EXT: Project Tesseract

Extension Key: tesseract
Language: en
Keywords: forAdmins, forIntermediates
Copyright 2008-2012, The Tesseract Team, <typo3@cobweb.ch>

This document is published under the Open Content License
available from http://www.opencontent.org/opl.shtml

The content of this document is related to TYPO3
- a GNU/GPL CMS/Framework available from www.typo3.org

Table of Contents

Introduction

Overview

The Tesseract Project is a suite of extensions that provides a global MVC framework to gather, filter and output data within TYPO3. It defines several types of objects that can interact with one another using a controller and that exchange data between themselves using standardized data formats. This standardization of interfaces and data exchange formats makes it possible to build a very flexible architecture.

The base use of this framework is to be able to query any table in the local TYPO3 database, filter and sort the retrieved records and display them in the FE. However the flexibility of the architecture makes it open to many other uses.

Additional information and fresh news about the project can be found on the dedicated web site: http://www.typo3-tesseract.com/.

What is a tesseract?

A tesseract is a 4-dimensional cube. The project was named after this structure because it symbolizes the various views that can be produced from a single set of data.

Questions?

If you have any questions about this extension, you may want to refer to the Tesseract Project web site (http://www.typo3-tesseract.com/) for support and tutorials. You may also ask questions in the TYPO3 English mailing list (typo3.english).

Keeping the developers happy

If you like this extension, do not hesitate to rate it. Go the Extension Repository, search for this extension, click on its title to go to the details view, then click on the "Ratings" tab and vote (you need to be logged in). Every new vote keeps the developers ticking. So just do it!

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.

Fundamentals

Displaying list of things (news items, flats, cars, movies, etc.) is a very common task in any web site. It is generally expected that such lists can be filtered and sorted. Each item should link to a detail view with cross-linking to related items (like actors to a movie, for example).

The common way to solve such tasks in TYPO3 is to create a specific extension, dedicated to a particular topic (cars, movies, etc.). Although TYPO3 does provide some libraries for easing this task, it still means a multiplication of extensions that basically perform the same task.

The aim of the Tesseract Project is to provide a generic way of achieving this, to avoid the need for repetitive, dedicated extensions. Such generalization can be achieved by defining a clean API and data exchange standards, orchestrated by controllers.

