Firebird 3.0 Developer's Guide PRE-BETA

Release 0.1

Denis Simonov
Megatranslations Ltd, translation of text from
the original Russian to English: Dimitry Borodin
Editor of the translated text: Helen Borrie

11 November 2017, document version 0.105

Firebird 3.0 Developer's Guide PRE-BETA

Release 0.1

11 November 2017, document version 0.105
by Denis Simonov
Megatranslations Ltd, translation of text from the original Russian to English: Dimitry Borodin
Editor of the translated text: Helen Borrie

Abstract

...

Table of Contents

List of Figures

List of Tables

Chapter 1

About the Firebird Developer's Guide

for Firebird 3.0

Blurb...

Subject Matter

This volume consists of chapters that walk through the development of a simple application for several language platforms, notably Delphi, Microsoft Entity Framework and MVC.NET (“Model-View-Controller”) for web applications, PHP and Java with the Spring framework. It is hoped that the work will grow in time, with contributions from authors using other stacks with Firebird.

About the Author

Denis Simonov....

Translation

Development of the original Russian version was sponsored by IBSurgeon and Moscow Exchange Bank. A crowd-funding campaign was launched by the Firebird Foundation in 2017 to fund the translation into English to provide this document as the foundation for translation by Firebird Project document writers into other languages.

The campaign suceeded in raising enough to get the process under way.

. . . and More Translation

Once the DocBook source appears in GitHub, we hope the trusty translators will start making versions in German, Japanese, Italian, French, Portuguese, Spanish, Czech. Certainly, we never have enough translators so please, you Firebirders who have English as a second language, do consider translating some chapters into your first language.

Acknowledgments

We acknowledge these contributions of sponsors and donors with gratitude and thank you all for stepping up.

Sponsors and Other Donors

Sponsors of the Russian Language version of this Guide

Moscow Exchange (Russia)

> *Moscow Exchange is the largest exchange holding in Russia and Eastern Europe, founded on December 19, 2011, through the consolidation of the MICEX (founded in 1992) and RTS (founded in 1995) exchange groups. Moscow Exchange ranks among the world's top 20 exchanges by trading in bonds and by the total capitalization of shares traded, as well as among the 10 largest exchange platforms for trading derivatives.*

IBSurgeon (ibase.ru) (Russia)

> *Technical support and developer of administrator tools for the Firebird DBMS.*

Sponsors of the Translation Project

Firebird Developers' Day contributors, 2017 (Brazil)
Syntess Software BV (Netherlands)

Other Donors

Listed below are the names of companies and individuals whose cash contributions covered the costs for translation into English, editing of the raw, translated text and conversion of the whole into the Firebird Project's standard DocBook 4 documentation source format.

Peter Lee (Australia)
Thomas M. Conrad (U.S.A.)
Doug Chamberlin (U.S.A)
Francis Moore (U.K.)
Juan Antonio Mendoza Gil
Robert Firl
Aparecido Silva
Deon van Niekerk
Martin Mutiku
Myles Wakeham (U.S.A.)
Pal Lillejord
Chong Ray
Jean-Marc Couret
Guiseppe Minutillo (Italy)
Chris Mathews (U.S.A.)
James Batson
Paolo Sciarrini (Italy)
Arkadiusz Wolanski (Poland)
Michele Denys
Marknadsinformation i Sverige AB (Sweden)
Transdata GmbH (Germany)
Francis Mullan (South Africa)
Laurent Guétin (Burkina Faso)
Massimilliano Coros
Roland van Morckhoven (Netherlands)
Andrew Kipcharsky (Russian Federation)
Hartmuth Prüfer (Germany)
Martin Köditz (Germany)
Mark Rotteveel (Netherlands)
Roknic Dusan
Solucionalia Consultores Auditores, S.L. (Spain)
Gabor Boros
Artur Henneberg
Ivan Cruz
Kjell Rilbe (Sweden)
Antonis Tsourinakis (Greece)
Gerdus van Zyl (South Africa)
Michael Trowe

Ralf Stegemann
Alessandro Marcellini
Alexander K. Bowie
Cserna Zsombor
Juergen Bachsteffel (Germany)
Michele Giordano
Shaymon Gracia Campos
Juan Carlos Ramirez
Alberto Fornes Llodra
Jozo Leko
Jose Antonio Amate Belchi
Vasily Vasilov

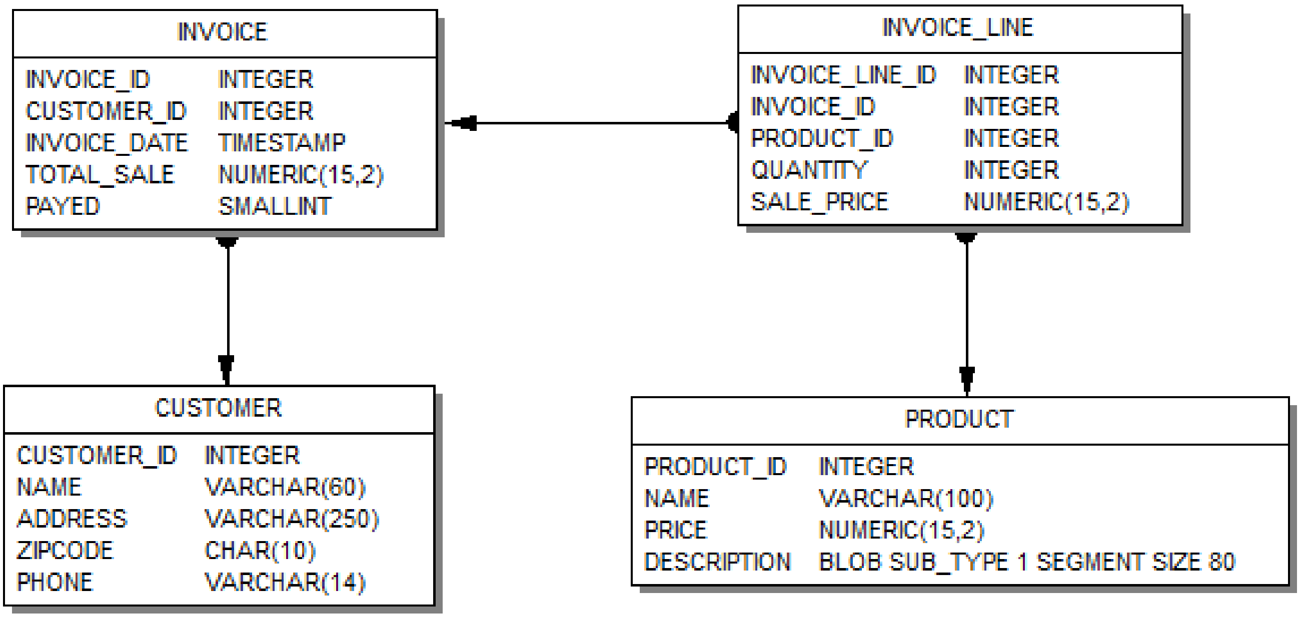
Chapter 2

The examples.fdb Database

Before we explore the process of creating applications in various programming languages, we will walk through the creation and preparation of the database that is used as the back-end to all of the sample projects.

The applications work with a database based on the model illustrated in this diagram:

Figure 2.1. Model of the examples.fdb database



> **Disclaimer**
>
> This chapter does not attempt to provide a tutorial about database design or SQL syntax. The model is made as simple as possible to avoid cluttering the application development techniques with topics about database modeling and development. We hope some readers might be enlightened by our approach to maintaining inter-related data using stored procedures. The scripts are all here for you to refer to as you work your way through the projects.
>
> The requirements for your real-life projects are undoubtedly different from and much more complicated than those for our example projects.

Database Creation Script

The tool used here to create the database from a script is *isql*, that is installed with all the other executables in every Firebird server installation. You could use any other administration tool for Firebird, such as FlameRobin, SQLLY Studio, IBExpert or others.

We will assume that you are working in Windows. Obviously, the formats of path names will differ on other file systems (Linux, Apple Mac, etc.) but the *isql* tool works the same on all platforms.

Run *isql* and enter the following script after the `SQL>` prompt appears :

```
CREATE DATABASE 'localhost:D:\fbdata\2.5\examples.fdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 8192 DEFAULT CHARACTER SET UTF8;
```

> **Important**
>
> The straight single quotes around the user and password arguments are not optional in Firebird 2.5 and lower versions because, in the CREATE DATABASE syntax, both are strings.
>
> In Firebird 3, the rules changed. User names became identifiers and no longer require single quotes. They can be made case-sensitive by enclosing the name in DOUBLE quotes, so you need to be aware of how that user is registered in the security database. Passwords are still strings.
>
> Quotes in the statement are not interchangeable with curly quotes, angle quotes or any other kind of quotes.

The user whose name and password are cited in the CREATE DATABASE statement becomes the owner of the database and has full access to all metadata objects. It is not essential that SYSDBA be the owner of a database. Any user can be the owner, which has the same access as SYSDBA in this database.

The actively supported versions of Firebird support the following page sizes: 4096, 8192 and 16384. The page size of 8192 is good for most cases.

The optional DEFAULT CHARACTER SET clause specifies the default character set for string data types. Character sets are applied to the CHAR, VARCHAR and BLOB TEXT data types. You can study the list of available language encodings in an Appendix to the Firebird Language Reference manual. All up-to-date programming languages support UTF8 so we choose this encoding.

Now we can exit the *isql* session by typing the following command:

```
EXIT;
```

Database Aliases

Databases are accessed locally and remotely by their physical file path on the server. Before you start to use a database, it is useful and wise to register an alias for its file path and to use the alias for all connections. It saves typing and, to some degree, it offers a little extra security from snoopers by obscuring the physical location of your database file in the connection string.

In Firebird 2.5, the alias of a database is registered in the `aliases.conf` file as follows:

```
examples = D:\fbdata\2.5\examples.fdb
```

In Firebird 3.0, the alias of a database is registered in the `databases.conf` file. Along with the alias for the database, some database-level parameters can be configured there: page cache size, the size of RAM for sorting and several others, e.g.,

```
examples = D:\fbdata\3.0\examples.fdb
{
    DefaultDbCachePages = 16K
    TempCacheLimit = 512M
}
```

> **Tip**
>
> You can use an alias even before the database exists. It is valid to substitute the full file path with the alias in the CREATE DATABASE statement.

Creating the Database Objects

Now let us create a script for building the database objects.

Domains

First, we define some domains that we will use in column definitions.

```
CREATE DOMAIN D_BOOLEAN AS
SMALLINT
CHECK (VALUE IN (0, 1));

COMMENT ON DOMAIN D_BOOLEAN IS
'Boolean type. 0 - FALSE, 1- TRUE';

CREATE DOMAIN D_MONEY AS
NUMERIC(15,2);

CREATE DOMAIN D_ZIPCODE AS
CHAR(10) CHARACTER SET UTF8
CHECK (TRIM(TRAILING FROM VALUE) SIMILAR TO '[0-9]+');

COMMENT ON DOMAIN D_ZIPCODE IS
'Zip code';
```

BOOLEAN Type

In Firebird 3.0, there is a native BOOLEAN type. Some drivers do not support it, due to its relatively recent appearance in Firebird's SQL lexicon. With that in mind,, our applications will be built on a database that will work with either Firebird 2.5 or Firebird 3.0.

> **Important**
>
> Before Firebird 3, servers could connect clients to databases that were created under older Firebird versions. Firebird 3 can connect only to databases that were created on or restored under Firebird 3.

