Watson Explorer Engine Connector SDK v1.0.0

Contents

Chapter 1. Watson Explorer Engine Connector SDK Overview

The Watson™ Explorer Engine Connector SDK Documentation describes how to build a custom connector that can enable you to crawl, retrieve, and index data from a data repository and make that data searchable with Watson Explorer Engine or Watson Explorer Application Builder.

The document will guide you through the connector development process with procedures, exercises, and sample code, that will enable you to develop a fully functional connector from start to finish.

Contents

The Connector SDK includes the scripts, code examples, exercises, procedures, and connector requirements to build a Watson Explorer connector in Java or Scala.

The Connector SDK documentation is structured as follows:

Connector Development Workflow
Outlines the planning that is required for developing a connector from start to finish using the Connector SDK.

Getting Started
Provides an overview of the Connector SDK package and includes procedures on compiling and building the example connectors.

Connector Examples
Delivers three complete examples of connectors, with increasing levels of complexity. The example connectors include the following (starting with the least complex):

Hello World Connector
The Hello World connector provides an example of a proof-of-concept basic connector. An enhanced version of the Hello World connector is the The Hello World Extended connector. The Hello World Extended connector builds upon the Hello World Connector with a series of exercises to enhance the Hello World proof-of-concept connector.

Note: The Connector SDK provides a Hello World connector example in both Java and Scala. The solution for either is the same. Therefore, only the Java version is described in the documentation.

Filesystem Connector
The Filesystem connector provides an example of a connector that can crawl a typical filesystem and use authentication. Additionally, the Filesystem with Continuous Update connector provides a similar example of the previous filesystem connector that is further enhanced to use some features of Watson Explorer Continuous Update mode.

Note: Because Filesystem does not provide information about deleted files, the Filesystem with Continuous Update connector provides only some aspects of integrating Continuous Update mode. It is intended to serve as a starting point for further

development and as an introduction to the complexity of adding the Continuous Update feature into connector code.

LDAP Connector
The LDAP connector provides an example of a protocol based-connector that can crawl repositories that are accessible by using the Lightweight Directory Access Protocol (LDAP).

Note: This connector is the most complex connector provided and serves as a model for other protocol based connectors.

Troubleshooting
Provides tips for troubleshooting common connector pitfalls and includes connector debugging techniques.

Additional Sources

Using the Connector SDK requires technical knowledge in several areas.

Authentication and Security
The Connector SDK assumes you are familiar with basic authentication mechanisms and securing web applications. The Connector SDK does not go into detail behind any one particular security protocol or method to invoke security. Beyond typical ACLs, many systems have custom security binding protocols and to describe a particular one here would be to provide a solution to that one alone. Custom security implementations require deep knowledge of the source data repository and it's composition.

For a primer on basic Watson Explorer Engine authentication see the authentication tutorial on IBM Knowledge Center at Tutorial: Applying Access Controls to Search Results.

AXL The Connector SDK does not provide a full function reference for AXL, an XML-based domain specific language that is used in the Watson Explorer Engine connector framework.

For more information on AXL functions, see the Watson Explorer Engine documentation on IBM Knowledge Center at http://www.ibm.com/support/knowledgecenter/SS8NLW_11.0.0/watsonexplorer_11.0.0.html.

Tip: You can search for AXL references in the IBM Knowledge Center by entering `viv:`*`function-name`* in the search box, where *`function-name`* is the actual name of the AXL reference. For example: Searching for `viv:crawl-enqueue-url` will take you to a search results page that provides a link to that AXL reference.

Converter Development
The Connector SDK does not describe how to develop the converters that may be needed to crawl a custom repository. However, Watson Explorer comes with predefined converters that can be used out of the box. Additionally, the Connector SDK describes the role of a converter within the context of a connector and provides a converter XML code sample as guidance.

For more information on Watson Explorer converters, see the converters documentation on IBM Knowledge Center at Converting

Java / Scala
The Connector SDK assumes proficiency at Java or Scala programming.

Although you do not need to know Scala, you will need to understand Java to follow along and build the connector examples.

Watson Explorer Administration
The Connector SDK assumes a familiarity with Watson Explorer Engine administration, system requirements, typical deployment scenarios and therefore does not explain the administrative tasks as part of the connector development procedures.

For more information on Watson Explorer Engine administration, see the Watson Explorer Foundational Components documentation on IBM Knowledge Center at IBM Watson Explorer.

Connector Development Workflow

Planning the development of a connector begins with a conceptual overview of the primary components of a connector and an understanding of the roles that they play in the architecture of a connector.

Connector SDK Framework

The framework of the SDK enables the crawler and connector to work recursively to retrieve, crawl, index and deliver data according to the node structure of a particular data repository. The example connectors provided in this Connector SDK are all built upon this conceptual framework. This framework can be described in terms of the following functional areas:

Connecting and communicating
Connecting and communication require the following connector capabilities:

- **Connecting** - Describes the process of making a connection between the connector and your data repository.
- **Authentication** - Describes the process of establishing that the user who is initiating the connection via a URL has the right privileges required by the remote data repository to access the repository.
- **Running** - Describes the process of a crawler actively indexing information in a remote repository.
- **Stopping** - Describes the process of stopping the crawler from indexing information.
- **Disconnecting** - Describes the process of disconnecting a connector from the remote data repository.

Crawling
Describes the crawler mechanism of actively identifying and retrieving data in the remote repository and passing that data along to be indexed.

Basic Connector Terminology

This section provides a description of basic connector terminology.

Connector Framework
The connector framework enables the crawler and connector to work recursively to retrieve, crawl, index and deliver data and have the same information.

ConnectorNode
The Java object-representation of a `crawl-url` XML element.

ConnectorWorker
Is a class that is called when an action is requested by the connector.

Continuous Update
The mechanism that enables new, updated, or deleted repository data to be continuously indexed. As a result, repository updates such as document modifications, and other changes, can be searched as quickly as possible. Using a connector in Continuous Update mode typically means the crawler should never need to be manually stopped and restarted, unless you are performing maintenance on your data repository.

Crawl-URL
The URL that the crawler uses as a starting point to access data in a remote data repository.

Important: This is not an actual URL but is an identifier for the location of data source repository.

Crawler
The Watson Explorer Engine mechanism to retrieve information in a data repository and pass it along into a conversion chain that normalizes the data into an indexable format.

Crawling Seed
The root URL to access resources such as fileshares, SMB shares, databases, email archives, and other data repositories that are accessible by various web protocols. Different seed URLs have different capabilities. Seeds can be repository-specific to enable connectors to crawl specific third-party applications such as Salesforce, SharePoint, IBM Domino, and other client relationship management (CRM) systems, product life cycle (PLC) systems, content management systems (CMS), cloud-based applications, and other web database applications. Each of the connector examples provided in this SDK provide a seed to crawl that particular resource.

Fetcher
Is a class that is responsible for communicating with a data repository.

Gradle
A freely available build automation tool. Gradle is used to build the connectors in the Connector SDK.

GuiceConnectorWorker
The GuiceConnectorWorker creates all of the appropriate bindings from the node for its `CrawlOptions`, `ConnectorOptions`, and XML attributes. It passes the modified `ConnectorNode` into a `HandlerChainNodeProcessor`.

Handler
Is a class that uses properties from the specialized node to perform some functions.

InputWorker
The InputWorker continually reads single nodes from the connection, identifying a node as either a `crawl-url` element or `pipeline-size-reply` element.

Scala Unlike Java, Scala has many features of functional programming languages like Scheme, Standard ML and Haskell, including currying, type inference, immutability, lazy evaluation, and pattern matching.

Specialized Node
Is a node that provides a specific instruction to a Handler. The specialized

node informs a Handler if it should be activated and, if appropriate, use properties from the specialized node, to perform a specific function.

Common Connector Architecture

The crawler and the connector interact through a sequence of processes that involve various connector mechanisms each with a specific task. The mechanisms and the sequence through which they interact comprise a common connector architecture as displayed Figure 1.

