# Creating web applications with Entity Framework

This chapter will describe how to create web applications with Firebird as the back-end, using Microsoft Entity Framework and the Visual Studio 2015 environment.

In this chapter, we examine the specifics of creating a web application with this framework. The basic principles for working with Entity Framework and Firebird are described in the previous chapter, Creating Applications with Entity Framework.

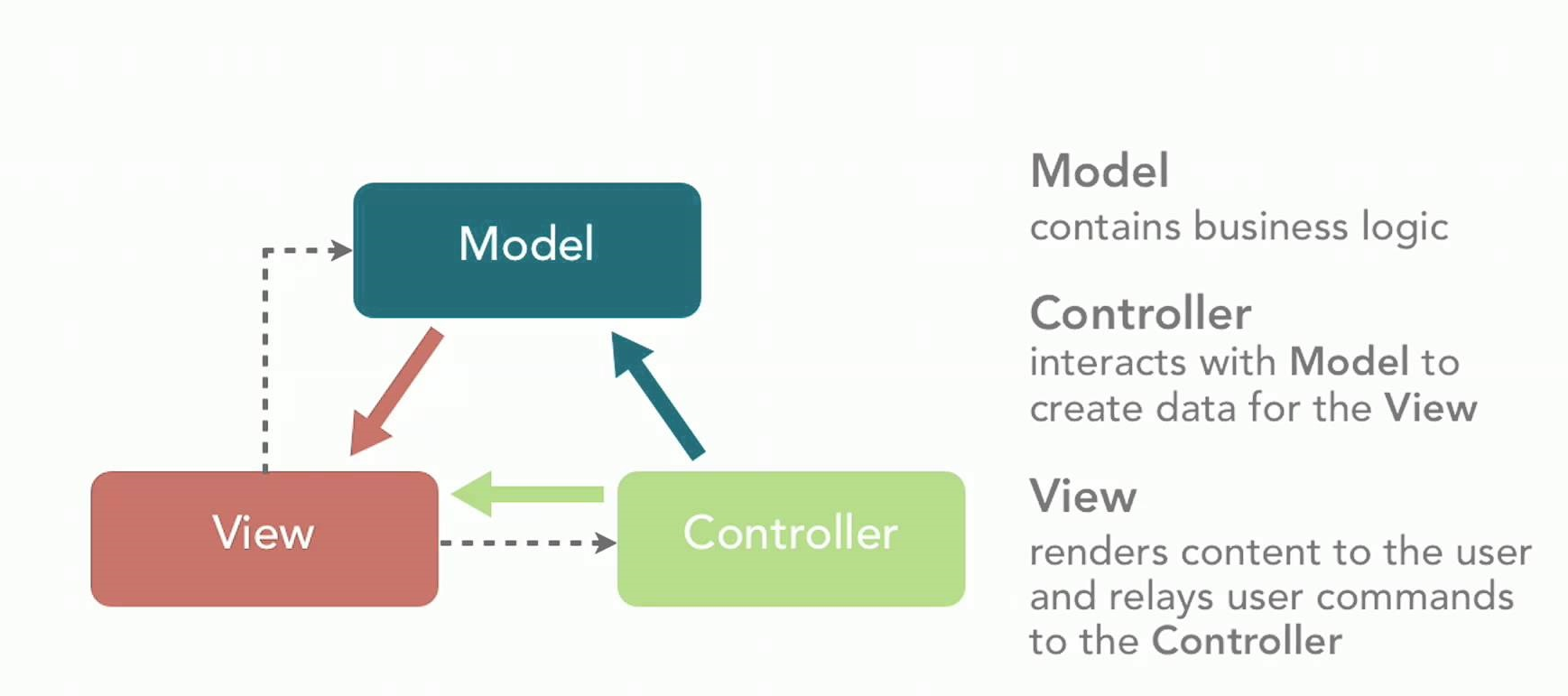
The .NET platform offers two main frameworks for creating web applications developed as "active server pages" (ASP): ASP.NET Web Forms and ASP.NET MVC. I prefer using the MVC pattern so it is this technology that we will be examining.

The **ASP.NET MVC** platform is a framework for creating websites and web applications through implementing the model-view-controller (MVC) pattern.

The concept behind the MVC pattern breaks down an application into three parts:

* **Controller**. Controllers work with the model and provide interaction with the user. They also provide view options for displaying the user interface. In an MVC application, views only display data while the controller handles the input and responds to user activities. As an example, the controller can process strings values in a query and send them to the model, which can use these values to send a query to the database.
* **View** —the visual part of application's user interface. The user interface is usually created to reflect the data from the model.
* **Model**. Model objects are the parts of the application that implement the logic for working with the application data. Model objects typically receive the status of the model and save it in the database.

Interaction between these components can be represented in the following general diagram:



The MVC pattern enables the creation of applications whose logical aspects – input, business and interface – are separated but interact closely with one another. The diagram illustrates the location of each logic type in the application:

* The user interface in the view
* The input logic the controller
* The business logic in the model

This separation allows you to work with complex structures while developing the application because it ensures discrete implementation of each aspect. The developer can focus on creating a view separately from implementing the business logic.

More comprehensive information about the ASP.NET MVC technology can be found at the website of the [ASP.NET community](http://www.asp.net/mvc/overview).

## Software Stack

Along with the libraries for working with Firebird, Entity Framework and MVC.NET, we will need a number of JavaScript libraries to support a responsive interface, such as jquery, jquery-ui, Bootstrap, jqGrid. In this example, we have tried to make a web application whose interface is similar to a desktop UI, by employing grids for views and modal windows for data input.

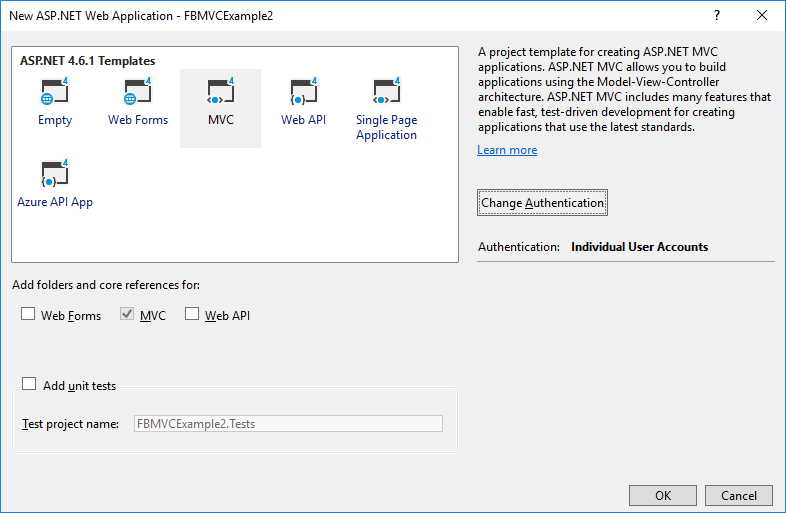
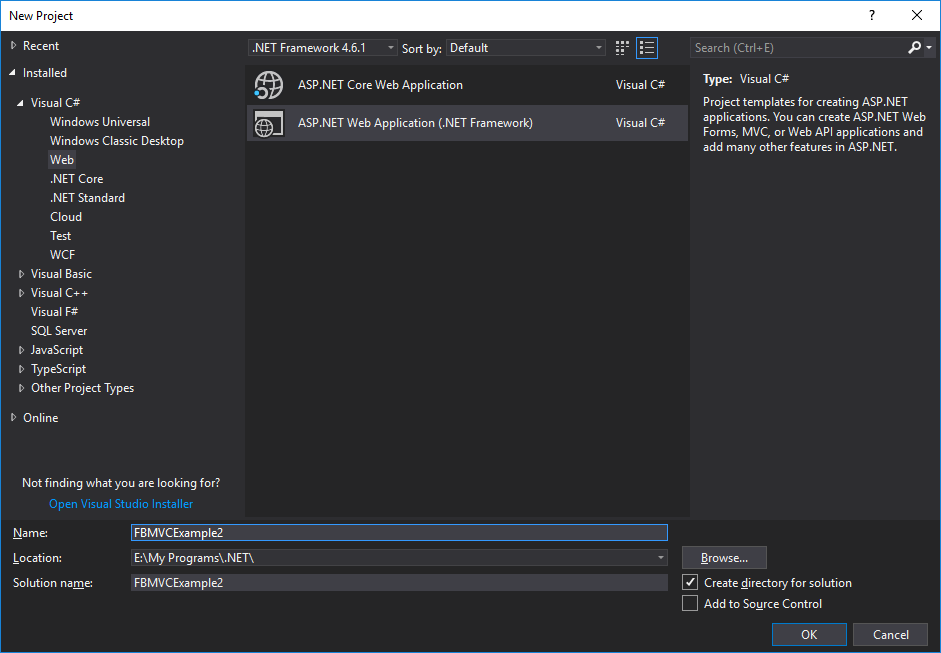
## Preparing Visual Studio 2015 for working with Firebird

For Visual Studio to work with the Firebird DBMS, you will have to take some additional steps that are described in detail in the "Preparing Visual Studio 2015 for working with Firebird" section of the previous chapter "Creating Applications with Entity Framework".

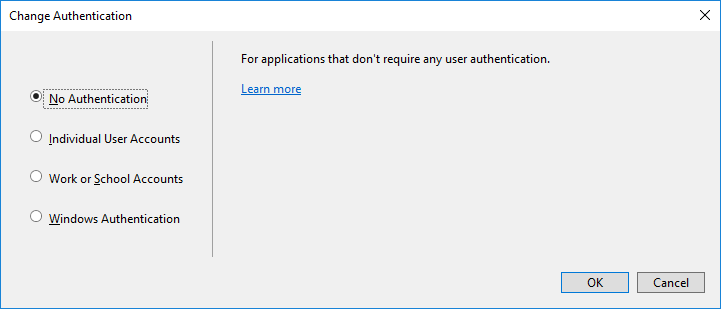
## Creating a project

In this part, we will show how use the Visual Studio wizards to create the framework of an MVC.NET application.

Open File -> New -> Project in Visual Studio 2015 and create a new project named FBMVCExample.



We will create a web application no authentication for now, so click the Change Authentication button to disable authentication. We will get back to this issue a bit later.



The project that you create will have virtually no functionality, but it already has its basic structure, described briefly in the following table:

|  |  |
| --- | --- |
| **Folder or file** | **Description** |
| /App\_Data | Folder where private web application data, such as XML files or database files, are located. |
| /App\_Start | Folder containing some main configuration settings for the project, including the definitions of routes and filters. |
| /Content | Static content goes in here, such as CSS files and images. It is an optional convention. You can store CSS files anywhere you want. |
| /Controllers | Controller classes are saved here. It is an optional convention. You can store controller classes anywhere. |
| /Models | View model and business model classes are saved here although it is better for all applications (except for the simplest ones) to define a business model in a separate project. It is an optional convention. You can store model classes anywhere you like. |
| /Scripts | This folder is used to store the JavaScript libraries being used in the application. By default, Visual Studio adds jQuery libraries and several other popular JavaScript libraries. It is an optional convention. |
| /Views | This folder stores the views and partial views. They are commonly grouped together in sub-folders name for the controllers they are connected with. |
| /Views/Shared | This folder stores layouts and views not specific to one controller. |
| /Views/Web.config | A configuration file containing the configuration information that ensures that views are processed within ASP.NET and not by the IIS web server. Also contains the namespaces imported into views by default. |
| /Global.asax | The global class of an ASP.NET application. A configuration for a route is registered in the file with its code (Global.asax.cs). This file contains also any code that is supposed to be executed during the launch or termination of an application or when an unhandled exception arises. |
| /Web.config | The configuration file for the application. |

