Paxata Data Library Custom Connector Developer's Guide

Paxata Release 2.12

Document Revision 1.0


Paxata™

Paxata Data Library Custom Connector Developer's Guide

www.paxata.com

Table of Contents

Introduction

The Paxata application is made up of several server components configured across multiple nodes in order to provide a distributed, scalable platform for data preparation. On-premises installations are supported on both bare-metal and virtualized platforms. The purpose of this guide is to provide detailed instructions for on-premise customers, as well as third parties, who want to build a custom Connector for importing and exporting data to and from the Paxata Data Library.

The instructions and tutorial provided in this document are intended for the developer audience with experience creating extensible applications using the Java platform.

For feedback, please send an email to ServiceDesk@paxata.com. Also, remember to check the Service Desk User Community at http://servicedesk.paxata.com.

How this guide is organized

This document is organized as follows:

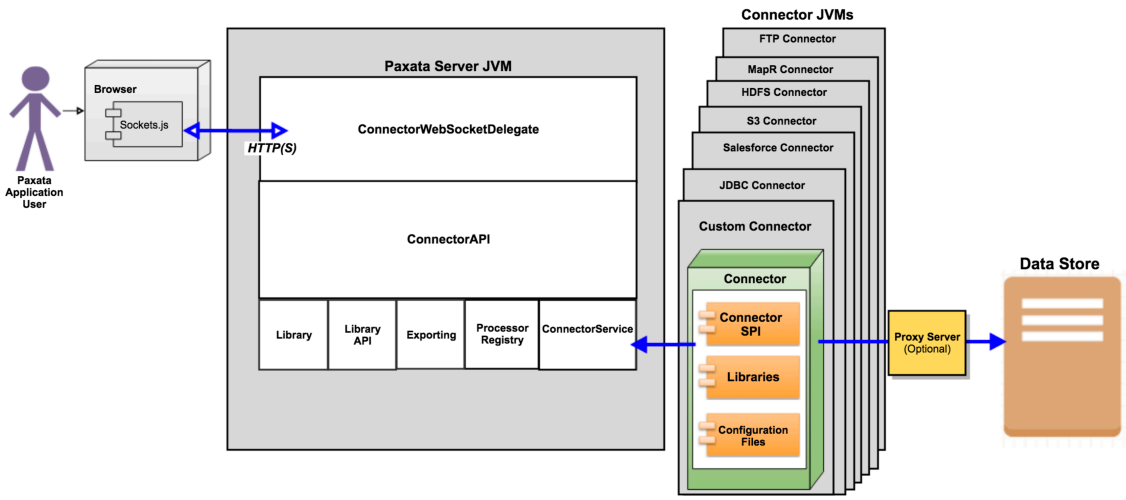
- An overview of Paxata's Connector Framework and the terminology you should familiarize yourself with prior to proceeding.
- An overview of all the steps you will take to create a custom Connector
- An explanation of concepts that you must understand before beginning your coding work.
- A complete tutorial that provides all of the steps required to write and implement a custom Connector plug-in from the ground up.

Appendices with complete examples of the Paxata Connector SPI and Configuration files.

Paxata Connector Framework

Overview

When Paxata is installed, out-of-the-box (OOTB) Connectors are provided for importing and exporting data to and from the Paxata Data Library: HDFS (CDH 5.5) for import and export, Hive (CDH 5.5) for import and export, JDBC for import, and Salesforce for import. Paxata's Connector Framework is also extensible; it supports the development of Connectors that can be "plugged into" the Paxata server to expand the Data Library's functionality to your specific Data Store.



The purpose of this document is to explain how you can create a custom Data Library Connector plug-in and deploy it for your Paxata users.

Before reviewing the concepts you will need to understand in order to develop your own Connector, please take a moment to review the *Connector Terminology* section below. The terms in the section are defined within the context of the Paxata Connector Framework and the process required to develop a custom Connector.

Connector Terminology

Connector: a Connector is something that can browse, read, and write external data (files and databases.)

ConnectorFactory: the top-level Java interface that you must implement in order to create a connection to your Data Store.

Data Source: an instance of a connector, fully configured, to access an external Data Store that has been configured through the Paxata UI and can be leveraged by the Paxata server to support the import and/or export of data.

Data Store: an external, internet-connected repository for persistently storing and managing collections of data. Examples of Data Stores include: relational databases, ftp servers, file systems (NFS mounted directories), distributed file systems (Hadoop), and Web APIs.

Export: the act of leveraging a Connector to write an AnswerSet from the Paxata Data Library into a remote Data Store.

Extensible Application Framework: an extensible application framework is a software design that can be extended and enhanced with new plug-ins or modules without modifying the original code base. Paxata's Connector Framework is extensible and supports the implementation of custom plug-in Connectors.

Import: the act of leveraging a Connector to copy a remote dataset into the Paxata Data Library.

Paxata Custom Connector: a ZIP file of: Java class files containing an implementation of the Paxata SPI (Service Provider Interface), the required Configuration files, and the 3rd party libraries required to enable connectivity from the Paxata server to your specific Data Store.

Service Provider Interface (SPI): an Application Protocol Interface (API) intended to be implemented or extended by a third party and can be used to enable framework extension to support plug-ins. The Paxata Connector SPI defines the contract that a third-party developer must support in order to integrate a custom Connector with the Paxata Server.

Creating Your Custom Connector

Overview of Steps

The following is a high-level outline of the steps you will take in order to successfully create a custom Connector plug-in for your Data Store.

1. Download a copy of the Paxata Connector SPI from the GitHub repository: https://github.com/Paxata/connector-spi
2. Code against the Paxata Connector SPI to implement the Java interfaces for your particular Data Store.
3. Create the JSON and Properties files, collectively referred to as the "Configuration Files", that define how your Connector will be presented and configured through the Paxata UI.
4. Test your Connector outside of Paxata.
5. After successful testing, create a ZIP archive that includes: your Java classes, your Configuration Files.
6. Install and deploy your ZIP archive at runtime using the appropriate REST API commands. (For installation instructions, see the separate document *REST API Guide*.
7. Login to the Paxata application and go to the "Connectors" page under the "Admin" menu. From the Connector page form, select and configure your Connector. After you save this form, your new Connector becomes available in Paxata for configuration.

Custom Connector Components

Overview

A custom Connector plug-in requires two key components that you must develop:

- the Java classes that satisfy the contract for the Paxata Connector Service Provider Interface (SPI)
- the Configuration Files—JSON and Properties files

Paxata Connector Service Provider Interface (SPI)

The Paxata Connector Service Provider Interface (SPI) defines the Java interfaces that a Connector must implement in order to integrate with Paxata's Data Library. The interfaces you implement form a contract between your Java classes and the Paxata server, and this contract is enforced at build time by your compiler. All methods defined by the Paxata SPI must appear in your source code before your Java classes will successfully compile. For more information on interfaces, refer to Oracle's Java documentation: https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

The Paxata SPI is a resource published by our product builds and made available here: https://github.com/Paxata/connector-spi

Configuration Files—JSON and Properties Files

The Configuration Files define a set of configuration fields that must be populated in the Paxata User Interface to enable connectivity with your Data Store. When the configuration field values are provided to your Java ConnectorFactory, the ConnectorFactory returns a configured Data Source that can be used for data import from/export to your external Data Store.

Recommendation

Develop a standalone Java test program to open a connection to your external Data Store and implement the methods required to get data from and put data to your Data Store. The resulting code can then be used to build your implementation of your SPI.

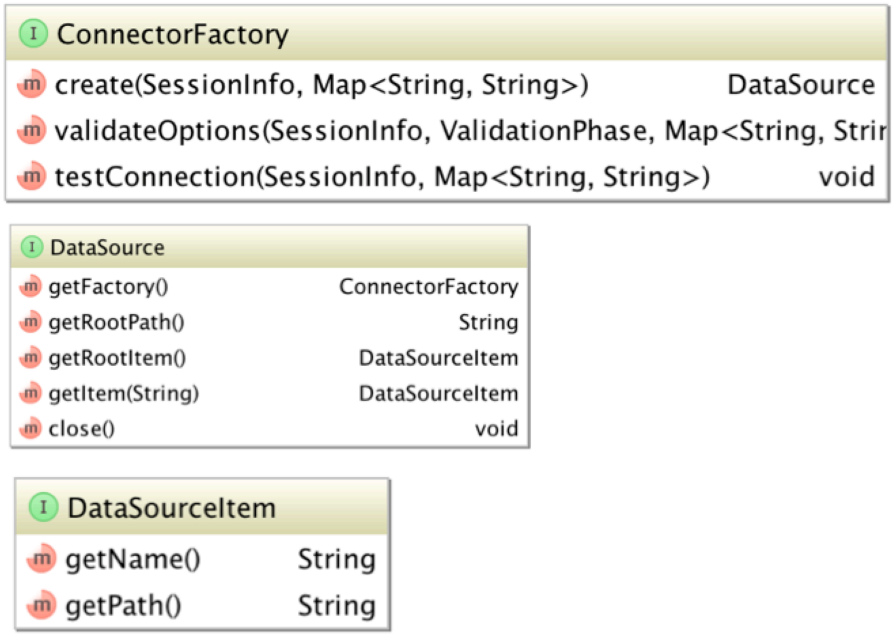
The next two chapters in this document review the SPI and Configuration Files in detail.

Paxata Connector SPI

Overview

The Paxata Connector SPI, or SPI, defines the Java interfaces that your Connector must implement in order to integrate with Paxata's Data Library. These interfaces form a wrapper around the appropriate Java library calls that interact with your external Data Store.

The following Java interface diagram displays the SPI's main interfaces and methods.

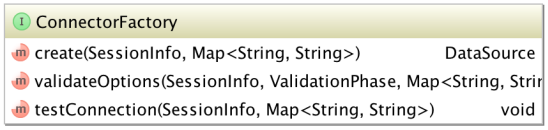


In addition to the main interfaces, there are also additional interfaces that you must implement based on the format of the data—stream or record—in your Data Store.

This chapter provides a high-level explanation of the SPI's interfaces and utility classes. For details on the class methods, refer to the Java docs referenced in Appendix A of this document.

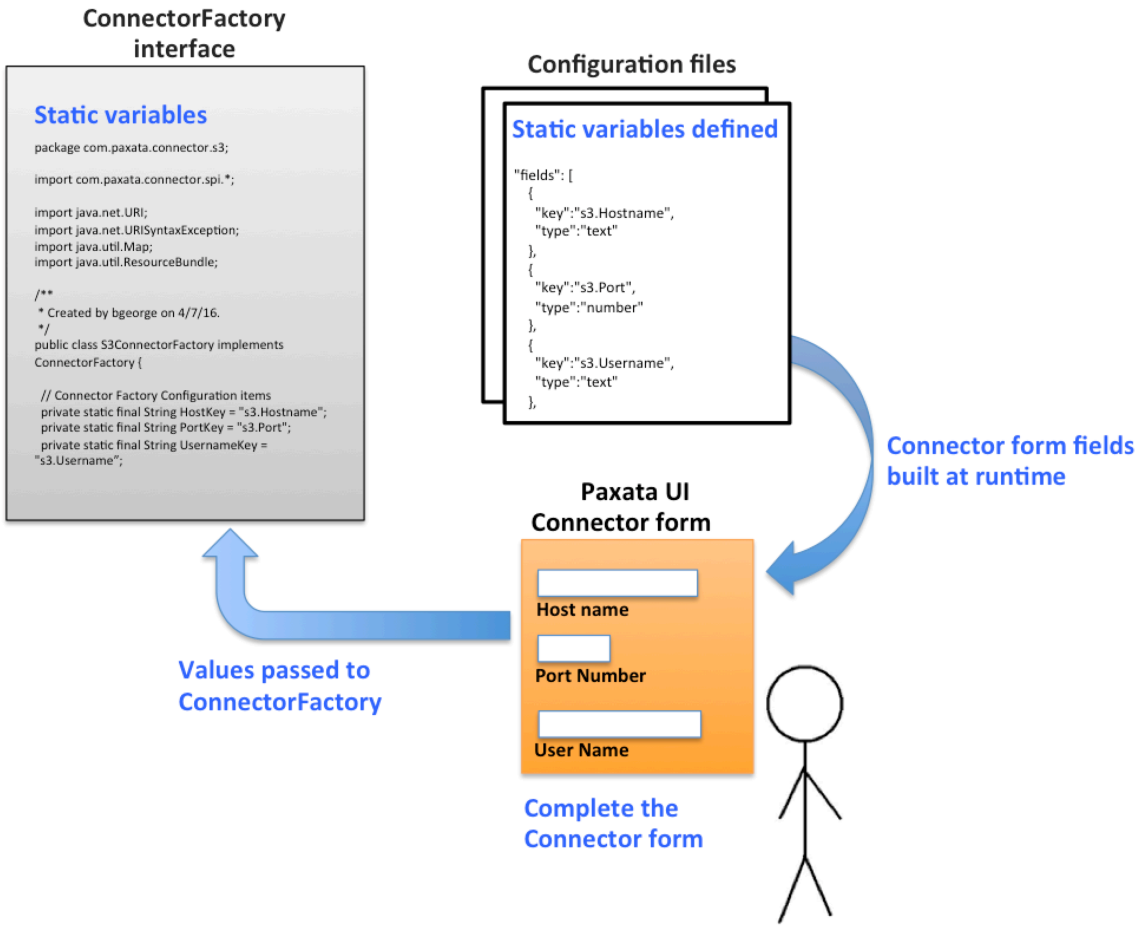
Main Interfaces

ConnectorFactory



The ConnectorFactory is the top-level interface you must implement in order to create a DataSource object that represents a connection to your Data Store.

The implementation of the ConnectorFactory class should contain static class variables that define the key names used to get the connection parameters for your Data Store. These key values are referenced in the **`fields`** array of your JSON Configuration File and they define the Connector form fields in the Paxata UI. Note: defining these fields as static variables referenced elsewhere simplifies the process of making changes later.



After you complete the Connector form in the Paxata UI, the ConnectorFactory class receives the form's input when the **`create`** method is called.

The **`create`** method takes:

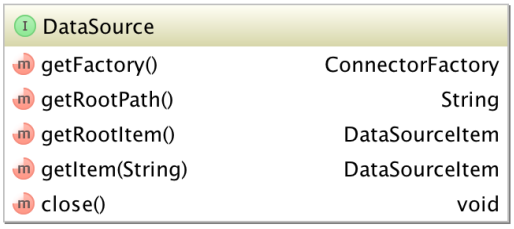
- the **`SessionInfo`** object—which has information about a Connector session passed from the Paxata UI Connector form;
- the **`options`** hashmap object—which has the configuration options passed from the Paxata UI Connector form

and returns an object of type **DataSource**:

```
DataSource create(SessionInfo sessionInfo, Map<String, String> options)
throws DataSourceConnectionException
```

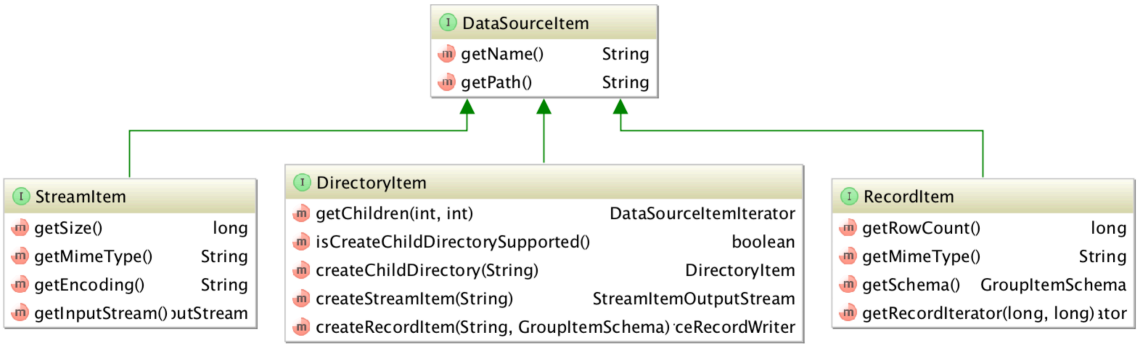
The **DataSource** object represents a connection to your Data Store.