The main concept

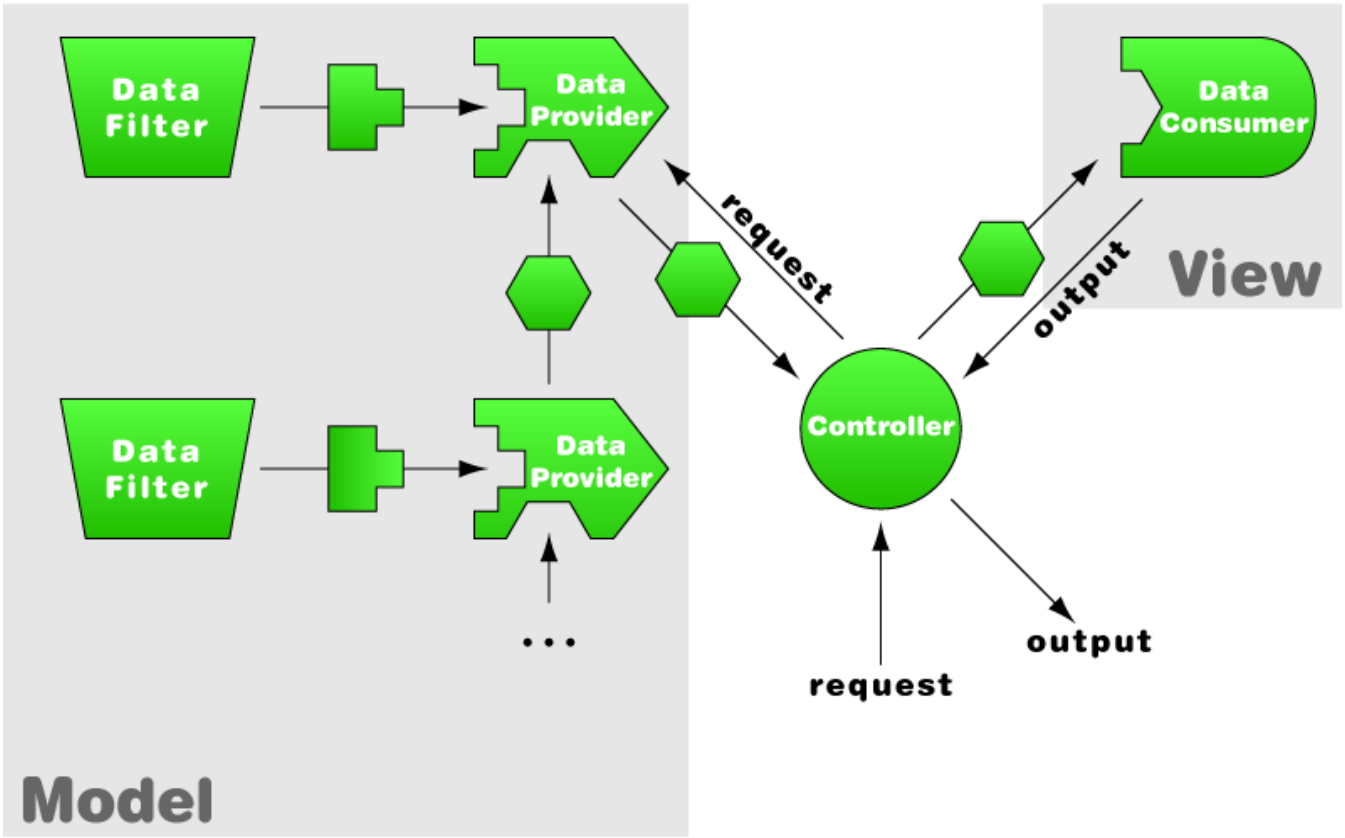
The basic idea is the following: there can exist a number of tools capable of extracting data from any number of sources, be it the TYPO3 database, external databases or other formats (plain text files, RSS feeds, etc.). Those tools are expected to provide that data in a standardized data structure (SDS). If the data comes from the TYPO3 database, the tools are expected to have done a clean work as far as handling the deleted flag, the enable fields and the language overlays are concerned (and – in an ideal world – also the workspace overlays). These tools are called **Data Providers** (or simply Providers).

There can be more than a single type of standardized data structure, but their number must be kept very low so that interoperability remains easy to implement. At the moment two SDS types exist: full set of records (*recordset*) and list of primary keys (*idlist*).

This standardized data is then used by output engines called by the controller. Each output engine is capable of handling one type of SDS. These output engines are called **Data Consumers** (or simply Consumers). Their goal is to produce some kind of output with the SDS they have received. In particular, this can be HTML code to be displayed in the TYPO3 frontend.

The Data Providers can receive input from other Data Providers to rework the data. A Data Provider can also receive input from a **Data Filter** (or simply Filter). This is another kind of standardized structure that describes filters to apply to the data, as well as sorting and capping (limit) parameters.

This architecture can be represented roughly with the following schema:



Standardized data structures (SDS)

Two SDS are defined and in use for now.

Full Recordset SDS (recordset)

The full recordset SDS is basically designed to return the record set resulting from a database query with added information and a (optional) recursive structure for joined tables. Obviously other sources than a database query can also return such a SDS.

An example is the best way to understand how this SDS is structured. Let's say you perform the following query using Data Query:

```
SELECT pages.uid, pages.title, tt_content.uid, tt_content.header FROM pages LEFT
JOIN tt_content ON tt_content.pid = pages.uid
```

In a usual situation you would get your result in a structure like this:

0	(value of pages.uid)	(value of pages.title)	(value of tt_content.uid)	(value of tt_content.header)
1	...			

