DOCUMENTS 4

GADGETS

Setup / Developer's Guide

GADGETS 1.0

Table of Contents

1. Introduction

This document describes installation, creation and use of *gadgets*.

Gadgets are an extension for DOCUMENTS 4 providing the option to integrate individual applications in different places in DOCUMENTS 4.

Placing multiple gadgets on so-called *dashboards* is also possible. Dashboards will then be used as overview pages for gadgets which users themselves can add, sort or delete.

Gadgets are based on *portal scripts*. For technical gadget functionality, therefore, the entire *portal script API* is available.

The *Gadget API* allows creating gadgets with predefined functions as overview pages, forms, tables, and diagrams.

2. Installation / Setup

2.1 Requirements

The use of *gadgets* requires DOCUMENTS 4.0a or a more current version. Moreover, the license must permit the use of the gadget API.

Gadget implementation requires PortalScript API knowledge. The use of the dashboard functionality requires Internet Explorer Version 8/9, or Firefox version 10 or higher.

2.2 Importing the gadget API

The gadget API comes with an archive with the name `Gadgets_x.x.x.zip`, where "`x.x.x`" represents the respective version number. The content of the archive is structured as follows:

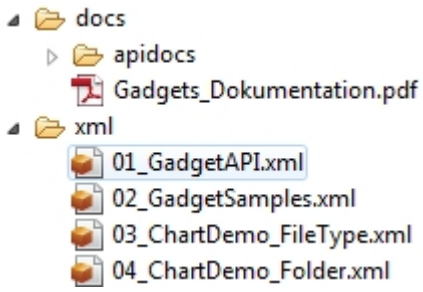

Fig. 1 Structure of the gadget package

To install *gadgets* in DOCUMENTS, you need to import the `01_GadgetAPI.xml` file into DOCUMENTS. The import is performed in the DOCUMENTS Manager via the *Server settings -> XML import* menu item. The gadget API is ready for use after import.

2.2.1 Importing the gadget samples

If you also want to install the gadget samples, you will have to import the following files (in this order):

- `02_GadgetSamples.xml`
- `03_ChartDemo_FileType.xml`
- `04_ChartDemo_Folder.xml`

For demonstration purposes, the "`GadgetDemo_InitPeachit`" script comes with the gadget samples. When this script is executed, some of the samples will be meaningfully integrated into the demo client *Peachit*. The following samples can be found in DOCUMENTS after execution:

- "Alternative overview" directly in tree view
- "Invoice overview" directly in tree view
- "Chart demo" directly in tree view
- "Last Used" in each Peachit sample file
- "Assigned employees" in each DOCUMENTS file of the "ftEmployee" type
- "Business card tab" in each DOCUMENTS file of the "ftEmployee" type
- "Quickorder" in each DOCUMENTS file of the "ftOrder" type

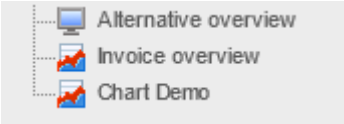

Fig. 2 The samples in tree view

3. Getting Started

3.1 What are gadgets?

Gadgets individually and principal-independently extend DOCUMENTS functionality. *Gadgets* can be integrated in different places within DOCUMENTS . They can interact both with other gadgets and, via a client-side API, with DOCUMENTS.

Because gadgets are implemented in portal scripts, the complete portal script API is available when using gadgets.

Additionally, gadgets will remain compatible in version updates.

3.2 What gadgets are not

Gadgets are not replacements of already existing DOCUMENTS program functions. Gadgets cannot be used to sidestep or modify permissions in DOCUMENTS.

Gadgets are not designed for complex and extensive applications because these are implemented in portal scripts and execution may involve a lot of performance.

3.3 "Hello Gadget"

This section describes how to create and integrate a simple gadget, using examples.

3.3.1 Creating a gadget

To create a gadget, you first have to store the server-side source text of the gadget in a portal script. A new portal script is created in the DOCUMENTS Manager under the tree entry "*Documents -> Scripting*".

The allocated script name will be required later to integrate the gadget. The name "`GadgetSample_HelloGadget`" is used in this example (see Fig. 3).

> ***Important note:***
>
> *The name of a portal script to be executed as a gadget must always start with "`Gadget`"!*

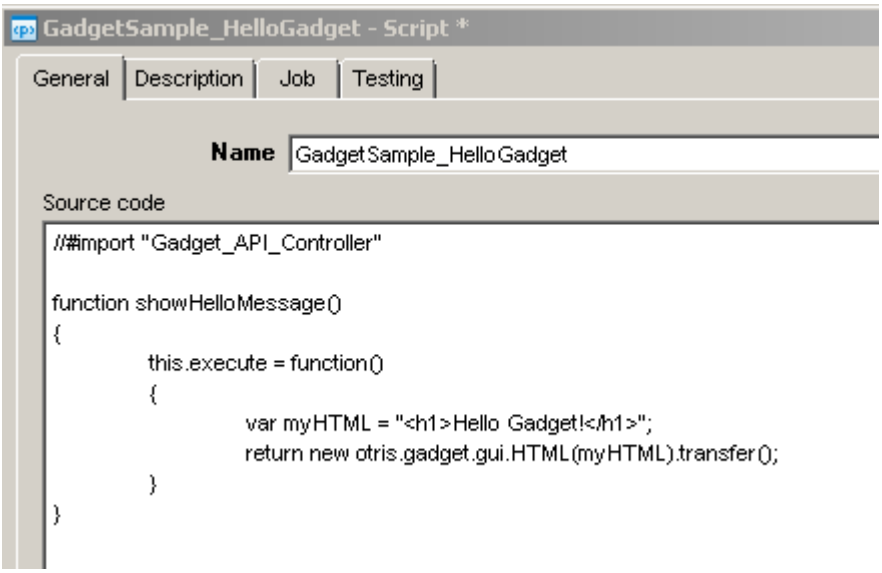

Fig. 3 Creating gadgets

The "Hello Gadget" example uses the following source code:

```
#import "Gadget_API_Controller"
function showHelloMessage()
{
  this.execute = function()
  {
      var myHTML = "<h1>Hello Gadget!</h1>";
      return new otris.gadget.gui.HTML(myHTML).transfer();
  }
}
```

This gadget script contains the import statement of the gadget API, followed by the gadget action `showHelloMessage()`.

This action returns the result of an HTML element with the heading "Hello Gadget!" and displays it. The gadget will have been successfully created after saving.

3.3.2 Deploying the gadget

A gadget stored in a portal script can be integrated in different places into DOCUMENTS. To do this, the name of the gadget script (in this example: `GadgetSample_HelloGadget`) and the name of the gadget action (in this example: "`showHelloMessage`") must be known. Together, this information makes up gadget configuration (see chapter 4.2: Gadget configuration).

To make gadgets accessible, for example, via an entry in the DOCUMENTS interface tree, you need to create a new public folder first.

This is performed in the DOCUMENTS Manager via the "*Documents -> Public folders -> New*" entry.

To assign the public folder a gadget, you need to create a new property named "gadgetConfig" with the following value on the "Properties" tab of the folder (see Fig. 4):

```
{gadgetScript:'GadgetSample_HelloGadget',gadgetAction:'showHelloMessage'}
```

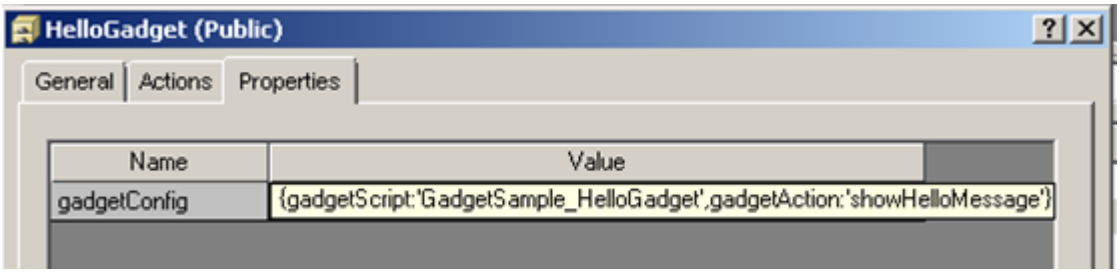

Fig. 4 Creating the gadget configuration

The created gadget can now be displayed via an entry in the tree. (See Fig. 5 and Fig. 6).