DataSource



The **DataSource** interface is implemented to create a connection to your Data Store. When implemented, the **DataSource** class represents a connection to a remote Data Store and provides a mechanism to get items in the store.

DataSourceItem



The **DataSourceItem** interface is the base interface for all data source interfaces. There are three data source interfaces: **DirectoryItem**, **RecordItem** and **StreamItem**. All Connectors will use the **DirectoryItem** interface. However, the format of the data in your Data Store—stream or record—determines if you will implement the **RecordItem** or **StreamItem** interface. Each of these interfaces are explained in the next topic for *Data Interfaces*.

Data Interfaces

DirectoryItem

All Connectors must use the **DirectoryItem** interface. The root directory of your Data Store is an example of a DirectoryItem. The **DirectoryItem** interface allows you to:

- browse the children/container
- create child container objects
- create child data objects (that contain data)

StreamItem and RecordItem

The format of the data in your Data Store—stream or record—determines if you will implement the **StreamItem** or **RecordItem** interface.

	StreamItem	**RecordItem**
Unique Attributes	• schema less • accessed as a raw data stream	• has a schema • accessed as distinct rows and columns
Shared Attributes	• mime type that indicates type of object—for example: "text/csv" • a measure of size—bytes (stream) or row count (record)	
Required Classes	• **StreamItemInputStream** class that is used to read from a stream item. • **StreamItemOutputStream** class that is used to write to a stream item.	• **DataSourceRecordIterator** class that is used to read records in an object. • **DataSourceRecordWriter** class that is used to write a record item, one record at a time, with a **GroupValue**. When you create a new record item, a name and schema are required. To define the schema, use the following utility interfaces and classes: ○ **ItemValue** ○ **PrimitiveValue** ○ **GroupValue** ○ **ItemSchema** ○ **PrimitiveItemSchema** ○ **GroupItemSchema** See the next section for details on the utility interfaces and classes.

Utility interfaces and classes for implementing RecordItem

The following interface and classes are used to define your schema.

- **ItemValue**: data in a record item. There are two different types of **ItemValues**:
  - **PrimitiveValue**: a single value in an individual cell. The data value will be one of the following types: **BinaryValue**, **BooleanValue**, **DateTimeValue**, **NumberValue**, **TextValue**
  - **GroupValue**: represents an ordered list of other item values and has a schema because it includes multiple items of various types. In the example below, both the highlighted cell and the highlighted row can be represented as a **GroupValue** because both include a list of values.

Record #	Name (string)	Address [string]
1	name one	"12345", "Happy Way", "Redwood City, CA"
2	name two	"54321", "Grumpy Way", "Redwood City, CA"
3	name three	"13579", "Sleepy Way", "Redwood City, CA"

A list of strings in an array column

A list of values for three columns, and this column is a GroupValue

- **ItemSchema**—the base class for a schema. The **ItemValue** class you use determines which one of the following schema classes you will use.
  - **PrimitiveItemSchema**: schema class for primitive values.
  - **GroupItemSchema**: schema class for group values. This schema item can hold primitive schema items or another group item for nested structures.

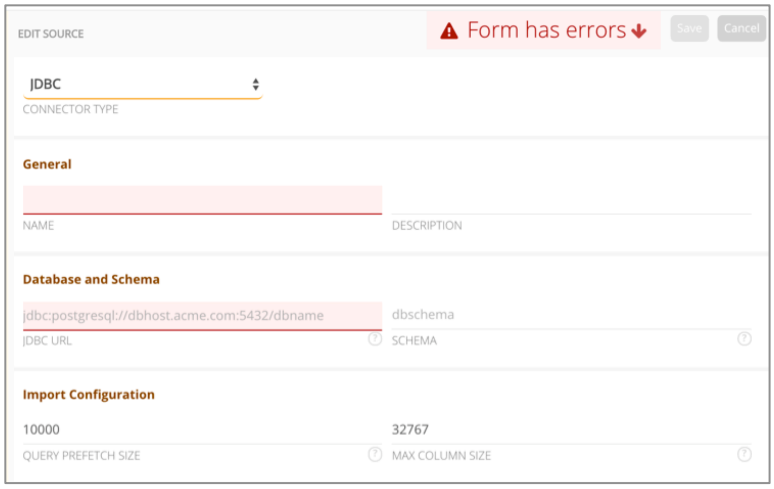
Configuration Files

Overview

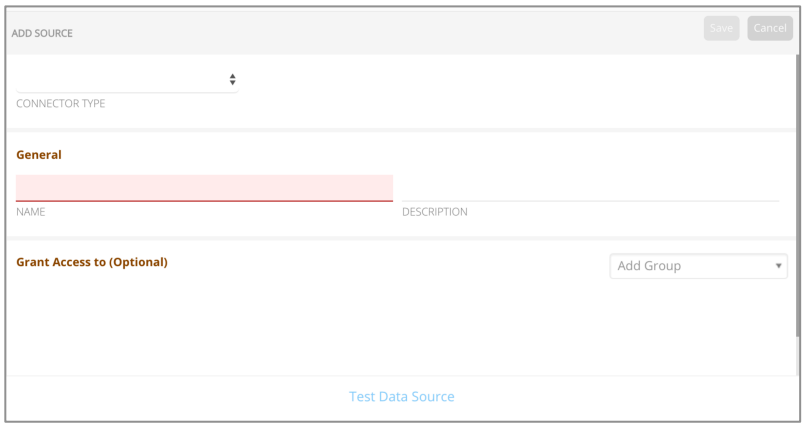
In addition to the Java interfaces that you must implement, there are two files you must write in order to configure your custom Connector through the Paxata UI: a JSON file and a Properties file. For the purpose of this developer's guide, these two files are collectively referred to as the "Configuration Files".

The Configuration Files are used to define the fields in the forms on the following UI pages:

Connector page in the Administration Menu includes the form used, typically by a Paxata administrator, to configure an instance of your Connector. The configuration fields assigned to this form should provide low-level configuration details about how a connection to your Data Store is made—for example, "hostname", "port", and "protocol" for an FTP connector.

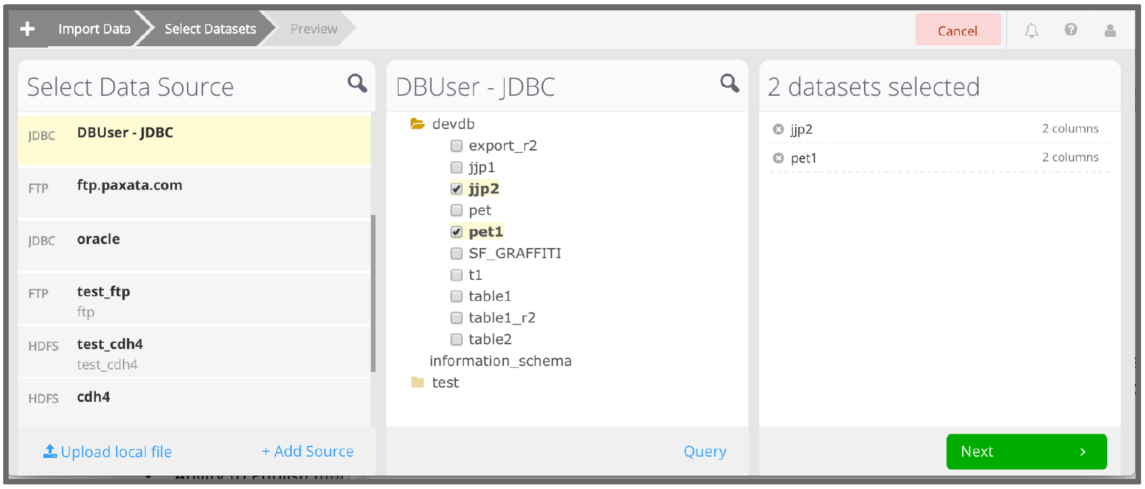


Data Source page in the Data Library includes the form used, typically by a Paxata administrator or data steward, to configure access to your Data Store based on the Configured Connector (that was created through the previous **Connector** page.) The configuration fields assigned to this form should provide the details that vary based on the user accessing the Data Store—for example, user credentials, the specific directories that a user can access, etc.

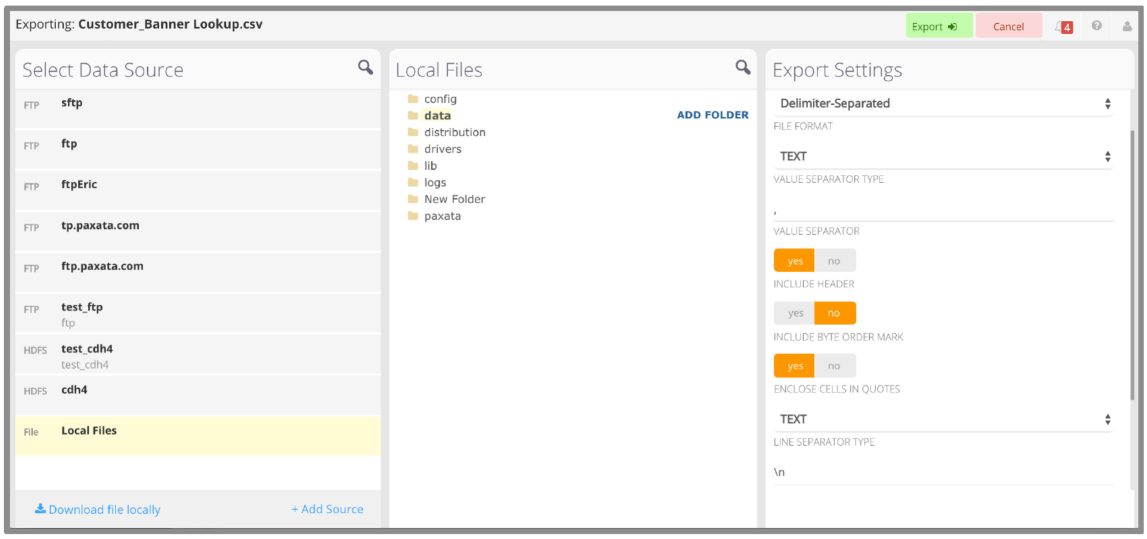


Import/Export session page in the Data Library is where import and export operations are performed by users. Configuration fields assigned to this page will be presented to users every time they import or export using the Data Source and should provide user access details —for example, user credentials, etc.

Import



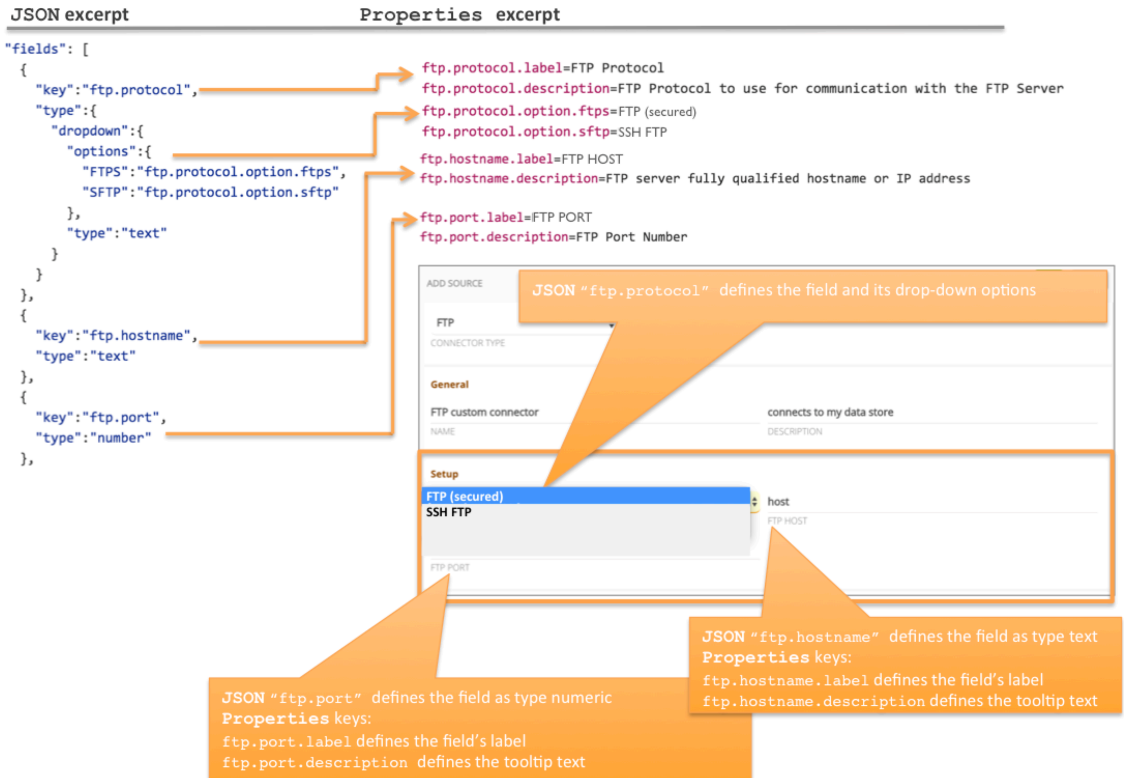
Export



How the Configuration Files Work Together

As part of the recommended practice for developing easily translatable applications, the Configuration Files were designed to support externalized text rather than hard coded strings in the HTML objects. The JSON file defines all of the configuration fields for the Connector, and the Properties file defines the text to display in the UI for the field labels and tooltips. The JSON file also defines the key values for your help text to display in the UI.

The following example displays excerpts from a JSON and Properties file. Notice how the excerpted "`fields`" array in the JSON file references the excerpted key/value pairs in Properties file—and then how both the JSON and Properties files configure your connector form in the Paxata UI:



The remainder of this chapter provides details for the content you must write and format you must follow when creating the Configuration Files.

JSON Configuration File

The JSON configuration file defines the field properties used to create the Connector, Data Source and import/export forms in Paxata UI. Each property is comprised of a name/value pair. Your JSON file can be authored in any standard text editor and must follow the format specification found at JSON.org (http://www.json.org/).

For the purpose of your custom connector, your JSON file must include the properties listed in the table below. See Appendix A for a complete JSON configuration file example.