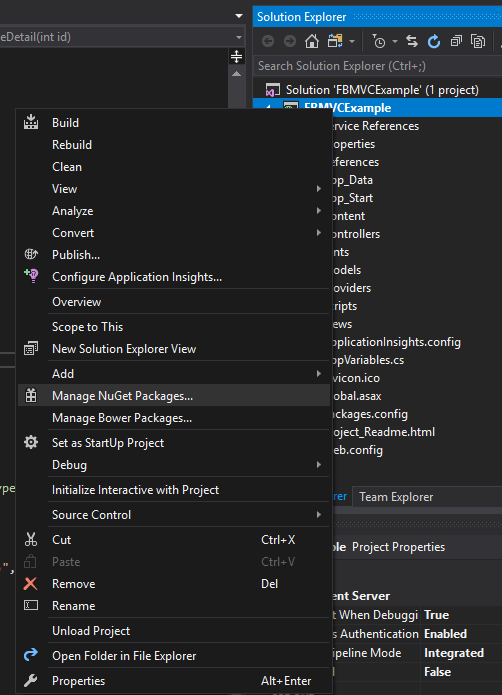
We will use the NuGet package manager to add the missing packages:

* FirebirdSql.Data.FirebirdClient
* EntityFramework (automatically added by the wizard)
* EntityFramework.Firebird
* Bootstrap (automatically added by the wizard)
* jQuery (automatically added by the wizard)
* jQuery.UI.Combined
* Respond (automatically added by the wizard)
* Newtonsoft.Json
* Moderninzr (automatically added by the wizard)
* Trirand.jqGrid

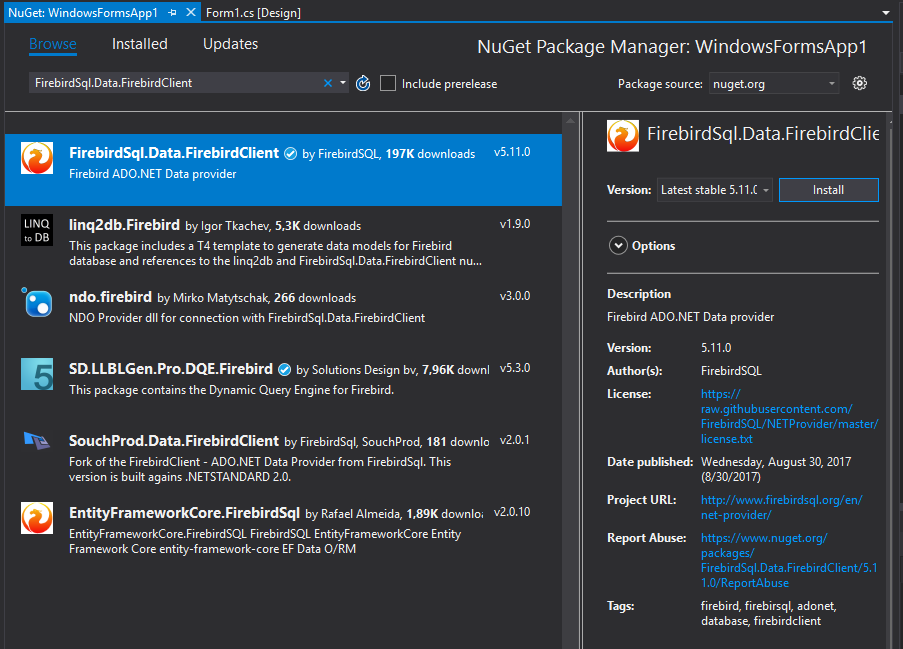
**Note**

Not all packages provided by NuGet are the latest version of the libraries. It is especially true for JavaScript libraries. You can install the latest versions of JavaScript libraries using a content delivery network (CDN) or by just downloading them and replacing the libraries provided by NuGet.

Right-click the project name in Solution Explorer and select the Manage NuGet Packages item in the drop-down menu.



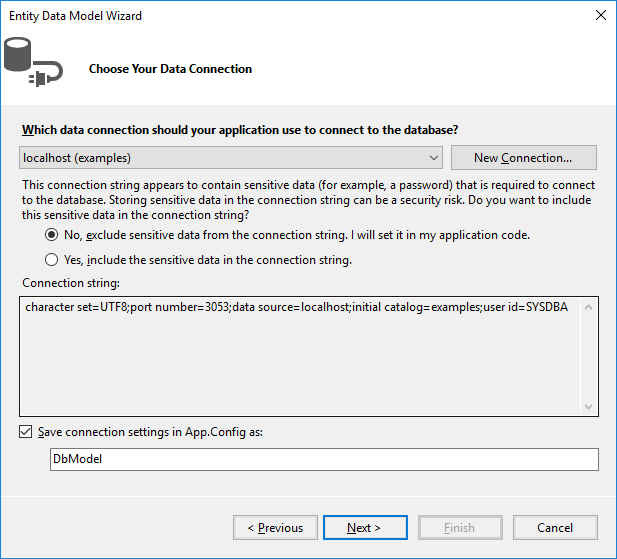
Find and install the necessary packages in the package manager.



## Creating an EDM

If you already have a Windows Forms application that uses Entity Framework, you can just model classes to the Models folder. Otherwise, you have to create them from scratch. The process of creating an EDM is described in the previous chapter "Creating Applications with Entity Framework" (see the "Creating an EDM" section).

There is one more small difference: your response to the EDM wizard's question about how to store the connection string:



When we create a web application, all users will work with the database using a single account, so select Yes for this question. Any user with enough privileges can be specified as the username. It is advisable not to use the SYSDBA user because it has more privileges than are required for a web application to work. You can always change the username in the application when it is ready for testing and deployment, by just editing the connection string in the AppName.exe.conf application configuration file. The connection string will be stored in the connectionStrings section and will look approximately as follows:

<add name="DbModel"

connectionString="character set=UTF8; data source=localhost;

initial catalog=examples; port number=3050;

user id=sysdba; dialect=3; isolationlevel=Snapshot;

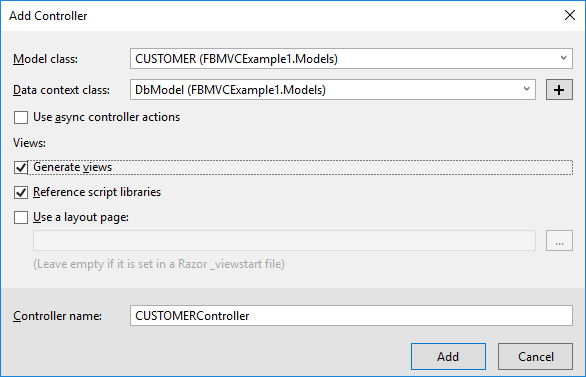
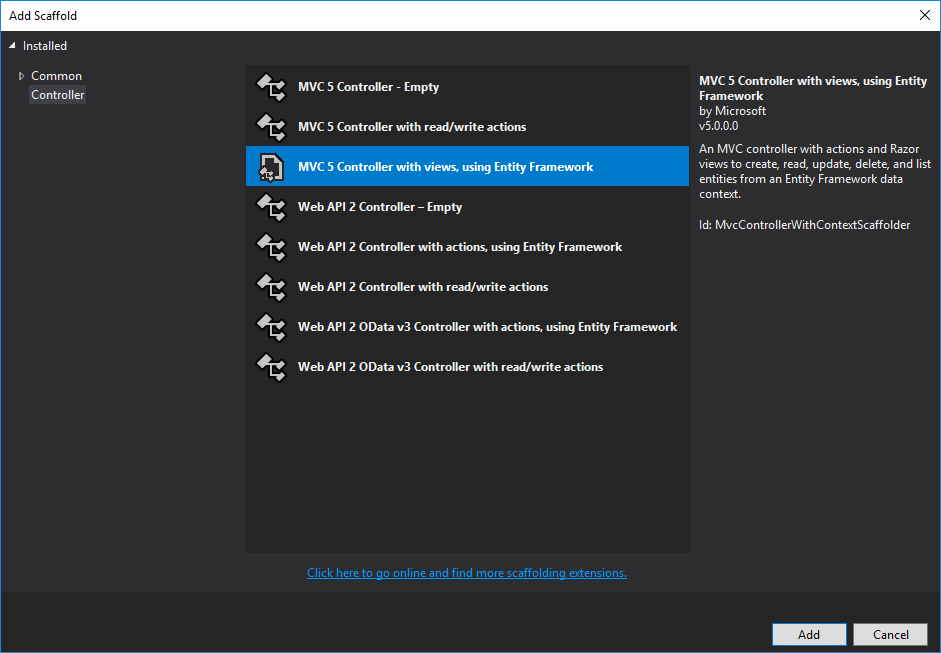
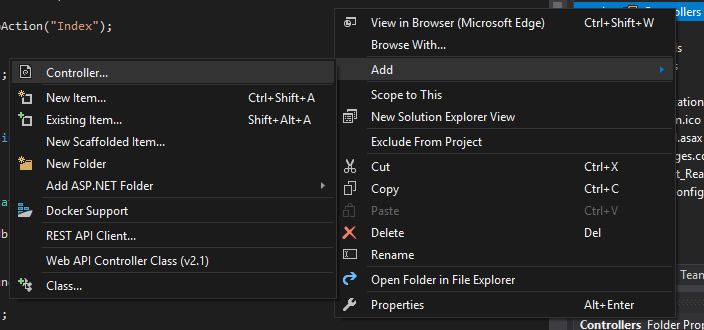
pooling=True; password=masterkey;"

providerName="FirebirdSql.Data.FirebirdClient" />

## Create a user interface for modules

### Controllers

And now let us create our first controller. It will be used to display and input customer data.



Once it is done, the controller CustomerController will be created, along with five views:

1. displays the customer list
2. displays the customer details for one customer
3. create (add) customer form
4. edit customer form
5. delete customer form

The Ajax technology and the jqGrid library will be used extensively in our project, so just the view for displaying the customer list as a table will be enough for our purposes. The rest of the operations will be performed with jqGrid.

The customer list may turn out to be quite big. The entire list from a big table is usually not returned in web applications because it could make the process of loading the page seriously slow. Instead, the data are usually split into pages or are dynamically loaded when the user scrolls down to the end of the page (or grid). We will use the first option in our example.

Another characteristic of web applications is that they do not keep any permanent connections to the database because the life of the page generation script is no longer than the time it takes to generate a response to the user request. A connection to the database is actually a rather expensive resource so we have to save it. Of course, there is a connection pool for reducing the time it takes to establish a connection to the database, but it is still advisable that a connection to the database is established only when it is really necessary. One of the ways to reduce the amount of interaction with the database is to check the correctness of the user input in the browser. Fortunately, modern HTML5 and JavaScript libraries can do that. For instance, you can check the presence of a required field or the maximum length of string fields in the input form.

So let us change the CustomerController controller so that it works with jqGrid. The code is quite lengthy, so track the comments to get a sense of the way the controller works.