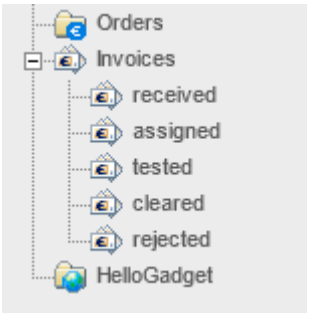

Fig. 7 "Hello Gadget" in tree view

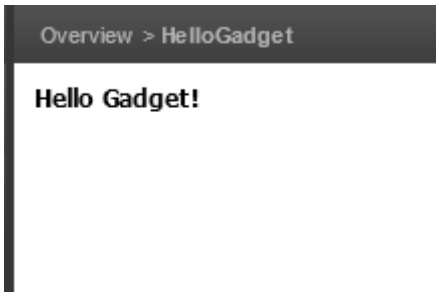

Fig. 6 Result of the gadget

Gadgets can be used in the following places within DOCUMENTS:

- In the overview
- In public folders (entry in the left menu tree)
- In the sidebar of DOCUMENTS files
- As tabs of a DOCUMENTS file
- As file field (type: Gadget)
- As user-defined action (in DOCUMENTS files, tabs and folders)
- As "user exit" (in a file field)
- Other integration options are under consideration

Chapter 6 "Integration von Gadgets" describes the different integration options.

4. Basic Concepts

4.1 Gadget structure

Gadgets are always composed of a *gadget script* and a *gadget action*, which is stored in that gadget script. Below we will explain the structure of gadgets using the "Hello Gadget" sample from chapter 3 Getting Started.

The gadget script always starts with importing the Gadget Controller API:

```
#import "Gadget_API_Controller"
```

This import statement ensures that the script is executable as a gadget scrip; it provides the functions of the gadget API.

The gadget action is a portal scripting function which must follow a specific structure so that it is executable as a gadget action:

```
function showHelloMessage()
{
  this.execute = function()
  {
      //Gadget Code
  }
}
```

A gadget script may contain any number of gadget actions and helper functions which can be used by the gadget actions.

4.2 Gadget configuration

In the entire gadget environment, the call of a specific gadget is always described with a "*gadget configuration*". This configuration is used to set the gadget call uniquely. A gadget configuration consists of:

gadgetId (optional):

A unique name that is assigned to the gadget. If you omit this parameter, a value will be automatically generated here. However, if you want to access the gadget at a later time (e.g. when communicating with other gadgets), you should allocate a name here to be able to address the gadget.

gadgetScript:

The name of the portal script containing the gadget action (`gadgetAction`) to be called.

gadgetAction:

The name of the gadget action to be called.

gadgetWidth:

The preferred width of the gadget in pixels. (This value may not be specified in quotation marks).

gadgetHeight:

The preferred height of the gadget in pixels. (This value may not be specified in quotation marks).

Parameters:

All other specified properties of the gadget configuration are automatically forwarded as parameters to the gadget. The gadget configuration is always created as an object in "JSON" format.

4.2.1 Example of a gadget configuration

```
{gadgetId: 'helloGadget', gadgetScript: 'GadgetSample_HelloGadget',
gadgetAction: 'showHelloMessage', message: 'Hello Gadget! '}
```

In this case, the "message" parameter is now accessible within the gadget context.

4.3 Gadget life cycle

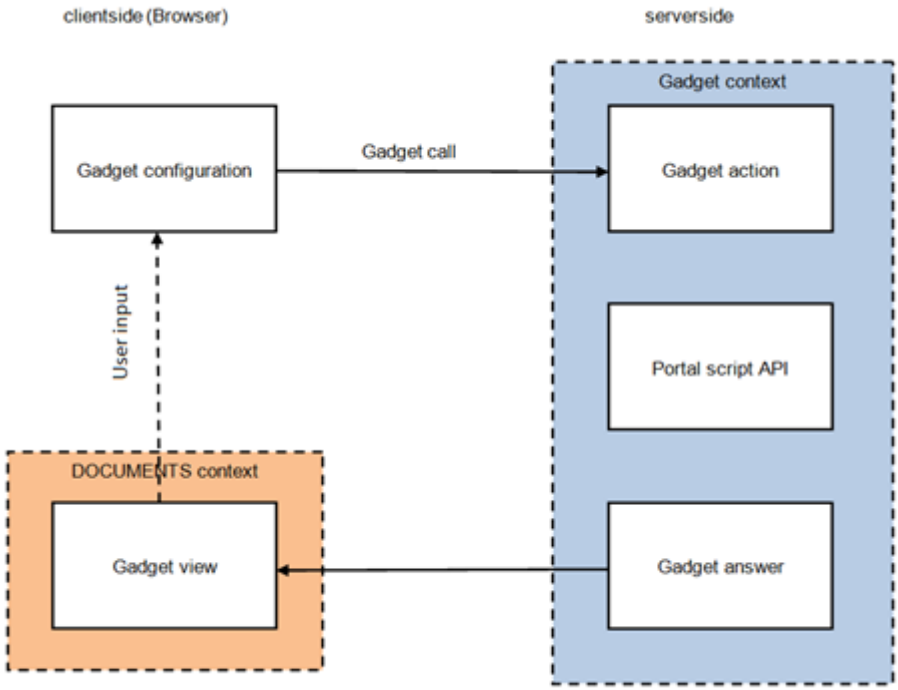

Fig. 8 Gadget life cycle

- Gadget execution always starts with setting the *gadget configuration.* (See section 4.2)
- Based on the *gadget configuration*, the gadget will then be actually called by the system.
- While executing the gadget, you can access information and functions of the *gadget context*. (See section 0)
- If the gadget has been executed and returns a response, this will be displayed. While the gadget is being displayed, the information and functions of the *DOCUMENTS context* are available (see section 4.6). The *gadget context* can no longer be reached here.
- The displayed gadget can now start another gadget call using, for example, `GadgetActionButtons`.

Fig. 7 provides an overview of a gadget life cycle.

4.4 Inter-gadget communication (Gadget ID)

As already described in section 4.2, you can specify a *gadget ID* within a gadget configuration. This unique name can be used to uniquely identify a displayed gadget.

When executing a gadget using a gadget ID that is already used by another gadget on the page, the display of the previously displayed gadget will be replaced with the new one.

