FastReport Mono Programmers Manual

Version 2022.3

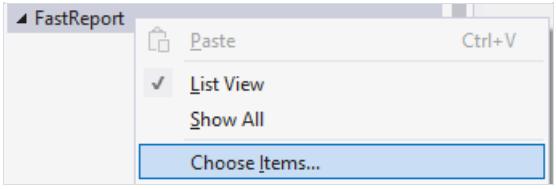
General information

Installing into VS Toolbox

Working with FastReport.Net* packages Toolbox is included in the package and should be automatically loaded by Visual Studio. Manual installation of Toolbox is also possible.

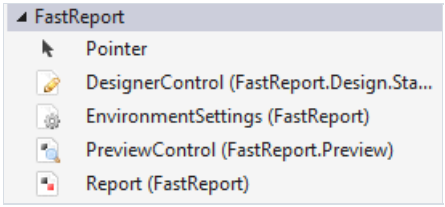
To add manually FastReport .NET components to the Visual Studio toolbox, you need to do the following steps:

- delete the "FastReport .NET" tab from the Toolbox, if it is there;
- create a new tab (to do this, right-click the Toolbox and select "Add Tab" item), or select an existing tab you would like to add FastReport components to;
- right-click on a tab and select "Choose Items...":



- in the dialog, press the "Browse..." button and choose FastReport.dll, FastReport.Web.dll files (they are located in the "C:\Program Files\FastReports\FastReport.Net" folder);
- close the dialog with OK button.

After this, you will see FastReport .NET components in a chosen tab:



- Report;
- PreviewControl;
- DesignerControl;
- EnvironmentSettings;
- WebReport (this component will be visible in ASP.NET project only).

Troubleshooting

If you face a problem when working with report designer (for example, some toolbars or tool windows are damaged), you should delete the configuration file. This file is created when you start FastReport .NET. It is located in the following folder:

Windows XP:

```
C:\Documents and Settings\user_name\Local Settings\Application Data\FastReport\FastReport.config
```

Windows Vista:

```
C:\Users\user_name\AppData\Local\FastReport\FastReport.config
```

The following information is stored in the config:

- sizes and locations of dialog windows;
- toolbar settings;
- recent used data connections;
- email settings (if you use the "Send Email" feature in the preview).

Deployment

You may redistribute the following files along with your application:

- FastReport.dll - the main FastReport .NET library;
- FastReport.Web.dll - the library that contains ASP.Net WebReport component;
- FastReport.Bars.dll - the toolbars and docking windows library;
- FastReport.Editor.dll - the code editor with syntax highlight. This library is not needed if you don't provide an end-user designer;
- FastReport.Compat.dll - the library for cross-platform compatibility;
- FastReport.DataVisualization.dll - the library for drawing charts;
- FastReport.xml - comments for classes, properties and methods used in FastReport. This file is used by the script editor, and also by the hint panels in the "Data" and "Properties" windows. It's not obligatory to distribute this file.

If your reports are stored in files, you have to deploy them as well.

FastReport NuGet packages:

- **FastReport.Core (demo on nuget.org)** - package containing the main logic of the program (obtaining the necessary data, rendering reports, exports, etc.). Some of the functionality from FastReport.NET is missing due to the cross-platform nature of this package.
- **FastReport.Net (demo on nuget.org)** - package with FastReport.dll library for .NET Framework 4.x, which is part of the 'Pro' and 'Demo' editions - for .NET Core 3.1, .NET 5 and .NET 6 exclusively for Windows. This package is available in several editions:
  - FastReport.NET.Demo - This package is available at nuget.org and in our official Trial installer, and it allows you to evaluate the product capabilities for various target frameworks on OS Windows (so-called FastReport.CoreWin). It has restrictions that are present in our other Demo edition products (5 page limit for exports, watermarked export pages, etc.).
  - FastReport.NET - This package is available in our official installer for licensed owners of the following editions: FastReport .NET WinForms and FastReport.Net Web. It includes FastReport .NET for .NET Framework 4.0 and higher.
  - FastReport.NET.Pro - This package is available in our official installer for licensed owners of the following editions: FastReport.NET Professional and Enterprise. It includes FastReport .NET for .NET Framework 4.0 or higher, FastReport.CoreWin for .NET Core 3.1 and for .NET 5.0 and higher.

- **FastReport.Web (demo on nuget.org)** - a package for integrating FastReport into scripts for working with web applications (rendering a report in a browser, exporting and printing from a browser, working with Online Designer) for ASP.NET Core. Includes components for Blazor Server and is used only with FastReport.Core.
- **FastReport.Core3.Web (demo on nuget.org)** - the same principle as FastReport.Web, but compatible with FastReport.CoreWin, which is included in the FastReport.Net.Demo/ FastReport.Net.Pro package.
- **FastReport.Localization (nuget.org)** - package with a set of FastReport localizations. Add it to your project if you need German or Russian localization, for example.
- **FastReport.Compat and FastReport.DataVisualization** - packages with basic logic (report compilation, MSChart support, etc.). You don't need to include them in your project, they are dependency packages.
- **FastReport.Data.*** - packages with connector plugins for FastReport to work with various databases, the connectors of which are not included in the source library. These packages are ◆common◆ for different FastReport editions and are suitable for both FastReport .NET and FastReport.Core and FastReport.CoreWin. Limitations: FastReport version 2021.4.0+ and NuGet Client 3.4.4+ are required.
  - FastReport.Data.ClickHouse

- FastReport.Data.Couchbase
- FastReport.Data.Firebird
- FastReport.Data.Json
- FastReport.Data.MongoDB
- FastReport.Data.MsSql
- FastReport.Data.MySql
- FastReport.Data.OracleODPCore
- FastReport.Data.Postgres
- FastReport.Data.RavenDB
- FastReport.Data.SQLite

Compiling the source code

Source code is shipped with FastReport .NET Professional edition. It includes source code of FastReport.dll, FastReport.Web.dll libraries. You may include it in your application's solution file. Let us demonstrate how to do this:

- open your project in the Visual Studio;
- open the Solution Explorer and right-click on the "Solution" item;
- choose the "Add/Existing Project..." item;
- add the "FastReport.csproj" file (it is located in the "C:\Program Files\FastReports\FastReport.Net\Source\FastReport" folder);
- add the "FastReport.Web.csproj" file (it is located in the "C:\Program Files\FastReports\FastReport.Net\Source\FastReport.Web" folder).

Turn off assembly signing for FastReport and FastReport.Web projects. To do this:

- right-click the "FastReport" project in the Solution Explorer;
- choose the "Properties" item;
- switch to the "Signing" tab and uncheck the "Sign the assembly" checkbox;
- do the same steps for FastReport.Web project.

Update the references to other FastReport .NET assemblies. To do this:

- expand the "FastReport\References" item in the Solution Explorer;
- remove the "FastReport.Bars", "FastReport.Editor" references;
- right-click the "References" item and choose the "Add Reference..." item;
- add references to the "FastReport.Bars.dll" and "FastReport.Editor.dll" files. These files are located in the "C:\Program Files\FastReports\FastReport.Net" folder.

Credits

Toolbars and docking - DevComponents (http://www.devcomponents.com)

Code editor with syntax highlight - Quantum Whale (http://www.qwhale.net)

Silk icons set (http://www.famfamfam.com/lab/icons/silk)

Working with Windows.Forms

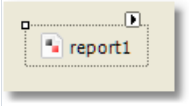
Using the Report component in Visual Studio

Let us consider the typical use of the Report component in Visual Studio. We will use the data from a typed dataset.

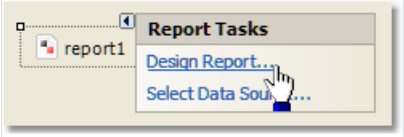
- create a new Windows Forms application;
- add a dataset into it ("Data|Add New Data Source..." menu item);
- switch to the Form designer;
- add the "DataSet" component on a form and connect it to the typed dataset that you have created.

To create a report, perform the following steps:

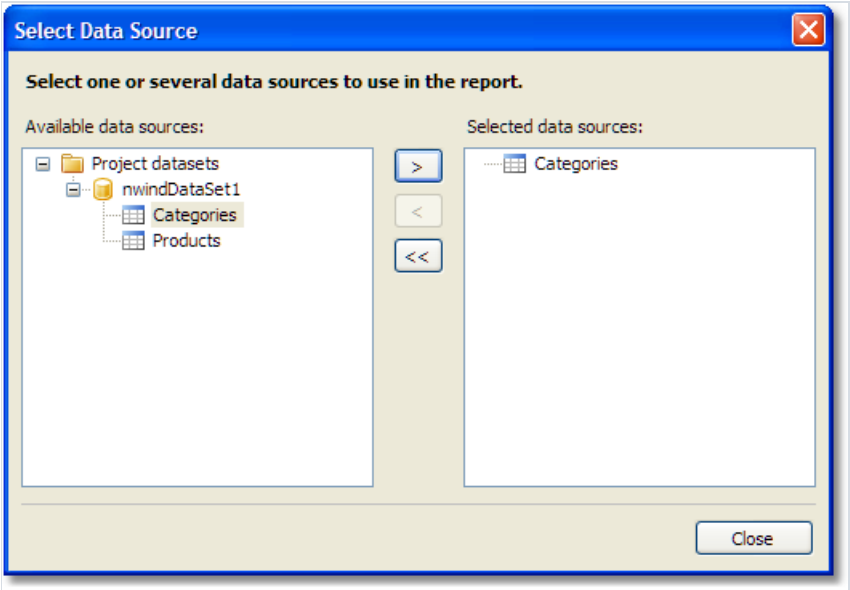
- put the "Report" component on a form:



- right-click it (or click on a smart tag button) and select the "Design Report..." item:



- choose the data source to use in a report:



- create your report. Read more about this in the User's Manual;
- close the report designer;
- add a "Button" control on your form;
- double-click it and write the following code in the button_Click event handler:

```
report1.Show();
```

- save the project and run it. When you click on a button you will see the prepared report.

Working with report in a code

To work with Report component in a code, you need to do the following:

- create a report instance;
- load a report file into it;
- register the application-defined data in a report;
- pass the values into the report parameters, if needed;
- run the report.

The following example demonstrates how to do this:

```
using (Report report = new Report())
{
  report.Load("report1.frx");
  report.RegisterData(dataSet1, "NorthWind");
  report.Show();
}
```

We will consider these steps in details in the following sections of this manual.

Storing and loading a report

You may store a report in the following ways:

Method	Description
in the application's resources	The typical scenario of using the Report, which we looked at before, uses this method. The StoreInResources property of the Report object is responsible for this. This property is set to true by default. This method has the following pros and cons:+ a report is embedded into your application, you don't need to deploy extra files;- if you need to change a report, you have to recompile your application.Loading a report is performed automatically. To do this, FastReport adds a code into the InitializeComponent method of your form.
in the .FRX file	This method is useful if you want to give your users the ability to change a report. In this case, set the report's StoreInResources property to false. To load the report from a file, use the Load method of the Report object: `report1.Load("filename.frx");`
in the database	You may store a report in the database, either as a string or in a blob-stream. To load the report from a string, use the LoadFromString method of the Report object. To load the report from a stream, use the overloaded version of the Load method: `report1.Load(stream);` To support the load/save operations in the report designer, you need to replace the "Open File" and "Save File" dialogs in the designer. Read here how to do this.
as a C#/VB.NET class	To work with a report as a class, design your report and save in to the .cs/.vb file. To do this, select "file type" in the "Save" dialog. The file type maybe either .cs or .vb - it depends on the script language in the report (it may be changed in the "Report

Registering data

If your report uses data from an application (for example, the typed dataset or a business-object), you have to register such data in a report. This can be done using the RegisterData method of the Report object.

> When you use the Report as described in the "Using the Report component in Visual Studio" section, you don't need to register the data. FastReport .NET does it automatically (it adds the RegisterData call in the InitializeComponent method of your form).

The RegisterData method must be called after you have loaded the report:

```
report1 = new Report();
report1.Load("report.frx");
report1.RegisterData(dataSet1, "NorthWind");
```

The RegisterData method is overloaded and allows to register the following data:

Method	Description
`void RegisterData(DataSet data)`	Registers the dataset. This method registers all tables, views and relations as well.Attention: if you register more than one dataset, use the RegisterData(DataSet data, string name) method instead.
`void RegisterData(DataSet data, string name)`	Registers the dataset. Specify any name in the name parameter (it must be persistent and unique if you register more than one dataset).
`void RegisterData(DataTable data, string name)`	Registers the data table.
`void RegisterData(DataView data, string name)`	Registers the data view.
`void RegisterDataAsp(IDataSource data, string name)`	Registers the ASP.NET data source such as AccessDataSource.
`void RegisterData(DataRelation data, string name)`	Registers the relation.
`void RegisterData(IEnumerable data, string name, BOConverterFlags flags, int maxNestingLevel)`	Registers the business object. Specify what items (properties, fields) should be used, in the flags parameter. Specify the maximum nesting level in the `maxNestingLevel` parameter (typically you need no more than 3 levels). Several nested objects may slow down the report.

Passing a value to a report parameter

The report may have parameters. Read more about this in the User's Manual. To pass a value to the parameter, use the SetParameterValue method of the Report object:

```
report1.Load("report.frx");
report1.SetParameterValue("MyParam", 10);
report1.Show();
```

This method is declared as follows:

```
publicvoid SetParameterValue(string complexName, object value)
```

Specify the parameter's name in the complexName parameter. To access a nested parameter, use its full name, for example:

```
"ParentParameter.ChildParameter"
```

Running a report

To run a report, use one of the following methods of the Report object:

Method	Description

`void Show()`	Runs a report and shows it in the preview window.
	This method is equal to:
	`if (Prepare())`
	`  ShowPrepared();`

`bool Prepare()`	Runs a report. If the report was prepared successfully, returns true. After this method, you need to call one of the following methods: ShowPrepared, PrintPrepared, SavePrepared, Export:
	`if (Prepare())`
	`  ShowPrepared();`

`bool Prepare(bool append)`	Runs a report. If the append parameter is set to true, the prepared report will be added to the existing one.
	So you can build several reports and display them in the preview as one report:
	`report1.Load("report1.frx");`
	`report1.Prepare();`
	`report1.Load("report2.frx");`
	`report1.Prepare(true);`
	`report.ShowPrepared();`

`void ShowPrepared()`	Shows a prepared report in the preview window. The report must be either prepared using the Prepare method, or loaded from the .FPX file using the LoadPrepared method:
	`if (Prepare())`
	`  ShowPrepared();`

`void ShowPrepared(bool modal)`	Shows a prepared report in the preview window. The modal parameter determines whether the preview should be shown modally.
`void ShowPrepared(bool modal, Form owner)`	The same as the previous method. The owner parameter determines a window that owns the preview window.
`void ShowPrepared(Form mdiParent)`	The same as the previous method. The mdiParent parameter determines the main MDI window.