**public class** CustomerController : Controller

{

**private** DbModel db = **new** DbModel();

*// Display view*

**public** ActionResult Index()

{

**return** View();

}

*// Receiving data in JSON for grid*

**public** ActionResult GetData(**int**? rows, **int**? page, **string** sidx, **string** sord,

**string** searchField, **string** searchString, **string** searchOper)

{

*// get the page number, the number of data displayed*

**int** pageNo = page ?? 1;

**int** limit = rows ?? 20;

*// calculate the offset*

**int** offset = (pageNo - 1) \* limit;

*// building a query for suppliers*

var customersQuery =

**from** customer **in** db.CUSTOMERS

**select new**

{

CUSTOMER\_ID = customer.CUSTOMER\_ID,

NAME = customer.NAME,

ADDRESS = customer.ADDRESS,

ZIPCODE = customer.ZIPCODE,

PHONE = customer.PHONE

};

*// adding a search condition to the query, if it is produced*

**if** (searchField != **null**)

{

**switch** (searchOper)

{

**case** "eq":

customersQuery = customersQuery.Where(

c => c.NAME == searchString);

**break**;

**case** "bw":

customersQuery = customersQuery.Where(

c => c.NAME.StartsWith(searchString));

**break**;

**case** "cn":

customersQuery = customersQuery.Where(

c => c.NAME.Contains(searchString));

**break**;

}

}

*// get the total number of suppliers*

**int** totalRows = customersQuery.Count();

*// add sorting*

**switch** (sord) {

**case** "asc":

customersQuery = customersQuery.OrderBy(

customer => customer.NAME);

**break**;

**case** "desc":

customersQuery = customersQuery.OrderByDescending(

customer => customer.NAME);

**break**;

}

*// get the list of suppliers*

var customers = customersQuery

.Skip(offset)

.Take(limit)

.ToList();

*// calculate the total number of pages*

**int** totalPages = totalRows / limit + 1;

*// create the result for jqGrid*

var result = **new**

{

page = pageNo,

total = totalPages,

records = totalRows,

rows = customers

};

*// convert the result to JSON*

**return** Json(result, JsonRequestBehavior.AllowGet);

}

*// Adding a new supplier*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Create(

[Bind(Include = "NAME,ADDRESS,ZIPCODE,PHONE")] CUSTOMER customer)

{

*// check the correctness of the model*

**if** (ModelState.IsValid)

{

*// get a new identifier using a generator*

customer.CUSTOMER\_ID = db.NextValueFor("GEN\_CUSTOMER\_ID");

*// add the model to the list*

db.CUSTOMERS.Add(customer);

*// save model*

db.SaveChanges();

*// return success in JSON format*

**return** Json(**true**);

}

**else** {

*// join model errors in one string*

**string** messages = **string**.Join("; ", ModelState.Values

.SelectMany(x => x.Errors)

.Select(x => x.ErrorMessage));

*// return error in JSON format*

**return** Json(**new** { error = messages });

}

}

*// Editing supplier*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Edit(

[Bind(Include = "CUSTOMER\_ID,NAME,ADDRESS,ZIPCODE,PHONE")] CUSTOMER customer)

{

*// check the correctness of the model*

**if** (ModelState.IsValid)

{

*// mark the model as modified*

db.Entry(customer).State = EntityState.Modified;

*// save model*

db.SaveChanges();

*// return success in JSON format*

**return** Json(**true**);

}

**else** {

*// join model errors in one string*

**string** messages = **string**.Join("; ", ModelState.Values

.SelectMany(x => x.Errors)

.Select(x => x.ErrorMessage));

*// return error in JSON format*

**return** Json(**new** { error = messages });

}

}

*// Deleting supplier*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Delete(**int** id)

{

*// find supplier by id*

CUSTOMER customer = db.CUSTOMERS.Find(id);

*// delete supplier*

db.CUSTOMERS.Remove(customer);

*// save model*

db.SaveChanges();

*// return success in JSON format*

**return** Json(**true**);

}

**protected override void** Dispose(**bool** disposing)

{

**if** (disposing)

{

db.Dispose();

}

**base**.Dispose(disposing);

}

}

The Index method is used to display the Views/Cusomter/Index.cshtml view. The view itself will be presented a bit later. This view is actually an html page template with markup and JavaScript for initiating jqGrid. The data itself will be obtained asynchronously in the JSON format, using the Ajax technology. The selected type of sorting, the page number and the search parameters will determine the format of an HTTP request that will be handled by the GetData action. The parameters of the HTTP request are displayed in the input parameters of the GetData method. We generate a LINQ query based on these parameters and send the retrieved result in the JSON format.

**Note**

Various libraries can assist with parsing the parameters of a query generated by jqGrid and make it easier to build the model. We have not used them in our examples so the code might be somewhat cumbersome. You can always improve it, of course.

The Create method is used to add a new customer record. The parameters of the HTTP POST request (the method has the [HttpPost] attribute specified for it) will be displayed on the Customer model. Check out the following line:

[Bind(Include = "NAME,ADDRESS,ZIPCODE,PHONE")] CUSTOMER customer

Here Bind specifies which parameters of the HTTP request are to be displayed in the properties of the model.

Note the **ValidateAntiforgeryToken** attribute. It is used to prevent forging requests between websites by verifying the tokens when the action method is called. The presence of this attribute requires that the HTTP request has an additional parameter named \_\_RequestVerificationToken. This parameter is automatically added to each form where the @Html.AntiForgeryToken() helper is specified. However, the jqGrid library uses dynamically generated Ajax requests instead of previously created web forms. To fix it, we need to change the shared view Views/Shared/\_Layout.cshtml as follows:

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title – ASP.NET application</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/jquery-ui")

<link href="~/Content/jquery.jqGrid/ui.jqgrid.css"

rel="stylesheet" type="text/css" />

<link href="~/Content/jquery.jqGrid/ui.jqgrid-bootstrap.css"

rel="stylesheet" type="text/css" />

<link href="~/Content/jquery.jqGrid/ui.jqgrid-bootstrap-ui.css"

rel="stylesheet" type="text/css" />

<script src="~/Scripts/jquery.jqGrid.min.js"

type="text/javascript"></script>

<script src="~/Scripts/i18n/grid.locale-en.js"

type="text/javascript"></script>

</head>

<body>

@Html.AntiForgeryToken()

<script>

**function** GetAntiForgeryToken() {

**var** tokenField =

$("input[type='hidden'][name$='RequestVerificationToken']");

**if** (tokenField.length == 0) {

**return** null;

} **else** {

**return** {

name: tokenField[0].name,

value: tokenField[0].value

};

}

}

*// add prefilter to all ajax requests*

*// it will add to any POST ajax request*

*// AntiForgery token*

$.ajaxPrefilter(

**function** (options, localOptions, jqXHR) {

**if** (options.type !== "GET") {

**var** token = GetAntiForgeryToken();

**if** (token !== null) {

**if** (options.data.indexOf("X-Requested-With") === -1) {

options.data = "X-Requested-With=XMLHttpRequest"

+ ((options.data === "") ? "" : "&" + options.data);

}

options.data = options.data + "&" + token.name + '='

+ token.value;

}

}

}

);

*// initialize the general properties of the jqGrid module*

$.jgrid.defaults.width = 780;

$.jgrid.defaults.responsive = true;

$.jgrid.defaults.styleUI = 'Bootstrap';

</script>

<!-- Navigation menu -->

<div **class**="navbar navbar-inverse navbar-fixed-top">

<div **class**="container">

<div **class**="navbar-header">

<button type="button" **class**="navbar-toggle" data-toggle="collapse"

data-target=".navbar-collapse">

<span **class**="icon-bar"></span>

<span **class**="icon-bar"></span>

<span **class**="icon-bar"></span>

</button>

</div>

<div **class**="navbar-collapse collapse">

<ul **class**="nav navbar-nav">

<li>@Html.ActionLink("Customers", "Index", "Customer")</li>

<li>@Html.ActionLink("Goods", "Index", "Product")</li>

<li>@Html.ActionLink("Invoices", "Index", "Invoice")</li>

</ul>

</div>

</div>

</div>

<div **class**="container body-content">

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year – ASP.NET application</p>

</footer>

</div>

@Scripts.Render("~/bundles/bootstrap")

@RenderSection("scripts", required: false)

</body>

</html>

### Bundles

Bundles are used to make it easier to link JavaScript scripts and CSS files. You can link CSS bundles with the Styles.Render helper and script bundles with the Scripts.Render helper.

Bundles are registered in the *BundleConfig.cs* file located in the *App\_Start* folder:

**public static void** RegisterBundles(BundleCollection bundles)

{

bundles.Add(**new** ScriptBundle("~/bundles/jquery").Include(

"~/Scripts/jquery-{version}.js"));

bundles.Add(**new** ScriptBundle("~/bundles/jqueryval").Include(

"~/Scripts/jquery.validate\*"));

bundles.Add(**new** ScriptBundle("~/bundles/jquery-ui").Include(

"~/Scripts/jquery-ui-{version}.js"));

bundles.Add(**new** ScriptBundle("~/bundles/modernizr").Include(

"~/Scripts/modernizr-\*"));

bundles.Add(**new** ScriptBundle("~/bundles/bootstrap").Include(

"~/Scripts/bootstrap.js",

"~/Scripts/respond.js"));

bundles.Add(**new** StyleBundle("~/Content/css").Include(

"~/Content/jquery-ui.min.css",

"~/Content/themes/ui-darkness/jquery-ui.min.css",

"~/Content/themes/ui-darkness/theme.css",

"~/Content/bootstrap.min.css",

"~/Content/Site.css"

));

}

The RegisterBundles method adds all created bundles to the bundles collection. A bundle is declared in the following way:

**new** ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-{version}.js")

The virtual path of the bundle is passed to the ScriptBundle construct. Specific script files are included in this bundle by utilizing the Include method.

The {version} parameter in the "~/Scripts/jquery-{version}.js" expression is a placeholder for any string referring to the script version. It is very convenient because it allows the version of the library to be changed later without having to change anything in the code. The system will accept the new version automatically.

The "~/Scripts/jquery.validate\*" expression changes the rest of the string with the asterisk character as a wildcard. For instance, the expression will include two files at once in the bundle: jquery.validate.js and jquery.validate.unobtrusive.js (and their minimized versions), because their names both start withjquery.validate\*".

The same applies when creating CSS bundles, using the StyleBundle class.

It is advisable to have the full versions of the scripts and cascading style sheets in the debug mode and the minimized ones in the release mode. Bundles allow you to solve this problem. When you run the application in the debug mode, the *web.config* files have the <compilation debug="true"> parameter. When you set this parameter to false (the Release mode), the minimized version of JavaScript modules and CSS files will be used instead of the full ones.

### Views

Since we need only the View/Customer/Index.cshtml view out of the five created for the Customer controller, you can delete the others from the folder.

@{

ViewBag.Title = "Index";

}

<h2>Customers</h2>

<table id="jqg"></table>

<div id="jqg-pager"></div>

<script type="text/javascript">

$(document).ready(**function** () {

**var** dbGrid = $("#jqg").jqGrid({

url: '@Url.Action("GetData")', *// URL to retrieve data*

datatype: "json", *// data format*

mtype: "GET", *// http type request*

*// model description*

colModel: [

{

label: 'Id',

name: 'CUSTOMER\_ID', *// field name*

key: true,

hidden: true

},

{

label: 'Name',

name: 'NAME',

width: 250,

sortable: true,

editable: true,

edittype: "text", *// field type in the editor*

search: true,

searchoptions: {

sopt: ['eq', 'bw', 'cn'] *// allowed search operators*

},

*// size and maximum length for the input field*

editoptions: { size: 30, maxlength: 60 },

*// mandatory field*

editrules: { required: true }

},

{

label: 'Address',

name: 'ADDRESS',

width: 300,

sortable: false, *// prohibit sorting*

editable: true,

search: false, *// prohibit searching*

edittype: "textarea",

editoptions: { maxlength: 250, cols: 30, rows: 4 }

},

{

label: 'Zip Code',

name: 'ZIPCODE',

width: 30,

sortable: false,

editable: true,

search: false,

edittype: "text",

editoptions: { size: 30, maxlength: 10 },

},

{

label: 'Phone',

name: 'PHONE',

width: 80,

sortable: false,

editable: true,

search: false,

edittype: "text",

editoptions: { size: 30, maxlength: 14 },

}

],

rowNum: 500, *// number of rows displayed*

loadonce: false, *// load only once*

sortname: 'NAME', *// sort by default by NAME column*

sortorder: "asc",

width: window.innerWidth - 80, *// grid width*

height: 500, *// grid height*

viewrecords: true, *// display the number of records*

caption: "Customers",

pager: 'jqg-pager' *// navigation item id*

});

dbGrid.jqGrid('navGrid', '#jqg-pager', {

search: true,

add: true,

edit: true,

del: true,

view: true,

refresh: true,

*// button labels*

searchtext: "Find",

addtext: "Add",

edittext: "Edit",

deltext: "Delete",

viewtext: "View",

viewtitle: "Selected record",

refreshtext: "Refresh"

},

update("edit"),

update("add"),

update("del")

);

*// function that returns the settings of the editor*

**function** update(act) {

**return** {

closeAfterAdd: true,

closeAfterEdit: true,

width: 400, *// editor width*

reloadAfterSubmit: true,

drag: true,

*// handler for sending the form of editing / deleting / adding*

onclickSubmit: **function** (params, postdata) {

*// get row id*

**var** selectedRow = dbGrid.getGridParam("selrow");

*// set URL depending on the operation*

**switch** (act) {

**case** "add":

params.url = '@Url.Action("Create")';

**break**;

**case** "edit":

params.url = '@Url.Action("Edit")';

postdata.CUSTOMER\_ID = selectedRow;

**break**;

**case** "del":

params.url = '@Url.Action("Delete")';

postdata.CUSTOMER\_ID = selectedRow;

**break**;

}

},

*// processing results of sending forms (operations)*

afterSubmit: **function** (response, postdata) {

**var** responseData = response.responseJSON;

*// check the result for error messages*

**if** (responseData.hasOwnProperty("error")) {

**if** (responseData.error.length) {

**return** [false, responseData.error];

}

}

**else** {

*// refresh grid*

$(**this**).jqGrid(

'setGridParam',

{

datatype: 'json'

}

).trigger('reloadGrid');

}

**return** [true, "", 0];

}

};

};

});

</script>

You can see that the entire view consists of the header, the jqg table and the jqg-pager block for displaying the navigation bar. The rest is occupied by the script for initiating the grid, the navigation bar and the dialog box for editing records. It is important to configure the model properties correctly in order to display the grid correctly, position input items on the edit form, configure validation for input forms and configure the sorting and search options.,. This configuration is not simple and has a lot of parameters. I have tried to describe the parameters being used in the comments. The full description of the model parameters can be found in the documentation for the jqGrid library in the [ColModel API](http://www.trirand.com/jqgridwiki/doku.php?id=wiki:colmodel_options) section.

