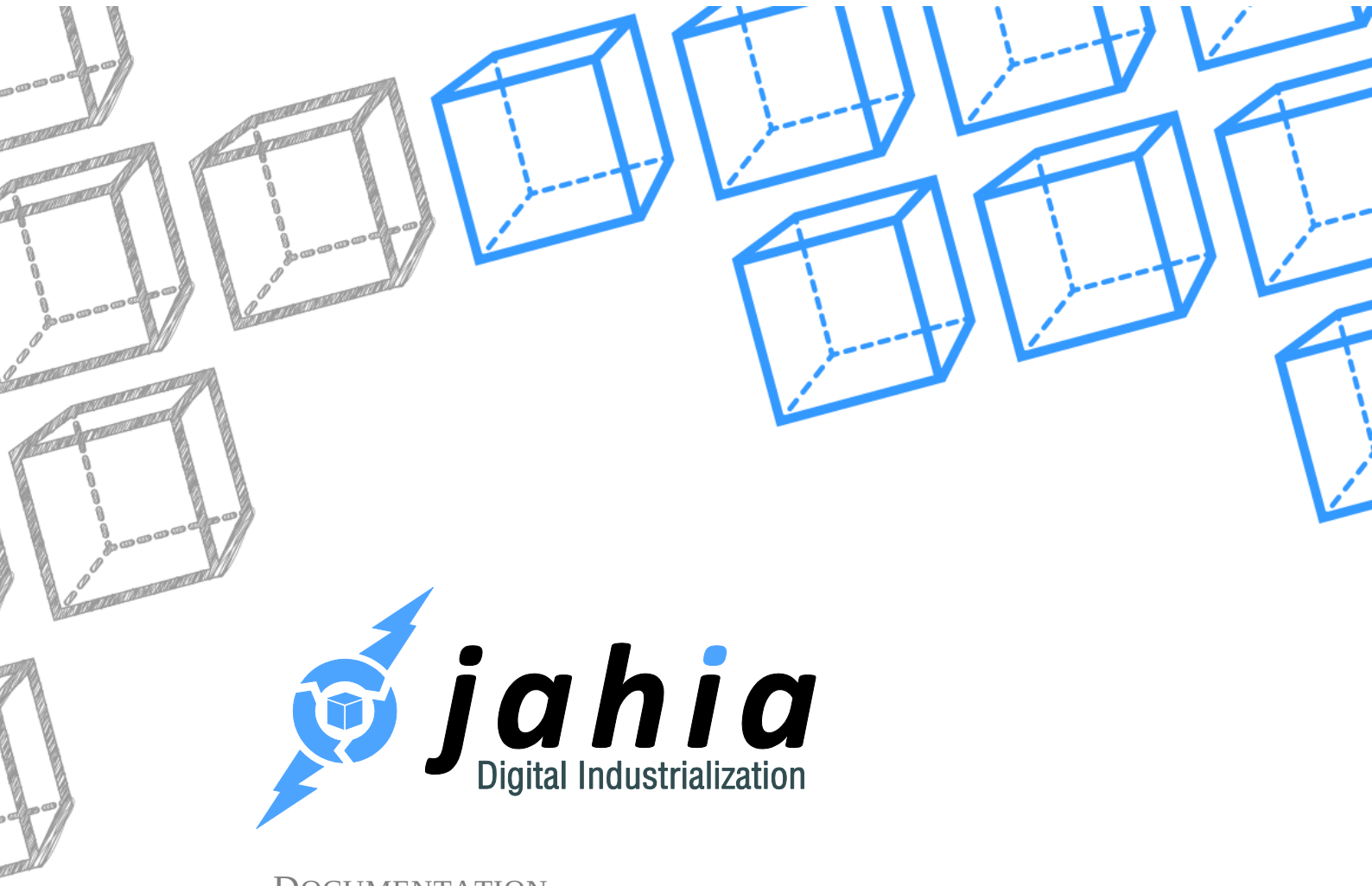


jahia
Digital Industrialization

DOCUMENTATION

Portal Factory 1.0 - External Data Provider Documentation

Rooted in Open Source CMS, Jahia's Digital Industrialization paradigm is about streamlining Enterprise digital projects across channels to truly control time-to-market and TCO, project after project.

Jahia Solutions Group SA

9 route des Jeunes,
CH-1227 Les acacias
Geneva, Switzerland

http://www.jahia.com

Summary

1 Introduction

External Data Provider is a module which provides a new API, allowing the integration of external system as content providers like the JCR.

Integration is done by implementing an External Data Source. The data source just need to focus in the connection to the external system and the data retrieval, when the external data provider do all the mapping work which will make the external content appear in the JCR tree.