Primary Tables

Now let us proceed to the primary tables. The first will be the CUSTOMER table. We will create a sequence (a generator) for its primary key and a corresponding trigger for implementing it as an auto-incrementing column. We will do the same for each of the tables.

```
CREATE GENERATOR GEN_CUSTOMER_ID;

CREATE TABLE CUSTOMER (
  CUSTOMER_ID INTEGER NOT NULL,
  NAME VARCHAR(60) NOT NULL,
  ADDRESS VARCHAR(250),
  ZIPCODE D_ZIPCODE,
  PHONE VARCHAR(14),
  CONSTRAINT PK_CUSTOMER PRIMARY KEY (CUSTOMER_ID)
);

SET TERM ^ ;

CREATE OR ALTER TRIGGER CUSTOMER_BI FOR CUSTOMER
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.CUSTOMER_ID IS NULL) THEN
    NEW.CUSTOMER_ID = NEXT VALUE FOR GEN_CUSTOMER_ID;
END
^

SET TERM ; ^

COMMENT ON TABLE CUSTOMER IS
'Customers';

COMMENT ON COLUMN CUSTOMER.CUSTOMER_ID IS
'Customer Id';

COMMENT ON COLUMN CUSTOMER.NAME IS
'Name';

COMMENT ON COLUMN CUSTOMER.ADDRESS IS
'Address';

COMMENT ON COLUMN CUSTOMER.ZIPCODE IS
'Zip Code';

COMMENT ON COLUMN CUSTOMER.PHONE IS
'Phone';
```

Note

- In Firebird 3.0, you can use IDENTITY columns as auto-incremental fields. The script for creating the table would then be as follows:

```
CREATE TABLE CUSTOMER (
  CUSTOMER_ID INTEGER GENERATED BY DEFAULT AS IDENTITY,
  NAME VARCHAR(60) NOT NULL,
  ADDRESS VARCHAR(250),
  ZIPCODE D_ZIPCODE,
  PHONE VARCHAR(14),
  CONSTRAINT PK_CUSTOMER PRIMARY KEY (CUSTOMER_ID)
);
```

- In Firebird 3.0, you need the USAGE privilege to use a sequence (generator) so you will have to add the following line to the script:

```
GRANT USAGE ON SEQUENCE GEN_CUSTOMER_ID TO TRIGGER CUSTOMER_BI;
```

Now we construct a script for creating the PRODUCT table:

```
CREATE GENERATOR GEN_PRODUCT_ID;

CREATE TABLE PRODUCT (
  PRODUCT_ID INTEGER NOT NULL,
  NAME VARCHAR(100) NOT NULL,
  PRICE D_MONEY NOT NULL,
  DESCRIPTION BLOB SUB_TYPE 1 SEGMENT SIZE 80,
  CONSTRAINT PK_PRODUCT PRIMARY KEY (PRODUCT_ID)
);

SET TERM ^;

CREATE OR ALTER TRIGGER PRODUCT_BI FOR PRODUCT
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.PRODUCT_ID IS NULL) THEN
    NEW.PRODUCT_ID = NEXT VALUE FOR GEN_PRODUCT_ID;
END
^

SET TERM ;^

COMMENT ON TABLE PRODUCT IS
'Goods';

COMMENT ON COLUMN PRODUCT.PRODUCT_ID IS
'Product Id';

COMMENT ON COLUMN PRODUCT.NAME IS
'Name';

COMMENT ON COLUMN PRODUCT.PRICE IS
```

```
'Price';

COMMENT ON COLUMN PRODUCT.DESCRIPTION IS
'Description';
```

> **Note**
>
> In Firebird 3.0, you need to add the command for granting the USAGE privilege for a sequence (generator) to the script:
>
> ```
> GRANT USAGE ON SEQUENCE GEN_PRODUCT_ID TO TRIGGER PRODUCT_BI;
> ```

Secondary Tables

The script for creating the INVOICE table:

```
CREATE GENERATOR GEN_INVOICE_ID;

CREATE TABLE INVOICE (
  INVOICE_ID INTEGER NOT NULL,
  CUSTOMER_ID INTEGER NOT NULL,
  INVOICE_DATE TIMESTAMP,
  TOTAL_SALE D_MONEY,
  PAID D_BOOLEAN DEFAULT 0 NOT NULL,
  CONSTRAINT PK_INVOICE PRIMARY KEY (INVOICE_ID)
);

ALTER TABLE INVOICE ADD CONSTRAINT FK_INVOCE_CUSTOMER
FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER (CUSTOMER_ID);

CREATE INDEX INVOICE_IDX_DATE ON INVOICE (INVOICE_DATE);

SET TERM ^;

CREATE OR ALTER TRIGGER INVOICE_BI FOR INVOICE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.INVOICE_ID IS NULL) THEN
    NEW.INVOICE_ID = GEN_ID(GEN_INVOICE_ID,1);
END
^

SET TERM ;^

COMMENT ON TABLE INVOICE IS
'Invoices';

COMMENT ON COLUMN INVOICE.INVOICE_ID IS
'Invoice number';
```

```
COMMENT ON COLUMN INVOICE.CUSTOMER_ID IS
'Customer Id';

COMMENT ON COLUMN INVOICE.INVOICE_DATE IS
'The date of issuance invoices';

COMMENT ON COLUMN INVOICE.TOTAL_SALE IS
'Total sum';

COMMENT ON COLUMN INVOICE.PAID IS
'Paid';
```

The INVOICE_DATE column is indexed because we will be filtering invoices by date to enable the records to be selected by a work period that will be application-defined by a start date and an end date.

> **Note**
>
> In Firebird 3.0, you need to add the command for granting the USAGE privilege for a sequence (generator) to the script:
>
> ```
> GRANT USAGE ON SEQUENCE GEN_INVOICE_ID TO TRIGGER INVOICE_BI;
> ```

The script for creating the INVOICE_LINE table:

```
CREATE GENERATOR GEN_INVOICE_LINE_ID;

CREATE TABLE INVOICE_LINE (
INVOICE_LINE_ID INTEGER NOT NULL,
INVOICE_ID INTEGER NOT NULL,
PRODUCT_ID INTEGER NOT NULL,
QUANTITY NUMERIC(15,0) NOT NULL,
SALE_PRICE D_MONEY NOT NULL,
CONSTRAINT PK_INVOICE_LINE PRIMARY KEY (INVOICE_LINE_ID)
);

ALTER TABLE INVOICE_LINE ADD CONSTRAINT FK_INVOICE_LINE_INVOICE
FOREIGN KEY (INVOICE_ID) REFERENCES INVOICE (INVOICE_ID);

ALTER TABLE INVOICE_LINE ADD CONSTRAINT FK_INVOICE_LINE_PRODUCT
FOREIGN KEY (PRODUCT_ID) REFERENCES PRODUCT (PRODUCT_ID);

SET TERM ^;

CREATE OR ALTER TRIGGER INVOICE_LINE_BI FOR INVOICE_LINE
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.INVOICE_LINE_ID IS NULL) THEN
    NEW.INVOICE_LINE_ID = NEXT VALUE FOR GEN_INVOICE_LINE_ID;
END
^

SET TERM ;^
```

```
COMMENT ON TABLE INVOICE_LINE IS
'Invoice lines';

COMMENT ON COLUMN INVOICE_LINE.INVOICE_LINE_ID IS
'Invoice line Id';

COMMENT ON COLUMN INVOICE_LINE.INVOICE_ID IS
'Invoice number';

COMMENT ON COLUMN INVOICE_LINE.PRODUCT_ID IS
'Product Id';

COMMENT ON COLUMN INVOICE_LINE.QUANTITY IS
'Quantity';

COMMENT ON COLUMN INVOICE_LINE.SALE_PRICE IS
'Price';
```

> **Note**
>
> In Firebird 3.0, you need to add the command for granting the USAGE privilege for a sequence (generator) to the script:
>
> ```
> GRANT USAGE ON SEQUENCE GEN_INVOICE_LINE_ID TO TRIGGER INVOICE_LINE_BI;
> ```

Stored Procedures

Some parts of the business logic will be implemented by means of stored procedures.

Adding a new invoice

The procedure for adding a new invoice is quite simple:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE SP_ADD_INVOICE (
  INVOICE_ID INTEGER,
  CUSTOMER_ID INTEGER,
  INVOICE_DATE TIMESTAMP = CURRENT_TIMESTAMP)
AS
BEGIN
  INSERT INTO INVOICE (
    INVOICE_ID,
    CUSTOMER_ID,
    INVOICE_DATE,
    TOTAL_SALE,
    PAID
  )
  VALUES (
```

```
    :INVOICE_ID,
    :CUSTOMER_ID,
    :INVOICE_DATE,
    0,
    0
  );
END
^

SET TERM ;^

COMMENT ON PROCEDURE SP_ADD_INVOICE IS
'Adding Invoice';

COMMENT ON PARAMETER SP_ADD_INVOICE.INVOICE_ID IS
'Invoice number';

COMMENT ON PARAMETER SP_ADD_INVOICE.CUSTOMER_ID IS
'Customer Id';

COMMENT ON PARAMETER SP_ADD_INVOICE.INVOICE_DATE IS
'Date';

GRANT INSERT ON INVOICE TO PROCEDURE SP_ADD_INVOICE;
```

Editing an invoice

The procedure for editing an invoice is a bit more complicated. We will include a rule to block further editing of an invoice once it is paid. We will create an exception that will be raised if an attempt is made to modify a paid invoice.

```
CREATE EXCEPTION E_INVOICE_ALREADY_PAYED 'Change is impossible, invoice paid.';
The stored procedure for editing an invoice:
SET TERM ^;

CREATE OR ALTER PROCEDURE SP_EDIT_INVOICE (
  INVOICE_ID INTEGER,
  CUSTOMER_ID INTEGER,
  INVOICE_DATE TIMESTAMP)
AS
BEGIN
  IF (EXISTS(SELECT *
             FROM INVOICE
             WHERE INVOICE_ID = :INVOICE_ID
               AND PAID = 1)) THEN
    EXCEPTION E_INVOICE_ALREADY_PAYED;

  UPDATE INVOICE
  SET CUSTOMER_ID = :CUSTOMER_ID,
  INVOICE_DATE = :INVOICE_DATE
  WHERE INVOICE_ID = :INVOICE_ID;
END
^
```

```
SET TERM ;^

COMMENT ON PROCEDURE SP_EDIT_INVOICE IS
'Editing invoice';

COMMENT ON PARAMETER SP_EDIT_INVOICE.INVOICE_ID IS
'Invoice number';

COMMENT ON PARAMETER SP_EDIT_INVOICE.CUSTOMER_ID IS
'Customer Id';

COMMENT ON PARAMETER SP_EDIT_INVOICE.INVOICE_DATE IS
'Date';

GRANT SELECT,UPDATE ON INVOICE TO PROCEDURE SP_EDIT_INVOICE;
```

Note

In Firebird 3.0, the USAGE privilege is required for exceptions so we need to add the following line:

```
GRANT USAGE ON EXCEPTION E_INVOICE_ALREADY_PAYED TO PROCEDURE SP_EDIT_INVOICE;
```

Deleting an invoice

The procedure SP_DELETE_INVOICE procedure checks whether the invoice is paid and raises an exception if it is:

```
SET TERM ^ ;