Designing a report

You can use the report designer in your application. This is possible for all FastReport .NET editions except the Basic. To do this, use the Design method of Report object:

```
report1 = new Report();
report1.Load("report1.frx");
report1.Design();
```

The Design method is overloaded:

Method	Description
`bool Design()`	Shows the designer.
`bool Design(bool modal)`	Shows the designer. The modal parameter determines whether it is necessary to show the designer modally.
`bool Design(Form mdiParent)`	Shows the designer. The mdiParent parameter defines the main MDI window.

Exporting a report

The prepared report can be exported to one of the supported formats. At this moment, the following formats can be used:

- PDF
- HTML
- RTF
- Excel XML (Excel 2003+)
- Excel 2007
- CSV
- TXT
- OpenOffice Calc
- Pictures (Bmp, Png, Jpeg, Gif, Tiff, Metafile)

The export is performed by the export filter. To do this:

- prepare a report using the Prepare method;
- create an instance of export filter and set up its properties;
- call the Export method of the Report object.

The following example exports a prepared report in the HTML format:

```
// prepare a report
report1.Prepare();
// create an instance of HTML export filter
FastReport.Export.Html.HTMLExport export = new FastReport.Export.Html.HTMLExport();
// show the export options dialog and do the export
if (export.ShowDialog())
  report1.Export(export, "result.html");
```

In this example, export settings are made using the dialog window.

Configuring the FastReport .NET environment

Using the EnvironmentSettings component which is available in the Toolbox, you can control some FastReport .NET environment settings. To do this, put the component on your form and set up its properties using the Properties window.

The EnvironmentSettings.ReportSettings property contains some report-related settings:

Property	Description
`Language DefaultLanguage`	The default script language for new reports.
`bool ShowProgress`	Determines whether it is necessary to show the progress window.
`bool ShowPerformance`	Determines whether to show the information about the report performance (report generation time, memory consumed) in the lower right corner of the preview window.

The EnvironmentSettings.DesignerSettings property contains some designer-related settings:

Property	Description
`Icon Icon`	The icon for the designer window.
`Font DefaultFont`	The default font used in a report.

The EnvironmentSettings.PreviewSettings property contains some preview-related settings:

Property	Description
`PreviewButtons Buttons`	Set of buttons that will be visible in the preview's toolbar.
`int PagesInCache`	The number of prepared pages that can be stored in the memory cache during preview.
`bool ShowInTaskbar`	Determines whether the preview window is displayed in the Windows taskbar.
`bool TopMost`	Determines whether the preview window should be displayed as a topmost form.
`Icon Icon`	The icon for the preview window.
`string Text`	The text for the preview window. If no text is set, the default text "Preview" will be used.

The EnvironmentSettings.EmailSettings property contains email account settings. These settings are used in the "Send Email" feature in the preview window:

Property	Description
`string Address`	Sender address (e.g. your email address).

Property	Description
`string Name`	Sender name (e.g. your name).
`string MessageTemplate`	The message template that will be used to create a new message. For example, "Hello, Best regards, ...".
`string Host`	SMTP host address.
`int Port`	SMTP port (25 by default).
`string UserName, string Password`	User name and password. Leave these properties empty if your server does not require authentication.
`bool AllowUI`	Allows to change these settings in the "Send Email" dialog. Settings will be stored in the FastReport .NET configuration file.

UI style settings are available in the following properties of EnvironmentSettings component:

Property	Description
`UIStyle UIStyle`	The style of designer and preview form. 6 styles are available - VisualStudio2005, Office2003, Office2007Blue, Office2007Silver, Office2007Black, VistaGlass. The default style is Office2007Black.
`bool UseOffice2007Form`	This property affects the designer and preview form. It determines whether the Office2007-style form should be used if one of the following styles is selected: Office2007Blue, Office2007Silver, Office2007Black, VistaGlass.Default value is true.

Besides these properties, the EnvironmentSettings component has some events. Using such events, you may do the following:

- replace standard "Open file" and "Save file" dialogs in the designer;
- replace standard progress window;
- pass own connection string to a connection defined in the report.

These tasks will be described in the following sections of this manual.

Replacing the "Open" and "Save" dialogs

If you decided to store a report in the database, you may need to change the designer in such a way that it can open and save reports from/to a database. That is, you need to replace standard "Open" and "Save" dialogs with your own dialogs that work with database. To do this, use the EnvironmentSettings component (see the Configuring the FastReport .NET environment). This component has the following events:

Event `CustomOpenDialog`

Occurs when the report designer is about to show the "Open" dialog. In the event handler, you must display a dialog window to allow user to choose a report file. If dialog was executed successfully, you must return e.Cancel = false and set the e.FileName to the selected file name. The following example demonstrates how to use this event:

```
private void CustomOpenDialog_Handler(object sender, OpenSaveDialogEventArgs e)
{
  using (OpenFileDialog dialog = new OpenFileDialog())
  {
    dialog.Filter = "Report files (*.frx)|*.frx";
    // set e.Cancel to false if dialog
    // was succesfully executed
    e.Cancel = dialog.ShowDialog() != DialogResult.OK;
    // set e.FileName to the selected file name
    e.FileName = dialog.FileName;
  }
}
```

Event `CustomSaveDialog`

Occurs when the report designer is about to show the "Save" dialog. In the event handler, you must display a dialog window to allow user to choose a report file. If dialog was executed successfully, you must return e.Cancel = false and set the e.FileName to the selected file name. The following example demonstrates how to use this event:

```
private void CustomSaveDialog_Handler(object sender, OpenSaveDialogEventArgs e)
{
  using (SaveFileDialog dialog = new SaveFileDialog())
  {
    dialog.Filter = "Report files (*.frx)|*.frx";
    // get default file name from e.FileName
    dialog.FileName = e.FileName;
    // set e.Cancel to false if dialog
    // was succesfully executed
    e.Cancel = dialog.ShowDialog() != DialogResult.OK;
    // set e.FileName to the selected file name
    e.FileName = dialog.FileName;
  }
}
```

Event `CustomOpenReport`

Occurs when the report designer is about to load the report. In the event handler, you must load the report specified in the e.Report property from the location specified in the e.FileName property. The latter property contains the name that was returned by the CustomOpenDialog event handler. It may be the file name, the database key value, etc. The following example demonstrates how to use this event:

```
private void CustomOpenReport_Handler(object sender, OpenSaveReportEventArgs e)
{
  // load the report from the given e.FileName
  e.Report.Load(e.FileName);
}
```

Event `CustomSaveReport`

Occurs when the report designer is about to save the report. In the event handler, you must save the report specified in the e.Report property to the location specified in the e.FileName property. The latter property contains the name that was returned by the CustomSaveDialog event handler. It may be the file name, the database key value, etc. The following example demonstrates how to use this event:

```
private void CustomSaveReport_Handler(object sender, OpenSaveReportEventArgs e)
{
  // save the report to the given e.FileName
  e.Report.Save(e.FileName);
}
```

Replacing the standard progress window

The progress window is shown during the following actions:

- running a report
- printing
- exporting

You may turn off the progress by setting the ReportSettings.ShowProgress property of the EnvironmentSettings component to false. Besides that, you may replace the standard progress window with your own. To do this, use the following events of the EnvironmentSettings component (see the "Configuring the FastReport .NET environment" section):

Event	Description
`StartProgress`	Occurs once before the operation. In this event, you have to create your own progress window and show it.
`Progress`	Occurs each time when current report page is handled. In this event, you have to show the progress state in your window.
`FinishProgress`	Occurs once after the operation. In this event, you have to destroy the progress window.

The Progress event takes e parameter of ProgressEventArgs type. It has the following properties:

Property	Description
`string Message`	The message text.
`int Progress`	The index of current report page being handled.
`int Total`	The number of total pages in a report. This parameter may be 0 when preparing a report, because the number of total pages is unknown.

In most cases, you need to display the text from the e.Message property, in the Progress event handler. Other parameters may be useful if you want to display a progress bar.

Passing own connection string

If you use data sources that are defined inside a report, you may need to pass an application-defined connection string to a report. This can be done in three ways.

The first method: you pass a connection string directly to the Connection object in a report. Do the following:

```
report1.Load(...);
// do it after loading the report, before running it
// assume we have one connection in the report
report1.Dictionary.Connections[0].ConnectionString = my_connection_string;
report1.Show();
```

The second method: you pass a connection string using the report parameter. Do the following:

- run the report designer;
- in the "Data" window, create a new report parameter (with "MyParameter" name, for example). See the User's Manual for more details;
- in the "Data" window, select the "Connection" object that contains a data source;
- switch to the "Properties" window and set the ConnectionStringExpression property to the following:

```
[MyParameter]
```

- pass the connection string to the MyParameter parameter:

```
report1.SetParameterValue("MyParameter", my_connection_string);
```

The third method: use the DatabaseLogin event of the EnvironmentSettings component (see the "Configuring the FastReport .NET environment" section). This event occurs each time when FastReport opens the connection. Here is an example of this event handler:

```
private void environmentSettings1_DatabaseLogin(
  object sender, DatabaseLoginEventArgs e)
{
  e.ConnectionString = my_connection_string;
}
```

> Keep in mind that the DatabaseLogin event is global, it works with all reports.

Passing custom SQL

The report may contain data sources that are added using the Data Wizard (via "Data|Add Data Source..." menu). Sometimes it is needed to pass custom SQL to that data source from your application. To do this, use the following code:

```
using FastReport.Data;
report1.Load(...);
// do it after loading the report, before running it
// find the table by its alias
TableDataSource table = report1.GetDataSource("MyTable") as TableDataSource;
table.SelectCommand = "new SQL text";
report1.Show();
```

Reference to a report object

When you work with a report as a class (see the "Storing a report and loading it" section), you may refer to the report objects directly. The following example demonstrates how to change the font of the "Text1" object contained in a report:

```
SimpleListReport report = new SimpleListReport();
report.Text1.Font = new Font("Arial", 12);
```

In other cases, you have to use the FindObject method of the Report object, if you need to get a reference to an object:

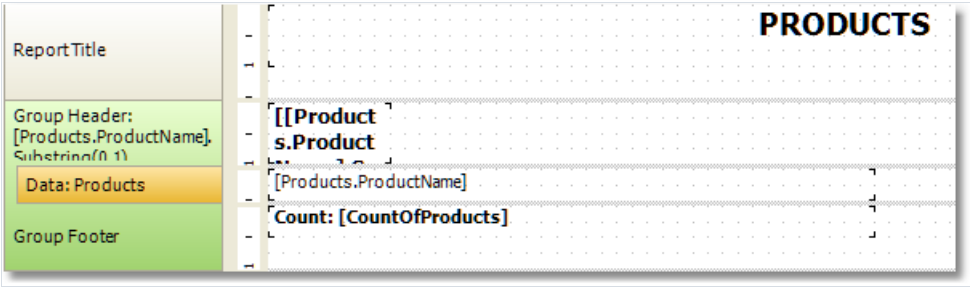
```
TextObject text1 = report1.FindObject("Text1") as TextObject;
text1.Font = new Font("Arial", 12);
```

To reference to a data source defined in a report, use the GetDataSource method of the Report object. This method takes a data source's alias as a parameter:

```
DataSourceBase ds = report1.GetDataSource("Products");
```

Creating a report by using code

Let us consider how to create a report in code. We will create the following report:




```
Report report = new Report();

// register the "Products" table
report.RegisterData(dataSet1.Tables["Products"], "Products");

// enable it to use in a report
report.GetDataSource("Products").Enabled = true;

// create A4 page with all margins set to 1cm
ReportPage page1 = new ReportPage();
page1.Name = "Page1";
report.Pages.Add(page1);

// create ReportTitle band
page1.ReportTitle = new ReportTitleBand();
page1.ReportTitle.Name = "ReportTitle1";

// set its height to 1.5cm
page1.ReportTitle.Height = Units.Centimeters * 1.5f;

// create group header
GroupHeaderBand group1 = new GroupHeaderBand();
group1.Name = "GroupHeader1";
group1.Height = Units.Centimeters * 1;

// set group condition
group1.Condition = "[Products.ProductName].Substring(0, 1)";

// add group to the page.Bands collection
page1.Bands.Add(group1);

// create group footer
group1.GroupFooter = new GroupFooterBand();
group1.GroupFooter.Name = "GroupFooter1";
group1.GroupFooter.Height = Units.Centimeters * 1;

// create DataBand
DataBand data1 = new DataBand();
data1.Name = "Data1";
data1.Height = Units.Centimeters * 0.5f;

// set data source
data1.DataSource = report.GetDataSource("Products");

// connect databand to a group
group1.Data = data1;

// create "Text" objects
```

```
// report title
TextObject text1 = new TextObject();
text1.Name = "Text1";

// set bounds
text1.Bounds = new RectangleF(0, 0, Units.Centimeters * 19, Units.Centimeters * 1);

// set text
text1.Text = "PRODUCTS";

// set appearance
text1.HorzAlign = HorzAlign.Center;
text1.Font = new Font("Tahoma", 14, FontStyle.Bold);

// add it to ReportTitle
page1.ReportTitle.Objects.Add(text1);

// group
TextObject text2 = new TextObject();
text2.Name = "Text2";
text2.Bounds = new RectangleF(0, 0, Units.Centimeters * 2, Units.Centimeters * 1);
text2.Text = "[[Products.ProductName].Substring(0, 1)]";
text2.Font = new Font("Tahoma", 10, FontStyle.Bold);

// add it to GroupHeader
group1.Objects.Add(text2);

// data band
TextObject text3 = new TextObject();
text3.Name = "Text3";
text3.Bounds = new RectangleF(0, 0, Units.Centimeters * 10, Units.Centimeters * 0.5f);
text3.Text = "[Products.ProductName]";
text3.Font = new Font("Tahoma", 8);

// add it to DataBand
data1.Objects.Add(text3);

// group footer
TextObject text4 = new TextObject();
text4.Name = "Text4";
text4.Bounds = new RectangleF(0, 0, Units.Centimeters * 10, Units.Centimeters * 0.5f);
text4.Text = "Count: [CountOfProducts]";
text4.Font = new Font("Tahoma", 8, FontStyle.Bold);

// add it to GroupFooter
group1.GroupFooter.Objects.Add(text4);

// add a total
Total groupTotal = new Total();
groupTotal.Name = "CountOfProducts";
groupTotal.TotalType = TotalType.Count;
groupTotal.Evaluator = data1;
groupTotal.PrintOn = group1.Footer;

// add it to report totals
report.Dictionary.Totals.Add(groupTotal);

// run the report
report.Show();
```

The prepared report looks as follows:

PRODUCTS

A

Aniseed Syrup
Alice Mutton
Count: 2

B

Boston Crab Meat
Count: 1

Using own preview window

Using the EnvironmentSettings component (see the "Configuring the FastReport .NET environment" section), you may tune up the standard preview window. The related properties are contained inside the EnvironmentSettings.PreviewSettings property.