Note that jqGrid does not automatically add hidden grid columns to the input form though I think it would make sense at least for key fields. Consequently, we have to add the customer identifier to the request parameters for editing and deleting:

**case** "edit":

params.url = '@Url.Action("Edit")';

postdata.CUSTOMER\_ID = selectedRow;

**break**;

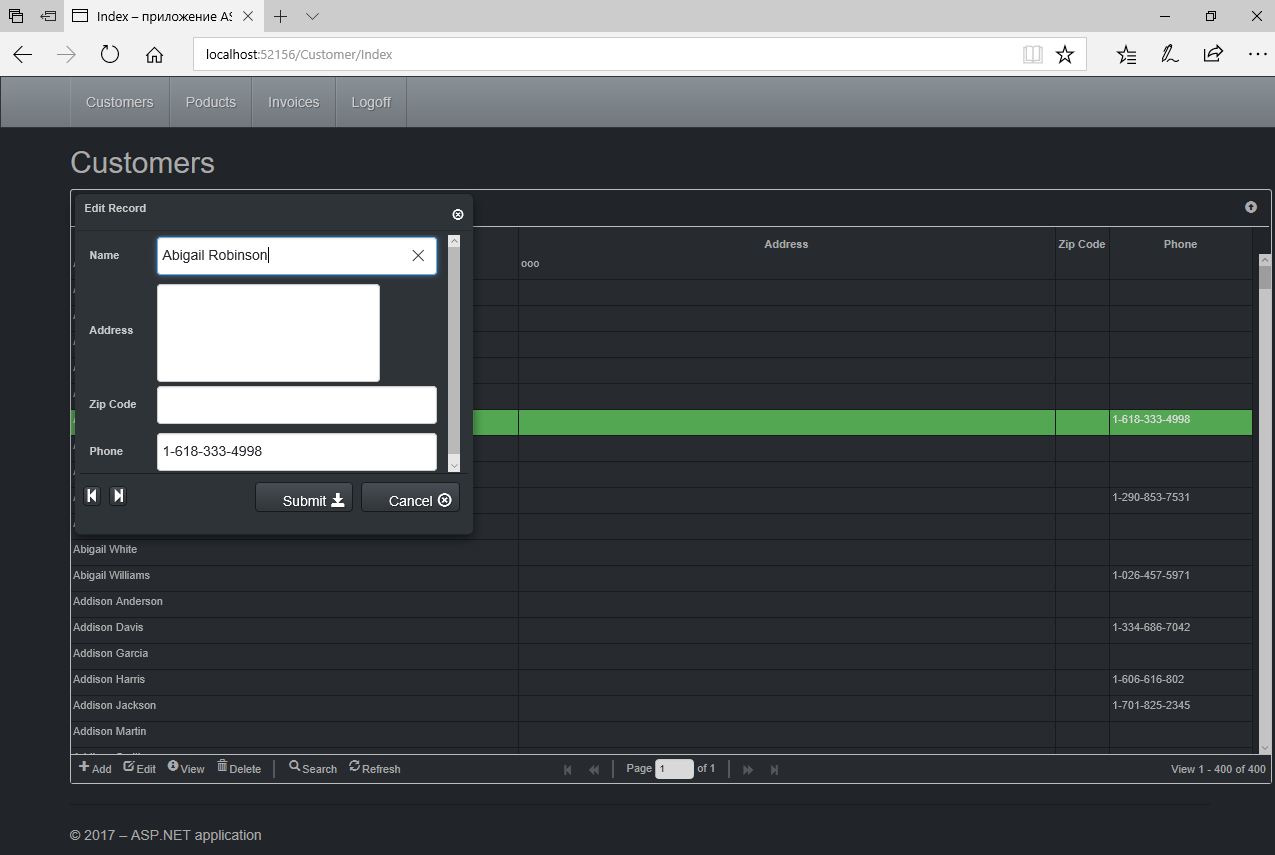
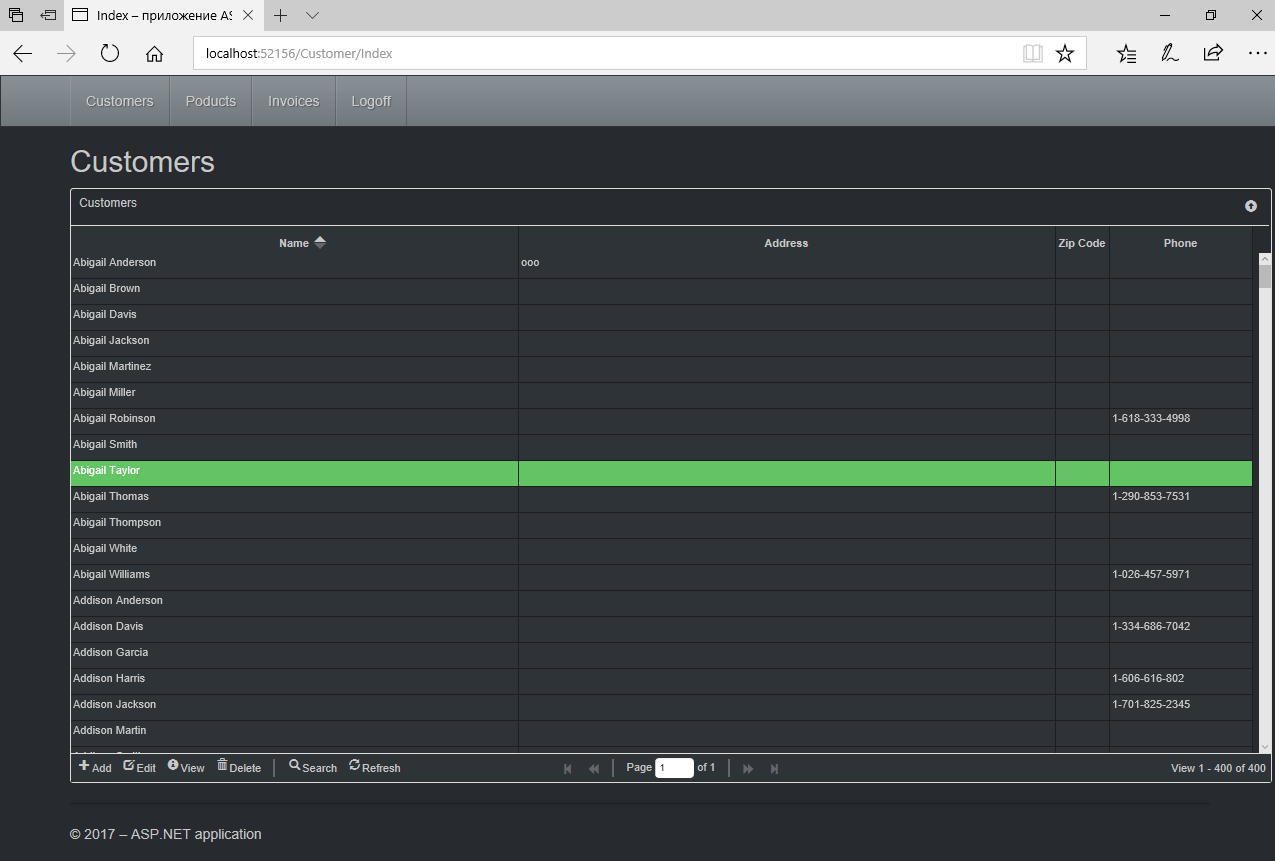
**case** "del":

params.url = '@Url.Action("Delete")';

postdata.CUSTOMER\_ID = selectedRow;

**break**;

The working page with the directory of customers will look like this:



The controller and view for the product UI are implemented in a similar way. We will not describe them here in detail. You can either write them yourself or use the source code attached to this article.

## Create a user interface for secondary modules

Our application will have only one secondary module, called "Invoices" Unlike our primary modules, the secondary module is likely to contain numerous records and new records are added more frequently.

An invoice consists of a header where some general attributes are described (number, date, customer …) and invoice detail lines with the list of products sold, their quantities, prices, etc. To save space on the page, we will hide the detail grid and display it only in response to a click on the icon with the + sign on it. Thus, we have our detailed grid embedded into the main one.

### Controllers

The controller of the invoice module must be able to return data for both invoice headers and the associate invoice lines. The same applies to the methods for adding, editing and deleting records.

[Authorize(Roles = "manager")]

**public class** InvoiceController : Controller

{

**private** DbModel db = **new** DbModel();

*// display view*

**public** ActionResult Index()

{

**return** View();

}

*// Receiving data in the JSON format for the main grid*

**public** ActionResult GetData(**int**? rows, **int**? page, **string** sidx, **string** sord,

**string** searchField, **string** searchString, **string** searchOper)

{

*// get the page number, the number of data displayed*

**int** pageNo = page ?? 1;

**int** limit = rows ?? 20;

*// calculate offset*

**int** offset = (pageNo - 1) \* limit;

*// building a request for receipt of invoices*

var invoicesQuery =

**from** invoice **in** db.INVOICES

**where** (invoice.INVOICE\_DATE >= AppVariables.StartDate) &&

(invoice.INVOICE\_DATE <= AppVariables.FinishDate)

**select new**

{

INVOICE\_ID = invoice.INVOICE\_ID,

CUSTOMER\_ID = invoice.CUSTOMER\_ID,

CUSTOMER\_NAME = invoice.CUSTOMER.NAME,

INVOICE\_DATE = invoice.INVOICE\_DATE,

TOTAL\_SALE = invoice.TOTAL\_SALE,

PAID = invoice.PAID

};

*// adding a search condition to the query, if it is produced*

*// for different fields, different comparison operators*

*// are available when searching*

**if** (searchField == "CUSTOMER\_NAME")

{

**switch** (searchOper)

{

**case** "eq": *// equal*

invoicesQuery = invoicesQuery.Where(

c => c.CUSTOMER\_NAME == searchString);

**break**;

**case** "bw": *// starting with*

invoicesQuery = invoicesQuery.Where(

c => c.CUSTOMER\_NAME.StartsWith(searchString));

**break**;

**case** "cn": *// containing*

invoicesQuery = invoicesQuery.Where(

c => c.CUSTOMER\_NAME.Contains(searchString));

**break**;

}

}

**if** (searchField == "INVOICE\_DATE")

{

var dateValue = DateTime.Parse(searchString);

**switch** (searchOper)

{

**case** "eq": *// =*

invoicesQuery = invoicesQuery.Where(

c => c.INVOICE\_DATE == dateValue);

**break**;

**case** "lt": *// <*

invoicesQuery = invoicesQuery.Where(

c => c.INVOICE\_DATE < dateValue);

**break**;

**case** "le": *// <=*

invoicesQuery = invoicesQuery.Where(

c => c.INVOICE\_DATE <= dateValue);

**break**;

**case** "gt": *// >*

invoicesQuery = invoicesQuery.Where(

c => c.INVOICE\_DATE > dateValue);

**break**;

**case** "ge": *// >=*

invoicesQuery = invoicesQuery.Where(

c => c.INVOICE\_DATE >= dateValue);

