list](#) and see what changes appeal to you.

“Beta” and “release candidate” versions may both have unreported bugs and are most suitable for people who are willing to help us troubleshoot these versions by [creating bug reports](#).

For the absolute latest development updates, see the [snapshot releases](#). These are created with every commit.

Installing or upgrading

Back up your data

If you are upgrading from an older version of OpenRefine and have projects already on your computer, you should create backups of those projects before you install a new version.

First, [locate your workspace directory](#). Then copy everything you find there and paste it into a folder elsewhere on your computer.

For extra security you can [export your existing OpenRefine projects](#).

CAUTION

Take note of the [extensions](#) you have currently installed. They may not be compatible with the upgraded version of OpenRefine. Installations can be installed in two places, so be sure to check both your workspace directory and the existing installation directory.

Install or upgrade OpenRefine

If you are upgrading an existing OpenRefine installation, you can delete the old program files and install the new files into the same space. Do not overwrite the files as some obsolete files may be left over unnecessarily.

CAUTION

If you have extensions installed, do not delete the [webapp\extensions](#) folder where you installed them. You may wish to install extensions into the workspace directory instead of the program directory. There is no guarantee that extensions will be forward-

compatible with new versions of OpenRefine, and we do not maintain extensions.

[Windows](#) [Mac](#) [Mac via Homebrew](#) [Linux](#)

Once you have downloaded the [.zip](#) file, extract it into a folder where you wish to store program files (such as [D:\Program Files\OpenRefine](#)).

You can right-click on [openrefine.exe](#) or [refine.bat](#) and pin one of those programs to your Start Menu or create shortcuts for easier access.

Once you have downloaded the [.dmg](#) file, open it and drag the OpenRefine icon onto the Applications folder icon (just like you would normally install Mac applications).

The quick version:

1. Install [Homebrew](#)
2. In Terminal enter [brew install --cask openrefine](#)
3. Then find OpenRefine in your Applications folder.

The long version:

[Homebrew](#) is a popular command-line package manager for Mac. Installing Homebrew is accomplished by pasting the installation command on the Homebrew website into a Terminal window. Once Homebrew is installed, applications like OpenRefine can be installed via a simple command. You can [install Homebrew from their website](#).

Install

Install OpenRefine with this command:

```
brew install --cask openrefine
```

You should see output like this:

```
==> Downloading https://github.com/OpenRefine/OpenRefine/releases/
download/3.4.1/openrefine-mac-3.4.1.dmg
#####
100.0%
==> Installing Cask openrefine
==> Moving App 'OpenRefine.app' to '/Applications/OpenRefine.app'
🍺 openrefine was successfully installed!
```

Behind the scenes, this command causes Homebrew to download the OpenRefine installer, verify the file's authenticity (using a SHA-256 checksum), mount the disk image, copy the [OpenRefine.app](#) application bundle into the Applications folder, unmount the disk image, and save a copy of the installer and metadata about the installation for future use.

If an existing [OpenRefine.app](#) is found in the Applications folder, Homebrew will not overwrite it, so installing via Homebrew requires either deleting or renaming previously installed copies.

Uninstall

To uninstall OpenRefine, paste this command into the Terminal:

```
brew uninstall --cask openrefine
```

You should see output like this:

```
==> Uninstalling Cask openrefine
==> Backing App 'OpenRefine.app' up to '/usr/local/Caskroom/openrefine/
3.4.1/OpenRefine.app'
==> Removing App '/Applications/OpenRefine.app'
==> Purging files for version 3.4.1 of Cask openrefine
```

Update

To update to the latest version of OpenRefine, paste these two commands into the Terminal:

```
brew update  
brew upgrade
```

You should see output like this:

```
==> Upgrading 1 outdated package:  
openrefine 3.4.0-> 3.4.1  
==> Upgrading openrefine  
==> Downloading https://github.com/OpenRefine/OpenRefine/releases/  
download/3.4.1/openrefine-mac-3.4.1.dmg  
#####
100.0%  
==> Backing App 'OpenRefine.app' up to '/usr/local/Caskroom/openrefine/  
3.4.0/OpenRefine.app'  
==> Removing App '/Applications/OpenRefine.app'  
==> Installing Cask openrefine  
==> Moving App 'OpenRefine.app' to '/Applications/OpenRefine.app'  
🍺 openrefine was successfully upgraded!
```

If you had previously installed the `openrefine-dev` cask (containing a release candidate) and you want to move to the stable release, you need to first uninstall the old cask and then install the new one:

```
brew uninstall --cask openrefine-dev  
brew install --cask openrefine
```

Once you have downloaded the `.tar.gz` file, open the command line (e.g., "Terminal") and navigate to the directory where the file is located.

For instance, if the file is in your `Downloads` directory, use the `cd` command to navigate there.

```
user@machine:~$ cd Downloads/
```

Next, extract the `.tar.gz` file. To extract to the same directory, use the following command (which you will need to adapt depending on the version of OpenRefine you are installing):

```
user@machine:~/Downloads$ tar xzf openrefine-3.6.2-linux.tar.gz
```

You can then go inside the directory just created by this extraction (again, to be adapted depending on the version):

```
user@machine:~/Downloads$ cd openrefine-3.6.2
```

And finally start OpenRefine with:

```
user@machine:~/Downloads/openrefine-3.6.2$ ./refine
```

Set where data is stored

OpenRefine stores data in two places:

- program files in the program directory, wherever it is you've installed it
- project files in what we call the "workspace directory."

You can access the workspace directory by:

- launch OpenRefine and click `Open Project` in the sidebar
- At the bottom of the screen, click `Browse workspace directory`
- A file-explorer or finder window will open in your workspace

By default its location is:

Depending on your version of Windows, the data is in one of these directories:

- `%appdata%\OpenRefine`
- `%localappdata%\OpenRefine`
- `C:\Documents and Settings\user id\Local Settings\Application Data\OpenRefine`
- `C:\Users\user id\AppData\Roaming\OpenRefine`
- `C:\Users\user id\AppData\Local\OpenRefine`
- `C:\Users\user id\OpenRefine`

For older Google Refine releases, replace `OpenRefine` with `Google\Refine`.

You can change this by adding this line to the file `openrefine.14j.ini` and specifying your desired drive and folder path:

```
-Drefine.data_dir=D:\MyDesiredFolder
```

If your folder path has spaces, use neutral quotation marks around it:

```
-Drefine.data_dir="D:\My Desired Folder"
```

If the folder does not exist, OpenRefine will create it.

```
~/Library/Application\ Support/OpenRefine/
```

For older versions, as Google Refine:

```
~/Library/Application\ Support/Google/Refine/
```

Logging is to `/var/log/daemon.log` - grep for `com.google.refine.Refine`.

```
~/.local/share/openrefine/
```

You can change this when you run OpenRefine from the terminal, by pointing to the workspace directory through the `-d` parameter:

```
./refine -p 3333 -i 0.0.0.0 -m 6000M -d /My/Desired/Folder
```

Logs

OpenRefine does not currently output an error log, but because the OpenRefine console window is always open (on Linux and Windows) while OpenRefine runs in your browser, you can copy information from the console if an error occurs.

Using a Mac, you can [run OpenRefine using the terminal](#) in order to capture errors.

Increasing memory allocation

OpenRefine relies on having computer memory available to it to work effectively. If you are planning to work with large datasets, you may wish to set up OpenRefine to handle it at the outset. By “large” we generally mean one of the following indicators:

- more than one million total cells
- an input file size of more than 50 megabytes (MB)
- more than 50 [rows per record in records mode](#)

By default OpenRefine is set to operate with 1 gigabyte (GB) of memory (1024MB). If you feel

that OpenRefine is running slowly, or you are getting “out of memory” errors (for example, `java.lang.OutOfMemoryError`), you can try allocating more memory.

A good practice is to start with no more than 50% of whatever memory is left over after the estimated usage of your operating system, to leave memory for your browser to run.

All of the settings below use a four-digit number to specify the megabytes (MB) used (actually [mebibytes](#)). The default is usually 1024MB, but the new value doesn't need to be a multiple of 1024.

DEALING WITH LARGE DATASETS

If your project is big enough to need more than the default amount of memory, consider turning off `Parse cell text into numbers, dates, ...` on import. It's convenient, but less efficient than explicitly converting any columns that you need as a data type other than the default “string” type.

[Windows](#) [Mac](#) [Linux](#)

Using `openrefine.exe`

If you run `openrefine.exe`, you will need to edit the `openrefine.l4j.ini` file found in the program directory and edit the line

```
# max memory memory heap size  
-Xmx1024M
```

The line “`-Xmx1024M`” defines the amount of memory available in megabytes. Change the number “1024” - for example, edit the line to “`-Xmx2048M`” to make 2048MB [2GB] of memory available.

OPENREFINE.EXE NOT RUNNING?

Once you increase the memory allocation, you may find that you cannot run `openrefine.exe`. In this case, your computer needs a 64-bit version of [Java](#) (this is different from [Java JDK](#)). Look for the “Windows Offline (64-bit)” download on the Downloads page and install that. Your system must also be set to use the 64-bit version of Java by [changing the Java configuration](#).

Using `refine.bat`

On Windows, OpenRefine can also be run by using the file `refine.bat` in the program directory. If you start OpenRefine using `refine.bat`, the memory available to OpenRefine can be specified either through command line options, or through the `refine.ini` file.

To set the maximum amount of memory on the command line when using `refine.bat`, `cd` to the program directory, then type

```
refine.bat /m 2048m
```

where “2048” is the maximum amount of MB that you want OpenRefine to use.

To change the default that `refine.bat` uses, edit the `refine.ini` line that reads

```
REFINE_MEMORY=1024M
```

Note that this file is only read if you use `refine.bat`, not `openrefine.exe`.

 **FIRST, ENSURE THAT YOU HAVE ALREADY FOLLOWED AND COMPLETED THE INSTALL STEPS ABOVE, OTHERWISE YOU WILL GET AN ERROR ABOUT A READ-ONLY VOLUME WHEN EDITING THE `Info.plist` FILE FOLLOWING THE NEXT STEPS.**

If you have downloaded the `.dmg` package and you start OpenRefine by double-clicking on it:

- close OpenRefine
- control-click on the OpenRefine icon (opens the contextual menu)

- click on "show package content" (a finder window opens)
- open the "Contents" folder
- open and edit the `Info.plist` file with any text editor (like Mac's default TextEdit)
- Change "-Xmx1024M" into, for example, "-Xmx2048M" or "-Xmx8G"
- save the file
- restart OpenRefine

If you have downloaded the `.tar.gz` package and you start OpenRefine from the command line, add the "-m xxxxM" parameter like this: `./refine -m 2048m`

Setting a default

If you don't want to set this option on the command line each time, you can also set it in the `refine.ini` file. Edit the line

```
REFINE_MEMORY=1024M
```

Make sure it is not commented out (that is, that the line doesn't start with a "#" character), and change "1024" to a higher value. Save the file, and when you next start OpenRefine it will use this value.

Installing extensions

Extensions have been created by our contributor community to add functionality or provide convenient shortcuts for common uses of OpenRefine. [We list extensions we know about on our extensions page.](#)

CONTRIBUTING EXTENSIONS

If you'd like to create or modify an extension, [see our developer documentation here](#). If

you're having a problem, [use our extensions page](#) to go to the page of the extension's project and report the issue there.

Two ways to install extensions

You can [install extensions in one of two places](#):

- Into your OpenRefine program folder, so they will only be available to that version/installation of OpenRefine (meaning the extension will not run if you upgrade OpenRefine), or
- Into your workspace, where your projects are stored, so they will be available no matter which version of OpenRefine you're using.

We provide these options because you may wish to reinstall a given extension manually each time you upgrade OpenRefine, in order to be sure it works properly.

Find the right place to install

If you want to install the extension into the program folder, go to your program directory and then go to [webapp\extensions](#) (or create it if it does not exist).

If you want to install the extension into your workspace, you can:

- [Locate your workspace directory](#)
- Create a new folder called "extensions" inside the workspace if it does not exist.

You can also [find your workspace on each operating system using these instructions](#).

Install the extension

Some extensions have their own instructions: make sure you read the documentation before

you begin installing.

Some extensions may have multiple versions, to match OpenRefine versions, so be sure to choose the right release for your installation. If you have questions about compatibility or want to request or voice your support for an update, [use our downloads page](#) to go to the extension's page and report the issue there.

Generally, the installation process will be:

- Download the extension (usually as a zip file from GitHub)
- Extract the zip contents into the “extensions” directory, making sure all the contents go into one folder with the name of the extension
- Start (or restart) OpenRefine.

To confirm that installation was a success, follow the instructions provided by the extension. Each extension will appear in its own way inside the OpenRefine interface. Make sure you read its documentation to know where the functionality will appear, such as under specific dropdown menus.

Running OpenRefine

Starting and exiting

OpenRefine does not require internet access to run its basic functions. Once you download and install it, it runs as a small web server on your own computer, and you access that local web server by using your browser.

You will see a command line window open when you run OpenRefine. Ignore that window while you work on datasets in your browser.

No matter how you start OpenRefine, it will load its interface in your computer's default browser. If you would like to use another browser instead, start OpenRefine and then point your chosen browser at the home screen: <http://127.0.0.1:3333/>.

OpenRefine works best on browsers based on WebKit, such as:

- [Google Chrome](#)
- [Chromium](#)
- [Opera](#)
- [Microsoft Edge](#)
- [Safari](#)

We are aware of some minor rendering and performance issues on other browsers such as Firefox. We don't support Internet Explorer.

You can view and work on multiple projects at the same time by simply having multiple tabs or browser windows open. From the [Open Project](#) screen, you can right-click on project names and open them in new tabs or windows.

[Windows](#) [Mac](#) [Linux](#)

With `openrefine.exe`

You can run OpenRefine by double-clicking `openrefine.exe` or calling it from the command line.

If you want to [modify the way](#) `openrefine.exe` [opens](#), you can edit the `openrefine.l4j.ini` file.

With refine.bat

On Windows, OpenRefine can also be run by using the file `refine.bat` in the program directory. If you start OpenRefine using `refine.bat`, you can do so by opening the file itself, or by calling it from the command line.

If you call `refine.bat` from the command line, you can [start OpenRefine with modifications](#). If you want to modify the way `refine.bat` opens through double-clicking or using a shortcut, you can edit the `refine.ini` file.

Exiting

To exit OpenRefine, close all the browser tabs or windows, then navigate to the command line window. To close this window and ensure OpenRefine exits properly, hold down `Control` and press `C` on your keyboard. This will save any last changes to your projects.

You can find OpenRefine in your Applications folder, or you can open it using Terminal.

To run OpenRefine using Terminal:

- Find the OpenRefine application / icon in Finder
- Control-click on the icon and select “Show Package Contents” from the context menu
- This should open a new Finder menu: navigate into the “MacOS” folder
- Control-click on “JavaAppLauncher”
- Choose “Open With” from the menu, and select “Terminal.”

To exit, close all your OpenRefine browser tabs, go back to the terminal window and press `Command` and `Q` to close it down.



PROBLEMS STARTING?

If you are using an older version of OpenRefine or are on an older version of MacOS, [check our Wiki for solutions to problems with MacOS](#).

Use a terminal to launch OpenRefine. First, navigate to the installation folder. Then call the program:

```
cd openrefine-3.4.1  
./refine
```

This will start OpenRefine and open your browser to the home screen.

To exit, close all the browser tabs, and then press `Control` and `C` in the terminal window.

⚠ DID YOU GET A JAVA_HOME ERROR?

"Error: Could not find the 'java' executable at ", are you sure your JAVA_HOME environment variable is pointing to a proper java installation?"

If you see this error, you need to [install and configure a JDK package](#), including setting up **JAVA_HOME**.

Troubleshooting

If you are having problems connecting to OpenRefine with your browser, [check our Wiki for information about browser settings and operating-system issues](#).

Starting with modifications

When you run OpenRefine from a command line, you can change a number of default settings.

[Windows](#) [Mac](#) [Linux](#)

On Windows, use a slash:

```
C:>refine /i 127.0.0.2 /p 3334
```

Get a list of all the commands with [refine /?](#).

Command	Use	Syntax example
/w	Path to the webapp	refine /w /path/to/openrefine
/m	Memory maximum heap	refine /m 6000M
/p	Port	refine /p 3334
/i	Interface (IP address, or IP and port)	refine /i 127.0.0.2:3334
/H	HTTP host to expect on incoming requests	refine /H openrefine.internal

Command	Use	Syntax example
/d	Path to the workspace	refine /d /where/you/want/the/workspace
/v	Verbosity (from low to high: error, warn, info, debug, trace)	refine /v info
/x	Additional Java configuration parameters (see Java documentation)	refine /x refine.autosave=5 refine /x refine.data_dir=/ refine /x refine.development=false refine /x refine.headless=false refine /x refine.host=127.0.0.1 refine /x refine.port=3333 refine /x refine.webapp=main/webapp refine /x refine.display.new.version.notice=true
/debug	Enable debugging (on port 8000)	refine /d
/jmx	Enable JMX monitoring for Jconsole and JvisualVM	refine /x

You cannot start the Mac version with modifications using Terminal, but you can modify the way the application starts with [settings within files](#).

To see the full list of command-line options, run [./refine -h](#).

Command	Use	Syntax example
-w	Path to the webapp	./refine -w /path/to/openrefine
-d	Path to the workspace	./refine -d /where/you/want/the/workspace
-m	Memory maximum heap	./refine -m 6000M
-p	Port	./refine -p 3334

Command	Use	Syntax example
-i	Interface (IP address, or IP and port)	./refine -i 127.0.0.2:3334
-H	HTTP host to expect on incoming requests	./refine -H openrefine.internal
-k	Add a Google API key	./refine -k YOUR_API_KEY
-v	Verbosity (from low to high: error, warn, info, debug, trace)	./refine -v info
-x	Additional Java configuration parameters (see Java documentation)	<pre>./refine -x refine.autosave=5 ./refine -x refine.data_dir=/ ./refine -x refine.development=false ./refine -x refine.headless=false ./refine -x refine.host=127.0.0.1 ./refine -x refine.port=3333 ./refine -x refine.webapp=main/webapp ./refine -x refine.display.new.version.notice=true</pre>
--debug	Enable debugging (on port 8000)	./refine --debug
--jmx	Enable JMX monitoring for Jconsole and JvisualVM	./refine --jmx

Modifications set within files

On Windows, you can modify the way `openrefine.exe` runs by editing `openrefine.l4j.ini`; you can modify the way `refine.bat` runs by editing `refine.ini`.

You can modify the Mac application by editing `info.plist`.

On Linux, you can edit `refine.ini`.

Some settings, such as changing memory allocations, are already set inside these files, and all you have to do is change the values. Some lines need to be un-commented to work.

For example, inside `refine.ini`, you should see:

```
no_proxy="localhost,127.0.0.1"
#REFINE_PORT=3334
#REFINE_HOST=127.0.0.1
#REFINE_WEBAPP=main\webapp

# Memory and max form size allocations
#REFINE_MAX_FORM_CONTENT_SIZE=1048576
REFINE_MEMORY=1400M

# Set initial java heap space (default: 256M) for better performance with large
datasets
REFINE_MIN_MEMORY=1400M
...
```

JVM preferences

Further modifications can be performed by using JVM preferences. These JVM preferences are different options and have different syntax than the key/value descriptions used on the command line.

Some of the most common keys (with their defaults) are:

Description	Argument	Syntax example
The project <code>autosave</code> frequency	<code>-Drefine.autosave</code>	5 [minutes]
The workspace director	<code>-Drefine.data_dir</code>	/
Development mode	<code>-Drefine.development</code>	false
Headless mode	<code>-Drefine.headless</code>	false
IP	<code>-Drefine.host</code>	127.0.0.1
Port	<code>-Drefine.port</code>	3333
The application folder	<code>-Drefine.webapp</code>	main/webapp
New version notice	<code>-Drefine.display.new.version.notice</code>	true

The syntax is as follows:

Windows **Mac** **Linux**

Locate the `refine.l4j.ini` file, and insert lines in this way:

```
-Drefine.port=3334  
-Drefine.host=127.0.0.2  
-Drefine.webapp=broker/core  
-Dhttp.proxyHost=yourproxyhost  
-Dhttp.proxyPort=8080
```

In `refine.ini`, use a similar syntax, but set multiple parameters within a single line starting with `JAVA_OPTIONS=`:

```
JAVA_OPTIONS=-Drefine.data_dir=C:\Users\user\Documents\OpenRefine\ -Drefine.port=3334
```

Locate the `info.plist`, and find the `array` element that follows the line

```
<key>JVMOptions</key>
```

Typically this looks something like:

```
<key>JVMOptions</key>  
<array>  
<string>-Xms256M</string>  
<string>-Xmx1024M</string>  
<string>-Drefine.version=2.6-beta.1</string>  
<string>-Drefine.webapp=$APP_ROOT/Contents/Resource/webapp</string>  
</array>
```

Add in values such as:

```
<key>JVMOptions</key>  
<array>  
<string>-Xms256M</string>  
<string>-Xmx1024M</string>  
<string>-Drefine.version=2.6-beta.1</string>  
<string>-Drefine.webapp=$APP_ROOT/Contents/Resource/webapp</string>
```

Locate the `refine.ini` file, and add `JAVA_OPTIONS=` before the `-Drefine.preference` declaration. You can un-comment and edit the existing suggested lines, or add lines:

```
JAVA_OPTIONS=-Drefine.autosave=2  
JAVA_OPTIONS=-Drefine.port=3334  
JAVA_OPTIONS=-Drefine.data_dir/usr/lib/OpenRefineWorkspace  
JAVA_OPTIONS=-Dhttp.proxyHost=yourproxyhost  
JAVA_OPTIONS=-Dhttp.proxyPort=8080
```

Refer to the [official Java documentation](#) for more preferences that can be set.

The home screen

When you first launch OpenRefine, you will see a screen with a menu on the left hand side that includes `Create Project`, `Open Project`, `Import Project`, and `Language Settings`. This is called the “home screen,” where you can manage your projects and general settings.

In the lower left-hand corner of the screen, you'll see `Preferences`, `Help`, and `About`.

Language settings

From the home screen, look in the options to the left for `Language Settings`. You can set your preferred interface language here. This language setting will persist until you change it again in the future. Languages are translated as a community effort; some languages are partially complete and default back to English where unfinished. Currently OpenRefine supports the following languages for 75% or more of the interface:

- Cebuano
- German
- English (UK)
- English (US)
- Spanish
- Filipino
- French
- Hebrew
- Magyar

- Italian
- Japanese (日)
- Portuguese (Brazil)
- Tagalog
- Chinese (中)

To leave the Language Settings screen, click on the diamond “OpenRefine” logo.

HELP US TRANSLATE OPENREFINE

We use Weblate to provide translations for the interface. You can check [our profile on Weblate](#) to see which languages are in the process of being supported. See [our technical reference](#) if you are interested in [contributing translation work](#) to make OpenRefine accessible to people in other languages.

Preferences

In the bottom left corner of the screen, look for [Preferences](#). At this time you can set preferences using a key/value pair: that is, selecting one of the keys below and setting a value for it.

Setting	Key	Value syntax	Default	Example	Version
Interface language	userLang	ISO 639-1 two-digit code	en	fr	—
Maximum facets	ui.browsing.listFacet.limit	Number	2000	5000	—
Timeout for Google Drive import	googleReadTimeOut	Number (microseconds)	180000	500000	—
Timeout for Google Drive authorization	googleConnectTimeOut	Number (microseconds)	180000	500000	—
Maximum lag for	wikibase.upload.maxLag	Number (seconds)	5	10	—

Setting	Key	Value syntax	Default	Example	Version
Wikibase edit retries					
Display of the reconciliation preview on hover	cell-ui.previewMatchedCells	Boolean	true	false	v3.2
Values for the choice of the number of rows to display	ui.browsing.pageSize	Array of number (JSON)	[5, 10, 25, 50]	[100, 500, 1000]	v3.5
Width of the panel for facets/ history	ui.browsing.facetsHistoryPanelWidth	Number (pixel)	300	500	v3.5
Default state of the reconciliation automatch option	ui.reconciliation.automatch	Boolean	true	false	v3.8
Value of clustering choice limit	ui.clustering.choices.limit	Number	5000	8000	v3.8

To leave the Preferences screen, click on the diamond “OpenRefine” logo.

If the preference you’re looking for isn’t here, look at the options you can set from the [command line or in an .ini file](#).

The project screen

The project screen (or work screen) is where you will spend most of your time once you have [begun to work on a project](#). This is a quick walkthrough of the parts of the interface you should familiarize yourself with.

The screenshot shows the OpenRefine interface for a project titled "100 Most Populous Cities from Wikidata".

Facet / Filter panel (left):

- Shows a facet for "countryLabel" with 20 choices: Bangladesh, Brazil, Chile, Democratic Republic of the Congo, India, Indonesia, Iran, Japan, Manchukuo, and Nigeria.
- Buttons: Refresh, Undo / Redo 0 / 0, Reset All, Remove All, Cluster.

Table View (right):

- 100 rows** displayed.
- Extensions:** Wikidata
- Show as: rows records. Show: 5 10 25 50 rows. Navigation: « first < previous 1 - 10 next > last ».
- Table headers: All, cityLabel, population, countryLabel.
- Data rows (10 shown):

	cityLabel	population	countryLabel
1.	Shanghai	23390000	People's Republic of China
2.	Beijing	21710000	People's Republic of China
3.	Lagos	21324000	Nigeria
4.	Dhaka	16800000	Bangladesh
5.	Mumbai	15414288	India
6.	Istanbul	14657434	Turkey
7.	Tokyo	13942856	Japan
8.	Tianjin	13245000	People's Republic of China
9.	Guangzhou	13080500	People's Republic of China
10.	São Paulo	12106920	Brazil

The project bar

The project bar runs across the very top of the project screen. It contains the the OpenRefine logo, the project title, and the project control buttons on the right side.

At any time you can close your current project and go back to the home screen by clicking on the OpenRefine logo. If you'd like to open another project in a new browser tab or window, you can right-click on the logo and use "Open in a new tab." You will lose [your current facets and view settings](#) if you close your project (but data transformations will be saved in the [History](#) of the project).

CAUTION

Don't click the "back" button on your browser - it will likely close your current project and you will lose your facets and view settings.

You can rename a project at any time by clicking inside the project title, which will turn into a text field. Project names don't have to be unique, as OpenRefine organizes them based on a unique identifier behind the scenes.

The [Permalink](#) allows you to return to a project at a specific view state - that is, with [facets and filters](#) applied. The [Permalink](#) can help you pick up where you left off if you have to close your project while working with facets and filters. It puts view-specific information directly into the URL: clicking on it will load this current-view URL in the existing tab. You can right-click and copy the [Permalink](#) URL to copy the current view state to your clipboard, without refreshing the tab you're using.

The [Open...](#) button will open up a new browser tab showing the [Create Project](#) screen. From here you can change settings, start a new project, or open an existing project.

The [Export](#) is a dropdown menu that allows you to pick a format for exporting a dataset. Many of the export options will only export rows and records that are currently visible - the currently selected facets and filters, not the total data in the project.

The [Help](#) will open up a new browser tab and bring you to this user manual on the web.

The grid header

The grid header sits below the project bar and above the project grid (where the data of your project is displayed). The grid header will tell you the total number of rows or records in your project, and indicate whether you are in [rows or records mode](#).

It will also tell you if you're currently looking at a select number of rows via facets or filtering, rather than the entire dataset, by displaying either, for example, "180 rows" or "67 matching rows (180 total)."

Directly below the row number, you have the ability to switch between [row mode and records mode](#). OpenRefine stores projects persistently in one of the two modes, and displays your data as records by default if you are.

To the right of the rows/records selection is the array of options for how many rows/records to view on screen at one time. At the far right of the screen you can navigate through your entire dataset one page at a time.

Extensions

The [Extensions](#) dropdown offers you options for extending your data - most commonly by uploading your edited statements to Wikidata, or by importing or exporting schema. You can learn more about these functions on the [Wikibase section](#). Other extensions may also add functions to this dropdown menu.

The grid

The area of the project screen that displays your dataset is called the “grid” (or the “data grid,” or the “project grid”). The grid presents data in a tabular format, which may look like a normal spreadsheet program to you.

Columns widths are automatically set based on their contents; some column headers may be cut off, but can be viewed by mousing over the headers.

In each column header you will see a small arrow. Clicking on this arrow brings up a dropdown menu containing column-specific data exploration and transformation options. You will learn about each of these options in the [Exploring data](#) and [Transforming data](#) sections.

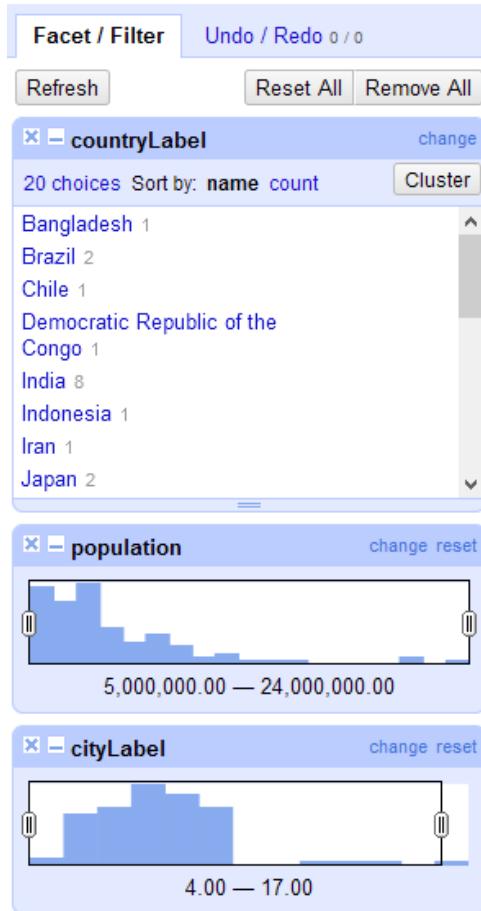
The first column in every project will always be [All](#), which contains options to flag, star, and do non-column-specific operations. The [All](#) column is also where rows/records are numbered. Numbering shows the permanent order of rows and records; a temporary sorting or facet may reorder the rows or show a limited set, but numbering will show you the original identifiers unless you make a permanent change.

The project grid may display with both vertical and horizontal scrolling, depending on the number and width of columns, and the number of rows/records displayed. You can control the display of the project grid by using [Sort and View options](#).

Mousing over individual cells will allow you to [edit cells individually](#).

Facet/Filter

The [Facet/Filter](#) tab is one of the main ways of exploring your data: displaying the patterns and trends in your data, and helping you narrow your focus and modify that data. [Facets](#) and [filters](#) are explained more in [Exploring data](#).



In the tab, you will see three buttons: **Refresh**, **Reset all**, and **Remove all**.

Refreshing your facets will ensure you are looking at the latest information about each facet, for example if you have changed the counts or eliminated some options.

Resetting your facets will remove any inclusion or exclusion you may have set - the facet options will stay in the sidebar, but your view settings will be undone.

Removing your facets will clear out the sidebar entirely. If you have written custom facets using [expressions](#), these will be lost.

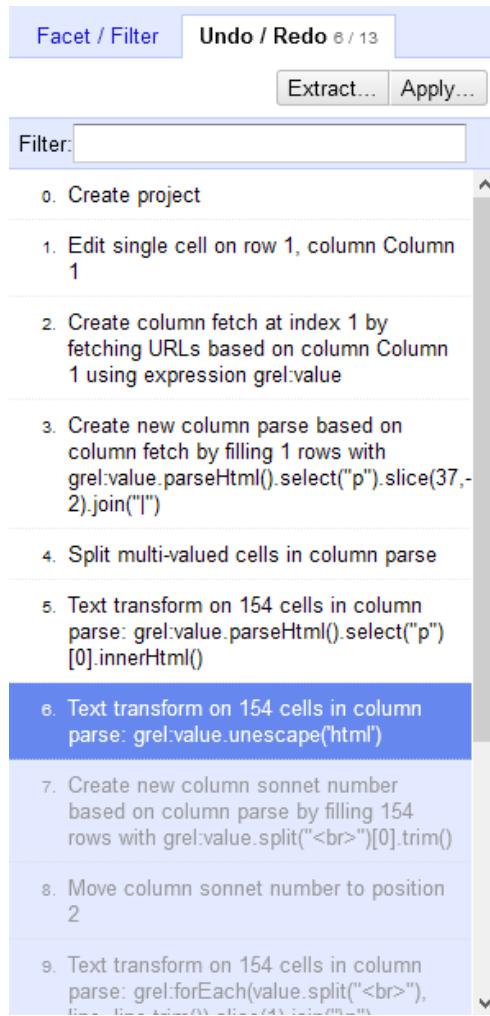
You can preserve your facets and filters for future use by copying a [Permalink](#).

History (Undo/Redo)

In OpenRefine, any activity that changes the data can be undone. Changes are tracked from the very beginning, when a project is first created. The change history of each project is saved with the project's data, so quitting

OpenRefine does not erase the steps you've taken. When you restart OpenRefine, you can view and undo changes that you made before you quit OpenRefine. OpenRefine [autosaves](#) your actions every five minutes by default, and when you close OpenRefine properly (using Ctrl + C). You can [change this interval](#).

Project history gets saved when you export a project archive, and restored when you import that archive to a new installation of OpenRefine.



When you click on the **Undo / Redo** tab in the sidebar of any project, that project's history is shown as a list of changes in order, with the first “change” being the action of creating the project itself. (That first change, indexed as step zero, cannot be undone.) Here is a sample history with 3 changes:

- 0. Create project
- 1. Remove 7 rows
- 2. Create new column Last Name based on column Name with grel:value.split(" ")
- 3. Split 230 cell(s) in column Address into several columns by separator

The current state of the project is highlighted with a dark blue background. If you move back and forth on the timeline you will see the current state become highlighted, while the actions that came after that state will be grayed out.

To revert your data back to an earlier state, simply click on the last action in the timeline you want to keep. In the example above, if we keep the removal of 7 rows but revert everything we did after that, then click on "Remove 7 rows." The last 2 changes will be undone, in order to bring the project back to state #1.

In this example, changes #2 and #3 will now be grayed out. You can redo a change by clicking on it in the history - everything up to and including it will be redone.

If you have moved back one or more states, and then you perform a new operation on your data, the later actions (everything that's greyed out) will be erased and cannot be re-applied.

The Undo/Redo tab will indicate which step you're on, and if you're about to risk erasing work - by saying something like "4/5" or "1/7" at the end.

Reusing operations

Operations that you perform in OpenRefine can be reused. For example, a formula you wrote inside one project can be copied and applied to another project later.

To reuse one or more operations, first extract it from the project where it was first applied. Click to the **Undo/Redo** tab and click **Extract...**. This brings up a box that lists all operations up to the current state (it does not show undone operations). Select the operation or operations you want to extract using the checkboxes on the left, and they will be encoded as JSON on the right. Copy that JSON to the clipboard.

Move to the second project, go to the **Undo/Redo** tab, click **Apply...** and paste in that JSON.

Not all operations can be extracted. Edits to a single cell, for example, can't be replicated.

Advanced OpenRefine uses

Running OpenRefine's Linux version on a Mac

You can run OpenRefine from the command line in Mac by using the Linux installation package. We do not promise support for this method. Follow the instructions in the Linux section.

Running as a server

CAUTION

Please note that if your machine has an external IP (is exposed to the Internet), you should not do this, or should protect it behind a proxy or firewall, such as nginx. Proceed at your own risk.

By default (and for security reasons), OpenRefine only listens to TCP requests coming from localhost (127.0.0.1) on port 3333. If you want to share your OpenRefine instance with colleagues and respond to TCP requests to any IP address of the machine, start it from the command line like this:

```
./refine -i 0.0.0.0
```

or set this option in `refine.ini`:

```
REFINE_HOST=0.0.0.0
```

or set this JVM option:

```
-Drefine.host=0.0.0.0
```

On Mac, you can add a specific entry to the `Info.plist` file located within the app bundle (`/Applications/OpenRefine.app/Contents/Info.plist`):

```
<key>JVMOptions</key>

<array>
  <string>-Drefine.host=0.0.0.0</string>
  ...
</array>
```

CAUTION

OpenRefine has no built-in security or version control for multi-user scenarios. OpenRefine has a single data model that is not shared, so there is a risk of data operations being overwritten by other users. Care must be taken by users.

Automating OpenRefine

Some users may wish to employ OpenRefine for batch processing as part of a larger automated pipeline. Not all OpenRefine features can work without human supervision and advancement (such as clustering), but many data transformation tasks can be automated.

CAUTION

The following are all third-party extensions and code; the OpenRefine team does not maintain them and cannot guarantee that any of them work.

Some examples:

- This project allows OpenRefine to be run from the command line using [operations saved in a JSON file](#): [OpenRefine batch processing](#)
- A Python project for applying a JSON file of operations to a data file, outputting the new file, and deleting the temporary project, written by David Huynh and Max Ogden: [Python client library for Google Refine](#)
- And the same in Ruby: [Refine-Ruby](#)
- Another Python client library, by Paul Makepeace: [OpenRefine Python Client Library](#)

To look for other instances, search our former Google Groups [for users](#) and [for developers](#), where [these projects were originally posted](#).

Starting a project

Overview

An OpenRefine project is started by importing in some existing data - OpenRefine doesn't allow you to create a dataset from nothing.

No matter where your data comes from, OpenRefine won't modify your original data source. It copies all the information from your input, creates its own project file, and stores it in your [workspace directory](#).

The data and all of your edits are [automatically saved](#) inside the project file. When you're finished modifying the data, you can [export it back out](#) into the file format of your choice.

You can also receive and open other people's projects, or send them yours, by [exporting a project archive](#) and [importing it](#).

Create a project by importing data

When you start OpenRefine, you'll be taken to the [Create Project](#) screen. You'll see on the left side of the screen that your options are to:

- import data from one or more files on your computer
- import data from one or more links on the web
- import data by pasting in text from your clipboard
- import data from a database (using SQL), and
- import one or more Sheets from Google Drive.

From these sources, you can load any of the following file formats:

- comma-separated values (CSV) or text-separated values (TSV)
- Text files
- Fixed-width columns
- JSON
- XML
- OpenDocument spreadsheet (ODS)
- Excel spreadsheet (XLS or XLSX)
- PC-Axis (PX)
- MARC
- RDF data (JSON-LD, N3, N-Triples, Turtle, RDF/XML)
- Wikitext

More formats can be imported by [adding extensions to provide that functionality](#).

If you supply two or more files for one project, the files' rows will be loaded in the order that you specify, and OpenRefine will create a column at the beginning of the dataset with the source URL or file name in it to help you identify where each row came from. If the files have columns with identical names, the data will load in those columns; if not, the successive files will append all of their new columns to the end of the dataset:

File	Fruit	Quantity	Berry	Berry source
fruits.csv	Orange	4		
fruits.csv	Apple	6		
berries.csv		9	Mulberry	Greece

File	Fruit	Quantity	Berry	Berry source
berries.csv		2	Blueberry	Canada

You cannot combine two datasets into one project by appending data within rows. You can, however, combine two projects later using functions such as [cross\(\)](#), or [fetch further data](#) using other methods.

For whichever method you choose to start your project, when you click [Next >>](#) you will be given a preview and a chance to configure the way OpenRefine interprets the data you input.

Get data from this computer

Click on [Browse...](#) and select a file (or several) on your hard drive. All files will be shown, not just compatible ones.

If you import an archive file (something with the extension [.zip](#), [.tar.gz](#), [.tgz](#), [.tar.bz2](#), [.gz](#), or [.bz2](#)), OpenRefine detects the files inside it, shows you a preview screen, and allows you to select which ones to load. This does not work with [.rar](#) files. When importing multiple archives you can store the name of the archive each file was extracted from by ticking the [Store archive file](#) option upon import.

Web addresses (URLs)

Type or paste the URL to a data file into the field provided. You can add as many fields as you want. OpenRefine will download the file and preview the project for you.

If you supply two or more file URLs, OpenRefine will identify each one and ask you to choose which (or all) to load.

Do not use this form to load a Google Sheet by its link; use [the Google Data form instead](#).

Clipboard

You can copy and paste in data from anywhere. OpenRefine will recognize comma-separated, tab-separated, or table-formatted information copied from sources such as word-processing documents, spreadsheets, and tables in PDFs. You can also just paste in a list of items that you want to turn into rows. OpenRefine recognizes each new text line as a row.

This can be useful if you want to pre-select a specific number of rows from your source data, or paste together rows from different places, rather than delete unwanted rows later in the project interface.

This can also be useful if you would like to paste in a list of URLs, which you can use later to [fetch more data](#).

Database (SQL)

If you are an administrator or have SQL access to a database of information, you may want to pull the latest dataset directly from there. This could include an online catalogue, a content management system, or a digital repository or collection management system. You can also load a database ([.db](#)) file saved locally. You will need to use an [SQL query](#) to import your intended data.

There are some publicly-accessible databases you can query, such as [one provided by Rfam](#). The instructions provided by Rfam can help you understand how to connect to and query from other databases.

OpenRefine can connect to PostgreSQL, MySQL, MariaDB, and SQLite database systems. It will automatically populate the [Port](#) field based on which of these you choose, but you

can manually edit this if needed.

If you have a `.db` file, you can supply the path to the file on your computer in the `Database` field at the bottom of the form. You can leave the rest of the fields blank.

To import data directly from a database, you will need the database type (such as MySQL), database name, the hostname (either an IP address or the domain that hosts the database), and the port on the host. You will need an account authorized for access, and you may need to add OpenRefine's IP address or host to the "allowable hosts" for that account. You can find that information by pressing `Test` and getting the IP address from the error message that results.

You can either connect just once to gather data, or save the connection to use it again later. If you press `Connect` without saving, OpenRefine will forget all the information you just entered. If you'd like to save the connection, name your connection in a way you will recognize later. Click `Save` and it will appear in the `Saved Connections` list on the left. From now on, you can click on the `...` ellipsis to the right of the connection you've saved, and click `Connect`.

If your connection is successful, you will see a Query Editor where you can run your SQL query. OpenRefine will give you an error if you write a statement that tries to modify the source database in any way.

Google data

You have two ways to load in data from Google Sheets:

- providing a link to an accessible Google Sheet (that is, one with link-sharing turned on), and
- selecting a Google Sheet in your Google Drive.

Google Sheet by URL

You can import data from any Google Sheet that has link-sharing turned on. Paste in a URL that looks something like

```
https://docs.google.com/spreadsheets/...../edit?usp=sharing
```

This will only work with Sheets, not with any other Google Drive file that might have an available link, including `.xls` and other valid files that are hosted in Google Drive. These links will not work when attempting to start a project [by URL](#) either, so you need to download those files to your computer.

Google Sheet from Drive

You can authorize OpenRefine to access your Google Drive data and import data from any Google Sheet it finds there. This will include Sheets that belong to you and Sheets that are shared with you, as well as Sheets that are in your trash.

When you select a Google option (either here, or [when exporting project data to Google Drive or Google Sheets](#), you will see a pop-up window that asks you to select a Google account to authorize with. You may see an error message when you authorize: if so, try your import or export operation again and it should succeed.

OpenRefine will not show spreadsheets that are in your email inbox or stored in any other Google property - only in Drive. It also won't show all compatible file formats, only Sheets files.

OpenRefine will generate a list of all Sheets it finds, with the most recently modified Sheets at the top. If a file you've just added isn't showing in this list, you can close and restart OpenRefine, or simply navigate to an existing project, open it, then head back to the [Create Project](#) window and check again.

When you click **Preview** the Sheet will open in a new browser tab. When you click the Sheet title, OpenRefine will begin to process the data.

Project preview

Once OpenRefine is ready to import the data, you will see a screen with **Configure Parsing Options** at the top. You'll see a preview of the first 100 rows and all identified columns.

At the bottom of the screen you will find options for telling OpenRefine how to process what it has found. You can tell it which row(s) to parse as column headers, as well as to ignore any number of rows at the top. You can also select a specific range of rows to work with, by discarding some rows at the top (excluding the header) and limiting the total number of rows it loads.

OpenRefine tries to guess how to parse your data based on the file extension. For example, **.xml** files are going to be parsed as though they are formatted in XML. An unknown file extension (or your clipboard copy-paste) is assumed to be either tab-separated or comma-separated. OpenRefine looks for a tab character, and if one is found, it assumes you have imported tab-separated data.

If OpenRefine isn't certain what format you imported, it will provide a list of possibilities under **Parse data as** and some settings. You can specify a custom separator now, or split columns later while [transforming your data](#).

If you imported a spreadsheet with multiple worksheets, they will be listed along with the number of rows they contain. You can only select data from one worksheet.

Note that OpenRefine does not preserve any formatting, such as cell or text colour, that may have been in the original data file. Hyperlinked text will be input as plain text, but OpenRefine will recognize links and make them clickable inside the project interface.

ENCODING ISSUES?

Look for character encoding issues at this stage. You may want to manually select an encoding, such as UTF-8, UTF-16, or ASCII, if OpenRefine does not display some characters correctly in the preview. Once your project is created, you can specify another encoding for specific columns using the [reinterpret\(\)](#) function.

You should create a project name at this stage. You can also supply tags to keep your projects organized. When you're happy with the preview, click [Create Project](#).

Import a project

Because OpenRefine only runs locally on your computer, you can't have a project accessible to more than one person at the same time.

The best way to collaborate with another person is to export and import projects that save all your changes, so that you can pick up where someone else left off. You can also [export projects](#) and import them to other computers, such as for working on the same project from the office and from home.

An exported project will include all of the [history](#), so you can see (and undo) all the changes from the previous user. It is essentially a point-in-time snapshot of their work. OpenRefine only exports projects as [.tar.gz](#) files at this time.

CAUTION

If you wish to hide the original state of your data and your history of edits (for example, if you are using OpenRefine to anonymize information), export your cleaned dataset only and do not share your project archive.

Once someone has sent you a project archive file from their computer, you can save it

anywhere. OpenRefine will import it like a new project and save its information to your workspace directory.

In the left-hand menu of the home screen, click **Import Project**. Click **Browse...** and navigate to wherever you saved the file you were sent (for example, your Downloads folder).

You can rename the project if you'd like - we recommend adding your name, a date, or a version number, if you're planning to continue collaborating with another person (or working from multiple computers).

Then, click **Import Project**. Your project should appear with a step count beside **Undo/Redo** if steps were saved by the exporter.

OpenRefine will store the project in its own workspace directory, so you can now delete the original file that was sent to you.

Project management

You can access all of your created projects by clicking on **Open Project**. Your project list can be organized by modification date, title, row count, and other metadata you can supply (such as subject, descriptor, tags, or creator). To edit the fields you see here, click **About** to the left of each project. There you can edit a number of available fields. You can also see the project ID that corresponds to the name of the folder in your work directory.

Naming projects

You may have multiple projects from the same dataset, or multiple versions from sharing a project with another person. OpenRefine automatically generates a project name from the imported file, or “clipboard” when you use **Clipboard** importing. Project names don’t

have to be unique, and OpenRefine will create many projects with the same name unless you intervene.

You can edit a project's name when you create it or import it, and you can rename a project later by opening it and clicking on the project name at the top of the screen.

Autosaving

OpenRefine [saves all of your actions](#) (everything you can see in the [Undo/Redo](#) panel). That includes flagging and starring rows.

It doesn't, however, save your facets, filters, or any kind of view you may have in place while you work. This includes the number of rows showing, and any sorting or column collapsing you may have done. A good rule of thumb is: if it's not showing in [Undo/Redo](#), you will lose it when you leave the project workspace.

Autosaving happens by default every five minutes. You can [change this preference by following these directions](#).

You can only save and share facets and filters, not any other type of view. To save current facets and filters, click [Permalink](#). The project will reload with a different URL, which you can then copy and save elsewhere. This permalink will save both the facets and filters you've set, and the settings for each one (such as sorting by count rather than by name).

Deleting projects

You can delete projects, which will erase the project files from the workspace directory on your computer. This is immediate and cannot be undone.

Go to [Open Project](#) and find the project you want to delete. Click on the  to the left of the project name. There will be a confirmation dialog.

Project files

You can find all of your raw project files in your work directory. They will be named according to the unique “Project ID” that OpenRefine has assigned them, which you can find on the [Open Project](#) screen, under the “About” link for each project.

Exploring data

Overview

OpenRefine offers lots of features to help you learn about your dataset, even if you don't change a single character. In this section we cover different ways for sorting through, filtering, and viewing your data.

Unlike spreadsheets, OpenRefine doesn't store formulas and display the output of those calculations; it only shows the value inside each cell. It doesn't support cell colors or text formatting.

Data types

Each piece of information (each cell) in OpenRefine is assigned a data type. Some file formats, when imported, can set data types that are recognized by OpenRefine. Cells without an associated data type on import will be considered a "string" at first, but you can have OpenRefine convert cell contents into other data types later. This is set at the cell level, not at the column level.

You can see data types in action when you preview a new project: check the box next to **Attempt to parse cell text into numbers**, and cells will be converted to the "number" data type based on their contents. You'll see numbers change from black text to green if they are recognized.

The data type will determine what you can do with the value. For example, if you want to add two values together, they must both be recognized as the number type.

You can check data types at any time by:

- clicking “edit” on a single cell (where you can also edit the type)
- creating a `Custom Text Facet` on a column, and inserting `type(value)` into the `Expression` field. This will generate the data type in the preview, and you can facet by data type if you press `OK`.

The data types supported are:

- string (one or more text characters)
- number (one or more characters of numbers only)
- boolean (values of “true” or “false”)
- `date` (ISO-8601-compliant extended format with time in UTC: YYYY-MM-DDTHH:MM:SSZ)

OpenRefine recognizes two further data types as a result of its own processes:

- error
- null

An “error” data type is created when the cell is storing an error generated during a transformation in OpenRefine.

A “null” data type is a special type that means “this cell has no value.” It’s distinct from cells that have values such as “0” or “false”, or cells that look empty but have whitespace in them, or cells that contain empty strings. When you use `type(value)`, it will show you that the cell’s value is “null” and its type is “undefined.” You can opt to [show “null” values](#), by going to `All` → `View` → `Show/Hide ‘null’ values in cells`.

Changing a cell’s data type is not the same operation as transforming its contents. For example, using a column-wide transform such as `Transform` → `Common transforms` → `To date` may not convert all values successfully, but going to an individual cell, clicking

“edit”, and changing the data type can successfully convert text to a date. These operations use different underlying code. Learn more about date formatting and transformations in the next section.

To transform data from one type to another, see [Transforming data](#) for information on using common transforms, and see [Expressions](#) for information on using `toString()`, `toDate()`, and other functions.

Dates

A “date” type is created when a column is [transformed into dates](#), when an expression is used to [convert cells to dates](#) or when individual cells are set to have the data type “date”.

Date-formatted data in OpenRefine relies on a number of conversion tools and standards. For something to be considered a date in OpenRefine, it will be converted into the ISO-8601-compliant extended format with time in UTC: YYYY-MM-DDTHH:MM:SSZ.

When you run `Edit cells` → `Common transforms` → `To date`, the following column of strings on the left will transform into the values on the right:

Input	→	Output
23/12/2019	→	2019-12-23T00:00:00Z
14-10-2015	→	2015-10-14T00:00:00Z
2012 02 16	→	2012-02-16T00:00:00Z
August 2nd 1964	→	1964-08-02T00:00:00Z
today	→	today

Input	→	Output
never	→	never

OpenRefine uses a variety of tools to recognize, convert, and format [dates](#) and so some of the values above can be reformatted using other methods. In this case, clicking the “today” cell and editing its data type manually will convert “today” into a value such as “2020-08-14T00:00:00Z”. Attempting the same data-type change on “never” will give you an error message and refuse to proceed.

You can do more precise conversion and formatting using expressions and arguments based on the state of your data: see the GREL functions reference section on [Date functions](#) for more help.

You can convert dates into a more human-readable format when you [export your data using the custom tabular exporter](#). You are given the option to keep your dates in the ISO 8601 format, to output short, medium, long, or full locale formats, or to specify a custom format. This means that you can format your dates into, for example, MM/DD/YY (the US short standard) with or without including the time, after working with ISO-8601-formatted dates in your project.

The following table shows some example [date and time formatting styles for the U.S. and French locales](#):

Style	U.S. Locale	French Locale
Default	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
Short	6/30/09 7:03 AM	30/06/09 07:03

Style	U.S. Locale	French Locale
Medium	Jun 30, 2009 7:03:47 AM	30 juin 2009 07:03:47
Long	June 30, 2009 7:03:47 AM PDT	30 juin 2009 07:03:47 PDT
Full	Tuesday, June 30, 2009 7:03:47 AM PDT	mardi 30 juin 2009 07 h 03 PDT

Rows vs. records

A row is a simple way to organize data: a series of cells, one cell per column. Sometimes there are multiple pieces of information in one cell, such as when a survey respondent can select more than one response.

In cases where there is more than one value for a single column in one or more rows, you may wish to use OpenRefine's records mode: this defines a single record as potentially containing more than one row. From there you can transform cells into multiple rows, each cell containing one value you'd like to work with.

Generally, when you import some data, OpenRefine reads that data in row mode. From the project screen, you can convert the project into records mode. OpenRefine remembers this action and will present you with records mode each time you open the project from then on.

OpenRefine understands records based on the content of the first column, what we call the "key column." Splitting a row into a multi-row record will base all association on the first column in your dataset.

If you have more than one column to split out into multiple rows, OpenRefine will keep your

data associated with its original record, and associate subgroups based on the top-most row in each group.

You can imagine the structure as a tree with many branches, all leading back to the same trunk.

For example, your key column may be a film or television show, with multiple cast members identified by name, associated to that work. You may have one or more roles listed for each person. The roles are linked to the actors, which are linked to the title.

Work	Actor	Role
The Wizard of Oz	Judy Garland	Dorothy Gale
	Ray Bolger	"Hunk"
		The Scarecrow
	Jack Haley	"Hickory"
		The Tin Man
	Bert Lahr	"Zeke"
		The Cowardly Lion
	Frank Morgan	Professor Marvel
		The Gatekeeper
		The Carriage Driver

Work	Actor	Role
		The Guard
		The Wizard of Oz
	Margaret Hamilton	Miss Almira Gulch
		The Wicked Witch of the West

Once you are in records mode, you can still move some columns around, but if you move a column to the beginning, you may find your data becomes misaligned. The new key column will sort into records based on empty cells, and values in the old key column will be assigned to the last row in the old record (the key value sitting above those values).

OpenRefine assigns a unique key behind the scenes, so your records don't need a unique identifier in the key column. You can keep track of which rows are assigned to each record by the record number that appears under the [All](#) column.

To [split multi-valued cells](#) and apply other operations that take advantage of records mode, see [Transforming data](#).

Be careful when in records mode that you do not accidentally delete rows based on being blank in one column where there is a value in another.

This feature is related to [Column Groups](#), which however is incomplete and deprecated.

Exploring facets

Overview

Facets are one of OpenRefine's strongest features - that's where the diamond logo comes from!

Faceting allows you to look for patterns and trends. Facets are essentially aspects or angles of data variance in a given column. For example, if you had survey data where respondents indicated one of five responses from "Strongly agree" to "Strongly disagree," those five responses make up a text facet, showing how many people selected each option.

Faceted browsing gives you a big-picture look at your data (do they agree or disagree?) and also allows you to filter down to a specific subset to explore it more (what do people who disagree say in other responses?).

Typically, you create a facet on a particular column. That facet selection appears on the left, in the **Facet/Filter** tab, and you can click on a displayed facet to view all the records that match. You can also "exclude" the facet, to view every record that does *not* match, and you can select more than one facet by clicking "include."

An example

You can learn about facets and filtering with the following example. You can copy the following table and paste it using the **Clipboard** method of starting a project if you would like to try it yourself. Check the "Attempt to parse cell text into numbers" option so that you can use numeric faceting.

We collected a list of the [10 most populous cities from Wikidata](#), using an example query of theirs. We removed the GPS coordinates and added the country.

cityLabel	population	countryLabel
Shanghai	23390000	People's Republic of China
Beijing	21710000	People's Republic of China
Lagos	21324000	Nigeria
Dhaka	16800000	Bangladesh
Mumbai	15414288	India
Istanbul	14657434	Turkey
Tokyo	13942856	Japan
Tianjin	13245000	People's Republic of China
Guangzhou	13080500	People's Republic of China
São Paulo	12106920	Brazil

If we want to see which countries have the most populous cities, we can create a text facet on the “countryLabel” column and OpenRefine will generate a list of all the different strings used in these cells.

We will see in the sidebar that the countries identified are displayed, along with the number of matches (the “count”). We can sort this list alphabetically or by the count. If you

sort by count at the top of the facet window, you'll learn which countries hold the most populous cities.

Facet	Count
People's Republic of China	4
Bangladesh	1
Brazil	1
India	1
Japan	1
Nigeria	1
Turkey	1

If we want to learn more about a particular country, we can click on its appearance in the facet sidebar. This narrows our dataset down temporarily to only rows matching that facet.

You'll see the "10 rows" indicator change to "4 matching rows (10 total)" if you click on "People's Republic of China". In the data grid, you'll see fewer rows: only the ones matching your current filter. Each row will maintain its original numbering, though - in this case, rows #1, 2, and 8.

If you want to go back to the original dataset, click **Reset All** or the small "exclude" text next to the facet. If you want to view the most populous cities in both China and India, click "include" next to each facet. Now you'll see 5 rows - #1, 2, 5, 8, 9.

We can also explore our data using the population information. In this case, because population is a number, we can create a numeric facet. This will give us the ability to explore by range rather than by exact matching values.

With the numeric facet, we are given a scale from the smallest to the largest value in the column. We can drag the range minimum and maximum to narrow the results. In this case, if we narrow down to only cities with more than 20 million in population, we get 3 matching rows out of the original 10.

When you look back at the text facet display of country names, you should see a smaller list with a reduced count: OpenRefine is now displaying the facets of the 3 matching rows, not the total dataset of 10 rows.

We can combine these facets - say, by narrowing to only the Chinese cities with populations greater than 20 million - simply by clicking in both. You should see 2 matching rows for both these criteria.

Things to know about facets

When you have facets applied, you will see “matching rows” in the [project grid header](#). If you click [Export](#) and copy your data out of OpenRefine while facets are active, many of the exporting options will only export the matching rows, not all the rows in your project.

OpenRefine has several default facets, which you’ll learn about below. The most powerful facets are the ones designed by you - custom facets, written using [expressions](#) to transform the data behind the scenes and help you narrow down to precisely what you’re looking for.

Facets are not saved in the project along with the data. But you can save a link to the current state of the application. Find the [Permalink](#) next to the project’s name.

You can modify any facet expression by clicking the “change” button to the right of the

column name in the facet sidebar.

Facet boxes that appear in the sidebar can be resized and rearranged. You can drag and drop the title bar of each box to reorder them, and drag on the bottom bar of text facet boxes.