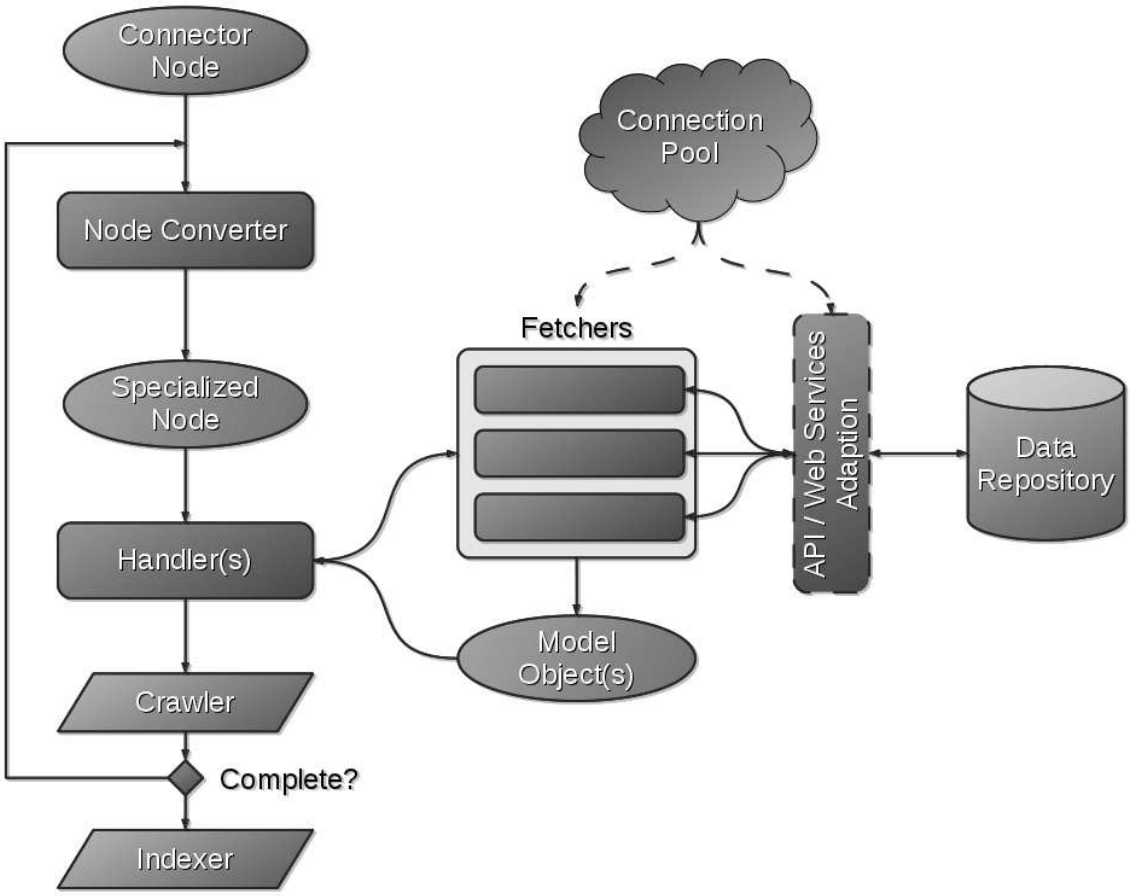

Figure 1. Common Connector Architecture

The Connector Framework passes a `ConnectorNode` into a Node Converter. The converter will use `GuiceConnectorWorker` to create all of the appropriate node bindings for its `CrawlOptions`, `ConnectorOptions`, and XML attributes. Next, it passes the modified `ConnectorNode` into a `HandlerChainNodeProcessor`. The `HandlerChainNodeProcessor` converts the `ConnectorNode` into a specialized node which is then passed into a chain of one or more Handlers.

A handler uses properties from the specialized node to perform some function. These function might include: querying a repository for document IDs, fetching data from a repository, and sending new `ConnectorNodes` (or `crawl-urls`) to the Crawler.

After processing the node, a handler has the option of passing the node to the next handler in the chain, or stopping further processing if there is not another handler in the chain. If a (child) `ConnectorNode` created at this stage contains all of the data it needs then the crawler does not need to send it back to the connector for more processing in which case the handler can mark the `ConnectorNode` as complete before sending it to the crawler.

When retrieving data from a repository, Handlers may employ one or more Fetchers. These Fetchers are responsible for communicating with the data repository, directly or potentially through an adapted API. Fetchers will create a Model Object by translating the repository data into a format expected by the handler using them.

Planning Overview

The development workflow that you can use to plan your connector starts with basics such as obtaining a development repository and getting sample data from that development repository.

At the planning stage, your goal is to identify the structure of the data contained in the external system that you are trying to crawl. The structure of that data might be comprised of systems that have users, communities, or documents. Moreover, the data may be nested, such as is typically the case with filesystem or email data. Next, you will consider if you have the right access to the data that you want to crawl. Other considerations include API usage limitations. You should be capable of performing a basic API call to your repository to determine the type of data you will get back. Does that data meet your needs?

Once you are able to retrieve data, you will want to consider the storage requirements for that data. Moreover, what will that data need to look like to be useful? Will that information need converted into other data types or is the data returned from an API call suitable enough in its native format to pass along with no additional conversion? At the conceptual level of planning a connector, you will want to have a notion of the connector features that you may want to include in the connector. For example, will your connector need to support a security model and if so, are you sufficiently knowledgeable about how that security model works?

Other features to consider may be server load considerations. Once you start crawling a resource, is that resource sufficiently provisioned to handle a crawl? What about detecting changes to your index? Will your connector require the ability to detect changes to the index continuously without restarting the crawler? This entails adding Continuous Update mode to your connector which increases the complexity of connector development. Understanding the structure of your data is a key starting point for connector planning and developing an initial connector workflow. Once you understand your data, and have thought about the above considerations, you can configure a crawler seed with the requirements that you need to connect to your data repository.

Identifying Connector Requirements

Identifying connector requirements entails reviewing your connector resources, performance needs, risk factors, and understanding the data you wish to crawl with your connector.

What follows are recommendations to consider when identifying your connector requirements:

- Identify the data size, number of documents, and the hardware and software components of the type of repository that you will crawl. If you are going to connect to a data repository, you should have a good idea of what that data repository contains. For example, does it contain millions of documents? Is the data structured or unstructured? How much storage does your repository use? Is your data distributed across many systems? If you are tasked with building a connector for a repository, then be sure to understand the data that you will be dealing with by consulting with the data repository administrator, if that person is not you.
- Determine your connector performance requirements. Do you have measurable performance metrics that your connector must satisfy? Again, if that person is not you, consult with the data repository administrator for the repository you want to crawl and learn what performance metrics your connector must satisfy for your environment.
- Asses development and test environment availability and comparability to a production environment You want a test environment that is as similar as possible to your production environment.

  Important: Be cautious not to execute any API calls that modify the data on your source system. Additionally, if you have a development system, be mindful that you do not use the connector to modify the source system unintentionally.
- Develop a robust set of test cases that will validate your connector development. If you have QA resources, consult them in outlining possible testing requirements that may have been overlooked.

By identifying connector requirements, you will have a better understanding of your data, what resources you have available to develop the connector, and will have a clear understanding of the risks associated with crawling your data repository.

Writing Overview

The connector writing process requires a conceptual understanding in several key areas to ensure that when you develop a connector you are prepared to do so. Generally, these key areas are followed sequentially. They include the following:

1. **Understanding the data repository** - First, you need to figure out the structure of the data contained in the external system. Is it a flat set of types, like Sharepoint (webs, lists, list items)? Is it a nested structure, like a filesystem or email? Understanding the structure of your data repository is critical to knowing how to develop a connector that will retrieve data from that repository. Therefore, if you are not familiar with the model of the repository that you are trying to access, then you will need to consult with someone knowledgeable about the repository data model.
2. **Defining the Seed** - The seed enables the connector to access information that will be necessary to connect to the data repository (for example: *username*, *password*, *filters*, *port numbers*, and more). The seed is defined using a VXML structure.
3. **Creating a URL structure** - You need to architect a URL scheme that uniquely represents each piece of data extracted from the repository. Each URL will have the following structure:
   a. **Protocol** - Each URL starts with a protocol. This should be brief, but descriptive. Examples include: `imap`, `exchange`, `sharepoint`, `smb` and others. Do not use an already existing protocol like `http` since the behavior might be unexpected.

Note: Remember that a *crawl-URL* is an identifier for the data and not an actual URL that can be accessed.

b. **Host** - We require that external systems run on an IP addressable machine, even if it is a single global host (such as Salesforce). The host is used by the crawler to control/balance the load. If you are accessing the local machine you can use `*localhost*` as your host.

c. **Path and CGI parameters** - The combination of the two should uniquely name any object/item in the system. However, be careful of "namespace" ambiguity. For example: in IMAP, there needs to be a way to distinguish between folders and messages. Thus `imap://host/folder/to/message/the_message` would be bad, as folders could have the same "name" as messages. While, `imap://host/folder/to/message?message=the_message` would be good. In general, the best practice recommendation for naming is to use the URL path for "types". Thus folders for file systems, tables for databases, object types for sharepoint, and others. Then, given a specific "type", a CGI parameter can be used to give the final name or identifier for an item.

Note: Options (name/value pairs) to be attached to URLs are supported. This enables authentication tokens and connection options that enable the crawler to process URLs in an highly parallel and efficient manner.

4. **Building the Core Connector Class** - You will need to create a class that will be the main entry point into the project. Usually we name this class as `<Repository>Connector.java`, where `<Repository>` is the name of the repository you are trying to access (for example: `FilesystemConnector.java`, `SalesForceConnector.java`). This class is responsible for the interaction between the crawler and the connector and it uses the `plugin.xml` file to let the crawler know where it can be found. Example of entry in the `plugin.xml` file:

```
<parameter id="class"
.helloworld.examples.HelloWorldConnector" />
```

This entry in the HelloWorld `plugin.xml` file lets the crawler know that `HelloWorldConnector.java` is the main class for the Hello World Connector.