**break**;

}

}

**if** (searchField == "PAID")

{

**int** iVal = (searchString == "on") ? 1 : 0;

invoicesQuery = invoicesQuery.Where(c => c.PAID == iVal);

}

*// get the total number of invoices*

**int** totalRows = invoicesQuery.Count();

*// add sorting*

**switch** (sord)

{

**case** "asc":

invoicesQuery = invoicesQuery.OrderBy(

invoice => invoice.INVOICE\_DATE);

**break**;

**case** "desc":

invoicesQuery = invoicesQuery.OrderByDescending(

invoice => invoice.INVOICE\_DATE);

**break**;

}

*// get invoice list*

var invoices = invoicesQuery

.Skip(offset)

.Take(limit)

.ToList();

*// calculate the total number of pages*

**int** totalPages = totalRows / limit + 1;

*// create the result for jqGrid*

var result = **new**

{

page = pageNo,

total = totalPages,

records = totalRows,

rows = invoices

};

*// convert the result to JSON*

**return** Json(result, JsonRequestBehavior.AllowGet);

}

*// Receiving data in the form of JSON for the detail grid*

**public** ActionResult GetDetailData(**int**? invoice\_id)

{

*// build a LINQ query for receiving invoice items*

*// filtered by invoice id*

var lines =

**from** line **in** db.INVOICE\_LINES

**where** line.INVOICE\_ID == invoice\_id

**select new**

{

INVOICE\_LINE\_ID = line.INVOICE\_LINE\_ID,

INVOICE\_ID = line.INVOICE\_ID,

PRODUCT\_ID = line.PRODUCT\_ID,

Product = line.PRODUCT.NAME,

Quantity = line.QUANTITY,

Price = line.SALE\_PRICE,

Total = line.QUANTITY \* line.SALE\_PRICE

};

*// get invoice position list*

var invoices = lines

.ToList();

*// create the result for jqGrid*

var result = **new**

{

rows = invoices

};

*// convert the result to JSON*

**return** Json(result, JsonRequestBehavior.AllowGet);

}

*// Add new invoice*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Create(

[Bind(Include = "CUSTOMER\_ID,INVOICE\_DATE")] INVOICE invoice)

{

*// check the correctness of the model*

**if** (ModelState.IsValid)

{

**try**

{

var INVOICE\_ID = **new** FbParameter("INVOICE\_ID", FbDbType.Integer);

var CUSTOMER\_ID = **new** FbParameter("CUSTOMER\_ID", FbDbType.Integer);

var INVOICE\_DATE = **new** FbParameter("INVOICE\_DATE",

FbDbType.TimeStamp);

*// initialize parameters query*

INVOICE\_ID.Value = db.NextValueFor("GEN\_INVOICE\_ID");

CUSTOMER\_ID.Value = invoice.CUSTOMER\_ID;

INVOICE\_DATE.Value = invoice.INVOICE\_DATE;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_ADD\_INVOICE(@INVOICE\_ID, @CUSTOMER\_ID, @INVOICE\_DATE)",

INVOICE\_ID,

CUSTOMER\_ID,

INVOICE\_DATE);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

**else** {

**string** messages = **string**.Join("; ", ModelState.Values

.SelectMany(x => x.Errors)

.Select(x => x.ErrorMessage));

*// return error in JSON format*

**return** Json(**new** { error = messages });

}

}

*// Edit invoice*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Edit(

[Bind(Include = "INVOICE\_ID,CUSTOMER\_ID,INVOICE\_DATE")] INVOICE invoice)

{

*// check the correctness of the model*

**if** (ModelState.IsValid)

{

**try**

{

var INVOICE\_ID = **new** FbParameter("INVOICE\_ID", FbDbType.Integer);

var CUSTOMER\_ID = **new** FbParameter("CUSTOMER\_ID", FbDbType.Integer);

var INVOICE\_DATE = **new** FbParameter("INVOICE\_DATE",

FbDbType.TimeStamp);

*// initialize parameters query*

INVOICE\_ID.Value = invoice.INVOICE\_ID;

CUSTOMER\_ID.Value = invoice.CUSTOMER\_ID;

INVOICE\_DATE.Value = invoice.INVOICE\_DATE;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_EDIT\_INVOICE(@INVOICE\_ID, @CUSTOMER\_ID, @INVOICE\_DATE)",

INVOICE\_ID,

CUSTOMER\_ID,

INVOICE\_DATE);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

**else** {

**string** messages = **string**.Join("; ", ModelState.Values

.SelectMany(x => x.Errors)

.Select(x => x.ErrorMessage));

*// return error in JSON format*

**return** Json(**new** { error = messages });

}

}

*// Delete invoice*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Delete(**int** id)

{

**try**

{

var INVOICE\_ID = **new** FbParameter("INVOICE\_ID", FbDbType.Integer);

*// initialize parameters query*

INVOICE\_ID.Value = id;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_DELETE\_INVOICE(@INVOICE\_ID)",

INVOICE\_ID);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

*// Payment of invoice*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Pay(**int** id)

{

**try**

{

var INVOICE\_ID = **new** FbParameter("INVOICE\_ID", FbDbType.Integer);

*// initialize parameters query*

INVOICE\_ID.Value = id;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_PAY\_FOR\_INOVICE(@INVOICE\_ID)",

INVOICE\_ID);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

*// Add invoice position*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult CreateDetail(

[Bind(Include = "INVOICE\_ID,PRODUCT\_ID,QUANTITY")] INVOICE\_LINE invoiceLine)

{

*// check the correctness of the model*

**if** (ModelState.IsValid)

{

**try**

{

var INVOICE\_ID = **new** FbParameter("INVOICE\_ID", FbDbType.Integer);

var PRODUCT\_ID = **new** FbParameter("PRODUCT\_ID", FbDbType.Integer);

var QUANTITY = **new** FbParameter("QUANTITY", FbDbType.Integer);

*// initialize parameters query*

INVOICE\_ID.Value = invoiceLine.INVOICE\_ID;

PRODUCT\_ID.Value = invoiceLine.PRODUCT\_ID;

QUANTITY.Value = invoiceLine.QUANTITY;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_ADD\_INVOICE\_LINE(@INVOICE\_ID, @PRODUCT\_ID, @QUANTITY)",

INVOICE\_ID,

PRODUCT\_ID,

QUANTITY);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

**else** {

**string** messages = **string**.Join("; ", ModelState.Values

.SelectMany(x => x.Errors)

.Select(x => x.ErrorMessage));

*// return error in JSON format*

**return** Json(**new** { error = messages });

}

}

*// Edit invoice position*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult EditDetail(

[Bind(Include = "INVOICE\_LINE\_ID,INVOICE\_ID,PRODUCT\_ID,QUANTITY")]

INVOICE\_LINE invoiceLine)

{

*// check the correctness of the model*

**if** (ModelState.IsValid)

{

**try**

{

*// Create parameters*

var INVOICE\_LINE\_ID = **new** FbParameter("INVOICE\_LINE\_ID",

FbDbType.Integer);

var QUANTITY = **new** FbParameter("QUANTITY", FbDbType.Integer);

*// initialize parameters query*

INVOICE\_LINE\_ID.Value = invoiceLine.INVOICE\_LINE\_ID;

QUANTITY.Value = invoiceLine.QUANTITY;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_EDIT\_INVOICE\_LINE(@INVOICE\_LINE\_ID, @QUANTITY)",

INVOICE\_LINE\_ID,

QUANTITY);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

**else** {

**string** messages = **string**.Join("; ", ModelState.Values

.SelectMany(x => x.Errors)

.Select(x => x.ErrorMessage));

*// return error in JSON format*

**return** Json(**new** { error = messages });

}

}

*// Delete invoice position*

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult DeleteDetail(**int** id)

{

**try**

{

*// create parameters*

var INVOICE\_LINE\_ID = **new** FbParameter("INVOICE\_LINE\_ID",

FbDbType.Integer);

*// initialize parameters query*

INVOICE\_LINE\_ID.Value = id;

*// execute stored procedure*

db.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_DELETE\_INVOICE\_LINE(@INVOICE\_LINE\_ID)",

INVOICE\_LINE\_ID);

*// return success in JSON format*

**return** Json(**true**);

}

**catch** (Exception ex)

{

*// return error in JSON format*

**return** Json(**new** { error = ex.Message });

}

}

**protected override void** Dispose(**bool** disposing)

{

**if** (disposing)

{

db.Dispose();

}

**base**.Dispose(disposing);

}

}

The GetDetailData method for retrieving the list of lines in an invoice lacks the code for page-by-page navigation. Realistically, a typical invoice does not have enough lines to justify using page-by-page navigation for them. Omitting it simplifies and speeds up the code.

In our sample, all data modification operations are performed in stored procedures, but you could do the same work using Entity Framework. Code for the stored procedures can be found in the database creation script.

### Views

A with the Customer controller, only one view View/Invoice/Index.cshtml is needed. The others can be deleted from this folder. The layout of the view is very simple, but the JavaScript code is quite extensive. We will examine the js code piece-by-piece.

@{

ViewBag.Title = "Index";

}

<h2>Invoices</h2>

<table id="jqg"></table>

<div id="jpager"></div>

<script type="text/javascript">

*/\*\**

*\* The code to work with jqGrid*

*\*/*

</script>

To begin with, we will take the code for working with the main grid. All we have to write into it is the properties of the model (field types and sizes, search, sorting, visibility parameters. etc.).

*// invoice grid*

**var** dbGrid = $("#jqg").jqGrid({

url: '@Url.Action("GetData")', *URL to retrieve data*

datatype: "json", *// format data*

mtype: "GET", *// type of http request*

*// model description*

colModel: [

{

label: 'Id',

name: 'INVOICE\_ID',

key: true,

hidden: true

},

{

label: 'CUSTOMER\_ID',

name: 'CUSTOMER\_ID',

hidden: true,

editrules: { edithidden: true, required: true },

editable: true,

edittype:'custom', *// own type*

editoptions: {

custom\_element: **function** (value, options) {

*// add hidden input*

**return** $("<input>")

.attr('type', 'hidden')

.attr('rowid', options.rowId)

.addClass("FormElement")

.addClass("form-control")

.val(value)

.get(0);

}

}

},

{

label: 'Date',

name: 'INVOICE\_DATE',

width: 60,

sortable: true,

editable: true,

search: true,

edittype: "text", *// type of input*

align: "right",

formatter: 'date', *// formatted as date*

sorttype: 'date', *// sorted as date*

formatoptions: { *// date format*

srcformat: 'd.m.Y H:i:s',

newformat: 'd.m.Y H:i:s'

},

editoptions: {

*// initializing the form element for editing*

dataInit: **function** (element) {

*// create datepicker*

$(element).datepicker({

id: 'invoiceDate\_datePicker',

dateFormat: 'dd.mm.yy',

minDate: **new Date**(2000, 0, 1),

maxDate: **new Date**(2030, 0, 1)

});

}

},

searchoptions: {

*// initializing the form element for searching*

dataInit: **function** (element) {

*// create datepicker*

$(element).datepicker({

id: 'invoiceDate\_datePicker',

dateFormat: 'dd.mm.yy',

minDate: **new Date**(2000, 0, 1),

maxDate: **new Date**(2030, 0, 1)

});

},

searchoptions: { *// searching types*

sopt: ['eq', 'lt', 'le', 'gt', 'ge']

},

}

},

{

label: 'Customer',

name: 'CUSTOMER\_NAME',

width: 250,

editable: true,

edittype: "text",

editoptions: {

size: 50,

maxlength: 60,

readonly: true

},

editrules: { required: true },

search: true,

searchoptions: {

sopt: ['eq', 'bw', 'cn']

},

},

{

label: 'Amount',

name: 'TOTAL\_SALE',

width: 60,

sortable: false,

editable: false,

search: false,

align: "right",

formatter: 'currency', *// format as currency*

sorttype: 'number',

searchrules: {

"required": true,

"number": true,

"minValue": 0

}

},

{

label: 'Paid',

name: 'PAID',

width: 30,

sortable: false,

editable: true,

search: true,

searchoptions: {

sopt: ['eq']

},

edittype: "checkbox",

formatter: "checkbox",

stype: "checkbox",

align: "center",

editoptions: {

value: "1",

offval: "0"

}

}

],

rowNum: 500, *// number of rows displayed*

loadonce: false,

sortname: 'INVOICE\_DATE', *// sort by default by NAME column*

sortorder: "desc",

width: window.innerWidth - 80, *// grid width*

height: 500, *// grid height*

viewrecords: true, *// display the number of records*

caption: "Invoices", *// grid caption*

pager: '#jpager', *// pagination element*

subGrid: true, *// show subgrid*

*// javascript function for displaying the parent grid*

subGridRowExpanded: showChildGrid,

subGridOptions: {

*// upload data only once*

reloadOnExpand: false,

*// load the subgrid rows only when you click on the icon "+"*

selectOnExpand: true

},

});

*// display the navigation bar*

dbGrid.jqGrid('navGrid', '#jpager',

{

search: true,

add: true,

edit: true,

del: true,

view: false,

refresh: true,

searchtext: "Search",

addtext: "Add",

edittext: "Edit",

deltext: "Delete",

viewtext: "View",

viewtitle: "Selected record",

refreshtext: "Refresh"

},

update("edit"),

update("add"),

update("del")

);

We'll add one more "custom" button to the main grid, for paying the invoice.