Text facet

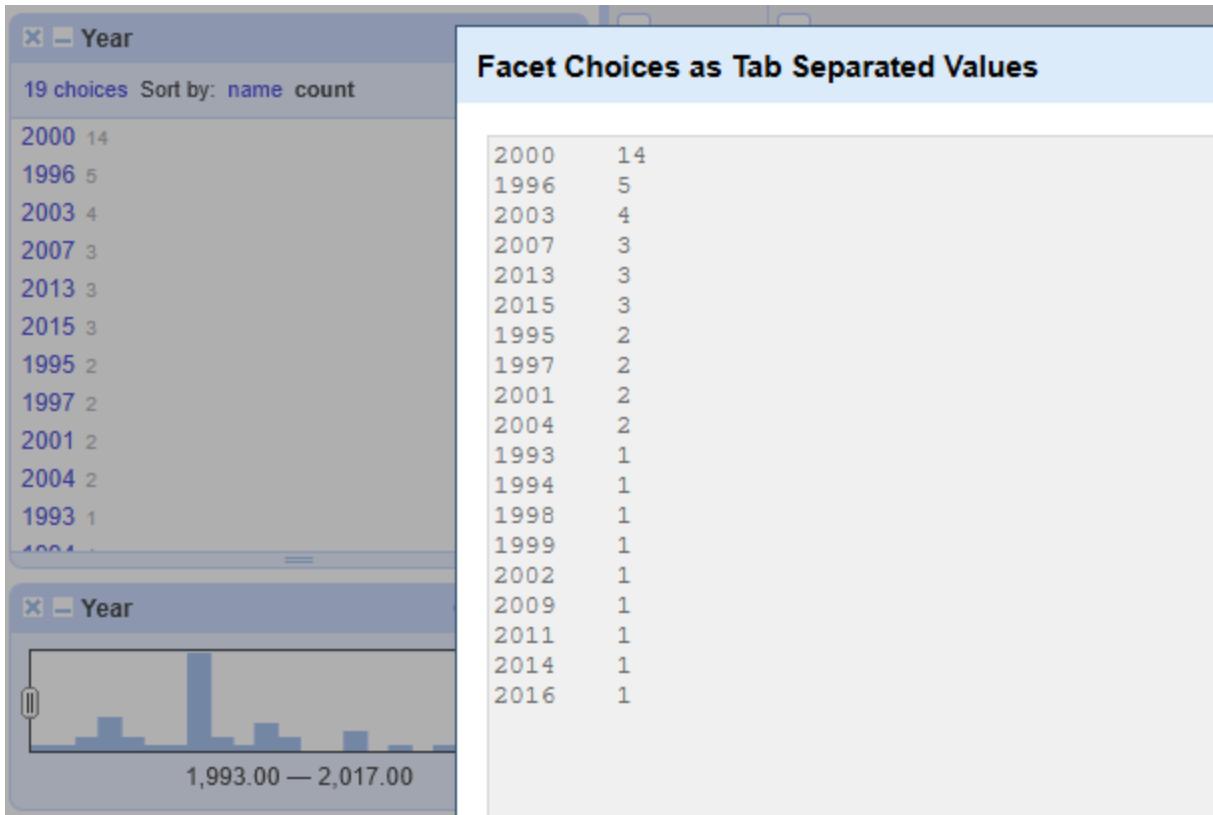
A text facet can be generated on any column with the “text” data type. Select the column dropdown and go to **Facet** → **Text facet**. The created facet will be sorted alphabetically, and can be sorted by count.

A text facet is very simple: it takes the total contents of the cells of the column in question and matches them up. It does no guessing about typos or near-matches.

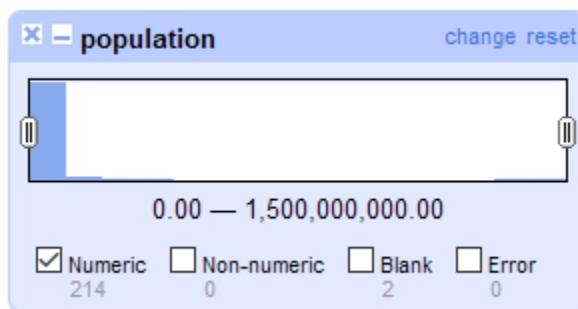
You can edit any entry that appears in the facet display, by hovering over the facet and clicking the “edit” button that appears. You can then type in a new value manually. This will mass-edit every identical cell in the column. This is a great way to fix typos, whitespace, and other issues that may be affecting the way facets appear. You can also automate the cleanup of facets by using [clustering](#): a “Cluster” button is displayed within the facet window. It may be most efficient to cluster cells to one value, and then mass-edit that value to your desired string within the clustering operation window.

Each text facet shows up to 2,000 choices by default. You can [increase this limit on the Preferences screen](#) if you need to, which may slow down your browser. If your applied facet has more choices than the current limit, you'll be offered the option to increase the limit, which will permanently edit that preference for you.

The choices and counts displayed in each facet can be copied as tab-separated values. To do so, click on the "X choices" link near the top left corner of the facet. This can be useful to generate small summary tables of your data.



Numeric facet



Whereas a text facet groups unique text values into groups, a numeric facet sorts numbers by their range - smallest to biggest. This displays visually as a histogram, and allows you to set a custom facet within that range. You can drag the minimum and maximum range markers to set a range. OpenRefine snaps to some basic equal-sized divisions - 19 in the example set above.

You will be offered the option to include blank, non-numeric, and error values in your numeric visualization; these will appear in the visual range as “0” values.

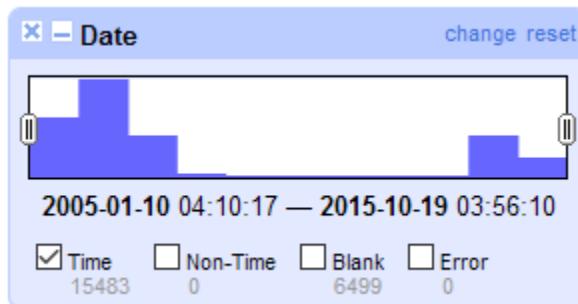
! NUMBERS AS TEXT

You can create a text facet on numeric data, which will treat each entry as a string. This can be useful if you wish, for example, to manually include facets instead of selecting a range, or sort by count, or copy that count.

! FACETING CUSTOMIZATION

As mentioned in the overview, facets can be modified or customized by GREL [expressions](#) in many ways. For example, to facet by clusters of [row](#) numbers with `row.index/100` or better visualizing numbers greater than 1000 with `max(row.index, 1000)`.

Timeline facet



Much like a numeric facet, a timeline facet will display as a small histogram with the values sorted: in this case, chronologically. A timeline facet only works on cells formatted as the “[date](#)” data type.

The facet appears with a count of blank cells and those with errors, which can help you

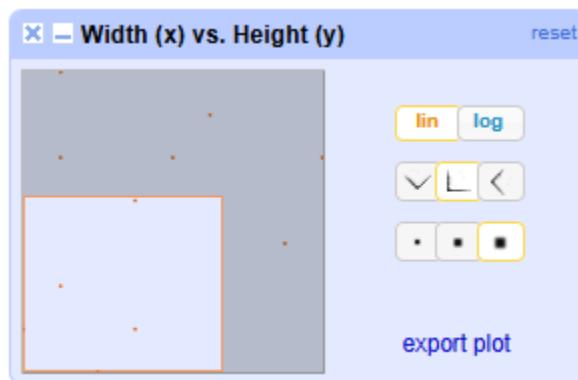
analyze whether your date cells are correctly converted.

Scatterplot facet

A scatterplot is a visual representation of two related sets of numeric data.

You have the option to generate linear scatterplots (where the X and Y axes show continuous increases) or logarithmic scatterplots (where the X and Y axes show exponential or scaled increases). You can also rotate the plot by 45 degrees in either direction, and you can choose the size of the dot indicating a datapoint. You can make these choices in both the preview and in the facet display.

A scatterplot facet can be generated on any column. You require two or more number columns to generate scatterplots. Selecting **Facet** → **Scatterplot facet** will create a preview of data plotted from every number-formatted column in your dataset, comparing every column against every other column. Each scatterplot will show in its own square, allowing you to choose which data comparison you would like to analyze further. You can control which columns are on the X and Y axes by rearranging the columns in your dataset.



When you click on your desired square, that two-column comparison will appear in the facets sidebar. From here, you can drag your mouse to draw a rectangle inside the scatterplot, which will narrow down to just the rows matching the points plotted inside

that rectangle (as shown by the rectangle inside the square in the image above). This rectangle can be resized by dragging any of the four edges. To draw a new rectangle, simply click and drag your mouse again. To add more scatterplots to the facet sidebar, re-run this process and select a different square.

If you have multiple facets applied, plotted points in your scatterplot displays will be greyed out if they are not part of the current matching data subset. If the rectangle you have drawn within a scatterplot display only includes grey dots, you will see no matching rows.

If you would like to export a scatterplot, OpenRefine will open a new tab with a generated PNG file that you can save.

Custom text facet

You may want to explore your textual data with modifications that aren't permanent. Creating custom text facets will load your column into memory, transform the data temporarily, and store those transformations inside the facet.

You can also use custom text facets to analyze numerical data, such as by analyzing a number as a string, or by creating a test that will return "true" and "false" as values.

Clicking on `Facet` → `Custom text facet...` will bring up an `expressions` window where you can enter in a GREL, Jython, or Clojure expression to modify how the facet works.

A custom text facet operates just like a `text facet` by default. Unlike a text facet, however, you cannot click "edit" on the facets that appear in the sidebar and change the matching cells in your dataset - because what they display is modified, not the original entries.

For example, you may wish to analyze only the first word in a text field - perhaps the first name in a column of "[First Name] [Last Name]" entries. In this case, you can tell

OpenRefine to facet only on the information that comes before the first space:

```
value.split(" ")[0]
```

In this case, `split()` is creating an array of text strings based on every space in the cells ["Firstname", "Lastname"]. Because arrays number their entries starting with 0, we want the first value, so we ask for `[0]`. (Assuming the first name is one word, not something like "Mary Anne.") We can do the same splitting and ask for the last name with

```
value.split(" ")[1]
```

You may want to create a facet that references several columns. For example, let's say you have two columns, "First Name" and "Last Name", and you want out how many people have the same initial letter for both names (e.g., Marilyn Monroe, Steven Segal). To do so, create a custom text facet on either column and enter the expression

```
cells["First Name"].value[0] == cells["Last Name"].value[0]
```

That expression will look for the first letter (the character at index 0) of each entry and compare them. Then it will facet your rows into "true" and "false."

You can learn more about text-modification functions on the [Expressions page](#).

Custom numeric facet

You may want to explore your numerical data with modifications that aren't permanent. You can also use custom numeric facets to analyze textual data, such as by getting the length of text strings (with `value.length()`), or by analyzing it as though it were formatted as numbers (with `toNumber(value)`).

If you would like to build your own version of a numeric facet, you can use the `Custom Numeric Facet` option. Clicking on `Facet` → `Custom Numeric Facet...` will bring up an [expressions](#) window where you can enter in a GREL, Jython, or Clojure expression to modify how the facet works. A custom numeric facet operates just like a [numeric facet](#) by default.

For example, you may wish to create a numeric facet that rounds your value to the nearest integer, enter

```
round(value)
```

If you have two columns of numbers and for each row you wish to create a numeric facet only on the larger of the two, enter

```
max(cells["Column1"].value, cells["Column2"].value)
```

If the numeric values in a column are drawn from a power law distribution, then it's better to group them by their logs:

```
value.log()
```

If the values are periodic you could take the modulus by the period to understand if there's a pattern:

```
mod(value, 7)
```

You can learn more about numeric-modification functions on the [Expressions page](#).

Customized facets

Customized facets have been added to expand the number of default facets users can apply with a single click. They represent some common and useful functions you shouldn't have to work out using an [expression](#).

All facets that display in the **Facet/Filter** tab can be edited by clicking on the “change” button to the right of the column title. This brings up the expressions window that will allow you to modify and preview the expression being used.

Word facet

A **Word facet** is a simple version of a text facet: it splits up the content of the cells based on spaces, and outputs each character string as a facet:

```
value.split(" ")
```

This can be useful for exploring the language used in a corpus, looking for common first and last names or titles, or seeing what's in multi-valued cells you don't wish to split up.

Word facet is case-sensitive and only splits by spaces, not by line breaks or other natural divisions.

Duplicates facet

A **Duplicates facet** will return only rows that have non-unique values in the column you've selected. It will create a facet of “true” and “false” values - true being cells that are not unique, and “false” being cells that are. The actual expression being used is

```
facetCount(value, 'value', '[Column]') > 1
```

Duplicates facets are case-sensitive and you may wish to filter out things like leading and trailing whitespace or other hard-to-see issues. You can modify the facet expression, for example, with:

```
facetCount(trim(toLowercase(value)), 'trim(toLowercase(value))',  
'cityLabel') > 1
```

Numeric log facet

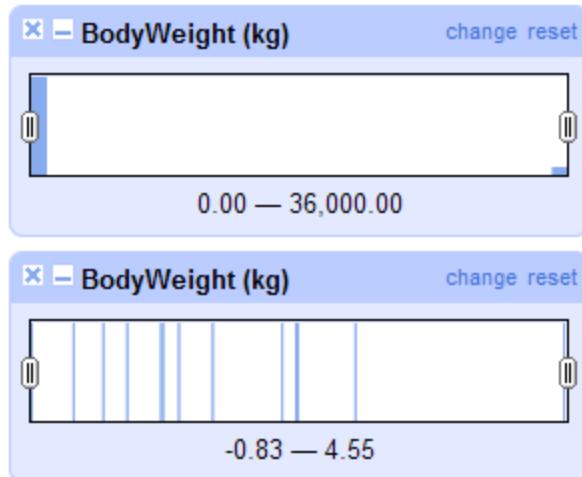
Logarithmic scales reduce wide-ranging quantities to more compact and manageable ranges. A log transformation can be used to make highly skewed distributions less skewed. If your numerical data is unevenly distributed (say, lots of values in one range, and then a long tail extending off into different magnitudes), a **Numeric log facet** can represent that range better than a simple numeric facet. It will break these values down into more navigable segments than the buckets of a numeric facet. This facet can make patterns in your data more visible. OpenRefine uses a base-10 log, the “common logarithm.”

For example, we can look at [this data about the body weight of various mammals](#):

Species	BodyWeight (kg)
Newborn_Human	3.2
Adult_Human	73
Pithecanthropus_Man	70

Species	BodyWeight (kg)
Squirrel	0.8
Hamster	0.15
Chimpanzee	50
Rabbit	1.4
Dog_(Beagle)	10
Cat	4.5
Rat	0.4
Sperm_Whale	35000
Turtle	3
Alligator	270

Most values will be clustered in the 0-100 range, but 35,000 is many magnitudes above that. A numeric facet will create 36 equal buckets of 1,000 each - containing almost all the cells in the first bucket. A numeric log facet will instead display the data more evenly across the visual range.



A 1-bounded numeric log facet can be used if you'd like to exclude all the values below 1 (including zero and negative numbers).

Text-length facet

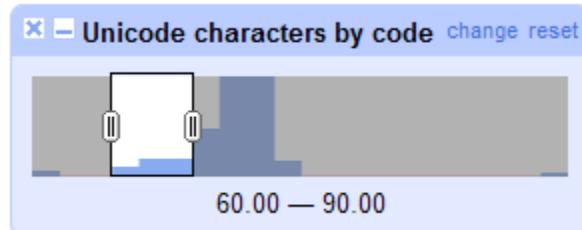
The `Text-length facet` returns a numerical value for each cell and plots it on a numeric facet chart. The expression used is

```
value.length()
```

This can be useful to, for example, look for values that did not successfully split on an earlier split operation, or to validate that data is a certain expected length (such as whether a date in YYYY/MM/DD is eight to ten characters).

You can also employ a `Log of text-length facet` that allows you to navigate more easily a wide range of string lengths. This can be useful in the case of web-scraping, where lots of textual data is loaded into single cells and needs to be parsed out.

Unicode character-code facet



The Unicode facet identifies and returns [Unicode decimal values](#). It generates a list of the Unicode numerical values of each character used in each text cell, which allows you to narrow down and search for special characters, punctuation, and other data formatting issues.

This facet creates a numerical chart, which offers you the ability to narrow down to a range of numbers. For example, lowercase characters are numbers 97-122, uppercase characters are numbers 65-90, and numerical digits are numbers 48-57.

Facet by error

An error is a data type created by OpenRefine in the process of transforming data. For example, say you had converted a column to the number data type. If one cell had text characters in it, OpenRefine could either output the original text string unchanged or output an error. If you allow errors to be created, you can facet by them later to search for them and fix them.

Expression

```
toNumber(value)
```

Preview History Starred Help

row	value	toNumber(value)
1.	1	1
2.	2	2
3.	3	3
4.	4	4
5.	five	Error: Unable to parse as number
6.	6	6

On error keep original
 set to blank
 store error

OK Cancel

To store errors in cells, ensure that you have **store error** selected for the “On error” option in the expressions window.

Facet by null, empty, or blank

Any column can be faceted for [null and/or empty cells](#). These can help you find cells where you want to manually enter content.

“Blank” means both null values and empty values. All three facets will generate “true” and “false” facets, “true” being blank.

An empty cell is a cell that is set to contain a string, but doesn’t have any characters in it (a zero-length string). This can be left over from an operation that removed characters, or

from manually editing a cell and deleting its contents.

Facet by star or flag

Stars and flags offer you the opportunity to mark specific rows for yourself for later focus. Stars and flags persist through closing and opening your project, and thus can provide a different function than using a permalink to persist your facets. Stars and flags can be used in any way you want, although they are designed to help you flag errors and star rows of particular importance.

You can manually star or flag rows simply by clicking on the icons to the left of each row.

You can also apply stars or flags to all matching rows by using the `All` dropdown menu (on the first column) and selecting `Edit rows` → `Star rows` or `Flag rows`. This will create “true” and “false” facets in the `Facet/Filter`. These operations will modify all matching rows in your current subset. You can unstar or unflag them as well.

You may wish to create a custom subset of your data through a series of separate faceting activities (rather than successively narrowing down with multiple facets applied). For example, you may wish to:

- apply a facet
- star all the matching rows
- remove that facet
- apply another, unrelated facet
- star all the new matching rows (which will not modify already-starred rows)
- remove that facet
- and then work with all of the cumulative starred rows.

You can also create a text facet on any column with the expression `row.starred` or `row.flagged`.

Text filter

Filters allow you to narrow down your data based on whether a given column includes a text string.

When you choose **Text filter** a box appears in the **Facet/Filter** tab that allows you to enter in text. Matching rows will narrow dynamically with every character you enter. You can set the search to be case-sensitive or not, and you can use this box to enter in a regular expression.

For example, you can enter in “side” as a text filter, and it will return all cells in that column containing “side,” “sideways,” “offside,” etc.

The text filter field supports [regular expressions](#). For example, you can employ a regular expression to view all properly-formatted emails:

```
([a-zA-Z0-9_-\.\+]+)@([a-zA-Z0-9\-\.\.]+)\.([a-zA-Z0-9\-\.]{2,15})
```

You can press “invert” on this facet to then see blank cells or invalid email addresses.

This filter works differently than facets because it is always active as long as it appears in the sidebar. If you “reset” it, you will delete all the text or expression you have entered.

You can apply multiple text filters in succession, which will successively narrow your data subset. This can be useful if you apply multiple inverted filters, such as to filter out all rows that respond “yes” or “maybe” and only look at the remaining responses.

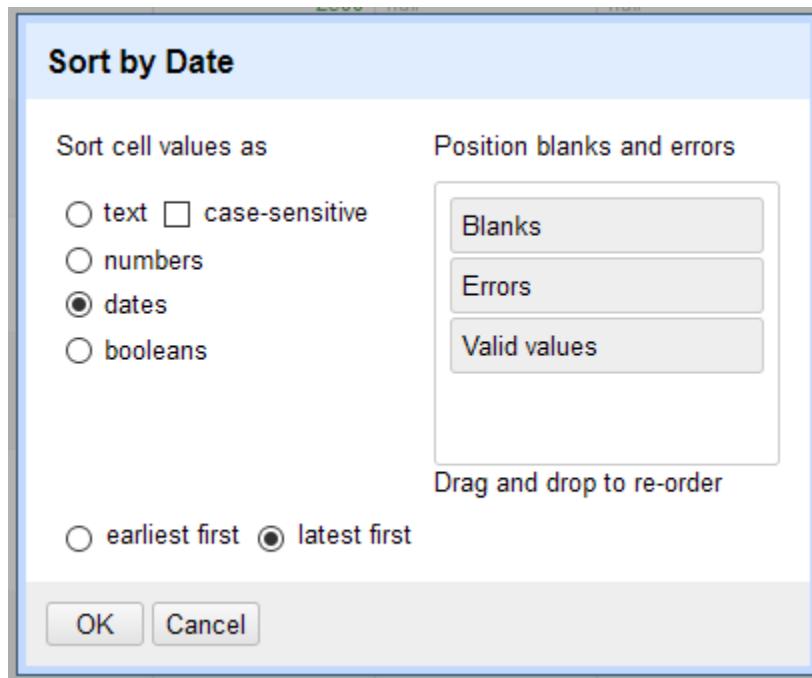
Sort and view

Sort

You can temporarily sort your rows by one column. You can sort based on [data type](#):

- text alphabetically or reverse
- numbers by largest or smallest
- dates by earliest or latest
- boolean values by false first or true first.

You can also choose where to place errors and blank cells in the sorting. Text can be case-sensitive or not: if so, cells that start with lowercase characters will appear ahead of those that start with uppercase characters.



After you apply a sorting method, you can make it permanent, remove it, reverse it, or apply a subsequent sorting. When it is applied, you'll find **Sort** in the project grid header to the right of the rows-display setting, which will show all current sorting settings.

If you have multiple sorting methods applied, they will work in the order you applied them (represented in order in the **Sort** menu). For example, you can sort an “authors” column alphabetically, and then sort their books by publication date, for those authors that have more than one book. If you apply those in a different order - sort all the publication dates in the dataset first, and then alphabetically by author - your dataset will look different.

55 rows				
Show as: rows records		Show: 5 10 25 50 rows	Sort ▾	
<input type="checkbox"/> All		<input type="checkbox"/> Author	<input type="checkbox"/> Title	
<input type="checkbox"/>	24.	A. S. Byatt	Possession	
<input type="checkbox"/>	39.	Alan Hollinghurst	The Line of Beauty	
<input type="checkbox"/>	18.	Anita Brookner	Hotel du Lac	
<input type="checkbox"/>	53.	Anna Burns	Milkman	
<input type="checkbox"/>	42.	Anne Enright	The Gathering	
<input type="checkbox"/>	43.	Aravind Adiga	The White Tiger	
<input type="checkbox"/>	32.	Arundhati Roy	The God of Small Things	
<input type="checkbox"/>	27.	Barry Unsworth	Sacred Hunger	
<input type="checkbox"/>	25.	Ben Okri	The Famished Road	
<input type="checkbox"/>	55.	Bernardine Evaristo	Girl, Woman, Other	
<input type="checkbox"/>	2.	Bernice Rubens	The Elected Member	
<input type="checkbox"/>	10.	David Storey	Saville	
<input type="checkbox"/>	38.	DBC Pierre	Vernon God Little	
<input type="checkbox"/>	48.	Eleanor Catton	The Luminaries	
<input type="checkbox"/>	52.	George Saunders	Lincoln in the Bardo	
<input type="checkbox"/>	31.	Graham Swift	Last Orders	
<input type="checkbox"/>	44.	Hilary Mantel	Wolf Hall	
<input type="checkbox"/>	47.	Hilary Mantel	Bring Up the Bodies	
<input type="checkbox"/>	45.	Howard Jacobson	The Finkler Question	
<input type="checkbox"/>	33.	Ian McEwan	Amsterdam	
<input type="checkbox"/>	12.	Iris Murdoch	The Sea, the Sea	
<input type="checkbox"/>	3.	J. G. Farrell	Troubles	
<input type="checkbox"/>	6.	J. G. Farrell	The Siege of Krishnapur	
<input type="checkbox"/>	17.	J. M. Coetzee	Life & Times of Michael K	

Remove sort
Reorder rows permanently

By Author ▾
By Publication year ▾

Sort...
Reverse
Remove sort

Country
 United Kingdom

When the sorting method you've applied is temporary, you will see that the rows retain their original numbering. When you make that sorting method permanent, by selecting

Reorder rows permanently, the row numbers will change and the Sort menu in the project grid header will disappear. This will apply all current sorting methods.

View

You can control what data you view in the grid. On each column, you will see a View menu option. From there, you can “collapse” (hide) that specific column, all other columns, all columns to the left, and all columns to the right. Using the View option that appears in the All column’s dropdown menu, you can collapse all columns, and expand all the columns that you previously collapsed.

Show/hide “null”

You can find, under All → View, the option to show and hide “null” values. A small grey “null” will appear in each applicable cell. Remember that a null cell is not the same thing as an empty cell.

TRUE	4
FALSE	
null	6
null	null

Page navigation

You can go directly to any page by changing the page number on the right-hand side. Using the up and down arrow keys in this input lets you go to the next and previous pages. You can also change the number of rows or records per page on the left-hand side of this view header bar.



Transforming data

Overview

OpenRefine gives you powerful ways to clean, correct, codify, and extend your data. Without ever needing to type inside a single cell, you can automatically fix typos, convert things to the right format, and add structured categories from trusted sources.

This section of ways to improve data are organized by their appearance in the menu options in OpenRefine. You can:

- change the order of [rows](#) or [columns](#)
- edit [cell contents](#) within a particular column
- [transform](#) rows into columns, and columns into rows
- [split or join columns](#)
- [add new columns](#) based on existing data, with fetching new information, or through [reconciliation](#)
- convert your rows of data into [multi-row records](#).

Edit rows

Moving rows around is a permanent change to your data.

You can [sort your data](#) based on the values in one column, but that change is a temporary view setting. With that setting applied, you can make that new order permanent.

100 rows

Show as: [rows](#) [records](#) Show: [5](#) [10](#) [25](#) [50](#) rows [Sort ▾](#)

All	cityLabel	population	country
46.	Ahmedabad	7645000	India
82.	Allahabad	5954391	India
92.	Bangkok	5696409	Thailand
19.	Baoding	10700000	People's Republic of China
2.	Beijing	21710000	People's Republic of China
15.	Bengaluru	11250000	India
70.	Bijie	6537498	People's Republic of China
64.	Cangzhou	6800000	People's Republic of China
44.	Changchun	7674439	People's Republic of China
45.	Changchun	7674439	Manchukuo
80.	Chennai	7747000	India

[Remove sort](#)

[Reorder rows permanently](#)

[By cityLabel](#)

In the project grid header, the word “Sort” will appear when a sort operation is applied.

Click on it to show the dropdown menu, and select [Reorder rows permanently](#). You will see the numbering of the rows change under the [All](#) column.

REORDERING ALL ROWS

Reordering rows permanently will affect all rows in the dataset, not just those currently viewed through [facets and filters](#).

You can undo this action using the [History](#) tab.

Cell editing

Overview

OpenRefine offers a number of features to edit and improve the contents of cells automatically and efficiently.

One way of doing this is editing through a [text facet](#). Once you have created a facet on a column, hover over the displayed results in the sidebar. Click on the small “edit” button that appears to the right of the facet, and type in a new value. This will apply to all the cells in the facet.

You can apply a text facet on numbers, boolean values, and dates, but if you edit a value it will be converted into the text [data type](#) (regardless of whether you edit a date into another correctly-formatted date, or a “true” value into “false”, etc.).

Transform

Select [Edit cells](#) → [Transform...](#) to open up an expressions window. From here, you can apply [expressions](#) to your data. The simplest examples are GREL functions such as [toUppercase\(\)](#) or [toLowercase\(\)](#), used in expressions as [toUppercase\(value\)](#) or [toLowercase\(value\)](#). When used on a column operation, [value](#) is the information in each cell in the selected column.

Use the preview to ensure your data is being transformed correctly.

You can also switch to the [Undo / Redo](#) tab inside the expressions window to reuse expressions you’ve already attempted in this project, whether they have been undone or

not.

OpenRefine offers you some frequently-used transformations in the next menu option, [Common transforms](#). For more custom transforms, read up on [expressions](#).

Common transforms

Trim leading and trailing whitespace

Often cell contents that should be identical, and look identical, are different because of space or line-break characters that are invisible to users. This function will get rid of any characters that sit before or after visible text characters.

Collapse consecutive whitespace

You may find that some text cells contain what look like spaces but are actually tabs, or contain multiple spaces in a row. This function will remove all space characters that sit in sequence and replace them with a single space.

Unescape HTML

Your data may come from an HTML-formatted source that expresses some characters through references (such as “ ” for a space, or “%u0107” for a Ć) instead of the actual Unicode characters. You can use the “unescape HTML entities” transform to look for these codes and replace them with the characters they represent. For other formatting that needs to be escaped, try a custom transformation with [escape\(\)](#).

Replace smart quotes with ASCII

Smart quotes (or curly quotes) recognize whether they come at the beginning or end of a string, and will generate an “open” quote (“) and a “close” quote (“”). These characters are not ASCII-compliant (though they are UTF8-compliant) so you can use this transform to replace them with a straight double quote character (") instead.

Case transforms

You can transform an entire column of text into UPPERCASE, lowercase, or Title Case using these three options. This can be useful if you are planning to do textual analysis and wish to avoid case-sensitivity (which some functions are) causing problems in your analysis. Consider also using a [custom facet](#) to temporarily modify cases instead of this permanent operation if appropriate.

Data-type transforms

As detailed in [Data types](#), OpenRefine recognizes different data types: string, number, boolean, and date. When you use these transforms, OpenRefine will check to see if the given values can be converted, then both transform the data in the cells (such as “3” as a text string to “3” as a number) and convert the data type on each successfully transformed cell. Cells that cannot be transformed will output the original value and maintain their original data type.

CAUTION

Be aware that dates may require manual intervention to transform successfully: see the section on [Dates](#) for more information.

Because these common transforms do not offer the ability to output an error instead of the original cell contents, be careful to look for unconverted and untransformed values. You will see a yellow alert at the top of screen that will tell you how many cells were converted - if this number does not match your current row set, you will need to look for and manually correct the remaining cells. Also consider facetting by data type, with the GREL function [type\(\)](#).

You can also convert cells into null values or empty strings. This can be useful if you wish to, for example, erase duplicates that you have identified and are analyzing as a subset.

Fill down and blank down

Fill down and blank down are two functions most frequently used when encountering data organized into [records](#) - that is, multiple rows associated with one specific entity.

If you receive information in rows mode and want to convert it to records mode, the easiest way is to sort your first column by the value that you want to use as a unique records key, [make that sorting permanent](#), then blank down all the duplicates in that column. OpenRefine will retain the first unique value and erase the rest. Then you can switch from “Show as rows” to “Show as records” and OpenRefine will associate rows to each other based on the remaining values in the first column.

Be careful that your data is sorted properly before you begin blanking down - not just the first column but other columns you may want to have in a certain order. For example, you may have multiple identical entries in the first column, one with a value in the second column and one with an empty cell in the second column. In this case you want the row with the second-column value to come first, so that you can clean up empty rows later, once you blank down.

If, conversely, you’ve received data with empty cells because it was already in something akin to records mode, you can fill down information to the rest of the rows. This will

duplicate whatever value exists in the topmost cell with a value: if the first row in the record is blank, it will take information from the next cell, or the cell after that, until it finds a value. The blank cells above this will remain blank.

Split multi-valued cells

Splitting cells with more than one value in them is a common way to get your data from single rows into [multi-row records](#). Survey data, for example, frequently allows respondents to “Select all that apply,” or an inventory list might have items filed under more than one category.

You can split a column based on any character or series of characters you input, such as a semi-colon (;) or a slash (/). The default is a comma. Splitting based on a separator will remove the separator characters, so you may wish to include a space with your separator (;) if it exists in your data.

You can use [expressions](#) to design the point at which a cell should split itself into two or more rows. This can be used to identify special characters or create more advanced evaluations. You can split on a line-break by entering `\n` and checking the “[regular expression](#)” checkbox.

Regular expressions can be useful if the split is not straightforward: say, if a capital letter (`[A-Z]`) indicates the beginning of a new string, or if you need to *not* always split on a character that appears in both the strings and as a separator. Remember that this will remove all the matching characters.

You can also split based on the lengths of the strings you expect to find. This can be useful if you have predictable data in the cells: for example, a 10-digit phone number, followed by a space, followed by another 10-digit phone number. Any characters past the explicit length you’ve specified will be discarded: if you split by “11, 10” any characters that may come after the 21st character will disappear. If some cells only have one phone

number, you will end up with blank rows.

If you have data that should be split into multiple columns instead of multiple rows, see [Split into several columns](#).

Join multi-valued cells

Joining will reverse the “split multi-valued cells” operation, or join up information from multiple rows into one row. All the strings will be compressed into the topmost cell in the record, in the order they appear. A window will appear where you can set the separator; the default is a comma and a space (,). This separator is optional. We suggest the separator | as a sufficiently rare character.

Cluster and edit

Creating a facet on a column is a great way to look for inconsistencies in your data; clustering is a great way to fix those inconsistencies. Clustering uses a variety of comparison methods to find text entries that are similar but not exact, then shares those results with you so that you can merge the cells that should match. Where editing a single cell or text facet at a time can be time-consuming and difficult, clustering is quick and streamlined.

Clustering always requires the user to approve each suggested edit - it will display values it thinks are variations on the same thing, and you can select which version to keep and apply across all the matching cells (or type in your own version).

OpenRefine will do a number of cleanup operations behind the scenes in order to do its analysis, but only the merges you approve will modify your data. Understanding those different behind-the-scenes cleanups can help you choose which clustering method will be

more accurate and effective.

You can start the process in two ways: using the dropdown menu on your column, select **Edit cells** → **Cluster and edit...**; or create a text facet and then press the “Cluster” button that appears in the facet box.

Cluster & edit column "Place"

This feature helps you find groups of different cell values that might be alternative representations of the same thing. For example, the two strings "New York" and "new york" are very likely to refer to the same concept and just have capitalization differences, and "Gödel" and "Godel" probably refer to the same person. [Find out more...](#)

Method **key collision** Keying Function **Fingerprint** 7 clusters found

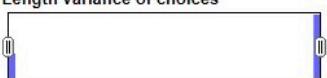
Cluster size	Row Count	Values in cluster	Merge?	New cell value
2	10	• PRINCESS FIELD (6 rows) • PRINCESS FIELD (4 rows)	<input type="checkbox"/>	PRINCESS FIELD
2	22	• MOUNT ZION (18 rows) • MOUNT ZION (4 rows)	<input type="checkbox"/>	MOUNT ZION
2	10	• SWABYS HOPE (5 rows) • SWABYS HOPE (5 rows)	<input type="checkbox"/>	SWABYS HOPE
2	9	• BROGUE HILL (5 rows) • BROGUE Hill (4 rows)	<input type="checkbox"/>	BROGUE HILL
2	9	• BALLARDS RIVER (7 rows) • BALLARDS RIVER (2 rows)	<input type="checkbox"/>	BALLARDS RIVER
2	5	• SPRING MOUNTAIN (3 rows) • MOUNTAIN SPRING (2 rows)	<input type="checkbox"/>	SPRING MOUNTAIN
2	19	• GRAVEL HILL (17 rows) • GRAVEL Hill (2 rows)	<input type="checkbox"/>	GRAVEL HILL

Rows in cluster

5 — 22

Average length of choices

11 — 15

Length variance of choices

0 — 1

Select all Deselect all Export clusters Merge selected & re-cluster Merge selected & Close Close

The clustering pop-up window will take a small amount of time to analyze your column, and then make some suggestions based on the clustering method currently active.

For each cluster identified, you can pick one of the existing values to apply to all cells, or manually type in a new value in the text box. And, of course, you can choose not to cluster them at all. OpenRefine will keep analyzing every time you make a change, with **Merge selected & re-cluster**, and you can work through all the methods this way.

You can also export the currently identified clusters as a JSON file, or close the window with or without applying your changes. You can also use the histograms on the right to

narrow down to, for example, clusters with lots of matching rows, or clusters of long or short values.

Clustering methods

You don't need to understand the details behind each clustering method to apply them successfully to your data.

The order in which these methods are presented in the interface and on this page is the order we recommend - starting with the most strict rules and moving to the laxest, which require more human supervision to apply correctly.

The clustering pop-up window offers you two categories of clustering methods: 6 different key collision clustering methods and 2 nearest neighbour clustering methods.

- [Key Collision](#)

The Key Collision category is itself subcategorized by [phonetic](#) or not-phonetic clustering.

- [Fingerprint](#)
- [N-gram Fingerprint](#)
- [Metaphone3](#)
- [Cologne Phonetic](#)
- [Daitch-Mokotoff](#)
- [Beider-Morse](#)

- [Nearest Neighbor](#)

- [Levenshtein](#)
- [PPM](#)

Key Collision

Key collisions are very fast and can process millions of cells in seconds:

Fingerprinting

Fingerprinting is the least likely to produce false positives, so it's a good place to start. It does the same kind of data cleaning behind the scenes that you might think to do manually:

- fix whitespace into single spaces
- put all uppercase letters into lowercase
- discard punctuation
- remove diacritics (e.g. accents) from characters
- split up all strings (words) and sort them alphabetically (so "Zhenyi, Wang" becomes "wang zhenyi").

For an in-depth understanding of fingerprinting, check this [document](#)

N-gram Fingerprinting

N-gram fingerprinting allows you to set the n value to whatever number you'd like and will create n-grams of n size (after doing some cleaning), alphabetize them, then join them back together into a fingerprint.

For example, a 1-gram fingerprint will simply organize all the letters in the cell into alphabetical order - by creating segments one character in length. A 2-gram fingerprint will find all the two-character segments, remove duplicates, alphabetize them, and join them back together (for example, "banana" generates "ba an na an na," which becomes "anbana").

This can help match cells that have typos, or incorrect spaces (such as matching

“lookout” and “look out,” which fingerprinting itself won’t identify because it separates words). The higher the n value, the fewer clusters will be identified. With 1-grams, keep an eye out for mismatched values that are near-anagrams of each other (such as “Wellington” and “Elgin Town”).

For an in-depth understanding of N-gram fingerprinting, check this [document](#)

Phonetic Clustering

The next four methods are phonetic algorithms: they identify letters that sound the same when pronounced out loud, and assess text values based on that (such as knowing that a word with an “S” might be a mistype of a word with a “Z”). They are great for spotting mistakes made by not knowing the spelling of a word or name after hearing it spoken aloud.

Metaphone3 Fingerprinting

Metaphone3 fingerprinting is an English-language phonetic algorithm. For example, “Reuben Gevorkiantz” and “Ruben Gevorkyants” share the same phonetic fingerprint in English.

Cologne Fingerprinting

Cologne fingerprinting is another phonetic algorithm, but for German pronunciation.

Daitch-Mokotoff

Daitch-Mokotoff is a phonetic algorithm for Slavic and Yiddish words, especially names.

Baider-Morse

Baider-Morse is a version of Daitch-Mokotoff that is slightly more strict.

Regardless of the language of your data, applying each of them might find different

potential matches: for example, Metaphone clusters “Cornwall” and “Corn Hill” and “Green Hill,” while Cologne clusters “Greenvale” and “Granville” and “Cornwall” and “Green Wall.”

For an in-depth understanding of phonetics, check this [document](#)

Nearest Neighbor

Nearest Neighbor clustering methods are slower than key collision methods.

They allow the user to set a radius - a threshold for matching or not matching. OpenRefine uses a “blocking” method first, which sorts values based on whether they have a certain amount of similarity (the default is “6” for a six-character string of identical characters) and then runs the nearest-neighbor operations on those sorted groups.

We recommend setting the block number to at least 3, and then increasing it if you need to be more strict (for example, if every value with “river” is being matched, you should increase it to 6 or more).

Note that bigger block values will take much longer to process, while smaller blocks may miss matches. Increasing the radius will make the matches more lax, as bigger differences will be clustered.

Levenshtein Distance

Levenshtein distance counts the number of edits required to make one value perfectly match another. As in the key collision methods above, it will do things like change uppercase to lowercase, fix whitespace, change special characters, etc. Each character that gets changed counts as 1 “distance.” “New York” and “newyork” have an edit distance value of 3 (“N” to “n”; “Y” to “y”; remove the space).

It can do relatively advanced edits, such as understanding the distance between “M. Makeba” and “Miriam Makeba” (5), but it may create false positives if these distances are greater than other, simpler transformations (such as the one-character distance to “B. Makeba,” another person entirely).

PPM (Prediction by Partial Matching)

PPM (Prediction by Partial Matching) uses compression to see whether two values are similar or different. In practice, this method is very lax even for small radius values and tends to generate many false positives, but because it operates at a sub-character level it is capable of finding substructures that are not easily identifiable by distances that work at the character level. So it should be used as a “last resort” clustering method. It is also more effective on longer strings than on shorter ones.

For more of the theory behind clustering, see [Clustering In Depth](#).

Replace

OpenRefine provides a find/replace function for you to edit your data. Selecting [Edit cells](#) → [Replace](#) will bring up a simple window where you can input a string to search and a string to replace it with. You can set case-sensitivity, and set it to only select whole words, defined by a string with spaces or punctuation around it (to prevent, for example, “house” selecting the “house” part of “doghouse”). You can use [regular expressions](#) in this field. You may wish to preview the results of this operation by testing it with a [Text filter](#) first.

You can also perform a sort of find/replace operation by editing one cell, and selecting “apply to all identical cells.”

Edit one cell at a time

You can edit individual cells by hovering your mouse over that cell. You should see a tiny blue link labeled “edit.” Click it to edit the cell. That pops up a window with a bigger text field for you to edit. You can change the [data type](#) of that cell, and you can apply these changes to all identical cells (in the same column), using this pop-up window.

You will likely want to avoid doing this except in rare cases - the more efficient means of improving your data will be through automated and bulk operations.

Column editing

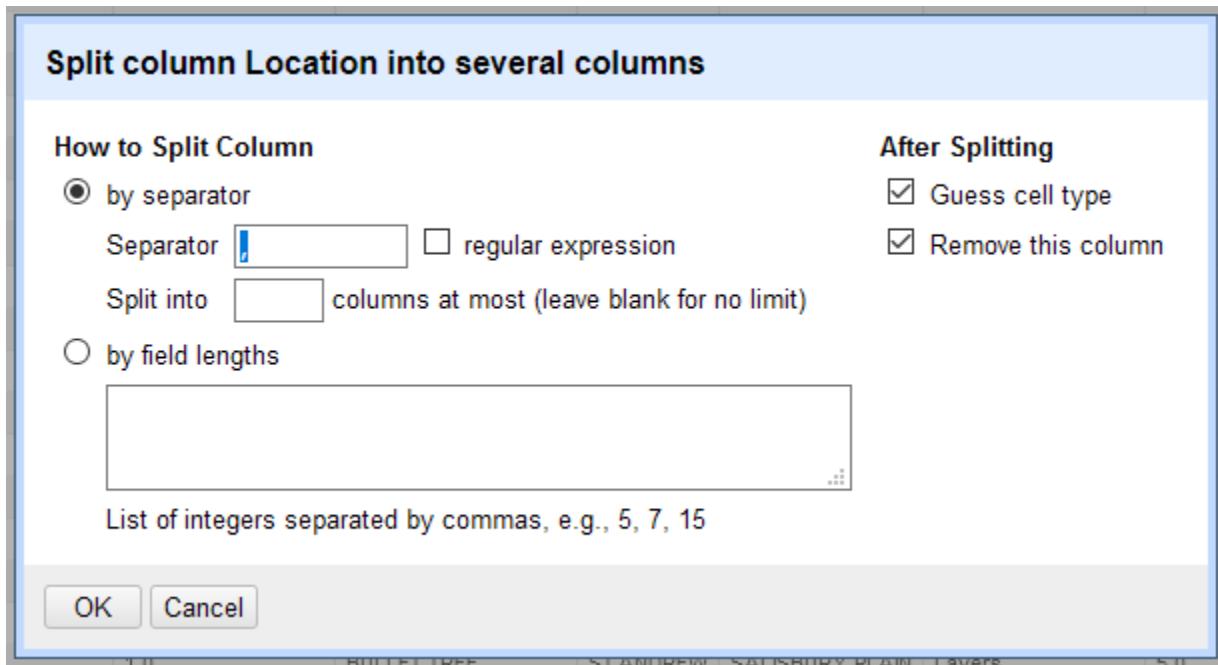
Overview

Column editing contains some of the most powerful data-improvement methods in OpenRefine. The operations in the `Edit column` menu involve using one column of data to add entirely new columns and fields to your dataset.

Splitting or joining

Many users find that they frequently need to make their data more granular: for example, splitting a “Firstname Lastname” column into two columns, one for first names and one for last names. The reverse is also often true: you may have several columns of category values that you want to join into one “category” column. .

Split into several columns

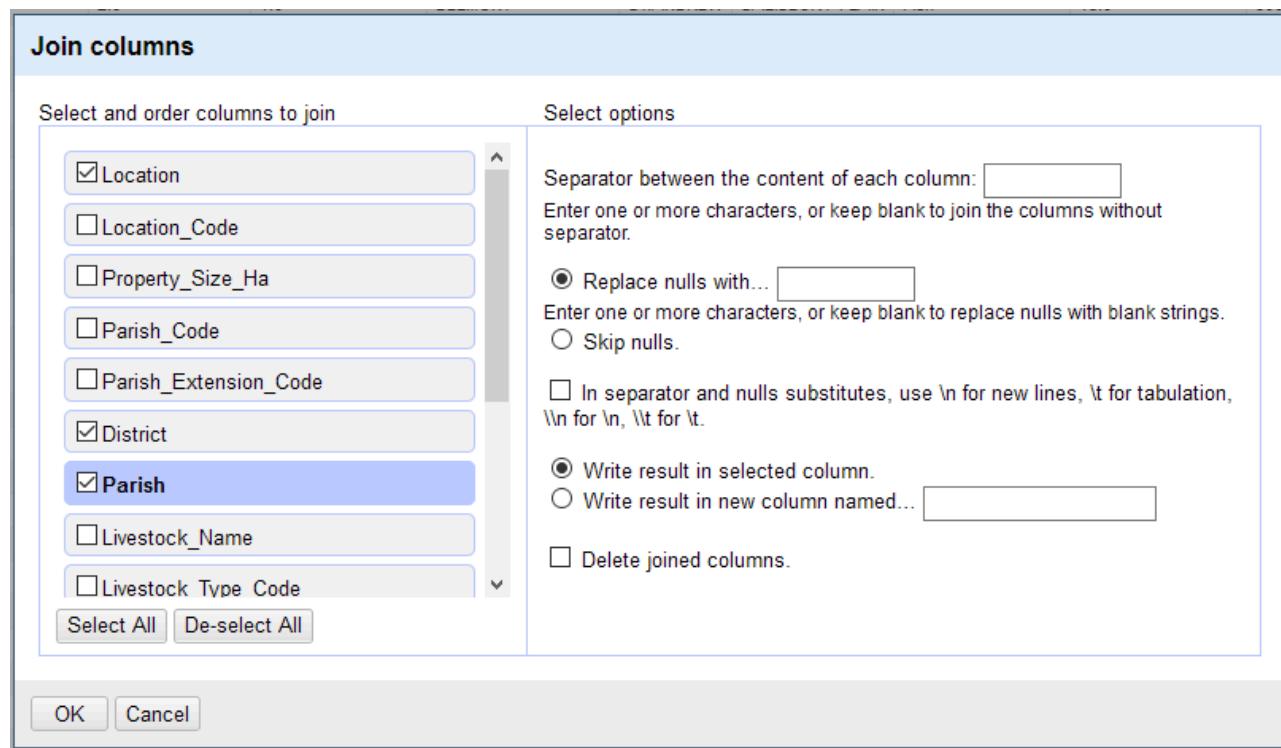


You can find this operation at [Edit column](#) → [Split into several columns...](#). Splitting one column into several columns requires you to identify the character, string lengths, or evaluating expression you want to split on. Just like [splitting multi-valued cells into rows](#), splitting cells into multiple columns will remove the separator character or string you indicate. Splitting by lengths will discard any information that comes after the specified total length.

You can also specify a maximum number of new columns to be made: separator characters after this limit will be ignored, and the remaining characters will end up in the last column.

New columns will be named after the original column, with a number: "Location 1," "Location 2," etc. You can choose to remove the original column with this operation, and you can have [data types](#) identified where possible. This function will work best with converting strings to numbers, and may not work with [dates](#).

Join columns



You can join columns by selecting [Edit column](#) → [Join columns...](#). All the columns currently in your dataset will appear in the pop-up window. You can select or un-select all the columns you want to join, and drag columns to put them in the order you want to join them in. You will define a separator character (optional) and define a string to insert into empty cells (nulls).

The joined data will appear in the column you originally selected, or you can create a new column for this content and specify a name. You can delete all the columns that were used in this join operation.

Add column based on this column

Selecting `Edit column` → `Add column based on this column...` will open up an [expressions](#) window where you can transform the data from this column (using `value`), or write a more complex expression that takes information from any number of columns or from external sources.

Expressions used in this operation will rely on your knowledge of variables. You can learn more in the [Expressions section on variables](#).

The simplest way to use this operation is simply leave the default `value` in the expression field, to create an exact copy of your column. For a column of [reconciled data](#), you can use the variable `cell` instead, to copy both the original string and the existing reconciliation data. This will include matched values, candidates, and new items.

One useful expression is to create a column based on concatenating (merging) two other columns. Select either of the source columns, choose `Edit column` → `Add column based on this column...`, name your new column, and use the following format in the expression window:

```
cells["Column 1"].value + cells["Column 2"].value
```

If your column names do not contain spaces, you can use the following format instead:

```
cells.Column1.value + cells.Column2.value
```

If you are in records mode instead of rows mode, you can concatenate using the following format:

```
row.record.cells.Column1.value + row.record.cells.Column2.value
```

You may wish to add separators or spaces, or modify your input during this operation with more advanced expressions.

Add column by fetching URLs

Through the **Add column by fetching URLs** function, OpenRefine supports the ability to fetch HTML or data from web pages or services. In this operation you will be building URL strings based on your column of data, by using **value** to insert a relevant substring. Your chosen column needs to contain parts of paths to valid HTML pages or files online.

If you have a column of URLs and want to fetch the information that they point to, you can simply run the expression as **value**. If your column has, for example, unique identifiers for Wikidata entities (numerical values starting with Q), you can download the JSON-formatted metadata about each entity with

```
"https://www.wikidata.org/wiki/Special:EntityData/" + value + ".json"
```

or whatever metadata format you prefer. Information about the format options in Wikidata can be found [here](#). The service you are fetching data from may have similar documentation on its provided options.

Add column by fetching URLs based on column Wikidata Entities

New column name: Throttle delay:

On error: set to blank store error Cache responses

HTTP headers to be used when fetching URLs: [Hide](#)

Authorization:

User-Agent:

Accept:

Formulate the URLs to fetch:

Expression: No syntax error.

Language: General Refine Expression Language (GREL) ▾

row	value	URL
1.	Q1125633	https://www.wikidata.org/wiki/Special:EntityData/Q1125633.json
2.	Q1376408	https://www.wikidata.org/wiki/Special:EntityData/Q1376408.json
3.	Q1741127	https://www.wikidata.org/wiki/Special:EntityData/Q1741127.json
4.	Q2156146	https://www.wikidata.org/wiki/Special:EntityData/Q2156146.json
5.	Q2405269	https://www.wikidata.org/wiki/Special:EntityData/Q2405269.json
6.	Q2864894	https://www.wikidata.org/wiki/Special:EntityData/Q2864894.json
7.	Q2864894	https://www.wikidata.org/wiki/Special:EntityData/Q2864894.json

Preview History Starred Help

OK Cancel

This service is more useful when getting metadata files instead of HTML, but you may wish to work with a page's entire HTML contents and then parse out information from that.

⚠ CAUTION

Be aware that the fetching process can take quite some time and that servers may not want to fulfill hundreds or thousands of page requests in seconds. Fetching

allows you to set a “throttle delay” which determines the amount of time between requests. The default is 5 seconds per row in your dataset (5000 milliseconds). We recommend leaving this at 1000 or greater.

Note the following:

- Before pressing “OK,” copy and paste a URL or two from the preview and test them in another browser tab to make sure they work.
- In some situations you may need to set [HTTP request headers](#). To set these, click the small “Show” button next to “HTTP headers to be used when fetching URLs” in the settings window. The authorization credentials get logged in your operation history in plain text, which may be a security concern for you. You can set the following request headers:
 - [User-Agent](#)
 - [Accept](#)
 - [Authorization](#)

Common errors

When OpenRefine attempts to fetch information from a web service, it can fail in a variety of ways. The following information is meant to help troubleshoot and fix problems encountered when using this function.

First, make sure that your fetching operation is storing errors (check “store error”). Then run the fetch and look at the error messages.

“**HTTP error 403 : Forbidden**” can be simply down to you not having access to the URL you are trying to use. If you can access the same URL with your browser, the remote site may be blocking OpenRefine because it doesn’t recognize its request as valid. Changing the [User-Agent](#) request header may help. If you believe you should have access to a site but

are “forbidden,” you may wish to contact the administrators.

“**HTTP error 404 : Not Found**” indicates that the information you are requesting does not exist, perhaps due to a problem with your cell values if it only happening in certain rows.

“**HTTP error 500 : Internal Server Error**” indicates the remote server is having a problem filling your request. You may wish to simply wait and try again later, or double-check the URLs.

“**error: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure**” can occur when you are trying to retrieve information over HTTPS but the remote site is using an encryption not supported by the Java virtual machine being used by OpenRefine.

You can check which encryption methods are supported by your OpenRefine/Java installation by using a service such as [How's my SSL](#). Add the URL <https://www.howsmyssl.com/a/check> to an OpenRefine cell and run “Add column by fetching URLs” on it, which will provide a description of the SSL client being used.

You can try installing additional encryption supports by installing the [Java Cryptography Extension](#). Note that for Mac users and for Windows users with the OpenRefine installation with bundled JRE, these updated cipher suites need to be dropped into the Java install within the OpenRefine application:

- On Mac, it will look something like /Applications/OpenRefine.app/Contents/PlugIns/jdk1.8.0_60.jdk/Contents/Home/jre/lib/security.
- On Windows: <\server\target\jre\lib\security>.

“**javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path building failed**” can appear when the remote site is using an HTTPS certificate not trusted by your local Java installation. You will need to make sure that the certificate, or (more likely) the root certificate, is trusted.

The list of trusted certificates is stored in an encrypted file called <cacerts> in your local

Java installation. This can be read and updated by a tool called “keytool.” You can find directions on how to add a security certificate to the list of trusted certificates for a Java installation [here](#) and [here](#).

Note that for Mac users and for Windows users with the OpenRefine installation with bundled JRE, the `cacerts` file within the OpenRefine application needs to be updated.

- On Mac, it will look something like `/Applications/OpenRefine.app/Contents/PlugIns/jdk1.8.0_60.jdk/Contents/Home/jre/lib/security/cacerts`.
- On Windows: `\server\target\jre\lib\security\`.

Renaming, removing, and moving

Every column's `Edit column` dropdown contains options to move it (to the beginning, end, left, or right), rename it, and delete it. These operations can be undone, but a removed column cannot be restored later if you keep modifying your data. If you wish to temporarily hide a column, go to `View` → `Collapse this column` instead.

Be cautious about moving columns in [records mode](#): if you change the first column in your dataset (the key column), your records may change in unintended ways.

Transposing

Overview

These functions were created to solve common problems with reshaping your data: pivoting cells from a row into a column, or pivoting cells from a column into a row. You can also transpose from a repeated set of values into multiple columns.

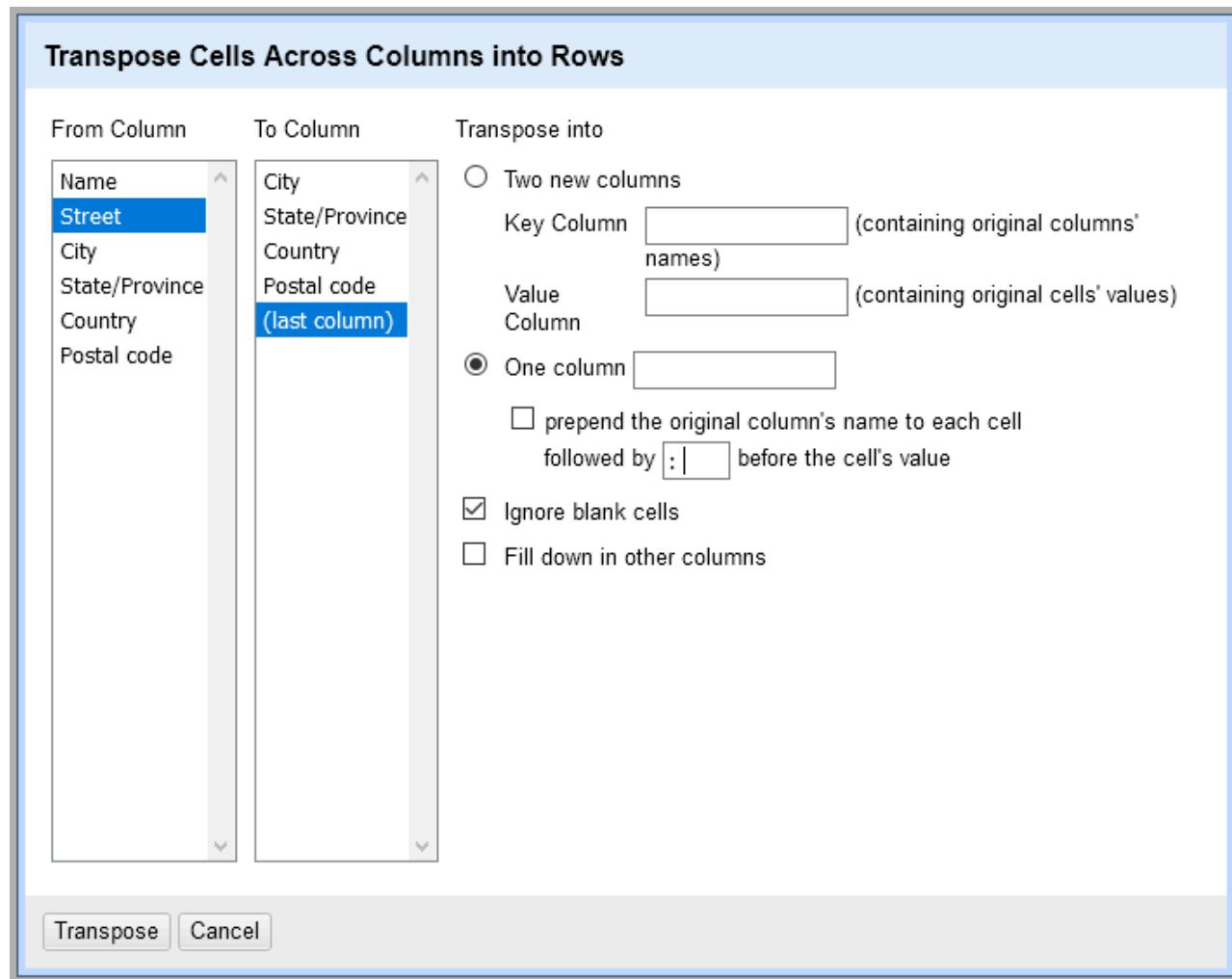
Transpose cells across columns into rows

Imagine personal data with addresses in this format:

Name	Street	City	State/ Province	Country	Postal code
Jacques Cousteau	23, quai de Conti	Paris		France	75270
Emmy Noether	010 N Merion Avenue	Bryn Mawr	Pennsylvania	USA	19010

You can transpose the address information from this format into multiple rows. Go to the “Street” column and select **Transpose** → **Transpose cells across columns into rows**. From there you can select all of the five columns, starting with “Street” and ending with “Postal code,” that correspond to address information. Once you begin, you should put your

project into [records mode](#) to associate the subsequent rows with “Name” as the key column.



