Image I/O-Ext – User Guide
V. 1.0

Eng. Daniele Romagnoli

Eng. Simone Giannecchini

1 – Introduction

This simple guide will provide you some instructions about how to use the Image I/O-Ext project capabilities and how to extend them by adding new plugins.

2 – Pre-Requirements

Before using the Image I/O-Ext, you should have already setup all the required elements. In case you need help on achieving this, you may look at the Image I/O-Ext Setup Guide, available here:

https://imageio-ext.dev.java.net/svn/imageio-ext/tags/1.0.1/documentation/ImageioExt-SetupGuide.odt[1]

After completing all the required steps suggested in that guide you should be ready to use the Image I/O-Ext Project. During the following instructions we assume you are using Eclipse as IDE.

First of all, you may download Eclipse from this site: http://www.eclipse.org/downloads/

From the “Eclipse IDE for Java Developers” section, click on the link located in the right side, referring the proper OS version you need (Windows/Linux).

3 – Usage

Image I/O-Ext extends the Java SUN's Image I/O which is a pluggable architecture for working with images stored in files and accessed across the network by means of a wide set of packages which allow to perform data access (read/write operations) and data manipulation, as well as a set of classes to define new image readers, image writers. Basically you would get access to a data source using a specific plugin which is able to manage that specific data format. Let us now introduce some tips on how to leverage on the Image I/O-Ext capabilities. As stated in the home page of the Image I/O-Ext project, it is composed of a main framework leveraging on GDAL which is a raster Geospatial Data Abstraction Library capable of managing a very large set of raster formats. The explanations available in the following sections are mainly focused on the main framework capabilities.

3.1 – Setup Customizations

When creating a new project which requires to use the Image I/O-Ext, you need to add several required libraries. Supposing you have already built and installed the Image I/O-Ext project as explained in the Setup Guide, you will find all what you need in your Maven2 Repository. Basically the Image I/O-Ext project core is built on top of 3 main libraries, available with the following JARs:

- **imageio-ext-gdal**
- **imageio-ext-gdalframework**
- **imageio-ext-customstreams**

Finally, depending on the specific format on which you need to get access, you must also add the proper library which provides access to it. As an instance, if you need to work on MrSID files, you also need to add the **imageio-ext-gdalmrsid** jar.

Before starting with some examples on how to perform data access and manipulation, plese note how each module composing the Image I/O-Ext project contains a set of Junit test case classes

[1]When prompted for user identification, just specify “guest” (without quotes) as user, with no password

which are mainly used by maven to test the project functionalities. To acquire confidence with the basic Image I/O (and Image I/O-Ext) data access/data manipulation ways, you can take a look on them with Eclipse, as explained in the following subsections.

3.1.1 – Build and import Eclipse projects

Maven allows to build ready-to-use Eclipse projects by automatically setting the required dependencies of each project. In case you need to setup eclipse projects for the main framework go in your `imageio-ext\library` folder and run:

`mvn eclipse:eclipse` (this will build ready-to-use projects containing the main framework).

Finally, if you are interested in building eclipse projects for all the available Image I/O-Ext plugins, you should enter in your `imageio-ext\plugin` folder and run again:

`mvn eclipse:eclipse -PconfigProfile` (Where configProfile represents one of the 3 profiles you can specify: **base** or **full** as explained in the setup guide, chapter 3.9.3 (Windows) or 4.8.2 (Linux))

Alternatively, if you are interested in a single plugin, you may enter in the proper subfolder, as an instance, `imageio-ext\plugin\gdalmrsid` and run again `mvn eclipse:eclipse`

At this point, you should be ready to run Eclipse and import the just produced projects as follow. From the Eclipse *File* menu: *File->"Import"->"General"->"Existings Projects into Workspace"->* and select the root directory where you previously downloaded the whole Image I/O-Ext project. When ready, the "*Projects:"* window should contain all the projects previously built with "`mvn eclipse:eclipse`". Select the ones you are interested in and go on.

