Server Side Pagination

Example Files (Builder project Advanced Modeling/UI):

<your example path>\Advanced Modeling\UI\uml\uiServerSideTablePagination.xml

On this Page:

- Configuring the Table Widget
- Extending the Table Data Service
- TableFilter Classes
  - TableFilterRequest Class
  - TableFilterResponse Class
- The Usage of Filter Classes
- Explicit Pagination Callback Service
- Download the Filtered Data

Related Pages:

- Open/Close Row Details

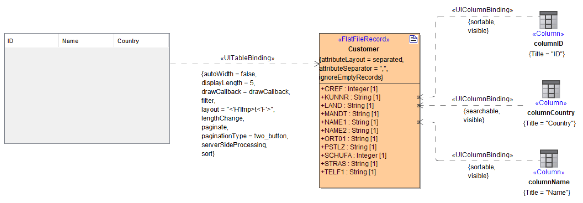
The table widget allows - next to the standard client side pagination - also for server side pagination. When to use which version depends on the system design. But in generally, server side processing is ideal if:

- large amounts of data are handled
- faster initial page load is required
- the service should be resilient to concurrent changes

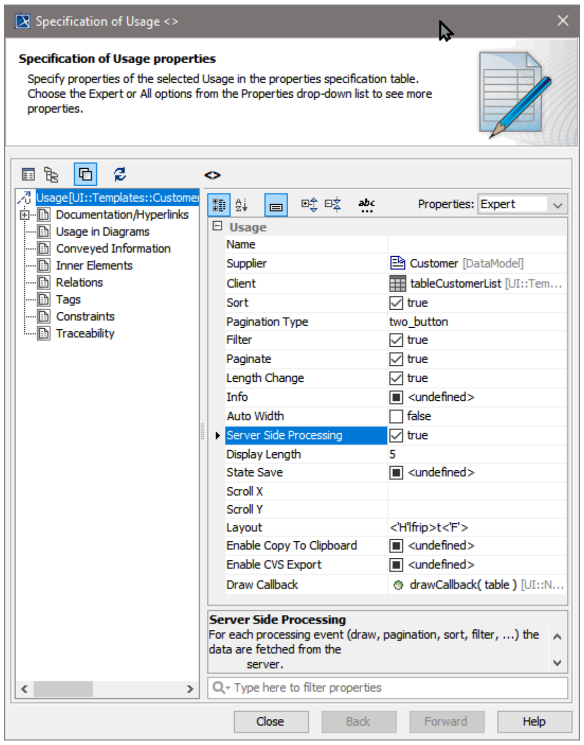
To enable server side pagination, you need to setup the frontend / UI part and the server side processing. The latter is not part of this UI documentation but a working example will give some insight on what is needed. The frontend part consists of configuring the table widget and binding as well as extending the service call's parameters.

Configuring the Table Widget

The Widget is configured on the <<UITableBinding>> dependency.



Within the tagged value section of the Specification Window, enable the tagged value Server Side Processing.



Specification of Usage properties

Specify properties of the selected Usage in the properties specification table. Choose the Expert or All options from the Properties drop-down list to see more properties.

<>

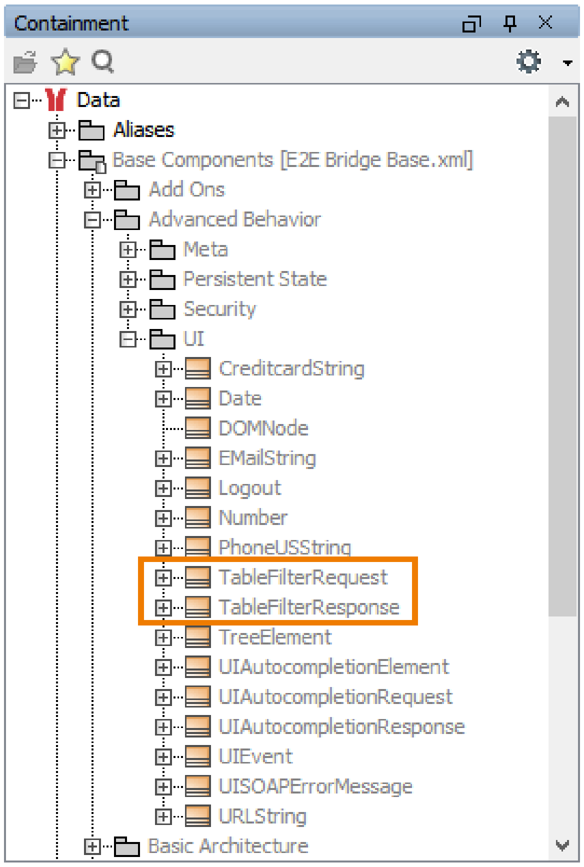
Usage	
Name	
Supplier	Customer [DataModel]
Client	tableCustomerList [UI::Tem...
Sort	✓ true
Pagination Type	two_button
Filter	✓ true
Paginate	✓ true
Length Change	✓ true
Info	■ <undefined>
Auto Width	☐ false
Server Side Processing	✓ true
Display Length	5
State Save	■ <undefined>
Scroll X	
Scroll Y	
Layout	<'H'lfrip>t<'F'>
Enable Copy To Clipboard	■ <undefined>
Enable CVS Export	■ <undefined>
Draw Callback	drawCallback(table) [UI::N...