*// Add a button to pay the invoice*

dbGrid.navButtonAdd('#jpager',

{

buttonicon: "glyphicon-usd",

title: "Pay",

caption: "Pay",

position: "last",

onClickButton: **function** () {

*// get the current record ID*

**var** id = dbGrid.getGridParam("selrow");

**if** (id) {

**var** url = '@Url.Action("Pay")';

$.ajax({

url: url,

type: 'POST',

data: { id: id },

success: **function** (data) {

*// check if an error has occurred*

**if** (data.hasOwnProperty("error")) {

alertDialog('Error', data.error);

}

**else** {

*// refresh grid*

$("#jqg").jqGrid(

'setGridParam',

{

datatype: 'json'

}

).trigger('reloadGrid');

}

}

});

}

}

});

The dialog boxes for editing secondary sets of data are much more complicated than for the primary sets. Since they often use options selected from other modules, it will not be possible to use the standard jqGrid methods to build these edit dialog boxes. However, this library has an option to build dialog boxes using templates, which we will use.

To enable customer selection, we will create a read-only field with a button at its right hand side for opening the form displaying the customer selection grid.

*// returns properties to create edit dialogs*

**function** update(act) {

*// editing dialog template*

**var** template = "<div style='margin-left:15px;' id='dlgEditInvoice'>";

template += "<div>{CUSTOMER\_ID} </div>";

template += "<div> Date: </div><div>{INVOICE\_DATE} </div>";

*// customer input field with a button*

template += "<div> Customer <sup>\*</sup>:</div>";

template += "<div>";

template += "<div style='float: left;'>{CUSTOMER\_NAME}</div> ";

template += "<a style='margin-left: 0.2em;' class='btn'";

template += " onclick='showCustomerWindow(); return false;'>";

template += "<span class='glyphicon glyphicon-folder-open'></span>";

template += " Select</a> ";

template += "<div style='clear: both;'></div>";

template += "</div>";

template += "<div> {PAID} Paid </div>";

template += "<hr style='width: 100%;'/>";

template += "<div> {sData} {cData} </div>";

template += "</div>";

**return** {

top: $(".container.body-content").position().top + 150,

left: $(".container.body-content").position().left + 150,

modal: true,

drag: true,

closeOnEscape: true,

closeAfterAdd: true,

closeAfterEdit: true,

reloadAfterSubmit: true,

template: (act != "del") ? template : null,

onclickSubmit: **function** (params, postdata) {

*// get row id*

**var** selectedRow = dbGrid.getGridParam("selrow");

**switch** (act) {

**case** "add":

params.url = '@Url.Action("Create")';

*// get customer id for current row*

postdata.CUSTOMER\_ID =

$('#dlgEditInvoice input[name=CUSTOMER\_ID]').val();

**break**;

**case** "edit":

params.url = '@Url.Action("Edit")';

postdata.INVOICE\_ID = selectedRow;

*// get customer id for current row*

postdata.CUSTOMER\_ID =

$('#dlgEditInvoice input[name=CUSTOMER\_ID]').val();

**break**;

**case** "del":

params.url = '@Url.Action("Delete")';

postdata.INVOICE\_ID = selectedRow;

**break**;

}

},

afterSubmit: **function** (response, postdata) {

**var** responseData = response.responseJSON;

*// check the result for error messages*

**if** (responseData.hasOwnProperty("error")) {

**if** (responseData.error.length) {

**return** [false, responseData.error];

}

}

**else** {

*// refresh grid*

$(**this**).jqGrid(

'setGridParam',

{

datatype: 'json'

}

).trigger('reloadGrid');

}

**return** [true, "", 0];

}

};

};

}

Now we will write a function for opening the customer module. The function will invoke the Bootstrap library to create a dialog box containing the grid from which a customer can be selected. It is actually the same grid we used earlier, but this time, it is enclosed by a dialog box. A click on the OK button will place the customer identifier and the customer name into the input fields of the parent dialog box for editing invoices.

*/\*\**

*\* Display a window for selecting a customer*

*\*/*

**function** showCustomerWindow() {

*// the main block of the dialog*

**var** dlg = $('<div>')

.attr('id', 'dlgChooseCustomer')

.attr('aria-hidden', 'true')

.attr('role', 'dialog')

.attr('data-backdrop', 'static')

.css("z-index", '2000')

.addClass('modal')

.appendTo($('body'));

*// block with the contents of the dialog*

**var** dlgContent = $("<div>")

.addClass("modal-content")

.css('width', '730px')

.appendTo($('<div>')

.addClass('modal-dialog')

.appendTo(dlg));

*// block with dialogue header*

**var** dlgHeader = $('<div>').addClass("modal-header").appendTo(dlgContent);

*// button "X" for closing*

$("<button>")

.addClass("close")

.attr('type', 'button')

.attr('aria-hidden', 'true')

.attr('data-dismiss', 'modal')

.html("&times;")

.appendTo(dlgHeader);

*// title*

$("<h5>").addClass("modal-title")

.html("Select customer")

.appendTo(dlgHeader);

*// body of dialogue*

**var** dlgBody = $('<div>')

.addClass("modal-body")

.appendTo(dlgContent);

*// footer of the dialogue*

**var** dlgFooter = $('<div>').addClass("modal-footer").appendTo(dlgContent);

*// button "OK"*

$("<button>")

.attr('type', 'button')

.addClass('btn')

.html('OK')

.on('click', **function** () {

**var** rowId = $("#jqgCustomer").jqGrid("getGridParam", "selrow");

**var** row = $("#jqgCustomer").jqGrid("getRowData", rowId);

*// To save the identifier and customer name*

*// to the input elements of the parent form*

$('#dlgEditInvoice input[name=CUSTOMER\_ID]').val(rowId);

$('#dlgEditInvoice input[name=CUSTOMER\_NAME]').val(row["NAME"]);

dlg.modal('hide');

})

.appendTo(dlgFooter);

*// button "Cancel"*

$("<button>")

.attr('type', 'button')

.addClass('btn')

.html('Cancel')

.on('click', **function** () { dlg.modal('hide'); })

.appendTo(dlgFooter);

*// add a table to display the customers in the body of the dialog*

$('<table>')

.attr('id', 'jqgCustomer')

.appendTo(dlgBody);

*// add the navigation bar*

$('<div>')

.attr('id', 'jqgCustomerPager')

.appendTo(dlgBody);

dlg.on('hidden.bs.modal', **function** () {

dlg.remove();

});

*// show dialog*

dlg.modal();

*// create and initialize jqGrid*

**var** dbGrid = $("#jqgCustomer").jqGrid({

url: '@Url.Action("GetData", "Customer")', *// URL to retrieve data*

mtype: "GET", *// http type of request*

datatype: "json", *// data format*

page: 1,

width: '100%',

*// view description*

colModel: [

{

label: 'Id',

name: 'CUSTOMER\_ID',

key: true,

hidden: true

},

{

label: 'Name',

name: 'NAME',

width: 250,

sortable: true,

editable: true,

edittype: "text", *// input type*

search: true,

searchoptions: {

sopt: ['eq', 'bw', 'cn'] *// allowed search operators*

},

*// size and maximum length for the input field*

editoptions: { size: 30, maxlength: 60 },

*// required input*

editrules: { required: true }

},

{

label: 'Address',

name: 'ADDRESS',

width: 300,

sortable: false,

editable: true,

search: false,

edittype: "textarea",

editoptions: { maxlength: 250, cols: 30, rows: 4 }

},

{

label: 'Zip Code',

name: 'ZIPCODE',

width: 60,

sortable: false,

editable: true,

search: false,

edittype: "text",

editoptions: { size: 30, maxlength: 10 },

},

{

label: 'Phone',

name: 'PHONE',

width: 85,

sortable: false,

editable: true,

search: false,

edittype: "text",

editoptions: { size: 30, maxlength: 14 },

}

],

loadonce: false,

pager: '#jqgCustomerPager',

rowNum: 500, *// number of rows displayed*

sortname: 'NAME', *// sort by default by NAME column*

sortorder: "asc",

height: 500

});

dbGrid.jqGrid('navGrid', '#jqgCustomerPager',

{

search: true,

add: false,

edit: false,

del: false,

view: false,

refresh: true,

searchtext: "Search",

viewtext: "View",

viewtitle: "Selected record",

refreshtext: "Refresh"

}

);

}

All there is left to write for the invoice module is the showChildGrid function that enables the invoice lines to be displayed and edited. Our function will create a grid with invoice lines dynamically after a click on the + button to show the details. To load data for the lines requires passing the primary key of the selected invoice header.

*// handler of the event of opening the parent grid*

*// takes two parameters: the identifier of the parent record*

*// and the value of the primary key*

**function** showChildGrid(parentRowID, parentRowKey) {

**var** childGridID = parentRowID + "\_table";

**var** childGridPagerID = parentRowID + "\_pager";

*// send the primary key of the parent record*

*// to filter the entries of the invoice items*

**var** childGridURL = '@Url.Action("GetDetailData")';

childGridURL = childGridURL + "?invoice\_id="

+ **encodeURIComponent**(parentRowKey)

*// add HTML elements to display the table and page navigation*

*// as children for the selected row in the master grid*

$('<table>')

.attr('id', childGridID)

.appendTo($('#' + parentRowID));

$('<div>')

.attr('id', childGridPagerID)

.addClass('scroll')

.appendTo($('#' + parentRowID));

*// create and initialize the child grid*

**var** detailGrid = $("#" + childGridID).jqGrid({

url: childGridURL,

mtype: "GET",

datatype: "json",

page: 1,

colModel: [

{

label: 'Invoice Line ID',

name: 'INVOICE\_LINE\_ID',

key: true,

hidden: true

},

{

label: 'Invoice ID',

name: 'INVOICE\_ID',

hidden: true,

editrules: { edithidden: true, required: true },

editable: true,

edittype: 'custom',

editoptions: {

custom\_element: **function** (value, options) {

*// create hidden input*

**return** $("<input>")

.attr('type', 'hidden')

.attr('rowid', options.rowId)

.addClass("FormElement")

.addClass("form-control")

.val(parentRowKey)

.get(0);

}

}

},

{

label: 'Product ID',

name: 'PRODUCT\_ID',

hidden: true,

editrules: { edithidden: true, required: true },

editable: true,

edittype: 'custom',

editoptions: {

custom\_element: **function** (value, options) {

*// create hidden input*

**return** $("<input>")

.attr('type', 'hidden')

.attr('rowid', options.rowId)

.addClass("FormElement")

.addClass("form-control")

.val(value)

.get(0);

}

}

},

{

label: 'Product',

name: 'Product',

width: 300,

editable: true,

edittype: "text",

editoptions: {

size: 50,

maxlength: 60,

readonly: true

},

editrules: { required: true }

},

{

label: 'Price',

name: 'Price',

formatter: 'currency',

editable: true,

editoptions: {

readonly: true

},

align: "right",

width: 100

},

{

label: 'Quantity',

name: 'Quantity',

align: "right",

width: 100,

editable: true,

editrules: { required: true, number: true, minValue: 1 },

editoptions: {

dataEvents: [

{

type: 'change',

fn: **function** (e) {

**var** quantity = $(**this**).val() - 0;

**var** price =

$('#dlgEditInvoiceLine input[name=Price]').val() - 0;