One column

You can transpose the multiple address columns into a series of rows:

Name	Address
Jacques Cousteau	23, quai de Conti

Name	Address
	Paris
	France
	75270
Emmy Noether	010 N Merion Avenue
	Bryn Mawr
	Pennsylvania
	USA
	19010

You can choose one column and include the column-name information in each cell by prepending it to the value, with or without a separator:

Name	Address
Jacques Cousteau	Street: 23, quai de Conti
	City: Paris
	Country: France
	Postal code: 75270

Name	Address
Emmy Noether	Street: 010 N Merion Avenue
	City: Bryn Mawr
	State/Province: Pennsylvania
	Country: USA
	Postal code: 19010

Two columns

You can retain the column names as separate cell values, by selecting [Two new columns](#) and naming the key and value columns.

Name	Address part	Address
Jacques Cousteau	Street	23, quai de Conti
	City	Paris
	Country	France
	Postal code	75270
Emmy Noether	Street	010 N Merion Avenue

Name	Address part	Address
	City	Bryn Mawr
	State/Province	Pennsylvania
	Country	USA
	Postal code	19010

Transpose cells in rows into columns

Imagine employee data in this format:

Column
Employee: Karen Chiu
Job title: Senior analyst
Office: New York
Employee: Joe Khoury
Job title: Junior analyst
Office: Beirut

Column
Employee: Samantha Martinez
Job title: CTO
Office: Tokyo

The goal is to sort out all of the information contained in one column into separate columns, but keep it organized by the person it represents:

Name	Job title	Office
Karen Chiu	Senior analyst	New York
Joe Khoury	Junior analyst	Beirut
Samantha Martinez	CTO	Tokyo

By selecting **Transpose** → **Transpose cells in rows into columns...** a window will appear that simply asks how many rows to transpose. In this case, each employee record has three rows, so input “3” (do not subtract one for the original column). The original column will disappear and be replaced with three columns, with the name of the original column plus a number appended.

Column 1	Column 2	Column 3
Employee: Karen Chiu	Job title: Senior analyst	Office: New York

Column 1	Column 2	Column 3
Employee: Joe Khoury	Job title: Junior analyst	Office: Beirut
Employee: Samantha Martinez	Job title: CTO	Office: Tokyo

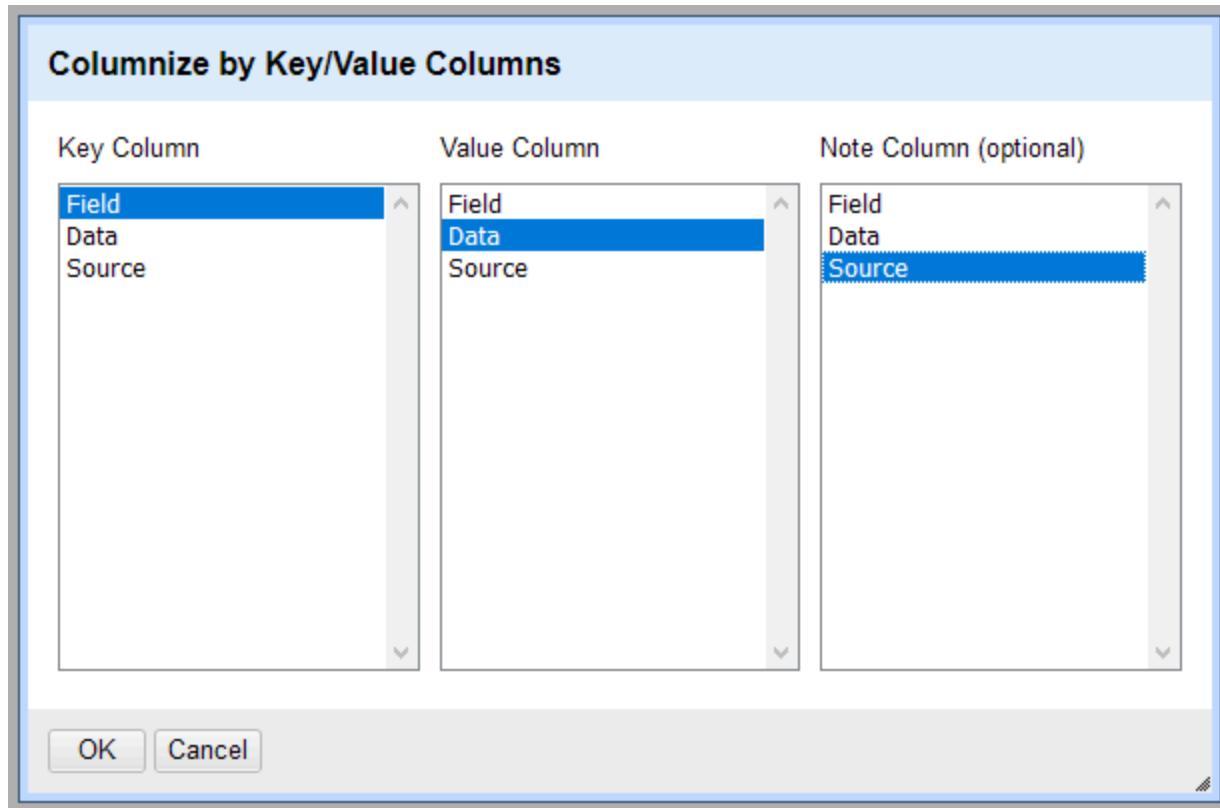
From here you can use [Cell editing](#) → [Replace](#) to remove “Employee:”, “Job title:”, and “Office:” if you wish, or use [expressions](#) with [Edit cells](#) → [Transform...](#) to clean out the extraneous characters:

```
value.replace("Employee: ", "")
```

If your dataset doesn't have a predictable number of cells per intended row, such that you cannot specify easily how many columns to create, try [Columnize by key/value columns](#).

Columnize by key/value columns

This operation can be used to reshape a dataset that contains key and value columns: the repeating strings in the key column become new column names, and the contents of the value column are moved to new columns. This operation can be found at [Transpose](#) → [Columnize by key/value columns](#).



Consider the following example, with flowers, their colours, and their International Union for Conservation of Nature (IUCN) identifiers:

Field	Data
Name	Galanthus nivalis
Color	White
IUCN ID	162168
Name	Narcissus cyclamineus
Color	Yellow

Field	Data
IUCN ID	161899

In this format, each flower species is described by multiple attributes on consecutive rows. The “Field” column contains the keys and the “Data” column contains the values. In the [Columnize by key/value columns](#) window you can select each of these from the available columns. It transforms the table as follows:

Name	Color	IUCN ID
Galanthus nivalis	White	162168
Narcissus cyclamineus	Yellow	161899

Entries with multiple values in the same column

If a new row would have multiple values for a given key, then these values will be grouped on consecutive rows, to form a [record structure](#).

For instance, flowers can have multiple colors:

Field	Data
Name	Galanthus nivalis
Color	White

Field	Data
Color	Green
IUCN ID	162168
Name	Narcissus cyclamineus
Color	Yellow
IUCN ID	161899

This table is transformed by the Columnize operation to:

Name	Color	IUCN ID
Galanthus nivalis	White	162168
	Green	
Narcissus cyclamineus	Yellow	161899

The first key encountered by the operation serves as the record key, so the “Green” value is attached to the “Galanthus nivalis” name. See the [Row order](#) section for more details about the influence of row order on the results of the operation.

Notes column

In addition to the key and value columns, you can optionally add a column for notes. This

can be used to store extra metadata associated to a key/value pair.

Consider the following example:

Field	Data	Source
Name	<i>Galanthus nivalis</i>	IUCN
Color	White	Contributed by Martha
IUCN ID	162168	
Name	<i>Narcissus cyclamineus</i>	Legacy
Color	Yellow	2009 survey
IUCN ID	161899	

If the “Source” column is selected as the notes column, this table is transformed to:

Name	Color	IUCN ID	Source: Name	Source: Color
<i>Galanthus nivalis</i>	White	162168	IUCN	Contributed by Martha
<i>Narcissus cyclamineus</i>	Yellow	161899	Legacy	2009 survey

Notes columns can therefore be used to preserve provenance or other context about a

particular key/value pair.

Row order

The order in which the key/value pairs appear matters. The Columnize operation will use the first key it encounters as the delimiter for entries: every time it encounters this key again, it will produce a new row, and add the following key/value pairs to that row.

Consider for instance the following table:

Field	Data
Name	Galanthus nivalis
Color	White
IUCN ID	162168
Name	Crinum variabile
Name	Narcissus cyclamineus
Color	Yellow
IUCN ID	161899

The occurrences of the “Name” value in the “Field” column define the boundaries of the entries. Because there is no other row between the “Crinum variabile” and the “Narcissus cyclamineus” rows, the “Color” and “IUCN ID” columns for the “Crinum variabile” entry will be empty:

Name	Color	IUCN ID
Galanthus nivalis	White	162168
Crinum variable		
Narcissus cyclamineus	Yellow	161899

This sensitivity to order is removed if there are extra columns: in that case, the first extra column will serve as the key for the new rows.

Extra columns

If your dataset contains extra columns, that are not being used as the key, value, or notes columns, they can be preserved by the operation. For this to work, they must have the same value in all old rows corresponding to a new row.

In the following example, the “Field” and “Data” columns are used as key and value columns respectively, and the “Wikidata ID” column is not selected:

Field	Data	Wikidata ID
Name	Galanthus nivalis	Q109995
Color	White	Q109995
IUCN ID	162168	Q109995
Name	Narcissus cyclamineus	Q1727024
Color	Yellow	Q1727024

Field	Data	Wikidata ID
IUCN ID	161899	Q1727024

This will be transformed to:

Wikidata ID	Name	Color	IUCN ID
Q109995	Galanthus nivalis	White	162168
Q1727024	Narcissus cyclamineus	Yellow	161899

This actually changes the operation: OpenRefine no longer looks for the first key ("Name") but simply pivots all information based on the first extra column's values. Every old row with the same value gets transposed into one new row. If you have more than one extra column, they are pivoted as well but not used as the new key.

You can use [Fill down](#) to put identical values in the extra columns if you need to.

Reconciling

Overview

Reconciliation is the process of matching your dataset with that of an external source. Datasets for comparison might be produced by libraries, archives, museums, academic organizations, scientific institutions, non-profits, or interest groups. You can also reconcile against user-edited data on [Wikidata or other Wikibase instances](#), or reconcile against [a local dataset that you yourself supply](#).

To reconcile your OpenRefine project against an external dataset, that dataset must offer a web service that conforms to the [Reconciliation Service API standards](#).

You may wish to reconcile in order to:

- fix spelling or variations in proper names
- clean up manually-entered subject headings against authorities such as the [Library of Congress Subject Headings \(LCSH\)](#)
- link your data to an existing dataset
- add to an editable platform such as [Wikidata](#)
- or see whether entities in your project appear in some specific list, such as the [Panama Papers](#).

Reconciliation is semi-automated: OpenRefine matches your cell values to the reconciliation information as best it can, but human judgment is required to review and approve the results. Reconciling happens by default through string searching, so typos, whitespace, and extraneous characters will have an effect on the results. You may wish to [clean and cluster](#) your data before reconciliaton.

WORKING ITERATIVELY

We recommend planning your reconciliation operations as iterative: reconcile multiple times with different settings, and with different subgroups of your data.

Sources

Start with [this current list of reconcilable authorities](#), which includes instructions for adding new services via Wikidata editing if you have one to add.

OpenRefine maintains a [further list of sources on the wiki](#), which can be edited by anyone. This list includes ways that you can reconcile against a [local dataset](#).

Other services may exist that are not yet listed in these two places: for example, the [310 datasets hosted by the Organized Crime and Corruption Reporting Project \(OCCRP\)](#) each have their own reconciliation URL, or you can reconcile against their entire database with the URL [shared on the reconciliation API list](#). For another example, you can reconcile against the entire Virtual International Authority File (VIAF) dataset, or [only the contributions from certain institutions](#). Search online to see if the authority you wish to reconcile against has an available service, or whether you can download a copy to reconcile against locally.

OpenRefine includes Wikidata reconciliation in the installation package - see the [Wikibase](#) page for more information particular to that service. Extensions can add reconciliation services, and can also add enhanced reconciliation capacities. Check the list of extensions on the [Downloads page](#) for more information.

Each source will have its own documentation on how it provides reconciliation. The table on [the reconciliation API list](#) indicates whether your chosen service supports the features described below. Refer to the service's documentation if you have questions about its behaviors and which OpenRefine features it supports.

In addition to the reconciliation services mentioned above, you may also choose to build your own service. You can either start from scratch using the [API specification](#) or use one of the frameworks mentioned in the [Reconciliation census](#).

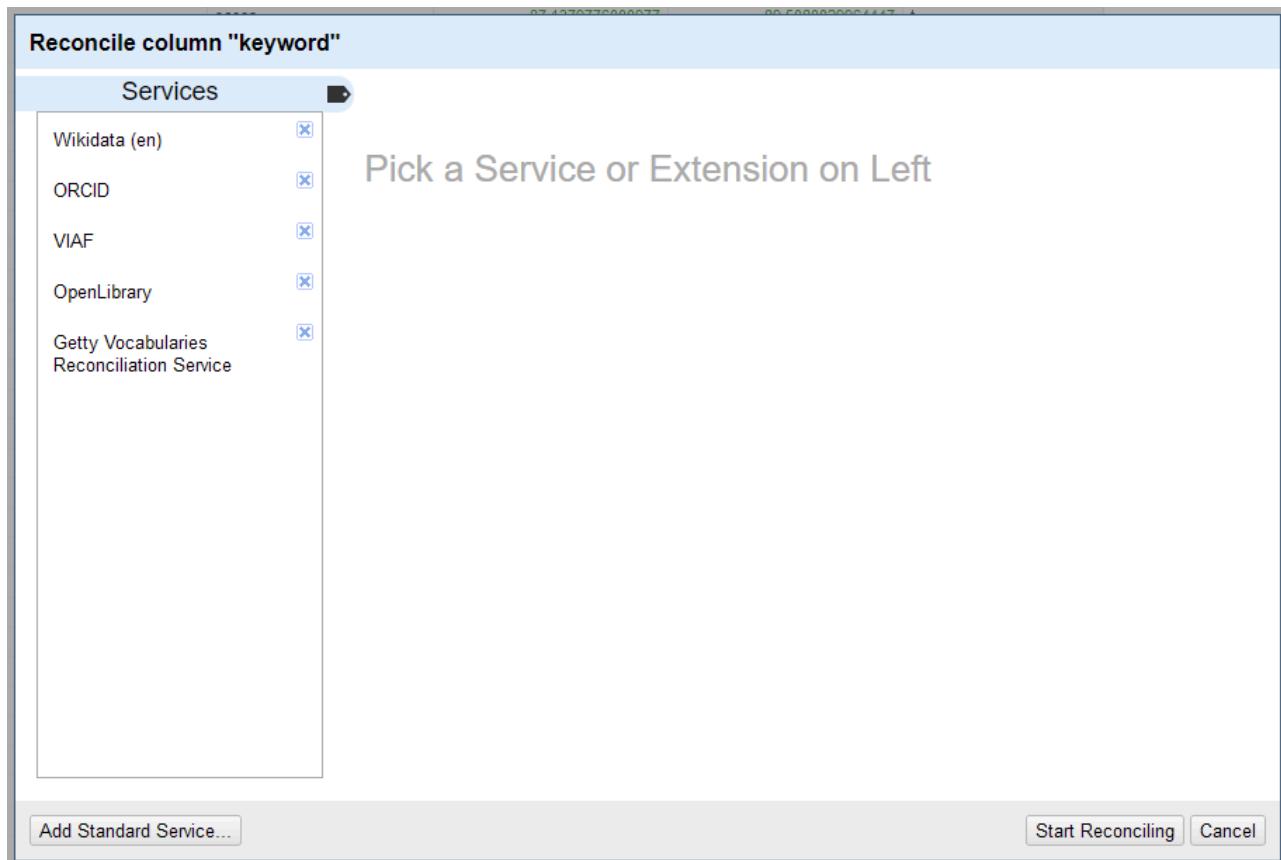
Of particular note is [reconcile-csv](#) which allows you to build a reconciliation service from a simple CSV file. Thus if you wanted to reconcile one OpenRefine project against another, you'd simply need to export the target project as a CSV, point [reconcile-csv](#) at it and you're good to go. A somewhat newer port of this project written in Python can be found at [csv-reconcile](#) which is more configurable and defaults to parsing tab separated files for convenience.

Similarly, you may choose to export some SPARQL output to a TSV to limit the scope of values you're reconciling against and/or for better performance.

Getting started

Choose a column to reconcile and use its dropdown menu to select [Reconcile](#) → [Start reconciling](#). If you want to reconcile only some cells in that column, first use filters and facets to isolate them.

In the reconciliation window, you will see Wikidata offered as a default service. To add another service, click [Add Standard Service...](#) and paste in the URL of a [service](#). You should see the name of the service appear in the list of [Services](#) if the URL is correct.



Once you select a service, your selected column may be sampled in order to suggest “types” (categories) to reconcile against. Other services will suggest their available types without sampling, and some services have no types.

For example, if you had a list of artists represented in a gallery collection, you could reconcile their names against the Getty Research Institute’s [Union List of Artist Names \(ULAN\)](#). The same [Getty reconciliation URL](#) will offer you ULAN, AAT (Art and Architecture Thesaurus), and TGN (Thesaurus of Geographic Names).

Reconcile column "keyword"

Reconcile each cell to an entity of one of these types:

- ULAN search
/ulan
- TGN search
/tgn
- AAT search
/aat
- Search all Vocab
/all

Also use relevant details from other columns:

Column	Include? As Property
photo_id	<input type="checkbox"/>
Photographer_username	<input type="checkbox"/>
ai_service_1_confidence	<input type="checkbox"/>
ai_service_2_confidence	<input type="checkbox"/>
suggested_by_user	<input type="checkbox"/>

Reconcile against type:

 Reconcile against no particular type

Auto-match candidates with high confidence

Maximum number of candidates to return

Refer to the [documentation specific to the reconciliation service](#) to learn whether types are offered, which types are offered, and which one is most appropriate for your column. You may wish to facet your data and reconcile batches against different types if available.

Reconciliation can be a time-consuming process, especially with large datasets. We suggest starting with a small test batch. There is no throttle (delay between requests) to set for the reconciliation process. The amount of time will vary for each service, and vary based on the options you select during the process.

When the process is done, you will see the reconciliation data in the cells. If the cell was successfully matched, it displays text as a single dark blue link. In this case, the reconciliation is confident that the match is correct, and you should not have to check it manually. If there is no clear match, one or more candidates are displayed, together with their reconciliation score, with the text in light blue links. You will need to select the correct one.

For each matching decision you make, you have two options: match this cell only (one checkmark), or also use the same identifier for all other cells containing the same original string (two checkmarks).

For services that offer the “[preview entities](#)” feature, you can hover your mouse over the suggestions to see more information about the candidates or matches. Each participating service (and each type) will deliver different structured data that may help you compare the candidates.

For example, the Getty ULAN shows an artist’s discipline, nationality, and birth and death years:

 **OpenRefine** Icelandic women artists [Permalink](#)

Facet / Filter Undo / Redo 29 / 29

12 rows

Show as: rows records Show: 5 10 25 50 rows

All	Artist	Lifespan	Profession
1.	Anna Jóelsdóttir <input checked="" type="checkbox"/> Hübschmannov'a, Anna (11) <input checked="" type="checkbox"/> Macková, Anna (11) <input checked="" type="checkbox"/> Neborová, Anna (11) <input checked="" type="checkbox"/> Tweeddale, Anna (11) <input checked="" type="checkbox"/> Anders, Anna (11) <input checked="" type="checkbox"/> Beeck, Anna (11) <input checked="" type="checkbox"/> Heindl, Anna (11) <input checked="" type="checkbox"/> Jermolaewa, Anna (11) <input checked="" type="checkbox"/> Waser, Anna (11) <input checked="" type="checkbox"/> Matoušková, Anna (11) <input checked="" type="checkbox"/> Create new item	born 1947	contemporary artist
2.	Fridriksdóttir, Gabriela Choose new match	born 1971	painter, sculptor
3.	Helgadóttir, Gerður Choose new match	1928–1975	sculptor, stained-glass artist
4.	Gunnfríður Jónsdóttir <input checked="" type="checkbox"/> Jónsdóttir, Jóní (19) <input checked="" type="checkbox"/> Create new item	edit 1889–1968	sculptor
5.	Sveinsdóttir, Júliana Choose new match		
6.	Sigurdardóttir, Katrín Choose new match		
7.	Kristín Jónsdóttir <input checked="" type="checkbox"/> Jónsdóttir, Jóní (19) <input checked="" type="checkbox"/> Create new item		
8.	Matthiassdóttir, Louisa Choose new match		
9.	Margret the Adroit		

Match this Cell Match All Identical Cells Cancel

Jónsdóttir, Jóní
(Icelandic artist, born 1972)

Hovering over the suggestion will also offer the two matching options as buttons.

For matched values (those appearing as dark blue links), the underlying cell value has not

been altered - the cell is storing both the original string and the matched entity link at the same time. If you were to copy your column to a new column at this point using [value](#), for example, the reconciliation data would not transfer - only the original strings. You can learn more about how OpenRefine stores different pieces of information in each cell in [the Variables section specific to reconciliation data](#).

For each cell, you can manually “Create new item,” which will take the cell’s original value and apply it, as though it is a match. This will not become a dark blue link, because at this time there is nothing to link to: it is a draft entity stored only in your project. You can use this feature to prepare these entries for eventual upload to an editable service such as [Wikibase](#), but most services do not yet support this feature.

Reconciliation facets

Under [Reconcile](#) → [Facets](#) there are a number of reconciliation-specific facetting options. OpenRefine automatically creates two facets when you reconcile some cells.

One is a numeric facet for “best candidate’s score,” the range of reconciliation scores of only the best candidate of each cell. Higher scores mean better matches, although each service calculates scores differently and has a different range. You can facet for higher scores using the numeric facet, and then approve them all in bulk, by using [Reconcile](#) → [Actions](#) → [Match each cell to its best candidate](#).

There is also a “judgment” facet created, which lets you filter for the cells that haven’t been matched (pick “none” in the facet). As you process each cell, its judgment changes from “none” to “matched” and it disappears from the view.

You can add other facets by selecting [Reconcile](#) → [Facets](#) on your reconciled column. You can facet by:

- your judgments (“matched,” or “none” for unreconciled cells, or “new” for entities you’ve created)

- the action you've performed on that cell (chosen a "single" match, or set a "mass" match, or no action, which appears as "unknown")
- the timestamps on the edits you've made so far (these appear as millisecond counts since an arbitrary point: they can be sorted alphabetically to move forward and back in time).

You can facet only the best candidates for each cell, based on:

- the score (calculated based on each service's own methods)
- the edit distance (using the [Levenshtein distance](#), a number based on how many single-character edits would be required to get your original value to the candidate value, with a larger value being a greater difference)
- the word similarity.

Word similarity is calculated as a percentage based on how many words (excluding [stop words](#)) in the original value match words in the candidate. For example, the value "Maria Luisa Zuloaga de Tovar" matched to the candidate "Palacios, Luisa Zuloaga de" results in a word similarity value of 0.6, or 60%, or 3 out of 5 words. Cells that are not yet matched to one candidate will show as 0.0).

You can also look at each best candidate's:

- type (the ones you have selected in successive reconciliation attempts, or other types returned by the service based on the cell values)
- type match ("true" if you selected a type and it succeeded, "false" if you reconciled against no particular type, and "(no type)" if it didn't reconcile)
- name match ("true" if you've matched, "false" if you haven't yet chosen from the candidates, or "(unreconciled)" if it didn't reconcile).

These facets are useful for doing successive reconciliation attempts, against different types, and with different supplementary information. The information represented by these facets are held in the cells themselves and can be called using the [reconciliation](#)

variables available in expressions.

Reconciliation actions

You can use the **Reconcile** → **Actions** menu options to perform bulk changes (which will apply only to your currently viewed set of rows or records):

- **Match each cell to its best candidate** (by highest score)
- **Create a new item for each cell** (discard any suggested matches)
- **Create one new item for similar cells** (a new entity will be created for each unique string)
- **Match all filtered cells to...** (a specific item from the chosen service, via a search box; only works with services that support the “suggest entities” property)
- **Discard all reconciliation judgments** (reverts back to multiple candidates per cell, including cells that may have been auto-matched in the original reconciliation process)
- **Clear reconciliation data**, reverting all cells back to their original values.

The other options available under **Reconcile** are:

- **Copy reconciliation data...** (to an existing column: if the original values in your reconciliation column are identical to those in your chosen column, the matched and new cells will copy over; unmatched values will not change)
- **Use values as identifiers** (if you are reconciling with unique identifiers instead of by doing string searches)
- **Add entity identifiers column**.

Reconciling with unique identifiers

Reconciliation services use unique identifiers for their entities. For example, the 14th Dalai Lama has the VIAF ID [38242123](#) and the Wikidata ID [Q17293](#). You can supply your known identifiers in a column to immediately match with the known IDs from a reconciliation service in order to subsequently pull more data about them.

Select the column with unique known identifiers of a service and apply the operation [Reconcile](#) → [Use values as identifiers](#). If you use this operation on a column of known IDs, you will not have access to the usual reconciliation settings. Instead, this will bring up a dialog to select a service from the list of reconciliation services you have already added (to add a new service, open the [Start reconciling...](#) window first).

! BEFORE RUNNING THIS OPERATION, ENSURE THAT THE IDS IN YOUR COLUMN ALREADY EXIST WITH THE RECONCILIATION SERVICE. [USE VALUES AS IDENTIFIERS](#) ASSUMES THEY EXIST WITH THE SERVICE, AND SO DOES NOT PERFORM A LOOKUP OR "RECONCILE" AGAINST THE SERVICE.

This operation is typically very fast and after it completes all cells will appear as dark blue “confirmed” matches.

You may get false positives (ex. the IDs no longer exist with the service), which you will need to hover over or click on to identify:

The screenshot shows the OpenRefine interface with a reconciliation service active. The top navigation bar includes 'Facet / Filter', 'Undo / Redo 1 / 1', and 'Extensions: Wikidata'. The main area displays '16 rows' with 'Show as: rows records' and 'Show: 5 10 25 50 rows' options. A sidebar on the left titled 'Using facets and filters' provides instructions and links to screencasts. The main workspace shows a column labeled 'Column 1' with five rows. Row 1 is selected and highlighted in blue, showing the value 'Q172931342edit' and the instruction 'Choose new match'. To the right of the table, a JSON response from the Wikidata service is displayed:

```
{"status": "error", "message": "invalid query", "details": "'Q172931342235'", "arguments": {"id": "Q172931342235", "lang": "en"}}
```

Reconciling by type

Reconciliation services, once added to OpenRefine, may suggest types from their databases. These types will usually be whatever the service specializes in: people, events, places, buildings, tools, plants, animals, organizations, etc.

Reconciling against a type may be faster and more accurate, but may result in fewer matches. Some services have hierarchical types (such as “mammal” as a subtype of “animal”). When you reconcile against a more specific type, unmatched values may fall back to the broader type; other services will not do this, so you may need to perform successive reconciliation attempts against different types. Refer to the documentation specific to the reconciliation service to learn more.

When you select a service from the list, OpenRefine will load some or all available types. Some services will sample the first ten rows of your column to suggest types (check the “Suggest types” column). You will see a service’s types in the reconciliation window:

Reconcile column "Artist"

» Access Service API

Reconcile each cell to an entity of one of these types:

Person
[/people/person](#)

Corporate Name
[/organization/organization](#)

Also use relevant details from other columns:

Column	Include? As Property
Lifespan	<input type="checkbox"/>
Profession	<input type="checkbox"/>

Reconcile against type:

Reconcile against no particular type

Auto-match candidates with high confidence

Maximum number of candidates to return

[Add Standard Service...](#) [Start Reconciling](#) [Cancel](#)

In this example, “Person” and “Corporate Name” are potential types offered by the reconciliation API for VIAF. You can also use the **Reconcile against type:** field to enter in another type that the service offers. When you start typing, this field may search and suggest existing types. For VIAF, you could enter “/book/book” if your column contained publications. You may need to enter the service's own strings precisely instead of attempting to search for a match.

Types are structured to fit their content: the Wikidata “human” type, for example, can include fields for birth and death dates, nationality, etc. The VIAF “person” type can include nationality and gender. You can use this to [include more properties](#) and find better matches.

If your column doesn't fit one specific type offered, you can [Reconcile against no particular type](#). This may take longer.

We recommend working in batches and reconciling against different types, moving from specific to broad. You can create a facet for **Best candidate's types** facet to see which types are being represented. Some candidates may return more than one type, depending on the service. Types may appear in facets by their unique IDs, rather than by their semantic labels (for example, Q5 for “human” in Wikidata).

Reconciling with additional columns

Some of your cells may be ambiguous, in the sense that a string can point to more than one entity: there are dozens of places called “Paris” and many characters, people, and pieces of culture, too. Selecting non-geographic or more localized types can help narrow that down, but if your chosen service doesn't provide a useful type, you can include more properties that make it clear whether you're looking for Paris, France.

The screenshot shows a reconciliation interface with two facets: 'All' and 'Place'. The 'Place' facet is selected and displays a list of entities matching the query 'Paris'. The list includes:

- Paris (100) (checked)
- Paris (100) (checked)
- 3317 Paris (100) (checked)
- Paris (100) (checked)
- Paris (100) (checked)
- Paris Bordone (100) (checked)
- Paris Saint-Germain (100) (checked)
- Paris (100) (checked)
- Paris (100) (checked)
- Create new item

At the bottom of the list, there is a link to 'Search for match'.

Including supplementary information can be useful, depending on the service (such as including birthdate information about each person you are trying to reconcile). You can reconcile unmatched cells with additional properties, in the right side of the

Start reconciling window, under “Also use relevant details from other columns.” The column names in your project will appear in the reconciliation window, with an **Include?** checkbox next to each one.

Fill in the **As Property** field with the type of information you are including. When you start typing, potential fields may pop up (depending on the “**suggest properties**” feature), such as “birthDate” in the case of ULAN or “Geburtsdatum” in the case of Integrated Authority File (GND). Use the documentation for your chosen service to identify the fields in their terms.

Some services will not be able to search for the exact name of your desired **As Property** entry, but you can still manually supply the field name. Refer to the service to choose the most appropriate field, and make sure you enter it correctly.

Reconcile column "Artist"

» Access Service API

Reconcile each cell to an entity of one of these types:

ULAN search /ulan
 TGN search /tgn
 AAT search /aat
 Search all Vocab /all

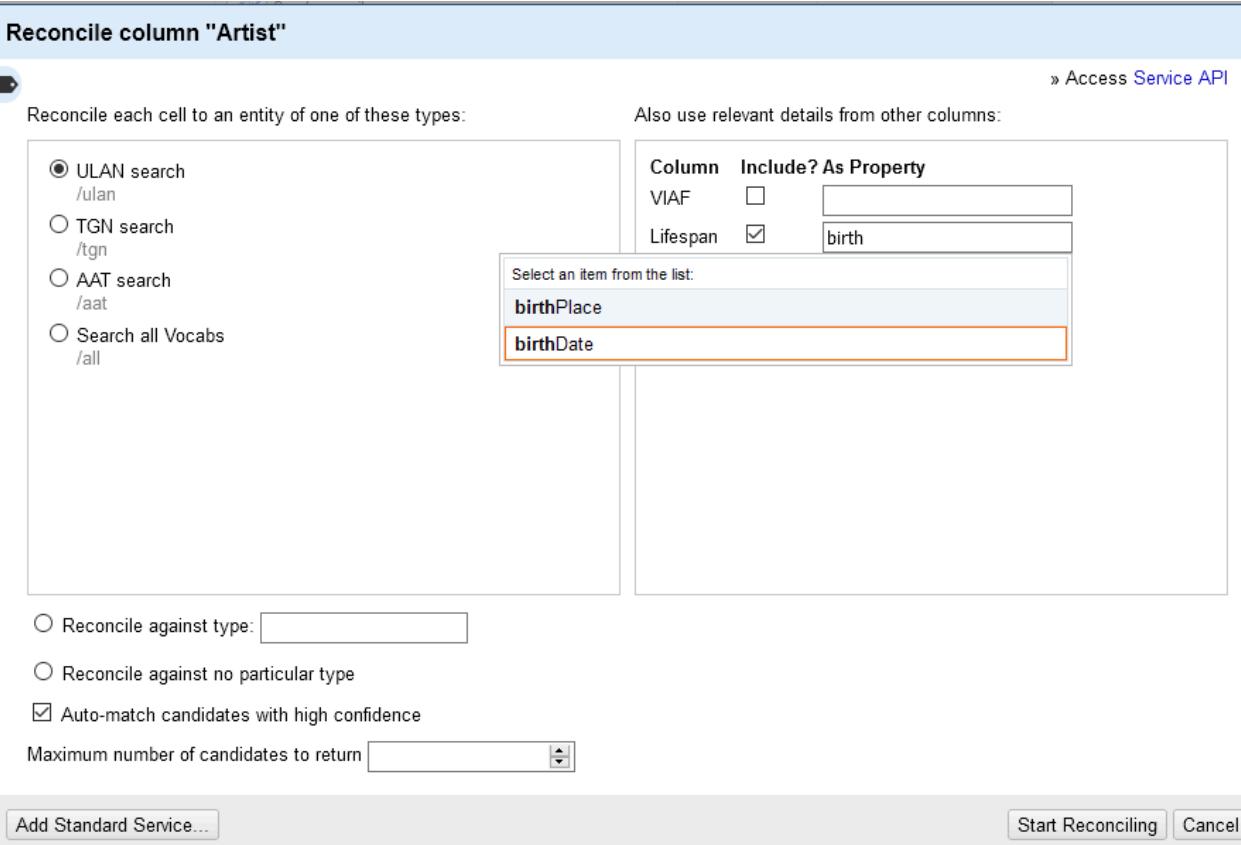
Also use relevant details from other columns:

Column	Include?	As Property
VIAF	<input type="checkbox"/>	
Lifespan	<input checked="" type="checkbox"/>	birth

Select an item from the list:
birthPlace
birthDate

Reconcile against type: []
 Reconcile against no particular type
 Auto-match candidates with high confidence
Maximum number of candidates to return []

Add Standard Service... Start Reconciling Cancel

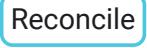


Fetching more data

One reason to reconcile to some external service is that it allows you to pull data from that service into your OpenRefine project. There are three ways to do this:

- Add identifiers for your values
- Add columns from reconciled values
- Add column by fetching URLs.

Add entity identifiers column

Once you have selected matches for your cells, you can retrieve the unique identifiers for those cells and create a new column for these, with  → . You will be asked to supply a column name. New items and other unmatched cells will generate null values in this column.

Add columns from reconciled values

If the reconciliation service supports [data extension](#), then you can augment your reconciled data with new columns using  → .

For example, if you have a column of chemical elements identified by name, you can fetch categorical information about them such as their atomic number and their element symbol:

<input type="checkbox"/> All	<input type="checkbox"/> chemical elem	
		1. hydrogen Choose new match
		2. helium Choose new match
		3. lithium Choose new match
		4. beryllium Choose new match
		5. boron Choose new match
		6. carbon Choose new match
		7. nitrogen Choose new match
		8. oxygen Choose new match
		9. fluorine Choose new match
		10. neon Choose new match
		11. sodium Choose new match
		12. magnesium Choose new match
		13. aluminum Choose new match
		14. silicon Choose new match
		15. francium Choose new match
		16. phosphorus Choose new match
		17. iron Choose new match
		18. sulfur Choose new match
		19. chlorine Choose new match
		20. argon Choose new match
		21. potassium Choose new match
		22. calcium Choose new match
		23. lead Choose new match

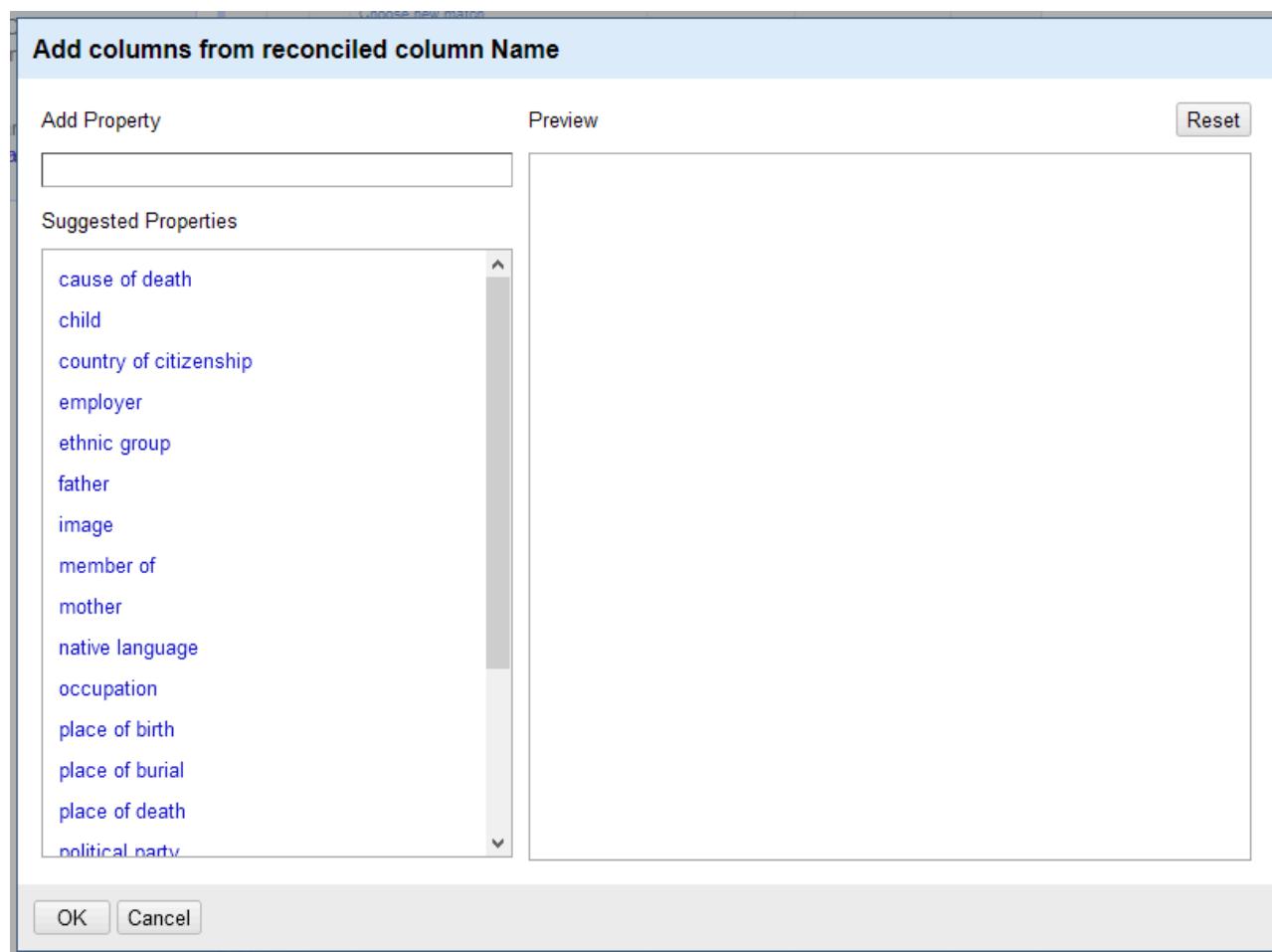
Once you have chosen reconciliation matches for your cells, selecting

Add column from reconciled values... will bring up a window to choose which related

information you'd like to import into new columns. You can manually enter desired properties, or select from a list of suggestions.

The quality of the suggested properties will depend on how you have reconciled your data beforehand: reconciling against a specific type will provide you with the associated

properties of that type. For example, GND suggests elements about the “people” type after you've reconciled with it, such as their parents, native languages, children, etc.



If you have left any values unreconciled in your column, you will see “<not reconciled>” in the preview. These will generate blank cells if you continue with the column addition process.

This process may pull more than one property per row in your data (such as multiple occupations), so you may need to switch into records mode after you've added columns.

Add columns by fetching URLs

If the reconciliation service cannot extend data, look for a generic web API for that data

source, or a structured URL that points to their dataset entities via unique IDs (such as “<https://viaf.org/viaf/000000>”). You can use the [Edit column](#) → [Add column by fetching URLs](#) operation to call this API or URL with the IDs obtained from the reconciliation process. This will require using [expressions](#).

You may not want to pull the entire HTML content of the pages at the ends of these URLs, so look to see whether the service offers a metadata endpoint, such as JSON-formatted data. You can either use a column of IDs, or you can pull the ID from each matched cell during the fetching process.

For example, if you have reconciled artists to the Getty's ULAN, and [have their unique ULAN IDs as a column](#), you can generate a new column of JSON-formatted data by using [Add column by fetching URLs](#) and entering the GREL expression `"http://vocab.getty.edu/" + value + ".json"`. For this service, the unique IDs are formatted “ulan/000000” and so the generated URLs look like “<http://vocab.getty.edu/ulan/000000.json>”.

Alternatively, you can insert the ID directly from the matched column's reconciliation variables, using a GREL expression like `"http://vocab.getty.edu/" + cell.recon.match.id + ".json"` instead.

Remember to set an appropriate throttle and to refer to the service documentation to ensure your compliance with their terms. See [the section about this operation](#) to learn more about the fetching process.

Keep all the suggestions made

To generate a list of each suggestion made, rather than only the best candidate, you can use a [GREL expression](#). Go to [Edit column](#) → [Add column based on this column](#). To create a list of all the possible matches, use something like

```
foreach(cell.recon.candidates,c,c.name).join(", ")
```

To get the unique identifiers of these matches instead, use

```
foreach(cell.recon.candidates,c,c.id).join(", ")
```

This information is stored as a string, without any attached reconciliation information.

Writing reconciliation expressions

OpenRefine supplies a number of variables related specifically to reconciled values. These can be used in GREL and Jython expressions. For example, some of the reconciliation variables are:

- `cell.recon.match.id` or `cell.recon.match.name` for matched values
- `cell.recon.best.name` or `cell.recon.best.id` for best-candidate values
- `cell.recon.candidates` for all listed candidates of each cell
- `cell.recon.judgment` (the values used in the “judgment” facet)
- `cell.recon.judgmentHistory` (the values used in the “judgment action timestamp” facet)
- `cell.recon.matched` (a “true” or “false” value)

You can find out more in the [reconciliation variables](#) section.



MAKE A COPY OF A RECONCILED COLUMN

To make a copy of a reconciled column and all its contents (the entire `recon` object for each cell) into a new column, just use the GREL expression `cell` only when using `Edit column` → `Add column based on this column`.

Exporting reconciled data

Once you have data that is reconciled to existing entities online, you may wish to export that data to a user-editable service such as Wikidata. See the section on [uploading your edits to Wikidata or other Wikibase instances](#) for more information, or the section on [exporting](#) to see other formats OpenRefine can produce.

You can share reconciled data in progress through a [project export or import](#), with some preparation. The importing user needs to have the appropriate reconciliation services installed on their OpenRefine instance (by going to [Start reconciling](#) and clicking on [Add Standard Service...](#)) in advance of opening the project, in order to use candidate and match links. Otherwise, the links will be broken and the user will need to add the reconciliation service and re-reconcile the columns in question. [Wikidata](#) reconciliation data can be shared more easily as the service comes bundled with OpenRefine.

Overview of Wikibase support

[Wikibase](#) is free software (a set of MediaWiki extensions) used by many organizations around the world to store and publish Linked Open Data. Wikibase is the software behind [Wikidata](#), a free, multilingual collaborative knowledge base and a sister project of Wikipedia. Wikidata offers structured data about the world and can be edited by anyone. Wikibase also powers [structured data](#) on [Wikimedia Commons](#), the media repository of Wikipedia.

OpenRefine provides powerful ways to both pull data from a Wikibase (including Wikidata and Wikimedia Commons) and to add data to it.

OpenRefine's Wikibase integration is provided by an extension which is available by default in OpenRefine. In this page, we present the functionalities for [Wikidata](#) and [Wikimedia Commons](#), but [any Wikibase instance can be connected to OpenRefine](#) to obtain a similar integration.

Editing Wikidata with OpenRefine

As a user-maintained data source, Wikidata can be edited by anyone. OpenRefine makes it simple to upload information in bulk. You simply need to get your information into the correct format, and ensure that it is new (not redundant to information already on Wikidata) and does not conflict with existing Wikidata information.

You do not need a Wikidata account to reconcile your local OpenRefine project to Wikidata, but to upload your cleaned dataset to Wikidata, you will need an [autoconfirmed](#) account, and you must [authorize OpenRefine with that account](#).

Wikidata is built by creating entities (such as people, organizations, or places, identified with unique numbers starting with Q), defining properties (unique numbers starting with

P), and using properties to define relationships between entities (a Q has a property P, with a value of another Q).

For example, you may wish to create entities for local authors and the books they've set in your community. Each writer will be an entity with the occupation [author \(Q482980\)](#), each book will be an entity with the property "instance of" ([P31](#)) linking it to a class such as [literary work \(Q7725634\)](#), and books will be related to authors through a property [author \(P50\)](#). Books can have places where they are set, with the property [narrative location \(P840\)](#).

To do this with OpenRefine, you'll need a column of publication titles that you have reconciled (and create new items where needed); each publication will have one or more locations in a "setting" column, which is also reconciled to municipalities or regions where they exist (and create new items where needed). Then you can add those new relationships, and create new entities for authors, books, and places where needed. You do not need columns for properties; those are defined later, in the creation of your [schema](#).

There is a list of [tutorials and walkthroughs on Wikidata](#) that will allow you to see the full process. You can save your schemas and drafts in OpenRefine, and your progress stays in draft until you are ready to upload it to Wikidata.

Batches of edits to Wikidata that are created with OpenRefine can be undone. You can test out the uploading process by reconciling to several "sandbox" entities created specifically for drafting edits and learning about Wikidata:

- <https://www.wikidata.org/wiki/Q4115189>
- <https://www.wikidata.org/wiki/Q13406268>
- <https://www.wikidata.org/wiki/Q15397819>
- <https://www.wikidata.org/wiki/Q64768399>

If you upload edits that are redundant (that is, all the statements you want to make have

already been made), nothing will happen. If you upload edits that conflict with existing information (such as a different birthdate than one already in Wikidata), it will be added as a second statement. OpenRefine produces no warnings as to whether your data replicates or conflicts with existing Wikidata elements.

You can use OpenRefine's reconciliation preview to look at the target Wikidata elements and see what information they already have, and whether the elements' histories have had similar edits reverted in the past.

Wikidata schema

A [schema](#) is a plan for how to structure information in a database. In OpenRefine, the schema operates as a template for how Wikidata edits should be applied: how to translate your tabular data into statements. With a schema, you can:

- preview the Wikidata edits and inspect them manually;
- analyze and fix any issues highlighted by OpenRefine;
- upload your changes to Wikidata by logging in with your own account;
- export the changes to the QuickStatements v1 format.

For example, if your dataset has columns for authors, publication titles, and publication years, your schema can be conceptualized as: [publication title] has the author [author], and was published in [publication year]. To establish these facts, you need to establish one or more columns as “items,” for which you will make “statements” that relate them to other columns.

You can export any schema you create, and import an existing schema for use with a new dataset. This can help you work in batches on a large amount of data while minimizing redundant labor.

Once you select [Edit Wikidata schema](#) under the [Extensions](#) dropdown menu, your project interface will change. You'll see new tabs added to the right of "X rows/records" in the grid

header: “Schema,” “Issues,” and “Preview.” You can now switch between the tabular grid format of your dataset and the screens that allow you to prepare data for uploading.

OpenRefine presents you with an easy visual way to map out the relationships in your dataset. Each of the columns of your project will appear at the top of the screen, and you can simply drag and drop them into the appropriate slots. To get start, select one column as an item.

The screenshot shows the OpenRefine interface for mapping a dataset to Wikidata. At the top, there are tabs for "Schema *", "Issues", and "Preview". Below the tabs, it says "55 records". On the right, there's a "Wikidata" extension dropdown. The main area shows a list of columns: "Author", "Title", "Publication year", "Literary genre", "Country", and "Reference".

Under each column, there's a "Terms" section which is currently empty ("no labels, descriptions or aliases added"). There are buttons to "+ add term" and "remove".

Below the terms section is a "Statements" section. It lists the Wikidata terms corresponding to the columns:

- Column "Author" maps to "Author".
 - 1 reference:
 - copy
 - reference URL
 - Referen...
 - Buttons: "+ add", "+ add qualifier", "+ add reference", "+ add value", "+ add statement".
- Column "Publication year" maps to "Publication date".
 - 1 reference:
 - copy
 - reference URL
 - Referen...
 - Buttons: "+ add", "+ add qualifier", "+ add reference", "+ add value".
- Column "Literary genre" maps to "Literary genre".
 - 1 reference:
 - copy
 - reference URL
 - Referen...
 - Buttons: "+ add", "+ add qualifier", "+ add reference", "+ add value".

You may wish to refer to [this Wikidata tutorial on how OpenRefine handles Wikidata schema](#). For details about how each data type is handled in the Wikibase schema, see

Schema alignment.

Editing terms with your schema

With OpenRefine, you can edit the terms (labels, aliases, descriptions) of Wikidata entities as well as establish relationships between entities. For example, you may wish to upload pseudonyms, pen names, maiden names, or married names for authors.

Anne Rice (Q184785)

American writer				 edit
Anne O'Brien A. N. Roquelaure Anne Rampling Howard Allen O'Brien Howard Allen Frances O'Brien				
▼ In more languages				
Configure				
Language	Label	Description	Also known as	
English	Anne Rice	American writer	Anne O'Brien A. N. Roquelaure Anne Rampling Howard Allen O'Brien Howard Allen Frances O'Brien	
French	Anne Rice	écrivain américaine	Howard Allen O'Brien	
Italian	Anne Rice	scrittrice statunitense	Anne Rampling	
German	Anne Rice	US-amerikanische Schriftstellerin	Anne O'Brien Howard Allen O'Brien Anne Rampling Anne O'Brien A. N. Roquelaure Howard Allen O'Brien	

You can do so by putting the preferred names in one column of your dataset and alternative names in another column. In the schema interface, add an item for the preferred values, then click “Add term” on the right-hand side of the screen. Select “Alias” from the dropdown, enter in “English” in the language field, and drop your alternative names column into the space. For this example, you should also consider adding those alternative names to the authors' entries using the property [pseudonym \(P742\)](#). The “description” and “label” terms can only contain one value, so there is an option to override existing values if needed. Aliases can be potentially infinite.

Author

Terms

Alias	en	Pseudonyms	remove
Label	lang	<input type="text"/>	<input type="checkbox"/> override if present remove + add term

Statements
no statements added
+ add statement
+ add item

Terms must always have an associated language. You can select the term's language by typing in the "lang" field, which will auto-complete for you. You cannot edit multiple languages at once, unless you supply a suitable column instead. For example, suppose you had translated publication titles, with data in the following format:

English title	Translated title	Translation language
Possession	Besessen	German
	Обладать	Russian
Disgrace	Disgrâce	French
	Vergogna	Italian
Wolf Hall	En la corte del lobo	Spanish
	❖❖❖❖❖❖❖	Japanese

You could upload the "Translated titles" to "Label" with the language specified by "Translation language." You may wish to fetch the two-letter language code and use that instead for better language matches.

The screenshot shows the OpenRefine interface with the 'English title' tab selected. The 'Translated title' field is highlighted with a green border. The 'Terms' section contains fields for 'Label' and 'Alias', each with a dropdown menu and a trash can icon. There are also two checkboxes: 'override if present' for each field. Below the terms is a 'Statements' section with the message 'no statements added'.

Unsupported field types

With OpenRefine, it is not yet possible to edit:

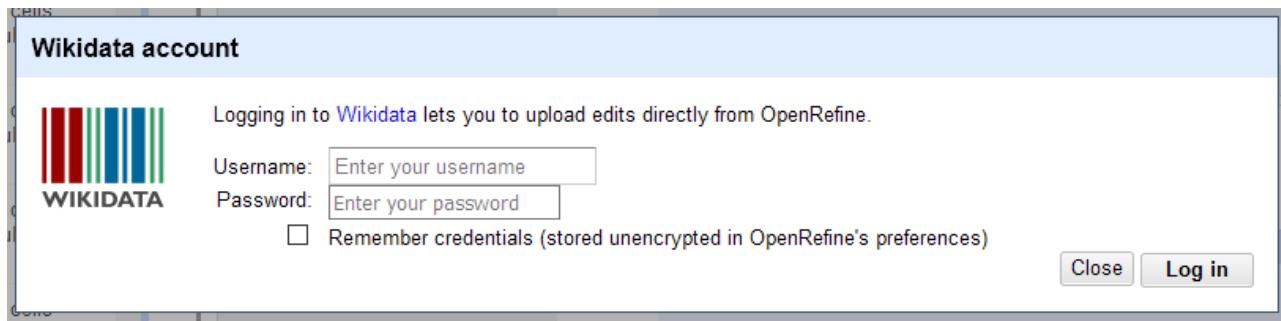
- sitelinks (links to Wikipedia or other Wikimedia projects, in the case of Wikidata);
- any field on Wikibase properties;
- lexemes, forms or senses.

Manage Wikidata account

To edit Wikidata directly from OpenRefine, you must log in with a Wikidata account.

OpenRefine can only upload edits with Wikidata user accounts that are “[autoconfirmed](#)” - at this time, that means accounts that have more than 50 edits and have existed for longer than four days.

Use the [Extensions](#) menu to select [Manage Wikidata account](#) and you will be presented with the following window:



For security reasons, you should not use your main account authorization with OpenRefine. Wikidata allows you to set special passwords to access your account through software. You can find [this setting for your account here](#) once logged in. Creating bot access will prompt you for a unique name. You should then enable the following required settings:

- High-volume editing
- Edit existing pages
- Create, edit, and move pages

It will then generate a username (in the form of "yourwikidatausername@yourbotname") and password for you to use with OpenRefine.

If your account or your bot is not properly authorized, OpenRefine will not display a warning or error when you try to upload your edits.

You can store your unencrypted username and password in OpenRefine, saved locally to your computer and available for future use. For security reasons, you may wish to leave this box unchecked. You can also save your OpenRefine-specific bot password in your browser or with a password management tool.

Import and export schema

You can save time on repetitive processes by defining a schema on one project, then exporting it and importing for use on new datasets in the future. Or you and your

colleagues can share a schema with each other to coordinate your work.

You can export a schema from a project using [Export](#) → [Wikidata schema](#), or by using [Extensions](#) → [Export schema](#). OpenRefine will generate a JSON file for you to save and share. You may experience issues with pop-up windows in your browser: consider allowing pop-ups from the OpenRefine URL (127.0.0.1) from now on.

You can import a schema using [Extensions](#) → [Import schema](#). You can upload a JSON file, or paste JSON statements directly into a field in the window. An imported schema will look for columns with the same names, and you will see an error message if your project doesn't contain matching columns.

Upload edits to Wikidata

There are two menu options in OpenRefine for applying your edits to Wikidata, and the details of the differences between the two can be found in the [Uploading page](#). Under [Export](#) you will see [Wikidata edits...](#) and under [Extensions](#) you will see [Upload edits to Wikidata](#). Both will bring up the same window for you to [log in with a Wikidata account](#).

Once you are authorized, you will see a window with any outstanding issues. You can ignore these issues, but we recommend you resolve them.

If you are ready to upload your edits, you can provide an “Edit summary” - a short message describing the batch of edits you are making. It can be helpful to leave notes for yourself, such as “batch 1: authors A-G” or other indicators of your workflow progress. OpenRefine will show the progress of the upload as it is happening, but does not show a confirmation window.

If your edits have been successful, you will see them listed on [your Wikidata user contributions page](#), and on the [Edit groups page](#). All edits can be undone from this second interface.

QuickStatements export

Your OpenRefine data can be exported in a format recognized by [QuickStatements](#), a tool that creates Wikidata edits using text commands. OpenRefine generates “version 1” QuickStatements commands.

There are advantages to using QuickStatements rather than uploading your edits directly to Wikidata, including the way QuickStatements resolves duplicates and redundancies. You can learn more on QuickStatements' [Help page](#), and on OpenRefine's [Uploading page](#).

In order to use QuickStatements, you must authorize it with a Wikidata account that is [autoconfirmed](#) (it may appear as “MediaWiki” when you authorize).

Follow the [steps listed on this page](#). To prepare your OpenRefine data into QuickStatements, select [Export](#) → [QuickStatements file](#), or [Extensions](#) → [Export to QuickStatements](#). Exporting your schema from OpenRefine will generate a text file called [statements.txt](#) by default. Paste the contents of the text file into a new QuickStatements batch using version 1. You can find [version 1 of the tool \(no longer maintained\) here](#). The text commands will be processed into Wikidata edits and previewed for you to review before submitting.

Issue detection

This section is an overview of the [Quality assurance page](#).

OpenRefine will analyze your schema and make suggestions. It does not check for conflicts in your proposed edits, or tell you about redundancies.

One of the most common suggestions is to attach [a reference to your edits](#) - a citation for where the information can be found. This can be a book or newspaper citation, a URL to

an online page, a reference to a physical source in an archival or special collection, or another source. If the source is itself an item on Wikidata, use the relationship [stated in](#) ([P248](#)); otherwise, use [reference URL](#) ([P854](#)) to identify an external source.

Editing Wikimedia Commons with OpenRefine



OPENREFINE VERSION 3.6 OR NEWER NEEDED!

Wikimedia Commons editing is possible with OpenRefine version 3.6 or newer. It is **not** supported in earlier versions.

More detailed guides and howtos for Wikimedia Commons editing can be found at OpenRefine's info page there: <https://commons.wikimedia.org/wiki/Commons:OpenRefine>

[Wikimedia Commons](#) is the shared media repository of the Wikimedia ecosystem; it is a sister project of Wikipedia. Wikimedia Commons contains (as of mid 2022) more than 85 million freely licensed media files (images, videos, scanned books, 3D files and more) that are used as illustrations on Wikipedia and that can be freely re-used by anyone.

Like other Wikimedia projects, Wikimedia Commons is a user-maintained resource which can be edited by anyone. As of version 3.6, OpenRefine can be used to bulk add structured, multilingual, machine-readable (linked) data to files on Wikimedia Commons.



STRUCTURED DATA ON WIKIMEDIA COMMONS

Structured data on Commons is multilingual information about a media file that can be understood by humans, with enough consistency that it can also be uniformly

processed by machines. Files on Wikimedia Commons can be described with multilingual concepts from Wikidata, Wikimedia's knowledge base.

Wikimedia Commons [has an information portal about structured data](#), which contains (among others) [information for cultural institutions](#), and [guidelines on data modeling](#).

In order to be able to edit Wikimedia Commons files with OpenRefine, you need a [Wikimedia account](#). You can use the same account across all Wikimedia projects (including Wikipedia, Wikidata and Wikimedia Commons); if you have already created an account on one of these projects, you can use that for Wikimedia Commons editing in OpenRefine as well.

Add the Wikimedia Commons manifest to OpenRefine

In order to make edits to Wikimedia Commons possible, start by adding the Wikimedia Commons manifest to OpenRefine. This process is the same as [the generic one for connecting OpenRefine to any Wikibase instance](#). For Wikimedia Commons specifically,

the process is as follows.