The full recordset SDS instead provides you with is a PHP array corresponding to the list of pages retrieved, where each page has a subtable containing its related content elements. The structure will look like this:

name	pages
count	Number of records returned (for table pages, not counting joined records)
totalCount	Total number of records without limits applied (except if limit was hard-coded in SQL query)
uidList	Comma-separated list of uid's, usable directly by TYPO3
header	uid: label: uid title: label: Title:
filter	Full Data Filter structure (see below for details about that structure) Note that this was introduced at a later point, so it may not be available on older versions of Tesseract components.
records	0: mainid: (value); title: (value); __substructure: tt_content: name: tt_content; count: Number of records returned (for table tt_content); uidList: Comma-separated list of uid's, usable directly by TYPO3; header: uid: label: uid; header: label: Header:; records: 0: uid: (value); header: (value); 1: ...; 2: ... 1: ...

As you can see, the data from the main table (the one in the FROM statement) and the data from each of the subtables (the ones in the JOIN statements) are separated for ease of use. Furthermore the "pages." or "tt_content." prefix that was necessary for disambiguating the fields in the SELECT statement are removed. Again this makes it easier to use. It also becomes possible to add data like a count of all records or a comma-separated list of all primary keys, which could be quite convenient to manipulate in TypoScript during the display.

Once inside records additional fields can be added but it is advised to prefix them with "__" (two underscores) to avoid overwriting a field from the database. The prime example for such fields is the one containing the subtable, which is called

"__substructure". Any other field can be imagined, although none are currently used.

The structure for a subtable is absolutely identical to the data for the main table. There can be any number of subtables for a given record and any level of recursion is possible. How many recursion levels can be prepared depends on the capabilities of the specific Data Provider. Whether all the levels can be used or not depends on the specific Data Consumer. The name of each subtable is used as associative index in the array of subtables.

List of primary keys SDS (idlist)

The idlist SDS is a simpler structure which is basically designed to return a list of primary keys. However these keys can be related to different tables. The structure is schematically described below:

uniqueTable	Name of a table if all primary keys come from the same table, empty otherwise
uidList	Comma-separated list of all primary keys
uidListWithTable	Comma-separated list of all primary keys including table names (*à la* RECORDS cObj). Example: pages_12,tt_content_23,tt_address_58
count	Number of records returned with limits applied
totalCount	Total number of records without limits applied
filter	Full Data Filter structure (see below for details about that structure) Note that this was introduced at a later point, so it may not be available on older versions of Tesseract components.

Data Filters Structure

Data Filters have their own standard structure. This structure makes it possible to define filters for the data that's going to be provided by the Data Providers. The Data Provider is expected to be able to understand such a structure and act on it. On top of true filters, the Data Filter can also be used to pass on data related to the ordering of the results or limiting the number of results.

Here's how such a structure looks like:

<table>
<tr><td>filters</td><td>
<table>
<tr><td>0</td><td>
<table>
<tr><td>table</td><td>Table to which the filter applies</td></tr>
<tr><td>field</td><td>Field to which the filter applies</td></tr>
<tr><td>conditions</td><td>
<table>
<tr><td>0</td><td>
<table>
<tr><td>operator</td><td>Operator to use for the test. Allowed values are: =, !=, >, >=, <, <=</td></tr>
<tr><td>value</td><td>Value to use for the test</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>...</td></tr>
</table>
</td></tr>
<tr><td>main</td><td>True if the condition should be applied in the main condition, false otherwise</td></tr>
<tr><td>void</td><td>True if the condition should be applied at all, false otherwise (see the "Data Filter" manual for a discussion on the usefulness of this kind of conditions)</td></tr>
<tr><td>string</td><td>Should contain the original filter information, untransformed (this can be used for debugging)</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>...</td></tr>
</table>
</td></tr>
<tr><td>logicalOperator</td><td>Logical operator used to link all filters together (AND/OR)</td></tr>
<tr><td>limit</td><td>
<table>
<tr><td>max</td><td>Maximum number of records to retrieve</td></tr>
<tr><td>offset</td><td>Number of records to shift from 0 as a multiple of max (i.e. shift is max * offset)</td></tr>
<tr><td>pointer</td><td>Direct pointer to a given record number (i.e. if pointer = 54, jump to the 54th record)</td></tr>
</table>
</td></tr>
<tr><td>orderby</td><td>
<table>
<tr><td>0</td><td>
<table>
<tr><td>table</td><td>Name of the table to sort</td></tr>
<tr><td>field</td><td>Name of the field to sort on</td></tr>
<tr><td>order</td><td>Direction of the ordering (asc/desc)</td></tr>
</table>
</td></tr>
<tr><td>1</td><td>...</td></tr>
</table>
</td></tr>
</table>