CREATE OR ALTER PROCEDURE SP_DELETE_INVOICE (
  INVOICE_ID INTEGER)
AS
BEGIN
  IF (EXISTS(SELECT * FROM INVOICE
             WHERE INVOICE_ID = :INVOICE_ID
               AND PAID = 1)) THEN
    EXCEPTION E_INVOICE_ALREADY_PAYED;

  DELETE FROM INVOICE WHERE INVOICE_ID = :INVOICE_ID;
END
^

SET TERM ;^

COMMENT ON PROCEDURE SP_DELETE_INVOICE IS
'Deleting invoices';

GRANT SELECT,DELETE ON INVOICE TO PROCEDURE SP_DELETE_INVOICE;
```

> **Note**
>
> In Firebird 3.0, the USAGE privilege is required for exceptions so we need to add the following line:
>
> ```
> GRANT USAGE ON EXCEPTION E_INVOICE_ALREADY_PAYED TO PROCEDURE SP_DELETE_INVOICE;
> ```

Paying an invoice

We will add one more procedure for paying an invoice:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE SP_PAY_FOR_INVOICE (
INVOICE_ID INTEGER)
AS
BEGIN
  IF (EXISTS(SELECT *
             FROM INVOICE
             WHERE INVOICE_ID = :INVOICE_ID
               AND PAID = 1)) THEN
EXCEPTION E_INVOICE_ALREADY_PAYED;
UPDATE INVOICE
SET PAID = 1
WHERE INVOICE_ID = :INVOICE_ID;
END
^

SET TERM ;^

COMMENT ON PROCEDURE SP_PAY_FOR_INVOICE IS
'Payment of invoices';

COMMENT ON PARAMETER SP_PAY_FOR_INVOICE.INVOICE_ID IS
'Invoice number';

GRANT SELECT,UPDATE ON INVOICE TO PROCEDURE SP_PAY_FOR_INVOICE;
```

> **Note**
>
> In Firebird 3.0, the USAGE privilege is required for exceptions so we need to add the following line:
>
> ```
> GRANT USAGE ON EXCEPTION E_INVOICE_ALREADY_PAYED TO PROCEDURE SP_PAY_FOR_INVOICE;
> ```

Invoice Line Items

Procedures for managing invoice items will check whether the invoice is paid and block any attempt to alter the line items of paid invoices. They will also correct the invoice total according to the amount of the product sold and its price.

Adding line items

The procedure for adding a line item to an invoice:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE SP_ADD_INVOICE_LINE (
  INVOICE_ID INTEGER,
  PRODUCT_ID INTEGER,
  QUANTITY INTEGER)
AS
  DECLARE sale_price D_MONEY;
  DECLARE paid D_BOOLEAN;
BEGIN
  SELECT
    paid
  FROM
    invoice
  WHERE
    invoice_id = :invoice_id
  INTO :paid;

  -- It does not allow you to edit already paid invoice.
  IF (paid = 1) THEN
    EXCEPTION E_INVOICE_ALREADY_PAYED;

  SELECT
    price
  FROM
    product
  WHERE
    product_id = :product_id
  INTO :sale_price;

INSERT INTO invoice_line (
  invoice_line_id,
  invoice_id,
  product_id,
  quantity,
  sale_price)
VALUES (
  NEXT VALUE FOR gen_invoice_line_id,
  :invoice_id,
  :product_id,
  :quantity,
  :sale_price);

  -- Increase the amount of the account.
  UPDATE invoice
  SET total_sale = COALESCE(total_sale, 0) + :sale_price * :quantity
  WHERE invoice_id = :invoice_id;

END
^
SET TERM ;^
```

```
COMMENT ON PROCEDURE SP_ADD_INVOICE_LINE IS
'Adding line invoices';

COMMENT ON PARAMETER SP_ADD_INVOICE_LINE.INVOICE_ID IS
'Invoice number';

COMMENT ON PARAMETER SP_ADD_INVOICE_LINE.PRODUCT_ID IS
'Product Id';

COMMENT ON PARAMETER SP_ADD_INVOICE_LINE.QUANTITY IS
'Quantity';

GRANT SELECT, UPDATE ON INVOICE TO PROCEDURE SP_ADD_INVOICE_LINE;
GRANT SELECT ON PRODUCT TO PROCEDURE SP_ADD_INVOICE_LINE;
GRANT INSERT ON INVOICE_LINE TO PROCEDURE SP_ADD_INVOICE_LINE;
-- only Firebird 3.0 and above
GRANT USAGE ON EXCEPTION E_INVOICE_ALREADY_PAYED TO PROCEDURE SP_ADD_INVOICE_LINE;
GRANT USAGE ON SEQUENCE GEN_INVOICE_LINE_ID TO PROCEDURE SP_ADD_INVOICE_LINE;
```

Editing line items

The procedure for editing an invoice line item:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE SP_EDIT_INVOICE_LINE (
  INVOICE_LINE_ID INTEGER,
  QUANTITY INTEGER)
AS
  DECLARE invoice_id INT;
  DECLARE price D_MONEY;
  DECLARE paid D_BOOLEAN;
BEGIN
  SELECT
    product.price,
    invoice.invoice_id,
    invoice.paid
  FROM
    invoice_line
  JOIN invoice ON invoice.invoice_id = invoice_line.invoice_id
  JOIN product ON product.product_id = invoice_line.product_id
  WHERE
    invoice_line.invoice_line_id = :invoice_line_id
  INTO
    :price,
    :invoice_id,
    :paid;

  -- It does not allow you to edit an already paid invoice.
  IF (paid = 1) THEN
    EXCEPTION E_INVOICE_ALREADY_PAYED;

  -- Update price and quantity.
  UPDATE invoice_line
```

```
  SET sale_price = :price,
      quantity = :quantity
  WHERE invoice_line_id = :invoice_line_id;
  -- Now update the amount of the account.
  MERGE INTO invoice
  USING (
    SELECT
      invoice_id,
      SUM(sale_price * quantity) AS total_sale
    FROM invoice_line
    WHERE invoice_id = :invoice_id
    GROUP BY invoice_id) L
  ON invoice.invoice_id = L.invoice_id
  WHEN MATCHED THEN
    UPDATE SET total_sale = L.total_sale;
END
^

SET TERM ;^

COMMENT ON PROCEDURE SP_EDIT_INVOICE_LINE IS
'Editing invoice line';

COMMENT ON PARAMETER SP_EDIT_INVOICE_LINE.INVOICE_LINE_ID IS
'Invoice line id';

COMMENT ON PARAMETER SP_EDIT_INVOICE_LINE.QUANTITY IS
'Quantity';

GRANT SELECT,UPDATE ON INVOICE_LINE TO PROCEDURE SP_EDIT_INVOICE_LINE;
GRANT SELECT,UPDATE ON INVOICE TO PROCEDURE SP_EDIT_INVOICE_LINE;
GRANT SELECT ON PRODUCT TO PROCEDURE SP_EDIT_INVOICE_LINE;
-- only Firebird 3.0 and above
GRANT USAGE ON EXCEPTION E_INVOICE_ALREADY_PAYED TO PROCEDURE SP_EDIT_INVOICE_LINE;
```

Deleting line items

The procedure for deleting an invoice line item from an invoice:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE SP_DELETE_INVOICE_LINE (
  INVOICE_LINE_ID INTEGER)
AS
  DECLARE invoice_id INT;
  DECLARE price D_MONEY;
  DECLARE quantity INT;
BEGIN
  IF (EXISTS(SELECT *
             FROM invoice_line
             JOIN invoice ON invoice.invoice_id = invoice_line.invoice_id
             WHERE invoice.paid = 1
               AND invoice_line.invoice_line_id = :invoice_line_id)) THEN
    EXCEPTION E_INVOICE_ALREADY_PAYED;
```

```
  DELETE FROM invoice_line
  WHERE invoice_line.invoice_line_id = :invoice_line_id
  RETURNING invoice_id, quantity, sale_price
  INTO invoice_id, quantity, price;

  -- Reduce the amount of the account.
  UPDATE invoice
  SET total_sale = total_sale - :quantity * :price
  WHERE invoice_id = :invoice_id;
END
^

SET TERM ;^

COMMENT ON PROCEDURE SP_DELETE_INVOICE_LINE IS
'Deleting invoice item';

COMMENT ON PARAMETER SP_DELETE_INVOICE_LINE.INVOICE_LINE_ID IS
'Code invoice item';
Privileges for Procedures
GRANT SELECT,DELETE ON INVOICE_LINE TO PROCEDURE SP_DELETE_INVOICE_LINE;
GRANT SELECT,UPDATE ON INVOICE TO PROCEDURE SP_DELETE_INVOICE_LINE;
-- only Firebird 3.0 and above
GRANT USAGE ON EXCEPTION E_INVOICE_ALREADY_PAYED TO PROCEDURE SP_DELETE_INVOICE_LINE;
```

Roles and Privileges for Users

Now we need to create roles and grant the corresponding privileges. We will create two roles: MANAGER and SUPERUSER. MANAGER will have a limited set of privileges while SUPERUSER will have access to practically everything in the database that is used by the project application.

```
CREATE ROLE MANAGER;
CREATE ROLE SUPERUSER;
The MANAGER role can read any table and use the corresponding procedures to manage invoices
GRANT SELECT ON CUSTOMER TO MANAGER;
GRANT SELECT ON INVOICE TO MANAGER;
GRANT SELECT ON INVOICE_LINE TO MANAGER;
GRANT SELECT ON PRODUCT TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_ADD_INVOICE TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_ADD_INVOICE_LINE TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_DELETE_INVOICE TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_DELETE_INVOICE_LINE TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_EDIT_INVOICE TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_EDIT_INVOICE_LINE TO MANAGER;
GRANT EXECUTE ON PROCEDURE SP_PAY_FOR_INVOICE TO MANAGER;
GRANT USAGE ON SEQUENCE GEN_INVOICE_ID TO MANAGER;
The SUPERUSER role can read any table, edit the primary tables directly and use the procedu
GRANT SELECT, INSERT, UPDATE, DELETE ON CUSTOMER TO SUPERUSER;
GRANT SELECT ON INVOICE TO SUPERUSER;
GRANT SELECT ON INVOICE_LINE TO SUPERUSER;
GRANT SELECT, INSERT, UPDATE, DELETE ON PRODUCT TO SUPERUSER;
GRANT EXECUTE ON PROCEDURE SP_ADD_INVOICE TO SUPERUSER;
GRANT EXECUTE ON PROCEDURE SP_ADD_INVOICE_LINE TO SUPERUSER;
GRANT EXECUTE ON PROCEDURE SP_DELETE_INVOICE TO SUPERUSER;
```

```
GRANT EXECUTE ON PROCEDURE SP_DELETE_INVOICE_LINE TO SUPERUSER;
GRANT EXECUTE ON PROCEDURE SP_EDIT_INVOICE TO SUPERUSER;
GRANT EXECUTE ON PROCEDURE SP_EDIT_INVOICE_LINE TO SUPERUSER;
GRANT EXECUTE ON PROCEDURE SP_PAY_FOR_INVICE TO SUPERUSER;
GRANT USAGE ON SEQUENCE GEN_CUSTOMER_ID TO SUPERUSER;
GRANT USAGE ON SEQUENCE GEN_INVOICE_ID TO SUPERUSER;
GRANT USAGE ON SEQUENCE GEN_PRODUCT_ID TO SUPERUSER;
```

These statements create some users and assign roles to them:

```
CREATE USER IVAN PASSWORD 'z12a';
CREATE USER ANNA PASSWORD 'lh67';