5. **Injecting the seed values** - There are generally two approaches to pass the values from the crawler to the connector:
   - **Guice** - This is the preferred method to create injection. The Filesystem and LDAP connector provide examples of this method.
   - **Manual** - This is the older method to create injection. The HelloWorld connector provides an example of this method. Although this is the older version, this solution is still important because it eliminates a lot of the complexity that Guice introduces, complexity that might not be necessary, especially with small connectors.
6. **Creating Specialized Nodes** - The specialized nodes define the entities. It describes what data can come from the repository. For example, the specialized node determines if data is a file or a document. These nodes describe the different forms the connector node can take. It can tell you if your data is a one type of entity or another.
7. **Defining the HandlerChainModule** - This class will create all the converter and handler bindings. The purpose of this class is to explain what handler/converter should be called when a particular node is provided and in which order the handlers/converters should be accessed.
8. **Adding the Handler node** - After processing the node, a handler has the option of passing the node to the next handler in the chain, or stopping

further processing if there is not another handler in the chain. When it needs more data from a repository, Handlers may employ one or more Fetchers.

9. **Adding Fetchers** - Fetchers are responsible for communicating with the data repository, directly or potentially through an adapted API. Fetchers will create a Model Object by translating the repository data into a format expected by the handler using them.
10. **Enhancing** - Hardening the connector is the process of adding connector features to your connector to support additional connector functionality. For example, adding Continuous Update mode to your connector enables the connector to detect changes in your index without having to restart the crawler.

Creating The Seed

This section provides an overview of the XML seed structure of a connector and describes the elements that comprise the seed. The connector seed is defined as an XML function with one or more children.

The first element if the function element defined using at least four attributes

1. name: the name of the function; the value is the name of the seed configuration file
2. type: type of the function used for display purpose in the admin interface; the value is `<crawl-seed>`
3. products: defined only for the internal use and its value should always be `<all>`
4. internal: a free text variable used to track the seed version for debug log purpose.

Note: More information can be found at:https://www.ibm.com/support/knowledgecenter/SS8NLW_11.0.1/com.ibm.swg.im.infosphere.dataexpl.engine.schema.doc/r_schema-ref-element-function.html

When the main function element is defined, with all its attributes, you need to start creating the function's children. There are at least two types of xml elements that can make the list of a function's children.

(1) prototype: defines a list of variable declarations which will be passed when calling the function The following example for the HelloWorld connector show how you can define a username and password variable in the Seed VXML file.

```
prototype>
   <declare required="required" name="username" type="string">
       <label name="label">Username</label>
       <description name="description"> A sample username. This will be used to greet you.</descri
   </declare>
   <declare required="required" name="password" type="password">
        <label name="label">Password></label>
        <description name="description> This is not be used for anything.</description>
   </declare>
</prototype>
```

Each `<declare>` element will declare a new variable that can be passed into the connector framework and is defined using three attributes (`name`, `required`, and `type`) and two children (`description` and `label`). The `required` attribute specifies if the value for this variable should be provided when a new seed is created. The `name` attribute is used to identify the variable in the connector framework and the

type attribute specifies how is the value of the variable stored. In the username case is just a simple string whereas in the password case is an encrypted value. Moreover, the <label> element is used to display a name next to the text field associated with the variable and <description> element will show up as a question mark next to the variable with the information provided in this element.

(2) process-xsl: holds any XSL code that need to be processed. Use <xml-to-text> element inside a process-xsl node so that the body of the section is nicely displayed by editors. This section declares all the configuration fields defined in the previous XML node, otherwise they will not be made available.

The section below is an example of such seed element, extracted from the HelloWorld connector.

```
<process-xsl>
   <xml-to-text xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
       <xsl:param name="username"/>
       <xsl:param name="password"/>
       <xsl:template match="/">
           <xsl: variable name="extra-options">
           <options>
```

Note: Repository type is not required to be defined, but is nice to have it.

```
<crawl-extender-option name="repositorytype">HelloWorld</crawl-extender-option>
```

Note: Reads the value provided for the username variable .

```
<crawl-extender-option name="username" value="{$username}</crawl-extender-option>
```

Note: Reads the value provided for the username variable.

```
<crawl-extender-option name="username" value="{$username}</crawl-extender-option>
```

Note: Reads the value provided for the password variable

```
<crawl-extender-option name="password" value="{$password}</crawl-extender-option> </options> </xsl:va
```

Note: The <next> element creates the seed URL using viv:url-build function.

Note: More information about this function can be found here: https://www.ibm.com/support/knowledgecenter/SS8NLW_11.0.1/com.ibm.swg.im.infosphere.dataexpl.engine.man.doc/r_viv_url-build.html

```
          <xsl:variable name="root-url">
          <xsl: value-of select="viv:url-build('hello-java',
          '', '', 'localhost', '-1', '', '')"/>
          </xsl:variable>
           <call-function name="vse-crawler-seed-url-common">
                 <with name="urls" value="{$root-url}" />
                 <with name="extra-curl-options"
                 value="{viv:node-to-str($extra-options)}"/>
          </call-function>
          <crawl-extender>
                  <call-function name="vse-crawler-seed-extender-java-common"
                  no-view-resolved="no-view-resolved">
                  <with name="protocol">hello-java</with>
```

Note: The class name matches the information provided in the plugin.xml file.

```
                <with name="classname">plugin:hellojava.plugin@hello-java</with>
                <with name="dns">dns</with> </call-function>
         </crawl-extender>
         </xml:template>
```

Implementation

Several Classes are required to produce a functional connector. When you build a connector, the `ConnectorNode` is the object that is exchanged between the crawler and the connector. Although not required, each connector will typically include a number of Handlers and Fetchers.

Most of the connector projects can be defined given the following directory tree structure:

- NEW_CONNECTOR_PROJECT
  - GRADLE
  - `build.gradle`
  - `settings.gradle`
  - SRC
    - PLUGIN
    - MAIN
      - NODES
      - RESOURCES
      - JAVA/SCALA CLASSES
        - NODES
        - MODULES
        - FETCHER
        - HANDLER
        - extra Java and Scala classes
  - TEST
    - RESOURCES
    - JAVA/SCALA CLASSES
      - NODES
      - MODULES
      - FETCHER
      - HANDLER
      - Extra Java and Scala classes

At the first level in the connector project tree there are two files used to define the project as a gradle build. First we have `build.gradle` file that is used to define all the dependencies and some basic properties of the project:

- project name
- project version
- project artifact ID

The second file, `settings.gradle`, is another mandatory file for a gradle build, and is used to define the name of the root project.

SRC This folder contains all the source code for the connector project.

PLUGIN
The most important file contained in this folder is plugin.xml. This file is used to define which class is the main class of the connector and which class is used to define the Guice injections.

MAIN Contains all the `JAVA` and `SCALA` classes.

Note: No matter if you are writing a Java connector or a Scala connector, the structure of the project will be the same. Therefore we use the notation `JAVA` or `SCALA` to represent one of the two implementations.

The main source code of the project can be divided into the following categories.

NODES
This folder contains the code that defines the main objects or tokens that are exchanged between connector and crawler. Depending on the connector you build, you might have one or multiple connector nodes.

MODULES
Used to cluster all classes that are used to define the connector modules.

For example, some of the most relevant modules that might be created are the following:

`ConnectionModule.java` Class
Contains all the necessary code to make a connection to the data source.

`ChainModule.java` Class
Used to bind the interfaces to their implementation. This class should extend the `HandlerChainModule` class and should overwrite the `configureHandlerChain()` method.

`DefaultOptionModule.java` Class
Used to define the binding between the connector seed options and the connector code `HandlerModule.java`

FETCHER
Not all connectors will have fetchers. If you want to allow your connector to interact with the data source in multiple stages, to collect more data when the connector node does not contain enough information, then you will define that behavior in one or more fetchers that will be stored in this folder.

HANDLER
Handlers are used to define what should the connector do when a particular data is retrieved from the data source. For example, in the File System Connector, there is a FolderHandler.java and a FileHandler.java file. Each file is used to describe what the connector should do when it reads a Folder and, respectively, a File from the datasource. This class should overwrite the `canProcess()` and `process()` methods.

OTHER JAVA/SCALA CLASSES
In addition to all the clusters defined above, we also need to create at least two other standard classes. First, we have the *`ConnectorName`*`Connector.java` Class which is the entry point in any connector. And second, the *`ConnectorName`*`LifeCycleListener.java` Class. This will create a new Life Cycle Listener with a group of pre-defined things that must be configured before the connector can start. This includes logging, `DefaultOptionModule` and enabling Continuous Update mode.

RESOURCES
This section of the project is used to hold the `build.properties` file used for the gradle build. This file usually has the same form and it contains the following information:

- `artifactId=@project.artifactId@`
- `groupId=@project.group@`
- `version=@project.version@`

- `tag=@GIT_BUILD_TAG@`

`build.properties` is a generic file used by the Gradle in conjunction to Jenkins build tool to configure the build environment variables. If you do not create a Jenkins job, you will not need this file. This file defines values for four variables: `artifactId`, `groupId`, `version` and `tag`. These values are given as parameters for the script and they are replaced with an actual value by the Jenkins job.

- `groupId` will identify your project uniquely across all projects
- `artifactId` is the name of the Jar without version
- `version` if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, ...). Don't use dates as they are usually associated with SNAPSHOT (nightly) builds.
- `* tag` is the tag used by the Jenkins job.

NODES
This folder contains the XML function files necessary to define a crawl seed and the authentication mechanism. The first one is usually defined in a function called: `function.vse-crawler-seed-connector_name.xml`.