In the Wikidata extension menu, choose **Select Wikibase instance...**. If you haven't added other Wikibases yet, you will only see Wikidata in the resulting dialog window. Click **Add Wikibase**. You will be prompted to paste either a manifest URL, or paste the JSON directly. Wikimedia Commons' manifest URL is:

```
https://raw.githubusercontent.com/OpenRefine/wikibase-manifests/master/wikimedia-commons-manifest.json
```

Add Wikibase manifest

The manifest should specify a reconciliation service linked to the Wikibase, the reconciliation service will be added to OpenRefine if not added yet.

Enter the Wikibase manifest's URL (recommended, this is helpful for keeping track of the latest manifest):

```
content.com/OpenRefine/wikibase-manifests/master/wikimedia-commons-manifest.json
```

Or paste the manifest JSON text directly (manifests for some public Wikibases can be found [here](#), you can also write one yourself according to this [tutorial](#)):

After adding this URL, you should now see Wikimedia Commons in your list of Wikibase instances. Click Wikimedia Commons to activate it. You can now close this dialog window by clicking the **Close** button.

An up-to-date list of Wikibase manifests, including Wikimedia Commons' most recent manifest, is available in [OpenRefine's Wikibase manifests repository](#).

Adding the Wikimedia Commons manifest in OpenRefine will also automatically add the [Wikimedia Commons reconciliation service](#).

Reconcile file names from Wikimedia Commons

Typically, an OpenRefine project for batch editing files on Wikimedia Commons will contain:

- A column with file names of files on Wikimedia Commons, reconciled through the Wikimedia Commons reconciliation service
- One or more columns with data that you want to add to these files, reconciled through the Wikidata reconciliation service

Usually, you will start editing Wikimedia Commons files in OpenRefine from a list of file names from Commons. You can obtain file names from one or more categories by, for instance, using the [PetScan tool](#) ([example of a PetScan query](#) to retrieve file names for one Commons category; change to Plain text format in the Output tab for a simple plain text list).

Make sure you have these file names in a column in an OpenRefine project. The file names can be in many different formats, and can also be bare/plain M-ids (MedialInfo identifiers, example: [M7620878](#)). Next, reconcile this column of file names against the Wikimedia Commons reconciliation service, which makes sure that OpenRefine recognizes these files and can edit them later. You start the reconciliation process by selecting [Reconcile](#) → [Start reconciling...](#) in the file column's menu. Then select the Wikimedia Commons reconciliation service and click the [Start reconciling...](#) button.

If you don't have the Wikimedia Commons reconciliation service installed in OpenRefine yet, click the button [Add standard service...](#) and paste

<https://commonsreconcile.toolforge.org/en/api> there. You can find more info and documentation about the Commons reconciliation service at <https://commonsreconcile.toolforge.org/>.

Extract Wikitext to process it further in OpenRefine

This step is optional, but may be very useful. Existing files on Wikimedia Commons are always described with so-called Wikitext, a plain-text description which contains (among other things) information about the file's creator, license, and one or more descriptive categories. Often, Wikitext contains information which is valuable to parse in OpenRefine and which can be converted to structured data (and reconciled with Wikidata) later.

For instance, the Wikitext of [this image](#) is:

```
== {{int:filedesc}} ==
{{Information
|Description={{en|1=Illustration from "The little folks of animal
land". Inscription below image: "The Bufkins Twins Were Swinging"}}
{{ru|1=Фотография [[:w:ru:Фрис, Гарри Виттер|Гарри Виттера Фриса]] из
книги «Маленький народец страны зверят» (The little folks of animal
land). Подпись под картинкой: «Близнецы Бафкинс качаются» («The
Bufkins Twins Were Swinging»)}}
|Source={{cite book |author=Harry Whittier Frees |year=1915
|title=The little folks of animal land |publisher=Lothrop, Lee &
Shepard co. |pages=25| url=https://books.google.com/
books?id=HcwTIRt2FvwC&pg=PA25}} Retrieved from Google Books.
|Author=[[w:Harry Whittier Frees|Harry Whittier Frees]]
|Date=1915
|Permission={{PD-US}}
|other_versions=
}}


[[Category:Photographs of kittens by Harry Whittier Frees]]
```

You can extract Wikitext (and structured data statements) from a list of file names. Select

[Edit column](#) → [Add columns from reconciled values...](#)

See [Add columns from reconciled values](#) for general information about this feature.

Prepare columns with structured data

Based on the Wikitext that you may have just extracted, or with data from other sources, you may have (or add) columns with additional structured data, such as the files'

- [creator](#)
- [source](#)
- what is [depicted](#) in the files
- [copyright status and license](#)

Some of these columns will need to be reconciled with Wikidata (*not* Wikimedia Commons!), since files on Wikimedia Commons are described with data from Wikidata.

19 matching rows (62 total)						Open...	Export	Help		
Show as:		rows	records	Show:	5 10 25 50 100 500 1000 rows	first	previous	1 of 1 page	next	last
All	filename	Wikitext	depicts1 (P180)	depicts2 (P180)	inception (P571)	copyright status (P6216)				
1.	File:Harry Whittier Frees -- Playtime.jpg Choose new match	<pre>== {{int:filedesc}} ==\n{{Information\n Description=[[en 1=Title: "Playtime". Two cats in human dress holding rope, which doll appears to be skipping.]]\n {{ru 1=Название: «Playtime» (Время игр). Две кошки держат веревку, через которую прыгает кукла.]]\n Source=[[LOC-Image cph.3b39359]]\n Author=[[w:Harry Whittier Frees]]\n Date={{other date ca 1914}}\n Permission=\n other_versions=\n}}\n== {{int:license-header}} ==\n{{PD-US}}\n[[Category:Photographs of kittens by Harry Whittier Frees]]\n[[Category:Cats with objects]]\n[[Category:Jump ropes]]\n[[Category:Cat acting as humans in art]]\n[[Category:Ball-jointed dolls]]</pre>	skipping rope Choose new match	doll Choose new match	1914	public domain Choose new match				
2.	File:Harry Whittier Frees - The Bufkins Twins Were Swinging.jpg Choose new match	<pre>== {{int:filedesc}} ==\n{{Information\n Description=[[en 1=Illustration from "The little folks of animal land". Inscription below image: "The Bufkins Twins Were Swinging"]]\n {{ru 1=Фотография [[w:ру:Фрис, Гарри Виттер Гарри Виттера Фриса]] из книги «Маленький народец стран зверят» (The little folks of animal land). Подпись под картинкой: «Близнецы Брафинс качаются» («The Bufkins Twins Were Swinging»))\n Source={{cite book author=Harry Whittier Frees year=1915 title=The little folks of animal land publisher=Lothrop, Lee & Shepard co. pages=25 url=https://books.google.com/books?id=HcwTIR2FvwC&pg=PA25}} Retrieved from Google Books.\n Author=[[w:Harry Whittier Frees Harry Whittier Frees]]\n Date=1915\n Permission={{PD-US}}\n other_versions=\n}}</pre>	swing Choose new match		1915	public domain Choose new match				

INFO

Data models for structured data about media files on Commons are explained and discussed at https://commons.wikimedia.org/wiki/Commons:Structured_data/Modeling.

Upload edits to Wikimedia Commons

Finally, you will build a schema to model the Wikimedia Commons edits that OpenRefine will perform for each row in your project. See [schema alignment](#) for general documentation about this feature.

As of OpenRefine 3.6, you will see Wikimedia Commons-specific fields in the schema editor, provided that you have installed and activated the Wikimedia Commons manifest [as described above](#). One such Commons-specific 'field' is the [multilingual file caption](#), for which [best practices are documented on Wikimedia Commons](#).

OpenRefine Harry Whittier Frees kittens [Permalink](#)

19 matching rows (62 total) Schema * Issues Preview Extensions Wikidata *

You are currently working against [Wikimedia Commons](#). You should have your data reconciled to reconciliation services linked to Wikimedia Commons first. The schema below specifies how your tabular data will be transformed into Wikimedia Commons edits. You can drag and drop the column names below in most input boxes: for each row, edits will be generated with the values in these columns.

[Save schema](#) [Discard changes](#)

filename	caption_en	Wikitext	depicts1 (P180)	depicts2 (P180)	inception (P571)	copyright status (P6216)
en	caption_en					
<input checked="" type="checkbox"/> override if present						
Statements						
depicts	<input type="text" value="kitten"/> <div style="float: right;"> remove + add caption remove configure </div>					
▶ 0 references						
<input type="text" value="depicts1 (P180)"/> <div style="float: right;"> remove + add reference remove configure </div>						
▶ 0 references						
<input type="text" value="depicts2 (P180)"/> <div style="float: right;"> remove + add reference remove configure </div>						
▶ 0 references						
creator	<input type="text" value="Harry Whittier Frees"/> <div style="float: right;"> remove + add value remove configure </div>					
<input type="text" value="object has"/> <input type="text" value="photographer"/> <div style="float: right;"> remove + add qualifier remove </div>						
▶ 0 references						
inception	<input type="text" value="inception (P571)"/> <div style="float: right;"> remove + add reference remove configure </div>					
▶ 0 references						
copyright status	<input type="text" value="copyright status (P6216)"/> <div style="float: right;"> remove + add value remove configure </div>					
▶ 0 references						

You can now drag and drop, and/or enter the desired terms and statements in the [schema](#), preview your edits, log in to Wikimedia Commons and [upload your edits](#) in the same way as for Wikidata or another Wikibase.

Revert mistakes with the EditGroups tool

When checking [your user contributions](#), you will see your recent Wikimedia Commons edits done with OpenRefine. Each OpenRefine edit displays a (*details*) hyperlink after the edit summary, which links to the edit batch in the [EditGroups](#) tool.

- 15:38, 30 May 2022 (diff | hist) . . (+2,613) . . File:Veronica stricta stricta 116146039.jpg (Changed label, description and/or aliases in en, and other parts: *depicted taxon + source of file (iNaturalist)* ([details](#))) (**current**) [rollback: 1 edit] ([Tag: openrefine-3.6](#))
- 15:38, 30 May 2022 (diff | hist) . . (+5,474) . . File:Pseudois nayaur 15896711.jpg (Changed label, description and/or aliases in en, and other parts: *depicted taxon + source of file (iNaturalist)* ([details](#))) (**current**) [rollback: 1 edit] ([Tag: openrefine-3.6](#))

In EditGroups, entire batches can be easily undone, in case some mistakes have been made.

All Wikimedia Commons batches with OpenRefine are listed at <https://editgroups-commons.toolforge.org/?tool=OR>.

Connecting OpenRefine to a Wikibase instance

This page explains how to connect OpenRefine to any Wikibase instance. If you just want to work with [Wikidata](#), you can ignore this page as Wikidata is configured out of the box in OpenRefine.

For Wikibase end users

All you need to configure OpenRefine to work with a Wikibase instance is a *manifest* for that instance, which provides some metadata and links required for the integration to work.

We offer some off-the-shelf manifests for some public Wikibase instances in the [wikibase-manifests](#) repository. But the administrators of your Wikibase instance should provide one that is potentially more up to date, so it makes sense to request it to them first.

For Wikibase administrators

To let your users contribute to your Wikibase instance with OpenRefine, you will need to write a manifest as described above. There is currently no canonical location where this manifest should be hosted - just make sure can be found easily by your users. This section explains the format of the manifest.

Requirements

To work with OpenRefine, your Wikibase instance needs an associated reconciliation service for each editable entity type:

- To enable editing items (entities with an identifier starting with Q), you can deploy a [Python wrapper](#) for this. It exposes a reconciliation service for items, built on top of Wikibase's own API and its Query Service. Note that this service requires the [UniversalLanguageSelector extension](#) should be installed.
- To enable editing media files (if your Wikibase instance accepts file uploads), you can use [another Python wrapper](#) which exposes a reconciliation service for media files.
- Editing properties or other entity types is not supported yet.

We are aware that deploying those additional web services can be difficult for some Wikibase users, and we think those web services should be replaced by a MediaWiki extension which exposes the reconciliation endpoints from MediaWiki itself. We are not aware of anyone planning to work on this, though.

The format of the manifest

The manifest is a JSON object describing all the configuration details necessary for OpenRefine to integrate with your Wikibase instance. As an example, here is the manifest of Wikimedia Commons:

```
{  
  "version": "2.0",  
  "mediawiki": {  
    "name": "Wikimedia Commons",  
    "root": "https://commons.wikimedia.org/wiki/",  
    "main_page": "https://commons.wikimedia.org/wiki/Main_Page",  
    "api": "https://commons.wikimedia.org/w/api.php"  
  },  
  "wikibase": {  
    "site_iri": "https://commons.wikimedia.org/entity/",  
    "maxlag": 5,  
    "max_edits_per_minute": 60,  
    "tag": "openrefine-${version}",  
    "properties": {  
      "instance_of": "P31",  
    }  
  }  
}
```

In general, there are several parts of the manifest: version, mediawiki, wikibase, oauth, entity_types and editgroups.

version

The version should in the format "2.x". The minor version should be increased when you update the manifest in a backward-compatible manner. The major version should be "2" if the manifest is in the format specified by [wikibase-manifest-schema-v2.json](#).

mediawiki

This part contains some basic information of the Wikibase.

name

The name of the Wikibase, should be unique for different Wikibase instances.

root

The root of the Wikibase. Typically in the form "<https://foo.bar/wiki/>". The trailing slash cannot be omitted.

main_page

The main page of the Wikibase. Typically in the form "https://foo.bar/wiki/Main_Page".

api

The MediaWiki API endpoint of the Wikibase. Typically in the form "<https://foo.bar/w/api.php>".

wikibase

This part contains configurations of the Wikibase extension.

site_iri

The IRI of the Wikibase, in the form '<http://foo.bar/entity/>'. This should match the IRI prefixes used in RDF serialization. Be careful about using "http" or "https", because any variation will break comparisons at various places. The trailing slash cannot be omitted.

maxlag

Maxlag is a parameter that controls how aggressive a mass-editing tool should be when uploading edits to a Wikibase instance. See https://www.mediawiki.org/wiki/Manual:Maxlag_parameter for more details. The value should be adapted according to the actual traffic of the Wikibase.

tag

Specifies a **tag** which should be applied to all edits made via the tool. The `${version}` variable will be replaced by the "major.minor" OpenRefine version before making edits.

max_edits_per_minute

Determines the editing speed expressed as the maximum number of edits to perform per minute, as an integer. The editing can still be slower than this rate if the performance of the Wikibase instance degrades. If set to 0, this will disable this cap.

properties

Some special properties of the Wikibase.

instance_of

The ID of the property "instance of".

subclass_of

The ID of the property "subclass of".

constraints

Not required. Should be configured if the Wikibase has the [WikibaseQualityConstraints extension](#) installed. Configurations of constraints consists of IDs of constraints related properties and items. For Wikidata, these IDs are retrieved from [extension.json](#). To configure this for another Wikibase instance, you should contact an admin of the Wikibase instance to get the content of [extension.json](#).

oauth

Not required. Should be configured if the Wikibase has the [OAuth extension](#) installed.

registration_page

The page to register an OAuth consumer of the Wikibase. Typically in the form "<https://foo.bar/wiki/Special:OAuthConsumerRegistration/propose>".

entity_types

The Wikibase instance can support several entity types (such as [item](#), [property](#) or [lexeme](#)), and this section stores parameters which are specific to those entity types.

The Wikibase instance must have at least a reconciliation service endpoint linked to it.

reconciliation_endpoint

The default reconciliation service endpoint for entities of this type. The endpoint must contain the [\\${lang}](#) variable such as [https://wikidata.reconcil.link/\\${lang}/api](https://wikidata.reconcil.link/${lang}/api), since the reconciliation service is expected to work for different languages. For the [item](#) entity type, you can get such a reconciliation service with [openrefine-wikibase](#). For the [mediainfo](#) entity type, you can use the [commons-recon-service](#) which can be configured to run for other Wikibase instances.

This parameter is optional: you do not need to run a reconciliation for all entity types available in your Wikibase instance. However, it is a prerequisite for being able to do edits to those entity types via OpenRefine.

site_iri

The base IRI for the entities of this type. This property is required. By default, this is expected to be the same as the site IRI for the Wikibase instance (see above), but if entities of this type are federated from another instance, then this should be set to the site IRI of that Wikibase instance.

mediawiki_api

The URL of the MediaWiki API to use with entities of this type. If not provided, it is expected to be the same as the MediaWiki API endpoint for this instance, but if entities of this type are federated from another instance, then this should be set to the MediaWiki API endpoint of that Wikibase instance.

hide_structured_fields_in_mediainfo

Not required. Set this flag to true if your Wikibase instance supports file uploads (in which case it should have a [mediainfo](#) section in the [entity_types](#) object above), but it does not support adding captions and statements directly on the files themselves (unlike Wikimedia Commons).

editgroups

Not required. Should be configured if the Wikibase instance has [EditGroups](#) service(s).

url_schema

The URL schema used in edits summary. This is used for EditGroups to extract the batch id from a batch of edits and for linking to the EditGroups page of the batch. The URL schema must contains the variable [\\${batch_id}](#), such as [\(\[\[:toollabs:editgroups/b/OR/\\${batch_id}|details\]\]\)](#) for Wikidata.

Check the format of the manifest

As mentioned above, the manifest should be in the format specified by [wikibase-manifest-schema-v2.json](#). You can check the format by adding the manifest directly to OpenRefine, and OpenRefine will complain if there is anything wrong with the format.

Test 1 Permalink				Open...	Export ▾	Help	
3 rows	Schema	Issues 1	Preview	Extensions: Wikidata ▾			
Show as:	rows records	Show:	5 10 25 50 rows	« first	< previous	1 of 1 page	next > last »
All	name	description	official website				
1.	OpenRefine Choose new match	data wrangling software	http://openrefine.org/				
2.	Wikibase Choose new match	collection of software (applications and libraries) for creating, managing and sharing structured data	https://wikiba.se/				
3.	Wikidata-Toolkit Choose new match	Java library to interact with Wikibase	https://www.mediawiki.org/wiki/Wikidata_Toolkit				

Migrate from the version 1 to the version 2 of the manifest format

If you have created a manifest for your Wikibase instance before OpenRefine 3.6, then you have used the version 1 format for manifests. This format was generalized (into version 2) in OpenRefine 3.6 to allow for editing different entity types. If you are interested in letting OpenRefine users edit not just items on your Wikibase instance, but also other types of entities (such as media files), then you should migrate your manifest from version 1 to 2.

To do so, you need to:

- Change the `version` field of your manifest to 2.0;
- Introduce the new `entity_types` field, into which the URL of your existing reconciliation service should go (inside the `item` subsection). See the documentation above for more details about the expected values of such fields;
- Deploy [the additional reconciliation service for media files](#) and reference it in the `entity_types` section of the manifest, following the documentation above.
- If your Wikibase instance supports file uploads, but does not use structured data on those files, add the `hide_structured_fields_in_mediainfo` field to your manifest, as documented above.

After you have made those changes to your manifest, OpenRefine users will need to add it again to their list of Wikibase instances for the changes to take effect.

Reconciling with Wikibase

The Wikidata [reconciliation service](#) for OpenRefine supports:

- A large number of potential types to reconcile against
- Previewing and viewing entities
- Suggesting entities, types, and properties
- Augmenting your project with more information pulled from Wikidata.

You can find documentation and further resources on the reconciliation API [here](#).

For the most part, Wikidata reconciliation behaves the same way other reconciliation services do, but there are a few processes and features specific to Wikidata.

Language settings

You can install a version of the Wikidata reconciliation service that uses your language. First, you need the language code: this is the [two-letter code found on this list](#), or in the domain name of the desired Wikipedia/Wikidata (for instance, “fr” if your Wikipedia is <https://fr.wikipedia.org/wiki/>).

Then, open the reconciliation window (under [Reconcile](#) → [Start reconciling...](#)) and click [Add Standard Service](#). The URL to enter is <https://wikidata.reconcil.link/fr/api>, where “fr” is your desired language code.

When reconciling using this interface, items and properties will be displayed in your chosen language if the label is available. The matching score of the reconciliation is not influenced by your choice of language for the service: items are matched by considering all labels and returning the best possible match. The language of your dataset is also

irrelevant to your choice of language for the reconciliation service; it simply determines which language labels to return based on the entity chosen.

Restricting matches by type

In Wikidata, types are items themselves. For instance, the [university of Ljubljana \(Q1377\)](#) has the type [public university \(Q875538\)](#), using the [instance of \(P31\)](#) property. Types can be subclasses of other types, using the [subclass of \(P279\)](#) property. For instance, [public university \(Q875538\)](#) is a subclass of [university \(Q3918\)](#). You can visualize these structures with the [Wikidata Graph Builder](#).

When you select or enter a type for reconciliation, OpenRefine will include that type and all of its subtypes. For instance, if you select [university \(Q3918\)](#), then [university of Ljubljana \(Q1377\)](#) will be a possible match, though that item isn't directly linked to Q3918 - because it is directly linked to Q875538, the subclass of Q3918.

Some items and types may not yet be set as an instance or subclass of anything (because Wikidata is crowdsourced). If you restrict reconciliation to a type, items without the chosen type will not appear in the results, except as a fallback, and will have a lower score.

Reconciling via unique identifiers

You can supply a column of unique identifiers (in the form "Q###" for entities) directly to Wikidata in order to pull more data, but [these strings will not be “reconciled” against the external dataset](#). Apply the operation [Reconcile](#) → [Use values as identifiers](#) on your column of QIDs. All cells will appear as dark blue “confirmed” matches. Some of the “matches” may be errors, which you will need to hover over or click on to identify. You cannot use this to reconcile properties (in the form "P###").

If the identifier you submit is assigned to multiple Wikidata items (because Wikidata is crowdsourced), all of the items are returned as candidates, with none automatically matched.

Property paths, special properties, and subfields

Wikidata's hierarchical property structure can be called by using property paths (using |, /, and . symbols). Labels, aliases, descriptions, and sitelinks can also be accessed. You can also match values against subfields, such as latitude and longitude subfields of a geographical coordinate.

Labels, aliases, descriptions and sitelinks can be accessed as follows (L for label , D for description, A for aliases, S for sitelink):

Len for Label in English Dfi for Description in Finnish Apt for Alias in Portuguese Sdewiki for Sitelink in German Wikipedia page titles Scommonswiki for Commons sitelink

The lowercase letters are Wikimedia language codes which select which language the terms will be fetched. No language fall-back is performed when retrieving the values.

For information on how to do this, read the [documentation and further resources here](#).

Schema alignment

A Wikibase schema is a template of Wikibase edits that is applied to each row in the project. This page describes how each part of this template works, and how it generates edits depending on the contents of the table cells.

Items

An item in the schema represents a set of changes on a particular Wikibase item, generated by a single row. This item can contain changes in [terms](#) (labels, descriptions and aliases) or [statements](#).

It is possible to make edits on different items for each row of your table: just add multiple items in your schema. Each item has a subject, which can be either entered manually (when the item on which the edits should be made is the same for all rows), or any reconciled column can be dropped in this field. In this case, the edits will depend on the reconciliation status of each cell:

- If the cell is matched to an item, edits will be made on that item;
- If the cell is marked as corresponding to a new item, a new item will be created for it. See [New items](#) for more details about how this works;
- If the cell has reconciliation candidates but has not been matched to any of them, the edit will be skipped (even if there is only one candidate with a high reconciliation score);
- If the cell is not reconciled or blank, the edit will be skipped.

Do not worry about the ordering of items in the schema or the order of your rows, as OpenRefine will rearrange your edits to optimize their upload. If your project makes edits on the same item across multiple rows, these edits will be merged together and

performed in one edit. See [Uploading your changes](#) about that.

Terms

Terms are the language-specific strings that you find at the top of Wikibase items: labels, descriptions and aliases. OpenRefine lets you edit these terms via the Wikibase schema.

Languages

Each term belongs to a particular language. Wikibase supports [hundreds of languages](#), which are designated by language codes. For each term that you want to add to an item, you will need to specify the language for this term. There are two cases:

- Either the language is constant across your dataset: you know that all the names in a given column are spelled in the same language. In this case, type the name of the language in the input and select the language in the drop-down suggestion dialog. This will place the appropriate language code in the input.
- Or the language varies across your dataset. In this case, you need to provide a column of Wikimedia language codes that indicates the language for each term that you want to add. Just drag and drop this column to the language field. If there are any invalid language codes in this column, the corresponding terms will be ignored. OpenRefine will translate any deprecated language codes to their preferred values silently.

Labels

This is because Wikibase items can have at most one label per language, so you need to choose whether to override any existing label (default behaviour before 3.2) or only insert your label if there is no such label in the given language (default behaviour starting from 3.2). When the content of the cell providing the label is blank, nothing will be changed (so,

it is not possible to remove labels).

Descriptions

Descriptions work like labels: there is at most one description per language, and OpenRefine can override existing descriptions or leave them unchanged. It is not possible to remove descriptions either.

Aliases

Aliases are added to the list of existing aliases in the given language. When adding an alias in a language where no label has been added yet, the alias is automatically promoted to a label for this language. It is not possible to remove aliases or to override any existing aliases.

Statements

You can add statements in the schema: this will generate new statements on the corresponding items. These statements will be merged with any existing statements on the actual Wikibase items and [this merging process depends on the upload medium](#). It is forecast to give more control over the merging strategy in the near future.

Main values

The main value of a statement is a data value whose type depends on the property used for the statement. If the main value cannot be evaluated (for instance because one of the cells it depends on is empty), then the entire statement will be skipped.

Statements with "no value" or "some value" can be inserted by using the special keywords

`#NOVALUE#` and `#SOMEVALUE#`, used in place of the value (either directly in the schema or via a column). This is supported since OpenRefine 3.7.

See the [data values](#) section for more details about how to specify each type of data value and when they are skipped.

Qualifiers

Qualifiers can be added on each statement. When their values are skipped, only the qualifier will be discarded: the rest of the statement will still be added.

References

References can (and should) be added to back each statement. If values inside the reference are skipped, the corresponding part of the reference will be discarded but the reference will still be added (unless the reference becomes empty).

Editing mode

The editing mode of a statement determines how it contributes to the corresponding entity. OpenRefine offers three editing modes:

- **Add or merge**, which adds the statement or merges it with the first existing statement that matches it;
- **Add**, which only adds the statement if there are no matching statements on the entity. Otherwise, leave those statements untouched;
- **Delete**, which deletes all matching statements.

The way statements are matched is controlled by the matching strategy, which can be configured for each statement in the schema.

Matching strategy

The matching strategy determines how the candidate statements generated by the schema are compared to the existing statements on the entity. OpenRefine offers three merging strategies:

- **Property**, which compares statements by their main property only. This means that any two statements using the same main property will be considered equivalent. For instance, using this merging strategy in conjunction with the **Delete** editing mode will delete all statements with a particular main property on the target entity.
- **Property and value**, which compares statements by their main property and main value only. This is what QuickStatements does. In addition, it is possible (and enabled by default) to match statement values in a lax way, for instance to ignore differences in trailing whitespace or rounding of quantities.
- **Qualifiers**, which compare statements using their property, main value and qualifiers. It is possible to define a list of property identifiers which determines which qualifiers are discriminating. Other qualifiers will not be taken into account when comparing statements. By default, all qualifiers are taken into account. This matching strategy also supports lax value matching.

These matching strategies are not honoured when exporting to QuickStatements, as the QuickStatements formats do not make it possible to represent them.

Lax value matching

When lax value matching is enabled, the following values are considered equal for statement matching purposes:

- strings which differ by whitespace at the beginning or end (such as `Berlin` and `Berlin`);
- URLs which differ by trailing slash or `http` / `https` differences (such as

<http://wikiba.se> and <https://wikiba.se/>);

- quantities with the same unit, whose uncertainty domain overlap (such as [47±1](#) and [48±0.5](#));
- geographical coordinates whose uncertainty domain overlap (note that since the uncertainty of geographical coordinates is expressed in degrees, this does not guarantee a distance threshold below which the coordinates will match);
- monolingual text values whose values differ by leading or trailing whitespace;
- dates which differ in attributes which are rendered irrelevant by the lowest precision of both values to compare (such as [1976-01-01](#) and [1976](#)).

Ranks

All statements ranks are set to **Normal**. It is currently not possible to set a different rank.

Data values

Data values are the data that you can find as target of a statement (or qualifier, or part of a reference). Each property dictates a particular type of data value. In each case, OpenRefine uses a particular process to translate cell contents to a data value of the appropriate type. This section explains the process for all data types.

Items

Items are evaluated in the same way as the subjects of items in the schema. They can be input directly using the auto-suggest service provided, or any column reconciled against Wikibase can be used. Refer to [the first Items section](#) to see how they are evaluated.

Strings and external identifiers

Bare strings and external identifiers can be input directly as constants (if they do not change across rows) or using any column. If a reconciled column is used for a string value, it is the value of the cell that is going to be used, not the name of the reconciled item (which is what OpenRefine displays). Values are skipped when the column is blank or null.

Monolingual texts

Monolingual texts consist of two parts:

- the language: see [Languages](#) for their structure;
- the value of the text: see [the section above](#).

A monolingual text is skipped when any of its parts is skipped (that is, if the language or the text are invalid).

Dates

Dates are parsed from cell contents (or from any constant provided in the schema) and the precision of the date is inferred from its format. Here are the valid formats:

- `YYYYM`, such as `2001M` (millenium precision)
- `YYYYC`, such as `1901C` (century precision)
- `YYYYD`, such as `1981D` (decade precision)
- `YYYY`, such as `1984` (year precision)
- `YYYY-MM`, such as `2019-03` (month precision)

- `YYYY-MM-DD`, such as `1897-08-14` (day precision)

Any value that does not match any of these formats will be ignored. All dates are represented in UTC, Gregorian calendar.

In OpenRefine 3.3, the following new formats have been introduced:

- `TODAY` returns today's date with day precision. This will be evaluated when performing the edits (or exporting to QuickStatements);
- `YYYY-MM-DD_QID` can be used to specify a date in a particular calendar (such as the [proleptic Julian calendar \(Q1985786\)](#)).

In OpenRefine 3.5, the following new format has been introduced:

- `-234` represents the year 234 [BCE](#)

Quantities

Quantities consist of two parts: the amount and the unit.

- the amount is mandatory and must be a string, such as `18,229.1020`. The precision that is displayed will be respected (the same number of trailing zeros will be shown in Wikibase). By default, no upper and lower bounds will be set. To define these, one needs to use the engineering notation, such as `3.45E+3`, which will be interpreted as `3,450±5`. As usual, the amount can be provided as a constant or as a column variable. In the latter case, the values in the column must be strings.
- the unit is optional. It is an item, so it can be provided either with the auto-suggest dialog or as a reconciled column. It is important to note that if a reconciled column is used, any unreconciled cells will discard the entire quantity value. So a template for a quantity value is either always unit-less, or always has a unit.

Globe coordinates

Geographic coordinates are specified as strings with the following formats, where all components are floating point numbers in degrees:

- `latitude, longitude` for a default precision of ten micro degrees (for instance: `49.265278, 4.028611` can be used indicate the position of Reims, France).
- `latitude, longitude, precision` when specifying an explicit precision (for instance: `49.265278, 4.028611, 0.1` can be used indicate the position of Reims within a tenth of a degree).

All globe coordinates are on Earth ([Q2](#)).

If your coordinates are in a different format, such as `49° 15' 55" N, 4° 1' 43" E`, you will need to convert them to decimal format first.

Media on Commons

Media on Wikimedia Commons is treated like strings, whose values must exactly match filenames on Commons. These values are not checked during schema evaluations: if they are wrong, uploading the statements will fail.

Tabular data and Geoshapes must be prefixed with the `Data:` namespace. This is indicated by the placeholder in the field that appears when constructing the schema.

Properties

Properties are always constants: there is currently no way to reconcile a column against properties. They have to be selected with the auto-suggest dialog.

Other data types

URLs, mathematical expressions and other textual datatypes are supported and treated as strings. At the time of writing, all datatypes supported by Wikibase are supported by OpenRefine.

Creating new items

OpenRefine can create new items. This page explains how they are generated.

Words of caution

- The fact that OpenRefine does not propose any item when reconciling a cell does not mean that the item is not present in the Wikibase instance: it can be missed for all sorts of reasons. Please make sure that you are not creating any duplicates!
- Make sure that the items that you want to create are admissible in the Wikibase instance. For Wikidata, see the [notability guidelines](#);
- Deleting items generally requires special rights: if you want to revert an edit group that includes new items in Wikidata, you will need to ask an administrator to do it.

Workflow overview

Here is how you would typically create new items with OpenRefine:

- Reconcile a column;
- Mark some of its cells as new items. This will not create items yet. If you need to mark many rows as new items, use the **Reconcile → Actions → Create a new item for each cell** operation.
- Create a Wikibase schema as usual, using the column where your new items are marked;
- Perform the edits: the new items will be created on Wikidata at this point;

- The cells that you had marked as new items will now be reconciled to the newly-created items.

It is often useful (but not mandatory) to treat new items in isolation and use a dedicated schema for them. This helps you add many statements on the new items (including labels and descriptions) without risking to clutter existing items with redundant edits. Use a facet on the judgment status of the reconciled column to isolate new items and perform their edits separately. As always in OpenRefine, only the rows covered by your facets will be considered when uploading the edits to Wikidata: if a cell is reconciled to a new item but is excluded by the facet, no new item will be created for it.¹

Note that even if you know that all items in your column are new, you will still need to make a first reconciliation pass by selecting the Wikidata reconciliation service, and then setting all reconciliation statuses to "new". If you skip the first part, OpenRefine will not know that this column is reconciled against your Wikibase instance (it could be reconciled to other services) so it will not let you use it in place of an item in a Wikibase schema.

You can also perform the edits with QuickStatements - in this case, your OpenRefine project will not be updated with the newly created Qids.

Adding labels to new items

The text that is in a cell reconciled to "new" is not automatically used as label for the newly-created item. This is because OpenRefine has no way to guess in which language this label should be. When adding new items, you need to explicitly add a label in the schema. This label can use the reconciled column as source, but if you have other cells matched to existing items, be careful not to override the labels of these items (if it is not your intention).

OpenRefine will refuse to perform edits where new items are created without any labels (as this is considered a critical issue). Other issues will be raised if insufficient basic

information is added on the items (but these other warnings will not prevent you from performing the edits).

Marking multiple cells as identical items

If you mark individual cells as new items, one new item per cell will be created. Sometimes multiple rows refer to the same item. OpenRefine makes it possible to mark all the corresponding cells as the *same* new item. Two conditions have to be met:

- the reconciled cells must be in the same column (it is not possible to mark two cells in different columns as the same new item);
- the cells must contain the same initial text value.

If these two conditions are met, then isolate these cells with facets and go to **Reconcile** → **Actions** → **Create one item for similar cells**. This will mark the cells as new and referring to the same item.

Retrieving the Qids of the newly-created items

Once you have performed your edits with OpenRefine, any new cells covered by the facet will be updated with their new Qids. You can retrieve these Qids with the **Edit column** → **Add column based on this column** action and using the `cell.recon.match.id` expression. Note that you will no longer be able to isolate new items with a judgment facet at this stage (because the judgment will be updated to **matched**) so it can be worth marking these rows (for instance with a star or flag) before performing the edits.

Footnotes

1. The only exception to this rule is when marking multiple cells as identical items: in this case, if one of such cells are included in the facet, then all the others will be updated with the newly created Qid once the edits are made. ↵

Quality assurance for Wikibase uploads

This page explains how the Wikidata extension of OpenRefine analyzes edits before they are uploaded to the Wikibase instance. Most of these checks rely on the use of the [Wikibase Quality Constraints](#) extension and the configuration of the property and item identifiers in the [Wikibase manifest](#).

Overview

Changes are scrutinized before they are uploaded, but also before the current content of the corresponding items is retrieved and merged with the updates. This means that some constraint violations cannot be predicted by the software (for instance, adding a new statement that conflicts with an existing statement on the item). However, this makes it possible to run the checks quickly, even for relatively large batches of edits. Issues are therefore refreshed in real time while the user builds the schema.

As a consequence, not all constraint violations can be detected: the ones that are supported are listed in the [Constraint violations](#) section. Conversely, not all issues reported will be flagged as constraint violations on the Wikibase site: see [Generic issues](#) for these.

Reconciliation

You should always assess the quality of your reconciliation results first. OpenRefine has various tools for quality assurance of reconciliation results. For instance:

- you can analyze the string similarity between your original names and those of the reconciled items (for instance with `Reconcile` → `Facets` → `Best candidate's name edit distance`);
- you can compare the values in your table with those on the items (via a text facet defined by a custom expression);
- you can facet by type on the reconciled items (add a new column with the types and use a text facet ordered by counts to get a sense of the distribution of types in your reconciled items).

Constraint violations

Constraints are retrieved as defined on the properties, using ([P2302](#)).

The following constraints are supported:

- [format constraint \(Q21502404\)](#), checked on all values
- [inverse constraint \(Q21510855\)](#): OpenRefine assumes that the inverses of the candidate statements are not in Wikidata yet. If you know that the inverse statements are already in Wikidata, you can safely ignore this issue.
- [used for values only constraint \(Q21528958\)](#), [used as qualifier constraint \(Q21510863\)](#) and [used as reference constraint \(Q21528959\)](#)
- [allowed qualifiers constraint \(Q21510851\)](#)
- [required qualifier constraint \(Q21510856\)](#)
- [single-value constraint \(Q19474404\)](#): this will only trigger if you are adding more than one statement with the property on the same item, but will not detect any existing statement with this property.
- [distinct values constraint \(Q21502410\)](#): similarly, this only checks for conflicts inside your edit batch.

A comparison of the supported constraints with respect to other implementations is available [here](#).

Generic issues

OpenRefine also detects issues that are not flagged (yet) by constraint violations on Wikidata:

- Statements without references. This does not rely on [citation needed constraint \(Q54554025\)](#): all statements are expected to have references. (The idea is that when importing a dataset, every statement you add
- should link to this dataset - it does not hurt to do it even for generic properties such as [instance of \(P31\)](#).)
- Spurious whitespace and non-printable characters in strings (including labels, descriptions and aliases);
- Self-referential statements (statements which mention the item they belong to);
- New items created without any label;
- New items created without any description;
- New items created without any [instance of \(P31\)](#) or [subclass of \(P279\)](#) statement.

Uploading edits to Wikibase

This page explains how to upload your edits to the target Wikibase. It assumes you already have a created a Wikibase schema in your OpenRefine project.

Uploading with OpenRefine

- Click `Wikidata` → `Upload edits to Wikidata`.
- Log in with your personal account or your bot account depending on which account you want to use to make the edits. It is a good practice to use a [bot password](#).
- Supply a meaningful edit summary. This is especially important because OpenRefine condenses all your changes on the same item as one edit: if you are making multiple changes, the edit summary generated by Wikibase will not indicate clearly what sort of change you made. If you are making atomic changes, such as adding a single alias or statement, the automatic edit summaries will be more meaningful. If supported by your Wikibase instance, OpenRefine will append a link to the [EditGroups](#) tool, which lets you track and analyze your edit batch after upload.
- Click `Perform edits` and wait for the operation to complete. You can watch your edits being made by checking your wiki contributions or the [EditGroups](#) tool.

Because performing edits in OpenRefine counts as an operation, you can extract this operation and reapply it to other projects. If you do so, you should also include the operation that saves the schema (only the last one is required), and make sure that the column names in the schema match those of the OpenRefine project where you are applying the operation.

Uploading with QuickStatements

This requires that the Wikibase site has an associated [QuickStatements](#) tool.

- Click `Wikibase` → `Export to QuickStatements` and copy the contents of the file;
- Go to QuickStatements (for Wikidata it can be found at <https://quickstatements.toolforge.org/>) and login to authorize the tool to use your account;
- Click `Version 1 format`;
- Paste the generated changes in the text area;
- Perform the edits with `Run` or `Run in background`.

Notable differences between the two methods

Merging strategy for terms and statements

OpenRefine offers various merging strategies for terms and statements. QuickStatements only supports one non-configurable merging strategy. Therefore, the merging strategies specified by the user in the schema are ignored when exporting to QuickStatements, which can result in unintended changes.

New item creation

OpenRefine supports creating new items with arbitrary relations between them.

QuickStatements supports creating new items with the `CREATE` instruction, and subsequent instructions can use the `LAST` placeholder to use the Qid of the last created item. When generating QuickStatements instructions, OpenRefine reorders your edits so that this syntax can be used. In rare cases, such as when a statement links two newly-created items, it is impossible to use QuickStatements to perform the edit. In this case, no QuickStatements script will be generated.

Speed and number of edits

OpenRefine generally performs one edit per item touched by an edit batch and at most two in general (in the case where new items contain links between them). This was chosen to minimize server load, speed up the upload and keep item histories compact. The downside is that the edit summaries can be less meaningful - it is therefore important that users supply informative summaries when uploading their batches. OpenRefine asymptotically edits at the rate of 60 edits per minute (so, usually 60 items per minute). The first edits are made more quickly, which is convenient for small batches.

QuickStatements performs incremental edits (for instance, when adding a statement with a qualifier and a reference, it will make three edits). That generally means lower speed, but more explicit item histories.

Expressions

Overview

You can use expressions in multiple places in OpenRefine to extend data cleanup and transformation. Expressions are available with the following functions:

- **Facet :**
 - **Custom text facet...**
 - **Custom numeric facet...**
 - **Customized facets** (click “change” after they have been created to bring up an expressions window)
- **Edit cells :**
 - **Transform...**
 - **Split multi-valued cells...**
 - **Join multi-valued cells...**
- **Edit column :**
 - **Split**
 - **Join**
 - **Add column based on this column**
 - **Add column by fetching URLs**.

In the expressions editor window you have the opportunity to select a supported language. The default is [GREL \(General Refine Expression Language\)](#); OpenRefine also comes with

support for [Clojure](#) and [Jython](#). Extensions may offer support for more expressions languages.

These languages have some syntax differences but support many of the same [variables](#). For example, the GREL expression `value.split(" ")[1]` would be written in Jython as `return value.split(" ")[1]`.

This page is a general reference for available functions, variables, and syntax. For examples that use these expressions for common data tasks, look at the [Recipes section on the wiki](#).

Expressions

There are significant differences between OpenRefine's expressions and the spreadsheet formulas you may be used to using for data manipulation. OpenRefine does not store formulas in cells and display output dynamically: OpenRefine's transformations are one-time operations that can change column contents or generate new columns. These are applied using variables such as `value` or `cell` to perform the same modification to each cell in a column.

Take the following example:

ID	Friend	Age
1.	John Smith	28
2.	Jane Doe	33

Were you to apply a transformation to the “friend” column with the expression

```
value.split(" ")[1]
```

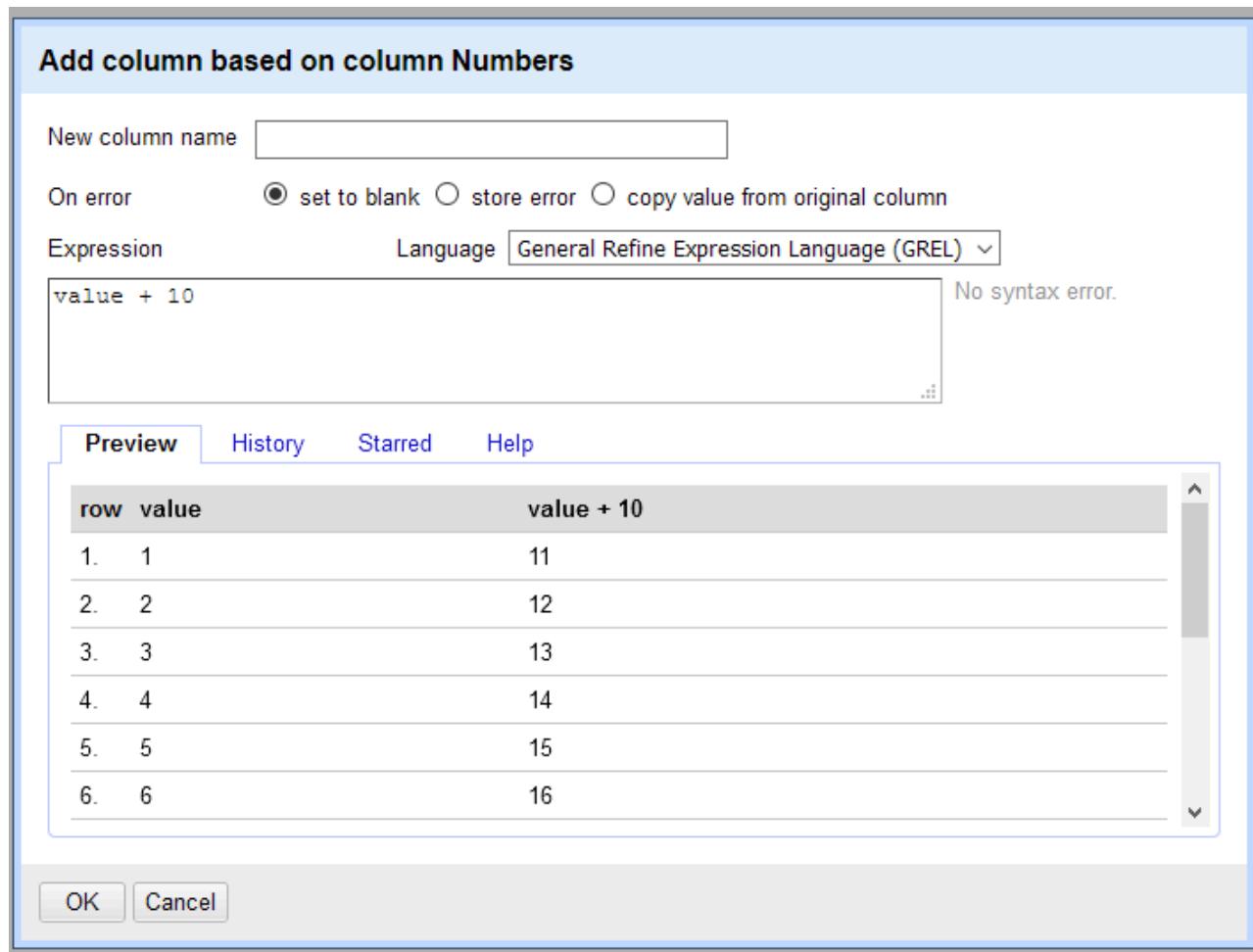
OpenRefine would work through each row, splitting the “friend” values based on a space character. The `value` for row 1 is “John Smith” so the output would be “Smith” (as “[1]” selects the second part of the created output); the `value` for row 2 is “Jane Doe” so the output would be “Doe”. Using variables, a single expression yields different results for different rows. The old information would be discarded; you couldn’t get “John” and “Jane” back unless you undid the operation in the [History](#) tab.

For another example, if you were to create a new column based on your data using the expression `row.starred`, it would generate a column of true and false values based on whether your rows were starred at that moment. If you were to then star more rows and unstar some rows, that data would not dynamically update - you would need to run the operation again to have current true/false values.

Note that an expression is typically based on one particular column in the data - the column whose drop-down menu is first selected. Many variables are created to stand for things about the cell in that “base column” of the current row on which the expression is evaluated. There are also variables about rows, which you can use to access cells in other columns.

The expressions editor

When you select a function that accepts expressions, you will see a window overlay the screen with what we call the expressions editor.



The expressions editor offers you a field for entering your formula and shows you a preview of its transformation on your first few rows of cells.

There is a dropdown menu from which you can choose an expression language. The default at first is GREL; if you begin working with another language, that selection will persist across OpenRefine. Jython and Clojure are also offered with the installation package, and you may be able to add more language support with third-party extensions and customizations.

There are also tabs for:

- [History](#), which shows you formulas you've recently used from across all your projects
- [Starred](#), which shows you formulas from your History that you've starred for reuse
- [Help](#), a quick reference to GREL functions.

Storing formulas you've used in the past can be helpful for repetitive tasks you're performing in batches.

You can also choose how formula errors are handled: replicate the original cell value, output an error message into the cell, or output a blank cell.

Regular expressions

OpenRefine offers several fields that support the use of regular expressions (regex), such as in a [Text filter](#) or a [Replace...](#) operation. GREL and other expressions can also use regular expression markup to extend their functionality.

If this is your first time working with regex, you may wish to read [this tutorial specific to the Java syntax that OpenRefine supports](#). We also recommend this [testing and learning tool](#).

GREL-supported regex

To write a regular expression inside a GREL expression, wrap it between a pair of forward slashes (/) much like the way you would in Javascript. For example, in

```
value.replace(/\s+/, " ")
```

the regular expression is `\s+`, and the syntax used in the expression wraps it with forward slashes (`/\s+/`). Though the regular expression syntax in OpenRefine follows that of Java (normally in Java, you would write regex as a string and escape it like "`\s+`"), a regular expression within a GREL expression is similar to Javascript.

Do not use slashes to wrap regular expressions outside of a GREL expression.

On the [GREL functions](#) page, functions that support regex will indicate that with a "p" for "pattern." The GREL functions that support regex are:

- [contains](#)
- [replace](#)
- [find](#)
- [match](#)
- [partition](#)
- [rpartition](#)
- [split](#)
- [smartSplit](#)

Jython-supported regex

You can also use [regex with Jython expressions](#), instead of GREL, for example with a [Custom Text Facet](#):

```
python import re g = re.search(ur"\u2014 (.*)",\s*BwV", value) return  
g.group(1)
```

Clojure-supported regex

[Clojure](#) uses the same regex engine as Java, and can be invoked with [re-find](#), [re-matches](#), etc. You can use the `#"pattern"` reader macro as described [in the Clojure documentation](#). For example, to get the nth element of a returned sequence, you can use the `nth` function:

```
clojure (nth (re-find #"\u2014 (.*)",\s*BwV" value) 1)
```

Variables

Most OpenRefine variables have attributes: aspects of the variables that can be called separately. We call these attributes “member fields” because they belong to certain variables. For example, you can query a record to find out how many rows it contains with `row.record.rowCount`: `rowCount` is a member field specific to the `record` variable, which is a member field of `row`. Member fields can be called using a dot separator, or with square brackets (`row["record"]`). The square bracket syntax is also used for variables that can call columns by name, for example, `cells["Postal Code"]`.

Variable	Meaning
<code>value</code>	The value of the cell in the current column of the current row (can be null)
<code>row</code>	The current row
<code>row.record</code>	One or more rows grouped together to form a record
<code>cells</code>	The cells of the current row, with fields that correspond to the column names (or <code>row.cells</code>)
<code>cell</code>	The cell in the current column of the current row, containing <code>value</code> and other attributes
<code>cell.recon</code>	The cell's reconciliation information returned from a reconciliation service or provider
<code>rowIndex</code>	The index value of the current row (the first row is 0)

Variable	Meaning
<code>columnName</code>	The name of the current cell's column, as a string

Row

The `row` variable itself is best used to access its member fields, which you can do using either a dot operator or square brackets: `row.index` or `row["index"]`.

Field	Meaning
<code>row.index</code>	The index value of the current row (the first row is 0)
<code>row.cells</code>	The cells of the row, returned as an array
<code>row.columnNames</code>	An array of the column names of the project. This will report all columns, even those with null cell values in that particular row. Call a column by number with <code>row.columnNames[3]</code>
<code>row.starred</code>	A boolean indicating if the row is starred
<code>row.flagged</code>	A boolean indicating if the row is flagged
<code>row.record</code>	The <code>record</code> object containing the current row

For array objects such as `row.columnNames` you can preview the array using the expressions window, and output it as a string using `toString(row.columnNames)` or with something like:

```
foreach(row.columnNames, v, v).join("; ")
```

Cells

The `cells` object is used to call information from the columns in your project. For example, `cells.Foo` returns a `cell` object representing the cell in the column named "Foo" of the current row. If the column name has spaces, use square brackets, e.g., `cells["Postal Code"]`. To get the corresponding column's value inside the `cells` variable, use `.value` at the end, for example, `cells["Postal Code"].value`. There is no `cells.value` - it can only be used with member fields.

Cell

A `cell` object contains all the data of a cell and is stored as a single object.

You can use `cell` on its own in the expressions editor to copy all the contents of a column to another column, including reconciliation information. Although the preview in the expressions editor will only show a small representation ("[object Cell]"), it will actually copy all the cell's data. Try this with `Edit Column` → `Add Column based on this column ...`.

Field	Meaning	Member fields
<code>cell</code>	An object containing the entire contents of the cell	<code>.value</code> , <code>.recon</code> , <code>.errorMessage</code>
<code>cell.value</code>	The value in the cell, which can be a string, a number, a boolean, null, or an error	
<code>cell.recon</code>	An object encapsulating reconciliation results	See the

Field	Meaning	Member fields
	for that cell	reconciliation section
<code>cell.errorMessage</code>	Returns the message of an <i>EvalError</i> instead of the error object itself (use value to return the error object)	.value

Reconciliation

Several of the fields here provide the data used in [reconciliation facets](#). You must type `cell.recon`; `recon` on its own will not work.

Field	Meaning	Member fields
<code>cell.recon.judgment</code>	A string: either "matched", "new", "none"	
<code>cell.recon.judgmentAction</code>	A string: either "single" or "similar" (or "unknown")	
<code>cell.recon.judgmentHistory</code>	A number, the epoch timestamp (in milliseconds) of	

Field	Meaning	Member fields
	your judgment	
<code>cell.recon.matched</code>	A boolean, true if judgment is “matched”	
<code>cell.recon.match</code>	The recon candidate that has been matched against this cell (or null)	.id, .name, .type
<code>cell.recon.best</code>	The highest scoring recon candidate from the reconciliation service (or null)	.id, .name, .type, .score
<code>cell.recon.features</code>	An array of reconciliation features to help you assess the accuracy of your matches	.typeMatch, .nameMatch, .nameLevenshtein, .nameWordDistance
<code>cell.recon.features.typeMatch</code>	A boolean, true if your chosen type is	

Field	Meaning	Member fields
	“matched” and false if not (or “(no type)” if unreconciled)	
<code>cell.recon.features.nameMatch</code>	A boolean, true if the cell and candidate strings are identical and false if not (or “(unreconciled)”)	
<code>cell.recon.features.nameLevenshtein</code>	A number representing the Levenshtein distance : larger if the difference is greater between value and candidate	
<code>cell.recon.features.nameWordDistance</code>	A number based on the word similarity	
<code>cell.recon.candidates</code>	An array of the top 3 candidates (default)	.id, .name, .type, .score

The `cell.recon.candidates` and `cell.recon.best` objects have a few deeper fields: `id`, `name`, `type`, and `score.type` is an array of type identifiers for a list of candidates, or a single string for the best candidate.

Arrays such as `cell.recon.candidates` and `cell.recon.candidates.type` can be joined into lists and stored as strings with something like:

```
foreach(cell.recon.candidates, v, v.name).join("; ")
```

Record

A `row.record` object encapsulates one or more rows that are grouped together, when your project is in records mode. You must call it as `row.record`; `record` will not return values.

Field	Meaning
<code>row.record.index</code>	The index of the current record (starting at 0)
<code>row.record.cells</code>	An array of the <code>cells</code> in the given column of the record
<code>row.record.fromRowIndex</code>	The row index of the first row in the record
<code>row.record.toRowIndex</code>	The row index of the last row in the record + 1 (i.e. the next record)
<code>row.record.rowCount</code>	A count of the number of rows in the record

For example, you can facet by number of rows in each record by creating a `Custom Numeric Facet` (or a `Custom Text Facet`) and entering `row.record.rowCount`.

General Refine Expression Language

Basics

GREL (General Refine Expression Language) is designed to resemble Javascript. Formulas use variables and depend on data types to do things like string manipulation or mathematical calculations:

Example	Output
<code>value + " (approved)"</code>	Concatenate two strings; whatever is in the cell gets converted to a string first
<code>value + 2.239</code>	Add 2.239 to the existing value (if a number); append text "2.239" to the end of the string otherwise
<code>value.trim().length()</code>	Trim leading and trailing whitespace of the cell value and then output the length of the result
<code>value.substring(7, 10)</code>	Output the substring of the value from character index 7, 8, and 9 (excluding character index 10)
<code>value.substring(13)</code>	Output the substring from index 13 to the end of the string

Note that the operator for string concatenation is `+` (not "&" as is used in Excel).

Evaluating conditions uses symbols such as `<`, `>`, `*`, `/`, etc. To check whether two objects are equal, use two equal signs (`value=="true"`).

See the [GREL functions page for a thorough reference](#) on each function and its inputs and outputs. Read on below for more about the general nature of GREL expressions.

Operators

Arithmetic Operators

Refer [GREL functions page](#) for details on Division Operator.

Modulus

When using the `%` operator, if both operands are numbers such as `1 % 2` the result will be a whole number. However, if either or both of the operands are floating-point numbers like `1.0 % 2` they will be promoted to floating point and the result will also be in floating-point format. It's important to note that the `%` operator may not behave as expected with floating-point numbers due to precision issues.

Multiplication

The behavior of the `*` operator is nuanced based on the data types of the operands. When both operands are integers such as `1 * 2`, the result is an integer. Conversely, if either or both operands are floating-point numbers the result becomes a floating-point number. You can use simple evaluations such as `3.5 * 2`

Relational Operators

`==` and `!=` operators are used to assess equality and inequality. For instance, `"a" == "b"` returns false and `"a" != "b"` returns true. When applied to integers, `5 == 5` returns true, while `3 != 3` returns false.

The `<` operator checks if the left operand is less than the right operand, while `<=` checks if it's less than or equal to. Similarly `>` verifies if the left operand is greater than the right and `>=` checks if it's greater than or equal to. These comparison operators apply to numbers, strings and dates.

References

- [String Concatenation](#)
- [Logical Functions](#)

Syntax

In GREL, functions can use either of these two forms:

- `functionName(arg0, arg1, ...)`
- `arg0.functionName(arg1, ...)`

The second form is a shorthand to make expressions easier to read. It simply pulls the first argument out and appends it to the front of the function, with a dot:

Dot notation	Full notation
<code>value.trim().length()</code>	<code>length(trim(value))</code>
<code>value.substring(7, 10)</code>	<code>substring(value, 7, 10)</code>

So, in the dot shorthand, the functions occur from left to right in the order of calling, rather than in the reverse order with parentheses. This allows you to string together multiple functions in a readable order.

The dot notation can also be used to access the member fields of [variables](#). For referring to column names that contain spaces (anything not a continuous string), use square brackets instead of dot notation:

Example	Description
<code>FirstName.cells</code>	Access the cell in the column named “FirstName” of the current row
<code>cells["First Name"]</code>	Access the cell in the column called “First Name” of the current row

Square brackets can also be used to get substrings and sub-arrays, and single items from arrays:

Example	Description
<code>value[1, 3]</code>	A substring of value, starting from character 1 up to but excluding character 3
<code>"internationalization"[1, -2]</code>	Will return “nternationalizati” (negative indexes are counted from the end)
<code>row.columnNames[5]</code>	Will return the name of the fifth column

Any function that outputs an array can use square brackets to select only one part of the array to output as a string (remember that the index of the items in an array starts with 0).

For example, `partition()` would normally output an array of three items: the part before your chosen fragment, the fragment you've identified, and the part after. Selecting only the third

part with `"internationalization".partition("nation")[2]` will output “alization” (and so will [-1], indicating the final item in the array).

Controls

GREL offers controls to support branching and looping (that is, “if” and “for” functions), but unlike functions, their arguments don’t all get evaluated before they get run. A control can decide which part of the code to execute and can affect the environment bindings. Functions, on the other hand, can’t do either. Each control decides which of their arguments to evaluate to `value`, and how.

Please note that the GREL control names are case-sensitive: for example, the `isError()` control can’t be called with `iserror()`.