> Important
>
> For Maven-specific properties in this file, you should adhere to the recommended Maven naming conventions. (https://maven.apache.org/guides/mini/guide-naming-conventions.html)

Property Name	Property Value description	Type	Required
`class`	Your Java class that implements **`ConnectorFactory.java`**	text	yes
`name`	Name of your connector as it should display in the UI on the Connector Configuration page for the "Connector Type" drop-down. Example: `ftp.connector.name` This "`key`" references an item (key and value pair) in your `Properties` file for the literal text to display, for example, "FTP"	text	yes
`description`	Description of what your connector does, for example, "Provides access to AWS S3 Data Store." This "`key`" references an item (key and value pair) in your `Properties` file for the literal text to display, for example, "FTP"	text	no
`bundle`	Resource bundle for locale-specific strings. Note: you must provide the name of your properties file without the .properties suffix. For example, "FTPConnector" for `FTPConnector.properties`	text	yes
`groupID`	Unique identifier for the organization that created the Connector, for example "Paxata". If you are using Maven to manage builds for your Connector, you can use your Maven `groupID` here.	text	yes
`artifactID`	Unique identifier for the Connector project. If you are using Maven to manage builds for your Connector, you can use your Maven `artifiactID` here.	text	yes
`version`	The version number for the Connector. Each new version of a connector deployed in the Paxata platform should have a unique version number. For example: 1.0, 1.1, 1.2, etc. Otherwise, previously installed versions will be overwritten.	number	yes
`external`	Boolean value (true or false) that determines if your custom connector should run in its own JVM outside of the Paxata JVM. The default is "false".	boolean	no

Property Name	Property Value description	Type	Required
`options`	Object with three items that define a connector's capabilities: • `importSupported`–defines support for importing from your Data Store to Paxata. • `exportSupported`–defines support for exporting from Paxata to your Data Store. • `querySupported`–defines support for the Paxata UI in receiving a query (like SQL) that can be passed to the Data Store. Note: if your connector does not support queries (the Queryable interface is not implemented) then you must specify "`false`" for "`querySupported`". • `wildcardSupported`–defines if the connector supports wildcard characters in path strings for import. Note: when a wildcard path is used to define an import, all of the files matching the path are concatenated. If undefined, the flag for these options defaults to `false`. Example: `"options": { "importSupported": true, "exportSupported": true, "querySupported": false, "wildcardSupported": true }`	object	yes
`fields`	Array of objects that define the configuration parameters used to generate the fields in your Paxata UI forms. The "`key`" references an item (key and value pair) in your `Properties` file; it is the key for the value passed in the HashMap to the ConnectorFactory. The "`type`" property defines the type of the field–text, number, boolean, etc. Each object "`key`" references an item (key and value pair) in your `Properties` file. Example: `[ { "key":"ftp.hostname", "type":"text" }, { "key":"ftp.port ", "type":"number" } ]` In the example above, two fields in the Paxata UI Connector page's form are defined. The first field is a text type field and references `ftp.hostname.label` and `ftp.hostname.description` in your Properties file for the field's UI label and description. The second field is a number type field and references `ftp.port.label` and `ftp.port.description` in your Properties file.	array of objects, one for each field	yes

Property Name	Property Value description	Type	Required
	Important The field types supported in in the JSON file are a subset of the HTML <input> Type standard. (http://www.w3schools.com/tags/att_input_type.asp) For more details on configuring objects in the `fields` array, refer to section: *Defining Connector Parameters through the "fields" Array*.		
`connector`	Object with two arrays that specify the required and optional "`fields`" to display on the UI Connector form. In addition, there is one item to specify the key values for the help text to display in the UI for your Paxata Connector page. Example: `"connector":{ "required":["ftp.protocol","ftp.hostname" ,"ftp.port"], "optional": ["ftp.rootDirectory"], "help": "ftp.connector.help" }` For more details on configuring this object, refer to section: *Configuring Parameters in Paxata UI forms through the JSON objects: "connector", "datasource" and "session"*	object	yes
`datasource`	Object with three arrays that specify the required and optional "`fields`" to display on the UI Data Source form. In addition, there is one item to specify the key values for the help text to display in the UI for your Paxata Data Source page. Example: `"datasource":{ "required": ["user.name","user.password"], "optional":["ftp.baseDirectory","username", "password"], "required_if_undefined": [], "help": "ftp.datasource.help" }` For more details on configuring this object, refer to section: *Configuring Parameters in Paxata UI forms through the JSON objects: "connector", "datasource" and "session"*	object	yes

<table>
<tr><th>Property Name</th><th>Property Value description</th><th>Type</th><th>Required</th></tr>
<tr><td>session</td><td>Object with three arrays that specify the required and optional "fields" to display for a Paxata user session that imports or exports data using the "datasource" object.

In addition, there is one item to specify the key values for the help text to display in the UI for Paxata users who connect to the Data Store for import/export operations.

Example:

<pre>"session":{
    "required": [user.pin],
    "optional": [],
    "required_if_undefined":
    ["user.name","user.password"],
    "help": "ftp.session.help"
  }</pre>
For more details on configuring this object, refer to section: Configuring Parameters in Paxata UI forms through the JSON objects: "connector", "datasource" and "session"</td><td>object</td><td>yes</td></tr>
<tr><td>groupings</td><td>Array of objects that allow you to conveniently group (in the UI) the fields you previously declared in the fields array.

The "name" defines a group label and references an item (key and value pair) in your Properties file for the group's name, and the fields array provides the key values for the fields to include in the group.

Example:

<pre>"groupings": [
 {
     name: "ftp.connector.legend"
     fields:
    ["ftp.protocol","ftp.hostname","ftp.port"
    ]
 },
 {
     name: "ftp.datasource.legend"
     fields: ["user.name","user.password"]
 },
 {
     name: "ftp.session.legend"
     fields: ["user.pin"]
 }
]</pre>
Important

• If you do not specify any groupings, then all fields in your forms will be grouped under a generic "Setup" label on the Connector, Data Source and session pages.

• If you choose to declare groupings, then you must provide field groupings for all fields as defined in the connector, datasource and session objects; you cannot specify a grouping for only one object's required fields.

For more details on configuring groupings, refer to section: Configuring the Grouping of Parameters in Paxata UI through the JSON "groupings" Array.</td><td>object</td><td>No</td></tr>
</table>

Property Name	Property Value description	Type	Required
supportedFormats	An object defining the data formats supported by this connector for import and export. This object contains two variables, import and export, that are themselves objects defining the relevant data formats. The import and export objects consist of two arrays, stream and record, that define the data formats supported by the connector. The stream array defines the stream-based file formats the connector will support; the record array defines the list of supported record-based data formats. Note: the list of supported stream-based formats is limited to those for which parsers are already defined in the Paxata platform. Refer to Appendix G: Stream Based File Types Supported by the Paxata Platform for more details. `"supportedFormats": { "import": { "stream": "Default", "record": [{ "key": "parquet", "displayName": "import.record.parser. parquet.name", "extensions": ["parquet"], "mimeTypes": ["application/x.parquet"] }] }, "export":{ "stream": "Default", "record": [{ "key": "parquet", "displayName":"export.record.parser. parquet.name", "extensions": ["parquet"], "mimeTypes": ["application/x.parquet"] }] } }` For more details on configuring groupings, refer to *Configuring the Supported File Formats with the "supportedFormats" Object*.	object	yes

Defining Connector Parameters through the "fields" Array

The "`fields`" array contains objects that define the configuration parameters used to generate the fields in your Paxata UI forms. There should be one object for every field that you need to generate, and the attributes you assign to an object determine how the associated field will display and function in the UI.

The following sections describe the required and optional attributes for objects in the "`fields`" array.

Required Attributes

key

The "`key`" attribute references a unique item (key and value pair) in your `Properties` file; it is the key for the value passed in the HashMap to the ConnectorFactory. The section *Properties Configuration File* provides details for the content you must write when authoring the `Properties` file.

type

The "`type`" attribute defines a field's type. The following types are supported.

- **text**
  The input for a "text" field is passed into the system exactly as it is entered in the form. A "text" input field is a single line. If you require more than a single line, use "textarea".

```
{
    "key": "ftp.hostname",
    "type": "text",
    "placeholder": "ftp.somedomain.com"
}
```

- **textarea**
  The input for a "textarea" field is passed into the system exactly as it is entered in the form. A "textarea" input field provides more than a single line for text entry.

```
{
    "key": "jdbc.beforeExport",
    "type": "textarea"
}
```

- **number**
  The input for this field type is passed to the server as a number with the JavaScript 'max number' enforced (http://www.w3schools.com/jsref/jsref_max_value.asp). If more than one dot or a longer number is required, use a "text" type field instead. Another important consideration when using a number type field: browser support. Because browsers support aspects of form field entry in different ways, you will want to ensure that your browser fully supports the number configuration that you expect to be entered into this field.

```
{
    "key": "ftp.port",
    "type": "number",
    "default": 21
}
```

- **boolean**
  A boolean field type that has one of two values: "true" or "false". (http://www.w3schools.com/jsref/jsref_obj_boolean.asp)

  Note: A boolean field is rendered in the Paxata UI as a checkbox.

  The following example defines a boolean field to enable auto-append of a timestamp value to data exported by the connector.

```
{
    "key": "append.timestamp",
    "type": "boolean",
    "default": true
}
```

- **drop-down**
  The drop-down is an HTML field type with nested "`options`" that allow you to define a list from which a user can select one value. Each property in the "`options`" object defines an option that will be available in the drop-down list. In the example below, the property names "FTP", "FTPS" , and "SFPT" are the values that will be passed to the ConnectorFactory, and the values for these properties—`ftp.display.name`, `ftps.display.name`, and `sfpt.display.name`—are keys that reference key/value pairs in the Properties file for the text to display in the Paxata UI.

  Use the following syntax to create a drop-down field:

```
 "fields": [
   {
     "key":"ftp.protocol",
     "type":{
       "dropdown":{
         "options":{
           "FTP":"ftp.display.name",
           "FTPS":"ftps.display.name",
           "SFTP":"sftp.display.name"
         },
         "type":"text"
       }
     }
   }
]
```

Optional Attributes

default

The `default` attribute defines a default value for an attribute that is pre-populated in the user interface.

placeholder

The `placeholder` attribute defines a short hint that describes the expected value for field input. This hint is displayed in the field before the user enters a value.

maxlength

The `maxlength` attribute specifies the maximum number of characters permitted in a "`text`" type field. Ensure you follow the syntax example below and capitalize the "L" for this attribute.
Note: this attribute is not supported for the other field types.

```
{
    "key":"ftp.hostname",
    "type":"text",
    "maxLength":"25"
}
```

hidden

The `hidden` attribute is used to hide a specific field in your form. When an expression that you define with the "hide" attribute evaluates to "true", then the field is hidden.

An example of when you may want to use this attribute: if your connector form presents different field options based on the selected authentication type. In this case, your JSON file defines a "`dropdown`" field type with the connection options. Then, a "`checkbox`" field with the hidden attribute is defined to hide a field that is not relevant for a particular authentication type.

The following example hides the field key "`fs.krb5.configure`" when the `fs.authentication` key is not equal to the value '`keytab`'.

```
  {
    "key":"fs.krb5.configure",
    "type":"text",
    "hidden": "fs.authentication != 'keytab'"
  }
```

Important

A strict syntax is enforced for the hidden attribute, which must follow this sequence:

hidden: {key} {operator} {literal value for key}

The key value must refer to a specific key as it is defined in your JSON file.

If the expression that you define here evaluates to "true", then the field will be hidden.
For example:

```
"hidden": "fs.authentication != 'keytab'"
```

or

```
"hidden": "fs.authentication == 'keytab'"
```

encrypt

The `encrypt` attribute specifies that input for the specified key value should be masked and encrypted when stored in the Paxata Mongo database. Input for this key value will then be masked as the user enters the string. Additionally, the input will be encrypted when stored in the Paxata Mongo database.

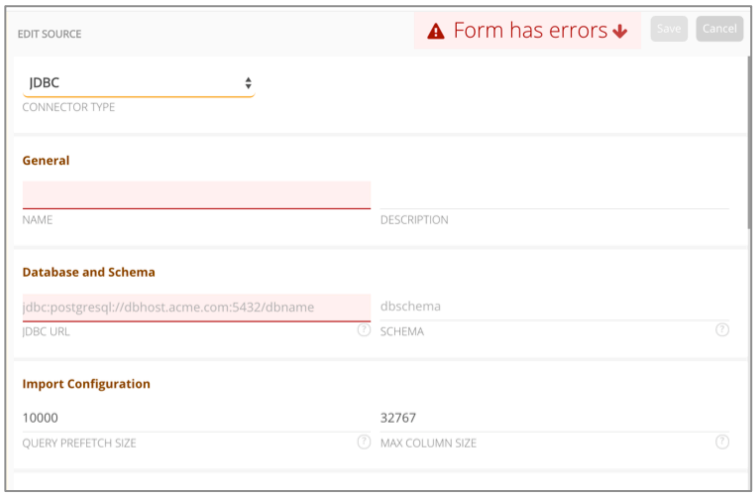
```
{
      "key":"connect.Password",
      "encrypt":true,
}
```

Configuring Parameters in Paxata UI forms through the JSON objects: "connector", "datasource" and "session"

The three JSON objects for "connector", "datasource" and "session" define the fields on the UI pages for Connector and Data Source, and during a user import/export session. The details for each object are explained below.

"connector" object

The form on the Connector page has the configuration fields that must be populated to define the external Data Store and how you will connect to it. Input to this form provides low-level configuration details about how a connection to your Data Store is made.



The "connector" object in your JSON file contains arrays that specify which field keys in the Connector page form are required or optional. In addition, if you plan to implement UI help for the Connector page, the help file key is also specified in this object:

```
"connector":{
  "required": ["ftp.protocol","ftp.hostname","ftp.port"],
  "optional": ["ftp.rootDirectory"],
  "help": "ftp.connector.help"
}
```

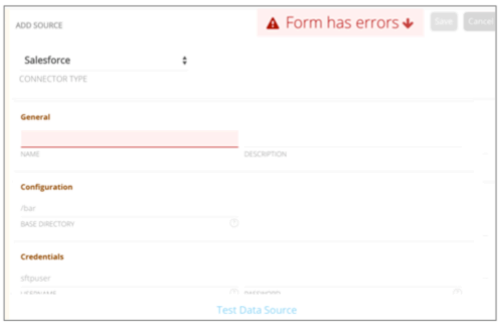
Important

Undefined parameters in this object default to an empty array, meaning that no fields will be displayed.

For more information on the help keys and how these are displayed in the UI, refer to section *Properties Configuration File*.

"datasource" and "session" objects

The Data Source page is where you configure user access to your Data Store and a session page is where a user performs an import or export operation.



The "datasource" and "session" objects in your JSON file contain arrays that specify which field keys are required or optional on the Data Source form and during a user import/export session. In addition, if you plan to implement UI help for the Data Source and user import/export pages, the help file keys are also specified in these objects:

```
"datasource":{
    "required": [],
    "optional": ["ftp.baseDirectory","username","password"],
    "required_if_undefined": [],
    "help": "ftp.datasource.help"
  },
  "session":{
    "required": [],
    "optional": [],
    "required_if_undefined": ["username","password"],
    "help": "ftp.session.help"
  }
```

The "required_if_undefined" option is an array of key fields that are required if the fields represented by those keys were not completed earlier on the Connector or Data Source pages. For example, this option allows you to prompt for user credentials at the Data Source level instead of requiring end user login for every import and export session. In short, the "required_if_undefined" array allows you to specify fields that must be completed if they were not completed earlier through either the Connector or Data Source page.

Important

Undefined parameters in this object default to an empty array, meaning that no fields will be displayed.

For more information on the help keys and how these are displayed in the UI, refer to section *Properties Configuration File*.

Configuring the Grouping of Parameters in Paxata UI through the JSON "groupings" Array

The "groupings" array allows you to conveniently group the fields in the "connector", "datasource" and "session" objects. This enables you to group the UI fields displayed on the Connector, Data Source and session pages.

The object "`name`" defines a group label and references an item (key and value pair) in your Properties file for the text to display for the group's label, and the "`fields`" array provides the key values for the fields to include in the group.

Note: the order in which you list the objects determines the order of the groupings in the form. Also, the order in which you list the field keys in an object's array determines the left-to-right order in which the associated fields appear in the UI.