TEST Usually, the TEST folder mimics the same structure as the MAIN folder, therefore there will be a test class for each class defined in the main folder.

Converters

This section describes how converters generally fit into the connector development framework, but does not provide guidance on developing custom converters that may be required to crawl a specific repository.

Converters produce content types specific to the connector. In the following example, the converter `type-in` attribute of the function element specifies the content type of the document.

```
<function name="vse-converter-ldap-to-xml" type="converter"
  type-in="application/ldap" products="all" internal="${project.version}"
  type-out="application/vxml-unnormalized" >
    <prototype>
        <label>Ldap Converter</label>
        <description>
            Basic converter for the Ldap connector.
        </description>
    </prototype>
</function>
```

If you are developing a connector for a repository not covered in the Connector SDK examples, you may need a custom converter to identify content types from that repository. The development of custom converters is not described in the Connector SDK documentation.

As in the seed case, a converter is another function. The main function attributes of a converter are the following:

- `name` - The name of the function.
- `type` - The type of the function.
- `type-in` - expected input data.
- `type-out` - expected output data.

Chapter 2. Getting Started

The Connector SDK documentation is designed to enable a Watson Explorer Engine applications developer to build a connector as quickly as possible. For developers that may not be familiar with developing Watson Explorer Engine applications and connectors, this section will serve as an outline of the steps you will need to complete in order to build your own connector. This guide describes how to build a connector through four different connector examples.

The Connector SDK Package

This section describes how to compile the connector examples that are provided as part of the Connector SDK package. The connectors are built using the Gradle automation tool.

Gradle is a build automation tool that builds upon the concepts of Apache Ant and Apache Maven and introduces a Groovy-based domain-specific language (DSL) instead of the more traditional XML form of declaring the project configuration.

Connector SDK Package Overview

The connector package contains a number of sub-directories with all the necessary scripts, JARs, examples, and documentation.

Important: You need to ensure that your JAVA_HOME variable should point to the location of your Java JDK or you will not be able to build the examples in the Connector SDK.

The connector package is delivered as a ZIP archive. The top level of the package contains a README file in Markdown syntax, Gradle build scripts for both UNIX and Microsoft Windows platforms, and scripts for creating symbolic links between the `lib` folder and each example connector. The package contains the following directories:

docs Contains Javadocs associated with each connector plus the Connector SDK documentation.

examples
Contains the Java and Scala version of the Hello World Connector, a Java version of the Hello World Extended Connector, a Java version of the Filesystem Connector, a Java version of the Filesystem with Continuous Update Mode Connector, and a Scala version of the LDAP Connector.

gradle Contains generated files for the Gradle wrapper.

lib Contains the Java libraries that the example connectors require.

Compiling the Connector Examples With Gradle

This section describes how to compile the example connectors that are shipped with the Connector SDK package using the Gradle automation scrips provided as part of the Connector SDK.

About this task

Note: Ensure that your `JAVA_HOME` environment variable correctly points to your JDK installation. Otherwise, you will not be able to build the examples.
To compile the connector examples, do the following:

Procedure

1. Unzip the sdk.zip package.
2. Run `./gradlew projects` to see what packages are available.
3. To create necessary links, run `mkLinks.sh` if you are compiling the connector package on a Unix platform. Alternatively, run `mkLinks.bat` if you are compiling the connector package on a Microsoft Windows platform. This will set up symbolic links for the examples to the lib directory that is contained in the Connector SDK package root.
4. To build all packages, run `./gradlew connectorDistribZip`. To run only a particular project, run `./gradlew $projectName: connectorDistribZip`

   Note: The *$projectName* is the name of the directory that contains the example. For example,you can run HelloWorld project by using `connector-plugin-helloworld-gradle`. You can find this name when you run `./gradlew projects` as described earlier in this procedure.

Results

The connectors are now built. The distribution version of each connector will be available in the `build/distribution` sub-directory of that particular connector's `examples` folder.

Using The Example Connectors

This section describes how to install and use the connector examples that are provided as part of the Connector SDK.

Once you have built the connector examples as described before, you will need to unpack the distribution zip and then install them in Watson Explorer Engine. Installation of the example connectors can be accomplished by using the standard connector installation procedure for all of Watson Explorer Engine connectors. Alternatively, you can manually install the connector examples. Both installation methods are described in the following subsections.

Using the Standard Connector Installation Procedure

About this task

To install a new or updated connector from an archive file (.zip file), do the following:

Procedure

1. Ensure that there is no other version of the same connector still installed.
2. Copy or move the archive file for the connector to the top level of your Watson Explorer Engine installation directory:

   Note: By default, the installation directory is `\Program Files\IBM\WEX\Engine` on Microsoft Windows systems, and `/opt/ibm/WEX/Engine` on Linux systems.
3. Extract the file. All files will be installed into the appropriate locations.

Specifically, the following files will be installed:

- `lib/java/plugins/CONNECTOR-VERSION.zip` - The connector plugin (where *CONNECTOR* is the name of the connector and *VERSION* is the specific version of the connector, such as 1.2.3).
- `data/repository-supplements/function.vse-crawler-seed-CONNECTOR.xml` - The connector's seed component, where *CONNECTOR* is the name of the connector. When the repository is unpacked, Watson Explorer Engine will identify the new file in the `data/repository-supplements` directory and will install it into the product repository.

4. Log in to your Watson Explorer Engine administration tool.
5. Navigate to **Management** > **Installation**. The installation screen displays.
6. In the **Repository** section of the **Installation** screen, click **unpack**. The message `Successfully unpacked repository files` displays when the new repository nodes have been successfully incorporated into the repository.

Results

After completing these steps, the new or updated connector will now be available as a seed when creating or modifying a site collection seed.

Manually Installing The Connector

About this task

To manually install a new or updated connector, perform the following actions:

Procedure

1. Move the `function.vse` crawl-seed XML files for the new connector into the `data/repository-supplements` directory of your Watson Explorer Engine installation directory:

   Note: By default, the installation directory is `\Program Files\IBM\WEX\Engine` on Microsoft Windows systems, and `/opt/ibm/WEX/Engine` on Linux systems.
2. Move the distribution file for the connector into the `lib/java/plugins` directory of your installation directory.
3. Move any JAR files for the connector into the `lib/java` directory in your installation directory. Alternately, they can also be installed into any directory that is located in the `Java CLASSPATH`.
4. To enable the new connector:
   a. Log into your Watson Explorer Engine administration tool.
   b. Navigate to **Management** > **Installation** > **Overview**. The installation screen displays.
   c. In the **Repository** section of the **Installation** screen, click **unpack**. The message `Successfully unpacked repository files` displays when the new repository node(s) have been successfully incorporated into the repository.

Results

After completing these steps, the new or updated connector will now be available as a seed when creating or modifying an existing site collection seed.

Chapter 3. HelloWorld Connector

The HelloWorld connector is a proof of concept connector that you can use to begin connector development. It requires a simple environment and is intended to demonstrate how to get up and running as quickly as possible with a working connector using the Connector SDK.

Note: The Connector SDK includes two version of the HelloWorld Connector, one written in Java and one written in Scala. The implementation for either one is identical and therefore only the Java example is described.

HelloWorld Project Overview

This section provides an overview of the Hello World Project. If you look inside the project package you will see that the project is organized as described in the “Connector Development Workflow” on page 3, "Implementation" section. However, you will not see the same structure inside the `main/java` node because this is a very small connector. This solution is provided to demonstrate a basic way to create a connector. Although, it is an older solution, it is sometimes faster to develop a connector using this solution as a model, especially when your goal is to develop a very simple connector, because the complexity introduced by Guice injection is eliminated.

HelloWorld Project Components

This section describes the components of the Hello World Project.

`HelloWorldConnector.java`

This connector has only one class that extends `ConnectorWorker` and overwrites the `processNode()` method. `ProcessNode()` function is necessary in order to explain how a node should be processed or handled when it is encountered.

The solution provided for this connector uses `VxmlDocBuilder` class, from the Connector Utility package , which provides helpful methods for creating document elements. All necessary characteristics of a document can be added using one of the add() methods defined for a `VxmlDocBuilder` object .

Crawl Seed

The crawl seed is defined in the `vse.function-crawl-seed` XML file, as explained in“Creating The Seed” on page 9. Our proof of concept HelloWorld connector is using a crawl seed that has two required parameters: **username** and **password**. When invoked, this seed will display a form with two text fields: one for the **username** and one for the **password**. Although we are not using the **password**, with this example we want to show how settings can be added into the crawler seed and how they can be passed to the connector. In this example the crawl options are passed to the connector through `getConnectorOption()` function which takes as argument the name of the parameter defined in the crawl seed file.

Hello World - Extended

The Hello World Connector can be enhanced by completing the following exercises. You are encouraged to try the exercises in this section first, before reading the solutions for them.

It is suggested that you enhance the Hello World Connector by completing these exercises in the following sequence:

Exercise 1: Add A Display URL
Add a display URL to the "Hello World!" document. For example: add a URL that when the user clicks on the result title, it will take them to www.ibm.com.

Exercise 2: Change Document Title
Change the title of the document to be "Hello, *your_name*!", where the value of *your_name* is the one specified in the seed configuration.

