EXT: Connector Services

Extension Key: svconnector
Language: en
Keywords: forDevelopers, forIntermediates

Table of Contents

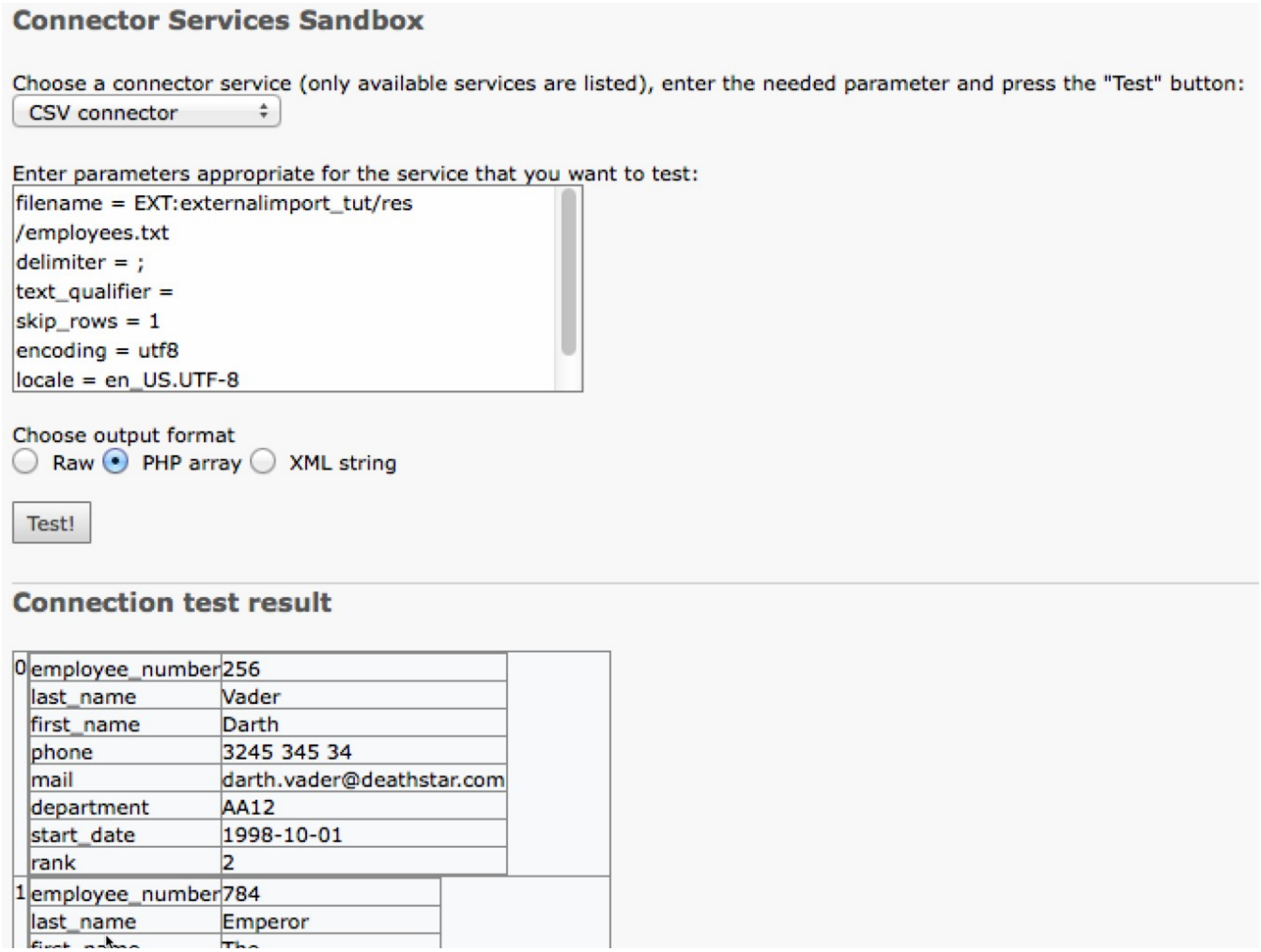
Introduction

The main idea of Connector services is to have a basic framework for developing scripts that connect to third-party applications and retrieve data from them. The new type for these services is "connector" and every third-party application is represented by a distinct sub-type.

This structure makes connection scripts reusable. It also makes them easier to use thanks to a general API. Connector services can be called both from the fronted and the backend. An example of extension using connector services is "external_import".