Custom properties may be added by extensions (e.g. the Data Query extension adds a “raw SQL” field).

Special values

A number of special values exist for the values found in conditions. Any implementation of the Data Filter pattern should be able to handle those values, if relevant. Special values begin with a backslash (\). They are expected to be handled correctly by the Data Filter (if needed) and to be interpreted by the Data Provider. The table below lists the base special values that should be recognized and how they should be handled by the Data Provider:

Name	Data Filter	Data Provider
\empty	Depending on the Data Filter implementation, it may not be easy to distinguish between an unset value and an empty one. The unset value should be ignored whereas the empty one should be equivalent to an empty string (or anything similar given the context). This special value should be used to explicitly mean an empty value.	This value should probably be interpreted as an empty string, or whatever is equivalent for each particular Data Provider.
\null	There's the same problem as with \empty, to distinguish between an unset value and a real test against null.	This value must be interpreted as a null or whatever is equivalent for each particular Data Provider.
\all	In some particular cases, it may be convenient to have a kind of wildcard value that means that the condition is actually not really a condition but accepts all values. This is the purpose of the \all keyword.	This value must be interpreted as meaning that all values for the given condition must be accepted. This probably means that the condition must simply be ignored by the Data Provider, but it will depend on specific implementations (it could be equivalent to “*” in some situations, for example).

A particular Data Filter implementation may use its own special values for its own purposes, but it should not expect a Data Provider to recognize them.

Explanations

A Data Filter's result is passed on to a Data Provider. It is up to each Data Provider to interpret the information it receives from the Data Filter. The Data Query, for example, transforms the Data Filter information into SQL statements. So the words “table” and “field” above must be considered from a general point of view, where they might not necessarily mean a database table and a column of that database table.

The same goes for the “main condition” mentioned above. Whatever the Data Provider does, it has to interpret all the conditions of the filter. When interpreted into SQL, conditions could either go to the WHERE clause (the “main condition” in this case) or to the ON clause of the relevant JOINs. The “main” flag is an indication to the Data Provider that it should apply this particular condition to the “main condition” (e.g. the WHERE clause) if it makes such a difference at all.

Note that wherever a “table” property exists, it may be empty. It is up to the Data Provider to know what to do in such a case. For example, it should know which is the main table it is querying and assume that such a filter applies to that table.

Currently available resources

Here is a list of the currently available Tesseract components.

Controllers

Display Controller (displaycontroller)

The Display Controller is a controller that acts as FE plug-in. It thus coordinates the work of the other components to produce output that is integrated into the TYPO3 page rendering process. The Display Controller defines relationships between the various components. This is described in details in the Display Controller manual, but here's a short overview to relate this to the general schema shown above.

The Display Controller refers to one Data Consumer for the display itself. It also refers to a Data Provider that feeds the Data Consumer. This provider is called "Primary Provider". It can also refer to a so-called "Secondary Provider" which feeds into the Primary Provider. Each provider can be linked to a Data Filter.

Advanced Display Controller (displaycontroller_advanced)

This controller works like a "multiple" Display Controller. It can take several groups of Data Providers and Data Filters and pass them to a single Data Consumer. It makes it possible to create sophisticated structures. Note that "templatedisplay" is not able to handle such constructs.