`if(e, eTrue, eFalse)`

Expression `e` is evaluated to a value. If that value is true, then expression `eTrue` is evaluated and the result is the value of the whole `if()` expression. Otherwise, expression `eFalse` is evaluated and that result is the value.

Examples:

Example expression	Result
<code>if("internationalization".length() > 10, "big string", "small string")</code>	“big string”
<code>if(mod(37, 2) == 0, "even", "odd")</code>	“odd”

Nested if (switch case) example:

```

if(value == 'Place', 'http://www.example.com/Location',
if(value == 'Person', 'http://www.example.com/Agent',
if(value == 'Book', 'http://www.example.com/Publication',
null)))

```

with(e1, variable v, e2)

Evaluates expression e1 and binds its value to variable v. Then evaluates expression e2 and returns that result.

Example expression	Result
<code>with("european union".split(" "), a, a.length())</code>	2
<code>with("european union".split(" "), a, forEach(a, v, v.length()))</code>	[8, 5]
<code>with("european union".split(" "), a, forEach(a, v, v.length()).sum() / a.length())</code>	6.5

filter(e1, v, e test)

Evaluates expression e1 to an array. Then for each array element, binds its value to variable v, evaluates expression test - which should return a boolean. If the boolean is true, pushes v onto the result array.

Expression	Result
<code>filter([3, 4, 8, 7, 9], v, mod(v, 2) == 1)</code>	[3, 7, 9]

forEach(e1, v, e2)

Evaluates expression e1 to an array. Then for each array element, binds its value to variable v, evaluates expression e2, and pushes the result onto the result array. When e1 is a JSON object, `forEach` iterates over its keys.

Expression	Result
<code>forEach([3, 4, 8, 7, 9], v, mod(v, 2))</code>	[1, 0, 0, 1, 1]

forEachIndex(e1, i, v, e2)

Evaluates expression e1 to an array. Then for each array element, binds its index to variable i and its value to variable v, evaluates expression e2, and pushes the result onto the result array.

Expression	Result
<code>forEachIndex(["anne", "ben", "cindy"], i, v, (i + 1) + ". " + v).join(", ")</code>	1. anne, 2. ben, 3. cindy

forRange(n from, n to, n step, v, e)

Iterates over the variable v starting at from, incrementing by the value of step each time while less than to. At each iteration, evaluates expression e, and pushes the result onto the

result array.

forNonBlank(e, v, eNonBlank, eBlank)

Evaluates expression e. If it is non-blank, forNonBlank() binds its value to variable v, evaluates expression eNonBlank and returns the result. Otherwise (if e evaluates to blank), forNonBlank() evaluates expression eBlank and returns that result instead.

Unlike other GREL functions beginning with “for,” forNonBlank() is not iterative. forNonBlank() essentially offers a shorter syntax to achieving the same outcome by using the isNonBlank() function within an “if” statement.

isBlank(e), isNonBlank(e), isNull(e), isNotNull(e), isNumeric(e), isError(e)

Evaluates the expression e, and returns a boolean based on the named evaluation.

Examples:

Expression	Result
<code>isBlank("abc")</code>	false
<code>isNonBlank("abc")</code>	true
<code>isNull("abc")</code>	false
<code>isNotNull("abc")</code>	true
<code>isNumeric(2)</code>	true
<code>isError(1)</code>	false

Expression	Result
<code>isError("abc")</code>	false
<code>isError(1 / 0)</code>	true

Remember that these are controls and not functions: you can't use dot notation (for example, the format `e.isX()` will not work).

Constants

Name	Meaning
true	The boolean constant true
false	The boolean constant false
PI	From Java's Math.PI , the value of pi (that is, 3.1415...)

GREL functions

Besides the reference guide below, OpenRefine's GitHub wiki has [a page with many examples and recipes of frequently-used GREL functions](#).

Reading this reference

For the reference below, the function is given in full-length notation and the in-text examples are written in dot notation. Shorthands are used to indicate the kind of [data type](#) used in each function: s for string, b for boolean, n for number, d for date, a for array, p for a regex pattern, and o for object (meaning any data type), as well as “null” and “error” data types.

If a function can take more than one kind of data as input or can output more than one kind of data, that is indicated with more than one letter (as with “s or a”) or with o for object, meaning it can take any type of data (string, boolean, date, number, etc.).

We also use shorthands for substring (“sub”) and separator string (“sep”). Optional arguments will say “(optional)”.

In places where OpenRefine will accept a string (s) or a regex pattern (p), you can supply a string by putting it in quotes. If you wish to use any [regex](#) notation, wrap the pattern in forward slashes.

Boolean functions

and(b1, b2, ...)

Uses the logical operator AND on two or more booleans to output a boolean. Evaluates

multiple statements into booleans, then returns true if all of the statements are true. For example, `(1 < 3).and(1 < 0)` returns false because one condition is true and one is false.

or(b1, b2, ...)

Uses the logical operator OR on two or more booleans to output a boolean. For example, `(1 < 3).or(1 > 7)` returns true because at least one of the conditions (the first one) is true.

not(b)

Uses the logical operator NOT on a boolean to output a boolean. For example, `not(1 > 7)` returns true because $1 > 7$ itself is false.

xor(b1, b2, ...)

Uses the logical operator XOR (exclusive-or) on two or more booleans to output a boolean. Evaluates multiple statements, then returns true if only one of them is true. For example, `(1 < 3).xor(1 < 7)` returns false because more than one of the conditions is true.

String functions

length(s)

Returns the length of string s as a number.

toString(o, string format (optional))

Takes any value type (string, number, date, boolean, error, null) and gives a string version of that value.

You can use `toString()` to convert numbers to strings with rounding, using an [optional](#)

[string format](#). For example, if you applied the expression `value.toString("%.0f")` to a column:

Input	Output
3.2	3
0.8	1
0.15	0
100.0	100

You can also convert dates to strings, using date parsing syntax built into OpenRefine (see [the toDate\(\) function for details](#)). For example, `value.toString("MMM-dd-yyyy")` would convert the date value [2024-10-15T00:00:00Z] to “Oct-15-2024”.

Note: In OpenRefine, using `toString()` on a null cell outputs the string “null”.

Testing string characteristics

`startsWith(s, sub)`

Returns a boolean indicating whether s starts with sub. For example,

`"food".startsWith("foo")` returns true, whereas `"food".startsWith("bar")` returns false.

`endsWith(s, sub)`

Returns a boolean indicating whether s ends with sub. For example,

`"food".endsWith("ood")` returns true, whereas `"food".endsWith("odd")` returns

false.

contains(s, sub or p)

Returns a boolean indicating whether s contains sub, which is either a substring or a regex pattern. For example, `"food".contains("oo")` returns true whereas `"food".contains("ee")` returns false.

You can search for a regular expression by wrapping it in forward slashes rather than quotes: `"rose is a rose".contains(/\s+/)` returns true. `startsWith()` and `endsWith()` can only take strings, while `contains()` can take a regex pattern, so you can use `contains()` to look for beginning and ending string patterns.

Basic string modification

Case conversion

toLowerCase(s)

Returns string s converted to all lowercase characters.

toUpperCase(s)

Returns string s converted to all uppercase characters.

toTitlecase(s)

Returns string s converted into titlecase: a capital letter starting each word, and the rest of the letters lowercase. For example, `"Once upon a midnight DREARY".toTitlecase()` returns the string “Once Upon A Midnight Dreary”.

Trimming

trim(s)

Returns a copy of the string s with leading and trailing whitespace removed. For example, `" island ".trim()` returns the string “island”. Identical to `strip()`.

strip(s)

Returns a copy of the string s with leading and trailing whitespace removed. For example, `" island ".strip()` returns the string “island”. Identical to `trim()`.

chomp(s, sep)

Returns a copy of string s with the string sep removed from the end if s ends with sep; otherwise, just returns s. For example, `"barely".chomp("ly")` and `"bare".chomp("ly")` both return the string “bare”.

Substring

substring(s, n from, n to (optional))

Returns the substring of s starting from character index from, and up to (excluding) character index to. If the to argument is omitted, substring will output to the end of s. For example, `"profound".substring(3)` returns the string “found”, and `"profound".substring(2, 4)` returns the string “of”.

Remember that character indices start from zero. A negative character index counts from the end of the string. For example, `"profound".substring(0, -1)` returns the string “profoun”.

slice(s, n from, n to (optional))

Identical to `substring()` in relation to strings. Also works with arrays; see [Array functions](#)

section.

get(s, n from, n to (optional))

Identical to substring() in relation to strings. Also works with named fields. Also works with arrays; see [Array functions section](#).

Find and replace

indexOf(s, sub)

Returns the first character index of sub as it first occurs in s; or, returns -1 if s does not contain sub. For example, `"internationalization".indexOf("nation")` returns 5, whereas `"internationalization".indexOf("world")` returns -1.

lastIndexOf(s, sub)

Returns the first character index of sub as it last occurs in s; or, returns -1 if s does not contain sub. For example, `"parallel".lastIndexOf("a")` returns 3 (pointing at the second "a").

replace(s, s or p find, s replace)

Returns the string obtained by replacing the find string with the replace string in the inputted string. For example, `"The cow jumps over the moon and moos".replace("oo", "ee")` returns the string "The cow jumps over the meen and mees". Find can be a regex pattern. For example, `"The cow jumps over the moon and moos".replace(/\s+/, "_")` will return "The_cow_jumps_over_the_moon_and_moos".

You cannot find or replace nulls with this, as null is not a string. You can instead:

1. Facet by null and then bulk-edit them to a string, or
2. Transform the column with an expression such as `if(value==null, "new", value)`.

replaceChars(s, s find, s replace)

Returns the string obtained by replacing a character in s, identified by find, with the corresponding character identified in replace. For example, "Téxt thát was optícálly recógnizéd".`replaceChars("áéíóú", "aeiou")` returns the string "Text that was optically recognized". You cannot use this to replace a single character with more than one character.

replaceEach(s, a find, a replace)

Returns the string obtained by replacing each element in s of a find array with the corresponding element of a replace array, sequentially. For example, "The cow jumps over the moon and moos".`replaceEach(["th", "moo"], ["ex", "mee"])` returns the string "The cow jumps over exe meen and mees".

The length of the find array must be the same as the length of the replace array. If the length of the find array is greater than the length of the replace array, the last element of the replace array is used for all remaining elements in the find array.

This function is available since OpenRefine 3.6.

find(s, sub or p)

Outputs an array of all consecutive substrings inside string s that match the substring or **regex** pattern p. For example, "`a beads ab mol oe i`".`find(/[aeio]+/)` would result in the array ["a", "ea", "a", "o", "oei"].

You can supply a substring instead of p, by putting it in quotes, and OpenRefine will compile it into a regex pattern. Anytime you supply quotes, OpenRefine interprets the contents as a string, not regex. If you wish to use any regex notation, wrap the pattern in forward slashes.



See also [match\(\)](#)

match(s, p)

Attempts to match the string s in its entirety against the [regex](#) pattern p and, if the pattern is found, outputs an array of all [capturing groups](#) (found in order). For example,

"230.22398, 12.3480".[match\(/.*\(\d\d\d\d\)/\)](#) returns an array of 1 substring: ["3480"]. It does not find 2239 as the first sequence with four digits, because the regex indicates the four digits must come at the end of the string.

You will need to convert the array to a string to store it in a cell, with a function such as [toString\(\)](#). An empty array [] is returned when there is no match to the desired substrings. A null is output when the entire regex does not match.

Remember to enclose your regex in forward slashes, and to escape characters and use parentheses as needed. Parentheses denote a desired substring (capturing group); for example, ".*(\d\d\d\d)" would return an array containing a single value, while "(.)(\d\d\d\d)" would return two. So, if you are looking for a desired substring anywhere within a string, use the syntax [value.match\(/.*\(desired-substring-regex\).*/\)](#).

For example, if [value](#) is "hello 123456 goodbye", the following would occur:

Expression	Result
value.match(/\d{6}/)	null (does not match the full string)
value.match(/.*\d{6}.*/)	[] (no indicated substring)
value.match(/.*(\d{6}).*/)	["123456"] (array with one value)

Expression	Result
<code>value.match(/(.*)(\d{6})(.*)/)</code>	["hello ", "123456", " goodbye"] (array with three values)



TIP

See also [find\(\)](#)

String parsing and splitting

`toNumber(s)`

Returns a string converted to a number. Will attempt to convert other formats into a string, then into a number. If the value is already a number, it will return the number.

`split(s, s or p sep, b preserveTokens (optional))`

Returns the array of strings obtained by splitting s by sep. The separator can be either a string or a regex pattern. For example, `"fire, water, earth, air".split(",")` returns an array of 4 strings: ["fire", " water", " earth", " air"]. Note that the space characters are retained but the separator is removed. If you include "true" for the `preserveTokens` boolean, empty segments are preserved.

`splitByLengths(s, n1, n2, ...)`

Returns the array of strings obtained by splitting s into substrings with the given lengths. For example, `"internationalization".splitByLengths(5, 6, 3)` returns an array of 3 strings: ["inter", "nation", "ali"]. Excess characters are discarded from the output array.

Like other functions that return an array, it also allows array slicing on the returned array. In

that case, it returns the array consisting of a subset of elements between $i1$ and $(i2 - 1)$.

For example,

Expression	Result
<code>"internationalization".splitByLengths(5, 6, 3)[0,3]</code>	Returns an array of 3 strings: ["inter", "nation", "ali"].
<code>"internationalization".splitByLengths(5, 6, 3)[0,2]</code>	Returns an array of 2 strings: ["inter", "nation"]
<code>"internationalization".splitByLengths(5, 6, 3)[1,3]</code>	Returns an array of 2 string: ["nation", "ali"]
<code>"internationalization".splitByLengths(5, 6, 3)[1]</code>	Returns string at position 1: "nation"

smartSplit(s, s or p sep (optional))

Returns the array of strings obtained by splitting s by sep , or by guessing either tab or comma separation if there is no sep given. Handles quotes properly and understands cancelled characters. The separator can be either a string or a regex pattern. For example, `value.smartSplit("\n")` will split at a carriage return or a new-line character.

Note: `value.escape('javascript')` is useful for previewing unprintable characters prior to using `smartSplit()`.

splitByCharType(s)

Returns an array of strings obtained by splitting s into groups of consecutive characters each time the characters change [Unicode categories](#). For example,

`"HenryCTaylor".splitByCharType()` will result in an array of ["H", "enry", "CT", "aylor"].

It is useful for separating letters and numbers: `"BE1A3E".splitByCharType()` will result in ["BE", "1", "A", "3", "E"].

partition(s, s or p fragment, b omitFragment (optional))

Returns an array of strings [a, fragment, z] where a is the substring within s before the first occurrence of fragment, and z is the substring after fragment. Fragment can be a string or a regex. For example, `"internationalization".partition("nation")` returns 3 strings: ["inter", "nation", "alization"]. If s does not contain fragment, it returns an array of [s, "", ""] (the original unpartitioned string, and two empty strings).

If the omitFragment boolean is true, for example with

`"internationalization".partition("nation", true)`, the fragment is not returned. The output is ["inter", "alization"].

As an example of using a regex for the fragment, the expression

`"abcdefghijklm".partition(/c.e/)` will output ["ab", "cde", "fg"].

rpartition(s, s or p fragment, b omitFragment (optional))

Returns an array of strings [a, fragment, z] where a is the substring within s before the last occurrence of fragment, and z is the substring after the last instance of fragment. (Rpartition means “reverse partition.”) For example, `"parallel".rpartition("a")` returns 3 strings: ["par", "a", "llel"]. Otherwise works identically to partition() above.

Encoding and hashing

diff(s1, s2, s timeUnit (optional))

Takes two strings and compares them, returning a string. Returns the remainder of s2 starting with the first character where they differ. For example, `"cacti".diff("cactus")` returns "us". Also works with dates; see [Date functions](#).

escape(s, s mode)

Escapes s in the given escaping mode. The mode can be one of: "html", "xml", "csv", "url", "javascript". Note that quotes are required around your mode. See the [recipes](#) for examples of escaping and unescaping.

unescape(s, s mode)

Unescapes s in the given escaping mode. The mode can be one of: "html", "xml", "csv", "url", "javascript". Note that quotes are required around your mode. See the [recipes](#) for examples of escaping and unescaping.

encode(s, s encoding)

Encodes the string, s in the specified encoding. The encoding can be one of: "base16", "base32", "base32hex", "base64", "base64url". For example, `encode("abc", "base64")` returns "YWJj".

This function is available since OpenRefine 3.6.

decode(s, s encoding)

Decodes the string, s in the specified encoding. The encoding can be one of: "base16", "base32", "base32hex", "base64", "base64url". For example, `decode("YWJj", "base64")` returns "abc".

This function is available since OpenRefine 3.6.

md5(o)

Returns the [MD5 hash](#) of an object. If fed something other than a string (array, number, date, etc.), md5() will convert it to a string and deliver the hash of the string. For example, `"internationalization".md5()` will return 2c55a1626e31b4e373ceedaa9adc12a3.

sha1(o)

Returns the [SHA-1 hash](#) of an object. If fed something other than a string (array, number, date, etc.), sha1() will convert it to a string and deliver the hash of the string. For example, `"internationalization".sha1()` will return `cd05286ee0ff8a830dbdc0c24f1cb68b83b0ef36`.

phonetic(s, s encoding)

Returns a phonetic encoding of a string, based on an available phonetic algorithm. See the [section on phonetic clustering](#) for more information. Can be one of the following supported phonetic methods: [metaphone](#), [doublemetaphone](#), [metaphone3](#), [soundex](#), [cologne](#). Quotes are required around your encoding method. For example, `"Ruth Prawer Jhabvala".phonetic("metaphone")` outputs the string "R0PRWRJHBFL".

reinterpret(s, s encoderTarget, s encoderSource)

Returns s reinterpreted through the given character encoders. You must supply one of the [supported encodings](#) for each of the original source and the target output. Note that quotes are required around your character encoder.

When an OpenRefine project is started, data is imported and interpreted. A specific character encoding is identified or manually selected at that time (such as UTF-8). You can reinterpret a column into another specified encoding using this function. This function may not fix your data; it may be better to use this in conjunction with new projects to test the interpretation, and pre-format your data as needed.

fingerprint(s)

Returns the fingerprint of s, a string that is the first step in [fingerprint clustering methods](#): it will trim whitespaces, convert all characters to lowercase, remove punctuation, sort words alphabetically, etc. For example, `"Ruth Prawer Jhabvala".fingerprint()` outputs the string "jhabvala praver ruth".

`ngram(s, n)`

Returns an array of the word n-grams of s. That is, it lists all the possible consecutive combinations of n words in the string. For example, `"Ruth Prawer Jhabvala".ngram(2)`

would output the array ["Ruth Prawer", "Prawer Jhabvala"]. A word n-gram of 1 simply lists all the words in original order; an n-gram larger than the number of words in the string will only return the original string inside an array (e.g. `"Ruth Prawer Jhabvala".ngram(4)` would simply return ["Ruth Prawer Jhabvala"]).

`ngramFingerprint(s, n)`

Returns the [n-gram fingerprint](#) of s. For example, `"banana".ngram(2)` would output "anbana", after first generating the 2-grams "ba an na an na", removing duplicates, and sorting them alphabetically.

`unicode(s)`

Returns an array of strings describing each character of s in their full unicode notation. For example, `"Bernice Rubens".unicode()` outputs [66, 101, 114, 110, 105, 99, 101, 32, 82, 117, 98, 101, 110, 115].

`unicodeType(s)`

Returns an array of strings describing each character of s by their unicode type. For example, `"Bernice Rubens".unicodeType()` outputs ["uppercase letter", "lowercase letter", "space separator", "uppercase letter", "lowercase letter", "lowercase letter", "lowercase letter", "lowercase letter", "lowercase letter"].

Translating

`detectLanguage(s)`

Returns a string locale describing the language of `s`, with support for 71 languages as stated [here](#). For example, `"Hello, world!".detectLanguage()` outputs "en".

This function is available since OpenRefine 3.6.

Format-based functions (JSON, HTML, XML)

`jsonize(o)`

Quotes a value as a JSON literal value.

`parseJson(s)`

Parses a string as JSON. `get()` can then be used with `parseJson()`: for example,

`parseJson(" { 'a' : 1 } ").get("a")` returns 1.

For example, from the following JSON array in `value`, we want to get all instances of "keywords" having the same object string name of "text", and combine them, using the `forEach()` function to iterate over the array.

```
{  
  "status": "OK",  
  "url": "",  
  "language": "english",  
  "keywords": [  
    {"text": "red", "color": "#FF0000"},  
    {"text": "blue", "color": "#0000FF"},  
    {"text": "green", "color": "#008000"}]
```

The GREL expression

`forEach(value.parseJson().keywords, v, v.text).join(":::")` will output "York en route:::Anthony Eden:::President Eisenhower".

Jsoup XML and HTML parsing

`parseHtml(s)`

Given a cell full of HTML-formatted text, `parseHtml()` simplifies HTML tags (such as by removing “ /” at the end of self-closing tags), closes any unclosed tags, and inserts linebreaks and indents for cleaner code. You cannot pass `parseHtml()` a URL, but you can pre-fetch HTML with the [Add column by fetching URLs](#) menu option.

A cell cannot store the output of `parseHtml()` unless you convert it with `toString()`: for example, `value.parseHtml().toString()`.

When `parseHtml()` simplifies HTML, it can sometimes introduce errors. When closing tags, it makes its best guesses based on line breaks, indentation, and the presence of other tags. You may need to manually check the results.

You can then extract or [select\(\)](#) which portions of the HTML document you need for further splitting, partitioning, etc. An example of extracting all table rows from a div using `parseHtml().select()` together is described more in depth at [StrippingHTML](#).

`parseXml(s)`

Given a cell full of XML-formatted text, `parseXml()` returns a full XML document and adds any missing closing tags. You can then extract or [select\(\)](#) which portions of the XML document you need for further splitting, partitioning, etc. Functions the same way as `parseHtml()` is described above.

select(s, element)

Returns an array of all the desired elements from an HTML or XML document, if the element exists. Elements are identified using the [Jsoup selector syntax](#). For example,

`value.parseHtml().select("img.portrait")[0]` would return the entirety of the first “img” tag with the “portrait” class found in the parsed HTML inside `value`. Returns an empty array if no matching element is found. Use with `toString()` to capture the results in a cell. A tutorial of `select()` is shown in [StrippingHTML](#).

You can use `select()` more than once:

```
value.parseHtml().select("div#content")[0].select("tr").toString()
```

htmlAttr(s, element)

Returns a string from an attribute on an HTML element. Use it in conjunction with `parseHtml()` as in the following example:

`value.parseHtml().select("a.email")[0].htmlAttr("href")` would retrieve the email address attached to a link with the “email” class.

xmlAttr(s, element)

Returns a string from an attribute on an XML element. Functions the same way `htmlAttr()` is described above. Use it in conjunction with `parseXml()`.

htmlText(element)

Returns a string of the text from within an HTML element (including all child elements), removing HTML tags and line breaks inside the string. Use it in conjunction with `parseHtml()` and `select()` to provide an element, as in the following example:

```
value.parseHtml().select("div.footer")[0].htmlText()
```

xmlText(element)

Returns a string of the text from within an XML element (including all child elements). Functions the same way `htmlText()` is described above. Use it in conjunction with `parseXml()` and `select()` to provide an element.

wholeText(element)

Selects the (unencoded) text of an element and its children, including any new lines and spaces, and returns a string of unencoded, un-normalized text. Use it in conjunction with `parseHtml()` and `select()` to provide an element as in the following example:

```
value.parseHtml().select("div.footer")[0].wholeText()
```

This function is available since OpenRefine 3.5.

innerHTML(element)

Returns the [inner HTML](#) of an HTML element. This will include text and children elements within the element selected. Use it in conjunction with `parseHtml()` and `select()` to provide an element.

innerHTML(element)

Returns the inner XML elements of an XML element. Does not return the text directly inside your chosen XML element - only the contents of its children. To select the direct text, use `ownText()`. To select both, use `xmlText()`. Use it in conjunction with `parseXml()` and `select()` to provide an element.

ownText(element)

Returns the text directly inside the selected XML or HTML element only, ignoring text inside children elements (for this, use `innerHTML()`). Use it in conjunction with a parser and `select()` to provide an element.

parent(element)

Returns the parent node or null if no parent. Use it in conjunction with `parseHtml()` and `select()` to provide an element.

This function is available since OpenRefine 3.6.

URI parsing

parseUri(s)

Given a valid URI string (for example: <https://www.openrefine.org:80/documentation#download?format=xml&os=mac>), `parseUri()` returns a JSON object with the following properties:

- `scheme`: The scheme of the URI, e.g. `http`
- `host`: the host of the URI (e.g. `www.openrefine.org`)
- `port`: the port of the URI (e.g. `80`)
- `path`: the path of the URI (e.g. `/documentation`)
- `query`: the query of the URI (e.g. `format=xml&os=mac`)
- `authority`: the authority of the URI (e.g. `www.openrefine.org:80`)
- `fragment`: the fragment of the URI (e.g. `download`)
- `query_params`: the query of the URI as an object (e.g. `{format: "xml", os: "mac"}`)

This function is available since OpenRefine 3.6.

Array functions

`length(a)`

Returns the size of an array, meaning the number of objects inside it. Arrays can be empty, in which case `length()` will return 0.

`slice(a, n from, n to (optional))`

Returns a sub-array of a given array, from the first index provided and up to and excluding the optional last index provided. Remember that array objects are indexed starting at 0. If the to value is omitted, it is understood to be the end of the array. For example, `[0, 1, 2, 3, 4].slice(1, 3)` returns [1, 2], and `[0, 1, 2, 3, 4].slice(2)` returns [2, 3, 4].

Also works with strings; see [String functions](#).

`get(a, n from, n to (optional))`

Returns a sub-array of a given array, from the first index provided and up to and excluding the optional last index provided. Remember that array objects are indexed starting at 0.

If the to value is omitted, only one array item is returned, as a string, instead of a sub-array. To return a sub-array from one index to the end, you can set the to argument to a very high number such as `value.get(2, 999)` or you can use something like `with(value, a, a.get(1, a.length()))` to count the length of each array.

Also works with strings; see [String functions](#).

`inArray(a, s)`

Returns true if the array contains the desired string, and false otherwise. Will not convert data types; for example, `[1, 2, 3, 4].inArray("3")` will return false.

reverse(a)

Reverses the array. For example, `[0, 1, 2, 3].reverse()` returns the array [3, 2, 1, 0].

sort(a)

Sorts the array in ascending order. Sorting is case-sensitive, uppercase first and lowercase second. For example, `["al", "Joe", "Bob", "jim"].sort()` returns the array ["Bob", "Joe", "al", "jim"].

sum(a)

Return the sum of the numbers in the array. For example, `[2, 1, 0, 3].sum()` returns 6.

join(a, sep)

Joins the items in the array with sep, and returns it all as a string. For example, `["and", "or", "not"].join("/")` returns the string "and/or/not".

uniques(a)

Returns the array with duplicates removed. Case-sensitive. For example, `["al", "Joe", "Bob", "Joe", "Al", "Bob"].uniques()` returns the array ["Joe", "al", "Al", "Bob"].

As of OpenRefine 3.4.1, uniques() reorders the array items it returns; in 3.4 beta 644 and onwards, it preserves the original order (in this case, ["al", "Joe", "Bob", "Al"]).

Date functions

now()

Returns the current time according to your system clock, in the [ISO 8601 extended format](#)

(converted to UTC). For example, 10:53am (and 00 seconds) on November 26th 2020 in EST returns [date 2020-11-26T15:53:00Z].

`toDate(o, b monthFirst, s format1, s format2, ...)`

Returns the inputted object converted to a date object. Without arguments, it returns the ISO 8601 extended format. With arguments, you can control the output format:

- `monthFirst`: set false if the date is formatted with the day before the month.
- `formatN`: attempt to parse the date using an ordered list of possible formats. Supply formats based on the [SimpleDateFormat](#) syntax (and see the table below for a handy reference).

For example, you can parse a column containing dates in different formats, such as cells with "Nov-09" and "11/09", using `value.toDate('MM/yy', 'MMM-yy').toString('yyyy-MM')` and both will output "2009-11". For another example, "1/4/2012 13:30:00" can be parsed into a date using `value.toDate('d/M/y H:m:s')`. If parsing a date with text components in a language other than your system language you can specify a language code as the `format1` argument. For example, a French language date such as "10 janvier 2023" could be parsed with `value.toDate('fr', 'dd MMM yyyy')`.

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year	Month	July; Jul; 07

Letter	Date or Time Component	Presentation	Examples
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	AM/PM marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in AM/PM (0-11)	Number	0
h	Hour in AM/PM (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55

Letter	Date or Time Component	Presentation	Examples
S	Millisecond	Number	978
n	Nanosecond	Number	789000
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

`diff(d1, d2, s timeUnit)`

Given two dates, returns a number indicating the difference in a given time unit (see the table below). For example, `diff(("Nov-11".toDate('MMM-yy')), ("Nov-09".toDate('MMM-yy')), "weeks")` will return 104, for 104 weeks, or two years. The later date should go first. If the output is negative, invert d1 and d2.

Also works with strings; see [diff\(\) in string functions](#).

`inc(d, n, s timeUnit)`

Returns a date changed by the given amount in the given unit of time (see the table below). The default unit is “hour”. A positive value increases the date, and a negative value moves it back in time. For example, if you want to move a date backwards by two months, use `value.inc(-2, "month")`.

datePart(d, s timeUnit)

Returns part of a date. The data type returned depends on the unit (see the table below).

OpenRefine supports the following values for timeUnit:

Unit	Date part returned	Returned data type	Example using [date 2014-03-14T05:30:04.000Z] as value
years, year	Year	Number	value.datePart("years") → 2014
months, month	Month	Number	value.datePart("months") → 2
weeks, week, w	Week of the month	Number	value.datePart("weeks") → 3
days, day, d	Day of the month	Number	value.datePart("days") → 14
weekday	Day of the week	String	value.datePart("weekday") → Friday
hours, hour, h	Hour	Number	value.datePart("hours") → 5
minutes, minute, min	Minute	Number	value.datePart("minutes") → 30
seconds, sec, s	Seconds	Number	value.datePart("seconds") → 04

Unit	Date part returned	Returned data type	Example using [date 2014-03-14T05:30:04.000789000Z] as value
milliseconds, ms, S	Milliseconds	Number	value.datePart("milliseconds") → 789
nanos, nano, n	Nanoseconds	Number	value.datePart("n") → 789000
time	Milliseconds between input and the Unix Epoch	Number	value.datePart("time") → 1394775004000

`timeSinceUnixEpochToDate(duration, scale)`

Converts a time as measured by the duration since the Unix Epoch (1970-01-01) to a date object. The second parameter indicates the unit of the duration, and can be ["second"](#), ["millisecond"](#) or ["microsecond"](#). If the unit is not provided, it is assumed to be ["second"](#).

This function is available since OpenRefine 3.6.

Math functions

For integer division and precision, you can use simple evaluations such as [1 / 2](#), which is equivalent to [floor\(1/2\)](#) - that is, it returns only whole number results. If either operand is a floating point number, they both get promoted to floating point and a floating

point result is returned. You can use `1 / 2.0` or `1.0 / 2` or `1.0 * x / y` (if you're working with variables of unknown contents).

⚠ CAUTION

Some of these math functions don't recognize integers when supplied as the first argument in dot notation (e.g., `5.cos()` simply returns 5 instead of the expected result). To ensure operations are successful, always wrap the first argument in brackets, such as `(value).cos()`.

Function	Use	Example
<code>abs(n)</code>	Returns the absolute value of a number.	<code>abs(-6)</code> returns 6.
<code>acos(n)</code>	Returns the arc cosine of an angle, in the range 0 through PI.	<code>acos(0.345)</code> returns 1.218557541697832.
<code>asin(n)</code>	Returns the arc sine of an angle in the range of -PI/2 through PI/2.	<code>asin(0.345)</code> returns 0.35223878509706474.
<code>atan(n)</code>	Returns the arc tangent of an angle in the range of -PI/2 through PI/2.	<code>atan(0.345)</code> returns 0.3322135507465967.
<code>atan2(n1, n2)</code>	Converts rectangular coordinates (n1, n2) to polar (r, theta). Returns number theta.	<code>atan2(0.345, 0.6)</code> returns 0.5218342798144103.
<code>ceil(n)</code>	Returns the ceiling of a number.	<code>3.7.ceil()</code> returns 4 and <code>-3.7.ceil()</code>

Function	Use	Example
		returns -3.
<code>combin(n1, n2)</code>	Returns the number of combinations for n2 elements as divided into n1.	<code>combin(20,2)</code> returns 190.
<code>cos(n)</code>	Returns the trigonometric cosine of a value.	<code>cos(5)</code> returns 0.28366218546322625.
<code>cosh(n)</code>	Returns the hyperbolic cosine of a value.	<code>cosh(5)</code> returns 74.20994852478785.
<code>degrees(n)</code>	Converts an angle from radians to degrees.	<code>degrees(5)</code> returns 286.4788975654116.
<code>even(n)</code>	Rounds the number up to the nearest even integer.	<code>even(5)</code> returns 6.
<code>exp(n)</code>	Returns e raised to the power of n.	<code>exp(5)</code> returns 148.4131591025766.
<code>fact(n)</code>	Returns the factorial of a number, starting from 1.	<code>fact(5)</code> returns 120.
<code>factn(n1, n2)</code>	Returns the factorial of n1, starting from n2.	<code>factn(10,3)</code> returns 280.
<code>floor(n)</code>	Returns the floor of a number.	<code>3.7.floor()</code> returns 3 and <code>-3.7.floor()</code>

Function	Use	Example
		returns -4.
<code>gcd(n1, n2)</code>	Returns the greatest common denominator of two numbers.	<code>gcd(95, 135)</code> returns 5.
<code>lcm(n1, n2)</code>	Returns the least common multiple of two numbers.	<code>lcm(95, 135)</code> returns 2565.
<code>ln(n)</code>	Returns the natural logarithm of n.	<code>ln(5)</code> returns 1.6094379124341003.
<code>log(n)</code>	Returns the base 10 logarithm of n.	<code>log(5)</code> returns 0.6989700043360189.
<code>max(n1, n2)</code>	Returns the larger of two numbers.	<code>max(3, 10)</code> returns 10.
<code>min(n1, n2)</code>	Returns the smaller of two numbers.	<code>min(3, 10)</code> returns 3.
<code>mod(n1, n2)</code>	Returns n_1 modulus n_2 . Note: <code>value.mod(9)</code> will work, whereas <code>74.mod(9)</code> will not work.	<code>mod(74, 9)</code> returns 2.
<code>multinomial(n1, n2 ... (optional))</code>	Calculates the multinomial of one number or a series of numbers.	<code>multinomial(2, 3)</code> returns 10.
<code>odd(n)</code>	Rounds the number up to the nearest odd integer.	<code>odd(10)</code> returns 11.

Function	Use	Example
<code>pow(n1, n2)</code>	Returns n1 raised to the power of n2. Note: <code>value.pow(3)</code> will work , whereas <code>2.pow(3)</code> will not work.	<code>pow(2, 3)</code> returns 8 (2 cubed) and <code>pow(3, 2)</code> returns 9 (3 squared). The square root of any numeric value can be called with <code>value.pow(0.5)</code> .
<code>quotient(n1, n2)</code>	Returns the integer portion of a division (truncated, not rounded), when supplied with a numerator and denominator.	<code>quotient(9, 2)</code> returns 4.
<code>radians(n)</code>	Converts an angle in degrees to radians.	<code>radians(10)</code> returns 0.17453292519943295.
<code>random(n lowerBound, n upperBound)</code>	Returns a random integer in the interval between the lower and upper bounds (inclusively). Will output a different random number in each cell in a column. If no arguments are provided, returns a number in the range <code>0.0 <= x < 1.0</code>	
<code>round(n)</code>	Rounds a number to the nearest integer.	<code>3.7.round()</code> returns 4 and <code>-3.7.round()</code> returns -4.
<code>sin(n)</code>	Returns the trigonometric sine of an	<code>sin(10)</code> returns

Function	Use	Example
	angle.	-0.5440211108893698.
<code>sinh(n)</code>	Returns the hyperbolic sine of an angle.	<code>sinh(10)</code> returns 11013.232874703393.
<code>sum(a)</code>	Sums the numbers in an array. Ignores non-number items. Returns 0 if the array does not contain numbers.	<code>sum([10, 2, three])</code> returns 12.
<code>tan(n)</code>	Returns the trigonometric tangent of an angle.	<code>tan(10)</code> returns 0.6483608274590866.
<code>tanh(n)</code>	Returns the hyperbolic tangent of a value.	<code>tanh(10)</code> returns 0.9999999958776927.

Other functions

`type(o)`

Returns a string with the data type of o, such as undefined, string, number, boolean, etc. For example, a `Transform` operation using `value.type()` will convert all cells in a column to strings of their data types.

`facetCount(choiceValue, s facetExpression, s columnName)`

Returns the facet count corresponding to the given choice value, by looking for the facetExpression in the choiceValue in columnName. For example, to create facet counts for the following table, we could generate a new column based on “Gift” and enter in

`value.facetCount("value", "Gift")`. This would add the column we've named "Count":

Gift	Recipient	Price	Count
lamp	Mary	20	1
clock	John	57	2
watch	Amit	80	1
clock	Claire	62	2

The facet expression, wrapped in quotes, can be useful to manipulate the inputted values before counting. For example, you could do a textual cleanup using `fingerprint()`:

`(value.fingerprint()).facetCount(value.fingerprint(), "Gift")`.

hasField(o, s name)

Returns a boolean indicating whether o has a member field called `name`. For example, `cell.recon.hasField("match")` will return false if a reconciliation match hasn't been selected yet, or true if it has. You cannot chain your desired fields: for example, `cell.hasField("recon.match")` will return false even if the above expression returns true).

coalesce(o1, o2, o3, ...)

Returns the first non-null from a series of objects. For example, `coalesce(value, "")` would return an empty string "" if `value` was null, but otherwise return `value`.

cross(cell, s projectName (optional), s columnName (optional))

Returns an array of zero or more rows in the project `projectName` for which the cells in their column `columnName` have the same content as the cell in your chosen column. For

example, if two projects contained matching names, and you wanted to pull addresses for people by their names from a project called “People” you would apply the following expression to your column of names:

```
cell.cross("People", "Name")[0].cells["Address"].value
```

This would match your current column to the “Name” column in “People” and, using those matches, pull the respective “Address” value into your current project.

You may need to do some data preparation with cross(), such as using trim() on your key columns or deduplicating values.

The first argument will be interpreted as `cell.value` if set to `cell`. If you omit projectName and columnName, they will default to the current project and index column (number 0).

Recipes and more examples for using cross() can be found [on our wiki](#).

Jython & Clojure

Jython

Jython 2.7.2 comes bundled with the default installation of OpenRefine 3.4.1. You can add libraries and code by following [this tutorial](#). A large number of Python files (`.py` or `.pyc`) are compatible.

Python code that depends on C bindings will not work in OpenRefine, which uses Java / Jython only. Since Jython is essentially Java, you can also import Java libraries and utilize those.

You will need to restart OpenRefine, so that new Jython or Python libraries are initialized during startup.

OpenRefine now has [most of the Jsoup.org library built into GREL functions](#) for parsing and working with HTML and XML elements.

Syntax

Expressions in Jython must have a `return` statement:

```
return value[1:-1]
```

```
return rowIndex%2
```

Fields have to be accessed using the bracket operator rather than dot notation:

```
return cells["col1"]["value"]
```

For example, to access the [edit distance](#) between a reconciled value and an original cell value using [recon variables](#):

```
return cell["recon"]["features"]["nameLevenshtein"]
```

To return the lower case of [value](#) (if the value is not null):

```
if value is not None:  
    return value.lower()  
else:  
    return None
```

Tutorials

- [Extending Jython with pypi modules](#)
- [Working with phone numbers using Java libraries inside Python](#)

Full documentation on the Jython language can be found on its official site:

<http://www.jython.org>.

Clojure

Clojure 1.10.1 comes bundled with the default installation of OpenRefine 3.4.1. At this time, not all [variables](#) can be used with Clojure expressions: only [value](#), [row](#), [rowIndex](#), [cell](#), and [cells](#) are available.

For example, functions can take the form

```
(.. value (toUpperCase) )
```

Or can look like

```
(-> value (str/split #" ") last )
```

which functions like `value.split(" ")` in GREL.

For help with syntax, see the [Clojure website's guide to syntax](#).

User-contributed Clojure recipes can be found on our wiki at <https://github.com/OpenRefine/OpenRefine/wiki/Recipes#11-clojure>.

Full documentation on the Clojure language can be found on its official site: <https://clojure.org/>.

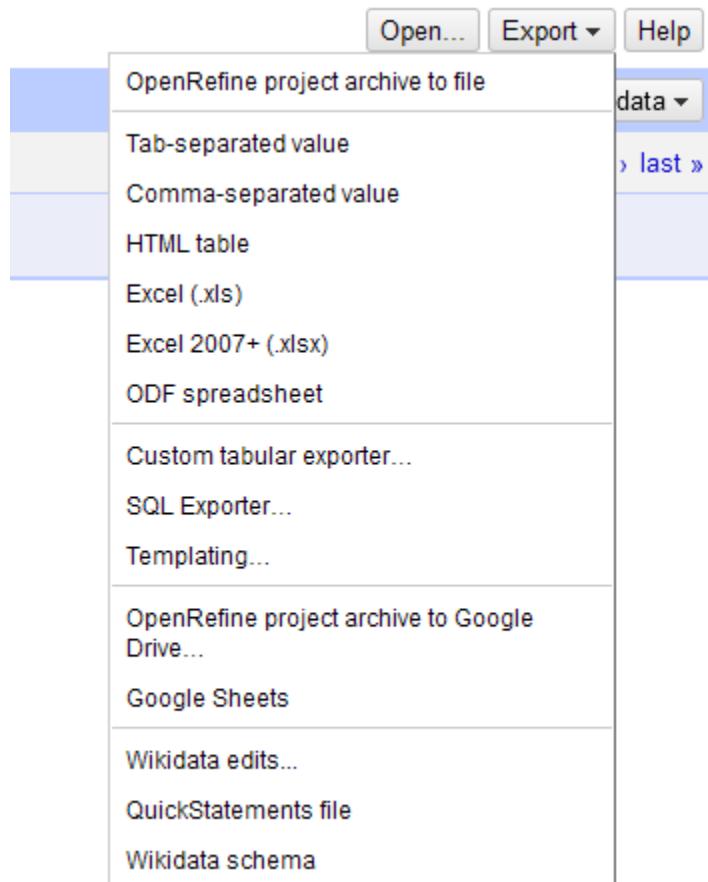
Exporting your work

Overview

Once your dataset is ready, you will need to get it out of OpenRefine and into the system of your choice. OpenRefine outputs a number of file formats, can upload your data directly into Google Sheets, and can create or update statements on Wikidata.

You can also [export your full project data](#) so that it can be opened by someone else using OpenRefine (or yourself, on another computer).

Export data



Many of the options only export data in the current view - that is, with current filters and facets applied. Some will give you the choice to export your entire dataset or just the currently-viewed rows.

To export data from a project, click the **Export** dropdown button in the top right corner and pick the format you want. Your options are:

- Tab-separated value (TSV) or Comma-separated value (CSV)
- HTML-formatted table

- Excel spreadsheet (XLS or XLSX)
- Open Document Format (ODF) spreadsheet (ODS)
- Upload to Google Sheets (requires [Google account authorization](#))
- [Custom tabular exporter](#)
- [SQL statement exporter](#)
- [Templating exporter](#), which generates JSON by default

You can also export reconciled data to Wikidata, or export your Wikidata schema for future use with other OpenRefine projects:

- [Upload edits to Wikidata](#)
- [Export to QuickStatements](#) (version 1)
- [Export Wikidata schema](#)

Custom tabular exporter

Custom Tabular Exporter

Content Download Upload Option Code

Select and Order Columns to Export

Author
 Title
 Publication year
 Literary genre
 Genre ID
 Country
 coordinate location
 continent
 Reference

Options for **Author**

For reconciled cells, output

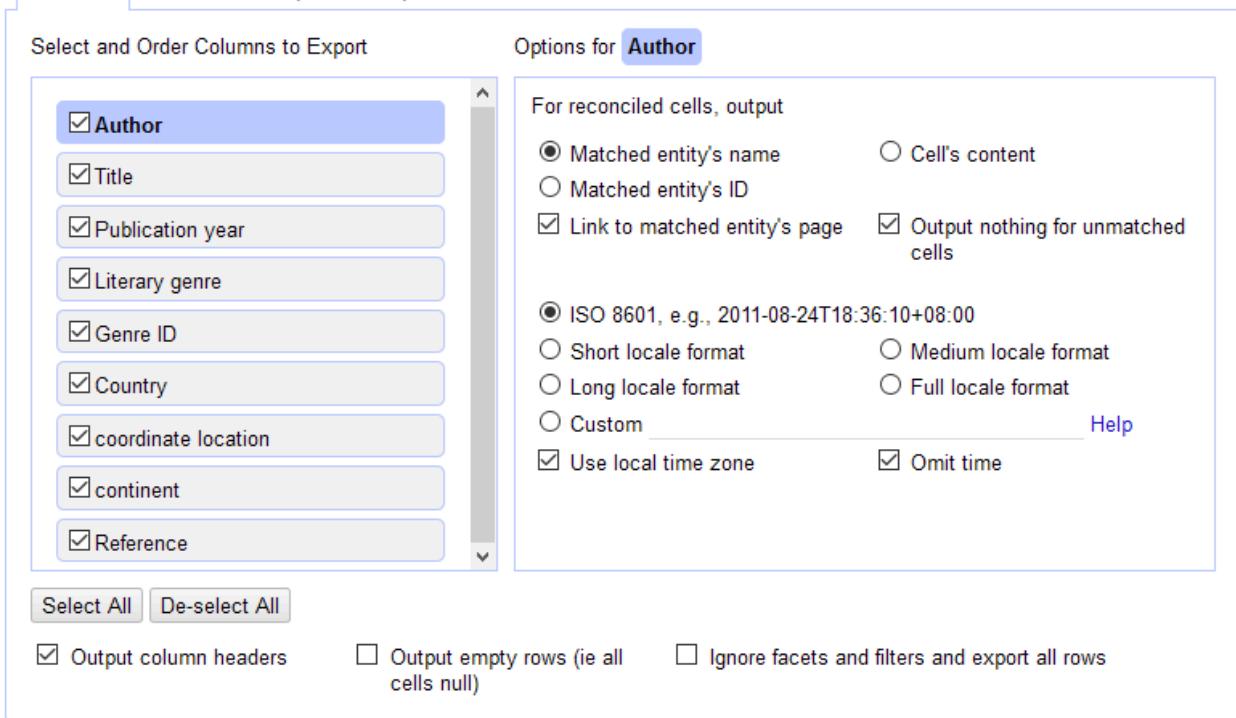
Matched entity's name Cell's content
 Matched entity's ID Link to matched entity's page Output nothing for unmatched cells
 ISO 8601, e.g., 2011-08-24T18:36:10+08:00 Short locale format Medium locale format
 Long locale format Full locale format
 Custom Use local time zone Omit time

[Help](#)

Select All De-select All

Output column headers Output empty rows (ie all cells null) Ignore facets and filters and export all rows

Cancel



With the custom tabular exporter, you can choose which of your data to export, the separator you wish to use, and whether you'd like to download the result to your computer or upload it into a Google Sheet.

On the **Content** tab, you can drag and drop the columns appearing in the column list to reorder the output. The options for reconciled and date data are applied to each column individually.

This exporter is especially useful with reconciled data, as you can choose whether you wish to output the cells' original values, the matched values, or the matched IDs. Ouputting

“match entity's name”, “matched entity's ID”, or “cell's content” will output, respectively, the contents of `cell.recon.match.name`, `cell.recon.match.id`, and `cell.value`.

“Output nothing for unmatched cells” will export empty cells for both newly-created matches and cells with no chosen matches. “Link to matched entity's page” will produce hyperlinked text in an HTML table output, but have no effect in other formats.

At this time, the date-formatting options in this window do not work. You can [keep track of this issue on Github](#). In the future, you will be able to choose how to [output date-formatted cells](#). You can create a custom date output by using [formatting according to the SimpleDateFormat parsing key found here](#).

Custom Tabular Exporter

Content Download Upload Option Code

Line-based text formats

Tab-separated values (TSV)
 Comma-separated values (CSV)
 Custom separator :

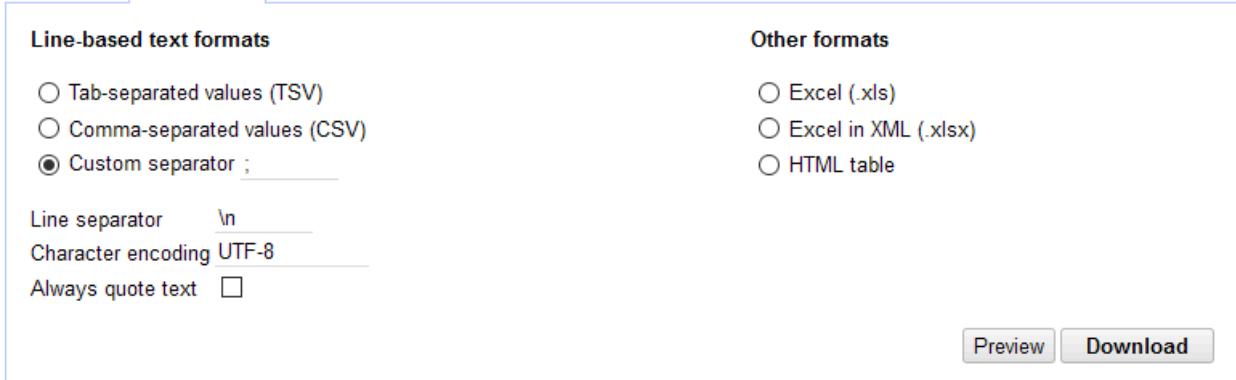
Line separator
Character encoding
Always quote text

Other formats

Excel (.xls)
 Excel in XML (.xlsx)
 HTML table

Preview **Download**

Cancel



On the **Download** tab, you can generate a preview of how the first ten rows of your dataset will output. If you do not choose one of the file formats on the right, the **Download** button will generate a text file. On the **Upload** tab, you can create a new Google Sheet.

With the **Option Code** tab, you can copy JSON of your current custom settings to reuse on another export, or you can paste in existing JSON settings to apply to the current project.

SQL exporter

The SQL exporter creates a SQL statement containing the data you've exported, which you can use to overwrite or add to an existing database. Choosing [Export](#) → [SQL exporter](#) will bring up a window with two tabs: one to define what data to output, and another to modify other aspects of the SQL statement, with options to preview and download the statement.

SQL Exporter

[Content](#) [Download](#)

Field Name	SQL Type	Size	Allow Null	Default
<input checked="" type="checkbox"/> Author	TEXT		Apply All	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Title	TEXT		Apply All	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Publication year	NUMERIC		Apply All	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Literary genre	TEXT		Apply All	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Genre ID	VARCHAR		Apply All	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Country	VARCHAR		Apply All	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> coordinate location	TEXT INT NUMERIC CHAR DATE TIMESTAMP		Apply All	<input checked="" type="checkbox"/>

Output empty row (i.e. facets and filters and export all) Trim Column Names

[Select All](#) [De-select All](#)

[Cancel](#)

The [Content](#) tab allows you to craft your dataset into an SQL table. From here, you can choose which columns to export, the data type to export for each (or choose "VARCHAR"), and the maximum character length for each field (if applicable based on the data type). You can set a default value for empty cells after unchecking "Allow null" in one or more columns.

With this output tool, you can choose whether to output only currently visible rows, or all the rows in your dataset, as well as whether to include empty rows. The option to "Trim column names" will remove their whitespace characters.

SQL Exporter

The screenshot shows the 'SQL Exporter' interface. At the top, there are two tabs: 'Content' and 'Download'. The 'Download' tab is highlighted with a blue border. Below the tabs, there is a form with the following fields:

- Table Name:** Novels
- Checkboxes (under 'Include Schema'):**
 - Include Schema
 - Include Drop Statement
 - Include 'IF EXISTS' in DROP statement
- Checkboxes (under 'Include Content'):**
 - Include Content
 - Convert null value to NULL in INSERT
- Buttons:** Preview (disabled) and Download

The **Download** tab allows you to finalize your complete SQL statement.

Include schema means that you will start your statement with the creation of a table. Without that, you will only have an INSERT statement.

Include content means including the INSERT statement with data from your project. Without that, you will only create empty columns.

You can include DROP and IF EXISTS if you require them, and set a name for the table to which the statement will refer.

You can then preview your statement, which will open up a new browser tab/window showing a statement with the first ten rows of your data (if included), or you can save a **.sql** file to your computer.

Templating exporter

If you pick **Templating...** from the **Export** dropdown menu, you can “roll your own” exporter. This is useful for formats that we don't support natively yet, or won't support. The Templating exporter generates JSON by default.

Templating Export

The screenshot shows the 'Templating Export' window with several sections:

- Prefix:** A code editor containing a JSON snippet starting with '{ "rows" : [...]'.
- Row Template:** A code editor containing a JSON snippet with a complex template for generating rows from table cells. It includes variables like 'Author', 'Title', 'Publication year', 'Literary genre', 'Genre ID', 'Country', 'coordinate location', 'continent', and 'Reference'.
- Row Separator:** A code editor containing a single character, a comma (',').
- Suffix:** A code editor containing a JSON snippet ending with ']'.
- Output Preview:** A large code editor on the right showing three examples of the generated JSON output based on the row template.
- Buttons at the bottom:** 'Reset Template', 'Export', and 'Cancel'.

The Templatting Export window allows you to set your own separators, prefix, and suffix to create a complete dataset in the language of your choice. In the **Row template** section, you can choose which columns to generate from each row by calling them with **variables**.

This can be used to:

- output **reconciliation data**, such as `cells["ColumnName"].recon.match.name`
- create multiple columns of output from different **member fields** of a single project column
- employ **expressions** to modify data for output: for example,
`cells["ColumnName"].value.toUpperCase()`.

Anything that appears inside doubled curly braces `({{ }})` is treated as a GREL

expression; anything outside is generated as straight text. You can use Jython or Clojure by declaring it at the start:

```
{jython:return cells["ColumnName"].value}
```

 CAUTION

Note that some syntax is different in this tool than elsewhere in OpenRefine: a forward slash or a closing curly brace () must be escaped with a backslash, while other characters do not need escaping.

You can include [regular expressions](#) as usual (inside forward slashes, with any GREL function that accepts them). For example, you could output a version of your cells with punctuation removed, using an expression such as

```
{jsonize(cells["ColumnName"].value.replaceChars("/[.!?$&,/]/", ""))}
```

You could also simply output a plain-text document inserting data from your project into sentences: for example, "In `{cells["Year"].value}` we received `{cells["RequestCount"].value}` requests."

You can use the shorthand `${ColumnName}` (no need for quotes) to insert column values directly. You cannot use this inside an expression, because of the closing curly brace.

If your project is in records mode, the `Row separator` field will insert a separator between records, rather than individual rows. Rows inside a single record will be directly appended to one another as per the content in the `Row Template` field.

Once you have created your template, you may wish to save the text you produced in each field, in order to reuse it in the future. Once you click `Export` OpenRefine will output a simple `.txt` file, and your template will be discarded.

We have recipes on using the Templating exporter to [produce several different formats](#).

Export a project

You can share a project in progress with another computer, a colleague, or with someone who wants to check your history. This can be useful for showing that your data cleanup didn't distort or manipulate the information in any way. Once you have exported a project, another OpenRefine installation can [import it as a new project](#).

You can either save it locally or upload it to Google Drive (which requires you to authorize a Google account).

CAUTION

OpenRefine project archives contain confidential data from previous steps, which will still be accessible to anyone who has the archive. If you are hoping to keep your original dataset hidden for privacy reasons, such as using OpenRefine to anonymize information, do not share your project archive.

To save your project archive locally: from the [Export](#) dropdown, select [OpenRefine project archive to file](#). OpenRefine exports your full project with all of its history. It does not export any current views or applied facets. Existing reconciliation information will be preserved, but the importing computer will need to add the same reconciliation services to keep working with that data.

OpenRefine exports files in [.tar.gz](#) format. You can rename the file when you save it; otherwise it will bear the project name.

To save your project archive to Google Drive: from the [Export](#) dropdown, select [OpenRefine project archive to Google Drive...](#). OpenRefine will not share the link with you, only confirm that the file was uploaded.

Export operations

You can [save and re-apply the history of any project](#) (all the operations shown in the Undo/Redo tab). This creates JSON that you can save for later reuse on another OpenRefine project.

Troubleshooting

Frequently asked questions

We collect and share FAQs and responses on Github at <https://github.com/OpenRefine/OpenRefine/wiki/FAQ>.

If you don't find your problem and solution there, continue on to the resources in the Community section below to see more conversations and look for solutions.

Community

If you're having a problem:

- Search the web to see if this is a known issue: it could be mentioned in the [forum](#), on our old mailing lists ([openrefine](#) and [openrefine-dev](#)), on [Stack Overflow](#)
- Report an issue:
 - Either as a new thread (conversation) in the [forum](#).
 - Or directly as [a Github issue](#).

If you want to contribute:

- [Help us translate the tool into more languages](#), using Weblate
- [We have a guide to contributing](#) in the technical section of our docs
- Contribute your feature requests in the [forum](#) or as [Github issues](#)
- Add your [blog posts, guides, tips, tricks, tutorials](#) to our list

- Keep an eye out for and respond to our biennial user survey.

Getting started

There are many ways you can contribute to OpenRefine. Choose which one fits you best and follow our guides to get started.

Code

Fix a bug or implement new functionality in the tool

Documentation

Improve the manual or develop new training material

Design

Propose new interfaces and make OpenRefine more usable

Translations

Make OpenRefine available in your language

Reporting and tracking issues

If you need to file a bug or request a feature, [create an Issue in the OpenRefine Github repository](#). Github issues should be used for reporting specific bugs and requesting specific features. If you just don't know how to do something using OpenRefine, or want to discuss some ideas, please:

- [Try the user manual](#)
- [Ask for help on our forum](#)

Why get involved

OpenRefine is a powerful open-source tool for cleaning, transforming, and enriching data. Originally developed by Google, it was released open source in 2013 and is now maintained by a diverse international community. Users like journalists, librarians and researchers value it for its powerful capabilities to work with messy data.

Design plays an important role in OpenRefine's popularity as its web-based interface is the main point of interaction between users and their data. They load datasets into the tool, where they can explore the data, identify issues, and apply transformations. Key OpenRefine features:

Key OpenRefine features:

- Open-source codebase: it is freely available for anyone to use and modify. This enables the tool to evolve continuously, benefiting from the contributions of developers, data enthusiasts, and designers across the globe.
- Capacity to streamline complex data transformation tasks: OpenRefine allows users to perform a wide array of operations – ranging from Complex data transformation tasks without requiring extensive programming skills: From basic formatting, filtering and sorting, to advanced data cleaning and connecting across heterogeneous sources, all via the user friendly interface.
- Empowering individuals who may not have extensive technical backgrounds to work confidently with data: the tool's accessibility reduces dependency on specialized data professionals and accelerates insights and decision-making.
- Educational resource: an extensive range of tutorials and learning materials online introduce newcomers to the concepts of data cleaning, transformation, and data

quality management.