Server Side Processing
For each processing event (draw, pagination, sort, filter, ...) the data are fetched from the server.

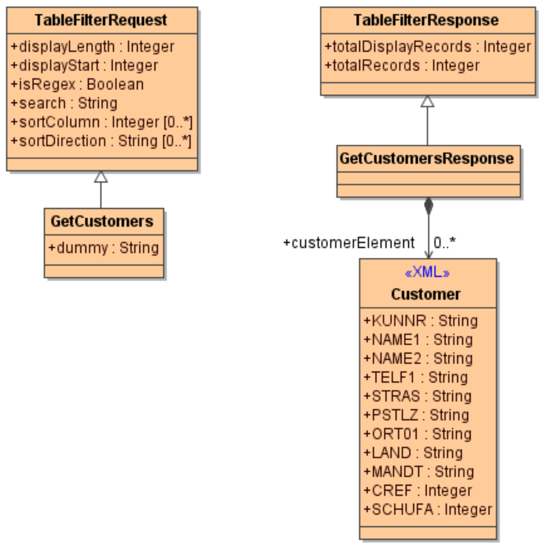
So far, this is all that has to be done in the widget configuration area.

Extending the Table Data Service

To be able to process data page sets on the server using e.g. SQL queries, additional information of the current table widget page set needs to be added to the request. For the model this means adding an additional class to the request and response class used by the service. These additional classes are **Tabl eFilterRequest** and **TableFilterResponse**, both a standard class within the **Base Components** repository.



The data service call request and response classes need to be extended using a generalization so that the table filter classes attributes get inherited as the following figure shows:



TableFilter Classes

The following tables give details on how to use the attributes of the TableFilter classes.

TableFilterRequest Class

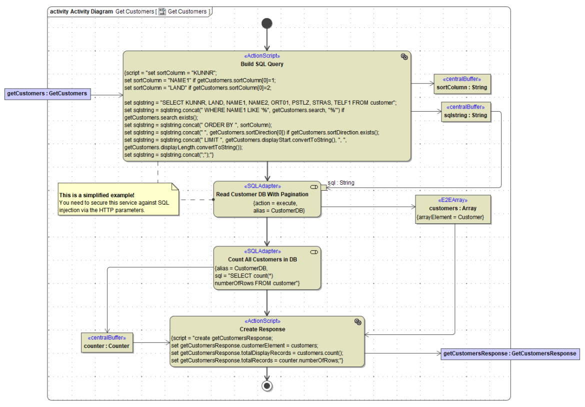
Attribute name	Description	Values	
displayLength	Contains the number of records to display.		
displayStart	Contains the index number of the first record to display.		
isRegex	Specify whether the search field contains a regular expression.	true	Search field is a regular expression.
		false	Search field contains a search term only.
search	Contains the global search field data. This attribute can be used as a search criteria within a SQL where clause.		
sortColumn [0..*]	Contains an array of column numbers to be sorted after. • If the user clicks to sort on one column, this array contains one element only. • If the user clicks more than one column holding the **Shift** key, all clicked columns are sorted hierarchically and are listed in this array in clicked order.		
sortColumn [0..*]	Contains an array of sort orders corresponding to the **sortColumn** array.	asc	Sort ascending.
		desc	Sort descending.

TableFilterResponse Class

Attribute name	Description
totalRecords	Contains the total number of records, before filtering (i.e. the total number of records in the table).
totalDisplayRecords	Contains the total number of records after the filtering has been applied (not just the number of records being returned in this result set).