Screenshots

A view of the BE module which makes it possible to test connections and the data they return.



Questions and support

If you have any questions about this extension, please ask them in the TYPO3 English mailing list, so that others can benefit from the answers. Please use the bug tracker on forge.typo3.org to report problem or suggest features (http://forge.typo3.org/projects/extension-svconnector/issues).

Keeping the developer happy

It's unfortunately not possible to rate extensions anymore (maybe that will change again), so feel free to send thanks to the developers or praise the extension in general. If you really want to give something back, you may consider my Amazon wish list: http://www.amazon.co.uk/registry/wishlist/G7DI2AN99Y4F

You may also take a step back and reflect about the beauty of sharing. Think about how much you are benefiting and how much yourself is giving back to the community.

Installation

This extension does nothing all by itself. Connectors must be developed for specific third-party applications. However this extension must be installed since it provides the base class from which all connector services inherit.

The Connector services family rely on **TYPO3 4.5 or above**.

Updating to 1.1.0

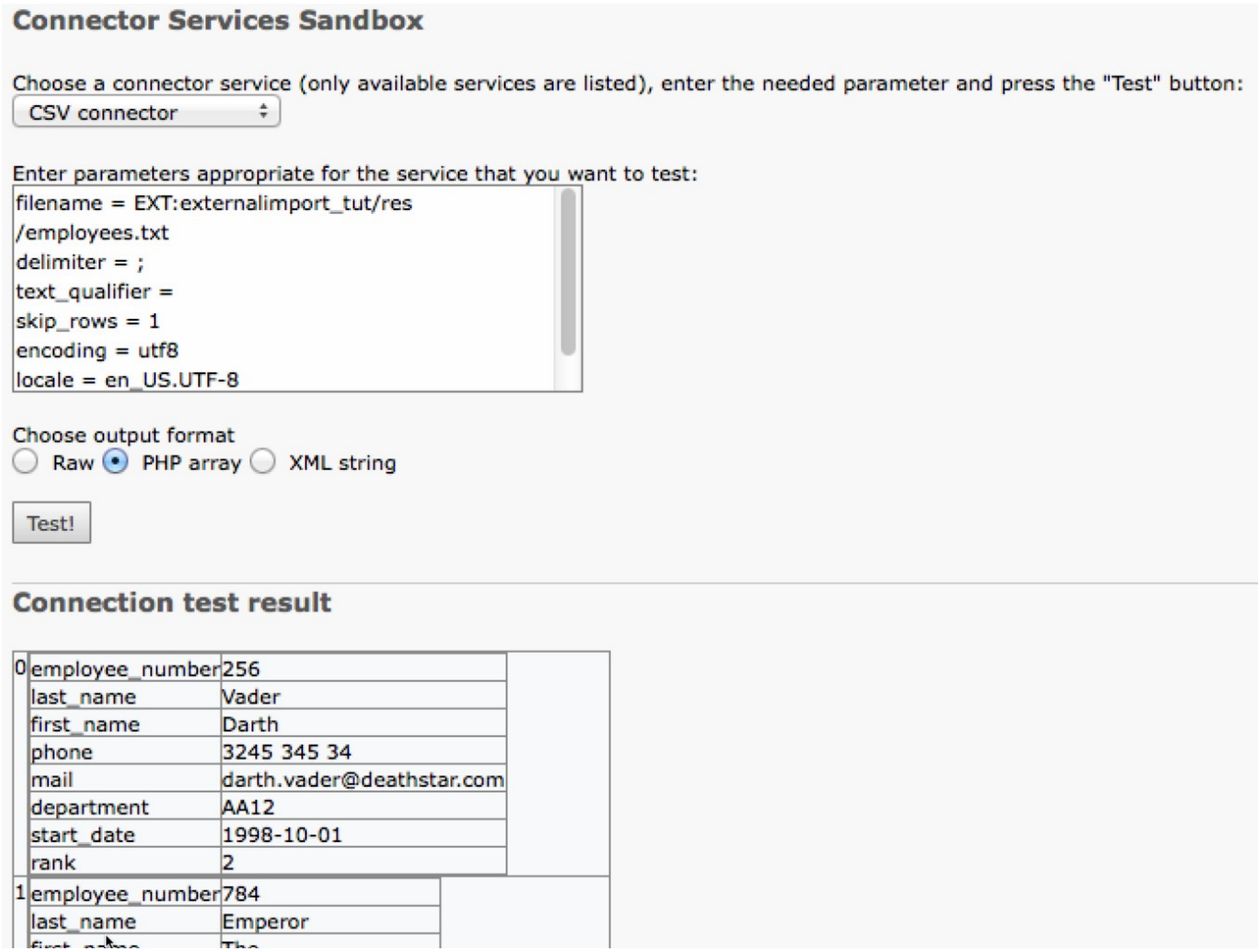
In version 1.1.0 the "sv1" class was moved to the root of the extension and renamed "sv_connector_base" as this made much more sense. The backward compatibility was ensured by keeping the "sv1" class, which is now just an empty wrapper for the base class.