If you don't want to use the standard preview window for some reason, you may create your own. To do this, use the PreviewControl control that can be added on your form. To show a report in this control, connect it to the Report object by the following code:

```
report1.Preview = previewControl1;
```

To prepare a report and show it in the PreviewControl, use the Show method of the Report object:

```
report1.Show();
your_form.ShowDialog();
```

or the following code:

```
if (report1.Prepare())
{
  report1.ShowPrepared();
  your_form.ShowDialog();
}
```

In these examples, the your_form is your form that contains the PreviewControl.

Using the methods of PreviewControl component, you can handle it from your code. You may even turn off the standard toolbar and/or statusbar, using the ToolbarVisible, StatusbarVisible properties. This is demonstrated in the Demos\C#\CustomPreview example project.

Filtering tables in the Data Wizard

The Data Wizard can be called from the "Data|Add Data Source..." menu. Here you can set up the connection and choose one or several data tables. By default, the wizard displays all tables available in the selected connection. If you want to filter unnecessary tables, use the Config.DesignerSettings.FilterConnectionTables event. The following example shows how to remove the "Table 1" table from the tables list:

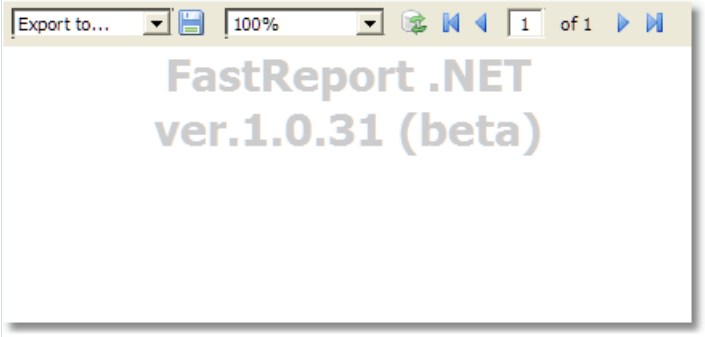
```
using FastReport.Utils;
Config.DesignerSettings.FilterConnectionTables += FilterConnectionTables;
private void FilterConnectionTables(
  object sender, FilterConnectionTablesEventArgs e)
{
  if (e.TableName == "Table 1")
    e.Skip = true;
}
```

Working with ASP.NET

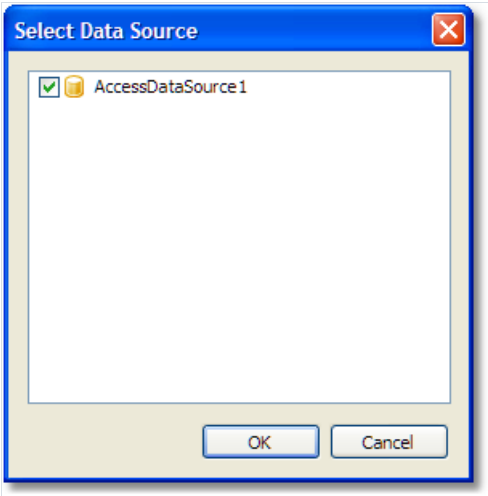
Using the WebReport component

Let us consider the typical case of using the WebReport component.

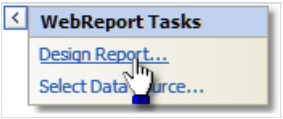
- assuming that you have a web project with all necessary data sources (for example, AccessDataSource);
- put the WebReport component on your web form:



- in the "smart tag" menu, select the "Select Data Source..." item and choose one or several data sources which you want to use in a report:



- in the "smart tag" menu, select the "Design Report..." item to run the report designer:



- create a report. Read more about this in the User's Manual;
- close the designer;
- save the changes in your project and run it. You will see a window with a prepared report.

Setting up web handler

WebReport requires a specific handler to be set in the web.config file. When you create a report object in Visual Studio, necessary lines are automatically written to the configuration file. The WebReport component checks the availability of the specified configuration at run time of the application. If the required lines are not found in the web.config file an error is thrown, requesting the file to be changed.

The web.config file should contain the following lines when used with an IIS6 server:

```
<system.web>
…
<httpHandlers>
 <add path="FastReport.Export.axd" verb="*" type="FastReport.Web.Handlers.WebExport"/>
</httpHandlers>
…
</system.web>
```

and with an IIS7 server, these lines:

```
<system.webServer>
…
<handlers>
 <add name="FastReportHandler" path="FastReport.Export.axd" verb="*"
type="FastReport.Web.Handlers.WebExport"/>
</handlers>
…
</system.webServer>
```

The correct WebReport configuration lines in the web.config file must be used when transferring your project from one server to another.

Check for correct working of the WebReport handler by means of the URL:

http://yoursite/app_folder/FastReport.Export.axd

An information message gives the version of FastReport and the server time.

Storing and loading a report

You may store a report in the following ways:

In a web form:

Typical scenario that we have looked at before, uses this method. The report is stored in the ReportResourceString property of the WebReport component. This method has the following pros and cons:

+ it's a simplest way to work with FastReport;
- the report template is stored in the ViewState of your web form. It will be transferred on a client side. It may slow down the work if the report has a big size;
- this method is not compatible with "Medium Trust" mode.

The report loading is performed automatically.

In the .FRX file:

This method assumes that the report is stored in a file in a special folder "App_Data". To do this:

- run the report designer;
- create a report and save it to the .FRX file;
- in the Solution Explorer, select the "App_Data" folder, right-click it and choose the "Add/Existing Item..." item. Select the report file that you just saved;
- select the WebReport component and clear its ReportResourceString property;
- select the "ReportFile" property, invoke its editor and choose the report from "App_Data" folder.

This method has the following pros and cons:

+ the report is not transferred to a client machine;
- this method is not compatible with "Medium Trust" mode.

The report loading is performed automatically.

A report can also be loaded from WebReport.StartReport event handler. Example code in StartReport:

```
(sender as WebReport).Report.Load(this.Server.MapPath("~/App_Data/report.frx"));
```

As a C#/VB.NET class:

In this method, you work with the report as a class. To do this:

- design your report and save in to the .cs/.vb file. To do this, select "file type" in the "Save" dialog. The file type maybe either .cs or .vb - it depends on the script language in the report (it may be changed in the "Report/Options..." menu);
- include that file into your project. It's better to save it in the "App_Code" folder;
- clear both ReportResourceString and ReportFile properties of the WebReport component.

This method has the following pros and cons:

+ you can work with the report as a regular class;
+ you can debug the report in the Visual Studio;
+ it's the only way to use a report in the "Medium Trust" mode;
- you cannot edit such a report. To do this, you need the original .FRX file.

To work with a report, create the WebReport.StartReport event handler. In this handler, you should do the following:

- create an instance of your report class;
- register the data;
- set the report to the Report property of the WebReport component.

Example of the StartReport event handler:

```
SimpleListReport report = new SimpleListReport();
report.RegisterDataAsp(your_data, "your_data_name");
WebReport1.Report = report;
```

The prepared report can be displayed from WebReport.StartReport event handler using the property WebReport.ReportDone. Example code in StartReport to load and display a prepared report:

```
(sender as WebReport).Report.LoadPrepared(this.Server.MapPath("~/App_Data/Prepared.fpx"));
(sender as WebReport).ReportDone = true;
```

Registering data

If you select the data source using the "smart tag" menu of the WebReport component, you don't need to register the data manually. In this case, FastReport stores the names of data sources in the ReportDataSources property of the WebReport component.

In case you don't want to use such method of registering data, you need to do it manually. It can be done by using the StartReport event of the WebReport component. In this event handler, you can call the RegisterData and RegisterDataAsp methods of the report. The report can be accessed through the WebReport.Report property:

```
webReport1.Report.RegisterData(myDataSet);
```

Read more about registering data in this section.

Passing a value to a report parameter

To pass a value to the report parameter, use the SetParameterValue method of the Report object. This method was described in details in the "Working with Windows.Forms" chapter.

To use this method in ASP.NET, you need to create the event handler for the StartReport event of the WebReport component. The report can be accessed through the WebReport.Report property:

```
webReport1.Report.SetParameterValue("MyParam", 10);
```

Working in the "Medium Trust" mode

This mode is used by many shared hosting providers. In this mode, the following actions are restricted:

- report compilation is impossible;
- impossible to use MS Access data source;
- impossible to use the RichObject;
- impossible to use some export filters that use WinAPI calls or temp files (PDF, Open Office);
- there may be other restrictions, depending on the provider.

To work with a report in this mode, you need to store a report as a C#/VB.NET class, as described in the "Storing and loading a report" section. In this case, the report compilation is not required.

Besides that, it is necessary to add System.Windows.Forms.DataVisualization.dll assembly into the GAC. This assembly is a part of Microsoft Chart Control and is used in FastReport to draw charts. Consult with your shared-hosting provider regarding adding this assembly into the GAC.

Working in Web Farm and Web Garden architectures

To use the FastReport report generator in a multi-server (Web Farm) or multi-processor (Web Garden) architecture there are additional requirements for creating special storage for data synchronization between WebReport objects.

Add the following lines to the configuration file web.config:

```
 <appSettings>
<add key="FastReportStoragePath" value="\\FS\WebReport_Exchange"/>
<add key="FastReportStorageTimeout" value="10"/>
<add key="FastReportStorageCleanup" value="1"/>
 </appSettings>
```

- FastReportStoragePath : path to the folder for temporary files when working in a multi-server architecture, each server must have access to this folder
- FastReportStorageTimeout : cache time for reports, in minutes
- FastReportStorageCleanup : time for checking the expired cache entries, in minutes

Check for correct configuration by means of the URL:

http://yoursite/app_folder/FastReport.Export.axd

You should see "Cluster mode: ON".

Working with ASP.NET MVC

You will not have any problems when using WebReport in ASPX (MVC 2) – it is only necessary to drag the control from the Toolbox to the page. WebReport will make all the required changes to web.config automatically. Let's look at a demo of WebReport in aspx, to be found in folder \Demos\C#\MvcDemo.

To use WebReport in Razor (MVC 3,4) you will need to add a line with the handler definitions to the web.config file in the root folder of your web-application.

Add this line in section <system.web> for use with IIS6:

```
<add path="FastReport.Export.axd" verb="*" type="FastReport.Web.Handlers.WebExport" />
```

and add this line in section <system.webServer> for use with IIS7:

```
<add name="FastReportHandler" path="FastReport.Export.axd" verb="*" type="FastReport.Web.Handlers.WebExport" />
```

Then modify the web.config file in the folder containing Views. Add these lines in section <system.web.webPages.razor> :

```
<add namespace="FastReport" />
<add namespace="FastReport.Web" />
```

Add these lines to file _Layout.cshtml in tag :

```
@WebReportGlobals.Scripts()
@WebReportGlobals.Styles()
```

Now you can draw the report on the View. Go to the controller and create a WebReport:

```
WebReport webReport = new WebReport(); // create object
```

```
webReport.Width = 600;  // set width
webReport.Height = 800; // set height
webReport.Report.RegisterData(dataSet, "AppData"); // data binding
webReport.ReportFile = this.Server.MapPath("~/App_Data/report.frx");  // load the report from the file
ViewBag.WebReport = webReport; // send object to the View
```

Go to View and add the line:

```
@ViewBag.WebReport.GetHtml()
```

Similar code to create WebReport you can also write directly in View.

Let's look at the demo of WebReport in Razor in folder \Demos\C#\MvcRazor. There are various samples for loading into the report, including pre-prepared, and there is an example of using the event StartReport.

> Don't forget to add the missing dll in the bin directory.

Example of export in MVC

When using FastReport.Net together with the ASP.Net MVC framework there is an easy method for creating a file in any supported format from a button press on the HTML form.

Add this code in the View:

```
@using (Html.BeginForm("GetFile", "Home"))
{
 <input id="pdf" type="submit" value="Export to PDF" />
}
```

- GetFile : name of the controller handler
- Home : name of the controller (eg: HomeController.cs)

Add the name space in the controller:

```
using FastReport.Export.Pdf;
```

Add method GetFile in the controller:

```
public FileResult GetFile()
{
  WebReport webReport = new WebReport();
  // bind data
  System.Data.DataSet dataSet = new System.Data.DataSet();
  dataSet.ReadXml(report_path + "nwind.xml");
  webReport.Report.RegisterData(dataSet, "NorthWind");

  // load report
  webReport.ReportFile = this.Server.MapPath("~/App_Data/report.frx");
 // prepare report
 webReport.Report.Prepare();
 // save file in stream
 Stream stream = new MemoryStream();
 webReport.Report.Export(new PDFExport(), stream);
 stream.Position = 0;
 // return stream in browser
 return File(stream, "application/zip", "report.pdf");
}
```

Example for Excel 2007:

```
using FastReport.Export.OoXML;
...
webReport.Report.Export(new Excel2007Export(), stream);
...
return File(stream, "application/xlsx", "report.xlsx");
```

FastReport .Net and jQuery

The WebReport object from FastReport.Net uses the jQuery library. You may already be using this library in your project.

To avoid duplication of jQuery boot scripts and styles in the client browser when working with markup Razor, you must use the following lines in _Layout.cshtml:

```
@WebReportGlobals.ScriptsWOjQuery()
@WebReportGlobals.StylesWOjQuery()
```

replacing these lines, which include all jQuery files:

```
@WebReportGlobals.Scripts()
@WebReportGlobals.Styles()
```

You must set the property ExternalJquery = true (defaults to false) when working with ASPX markup.

Working with WCF

WCF service library FastReport.Service.dll

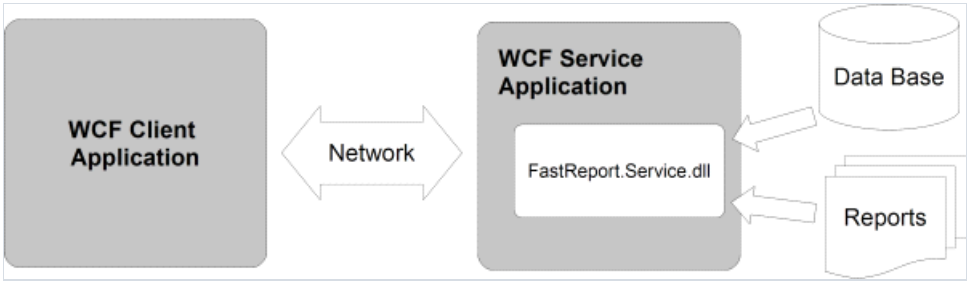
Simple example of WCF service

Creating Web Service using FastReport.Service.dll

Creating the WCF service hosted in windows service

WCF service library FastReport.Service.dll

FastReport .NET contains the library FastReport.Service.dll (only in the .NET 4.0 package). This library is a WCF Service Library and is intended for use in custom applications that perform the functions of the service.