The Usage of Filter Classes

As described previously, the TableFilter classes hold information which is essential for the server side processing. The following activity diagram is an example on how to use the attributes to retrieve and send back the correct paging data.


activity Activity Diagram Get Customers [Get Customers]
getCustomers : GetCustomers
«ActionScript»
Build SQL Query
«centralBuffer»
sortColumn : String
«centralBuffer»
sqlstring : String
sql : String
This is a simplified example!
You need to secure this service against SQL injection via the HTTP parameters.
«SQLAdapter»
Read Customer DB With Pagination
«E2EArray»
customers : Array
{arrayElement = Customer}
«SQLAdapter»
Count All Customers in DB
«centralBuffer»
counter : Counter
«ActionScript»
Create Response
getCustomersResponse : GetCustomersResponse

```
{script = "set sortColumn = "KUNNR";
set sortColumn = "NAME1" if getCustomers.sortColumn[0]=1;
set sortColumn = "LAND" if getCustomers.sortColumn[0]=2;

set sqlstring = "SELECT KUNNR, LAND, NAME1, NAME2, ORT01, PSTLZ, STRAS, TELF1 FROM customer";
set sqlstring = sqlstring.concat(" WHERE NAME1 LIKE '%", getCustomers.search, "%'") if
getCustomers.search.exists();
set sqlstring = sqlstring.concat(" ORDER BY ", sortColumn);
set sqlstring = sqlstring.concat(" ", getCustomers.sortDirection[0]) if getCustomers.sortDirection.exists();
set sqlstring = sqlstring.concat(" LIMIT ", getCustomers.displayStart.convertToString(), ", ",
getCustomers.displayLength.convertToString());
set sqlstring = sqlstring.concat(";");"}
```

```
{action = execute,
alias = CustomerDB}
```

```
{alias = CustomerDB,
sql = "SELECT count(*)
numberOfRows FROM customer"}
```

```
{script = "create getCustomersResponse;
set getCustomersResponse.customerElement = customers;
set getCustomersResponse.totalDisplayRecords = customers.count();
set getCustomersResponse.totalRecords = counter.numberOfRows;"}
```

The activity above shows how the parameters of the **TableFilter** classes are used to query a database and receiving the data which the widget dictates to retrieve. In the example, the query is build dynamically to respect the **search** and **order** parameters.

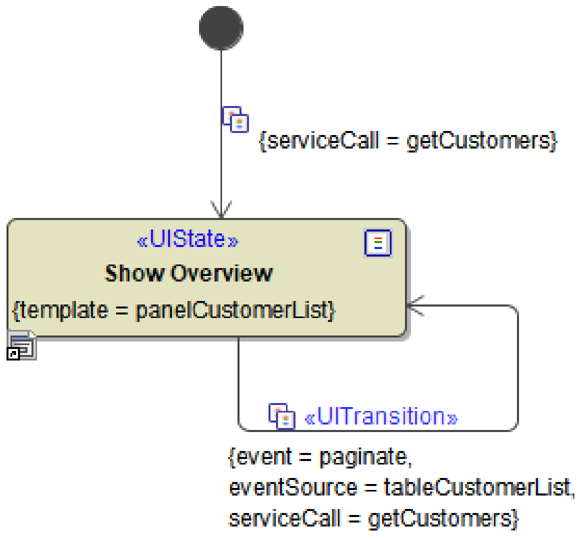
Action	Description
Build SQL Query	This is where the SQL statement is build. For the pagination, it is important to identify the position of the current and next data set (page) to retrieve. In the SQL statement, we use the `LIMIT` keyword for reading pagewise: `LIMIT getCustomers.displayStart.convertToString(), getCustomers.displayLength.getCustomers.displayStart.convertToString()` As it is possible to define a search key, the `ORDER BY` clause is set by the **search** criteria delivered by the service request input. In the above example, the search criteria is set in case it exists.
Read Customer DB with Pagination	This is where the SQLite database is queried. The result will be a data set which corresponds to the page size of the table widget. The result of the query will be the array of **customers**.
Count All Customers in DB	The output **getCustomerResponse** needs to know the **totalRecords** that the entire database table holds. The result of the SQLite query gives back the count value.
Create Response	As a last step the the response gets populated with the data needed for the table widget to display the data set (page). The actual data (**customers**) is assigned to the **customerElement** of the response. This data will be displayed in the widget as it is defined by the tables binding. The **totalDisplayRecords** holds the count of the actual data set result and is needed for the widget to do calculations for further paging requests as well as the third parameter **totalRecords**.

The example handles the paging using a SQLite database, and shows how the logic should be used in general. Other systems will allow similar handling of paging mechanisms or even need some more complex logic.