If you designed an extension which extended the "sv1" class, it would be good to change it to extend the "base" class instead, although this is not necessary.

Configuration

The configuration is up to each specific connector according to its needs.

A BE module is provided to test connections.



The steps to use this tool are simple:

1. Choose a particular service from the drop-down menu. Only available services are in the list. If some services are unavailable, a warning will be displayed at the top of the screen.
2. Enter all the necessary parameters in the text field. For each parameter, enter its name, an equal (=) sign and its value (as in the screenshot above).
3. Choose the output format ("raw" will return the native format for the type of resource being connected to).
4. Click on the "Test!" button. If the connection is successful, the data fetched by the connector service will be displayed below the form. If some error happens, a message will be displayed at the top of the page.

Developer's manual

The main interest of the connector services is to manage connections to remote systems and provide a common API for extensions needing such connections. This chapter describes said API and gives some indications about how to implement your own connector services.

Connector API

The table below describes the methods that make up the connector API. These are the methods that you must absolutely implement in your own connector services if you want to make them usable by extensions that rely on such services.

Method	Input	Description	Output
init()	none	This method is called when the connector is instantiated by the TYPO3 service API. It is expected to return a boolean value: true if the distant source is available, false otherwise.	boolean
query()	array of parameters	Strictly speaking this method is not part of the API, since it is protected and thus not designed to be called directly. It is designed to encapsulate the distant source querying mechanism, so it is good programming practice to use it.	mixed (result from the distant source)
fetchRaw()	array of parameters	This method is expected to return the result of the query to the distant source as is, without transformation.	mixed (result from the distant source)
fetchXML()	array of parameters	This method is expected to return the result of the query to the distant source transformed into a XML structure (as a string).	string (XML structure)
fetchArray()	array of parameters	This method is expected to return the result of the query to the distant source transformed into a PHP array.	array
postProcessOperations()	array of parameters and a status	This method is designed to be called back by whatever process called the connector in the first place when said process is done. It receives as argument the usual list of parameters, plus some variable indicating the status of the process (typically this could be an indication of success or failure). It doesn't do anything by itself, but just calls hooks.	void

Implementing a connector service

The first step is to create a class derived from the base connector service and implement all the methods described in the API above. Your class declaration should look something like:

```
class tx_svconnectorspecial_sv1 extends tx_svconnector_base {
}
```

after having included the base connector class:

```
require_once(t3lib_extMgm::extPath('svconnector', 'class.tx_svconnector_base.php'));
```

You must then register your service with the TYPO3 service API. This goes into your extension's ext_localconf.php file and will look like that:

```
t3lib_extMgm::addService($_EXTKEY,  'connector' /* sv type */,  'tx_svconnectorspecial_sv1' /* sv key
*/,
            array(
                  'title' => 'Special Connector',
                  'description' => 'Connect to a special server',
                  'subtype' => 'special',
                  'available' => TRUE,
                  'priority' => 50,
                  'quality' => 50,
                  'os' => '',
                  'exec' => '',
                  'classFile' =>
t3lib_extMgm::extPath($_EXTKEY).'sv1/class.tx_svconnectorspecial_sv1.php',
                  'className' => 'tx_svconnectorspecial_sv1',
            )
      );
```

Initializing the service

Every service must have an init() method which tells TYPO3 whether the service is available or not. This is where you should test the connection to whatever remote application you are connecting to. If that test connection fails, the init() method should return false. Otherwise it should return true.

If your connector has no risk of failing (for example, because it operates locally and is not dependent on anything special), you must still implement the init() method and have it return true all the time.