GRANT MANAGER TO ANNA;
GRANT MANAGER TO IVAN WITH ADMIN OPTION;
GRANT SUPERUSER TO IVAN;
```

The user IVAN can assign the MANAGER role to other users.

Saving and Running the Script

Save our script to a text file named `examples.sql`.

Now you have three choices: you can

- download the ready-made script files using the following links:

  https://github.com/sim1984/example-db_2_5/archive/1.0.zip
  or https://github.com/sim1984/example-db_3_0/archive/1.0.zip

- OR run the script examples.sql that you just created yourself;

- OR download the ready-made database, complete with sample data. Links are provided at the end of this chapter.

Now, to run our script in the database created earlier:

```
  isql -user sysdba -password masterkey "localhost:examples"
                                  -i "d:\examples-db\examples.sql"
```

> **Warning**
> Do not split this command!

The argument "localhost:examples" uses an alias in place of the file path. It assumes that an alias named 'examples' actually exists, of course! The `-i` switch is an abbreviation of `-input` and its argument should be the path to the script file you just saved.

Loading Test Data

Now that the database is created and built, you can populate it with test data. Various tools are available to help with that. If you prefer not to do it yourself, you can download a copy of the built database already loaded with the test data we used in the sample projects, from one of the following links:

- https://github.com/sim1984/example-db_2_5/releases/download/1.0/examples.fdb
- or https://github.com/sim1984/example-db_3_0/releases/download/1.0/examples.fdb

> **Reminder**
>
> A database built by Firebird 2.5 will not be accessible by a Firebird 3 server, nor vice versa. Make sure you download the correct database for your needs.

Chapter 3

Developing Firebird Applications in Delphi

This chapter will describe the process of developing applications for Firebird databases with the FireDac™ data access components in the Embarcadero Delphi™ XE5 environment. FireDac™ is a standard set of components for accessing various databases in Delphi XE3 and higher versions.

Starting a Project

Create a new project using *File—>New—>VCL Forms Application - Delphi*

Add a new data module using *File—>New—>Other* and selecting *Delphi Projects—>Delphi Files—>Data Module* in the wizard. This will be the main data module in our project. It will contain some instances of global access components that must be accessible to all forms that are intended to work with data. *TFDConnection* is an example of this kind of component.

TFDConnection Component

The *TFDConnection* component provides connectivity to various types of databases. We will specify an instance of this component in the *Connection* properties of other FireDac components. The particular type of the database to which the connection will be established depends on the value of the *DriverName* property. To access Firebird, you need to set this property to *FB*.

For the connection to know exactly which access library it should work with, place the *TFBPhysFBDriverLink* component in the main data module. Its *VendorLib* property enables the path to the client library to be specified precisely. If it is not specified, the component will attempt to establish a connection via libraries registered in the system, for example, in `system32`, which might not be what you want at all.

Path to the Client Library

We will place the necessary library in the `fbclient` folder located in the application folder and use the following code for the OnCreate event of the data module:

```
xAppPath := ExtractFileDir(Application.ExeName) + PathDelim;
FDPhysFBDriverLink.VendorLib := xAppPath + 'fbclient' + PathDelim + 'fbclient.dll';
```

> **Important notes about "bitness"**
>
> If you compile a 32-bit application, you should use the 32-bit fbclient.dll library. For a 64-bit application, it should be the 64-bit library.
>
> Along with the file fbclient.dll, it is advisable to place the following libraries in the same folder: `msvcp80.dll` and `msvcr80.dll` (for Firebird 2.5) as well as `msvcp100.dll` and `msvcr100.dll` (for Firebird 3.0). These libraries are located either in the `bin` subfolder (Firebird 2.5) or in the root folder of the server (Firebird 3.0).
>
> For the application to show internal firebird errors correctly, it is necessary to copy the file `firebird.msg` as well.
>
> - For Firebird 2.5 or earlier, the libraries must be one level up from the folder with the client library, i.e., in the application folder for our purposes.
> - For Firebird 3, they must be in the same folder as the client library, i.e. in the `fbclient` folder.

Developing for Embedded Applications

If you need your application to run without the installed Firebird server, i.e. in the Embedded mode, for Firebird 2.5 you should replace `fbclient.dll` with `fbembed.dll`. Make sure that the width of the CPU register (64-bit or 32-bit) matches the application. If necessary, the name of the library can be placed in the configuration file of your application.

It is not necessary to change anything for Firebird 3.0, in which the working mode depends on the connection string and the value of the *Providers* parameter in the file firebird.conf/databases.conf.

> **TIP**
>
> Even if your application is intended to work with Firebird in the Embedded mode, it is advisable to attach to the full server during development. The reason is that embedded Firebird runs in the same address space as the application and any application connecting to a database in embedded mode must be able to obtain exclusive access to that database. Once that connection succeeds, no other embedded connections are possible. When you are connected to your database in the Delphi IDE, the established connection is in Delphi's application space, thus preventing your application from being run successfully from the IDE.
>
> Note, Firebird 3 embedded still requires exclusive access if the installed full server is in Super (Superserver) mode.

Connection parameters

The *Params* property of the *TFDConnection* component contains the database connection parameters (username, password, connection character set, etc.). If you invoke the *TFDConnection* property editor by double-clicking on the component, you will see that those properties have been filled automatically. The property set depends on the database type.

Figure 3.1. TFDConnection property editor

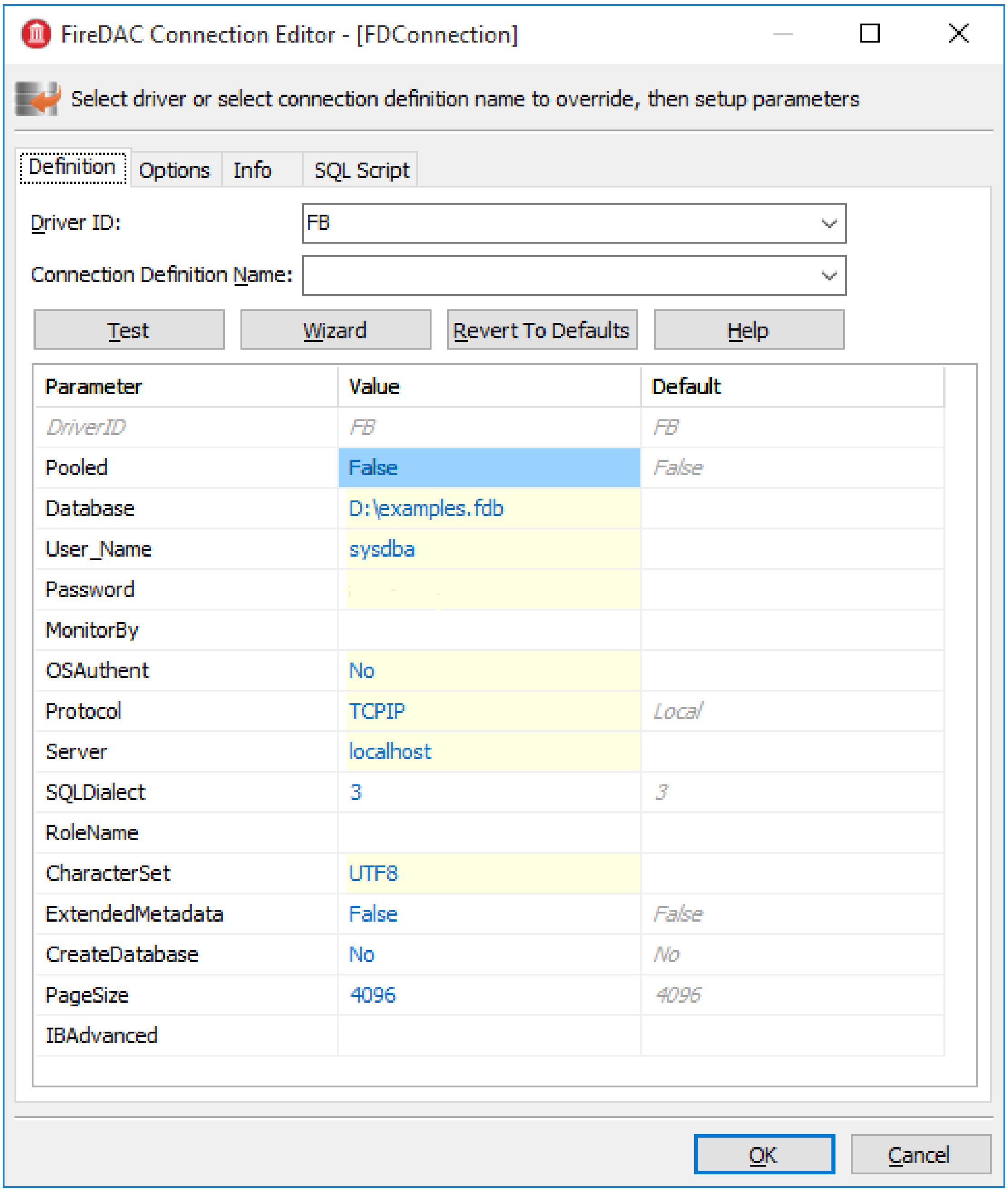


Table 3.1. TFDConnection component main properties

Property	Purpose
Pooled	Whether a connection pool is used
Database	The path to the database or its alias as defined in the aliases.conf configuration file (or in databases.conf) of the Firebird server
User_Name	Firebird user name. Not used if OSAuthent is True.
Password	Firebird password. Not used if OSAuthent is True.
OSAuthent	Whether operating system authentication is used

Property	Purpose
Protocol	Connection protocol. Possible values: • Local—local protocol • NetBEUI—named pipes, WNET • SPX—This property is for Novell's IPX/SPX protocol, which has never been supported in Firebird • TCPIP—TCP/IP
Server	Server name or its IP address. If the server is run on a non-standard port, you also need to append the port number after a slash, e.g., localhost/3051
SQLDialect	SQL Dialect. It must match that of the database
RoleName	Role name, if required
CharacterSet	Connection character set name
Additional Properties:	
Connected	Used to manage the database connection or check the connection status. This property must be set to True in order for the wizards of other FireDac components to work. If your application needs to request authentication data, it is important to remember to reset this property to False before compiling your application.
LoginPrompt	Whether to request the username and password during a connection attempt
Transaction	The TFDTransaction component that will be used as default to conduct various TFDConnection transactions. If this property is not explicitly specified, TFDConnection will create its own TFDTransaction instance. Its parameters can be configured in the TxOptions property.
UpdateTransaction	The TFDTransaction component that is to be used as default for the UpdateTransaction property of TFDQuery components, unless explicitly specified for the dataset. If this property is not specified explicitly, the value from the Transaction property of the connection will be used, unless it is explicitly specified for the dataset.

Connection Parameters in a Configuration File

Since the connection parameters, except for the username and password and possibly the role, are usually common to all instances the application, we will read them from the configuration file:

```
xIniFile := TIniFile.Create(xAppPath + 'config.ini');
try
  xIniFile.ReadSectionValues('connection', FDConnection.Params);
finally
  xIniFile.Free;
end;
```

A Typical Configuration File

Typically, the `config.ini` file contains the following lines:

```
[connection]
DriverID=FB
Protocol=TCPIP
Server=localhost/3051
Database=examples
OSAuthent=No
RoleName=
CharacterSet=UTF8
```

You can get the contents of the connection section by copying the contents of the *Params* property of the *TFD-Connection* component after the wizard finishes its work.

> **Note**
>
> Actually, the common settings are usually located in `%AppData%\Manufacturer\AppName` and are saved to that location by the application installation software. However, it is convenient for the configuration file to be stored somewhere closer during the development, for instance, in the application folder.
>
> Note that if your application is installed into the Program Files folder and the configuration file is located there as well, it is likely that the file will be virtualized in Program Data and issues could arise with modifying it and reading the new settings subsequently.

Connecting to the database

To connect to the database, it is necessary to change the *Connected* property of the *TFDConnection* component to True or call the *Open* method. You can use the *Open* method to pass the username and password as parameters.

A Little Modification

We will replace the standard database connection dialog box in our application and allow users to make three mistakes while entering the authentication information. After three failures, the application will be closed.

To implement it, we will write the following code in the *OnCreate* event handler of the main data module.