The library contains the following functions:

```
List<ReportItem> GetReportsList();
```

returns a list of available reports. Each item is returned as a ReportItem object. Reports are stored on a hard drive on a server that is running the service. Files are sorted in alphabetical order.

```
List<ReportItem> GetReportsListByPath(string path);
```

returns a list of available reports by path. Files are sorted in alphabetical order.

`List<GearItem> GetGearList();`

returns a list of available formats that can generate service reports as elements GearItem.

`Stream GetReport(ReportItem report, GearItem gear);`

returns a stream of the result of building a report. Parameters "report" and "gear" can be used from the lists previously obtained, or by creating new objects with the required properties. The returned stream does not support positioning.

Let's look at list elements.

ReportItem

```
public class ReportItem
{
 public string Path;
 public string Name;
 public string Description;
 public Dictionary<string, string> Parameters;
}
```

Path – path to the report file on the server, relative to the root folder for storing reports. The file extension of the report must be *.frx. This property is used to identify a specific report with further queries.

Name – name of the report, taken from the metadata of the report. If the metadata of the report contains an empty name then the property contains a filename without an extension. This property can be used to build an interactive list of available reports in your application (eg: in a ListBox).

Description – description of the report, taken from the metadata of the report.

Dictionary<string, string> Parameters – dictionary of report parameters, may be filling parameters, which will be subsequently transferred to the report. It supports only the string values that must be considered when designing a report template.

GearItem

```
public class GearItem
{
 public string Name;
 public Dictionary<string, string> Properties;
}
```

Name – name of the format : may contain one of the following strings:

Name	Description
PDF	Adobe Acrobat file
DOCX	Microsoft Word 2007 file
XLSX	Microsoft Excel 2007 file
PPTX	Microsoft PowerPoint 2007 file
RTF	Rich Text file – supported by many text editors
ODS	Open Office Spreadsheet file
ODT	Open Office Text file
MHT	Compressed HTML file together with the images, can be opened in Internet Explorer
CSV	Comma separated values file
DBF	dBase file
XML	Excel XML table – without images
TXT	Text file
FPX	FastReport.Net Prepared report file

Dictionary<string, string> Properties – dictionary of parameters of a report. A complete list of supported parameters with default values is available on requesting the server to list formats.

When creating a service you must add the following lines in your App.config or Web.config:

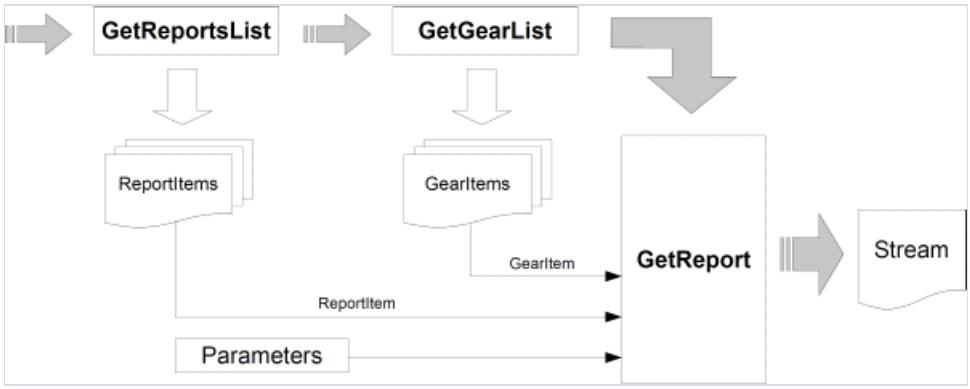
```
<appSettings>
<add key="FastReport.ReportsPath" value="C:\Program files\FastReports\FastReport.Net\Demos\WCF" />
<add key="FastReport.ConnectionStringName" value="FastReportDemo" />
<add key="FastReport.Gear" value="PDF,DOCX,XLSX,PPTX,RTF,ODS,ODT,MHT,CSV,DBF,XML,TXT,FPX" />
</appSettings>
```

FastReport.ReportsPath – specifies the path to the folder with the reports, a list of which will be transmitted to the client.

FastReport.ConnectionStringName – name of the connection string to the database, which is stored in the configuration section . Used to replace the internal connection string in the report template.

FastReport.Gear – list of available formats. You can select only those required and change the order of the names.

Schematic of using FastReport.Service:



And, if you already know exactly what to report and in which format to receive it (this reduces the number of queries made to the service):

Important points to note when you create report templates for use in the services:

- dialogs in the reports are not supported and will be ignored;
- each report must include an internal DataConnection, whose connection string for the report service is replaced by a string from the configuration.

Examples of use of FastReport.Service.dll can be found in the folders \Demos\C#\WCFWebService , \Demos\C#\WCFWindowsService , \Demos\C#\WCFWebClient , \Demos\C#\WCFClient.

An example configuration file service - FastReport.Service.dll.config.

Simple example of WCF service

This example does not require programming and is intended for testing the library and the configuration file. To complete the task, we will use the program WcfSvcHost.exe, that comes with Visual Studio:

1. Create a folder for our project anywhere on the disk, eg: as C:\WCF\FastReport
2. Copy these files to the folder : FastReport.Service.dll, FastReport.Service.dll.config, FastReport.dll and FastReport.Bars.dll
3. Create two sub-folders \Data and \Reports
4. Copy the database file to the \Data folder from the Demos folder \FastReport.Net\Demos\Reports\nwind.xml
5. Copy the contents of folder \FastReports\FastReport.Net\Demos\WCF to \Reports – it contains test reports with built-in connections to the database, which are essential when used with library FastReport.Service.dll
6. Open the configuration file FastReport.Service.dll.config in any text editor
7. Change the path to the reports in section

```
<add key="FastReport.ReportsPath" value="C:\WCF\FastReport\Reports" />
```

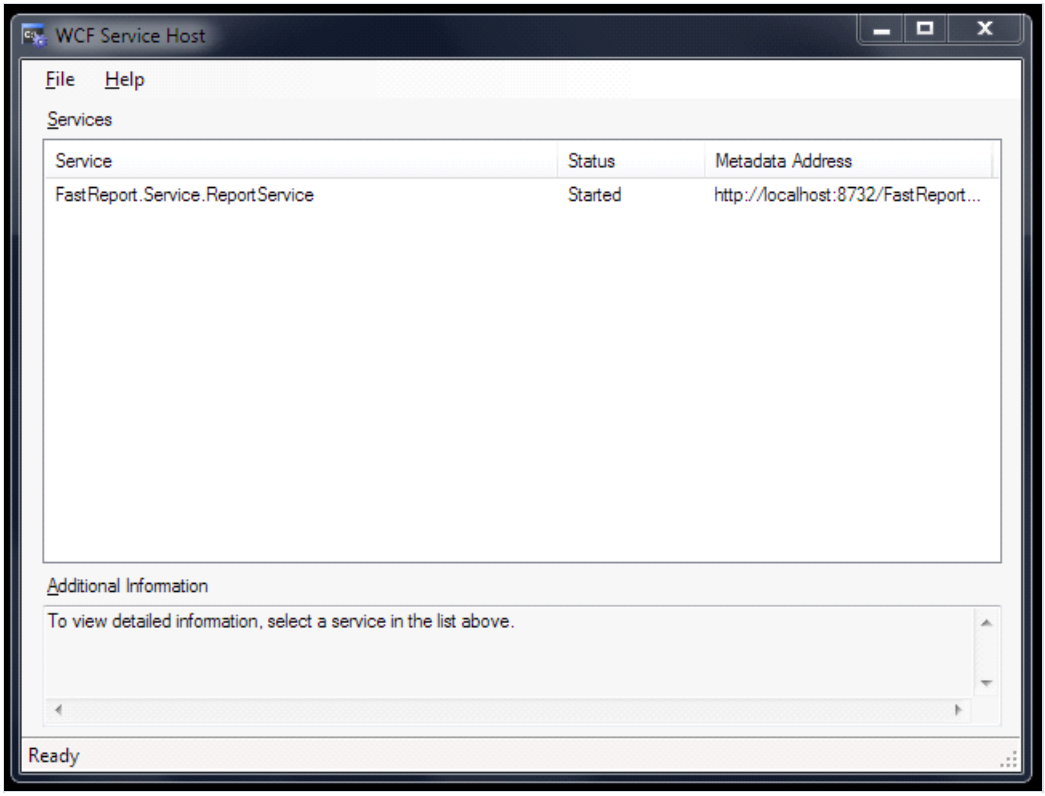
8. Change the connection string in section :

```
<add name="FastReportDemo" connectionString="XsdFile=;XmlFile=C:\WCF\FastReport\Data\nwind.xml"/>
```

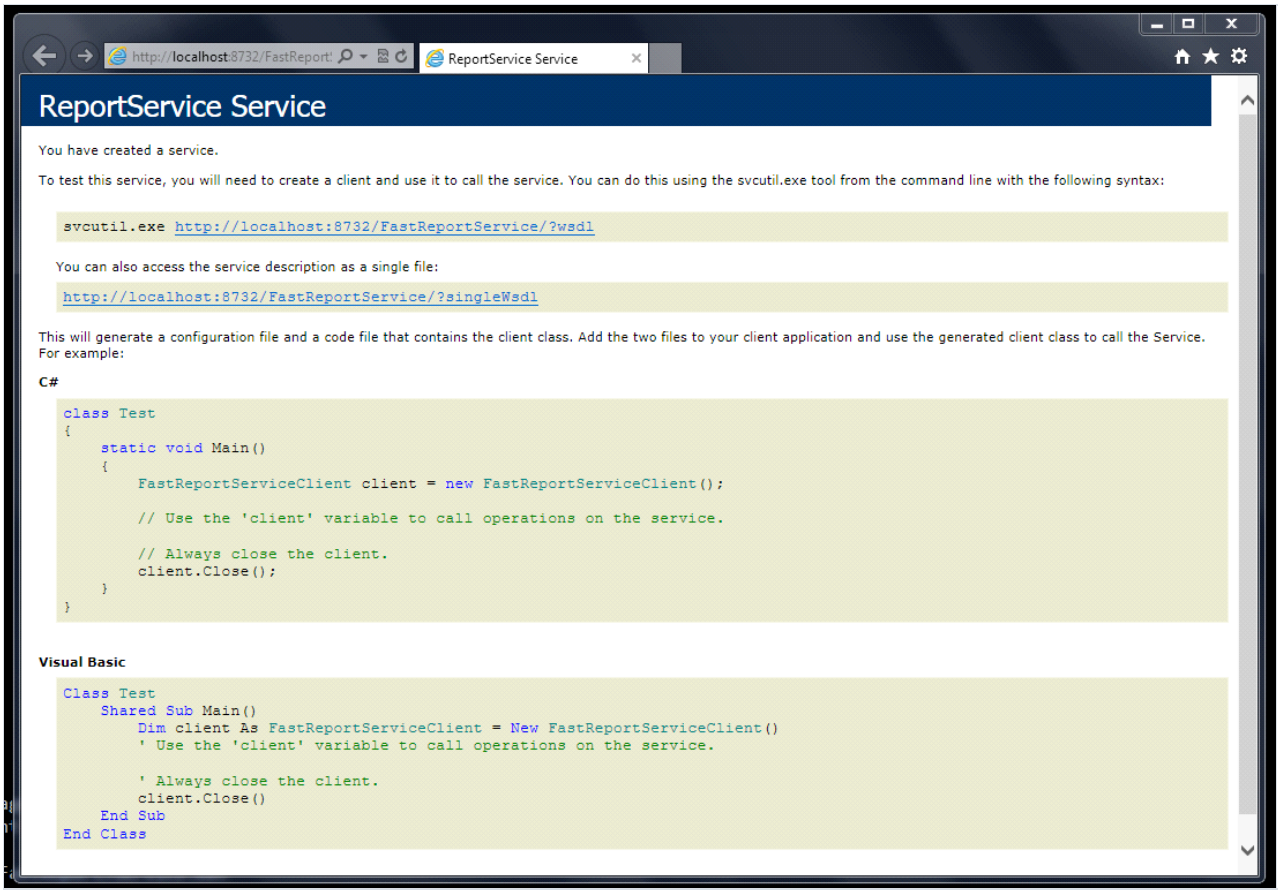
9. Create batch file service.bat containing the line:

"C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\WcfSvcHost.exe" /service:C:\WCF\FastReport\FastReport.Service.dll /config:C:\WCF\FastReport\FastReport.Service.dll.config

10. Run service.bat from Explorer with administrator rights ('Run as administrator'). You will see an icon for WCF Service Host in the system tray. Double-click on the icon:



11. Open a web browser and go to address http://localhost:8732/FastReportService/

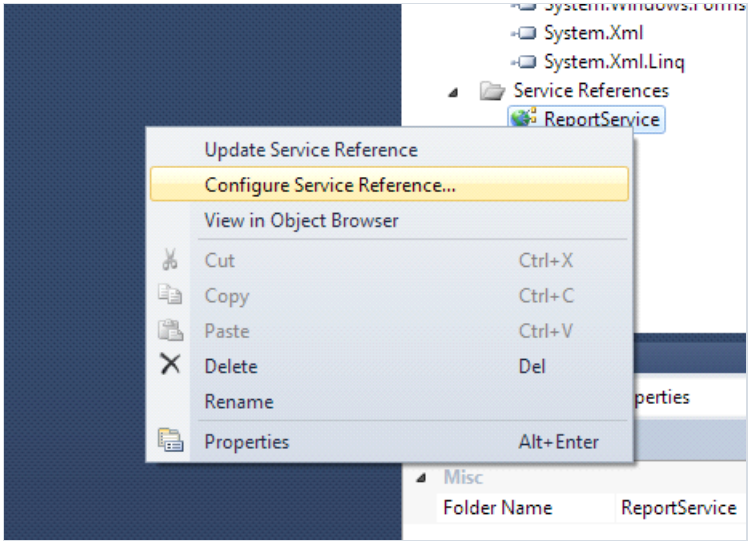


This shows the Service working normally. You can change the port number of the service in the configuration file:

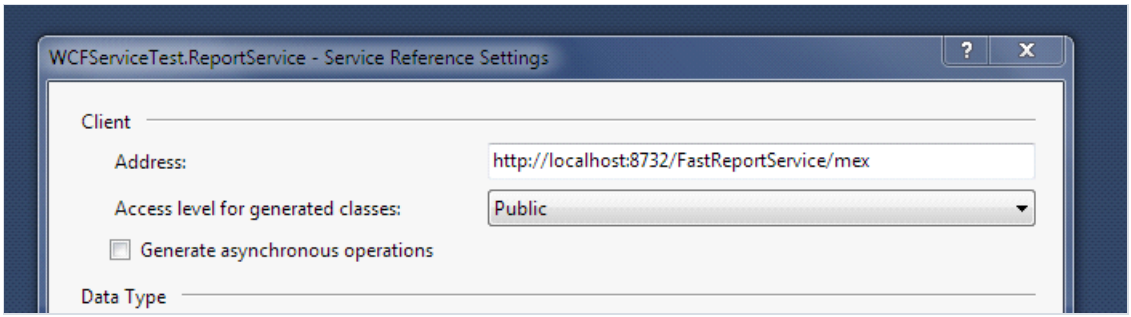
```
<add baseAddress="http://localhost:8732/FastReportService/" />
```