Explicit Pagination Callback Service

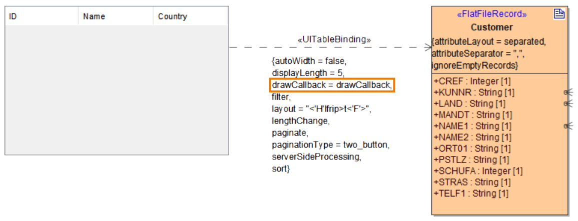
By default, the **last service providing the table data** will also be called when paginating through the table. This is feasible for simple cases - however, often it makes more sense to explicitly define a service to be called for pagination events.

In the following example the service **getCustomers** is called to initially build the table, but for pagination events the service **paginationCallback** is invoked. Both services must follow the same implementation rules as described above. The only difference is that the pagination service is called by a <<UITransition>> triggered by an **paginate** event. In contrast to other UI transitions it must start and end at the *same* state. Additionally, calling scripts or defining timeouts is not allowed on such a transition.

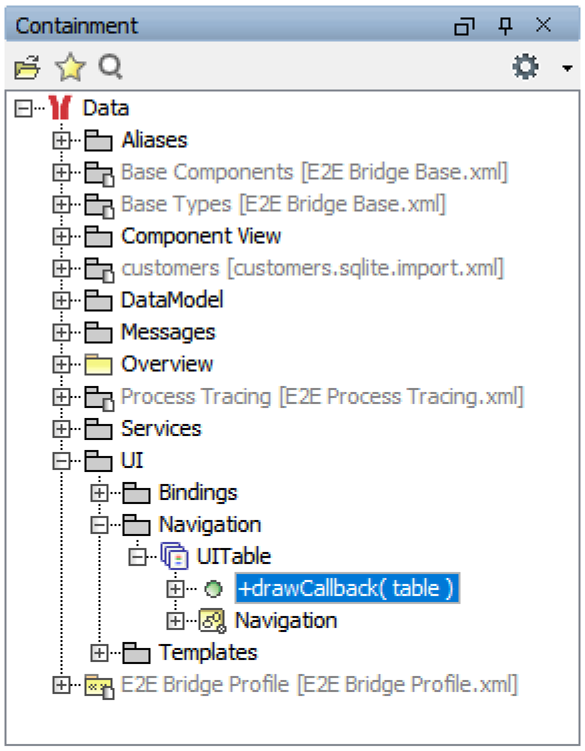

{serviceCall = getCustomers}
«UIState»
Show Overview
{template = panelCustomerList}
«UITransition»
{event = paginate,
eventSource = tableCustomerList,
serviceCall = getCustomers}

Download the Filtered Data

To provide a link to download not only the table page but the filtered table data, you need to define a **dra wCallback** that contains JavaScript to calculate the download link. Every time the table is drawn anew (e. g. due to filter changes) the download link will be updated.



The callback resides in the UI navigation:



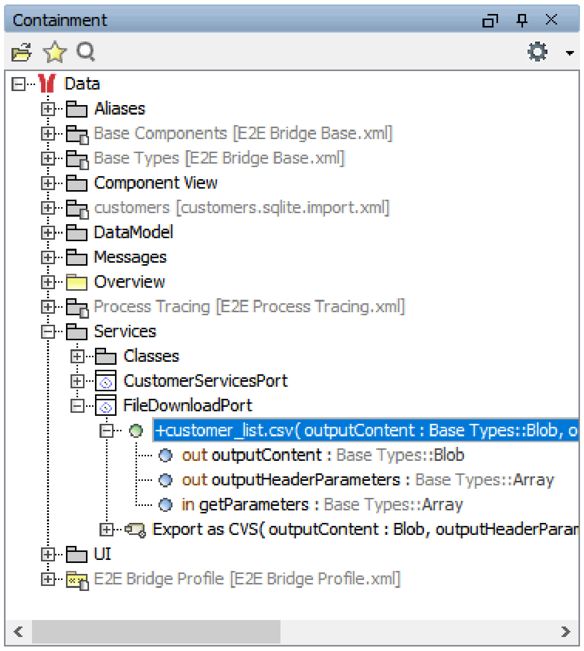
```
Data
    Aliases
    Base Components [E2E Bridge Base.xml]
    Base Types [E2E Bridge Base.xml]
    Component View
    customers [customers.sqlite.import.xml]
    DataModel
    Messages
    Overview
    Process Tracing [E2E Process Tracing.xml]
    Services
    UI
        Bindings
        Navigation
            UITable
                +drawCallback( table )
                Navigation
        Templates
    E2E Bridge Profile [E2E Bridge Profile.xml]
```