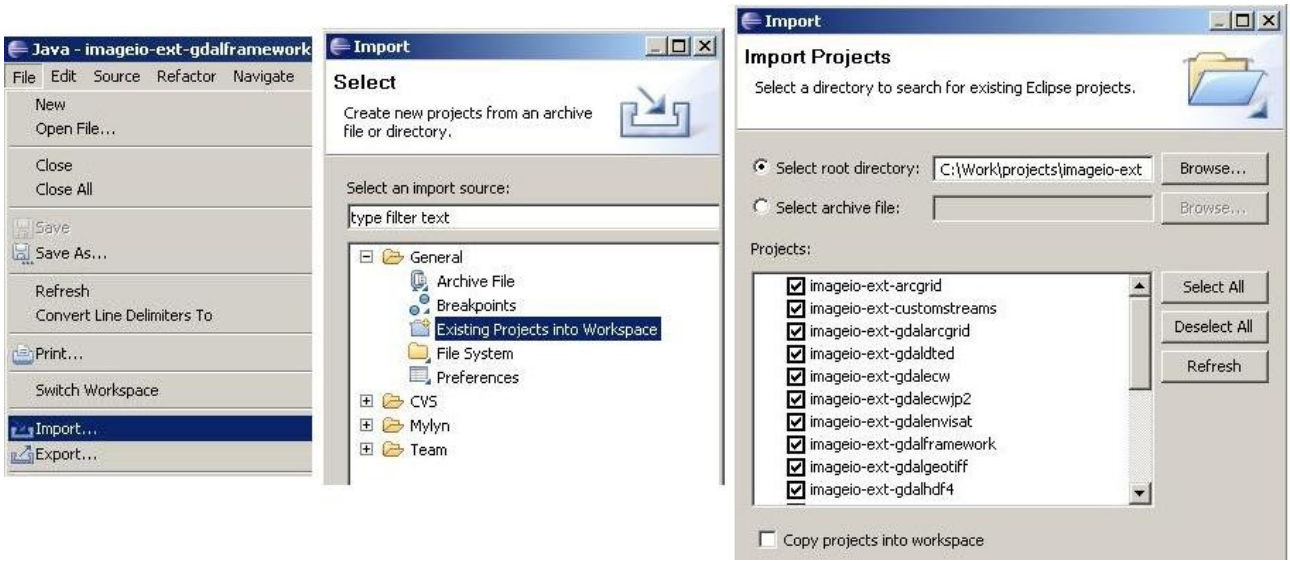

Figure 1: Importing Projects from Eclipse

3.1.2 – Setup dependencies on Eclipse

As stated in 3.1, some of these projects need several dependencies (as an instance, the GDAL-based plugins require `imageio-ext-gdalframework.jar`, `imageio-ext-gdal.jar`, `imageio-ext-customstream.jar` and some others) which are contained in the Maven2 repository.

Be sure you have properly set the *M2_REPO* classpath variable as explained here below. Open the properties of one of your just imported projects and select "*Java Build Path*" entry in the left column. Then go to the "*Libraries*" Tab and check if your M2_REPO variable has been defined. To define it, click on "*Add Variable"->"Configure Variables..."->"New..."* Set "M2_REPO" as Name and a proper location as Path. Usually, the local maven2 repository is located on the user folder of your OS installation, as an instance:

- "`C:\Documents and settings\YourUser\.m2\repository`" on Windows XP
- "`C:\Users\YourUser\.m2\repository`" on Windows Vista
- "`/home/youruser/.m2/repository`" on Linux