Example JSON excerpt:

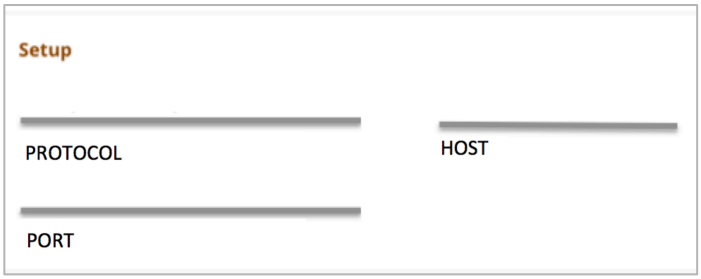
```
"groupings": [
 {
     "name": "grouping.host.legend"
     "fields": ["ftp.protocol","ftp.hostname","ftp.port"]
 },
 {
     "name": "grouping.credentials.legend"
     "fields": ["user.name","user.password"]
 },
 {
     "name": "grouping.pin.legend"
     "fields": ["user.pin"]
 }
]
```

Example Properties excerpt:

```
# Groupings
grouping.host.legend=Setup
grouping.credentials.legend=Authentication
grouping.pin.legend=PIN ID
```

Using the example on the previous page, the following labels and groupings are created in the UI.

Connector page:



Data Source page:

Session page:

Important

- When you declare groupings, you must provide field groupings for *all* of the fields defined in the "connector", "datasource" and "session" objects; you cannot specify a grouping for only one object's required fields.
- It is imperative that you account for all of your required fields in the groupings. Otherwise, they will not be displayed in UI.
- If you do not specify any groupings, then all required fields will be grouped under the generic label "Setup".

Configuring the Supported File Formats with the "supportedFormats" Object

The "supportedFormats" object defines the types of formats that your Connector supports. These formats are defined separately for import and export using the "import" and "export" objects. Each object contains two fields—"stream" and "record"—that define the supported formats. Note that if your Connector supports both import and export, then you must define at least one format for each.

For stream-based formats, the supported types are limited to those already supported by the platform. See *Appendix E: Stream-Based File Types Supported by the Platform* for details of the supported types.
Important:

- If the "stream" field is defined as "DEFAULT", then all platform formats will be supported for your Connector.
- If no stream-based formats are supported, then define this field as an empty array.
- If a limited set of platform-supported formats will be supported, refer to Appendix E for the correct entries and syntax.

For record-based, the Connector defines the supported format using the "record" field.
Important:

- If no record-based formats are supported, then define this field as an empty array.

The example below is for a Connector that supports import and export, where:

- Import
  - "stream" uses "DEFAULT" to include all platform-supported formats
  - "record" defines a single supported format for parquet files
- Export
  - "stream" limits to the "fixed-width" and "separator" formats defined in the platform
  - "record" is an empty array to indicate no record-based export is supported

```
"supportedFormats": {
  "import": {
    "stream": "DEFAULT",
    "record": [
      {
        "key": "parquet",
        "displayName": "import.record.parser.parquet.name",
        "extensions": ["parquet"],
        "mimeTypes": ["application/x.parquet"]
      }
    ]
  },
```

```
  "export":{
    "stream": [
      {
         "key": "separator",
         "displayName": "export.stream.parser.delimited.name",
         "extensions": ["csv"],
         "mimeTypes": ["text/csv"]
      },
      {
         "key": "fixed-width",
         "displayName": "export.stream.parser.fixed.name",
         "extensions": ["txt"],
         "mimeTypes": ["text/fixed-width"]
      }
    ],
    "record": []
  }
}
```

The objects in the arrays follow this structure:

```
{
  "key":"delimited",
  "displayName":"export.csv",
  "extensions":["csv", "tsv"],
  "mimeTypes":["text/csv", "text/x-csv"]
}
```

- `key`: this value defines a unique identifier for this file type.
- `displayName`: this value defines a key that references an item (key and value pair) in your Properties file for the text to display for the file type name in the UI.
- `extensions`: a JSON array of strings listing each file extension (without the period) that is supported for this format. Leave this array empty if there are no meaningful file extensions—for example JDBC and Hive.
- `mimeTypes`: a JSON array of strings listing each mime type supported by this format. At least one mimeType must be defined. For a format where a registered mime type does not exist, use the "x." syntax to define an unregistered mime type. For example: "application/x.dbtable" for importing from a database table.
  For details, refer to: https://en.wikipedia.org/wiki/Media_type

Properties Configuration File

Your Properties file provides the text to display for every key value referenced in your JSON file. The Properties file can be authored in any standard text editor but must define all of the field text to display in the Paxata UI. The following four sections define the categories of information that should be included in your file:

- **#Connector top-level strings**: the name of the connector as it should display in the UI.
- #**Options**: the UI labels and descriptions for every `field` array object referenced in the JSON file. Note: the `label` provides the field label and the `description` provides the tooltip text on hover. You may find it useful to organize the options by sections that reflect the pages in the UI where the fields are displayed—see the example below.
- **#Exceptions**: the error message text to display on import or export failure.
- **#Online Help**: the help text for the Connector, Data Source and session pages in the Paxata UI. Note that if you do not include any text for these keys, the help panel in the UI for these pages will be blank.

Example Properties file:

```
# Connector top-level strings
connector.name=Foo Data Store
connector.description=Foo Data Store Connector

# Connector Options
host.label=Host Name
host.description=The name or IP address of the host server

# Data Source Options
proxy.host.username.label=Proxy Username
proxy.host.username.description=Required to login to proxy server

proxy.host.password.label=Proxy Password
proxy.host.password.description=Required to login to proxy server

# Session Options
rootDirectory.label=Data Store Root Directory
rootDirectory.description=Top-level directory for Foo Data Store

baseDirectory.label=Base Directory
baseDirectory.description=sub-directory in Foo Data Store
```

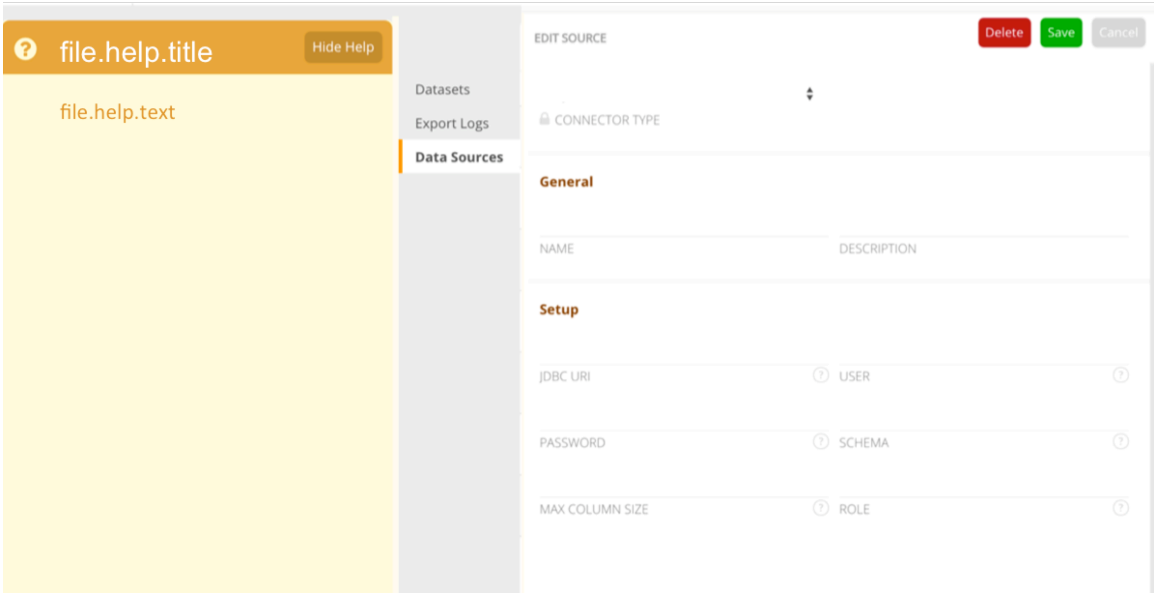
```
# exceptions
connector.file.output.exception=Exception getting output stream for {0}
connector.file.input.exception=Exception getting input stream for {0}

# Online Help
file.connector.help.title=Data Store Connection Configuration
file.connector.help.text=This connector…

file.datasource.help.title=Data Source Configuration
file.datasource.help.text=This connector…

file.session.help.title=Data Import and Export
file.session.help.text=To import and export data from…
```

For your online help, the `.title` and `.text` keys are used to display the help text in the Paxata UI as follows:



See Appendix B for another and longer example of a Properties file.

Custom Connector Tutorial

Terminology

Important Terminology to review before beginning this tutorial.

AWS: Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality

More info: https://aws.amazon.com/

AWS S3: Amazon Simple Storage Service (Amazon S3), provides developers and IT teams with secure, durable, highly-scalable cloud storage. Amazon S3 is easy to use object storage, with a simple web service interface to store and retrieve any amount of data from anywhere on the web.

More info: https://aws.amazon.com/s3/.

AWS S3 Region: AWS users can create data storage Buckets in many of the geographically distributed Amazon data centers. These data centers are identified as "regions".

More info: http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region

AWS AccessKey / AWS SecretKey:

An alphanumeric text string that uniquely identifies the AWS user

More info: http://docs.aws.amazon.com/general/latest/gr/getting-aws-sec-creds.html

S3 Bucket: A **bucket** is a logical unit of storage in Amazon Web Services (AWS) S3 service. **Buckets** are used to store objects, which consist of data and metadata that describes the data. Users can consider a bucket to be a directory. The AWS S3 service allows a maximum of 100 uniquely named buckets.

More Info: http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingBucket.html

S3 Object: An S3 Object is a unit of key - value data storage. The key, in this case, is the name of the data storage object (file) that is uploaded to the S3 Service and the value is the data within the object. S3 Objects also support metadata, versioning and permissions.

More info: http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingObjects.html

S3 Object Key: The name of an S3 Object. The name can represent a hierarchy of structure similar to that of a file path. These hierarchical names are known as **Prefixes.** Examples: 'my-data-file.csv', 'US/OH/Columbus/census.csv'

More info: Object Keyshttp://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html#object-keys

S3 Object Prefix: An S3 Object prefix is the initial delimited portion of the S3 Object Key that allows users to form a virtual hierarchy of their data without an actual directory structure. In the object key 'US/OH/Columbus/census.csv', we can use several possible prefixes to address subsets of data ("US/", "US/OH/", "US/OH/Columbus/").

Connector: A Paxata Connector is a package of Java software, configuration files and dependencies that are packaged as a zip file.

Primary contents of zip file:

- A Java Jar file containing an implementation of the Paxata Connector API
- 3rd party libraries leveraged in the connector, including those that enable connectivity to a specific Data Store.
- A JSON formatted configuration file that defines the set of configuration fields that are presented in the Paxata UI.
- A properties file that contains externalized strings and messages to support localization messages for Paxata end users.

Data Source: A Data Source is configured instance of a connection to an external Data Store that can be leveraged by the Paxata server to support the import and/or export of data.

Data Store: A data store is an external, internet-connected repository for persistently storing and managing collections of data**.** Examples of data stores include relational databases, ftp servers, file systems (NFA mounted directories), distributed file systems (Hadoop), Web APIs.

Service Provider Interface (SPI): The set of public interfaces and abstract classes that specify the features so be implemented by Paxata Connector Services. The SPI defines the contract between the Paxata server and the Connectors that provide extensibly connectivity to data stores.

DataSourceItem: This is Paxata terminology for an item stored with in the data store. These items can represent a directory, file, database schema, table, etc.

RootItem: Each **Data Source** created by the ConnectorFactory will establish connectivity with the specified connection options and establish a root item, typically a directory or schema, that serves as the top-level of a hierarchy of data that is available to the Paxata user.

Prerequisites

- An operational Paxata server that supports external Connectors.
    - It is imperative that you leverage the version of the Paxata Connector SPI that is supported by your Paxata installation.

> ⚠ **Find the SPI in your installation**
> The Connector SPI jar file is shipped with the Paxata server. Connector developers will need to be provided with this artifact in order to build Paxata Connectors.

- A Java software development environment to support implementation and testing.
    - Paxata allows for Java projects written and tested using Java 7 or Java 8.
    - Paxata Runtime environment of leverages Java 8.

- AWS (https://aws.amazon.com/sdk-for-java/)
    - A suitable *http://docs.aws.amazon.com/AWSSdkDocsJava/latest/DeveloperGuide/java-dg-java-env.html*.

- The necessary SSL certificates installed. If you are using Java 1.6 or newer, you should already have the correct certificates installed. For more information, see *http://docs.aws.amazon.com/AWSSdkDocsJava/latest/DeveloperGuide/use-sha256.html*.

- Existing AWS S3 service setup with at least 1 Bucket created.
  - Please note the AWS S3 Region used when setting up S3.

- AWS account and access keys. For instructions, see *http://docs.aws.amazon.com/AWSSdkDocsJava/latest/DeveloperGuide/getting-started-signup.html*
  .

Notes

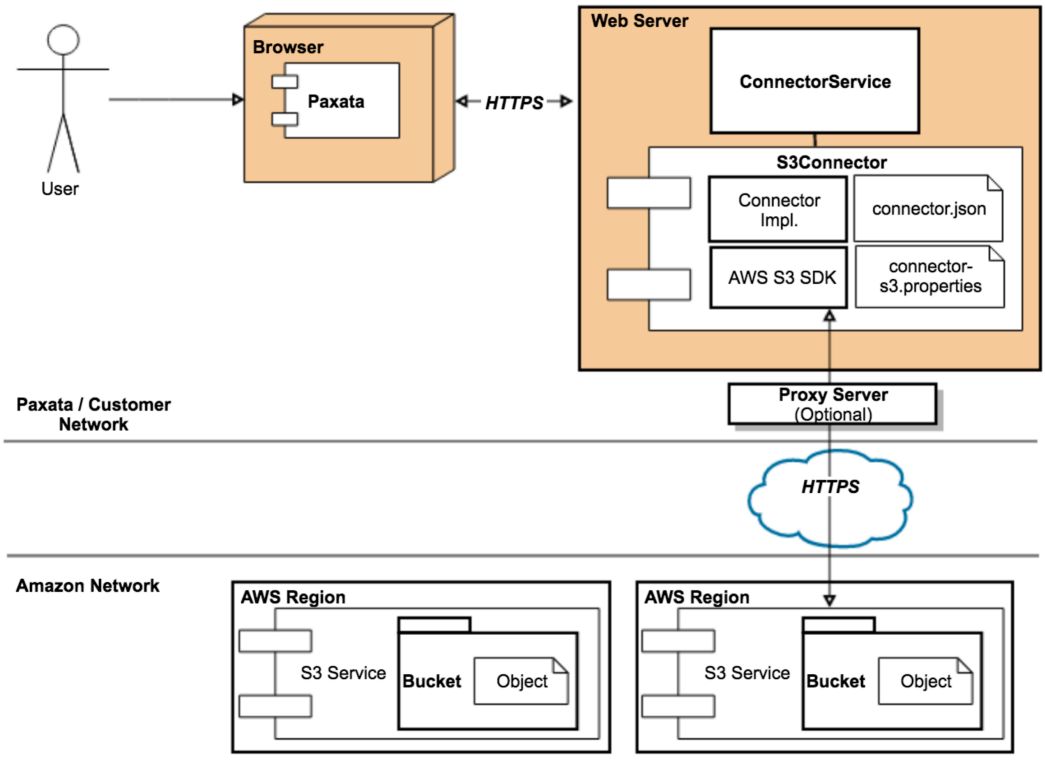
- This tutorial is built using IntelliJ IDEA development environment. However, we've made efforts to avoid providing IDE-specific commands and conveniences at all possible points.