In this way, a gadget may also replace the content of another gadget through a call. The following example allows modifying the color of the square in the other gadget by pressing the buttons in a gadget:

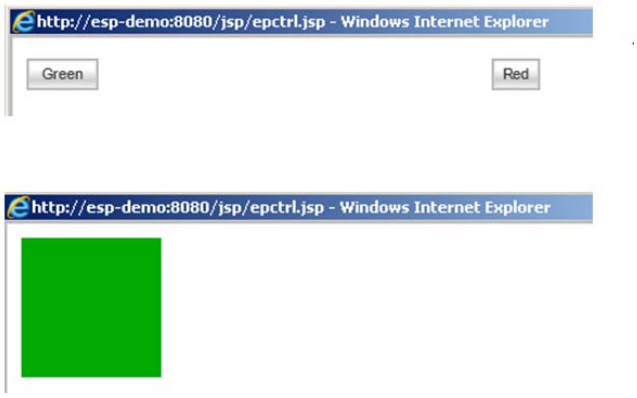

Fig. 9 Example of inter-gadget communication

```
#import "Gadget_API_Controller"


function showRemote()
{
  this.execute = function()
  {
      var form = new otris.gadget.gui.Form();
      form.addGadgetActionButton("buttonGreen", "Green",
                       "GadgetSample_RemoteGadget", "showColorGadget",
                       {color: '#00aa00'}, "colorGadget");


      form.addGadgetActionButton("buttonRed", "Red",
                      "GadgetSample_RemoteGadget", "showColorGadget",
                      {color: '#aa0000'}, "colorGadget").setInLine(true);
      return form.transfer();
  }
}


function showColorGadget()
{
  this.execute = function()
  {
      var color = gadgetContext.gadgetParams.color ?
                  gadgetContext.gadgetParams.color : '#00aa00';


      var htmlText = "<div style='width: 100px; height:\
                     100px; background-color: "+ color +";'>";


      var html = new otris.gadget.gui.HTML(htmlText);
      return html.transfer();
  }
}
```

To attain the view displayed in Fig. 8, the following two gadgets were configured for a file type:

```
{gadgetScript: 'GadgetSample_RemoteGadget', gadgetAction: 'showRemote'}
{gadgetScript: 'GadgetSample_RemoteGadget', gadgetAction:'showColorGadget',
gadgetId: 'colorGadget'}
```

Chapter 6 "Integration von Gadgets" explains how to configure multiple gadgets.

When executing a gadget on a page that can only contain a single gadget, each new gadget call will replace the currently displayed gadget.

4.5 The gadget context (server-side)

The gadget context is an implicit object (`gadgetContext`), which is available **while executing** a gadget. It contains all important information about the executed gadget and the parameters passed during the call.

4.5.1 Context information

Information on the gadget itself and the context in which it has been executed is directly accessible in the `gadgetContext`. The most important information is:

- `gadgetContext.gadgetId`: Gadget ID of the current gadget.
- `gadgetContext.gadgetScript`: The gadget script of the current gadget.
- `gadgetContext.gadgetAction`: The gadget action of the current gadget.
- `gadgetContext.folderId`: The ID of the folder in which the gadget is executed.
- `gadgetContext.fileId`: The ID of the DOCUMENTS file in which the gadget is executed.
- `gadgetContext.registerId`: The ID of the tab in which the gadget is executed (if the gadget is executed in a tab other than the file cover).
- `gadgetContext.fieldId`: The ID of the field in which the gadget is executed.
- `gadgetContext.documentId`: The ID of the document being displayed when the gadget is executed on a document tab.

4.5.2 Gadget parameters

Gadget parameters are parameters which are passed via the gadget configuration directly on calling a gadget. These can be accessed via the `gadgetContext-gadgetParams` object.

If, for instance, a parameter named "name" is passed during gadget configuration, this can be retrieved as follows:

```
get and output //parameter "name"
util.out(gadgetContext.gadgetParams.name);
```

4.5.3 Form parameter

If a gadget contains forms, the parameters of the forms will be passed on to the called gadget when pressing an Action button within that form. If the form contains, for example, a parameter named "company", this can be retrieved as follows:

```
get and output //parameter "company"
util.out(gadgetContext.formParams.company);
```

4.5.4 Ext parameter

Specific gadget elements that use the components of the *Ext JS Framework* automatically call a separate gadget to retrieve their data. This gadget is set for these elements via the "`setDataScript`" and "`setDataAction`" functions. The respective component then indicates to the gadget via the Ext parameters which data is to be retrieved. These parameters are:

- `start`: Number of first record to be retrieved.
- `limit`: Number of records to be retrieved.
- `sort` : Name of data field by which the results are to be sorted.
- `searchExpression`: Search term by which to filter the results.

You can get the Ext parameters as follows:

```
get and output //parameter "start"
util.out(gadgetContext.extParams.start);
```

4.6 The DOCUMENTS context (client-side)

The *DOCUMENTS context* is an implicit object (`documentsContext`) which is available on the client-side **while displaying** a gadget in the browser. It contains all important functions to enable executing actions in the DOCUMENTS environment. This can be, for example, displaying or editing a DOCUMENTS file.

For a detailed description of the DOCUMENTS context functions, see the API documentation in the section "DOCUMENTS context".

5. API Overview

This chapter gives an overview of the elements contained in the gadget API, and their use. The gadget package includes complete API documentation.

5.1 General

Basically, gadgets can return any element as a result which has a *transfer()* method from the gadget API. The returned element is the result of the gadget and is displayed in the DOCUMENTS interface.

5.2 HTML elements

The "`otris.gadget.gui.HTML`" element can be used to display any HTML text.

```
var htmltext = '<h1>Hello!,</h1>\
        <p>Sie verwenden gerade das HTML Gadget.</p>\
        <p>Und hier noch ein zweiter Absatz.</p>';

var htmlerg = new otris.gadget.gui.HTML(htmltext);
return htmlerg.transfer();
```

If data to be displayed in HTML is already available, this data can be simply inserted into an HTML structure using the "`compileTemplate()`" method. To do this, you set specific *markers* in the HTML text which will then be replaced by the content of the data.

```
var person = {
             first name: 'Willi',
             name: 'Schreiber',
      };

var htmltext = '<h1>Hello {first name} {name},</h1>\
        <p>Sie verwenden gerade das HTML Gadget.</p>\
        <p>Und hier noch ein zweiter Absatz.</p>';

var htmlresult = new otris.gadget.gui.HTML(htmltext);
htmlresult.compileTemplate(person);
return htmlresult.transfer();
```

The markers are enclosed in curly braces {}; they contain the respective name of the variable in the object used. The result contains the data from the "`person`" object.

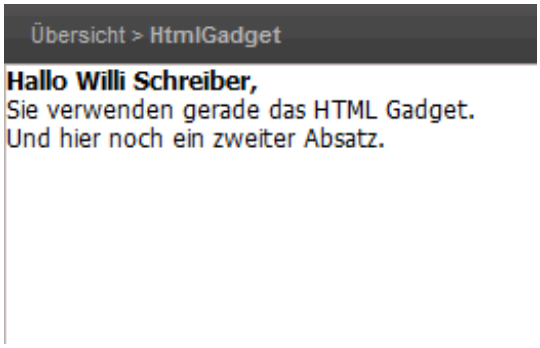

Fig. 10 Result of the HTML gadget

In addition, the HTML element provides the option to deploy functions from the gadget script to execute the gadget. The HTML element is made aware of such functions using the "`addClientFunction (clientFunction`" method.

```
//#import "Gadget_API_Controller"
function showHtmlText()
{
  this.execute = function()
  {
      var htmltext = '<h1>Hello!,</h1>\
      <p>Sie verwenden gerade das HTML Gadget.</p> \
      <p><a href=\"#\" onClick=\"eineFunktion()\">Ausführen!</a>';
      var htmlresult = new otris.gadget.gui.HTML(htmltext);
      htmlresult.addClientFunction(eineFunktion);
      return htmlresult.transfer();
  }
}
function eineFunktion(){
  alert("eine Funktion wurde ausgeführt");
}
```

If you click "Execute!" in the gadget, the function will be executed.

> ***IMPORTANT NOTE:***
>
> *Functions called in this manner are executed on the client side within the DOCUMENTS context in the browser. This means that the "gadgetContext" object and all information that it contains are **not** available in such functions.*

5.3 Messages and error messages

Gadgets can display messages and error messages. For this purpose, the "`otris.gadget.gui.Message`" class is returned as the result of the gadget.