Data Providers

Data Query (dataquery)

Data Query is a SQL-based Data Provider. This means it relies on SQL queries to select data that is then transformed into a recordset-type SDS. Data Query hides all the handling of the TYPO3-specific fields (deleted flag and enable fields) and also transparently handles language overlays.

It can receive filtering information from a Data Filter, which is transformed into SQL syntax.

In relation with the Display Controller, it acts as a Primary Provider.

Google Query (googlequery)

Google Query is a Data Provider based on Google Mini or Google Search Appliance. It actually contains two providers, one returning recordset-type SDS (which acts as a Primary Provider with the Display Controller) and one returning idlist-type SDS (which acts as a Secondary Provider with the Display Controller).

Tag Pack Provider (tagpackprovider)

This Data Provider is based on the Tag Pack extension. It uses tags to make selections of records from various tables and returns the keys to those records in a idlist-type SDS. It can act as a Secondary Provider with the Display Controller.

The Tag Pack Provider does not interact with Data Filters.

Data Consumers

Template-based display (templatedisplay)

This extension provides a way to transform the data returned by Data Providers into HTML using templates based on TYPO3's famous ### marker syntax. Thanks to a nice visual interface, it's possible to simply click on a marker and match it to a field from the list of available data. Local TypoScript processing can then be added for rendering.

Fluid-based display (fluiddisplay)

This extensions makes it possible to use the Fluid templating engine for rendering. This makes it possible to benefit from all the power and feature of Fluid, plus the numerous view helpers developed by the community.

PHP-based display (phpdisplay)

For even more flexibility, this extension makes it possible to use raw PHP for rendering. Useful in special cases where none of the above Data Consumer are enough.

Data Filters

Data Filter (datafilter)

The Data Filter extension is currently the only implementation of the Data Filter concept. It is a very flexible tool allowing to pick data from a variety of sources (in particular GET/POST variables, but also contextual values set in TypoScript or results from some common PHP functions like date()) and set it to be matched against fields used by Data Providers, so as to restrict the records that those tools return.

It also handles sorting (field and order) and limit and offset parameters.

Utilities

Tesseract (tesseract)

This very extension is of course the basic brick of the Tesseract architecture. On top of this general documentation, it provides TYPO3 services for each of component types used by Tesseract (controller, data provider, data consumer, data filter), PHP interfaces for defining the API and some utility classes.

Improved Overlays API (overlays)

This extension is a tool box designed to simplify work with language overlays. On the one hand it provides very general methods which make it easy to retrieve properly overlaid records. On the other hand it also provides a number of very basic methods that handle such tasks as assembling the proper language fields conditions in a SQL statement, based on TCA definitions.

Generic Expression Parser (expressions)

This extension provides a library inspired by TypoScript's getText function. It makes it possible to retrieve values from a variety of sources with a simple syntax and provides a simple API for parsing strings containing such expressions. Retrieved values can also be post-processed by functions, which make it possible, for example, to quote strings tso that they are safe to use inside a SQL query.

The expression parser is at the heart of the Data Filter extension. It is also used by several other Tesseract components.

Contexts (context)

This extension is still in development. It aims to replace the current context mechanism found in the Display Controller.

One major aspect of Tesseract is to be able to reuse components in various places of a given web site. In particular Data Providers, which often to query the same data but with different parameters. This is possible thanks to Data Filters. Contexts take this one step further, by making it possible to define values in TypoScript which can be retrieved easily by Data Filters. This allows to have a given Data Provider be the same for a whole site and vary the content it returns just thanks to some TypoScript values that vary along the page tree.

Ideally the Contexts extension should provide a simple way to work with these values, rather than having to edit TypoScript templates.