- This tutorial is intended to accompany the completed Paxata S3 Connector source code.

Connector Overview

Paxata Connectors are primarily intended to provide connectivity to an external data store. In order to perform this function and to integrate with Paxata, the Connector must implement the Paxata Connector SPI (Service Provider Interface) which provides some common functionality to each connector as well as defining the interface that the Paxata server uses when communicating with the Connector.

In this tutorial, we'll be walking through the decisions and approaches taken when creating a Paxata Connector to allow for import and export of data stored within the Amazon Simple Storage Service (S3). The Connector will implement the Paxata Connector SPI (Service Provider Interface) which enables integration with Paxata. In order to establish connectivity with the S3 Service, it will also use the AWS S3 SDK for Java.



Connector Decisions

The S3 Service allows users to define up to 100 Buckets of data that can contain a potentially unlimited number of data sets, called S3Objects. Our connector needs to be able to support configurations that allow 3 levels of connectivity in order to allow administrators to limit access to specific subsets of data held within the S3 Service.

1. Connection to the top-level S3 Service that allows selection of any Bucket or S3Object within the specified S3 Region. We will refer to this type of connectivity as a **Service** connection.

2. Connection to a specific named Bucket of data within the S3 Service. We will refer to this type of connectivity as a **Bucket** connection.

3. Connection to a subset of S3Objects within a Bucket as identified by an S3 **Prefix**. We will refer to this type of connectivity as a **Prefix** connection.

Our Connector will implement a class, PXS3Object, which serves as an abstraction layer to support the definition of Service, Buckets, Prefixes and S3Objects within our code.

When the Connector is configure for each of these connection types, it will return a **Data Source** to Paxata. This data source serves as the primary interface between Paxata and the Data Store. In each connectivity scenario, above, the Data Store will be configure to hold a connection to a root DataSourceItem, called the **RootItem**, which serves as the top-level of a hierarchy of data that is available to the Paxata user. In each case, the path of the rootItem is defined publicly as "/", while the actual path is hidden and used when the Connector needs to import or export data. This allows Paxata to get the root item as a means of navigating the data hierarchy by calling the DataSource getItem("/") method.

For items held beneath the root item, such as files within a directory, the system will request them using the relative path to the file. Internally, the connector can leverage the internal path of the rootItem and the relative path of the child item to reference the actual

S3 Connection / Root Items

Connection Type	Root Item	Public Root Path	Private Root Path	Child Item Relative Path	Child Item Absolute Path
Service	A connection that allows access to all Buckets	/	/	bucketName	/bucketName
Bucket	A named S3 Bucket	/	/bucketName	data.csv	/bucketName/data.csv
Prefix	A named Prefix within a named S3 Bucket	/	/bucketName/Prefix	data.json	/bucketName/Prefix/data.json
Extended Prefix	A multi-level named Prefix within a named S3 Bucket	/	/bucketName/Prefix1/Prefix2/Prefix3	sales.xml	/bucketName/Prefix1/Prefix2/Prefix3/sales.xml

Although the following S3 features exist in the S3 SDK, they cannot be supported by the Paxata Connector SPI at this time.

- Bucket creation
- Bucket deletion
- S3 Object deletion

Connector S3 Implemented Features

Our S3 Connector will support the following features:

- Specification of a Web Proxy
    - Purpose: To allow on-premises Paxata users to leverage proxied access to external data resources.
    - Proxy Host / Proxy Port: Hostname and port number of web proxy server.
    - Proxy Username / Proxy Password: User credentials for authenticated proxy usage
- Specification of S3 Service details
    - AWS Region: Select the AWS S3 service region. Provided as a drop-down list for single selection in the Paxata UI.
    - Bucket name: Optional text field to allow specification of a named S3 Bucket. Allows configuration of connections with access to specific data sets.
    - Prefix: Optional field to allow specification of subsection of objects within an S3 Bucket.
    - Prefix Delimiter: Although default behavior for Prefixes is to use the "/" as a delimiter, we allow specification of alternate characters.

- Authentication leveraging AWS standard credentials
    - AccessKey
    - SecretKey
- Support for UI-based navigation of S3 buckets and objects:
    - Get List of all S3 Buckets
    - Get List of objects within an S3 Bucket
    - Get list of S3Object Prefixes, present as browsable directory hierarchy
- Import
    - Select one or more S3Objects for import
- Export
    - Select a Bucket or Prefix as export location for published Answer Set.
    - Export a prepared Answer Set to a Bucket

Determine Connector Options

Connector Options define the user-specified data elements to be collected in the Paxata UI and passed to the S3 Connector in order to enable connectivity, browsing, import and export features. Based upon the enabled features above, our connector options are:

Field	Configuration Field Name Localization Key	Data Type	Encrypt
Proxy Hostname	s3.proxyHostname	String	No
Proxy Port	s3.proxyPort	String	No
Proxy Username	s3.proxyUsername	String	No
Proxy Password	s3.password	String	Yes
S3 Region name	s3.awsRegion	String	No
S3 Bucket name	s3.bucketName	String	No
S3 Prefix	s3.prefix	String	No
S3 Prefix Delimiter	s3.prefixDelimiter	String	No
S3 Access Key	s3.accessKey	String	Yes
S3 Secret Key	s3.secretKey	String	Yes

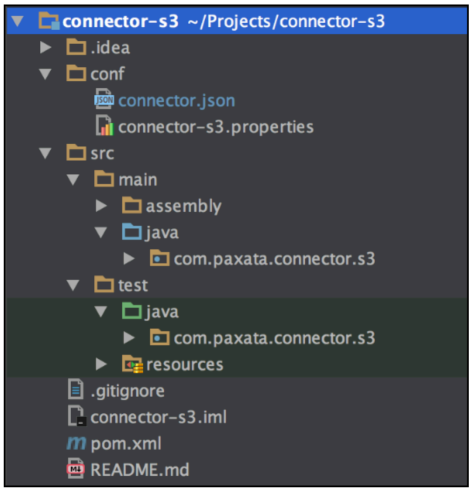
Building your Connector

Create a Java Project

Using your Java development tools, create a new standard Java project.

Create Project Structure

Create a directory structure for your Java project.



Details

- Ignore these:
  - .idea: tool-specific directory
  - .gitignore: code repository-specific file
  - connector-s3.iml: tool-specific metadata file
  - README.md: code repository-specific documentation file
- conf:
  - connector.json:
    - Will hold the configuration that drives the collection of your connector's options from the user.
  - connector-s3.properties:
    - Serves as the ResourceBundle property file and holds the externalized strings and messages that allow your Connector to be localized to support additional languages.
    - Referred to as the connector "bundle" file.
- src
  - main
    - assembly: Directory that will hold the Maven assembly.xml file, which enables creation of a packaged connector Zip archive.
    - java: holds Java source code for your Connector.
      - Create a java package named "com.paxata.connector.s3" to hold your connector source code.
  - test

        - java: holds Java test source code for your Connector.
            - Create a java package named "com.paxata.connector.s3" to hold your connector test source code.
        - resources: A directory to hold necessary files that will be used by your test.
- pom.xml: The Maven build file that will handle dependency resolution, compilation and packaging of your Connector.

Assembly File

The Maven Assembly plugin will be leveraged to build the Zip archive of your Connector. For more info: http://maven.apache.org/plugins/maven-assembly-plugin/

Steps:

1. Create an empty file named "**assembly.xml**" in the directory **src/main/assembly**
2. Open the file in your code editor
3. Paste the following contents into the file
4. Save the file.

Maven Assembly Configuration

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/plugins/maven-
assembly-plugin/assembly/1.1.0 http://maven.apache.org/xsd/assembly-
1.1.0.xsd">
   <id>bin</id>
   <formats>
     <format>zip</format>
   </formats>
   <fileSets>
     <fileSet>
       <directory>${project.basedir}</directory>
       <outputDirectory>/</outputDirectory>
       <includes>
         <include>conf/*</include>
       </includes>
```

```
    </fileSet>
  </fileSets>
  <dependencySets>
    <dependencySet>
      <outputDirectory>/lib</outputDirectory>
      <useProjectArtifact>true</useProjectArtifact>
      <scope>runtime</scope>
    </dependencySet>
  </dependencySets>
</assembly>
```

Maven Project File

1. Create a Maven *pom.xml* file in the root directory of your Java project.
2. Copy and paste the following contents into the pom.xml file.

Initial pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.paxata</groupId>
   <artifactId>connector-s3</artifactId>
   <version>1.0-SNAPSHOT</version>

   <!-- Specify the maven Repositories from which you will resolve
dependencies -->
   <repositories>
     <repository>
       <id>central</id>
       <name>Central</name>
       <url>http://repo1.maven.org/maven2</url>
     </repository>
   </repositories>

   <build>
     <plugins>
       <plugin>
       <groupId>org.apache.maven.plugins</groupId>
```

```
       <artifactId>maven-compiler-plugin</artifactId>
       <version>3.5.1</version>
       <configuration>
         <source>1.7</source>
         <target>1.7</target>
       </configuration>
       </plugin>
       <!-- Include the assembly plugin for building a zip archive of the
connectorm-->
       <plugin>
         <artifactId>maven-assembly-plugin</artifactId>
         <version>2.2.1</version>
         <configuration>
           <appendAssemblyId>false</appendAssemblyId>
           <descriptors>
             <descriptor>src/main/assembly/assembly.xml</descriptor>
           </descriptors>
         </configuration>
       </plugin>
     </plugins>
   </build>

   <!-- Specify the libraries that will be leveraged within your Connector
-->
   <dependencies>
   </dependencies>

 </project>
```

Configure dependencies

Our project needs to depend on several external software resources in order to implement the necessary functionality.

Paxata Connector SPI

The Paxata Connector SPI jar file is provided with your Paxata installation. Obtain this jar file from your administrator and place this jar on the classpath of your Connector project.

Example filename: connector-spi_2.10-2.2.13.jar

Amazon S3 Java SDK

Provides the library necessary to programmatically integrate from Java to the Amazon S3 service.

4. Copy and paste the following contents into the dependencies section of pom.xml file.

AWS S3 SDK dependency

```
<dependency>
   <groupId>com.amazonaws</groupId>
   <artifactId>aws-java-sdk-s3</artifactId>
   <version>1.10.67</version>
 </dependency>
```

Generic Utilities

These dependencies provide support for testing (junit) and utility functionality.

5. Copy and paste the following contents into the dependencies section of pom.xml file.

Additional dependencies

```
<dependency>
   <groupId>commons-io</groupId>
   <artifactId>commons-io</artifactId>
   <version>2.4</version>
 </dependency>
 <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
 </dependency>
```

Final Pom.xml with all dependencies

Final dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.paxata</groupId>
   <artifactId>connector-s3</artifactId>
   <version>2.2.1</version>
```

```
   <!-- Specify the maven Repositories from which you will resolve
dependencies -->
   <repositories>
     <repository>
       <id>central</id>
       <name>Central</name>
       <url>http://repo1.maven.org/maven2</url>
     </repository>
   </repositories>

   <build>
     <plugins>
       <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-compiler-plugin</artifactId>
       <version>3.5.1</version>
       <configuration>
         <source>1.7</source>
         <target>1.7</target>
       </configuration>
       </plugin>
       <!-- Include the assembly plugin for building a zip archive of the
connectorm-->
       <plugin>
         <artifactId>maven-assembly-plugin</artifactId>
         <version>2.2.1</version>
         <configuration>
           <appendAssemblyId>false</appendAssemblyId>
           <descriptors>
             <descriptor>src/main/assembly/assembly.xml</descriptor>
           </descriptors>
         </configuration>
       </plugin>
     </plugins>
   </build>

   <!-- Specify the libraries that will be leveraged within your Connector
-->
   <dependencies>
     <dependency>
       <groupId>paxata</groupId>
       <artifactId>connector-spi_2.10</artifactId>
```

```
      <version>2.2.13</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-java-sdk-s3</artifactId>
      <version>1.10.77</version>
    </dependency>
    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>2.4</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
  </dependencies>
</project>
```

Building your Project

Using the provided Maven Pom and Assembly files, the developer can use the following commands to build and package their Connector:

Task	Command
Clean your workspace	mvn clean
Compile your project	mvn compile
Package Zip Archive, run tests	mvn package assembly:assembly
Package Zip Archive, skip tests	mvn package assembly:assembly -DskipTests=true

Create Connector Options JSON file

Each Connector provides a JSON document that defines how the Paxata UI will present the configuration options that are required to establish connectivity with the remove data store. A completed Connector defines each Connector Option field, the UI groupings of these individual fields and whether the

1. Create file in the **conf** directory named "**connector.json**"
2. Open this file in the editor.
3. Open the file connector-template.json and paste the contents into your connector.json file.
4. Replace the "CONN_PREFIX" placeholder with "s3"
5. Using the following information, update the **top-level fields**
    a. class: com.paxata.connector.s3.S3ConnectorFactory
    b. name: s3.connector.name
    c. description: s3.connector.description
    d. bundle: connector-s3.properties
    e. groupId: paxata
    f. artifactId: connector-s3
    g. version: 0.1.0
    h. external: false
6. Using the following information, update the **options** settings to specify that this Connector will support import and export, but not querying.
    a. importSupported: true
    b. exportSupported: true
    c. querySupported: false
7. The **fields** values have been defined above. One notable change has been made for the convenience of the end user. The AWS Region field has been converted from a text field into a dropdown list containing the AWS specified Region names (left-side) and the localization keys (right-side) of the string presented in the Paxata UI form.
    a. Copy the following information and paste them into the fields JSON array.

Fields

```
{
  "key":"s3.proxyHostname",
  "fieldType":"text"
},
{
  "key":"s3.proxyPort",
```

```
  "fieldType":"number"
},
{
  "key":"s3.proxyUsername",
  "fieldType":"text"
},
{
  "key":"s3.proxyPassword",
  "encrypt":true,
  "fieldType":"text"
},
{
  "key": "s3.awsRegion",
  "fieldType": "dropdown",
  "fieldOptions": {
    "govcloud": "s3.awsRegion.option.GovCloud",
    "us-east-1": "s3.awsRegion.option.US_EAST_1",
    "us-west-1": "s3.awsRegion.option.US_WEST_1",
    "us-west-2": "s3.awsRegion.option.US_WEST_2",
    "eu-west-1": "s3.awsRegion.option.EU_WEST_1",
    "eu-central-1": "s3.awsRegion.option.EU_CENTRAL_1",
    "ap-southeast-1": "s3.awsRegion.option.AP_SOUTHEAST_1",
    "ap-southeast-2": "s3.awsRegion.option.AP_SOUTHEAST_2",
    "ap-northeast-1": "s3.awsRegion.option.AP_NORTHEAST_1",
    "ap-northeast-2": "s3.awsRegion.option.AP_NORTHEAST_2",
    "sa-east-1": "s3.awsRegion.option.SA_EAST_1",
    "cn-north-1": "s3.awsRegion.option.CN_NORTH_1"
  }
},
{
  "key":"s3.bucketName",
  "fieldType":"text"
},
{
  "key":"s3.prefix",
  "fieldType":"text"
},
{
  "key":"s3.prefixDelimiter",
  "fieldType":"text"
},
{
```

```
  "key":"s3.accessKey",
  "encrypt":true,
  "fieldType":"text"
},
{
  "key":"s3.secretKey",
  "encrypt":true,
  "fieldType":"text"
}
```

8. Define your Connector groupings specify the form groups that will be presented to users when collection your connector options

   a. Proxy settings form group

      i. Copy and paste the following text into the groupings JSON array to define the proxy settings form group.

   Proxy Grouping