The following message types are supported:

- *error*: Error message
- *warning*: Warning
- *info*: Any information

Depending on the message type and browser, different icons or notes are displayed.

```
//#import "Gadget_API_Controller"

function showMessageGadget()
{
  this.execute = function()
  {
      if(gadgetContext.gadgetEvent == 'klickmich')
      {
              return new otris.gadget.gui.Message('Die Schaltfläche\
 wurde geklickt!', 'info').transfer();
      }
      else
      {
              myForm = new otris.gadget.gui.Form();
              myForm.addHeadLine('Bitte klicken Sie auf die Schaltfläche:');
              myForm.addGadgetActionButton('klickmich', 'Klick mich!');

              return myForm.transfer();
      }
  }
}
```

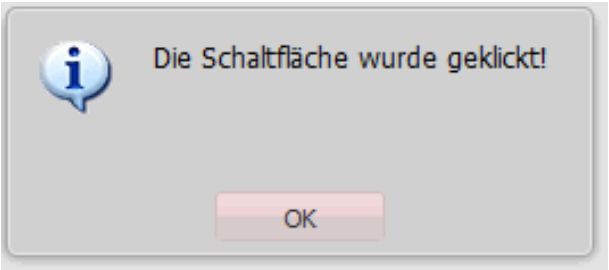

Fig. 11 Displayed message (type: Info)

5.4 Forms and GadgetActionButtons

Gadgets may display forms and send the entered data to any gadget using *GadgetActionButtons*. The "`otris.gadget.gui.style`" is used to create forms. This class contains a variety of functions that can be used to add form elements.

GadgetActionButtons are buttons that can be inserted into forms and that will start a gadget call when pressed. If you press a GadgetActionButton in a form, the contents of the form elements will be automatically sent to the called gadget.

In the following example, a simple form is used to generate a personal salutation:

```
function(){ showFormGadget()
{
  this.execute = function()
  {
      if(gadgetContext.gadgetEvent == 'send')
      {
              var htmlText = "Hello ";

              if(gadgetContext.formParams.geschlecht == "male"){
                     htmlText += "Herr ";
              }
              else if (gadgetContext.formParams.geschlecht == "female"){
                     htmlText += "Frau ";
              }

              htmlText = "<h1>" + htmlText
              + gadgetContext.formParams.firstname + " "
              + gadgetContext.formParams.name + "</h1>";

              var html = new otris.gadget.gui.HTML(htmlText, 'testform');
              return html.transfer();
      }
      else
      {
              myForm = new otris.gadget.gui.Form();
              myForm.addTextField('name', "Name");
              myForm.addTextField('firstname', "Vorname");
              var options = [['male','männlich'],['female','weiblich']];
              myForm.addSingleSelectList('geschlecht',
                             "Geschlecht", options);
              myForm.addGadgetActionButton('send', 'absenden');

              return myForm.transfer();
      }
  }
}
```

If the gadget is displayed for the first time, the `gadgetContext.gadgetEvent` parameter has not yet been set and a simple form will be displayed which contains a GadgetActionButton (see Fig. 11).

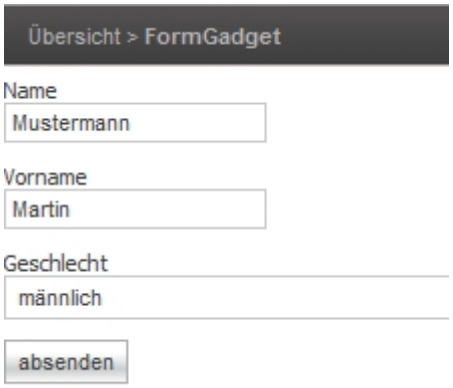

Fig. 12 Displaying the form gadget

When you press the GadgetActionButton, the gadget is restarted, but this time using the `gadgetContext.gadgetEvent` parameter, which now contains the name of the GadgetActionButton.

The salutation is displayed (See Fig. 12).

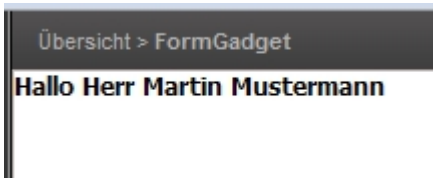

Fig. 13 Result of the form gadget

5.5 Tables and the External JS Framework

The *Ext JS Framework* from Sencha Inc. (www.sencha.com) is used to integrate more complex display elements such as tables and diagrams into gadgets.

The `otris.gadget.gui.ExtComponent` class facilitates configuring the bulk of elements from Ext JS and use them in gadgets.

For the most important framework elements, Gadget API additionally provides, however, separate components that significantly simplify the use of tables and charts in the gadget API. The following gadget generates a simple table:

```
//#import "Gadget_API_Controller"

function showTable()
{
  this.execute = function()
  {
      var table = new otris.gadget.gui.ExtTable();
      table.setHeader(["id","Name", "Vorname"]);
      table.setDataIndex(["id","name","firstname"]);

      var data = new Array();
      data.push({id: 1, name: 'Schreiber', firstname: 'Willi'});
      data.push({id: 2, name: 'Oppen', firstname: 'Bernhard'});
      data.push({id: 3, name: 'Schlepp', firstname: 'Stefan'});

      table.setData(data);
      table.setHideFirstColumn(true);

      return table.transfer();
  }
}
```

When executing the above gadget, the table shown in Fig. 13 will be displayed.

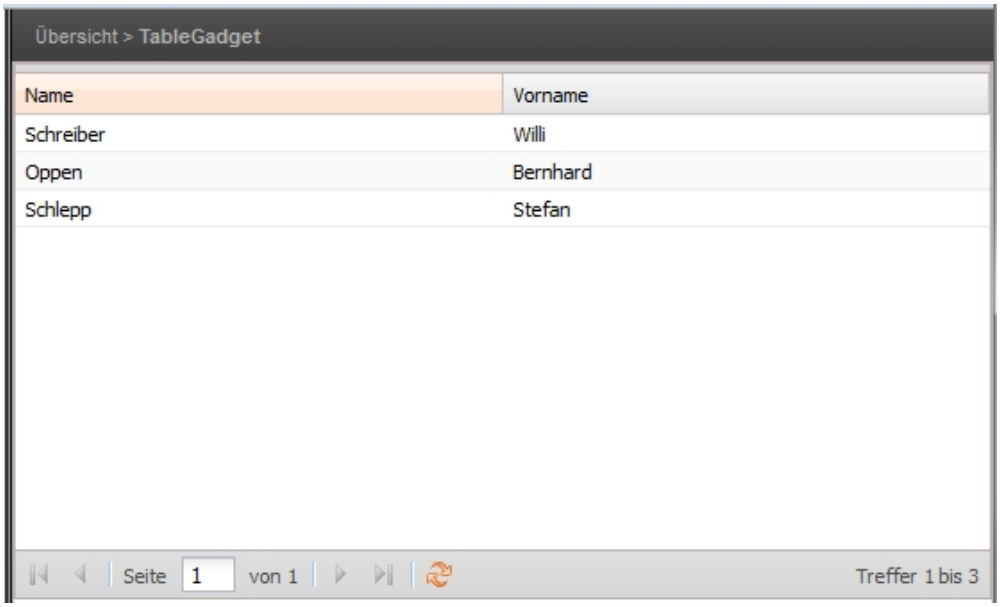

Fig. 14 A simple table

In this case, the data was entered directly for the table. To use the complete capabilities of the tables, the data can also be obtained from another gadget call.

Section 7.1 Das „LastUsed-Gadget“ contains an in-depth description of the use of tables, explaining it using an extended example.