$('#dlgEditInvoiceLine input[name=Total]').val(quantity \* price);

}

}

],

defaultValue: 1

}

},

{

label: 'Total',

name: 'Total',

formatter: 'currency',

align: "right",

width: 100,

editable: true,

editoptions: {

readonly: true

}

}

],

loadonce: false,

width: '100%',

height: '100%',

pager: "#" + childGridPagerID

});

*// displaying the toolbar*

$("#" + childGridID).jqGrid('navGrid', '#' + childGridPagerID,

{

search: false,

add: true,

edit: true,

del: true,

refresh: true

},

updateDetail("edit"),

updateDetail("add"),

updateDetail("del")

);

*// function that returns settings for the editing dialog*

**function** updateDetail(act) {

*// editing dialog template*

**var** template = "<div style='margin-left:15px;' id='dlgEditInvoiceLine'>";

template += "<div>{INVOICE\_ID} </div>";

template += "<div>{PRODUCT\_ID} </div>";

*// input field for goods with a button*

template += "<div> Product <sup>\*</sup>:</div>";

template += "<div>";

template += "<div style='float: left;'>{Product}</div> ";

template += "<a style='margin-left: 0.2em;' class='btn' ";

template += "onclick='showProductWindow(); return false;'>";

template += "<span class='glyphicon glyphicon-folder-open'></span>";

template += " Выбрать</a> ";

template += "<div style='clear: both;'></div>";

template += "</div>";

template += "<div> Quantity: </div><div>{Quantity} </div>";

template += "<div> Price: </div><div>{Price} </div>";

template += "<div> Total: </div><div>{Total} </div>";

template += "<hr style='width: 100%;'/>";

template += "<div> {sData} {cData} </div>";

template += "</div>";

**return** {

top: $(".container.body-content").position().top + 150,

left: $(".container.body-content").position().left + 150,

modal: true,

drag: true,

closeOnEscape: true,

closeAfterAdd: true,

closeAfterEdit: true,

reloadAfterSubmit: true,

template: (act != "del") ? template : null,

onclickSubmit: **function** (params, postdata) {

**var** selectedRow = detailGrid.getGridParam("selrow");

**switch** (act) {

**case** "add":

params.url = '@Url.Action("CreateDetail")';

*// get invoice id*

postdata.INVOICE\_ID =

$('#dlgEditInvoiceLine input[name=INVOICE\_ID]').val();

*// get the product ID for the current record*

postdata.PRODUCT\_ID =

$('#dlgEditInvoiceLine input[name=PRODUCT\_ID]').val();

**break**;

**case** "edit":

params.url = '@Url.Action("EditDetail")';

*// get current record id*

postdata.INVOICE\_LINE\_ID = selectedRow;

**break**;

**case** "del":

params.url = '@Url.Action("DeleteDetail")';

*// get current record id*

postdata.INVOICE\_LINE\_ID = selectedRow;

**break**;

}

},

afterSubmit: **function** (response, postdata) {

**var** responseData = response.responseJSON;

*// check the result for error messages*

**if** (responseData.hasOwnProperty("error")) {

**if** (responseData.error.length) {

**return** [false, responseData.error];

}

}

**else** {

*// refresh grid*

$(**this**).jqGrid(

'setGridParam',

{

datatype: 'json'

}

).trigger('reloadGrid');

}

**return** [true, "", 0];

}

};

};

}

Now we are done with creating the invoice module. Although the showProductWindow function that is used to select a product from the directory while filling out invoice lines is not examined here, it is totally similar to the showCustomerWindow function that we examined earlier to implement the selection of customers from the customer module.

An observant reader might have noticed that the functions for displaying the selection from the module and for displaying the module itself were almost identical. Something you could do yourself to improve the code is to move these functions into separate .js script files.

## Authentication

The ASP.NET technology has a powerful mechanism for managing authentication in .NET applications called ASP.NET Identity. The infrastructure of OWIN and AspNet Identity make it possible to perform both standard authentication and authentication via external services through accounts in Google, Twitter, Facebook, etc. The description of the ASP.NET Identity technology is quite comprehensive and goes beyond the scope of this publication but you can about it at <http://www.asp.net/identity>.

For our application, we will take a less complicated approach based on form authentication. To enable form authentication requires changes in the *web.config* configuration file. Find the <system.web> section and insert the following subsection inside it:

<authentication mode="Forms">

<forms name="cookies" timeout="2880" loginUrl="~/Account/Login"

defaultUrl="~/Invoice/Index"/>

</authentication>

Setting mode="Forms" enables form authentication. Some parameters need to follow it. The following list of parameters is available:

* **cookieless**: specifies whether cookie sets are used and how they are used. It can take the following values: **UseCookies** (specifies that the cookie sets will always be used independently of the device), **UseUri** (cookies sets are never used), **AutoDetect** (if the device supports cookie sets, they are used, otherwise, they are not used, a test determining their support is run in this case), **UseDeviceProfile** (if the device supports cookie sets, they are used, otherwise, they are not used, no test is run unlike in the previous case. Used by default).
* **defaultUrl**: specifies the URL to redirect to after authentication
* **domain**: specifies cookie sets for the entire domain. Thanks to that, it is possible to use the same cookie sets for the main domain and its subdomains. By default, its value is an empty string
* **loginUrl**: the URL for user authentication. The default value is "~/Account/Login"
* **name**: specifies the name for the cookie set. The default value is ".ASPXAUTH"
* **path**: specifies the path for the cookie set. The default value is "/"
* **requireSSL**: specifies whether an SSL connection is required for sending cookie sets. The default value is false
* **timeout**: specifies the timeout for cookies in minutes

In our application, we will store authentication data in the same database that stores all other data so that we will not need an additional connection string.

Now we need to create all the infrastructure required for authentication - models, controllers and views. We will create the WebUser model that will describe the user:

[Table("Firebird.WEBUSER")]

**public partial class** WEBUSER

{

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",

"CA2214:DoNotCallOverridableMethodsInConstructors")]

**public** WEBUSER()

{

WEBUSERINROLES = **new** HashSet<WEBUSERINROLE>();

}

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.None)]

**public int** WEBUSER\_ID { **get**; **set**; }

[Required]

[StringLength(63)]

**public string** EMAIL { **get**; **set**; }

[Required]

[StringLength(63)]

**public string** PASSWD { **get**; **set**; }

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",

"CA2227:CollectionPropertiesShouldBeReadOnly")]

**public virtual** ICollection<WEBUSERINROLE> WEBUSERINROLES { **get**; **set**; }

}

We'll add two more models: one for the description of roles (WEBROLE) and another one for binding the roles to users (WEBUSERINROLE).

[Table("Firebird.WEBROLE")]

**public partial class** WEBROLE

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.None)]

**public int** WEBROLE\_ID { **get**; **set**; }

[Required]

[StringLength(63)]

**public string** NAME { **get**; **set**; }

}

[Table("Firebird.WEBUSERINROLE")]

**public partial class** WEBUSERINROLE

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.None)]

**public int** ID { **get**; **set**; }

[Required]

**public int** WEBUSER\_ID { **get**; **set**; }

[Required]

**public int** WEBROLE\_ID { **get**; **set**; }

**public virtual** WEBUSER WEBUSER { **get**; **set**; }

**public virtual** WEBROLE WEBROLE { **get**; **set**; }

}

We will use the Fluent API to specify relations between WEBUSER and WEBUSERINROLE in the DbModel class.

…

**public virtual** DbSet<WEBUSER> WEBUSERS { **get**; **set**; }

**public virtual** DbSet<WEBROLE> WEBROLES { **get**; **set**; }

**public virtual** DbSet<WEBUSERINROLE> WEBUSERINROLES { **get**; **set**; }

…

**protected override void** OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<WEBUSER>()

.HasMany(e => e.WEBUSERINROLES)

.WithRequired(e => e.WEBUSER)

.WillCascadeOnDelete(**false**);

…

}

…

Since we use the Database First technology, tables in the database can be created automatically. I prefer to control the process so I am providing the following script for creating the additional tables:

**RECREATE TABLE** WEBUSER (

WEBUSER\_ID **INT NOT NULL**,

EMAIL **VARCHAR**(63) **NOT NULL**,

PASSWD **VARCHAR**(63) **NOT NULL**,

**CONSTRAINT** PK\_WEBUSER **PRIMARY KEY**(WEBUSER\_ID),

**CONSTRAINT** UNQ\_WEBUSER **UNIQUE**(EMAIL)

);

**RECREATE TABLE** WEBROLE (

WEBROLE\_ID **INT NOT NULL**,

**NAME VARCHAR**(63) **NOT NULL**,

**CONSTRAINT** PK\_WEBROLE **PRIMARY KEY**(WEBROLE\_ID),

**CONSTRAINT** UNQ\_WEBROLE **UNIQUE**(**NAME**)

);

**RECREATE TABLE** WEBUSERINROLE (

ID **INT NOT NULL**,

WEBUSER\_ID **INT NOT NULL**,

WEBROLE\_ID **INT NOT NULL**,

**CONSTRAINT** PK\_WEBUSERINROLE **PRIMARY KEY**(ID)

);

**ALTER TABLE** WEBUSERINROLE

**ADD CONSTRAINT** FK\_WEBUSERINROLE\_USER

**FOREIGN KEY** (WEBUSER\_ID) **REFERENCES** WEBUSER (WEBUSER\_ID);

**ALTER TABLE** WEBUSERINROLE

**ADD CONSTRAINT** FK\_WEBUSERINROLE\_ROLE

**FOREIGN KEY** (WEBROLE\_ID) **REFERENCES** WEBROLE (WEBROLE\_ID);

**RECREATE SEQUENCE** SEQ\_WEBUSER;

**RECREATE SEQUENCE** SEQ\_WEBROLE;

**RECREATE SEQUENCE** SEQ\_WEBUSERINROLE;

**SET** TERM ^;

**RECREATE TRIGGER** TBI\_WEBUSER

**FOR** WEBUSER

**ACTIVE BEFORE INSERT**

**AS**

**BEGIN**

**IF** (NEW.WEBUSER\_ID **IS NULL**) **THEN**

NEW.WEBUSER\_ID = **NEXT VALUE FOR** SEQ\_WEBUSER;

**END**^

**RECREATE TRIGGER** TBI\_WEBROLE

**FOR** WEBROLE

**ACTIVE BEFORE INSERT**

**AS**

**BEGIN**

**IF** (NEW.WEBROLE\_ID **IS NULL**) **THEN**

NEW.WEBROLE\_ID = **NEXT VALUE FOR** SEQ\_WEBROLE;

**END**^

**RECREATE TRIGGER** TBI\_WEBUSERINROLE

**FOR** WEBUSERINROLE

**ACTIVE BEFORE INSERT**

**AS**

**BEGIN**

**IF** (NEW.ID **IS NULL**) **THEN**

NEW.ID = **NEXT VALUE FOR** SEQ\_WEBUSERINROLE;

**END**^

**SET** TERM ;^

To test it, we'll add two users and two roles:

**INSERT INTO** WEBUSER (EMAIL, PASSWD) **VALUES** ('john', '12345');

**INSERT INTO** WEBUSER (EMAIL, PASSWD) **VALUES** ('alex', '123');

**COMMIT**;

**INSERT INTO** WEBROLE (**NAME**) **VALUES** ('admin');

**INSERT INTO** WEBROLE (**NAME**) **VALUES** ('manager');

**COMMIT**;

*-- Link users and roles*

**INSERT INTO** WEBUSERINROLE(WEBUSER\_ID, WEBROLE\_ID) **VALUES**(1, 1);

**INSERT INTO** WEBUSERINROLE(WEBUSER\_ID, WEBROLE\_ID) **VALUES**(1, 2);

**INSERT INTO** WEBUSERINROLE(WEBUSER\_ID, WEBROLE\_ID) **VALUES**(2, 2);

**COMMIT**;

**Comment about passwords**

Usually, some hash from the password is stored in an open form instead of the password itself, using the md5 algorithm, for example. For our example, we have simplified authentication somewhat.