Installation

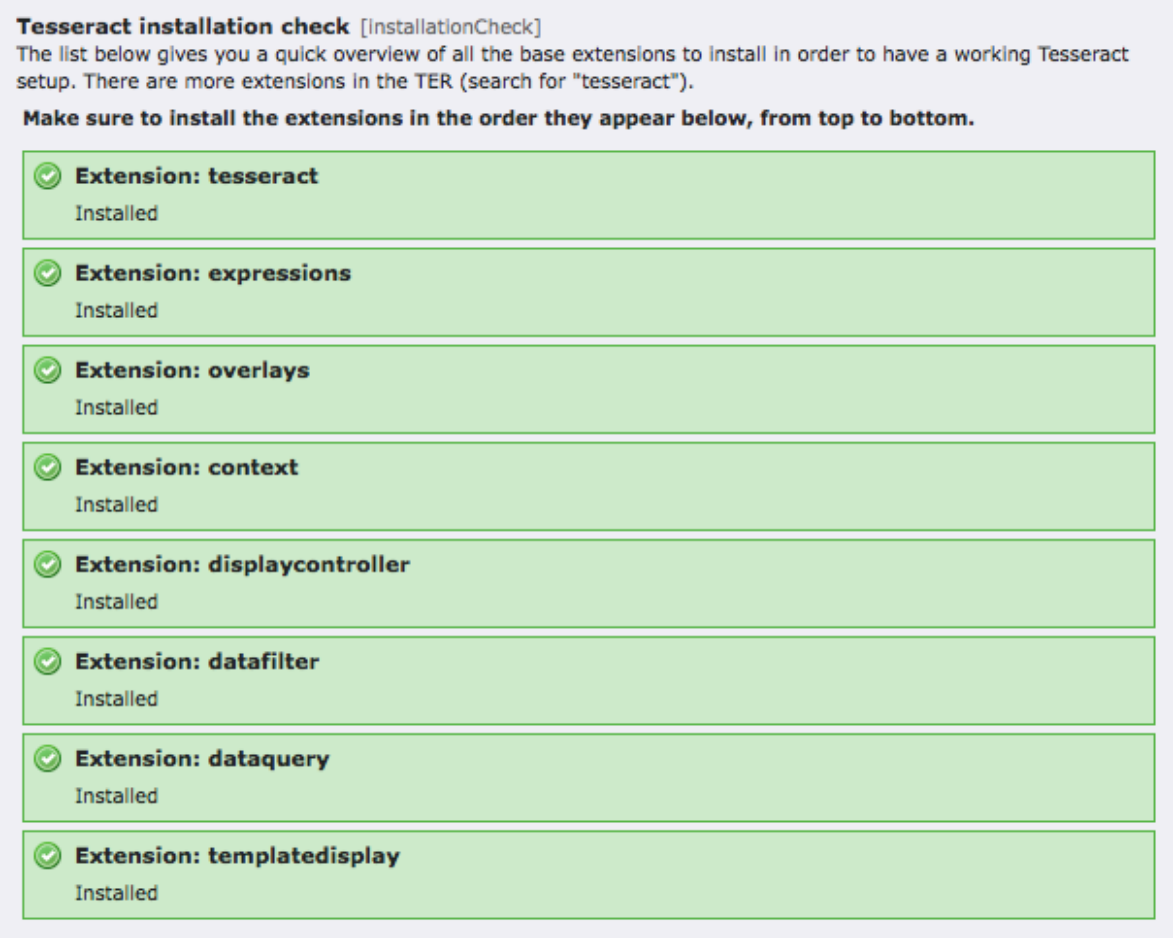
As was described above the Tesseract Project is not a single extension but a whole suite of them. To facilitate installation, extension “tesseract” makes suggestions for all extensions that should be installed. All these suggestions should be accepted, in order to have a complete set-up. Some of the suggested extensions in turn have requirements and suggestions.

Refer to the manual of each extension for specific installation details.

Installing “tesseract” alone does nothing as this extension only provides this documentation and a number of bases classes used by the other extensions.

The extension configuration screen provides a quick way to check if the main extensions that make up Tesseract are installed. Tesseract is of a modular nature. This means that you may perfectly run without some of the suggested extensions. You may also have other extensions installed, but which won't appear in the this check screen.

A typical screen may look like the one below. **Note that extensions must be installed in the exact order in which they are presented in this screen!**



Compatibility issues

In general the Tesseract Project requires TYPO3 4.3+ to run. It will not run with a lower version. It also requires PHP 5.2 or above.