6. Integrating Gadgets

6.1 General

When integrating gadgets into DOCUMENTS, the gadget configurations are entered in different places as properties. Normally, you enter an individual gadget configuration as follows:

```
{gadgetScript:'GadgetSample_HelloGadget', gadgetAction:'showHelloMessage'}
```

The following properties also support specifying multiple gadgets:

- *marginGadgetConfigs* (See section 6.3 Gadgets in the sidebar of DOCUMENTS files)
- *dashboardGadgetConfigs*
- *overviewGadgetConfigs*
- *overviewDashboardGadgetConfigs*

For these properties, you always need to use the following format:

```
[{gadgetScript:'GadgetSample_HelloGadget',gadgetAction:'showHelloMessage'},
{gadgetScript: 'GadgetSample_HtmlGadget', gadgetAction: 'showHtmlText'}]
```

If only a single gadget is to be used within such a property, this must be enclosed in square brackets anyway:

```
[{gadgetScript:'GadgetSample_HelloGadget',gadgetAction:'showHelloMessage'}]
```

6.2 Gadgets for public folders

To use a gadget for a public folder, the gadget configuration must be entered as property "`gadgetConfig`" for the folder (see Fig. 14).

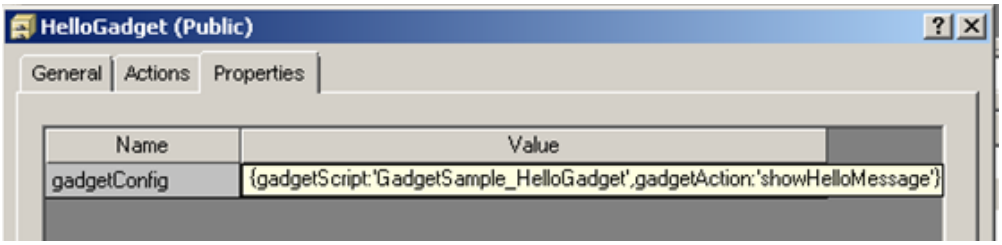

Fig. 15 Gadget configuration for the public folder

The gadget configured here then fills the entire space of the main scope of DOCUMENTS (see Fig. 15).

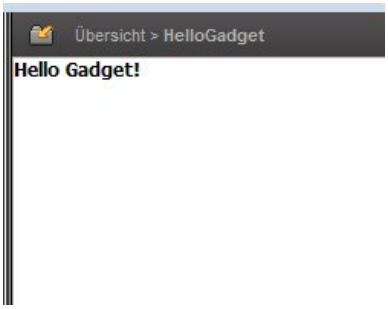

Fig. 16 Gadget for folder

6.3 Gadgets in the sidebar of DOCUMENTS files

Gadgets can also be used in the sidebar of DOCUMENTS files. For this purpose, the "`marginGadgetConfigs`" property must be set for the respective file type. Multiple gadgets can be used in this property; these are displayed vertically in the sidebar.

The following gadget configuration was used for the example below (see Fig. 16):

```
[{gadgetScript:'GadgetSample_HelloGadget',gadgetAction:'showHelloMessage'},
{gadgetScript: 'GadgetSample_HtmlGadget', gadgetAction: 'showHtmlText'}]
```

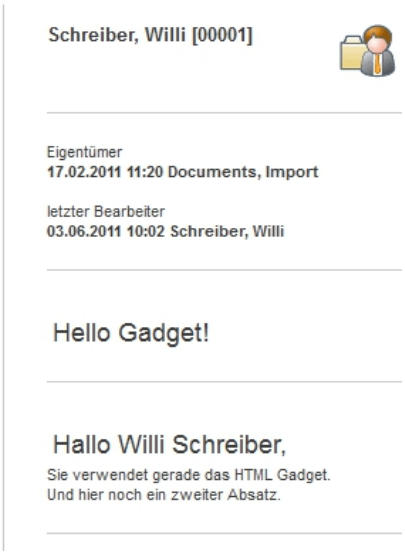

Fig. 17 Two gadgets in the sidebar

6.4 Gadgets in DOCUMENTS file tabs

Gadgets can also be used on DOCUMENTS file tabs. For this purpose, a tab of the "External call" type is added to the file type (see Fig. 17).

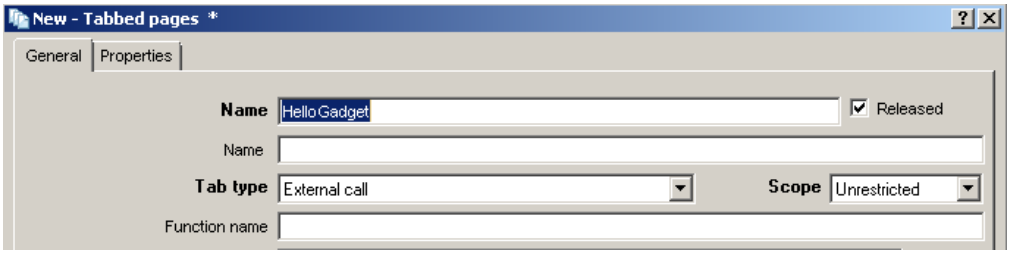

Fig. 18 Creating a new tab

The gadget configuration is then entered in the "`gadgetConfig`" property of the "Properties" tab (see Fig. 18).

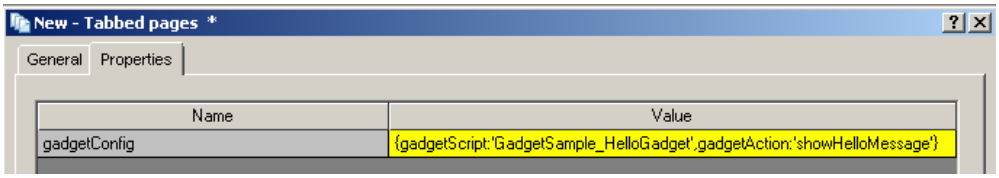

Fig. 19 Gadget configuration on tab

> *Important note:*
>
> *When the new gadget tab is to be also used in already existing DOCUMENTS files of the file type, these must be customized after creating the tab via the "Change files" button.*

The configured gadget will then be displayed on selecting the created tab (see Fig. 19).

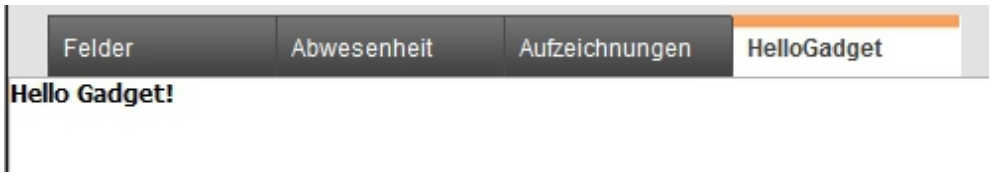

Fig. 20 Gadget on a tab

6.5 Gadgets as field of a DOCUMENTS file

Moreover, gadgets be used as a field within a DOCUMENTS file. To do this, a new field of the "Gadget" type is created in a file type (see Fig. 20). On the "Properties" tab, you can then enter a gadget configuration in the "`gadgetConfig`" property.

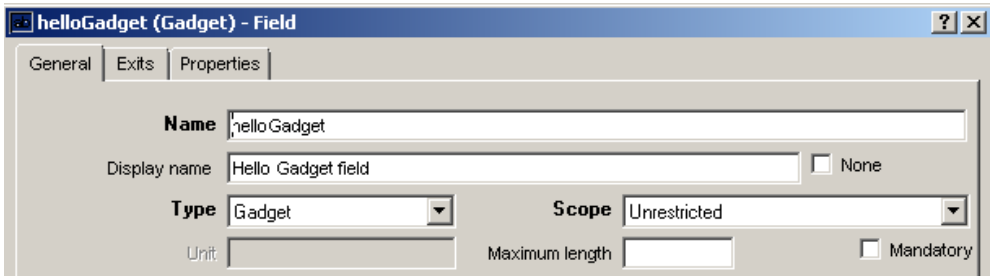

Fig. 21 File field of the Gadget type

The gadget set in "`gadgetConfig`" is then integrated as a field into file view. (See Fig. 21)

Fig. 22 Gadget as a file field

If the gadget is to save values in the DOCUMENTS file, an additional, hidden field will be required in which the gadget can save contents via the DOCUMENTS context.