Exercise 3: Create A New Document
Create a separate document "Good bye, *your_name*!", where the value of *your_name* is the one specified in the seed configuration.

Exercise 4: Add New Content
Add a new content to the document. The name of the content should be `last-modified-time` and the value should be a made-up date.

Exercise 5: Create A Virtual Document
Create a virtual document out of the two documents "Hello, *your_name*!" and "Good bye, *your_name*!" Use the `vse-key` attribute of the two documents to create the virtual document.

Exercise 6: Add A PDF File
Create a separate document and add a PDF file to it. The solution reads the file name and path from the seed.

Exercise 1: Adding A URL

This exercise describes how to add the URL to the Hello World connector example.

About this task

The `VxmlDocBuilder` object contains all the content that defines the document including the display URLs. To add the URL, do the following:

Procedure

1. In the Hello World Example, locate where the `VxmlDocBuilder` object is created.
2. Find where the URLs are added to the document.
3. Replace the value of the URL with www.ibm.com.

Results

When you have completed this exercise, your Hello World connector code should display as follows:

```
return new VxmlDocBuilder().url("http://www.ibm.com")
```

Exercise 2: Changing The Document Title

This exercise describes how to change a document title. As in the previous exercise, we know that all the content of a document is contained in the `VxmlDocBuilder` object and is added using an `add()` method.

About this task

To change the document title contained in the `VxmlDocBuilder` object, do the following:

Procedure

1. In the Hello World Connector example, locate where `VxmlDocBuilder` object is created.
2. Find where the `title` for the document is added.
3. Change the given value with `hello` and the name given as an option in the crawl seed. As explained previously, the option can be retrieved using the `getConnectorOption()` function.

Results

When you have completed this exercise, the part of the code that takes care of the document title should look like this:

```
.addContent("name", "Hello, " +
node.getConnectorOption(HelloWorldConnectorConstants.SEED_OPT_USERNAME)+ "!")
```

Exercise 3: Creating A New Document

This exercise describes how to create a new document and builds upon the previous connector exercise. If you take a close look at the code, you will see that the original document is created in a function called `createPersonalizedHelloWorldDocument(ConnectorNode node)`.

About this task

For this task we need to create a new function, similar to the one mentioned above, that will create another `VxmlDocBuilder` object which will contain the new file. To make a unique new document you need to assign an `ID` (or a `vse-key`) to your document that will be different than the `ID` of the previous document. To create a new document, do the following:

Procedure

1. In the Hello World Example, write a new function that will create a document, same as `createPersonalizedHelloWorldDocument`.
2. Add an attribute called `vse-key` to the new document.
3. Modify the old document to assign a redefined `vse-key`, different than the new one assigned.

Results

When you have completed this exercise, the part of the code that takes care of the creating of a new document should look like this:

```
return new VxmlDocBuilder()
        .url("http://" + HelloWorldConnectorConstants.SEED_OPT_HOST_GOODBYE)
        .setAttribute("vse-key", "2")
        .addContent("title", "Goodbye and nice meeting you!")
```

```
.addContent("name", "Goodbye, Java World!")
.addContent("body", "Goodbye, " + node.getConnectorOption
(HelloWorldConnectorConstants.SEED_OPT_USERNAME) + "!")
.toElement();
```

Exercise 4: Adding New Content

This exercise describes how to add new content to the document. Adding new content is as easy as calling the `addContent()` function on the `VxmlDocBuilder` object. Adding new content is as straightforward as calling `addContent()` function on the `VxmlDocBuilder` object. Since we want to add a time stamp, the name of the content should be `last-modified-time` and the value should be something we make up.

About this task

To add new content to your document, do the following:

Procedure

1. In the Hello World example find the function where the `VxmlDocBuilder` object is created.
2. Apply another `addContent()` function with the first argument `last-modified-date` and the second argument, a *`date/time value`* in any format you prefer.

Results

When you have completed this exercise, the part of the code that takes care of adding a new content should look like this:

```
.addContent("last-modified-time", "07/29/2015")
```

Exercise 5: Create A Virtual Document

This exercise describes how to create a virtual document. A virtual document is a document that is created by joining two other existing documents.

About this task

As we saw in previous exercise, what makes a document unique is the `vse-key` assigned to it. To make a virtual document, you can reuse the same `vse-key` on two different documents. To create a virtual document, do the following:

Procedure

1. In the Hello World example create a variable that will hold the `vse-key` defined for one of the documents.
2. Change the `vse-key` attribute value for the second document to the value you saved in the previous step.
3. Append the new node/document to the connector node object.

Results

When you have completed this exercise, the part of the code that takes care of the creation of the virtual document should look as follows:

```
    public void createVirtualDocument(Element documentToAttach,
    ConnectorNode node, String vse_key){
        documentToAttach.setAttribute("vse-key", vse_key);
        appendDocumentToNode(node, documentToAttach);
}
```

Exercise 6: Add A PDF File In A New Document

This section describes how to add a PDF file as the content of a new document. For this task, the file should be read from a local path given as a crawl seed option and the content of the file should be added as a content of a new document.

About this task

To add a PDF file in a new document, do the following:

Procedure

1. Read the content of a PDF file in an `FileInputStream` object
2. Use `readStream()` method from the `ConnectorData` class to read and append the data from the input stream into the current node.
3. Set the content type of the node to **application/pdf**. For this step, use the `ConnectorData` object used in the previous step and the `setContentType()` method.
4. Make sure you handle any exception that can be thrown and you close the input stream when is not needed anymore.

Results

You have added A PDF file as the content of a new document to the Hello World example.

Chapter 4. Filesystem Connector

The Filesystem connector provides a typical example of a connector that can crawl a filesystem. The connector contains all the code needed to develop a filesystem connector.

Filesystem Project Overview

This section provides an overview of the Filesystem project.

If you look inside the project package you will see that the project is organized as described in “Connector Development Workflow” on page 3, "Implementation" section. The Filesystem project components are comprised of a first level project source folder that contains three main components: `main`, `plugin` and `test`. The `plugin.xml` file defines the following:

1. The location of the connector's main class that extends `ConnectorWorker` class. This class is used to pass information from the crawler to the connector.
2. The location of the connector 's event listener class that extends the `GuiceLifeCycleListener` class.

The `Main` folder contains all the code that defines the connector. This folder contains three separate folders, each with its own purpose:

1. `nodes` - The nodes folder contains the crawler-seed function(s). The first function, and common to all connectors, if `function.vse-crawler-seed-filesystem.xml`. This function defines the main components of the crawler seed. The second function, `function.filesystem-posix-rights.xml` is used in the authentication process.
2. `resources` - This folder contains all the necessary resources used in the connector's code. In this case, the only necessary resource is the `build.properties` file. Please see chapter for more details about `build.properties` file.
3. `java` or `scala` - All Java and Scala code necessary to create the connector will be contained in this folder. For a better organization, the code is divided in multiple folders, that cluster together the classes within same scope. The main class is `FileSystemConnector.java`. It extends a `ConnectorWorker` class and its only purpose is to start the communication between the crawler and the connector.

   The second most important class is the `FilesystemLifeCycleListener.java`. This class is responsible for creating a new Life Cycle Listener with a group of pre-defined elements that must be configured before the connector can start. For this connector, we only use the `FilesystemDefaultOptionModule` and `LogConfigurationStartupModule`. You will probably have these two classes in all your projects because, the first class it is used to bind the crawler seed options with their default values, and the second one is used for the logging messages throughout the code.

Next, we need to define a set of nodes. Each node represents a datasource, an entity which in this case can be a file. The purpose of `FileNode.java` is to create a node of type file from a `ConnectorNode`. This class has an inner converter class that overwrites the `canConvert()` and `convert()` methods.

So far we have the connection to the crawler and the conversion from a connector node to a file node. Next we need to define how this node can be handled. For this, we define two handlers: `FileHandler.java` and `FolderHandler.java`.

Both files have the same purpose but one is used when a file is read whereas the other one is used when a folder is read from the source. The two most important functions of a handler are the `canProcess()` and `process()`. The `canProcess()` function determines if the object received can be process, whereas the `process()` function it actually processes the information.

A handler is defined for only one particular data received from the datasource, therefore the function `canProcess()` will refurn `FALSE` if the respective data is not received. For example, `FileHandler.canProcess()` function will return false if a folder is received. In the case when the node can be processed, the fetcher for that particular node is called, the node is processed and eventually sent to the crawler queue. If everything works accordingly, a `NextAction.STOP_PROCESSING` is returned by this function.

The fetchers are responsible for communicating with the datasource and retrieving streams of data. For this project, we have a `FileFetcher.java` class in which the main function is the `retrieveAndProcessFileContent(Path, StreamConsumer)`. This function will connect to the given path and will read and consume the data.

Finally, the last part of most of the projects is the `modules` folder. For the Filesystem connector this folder contains the `FileSystemChainModule.java` and `FileSystemDefaultOptionModule.java`. `FileSystemChainModule.java` contains all the bindings between classes, from the main node class to the available handlers and fetchers. The Binder class collects configuration information (primarily bindings) which will be used to create an Injector. Guice provides this object to your application's Module implementors so they may each contribute their own bindings and other registrations. `FileSystemDefaultOptionModule.java` contains all necessary bindings. Anything that has a default value in the crawler-seed xml file, should be bind in the `DefaultOptionModule` class.