We will not interact directly with the WebUser model during registration and authentication. Instead, we will add some special models to the project:

**namespace** FBMVCExample.Models

{

**using** System;

**using** System.Collections.Generic;

**using** System.ComponentModel.DataAnnotations;

**using** System.ComponentModel.DataAnnotations.Schema;

**using** System.Data.Entity.Spatial;

*// Login model*

**public class** LoginModel

{

[Required]

**public string** Name { **get**; **set**; }

[Required]

[DataType(DataType.Password)]

**public string** Password { **get**; **set**; }

}

*// Model for registering a new user*

**public class** RegisterModel

{

[Required]

**public string** Name { **get**; **set**; }

[Required]

[DataType(DataType.Password)]

**public string** Password { **get**; **set**; }

[Required]

[DataType(DataType.Password)]

[Compare("Password", ErrorMessage = " Passwords do not match ")]

**public string** ConfirmPassword { **get**; **set**; }

}

}

These models will be used for the authentication and registration views respectively. The authentication view is coded as follows:

@model FBMVCExample.Models.LoginModel

@{

ViewBag.Title = "Login";

}

<h2>Login</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div **class**="form-horizontal">

@Html.ValidationSummary(true)

<div **class**="form-group">

@Html.LabelFor(model => model.Name,

**new** { @**class** = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.Name)

@Html.ValidationMessageFor(model => model.Name)

</div>

</div>

<div **class**="form-group">

@Html.LabelFor(model => model.Password,

**new** { @**class** = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.Password)

@Html.ValidationMessageFor(model => model.Password)

</div>

</div>

<div **class**="form-group">

<div **class**="col-md-offset-2 col-md-10">

<input type="submit" value="Logon" **class**="btn btn-default" />

</div>

</div>

</div>

}

@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

}

The registration view, in turn, is coded as follows:

@model FBMVCExample.Models.RegisterModel

@{

ViewBag.Title = "Registration";

}

<h2>Регистрация</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div **class**="form-horizontal">

@Html.ValidationSummary(true)

<div **class**="form-group">

@Html.LabelFor(model => model.Name,

**new** { @**class** = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.Name)

@Html.ValidationMessageFor(model => model.Name)

</div>

</div>

<div **class**="form-group">

@Html.LabelFor(model => model.Password,

**new** { @**class** = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.Password)

@Html.ValidationMessageFor(model => model.Password)

</div>

</div>

<div **class**="form-group">

@Html.LabelFor(model => model.ConfirmPassword,

**new** { @**class** = "control-label col-md-2" })

<div **class**="col-md-10">

@Html.EditorFor(model => model.ConfirmPassword)

@Html.ValidationMessageFor(model => model.ConfirmPassword)

</div>

</div>

<div **class**="form-group">

<div **class**="col-md-offset-2 col-md-10">

<input type="submit" value="Register"

**class**="btn btn-default" />

</div>

</div>

</div>

}

@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

}

**Comment about users**

The model, views and controllers for user authentication and registration are made as simple as possible in this example. A user usually has a lot more attributes than just a username and a password.

Now let us add one more controller--AccountController-- with the following contents:

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Web;

**using** System.Web.Mvc;

**using** System.Web.Security;

**using** FBMVCExample.Models;

**namespace** FBMVCExample.Controllers

{

**public class** AccountController : Controller

{

**public** ActionResult Login()

{

**return** View();

}

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Login(LoginModel model)

{

**if** (ModelState.IsValid)

{

*// search user in db*

WEBUSER user = **null**;

**using** (DbModel db = **new** DbModel())

{

user = db.WEBUSERS.FirstOrDefault(

u => u.EMAIL == model.Name &&

u.PASSWD == model.Password);

}

*// if you find a user with a login and password,*

*// then remember it and do a redirect to the start page*

**if** (user != **null**)

{

FormsAuthentication.SetAuthCookie(model.Name, **true**);

**return** RedirectToAction("Index", "Invoice");

}

**else**

{

ModelState.AddModelError("",

" A user with such a username and password does not exist ");

}

}

**return** View(model);

}

[Authorize(Roles = "admin")]

**public** ActionResult Register()

{

**return** View();

}

[HttpPost]

[ValidateAntiForgeryToken]

**public** ActionResult Register(RegisterModel model)

{

**if** (ModelState.IsValid)

{

WEBUSER user = **null**;

**using** (DbModel db = **new** DbModel())

{

user = db.WEBUSERS.FirstOrDefault(u => u.EMAIL == model.Name);

}

**if** (user == **null**)

{

*// create a new user*

**using** (DbModel db = **new** DbModel())

{

*// get a new identifier using a sequence*

**int** userId = db.NextValueFor("SEQ\_WEBUSER");

db.WEBUSERS.Add(**new** WEBUSER {

WEBUSER\_ID = userId,

EMAIL = model.Name,

PASSWD = model.Password

});

db.SaveChanges();

user = db.WEBUSERS.Where(u => u.WEBUSER\_ID == userId)

.FirstOrDefault();

*// find the role of manager*

*// This role will be the default role, i.e.*

*// will be issued automatically upon registration*

var defaultRole =

db.WEBROLES

.Where(r => r.NAME == "manager")

.FirstOrDefault();

*// Assign the default role to the newly added user*

**if** (user != **null** && defaultRole != **null**)

{

db.WEBUSERINROLES.Add(**new** WEBUSERINROLE

{

WEBUSER\_ID = user.WEBUSER\_ID,

WEBROLE\_ID = defaultRole.WEBROLE\_ID

});

db.SaveChanges();

}

}

*// if the user is successfully added to the database*

**if** (user != **null**)

{

FormsAuthentication.SetAuthCookie(model.Name, **true**);

**return** RedirectToAction("Login", "Account");

}

}

**else**

{

ModelState.AddModelError("",

"User with such login already exists");

}

}

**return** View(model);

}

**public** ActionResult Logoff()

{

FormsAuthentication.SignOut();

**return** RedirectToAction("Login", "Account");

}

}

}

Note the attribute [Authorize(Roles = "admin")] to stipulate that only a user with the admin role can perform the user registration operation. This mechanism is called an authentication filter. We will get back to it a bit later.

We add a new user to the database during registration and check during authentication as to whether that user exists. If the user is found, we use form authentication to set a cookie, as follows:

FormsAuthentication.SetAuthCookie(model.Name, **true**);

All information about a user in Asp.Net MVC is stored in the HttpContext.User property that implements the **IPrincipal** interface defined in the System.Security.Principal namespace.

The **IPrincipal** interface defines the **Identity** property that stores the object of the **IIdentity** interface describing the current user.

The **IIdentity** interface has the following properties:

* **AuthenticationType**: authentication type
* **IsAuthenticated**: returns true if the user is logged in
* **Name**: the username in the system

To determine whether a user is logged in, ASP.NET MVC receives cookies from the browser and if the user is logged in, the IIdentity.IsAuthenticated property is set to true, and the Name property gets the username as its value.

Now let us add authentication items using universal providers mechanism.

Universal providers offer a ready-made authentication functionality. But at the same time, these providers are flexible enough that we can redefine them to work in whatever way we need them to. It is not necessary to redefine and use all four providers which is convenient if we do not need all of the fancy ASP.NET Identity features, but just a very simple authentication system.

Next, we will redefine the role provider. We need to add the Microsoft.AspNet.Providers package via NuGet to do this.

To define the role provider itself, first we add the *Providers* folder to the project and then add a new MyRoleProvider class to it:

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Web;

**using** System.Web.Security;

**using** FBMVCExample.Models;

**namespace** FBMVCExample.Providers

{

**public class** MyRoleProvider : RoleProvider

{

*/// <summary>*

*/// Returns the list of user roles*

*/// </summary>*

*/// <param name="username">Username</param>*

*/// <returns></returns>*

**public override string**[] GetRolesForUser(**string** username)

{

**string**[] roles = **new string**[] { };

**using** (DbModel db = **new** DbModel())

{

*// Get the user*

WEBUSER user = db.WEBUSERS.FirstOrDefault(

u => u.EMAIL == username);

**if** (user != **null**)

{

*// fill in an array of available roles*

**int** i = 0;

roles = **new string**[user.WEBUSERINROLES.Count];

**foreach** (var rolesInUser **in** user.WEBUSERINROLES)

{

roles[i] = rolesInUser.WEBROLE.NAME;

i++;

}

}

}

**return** roles;

}

*/// <summary>*

*/// Creating a new role*

*/// </summary>*

*/// <param name="roleName">Role name</param>*

**public override void** CreateRole(**string** roleName)

{

**using** (DbModel db = **new** DbModel())

{

WEBROLE newRole = **new** WEBROLE() { NAME = roleName };

db.WEBROLES.Add(newRole);

db.SaveChanges();

}

}

*/// <summary>*

*/// Returns whether the user role is present*

*/// </summary>*

*/// <param name="username">User name</param>*

*/// <param name="roleName">Role name</param>*

*/// <returns></returns>*

**public override bool** IsUserInRole(**string** username, **string** roleName)

{

**bool** outputResult = **false**;

**using** (DbModel db = **new** DbModel())

{

var userInRole =

**from** ur **in** db.WEBUSERINROLES

**where** ur.WEBUSER.EMAIL == username &&

ur.WEBROLE.NAME == roleName

**select new** { id = ur.ID };

outputResult = userInRole.Count() > 0;

}

**return** outputResult;

}

**public override void** AddUsersToRoles(**string**[] usernames,

**string**[] roleNames)

{

**throw new** NotImplementedException();

}

**public override string** ApplicationName

{

**get** { **throw new** NotImplementedException(); }

**set** { **throw new** NotImplementedException(); }

}

**public override bool** DeleteRole(**string** roleName,

**bool** throwOnPopulatedRole)

{

**throw new** NotImplementedException();

}

**public override string**[] FindUsersInRole(**string** roleName,

**string** usernameToMatch)

{

**throw new** NotImplementedException();

}

**public override string**[] GetAllRoles()

{

**throw new** NotImplementedException();

}

**public override string**[] GetUsersInRole(**string** roleName)

{

**throw new** NotImplementedException();

}

**public override void** RemoveUsersFromRoles(**string**[] usernames,

**string**[] roleNames)

{

**throw new** NotImplementedException();

}

**public override bool** RoleExists(**string** roleName)

{

**throw new** NotImplementedException();

}

}

}

For the purpose of illustration, three methods are redefined. The first one is GetRolesForUser, for obtaining a set of roles for a specified user. The second method is CreateRole, for creating a role. The third, IsUserInRole, determines whether the user has a specified role in the system.

To use the role provider in the application, we need to add its definition to the configuration file. Open the web.config file and remove the definition of providers added automatically during the installation of the Microsoft.AspNet.Providers package. Next, we insert our provider within the system.web section:

<system.web>

<authentication mode="Forms">

<forms name="cookies" timeout="2880" loginUrl="~/Account/Login"

defaultUrl="~/Invoice/Index"/>

</authentication>

<roleManager enabled="true" defaultProvider="MyRoleProvider">

<providers>

<add name="MyRoleProvider"

type="FBMVCExample.Providers.MyRoleProvider" />

</providers>

</roleManager>

</system.web>

Now we can limit access to the methods of various controllers using the *Authorize* attribute. We have already seen how it is used in the AccountController controller:

[Authorize(Roles = "admin")]

**public** ActionResult Register()

{…

This filter can be used at two levels: on a controller as a whole and on an individual operation of a controller. We will set different rights for our main controllers: CustomerController, InvoiceController and ProductController. In our sample, a user with the manager role can view and edit data in all three tables. Setting a filter for the InvoiceController controller would be coded as follows:

[Authorize(Roles = "manager")]

**public class** InvoiceController : Controller

{

**private** DbModel db = **new** DbModel();

*// Show view*

**public** ActionResult Index()

{

**return** View();

}

…

Setting filters in the other controllers can be implemented in a similar manner.

## Conclusion

The source code for the sample application can be obtained from the URL <https://github.com/sim1984/FBMVCExample>