6.6 Gadgets as a user-defined action

Gadgets can be executed as a *user-defined action*. To do this, you need to add a new user-defined action to the respective file type (see Fig. 22). "Gadget" is selected as the type. On the "Properties" tab, you can then enter a gadget configuration in the "`gadgetConfig`" property.

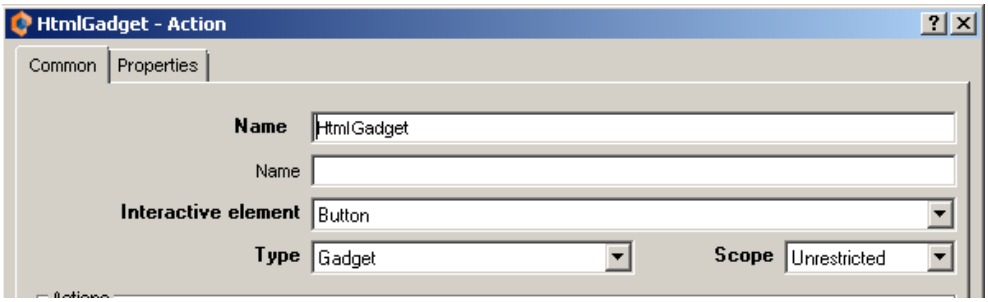

Fig. 23 User-defined action (gadget)

When executing the action, the gadget is opened in a new window (see Fig. 23).

Fig. 24 Gadget in new window

6.7 Gadgets as a user exit

Gadgets can be configured as a *user exit* for fields of file types. For this purpose, the value "Action button next to field" is selected for the "Type" field in the properties dialog of the field on the "Exits" tab. (See Fig. 24)

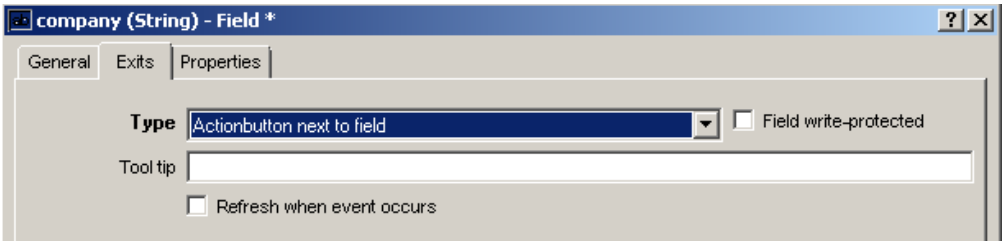

Fig. 25 Setting up a user exit

On the "Properties" tab, you can then enter a gadget configuration in the "`gadgetConfig`" property.

Next to the respective field, an action button then appears (see Fig. 25) via which the gadget is started in a new window. The respective field value can then be customized from the gadget, e.g. via the DOCUMENTS context.

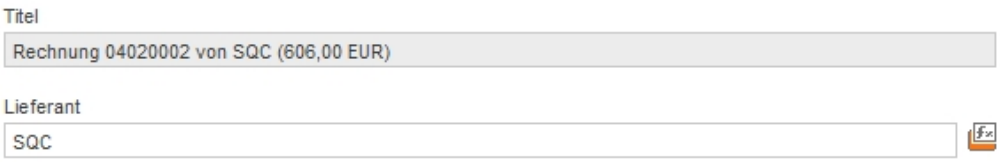

Fig. 26 Action button next to a file field

6.8 Gadgets as e-mail exit

Gadgets can be configured as e-mail exit on sending a DOCUMENTS file of a specific file type. To do this, a gadget configuration is entered in the "`emailGadgetConfig`" property in the file type's properties dialog.

When sending DOCUMENTS files of this file type via the "Send e-mail" action, a function icon will respectively appear next to the fields for address input via which the configured gadget can be started (see Fig. 26).

Fig. 27 Function icons next to address fields

When clicking the icon, the gadget will be executed in a separate window (see Fig. 27).

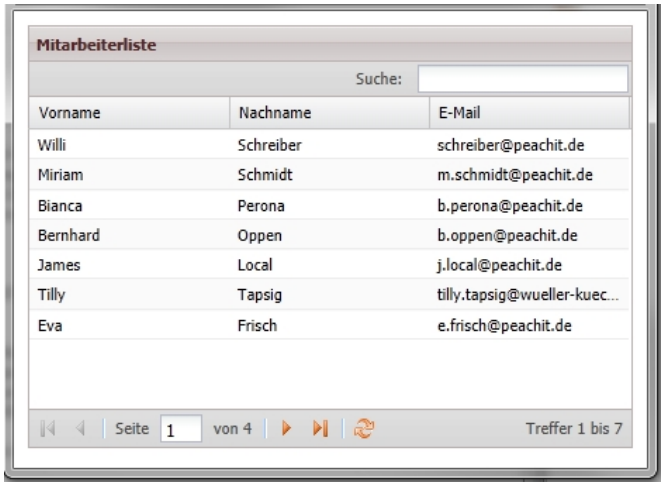

Fig. 28 Gadget as e-mail exit

6.9 Integrating dashboards

Dashboards allow displaying multiple gadgets on a single page. The individual gadgets are displayed in windows that can be arranged and sorted independently by the user.

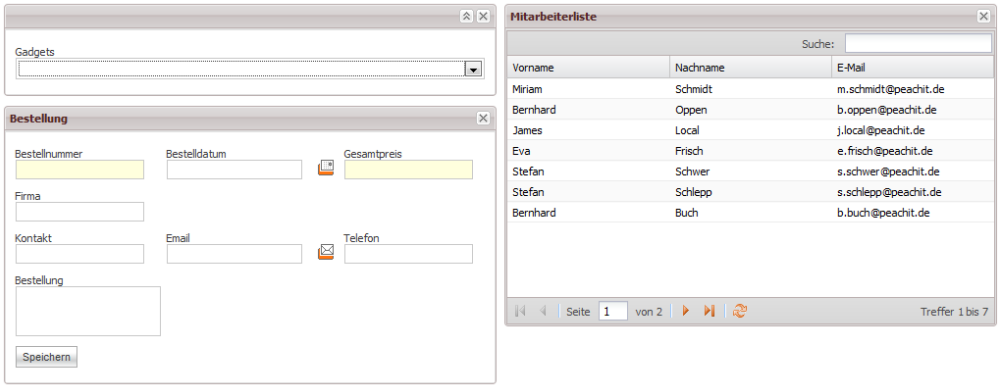

Fig. 29 Example: Dashboard

Dashboards can be used in the following places:

- As an alternative overview page
- For public folders
- In file tabs

If a dashboard is to be integrated into one of these places instead of an individual gadget, you will have to use the "`dashboardGadgetConfigs`" property instead of the "`gadgetConfig`" property. Any number of gadgets to be started on displaying the dashboard can then be specified in this property.

Additionally, an individual gadget that can be used to add other gadgets to the dashboard can be configured in the "`dashboardControl`" property. This gadget must contain a form with a "`gadgetActionList`" element. This provides the gadget configurations for those gadgets that can be added.