- Supportive environment for knowledge exchange, troubleshooting, and collaborative problem-solving: this fosters a sense of belonging and encourages continuous learning.

Diverse user communities

- **Data analysts and scientists** Data analysts and scientists often use OpenRefine to preprocess and clean datasets before conducting in-depth analyses. They leverage its features to identify anomalies, correct errors, and ensure data consistency, enabling them to generate more accurate insights.
- **Data engineers** Data engineers use OpenRefine to transform and prepare raw data for downstream processes. They perform data normalization, standardization, and enrichment to ensure that data is well-structured and ready for integration into databases or data pipeline
- **Researchers** Researchers across various domains use OpenRefine to clean and prepare data for academic studies and research projects. It allows them to focus on the core aspects of their research rather than getting bogged down by data quality issues.
- **Librarians and archivists** OpenRefine is valuable for librarians and archivists who work with large collections of data, such as catalog records or historical documents. It helps them clean, categorize, and enrich metadata, making it easier to organize and retrieve information.
- **Business analysts** Business analysts leverage OpenRefine to process and transform datasets for business intelligence purposes. They ensure data accuracy and consistency, enabling more reliable decision-making within organizations.

- **Journalists** Investigative journalists use OpenRefine to clean and analyze datasets relevant to their stories. It helps them uncover patterns, discrepancies, and insights that contribute to impactful news reporting.
- **Non-technical professionals** OpenRefine's user-friendly interface makes it accessible to individuals who may not have strong technical skills. Marketing professionals, for instance, can clean and prepare customer data for targeted campaigns without needing programming expertise.
- **Educators** OpenRefine serves as an educational tool for teaching data cleaning, transformation, and data quality concepts. Educators can introduce students to real-world data challenges and provide hands-on experience in managing messy datasets.

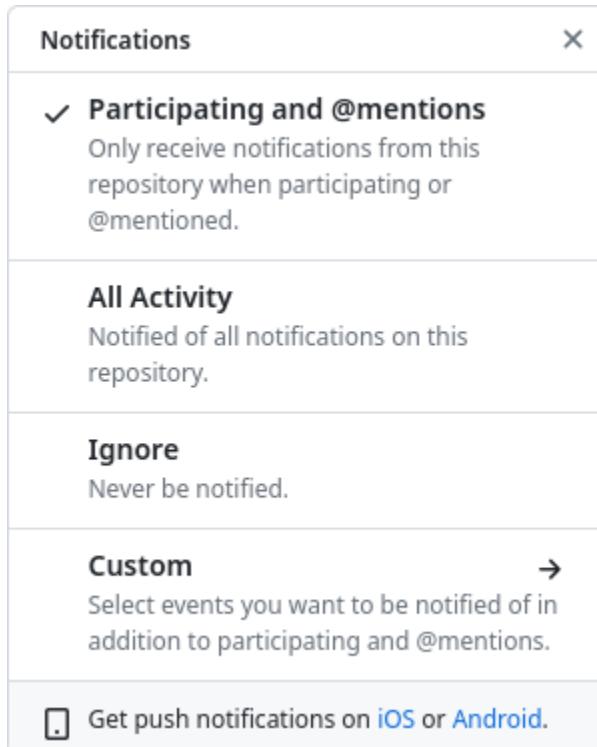
Communicating via GitHub

We recommend new members of OpenRefine's GitHub organization to apply some GitHub configuration settings, to help them integrate in the project smoothly.

Notifications

As a member of the organization, GitHub subscribes you automatically to a lot of notifications about everything that is happening in the project. For most people, this will generate much more noise than desired.

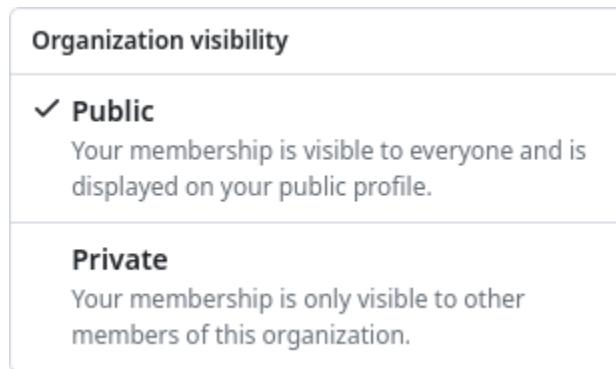
Go to [the OpenRefine repository](#), click on "Unwatch" and pick "Participating and @mentions". This will ensure that you only get notifications in discussions in which you left a comment, or if you were explicitly pinged.



We encourage you to do the same for any other repository which generates unwanted notifications in your feed. This will help you ensure your GitHub notifications stay relevant and help you get involved in OpenRefine, focusing on the topics that you care about.

Publicizing your membership

By default, your membership of the OpenRefine project is kept hidden. We encourage you to make it public by going to [the list of members of the OpenRefine organization](#), looking for your own user account there, and switching from "Private" to "Public".



This will include you in the list of project members and display an OpenRefine badge on your GitHub profile:

Organizations



Block or Report

Code contributions

Your first code pull request

This describes the overall steps to your first code contribution in OpenRefine. If you have trouble with any of these steps feel free to reach out on the [developer forum](#) or the [Gitter channel](#).

First, why do you want to contribute to OpenRefine?

- Are you an OpenRefine user who wants to fix or improve an aspect of the tool? Great! Before you dive in and make your changes, make sure you discuss what you want to change with the community first. Check if there is a [GitHub issue](#) on this topic already. If there is one, is there consensus around it? If not, it would be worth opening one. You are also welcome to discuss your plans in the [developer forum](#).
- Do you want to contribute to an open source project and picked OpenRefine for that? Great too! It is useful if you first get a sense of what the tool does. [Install OpenRefine](#) and learn to use it by following [some tutorials](#) or watching [some videos](#). Once you have a basic understanding of the use cases around OpenRefine, browse the [list of issues](#) to find an one that you find interesting. You should pick one where you understand what the problem is as a user, you can see why fixing it would be an improvement to the tool. It is also a good idea to pick an issue that matches your technical skills: some require work on the backend (in Java) or on the frontend (Javascript), or sometimes both. We try to maintain a list of [good first issues](#) which should be easier than others and should not require any difficult design decision.

Set up the development environment

Once you know what to work on, you can set up a development environment so that you

can make changes to OpenRefine and test them. We have [detailed instructions for this](#).

If you want to tackle an existing bug, try to reproduce this bug in your development environment. You might need to locate a particular dialog, use a specific importer on a sample file, or follow any other user workflow. If you have followed all the steps described in the issue and cannot observe the bug mentioned, write a comment on the issue explaining that you are not able to reproduce it (perhaps it was fixed by another change).

Locating the relevant part of the code to change

OpenRefine being a web app, it consists of a back-end (written in Java) and a front-end (written in Javascript using jQuery). Depending on the issue you want to tackle, you might need to make changes to either part, or sometimes both. You do not need to be familiar with the directory structure of OpenRefine to locate which files to work on: you can often use text search across files to find out what to edit.

For instance, suppose the issue you want to solve is about a dialog entitled "Columnize by key/values". The "columnize" term is likely rather specific and likely to be only used in this dialog or related menu entries, so you can search for this word in the entire repository. You can do so with [GitHub's search engine](#) or locally in your git clone, for instance with `grep -iR "columnize" .` (to search in file contents) or `find . | grep columnize` (to search in file names) on UNIX systems.

This will give you the following sorts of results, as of January 2023:

- `main/webapp/modules/core/scripts/views/data-table/key-value-columnize.html`: This file stores the HTML source of the dialog which configures the operation. This would be the place to change the layout of the dialog or add a button inside it, for instance.
- `main/webapp/modules/core/scripts/views/data-table/menu-edit-cells.js`: This file defines the contents of the "Edit cells" menu in each column, where the columnize operation can be found. It also defines the logic of the corresponding

dialog.

- `main/webapp/modules/core/langs/translation-en.json`: This file defines the translations (here in English) for any text shown in the frontend, such as the menu item for the columnize operation, or text in its dialog.
- `main/tests/cypress/cypress/e2e/project/grid/column transpose/columnize.cy.js`: The files under `main/tests/cypress` are **integration tests written in the Cypress framework**: those describe a user interaction in the web-based UI and check that the expected results are obtained.
- `main/src/com/google/refine/operations/cell/KeyValueColumnizeOperation.java`: This file is part of the backend and implements the columnize operation itself. This is likely the file you will need to edit if you want to change the behaviour of the operation in some special cases.
- `main/tests/server/src/com/google/refine/operations/cell/KeyValueColumnizeTests.java`: Such a file defines a set of Java tests, used to check back-end functionality. This file checks that the operation behaves as expected in various scenarios.
- `main/resources/com/google/refine/operations/OperationDescription.properties`: Such a `.properties` file contains localized strings for the backend, in this case the description of the operation;
- `main/webapp/modules/core/MOD-INF/controller.js`: This file acts as a central registry which holds links to all the back-end and front-end components which make up the application. The columnize operation (implemented in the back-end) and the corresponding UI (implemented in the front-end) are all registered there.

Testing your changes

Once you have made changes to the source code, you should test them to make sure they work as expected. Manual testing is very useful for that, as it lets you judge the quality of the final user experience. On top of that, it is useful (and often required for your contribution to be accepted) to have automated tests. As mentioned above, those can

come in two forms:

- [Cypress integration tests](#), which should be used when you made changes to the frontend. Those tests are relatively slow to execute: therefore, consider adding assertions to an existing test which covers the area you are touching, if possible;
- Java unit tests, which should be used when you made changes to the backend. Those tests are faster to execute and generally more reliable. Most Java classes have a corresponding test class (as in the example above). In most cases you should be able to write your unit test by imitating existing tests from the same test class.

Submitting your changes

Once you have made sure your changes work as expected, you are ready to submit them for review.

- Create a git branch for your fix. The name of your branch should contain the issue number, and a few words to describe the topic of the fix, for instance "issue-1234-columnize-layout".
- If you made Java changes, run linting to make sure they conform to our code style, with `./refine lint`.
- Commit your changes, using a message that contains one of the special words "closes" and "fixes" which are detected by Github, followed by the issue number, e.g. "closes #1234" or "fixes #1234", this will link the commit to the issue you are working on.
- Push your branch to your fork and create a pull request for it, explaining the approach you have used, any design decisions you have made.

Your changes will be reviewed and we might suggest improvements to your pull request. You can submit those follow-up changes by adding more commits to your branch. Your commits will generally be squashed when merging the pull request. Thank you for your

contribution!

Translate OpenRefine's interface

Currently supported languages include English, Spanish, Chinese, French, Hebrew, Italian and Japanese.



You can help translate OpenRefine into your language by visiting [Weblate](#) which provides a web based UI to edit and add translations and sends automatic pull requests back to our project.

Click to help translate --> [Weblate](#)

Manual translation process

Localized strings are entered in a .json file, one per language. They are located in the folder `main/webapp/modules/core/langs/` in a file named `translation-xx.json`, where xx is the language code (i.e. fr for French).

Simple case of localized string

This is an example of a simple string, with the start of the JSON file. This example is for French.

```
{  
    "name": "Français",  
    "core-index/help": "Aide",  
    (... more lines)  
}
```

So the key `core-index/help` will render as `"Aide"` in French.

Localization with a parameterized value

In this example, the name of the column (represented by `$1` in this example), will be substituted with the string of the name of the column.

```
"core-facets/edit-facet-title": "Cliquez ici pour éditer le nom de la  
facette\nColonne : $1",
```

Localization with a singular/plural value

In this example, one of the parameter will have a different string depending if the value is 1 or another value. In this example, the string for page, the second parameter, `$2`, will have an « s » or not depending on the value of `$2`.

```
"core-views/goto-page": "$1 de $2 {{plural:$2|page|pages}}"
```

Front-end development

The OpenRefine front end has been localized using the [Wikidata jquery.i18n library](#). The localized text is stored in a JSON dictionary on the server and retrieved with a new OpenRefine command.

Adding a new string

There should be no hard-coded language strings in the HTML or JSON used for the front end. If you need a new string, first check the existing strings to make sure there isn't an equivalent string, **in an equivalent context**, that you can reuse. Context is important because it can affect how the same literal English text is translated. This cuts down on the amount of text which needs to be translated.

Strings should be entire sentences or phrases and should include substitution variables for any parameters. Do not concatenate strings in either Java or Javascript (or implicitly by laying them out in a specific order). So, instead of `"You have " + count + "` `row(s)"` (or worse `count != 1 ? " rows" : " row"`), internationalize everything together so that it can be translated taking into account word ordering and plurals for different languages, ie `"You have $1 {{plural $1: row|rows}}"`, passing the parameter(s) into the `$.i18n` call.

If there's no string you can reuse, allocate an available key in the appropriate translation dictionary and add the default string, e.g.

```
...  
"section/newkey": "new default string for this key",  
...
```

and then set the text (or HTML) of your HTML element using i18n helper method. So given an HTML fragment like:

```
<label id="new-element-id">[untranslated text would have appeared  
here before]</label>
```

we could set its text using:

```
$( '#new-html-element-id' ).text($.i18n('section/newkey'));
```

or, if you need to embed HTML tags:

```
$( '#new-html-element-id' ).html($.i18n('section/newkey'));
```

Adding a new language

The language dictionaries are stored in the `langs` subdirectory for the module e.g.

- <https://github.com/OpenRefine/OpenRefine/tree/master/main/webapp/modules/core/langs> for the main interface
- <https://github.com/OpenRefine/OpenRefine/tree/master/extensions/gdata/module/langs> for google spreadsheet connection
- <https://github.com/OpenRefine/OpenRefine/tree/master/extensions/database/module/langs> for database via JDBC
- <https://github.com/OpenRefine/OpenRefine/tree/master/extensions/wikidata/module/langs> for Wikidata

To add support for a new language, the easiest way is to do it directly in Weblate. To do it manually, copy `translation-en.json` to `translation-<locale>.json` and have your translator translate all the value strings (ie right hand side).

Main interface

The translation is best done [with Weblate](#). Files are periodically merged by the developer team.

Run the latest (hopefully cloned from github) version and check whether translated words fit to the layout. Not all items can be translated word by word, especially into non-Indo-

European languages.

If you see any text which remains in English even when you have checked all items, please create bug report in the issue tracker so that the developers can fix it.

Extensions

Extensions can be translated via Weblate just like the core software.

The new extension for Wikidata contains lots of domain-specific concepts, with which you may not be familiar. The Wikidata may not have reconciliation service for your language. I recommend checking the glossary(<https://www.wikidata.org/wiki/Wikidata:Glossary>) to be consistent.

By default, the system tries to load the language file corresponding to the currently in-use browser language. To override this setting a new menu item ("Language Settings") has been added at the index page. To support a new language file, the developer should add a corresponding entry to the dropdown menu in this file: `/OpenRefine/main/webapp/modules/core/scripts/index/lang-settings-ui.html`. The entry should look like:

```
<option value="<locale>">[Language Label]</option>
```

Server-side localisation

Currently no back end functions are translated, so things like error messages, undo history, etc may appear in English form. Rather than sending raw error text to the front end, it's better to send an error code which is translated into text on the front end. This allows for multiple languages to be supported.

How to build, test and run

This page explains how to install the tools you need to run OpenRefine from source and develop. This consists of:

- A Unix/Linux shell environment or the Windows command line, that should be installed on your machine already;
- [OpenRefine's source code](#);
- a [Java Development Kit \(JDK\)](#) (version 11 or later);
- [Apache Maven](#);
- [Node.js and NPM](#) (version 16 or later).

Get OpenRefine's source code

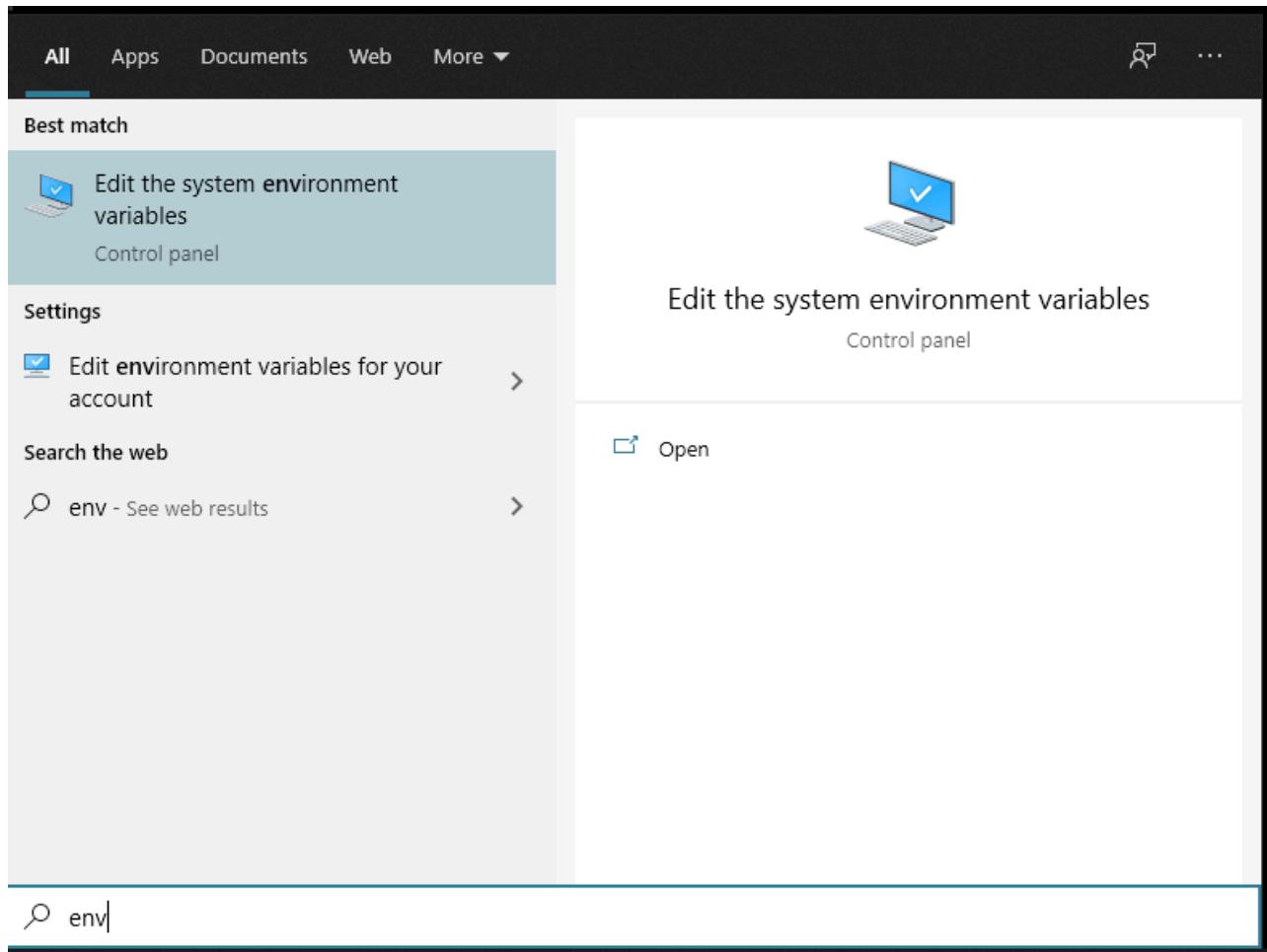
With Git installed, use the `git clone` command to download the [project's repo](#) to a directory of your choice.

Set up JDK

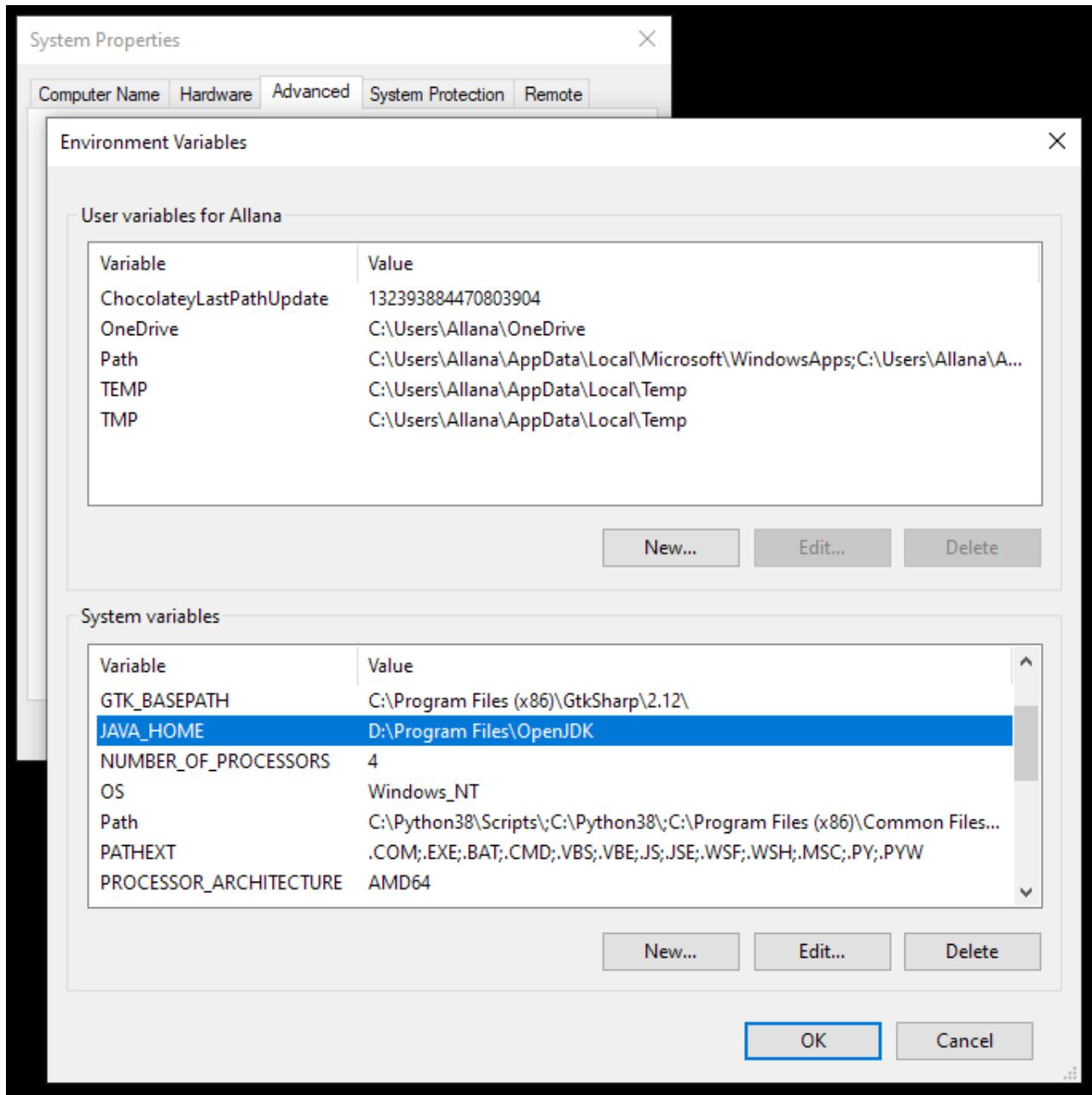
You must [install JDK](#) and set the `JAVA_HOME` environment variable (please ensure it points to the JDK, and not the JRE). OpenRefine is known to work with Java 11 to 21.

[Windows](#) [Mac](#) [Linux](#)

1. On Windows 10, click the Start Menu button, type `env`, and look at the search results. Click [Edit the system environment variables](#). (If you are using an earlier version of Windows, use the “Search” or “Search programs and files” box in the Start Menu.)



2. Click **Environment Variables...** at the bottom of the **Advanced** window.
3. In the **Environment Variables** window that appears, click **New...** and create a variable with the key **JAVA_HOME**. You can set the variable for only your user account, as in the screenshot below, or set it as a system variable - it will work either way.



- Set the **Value** to the folder where you installed JDK, in the format `D:\Programs\OpenJDK`. You can locate this folder with the **Browse directory...** button.

First, install Java. You can do so either with Homebrew, with `brew install java`, or by [downloading it from Adoptium and installing it manually](#).

You then need to make sure the `JAVA_HOME` environment is properly set, so that

OpenRefine can find your Java install.

Check the environment variable `JAVA_HOME` with:

```
$JAVA_HOME/bin/java --version
```

If this shows your Java version, your `JAVA_HOME` variable is set up correctly. If it shows an error, you need to adjust it. To do so, you can use:

```
export JAVA_HOME=$(/usr/libexec/java_home)"
```

Or, for Java 13.x:

```
export JAVA_HOME=$(/usr/libexec/java_home -v 13)"
```

On Debian/Ubuntu derivatives, enter the following:

```
sudo apt install default-jdk
```

On Fedora/CentOS, use:

```
sudo dnf install java-devel
```

On ArchLinux, use:

```
sudo pacman -S jdk-openjdk
```

For other distributions, search for any JDK in your package repository: most should be compatible with OpenRefine.

Maven

OpenRefine development requires Apache Maven for its build, test, and packaging processing. We encourage using the latest version of Apache Maven for development of OpenRefine, otherwise sometimes spurious errors appear in your IDE regarding POM, dependencies, or packages.

[Windows](#) [Mac](#) [Linux](#)

[Install Maven](#). Then ensure the `M2_HOME` or `MAVEN_HOME` environment variable is set or 'mvn' is in your system `PATH`:

```
MAVEN_HOME=E:\Downloads\apache-maven-3.8.4-bin\apache-maven-3.8.4\
```

Install Maven via Homebrew with `brew install maven`.

Otherwise, [Install Maven](#). Then ensure the `M2_HOME` or `MAVEN_HOME` environment variable is set or 'mvn' is in your system `PATH`:

```
MAVEN_HOME=/opt/apache-maven-3.8.7
```

Install Maven with the package manager of your Linux distribution. For instance:

- On Debian/Ubuntu derivatives, use `sudo apt install maven`
- On Fedora/CentOS, use `sudo dnf install maven`
- On ArchLinux, use `sudo pacman -S maven`

Other distributions are likely to offer Maven in their official package repository as well.

Node.js and npm

The OpenRefine webapp requires [Node.js](#) and npm to install package dependencies. We require Node.js 16 or newer. Download and install Node.js (On Windows, you can alternatively install [nvm](#) to easily manage multiple npm versions on your system). You should then have node and npm installed. You can check the versions by typing:

```
node -v  
npm -v
```

You can update the version of npm to the latest by typing

```
npm install -g npm@latest
```

Building

To see what functions are supported by OpenRefine's build system, type

```
./refine -h
```

To build the OpenRefine application from source type:

```
./refine clean  
./refine build
```

Note that the [refine](#) script is a wrapper over the Maven build system. You can often use Maven commands directly, but running some goals in isolation might fail (try adding the

`compile test-compile` goals in your invocation if that is the case).

Testing

Since OpenRefine is composed of two parts, a server and a in-browser UI, the testing system reflects that:

- on the server side, it's powered by [TestNG](#) and the unit tests are written in Java;
- on the client side, we use [Cypress](#) and the tests are written in Javascript

To run server tests, use:

```
./refine test
```

To run the Cypress tests for the first time, [you must go through the installation process..](#)

Then, you need to run two processes in parallel:

- OpenRefine itself, ideally running off a fresh workspace directory: `./refine -d /tmp/openrefine_workspace`
- Cypress, with the command `yarn --cwd ./main/tests/cypress run cypress open`

We recommend running only individual test suites locally and relying on our continuous integration infrastructure to run the entire test suite, as this is rather time consuming.

Running

From the top level directory in the OpenRefine application you can build, test and run OpenRefine using the `./refine` shell script (if you are working in a *nix shell), or using

the `refine.bat` script from the Windows command line. Note that the `refine.bat` on Windows only supports a subset of the functionality, supported by the `refine` shell script. The example commands below are using the `./refine` shell script, and you will need to use `refine.bat` if you are working from the Windows command line.

To run OpenRefine from the command line (assuming you have been able to build from the source code successfully)

```
./refine
```

By default, OpenRefine will use `refine.ini` for configuration. You can copy it and rename it to `refine-dev.ini`, which will be used for configuration instead. `refine-dev.ini` won't be tracked by Git, so feel free to put your custom configurations into it.

If you wish to run the application manually, without using the `refine` script, you can do so via Maven with `mvn exec:java`. The entry point of the application is the `com.google.refine.Refine` class.

Building distributions (packaged versions)

The Refine build system uses Apache Maven to automate the creation of the installation packages for the different operating systems. The packages are currently optimized to run on Mac OS X which is the only platform capable of creating the packages for all three OS that we support.

To build the distributions type

```
./refine dist <version>
```

where 'version' is the release version.

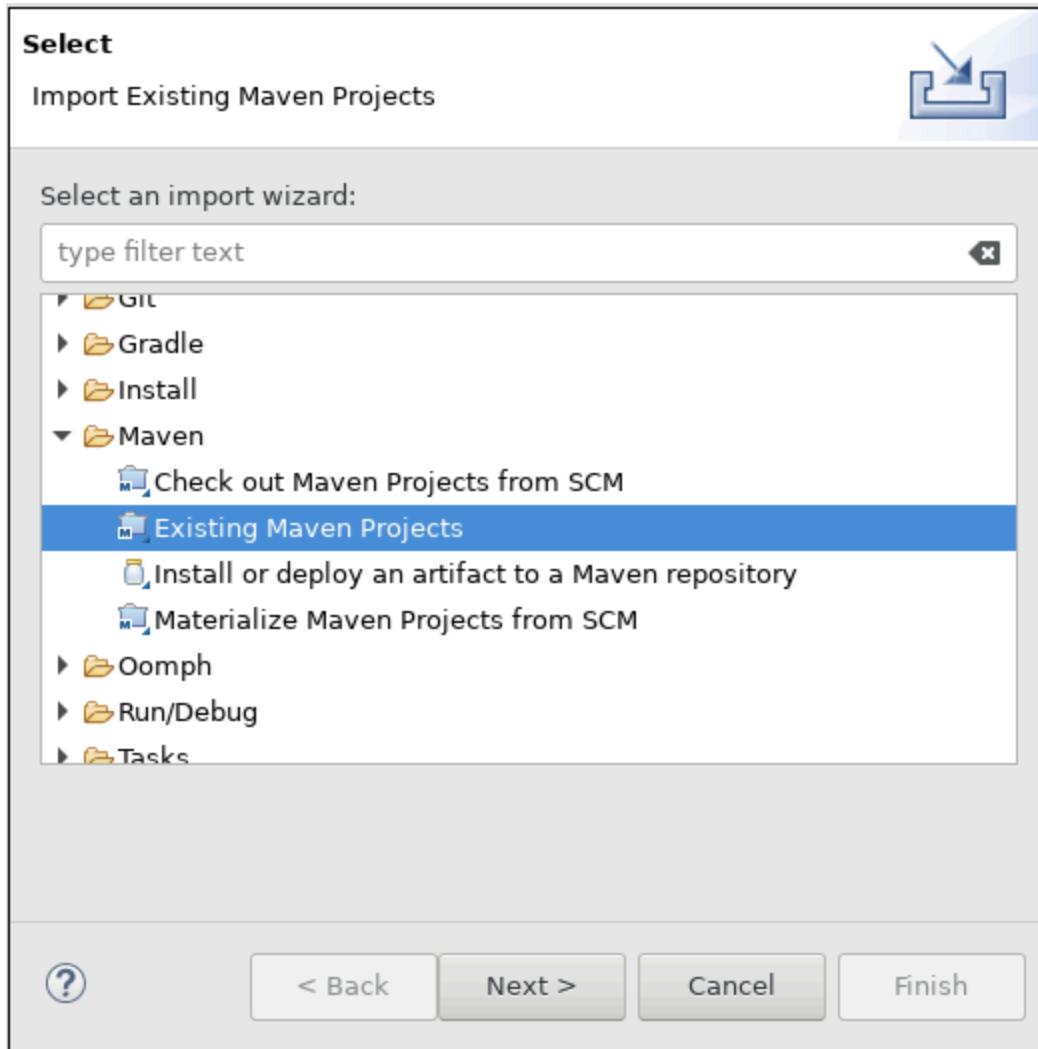
Developing with Eclipse

'OpenRefine' source comes with Maven configuration files which are recognized by [Eclipse](#) if the Eclipse Maven plugin (m2e) is installed.

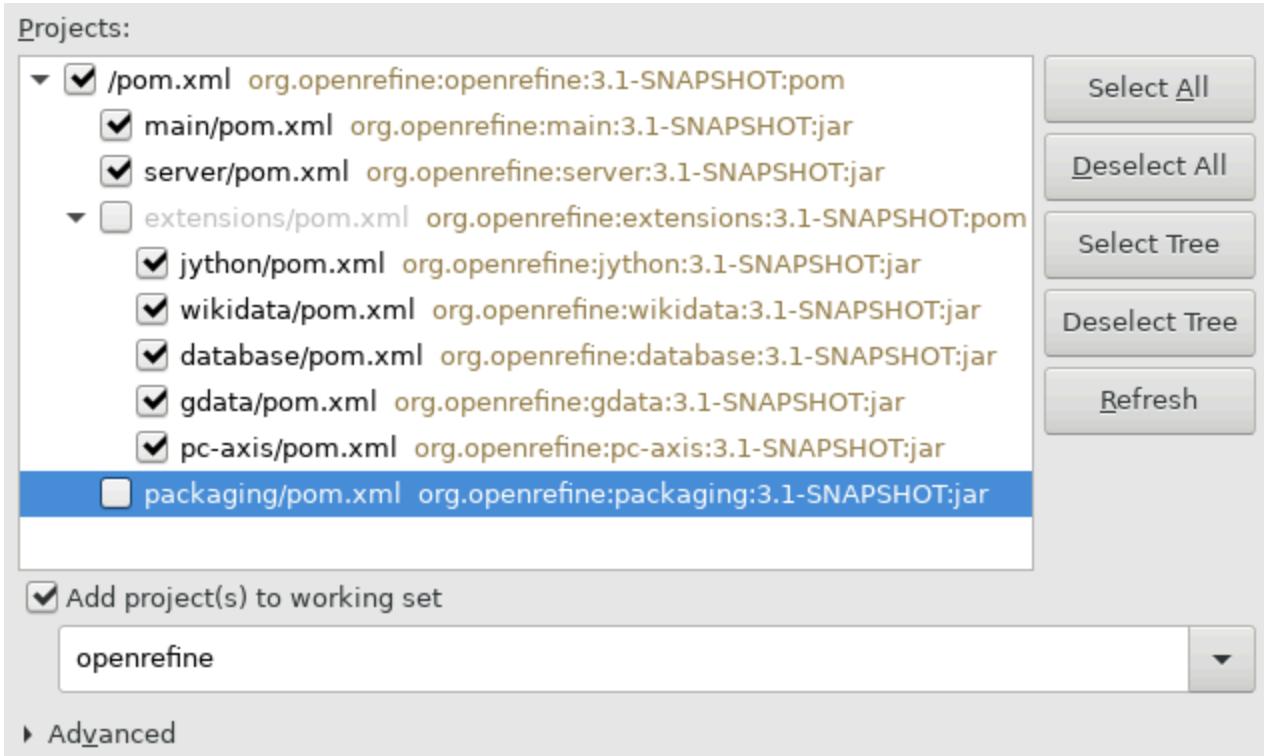
At the command line, go to a directory **not** under your Eclipse workspace directory and check out the source:

```
git clone https://github.com/OpenRefine/OpenRefine.git
```

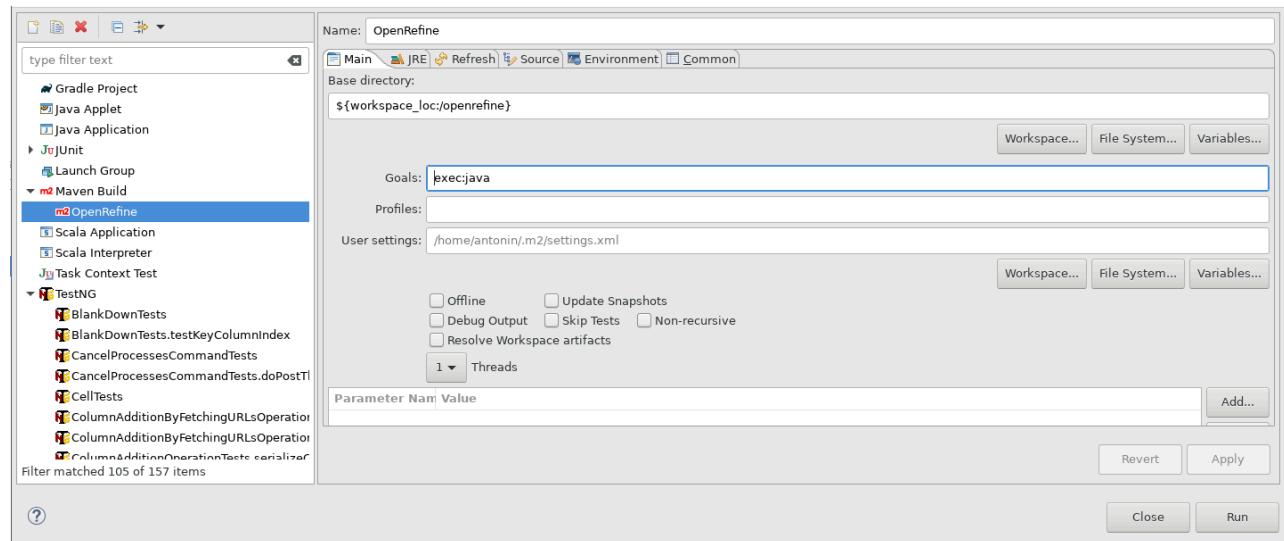
In Eclipse, invoke the [Import...](#) command and select [Existing Maven Projects](#).



Choose the root directory of your clone of the repository. You get to choose which modules of the project will be imported. You can safely leave out the `packaging` module which is only used to generate the Linux, Windows and MacOS distributions.



To run and debug OpenRefine from Eclipse, you will need to add an execution configuration on the `server` sub-project. Right click on the `server` subproject, click `Run as...` and `Run configurations...` and create a new `Maven Build` run configuration. Rename the run configuration `OpenRefine`. Enter the root directory of the project as `Base directory` and use `exec:java` as a Maven goal.



This will add a run configuration that you can then use to run OpenRefine from Eclipse.

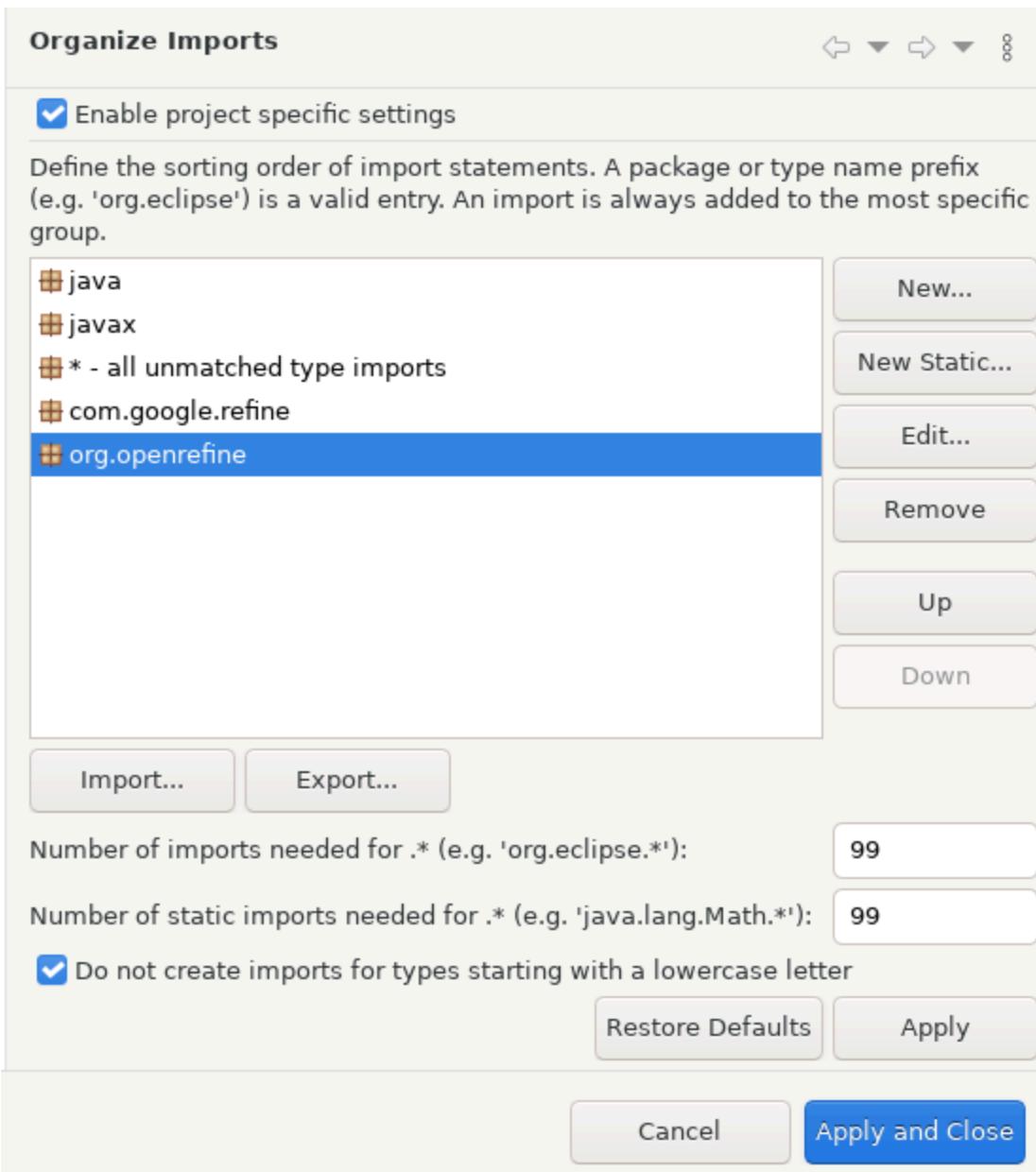
Code style

You can apply the supplied Eclipse code style (in `IDEs/eclipse/Refine.style.xml`) to make sure Eclipse lints your files according to the existing style. To do so, go to `Window -> Preferences -> Java Code Style -> Formatter` menu and enable a project-specific formatter that you can import from the XML file.

You can also configure Eclipse to sort `import` statements according to our conventions, by going to the `Window -> Preferences -> Java -> Code Style -> Organize imports` menu and enabling project-specific import order:

- `java`
- `javax`
- `*`
- `com.google.refine`
- `org.openrefine`

The dialog should look as follows:



Pull requests deviating from this style will fail in the CI.

You can manually apply the code style (regardless of your IDE) with the [./refine lint](#) command (or [refine.bat lint](#) on Windows).

Testing

You can run the server tests directly from Eclipse. To do that you need to have the TestNG launcher plugin installed, as well as the TestNG M2E plugin (for integration with Maven). If you don't have it, you can get it by [installing new software](#) from this update URL <https://testng.org/doc/download.html>

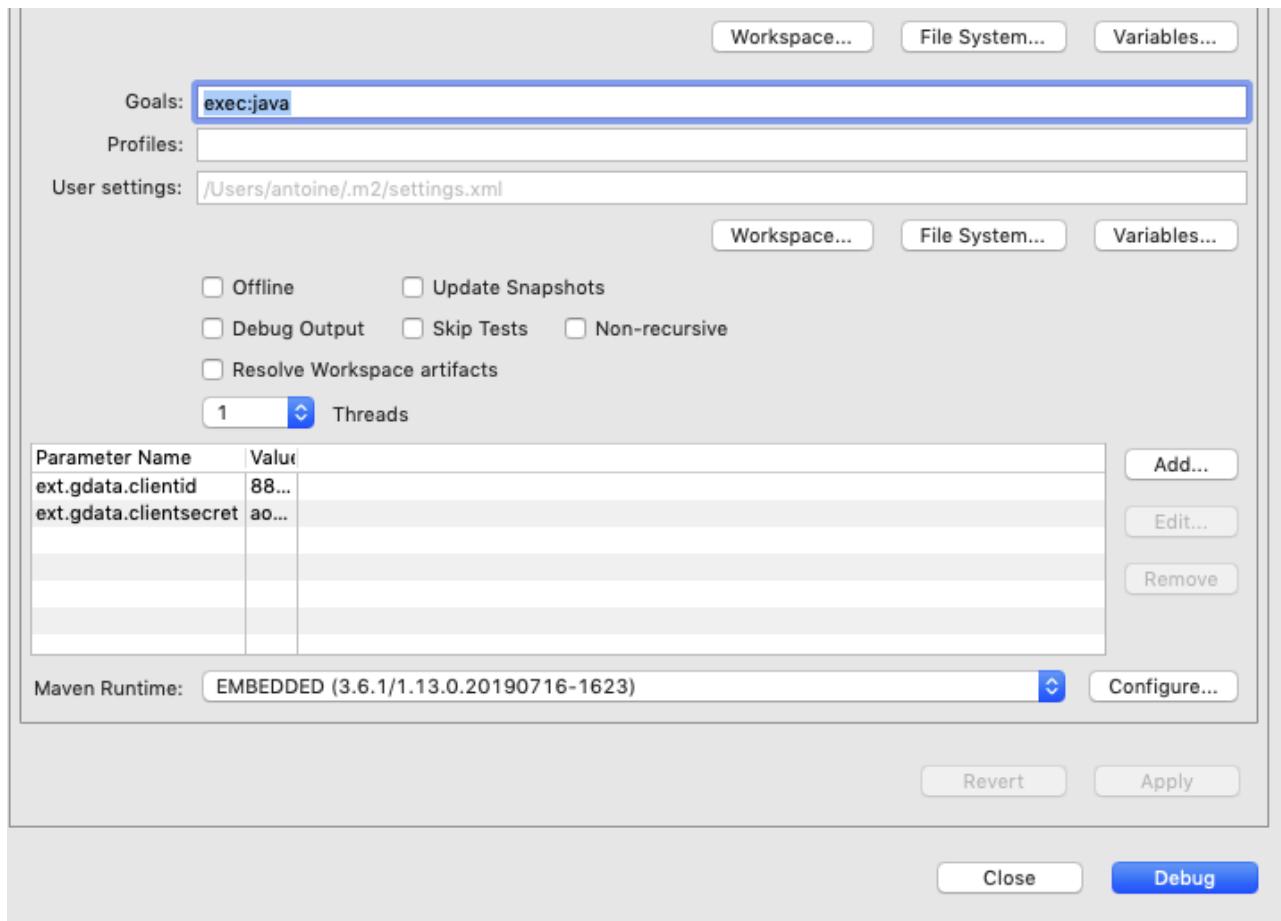
Once the TestNG launching plugin is installed in your Eclipse, right click on the source folder "main/tests/server/src", select `Run As` -> `TestNG Test`. This should open a new tab with the TestNG launcher running the OpenRefine tests.

Test coverage

It is possible to analyze test coverage in Eclipse with the `EclEmma Java Code Coverage` plugin. It will add a `Coverage as...` menu similar to the `Run as...` and `Debug as...` menus which will then display the covered and missed lines in the source editor.

Debugging

Here's an example of putting configuration in Eclipse for debugging, like putting values for the Google Data extension. Other type of configurations that can be set are memory, Wikidata login information and more.

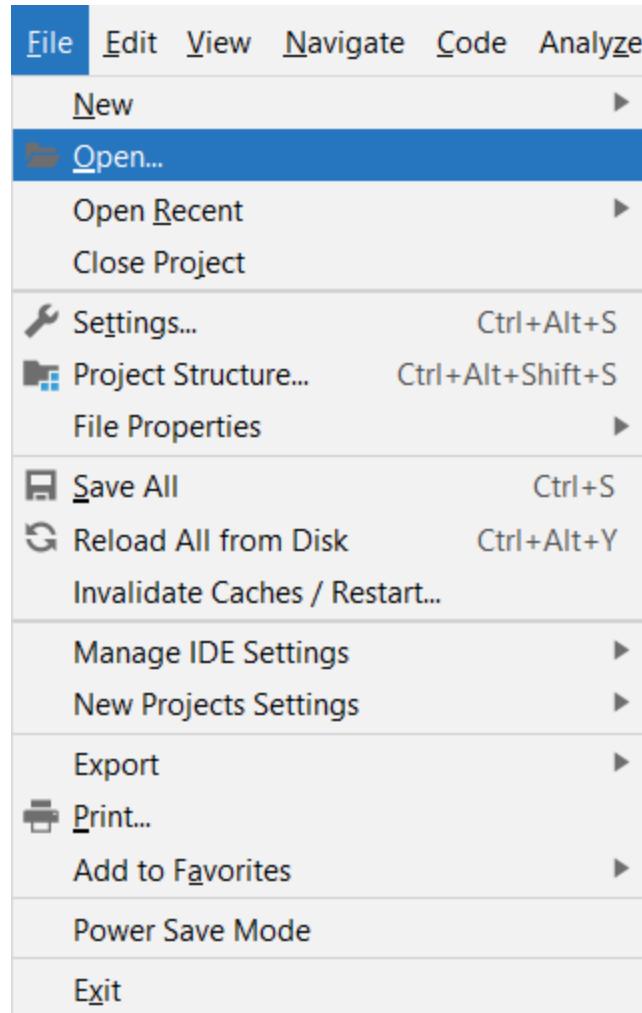


Developing with IntelliJ IDEA

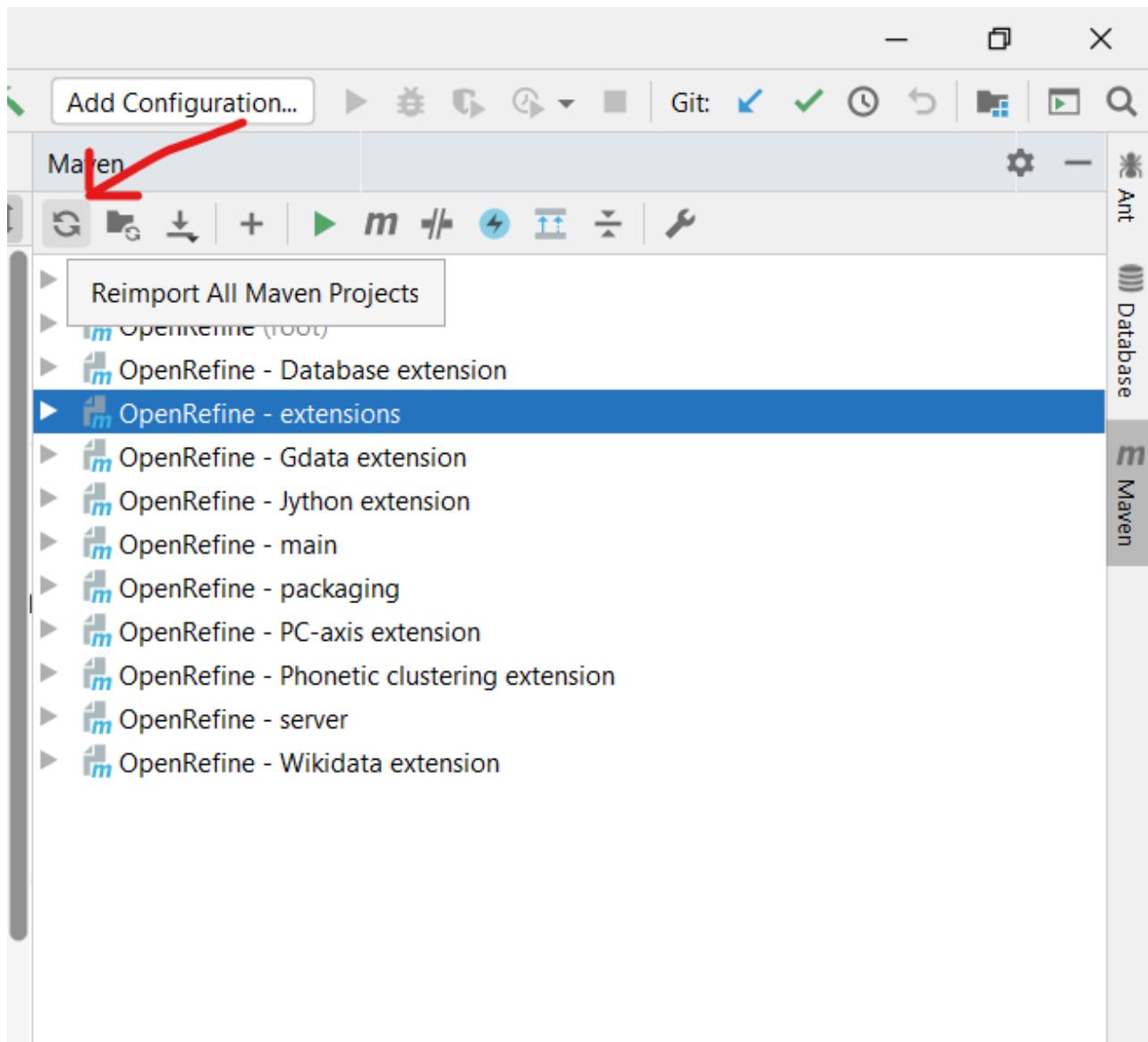
At the command line, go to a directory you want to save the OpenRefine project and execute the following command to clone the repository:

```
git clone https://github.com/OpenRefine/OpenRefine.git
```

Then, open the IntelliJ idea and go to [file -> open](#) and select the location of the cloned repository.



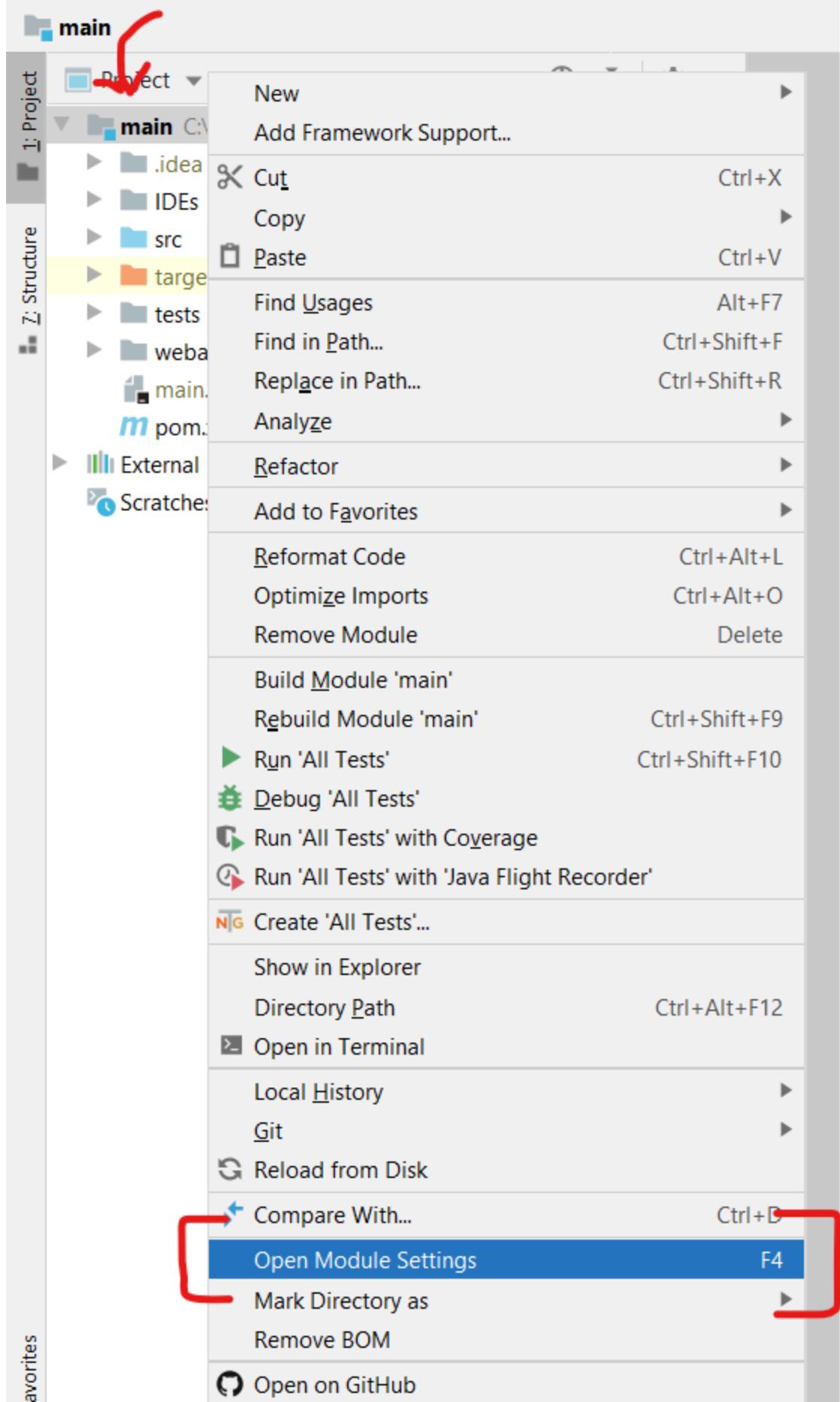
It will prompt you to add as a maven project as the source code contains a pom.xml file in it. Allow `auto-import` so that it can add it as a maven project. If it doesn't prompt something like this then you can go on the right side of the IDE and click on maven then, click on `reimport all the maven projects` that will add all the dependencies and jar files required for the project.



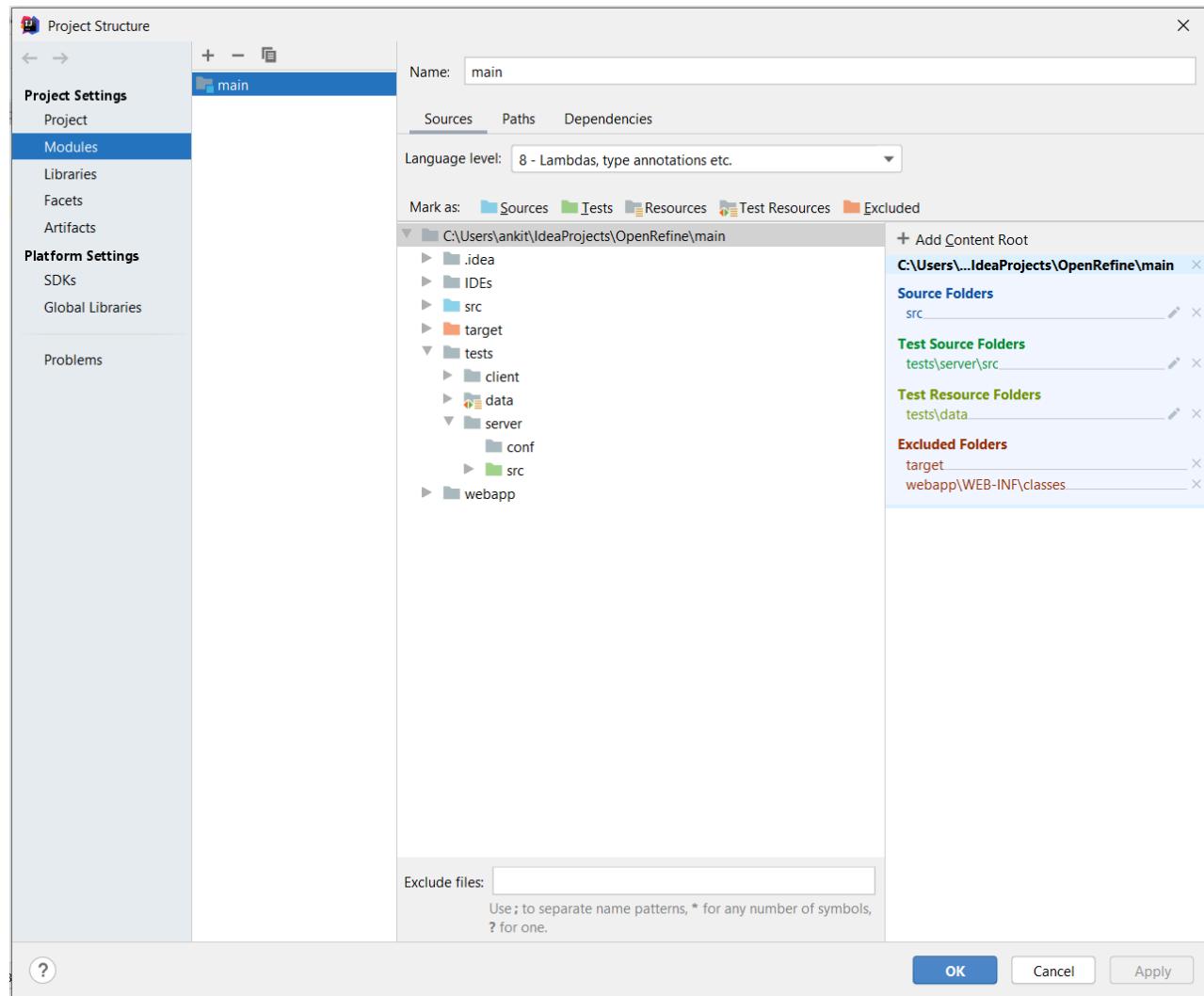
After this, you will be able to properly build, test, and run the OpenRefine project from the terminal. But if you will go to any of the test folders and open some file it will show you some import errors because the project isn't yet set up at the module level.