Upgrading to version 1.3.0

Version 1.3.0 of extension "tesseract" came with some API changes. A new base class was introduced for all components: `tx_tesseract_component`. This class is already extended by all the existing consumer, filter and provider classes, so this change should be transparent for any Tesseract component, unless it did not extend the existing base classes (which is not recommended).

However these changes have an impact on controllers. If you have designed a custom Tesseract controller, you should check the changes to the "displaycontroller" or "displaycontroller_advanced" extensions to check what should be changed in your own controllers.

Note: at the time of this writing, there are no custom controllers out there known to the authors, which is why we allowed ourselves such an API change.

Configuration

As of Tesseract 1.1.0, it is possible to use TypoScript to configure default values for the Data Filters, Data Providers and Data Consumers. The general syntax is the following:

```
config.tx_tesseract.[table name].default.[field name] = foo
```

Example:

Let's say you want to have a default value of 20 for the "Max items per view" in the "datafilter" extension. The syntax would be:

```
config.tx_tesseract.tx_datafilter_filters.default.limit_start = 20
```

Developer's Guide

API

Each object type is expected to implement a specific interface so as to be able to behave properly. These interfaces are provided by the Tesseract extension.

Base component

All providers, filters and consumers are supposed to inherit from a base Tesseract component class (`tx_tesseract_component`). This class defines the following methods:

Name	Description
loadData()	This method is used to load the necessary information about the component. It takes a type and a primary key and retrieves the information from the database. It is the proper way of retrieving such data. The `setData()` method (see below) is used in special cases.
setData()	This method can be used to (forcefully) set the information related to the Tesseract component. This is usually achieved via `loadData()`, but is sometimes useful. In particular it is used in the context of unit testing.
getData()	This method returns the information related to the Tesseract component.

Data Provider

The DataProvider interface defines the methods that a data source must implement in order to be recognized as a Data Provider. These methods are:

Name	Description
getProvidedDataStructures()	This method returns a list (array) of all SDS types that the Data Provider can return (so one or more of "recordset", "idlist", "tree" or any other that may be added at a later point).
providesDataStructure()	This method takes a SDS type as an input and returns true or false depending on whether the Data Provider can provide it as an output or not.
getAcceptedDataStructures()	This method returns a list (array) of all SDS types that the Data Provider can accept as an input.
acceptsDataStructure()	This method takes a SDS type as an input and returns true or false depending on whether the Data Provider can handle it as an input or not.
initEmptyDataStructure()	This method is used to prepare an empty SDS, for example setting an empty "records" array and set the count to 0.
getDataStructure()	This method returns the SDS created by the Data Provider
setDataStructure()	This method is used to pass to the Data Provider an input SDS
setDataFilter()	This method is used to pass a Data Filter structure to the Data Provider.
getTablesAndFields()	This method returns a list of tables and fields available in the data structure, complete with localized labels.
setEmptyDataStructureFlag()	This method is used to set a flag that indicates whether an empty SDS should be returned or not.
getEmptyDataStructureFlag()	This method is used to retrieve the status of the flag mentioned above.

Data Consumer

Similarly the DataConsumer interface defines what methods are expected from a class in order to mark it as a Data Consumer. The methods are:

Name	Description
getAcceptedDataStructures()	This method returns a list (array) of all SDS types that the Data Consumer can accept as an input.
acceptsDataStructure()	This method takes a SDS type as an input and returns true or false depending on whether the Data Consumer can handle it as an input or not.
setDataStructure()	This method is used to pass to the Data Consumer an input SDS
setDataFilter()	This method is used to pass a Data Filter structure to the Data Consumer.
startProcess()	This method tells the Data Consumer to start rendering the SDS into its final output, but that output is not returned yet. It is just stored internally.