```
// After three unsuccessful login attempts, we close the application.
xLoginCount := 0;
xLoginPromptDlg := TLoginPromptForm.Create(Self);
while (xLoginCount < MAX_LOGIN_COUNT) and
      (not FDConnection.Connected) do
begin
```

```
  try
    if xLoginPromptDlg.ShowModal = mrOK then
      FDConnection.Open(
        xLoginPromptDlg.UserName, xLoginPromptDlg.Password)
    else
      xLoginCount := MAX_LOGIN_COUNT;
  except
    on E: Exception do
    begin
      Inc(xLoginCount);
      Application.ShowException(E);
    end
  end;
end;
xLoginPromptDlg.Free;
if not FDConnection.Connected then
  Halt;
```

Working with Transactions

The Firebird client allows any operations to be made only in the context of a transaction so, if you manage to access data without explicitly calling *TFDTransaction.StartTransaction*, it means that it was called automatically somewhere deep in FireDac. It is highly recommended to avoid this practice. For applications to work correctly with databases, it is advisable to manage transactions manually, which means starting and committing them or rolling them back with explicit calls.

The *TFDTransaction* component is used to manage transactions explicitly.

TFDTransaction Component

TFDTransaction has three methods for managing a transaction explicitly: *StartTransaction*, *Commit* and *Rollback*. The following table summarises the properties available to configure this component.

Table 3.2. TFDTransaction component main properties

Property	Purpose
Connection	Reference to the FDConnection component
Options.AutoCommit	Controls the automatic start and end of a transaction, emulating Firebird's own transaction management. The default value is True. See note (1) below for more details about behaviour if the Autocommit option is True.
Options.AutoStart	Controls the automatic start of a transaction. The default value is True.
Options.AutoStop	Controls the automatic end of a transaction. The default value is True.
Options.DisconnectAction	

Property	Purpose
	The action that will be performed when the connection is closed while the transaction is active. The default value is xdCommit—the transaction will be committed. See note (2) below for details of the other options.
Options.EnableNested	Controls nested transactions. The default value is True. Firebird does not support nested transactions as such but FireDac can emulate them using savepoints. For more details, see note(3) below.
Options.Isolation	Specifies the transaction isolation level. It is the most important transaction property. The default value is xiReadCommitted. The other values that Firebird supports are xiSnapshot and xiUnspecified; also xiSerializable, to some degree. For more details about the available isolation levels, see note (4) below.
Options.Params	Firebird-specific transaction attributes that can be applied to refine the transaction parameters, overriding attributes applied by the standard implementation of the selected isolation level. For the attributes that can be set and the "legal" combinations, see note (5) below.
Options.ReadOnly	Indicates whether it is a read-only transaction. The default value is False. Setting it to True disables any write activity. Long-running read-only transactions in READ COMMITTED isolation are recommended for activities that do not change anything in the database because they use fewer resources and do not interfere with garbage collection.

Note 1: AutoCommit=True

If the value of *AutoCommit* is set to True, FireDAC behaves as follows:

- Starts a transaction (if required) before each SQL command and ends the transaction after the SQL command completes execution
- If the command is successfully executed, the transaction will be ended by COMMIT. Otherwise, it will be ended by ROLLBACK.
- If the application calls the StartTransaction method, automatic transaction management will be disabled until that transaction is ended by Commit or Rollback.

Note 2: DisconnectAction

The following values are possible:

- xdNone—nothing will be done. The DBMS will perform its default action.
- xdCommit—the transaction will be committed
- xdRollback—the transaction will be rolled back

Note that, in some other data access components, the default value for the *DisconnectAction* property is *xdRollback* and will need to be set manually with Firebird to match the *FDTransaction* setting.

Note 3: EnableNested

If *StartTransaction* is called from within an active transaction, FireDac will emulate a nested transaction by creating a savepoint. Unless you are very confident in the effect of enabling nested transactions, set *EnableNested* to False. With this setting, calling *StartTransaction* inside the transaction will raise an exception.

Note 4: Isolation

FireBird has three isolation levels: READ COMMITTED, SNAPSHOT (“concurrency”) and SNAPSHOT TABLE STABILITY (“consistency”, rarely used). FireDac supports some but not all configurations for READ COMMITTED and SNAPSHOT. It uses the third level partially to emulate the SERIALIZABLE isolation that Firebird does not support.

- xiReadCommitted—the READ COMMITTED isolation level. FireDac starts ReadCommitted transactions in Firebird with the following parameters: read/write, rec_version, nowait
- xiSnapshot—the SNAPSHOT (concurrency) isolation level. FireDac starts Snapshot transactions in Firebird with the following parameters: read/write, wait
- xiUnspecified—Firebird's default isolation level (SNAPSHOT) with the following parameters: read/write, wait
- xiSerializable—the SERIALIZABLE isolation level. Firebird does not support serializable isolation, but FireDac emulates it by starting a SNAPSHOT TABLE STABILITY (“consistency”) transaction with the following parameters: read/write, wait.

Other parameters, not supported by Firebird at all, are:

- xiDirtyRead—if this is selected (not a good idea!) READ COMMITTED will be used instead
- xiRepeatableRead—if this is selected, SNAPSHOT will be used instead

Note 5: Firebird-specific Transaction Attributes

Attributes that can be customised in Options.Params are:

- *read write*, the default read mode for all of the options.isolation selections—see note (4) above. Set *write* off if you want read-only mode. Alternatively, you can set *Options.ReadOnly* to True to achieve the same thing. There is no such thing as a “write-only” transaction.
- *read_committed*, *concurrency* and *consistency* are isolation levels.
- *wait* and *nowait* are conflict resolution settings, determining whether the transaction is to wait for a conflict to resolve
- *rec_version* and *no rec_version* provide an option that is applicable only to READ COMMITTED transactions. The default *rec_version* lets this transaction read the latest committed version of a record and overwrite it if the transaction ID of the latest committed version is newer (higher) than the ID of this transaction. The *no rec_version* setting will block this transaction from reading the latest committed version if an update is pending from any other transaction.

Multiple Transactions

Unlike many other DBMSs, Firebird allows as many *TFDTransaction* objects as you need to associate with the same connection. In our application, we will use one common read transaction for all primary and secondary modules and one read/write transaction for each dataset.

We do not want to rely on starting and ending transactions automatically: we want to have full control. That is why *Options.AutoCommit=False*, *Options.AutoStart=False* and *Options.AutoStop=False* are set in all of our transactions.

Datasets

The components *TFDQuery*, *TFDTable*, *TFDStoredProc* and *TFDCommand* are the components for working with data in FireDac. *TFDCommand* does not deliver a dataset and, when TFDStoredProc is used with an executable stored procedure, rather than a selectable one, it does not deliver a dataset, either.

TFDQuery, *TFDTable* and *TFDStoredProc* are inherited from *TFDRdbmsDataSet*.

Apart from datasets for working with the database directly, FireDac also has the *TFDMemTable* component for working with in-memory datasets. It is functionally equivalent to *TClientDataSet*.

The main component for working with datasets, *TFDQuery*, can be used for practically any purpose. The *TFDTable* and *TFDStoredProc* components are just variants, expanded or reduced to meet differences in functionality. No more will be said about them and we will not be using them in our application. If you wish, you can learn about them in the FireDac documentation.

The purpose of a dataset component is to buffer records retrieved by the SELECT statement, commonly for displaying in a grid and providing for the current record in the buffer (grid) to be editable. Unlike the IBX *TIBDataSet* component, *TFDQuery* component does not have the properties *RefreshSQL*, *InsertSQL*, *UpdateSQL* and *DeleteSQL*. Instead, a separate *TFDUpdateSQL* object specifies the statement for dataset modifications and the dataset component carries a reference to that component in its *UpdateObject* property.

> **RequestLive Property**
>
> Sometimes it is possible to make an *FDQuery* object editable without referring, through the *UpdateObject* property, to an *FDUpdateSQL* object that specifies queries for insert, update and delete. The property *UpdateOptions.RequestLive* can be set to True for sets that are naturally updatable and the object will generate the modification queries for you. However, because this approach puts strict limitations on the SELECT query, it is not always useful to rely on it.

TFDQuery Component

Table 3.3. TFDQuery component main properties

Property	Purpose
Connection	Reference to the FDConnection object
MasterSource	If the dataset is to be used as detail to a master dataset, this property refers to the data source (TDataSource) of the master set
Transaction	If specified, refers to the transaction within which the query will be executed. If not specified, the default transaction for the connection will be used.
UpdateObject	Reference to the FDUpdateSQL object providing for the dataset to be editable when the SELECT query does not meet the requirements for automatic generation of modification queries with UpdateOptions.RequestLive=True.
UpdateTransaction	

Property	Purpose
	The transaction within which modification queries will be executed. If the property is not specified the transaction from the Transaction property of the connection will be used.
UpdateOptions.CheckRequired	If set to True (the default) FireDac controls the Required property of the corresponding NOT NULL fields. If you keep it True and a field with the Required=True has no value assigned to it, an exception will be raised when the Post method is called. This might not be what you want if a value is going to be assigned to this field later in BEFORE triggers.
UpdateOptions.EnableDelete	Specifies whether a record can be deleted from the dataset. If EnableDelete=False, an exception will be raised when the Delete method is called.
UpdateOptions.EnableInsert	Specifies whether a record can be inserted into the dataset. If EnableInsert=False, an exception will be raised when the Insert/Append method is called.
UpdateOptions.EnableInsert	Specifies whether a record can be inserted into the dataset. If EnableInsert=False, an exception will be raised when the Insert/Append method is called.
UpdateOptions.EnableUpdate	Specifies whether a record can be edited in the dataset. If EnableUpdate=False, an exception will be raised when the Edit method is called.
UpdateOptions.FetchGeneratorPoint	Controls the moment when the next value is fetched from the generator specified in the UpdateOptions.GeneratorName property or in the GeneratorName property of the auto-incremental field AutoGenerateValue=arAutoInc. The default is *gpDeferred*, causing the next value to be fetched from the generator before a new record is posted in the database, i.e., during Post or ApplyUpdates. For the full set of possible values, see note (1) below.
UpdateOptions.GeneratorName	The name of the generator from which the next value for an auto-incremental field is to be fetched.
UpdateOptions.ReadOnly	Specifies whether it is a read-only dataset. The default value is False. If the value of this property is set to True, the EnableDelete, EnableInsert and EnableUpdate properties will be automatically set to False.
UpdateOptions.RequestLive	Setting RequestLive to True makes a query editable, if possible. Queries for insert, update and delete will be generated automatically. This setting imposes strict limitations on the SELECT query. It is supported for backward compatibility with the ancient BDE and is not recommended.
UpdateOptions.UpdateMode	Controls how to check whether a record has been modified. This property allows control over possible overwriting of updates in cases where one user is taking a long time to edit a record while another user has been editing the same record simultaneously and

Property	Purpose
	completes the update earlier. The default is upWhereKeyOnly. For information about the available modes, see note (2) below.
CachedUpdates	Specifies whether the dataset cache defers changes in the dataset buffer. If this property is set to True, any changes (Insert/Post, Update/Post, Delete) are saved to a special log and the application must apply them explicitly by calling the ApplyUpdates method. All changes will be made within a small period of time and within one short transaction. The default value of this property is False.
SQL	Contains the text of the SQL query. If this property is a SELECT statement, execute it by calling the Open methold. Use the Execute or ExecSQL for executing a statement that does not return a dataset.

Note 1: UpdateOptions.FetchGeneratorPoint

The property UpdateOptions.FetchGeneratorPoint can take the following values:

- *gpNone*—no value is fetched from the generator
- *gpImmediate*—the next value is fetched from the generator right after the Insert/Append method is called
- *gpDeferred*—the next value is fetched during Post or ApplyUpdates

Note 2: UpdateOptions.UpdateMode

The user in a lengthy editing session could be unaware that a record has been updated one or more times during his editing session, perhaps causing his own changes to overwrite someone else's updates. The *UpdateOptions.UpdateMode* property allows a choice of behaviours to lessen or avoid this risk:

- *upWhereAll*—check whether a record exists by its primary key + check all columns for old values, e.g.,