Let's connect to our service from the demo example \FastReport.Net\Demos\C#\WCFClient

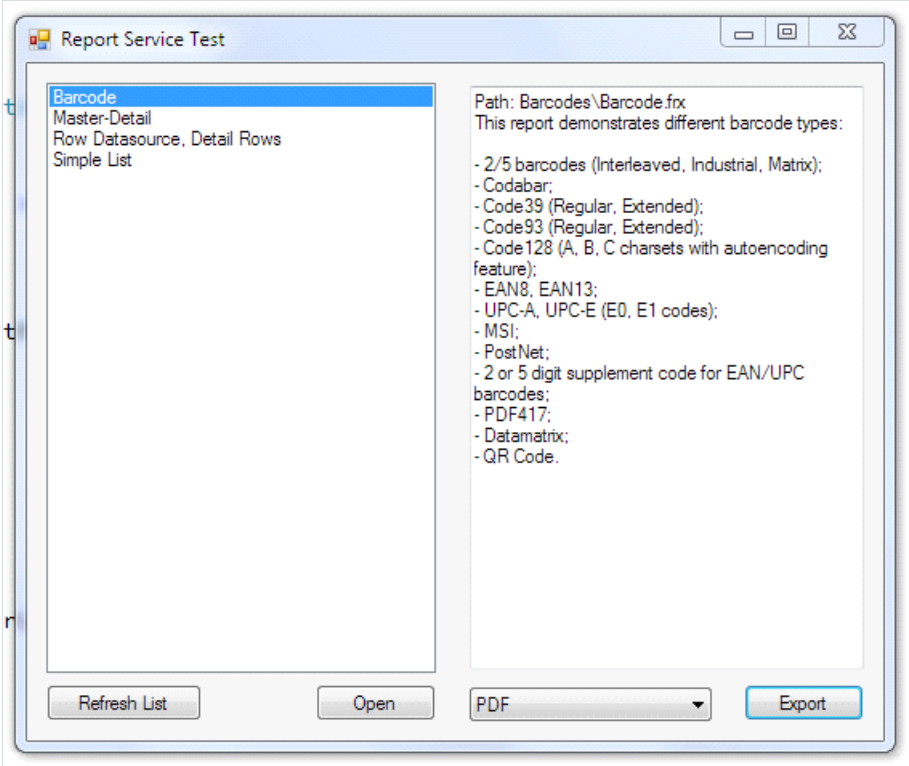
1. Open WCFServiceClient.csproj in Visual Studio
2. Right-click in Solution Explorer on "Service References:ReportService" and select "Configure Service Reference" in the popup



3. Review the service address, which should end with "/mex" (metadata exchange)



4. Compile and run an example.

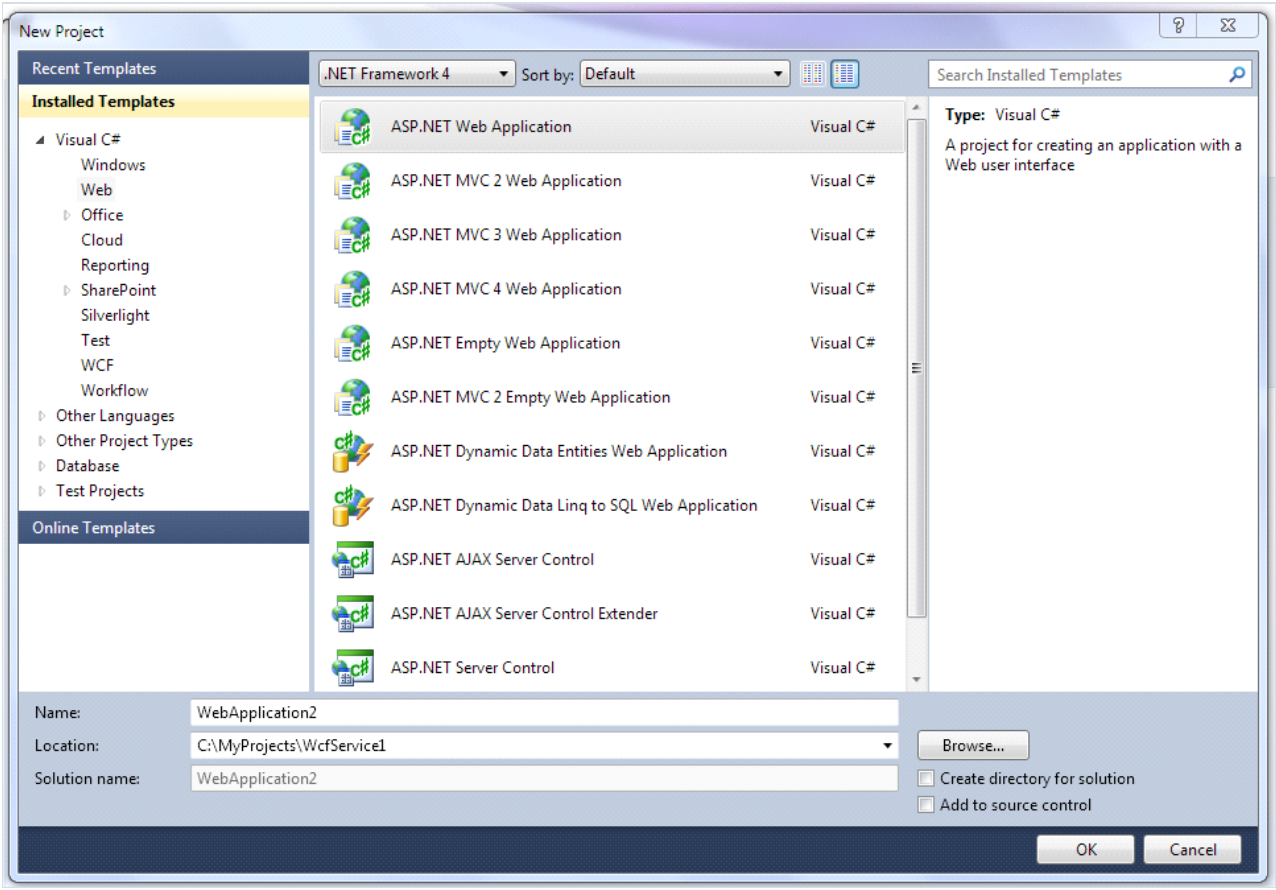

Report Service Test
Barcode
Master-Detail
Row Datasource, Detail Rows
Simple List
Path: Barcodes\Barcode.frx
This report demonstrates different barcode types:
- 2/5 barcodes (Interleaved, Industrial, Matrix);
- Codabar;
- Code39 (Regular, Extended);
- Code93 (Regular, Extended);
- Code128 (A, B, C charsets with autoencoding feature);
- EAN8, EAN13;
- UPC-A, UPC-E (E0, E1 codes);
- MSI;
- PostNet;
- 2 or 5 digit supplement code for EAN/UPC barcodes;
- PDF417;
- Datamatrix;
- QR Code.
Refresh List
Open
PDF
Export

Creating Web Service using FastReport.Service.dll

There is an easy way to implement a web service using the library FastReport.Service.dll (WCF Service Library), which is supplied with FastReport .Net.

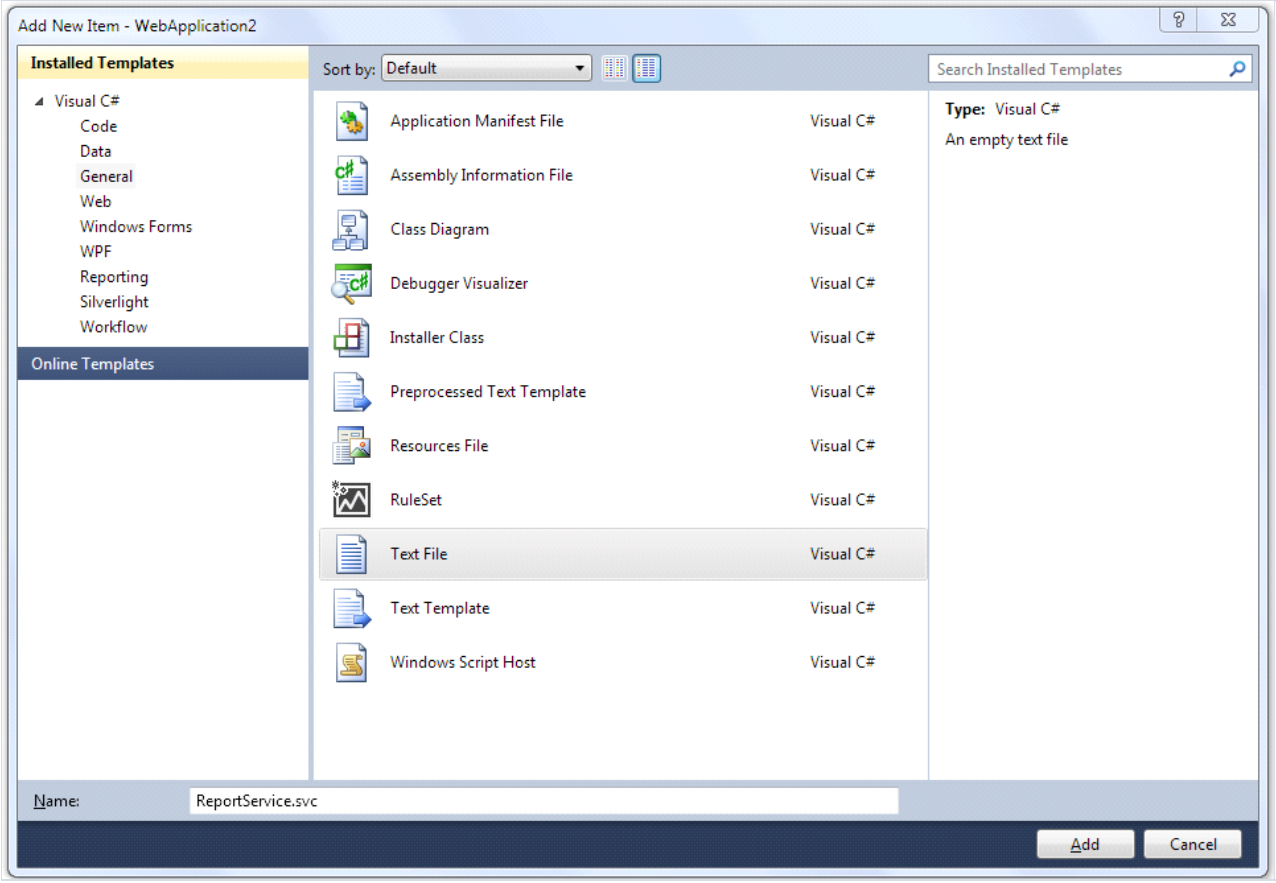
Our example is based on creating a simple web application with web service functions, but you can modify your existing project based on .NET Framework 4.0 or later.

Run Visual Studio and create a new ASP.NET Web Application project under .NET Framework 4.0.



Add references to libraries FastReport.dll, FastReport.Bars.dll, FastReport.Service.dll

Create a new text file with name ReportService.svc in the site root.



Add these lines to the file:

```
<%@ ServiceHost Service="FastReport.Service.ReportService" %>
<%@ Assembly Name="FastReport.Service" %>
```

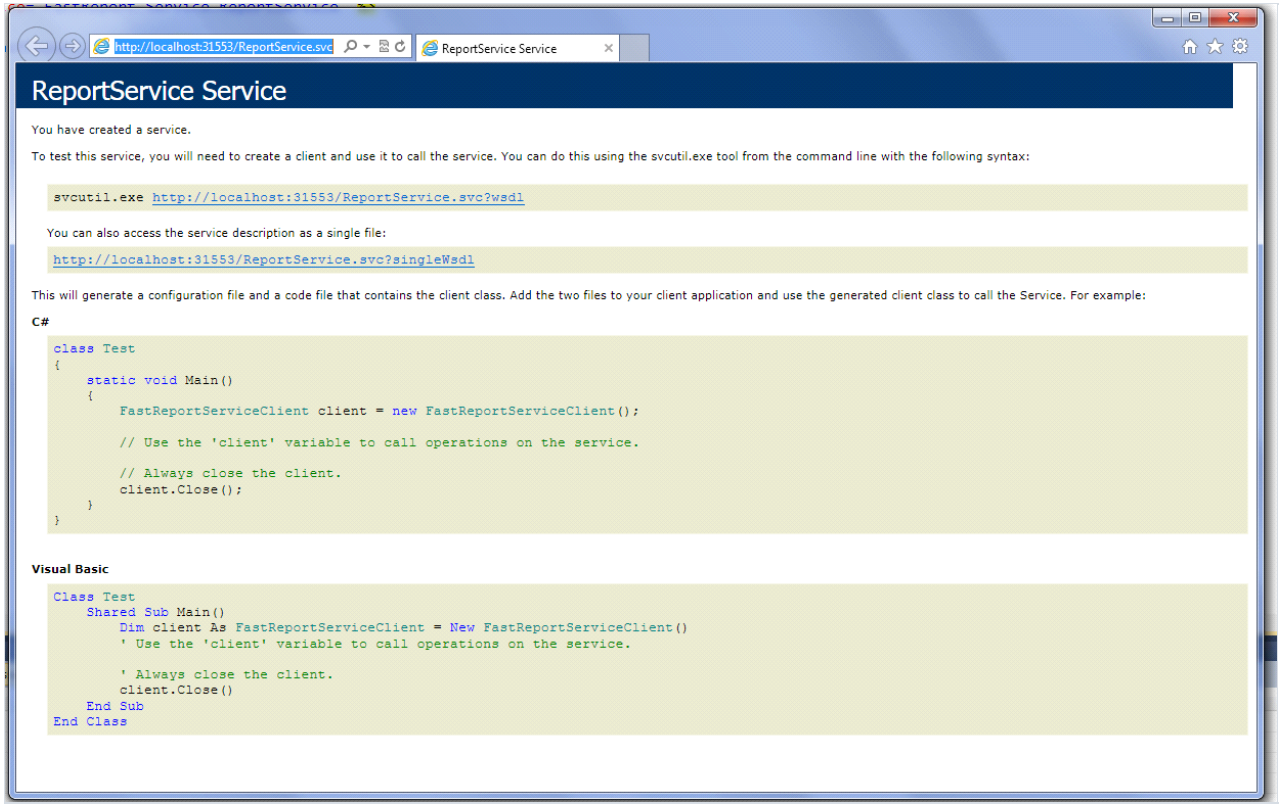
Open web.config and add this code in section:

```
<appSettings>
 <!-- path to folder with reports -->
 <add key="FastReport.ReportsPath" value="C:\Program files\FastReports\FastReport.Net\Demos\WCF" />
 <!-- name of connection string for reports -->
 <add key="FastReport.ConnectionStringName" value="FastReportDemo" />
 <!-- Comma-separated list of available formats PDF,DOCX,XLSX,PPTX,RTF,ODS,ODT,MHT,CSV,DBF,XML,TXT,FPX.
 You can delete any or change order in this list. -->
 <add key="FastReport.Gear" value="PDF,DOCX,XLSX,PPTX,RTF,ODS,ODT,MHT,CSV,DBF,XML,TXT,FPX" />
 </appSettings>
 <connectionStrings>
 <add name="FastReportDemo" connectionString="XsdFile=;XmlFile=C:\Program
Files\FastReports\FastReport.Net\Demos\Reports\nwind.xml"/>
 </connectionStrings>
 <system.serviceModel>
 <services>
 <service behaviorConfiguration="FastReportServiceBehavior" name="FastReport.Service.ReportService">
 <endpoint address="" binding="wsHttpBinding" contract="FastReport.Service.IFastReportService">
 <identity>
 <dns value="localhost" />
 </identity>
 </endpoint>
 <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
 </service>
 </services>
 <behaviors>
 <serviceBehaviors>
 <behavior name="FastReportServiceBehavior">
 <serviceMetadata httpGetEnabled="True" />
 <serviceDebug includeExceptionDetailInFaults="True" />
 </behavior>
 </serviceBehaviors>
 </behaviors>
 <bindings>
 <basicHttpBinding>
 <binding messageEncoding="Mtom"
 closeTimeout="00:02:00" openTimeout="00:02:00"
 receiveTimeout="00:10:00" sendTimeout="00:02:00"
 maxReceivedMessageSize="67108864" maxBufferSize="65536"
 transferMode="Streamed">
 <security mode="None">
 <transport clientCredentialType="None" />
 </security>
 </binding>
 </basicHttpBinding>
 </bindings>
 </system.serviceModel>
```

The key "FastReport.ReportsPath" should contain a path to the folder with the reports. You can set it to the demo folder «\FastReport.Net\Demos\WCF», for example.