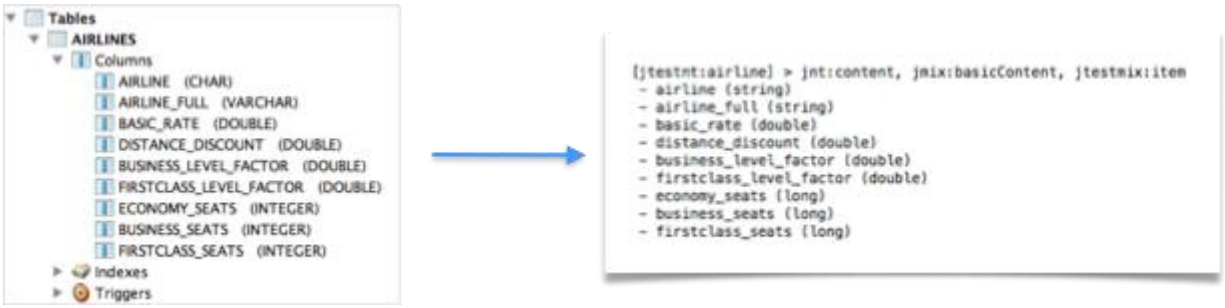
All data source must provide content (reading). They can provide search capabilities, provide write access to create/update content. They can also be enhanceable - meaning that the ‘raw’ content they provide can be enhanced by Digital Factory content (such as for example being able to add comments to an object provided by an External Data Provider).

2 How it works

2.1 Specify your mapping

Your external content has to be mapped as nodes inside Digital Factory so they can be used by Digital Factory as regular nodes (edit/copy/paste/reference etc.). This means that your external provider module must provide a definition cnd file for each type of content you plan to map into Digital Factory.

As a simple example, you can map a database table to a nodetype, defining each column as a JCR property :



Then, you have to define a tree structure for your objects. As they will appear in the Digital Factory repository, you'll have to decide for each entry what will be its parent and children.

It is very important that each node have a unique path - you must be able to find an object from a path, and also to know the path from an object. The node returned by a path must always be the same, and not depend on contextual information. If your nodes depends on the context (for example, the current user), you'll need to have different paths. In order to correctly create a node hierarchy, it's perfectly allowed to add some "virtual nodes" which will act container to organize your data.

Optionally, you can define a unique identifier for every node. The External Data Provider will map this identifier to a JCR compatible UUID if needed, so that it can be used in Digital Factory as any other node.

2.2 Declaring your Data Source

Those external data are accessed through a JCR provider declared in Spring, where you will set some information like provider key, the mount point, the data source implementation, ...

```
    <bean id="TMDBProvider"
class="org.jahia.modules.external.ExternalContentStoreProvider"
parent="AbstractJCRStoreProvider" >
        <property name="key" value="TMDBProvider"/>
        <property name="mountPoint"
value="/sites/movies/contents/tmdb"/>
```

```
        <property name="externalProviderInitializerService"
ref="ExternalProviderInitializerService"/>
        <property name="extendableTypes">
            <list>
                <value>nt:base</value>
            </list>
        </property>
        <property name="dataSource" ref="TMDBDataSource"/>
    </bean>

    <bean name="TMDBDataSource"
class="org.jahia.modules.tmdbprovider.TMDBDataSource" init-
method="start">
        <property name="cacheProvider" ref="ehCacheProvider"/>
        <property name="apiKeyValue"
value="${com.jahia.tmdb.apiKeyValue}"/>
    </bean>
```

This provider then access the underlying data source (implementing ExternalDataSource and other optional interface if needed to read,save the data).

Your implementation of ExternalDataSource must also list the node types you are handling so that Digital Factory knows which node types this data source is able to handle. This can be done programmatically or inside your spring file, here an example of declarative nodeType support from the ModuleDataSource.

```
<bean id="ModulesDataSourcePrototype"
class="org.jahia.modules.external.modules.ModulesDataSource"
          scope="prototype">
        <property name="supportedNodeTypes">
            <set>
                <value>jnt:cssFolder</value>
                <value>jnt:cssFile</value>
                <value>jnt:javascriptFolder</value>
                <value>jnt:javascriptFile</value>
            </set>
        </property>
    </bean>
```

3 Implementation

3.1 Providing/Reading Content

The main point to define a new provider is to implement the ExternalDataSource interface provided by the external-provider module (org.jahia.modules.external.ExternalDataSource).

This interface require from you to implement 7 methods to be able to mount/browse your data as if they were part of the Digital Factory Content tree.