The `test` folder contains all the unit tests defined for the classes created in the Main folder.

Filesystem Connector with Continuous Update

The Filesystem with Continuous Update mode connector serves as a starting point into the complexity of developing connectors to take advantage of the full capabilities of Continuous Update mode.

Continuous Update mode enables you to detect changes in your repository as they happen without requiring a full recrawl of your repositories. This functionality must be configured and customized specifically for each particular repository. Because Filesystem does not provide information about deleted files, only a partial example of the Continuous Update mode is implemented. However, the partial implementation of Continuous Update mode functionality in this example provides the hooks needed to build a connector with enhanced Continuous Update mode and tailored for your particular data repository.

Filesystem with Continuous Update Project Overview

This section provides an overview of the Filesystem with Continuous Update Project.

If you look inside the project package you will see that the project is organized as described in“Connector Development Workflow” on page 3, "Implementation" section. Moreover, there are only few differences between this connector and the Filesystem Connector and we are going to restrict our discussion to what is new in the FileSystem Continuous Update connector.

As the name suggests, a light form of Continuous Update is introduced in this version of the FileSystem connector. When a connector is configured to operate in Continuous Update mode, new, updated, and deleted data is continuously indexed. As a result, repository updates, document modifications, and other changes can be searched with Application Builder or Watson Explorer Engine applications as quickly as possible. The Watson Explorer Engine crawler should never need to be manually stopped and restarted, unless for maintenance. Because of the way a file system is defined, we cannot detect when a file is deleted therefore the Continuous Update cannot be fully implemented.

Preparing The Filesystem Connector Crawl Seed For Continuous Update

To implement Continuous Update mode you will need to make a series of additions to the crawl-seed file. First, you will need to include in the crawl seed the option for Continuous Update mode. Next, in the `process-xls` section of the crawl-seed file will you need to include additional options.

About this task

This task describes how to prepare the crawl-seed file for enabling Continuous Update mode in the Filesystem connector.

To prepare the crawl seed to enable Continuous Update mode, do the following:

Procedure

1. In the `process-xls` section of the crawl-seed file, add the following XSL elements:

   ```
   <xsl:param name="continuous-update-enabled"/>
   <xsl:param name="continuous-update-delay-in-seconds"/>
   ```

   After adding the XSL elements, the `process-xls` section should look as follows:

   ```
   <!-- Declare all configuration variables here. -->
   <xsl:param name="files"/>
   <xsl:param name="bootstrap-logging-cfg"/>
   <xsl:param name="bootstrap-logging-enabled"/>
   <xsl:param name="logging-config"/>
   <xsl:param name="continuous-update-enabled"/>
   <xsl:param name="continuous-update-delay-in-seconds"/>
   ```

   Along with the original files, bootstrapping, and logging options, you now have two options that are associated with Continuous Update mode.
2. In the crawl extender, you will need to ensure that the crawler always starts the connector and generates an error if the JVM shuts down during a crawl. Therefore, you will need to ensure that your crawl extender is set as follows:

```
<crawl-extender>
   <xsl:if test="$continuous-update-enabled">
       <xsl:attribute name="always-run">always-run</xsl:attribute>
       <!-- Setting the next attribute causes the crawler to generate a fatal error
       if the JVM shuts down. -->
       <xsl:attribute name="continuous-update-mode">
       continuous-update-mode</xsl:attribute>
  </xsl:if>
```

3. In the `crawler-extender-option` section you will need to ensure that the default values for the continuous update options are set as follows:

```
<crawl-extender-option name="continuous-update-enabled"
value="{viv:if-else($continuous-update-enabled, 'true', 'false')}"/>
<crawl-extender-option name="continuous-update-delay-in-seconds"
value="{$continuous-update-delay-in-seconds}"/>
```

Results

The crawler seed is now prepared to support Continuous Update mode. The next section describes the code that you will need to add for the connector to handle continuous updates.

Adding Continuous Update Code To The Filesystem Connector

When the crawler seed is ready, we can add the necessary code to the connector body that will handle continuous updates.

About this task

To add the code to support Continuous Update mode, do the following:

Procedure

1. Add a FileSystem Continuous Update Module. This class will only bind tasks for the continuous update module.
2. In the `FileSystemLifeCycleListener.java file`, we will add the following call: <codeph>FileSystemContinuousUpdateModule</codeph>: <codeph>super(new FileSystemDefaultOptionsModule()</codeph>, <codeph> new LogConfiguringStartupModule()</codeph>, <codeph>new FileSystemContinuousUpdateModule()</codeph>.
3. Create a `FileSystemChange.java` file that will be used to check when a file is changed or deleted. As we mentioned before deleted files cannot be detected in a file system, however we added this incomplete functionality to demonstrate how should this be implemented, when information is available.
4. `Create FileSystemChangeFetcher.java class`. This class determines which files have been modified for one given path (given in the seed configuration) and encapsulates them in `FileSystemChange` object which will be handled by the `FileSystemChangeHandler`.
5. `FileSystemChangeHandler` overrides the *process* method from the Handler class. It creates a new connector node and it enqueues it into the crawler using the type of the change. If you look closely at the code, it will not be executed, since we cannot detect that the file was deleted.

Results

These are the only additions that have to be done to make a connector work in a continuous mode.

Chapter 5. LDAP Connector

The most advanced connector example provided in the Connector SDK is the LDAP connector, which connects to repositories using the Lightweight Directory Active Protocol. (LDAP).

LDAP runs on a layer above the TCP/IP stack. It provides a mechanism used to connect to, search, and modify Internet directories. The LDAP connector is a full scale connector, with all the capabilities that can be found in many other connectors. This connector example is intended to serve as a model and a teaching tool for a developing a full scale connector with the Connector SDK.

LDAP Project Overview

This section provides an overview of the most complex connector example, the LDAP connector. The project has the same structure as described in the “Connector Development Workflow” on page 3, "Implementation" section.

LDAP (Lightweight Directory Access Protocol) is an Internet protocol that email and other programs use to look up information from a server, usually contact information. But LDAP is not limited to contact information, or even information about people. LDAP is used to look up encryption certificates, pointers to printers and other services on a network, and provide single sign-on where one password for a user is shared between many services. LDAP is appropriate for any kind of directory-like information, where fast look ups and less-frequent updates are the norm.

An LDAP directory is a collection of entries, which consist of one or more attributes each. Each attribute has one or more values and a type that determines the kind of information the values can hold and how those values behave during directory operations. The attribute types and object classes defined for LDAP clients are described in RFC 4510.

Additionally, LDAP defines the following:

- Permissions - set by the administrator to allow only certain people to access the LDAP database, and optionally keep certain data private.
- Schema - a way to describe the format and attributes of data in the server. For example: a schema entered in an LDAP server might define a `groovyPerson` entry type, which has attributes of `instantMessageAddress`, and `coffeeRoastPreference`.

LDAP directory service is based on a client-server model. One or more LDAP servers contain the data making up the LDAP directory tree or LDAP back-end database. An LDAP client connects to an LDAP server and asks it a question. The server responds with the answer, or with a pointer to where the client can get more information (typically, another LDAP server).

The entries are arranged hierarchically in a tree that is structured geographically and organizationally. Global entries, such as countries/regions, reside at the top of the tree, followed by state or national organizations, then organizational units, people, devices, or anything else that might be represented in a directory.

A directory entry is represented by its entry name, or relative distinguished name (RDN), and by its distinguished name (DN). The DN uniquely identifies each entry on a global level, and is derived by concatenating the RDN of an entry with the RDN of each of its ancestor entries.

LDAP Project Components

First level of the LDAP project source folder contains three main components: `main`, `plugin` and `test`.

The `plugin.xml` defines the following:

1. The location of the connector's main class that extends `ConnectorWorker` class. This class is used to pass information from the crawler to the connector.
2. The location of the connector 's event listener class that extends `GuiceLifeCycleListener` class.

The `Main` folder contains all the code that defines the connector. This folder contains three separate folders, each with its own purpose:

- `Nodes`
- `Resources`
- `Java` or `Scala` Classes

`Nodes` The nodes folder contains the crawler-seed function(s). First and most important function, `function.vse.crawler-seed-ldap.xml` , is used to define the main components of the crawler seed. First section of the crawler seed is used to declare the name of the seed, under wich it will be displayed in the seed list, in Watson Explorer Engine.

```
<label>LDAP</label>
  <description>
    <p>
      Crawl an LDAP server by indexing a certain search result.
    </p>
   </description>
```

Next we define some necessary attributes used to connect to the LDAP server -- the **hostname** of the LDAP server and the **port** number.

```
<declare name="host" type="string" required="required"> ... </declare>
<declare name="port" type="int" min="0"
max="65535" initial-value="389"> ... </declare>
```

Next section is dedicated to the authentication procedure, and is define between the `<proto-section>` tags.

```
<proto-section section="Authentication"> ... </proto-section>
```

This section defines the necessary attributes that will be used during an LDAP connection session and it includes a username, a password and a method of authentication. Also, as described in the previous paragraph/section, an LDAP connection needs a Distinguished Name (DN) for authentication.