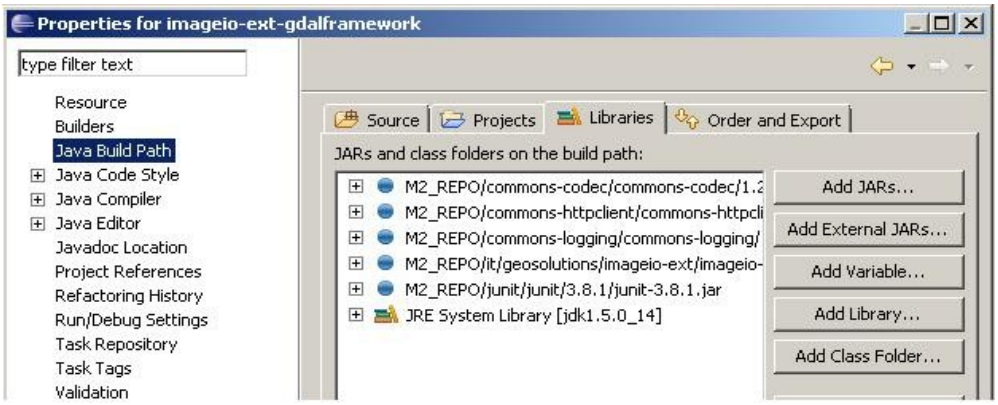

Figure 2: Configuring M2_REPO dependencies

At this point, everything should be ready to run your first test.

3.1.3 – Test run customizations on Eclipse

Select a test class from src/test/java folder of an imported plugin and select *Run As->Java Application*. Note that all tests do not display image loaded. If you want to view the images in the available tests, just specify a proper JVM argument for the test run as follow. Select a test class and select "*Run As" -> "Run..."*[2] . Then go in the *Arguments* TAB and add the following line in the "*VM arguments:*" box: `-Dorg.geotools.test.interactive=true` as shown in Figure 3.

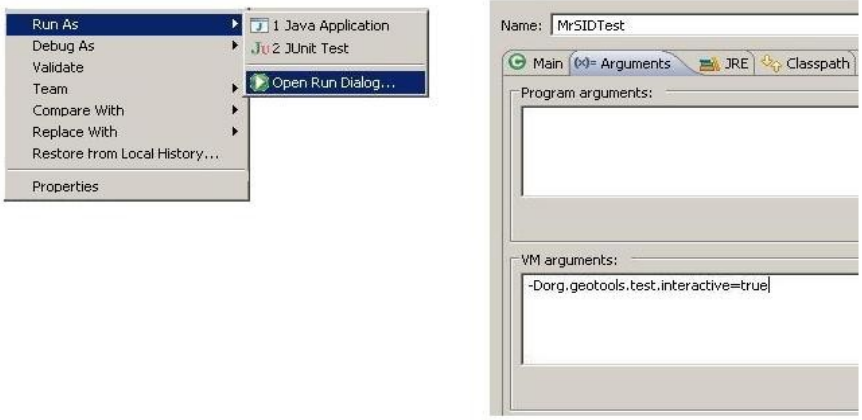

Figure 3: Customizing Run

[2]Depending on your Eclipse IDE version, the available menu command may be "*Open Run Dialog...*" instead of "*Run...*"

On Linux, in case the required libraries (.SO) are not found during tests, you should check the LD_LIBRARY_PATH environment variable[3]. Select the test class from src/test/java of a plugin project and select *Run As->Open Run Dialog*. Then select the "*Environment*" tab and check whether the LD_LIBRARY_PATH has been properly specified. If such a variable is not yet defined, just add it using the "Select..." button. Be sure it also contains a path where the required libraries have be placed, as an instance, the "`/usr/local/lib/`" path. If missing, simply add it using the "Edit..." button, and append the "`:/usr/local/lib/`" string to the value of such a variable.

3.2 – Read Access

Let suppose you need to obtain an image from a file stored on your disk (as an instance on `C:\data\sample.sid`). Depending on your needs, the read access could be performed in several different ways.

3.2.1 – Simplest Read Access

Use the following code in a method of your class to get an image by reading all the data available from the underlying file:

```
final File file = new File("C:/data/sample.sid");
final ImageReader reader = new MrSIDImageReaderSpi().createReaderInstance();
reader.setInput(file);
final RenderedImage image = reader.read(0);
```

This example may be useful to read a whole dataset. However, customizing a read operation is a more frequent task since, for example, you could need a subset of the dataset, or you could need to reduce the memory requirements. To achieve this objective, you may provide an image read parameter as argument of the read operation. The following example illustrates how to extract a portion of a bigger image and scale its resolution down to a lower level.

3.2.2 – Source settings parametrization read

Let's suppose your MrSID file is big (a 10000x10000 pixels dataset representing sea and shores) and you need to load a rescaled view (4 times smaller) of a portion of the original image: a region composed of 5000x3600 pixels starting at the x,y pixel coordinates (5000, 6400), as illustrated in Figure 4.