For removing those errors, and enjoying the features of the IDE like **ctrl + click**, etc you need to set up the project at the module level too. Open the different modules like `extensions/wikidata`, `main` as a project in the IDE. Then, right-click on the project folder and open the module settings.

File Edit View Navigate Code Analyze Refactor Build Run Tools



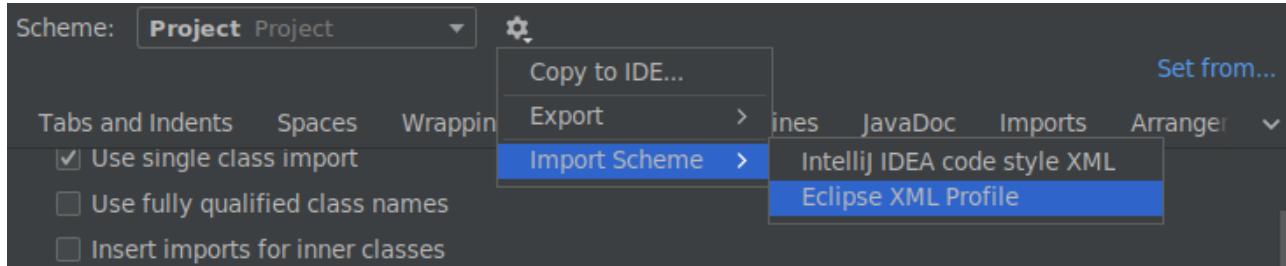
In the module settings, add the source folder and test source folders of that module.



Then, do the same thing for the main OpenRefine project and now you are good to go.

Code style

You can set up IntelliJ to follow the style conventions we use in OpenRefine, as [IntelliJ is able to import Eclipse style files](#). Go to [Settings -> Editor -> Code style -> Java](#) and import the style configuration file as follows:



The style file is located at [IDEs/eclipse/Refine.style.xml](#) in the repository. Note that this won't configure import ordering since this isn't included in the Eclipse code style export.

You can also configure the import order for the OpenRefine to follow the following order:

- `java`
- `javax`
- `*`
- `com.google.refine`
- `org.openrefine`

And disable the use of star imports by setting the thresholds for their activation at 99.

Use fully qualified class names in JavaDoc: ▾

Class count to use import with '*':

Names count to use static import with '*':

Packages to Use Import with '*'

+ -

Static	Package	With Subpackages
Nothing to show		

Import Layout

Layout static imports separately

+ - ▲ ▼

Static	Package	With Subpackages
<code>import static all other imports</code>		
<blank line>		
<input type="checkbox"/>	<code>import java.*</code>	<input checked="" type="checkbox"/>
<blank line>		
<input type="checkbox"/>	<code>import javax.*</code>	<input checked="" type="checkbox"/>
<blank line>		

To automatically format the code from the command-line, you can also use [./refine](#) [lint](#) (or `refine.bat lint` on Windows).

Functional tests

Introduction

OpenRefine's web interface is tested with the [Cypress framework](#). With Cypress, tests are performing assertions using a real browser, the same way a real user would use the software. Those are "end to end" (e2e) tests because they also rely on OpenRefine's backend (server).

Cypress tests can be ran

- using the Cypress test runner (development mode)
- using a command line (CI/CD mode)

If you are writing tests, the Cypress test runner is good enough, and the command-line is mainly used by the CI/CD platform (Github actions)

Cypress brief overview

Cypress operates insides a browser, it's internally using NodeJS. That's a key difference with tools such as Selenium.

From the Cypress documentation:

But what this also means is that your test code **is being evaluated inside the browser**. Test code is not evaluated in Node, or any other server side language. The **only** language we will ever support is the language of the web: JavaScript.

Good starting points with Cypress are the [Getting started guide](#), and the [Trade-offs](#)

The general workflow of a Cypress test is to

- Start a browser (yarn run cypress open)
- Visit a URL
- Trigger user actions
- Assert that the DOM contains expected texts and elements using selectors

Getting started

If this is the first time you use Cypress, it is recommended for you to get familiar with the tool.

- [Cypress overview](#)
- [Cypress examples of tests and syntax](#)

1. Install Cypress

You will need:

- [Node.js 10 or 12 and above](#)
- [Yarn or NPM](#)
- A Unix/Linux shell environment or the Windows command line

To install Cypress and dependencies, run :

```
cd ./main/tests/cypress  
yarn install
```

2. Start the test runner

The test runner assumes that OpenRefine is up and running on the local machine, the tests themselves do not launch OpenRefine, nor restarts it.

Start OpenRefine with

```
./refine
```

Then start Cypress

```
yarn --cwd ./main/tests/cypress run cypress open
```

3. Run the existing tests

Once the test runner is up, you can choose to run one or several tests by selecting them from the interface.

Click on one of them and the test will start.

4. Add your first test

- Add a `test.spec.js` into the `main/tests/cypress/cypress/e2e` folder.
- The test is instantly available in the list
- Click on the test
- Start to add some code

Tests technical documentation

A typical test

A typical OpenRefine test starts with the following code

```
it('Ensure cells are blanked down', function () {
  cy.loadAndVisitProject('food.mini');
  cy.get('.viewpanel-sorting a').contains('Sort').click();
  cy.get('.viewpanel').should('to.contain', 'Something');
});
```

The first noticeable thing about a test is the description (`Ensure cells are blanked down`), which describes what the test is doing.

Lines usually starts with `cy.something...`, which is the main way to interact with the Cypress framework.

A few examples:

- `cy.get('a.my-class')` will retrieve the `` element
- `cy.click()` will click on the element
- eventually, `cy.should()` will perform an assertion, for example that the element contains an expected text with `cy.should('to.contains', 'my text')`

On top of that, OpenRefine contributors have added some functions for common OpenRefine interactions. For example

- `cy.loadAndVisitProject` will create a fresh project in OpenRefine
- `cy.assertCellEquals` will ensure that a cell contains a given value

See below on the dedicated section 'Testing utilities'

Testing guidelines

- `cy.wait` should be used in the last resort scenario. It's considered a bad practice, though sometimes there is no other choice
- Tests should remain isolated from each other. It's best to try one feature at the time
- A test should always start with a fresh project
- The name of the files should mirror the OpenRefine UI organization

Testing utilities

OpenRefine contributors have added some utility methods on the top of the Cypress framework. Those methods perform some common actions or assertions on OpenRefine, to avoid code duplication.

Utilities can be found in `cypress/support/commands.js`.

The most important utility method is `loadAndVisitProject`.

This method will create a fresh OpenRefine project based on a dataset given as a parameter.

The fixture parameter can be

- An arbitrary array, the first row is for the column names, other rows are for the values
Use an arbitrary array **only** if the test requires some specific grid values

Example:

```
const fixture = [
  ['Column A', 'Column B', 'Column C'],
  ['0A', '0B', '0C'],
```

- A referenced dataset: `food.small` or `food.mini`
Most of the time, tests does not require any specific grid values
Use `food.mini` as much as possible, it loads 2 rows and very few columns in the grid
Use `food.small` if the test requires a few hundred rows in the grid
- Those datasets live in `cypress/fixtures`

Browsers

In terms of browsers, Cypress is using what is installed on your operating system. See the [Cypress documentation](#) for a list of supported browsers

Folder organization

Tests are located in `main/tests/cypress/cypress` folder. The test should not use any file outside the cypress folder.

- `/fixtures` contains CSVs and OpenRefine project files used by the tests
- `/integration` contains the tests
- `/plugins` contains custom plugins for the OR project
- `/screenshots` and `/videos` contains the recording of the tests, Git ignored
- `/support` is a custom library of assertion and common user actions, to avoid code duplication in the tests themselves

Configuration

Cypress execution can be configured with environment variables, they can be declared at the OS level, or when running the test

Available variables are

- OPENREFINE_URL, determine on which scheme://url:port to access OpenRefine, default to <http://localhost:333>
- DISABLE_PROJECT_CLEANUP, If set to 1, projects will not be deleted after each run. Default to 0 to keep the OpenRefine instance clean

Cypress contains [exhaustive documentation](#) about configuration, but here are two simple ways to configure the execution of the tests:

Overriding with a cypress.env.json file

This file is ignored by Git, and you can use it to configure Cypress locally

Command-line

You can pass variables at the command-line level

```
yarn --cwd ./main/tests/cypress run cypress open --env  
OPENREFINE_URL="http://localhost:1234"
```

Visual testing

Tests generally ensure application behavior by making assertions against the DOM, to ensure specific texts or css attributes are present in the document body.

Visual testing, on the contrary, is a way to test applications by comparing images. A reference screenshot is taken the first time the test runs, and subsequent runs will compare a new screenshot against the reference, at the pixel level.

Here is an [introduction to visual testing by Cypress](#).

In some cases, we are using visual testing.

We are using [Cypress Image Snapshot](#)

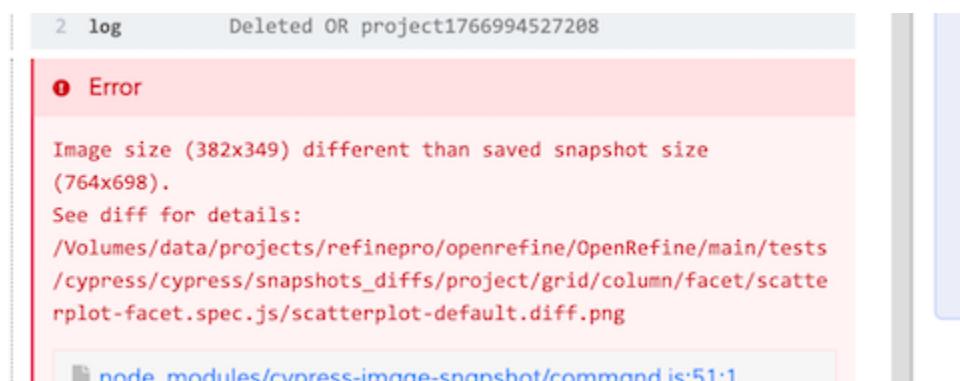
Identified cases are so far:

- testing images created by OpenRefine backend (scatterplots for example)

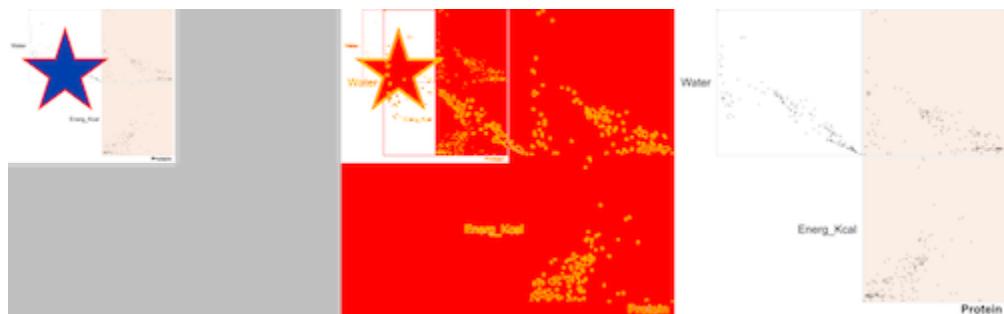
Reference screenshots (Called snapshots), are stored in /cypress/snapshots. And a snapshot can be taken for the whole page, or just a single part of the page.

When a visual test fails

First, Cypress will display the following error message:



Then, a diff image will be created in /cypress/snapshots, this directory is ignored by Git. The diff images shows the reference image on the left, the image that was taken during the test run on the right, and the diff in the middle.



CI/CD

In CI/CD, tests are run headless, with the following command-line

```
./refine e2e_tests
```

Results are displayed in the standard output

Resources

[Cypress command line options](#) [Lots of good Cypress examples](#)

Architecture before version 4

OpenRefine is a web application, but is designed to be run locally on your own machine. The server-side maintains states of the data (undo/redo history, long-running processes, etc.) while the client-side maintains states of the user interface (facets and their selections, view pagination, etc.). The client-side makes GET and POST ajax calls to cause changes to the data and to fetch data and data-related states from the server-side.

This architecture provides a good separation of concerns (data vs. UI); allows the use of familiar web technologies (HTML, CSS, Javascript) to implement user interface features; and enables the server side to be called by third-party software through standard GET and POST requests.

Technology stack

The server-side (back-end) part of OpenRefine is implemented in Java as one single servlet which is executed by the [Jetty](#) web server and servlet container. The use of Java strikes a balance between performance and portability across operating systems (there is very little OS-specific code and has mostly to do with starting the application).

The functional extensibility of OpenRefine is provided by a fork of the [SIMILE Butterfly](#) modular web application framework. With this framework, extensions are able to provide new functionality both in the server- and client-side. A [list of known extensions](#) is maintained on our website and we have [specific documentation for extension developers](#).

The client-side part of OpenRefine is implemented in HTML, CSS and plain Javascript. It primarily uses the following libraries:

- [jQuery](#)
- [Wikimedia's jQuery.i18n](#) The front-end dependencies are fetched at build time via [NPM](#).

The server-side part of OpenRefine relies on many libraries, for instance to implement import and export in many different formats. Those are fetched at build time via [Apache Maven](#).

The data storage and processing architecture is being transformed. Up to version 3.x, OpenRefine uses an in-memory storage, where the entire project grid is loaded in the Java heap, with operations mutating that state. From 4.x on, OpenRefine uses a different architecture, where data is stored on disk by default and cached in memory if the project is small enough.

Server-side architecture

OpenRefine's server-side is written entirely in Java ([main/src/](#)) and its entry point is the Java servlet [com.google.refine.RefineServlet](#). By default, the servlet is hosted in the lightweight Jetty web server instantiated by [server/src/com.google.refine.Refine](#). Note that the server class itself is under [server/src/](#), not [main/src/](#); this separation leaves the possibility of hosting [RefineServlet](#) in a different servlet container.

The web server configuration is in [main/webapp/WEB-INF/web.xml](#); that's where [RefineServlet](#) is hooked up. [RefineServlet](#) itself is simple: it just reacts to requests from the client-side by routing them to the right [Command](#) class in the packages [com.google.refine.commands](#).**

As mentioned before, the server-side maintains states of the data, and the primary class involved is [com.google.refine.ProjectManager](#).

Projects

In OpenRefine there's the concept of a workspace similar to that in [Eclipse IDE](#). When you run OpenRefine it manages projects within a single workspace, and the workspace is organized in a file directory with sub-directories. The default workspace directories are listed [in the manual](#) and it also explains how to change them.

The class [ProjectManager](#) is what manages the workspace. It keeps in memory the metadata of every project (in the class [ProjectMetadata](#)). This metadata includes the project's name and last modified date, and any other information necessary to present and let the user interact with the

project as a whole. Only when the user decides to look at the project's data would [ProjectManager](#) load the project's actual data. The separation of project metadata and data is to minimize the amount of stuff loaded into memory.

A project's *actual* data includes the columns, rows, cells, reconciliation records, and history entries.

A project is loaded when it needs to be displayed or modified, and it remains in memory until 1 hour after the last time it gets modified. Periodically the project manager tries to save modified projects, and it saves as many modified projects as possible within 30 seconds.

Data Model

A project's data consists of

- *raw data*: a list of rows, each row consisting of a list of cells
- *models* on top of that raw data that give high-level presentation or interpretation of that data. This design lets the same raw data be viewed in different ways by different models, and let the models be changed without costly changes to the raw data.

Column Model

Cells in rows are not named and can only be addressed by their list position indices. So, a *column model* is needed to give a name to each list position. The column model also stores other metadata for each column, including the type that cells in the column have been reconciled to and the overall reconciliation statistics of those cells.

Each column also acts as a cache for data computed from the raw data related to that column.

Columns in the column model can be removed and re-ordered without changing the raw data—the cells in the rows. This makes column removal and ordering operations really quick.

Column Groups



CAUTION

This feature is partially implemented, buggy and deprecated. It will be removed in OpenRefine 4.0. See the following links for details:

- [Issue#5122](#) that first argues it's a useful feature, but then agrees with its deprecation
- [Discussion Who uses column groups?](#)
- [Discussion The future of the records mode](#) about better ways of implementing grouping and a hierarchical model in OpenRefine.

This feature is related to [Rows vs Records](#), which however continues to be supported.

Consider the following data:

primary key column	movie title	year	actor	character	award	language	country	release_date	editions
1.	Austin Powers	1997	Elizabeth Hurley	Vanessa Kensington		English	USA	May 2 1997	Widescreen
2.			Mike Myers	Austin Powers		French	Germany	N/A	HD
3.				Dr. Evil	MTV Best Villain	Japanese			
4.					Best Comedy Perf				
5.			Robert Wagner	Number Two					
6.			Seth Green	Scott Evil					

Although the data is in a grid, we humans can understand that it is a tree. First of all, all rows contain data ultimately linked to the movie Austin Powers, although only one row contains the text "Austin Powers" in the "movie title" column. We also know that "USA" and "Germany" are not related

to Elizabeth Hurley and Mike Myers respectively (say, as their nationality), but rather, "USA" and "Germany" are related to the movie (where it was released). We know that Mike Myers played both the character "Austin Powers" and the character "Dr. Evil"; and for the latter he received 2 awards. We humans can understand how to interpret the grid as a tree based on its visual layout as well as some knowledge we have about the movie domain but is not encoded in the table.

OpenRefine can capture our knowledge of this transformation from grid to tree using *column groups*, also stored in the column model. Each column group illustrated as a blue bracket above specifies which columns are grouped together, as well as which of those columns is the key column in that group (blue triangle). One column group can span over columns grouped by another column group, and in this way, column groups form a hierarchy determined by which column group envelopes another. This hierarchy of column groups allows the 2-dimensional (grid-shaped) table of rows and cells to be interpreted as a list of hierarchical (tree-shaped) data records.

Blank cells play a very important role. The blank cell in a key column of a row (e.g., cell "character" on row 4) makes that row (row 4) *depend* on the first preceding row with that column filled in (row 3). This means that "Best Comedy Perf" on row 4 applies to "Dr. Evil" on row 3. Row 3 is said to be a *context row* for row 4. Similarly, since rows 2 - 6 all have blank cells in the first column, they all depend on row 1, and all their data ultimately applies to the movie Austin Powers. Row 1 depends on no other row and is said to be a *record row*. Rows 1 - 6 together form one *record*.

Currently (as of 12th December 2017) only the XML and JSON importers create column groups, and while the data table view does display column groups but it doesn't support modifying them.

Changes, History, Processes, and Operations

All changes to the project's data are tracked (N.B. this does not include changes to a project's metadata - such as the project name.)

Changes are stored as `com.google.refine.history.Change` objects. `com.google.refine.history.Change` is an interface, and implementing classes are in `com.google.refine.model.changes.**`. Each change object stores enough data to modify the project's data when its `apply()` method is called, and enough data to revert its effect when its `revert()` method is called. It's only supposed to *store* data, not to actually *compute* the change. In this way, it's like a .diff patch file for a code base.

Some change objects can be huge, as huge as the project itself. So change objects are not kept in memory except when they are to be applied or reverted. However, since we still need to show the user some information about changes (as displayed in the History panel in the UI), we keep metadata of changes separate from the change objects. For each change object there is one corresponding `com.google.refine.history.HistoryEntry` for storing its metadata, such as the change's human-friendly description and timestamp.

Each project has a `com.google.refine.history.History` object that contains an ordered list of all `HistoryEntry` objects storing metadata for all changes that have been done since after the project was created. Actually, there are 2 ordered lists: one for done changes that can be reverted (undone), an done for undone changes that can be re-applied (redone). Changes must be done or redone in their exact orders in these lists because each change makes certain assumptions about the state of the project before and after it is applied. As changes cannot be undone/redone out of order, when one change fails to revert, it blocks the whole history from being reverted to any state preceding that change (as happened in [Issue #2](#)).

As mentioned before, a change contains only the diff and does not actually compute that diff. The computation is performed by a `com.google.refine.process.Process` object—every change object is created by a process object. A process can be immediate, producing its change object synchronously within a very short period of time (e.g., starring one row); or a process can be long-running, producing its change object after a long time and a lot of computation, including network calls (e.g., reconciling a column).

As the user interacts with the UI on the client-side, their interactions trigger ajax calls to the server-side. Some calls are meant to modify the project. Those are handled by commands that instantiates processes. Processes are queued in a first-in-first-out basis. The first-in process gets run and until it is done all the other processes are stuck in the queue.

A process can effect a change in one thing in the project (e.g., edit one particular cell, star one particular row), or a process can effect changes in *potentially* many things in the project (e.g., edit zero or more cells sharing the same content, starring all rows filtered by some facets). The latter kind of process is generalizable: it is meaningful to apply them on another similar project. Such a process is associated with an *abstract operation* `com.google.refine.model.AbstractOperation` that encodes the information necessary to create another instance of that process, but potentially for a different project. When you click "extract" in the History panel, these abstract operations are called to serialize their information to JSON; and when you click "apply" in the History panel, the JSON you paste in is used to re-construct these abstract operations, which in turn create

processes, which get run sequentially in a queue to generate change object and history entry pairs.

In summary,

- change objects store diffs
- history entries store metadata of change objects
- processes compute diffs and create change object and history entry pairs
- some processes are long-running and some are immediate; processes are run sequentially in a queue
- generalizable processes can be re-constructed from abstract operations

Client-side architecture

The client-side part of OpenRefine is implemented in HTML, CSS and Javascript and uses the following Javascript libraries:

- [jQuery](#)
- [jQueryUI](#)
- [Recurser jquery-i18n](#)

Importing architecture

OpenRefine has a sophisticated architecture for accommodating a diverse and extensible set of importable file formats and workflows. The formats range from simple CSV, TSV to fixed-width fields to line-based records to hierarchical XML and JSON. The workflows allow the user to preview and tweak many different import settings before creating the project. In some cases, such as XML and JSON, the user also has to select which elements in the data file to import. Additionally, a data file can also be an archive file (e.g., .zip) that contains many files inside; the user can select which of those files to import. Finally, extensions to OpenRefine can inject functionalities into any part of this architecture.

The Index Page and Action Areas

The opening screen of OpenRefine is implemented by the file [main/webapp/modules/core/index.vt](#) and will be referred to here as the index page. Its default implementation contains 3 finger tabs labeled Create Project, Open Project, and Import Project. Each tab selects an "action area". The 3 default action areas are for, obviously, creating a new project, opening an existing project, and importing a project .tar file.

Extensions can add more action areas in Javascript. For example, this is how the Create Project action area is added ([main/webapp/modules/core/scripts/index/create-project-ui.js](#)):

```
Refine.actionAreas.push({  
  id: "create-project",  
  label: "Create Project",  
  uiClass: Refine.CreateProjectUI  
});
```

The UI class is a constructor function that takes one argument, a jQuery-wrapped HTML element where the tab body of the action area should be rendered.

If your extension requires a very unique importing work flow, or a very novel feature that should be exposed on the index page, then add a new action area. Otherwise, try to use the existing work flows as much as possible.

The Create Project Action Area

The Create Project action area is itself extensible. Initially, it embeds a set of finger tabs corresponding to a variety of "source selection UIs": you can

select a source of data by specifying a file on your computer, or you can specify the URL to a publicly accessible data file or data feed, or you can paste in from the clipboard a chunk of data.

There are actually 3 points of extension in the Create Project action area, and the first is invisible.

Importing Controllers

The Create Project action area manages a list of "importing controllers". Each controller follows a particular work flow (in UI terms, think "wizard"). Refine comes with a "default importing controller" (`refine/main/webapp/modules/core/scripts/index/default-importing-controller/controller.js`) and its work flow assumes that the data can be retrieved and cached in whole before getting processed in order to generate a preview for the user to inspect. (If the data cannot be retrieved and cached in whole before previewing, then another importing controller is needed.)

An importing controller is just programming logic, but it can manifest itself visually by registering one or more data source UIs and one or more custom panels in the Create Project action area. The default importing controller registers 3 such custom panels, which act like pages of a wizard.

An extension can register any number of importing controller. Each controller has a client-side part and a server-side part. Its client-side part is just a constructor function that takes an object representing the Create Project action area (usually named `createProjectUI`). The controller (client-side) is expected to use that object to register data source UIs and/or create custom panels. The controller is not expected to have any particular interface method. The default importing controller's client-side code looks like this (`main/webapp/modules/core/scripts/index/default-importing-controller/controller.js`):

```
Refine.DefaultImportingController = function(createProjectUI) {
  this._createProjectUI = createProjectUI; // save a reference to the create project action area

  this._progressPanel = createProjectUI.addCustomPanel(); // create a custom panel
  this._progressPanel.html('...'); // render the custom panel
  ... do other stuff ...
};

Refine.CreateProjectUI.controllers.push(Refine.DefaultImportingController); // register the controller
```

We will cover the server-side code [below](#).

Data Source Selection UIs

Data source selection UIs are another point of extensibility in the Create Project action area. As mentioned previously, by default there are 3 data source UIs. Those are added by the default importing controller.

Extensions can also add their own data source UIs. A data source selection UI object can be registered like so

```
createProjectUI.addSourceSelectionUI({
  label: "This Computer",
  id: "local-computer-source",
  ui: theDataSourceSelectionUIObject
});
```

`theDataSourceSelectionUIObject` is an object that has the following member methods:

- `attachUI(bodyDiv)`
- `focus()`

If you want to install a data source selection UI that is managed by the default importing controller, then register its UI class with the default importing controller, like so (`main/webapp/modules/core/scripts/index/default-importing-sources/sources.js`):

```
Refine.DefaultImportingController.sources.push({
  "label": "This Computer",
```

The default importing controller will assume that the `uiClass` field is a constructor function and call it with one argument—the controller object itself. That constructor function should save the controller object for later use. More specifically, for data source UIs that use the default importing controller, they can call the controller to kickstart the process that retrieves and caches the data to import:

```
controller.startImportJob(form, "... status message ...");
```

The argument `form` is a jQuery-wrapped FORM element that will get submitted to the server side at the command [/command/core/create-importing-job](#). That command and the default importing controller will take care of uploading or downloading the data, caching it, updating the client side's progress display, and then showing the next importing step when the data is fully cached.

See [main/webapp/modules/core/scripts/index/default-importing-sources/sources.js](#) for examples of such source selection UIs. While we write about source selection UIs managed by the default importing controller here, chances are your own extension will not be adding such a new source selection UI. Your extension probably adds with a new importing controller as well as a new source selection UI that work together.

File Selection Panel

This screen is shown when there are multiple files to choose from when creating a project, for instance after uploading a zip file with multiple files in it. This interface lets the user choose which files to import to create a new project. Although OpenRefine only supports one table per project so far, it is possible to select multiple files to import. Their contents will be concatenated into a single table.

Parsing UI Panel

The parsing UI panel is shown when importing data into a new project. Primarily, it lets the user select in which format the data is, which determines how it is read and transformed into an OpenRefine project. The back-end will try to supply an informed guess for the format using the [format guesser](#), but it is not uncommon that this initial choice must be overridden by the user.

Beyond this choice of format, the parsing UI panel offers a configuration panel for the chosen importer. This part of the UI can be defined independently for each input format, given that not all options are relevant for all formats. For instance, when selecting the "Text file" option, the specific UI of the [LineBasedImporter](#) will be shown. This UI is defined in:

- [main/webapp/modules/core/scripts/index/parser-interfaces/line-based-parser-ui.html](#)
- [main/webapp/modules/core/scripts/index/parser-interfaces/line-based-parser-ui.js](#)

Other importers generally define their own parsing configuration panel as well.

The link between the format's identifier (MIME type), importer (Java class which defines the parsing logic) and parsing options UI (Javascript class that defines the rendering of this options area) is made in the [main/webapp/modules/core/MOD-INF/controller.js](#) file, where those components are registered together in the [ImportingManager](#).

Server-side Components

ImportingController

An importing controller is a component of the back-end which is in charge of the entire importing workflow, from the initial transfer of the raw data to be imported to the created project, with all the configuration steps in between, [as described in the earlier section](#). OpenRefine comes with a default importing controller which implements this for data coming from:

- file upload by the user via the web interface
- upload of textual information using the clipboard import form
- download of a file by supplying a URL

For all of these data sources, the first step consists of storing the corresponding input files in a temporary directory inside the workspace. The default importing controller provides an HTTP API used by the front-end to select which files to import, [predict the format they are in](#), provide default importing options for the selected format, preview the project's first few rows with the given options, and finally create the project.

The importing controller is not used for loading existing projects or importing OpenRefine project archives: the project manager is responsible for both of those.

Extensions can define other importing controllers to implement other importing flows depending on the data source. For instance, importing data from a SQL database requires different steps such as selecting the database and providing a SQL query. The `database` extension implements such a workflow by providing its own importing controller.

FormatGuesser

A format guesser is a class that tries to determine the MIME type of a file, considering its contents. The `FormatGuesser` interface has multiple implementations, which can be used to determine the format depending on its basic type (binary, text based). For text files, this relies on heuristics which are quite ad-hoc and brittle. For binary files, we do not currently try to do anything, while one would at least expect that we check for some so-called magic numbers at the beginning of files, which can be used to detect many file formats.

Worse, our format guessing logic does not actually attempt to parse the files with the guessed formats, so it is not uncommon that the user is directly presented with a parsing error (in the form of a javascript alert) upon importing files.

This could be avoided by trying to read the given files with the predicted importer before suggesting the format to the user, making sure that at least that does not throw an exception.

ImportingParser

An `ImportingParser` is a class that is responsible for parsing a file into OpenRefine's project model. It takes a range of importing options passed on from the frontend, input by the user into a dedicated UI, specific to the format being parsed.

When possible, parsers are designed so that they can import the first few rows of the project without reading the entire input file in memory. This helps provide fast previews of the project to be created when the user changes importing options. Every change in the importing options triggers a new parse of the source files (unless the user has disabled auto preview option in the parsing configuration panel).

Faceted browsing architecture

Faceted browsing support is core to OpenRefine as it is the primary and only mechanism for filtering to a subset of rows on which to do something *en masse* (ie in bulk). Without faceted browsing or an equivalent querying/browsing mechanism, you can only change one thing at a time (one cell or row) or else change everything all at once; both kinds of editing are practically useless when dealing with large data sets.

In OpenRefine, different components of the code need to know which rows to process from the faceted browsing state (how the facets are constrained). For example, when the user applies some facet selections and then exports the data, the exporter serializes only the matching rows, not all rows in the project. Thus, faceted browsing isn't only hooked up to the data view for displaying data to the user, but it is also hooked up to almost all other parts of the system.

Engine Configuration

As OpenRefine is a web app, there might be several browser windows opened on the same project, each in a different faceted browsing state. It is best to maintain the faceted browsing state in each browser window while keeping the server side completely stateless with regard to faceted browsing. Whenever the client-side needs something done by the server, it transfers the entire faceted browsing state over to the server-side. The faceted browsing state behaves much like the `WHERE` clause in a SQL query, telling the server-side how to select the rows to process.

In fact, it is best to think of the faceted browsing state as just a database query much like a SQL query. It can be passed around the whole system, to any component needing to know which rows to process. It is serialized into JSON to pass between the client-side and the server side, or to save in an abstract operation's specification. The job of the faceted browsing subsystem on the client-side is to let the user interactively modify this "faceted browsing query", and the job of the faceted browsing subsystem on the server side is to resolve that query.

In the code, the faceted browsing state, or faceted browsing query, is actually called the *engine configuration* or *engine config* for short. It consists

mostly of an array facet configurations. For each facet, it stores the name of the column on which the facet is based (or an empty string if there is no base column). Each type of facet has different configuration. Text search facets have queries and flags for case-sensitivity mode and regular expression mode. Text facets (aka list facets) and numeric range facets have expressions. Each list facet also has an array of selected choices, an invert flag, and flags for whether blank and error cells are selected. Each numeric range facet has, among other things, a "from" and a "to" values. If you trace the AJAX calls, you'd see the engine configs being shuttled, e.g.,

```
{
  "mode": "rows",
  "facets" : [
    {
      "type": "text",
      "name": "Shrt_Desc",
      "columnName": "Shrt_Desc",
      "mode": "text",
      "caseSensitive": false,
      "query": "cheese"
    },
    {
      "type": "list",
      "name": "Shrt_Desc",
      "columnName": "Shrt_Desc",
      "expression": "grel:value.toLowerCase().split('\\', '\\')",
      "omitBlank": false,
      "omitError": false,
      "selection": [],
      "selectBlank": false,
      "selectError": false,
      "invert": false
    },
    {
      "type": "range",
      "name": "Water",
      "expression": "value",
      "columnName": "Water",
      "selectNumeric": true,
      "selectNonNumeric": true,
      "selectBlank": true,
      "selectError": true,
      "from": 0,
      "to": 53
    }
  ]
}
```

Server-Side Subsystem

From an engine configuration like the one above, the server-side faceted browsing subsystem is capable of producing:

- an iteration over the rows matching the facets' constraints
- information on how to render the facets (e.g., choice and count pairs for a list facet, histogram for a numeric range facet)

When the engine config JSON arrives in an HTTP request on the server-side, a `com.google.refine.browsing.Engine` object is constructed and initialized with that JSON. It turns constructs zero or more `com.google.refine.browsing.facets.Facet` objects. Then for each facet, the engine calls its `getRowFilter()` method, which returns `null` if the facet isn't constrained in anyway, or a `com.google.refine.browsing.filters.RowFilter` object. Then, to when iterating over a project's rows, the engine calls on all row filters' `filterRow()` method. If and only if all row filters return `true` the row is considered to match the facets' constraints. How each row filter works depends on the corresponding type of facet.

To produce information on how to render a particular facet in the UI, the engine follows the same procedure described in the previous except it

skips over the facet in question. In other words, it produces an iteration over all rows constrained by the other facets. Then it feeds that iteration to the facet in question by calling the facet's `computeChoices()` method. This gives the method a chance to compute the rendering information for its UI counterpart on the client-side. When all facets have been given a chance to compute their rendering information, the engine calls all facets to serialize their information as JSON and returns the JSON to the client-side. Only one HTTP call is needed to compute all facets.

Client-side subsystem

On the client-side there is also an engine object (implemented in Javascript rather than Java) and zero or more facet objects (also in Javascript, obviously). The engine is responsible for distributing the rendering information computed on the server-side to the right facets, and when the user interacts with a facet, the facet tells the engine to update the whole UI. To do so, the engine gathers the configuration of each facet and composes the whole engine config as a single JSON object. Two separate AJAX calls are made with that engine config, one to retrieve the rows to render, and one to re-compute the rendering information for the facets because changing one facet does affect all the other facets.

Architecture in version 4

This page explains the architecture of OpenRefine 4, which is currently being developed and differs from the previous versions in important ways. Projects and their history are represented differently, which makes it possible to work on tables which do not fit in RAM, improve the user experience around long-running operations in various ways and the reproducibility of OpenRefine workflows more broadly.

Project history structure

The history of a project is a succession of different states of the project. Each state is a [Grid](#), which comprises the following data:

- column headers, containing column metadata such as the column names;
- the project table itself;
- the overlay models stored alongside the table (such as a Wikibase schema).

The initial grid of the project is generated by the importer which created the project. Each following is obtained by applying a [Change](#) on the previous grid. A change is a function which takes the previous grid and produces the new grid.

Grids are immutable objects. This means that a change cannot mutate the grid in place: it must create a modified copy of the grid instead. We will see in [the Runners section](#) why can be done efficiently and does not require a lot of data copy in general. The immutability of grids provides useful guarantees about thread safety. For instance, this helps ensure that the evaluation of facets happened at a precise point in project history, and does not give a mixed view of different grid states.

In the previous architecture, changes were not only responsible for applying themselves to

a grid, but also to revert themselves, meaning that they had to implement the reverse operation as well. Only requiring the forward application has important consequences:

- Implementing a change is easier, since we only need to implement the function in one direction. Implementing both directions and making sure they were indeed inverses of each other is non-trivial and gave rise to certain bugs, such as [#2](#). Because we let extensions define their own changes, there was a significant risk that an extension implements a change incorrectly, leaving the project in an inconsistent state after using the undo feature.
- Many operations in OpenRefine are destructive, meaning that reverting them requires storing the deleted or altered data in the change object itself, to be able to restore it upon reversion. This is no longer necessary, making it possible for changes to be much lighter;
- To undo a change, we need to have kept the earlier version of the grid accessible. We will see in [the Runners section](#) why this can be done efficiently, without keeping all grids in memory explicitly.

Runners

The implementation of grids and of the transformations on them is pluggable. This means that the way a grid is concretely represented in memory can be adapted, depending on the resources available and the broader context of execution. This relies on the notion of [Runner](#), which is essentially a factory class for grids which picks a particular implementation of the Grid interface when creating such grids.

- The [local runner](#) is the default one. It is designed to be run when all of the data to transform is located on the same machine (where OpenRefine is running). Project data is read from disk in a lazy fashion, i.e. only when the corresponding grid values need to be displayed, aggregated or exported. Therefore it makes it possible to run OpenRefine on large datasets without the need for a large working memory (RAM).

We explain the differences between [lazy and eager evaluation](#) in the dedicated section.

- The Spark runner is designed to run on distributed datasets. Those datasets can be split into blocks which are stored on different machines. The execution of the workflow can be shared between the various machines, which form a Spark cluster. As in the local runner, changes are evaluated lazily.
- The testing runner is a very simple runner, which loads all the data it works on in memory. It is not optimized for performance at all: it simply meets the specification of the runner interface in the simplest possible way. It is used in tests of operations, importers and other basic blocks of workflows. Its simplicity makes it fast enough on small testing datasets generally used in such tests. This runner also performs additional checks during execution, so that incorrect behaviours can be detected more easily during testing. Unlike the local and Spark runners, the testing runner uses eager evaluation.

The runner can be configured on the command-line with the `-r` option (`/r` on Windows), by providing the class name of the data model runner to use. In the `refine.ini` configuration file, the corresponding option is `refine.runner.class`. The class names of the available runners are:

- `org.openrefine.runners.local.LocalRunner` for the **local runner** (default)
- `org.openrefine.runners.spark.SparkRunner` for the Spark-based runner
- `org.openrefine.runners.testing.TestingRunner` for the testing runner

A common test suite, gathering test cases that all runners should pass, is available in [RunnerTestBase](#). This can be used by creating a test class which extends this base class.

Lazy and eager evaluation

Given that grids are immutable, changing even just a single cell in a grid requires making a copy of that grid. One strategy to achieve this, which is used by the testing runner, is to create a copy of all cells of the original grid, with the change on the desired cell, and return this as a new grid. But grids do not have to be backed by a list of lists containing the value of each cell.

Instead, the lazy runners use a different strategy. The returned grid is a proxy object, which contains a pointer to the original grid, and implements all operations on the new grid by forwarding them to the original grid, and adapting their results to match the ones expected of the new grid.

We call those runners lazy because the operations on their grids are [lazily evaluated](#): the actual computation does not happen when creating the modified grid, but is delayed to the later use of the data in the grid (such as running an aggregation on it to compute facet statistics, or retrieving a range of rows).

This has a few important consequences:

- the proxy objects representing those transformed grids are very lightweight: they just hold the recipe to compute the new grid out of the old one, not the data itself. This means that we can generally afford to keep many such proxy objects loaded in memory. In a lot of cases, OpenRefine will indeed hold a Grid object for every single step of a project's history. As a result, it becomes possible to undo a transformation even though Change objects do not support reverting their effects: we simply roll back to the earlier Grid object.
- when the history grows longer, the project data is accessed via a long chain of proxy objects, meaning that the project workflow is re-executed many times. When that becomes too expensive, it is possible to turn on caching in one of the intermediate

grids, meaning that we compute all the contents of that grid once and store that in memory or on disk. This strategy is detailed in the [Memory management](#) section.

- with lazy evaluation, some computations required by changes can be evaluated many times (for instance, each time facets are refreshed). This is undesirable for expensive or effectful computations (for instance, fetching data from a URL). To prevent this from happening, changes are allowed to store the results of such expensive computations in [ChangeData objects](#). Such objects are evaluated only once, persisted on disk, and the new grid can be derived by combining the old grid with some ChangeData objects (generally by joining them).

ChangeData objects

A [ChangeData](#) object is a data structure which holds change data which should be computed only once given the cost or side-effects of recomputing it multiple times. It is for instance used to store reconciliation data, data fetched from URLs, or the results of evaluating expressions which are not guaranteed to be [pure](#).

ChangeData objects are able to index arbitrary data items associated to rows or records in a Grid. To each row or record id corresponds at most one such item. The typical lifecycle for a ChangeData object is as follows:

- The ChangeData object is derived from a Grid by applying a function row or record-wise to the grid, using [Grid::mapRows](#) or [Grid::mapRecords](#). This is typically a lazy operation, meaning that the expensive computations involved are not done yet;
- Immediately after, it is saved to a file within the project's storage directory (see [Project serialization](#)). This saving can take time as this triggers the expensive computations applied at the earlier step.
- As soon as the saving has started (and while it is still in progress), the ChangeData can be read back from its serialization, using [Runner.loadChangeData](#). Only the items computed and written at that stage are read back.

- The ChangeData can be joined back with the Grid it was derived from, inserting the change data into cells (for instance by creating a new column where fetched URLs are stored). This is done by means of methods such as `Grid::join`.

Project serialization

A single grid is serialized in its own directory, with the following components:

- `metadata.json`, containing the column model and overlay models associated with the grid;
- `grid/part-* .gz` files, which are Gzip-compressed text files containing for each line the JSON serialization of a row in the project.

The fact that the grid's serialization is spread on multiple files makes it possible to process it in parallel (with one worker per partition). Rows are sorted and each partition contains a sequential chunk of rows, meaning that to access a given row by row number, it is possible to read the relevant partition file only. The number of partitions depends on the size of the project and the options of the runner.

A project is serialized in its own directory too, with the following components:

- `history.json`, containing the list of history entries, which includes the associated Changes and Operations;
- `initial/`, a directory where the first grid of the project (the one created by the importer) is stored;
- `changes/`, a directory where all the ChangeData objects are serialized. They are stored in subdirectories determined by the history entry id they are associated to.
- `cache/`, a directory where grids other than the initial one can be stored. This is useful when the project history grows longer to avoid recomputing all operations from the initial grid at every HTTP request.

Memory management

TODO describe here the caching strategy (being reworked)

Local runner

The local runner is the default one, as it is designed to be efficient in OpenRefine's intended usage conditions: running locally on the machine where the data cleaning is being done. Its design is inspired by Spark. Spark itself could not be used in place of this runner because its support for distributed computations and redundancy adds significant overheads which make the tool less responsive when run locally. Also, OpenRefine relies on the order of rows in many contexts, and row order is not preserved by many Spark primitives, making its abstractions less useful for OpenRefine as default implementation.

Options

The following configuration parameters can be used with this runner:

Configuration key	Default value	Description
<code>refine.runner.defaultParallelism</code>	4	how many partitions datasets should generally be split, unless they are very small or very big

Configuration key	Default value	Description
<code>refine.runner.minSplitSize</code>	4096	minimum size of a partition in bytes. Datasets which are smaller than this value will not be split at all.
<code>refine.runner.maxSplitSize</code>	16777216	maximum size of a partition in bytes. Datasets which are larger than <code>defaultParallelism * maxSplitSize</code> will be split in more partitions than the default parallelism.

Partitioned Lazy Lists

The core data structure which underpins the lazy representation of Grids and ChangeData objects in the local runner is the Partitioned Lazy List (PLL). It is a lightweight version of Spark's [Resilient Distributed Dataset \(RDD\)](#). It is:

- a list, because it represents an ordered collections of objects;
- lazy, because by default it does not store its contents as explicit objects in memory. Instead, the elements are computed on-demand, when they are accessed.
- partitioned, because it divides its contents into contiguous groups of elements called partitions. Each partition can be enumerated from independently, making it possible to run processes in parallel on different parts of the list.

In contrast to RDDs, PLLs are:

- not distributed: all of the data must be locally accessible, all the computations are happening in the same JVM
- not resilient: there is no support for redundancy.

The concurrency in PLLs is implemented with Java threads. When instantiated, the local runner starts a thread pool which is used on demand when computations are executed.

Methods and theory behind the clustering functionality in OpenRefine.

Introduction

In OpenRefine, [clustering](#) refers to the operation of "finding groups of different values that might be alternative representations of the same thing."

It is worth noting that clustering in OpenRefine works only at the syntactic level (the character composition of the cell value) and, while very useful to spot errors, typos, and inconsistencies, it's by no means enough to perform effective semantically-aware reconciliation. This is why OpenRefine uses [external semantically-aware reconciliation services](#) (such as Wikidata) to compensate for the deficiencies of syntax-level clustering alone.

Methodologies

To strike a balance between general applicability and usefulness, OpenRefine ships with a selected number of clustering methods and algorithms that have proven effective and fast enough to use in a wide variety of situations.

OpenRefine currently offers 2 broad categories of clustering methods:

1. Token-based (n-gram, key collision, etc.)
2. Character-based, also known as Edit distance (Levenshtein distance, PPM, etc.)

NOTE: Performance differs depending on the strings that you want to cluster in your data which might be short or very long or varying. String length complexity has a large part to do with the algorithm that might perform faster (but not necessarily better!). In general, it's usually best to use heavy algorithms for shorter strings that can provide better quality – like Levenshtein distance. And token-based algorithms for longer strings such as n-grams (q-grams), bag distance, Jaccard similarity, Dice coefficient, etc. (some of which we do not provide currently).

Key Collision Methods

"Key Collision" methods are based on the idea of creating an alternative representation of a value (a "key") that contains only the most valuable or meaningful part of the string and "buckets" (or "bin" as it's described inside OpenRefine's code) together different strings based on the fact that their key is the same (hence the name "key collision").

Fingerprint

The fingerprinting method is the least likely to produce false positives, which is why it is the default method.

The process that generates the key from a string value is the following (note that the order of these operations is significant):

- remove leading and trailing whitespace
- change all characters to their lowercase representation
- remove all punctuation and control characters
- normalize extended western characters to their ASCII representation (for example "gödel" → "godel")
- split the string into whitespace-separated tokens
- sort the tokens and remove duplicates
- join the tokens back together

If you're curious, the code that performs this is [here](#).

Several factors play a role in this fingerprint:

- because whitespace is normalized, characters are lowercased, and punctuation is

removed, those parts don't play a differentiation role in the fingerprint. Because these attributes of the string are the least significant in terms of meaning differentiation, these turn out to be the most varying parts of the strings, and removing them has a substantial benefit in emerging clusters.

- because the string parts are sorted, the given order of tokens doesn't matter (so "Cruise, Tom" and "Tom Cruise" both end up with a fingerprint "cruise tom" and therefore end up in the same cluster)
- normalizing extended western characters plays the role of reproducing data entry mistakes performed when entering extended characters with an ASCII-only keyboard. Note that this procedure can also lead to false positives. For example, "gödel" and "godél" would both end up with "godel" as their fingerprint, but they're likely to be different names, so this might work less effectively for datasets where extended characters play a substantial differentiation role.

N-Gram Fingerprint

The [n-gram](#) fingerprint method does the following:

- change all characters to their lowercase representation
- remove all punctuation, whitespace, and control characters
- obtain all the string n-grams
- sort the n-grams and remove duplicates
- join the sorted n-grams back together
- normalize extended western characters to their ASCII representation

So, for example, the 2-gram fingerprint of "Paris" is "arispari" and the 1-gram fingerprint is "aiprs".

[Check the code](#) if you're curious about the details.

Why is this useful? In practice, using big values for n-grams doesn't yield any advantage

over the previous fingerprint method, but using 2-grams and 1-grams, while yielding many false positives, can find clusters that the previous method didn't find even with strings that have small differences, with a very small performance price.

For example "Krzysztof", "Kryzysztof", and "Krzystof" have different lengths and different regular fingerprints, but share the same 1-gram fingerprint because they use the same letters.

Phonetic Fingerprint

Phonetic fingerprinting is a way to transform tokens into the way they are pronounced. This is useful to spot errors that are due to people misunderstanding or not knowing the spelling of a word after only hearing it. The idea is that similar-sounding words will end up sharing the same key and thus being binned in the same cluster.

For example, "Reuben Gevorkiantz" and "Ruben Gevorkyants" share the same phonetic fingerprint for English pronunciation but they have different fingerprints for both the regular and n-gram fingerprinting methods above, no matter the size of the n-gram.

OpenRefine supports multiple "Phonetic" algorithms including:

Metaphone3

Metaphone3 improves phonetic encoding for not just words in the English Language but also non-English words, first names, and last names common in the United States.

The 'approximate' aspect of the encoding for OpenRefine (version 2.1.3) is implemented according to the following rules:

- All vowels are encoded to the same value - 'A'.
- If the parameter `encodeVowels` is set to false, only *initial* vowels will be encoded at all.

- If `encodeVowels` is set to true, 'A' will be encoded at all places in the word that any vowels are normally pronounced.
- 'W', as well as 'Y', are treated as vowels. Although there are differences in their pronunciation in different circumstances that lead to their being classified as vowels under some circumstances and as consonants in others, for the 'fuzziness' component of the Soundex and Metaphone family of algorithms they will be always be treated here as vowels.
- Voiced and un-voiced consonant pairs are mapped to the same encoded value. That is:
 - 'D' and 'T' -> 'T'
 - 'B' and 'P' -> 'P'
 - 'G' and 'K' -> 'K'
 - 'Z' and 'S' -> 'S'
 - 'V' and 'F' -> 'F'
- In addition to the above voiced/unvoiced rules, 'CH' and 'SH' -> 'X', where 'X' represents the "-SH-" and "-CH-" sounds in Metaphone 3 encoding.
- Also, the sound that is spelled as "TH" in English is encoded to '0' (zero).

Cologne Phonetics

Cologne Phonetics, also called Kölner Phonetik, is a phonetic algorithm that assigns a sequence of digits (called the **phonetic code**) to words.

Unlike Metaphone3, Cologne Phonetics is optimized for the German language.

The process of encoding words using Cologne phonetics can be broken down to the following steps: **Step 1:**

The encoding process is as follows:

- The word is preprocessed. That is,
 - conversion to upper case
 - transcription of germanic umlauts
 - removal of non-alphabetical characters

-The letters are then replaced by their phonetic codes according to the following table:

Letter	Context	Code
A, E, I, O, U		0
H		-
B		1
P	not before H	
D, T	not before C, S, Z	2
F, V, W		3
P	before H	
G, K, Q		
C	at onset before A, H, K, L, O, Q, R, U, X	4
	before A, H, K, O, Q, U, X except after S, Z	

Letter	Context	Code
X	not after C, K, Q	48
L		5
M, N		6
R		7
S, Z		
C	after S, Z	8
	at onset except before A, H, K, L, O, Q, R, U, X	
	not before A, H, K, O, Q, U, X	
D, T	before C, S, Z	
X	after C, K, Q	

For example Following the rules stated above, the phrase "Good morning" in German could be encoded as:

Guten Morgen -> GUTENMORGEN -> 40206607406

Step 2:

Any consecutive duplicate digit is removed

40206607406 -> 4020607406

Step 3:

Removal of all codes "0" **except** at the beginning.

4020607406 -> 426746

Hence, by the Cologne phonetic, Guteng Morgen is encoded as 426746

Note that two or more identical consecutive digits can occur if they occur after removing the "0" digits.

Daitch-Moktoff

The Daitch-Moktoff phonetic algorithm was created by Randy Daitch and Gary Mokotoff of the Jewish Genealogical Society (New York), hence its name.

It is a refinement of the [Soundex](#) algorithms designed to allow greater accuracy in matching Slavic and Yiddish surnames with similar pronunciation but differences in spelling.

The rules for converting surnames into D-M codes are as follows:

- Names are coded to six digits, each digit representing a sound listed in the [coding chart](#)
- When a name lacks enough coded sounds for six digits, use zeros to fill to six digits.
- The letters A, E, I, O, U, J, and Y are always coded at the beginning of a name as in Alice. In any other situation, they are ignored except when two of them form a pair and the pair comes before a vowel, as in Isaiah but not Freud.
- The letter H is coded at the beginning of a name, as in Haber, or preceding a vowel, as in Manheim, otherwise, it is not coded.
- When adjacent sounds can combine to form a larger sound, they are given the code number of the larger sound. Mintz is not coded MIN-T-Z but MIN-TZ.
- When adjacent letters have the same code number, they are coded as one sound, as

in TOPF, which is not coded TO-P-F but TO-PF. Exceptions to this rule are the letter combinations MN and NM, whose letters are coded separately, as in Kleinman.

- When a surname consists of more than one word, it is coded as if one word, such as "Ben Aron", is treated as "Benaron".
- Several letters and letter combinations pose the problem that they may sound in one of two ways. The letter and letter combinations CH, CK, C, J, and RS are assigned two possible code numbers.

Beider-Morse

The [Beider-Morse Phonetic Matching](#) (BMPM) is a very intelligent algorithm, compared to [Metaphone](#), whose purpose is to match names that are phonetically equivalent to the expected name. However, unlike [soundex](#) methods, the "sounds-alike" test is based not only on the spelling but on the linguistic properties of various languages.

The steps for comparison are as follows:

Step 1: Determines the language from the spelling of the name

The spelling of a name can include some letters or letter combinations that allow the language to be determined. Some examples are:

- "tsch", final "mann" and "witz" are specifically German
- final and initial "cs" and "zs" are necessarily Hungarian
- "cz", "cy", initial "rz" and "wl", final "cki", letters "Ś", "ł" and "ż" can be only Polish

Step 2: Applies phonetic rules to identify the language and translates the name into phonetic alphabets

Step 3: Calculating the Approximate Phonetic Value

Step 4: Calculating the Hebrew Phonetic Value

The entire process is described in detail in this [document](#)

Nearest Neighbor Methods

While key collisions methods are very fast, they tend to be either too strict or too lax with no way to fine-tune how much difference between strings we are willing to tolerate.

The [Nearest Neighbor](#) methods (also known as kNN), on the other hand, provide a parameter (the radius or k) which represents a distance threshold: any pair of strings that are closer than a certain value will be binned together.

Unfortunately, given n strings, there are $\frac{n(n-1)}{2}$ pairs of strings (and relative distances) that need to be compared and this turns out to be too slow even for small datasets (a dataset with 3000 rows require 4.5 million distance calculations!).

We have tried various methods to speed up this process but the one that works the best is called 'blocking' and is, in fact, a hybrid between key collision and kNN. This works by performing a first pass over the sequence of strings to evaluate and obtain 'blocks' in which all strings share a substring of a given 'blocking size' (which defaults to 6 chars in OpenRefine).

Blocking doesn't change the computational complexity of the kNN method but drastically reduces the number of strings that will be matched against one another (because strings are matched only inside the block that contains them). So instead of $\frac{n(n-1)}{2}$ we now have $\frac{nm(m-1)}{2}$ but n is the number of blocks and m is the average size of the block. In practice, this turns out to be dramatically faster because the block size is comparable to the number of strings and the blocks are normally much smaller. For example, for 3000 strings, you can have a thousand blocks composed of 10 strings each, which requires 45k distances to calculate instead of 4.5M!

If you're not in a hurry, OpenRefine lets you select the size of the blocking substring and

you can lower it down to 2 or 1 and make sure that blocking is not hiding a potential pair from your search... although in practice, anything lower than 3 normally turns out to be a waste of time.

All the above is shared between all the kNN methods, the difference of operation lies in the method used to evaluate the distance between the two strings.

For kNN distances, we found that blocking with less than 3 or 4 chars explodes the amount of time clustering takes and yields very few new valuable results, but your mileage may vary.

Levenshtein Distance

The [Levenshtein](#) distance (also known as "edit distance") is probably the simplest and most intuitive distance function between strings and is often still very effective due to its general applicability. It measures the minimal number of 'edit operations' that are required to change one string into the other.

It's worth noting that there are many flavors of edit-based distance functions (say, the [Damerau-Levenshtein distance](#), which considers 'transposition' as a single operation) but in practice, for clustering purposes, they tend to be equally functional (as long as the user has control over the distance threshold).

PPM

This distance is an implementation of [a seminal paper](#) about the use of the [Kolmogorov complexity](#) to estimate 'similarity' between strings and has been widely applied to the comparison of strings originating from DNA sequencing.

The idea is that because text compressors work by estimating the information content of a string, if two strings A and B are identical, compressing A or compressing A+B

(concatenating the strings) should yield very little difference (ideally, a single extra bit to indicate the presence of the redundant information). On the other hand, if A and B are very different, compressing A and compressing A+B should yield dramatic differences in length.

OpenRefine uses a normalized version of the algorithm, where the distance between A and B is given by

$$d(A, B) = \text{comp}(A+B) + \text{comp}(B+A) / (\text{comp}(A+A) + \text{comp}(B+B));$$

where `comp(s)` is the length of bytes of the compressed sequence of the string `s` and `+` is the append operator. This is used to account for deviation in the optimality of the given compressors.

While many different compressors can be used, the closer to Kolmogorov optimality they are (meaning, the better they encode) the more effective their result.

For this reason, we have used [Prediction by Partial Matching](#) as the compressor algorithm as it is one of the most effective compression algorithms for text and works by performing statistical analysis and predicting what character will come next in a string.

In practice, this method is very lax even for small radius values and tends to generate many false positives, but because it operates at a sub-character level it is capable of finding substructures that are not easily identifiable by distances that work at the character level. So it should be used as a 'last resort' clustering method; that's why it is listed last here despite its phenomenal efficacy in other realms.

It is also important to note that in practice similar distances are more effective on longer strings than on shorter ones; this is mostly an artifact of the need for the statistical compressors to 'warm up' and gather enough statistics to start performing well.

Cluster New Value Suggestions

For each cluster identified, one value is chosen as the initial 'New Cell Value' to use as the common value for all values in the cluster. The value chosen is the first value in the Cluster: see the `ClusteringDialog.prototype._updateData` function in [/main/webapp/modules/core/scripts/dialogs/clustering-dialog.js](#).