The next `<proto-section>` is dedicated to the search attributes. In this section, the user can define multiple parameters for the search as the scope, the search time limit, search count limit. They can also restrict the search to only a list of attributes or she can completely remove some attributes (for example: blacklist attributes) to speed up the crawl.

The next <proto-section> is dedicated to the connection security. It defines the protocols that can be used by the LDAP server and it allows users to upgrade the unencrypted TCP socket connection to an encrypted SSL connection by enabling the StartTLS attribute <declare name="enable-starttls" type="boolean" initial-value="false">.

Also, LDAP security requires SSL Server Certificate Verification which is defined in the following declaration:

```
<declare name="ssl-cert-policy" type="enum"
initial-value="JVM default" enum-values="Trust all|JVM default"
other-value="SSL fingerprint"> ... </declare>
```

The next few advanced <proto-section> are created to add more attributes for the authentication method selected before. The crawl seed also contains a full section for debugging introduced by the following <proto-section><proto-section section="Advanced - Debugging" toggle-section="toggle-section">.

The next two xml functions are converters.

1. function.vse-converter-ldap.xml – Very basic converter that just outputs application/ldap as application/vxml-unnormalized without doing any conversion.
2. function.vse-converter-ldap-binary.xml – Very basic converter that just outputs application/ldap-binary as application/vxml-unnormalized without doing any real conversion.

Note: When you create a search collection using the LDAP seed, you need to make sure you add the LDAP converter too, otherwise the crawler will return errors.

Resources

This folder contains all the necessary resources used in the connector's code. In this case, the only necessary resource is the build.properties file. Please see chapter"The Connector SDK Package" on page 15 for more details about build.properties file.

Scala Main class for the LDAP connector is LdapConnector.scala. The only scope of this class is to start the communication between the crawler and the connector. The next important class is the LdapLifeCycleListener.scala.

This class is responsible for creating a new **Life Cycle Listener** with a group of pre-defined elements that must be configured before the connector can start. For this connector, we use LdapDefaultOptionsModule, LogCongiguringstartupModule, StdOutStdErrRedirectingStartupModule and LdapConnectionPoolStartupModule.

Next, we need to define what is a node in an LDAP environment. One class is created for each entity that will be referred as a node in the LDAP connector: LdapRootNode.scala , LdapPagingNode.scala and LdapResumeNode.scala.

All three functions override the canConvert() and convert() functions. The first one checks if the data received is of the expected type, whereas the second function converts the ConnectorNode into the necessary Ldap type of node.

As the name suggests, LdapRootNode is used to create a reference to the root node of an LDAP call. This node will contain the main characteristics of an LDAP node. The LdapResumeNode is used when resume function is

activated in Watson Explorer Engine. Finally, the LdapPagingNode during search to cut the result set into pages.

So far we took care of the interaction between the crawler and the connector and the conversion step, when a ConnectorNode is received.

Next we need to define how the data is handled. Since we defined three different Ldap nodes we will also define three different handlers: LdapRootNodeHandler.scala, LdapResumeHandler.scala and LdapPagingNodeHandler.scala.

LdapRootNodeHandler: processes a generic Ldap node by initiating a search given the attributes setting in the crawl seed (for example: search base DN, search filters, search controls) and sets the NextAction to STOP_PROCESSING.

LdapResumeHandler: defines how the data hold in a resume node should be processed. Since the LDAP does not support resume options, the process() method will only throw a ResumeUnsupportedException error. However, we commented out the code that would have been written here if resume would have been possible. Use this as an example of how a resume option can be treated:

```
 override def process(node: LdapResumeNode): NextAction =
{ //In case resume should be interpreted as refresh:
crawler.enqueueURLWithChangeID(node.seedUrl,
NodeUtils.getCurrentTimeAsChangeId) NextAction.STOP_PROCESSING }
```

LdapPagingNodeHandler: processes the data contained in a paging LDAP node. First, it creates a page result control, or an encoded cookie, from the data contained in the LdapPagingNode. Then, this cookie is sent to the processLdapNode() function along with the node information. In the end, the NextAction is set to STOP_PROCESSING.

Due to its particularity, LDAP connector project has an extra folder, CONNECTION, that contains all the classes necessary during the authentication mechanism.

LdapBindManager.scala is the main binding class for the LDAP connector and it creates a binding connection and a binding context for the connector.

Finally, the last part of the project is the MODULES folder. LDAP connector contains five modules:

1. LdapDefaultOptionsModule.scala: contains all necessary bindings. Anything that has a default value in the crawler-seed xml file, should be bind in this class.
2. LdapHandlerChainModule.scala: contains buildings of the converters and handlers for each entity node defined in the nodes folder. This class contains handlers binding for LdapRootNode, LdapPagingNode and LdapResumeNode. It also contains a binding for a ConnectorNode handler in case none of the predefined LDAP entities are found. This binding is to an exception Handler class.
3. LdapConnectionModule.scala: Binds the values given in the crawler seed to the connector framework for the LDAP connection.
4. LdapConnectionPoolStartupModule.scala: Binds the values given in the crawler seed to the connector framework for the LDAP connection pool startup module.
5. PipelineCapacityMonitorModule.scala: Binds the pipeline capacity monitor factory to an instance of a factory.

Chapter 6. Troubleshooting

This section enables you to troubleshoot the connector that you develop with the Connector SDK. It is divided into the follow sections:

Common Pitfalls
This section provides an overview of common development mistakes that you may encounter following along with the procedures of this SDK.

Debugging
This section provides a list of debugging tools and techniques that can help you identify problems that you may be encountering when developing connectors with the Connector SDK.

Common Pitfalls

This section describes common pitfalls that may occur when developing a connector using the Connector SDK. You can use the information here to troubleshoot your connector.

Connector could not be started
Make sure you are running the right seed. Click on the XML tab. Is the *vse-crawler-seed-XXX* the right function? You can also click on the link that represents the `vse-crawler-seed` function and check the function code. Is this the right function? Are you calling the right class for the connector code?

No Error
The description of the error usually says it cannot find different plugins, none of which is your connector's plugin.

- Make sure that Watson Explorer Engine can see the plugin file for your connector. Are the values provided in plugin.xml and vse-crawler-seed.xml matching?
- Do a gradle clean followed by a gradle `ConnectorDistribZip` and redeploy your connector.

The connector could not initialize
The connector could not initialize:

```
java.lang.IllegalArgumentException: extension confluence.plugin@confluence
not available in point
main.plugin@com.vivisimo.connector.ConnectorWorker at
org.java.plugin.registry.xml.ExtensionPointImpl.getAvailableExtension(
ExtensionPointImpl.java:150)
at com.vivisimo.libmisc.PluginBridge.getPlugin(Unknown Source) at
com.vivisimo.connector.ConnectorRunner.setupThreads(Unknown Source)
at com.vivisimo.connector.ConnectorRunner.main(Unknown Source)
```

- Make sure that the connector zip file is in the plugins directory `\lib\java\plugins\`
- Make sure that the connector has the right permissions set. You can check the permissions on the other connector zip files to apply the same for the connector that is failing to initialize.
- Make sure you are running the right seed. Click on the XML tab. Is the *vse-crawler-seed-XXX* the right function.

java.lang.UnsatisfiedLinkError: no misc in java.library.path

```
Exception in thread "main" java.lang.UnsatisfiedLinkError:
no misc in java.library.path at
java.lang.ClassLoader.loadLibrary(ClassLoader.java:1682) at
java.lang.Runtime.loadLibrary0(Runtime.java:822) at
java.lang.System.loadLibrary(System.java:993) at
com.vivisimo.libmisc.Libmisc.<clinit>(Unknown Source) at
com.vivisimo.libmisc.StringUtils.<clinit>(Unknown Source) at
com.vivisimo.libmisc.XMLUtil.stringToNode(Unknown Source) at
com.vivisimo.com.XMLTest.getChildNodesTest(XMLTest.java:80) at
com.vivisimo.com.XMLTest.main(XMLTest.java:56)
```

- On Microsoft Windows systems, make sure that the `\lib` folder contains `misc.dll` and it has the right permissions.
- On Unix/Linux systems, make sure that the `/lib` folder contains the file `libmisc.so` and has the right permissions.

Connection Refused or Port out of Range

```
Error: content-type [vivisimo/crawler-error]
Problem with logon: Connection refused
```

Or

```
Error: content-type [vivisimo/crawler-error]
The connector did not catch an exception:
java.lang.IllegalArgumentException: port out of range:
<port used in configuration>
```

- The port was incorrectly set in the search collection configuration. Ensure that the port in the configuration matches the port listed in the repository information.

FROM keyword not found where expected

```
Error: content-type [vivisimo/crawler-error]
Could not execute full statement: ORA-00923: FROM keyword not found where expected.
```

- Click on the Configuration tab of the Search Collection from which the error is generated, and select the Crawling tab.
- Click on Seed Component of your collection.
- Scroll down to the Free XML section and click Edit.
- Find the with tag where name equals sql.
- Replace any single quotes with double quotes inside of the tag and click OK.
- Test the crawl again.