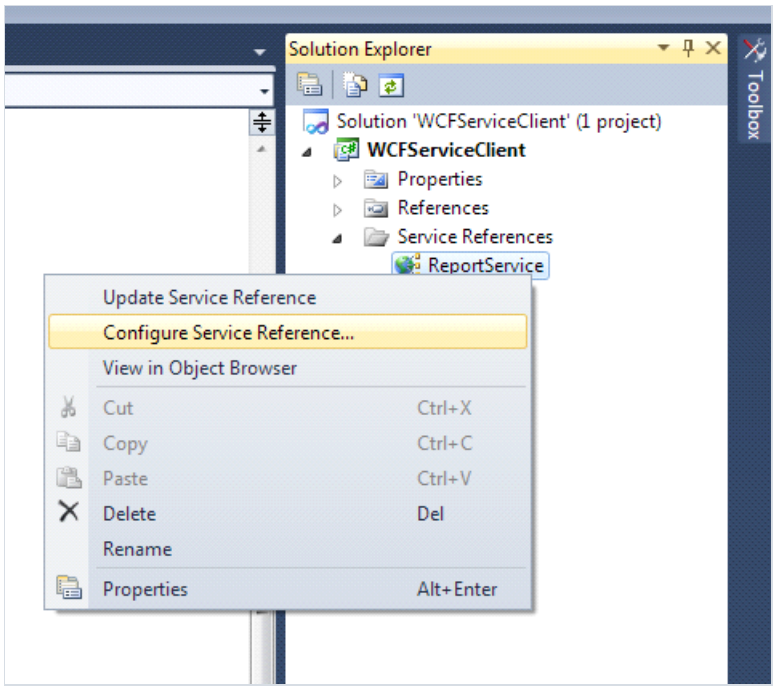
The key "FastReport.ConnectionStringName" should contain the connection string name. This line should be registered in section .

Let's run our site and check the availability of a Web service by accessing the file ReportService.svc.



When you deploy the project on the server, be sure to check that files FastReport.dll, FastReport.Bars.dll, FastReport.Service.dll are in the folder \bin.

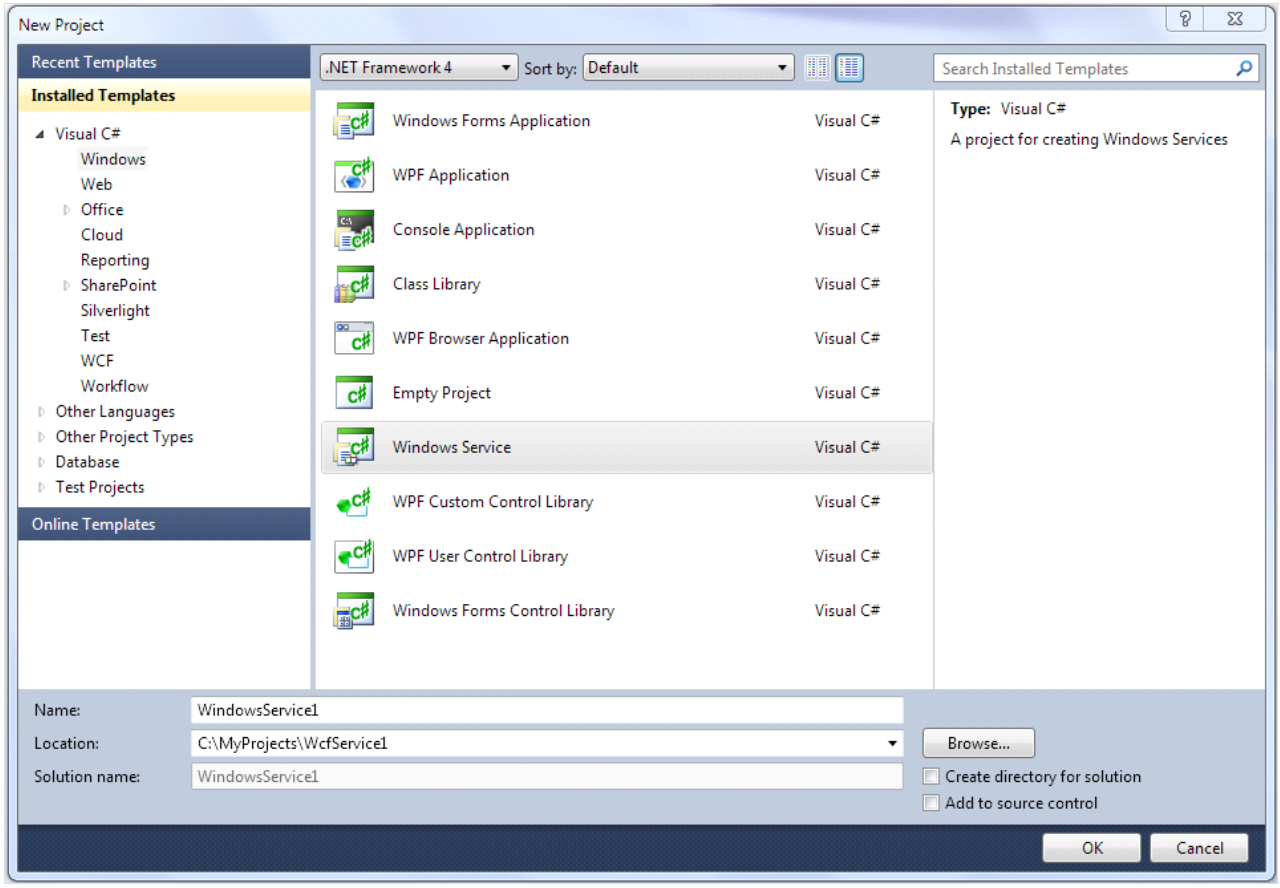
Examples of client programs can be found in the folders \FastReport.Net\Demos\C#\WCFClient and \FastReport.Net\Demos\C#\WCFWebClient. Open each project in Visual Studio, right-click on ReportService and select Configure Service Reference in the popup.



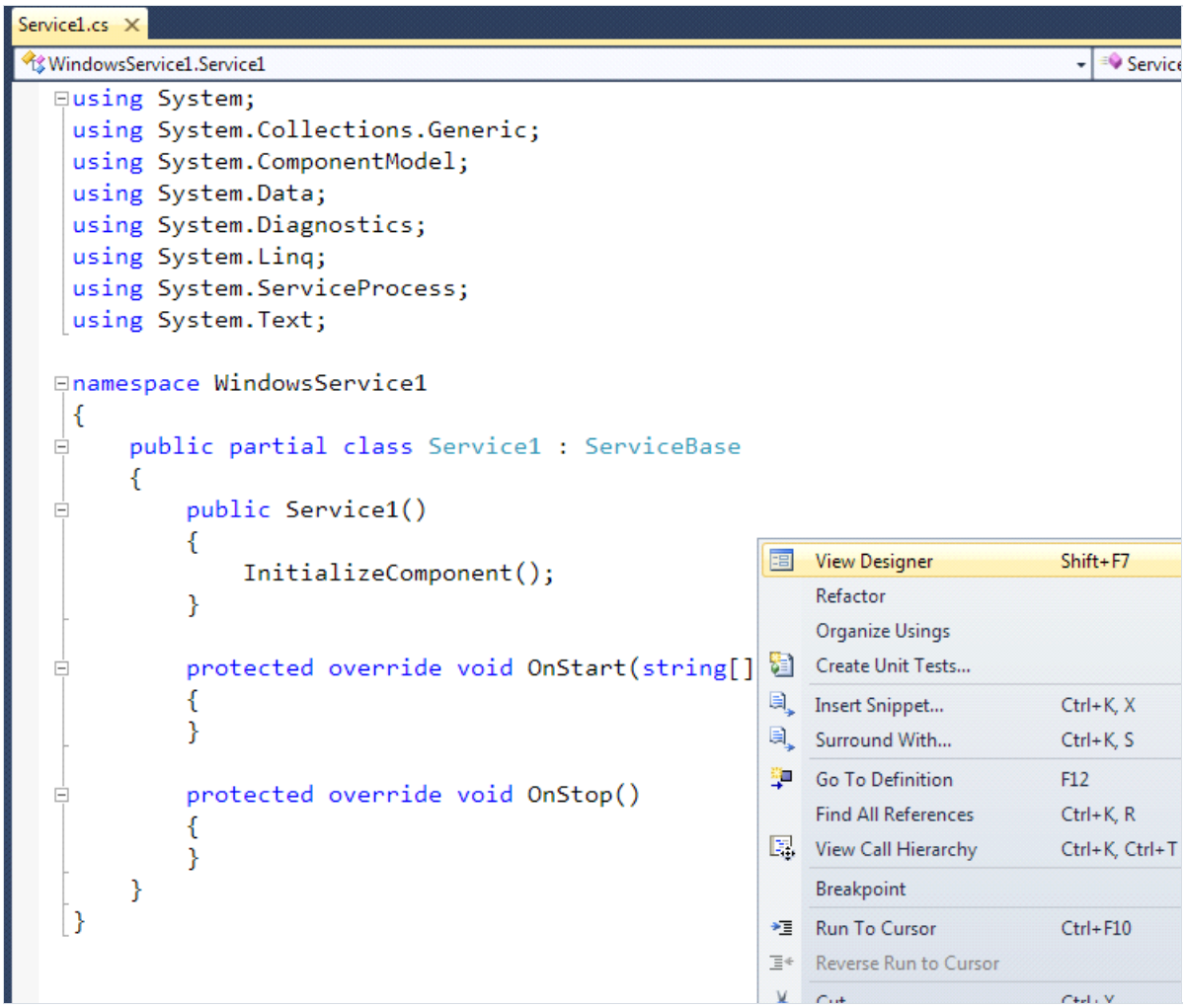
Specify the address of an existing web service in the configuration window.

Creating the WCF service hosted in windows service

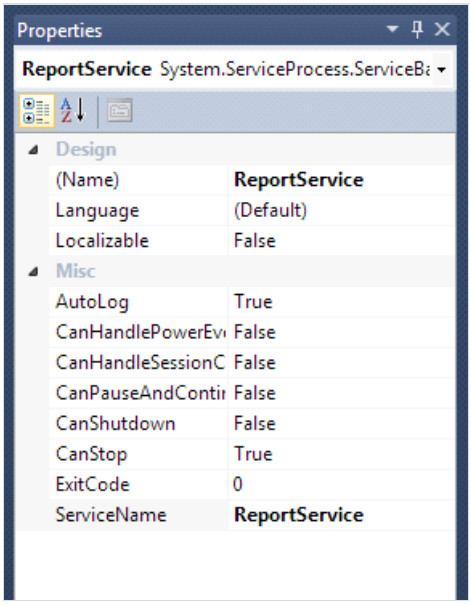
Open Visual Studio and create a project WindowsService.