Such a gadget could look like this, for example:

```
#import "Gadget_API_Controller"

function getGadgetContainerActionList()
{
  this.execute = function()
  {
      var myForm = new otris.gadget.gui.Form();

      var gadgetConfigs = {
             'helloGadget': {label: 'Hello Gadget',
                             gadgetScript: 'GadgetSample_HelloGadget',
                             gadgetAction: 'showHelloMessage'},

             'htmlGadget': {label: 'Html Gadget',
                            gadgetScript: 'GadgetSample_HtmlGadget',
                            gadgetAction: 'showHtmlText'},

             'messageGadget': {label: 'Message Gadget',
                               gadgetScript: 'GadgetSample_MessageGadget',
                               gadgetAction: 'showMessageGadget'},

             'tableGadget': {label: 'Table Gadget',
                             gadgetScript: 'GadgetSample_TableGadget',
                             gadgetAction: 'showTable'},
      }
      myForm.addGadgetActionList('', 'Gadgets', gadgetConfigs);
      return myForm.transfer();
  }
}
```

The following gadget configuration for a public folder, for example, allows creating a simple dashboard:

```
{gadgetId: 'control', gadgetScript: 'GadgetSample_DashboardControl',
gadgetAction: 'getGadgetContainerActionList'}
```

The dashboard contains a gadget which provides a list including gadgets that can be added.

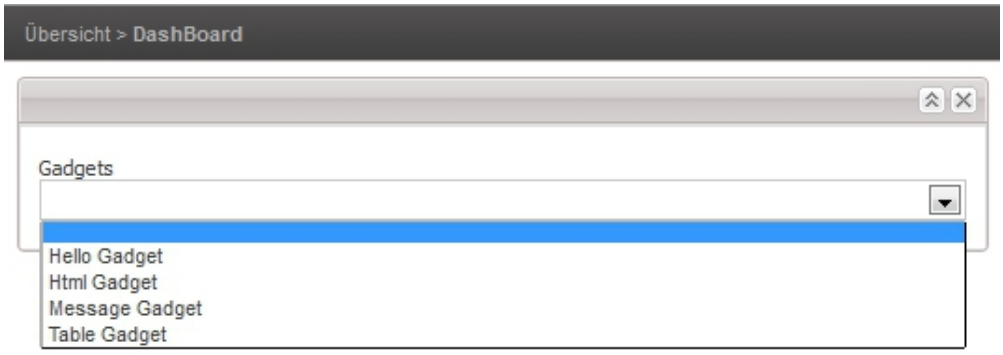

Fig. 30 A gadget as dashboard control

6.10 Gadgets on the overview page

To place gadgets on the *overview page*, the gadget configurations must be entered in the global property "`overviewGadgetConfigs`". Global properties are entered on the "*Documents -> Settings*" menu item (Properties tab). The gadgets are then displayed, by default, at the bottom of the overview page. Here you should consider the note on specifying multiple gadgets in section 6.

If the gadgets are to be displayed at the top of the overview page, the global property "`overviewGadgetPosition`" can be set to "top".

7. Samples

This section provides an in-depth description of complete gadget samples which resolve a respective issue.

7.1 The "LastUsed gadget"

> *The complete program source text of the following example can be found in the* **GadgetSample_LastUsedGadget.js** *file.*

The "LastUsed gadget" is to display a single column table that displays a list of recently displayed documents of the currently logged-in user. Double-clicking an entry is to open the corresponding document.

7.1.1 Implementing the gadget

Two gadget actions are required for the LastUsed gadget. The `getLastUsedData` action collects the data for the table and returns it. The `showLastUsed` action displays the table.

The `showLastUsed`

```
3:  function showLastUsed()
4:  {
5:    this.execute = function()
6:    {
7:      var table = new otris.gadget.gui.ExtTable('Last used');
8:      table.setHeader(['Id','Titel']);
9:      table.setDataIndex(['id','fileTitle']);
10:      table.setDataGadgetConfig({
11:        gadgetScript: 'GadgetSample_LastUsedGadget',
12:        gadgetAction: 'getLastUsedData'
13:      });
14:      table.setHideFirstColumn(true);
15:
16:      table.addHandler('itemdblclick',function(grid, record){
17:        documentsContext.showFileView(record.data.id,'');
18:      });
19:
20:      table.setDisableSearch(true);
21:      table.setShowTopToolbar(false);
22:      table.setShowBottomToolbar(false);
23:      return table.transfer();
24:    }
```

```
25:  }
```

A new table object is created in Line 7, and given the title "Last used".

```
7:      var table = new otris.gadget.gui.ExtTable('Last used');
8:      table.setHeader(['Id','Titel']);
9:      table.setDataIndex(['id','fileTitle']);
```

The column headings of the table are set in Line 8. The "`Id`" column will later include the *Id* of the respective document that appears on the list. Because this column will later be hidden, this heading will later be invisible. The `Title` column will contain the document's title.

The *Data index* (Line 9) describes for each column which data it is to contain from the data object. The data object is automatically retrieved by the table by calling a data gadget.

```
10:     table.setDataGadgetConfig({
11:       gadgetScript: 'GadgetSample_LastUsedGadget',
12:       gadgetAction: 'getLastUsedData'
13:     });
```

Which gadget is called for this purpose can be set via the table constructor or via the `setDataGadgetConfig` method (see Lines 10-13).

```
14:     table.setHideFirstColumn(true);
```

In Line 14, the first column of the table is hidden.

```
16:     table.addHandler('itemdblclick',function(grid, record){
17:       documentsContext.showFileView(record.data.id,'');
18:     });
```

A "handler" is added to the table in Lines 16-18. Handlers are functions that can be entered on tables for specific events. Because this is an element from the Ext JS Framework, the supported events should be taken from the Ext JS documentation of the `Ext.grid.Panel` object.

These functions are executed within the DOCUMENTS context. In this case, the function will be executed when double-clicking a table line and the document will be displayed with the respective ID.

```
20:     table.setDisableSearch(true);
21:     table.setShowTopToolbar(false);
22:     table.setShowBottomToolbar(false);
```

In Lines 20-22, the display elements of the table which are not required are hidden.

The `getLastUsedData` action

```
6
27:  function getLastUsedData()
28:  {
29:    this.execute = function()
30:    {
31:      var data = {};
32:
33:      var systemuser = context.getSystemUser();
34:      var lastusedfolder = systemuser.getPrivateFolder('lastused');
35:
36:      var myFRS = lastusedfolder.getFiles();
37:
38:      var data = otris.gadget.util.DataUtils.getTableData(myFRS,[]);
39:
40:      return data;
41:    }
42:  }
```

Initially, an empty object is created for the data (31). After that, the system user object of the currently logged-in user is retrieved (33). The user's "`lastused`" folder is queried in Line 34. A *FileResultSet* of all contained documents is requested in Line 36. The `getTableData` function call occurs in Line 38. This function converts an existing FileResultSet to a data object that can be used with a table. The respective fields of the documents are now available as columns within the object. For options to limit the amount of data and others, see Gadget API documentation.

7.2 The "Quickorder gadget"

> *The complete program source text of the example below can be found in the* **GadgetSample_QuickOrder.js** *file.*

The "*QuickOrder gadget*" is to display a form that enables creating a new order directly from the gadget, i.e. a new DOCUMENTS file of the "*ftOrder*" file type.

7.2.1 Implementing the gadget

Two gadget actions are required for the Quickorder gadget. The *showOrderFile* action shows a form with input fields to create an order. The *saveOrderFile* action creates a new order and saves the data collected within the form. The *showSmallOrderFile* action shows the same form in a reduced version, so the gadget can also be used in the sidebar of DOCUMENTS files; in this example, this action will not be discussed in detail though.