  ```
  update table set ...
  where pkfield = :old_ pkfield and
            client_name =  :old_client_name and
           info = :old_info ...
  ```

  With *upWhereAll* set, the update query will change content in a record only if the record has not been edited by anyone else since our transaction started. It is especially important if there are dependencies between values in columns, such as minimum and maximum wages, etc.

- *upWhereChanged*—check whether a record exists by its primary key + check for old values only in the columns being edited.

  ```
  update table set ...
  where pkfield = :old_pkfield and
      client_name = :old_client
  ```

- *upWhereKeyOnly*—check whether a record exists by its primary key. This check corresponds to the automatically generated UpdateSQL query.

  To avoid (or handle) update conflicts in a multi-user environment, typically you need to add WHERE conditions manually. You would need a similar tactic, of course, to implement a process that emulates upWhereChanged, removing the unused column modifications from the update table set, leaving in the update list only the columns that are actually modified. The update query could otherwise overwrite someone else's updates of this record.

  Obviously, the UpdateSQL needs to be created dynamically.

  If you want to specify the settings for detecting update conflicts individually for each field, you can use the ProviderFlags property for each field.

TFDUpdateSQL component

The *TFDUpdateSQL* component enables you to refine or redefine the SQL command that Delphi generates automatically for updating a dataset. It can be used to update an *FDQuery* object, an *FDTable* object or data underlying an *FDStoredProc* object.

Using *TFDUpdateSQL* is optional for *TFDQuery* and *TFDTable* because these components can generate statements automatically, that can sometimes be used for posting updates from a dataset to the database. For updating a dataset that is delivered into an *FDStoredProc* object, use of the *TFDUpdateSQL* is not optional. The developer must figure out a statement that will result in the desired updates. If only one table is updated, a direct DML statement might be sufficient. Where multiple tables are affected, an executable stored procedure will be unavoidable.

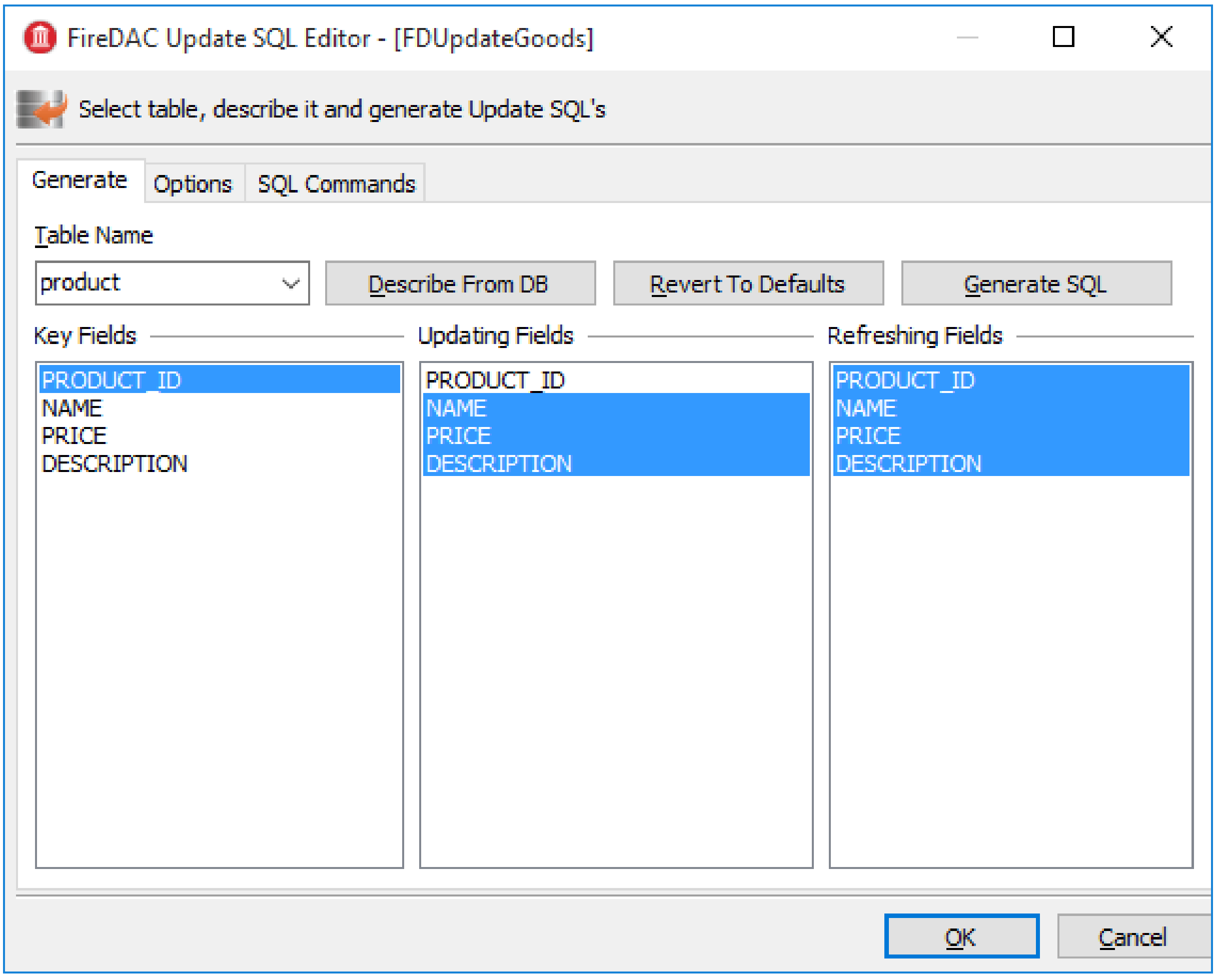
We recommend that you always use it, even in the simplest cases, to give yourself full control over the queries that are requested from your application.

TFDUpdateSQL Properties

To specify the SQL DML statements at design time, double-click on the *TFDUpdateSQL* component in your data module to open the property editor.

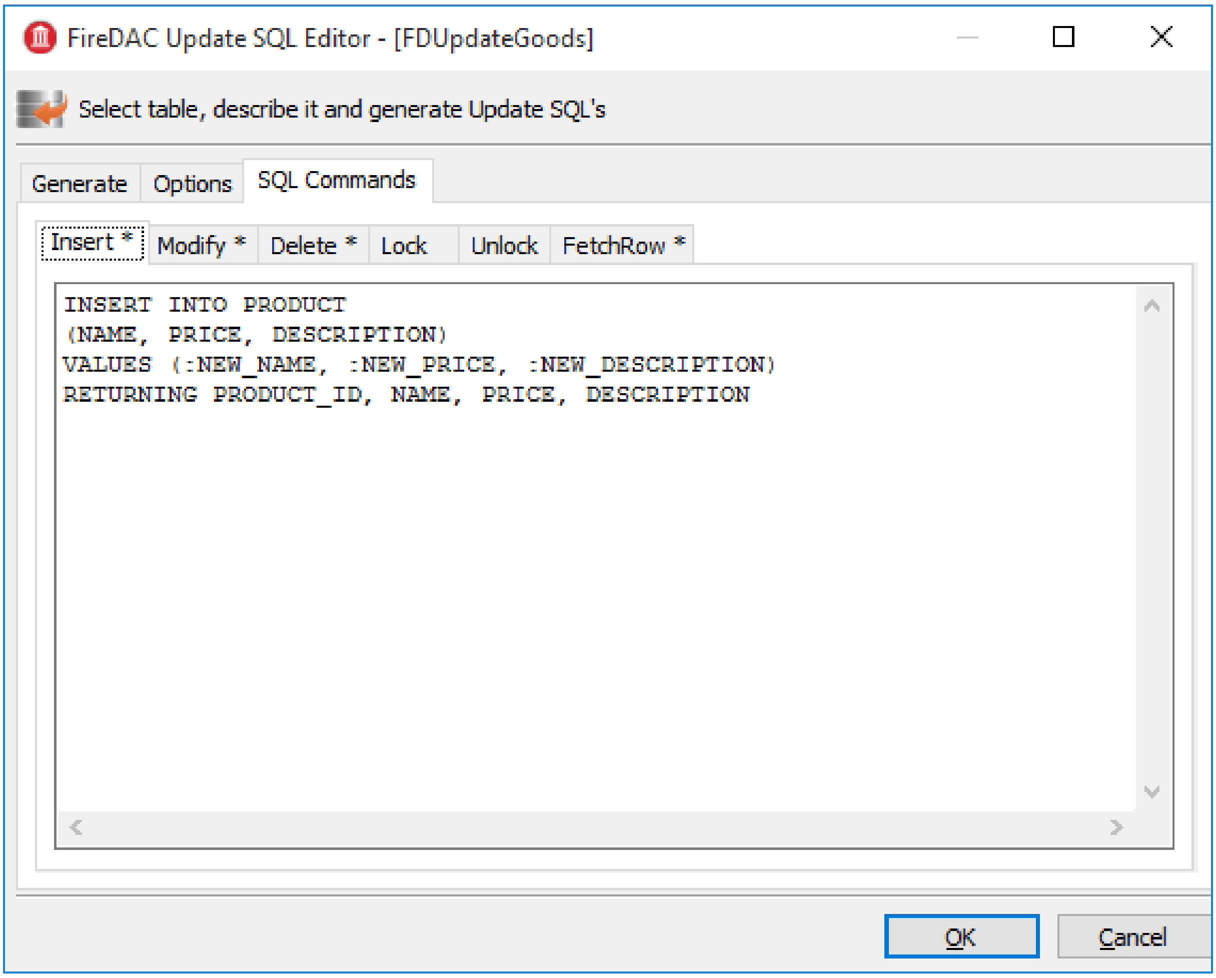
> **Important**
>
> Each component has its own design-time property editor. For multiple data-aware editors to run, FireDac needs an active connection to the database (*TFDConnection.Connected = True*) and a transaction in the autostart mode (TFDTransaction.Options.AutoStart = True) for each one.
>
> Design-time settings could interfere with the way the application is intended to work. For instance, the user is supposed to log in to the program using his username, but the FDConnection object connects to the database as SYSDBA.
>
> It is advisable to check the *Connected* property of the *FDConnection* object and reset it each time you use the data-aware editors. *AutoStart* will have to be enabled and disabled for a a read-only transaction as well.

Figure 3.2. TFDUpdateSQL property editor



You can use the Generate tab to make writing Insert/Update/Delete/Refresh queries easier for yourself. Select the table to be updated, its key fields, the fields to be updated and the fields that will be reread after the update and click the Generate SQL button to have Delphi generate the queries automatically. You will be switched to the SQL Commands tab where you can correct each query.

Figure 3.3. TFDUpdateSQL SQL command editor



> **Note**
>
> Since product_id is not included in *Updating Fields*, it is absent from the generated Insert query. It is assumed that this column is filled automatically by a generator call in a BEFORE INSERT trigger or, from Firebird 3.0 forward, it could be an IDENTITY column. When a value is fetched from the generator for this column at the server side, it is recommended to add the PRODUCT_ID column manually to the RETURNING clause of the INSERT statement.