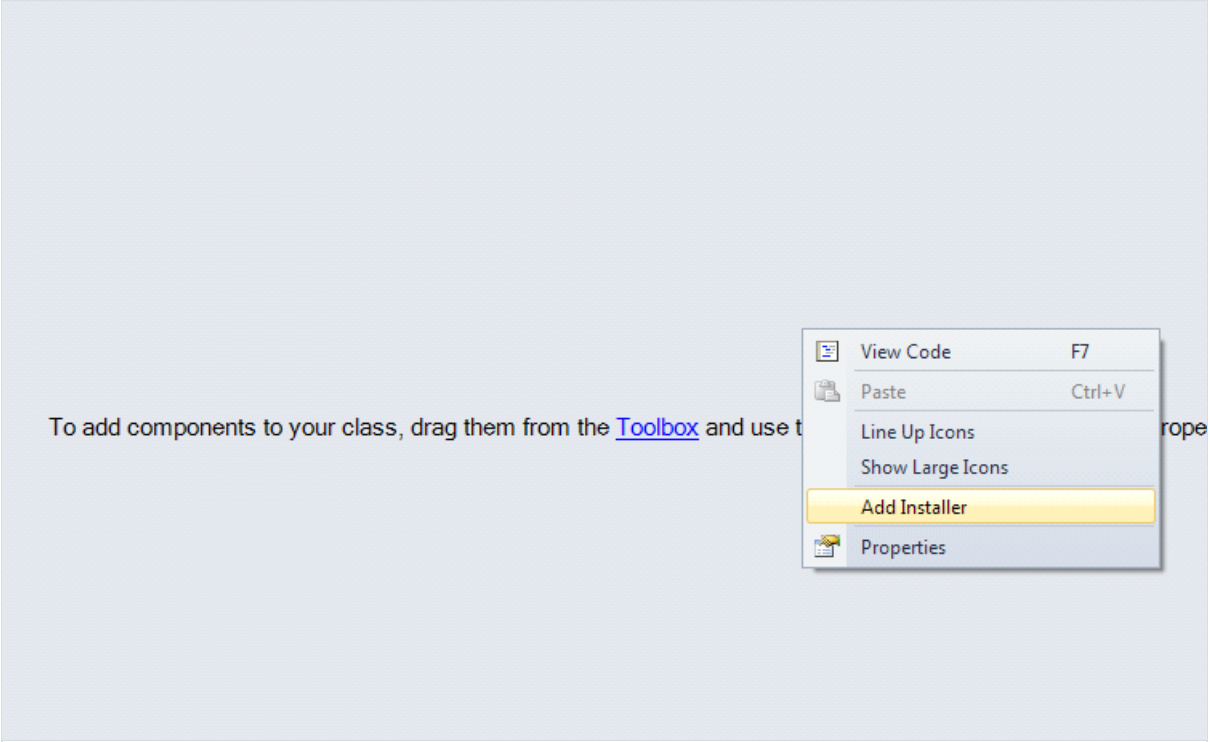
Open the designer of Service1.cs



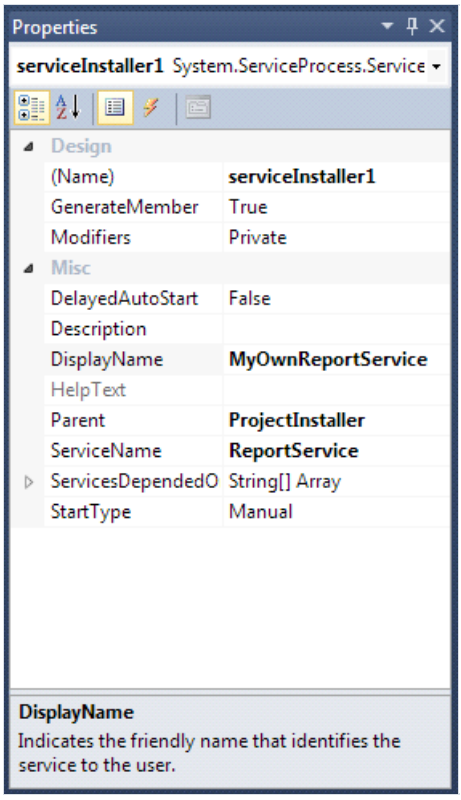
Change the name of the service to your own choice:



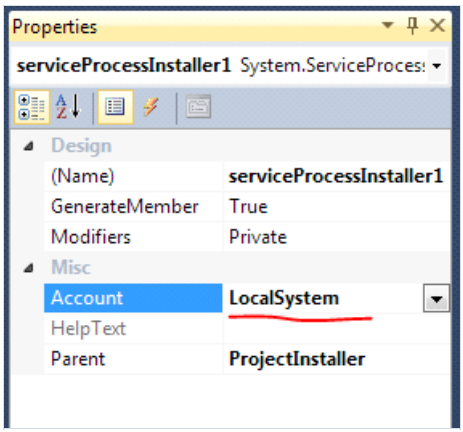
Right-click on window and select "Add Installer" in the popup:



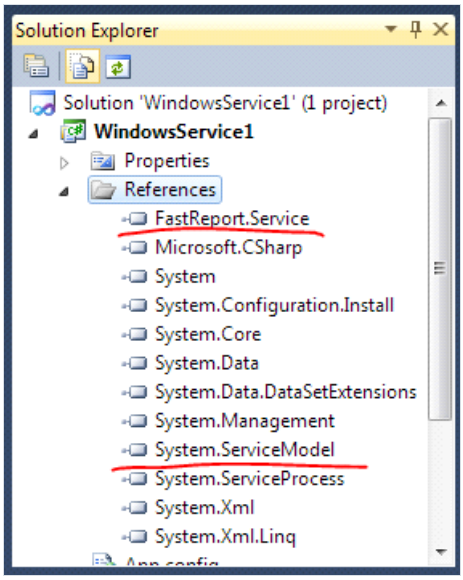
Edit the properties of the component serviceInstaller1 - set up a DisplayName.



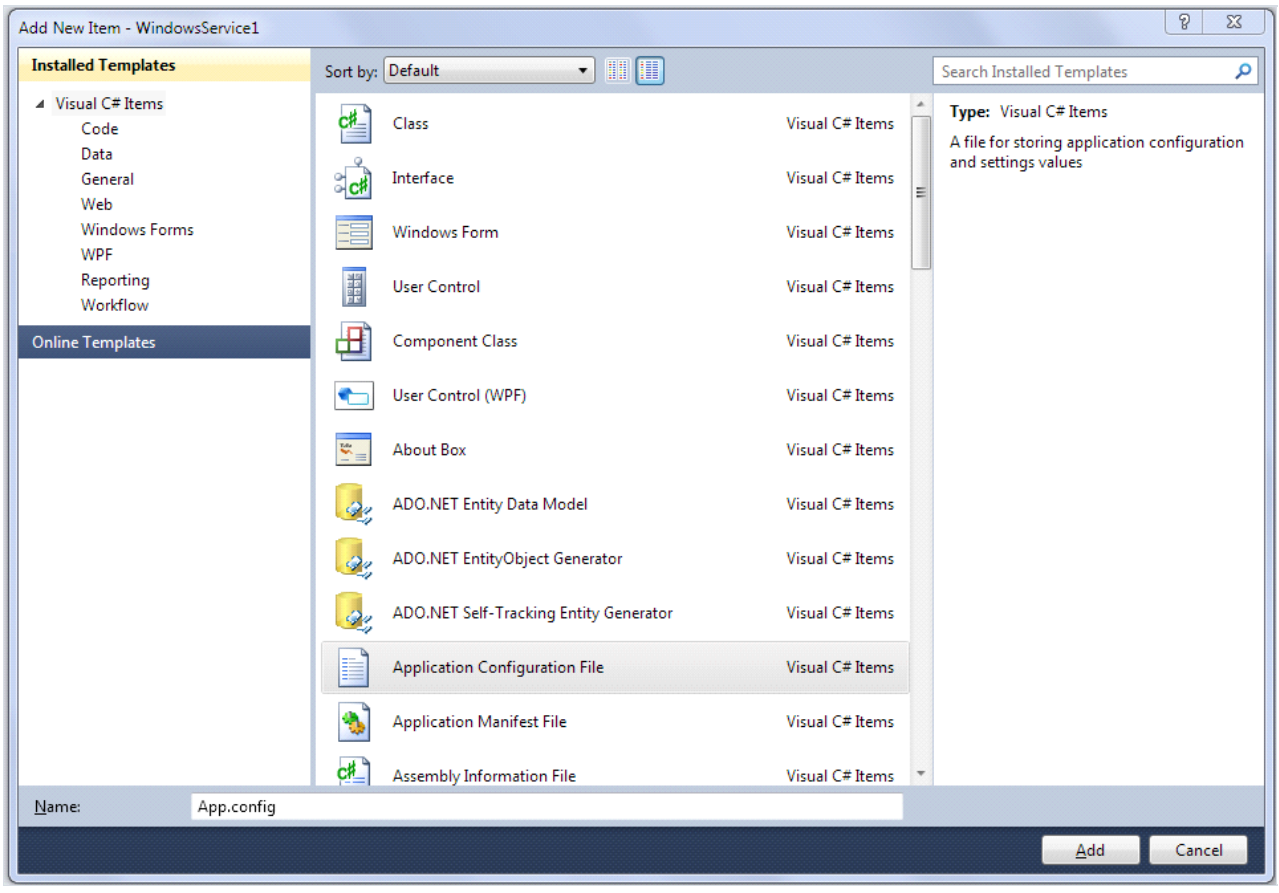
In the component properties of serviceProcessInstaller1 set the type of account for the service as LocalSystem.



Add references in the project to System.ServiceModel and FastReport.Service.dll :



Create an application configuration file:



Copy the following text into the new app.config file:

```
<?xml version="1.0"?>
<configuration>
 <appSettings>
 <!-- path to folder with reports -->
 <add key="FastReport.ReportsPath" value="C:\Program files\FastReports\FastReport.Net\Demos\WCF" />
 <!-- name of connection string for reports -->
 <add key="FastReport.ConnectionStringName" value="FastReportDemo" />
 <!-- Comma-separated list of available formats PDF,DOCX,XLSX,PPTX,RTF,ODS,ODT,MHT,CSV,DBF,XML,TXT,FPX.
 You can delete any or change order in this list. -->
 <add key="FastReport.Gear" value="PDF,DOCX,XLSX,PPTX,RTF,ODS,ODT,MHT,CSV,DBF,XML,TXT,FPX" />
 </appSettings>
 <connectionStrings>
 <add name="FastReportDemo" connectionString="XsdFile=;XmlFile=C:\Program
Files\FastReports\FastReport.Net\Demos\Reports\nwind.xml"/>
 </connectionStrings>
 <system.web>
 <compilation debug="true" />
 <membership defaultProvider="ClientAuthenticationMembershipProvider">
 <providers>
 <add name="ClientAuthenticationMembershipProvider"
type="System.Web.ClientServices.Providers.ClientFormsAuthenticationMembershipProvider,
System.Web.Extensions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" serviceUri=""
/>
 </providers>
 </membership>
 <roleManager defaultProvider="ClientRoleProvider" enabled="true">
 <providers>
 <add name="ClientRoleProvider" type="System.Web.ClientServices.Providers.ClientRoleProvider,
System.Web.Extensions, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" serviceUri=""
cacheTimeout="86400" />
 </providers>
 </roleManager>
 </system.web>
 <!-- When deploying the service library project, the content of the config file must be added to the
host's
 app.config file. System.Configuration does not support config files for libraries. -->
 <system.serviceModel>
 <services>
 <service behaviorConfiguration="FastReportServiceBehavior" name="FastReport.Service.ReportService">
 <endpoint address="" binding="wsHttpBinding" contract="FastReport.Service.IFastReportService">
 <identity>
 <dns value="localhost" />
 </identity>
 </endpoint>
 <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
 <host>
 <baseAddresses>
 <add baseAddress="http://localhost:8732/FastReportService/" />
 </baseAddresses>
 </host>
 </service>
 </services>
 <behaviors>
 <serviceBehaviors>
 <behavior name="FastReportServiceBehavior">
 <serviceMetadata httpGetEnabled="True" />
 <serviceDebug includeExceptionDetailInFaults="True" />
 </behavior>
 </serviceBehaviors>
 </behaviors>
 <bindings>
 <basicHttpBinding>
 <binding messageEncoding="Mtom"
 closeTimeout="00:02:00" openTimeout="00:02:00"
 receiveTimeout="00:10:00" sendTimeout="00:02:00"
```

```
 maxReceivedMessageSize="67108864" maxBufferSize="65536"
 transferMode="Streamed">
 <security mode="None">
 <transport clientCredentialType="None" />
 </security>
 </binding>
 </basicHttpBinding>
 </bindings>
 </system.serviceModel>
 <startup>
 <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0" />
 </startup>
</configuration>
```

Go to the editor of Service1.cs and add the line:

```
using System.ServiceModel;
```

Modify the class of service so it looks like:

```
public partial class ReportService : ServiceBase
{
 ServiceHost reportHost;

 public ReportService()
 {
 InitializeComponent();
 }

 protected override void OnStart(string[] args)
 {
 if (reportHost != null)
 reportHost.Close();
 reportHost = new ServiceHost(typeof(FastReport.Service.ReportService));
 reportHost.Open();
 }

 protected override void OnStop()
 {
 reportHost.Close();
 reportHost = null;
 }
}
```

You can install the service using the command line utility InstallUtil.exe, which comes with .NET Framework, for instance:

C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
"C:\MyProjects\WcfService1\WindowsService1\bin\Debug\WindowsService1.exe"

And you can start the service with the command:

net start ReportService

Open a web browser and check the address http://localhost:8732/FastReportService/, which was set in app.config in baseAddress. You can change the folder and port to your own choice.

Commands to stop and to uninstall the service:

```
net stop ReportService

C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe /u
"C:\MyProjects\WcfService1\WindowsService1\bin\Debug\WindowsService1.exe"
```

This example is located in folder "\Demos\C#\WCFWindowsService".

Extending FastReport functionality

Create your own connection types

Creating custom connection types

Data connections are used to add a data source to a report. This allows you to connect to data directly from the report, rather than using data provided by the application.

To create your own connection you need to do the following:

- create a connection class - a descendant of `FastReport.Data.DataConnectionBase` and implement some of its methods;
- create a connection editor - an heir of `FastReport.Data.ConnectionEditors.ConnectionEditorBase` and implement the user interface and methods `GetConnectionString`, `SetConnectionString`;
- register the connection in FastReport.

These steps will be described below.

DataConnectionBase class

To create your own connection, use the FastReport.Data.DataConnectionBase class. This class has the following set of methods that you must override when creating your own connection:

```
public abstract class DataConnectionBase : DataComponentBase
{
  protected virtual string GetConnectionStringWithLoginInfo(string userName, string password)
  public abstract string QuoteIdentifier(string value, DbConnection connection);
  public virtual DbConnection GetConnection();
  public virtual DbDataAdapter GetAdapter(string selectCommand, DbConnection connection,
    CommandParameterCollection parameters);
  public virtual ConnectionEditorBase GetEditor();
  public virtual Type GetParameterType();
  public virtual string GetConnectionId();
  public virtual string[] GetTableNames();
  public virtual void TestConnection();
  public virtual void FillTableSchema(DataTable table, string selectCommand,
    CommandParameterCollection parameters);
  public virtual void FillTableData(DataTable table, string selectCommand,
    CommandParameterCollection parameters);
}
```

Not all methods need to be overridden - some of them have a default implementation, which will be sufficient for most cases. So, if your connection uses objects of type DbConnection and DbDataAdapter (and this is standard for the vast majority of connections), you need to implement the following methods:

```
  public abstract string QuoteIdentifier(string value, DbConnection connection);
  protected virtual string GetConnectionStringWithLoginInfo(string userName, string password)
  public virtual DbConnection GetConnection();
  public virtual DbDataAdapter GetAdapter(string selectCommand, DbConnection connection,
    CommandParameterCollection parameters);
  public virtual ConnectionEditorBase GetEditor();
  public virtual Type GetParameterType();
  public virtual string GetConnectionId();
  public virtual string[] GetTableNames();
```

If your connection does not work with such objects, but, for example, is a bridge between the application server and the report, you will need to implement the following methods:

```
  public abstract string QuoteIdentifier(string value, DbConnection connection);
  public virtual ConnectionEditorBase GetEditor();
  public virtual Type GetParameterType();
  public virtual string GetConnectionId();
  public virtual string[] GetTableNames();
  public virtual void TestConnection();
  public virtual void FillTableSchema(DataTable table, string selectCommand,
    CommandParameterCollection parameters);
  public virtual void FillTableData(DataTable table, string selectCommand,
    CommandParameterCollection parameters);
```

Below is a description of each method.