DNS Server Failure

```
Error: content-type [vivisimo/crawler-error]
DNS error: DNS server failure/
```

The host URL of the data crawl is incorrect or you are not connected to the intranet/internet.

- Double check the host information listed in the Search Collection configuration for a typo.
- Check your network connection.

The connector could not initialize: java.lang.NoClassDefFoundError

```
Error: content-type [vivisimo/crawler-error]
The connector could not initialize:
java.lang.NoClassDefFoundError: com/plumtree/remote/prc/PortalException at
java.lang.Class.getDeclaredConstructors0(Native Method) at
java.lang.Class.privateGetDeclaredConstructors(Unknown Source) at
```

```
java.lang.Class.getConstructor0(Unknown Source) at
java.lang.Class.getConstructor(Unknown Source) at
com.vivisimo.connector.ConnectorRunner.setupThreads(Unknown Source) at
com.vivisimo.connector.ConnectorRunner.main(Unknown Source)
```

The path to the JARs directory is incorrect.

- Double check the JARs directory path.

com.google.inject.CreationException: Guice creation errors

While locating `java.lang.String` annotated with

```
@com.ibm.dataexplorer.connector.extensions.inject.ConnectorOption
for parameter 0 at
com.ibm.dataexplorer.connector.extensions.inject.
LogConfiguringStartupModule$LogConfigurator.&lt;init>
(LogConfiguringStartupModule.java:188) at
com.ibm.dataexplorer.connector.extensions
.inject.LogConfiguringStartupModule.configureBeforeCrawlStarts
(LogConfiguringStartupModule.java:175)
```

When you write tests function, first thing you do is to create a test Crawler. This error says that an option is missing.

- The error is related with the `LogConfiguration` class
- To fix it, you need to add the missing option. In this case the following option was missing:
  ```
  connector_option(Constants.CollectionSettings.LOGGING_CONFIG,
                                        "<root />")
  ```

Debugging

This section describes connector debugging techniques and methods that can help resolve common connector problems.

Validate user and password

Often simple connector issues can be attributed to basic account permission errors. Confirm that you are using an account with the appropriate permissions to crawl your repository and that you are using the right password for it.

Check the documentation

Be sure that you have correctly configured all installed Watson Explorer components and your connector, and that there are no missing steps, or incorrect configuration settings, which might be causing a problem in using the connector.

Tip: Rights functions for user collections are common connector pitfalls.

Eliminate resource-side errors

It is a good tactical step to "assume" the issue is with a Watson Explorer Engine connector but, at the same time, to make the administrator of the resource that you are crawling aware of any problems crawling that resource. The administrator may be aware of the issue and have a patch available. It never hurts to check.

Test multi-threaded versus single-threaded

To determine if a connector issue is related to multithreading, set the thread count to 1 and then test a new crawl. If an error is encountered, multithreading is not the source of the problem. Setting the thread count to 1 also has the benefit of making the log easier to read.

Enable bootstrap logging

If a connector is not starting at all, enable bootstrap logging to determine where the failure occurs when the connector is initiated. Bootstrap logging can be enabled in the Watson Explorer Engine administration tool's seed configuration screen.

To activate bootstrap logging do the following:

1. From the seed configuration page of your site collection, go to **Configuration** > **Crawling**. The crawling configuration page displays.
2. In the **Seeds** section, click **edit** and expand the **Advanced - Logging** collapsible menu.
3. Check the **enable connector bootstrap logging** box. Additionally, enter Log4j settings in the **Connector Logging Configuration** text box.
4. Click **OK**.

Enable connector logging

If **Bootstrap Logging** is not available, you can enable a logging condition. To add a logging condition to the connector seed, do the following:

1. In the Watson Explorer Engine administration tool, select **Add A New Condition** from the **Configuration** > **Crawling** > **Conditional Settings** section.

   A pop-up window displays with a list of new conditions.
2. Scroll down and select **connector logging**.

Your goal is to capture a stack trace, which can help pinpoint what might be causing your connector problems.

Enable Log4J logging levels

Log4j enables you to activate different levels of logging without modifying the application binary thus avoiding a heavy performance cost. Logging behavior can be controlled by editing a Log4J configuration file.

Key logging levels that can be applied using the Log4j utility are the following:

- `OFF` - The `OFF` level has the highest possible rank and is intended to turn off logging.
- `FATAL` - The `FATAL` level designates very severe error events that will presumably lead the application to abort.
- `ERROR` - The `ERROR` level designates error events that might still allow the application to continue running.
- `WARN` - The `WARN` level designates potentially harmful situations.
- `INFO` - The `INFO` level designates informational messages that highlight the progress of the application at coarse-grained level.
- `DEBUG` - The `DEBUG` Level designates fine-grained informational events that are most useful to debug an application.
- `TRACE` - The `TRACE` Level designates finer-grained informational events than the DEBUG
- `ALL` - The `ALL` has the lowest possible rank and is intended to turn on all logging.

For more detailed information about Log4j and its configuration, see the online resources for Log4j.

Enable Oakland HTTP wire logging

Enabling logging for wire-level activity is useful for Watson Explorer Engine connectors that use HTTP connections. This is because the wire log

records all data transmitted to and from your server(s) when executing HTTP requests. The wire log uses the `org.apache.http.wire` logging category, which should only be enabled to debug problems. Be aware that wire logging will produce a large amount of log data.

Check for missing JAR files
Be sure that you have all the JAR files needed. If the connector was installed correctly, the necessary JAR files should have been copied to the right location by default.

Open JMX port to profile resources
Java Management Extensions (JMX) supply tools for managing and monitoring applications, system objects, devices and service oriented networks.

To enable the JMX agent and configure its operation, you must set certain system properties when you start the Java virtual machine (JVM). For detailed instruction, consult help resources for using JMX and other JMX compliant tools.

Packet trace with Wireshark
If you are familiar with Wireshark and its advanced packet trace capabilities, it can be used instead of, or to augment, any packet tracing capabilities in the connector that you are using. Consult your Wireshark help resources for using the more powerful features of Wireshark tracing.

Profile resources
Use common performance testing methods to determine how fast the connector performs under a particular workload. Profiling the resources used under various work loads serve to pinpoint bugs relating to scalability, reliability, and resource usage.

Replicate in development environment
Replicate the production environment issue in your development environment and test for the same bug.

Reproduce without connector
Another simple test to determine if the connector is the source of the error, is to attempt to probe the remote resource without it. If you are unable to contact the remote resource without the connector, there may be a problem with your environment rather than with the connector. Common tools used to help in this regard include the following:

- Curl is a command line tool for sending and receiving files using URL syntax. Since Curl is used by many Watson Explorer Engine connectors, it is a great tool to help pinpoint the source of problems when crawling associated resource sites.
- Check that your problems are not browser specific. To do so, attempt to display search results in modern browsers such as Firefox, Internet Explorer, Chrome, and Safari. Test in the browser versions that are relevant to your users.
- Ping and Traceroute can be used to send packets of information to the remote data resource for the purpose of retrieving information, which can useful for testing your internet connection. Consult your operating system documentation on how to locate and execute the ping and traceroute utilities that are available in your environment.

Adjust crawler delay

In **Global Settings** > **Crawler Aggressiveness**, set the **Delay value** to 1. This will increase requests on your server to help identify potential problems.

Note: We do not recommend setting the delay to `0`. Doing so can cause excessive resource usage on your crawling server, repository server, or both.

Validate web services

Check that all web services are performing correctly and that all the needed web services are activated in the server(s) where the data you are crawling is hosted. You can use a Web test to test Web services. Check online resources for writing specific web tests based on your environment.

Remote Debugging with Eclipse

About this task

Using Eclipse you can remotely debug connectors that are running on a Watson Explorer Engine instance. The procedure to do so is the following:

Procedure

1. On the server side, set the **debug-port** value in the collection definition, and set the value to `y` in function **vse-crawler-seed-extender-java-common**.
   a. Select the collection's seed component.
   b. Select the XML view of the seed component.
   c. Under the **crawl-extender** change the **debug_port** value at the bottom of the XML file, by adding the port number you selected. For example, if the port number is 5005 the line should be: `<with name="debug-port">5005</with>`.
2. Access the XML view of the function **vse-crawler-seed-extender-java-common**.
3. Search for the `test="$debug-port > -1` condition and set suspend the suspend attribute of this XML element to `y`.

   Note: Set suspend to `y` if the target VM is to be suspended immediately before the main class is loaded. Otherwise, set it to `n`.
4. In Eclipse, open **Debug Configurations** window, right click on **Remote Java Application** and select **New**.
5. On the right panel give a name for the current remote debug settings, and select a project that will be debugged.
6. Keep the connection type Standard (Socket attached) and add the remote host and the port number that was set in the collection's seed.

   Note: Be sure that the debug port number is not used by another process and is the same value set in the seed file as described in step 1.
7. In the **Source** tab add the project or projects whose source might be debugged.
8. Click **Apply** to accept the settings then start a debugging session by clicking **Debug**.

   Note: If suspend is set to `n`, and because the remote connector JVM could have a very short life time, clicking **Debug** does not guarantee a remote debugging session. Make sure that you have set a break point in the code and you have started the debug session after the remote connector started and before it ended.

Results

You have remotely debugged a connector using Eclipse.