Add a script like

```
var settings = $(table).dataTable().fnSettings();

var query = "?displayStart=" + settings._iDisplayStart + "&displayLength="
+ settings._iDisplayLength;
query += "&search=" + settings.oPreviousSearch.sSearch;

for(var i = 0; i < settings.aaSorting.length; i++){
    query += "&sortColumn=" + settings.aaSorting[i][0];
    query += "&sortDirection=" + settings.aaSorting[i][1];
}

$("#ID::linkExport").attr("href", "../../data/download/customer_list.csv"
+ query);
```

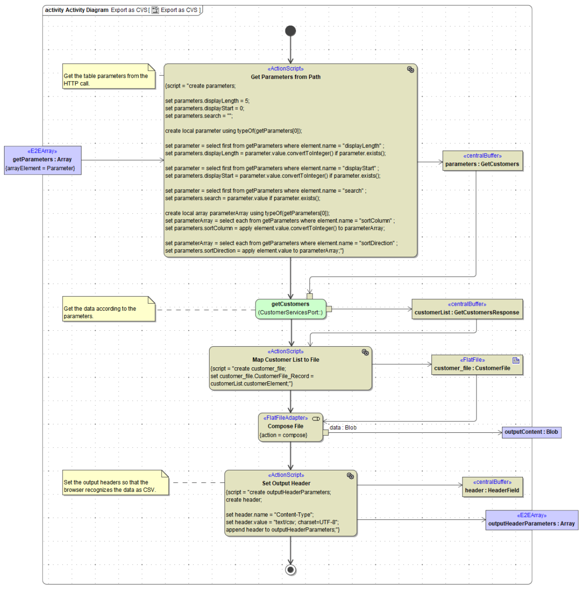
that builds the download link and adds the table parameters.

The link points to an operations that allows to download the displayed data:



```
Data
    Aliases
    Base Components [E2E Bridge Base.xml]
    Base Types [E2E Bridge Base.xml]
    Component View
    customers [customers.sqlite.import.xml]
    DataModel
    Messages
    Overview
    Process Tracing [E2E Process Tracing.xml]
    Services
        Classes
        CustomerServicesPort
        FileDownloadPort
            +customer_list.csv( outputContent : Base Types::Blob, o
                out outputContent : Base Types::Blob
                out outputHeaderParameters : Base Types::Array
                in getParameters : Base Types::Array
            Export as CVS( outputContent : Blob, outputHeaderParar
    UI
    E2E Bridge Profile [E2E Bridge Profile.xml]
```

The download operation reads the table parameters from the HTTP parameters and calls the **getCustomers** operation accordingly:


activity Activity Diagram Export as CVS [Export as CVS]
Get the table parameters from the HTTP call.
«E2EArray»
getParameters : Array
{arrayElement = Parameter}
«ActionScript»
Get Parameters from Path
«centralBuffer»
parameters : GetCustomers
Get the data according to the parameters.
getCustomers
(CustomerServicesPort::)
«centralBuffer»
customerList : GetCustomersResponse
«ActionScript»
Map Customer List to File
«FlatFile»
customer_file : CustomerFile
«FlatFileAdapter»
Compose File
{action = compose}
data : Blob
outputContent : Blob
Set the output headers so that the browser recognizes the data as CSV.
«ActionScript»
Set Output Header
«centralBuffer»
header : HeaderField
«E2EArray»
outputHeaderParameters : Array

```
{script = "create parameters;

set parameters.displayLength = 5;
set parameters.displayStart = 0;
set parameters.search = "";

create local parameter using typeOf(getParameters[0]);

set parameter = select first from getParameters where element.name = "displayLength" ;
set parameters.displayLength = parameter.value.convertToInteger() if parameter.exists();

set parameter = select first from getParameters where element.name = "displayStart" ;
set parameters.displayStart = parameter.value.convertToInteger() if parameter.exists();

set parameter = select first from getParameters where element.name = "search" ;
set parameters.search = parameter.value if parameter.exists();

create local array parameterArray using typeOf(getParameters[0]);
set parameterArray = select each from getParameters where element.name = "sortColumn" ;
set parameters.sortColumn = apply element.value.convertToInteger() to parameterArray;

set parameterArray = select each from getParameters where element.name = "sortDirection" ;
set parameters.sortDirection = apply element.value to parameterArray;"}
```

```
{script = "create customer_file;
set customer_file.CustomerFile_Record =
customerList.customerElement;"}
```

```
{script = "create outputHeaderParameters;
create header;

set header.name = "Content-Type";
set header.value = "text/csv; charset=UTF-8";
append header to outputHeaderParameters;"}
```

After having read the data, the file is composed and returned.