ConnectionString propery

Each connection type has its own set of parameters, such as database path, SQL dialect, username and password. The ConnectionString property, which has a string type, is used to store the parameters.

This property is used as follows:

- in the GetConnection method when creating an object of type DbConnection;
- In the connection editor. In the latter case, individual parameters are selected from the connection string (e.g., the path to the database file). This uses a class of type `DbConnectionStringBuilder`, specific to each connection type. For example, for MS SQL it is `SqlConnectionStringBuilder`.

QuoteIdentifier method

An identifier (table or column name) is passed to this method. The method must return a quoted identifier. Quotation marks are specific to each database. For example, MS Access uses square brackets:

```
select * from [Table with long name]
```

The method in `MsAccessDataConnection` looks like this:

```
public override string QuoteIdentifier(string value, DbConnection connection)
{
  return "[" + value + "]";
}
```

You must override this method. Refer to the manual for this connection type to find out which characters are used to refer to tables with long names.

GetConnectionStringWithLoginInfo method

This method should take the existing connection string (from the `ConnectionString` property), include the username and password information, and return the modified string. This method is used if the connection settings specify "Ask for password when connecting".

For example, the implementation of this method for `MsSqlDataConnection` looks like this:

```
protected override string GetConnectionStringWithLoginInfo(string userName, string password)
{
  // get the existing connection string
  SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(ConnectionString);

  // include the username and password
  builder.IntegratedSecurity = false;
  builder.UserID = userName;
  builder.Password = password;

  // return the modified string
  return builder.ToString();
}
```

You have to override this method.

GetConnection method

The method returns a new object of type `DbConnection`, specific to this connection. This object is used to connect to the database when the table needs to be filled with data.

The connection parameters are taken from the `ConnectionString` property. For example, for `MsSqlDataConnection`, the method looks like this:

```
public override DbConnection GetConnection()
{
  return new SqlConnection(ConnectionString);
}
```

Most of the time you have to override this method. It is used in two other methods, `FillTableSchema` and `FillTableData`. If your connection type does not use `DbConnection` and `DbDataAdapter` objects to get information about the table and load data into it, you may not override this method. In that case, you must override the `FillTableSchema` and `FillTableData` methods.

GetAdapter method

The method returns a new object of type `DbDataAdapter`, specific to this connection. This object is used to fill the table with data.

The following parameters are passed to the method:

Parameter	Description
selectCommand	SQL query text.
connection	Object of type `DbConnection`, created in the `GetConnection` method.
Parameters	Query parameters, if defined.

Let's look at an example implementation of this method in `MsSqlDataConnection`:

```
public override DbDataAdapter GetAdapter(string selectCommand, DbConnection connection,
  CommandParameterCollection parameters)
{
  SqlDataAdapter adapter = new SqlDataAdapter(selectCommand, connection as SqlConnection);
  foreach (CommandParameter p in parameters)
  {
    SqlParameter parameter = adapter.SelectCommand.Parameters.Add(p.Name, (SqlDbType)p.DataType, p.Size);
    parameter.Value = p.Value;
  }
  return adapter;
}
```

The method creates an adapter specific to MS SQL, fills the query parameters and returns the adapter. The parameters should be described in more detail. The query text may contain parameters. In this case, a collection of parameters in the parameters variable is passed to the method - these are the parameters defined in the FastReport designer when the query was created. You must add the parameters to the `adapter.SelectCommand.Parameters` list. Each parameter in the parameters collection has the following properties:

Property	Description
Name	Parameter name.
DataType	The data type of the parameter. This is an int type property; you must cast it to the type used for the parameters in this connection.
Size	The size of the parameter data.
Value	The value of the parameter.

Most of the time you have to override this method. It is used in two other methods, `FillTableSchema` and `FillTableData`. If your connection type doesn't use the `DbConnection` and `DbDataAdapter` objects to get information about the table and load data into it, you may not override this method. In that case, you must override the `FillTableSchema` and `FillTableData` methods.

GetEditor method

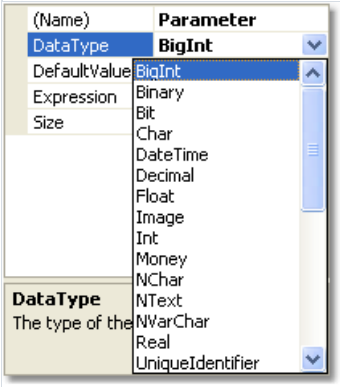
This method returns an instance of the editor for the given connection type. The implementation of the editor will be discussed in "Connection Editor".

An example implementation of this method in `MsSqlDataConnection`:

```
public override ConnectionEditorBase GetEditor()
{
  return new MsSqlConnectionEditor();
}
```

GetParameterType method

The value returned by this method is used in the query parameter editor to select the data type of the parameter:



For example, for `MsSqlDataConnection` the parameter is of type `SqlDbType`:

```
public override Type GetParameterType()
{
  return typeof(SqlDbType);
}
```

You have to override this method.

GetConnectionId method

The value returned by this method is used in the "Connection Wizard" to display short information about the connection.

This value usually contains the name of the connection type and the name of the database. In `MsSqlDataConnection` the implementation of the method looks like this:

```
public override string GetConnectionId()
{
  SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(ConnectionString);
  string info = builder.InitialCatalog;
  if (String.IsNullOrEmpty(info))
    info = builder.AttachDBFilename;
  return "MS SQL: " + info;
}
```

Note that `SqlConnectionStringBuilder` is used to parse the connection string (`ConnectionString` property).

You have to override this method.

GetTableNames method

This method returns a list of tables and views in the database. As a rule, the `GetSchema` method of the `DbConnection` object is used for this, which is returned by the `GetConnection` method.

This method has a standard implementation. In case the standard implementation is incompatible with your connection type (i.e., it does not return a list of database entities), you will have to override this method. Consider the `MsSqlDataConnection` implementation of this method as an example:

```
public override string[] GetTableNames()
{
  List<string> list = new List<string>();
  GetDBObjectNames("BASE TABLE", list);
  GetDBObjectNames("VIEW", list);
  return list.ToArray();
}

private void GetDBObjectNames(string name, List<string> list)
{
  DataTable schema = null;
  using (DbConnection connection = GetConnection())
  {
    connection.Open();
    schema = connection.GetSchema("Tables", newstring[] { null, null, null, name });
  }
  foreach (DataRow row in schema.Rows)
  {
    list.Add(row["TABLE_NAME"].ToString());
  }
}
```

TestConnection method

This method is called when you click the "Test" button in the connection settings.

The method has a default implementation that creates the connection and tries to open it:

```
public virtual void TestConnection()
{
  DbConnection conn = GetConnection();
  if (conn != null)
  {
    try
    {
      conn.Open();
    }
    finally
    {
      conn.Dispose();
    }
  }
}
```

If the test was successful, the method does nothing. Otherwise, an exception occurs when the connection is opened, which is handled in the "Connection Wizard" and gives a window with an error text.

As a rule, you don't need to override this method. You may need it if your connection does not use a `DbConnection` object to access the database (and therefore you do not return it in the `GetConnection` method).

FillTableSchema method

The method fills in the table schema (i.e. field names and types).

The following parameters are passed to the method:

Parameter	Description
table	The DataTable object whose schema you want to fill.
selectCommand	Query text in SQL.
parameters	Query parameters, if defined.

The method has a default implementation that uses objects returned by the `GetConnection` and `GetAdapter` methods:

```
public virtual void FillTableSchema(DataTable table, string selectCommand, CommandParameterCollection
parameters)
{
 using (DbConnection conn = GetConnection())
 {
  OpenConnection(conn);

  // prepare select command
  selectCommand = PrepareSelectCommand(selectCommand, table.TableName, conn);

  // read the table schema
  using (DbDataAdapter adapter = GetAdapter(selectCommand, conn, parameters))
  {
   adapter.SelectCommand.CommandTimeout = CommandTimeout;
   adapter.FillSchema(table, SchemaType.Source);
  }
 }
}
```

In most cases, you do not need to override this method. You may need to do this if you don't use the `DbConnection` and `DbDataAdapter` objects to access the data (and therefore don't implement the `GetConnection` and `GetAdapter` methods).

FillTableData method

The method fills the table with data.

The following parameters are passed to the method:

Parameter	Description
table	The DataTable object to be filled.
selectCommand	Query text in SQL.
parameters	Query parameters, if defined.

The method has a default implementation that uses objects returned by the GetConnection and GetAdapter methods:

```
public virtual void FillTableData(DataTable table, string selectCommand, CommandParameterCollection
parameters)
{
 using (DbConnection conn = GetConnection())
 {
  OpenConnection(conn);

  // prepare select command
  selectCommand = PrepareSelectCommand(selectCommand, table.TableName, conn);

  // read the table
  using (DbDataAdapter adapter = GetAdapter(selectCommand, conn, parameters))
  {
   adapter.SelectCommand.CommandTimeout = CommandTimeout;
   table.Clear();
   adapter.Fill(table);
  }
 }
}
```

In most cases, you do not need to override this method. You may need to do this if you don't use the DbConnection and DbDataAdapter objects to access the data (and therefore don't implement the `GetConnection` and `GetAdapter` methods).

Connection Editor

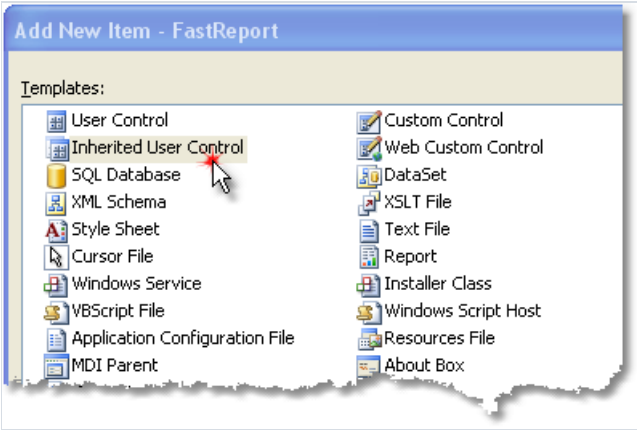
To edit the connection, use a control of type UserControl, which is displayed in the connection selection window (in the figure the control is highlighted with a red frame).

Each connection type has its own editor. All editors are inherited from the base class - `FastReport.Data.ConnectionEditors.ConnectionEditorBase` :

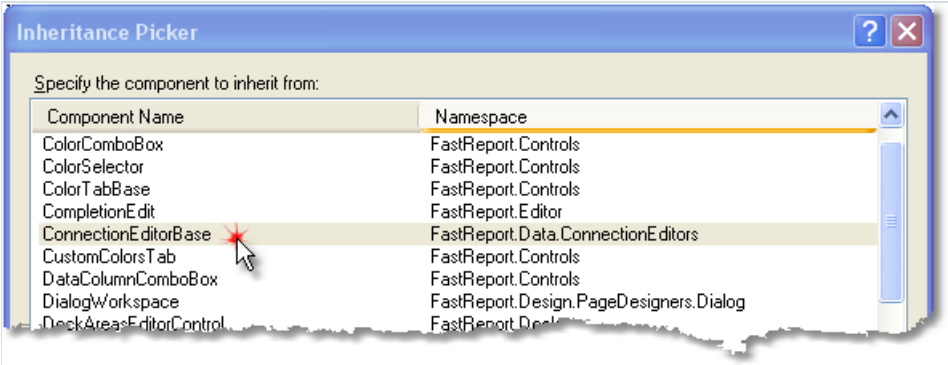
```
public class ConnectionEditorBase : UserControl
{
  protected virtual string GetConnectionString();
  protected virtual void SetConnectionString(string value);
}
```

To create your own editor, do the following:

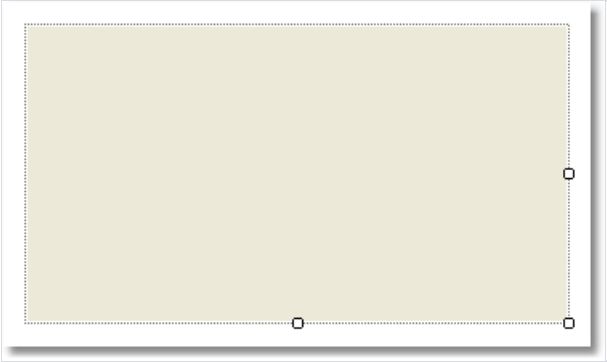
- Add a new control to the project (Add->User Control...) and select "Inherited User Control" in the "Add New Item" window:



- select the ConnectionEditorBase base class type:



A control that looks like this will be added to the project:



On the form of the editor, place the controls you need. Change the height of the editor to accommodate all the elements. The width of the editor is fixed and cannot be changed.

The methods of the base class allow you to fill the controls with values from the connection and, conversely, to pass values from the elements to the connection.

GetConnectionString method

This method is called when you close the editor window with the OK button. It should return a connection string that contains the values entered in the editor.

SetConnectionString method

This method is called the first time the editor is displayed. It must parse the connection string into its component parts and display their values in the editor. To parse a connection string, it is convenient to use a class of type DbConnectionStringBuilder, which is available in every connection type. For example, for MS SQL it is `SqlConnectionStringBuilder`.

Registering a connection in FastReport

In order for your connection to be used in the FastReport designer, it must be registered. There are two ways to do this.

Method 1: Your connection is part of your project (i.e. it is not in a separate .dll plugin). Registration is done using the following code:

```
FastReport.Utils.RegisteredObjects.AddConnection(typeof(MyDataConnection));
```

This code must be executed once in the entire lifetime of the application, before you start the designer.

Method 2. Your connection is contained in a separate .dll plugin. In this case you can connect the plugin to FastReport, and it will be loaded every time you work with the designer. To do this, you need to declare a public class like `FastReport.Utils.AssemblyInializerBase` in the plugin and register your connection in its constructor:

```
public class AssemblyInitializer : FastReport.Utils.AssemblyInitializerBase
{
  public AssemblyInitializer()
  {
    FastReport.Utils.RegisteredObjects.AddConnection(typeof(MyDataConnection));
  }
}
```

When you load your plugin, FastReport will find classes like AssemblyInitializerBase and initialize them.

To attach a plugin to FastReport, bring up the designer and select the `Settings...` item in the View menu. In the window that opens, select the "Plugins" tab and add your .dll plugin.

This can also be done by adding the module name to the FastReport configuration file. This file is created in the user directory by default:

```
C:\Documents and Settings\User\Local Settings\Application Data\FastReport\FastReport.config
```

Add a plugin inside the `Plugins` tag:

```
<?xml version="1.0" encoding="utf-8"?>
<Config>
  ...
  <Plugins>
    <Plugin Name="c:\Program Files\MyProgram\MyPlugin.dll"/>
  </Plugins>
</Config>
```

Contacts and technical support

You can always ask questions about using the product by email, or by using the form on the website.

We also welcome your suggestions on how to improve our product.