```
{
  "name": "s3.proxy.group.legend",
  "fields": ["s3.proxyHostname", "s3.proxyPort", "s3.proxyUsername",
"s3.proxyPassword"]
},
```

   b. AWS settings form group

      i. Copy and paste the following text into the groupings JSON array to define the AWS settings form group

   AWS Grouping

```
{
  "name": "s3.aws.group.legend",
  "fields": ["s3.awsRegion","s3.bucketName", "s3.prefix",
"s3.prefixDelimiter","s3.accessKey", "s3.secretKey"]
}
```

9. Define the required and optional field settings for connector configuration fields that are defined by Paxata administrators. This is the first step of...

   a. Required fields

i. Copy the following entries into the file to specify that the AWS region must be specified during connector configuration

```
"required": ["s3.awsRegion"]
```

b. Optional fields

i. Copy the following entries into the file to specify that the proxy fields can optionally be specified during connector configuration

```
"optional": ["s3.proxyHostname", "s3.proxyPort",
"s3.proxyUsername", "s3.proxyPassword"],
```

10. Define the required and optional field settings for data source configuration fields that are defined by Paxata Tenant administrators

a. Optional fields

i. Copy the following entries into the file to indicate that they can optionally be populated during Data Source configuraiton

```
"optional": ["s3.bucketName", "s3.accessKey",
"s3.secretKey", "s3.prefix", "s3.prefixDelimiter"],
```

11. Define the required and optional field settings for session configuration fields that are defined by Paxata Tenant administrators and users

a. Required-if-undefined fields

i. Copy the following entries into the file to indicate that these fields must be populated if they have not been populated in previous steps

```
"required_if_undefined": ["s3.accessKey",
"s3.secretKey"],
```

12. supportedFormats

a. This connector will leverage the default settings for stream-based importing. This allows the connector to leverage the built-in parsing capabilities within Paxata.

Create the Resource Bundle Properties File

Our standard approach to support localization/internationalization of strings and messages that are presented in the Paxata UI is to leverage a Java Resource Bundle backed by properties files. In this approach, all strings and messages that appear in the Paxata UI have a unique key created. This key and the actual message are placed into the properties file. The key is then leveraged within the connector.json file and in Exceptions that are thrown from the Connector. The Paxata server handles resolution of the key to the message for the Connector Developer.

- More info: https://docs.oracle.com/javase/tutorial/i18n/resbundle/index.html

⚠ You will notice that each entry in the resource bundle properties file is prefaced with "s3.". This is done to avoid any potential collision with other properties within Paxata.

1. Create file in the **conf** directory named "**connector-s3.properties**"
2. Open this file in the editor.
3. Populate the entries that have been used for the connector.json **top-level fields**
    a. Add an entry for **name**:
        i. s3.connector.name=Amazon S3
    b. Add an entry for **description**:
        i. s3.connector.description=Amazon S3 Connector
4. Populate the entries that have been used for connnector.json **fields**

⚠ Fields presented in the UI have both a **label** and a **description**

    a.
    b. Add entries for Proxy fields
        i. Copy the following properties into the file

Proxy entries

```
s3.proxyHostname.label=Proxy Host
s3.proxyHostname.description=The host name or IP address of a
web proxy server used to access external web resources
s3.proxyPort.label=Proxy Port
s3.proxyPort.description=The port number of a web proxy
server used to access external web resources
s3.proxyUsername.label=Proxy Username
```

```
s3.proxyUsername.description=The username of a web proxy
server used to access external web resources
s3.proxyPassword.label=Proxy Password
s3.proxyPassword.description=The user password of a web proxy
server used to access external web resources
```

c. Add entries for AWS text fields

i. Copy the following properties into the file

AWS entries

```
# AWS options
s3.bucketName.label=Bucket Name
s3.bucketName.description=An S3 Bucket name, which represents
a directory of objects stored in Amazon S3
s3.prefix.label=Object Prefix
s3.prefix.description=A string that represents a hierarchical
prefix of a subset S3 data objects within a Bucket
s3.prefixDelimiter.label=Prefix Delimiter
s3.prefixDelimiter.description=A single character that
represents the delimiter within a hierarchical prefix.
s3.accessKey.label=Access Key ID
s3.accessKey.description=The ID associated with an AWS user's
Access Key.
s3.secretKey.label=Secret Key
s3.secretKey.description=The Secret Key associated with
an AWS user's Access Key.
```

d. Add entries for AWS Region

i. Copy the following properties into the file

AWS Region entries

```
# AWS Region
s3.awsRegion.label=AWS Region
s3.awsRegion.description=The Region name of a geographic
subdivision of the Amazon S3 Service
s3.awsRegion.option.GovCloud=us-gov-west-1
s3.awsRegion.option.US_EAST_1=us-east-1
s3.awsRegion.option.US_WEST_1=us-west-1
s3.awsRegion.option.US_WEST_2=us-west-2
s3.awsRegion.option.EU_WEST_1=eu-west-1
s3.awsRegion.option.EU_CENTRAL_1=eu-central-1
s3.awsRegion.option.AP_SOUTHEAST_1=ap-southeast-1
```

```
s3.awsRegion.option.AP_SOUTHEAST_2=ap-southeast-2
s3.awsRegion.option.AP_NORTHEAST_1=ap-northeast-1
s3.awsRegion.option.AP_NORTHEAST_2=ap-northeast-2
s3.awsRegion.option.SA_EAST_1=sa-east-1
s3.awsRegion.option.CN_NORTH_1=cn-north-1
```

e. Add entries for Paxata form **groupings**

i. Copy the following properties into the file to provide the names for the Paxata UI Form groups

Form groupings

```
# group keys
s3.proxy.group.legend=Web Proxy
s3.aws.group.legend=AWS S3
```

f. Add entries for Paxata Help text

i. Copy the following properties into the file to provide the help titles and placeholders for help text.

```
# Application Help
s3.connector.help.title=S3 Connector Configuration
s3.connector.help.text=TBD
s3.datasource.help.title=S3 Datasource Configuration
s3.datasource.help.text=TBD
s3.session.help.title=S3 Session
s3.session.help.text=TBD
```

g. Add entries for Exception messages

i. Copy the following properties into the file to provide keys and messages for expected exceptions that may arise

```
# Exception messages
s3.proxy.auth.error=Web Proxy username and password must both be
provided to leverage proxy authentication
s3.proxy.host.error=Web Proxy hostname and port must both be
provided to leverage proxied access
s3.proxyhost.option.exception=Both proxy host and proxy port are
required if configuring proxied access
s3.proxyuser.option.exception=Both proxy username and proxy
password are required if configuring authenticated proxied
access
s3.prefix.option.exception=Bucket name required, but not
provided with Prefix {0}. Provide Bucket name.
```

```
s3.service.unavailable=S3 service is not supported in specified
AWS Region
s3.region.endpoint.error=Unable to identify the Endpoint for the
selected AWS Region
s3.missing.option.exception=The required option {0} was not
provided
s3.service.region.exception=The S3 Region Name {0} is not valid
s3.service.exception=S3 service exception {0}
s3.client.exception=S3 client exception {0}
s3.configuration.exception=Exception while validating S3
connectivity. Verify configuration. {0}
s3.item.path.exception=Path provided for item retrieval is
invalid {0}
s3.itemPath.bucket.exception=Requested item path is missing S3
Bucket name {0}
s3.item.root.exception=The requested item {0} is not beneath the
configured base directory {1}
s3.export.exception=Exception while attempting to write output
to S3 {0}
s3.export.root.exception=Attempt to export file outside of a
Bucket
```

5. Save the file.

Create PxS3Object Class

Our PxS3Object class allows us to create an internal data type that wraps the object types provided by S3 Buckets, Prefix and S3Object. It also allows us to hold some context around these objects to support navigation and Connector DataSource configuration.

1. In your development environment, create the class "PXS3Client.java" in the com.paxata.connector.s3 package
2. Open the file in your editor.
3. Implementation:

   a. Create member variables:

   Copy the following code into your class:

```
private String name;
private Long size;
private S3Object s3Object = null;
private Bucket bucket = null;
private ObjectMetadata objectMetadata = null;
private S3ObjectType type = null;
```

b. Implement an enumeration that defines our PXS3Object types

i. Copy the following into your class

```
public enum S3ObjectType {
  BUCKET, PREFIX, S3OBJECT, SERVICE
}
```

c. Implement 3 constructors to support Buckets, Prefixes and S3Objects

i. Copy the following into your class

```
public PXS3Object (Bucket bucket) {
  this.type = S3ObjectType.BUCKET;
  this.bucket = bucket;
  this.name = bucket.getName();
  this.size = 0L;
}

public PXS3Object (S3Object s3Object) {
  this.type = S3ObjectType.S3OBJECT;
  this.s3Object = s3Object;
  this.name = s3Object.getKey();
  this.objectMetadata = s3Object.getObjectMetadata();
  this.size = objectMetadata.getContentLength();
}

public PXS3Object (Bucket bucket, String prefix) {
  this.type = S3ObjectType.PREFIX;
  this.bucket = bucket;
  this.name = prefix;
  this.size = 0L;
}
```

d. Implement the following getters and setters:

i. getName

ii. setName

iii. getSize

iv. getBucket

v. setBucket

vi. getS3Object

vii. getType

viii. getBucketName

1. This is the only methods with a special case.
2. Use the following implementation to return the Bucket name form the S3Object for S3ObjectType.S3Object