Here the listing of those methods :

- `getItemByPath`
- `getChildren`
- `getItemByIdentifier`
- `itemExists`
- `getSupportedNodeTypes`
- `isSupportsUuid`
- `isSupportsHierarchicalIdentifiers`

The first method, getItemByPath(), is the entry point for the external data. It has to return an ExternalData node for all valid paths - including the root path (“/”). ExternalData is a simple java object that represent an entry of your external data. It contains the id, path, node types and the properties encoded as string (or Binary objects for binaries properties).

The getChildren method also need to be implemented for all valid paths - it has to return the names of all sub nodes, as you want them to appear in the Digital Factory repository. For example, if you map a table or the result of a SQL query then this is the method that will return all the results. Note that it is not required that all valid nodes are listed here. If they don’t appear here, you won’t see them in the repository tree, but you may still be able to access them directly by path or by doing a search. This is especially useful if you have thousands of nodes at the same level.

These two methods reflects the hierarchy you will give to Digital Factory.

The getItemByIdentifier() method return the same ExternalData node, but based on the internal identifier you want to use.

The getSupportedNodeTypes() method simply return the list of node types that your data source may contains.

isSupportsUuid() tells the External Data Provider that your external data have identifier in the UUID format. This prevent Digital Factory to create its own identifiers and maintain a mapping between its uuids and your identifiers. In most of the cases, return false.

isSupportsHierarchicalIdentifiers() tells if your identifier actually looks like the path of the node, and allows the provider to optimize some operation like the move - where your identifier will be “updated”. This is for example useful in a file system provider, if you want to use the path of the file as its identifier.

itemExists() simply tests if the item at the give path exists.

3.2 Identifier Mapping

Every time we read an external node for the first time we generate a unique identifier for it inside Digital Factory. Those mapped identifier are stored inside a table called ‘jahia_external_mapping’.

This table map the internal id to a pair of provider key and the external id returned by ExternalData.getIdentifier method.

3.3 ExternalData

The External Data Source is responsible for mapping its data content into ExternalData object. ExternalData provides access to the properties of your content, those properties has to be converted to one of two type String or Binary. String can be internationalized or not, as they are declared in the cnd file.

3.4 Lazy Loading

If your External provider is accessing expansive data (performance/memory wise) to read then you can implement the ExternalDataSource.LazyProperty interface and fill the lazyProperties,

lazyI18nProperties and lazyBinaryProperties sets inside ExternalData. If somebody tries to get a property which is not the properties map in ExternalData, but which is in one of thoses sets, the system will call one of these methods to get the values :

- `getBinaryPropertyValues`
- `getI18nPropertyValues`
- `getPropertyValues`

For example the ModuleDataSource retrieve the source code as a LazyProperties so this way the source code will be read from the disk only when displayed not when you display the file inside the tree for exploration.

You have to decide which type of loading you want to implement, for example on a DB it must be more interesting to read all the data at once (if not binaries) depending on the number of rows and columns.

3.5 Searching Content

This capability will require you to implement ExternalDataSource.Searchable interface which define only one method :

- search(ExternalQuery query)

Where query is an ExternalQuery, more information here :

http://jackrabbit.apache.org/api/1.4/org/apache/jackrabbit/core/query/jsr283/qom/QueryObjectModel.html

Your method should be able to handle a list of constraint from the query (AND, OR, NOT, DESCENDANTNODE, CHILDNODE, etc.)

You do not have to handle everything if it does not makes sense in your case.

The QueryHelper class provide some helpful methods to parse the constraints :

- getNodeType

- getRootPath
- includeSubChild
- getSimpleAndConstraints
- getSimpleOrConstraints

The getSimpleAndConstraints method will return you a map of the properties and their expected values from the ‘AND’ constraints in the query.

The getSimpleOrConstraints method will return you a map of the properties and their expected values from the ‘OR’ constraints in the query

From the constraints you build a query that means something for your external provider (for example if it is an SQL DB, map those constraints as ‘AND’ constraint in the ‘WHERE’ clause of your request).

Query are expressed using the JCR SQL-2 language definition.

3.6 Enhancing/Merging External Content with Digital Factory Content

Digital Factory allows you to extend your external data content with some of its own mixins or to override some properties of your nodes from Digital Factory. This allow in your definition to mix for example external data and data defined in Digital Factory.

In your Spring file you can declare two things, which of your nodes are extensible by additional mixin and properties, and which properties from your definition can be overridden/merge. Here how you do that :

```
<property name="extendableTypes">
      <list>
            <value>nt:base</value>
      </list>
</property>
```

This is saying that all your types are extendable, but you can limit that to only certain nodes by listing their definitions. Any mixin can be added on nodes that are extendable.

```
<property name="overridableItems">
      <list>
            <value>jtestnt:directory.*</value>
            <value>jtestnt:airline.firstclass_seats</value>
      </list>
</property>
```

This one is saying that all properties from `jtestnt:directory` can be overridden inside Digital Factory. The next one is saying that only the property firstclass_seats from airline definition can be overridden.