The first value in the Cluster is determined by two steps:

- a) The order of the items in the Cluster as the Cluster is built
- b) The order of the items in the Cluster after sorting by the count of the occurrences of each value

(a) is achieved via a Collections.sort - which is "[guaranteed to be stable: equal elements will not be reordered as a result of the sort.](#)"

(b) is achieved by different methods depending on whether you are doing a Nearest Neighbour or Key Collisions (aka Binning) cluster

If you are using Key Collision/Binning then the Cluster is created using a TreeMap which by default "[is sorted according to the natural ordering of its keys](#)".

The key is the string in the cell - so that means it will sort by the natural ordering of the strings in the cluster - which means that it uses a '[lexicographical order](#)' - basically based on the Unicode values in the string

If you are using the Nearest Neighbour sort the Cluster is created in a different way which is (as yet) undocumented. Testing indicates that it may be something like reverse natural ordering.

Contribute

- We've been focusing mostly on English content or data ported to English. We know some of the methods might be biased towards it but we're willing to introduce more methods once the OpenRefine community gathers more insights into these problems;

- OpenRefine's internals support a lot more methods but we have turned off many of them because they don't seem to have much practical advantage over the ones described here. If you think that OpenRefine should use other methods, feel free to suggest them to us because we might have overlooked them.

Suggested Reading

A lot of the code that OpenRefine uses for clustering originates from research done by the [SIMILE Project](#) at MIT which later [graduated as the Vicino project](#) ('vicino', pronounced "vitch-ee-no", means 'near' in Italian).

For more information on clustering methods and related research we suggest you look at the [bibliography of the Vicino project](#).

OpenRefine API

This is a generic API reference for interacting with OpenRefine's HTTP API.

NOTE: This protocol is subject to change without warning at any time (and has in the past) and is not versioned. Use at your own risk!

For OpenRefine 3.3 and later, all POST requests need to include a CSRF token as described [in the release notes](#)

Create project:

Command: *POST /command/core/create-project-from-upload*

When uploading files you will need to send the data as `multipart/form-data`. This is different to all other API calls which use a mixture of query string and POST parameters.

multipart form-data:

'project-file' : file contents 'project-name' : project name 'format' : format of data in project-file (e.g. 'text/line-based/*sv') [optional] 'options' : json object containing options relevant to the file format [optional - however, some importers may have required options, such as `recordPath` for the JSON & XML importers].

The formats supported will depend on the version of OpenRefine you are using and any Extensions you have installed. The common formats include:

- 'text/line-based': Line-based text files
- 'text/line-based/*sv': CSV / TSV / separator-based files [separator to be used in specified in the json submitted to the options parameter]

- 'text/line-based/fixed-width': Fixed-width field text files
- 'binary/text/xml/xls/xlsx': Excel files
- 'text/json': JSON files
- 'text/xml': XML files

If the format is omitted OpenRefine will try to guess the format based on the file extension and MIME type. The values which can be specified in the JSON object submitted to the 'options' parameter will vary depending on the format being imported. If not specified the options will either be guessed at by OpenRefine (e.g. separator being used in a separated values file) or a default value used. The import options for each file format are not currently documented, but can be seen in the OpenRefine GUI interface when importing a file of the relevant format.

If the project creation is successful, you will be redirected to a URL of the form:

```
http://127.0.0.1:3333/project?project=<project id>
```

From the project parameter you can extract the project id for use in future API calls. The content of the response is the HTML for the OpenRefine interface for viewing the project.

Get project models:

Command: *GET /command/core/get-models?*

'project' : project id

Recovers the models for the specific project. This includes columns, records, overlay models, scripting. In the columnModel a list of the columns is displayed, key index and name, and column groupings.

Response:

On success:

```
{  
  "columnModel":{  
    "columns": [  
      {  
        "cellIndex":0,  
        "originalName":"email",  
        "name":"email"  
      },  
      {  
        "cellIndex":1,  
        "originalName":"name",  
        "name":"name"  
      },  
      {  
        "cellIndex":2,  
        "originalName":"state",  
        "name":"state"  
      },  
      {  
        "cellIndex":3,  
        "originalName":"gender",  
        "name":"gender"  
      },  
      {  
        "cellIndex":4,  
        "originalName":"purchase",  
        "name":"purchase"  
      }  
    ],  
    "keyCellIndex":0,  
    "keyColumnName":"email",  
    "columnGroups": [  
    ]  
  }  
}
```

Rename Project or Change Metadata

Command: *POST /command/core/set-project-metadata*

Use this command to rename a project or change project metadata as described in [Project management](#).

In the form data

'project' : project id 'name': metadata field, one of: name, creator, contributors, subject, description, title, version, license, homepage, image, customMetadata 'value': metadata value

- To rename a project, use the `name` metadata field.
- To set custom metadata fields, use `customMetadata` and a JSON object `value`

Change Project Tags

Command: *POST /command/core/set-project-tags*

Tags are used to organize projects, see [Project management](#).

In the form data

'project' : project id 'old': tags to remove (comma-separated list) 'new': tags to add (comma-separated list)

Apply operations

Command: *POST /command/core/apply-operations?*

In the parameter

'project' : project id

In the form data

'operations' : Valid JSON **Array** of OpenRefine operations

Example of a Valid JSON **Array**

```
[  
  {  
    "op": "core/column-addition",  
    "description": "Create column zip type at index 15 based on  
column Zip Code 2 using expression grel:value.type()",  
    "engineConfig": {  
      "mode": "row-based",  
      "facets": []  
    },  
    "newColumnName": "zip type",  
    "columnInsertIndex": 15,  
    "baseColumnName": "Zip Code 2",  
    "expression": "grel:value.type()",  
    "onError": "set-to-blank"  
  },  
  {  
    "op": "core/column-addition",  
    "description": "Create column testing at index 15 based on  
column Zip Code 2 using expression grel:value.toString()0,5]",  
    "engineConfig": {
```

On success returns JSON response `{ "code" : "ok" }`

Export rows

Command: `POST /command/core/export-rows`

In the parameter

'project' : project id 'format' : format... (e.g 'tsv', 'csv')

In the form data

```
"engine" : JSON string... (e.g. '{"facets":[],"mode":"row-based"})'
```

Returns exported row data in the specified format. The formats supported will depend on the version of OpenRefine you are using and any Extensions you have installed. The common formats include:

- csv
- tsv
- xls
- xlsx
- ods
- html

Delete project

Command: `POST /command/core/delete-project`

'project' : project id...

Returns JSON response

Check status of async processes

Command: *GET /command/core/get-processes*

'project' : project id...

Returns JSON response

Get all projects metadata:

Command: *GET /command/core/get-all-project-metadata*

Recovers the meta data for all projects. This includes the project's id, name, time of creation and last time of modification.

Response:

```
{  
  "projects":{  
    "[project_id]":{  
      "name":"[project_name]",  
      "created":"[project_creation_time]",  
      "modified":"[project_modification_time]"  
    },  
    ...[More projects]...  
  }  
}
```

Expression Preview

Command: *POST /command/core/preview-expression*

Pass some expression (GREL or otherwise) to the server where it will be executed on selected columns and the result returned.

Parameters:

- **cellIndex:** *[column]* The cell/column you wish to execute the expression on.
- **rowIndices:** *[rows]* The rows to execute the expression on as JSON array. Example:
`[0,1]`
- **expression:** *[language]:[expression]* The expression to execute. The language can either be grel, jython or clojure. Example: grel:value.toLowerCase()
- **project:** *[project_id]* The project id to execute the expression on.
- **repeat:** *[repeat]* A boolean value (true/false) indicating whether or not this command should be repeated multiple times. A repeated command will be executed until the result of the current iteration equals the result of the previous iteration.
- **repeatCount:** *[repeatCount]* The maximum amount of times a command will be repeated.

Response:

On success:

```
{  
  "code": "ok",
```

The result array will hold up to ten results, depending on how many rows there are in the project that was specified by the [project_id] parameter. Each result is the string that would be put in the cell if the GREL command was executed on that cell. Note that any expression that would return an array or JSON object will be jsonized, although the output can differ slightly from the `jsonize()` function.

On error:

```
{  
  "code": "error",  
  "type": "[error_type]",  
  "message": "[error message]"  
}
```

Third-party software libraries

Libraries using the [OpenRefine API](#):

Python

- [refine-client-py](#)
- [openrefine-client](#), a fork of the above with an extended CLI
- [refine-python](#)

Ruby

- [refine-ruby](#)
 - The above is a maintained fork of [refine-ruby](#)
- [google_refine](#)

NodeJS

- [node-openrefine](#)

R

- [rrefine](#)

PHP

- [openrefine-php-client](#)

Java

- [refine-java](#)

Bash

- [bash-refine.sh](#) (templates for shell scripts)

Reconciliation API

This is a technical description of the mechanisms behind the reconciliation system in OpenRefine. For usage instructions, see [Reconciliation](#).

A reconciliation service is a web service that, given some text which is a name or label for something, and optionally some additional details, returns a ranked list of potential entities matching the criteria. The candidate text does not have to match each entity's official name perfectly, and that's the whole point of reconciliation-to get from ambiguous text name to precisely identified entities. For instance, given the text "apple", a reconciliation service probably should return the fruit apple, the Apple Inc. company, and New York city (also known as the Big Apple).

Entities are identified by strong identifiers in some particular identifier space. In the same identifier space, identifiers follow the same syntax. For example, given the string "apple", a reconciliation service might return entities identified by the strings "[Q89](#)", "[Q312](#)", and "[Q60](#)", in the Wikidata ID space. Each reconciliation service can only reconcile to one single identifier space, but several reconciliation services can reconcile to the same identifier space.

OpenRefine can connect to any reconciliation service which follows the [reconciliation API v0.2](#). This was formerly a specification edited by the OpenRefine project, which has now transitioned to its own [W3C Entity Reconciliation Community Group](#).

Informally, the main function of any reconciliation service is to find good candidates in the underlying database, given the following data:

- A string, which is normally the name or title of the entity, in some language.
- Optionally, a type which can be used to narrow down the search to entities of this type. OpenRefine does not define a particular set of acceptable types: this choice is left to the reconciliation service (see the suggest API for that).

- Optionally, a list of properties and their values, which can be used to refine the search. For instance, when reconciling a database of books, the author name or the publication date are useful bits of information that can be transferred to the reconciliation service. This information will be sent to the reconciliation service if the user binds columns to properties. Again, the notion of property is not predefined in OpenRefine: its definition depends on the reconciliation service.

In a sense, the reconciliation protocol is a standardized search API tailored to the specific needs of data matching. Beyond searching for candidate matches, it also comes with other features to help the user review and correct a matching (by offering previews and auto-completion for the target dataset).

See [the specifications of the protocol](#) for more details about it. You can suggest changes on its [issues tracker](#) or on the [group mailing list](#).

API versions supported by OpenRefine

There are multiple versions of the protocol available:

- [version 0.1](#), supported since OpenRefine 2.7. This version of the protocol is based on [JSONP](#), which represents a security risk. Therefore we discourage services to implement this version of the specifications.
- [version 0.2](#), supported since OpenRefine 3.3. This is the current stable version of the protocol, which we encourage services to implement.
- [the current draft of the next version](#), which is not supported by OpenRefine yet.

Create a new reconciliation service

If you want to create a reconciliation service for a new data source, we have resources to

help you. You can work from the specification and implement the API in your own service, but there are also other options.

- you can reuse an [existing library or framework to expose the required web API](#). Note that it is worth paying attention to which version of the specification they implement.
- you can also use a [standalone tool to expose a reconciliation service on top of a dataset](#).

In any case, you can use the [reconciliation test bench](#) to test your service interactively on some examples and validate its behaviour. Note that this page also features a list of known public reconciliation services: you could also add yours there if it is hosted publicly.

Writing extensions

Introduction

This is a very brief overview of the structure of OpenRefine extensions. For more detailed documentation and step-by-step guides please see the following external documentation/tutorials:

- Giuliano Tortoreto has [written documentation detailing how to build extension for OpenRefine](#)
- Owen Stephens has written [a guide to developing an extension which adds new GREL functions to OpenRefine](#).

OpenRefine makes use of a modified version of the [Butterfly framework](#) to provide an extension architecture. OpenRefine extensions are Butterfly modules. You don't really need to know about Butterfly itself, but you might encounter "butterfly" here and there in the code base.

Extensions that come with the code base are located under [the extensions sub-directory](#), but when you develop your own extension, you can put its code anywhere as long as you point Butterfly to it. That is done by any one of the following methods

- refer to your extension's directory in [the butterfly.properties file](#) through a `butterfly.modules.path` setting.
- specify the `butterfly.modules.path` property on the command line when you run OpenRefine. This overrides the values in the property file, so you need to include the default values first e.g.
`-Dbutterfly.modules.path=modules, ../../extensions,/path/to/your/extension`

Please note that you should bundle any dependencies yourself, so you are insulated from OpenRefine packaging changes over time.

Directory Layout

A OpenRefine extension sits in a file directory that contains the following files and sub-directories:

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

The file named `module.properties` (see [example](#)) contains the extension's metadata. Of importance is the name field, which gives the extension a name that's used in many other places to refer to it. This can be different from the extension's directory name.

```
name = my-extension-name
```

Your extension's client-side resources (.html, .js, .css files) stored in the `module/` sub-directory will be accessible from <http://127.0.0.1:3333/extension/my-extension-name/> when OpenRefine is running.

Also of importance is the dependency

```
requires = core
```

which makes sure that the core module of OpenRefine is loaded before the extension attempts to hook into it.

The file named `controller.js` is responsible for registering the extension's hooks into OpenRefine. Look at the sample-extension extension's `controller.js` file for an example. It should have a function called `init()` that does the hook registrations.

The `pom.xml` file is an [Apache Maven](#) build file. You can make a copy of the sample extension's `pom.xml` file to get started. The important point here is that the Java classes should be built into the `module/MOD-INF/classes` sub-directory.

Note that your extension's Java code would need to reference some libraries used in OpenRefine and OpenRefine's Java classes themselves. These dependencies are reflected in the Maven configuration for the extension.

Sample extension

The sample extension is included in the code base so that you can copy it and get started on writing your own extension. After you copy it, make sure you change its name inside its `module/MOD-INF/controller.js` file.

Basic Structure

The sample extension's code is in `refine/extensions/sample/`. In that directory, Java source code is contained under the `src` sub-directory, and webapp code is under the `module` sub-directory. Here is the full directory layout:

```
refine/extensions/sample/
    build.xml (ant build script)
    src/
        com/google/refine/sampleExtension/
            ... Java source code ...
```

The sub-directory `MOD-INF` contains the Butterfly module's metadata and is what Butterfly looks for when it scans directories for modules. `MOD-INF` serves similar functions as `WEB-INF` in other web frameworks.

Java code is built into the sub-directory `classes` inside `MOD-INF`, and supporting external Java jars are in the `lib` sub-directory. Those will be automatically loaded by Butterfly. (The `build.xml` script is wired to compile into the `classes` sub-directory.)

Client-side code is in the inner `module` sub-directory. They can be plain old `.html`, `.css`, `.js`, and image files. There are also Velocity `.vt` files, but they need to be routed inside `MOD-INF/controller.js`.

`MOD-INF/controller.js` lets you configure the extension's initialization and URL routing in Javascript rather than in Java. For example, when the requested URL path is either `/` or an empty string, we process and return `MOD-INF/index.vt` (see <http://127.0.0.1:3333/extension/sample/> if OpenRefine is running).

The `init()` function in `controller.js` allows the extension to register various client-side handlers for augmenting pages served by Refine's core. These handlers are feature-specific. For example, [this is where the python extension adds its parser](#). As for the sample extension, it adds its script `project-injection.js` and style `project-injection.css` into the `/project` page. If you [view the source of the /project page](#), you will see references to those two files.

Wiring Up the Extension

The Extensions are loaded by the Butterfly framework. Butterfly refers to these as 'modules'. [The location of modules is set in the main/webapp/butterfly.properties file](#). Butterfly simply descends into each of those paths and looks for any `MOD-INF` directories.

For more information, see [Extension Points](#).

Extension points

Client-side: Javascript and CSS

The UI in OpenRefine for working with a project is coded in [the /main/webapp/modules/core/project.vt file](#). The file is quite small, and that's because almost all of its content is to be expanded dynamically through the Velocity variables `$scriptInjection` and `$styleInjection`. So that your own Javascript and CSS files get loaded, you need to register them with the `ClientSideResourceManager`, which is done in the `/module/MOD-INF/controller.js` file. See [the controller.js file in this sample extension code](#) for an example.

In the registration call, the variable `module` is already available to your code by default, and it refers to your own extension.

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

You can specify one or more files for registration, and their paths are relative to the `module` sub-directory of your extension. They are included in the order listed.

Javascript Bundling: Note that `project.vt` belongs to the core module and is thus under the control of the core module's [controller.js file](#). The Javascript files to be included in `project.vt` are by default bundled together for performance. When debugging, you can prevent this bundling behavior by setting `bundle` to `false` near the top of that `controller.js` file. (If you have commit access to this code base, be sure not to check that

change in.)

Client-side: Images

We recommend that you always refer to images through your CSS files rather than in your Javascript code. URLs to images will thus be relative to your CSS files, e.g.,

```
.foo {  
    background: url(../images/x.png);  
}
```

If you really really absolutely need to refer to your images in your Javascript code, then look up your extension's URL path in the global Javascript variable `ModuleWirings`:

```
ModuleWirings["my-extension"] + "images/x.png"
```

Client-side: HTML Templates

Beside Javascript, CSS, and images, your extension might also include HTML templates that get loaded on the fly by your Javascript code and injected into the page's DOM. For example, here is [the Cluster edit dialog template](#), which gets loaded by code in [the equivalent javascript file 'clustering-dialog.js'](#):

```
var dialog = $(DOM.loadHTML("core", "scripts/dialogs/clustering-  
dialog.html"));
```

`DOM.loadHTML` returns the content of the file as a string, and `$(...)` turns it into a DOM fragment. Where `"core"` is, you would want your extension's name. The path of the HTML file is relative to your extension's `module` sub-directory.

Client-side: Project UI Extension Points

Getting your extension's Javascript code included in `project.vt` doesn't accomplish much by itself unless your code also registers hooks into the UI. For example, you can surely implement an exporter in Javascript, but unless you add a corresponding menu command in the UI, your user can't use your exporter.

Main Menu

The main menu can be extended by calling any one of the methods `MenuBar.appendTo`, `MenuBar.insertBefore`, and `MenuBar.insertAfter`. Each method takes 2 arguments: an array of strings that identify a particular existing menu item or submenu, and one new single menu item or submenu or an array of menu items and submenus. For example, to insert 2 menu items and a menu separator before the menu item Project > Export Filtered Rows > Templating..., write this Javascript code wherever that would execute when your Javascript files get loaded:

```
MenuBar.insertBefore(
    ["core/project", "core/export", "core/export-templating"],
    [
        {
            "label": "Menu item 1",
            "click": function() { ... }
        },
        {
            "label": "Menu item 2",
            "click": function() { ... }
        },
        {} // separator
    ]
);
```

The array `["core/project", "core/export", "core/export-templating"]` pinpoints the reference menu item.

See the beginning of [/main/webapp/modules/core/scripts/project/menu-bar.js](#) for IDs of menu items and submenus.

Column Header Menu

The drop-down menu of each column can also be extended, but the mechanism is slightly different compared to the main menu. Because the drop-down menu for a particular column is constructed on the fly when the user actually clicks the drop-down menu button, extending the column header menu can't really be done once at start-up time, but must be done every time a column header menu gets created. So, registration in this case involves providing a function that gets called each such time:

```
DataTableColumnHeaderUI.extendMenu(function(column, columnHeaderUI, menu)
{ ... do stuff to menu ... });
```

That function takes in the column object (which contains the column's name), the column header UI object (generally not so useful), and the menu to extend. In the previous code line where it says "do stuff to menu", you can write something like this:

```
MenuSystem.appendTo(menu, ["core/facet"], [
    {
        id: "core/text-facet",
        label: "My Facet on " + column.name,
        click: function() {
            ... use column.name and do something ...
        }
    },
]);
```

In addition to `MenuSystem.appendTo`, you can also call `MenuSystem.insertBefore` and `MenuSystem.insertAfter` which take the same 3 arguments. To see what IDs you can use, see the function `DataTableColumnHeaderUI.prototype._createMenuForColumnHeader` in [/main/webapp/modules/core/scripts/views/data-table/column-header-ui.js](#).

Cell renderers

From OpenRefine 3.7 onwards, extensions can also customize the way cells are rendered. This is done by registering a renderer, which is responsible for transforming the JSON representation of a cell into DOM elements rendering it:

```
class MyCellRenderer {  
  
    constructor() {  
        super();  
        // some initialization code can be added here  
    }  
  
    render(  
        rowIndex, // the 0-based row index  
        cellIndex, // the position of the column to which the cell belongs  
        cell, // the deserialized JSON representation of the cell  
        cellUI // the parent CellUI object which called this renderer  
    ) {  
        // this renderer has the opportunity to return a DOM element  
        // representing the cell, as follows:  
        return $('rendered cell</span>');  
        // or it may not return anything, in which case the next cell  
        // renderer will be executed  
    }  
}
```

OpenRefine holds an ordered list of cell renderers in its `CellRendererRegistry`. To render a cell, OpenRefine will execute each cell renderer in order, until the first renderer which returns a DOM element. The following renderers are not executed and that DOM element is used as the cell representation.

The registration of the renderer is done with

```
CellRendererRegistry.addRenderer(  
    'my-renderer-identifier', // a string identifying our new renderer
```

The default renderers available in OpenRefine itself can be found in [main/webapp/modules/core/scripts/views/data-table/cell-renderers/registry.js](#).

Server-side: Ajax Commands

The client-side of OpenRefine gets things done by calling AJAX commands on the server-side. These commands must be registered with the OpenRefine servlet, so that the servlet knows how to route AJAX calls from the client-side. This can be done inside the `init` function in your extension's `controller.js` file, e.g.,

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

Your command will then be accessible at <http://127.0.0.1:3333/command/my-extension/my-command>.

Server-side: Operations

Most commands change the project's data. Most of them do so by creating abstract operations. See the Changes, History, Processes, and Operations section of the [Server Side Architecture](#) document.

You can register an operation `class` in the `init` function as follows:

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

Do not call `new` to construct an operation instance. You must register the class itself. The class should have a static function for reconstructing an operation instance from a JSON blob:

```
static public AbstractOperation reconstruct(Project project, JSONObject
obj) throws Exception {
    ...
}
```

Server-side: GREL

GREL can be extended with new functions. This is also done in the `init` function in `controller.js`, e.g.,

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

You might also want to provide new variables (beyond just `value`, `cells`, `row`, etc.) available to expressions. This is done by registering a binder that implements the interface `com.google.refine.expr.Binder`:

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

Server-side: Importers

You can register an importer as follows:

```
Packages.com.google.refine.importers.ImporterRegistry.registerImporter(
    "importer-name", new Packages.com.foo.bar.MyImporter());
```

The string `"importer-name"` isn't important at all. It's not really related to file extension or

mime-type. Just use something unique. Your importer will be explicitly called to test if it can import something.

Server-side: Exporters

You can register an exporter as follows:

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

The string `"exporter-name"` isn't important at all. It's only used by the client-side to tell the server-side which exporter to use. Just use something unique and, of course, relevant.

Server-side: Overlay Models

Overlay models are objects attached onto a core Project object to store and manage additional data for that project. For example, the schema alignment skeleton is managed by the Protograph overlay model. An overlay model implements the interface

`com.google.refine.model.OverlayModel` and can be registered like so:

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

Note that you register the **class**, not an instance. The class should implement the following static method for reconstructing an overlay model instance from a JSON blob:

```
static public OverlayModel reconstruct(JSONObject o) throws JSONException  
{  
    ...  
}
```

When the project gets saved, the overlay model instance's `write` method will be called:

```
public void write(JSONWriter writer, Properties options) throws  
JSONException {  
    ...  
}
```

Server-side: Scripting Languages

A scripting language (such as Jython) can be registered as follows:

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

The first string is the prefix that gets prepended to each expression so that we know which language the expression is in. This should be short, unique, and identifying. The second string is a user-friendly name of the language. The third is an object that implements the interface `com.google.refine.expr.LanguageSpecificParser`. The final string is the default expression in that language that would return the cell's value.

In 2018 we are making important changes to OpenRefine to modernize it, for the benefit of users and contributors. This page describes the changes that impact developers of extensions or forks and is intended to minimize the effort required on their end to follow the transition. The instructions are written specifically with extension maintainers in mind, but fork maintainers should also find it useful.

This document describes the migrations in the order they are committed to the master branch. This means that it should be possible to perform each migration in turn, with the ability to run the software between each stage by checking out the appropriate git commit.

Migrating older extensions

Migrating from Ant to Maven

Why are we doing this change?

Ant is a fairly old (antique?) build system that does not incorporate any dependency management. By migrating to Maven we are making it easier for developers to extend OpenRefine with new libraries, and stop having to ship dozens of .jar files in the repository. Using the Maven repository also encourages developers to add dependencies to released versions of libraries instead of custom snapshots that are hard to update.

When was this change made?

The migration was done between 3.0 and 3.1-beta with this commit: <https://github.com/OpenRefine/OpenRefine/commit/47323a9e750a3bc9d43af606006b5eb20ca397b8>

How to migrate an extension

You will need to write a `pom.xml` in the root folder of your extension to configure the compilation process with Maven. Sample `pom.xml` files for extensions can be found in the extensions that are shipped with OpenRefine (`gdata`, `database`, `jython`, `pc-axis` and `wikidata`). A sample extension (`sample`) is also provided, with a minimal build file.

For any library that your extension depends on, you should try to find a matching artifact in the Maven Central repository. If you can find such an artifact, delete the `.jar` file from your extension and add the dependency in your `pom.xml` file. If you cannot find such an

artifact, it is still possible to incorporate your own `.jar` file using `maven-install-plugin` that you can configure in your `pom.xml` file as follows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-install-plugin</artifactId>
  <version>2.5.2</version>
  <executions>
    <execution>
      <id>install-wdtk-datamodel</id>
      <phase>process-resources</phase>
      <configuration>
        <file>${basedir}/lib/my-proprietary-
library.jar</file>
        <repositoryLayout>default</repositoryLayout>
        <groupId>com.my.company</groupId>
        <artifactId>my-library</artifactId>
        <version>0.5.3-SNAPSHOT</version>
        <packaging>jar</packaging>
        <generatePom>true</generatePom>
      </configuration>
      <goals>
        <goal>install-file</goal>
      </goals>
    </execution>
    <!-- if you need to add more than one jar, add more
execution blocks here -->
  </executions>
</plugin>
```

And add the dependency to the `<dependencies>` section as usual:

```
com.my.company my-library 0.5.3-SNAPSHOT
```

Migrating to Wikimedia's i18n jQuery

plugin

Why are we doing this change?

This adds various important localization features, such as the ability to handle plurals or interpolation. This also restores the language fallback (displaying strings in English if they are not available in the target language) which did not work with the previous set up.

When was the migration made?

The migration was made between 3.1-beta and 3.1, with this commit: <https://github.com/OpenRefine/OpenRefine/commit/22322bd0272e99869ab8381b1f28696cc7a26721>

How to migrate an extension

You will need to update your translation files, merging nested objects in one global object, concatenating keys. You can do this by running the following Python script on all your JSON translation files:

```
import json import sys

with open(sys.argv[1], 'r') as f: j = json.loads(f.read())

result = def translate(obj, path): res = if type(obj) == str: result['/'.join(path)] = obj else: for k, v in obj.items(): new_path = path + [k] translate(v, new_path)

translate(j, [])
```

```
with open(sys.argv[1], 'w') as f: f.write(json.dumps(result, ensure_ascii=False, indent=4))
```

Then your javascript files which retrieve the translated strings should be updated:

`$.i18n._('core-dialogs')['cancel']` becomes `$.i18n('core-dialogs/cancel')`.

You can do this with the following `sed` script:

```
sed -i "s/\.i18n\.\([""]\([A-Za-z0-9\^-\]\)["]\)\[""]\([A-Za-z0-9_]\)["]\]/\.$i18n(\\"1\\2")/g"  
my_javascript_file.js
```

You can then chase down the places where you are concatenating translated strings, and replace that with more flexible patterns using [the plugin's features](#).

Migrating from org.json to Jackson

Why are we doing this change?

The org.json (or json-java) library has multiple drawbacks.

- First, it has limited functionality - all the serialization and deserialization has to be done explicitly - an important proportion of OpenRefine's code was dedicated to implementing these;
- Second, its implementation is not optimized for speed - multiple projects have reported speedups when migrating to more modern JSON libraries;
- Third, and this was the decisive factor to initiate the migration: [its license](#) is the MIT license with an additional condition which makes it non-free. Getting rid of this dependency was required by the Software Freedom Conservancy as a prerequisite to become a fiscal sponsor for the project.

When was the migration made?

This change was made between 3.1 and 3.2-beta, with this commit: <https://github.com/OpenRefine/OpenRefine/commit/5639f1b2f17303b03026629d763dcb6fef98550b>

How to migrate an extension or fork

You will need to use the Jackson library to serialize the classes that implement interfaces or extend classes exposed by OpenRefine. The interface `Jsonizable` was deleted. Any class that used to implement this now needs to be serializable by Jackson, producing the same format as the previous serialization code. This applies to any operation, facet, overlay model or GREL function. If you are new to Jackson, have a look at [this tutorial](#) to learn how to annotate your class for serialization. Once this is done, you can remove the `void write(JSONWriter writer, Properties options)` method from your class.

Note that it is important that you do this migration for all classes implementing the `Jsonizable` interface that are exposed to OpenRefine's core.

We encourage you to migrate out of org.json completely, but this is only required for the classes that interact with OpenRefine's core.

General notes about migrating

OpenRefine's ObjectMapper is available at `ParsingUtilities.mapper`. It is configured to only serialize the fields and getters that have been explicitly marked with `@JsonProperty` (to avoid accidental JSON format changes due to refactoring). On deserialization it will ignore any field in the JSON payload that does not correspond to a field in the Java class. It has serializers and deserializers for `OffsetDateTime` and `LocalDateTime`.

Useful snippets to use in tests:

- deserialize an instance: `MyClass instance = ParsingUtilities.mapper.readValue(jsonString, MyClass.class);` (replaces calls to `Jsonizable.write`);
- serialize an instance: `String json = ParsingUtilities.mapper.writeValueAsString(myInstance);` (replaces calls to static methods such as `load`, `loadStreaming` or `reconstruct`);
- the equivalent of `JSONObject` is `ObjectNode`, the equivalent of `JSONArray` is `ArrayNode`;
- create an empty JSON object: `ParsingUtilities.mapper.createObjectNode()` (replaces `new JSONObject()`);
- create an empty JSON array: `ParsingUtilities.mapper.createArrayNode()` (replaces `new JSONArray()`).

Before undertaking the migration, we recommend that you write some tests which serialize and deserialize your objects. This will help you make sure that the JSON format is preserved during the migration. One way to do this is to collect some sample JSON representations of your objects, and check in your tests that deserializing these JSON payloads and serializing them back to JSON preserves the JSON payload. Some utilities are available to help you with that in `TestUtils` (we had [some to test org.json serialization](#) before we got rid of the dependency, feel free to copy them).

For functions

Before the migration, you had to explicitly define JSON serialization of functions with a `write` method. You should now override the getters returning the various documentation fields.

Example: `Cos` function [before](#) and [after](#).

For operations

Before the JSON migration we refactored engine-dependent operations so that the engine

configuration is represented by an `EngineConfig` object instead of a `JSONObject`. Therefore the constructor for your operation should be updated to use this new class. Your constructor should also be annotated to be used during deserialization.

Note that you do not need to explicitly serialize the operation type, this is already done for you by `AbstractOperation`.

Example: `ColumnRemovalOperation` [before](#) and [after](#).

For changes

Changes are serialized in plain text but often relies on JSON serialization for parts of the data. Just use the methods above with `ParsingUtilities.mapper` to maintain this behaviour.

Example: `ReconChange` [before](#) and [after](#).

For importers

The importing options have been migrated from `JSONObject` to `ObjectNode`. Your compiler should help you propagate this change. Utility functions in `JSONUtilities` have been migrated to Jackson so you should have minimal changes if you used them.

Example: `TabularImportingParserBase` [before](#) and [after](#).

For overlay models

Migrate serialization and deserialization as for other objects.

Example: `wikibaseSchema` [before](#) and [after](#)

For preference values

Any class that is stored in OpenRefine's preference now needs to implement the `com.google.refine.preferences.PreferenceValue` interface. The static `load`

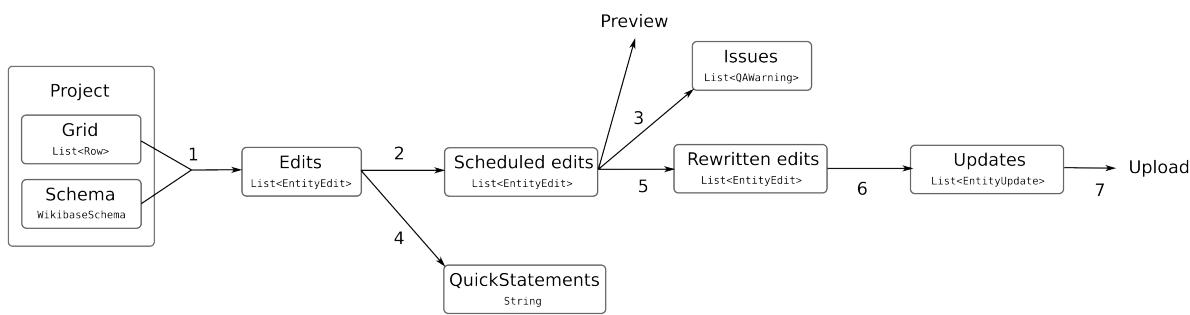
method and the `write` method used previously for deserialization should be deleted and regular Jackson serialization and deserialization should be implemented instead. Note that you do not need to explicitly serialize the class name, this is already done for you by the interface.

Example: `TopList` before and after

Overview of the Wikibase extension

The Wikibase extension provides upload functionalities into [Wikibase](#) instances, by helping the user transform data from a tabular format to [the data model of Wikibase entities](#).

The following graph gives an overview of the editing pipeline:



1. The schema is evaluated ([WikibaseSchema::evaluate](#)) on each visible row in the project. This gives rise to one or more [EntityEdit](#) objects for each row. Those objects represent a candidate edit, including the configuration of how this edit should be matched to any existing data on the entity to be edited.
2. The [WikibaseAPIScheduler](#) groups the edits together, so that they can later be performed with the HTTP Wikibase API efficiently. This essentially means bundling up all changes made to the same entity across the project into a single object. If new edits are made, those edits also need to be ordered so that any new entity referenced in an edit already exists by the time this edit is made.
3. A collection of [EditScrutinizer](#)s are executed on the candidate edits, to check for detectable issues about those. This generates [QAWarning](#) objects which are aggregated according to keys defined by the scrutinizers. This collection of scrutinizers is largely based on the constraint system offered by

[WikibaseQualityConstraints](#), when it is used by the target Wikibase.

The steps above are executed every time the schema is previewed. When the user is ready to upload their edits to Wikibase, they have a choice between two routes:

4. Going through [QuickStatements](#). The [QuickStatementsScheduler](#) offers another way to group edits together, such that they can be expressed in the QuickStatements v1 format. The [QuickStatementsExporter](#) translates them to that format.

They can also upload their edits directly from OpenRefine, in which case the following operations are executed for each edit:

5. The edit is rewritten, so that if it references any newly created Wikibase entity, the identifier of this entity is inserted in the edit. This is done by [ReconEntityRewriter](#).
6. For edits to existing entities, the edit is compared to any existing data on the item. Following the matching criteria stored in the edit, we generate a concrete update of entity data. This is done by [EntityEdit::toUpdate](#).
7. Finally, the update is sent to the Wikibase instance using [Wikidata-Toolkit](#).

Guidelines for maintaining OpenRefine

This page describes our practices to review issues and pull requests in the OpenRefine project.

Reviewing issues

When people create new issues, they automatically get assigned [the "to be reviewed" tag](#).

Ideally, for each of these issues, someone familiar with OpenRefine (not necessarily a developer!) should read the issue and try to determine if there is a genuine bug to fix, or if the enhancement request is legitimate. In those cases, we can remove the "to be reviewed" tag and leave the issue open. In the others, the issue should be politely closed.

Bugs

For a bug, we should first check if it is a real unexpected behaviour or if just comes from a misunderstanding of the intended behaviour of the tool (which could suggest an improvement to the documentation). Then, if it sounds like a genuine problem, we need to check if it can be reproduced independently on the master branch. If the issue does not give enough details about the bug to reproduce it on master, mark it as "not reproducible" and ask the reporter for more information. After some time without any information from the reporter, we can close the issue.

Enhancement requests

For an enhancement, we need to make a judgment call of whether the proposed functionality is in the scope of the project. There is no universal rule for this of course, so just use your own intuition: do you think this would improve the tool? Would it be consistent with the spirit of the project? Trust your own opinion - if people disagree, they can have a discussion on the issue.

Tagging good first issues

Adding [the "good first issue" tag](#) is something that requires a bit more familiarity with the development process. This tag is used by GitHub to showcase issues in some project lists and we point interested potential contributors to it. It is therefore important that tackling these issues gives them a nice onboarding experience, with as few hurdles as possible.

Developers should add the "good first issue" tag when they are confident that they can provide a good pull request for the issue with at most a few hours of work. Also, solving the issue should not require any difficult design decision. The issue should be uncontentious: it should be clear that the proposed solution should be accepted by the team.

Reviewing pull requests

Process

1. A committer reviews the PR to check for the requirements below and tests it. Each PR should be linked to one or more corresponding issues and the reviewer should check

that those are correctly addressed by the PR. The reviewer should be someone else than the PR author. For PRs with an important impact or contentious issues, it is important to leave enough time for other contributors to give their opinion.

2. The reviewer merges the pull request by squashing its commits into one (except for Weblate PRs which should be merged without squashing).
3. The reviewer adds the linked issues to the milestone for the next release (such as [the 3.5 milestone](#))
4. If the change is worth noting for users or developers, the reviewer adds an entry in the changelog for the next release (such as [Changes for 3.5](#))

Requirements

Code style

Currently, only our code style for integration tests (using Cypress) is codified and enforced by the CI. For the rest, we rely on imitating the surrounding code. [We should decide on a code style and check it in the CI for other areas of the tool.](#)

Testing

We currently rely have two sorts of tests:

- Backend tests, in Java, written with the TestNG framework. Their granularity varies, but generally speaking they are unit tests which test components in isolation.
- UI tests, in Javascript, written with the Cypress framework. They are integration tests which test both the frontend and the backend at the same time.

Changes to the backend should generally come with the accompanying TestNG tests. Functional changes to the UI should ideally come with corresponding Cypress tests as

well.

Those tests should be supplied in the same PR as the one that touches the product code.

Documentation

Changes to user-facing functionality should be reflected in the docs. Those documentation changes should happen in the same PR as the one that touches the product code.

UI style

We do not have formally defined UI style guidelines. Contributors are invited to imitate the existing style.

Licensing and dependencies

Dependencies can only be added if they are released under a license that is compatible with our BSD Clause-3 license. One should pay attention to the size of the dependencies since they inflate the size of the release bundles.

Continuous integration

The various check statuses reported by our continuous integration suite should be green.

Special pull requests

Weblate PRs

Weblate PRs should not be squashed as it prevents Weblate from recognizing that the corresponding changes have been made in master. They should be merged without squashing.

Reviewing Weblate PRs only amounts to a quick visual sanity check as maintainers are not

expected to master the languages involved. If corrections need to be made, they should be done in Weblate itself.

Dependabot PRs

When reviewing a Dependabot PR it is generally useful to pay attention to:

- the type of version change: most libraries follow the "semver" versioning convention, which indicates the nature of the change.
- the library's changelog, especially if the version change is more significant than a patch release

When releasing a new version of Refine, the following steps should be followed:

1. Make sure the `master` branch is stable and nothing has broken since the previous version. We need developers to stabilize the trunk and some volunteers to try out `master` for a few days.
2. Change the version number in `RefineServlet.java` and in the POM files using `mvn versions:set -DnewVersion=2.6-beta -DgenerateBackupPoms=false`. Commit the changes.
3. Compose the list of changes in the code and on the wiki. If the issues have been updated with the appropriate milestone, the Github issue tracker should be able to provide a good starting point for this.
4. Tag the release candidate in git and push the tag to Github. For example:

```
git tag -a -m "Second beta" 2.6-beta.2  
git push origin --tags
```

5. Create a GitHub release based on that tag, with a summary of the changes as description of the release. Publishing the GitHub release will trigger the generation of the packaged artifacts, which will be added to the release on GitHub.
6. Once the artifacts are ready, update the `releases.json` file in the `openrefine.org` repository to reflect this new version. Verify that the correct versions are shown at <http://openrefine.org/download>
7. Update the OpenRefine Homebrew cask or coordinate an update via the developer forum
8. Announce on the [announcements section of the OpenRefine forum](#), or even on the [blog](#) if this is a major release (the blog is imported automatically into the announcements category of the forum)
9. Update the version in master to the next version number with `-SNAPSHOT` (such as `4.3-SNAPSHOT`), in the same places as in step 2.

Apple code signing

We have code signing certificates for our iOS distributions. Those are available in our build environment and you should not need to import them to your own machine to make a release (since you are not building the release yourself). But if the signing process fails, it can be useful to import the certificates on your own machine to debug the process. To do so:

- Request advisory.committee@openrefine.org to be added to the Apple team: you need to provide the email address that corresponds to your AppleID account;
- Create a certificate signing request from your Mac: <https://help.apple.com/developer-account/#/devbfa00fef7>
- Go to <https://developer.apple.com/account/resources/certificates/add> and select "Apple Distribution" as certificate type
- Upload the certificate signing request in the form
- Download the generated certificate
- Import this certificate in the "Keychain Access" app on your mac
- The signing workflow can be found in [.github/workflows/snapshot_release.yml](#)

Maintaining OpenRefine's Homebrew cask

Homebrew is a popular command-line package manager for macOS. Once Homebrew is installed, OpenRefine can be installed via the simple command, `brew install openrefine`. OpenRefine's presence on Homebrew is found in the Homebrew Cask repository project, as a "cask", at <https://github.com/Homebrew/homebrew-cask/blob/HEAD/Casks/openrefine.rb>.

Terminology: "Homebrew Cask" is the segment of Homebrew where pre-built binaries and GUI applications go, whereas the original "Homebrew" project is reserved for command-line utilities that can be built from source. Because the macOS version of OpenRefine is released as an app bundle with GUI components, it is handled as a Homebrew Cask.

When there is a new release of OpenRefine, registering the new release with Homebrew can be easily accomplished using Homebrew's `brew bump-cask-pr` command. Full directions for this utility as well as procedures for more complex PRs can be found on [the Homebrew Cask CONTRIBUTING page](#), but, a simple version bump is a one-line command. For example, to update Homebrew's version of OpenRefine to 3.4.1, use this command:

```
brew bump-cask-pr --version 3.4.1 openrefine
```

This command will cause your local Homebrew installation to download the new version of OpenRefine, calculate the installer's new SHA-256 fingerprint value, and construct a pull request under your GitHub account. Once the pull request is submitted, continuous integration tests will run, and a member of the Homebrew community will review the PR. At times there is a backlog on the CI servers, but once tests pass, the community review is typically completed in a matter of hours.

Note: It is important that the OpenRefine release tag and version number are identical, so that the Homebrew cask can find the installer's URL. This is because Homebrew's cask for OpenRefine uses the following formula for constructing the URL to OpenRefine's installer for a given release:

```
https://github.com/OpenRefine/OpenRefine/releases/  
download/#{version}/openrefine-mac-#{version}.dmg
```

That is, when you tell Homebrew that OpenRefine is now at version [3.4.1](#), Homebrew will try to download the OpenRefine installed from the following URL:

```
https://github.com/OpenRefine/OpenRefine/releases/download/3.4.1/  
openrefine-mac-3.4.1.dmg
```

Introduction

Welcome to the OpenRefine design contribution documentation!

This documentation is a comprehensive guide to contributing and collaborating on design for OpenRefine. It covers everything from joining the community to creating and implementing design changes.

We invite designers to help with usability issues (UX/UI), aesthetics and visual design input, and more.

The goal of this documentation is to help you make impactful contributions to OpenRefine's user experience. We emphasize the importance of open communication, feedback, and collaboration with developers.

Whether you're a seasoned designer or new to open source, we believe that your contributions can make a significant impact on OpenRefine.

Get involved

Join the OpenRefine community

Here's how you can take your first steps to become a part of the OpenRefine design team:

OpenRefine's communication channels are the heart of our collaboration. Join the [OpenRefine forum](#), where discussions, announcements, and design-related topics are shared. It is a great place to ask questions, get help, and meet other OpenRefine users and developers.

Once you've joined the forum, take a moment to introduce yourself in the [introduction thread](#). Let us know about your background, interests, and what aspects of design you're passionate about. This is a great way for us to get to know you better and for you to connect with fellow community members.

Check out the available avenues for contributing to the design team. Whether it's enhancing the core design elements, creating visual assets, or collaborating on user experience improvements, carrying out user testing or research, there's a role for every design enthusiast.

Tools

Figma

[Figma](#) is a popular web-based design tool, which is used to document the OpenRefine design system. You don't have to use it to contribute to the OpenRefine project, but you

may want to take advantage of the possibilities to reuse design components in your design work, too. To do that you will need to create a free account, if you don't already have one. If you need guidance, explore [Figma's Help Center](#) for tutorials and more to master Figma's features.

GitHub

GitHub serves as the primary issue tracker for OpenRefine, hosting its codebase. It's worth noting that GitHub's significance extends beyond developers, as users, educators and designers also contribute significantly.

You can create a free account on [Github.com](#), if you don't already have one. To learn more on how designers can use GitHub, the article [Git for Designers – All You Need to Know](#) by [UXPIN](#) offers an extensive breakdown of Github.

OpenRefine design system

The [OpenRefine Design System](#) was created by careful analysis of the tool's interactive components, providing helpful reference for designers and contributors regarding the styling, application and integration of components in different user workflow scenarios. Designed to ensure consistency and usability, it also serves as a normative document. Use the page navigation in the Figma document to check guidelines for colors, typography styles, individual buttons, complete dialogs and interaction sequences.

Accessing the OpenRefine design system

Duplicate for edit access: The OpenRefine Design System is accessible to all for viewing. For edit access, duplicate the file to your drafts. This empowers you to make changes and contribute. You can also copy components to your own Figma files.

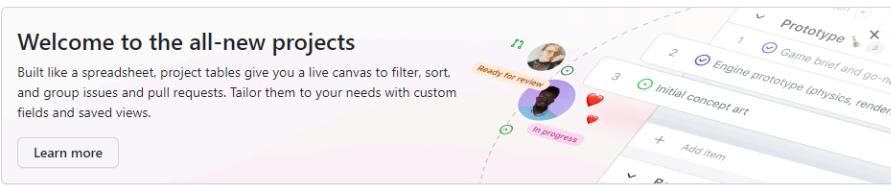
Request edit access: If you're an active contributor, you might be granted direct edit access. Feel free to inquire in the development and design category channel on the OpenRefine forum.

Design contribution workflows

Identifying design issues

The OpenRefine project contains several repositories, but you're most likely to find design-related issues in the main application repository here: <https://github.com/OpenRefine/OpenRefine>. Alternatively, further design issues might be linked to the main [OpenRefine website repository](#), or some additional extensions.

Within OpenRefine's GitHub, the [OpenRefine Design Project](#) is the main entry point for designers to identify and select suitable issues to work on.



The screenshot shows the GitHub Projects interface for the 'OpenRefine Design' project. At the top, there's a search bar with 'isopen' and a 'New project' button. Below it, a section titled 'Welcome to the all-new projects' describes the feature as a live canvas for filtering, sorting, and grouping issues. A 'Learn more' button is present. The main area displays a board with cards representing issues. One card is highlighted with a pink border: '#1 OpenRefine Design' (updated 2 days ago), which is described as 'This project is meant to gather and organize UX and UI design across the whole OpenRefine project'. Other visible cards include '#8 Improving OpenRefine's design workflows' (updated on Jun 9), '#7 Ayushi's reconciliation improvements' (updated last week), '#6 Improving OpenRefine's reproducibility' (updated on Jul 17), and '#2 Structured Data on Commons' (updated last week). Each card has a three-dot menu icon on the right.

The OpenRefine design team uses GitHub Projects to effectively manage and track design-related issues across all relevant repositories. GitHub Projects allows us to group

issues that have been tagged with a design-related theme and track progress from start to finish.

The screenshot shows the OpenRefine interface with the following details:

- Project:** OpenRefine / Projects / OpenRefine Design
- View:** Table
- Issues:** 17 items listed in a table format.
- Table Headers:** Title, Assignees, Status
- Issue Details:**
 - 1. Include a default prefix in column names to indicate the data source following data exte... #5130
 - 2. Better display of which reconciliation service a specific column is reconciled against #4824
 - 3. Update the UI for adding media in the schema builder #5287
 - 4. Wikitext Builder UI #5267
 - 5. UI edits to schema management dialog in Wikibase extension #5347
 - 6. UI improvements for the "Add entity identifiers column" action #5998
 - 7. Improve look and feel of the issues tab of the Wikibase extension #5702
 - 8. Show endpoint URL in reconciliation service list #5878
 - 9. Possibility for a more robust undo/redo functionality to allow users to easily revert chan... #5745
 - 10. Padding of buttons, alignment of input field and increase of font-sizes for texts/urls to b... #5706
 - 11. Improve look and feel of project metadata dialog #5701
 - 12. Improve look and feel of database import panel #5700
 - 13. Improve UI and UX of the create project tab #5675
 - 14. 'Add column(s) from other OpenRefine project' as default feature #5393
 - 15. Highlight the first column when the records mode is on #5175
 - 16. Add new data quality or data profiling UI enhancements #5315
 - 17. Delete multiple projects at once #4965
- Assignees:** Yemihun..., elebitzero, prashasti..., Lydiaoffi..., wumion...
- Status:** In Progress, Triage, Todo
- Actions:** Filter by keyword or by field, New View, +

Project views:

- Once you open the UI / UX Project, you have the option to look at current issues in either Table or Board view. Both views allow you to get a quick overview of the status of various tasks and who is assigned to work on them. Typically, you want to find tasks that have not been assigned to anyone yet and have status “To Do”.

Getting assigned to an issue:

- When you'd like to contribute to a design task, you can request assignment or wait to be assigned by a project maintainer or someone with appropriate permissions. Click on the issue card to open it, and within the right-hand side "Assignees" section, you can request to be assigned to the issue or express your interest via a comment to the project maintainers. Getting assigned to an issue signals to the community that this issue will be actively worked on, and unless you explicitly request help from others, you will be responsible for carrying out the actions needed to close the issue.

Tracking the issue:

- Once assigned, you can move the issue card to the "In Progress" column, indicating that you're actively working on it, unless another project maintainer already does it for you. You can update the issue's description, comments, and share any relevant design assets. To demonstrate your progress, consider using checklists within the issue description to outline the specific design elements you're addressing. You can also add comments to the issue at relevant progress intervals to discuss particular pain points with other contributors or to mark any significant breakthroughs. Once an issue has been completed and/or moved to a pull request, you can also mark it as done.

Creating an issue

If you can't find relevant issues to work on in the OpenRefine Design Project, or you already have new ideas for OpenRefine that you'd like to contribute to, you are welcome to create new issues. To do that you can either click on "New issue" from the Issues page of the relevant repository, or you can click on the "Add new item" buttons from the bottom of the Table or Board views in the OpenRefine Design Project page. You may be prompted to select a template if applicable.

Specify the type of issue: Choose one of the following options that best describes your issue:

- **Bug report:** Create a report to help us improve OpenRefine.
- **Feature request:** Suggest an idea for OpenRefine.
- **Design discussion:** Start a discussion on issues related to UI/visual design, UX/usability or accessibility.
- **Documentation:** Report issues related to improving project documentation or tutorials.
- **Report a security vulnerability:** Privately report a security vulnerability.
- **Ask a question (forum):** Please ask and answer questions here.
- **Gitter chat:** Engage in live discussions on Gitter.

Selecting the appropriate issue type ensures that your contribution reaches the right audience and receives the attention it needs.

Title and description: Craft a clear and descriptive title (e.g., "Improve Consistency in Button Styling" rather than just "Button Design") for your issue. Detail the design problem or idea following the template provided in the description space. Screenshots or links to mockups can add clarity.

Assigning labels: Labels are used to categorize issues based on their content. It's

important to note that label assignment is generally restricted to more experienced contributors, and you need to wait to become a member of the OpenRefine GitHub organization before being able to perform this yourself. In any case, it's helpful to be aware of the current labeling conventions in order to be able to better identify issues suitable to contribute to, as well as being able to assign labels yourself in the future.

Label structure: Generally, issues can be tagged with labels that fall within one of the following categories: Type / Theme / Status / PR [pull request] / Priority / Difficulty / Platform / Module / Data format

You would have already seen the templates for the Type category labels, these were Bug, Feature Request, Design Discussion, and Documentation. The next set of labels relevant for design contributions are the Theme category labels, these can be one of the following: UI/Visual Design, UX/Usability and Accessibility. Status is assigned via the Project views. PR, Priority, and the rest of the categories are optional, but still helpful to assign whenever possible. Difficulty is a category that can be particularly useful when identifying issues to contribute to. The Good first issue label indicates an issue that is suitable to newcomers, whereas Difficulty: Intermediate or Advanced indicate issues better suited to more experienced contributors. Check out the Resources section for a complete guide to the GitHub label structure used by the OpenRefine community.

Submitting your issue: Click "Submit new issue" to publish your issue. It will now be visible to the OpenRefine community.

Brainstorming and ideation

- **Understanding the issue:** Read the issue description, any accompanying comments, and relevant documentation to gain a clear understanding of the problem at hand.
- **Brainstorm and research:** Engage in brainstorming sessions to generate creative ideas and potential solutions. Research industry best practices, user preferences, and existing design patterns to inform your approach.
- **Comment and discuss:** Within the GitHub issue, share your initial thoughts, concepts, and potential directions for the design. Engage in discussions with maintainers, other contributors, and designers to refine your ideas based on feedback.
- **Sketch concepts:** If applicable, sketch rough concepts on paper or using digital tools. These sketches can help visualize your ideas before delving into detailed design work.

Creating wireframes and concepts

- **Wireframe creation:** Based on the chosen concept, create wireframes – basic, low-fidelity representations of the user interface. Focus on layout, structure, and the flow of interactions.
- **Iterate and refine:** Share your wireframes with the community through the GitHub issue. Gather feedback and iterate on your designs, making necessary adjustments to address any concerns or suggestions.

Providing design assets and implementation

Providing design assets and design implementation

- **High-fidelity designs:** Once wireframes are approved by community consensus, proceed to create high-fidelity designs of the user interface using design tools like Figma or Sketch.
- **Visual consistency:** Ensure your designs align with the existing design system, incorporating colors, typography, and UI components as outlined in the OpenRefine Design System. Note: Using Figma in this regard will probably save you time.
- **Export assets:** Export design assets such as icons, buttons, and graphics in formats suitable for implementation and link them directly in the relevant GitHub issues.
- **Implementing your design ideas:** If you're fluent in frontend development, as well as design, you will ideally be able to also implement your own design proposals in html and css. You can follow the contributor guide for developers for detailed instructions on how the process for contributing code via GitHub is organized.

Feedback and prioritization

When engaging in design discussions and reviews, share your work within GitHub issues to gather feedback on usability, aesthetics, and project alignment. Incorporate received feedback by refining your designs, ensuring they adhere to the user goals identified in the original formulation of the issue.

Collaborate with maintainers and contributors to prioritize the order of addressing and implementing design tasks, considering the community's size and potential competing responsibilities. Be aware that feedback from project maintainers and experienced contributors might take time due to the community's small scale and other ongoing tasks.

For the same reason, implementation might also take time. The asynchronous feedback and implementation process does not remove any value from the design suggestion as it helps elicit future functionalities or improvements.

Community engagement

Active participation in forum

Besides specific issue discussions on GitHub, engage in discussions on the project's Forum to share more general insights, seek guidance, and collaborate with fellow contributors. Your active involvement helps foster a supportive community and enables valuable knowledge exchange, contributing to the collective growth of the OpenRefine project.

Monthly contributor meetup

Join the monthly contributor meetup to connect with others, discuss progress, and align

efforts. These gatherings provide an opportunity to share ideas, address challenges, and collectively steer the project forward, reinforcing a sense of camaraderie and shared purpose within the OpenRefine community.

Resources

[Past user research and testing](#)

[The OpenRefine Design System](#)

[The OpenRefine Forum](#)

[OpenRefine's GitHub Label Structure](#)

Documentation contributions

We welcome contributions to our documentation, which is maintained as a set of [Markdown](#) files in the [OpenRefine/openrefine.org](#) Git repository.

Single page edit via GitHub web interface

For changes made to a single page, you do not need to be familiar with Git. Go to the page you want to change and click "Edit this page" at the bottom of the page. This will lead you to GitHub, where you will need to have an account. You will be then guided through the process of proposing those changes (which will create a "Pull Request" in GitHub's terms).

Once your pull request is open, your changes will be checked automatically for common issues, and if all goes well a preview of the website with your changes will be generated. We will then review your changes and integrate them to the website.

Multi-page edit by editing the documentation locally

For more complex changes, it is helpful to check how the website looks after your changes, before you submit them. For this workflow you will need to install:

- a Git client, such as GitHub Desktop or the [git](#) command-line tool if you are comfortable working from the terminal;
- [Node.js](#)

- [Yarn](#), which can be installed with

```
npm install -g yarn
```

after having installed Node.js.

Once you have installed those tools, create a personal fork of the [OpenRefine/openrefine.org](#) repository and clone your fork, then change to the clone:

```
git clone git@github.com:<myaccount>/openrefine.org.git
cd openrefine.org
yarn
```

This will install the dependencies required to generate the site, which is mainly [Docusaurus](#).

Once this is done, generate the docs with:

```
yarn start
```

This will spin a local web server to serve the docs for you. A browser window will open at <http://localhost:3000> and will auto-refresh when you edit the source files.

You could also generate the website as a set of static files, although this should generally not be needed:

```
yarn build
```

Once you are happy with your changes, you can [submit them as a pull request](#).

Development roadmap

Please be aware that the OpenRefine roadmap is subject to change at any time, so please check back regularly, and monitor [milestones](#), [projects](#) and [issues](#) in Github to keep up to date with current plans.

If there are features you would like to see that are not currently listed here or in current [milestones](#), [projects](#) and [issues](#), please add them to the [issue tracker](#).

Planned releases

4.0

New backend storage option to allow using much bigger datasets at the expense of real-time feedback.

New UI (possibly Vue or React based)

Work in progress

Alongside the planned releases there are often smaller pieces of work in progress. Check for [recently updated issues](#) and [pull requests](#) to see what is currently in the works.

On the back burner

Some aspects of OpenRefine have previously been targeted for release, but have not made

it into a release and have not been worked on recently. If you would like to see features in these areas, please create an issue the describes what development you would like to see:

- Streamlining traditional features
- Views: map, timeline, protovis (D3.js) charts
- Better machinery to guess and re-encode cell values (useful for fixing encoding issues)
- Collaborative editing support (see documentation on the '[broker protocol](#)' to see where this work was going)
- Column groups