```
public String getBucketName () {
  if (this.type == S3ObjectType.BUCKET || this.type ==
S3ObjectType.PREFIX) {
    return this.bucket.getName();
  }
  return this.s3Object.getBucketName();
}
```

e. Save the file.

f. Compile.

4. Create a jUnit test class called PxS3ObjectTest to contain tests that validate this class functionality

a. Example Tests:

i. createPxS3FileObject

ii. createPxS3BucketObject

iii. createPXS3PrefixObject

Create a Client Class

Although the S3 SDK provides a client that we can use to interact with the S3 Service and our data stored there, we often follow the practice of creating our own Data Store client class to encapsulate all of the functionality that is leveraged by our Paxata Connector. Creation of this class will allow the developer to test connectivity, navigation, import and export outside of the scope of Paxata integration.

1. In your development environment, create the class "S3Client.java" in the com.paxata.connector.s3 package
2. Open the file in your editor.
3. Implement the following methods
    a. Constructor without input parameters
        i. Purpose: Provided as a convenience for unit testing.
    b. Constructor with input parameters
        i. Purpose: Instantiates a connection to the AWS S3 service targeting the specified S3 Region

ii. Input Parameters

1. String accessKey
2. String secretKey
3. String regionName
4. String proxyHost
5. Integer proxyPort
6. String proxyUserName
7. String proxyPassword

iii. Activities:

1. Instantiate an AWSCredentials object using the accessKey and secretKey
2. Instantiate an S3 ClientConfiguration object
3. if specified, use appropriate setters to populate the proxy settings on the ClientConfiguration object
4. Instantiate an AmazonS3Client object using your AWSCredentials and ClientConfiguration objects as input
5. Determine the S3 Endpoint URI using the provided S3 Region Name and set it on the AmazonS3Client
    a. **Note**: The S3 SDK does not provide a method for this. This requires a lookup against a Map within our Client to validate the region and return the appropriate URI.
6. Validate the connection
    a. **Note**: at this point you have an S3 Client, but it has not yet been connected to S3. Validation is essential at this point to verify that the client is configured correctly
    b. Our example code uses the client to execute the listBuckets() method.

c. createEndpointURI

i. Purpose: Using the provided AWS S3 RegionName, validate the region and create the appropriate S3 Endpoint URL.

ii. Input Parameters

1. String regionName

iii. Activities:

1. Lookup the S3 URI using the provided Region Name

d. validateClient:

i. Purpose: Validate that the connection instantiated in the constructor can

successfully connect using the configuration options that were provided. If listBuckets call throws an exception then rethrow to the constructor, else return true.

ii. Activities:

1. Make a call to the AmazonS3Client listBuckets() method within try/catch block

e. getBuckets:

i. Purpose: Obtain the list of Buckets that exist for the configured S3 connection.

ii. Activities:

1. Within a try/catch block, call the AmazonS3Client listBuckets()
2. Catch AmazonServiceException and AmazonClientException, rethrow as ConnectorIOException
3. Return List<Bucket>

f. getBucket

i. Purpose: Retrieve the S3 Bucket that matches the provided bucketName.

ii. Input Parameters:

1. String bucketName

iii. Activities:

1. Call AmazonS3Client doesBucketExist() method to verify existence of named Bucket
2. Within a try/catch block, call the AmazonS3Client listBuckets()
3. Iterate the list of returned Buckets and return specified Bucket

g. listObjects

i. Purpose: List the available S3Objects **and** Prefixes that exist for a specified Bucket and Prefix location.

ii. Input Parameters:

1. Bucket
2. String prefix
3. String delimiter

iii. Activities:

1. Catch AmazonServiceException and AmazonClientException, rethrow as ConnectorIOException
2. Create a List<PXS3Object> to hold values to be returned to caller
3. Create a ListObjectsRequest

a. Optionally set the prefix and delimiter if they were provided (neither null nor empty)

4. Call AmazonS3Client listObjects() proving the ListObjectRequest as input, receive an ObjectListing object
5. Extranct Common Prefixes
    a. Call ObjectListing's getCommonPrefixes() method
    b. Iterate the list of Prefixes
        i. Creating a PxS3Object for each
        ii. Add item to List.
6. Extract S3Objects
    a. Call ObjectListing's getObjectSummaries() method
    b. Iterate the list of ObjctSummaries
        i. Obtain the S3Object using the AmazonS3Client getObject() method
        ii. Creating a PxS3Object for each S3Object
        iii. Add item to List.
7. Return List<PXS3Object>

h. getObject:
    i. Purpose: Obtain an S3 Object identified by the provided key that exists in the specified Bucket
    ii. Input Parameters
        1. String bucketName
        2. String key
    iii. Activities:
        1. Call AmazonS3Client doesObjectExist() method to verify existence of named S3Object within specified Bucket
        2. If exists
            a. Create GetObjectRequest using bucketName and key
            b. Obtain the S3Object using the AmazonS3Client getObject() method
        3. Return S3Object or null if it does not exist

i. getObjectStream
    i. Purpose: Obtain an InputStream associated with an S3Object key in a named Bucket to enable reading file contents.

ii. Input Parameters:

1. String bucketName
2. String key

iii. Activities:

1. Create GetObjectRequest using bucketName and key
2. Obtain the S3Object using the AmazonS3Client getObject() method
3. Return S3ObjectInputStream using the AmazonS3Client getObjectContent() method
4. Catch AmazonServiceException and AmazonClientException, rethrow as ConnectorIOException

j. createFile

i. Purpose: Create an OutputStream to allow writing data to an S3Object within the specified Bucket

ii. Input Parameters:

1. Bucket bucket
2. String key
3. String encoding: the character encoding of the data being written to the file
4. String contentType: the mime type of the data being written to the file

iii. Notes:

1. S3 expects an InputStream to the data content to be written to the S3Object, but Paxata requires an OutputStream.
2. We have bridged this gap using Java Piped Streams and the ExecutorService to wrap a PipedInputStream in a PipedOutputStream

iv. Activities:

1. Instantiate an S3 ObjectMetadata object
2. If encoding and contentType are provided (neither null nor empty)
    a. Set encoding using the setContentEncoding() method
    b. Set contentType using the setContentType() method
3. Create PipedInputStream with a pipe size of **1048576**
4. Create a PipedOutputStream
5. Connect the inputStream to the outputStream
6. Using the ExecutorService, create a Runnable thread
    a. Within the thread, call the AmazonS3Client putObject() method to

write output to a new S3Object

b. Close input and output stream

7. return the PipedOutputStream to the caller

4. Save the file
5. Compile
6. Create a jUnit test class called S3ClientTest to contain tests that validate this class functionality
    a. Example Tests:
        i. createEndpointURI_Success
        ii. getEndpointURI_Fail
        iii. CreateS3ClientWithBadCredential
        iv. CreateS3Client
        v. CreateS3ClientWithProxy
        vi. CreateS3ClientWithProxyUser
        vii. CreateS3ClientWithBadProxyHost
        viii. CreateS3ClientWithBadProxyUser
        ix. getBucket_Succeed
        x. getBucketReadFailure
        xi. getBucketWrongRegion
        xii. getBuckets
        xiii. getObject
        xiv. getObjectNotExist
        xv. writeFileToBucket
        xvi. writePrefixedFileToBucket

Create a Path Utility Class

The S3PathUtil class is provided to consistently manage creation and parsing of absolute and relative paths according to the requirements of providing support or **Service, Bucket and Prefix** connections and relative item paths.

1. Create file S3PathUtil.java
2. Open the file in your editor.
3. Implement the following methods
    a. public static String relativizePath(String basePath, String absPath)

i. Purpose: Remove the scheme, host and port to produce a relative path to the requested S3 Object. Final relative path should be /[bucketName]/[object key]

ii. Use File and URI classes to relativize the absPath to the basePath

b. public static String buildRootPath (String rootPath, String bucketName, String prefix)

i. Purpose: Create an absolute path for the root item that corresponds to the user provided bucket name and prefix

ii. Combine rootPath, bucketName and prefix into a complete absolute path that represents the root item according to S3 path requirements.

iii. Path = /[bucketName]/[prefix]

c. public static PXS3Object.S3ObjectType getTypeForRootPath(String path)

i. Purpose: Determine whether the root path represents a Service, Bucket or Prefix connection

ii. Parse path to identify whether bucketName and prefix are populted

1. Both bucketName and prefix are populated => PREFIX

2. Only bucketName populated => BUCKET

3. Otherwise, SERVICE

d. public static String getLastPrefixName (String prefix, String delimiter)

i. Purpose: Get the right-most Prefix element from the provided full prefix path

ii. To support navigation through prefixes, we must extract the right-most prefix element for presentation in the navigation UI

e. public static String createAbsolutePathForItemPath (String rootPath, String path, String delimiter)

i. Purpose: Given the path elements provided, build an absolute path to the Item

ii. Combine rootPath, path and delimiter into a complete absolute path that represents the root item according to S3 path requirements.

f. public static String createAbsolutePathForItem (String rootPath, String bucketName, String objectName, String delimiter)

i. Purpose: Given the path elements provided for the S3Object, build an absolute path to the Item

ii. Combine rootPath, bucketName, objectName and delimiter into a complete absolute path that represents the root item according to S3 path requirements.

g. public static String getBucketName (String path, String delimiter)

i. Purpose: Extract the Bucket name from the provided absolute path to the item

h. public static String getS3ObjectName (String path, String delimiter)

i. Purpose: Extract the S3Object name from the provided absolute path to the item

4. Save the file
5. Compile
6. Create a jUnit test class called S3PathUtilTest to contain tests that validate this class functionality
    a. Example Tests
        i. buildPathforBucket
        ii. buildPathforBucketAndPrefix
        iii. buildPathforBucketAndPrefixAndFile
        iv. buildPathforService
        v. createAbsolutePathForBucket
        vi. createAbsolutePathForBucketItemWithBucket
        vii. createAbsolutePathForBucketItemWithBucketAndPrefix
        viii. createAbsolutePathForBucketWithBucketPath
        ix. createAbsolutePathForBucketWithPrefixPath
        x. createAbsolutePathForPrefix
        xi. createAbsolutePathForPrefixItemWithBucket
        xii. createAbsolutePathForPrefixItemWithBucketAndPrefix
        xiii. createAbsolutePathForPrefixWithBucketPath
        xiv. createAbsolutePathForPrefixWithPrefixPath
        xv. createAbsolutePathForService
        xvi. createAbsolutePathForServiceItemWithBucket
        xvii. createAbsolutePathForServiceItemWithBucketAndPrefix
        xviii. getBucketFromAbsPathForBucket
        xix. getBucketFromAbsPathForFile
        xx. getBucketFromAbsPathForPrefix
        xxi. getObjectFromAbsPathForBucket
        xxii. getObjectFromAbsPathForS3Object
        xxiii. getPrefixFromAbsPathForMultiPrefix
        xxiv. getPrefixFromAbsPathForPrefix
        xxv. relativizeBucketPathforService
        xxvi. relativizeMultiPrefixPathforBucket

xxvii. relativizeMultiPrefixPathforMultiPrefix

xxviii. relativizePrefixPathforBucket

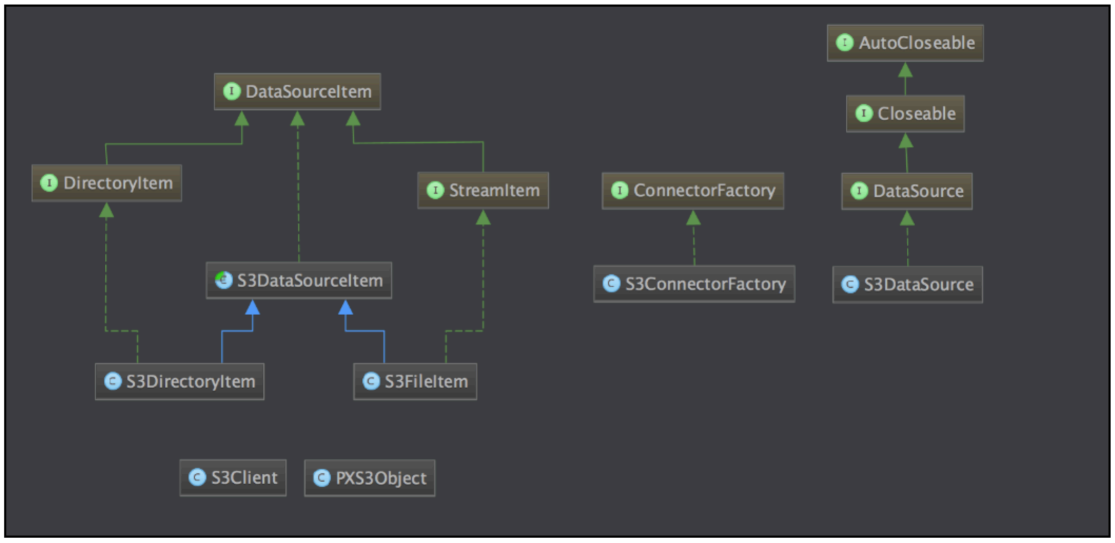
xxix. relativizePrefixPathforPrefix

xxx. relativizePrefixPathforService

Create Connector Classes

- Target Directory: connector-s3/src/main/java/com/paxata/connector/s3
- Package: com.paxata.connector.s3

Class inheritance structure



S3 Connector Factory

The S3ConnectorFactory is responsible for validation of user-specified

1. Create file S3ConnectorFactory.java
2. Update the class definition to specify that it "**implements ConnectorFactory**"
3. Add empty implementations of methods defined in ConnectorFactory interface. Your IDE may provide options to perform this.
    a. create(SessionInfo sessionInfo, Map<String, String> options)

b. validateOptions(SessionInfo sessionInfo, ValidationPhase validationPhase, Map<String, String> map, ConnectorDefinition connectorDefinition)

c. testConnection(SessionInfo sessionInfo, Map<String, String> map)

4. Add member variables to support reading options passed from the User. These must match the "keys" from the JSON document

   a. Copy the following code into your class

```
// Connector Factory Configuration items
private static final String proxyHostKey = "s3.proxyHostname";
private static final String proxyPortKey = "s3.proxyPort";
private static final String proxyUsernameKey
= "s3.proxyUsername";
private static final String proxyPasswordKey
= "s3.proxyPassword";
private static final String s3RegionKey = "s3.awsRegion";

// DataSource Configuration items
private static final String bucketNameKey = "s3.bucketName";
private static final String prefixKey = "s3.prefix";
private static final String delimiterKey = "s3.prefixDelimiter";

// Credentials Configuration items
private static final String accessKeyKey = "s3.accessKey";
private static final String secretKeyKey = "s3.secretKey";
```

5. Implement methods

a. validateOptions

i. Copy the following code into the method body

```
return ConnectorConfigurationParser.validateOptions(validationPhase
, map, connectorDefinition);
```

b. testConnection

i. Purpose: using the provided SessionInfo and user defined connection options, test connection to the Data Store

ii. Copy the following code into the method body to instantiate a DataSource and make a call to obtain the rootItem as defined by the provided options map

```
DataSource dataSource = create(sessionInfo, map);
try {
  dataSource.getRootItem();
} finally {
  try {
    dataSource.close();
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
}
```

c. create

i. Purpose: Validate the user supplied configuration items against the specification in the Connectors JSON file.

ii. Activities:

1. Obtain any needed SessionInfo values. Locale can occasionally be provided to WebAPIs to support localized messages
2. Using constants from step 4, read all user-supplied configuration values from options Map
    a. Default delimiter to "/"
    b. Convert values that should be numeric to the appropriate types. Example: proxy port
3. Instantiate an S3Client using provided configuration options
4. Instantiate and return a new S3DataSource object

6. Save the file
7. Compile
    a. Expect failure as we have not yet created the S3DataSource class.

8. Create a jUnit test class called S3ConnectorFactoryTest to contain tests that validate this class functionality
    a. Example Tests:
    i. FailCreateDataSourceWithMissingRegionKey
    ii. FailCreateDataSourceWithMissingAccessKey
    iii. FailCreateDataSourceWithMissingSecretKey
    iv. FailCreateDataSourceWithBadRegionName
    v. SucceedCreateDataSourceWithGoodOptions
    vi. VerifyConfig

S3 Connector Data Source Class

S3 Data Source is the primary integration point between Paxata and the Connector. This class implements the SPI DataSource interface.

1. Create Java class file S3DataSource.java
2. Update the class definition to specify that it "**implements DataSource**"
3. Add empty implementations of methods defined in ConnectorFactory interface. Your IDE may provide options to perform this.
    a. getFactory
    b. getRootPath
    c. getRootItem
    d. getItem
    e. close
4. Add member variables
    a. Copy the following code into your class

```
private S3ConnectorFactory s3ConnectorFactory;
private S3Client cnx;
private String prefix;
private String delimiter;
private S3DataSourceItem rootItem;
private PXS3Object.S3ObjectType rootType;
private String ROOTPATH = "/";
private String rootPath = "";
```

5. Implement Methods
    a. Constructor
        i. Purpose: Instantiate an S3DataSource, creating a root item containing an PxS3Object that is appropriate for the Bucket, Prefix and delimiter that is provided.
        ii. InputParameters:
            1. S3ConnectorFactory factory
            2. S3Client cnx
            3. String bucketName
            4. String prefix
            5. String delimiter
        iii. Activities:
            1. Set class member variables for factory, delimiter, cnx and prefix
            2. Determine rootPath for provided options
                a. Copy the following code into your method

```
rootPath = S3PathUtil.buildRootPath(ROOTPATH,
bucketName, prefix);
```

            3. Determine connection type of rootPath
                a. Copy the following code into your method

```
rootType = S3PathUtil.getTypeForRootPath(rootPath);
```

            4. Create an S3DirectoryItem to serve as the rootItem
                a. For SERVICE, use null as PXS3Object
                b. For Bucket and Prefix, user the appropriate PXS3Object constructor to create a PXS3Object

    b. getFactory
        i. Purpose: Getter that returns the ConnectorFactory class variable
    c. getRootPath
        i. Purpose: Getter that returns the public root path (ROOTPATH) class variable
    d. getRootItem
        i. Purpose: Getter that returns the root S3DataSourceItem class variable

e. getItem

i. Purpose: Return the DataSourceItem that corresponds to the provided relative path beneath our root item

ii. Activities:

1. Throw DataSourceConnectionException if path is null or empty.
2. If path == "/", return rootItem
3. Create absolute path for item leveraging rootPath, path and delimiter
4. Extract bucket name using S3PathUtil method
5. Extract object name using S3PathUtil method
6. If Bucket has been requested
    a. Instantiate an PXS3Object for the bucket
    b. Instantiate an S3DirectortyItem and return
7. If Prefix has been requested
    a. Instantiate ax PXS3Object for the bucket and object name
    b. Instantiate an S3DirectortyItem and return
8. If S3Object has been requested
    a. Instantiate ax PXS3Object for the S3object
    b. Instantiate an S3FileItem and return

f. close

i. Purpose: Provides the ability to close the S3Client connection

ii. There is no SDK method to close the client connection, here we just null out the cnx class variable

6. Save the file
7. Compile
    a. Expect failure as we have not yet created the S3DirectortyItem or S3FileItem classes.
8. Create a jUnit test class called S3DataSourceTest to contain tests that validate this class functionality
    a. CreateDataSourceForBucket
    b. CreateDataSourceForBucketWithPrefix
    c. CreateDataSourceForService
    d. CreateDataSourceForServiceAndGetItem
    e. GetAllBucketsForService
    f. GetBucketItemForService

g. GetFileItemForService

h. GetItemWithBadPath

i. GetMultiPrefixItemForService

j. GetPrefixItemForBucket

k. GetPrefixItemForService

l. NavigateChildrenOfRootItemWithBucket

S3 Connector Data Source Item

S3DataSourceItem is an abstract class that implements the SPI DataSourceItem and serves as the parent class for both S3DirectoryItem and S3FileItem

1. Create Abstract Java class file S3DataSourceItem.java
2. Update the class definition to specify that it "**implements DataSourceItem**"
3. Save the file
4. Compile

S3 Connector Directory Item

S3DirectoryItem class is intended to handle S3 objects that are treated as directories in Paxata. These objects are the S3 Service, Buckets and Prefixes.

1. Create Java class file S3DirectoryItem.java
2. Update the class definition to specify that it "**extends S3DataSourceItem implementsDirectoryItem**"
3. Add empty implementations of methods defined in DataSourceItem interface. Your IDE may provide options to perform this.
    a. getName
    b. getPath
4. Add empty implementations of methods defined in DirectoryItem interface. Your IDE may provide options to perform this.
    a. getChildren
    b. isCreateChildDirectorySupported
    c. createChildDirectory
    d. createStreamItem
    e. createRecordItem
5. Add member variables

    Copy the following code into your class:

```
private DataSource ds;
private S3Client cnx;
private String rootPath;
private String prefix;
private String delimiter;
private String path;
private String name;
private PXS3Object pxs3Object;
```

6. Implement Methods
    a. Constructor
        i. Purpose: Instantiate a S3DirectoryItem that represents an S3 Bucket or Prefix
        ii. Input Parameters
            1. DataSource ds
            2. S3Client cnx
            3. Boolean isRoot
            4. String rootPath
            5. String prefix
            6. String delimiter
            7. PXS3Object pxs3Object
        iii. Activities:
            1. Set class member variables for ds, cnx, rootPath, prefix, delimiter, pxs3Object
            2. If isRoot == true
                a. set path and name to "/"
            3. if isRoot != true
                a. if pxs3Object represents a bucket, create absolute path for Bucket
                b. else create absolute path for Prefix
                c. Relativize path against rootItem path
                d. set path == relativePath
                e. if type == BUCKET
                    i. set name = pxs3Object.getName
                f. else

i. set name = Right-most prefix of prefix path

b. getName

i. Purpose: Getter that returns the name class variable

c. getPath

i. Purpose: Getter that returns the path class variable

d. getChildren

i. Purpose: Obtain the list of child Prefixes and S3Objects that exist within the Bucket or Prefix represented by this S3DirectoryItem

ii. Activities

1. Test boundaries of start and count
    a. If start < 0 || count == 0, return an empty List<DataSourceItem>
2. If this item's root item is a SERVICE (pxs3Object is null)
    a. Get list of Buckets
    b. Create a PXS3Object for each Bucket
    c. Create a S3DirectoryItem for each PXS3Object
    d. Add S3DataSourceItem to result list
3. If this item's root item is a Bucket
    a. Call S3Client listObjects() method to obtain list of Prefixes and S3Objects
    b. Create S3Directory item for Preffixes as S3DataSourceItem
    c. Create S3FileItem for S3Objects as S3DataSourceItem
    d. Add S3DataSourceItem to result list
4. Apply pagination start and count to list
5. Returning DataSourceItemIteratorDelegate for sliced List of DataSourceItem

e. isCreateChildDirectorySupported

i. Purpose: Identify support for creation of child directories

ii. This method should return false.

f. createChildDirectory

i. Purpose: Creation of child directories is not supported for S3.

ii. Should return null.

g. createStreamItem

i. Purpose: Create a S3Object child object within the Bucket Prefix of this

S3DirectoryItem

h. createRecordItem

i. RecordItems are not supported for S3 at this time. This method should return null.

7. Save the file
8. Compile
9. Create a jUnit test class called S3DirectoryItemTest to contain tests that validate this class functionality

- Example Tests

i. CreateDirectoryItemForBucket
ii. CreateDirectoryItemForBucketWithPrefix
iii. CreateDirectoryItemForService
iv. getChildrenFromBucket
v. getChildrenFromBucketWithPrefix
vi. getChildrenFromService
vii. getChildrenLargeCount
viii. getChildrenLargeStart
ix. getChildrenNegativeCount
x. getChildrenNegativeStart
xi. getChildrenSmallCountRepeated
xii. getChildrenZeroCount
xiii. putObjectToBucket
xiv. putObjectToPrefix
xv. putObjectToService
xvi. putPrefixObjectToBucket
xvii. RootItemAsS3Bucket
xviii. RootItemAsTopLevelS3Connection

S3 Connector File Item

S3FileItem class is intended to handle S3 Objects that are treated as files in Paxata. These objects are the S3Objects.

1. Create Java class file S3FileItem.java
2. Update the class definition to specify that it "**extends S3DataSourceItem implementsStreamItem**"

3. Add empty implementations of methods defined in DataSourceItem interface. Your IDE may provide options to perform this.
    a. getName
    b. getPath
4. Add empty implementations of methods defined in StreamItem interface. Your IDE may provide options to perform this.
    a. getSize
    b. getMimeType
    c. getEncoding
    d. getInputStream
5. Add member variables
    a. Copy this code into your class file

```
private DataSource ds;
private S3Client cnx;
private String delimiter;
private String path;
private S3Object s3Object;
```

6. Implement Methods
    a. Constructor
        i. Purpose: Instantiate this S3FileItem that represents an S3Object that contains data
        ii. Input Parameters
            1. DataSource ds
            2. S3Client cnx
            3. String rootPath
            4. String prefix
            5. String delimiter
            6. PXS3Object pxs3Object
        iii. Activities:
            1. Set class member variables for ds, cnx, delimiter, pxs3Object
            2. Create the absolute path for this S3FileItem using the rootPath, BucketName, object name and delimiter
            3. Set path to the relativized absolute path against the rootPath

b. getName
    i. Purpose: Returns the name of this S3FileItem
c. getPath
    i. Purpose: Returns the relative path to this S3FileItem
d. getSize
    i. Purpose: Return the size, in bytes, of the s3Object represented by this S3FileItem
e. getMimeType
    i. Purpose: Obtain the mime type of the s3Object represented by this S3FileItem to drive parsing of the data stream
    ii. S3 mime type issue:

> ⚠ The mime type (contentType) obtained from the S3Object metadata is prone to errors based upon the client that is used to upload the data. We've experienced issued importing data because the mime type was inaccurate. For example, A CSV file was identified as application/octet-stream rather than text/csv. For this reason, we do not rely on S3 metadata content types when importing data onto Paxata.

f. getEncoding
    i. Purpose: Return the character set encoding of the s3Object represented by this S3FileItem to drive parsing
g. getInputStream
    ii. Purpose: Obtain the InputStream of the s3Object represented by this S3FileItem to support import into Paxata

7. Save the file
8. Compile
9. Create a jUnit test class called S3FileItemTest to contain tests that validate this class functionality
    a. Example Tests
        i. getS3ObjectForService
        ii. getS3ObjectForBucket
        iii. getS3ObjectForBucketWithPrefix
        iv. getPrefixedS3ObjectForService
        v. getPrefixedS3ObjectForBucket

Appendix A: Java docs for Connector Interfaces and Methods

Java docs for the Connector SPI are included in the **javadoc.jar** file as part of the **Paxata Connector SPI** build. Alternatively, you can create your own Java docs from the SPI source code that you download from the GitHub repository: https://github.com/Paxata/connector-spi

Appendix B: JSON Configuration File Example

The following JSON format defines data objects for a Paxata FTP connector.

```
{
  "class":"com.paxata.connector.ftp.FTPConnectorFactory",
  "name":"ftp.connector.name",
  "description":"ftp.connector.description",
  "bundle":"connector-ftp",
  "groupId":"paxata",
  "artifactId":"connector-ftp",
  "version":"${version}",
  "options": {
    "importSupported": true,
    "exportSupported": true,
    "querySupported": false,
    "wildcardSupported": false
  },
  "fields": [
    {
      "key":"ftp.protocol",
      "type":{
        "dropdown":{
          "options":{
            "FTP":"ftp.protocol.option.ftp",
            "FTPS":"ftp.protocol.option.ftps",
            "SFTP":"ftp.protocol.option.sftp"
          },
          "type":"text"
        }
      }
    },
    {
      "key":"ftp.hostname",
      "type":"text"
    },
    {
      "key":"ftp.port",
      "type":"number"
    },
    {
      "key":"ftp.rootDirectory",
      "type":"text"
    },
    {
      "key":"ftp.baseDirectory",
      "type":"text"
    },
```

```
    {
      "key":"username",
      "type":"text"
    },
    {
      "key":"password",
      "encrypt":true,
      "type":"text"
    }
  ],

  "groupings": [
    {
      "name": "ftp.host.group.legend",
      "fields": ["ftp.protocol", "ftp.hostname", "ftp.port"]
    },
    {
      "name": "ftp.config.group.legend",
      "fields": ["ftp.rootDirectory", "ftp.baseDirectory"]
    },
    {
      "name": "ftp.credentials.group.legend",
      "fields": ["username", "password"]
    }
  ],

  "connector":{
    "required": ["ftp.protocol", "ftp.hostname", "ftp.port"],
    "optional": ["ftp.rootDirectory"],
    "required_if_undefined": [],
    "help": "ftp.connector.help"
  },
  "datasource":{
    "required": [],
    "optional": ["ftp.baseDirectory", "username", "password"],
    "required_if_undefined": [],
    "help": "ftp.datasource.help"
  },
  "session":{
    "required": [],
    "optional": [],
    "required_if_undefined": ["username", "password"],
    "help": "ftp.session.help"
  },

  "supportedFormats": {
    "import": {
      "stream": "Default",
```

```
      "record": []
    },
    "export":{
      "stream": "Default",
      "record": []
    }
  }
}
```

Appendix C: Properties Configuration File Example

The following example is associated with the FTP JSON configuration file example from Appendix B.

```
ftp.connector.name=FTP
ftp.connector.description=FTP Connector

# Field Definitions
ftp.protocol.label=FTP Protocol
ftp.protocol.description=FTP Protocol to use for communication with the FTP
Server
ftp.protocol.option.ftp=FTP
ftp.protocol.option.ftps=FTPS
ftp.protocol.option.sftp=SFTP

ftp.hostname.label=SFTP Hostname
ftp.hostname.placeholder=sftp.acme.com
ftp.hostname.description=SFTP server hostname or IP address

ftp.port.label=FTP Port
ftp.port.description=FTP Port Number

ftp.rootDirectory.label=Root Directory
ftp.rootDirectory.placeholder=/user/foo
ftp.rootDirectory.description=The root path on the Data Store accessible by
this connector

ftp.baseDirectory.label=Base Directory
ftp.baseDirectory.placeholder=/bar
ftp.baseDirectory.description=The root path on the Data Store accessible by
this Data Source

username.label=Username
username.placeholder=ftpuser
username.description=User for connecting to the FTP server

password.label=Password
password.description=Password for connecting to the FTP server

# Group Labels
ftp.host.group.legend=FTP Host
ftp.config.group.legend=Configuration
ftp.credentials.group.legend=Credentials
```

```
# Online Help
ftp.connector.help.title=FTP Connector Parameters
ftp.connector.help.text=Allows you to connect to an FTP server for Data
Library imports and exports. The following fields are used to define the
connection parameters.<br><br><strong>FTP Host</strong><br><b>FTP
Protocol</b>: unsecured, secured or SSH SFTP<br><b>FTP hostname</b>: the
server hostname<br><b>FTP Port</b>: the server port
number.<br><br><strong>Configuration</strong><br><b>Root Directory</b>:
specifies the top-level of the directory structure from which import/export
of data is enabled.

ftp.datasource.help.title=FTP Data Source Parameters
ftp.datasource.help.text=Allows you to connect to an FTP server for Data
Library imports and exports. The following fields are used to define the
connection parameters.<hr><strong>Configuration</strong><br>Base Directory:
specifies the top-level of the directory structure from which import/export
of data is enabled using this Data Source.<br><br>Note: this directory must
be a child of the Root Directory provided in the Connector
Configuration.<hr><strong>Credentials</strong><br>User authentication can be
done through a Shared Account or an Individual Account. If credentials are
not configured with the Data Source, they user will be prompted for
credentials.<br><br>Username: The username used to authenticate on the FTP
server.<br>Password: The password used to authenticate on the FTP server.

ftp.session.help.title=FTP Session
ftp.session.help.text=In order to connect to the FTP server, credentials must
be provided.<br><b>Username</b>: The username used to authenticate on the
SFTP server<br><b>Password</b>: The password used to authenticate on the SFTP
server.
```

Appendix D: Packaging Your Connector

Your Connector is deployed into the platform as a zip file using the REST API.
Your zip file must contain the following two folders:

- **lib/**
  this folder contains all **.jar** and **.lib** files that your Connector requires—including the **Paxata Connector SPI** jar file. This folder may also contain any configuration files that your Connector requires in the class path.
  Note: this folder and all of the .jar files within it are included in the Java class path at run time.

- **conf/**
  this folder contains the JSON configuration file—"`connector.json`"—and all `.properties` files that include the resource bundles required by your Connector.
  Note: this folder is included in the Java class path at run time.

Example:

```
conf
conf/connector-jdbc.properties
conf/connector.json
lib
lib/com.lib1.jar
lib/com.lib2.jar
lib/com.lib3.jar
lib/com.lib4.jar
lib/paxata.connector-jdbc.jar
lib/paxata.connector-spi.jar
lib/runtime_conf.properties
```

Your Connector is deployed to the platform using the "Connector Factory" endpoint defined in the *REST API*. See the *REST API* documentation for details.

Appendix E: Stream-Based File Types Supported by the Platform

The following provides an inclusive example of all platform-supported "import" stream formats. If your Connector will not support all stream-based formats, copy the formats you need from this example.

```
"import":{
   "stream":[
      {
         "key": "separator",
         "displayName": "import.stream.parser.delimited.name",
         "extensions": ["csv", "tsv"],
         "mimeTypes": ["text/csv", "text/x-csv", "text/tab-separated-value"]
      },
      {
         "key": "fixed-width",
         "displayName": "import.stream.parser.fixed.name",
         "extensions": ["txt"],
         "mimeTypes": ["text/plain", "text/fixed-width"]
      },
      {
         "key": "excel",
         "displayName": "import.stream.parser.excel.name",
         "extensions": ["xls", "xlsm", "xlsx"],
         "mimeTypes": [
           "application/msexcel application/x-msexcel",
           "application/x-ms-excel",
           "application/vnd.ms-excel",
           "application/x-excel",
           "application/xls application/x-xls",
           "text/xml/xlsx",
           "application/vnd.openxmlformatsofficedocument.spreadsheetml.sheet"
         ]
      },
      {
         "key": "json",
         "displayName": "import.stream.parser.json.name",
         "extensions": ["json", "js"],
         "mimeTypes": [
            "application/json",
            "application/javascript",
         "text/json"]
      },
      {
         "key": "xml",
         "displayName": "import.stream.parser.xml.name",
```

```
            "extensions": ["xml", "rdf"],
            "mimeTypes": ["application/xml", "application/rdf+xml"]
         },
         {
            "key": "avro",
            "displayName": "import.stream.parser.avro.name",
            "extensions": ["avro"],
            "mimeTypes": ["avro/binary"]
         }
      ]
},
```

The following provides an inclusive example of all platform-supported "export" stream formats. If your Connector will not support all stream-based formats, copy the formats you need from this example.

```
"export":{
   "stream":[
      {
         "key": "separator",
         "displayName": "export.stream.parser.delimited.name",
         "extensions": ["csv"],
         "mimeTypes": ["text/csv"]
      },
      {
         "key": "fixed-width",
         "displayName": "export.stream.parser.fixed.name",
         "extensions": ["txt"],
         "mimeTypes": ["text/fixed-width"]
      },
      {
         "key": "json",
         "displayName": "export.stream.parser.json.name",
         "extensions": ["json"],
         "mimeTypes": ["application/json"]
      },
      {
         "key": "xml",
         "displayName": "export.stream.parser.xml.name",
         "extensions": ["xml"],
         "mimeTypes": ["application/xml"]
      },
      {
         "key": "avro",
         "displayName": "export.stream.parser.avro.name",
         "extensions": ["avro"],
         "mimeTypes": ["avro/binary"]
      }
   ]
}
```

Contacting Support

Overview

Support will be provided for technical issues and enhancement requests for Paxata-supplied software. This includes features and functions, error-messages, documentation, and feature requests.

Self-help and community forums are available on the Paxata Service Desk web site; they are accessible to current application subscribers.

Response Times

After a user request is received, a follow-up response from Paxata Service Desk personnel is sent to the user in a period of time no longer than the intervals defined in the table below:

Priority	Minimum response time
Urgent	1 business day
High	1 business day
Normal	2 business days
Low	2 business days

Business Hours

A business day is defined as eight hours within normal operating hours for the Paxata Service Desk. The Paxata Service Desk is available from 8:00am to 4:00pm in the U.S. Pacific time zone; it operates Monday through Friday. The Paxata Service Desk does not have business hours on U.S. Holidays.

Priority

A user request for support is assigned a priority based on the following table:

Priority	Characteristics
Urgent	This type of request demonstrates that the application failed completely or that the incident reasonably represents a significant work stoppage.
High	This classification indicates that software is operational but features do not perform as expected; also applicable if the fault has a significant impact on execution of daily tasks.
Normal	This priority is given if the incident represents behaviors that are unexpected but do not materially impact daily work.
Low	A request for how to perform a task or use a feature will carry this classification. This applies also to a request for a future feature or function.

Service Desk Channels

The Paxata Service Desk maintains connection with users with the following methods:

Method	Contact information
Phone	+1 (650) 542.7869
Email	`servicedesk@paxata.com`
Twitter	`@Paxata_News`
Facebook	`www.facebook.com/paxata`
Service Desk web site	`servicedesk.paxata.com`