The init() method of the base connector class (tx_svconnector_sv1) also takes care of providing a proper language object (useful e.g. for localizing error messages) and of reading its own configuration. In your init() method, you should take care of not overwriting this configuration, but instead merge it with the base class' configuration. The code should look something like:

```
protected function init() {
	parent::init();
	$localConfiguration = unserialize($GLOBALS['TYPO3_CONF_VARS']['EXT']['extConf'][$this->extKey]);
	$this->extConfiguration = array_merge($this->extConfiguration, $localConfiguration);
}
```

Obviously you can ignore this if your extension has no configuration.

Resetting the service

A little known feature of services is that there's only a single instance of a given service existing at any one time. Services instances are kept in a global array and are served again when a service is called multiple times. This may have the undesirable effect that the instance you get is not brand new and shiny, but loaded with data from its previous call. To avoid that, whenever an instance is recalled, TYPO3 will call the reset() method of the service, where any necessary clean up can be performed.

If you think this is needed for the particular service that you are developing, then don't forget to implement the reset() method.

Throwing exceptions

A connector may encounter errors while performing its task that could not be detected upstream during the initialization phase. In such a case, it is recommended to interrupt the process and throw an exception with an informative error message.

Applications using connector services should be ready to receive exceptions and should thus encapsulate calls to any of the "fetch" methods in a try/catch block:

```
try {
	$result = $serviceObject->fetchRaw($parameters);
	// Do something...
}
catch (Exception $e) {
	// Issue error message or log error, whatever...
}
```

Hooks

It doesn't really make sense to have hooks in this base connector class, since it is not supposed to be instantiated directly, but they have been placed as example of hooks that may be useful to other people. If you don't plan to release your extension, you may not be bothered about this, but if you intend to make it public, please consider making available some or all of the following hooks:

- **processParameters**: this hook should be implemented in the query() method. The idea is that it makes it possible to manipulate the parameters of the call and assemble a customized query structure before actually querying the distant source. This should provide enough flexibility to other developers that they can use your connector service without modifications. Calling this hook should replace the normal assembly of the query structure.
  The method called by the hook receives as parameters the array of parameters passed to the query() method and a back-reference to the calling connector object.
- **processResponse**: this hook is designed to process the data inside the query() method, just as it is returned from the distant source. Again this gives the flexibility to manipulate that data for special cases without changing the whole connector. Note that since all "fetch" methods are supposed to call the query() method to get the data from the distant source, this hook actually has an impact on all data fetching methods.
  The method called by the hook receives as parameters the response of the distant source and a back-reference to the calling connector object.
- **processRawData**: this hook is very similar to the "processResponse" hook, but it is designed to be called inside fetchRaw(), so that it will affect the output of that method only.

The method called by the hook receives as parameters the output of the query() method and a back-reference to the calling connector object.

- **processArray**, **processXML**: this is similar to processRawData, but inside the fetchArray() and fetchXML() methods respectively. The first parameter received by the hook's method are the PHP array and the XML string respectively.
- **postProcessOperations**: this hook is designed to perform operations after the process that initially called the connector is done. This can be any kind of clean up that might be necessary.
  The method called by the hook receives as parameters the array of parameters passed to the query() method, a status indicator and a back-reference to the calling connector object. The nature of the status indicator is not clearly defined and will depend on the process that calls back the connector. In the simplest case, it may be a boolean value indicating success or failure.

Format conversions

The extension also provides a utility class. The main method provided by this class is a XML to array conversion utility, which transforms a XML structure into a PHP array without losing any information. This method is simply called as:

```
$phpArray = tx_svconnector_utility::convertXmlToArray($xml)
```

Of course one's own conversion method may be used if needed. The conversion from a PHP array to a XML structure can safely rely on TYPO3's API. e.g.:

```
$xml = t3lib_div::array2xml_cs($phpArray);
```

Again one's own conversion method may be used if needed.

Configuration sample

Since version 2.1.0 of svconnector, the testing BE module can read configuration samples from existing connector services. This makes it easier enter a configuration for testing, in particular to avoid forgetting some important parameter. This sample configuration does not have to be declared in any way, but is expected to be strictly located in `Resources/Public/Samples/Configuration.txt`.

It consists of a simple text file, with one configuration option per line, as you would input it in the BE module. Please check the existing connector services (feed, SQL, CSV) for examples.

To-Do list

There is a roadmap on Forge for the continuing development of this extension:

http://forge.typo3.org/projects/extension-svconnector/roadmap