With the following settings the read operation will get an image of 1250x900 pixels.

The following lines of code allow to obtain the requested parametrized read operation:

Figure 4: source settings

```
final File file = new File("C:/data/bigsample.sid");
final ImageReader reader = new MrSIDImageReaderSpi().createReaderInstance();
reader.setInput(file);
final ImageReadParam param = reader.getDefaultReadParam();
```

[3]In case you have used the deploy module, all the required libraries should have be placed by Maven in your JRE and everything should already work.

```
param.setSourceSubsampling(4, 4, 0, 0);
param.setSourceRegion(new Rectangle(5000, 6400, 5000, 3600));
final RenderedImage image = reader.read(0, param);
```

It is also possible to specify a subset of the source bands to be read. Let's suppose you want to generate a Gray image from the first component of the RGB image of the previous MrSID example. Using the image read parameters, you need to specify the desired source band as well as the destination type of the image to be generated. When specifying a destination image or a destination type you need to take on account the additional read parameters such as subsampling and source region which can be involved in the destination sizes computation. The following lines of code contains[4] a possible example of the instructions needed to obtain this:

```
...
SampleModel sm = spec.getSampleModel();
final int width = reader.getWidth(0);
final int height = reader.getHeight(0);
final ImageReadParam param = new ImageReadParam();
final int ssx = 2;
final int ssy = 2;
param.setSourceSubsampling(ssx, ssy, 0, 0);

final Rectangle sourceRegion = new Rectangle(50, 50, 300, 300);
param.setSourceRegion(sourceRegion);

param.setSourceBands(new int[]{0});

Rectangle intersectedRegion = new Rectangle(0, 0, width, height);
intersectedRegion = intersectedRegion.intersection(sourceRegion);

final int subsampledWidth = (intersectedRegion.width + ssx - 1) / ssx;
final int subsampledHeight = (intersectedRegion.height + ssy - 1) / ssy;

ColorSpace cs = ColorSpace.getInstance(ColorSpace.CS_GRAY);
ColorModel cm = RasterFactory.createComponentColorModel(sm.getDataType(), cs, false,
       false, Transparency.OPAQUE);

param.setDestinationType(new ImageTypeSpecifier(cm, sm.createCompatibleSampleModel(
       subsampledWidth, subsampledHeight).createSubsetSampleModel(new int[]{0})));

final RenderedImage image = reader.read(0, param);
```

3.2.2.1 – Note on destination settings

The Image I/O specification states that when defining a destination image, its color model and sample model needs to correspond to one of the image type specifiers returned by the image

[4]Code for creating reader instance, file and setting input are omitted

reader, otherwise an exceptions will be thrown during the read. Due to the general purpose nature of the Image I/O-Ext framework, the available image readers will always return an `ImageTypeSpecifier` built in compliance with the main properties of the underlying dataset[5] such as width, height, number of bands and color interpretation. For this reason, if you need to build, as an instance, a destination image with a reduced set of bands, setting the destination, using the `setDestinationType` method should be preferred since this approach avoids the `ImageTypeSpecifier` compliance checks which will fail using the `setDestination` method.

3.2.3 – JAI ImageRead

In the previous examples we have performed data access directly using the `read` methods of an `ImageReader` instance. However, there is another way to perform data access and data manipulation with better performances: using the JAI-Image I/O Toolkit.

3.2.3.1 – Little Introduction on JAI and JAI-Image I/O Toolkit

JAI-Image I/O Toolkit provides Image I/O-based read and write operations for Java Advanced Imaging (JAI ImageRead and ImageWrite operations). The JAI is a set of APIs which allows sophisticated, high performance image processing functionalities, such as rescales, rotations, crops, convolutions, bands compositions, shears, sub-samplings and much more. Moreover, JAI provides built-in support for a wide set of mechanisms such as tiling, tile-caching, deferred execution and operations chaining. Let us provide a minimal introduction on these topics in order to know how data may be accessed/manipulated. Basically:

- Tiling refers to the technique of building a tessellation of a big image in smaller squares, allowing to load and process only a reduced subset of this, with the advantage of a reduced memory consumption and a minor loading time.
- Tile-Caching refers to the capability of caching tiles which need to be frequently used or involved in some type of processing.
- Deferred execution refers to a mechanism which allows loading data only when they are really need.
- Operations chaining refers to a technique which allows the user to sets a chain of operations by concatenating them one after the other as needed, building directed acyclic graph. The graph starts from a source (as an instance an originating image) and ends with a sink (as an instance, the rendering on the monitor). In such a context, the meaning of the term deferred execution is that no data pixels are loaded in memory until a sink needs to actually use data from the preceding operations (like in pixel evaluation or image rendering).

The JAI ImageRead operation is the bound between the JAI and the Image I/O for reading images using the Deferred Execution Model. Internally, it basically computes and get tiles when needed, involving the tile-caching mechanism.

Being the Image I/O-Ext, as its name suggests, an extension of the standard Image I/O architecture, it may be easily involved in JAI ImageRead and JAI ImageWrite operations.

Note that when using the JAI ImageRead, a call to the underlying `ImageReader`'s `read` method will be performed for any tile which needs to be accessed for the data loading. This approach is different with respect to the type of data loading performed by a manual call to the `read` method which simply loads all you need at one time. For this reason, in several circumstances you may

[5]Moreover, due to an inner limitation of the SampleModel class, the SampleModel dimensions are equal to the Tile sizes in case the image width * image height product exceeds the 2^31 -1 value.

notice that rendering an image using a not properly configured JAI ImageRead operation may require a lot of time. This mainly happens when the underlying dataset is striped, having each tile composed of a data row, requiring a JNI access to the underlying GDAL DLLs for each row/tile to be managed. However, this issue may be easily solved by specifying an ImageLayout when creating the JAI operation.

3.2.3.2 – JAI ImageRead Example code

The following lines of code allow to perform a simple ImageRead operation using the JAI-Image I/O toolkit with Image I/O-Ext:

```
final File file = new File("C:/data/bigsample.sid");
final ImageReader reader = new MrSIDImageReaderSpi().createReaderInstance();
final ParameterBlockJAI pbjImageRead;
final ImageReadParam param = new ImageReadParam();

param.setSourceSubsampling(4, 4, 0, 0);
param.setSourceRegion(new Rectangle(5000, 6400, 5000, 3600));

final ImageLayout l = new ImageLayout();
l.setTileGridXOffset(0).setTileGridYOffset(0).setTileHeight(512)
                       .setTileWidth(512);
pbjImageRead = new ParameterBlockJAI("ImageRead");
pbjImageRead.setParameter("Input", file);
pbjImageRead.setParameter("readParam", param);
pbjImageRead.setParameter("reader", reader);

RenderedOp image = JAI.create("ImageRead", pbjImageRead,
            new RenderingHints(JAI.KEY_IMAGE_LAYOUT, l));
```

After the last instruction it is possible to concatenate further JAI operations such as, as an instance, rotate, rescale, crops and much more. It is also worth to point out that no data has yet been loaded, due to the deferred execution.

3.3 – Metadata

The Image I/O architecture is capable to expose additional information (metadata) related to the supplied source. Since some data formats allow to store different images within the same data source, Image I/O distinguishes between the concept of stream metadata, which is used to report information about the whole data set we are referring to, and image metadata which is used to report information about a single image belonging a wider set.

As an extension of the Image I/O architecture, the main framework of Image I/O-Ext is able to return image and stream metadata (the last one simply representing names and descriptions of all the datasets available in the same datasource). In such a context, the main framework defines a `GDALCommonIIOImageMetadata` used to represent image metadata with a set of properties common to any dataset independently by the specific format implementation. The typology of information common to any dataset are:

- descriptive info (dataset name and description, used driver, CRS)
- raster properties (width, height, tile sizes, number of bands)

- bands properties
- geoTransformation

See the source code for further information about the `GDALCommonIIOImageMetadata`'s format structure. In case a format may contain specific metadata, a suitable `GDALCommonIIOImageMetadata`'s subclass instance could be used[6].