Name	Description
getResult()	This method gets the output produced by the Data Consumer. This means it's normally called after startProcess(). It can also be called directly (i.e. not after startProcess()) when the Data Consumer should not be passed any structure. In this case, it is expected to render some kind of error output. This situation is likely to happen when a filter didn't return any filtering values. In this you may either want to get everything from the Data Provider, or nothing. In the latter case it's not necessary to call the Data Provider at all and the Data Consumer may be called without SDS.

Data Filters

The same goes again for the Data Filters, with the DataFilter interface:

Name	Description
getFilterStructure()	This method processes the Data Filter's configuration and returns the filter structure.
isFilterEmpty()	This method returns true if the "filters" part of the filter structure is empty, false otherwise.
getFilter()	This method is called to get the Data Filter itself. It will start the processing of the filter configuration and return the standard Data Filter structure.
setFilter()	This method is used to pass an existing Data Filter structure to the Data Filter. This will normally be a Data Filter structure that was saved in cache.
saveFilter()	This method saves the Data Filter into the transient session object ("ses").

Controller

Each controller will probably have very different logics. There are however some minimal requirements represented by two different PHP interfaces.

When setting up relationships between components a service exists to "link back" from a Consumer to a Provider. Let's take the example of the Template Display Consumer. It provides an interface for mapping fields retrieved by a related Provider to markers in a HTML template. To be able to achieve that from within the TYPO3 BE, Template Display must be "put in touch" with a Provider to which it is related via a controller. The controller service exists for that purpose.

It implements the following interface (`tx_tesseract_datacontroller`):

Name	Description
loadData()	This method takes whatever information is relevant to identify the controller and performs any appropriate action.
getRelatedProvider()	This method looks for the Data Consumer that is managed by that particular controller instance and returns it as a Data Consumer object.

The "other side" of the process is when the controller prepares for output. A different interface must be implemented for that purprose (`tx_tesseract_datacontroller_output`):

Name	Description
getPrefixId()	This method is expected to return a key uniquely identifying the controller. In the fronted this would typically take the form of "tx_*extensionname*" to be used in prefixing GET/POST vars.
addMessage	Add a message to the debugging message queue.
getMessageQueue	Get all the messages in the debugging message queue.
getMessageQueueForKey	Get messages in the debugging message queue related to a specific key (normally a particular Tesseract component).

`tx_tesseract_datacontroller_output::getPrefixId()` is used – for example – by extension "templatedisplay" while assembling links with query variables. The Display Controller extension returns "tx_displaycontroller" as a prefix, so "templatedisplay" will use the common TYPO3 syntax:

```
tx_displaycontroller[foo]
```

When creating a link with query variables. This ensure that the controller knows how to retrieve variables that are meaning for itself. In the case of the "displaycontroller", all such properly constructed variables will be available in the well-know `piVars` array.

Each controller is expected to manage an internal message queue for storing debugging messages. Any Tesseract component can add a message to that queue by referring to its controller instance. Example:

```
$this->controller->addMessage(
      'extensionkey_uid',
      'This is the debug message',
      'Title of the message (optional)',
      t3lib_FlashMessage::OK,
      array('foo', 'bar')
);
```

The controller should store such messages only if some debugging mode is active. It is then expected to also display or store these messages in some way to help debugging. The messages should by stored by key, the key helping identify which component sent the message (so the key will usually be the extension key of the component, plus the uid of the specific instance (or some other identifier, should the component not be related to a database record)). This could then be used by said component to retrieve its messages (using `getMessageQueueForKey()`), although this is not currently used and does not seem useful at the moment.

Known problems

None to date. If you have any issues, please refer to the Tesseract Project web site (http://www.typo3-tesseract.com/). You may also post your problems to the TYPO3 English mailing list (typo3.english), so that others may benefit from the answers too. For bugs or feature requests, please open an entry in the extension's bug tracker on Forge (http://forge.typo3.org/projects/extension-tesseract/issues).

To-Do List

The roadmap for the evolution of this extension can be found on Forge: http://forge.typo3.org/projects/roadmap/extension-tesseract

For feature requests, please open a report on Forge issue tracker: http://forge.typo3.org/projects/extension-tesseract/issues