On regular usage those nodes will only be available to end users/editors if the external provider is mounted. If you unmount your external provider those data will only be accessible from Jahia tools for administrative purpose.

As all content coming from the external provider, these content are not subject to publication. Any extension will be visible in both default and live workspace immediately.

3.7 Writing/Updating Content

The external provider can be writeable, this means that you will be able to create new content, or update existing one from within Digital Factory.

This capability will require you to implement ExternalDataSource.Writable interface which define 4 methods :

- `move`
- `order`
- `removeItemByPath`
- `saveItem`

Your provider should at least implement saveItem. saveItem will receive ExternalData with all modified properties. Note that if you are using lazy properties, modified properties will be moved

from the set of lazy properties to the map of properties. Removed properties will be removed from both properties map and lazy properties set.

If content can be deleted then you should implement removeItemsByPath.

The other two methods (move and order) are optional behavior, that need to be implemented only if your provider support them (for example the VFSDataSource does not implement order as files are not ordered on a filesystem but moving is implemented).

Here is an example of how to access binary data from ExternalDataSource and save them into the filesystem using VFS API(example from the VFSDataSource).

```
public void saveItem(ExternalData data) throws RepositoryException {
        try {
            ExtendedNodeType nodeType =
NodeTypeRegistry.getInstance().getNodeType(data.getType());
            if (nodeType.isNodeType(Constants.NT_RESOURCE)) {
                OutputStream outputStream = null;
                try {
                    final Binary[] binaries =
data.getBinaryProperties().get(Constants.JCR_DATA);
                    if (binaries.length > 0) {
                        outputStream =
getFile(data.getPath().substring(0, data.getPath().indexOf("/" +
Constants.JCR_CONTENT))).getContent().getOutputStream();
                        for (Binary binary : binaries) {
                            InputStream stream = null;
                            try {
                                stream = binary.getStream();
                                IOUtils.copy(stream, outputStream);
                            } finally {
                                IOUtils.closeQuietly(stream);
                                binary.dispose();
                            }
                        }
                    }
                } catch (IOException e) {
                    throw new PathNotFoundException("I/O on file : " +
data.getPath(),e);
                } catch (RepositoryException e) {
                    throw new PathNotFoundException("unable to get
outputStream of : " + data.getPath(),e);
                } finally {
```

```
                    IOUtils.closeQuietly(outputStream);
                }
            } else if (nodeType.isNodeType("jnt:folder")) {
                try {
                    getFile(data.getPath()).createFolder();
                } catch (FileSystemException e) {
                    throw new PathNotFoundException(e);
                }
            }
        } catch (NoSuchNodeTypeException e) {
            throw new PathNotFoundException(e);
        }
    }
```

3.8 Provider factories

It is possible to create a configurable external data source that will be mounted and unmounted on demand by the server administrator. Instead of declaring a mount point in the spring declaration, you can add a bean implementing the ProviderFactory interface, which will be reponsible of mounting the provider.

The factory need to be associated with a node type which inherits from jnt:mountPoint, and that will define all required properties to correctly initialize the Data Source. Thenthe moutProvider method will instantiate the External Data Provider instance based on a prototype, and initialize the Data Source. Here's the code the definition of a mount point from the VFS Provider :

```
[jnt:vfsMountPoint] > jnt:mountPoint
 - j:rootPath (string) nofulltext
```

And the associated code, which create the provider by mount the VFS url passed in j:rootPath :

```
    public JCRStoreProvider mountProvider(JCRNodeWrapper mountPoint)
throws RepositoryException {
        ExternalContentStoreProvider provider =
(ExternalContentStoreProvider)
SpringContextSingleton.getBean("ExternalStoreProviderPrototype");
```

```
        provider.setKey(mountPoint.getIdentifier());
        provider.setMountPoint(mountPoint.getPath());

        VFSDataSource dataSource = new VFSDataSource();

dataSource.setRoot(mountPoint.getProperty("j:rootPath").getString());
        provider.setDataSource(dataSource);
        provider.setDynamicallyMounted(true);
        provider.setSessionFactory(JCRSessionFactory.getInstance());
        try {
            provider.start();
        } catch (JahiaInitializationException e) {
            throw new RepositoryException(e);
        }
        return provider;

    }
```

Once the provider factory is declared, the “mount” button in document manager will display the new node type, allowing the administrator to create a new mount point with this Data Source.

Jahia Solutions Group SA

9 route des Jeunes,
CH-1227 Les acacias
Geneva, Switzerland

http://www.jahia.com