3.3.1 – Writable Metadata

As you will see on chapter 3.4, for a write operation it is possible to specify image metadata to be written to the output. The writer attempts to get several properties from the provided metadata such as georeferencing, projection, as well as auxiliary information to be written to the destination file. If the image to be written comes from a previous read which has been performed by an Image I/O-Ext reader leveraging on GDAL, you can simply ask it to obtain the related image metadata and reuse this as argument of the write operation. In other cases, such a metadata could be unavailable, as an instance, if your image has been created from scratch or the originating image has been loaded by an external image reader. In such a case you may leverage on the writable metadata class (`GDALWritableCommonIIOImageMetadata`) to set all the required fields and then use this as argument of the write operation. Note that you can also obtain a writable instance from a `GDALCommonIIOImageMetadata` object using the `asWritable` method and then set the properties of this new writable copy.

3.4 – Write Access

After a brief introduction on read access topics, it's time to introduce some explanations about the write capabilities. First of all, in the main page of Image I/O-Ext project you will find a brief list containing the type of data access supported (R/W) for each format in order to know which plugin is actually able to perform write operations.

Let us now start with a very simple write operation defining a JAI ImageWrite operation using the GeoTIFF plugin, where we suppose the originating RenderedImage has already been created using a previous read operation as explained in chapter 3.2.

```
ImageWriter writer = new GeoTiffImageWriterSpi().createWriterInstance();
final File file = new File("C:/data/output.tif");
final ParameterBlockJAI pbjImageWrite = new ParameterBlockJAI("ImageWrite");
pbjImageWrite.setParameter("Output", outputFile);
pbjImageWrite.setParameter("writer", writer);
pbjImageWrite.addSource(image);
final RenderedOp op = JAI.create("ImageWrite", pbjImageWrite);
```

In addition to the classic image write parameters customization, such as source region settings and subsampling factors, some format specific image writers provides support for special customizations of the write process. In this guide, no specific examples will be offered in such a context since this type of customization is strictly related to the format specifications. Anyway, the JPEG2000 (Kakadu based) as well as the GeoTiff plugins contains useful code for learning on such a topic. As an instance, a JPEG2000 data source may contain several views of the same image representing different resolution levels. When writing a new JPEG2000 sample, it is possible to define the number of desired resolution levels using a proper image write parameter.

[6]Take a look on the MrSID plugin to see an example.

3.4.1 – Notes on write operations

3.4.1.1 – Operations modifying the bounding box

Although this project may be exploited within a GIS context to obtain data access and raster manipulations capabilities, some concepts GIS-related are not handled since very powerful GIS applications already exist to achieve this task. For this reason, in case you need to write an image coming from a customized read operation, or in case your image has been manipulated with the result of the invalidation of the original geoTransformation, you should specify a proper image metadata instance, containing updated geoTransformation information.

3.4.1.2 – Memory allocation setting

The write operations are built on top of the underlying GDAL drivers capabilities. Some GDAL drivers only allows to create a copy of an already existent GDAL dataset. For this reason, the Image I/O-Ext framework attempts to build an in memory dataset (MEM dataset) from an input Image to be used as a source of the copy. However, since data for the source image could be very big in size (gigabytes), creating a MEM dataset could be memory wasteful. For this reason it is possible to define a maximum amount of memory to be used for this approach to prevent drastic memory uses. A runtime, in case the memory requested to setup a MEM dataset is greater than the specified threshold, a temporary GeoTIFF dataset is created on top of the available data as a source of the copy. To define the threshold value you need to define the `it.geosolutions.gdalmemoryrastermaxsize` system property with a value specifying the maximum amount of memory to be used to create a MEM dataset. As an instance, to set a maximum amount of 32 Megabytes you can specify as argument of the JVM the line:

```
-Dit.geosolutions.gdalmemoryrastermaxsize=32M
```

If specifying:

- a simple integer value: it will be considered as size in bytes
- a value ending with the “K” char: it will be considered as size in kilobytes
- a value ending with the “M” char: it will be considered as size in megabytes

3.4.1.3 – Persistable Auxiliary Metadata

A format, depending on its specifications, may support a reduced set of metadata. To overcome this limitation it is possible to enable the Persistable Auxiliary Metadata which allows to add extra metadata to a dataset. The additional metadata are stored within a file having the same name of the data file, with extension “aux.xml”. To enable the PAM mechanism you can externally setup an environment variable “GDAL_PAM_ENABLED” with the value “YES” or use the `setGdalPAM` method of the `GDALUtilities` class with a boolean specifying if you need to enable or disable PAM.