The `showOrderFile` action

```
3:  function showOrderFile()
4:  {
5:    this.execute = function()
6:    {
7:       var isUserAction = gadgetContext.gadgetParams.isUserAction;
8:       var form = new otris.gadget.gui.Form('order');
9:       form.setTitle("New File Order");
10:      var orderId = form.addTextField('orderId', 'Order number', '');
11:      orderId.setMandatory(true);
12:      var orderDate = form.addDateField('orderDate',
                                           'OrderDate', '');
13:      orderDate.setInLine(true);
14:      var total = form.addNumberField('total', 'Total price', '');
15:      total.setMandatory(true);
16:      total.setInLine(true);
17:      var company = form.addTextField('company', 'Firma', '');
18:      var contact = form.addTextField('contact', 'Kontakt', '');
19:      var email = form.addEMailField('email', 'Email', '');
20:      email.setInLine(true);
21:      var telephone = form.addTextField('telephone', 'Telefon', '');
22:      telephone.setInLine(true);
23:      var comment = form.addTextArea('comment', 'order', '');
24:
25:      var actionSave = form.addGadgetActionButton('actionSave',
26:                              'Save',
27:                              {gadgetScript: 'GadgetSample_QuickOrder',
28:                               gadgetAction: 'saveOrderFile',
29:                               isUserAction: isUserAction});
30:      actionSave.setInLine(false);
31:
32:      return form.transfer();
33:    }
34:  }
```

The "isUserAction" is queried from the gadget parameters in Line 7. This is to be set in the gadget configuration when the gadget is executed in a user-defined action. This information is important to close the window opened by the user-defined action again on sending the form.

```
7:       var isUserAction = gadgetContext.gadgetParams.isUserAction;
8:       var form = new otris.gadget.gui.Form('order');
9:       form.setTitle("New File Order");
```

A new form object is created in Lines 8 and 9. Input fields are added to the form in Lines 10-23. The procedure is explained once more using the example of the "total" field:

```
14:      var total = form.addNumberField('total', 'Total price', '');
15:      total.setMandatory(true);
16:      total.setInLine(true);
```

In Line 14, the field is added to the form using the "form.addNumberField" method. To ensure that the form elements can still be edited at a later time, the methods for adding form elements always return the newly added element. In Line 15, the field is highlighted as a mandatory field. The setInLine(true) method (Line 16) is used to display the form field in the same line as the previous form field.

```
25:      var actionSave = form.addGadgetActionButton('actionSave',
26:                              'Save',
27:                              {gadgetScript: 'GadgetSample_QuickOrder',
28:                               gadgetAction: 'saveOrderFile',
29:                               isUserAction: isUserAction});
```

A GadgetActionButton is added as the last form field in Line 25. This contains a name, a label and a gadget configuration that contains the gadget call to which the form is to be sent.

The complete form is then returned as the result of the gadget action in Line 32.

The saveOrderFile action

```
56:  function saveOrderFile()
57:  {
58:    this.execute = function()
59:    {
60:      var $FORM = gadgetContext.formParams;
61:      if($FORM.orderId && $FORM.orderId !=""
62:        && $FORM.total && $FORM.total != "") {
63:        var newFtOrder = context.createFile('ftOrder');
64:        newFtOrder.orderId = $FORM.orderId;
65:        newFtOrder.orderDate = $FORM.orderDate;
66:        newFtOrder.total = $FORM.total;
67:        newFtOrder.company = $FORM.company;
```

```
68:        newFtOrder.contact = $FORM.contact;
69:        newFtOrder.emailAddress = $FORM.email;
70:        newFtOrder.telephone = $FORM.telephone;
71:        newFtOrder.comment = $FORM.comment;
72:        newFtOrder.sync();
73:
74:        if(gadgetContext.gadgetEvent == "actionSaveAndShow")
75:        {
76:          var myForm = new otris.gadget.gui.Form();
77:          myForm.addHeadLine("Die Mappe wurde erfolgreich angelegt.");
78:          myForm.addGadgetActionButton("backToQuickOrder",
79:                         "Zurück zum Formular",
80:                         {gadgetScript: 'GadgetSample_QuickOrder',
81:                         gadgetAction: 'showSmallOrderFile'});
82:
83:          myForm.onGadgetLoad("function(){\
84:            if(!documentsContext.isEditMode()){\
85:            documentsContext.showFileEditView('" +
                     newFtOrder.getAutoText('%id%') + "');\
86:            }\
87:          }");
88:          return myForm.transfer();
89:        }
90:        else
91:        {
92:          var msg = new otris.gadget.gui.Message("Mappe wurde
                                                    erfolgreich angelegt!",
93:                                'info');
94:          if(gadgetContext.gadgetParams.isUserAction)
95:          {
96:             msg.onGadgetLoad(function(){window.close()});
97:          }
98:          return msg.transfer();
99:        }
100:      }
101:      else {
102:        if($FORM.orderId == "") {
103:          return new otris.gadget.gui.Message('orderId ist ein
                                                  Pflichtfeld',
104:                               'error').transfer();
105:        }
106:        if($FORM.total == "") {
107:          return new otris.gadget.gui.Message('Gesamtpreis ist ein
                                                   Pflichtfeld',
108:                               'error').transfer();
109:        }
110:      }
```

```
111:    }
112:  }
```

In Line 60, an "abbreviation" is initially created, i.e. a variable with the name "$FORM" that contains a reference to the "gadgetContext.formParams" variable, so that the full name of the variable need not be written at each access to the form parameters.

In Lines 61 and 62, a check is made on whether the mandatory fields of the form have been filled out.

```
60:      var $FORM = gadgetContext.formParams;
61:      if($FORM.orderId && $FORM.orderId !=""
62:        && $FORM.total && $FORM.total != "") {
```

If this is the case, a new DOCUMENTS file of the *ftOrder* type will be created in Line 63. The form values will then be transmitted to the newly created DOCUMENTS file from Line 64 on. In Line 72, the changes are carried over into the DOCUMENTS file.

```
63:        var newFtOrder = context.createFile('ftOrder');
64:        newFtOrder.orderId = $FORM.orderId;
65:        newFtOrder.orderDate = $FORM.orderDate;
66:        newFtOrder.total = $FORM.total;
67:        newFtOrder.company = $FORM.company;
68:        newFtOrder.contact = $FORM.contact;
69:        newFtOrder.emailAddress = $FORM.email;
70:        newFtOrder.telephone = $FORM.telephone;
71:        newFtOrder.comment = $FORM.comment;
72:        newFtOrder.sync();
```

Lines 74-89 only refer to the reduced version of the Quickorder gadget for the sidebar; here it is ensured that the newly created DOCUMENTS file is displayed after sending the form in DOCUMENTS.

If this is not the reduced variant, a message will be returned in Lines 92-98 as the result of the gadget action. If the gadget parameter "isUserAction" has been set during gadget configuration, the window in which the gadget resides will additionally close after displaying the message.

```
92:          var msg = new otris.gadget.gui.Message("Mappe wurde
                                                    erfolgreich angelegt!",
93:                                'info');
94:          if(gadgetContext.gadgetParams.isUserAction)
95:          {
96:             msg.onGadgetLoad(function(){window.close()});
97:          }
98:          return msg.transfer();
```

An error message will be returned in Lines 101-109 if one of the mandatory fields has not been filled out.

```
101:      else {
102:        if($FORM.orderId == "") {
103:          return new otris.gadget.gui.Message('orderId ist ein
                                                  Pflichtfeld',
104:                              'error').transfer();
105:        }
106:        if($FORM.total == "") {
107:          return new otris.gadget.gui.Message('Gesamtpreis ist ein
                                                   Pflichtfeld',
108:                              'error').transfer();
109:        }
```

8. Table of Figures