4 – Capabilities Extensions

As stated in chapter 3, although Image I/O-Ext provides several unrelated capabilities, its main feature is allowing to access and manipulate a set[7] of raster data formats via GDAL. For this reason, exposing a plugin for almost any raster format supported by GDAL, could be a good

[7]The home page of the project (https://imageio-ext.dev.java.net/) reports a list of the actually supported formats as well as the type of supported access (read/write)

objective of this project. You may find the list of all GDAL's supported formats at: http://www.gdal.org/formats_list.html

In case a format is not yet supported by Image I/O-Ext, it is possible to define a new plugin for it.

Basically all you need to do is writing a specific `ImageReader`/`ImageReaderSpi` as well as a specific `ImageWriter`/`ImageWriterSpi` in case you need to support write operations too[8]. You can take a look on the already defined plugins to understand how they are formed.

4.1 – Basic Steps

Any `ImageReaderSpi` needs to specify a set of basic properties which describes the capabilities of the specific image reader provided, such as, as an instance:

- the suffixes associated with the supported formats
- a set of human-readable names for the supported formats
- a list of MIME types associated with the supported formats
- the name of the associated plugin

For this reason, when defining a new plugin, the developer needs to set all these properties in the `ImageReaderSpi`'s subclass. You can take a look on the source code of the already defined plugins to see how to set these properties.

Moreover, any plugin leveraging on GDAL needs to specify the formats it is supporting in order to let the main framework interact with the proper GDAL driver. For this reason, the `GDALImageReaderSpi`'s subclass need to specify in the constructor, the name of the GDAL driver which will be used by the plugin.

To better understand how to set these properties you can take a look on the source code of the already defined plugins.

4.2 – Writer related settings (if available)

The basic operations required when defining a new writer are similar to the ones involved when setting up the reader. You need to define all the SPI properties as well as the constructor body.

Defining a writer plugin may require an additional set of settings since the underlying GDAL format drivers allow to specify a set of options before the creation of a file (a write operation), by setting a list of Strings having the form ”OptionName=OptionValue”. For this reason, before introducing how to define format specific capabilities, let us briefly illustrate how the framework allows to customize write operations.

4.2.1 – Internal Architecture

The framework contains a `GDALCreateOptionsHandler` class which allows to properly handle a set of format specific create options, each one is represented by an instance of the `GDALCreateOption` class. This structure allows to set create option properties such as:

- name of the option
- range of accepted values
- data type of the value[9]

[8]Be sure the underlying GDAL Driver supports creation for that format.

[9]Although create operation are always set as strings, the related value may represent, as an instance, a numeric quantity. Specifying the data type allows proper checks be performed.

- optional default value

The main framework also defines a `GDALImageWriteParam` extending the Image I/O `ImageWriteParam`. Basically, it allows adapting image write parameters by means of an internal instance of a `GDALCreateOptionsHandler`.

4.2.2 – Required settings for new plugins

When defining a plugin for a format supporting create options mechanism, it is worth to define a proper `GDALCreateOptionsHandler`'s subclass as well as a proper `GDALImageWriteParam`'s. By this way, a write operation may be parametrized by setting the properties of an instance of this class which leverages on the underlying options handler. Basically, you need to define a specific `XXXCreateOptionsHandler` extending `GDALCreateOptionsHandler`. In the constructor, you need to setup the list of all the create options available for that format, by defining names, value type and validity values. To acquire confidence on these operations you can take a look on the source code of the Image I/O-EXT's Jpeg2000 (Kakadu based) or the GeoTIFF plugins. After this step, you need to define a specific `XXXImageWriteParam` extending `GDALImageWriteParam` where to define a proper set of setter methods.