The Options Tab

The Options tab contains some properties that can affect the process of query generation. These properties are not related to the *TFDUpdateSQL* component itself. Rather, for convenience, they are references to the *UpdateOptions* properties of the dataset that has the current *TFDUpdateSQL* specified in its *UpdateObject* property.

Table 3.4. TFDUpdateSQL component main properties

Property	Purpose
Connection	Reference to the TFDConnection component
DeleteSQL	The SQL query for deleting a record

Property	Purpose
FetchRowSQL	The SQL query for returning a current record after it has been updated or inserted—“RefreshSQL”
InsertSQL	The SQL query for inserting a record
LockSQL	The SQL query for locking a current record. (FOR UPDATE WITH LOCK)
ModifySQL	The SQL query for modifying a record
UnlockSQL	The SQL query for unlocking a current record. It is not used in Firebird.

Notice that, because the *TFDUpdateSQL* component does not execute modification queries directly, it has no *Transaction* property. It acts as a replacement for queries automatically generated in the parent *TFDRdbmsDataSet*.

TFDCommand component

The *TFDCommand* component is used to execute SQL queries. It is not descended from *TDataSet* so it is valid to use only for executing SQL queries that do not return datasets.

Table 3.5. TFDCommand component main properties

Property	Purpose
Connection	Reference to the TFDConnection component
Transaction	The transaction within which the SQL command will be executed
CommandKind	Type of command. The types are described in the section below.
CommandText	SQL query text

Types of Command

Usually, the command type is determined automatically from the text of the SQL statement. The following values are available for the property *TFDCommand.CommandKind* to cater for cases where the internal parser might be unable to make correct, unambiguous assumptions based on the statement text alone:

- skUnknown—unknown. Tells the internal parser to determine the command type automatically from its analysis of the text of the command
- skStartTransaction—a command for starting a transaction

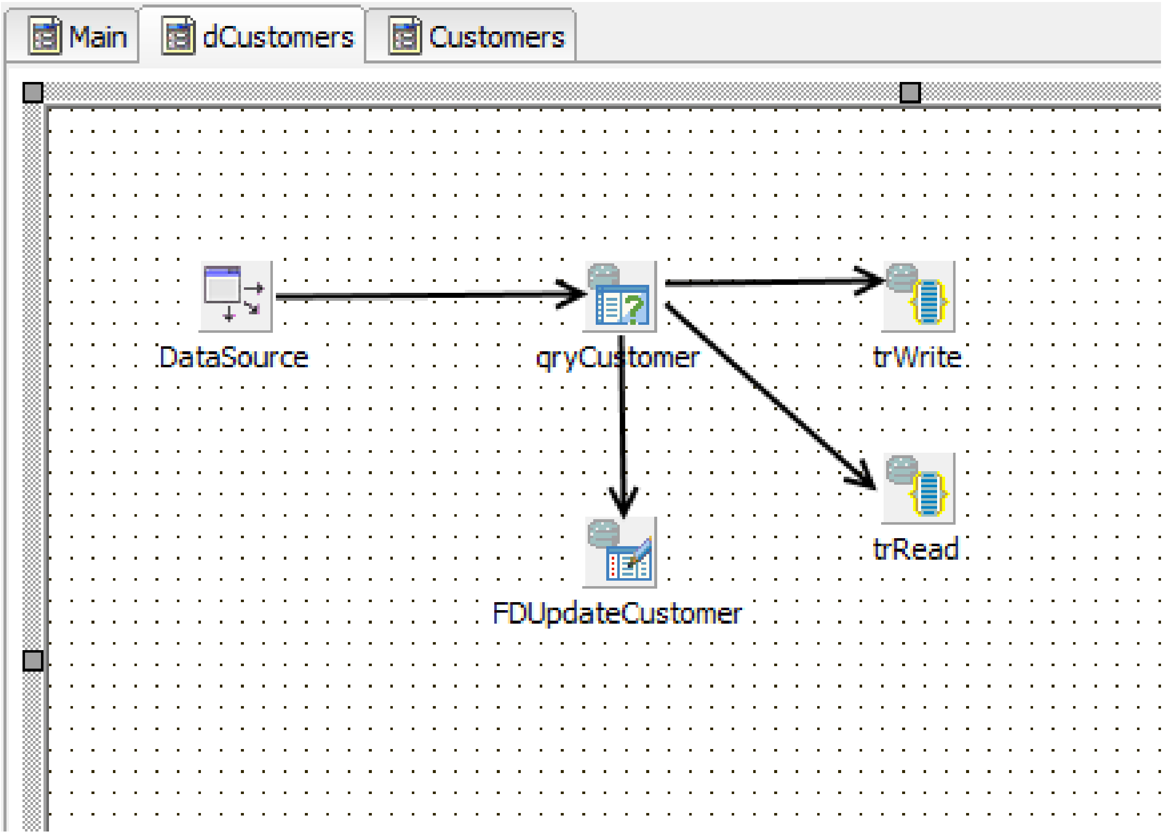
- skCommit—a command for ending and committing a transaction
- skRollback—a command for ending and rolling back a transaction
- skCreate—a CREATE … command for creating a new metadata object
- skAlter—an ALTER … command for altering a metadata object
- skDrop—a DROP … command for deleting a metadata object
- skSelect—a SELECT command for retrieving data
- skSelectForLock—a SELECT … WITH LOCK command for locking the selected rows
- skInsert—an INSERT … command for inserting a new record
- skUpdate—an UPDATE … command for modifying records
- skDelete—a DELETE … command for deleting records
- skMerge—a MERGE INTO … command
- skExecute—an EXECUTE PROCEDURE or or EXECUTE BLOCK command
- skStoredProc—a stored procedure call
- skStoredProcNoCrs—a call to a stored procedure that does not return a cursor
- skStoredProcWithCrs—a call to a stored procedure that returns a cursor

Creating the Primary Modules

We will create two primary modules in our application: a product module and a customer module. Each primary dataset is displayed on a form by means of a *TDBGrid* grid and a toolbar with buttons. The business logic of working with the dataset will be located in a separate DataModule that contains a *TDataSource* data source, a *TFDQuery* dataset, and two *TFDTransaction* transactions, one read-only and one read/write.

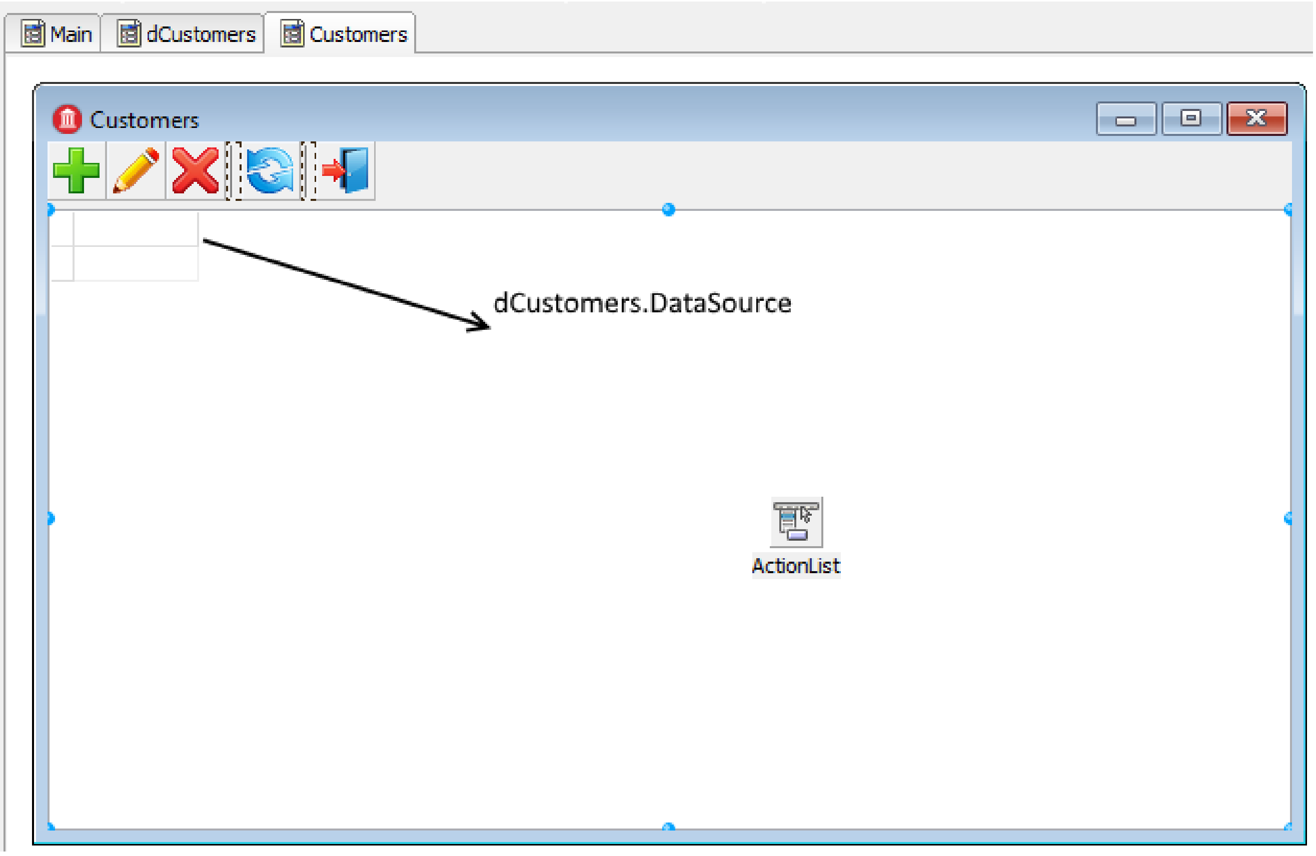
As our model for creating datasets, we will create the Customer dataset on the dCustomers datamodule:

Figure 3.4. dCustomers datamodule



On tabbing to the Customers form, this is the initial view. The DataSource component is not visible on the form because it is located in the dCustomers datamodule.

Figure 3.5. Customers form, initial view



We have placed the *TFDQuery* component in the dCustomers datamodule and named it qryCustomers. This dataset will be referred to in the *DataSet* property of the DataSource data source in DCustomers. We specify the read-only transaction trRead in the *Transaction* property, the trWritetransaction in the *UpdateTransaction* property and, for the *Connection* property, the connection located in the main data module. We populate the *SQL* property with the following query:

```
SELECT
  customer_id,
  name,
  address,
  zipcode,
  phone
FROM
  customer
ORDER BY name
```

The Read-only Transaction

The trRead read transaction is started when the dataset form is displayed (the *OnActivate* event) and is ended when the form is closed. READ COMMITTED isolation level (*Options.Isolation = xiReadCommitted*) is usually used to show data in grids because it allows the transaction to see changes committed in the database by other users by just repeating queries (rereading data) without the transaction being restarted.

Since this transaction is used only to read data, we set the *Options.ReadOnly* property to True. Thus, our transaction will have the following parameters: read read_committed rec_version.

> **Why?**
>
> A transaction with exactly these parameters can remain open in Firebird as long as necessary (days, weeks, months) without locking other transactions or affecting the accumulation of garbage in the database because, with these parameters, a transaction is started on the server as committed.

We set the property *Options.DisconnectAction* to *xdCommit*, which perfectly fits a read-only transaction. Finally, the read transaction will have the following properties:

```
Options.AutoStart = False
Options.AutoCommit = False
Options.AutoStop = False
Options.DisconnectAction = xdCommit
Options.Isolations = xiReadCommitted
Options.ReadOnly = True
```

> **Important**
>
> Although we do not discuss reporting in this manual, be aware that you should not use such a transaction for reports, especially if they use several queries in sequence. A transaction with READ COMMITTED isolation will see all new committed changes when rereading data. The recommended configuration for reports is a short read-only transaction with SNAPSHOT isolation (*Options.Isolation = xiSnapshot and Options.ReadOnly= True*).

The Read/Write Transaction

The write transaction trWrite that we use for our *FDUpdateSQL* object must be as short as possible to prevent the oldest active transaction from getting “stuck” and inhibiting garbage collection. High levels of uncollected garbage will lead to lower performance. Since the write transaction is very short, we can use the SNAPSHOT isolation level. The default value of the *Options.DisconnectAction* property, *xdCommit*, is not appropriate for write transactions, so it should be set to *xdRollback*. We will not rely on starting and ending transactions automatically. Instead, we will start and end a transaction explicitly. Thus, our transaction should have the following properties:

```
Options.AutoStart = False
Options.AutoCommit = False
Options.AutoStop = False
Options.DisconnectAction = xdRollback
Options.Isolations = xiSnapshot
Options.ReadOnly = False
```

SNAPSHOT vs READ COMMITTED Isolation

It is not absolutely necessary to specify SNAPSHOT isolation for simple INSERT/UPDATE/DELETE operations. However, if a table has complex triggers or a stored procedure is executed instead of a simple INSERT/UP-

DATE/DELETE query, it is advisable to use SNAPSHOT. The reason is that READ COMMITTED isolation does not ensure the read consistency of the statement within one transaction, since the SELECT statement in this isolation can return data that were committed to the database after the transaction began. In principle, SNAPSHOT isolation is recommended for short-running transactions.

Configuring the Customer Module for Editing

In this section, we will configure some properties in the *qryCustomer* and *FDUpdateCustomer* objects to make the Customer dataset editable.

The TFDUpdateSQL Settings

To make the dataset editable, the *InsertSQL*, *ModifySQL*, *DeleteSQL* and *FetchRowSQL* properties should be specified in the *FDUpdateSQL* object that is linked to the dataset. The wizard can generate these statements but it may be necessary to correct some things afterwards. For example, you can add a RETURNING clause, remove some columns from the update list or cancel an automatically generated stored procedure call entirely.

InsertSQL

```
INSERT INTO customer (
  customer_id,
  name,
  address,
  zipcode,
  phone)
VALUES (:new_customer_id,
  :new_name,
  :new_address,
  :new_zipcode,
  :new_phone)
```

ModifySQL

```
UPDATE customer
SET name = :new_name,
    address = :new_address,
    zipcode = :new_zipcode,
    phone = :new_phone
WHERE (customer_id = :old_customer_id)
```

DeleteSQL

```
DELETE FROM customer
```

```
WHERE (customer_id = :old_customer_id)
```

FetchRowSQL

```
SELECT
  customer_id,
  name,
  address,
  zipcode,
  phone
FROM
  customer
WHERE customer_id = :old_customer_id
```

Getting a Generator Value

In this project, we will get the value from the generator before making an insert into the table. To enable that, specify the following values for the properties of the *TFDQuery* component:

```
  UpdateOptions.GeneratorName = GEN_CUSTOMER_ID
     and
  UpdateOptions.AutoIncFields = CUSTOMER_ID
```

> **Note**
>
> This method works only for autoinc fields that are populated by explicit generators (sequences). It is not applicable to the IDENTITY type of autoinc key introduced in Firebird 3.0.

Another way to get the value from the generator is to return it after the INSERT is executed by means of a RETURNING clause. This method, which works for IDENTITY fields as well, will be shown later, in the topic Using a RETURNING Clause to Acquire an Autoinc Value.

Implementing the Customer Module

Modal forms are often used to add a new record or to edit an existing one. Once the modal form is closed by the mrOK result, the changes are posted to the database. Database-aware visual components are usually used to create this kind of form. These components enable you to display the values of some fields from the current record and immediately accept the user's changes in the corresponding fields if the dataset is in the Insert/Edit mode, i.e. before Post.

The only way to switch the dataset to Insert/Edit mode is by starting a write transaction. So, if somebody opens a form for adding a new record and leaves for a lunch break, we will have an active transaction hanging until the user comes back from lunch and closes the form. This uncommitted edit can inhibit garbage collection, which will reduce performance. There are two ways to solve this problem:

1. Use the *CachedUpdates* mode, which enables the transaction to be active just for a very short period (to be exact, just for the time it takes for the changes to be applied to the database).

2. Give up using visual components that are data-aware. This approach requires some additional effort from you to activate the data source and pass user input to it.

We will show how both methods are implemented. The first method is much more convenient to use. Let's examine the code for editing a customer record:

```
procedure TCustomerForm.actEditRecordExecute(Sender: TObject);
var
  xEditorForm: TEditCustomerForm;
begin
  xEditorForm := TEditCustomerForm.Create(Self);
  try
    xEditorForm.OnClose := CustomerEditorClose;
    xEditorForm.DataSource := Customers.DataSource;
    xEditorForm.Caption := 'Edit customer';
    Customers.Edit;
    xEditorForm.ShowModal;
  finally
    xEditorForm.Free;
  end;
end;
The Customers property is initiated in the OnCreate event:
procedure TCustomerForm.FormCreate(Sender: TObject);
begin
  FCustomers := TDMCustomers.Create(Self);
  DBGrid.DataSource := Customers.DataSource;
end;
```

We set the *CachedUpdates* mode for the dataset in the Edit method of the dCustomers module before switching it to the edit mode:

```
procedure TdmCustomers.Edit;
begin
  qryCustomer.CachedUpdates := True;
  qryCustomer.Edit;
end;
```

The logic of handling the process of editing and adding a record is implemented in the *OnClose* event handler for the modal edit form:

```
procedure TCustomerForm.CustomerEditorClose(Sender: TObject;
  var Action: TCloseAction);
begin
  if TEditCustomerForm(Sender).ModalResult <> mrOK then
  begin
    Customers.Cancel;
    Action := caFree;
    Exit;
  end;
  try
```

```
    Customers.Post;
    Customers.Save;
    Action := caFree;
  except
    on E: Exception do
    begin
      Application.ShowException(E);
      // It does not close the window give the user correct the error
      Action := caNone;
    end;
  end;
end;
```

To understand the internal processes, we can study the code for the *Cancel, Post* and *Save* methods of the dCustomer data module:

```
procedure TdmCustomers.Cancel;
begin
  qryCustomer.Cancel;
  qryCustomer.CancelUpdates;
  qryCustomer.CachedUpdates := False;
end;

procedure TdmCustomers.Post;
begin
  qryCustomer.Post;
end;

procedure TdmCustomers.Save;
begin
  // We do everything in a short transaction
  // In CachedUpdates mode an error does not interrupt the running code.
  // The ApplyUpdates method returns the number of errors.
  // The error can be obtained from the property RowError
  try
    trWrite.StartTransaction;
    if (qryCustomer.ApplyUpdates = 0) then
    begin
      qryCustomer.CommitUpdates;
      trWrite.Commit;
    end
    else
      raise Exception.Create(qryCustomer.RowError.Message);
    qryCustomer.CachedUpdates := False;
  except
    on E: Exception do
    begin
      if trWrite.Active then
        trWrite.Rollback;
      raise;
    end;
  end;
end;
```

Observe that the write transaction is not started at all until the OK button is clicked. Thus, the write transaction is active only while the data are being transferred from the dataset buffer to the database. Since we access not more than one record in the buffer, the transaction will be active for a very short time, which is exactly what we want.

Using a RETURNING Clause to Acquire an Autoinc Value

Creating the product is similar to creating the customer one. We will use it to demonstrate the method of getting an auto-incremented value by means of a RETURNING clause.

The main query:

```
SELECT
  product_id,
  name,
  price,
  description
FROM product
ORDER BY name
```

The *TFDUpdateSQL.InsertSQL* property will contain the following statement:

```
INSERT INTO PRODUCT (NAME, PRICE, DESCRIPTION)
VALUES (:NEW_NAME, :NEW_PRICE, :NEW_DESCRIPTION)
RETURNING PRODUCT_ID
```

The RETURNING clause in this statement will return the value of the PRODUCT_ID field after it has been populated by the BEFORE INSERT trigger. The client side in this case has no need to know the name of the generator, since it all happens on the server. Leave the *UpdateOptions.GeneratorName* property as nil.

To acquire the autoinc value by this method also requires filling a couple of properties for the PRODUCT_ID field because the value is being entered indirectly:

```
  Required = False
     and
  ReadOnly = True
```

Everything else is set up similarly to the way it was done for the Customer module.

Creating a Secondary Module

Secondary datasets typically contain larger numbers of records than primary datasets and new records are added frequently. Our application will have only one secondary module, named “Invoices”.

An invoice consists of a header where some general attributes are described (number, date, customer …) and invoice lines with the list of products, their quantities, prices, etc. It is convenient to have two grids for such documents: the main one (master) showing the data invoice header data and the detail one showing the invoice lines.

We want to place two *TDBGrid* components on the invoice form and link a separate *TDataSource* to each of them that will be linked to its respective *TFDQuery*. In our project, the dataset with the invoice headers (the master set) will be called qryInvoice, and the one with the invoice lines (the detail set) will be called qryInvoiceLine.

The Transactions for Invoice Data

The *Transaction* property of each dataset will specify the read-only transaction trRead that is located in the dmInvoicedata module. Use the *UpdateTransaction* property to specify the trWrite transaction and the *Connection* property to specify the connection located in the main data module.

A Filter for the Data

Secondary datasets usually contain a field with the record creation date. In order to reduce the amount of retrieved data, a notion such as “a work period” is commonly incorporated in the application to filter the set of data sent to the client. A work period is a range of dates for which the records are required.

Since the application could have more than one secondary dataset, it makes sense to add variables containing the start and end dates of a work period to the global dmMain data module that is used by all modules working with the database in one way or another. Once the application is started, the work period could be defined by the start and end dates of the current quarter, or some other appropriate start/end date pair. The application could allow the user to change the work period while working with the application.

Configuring the Module

Figure 3.6. The Invoice form tab

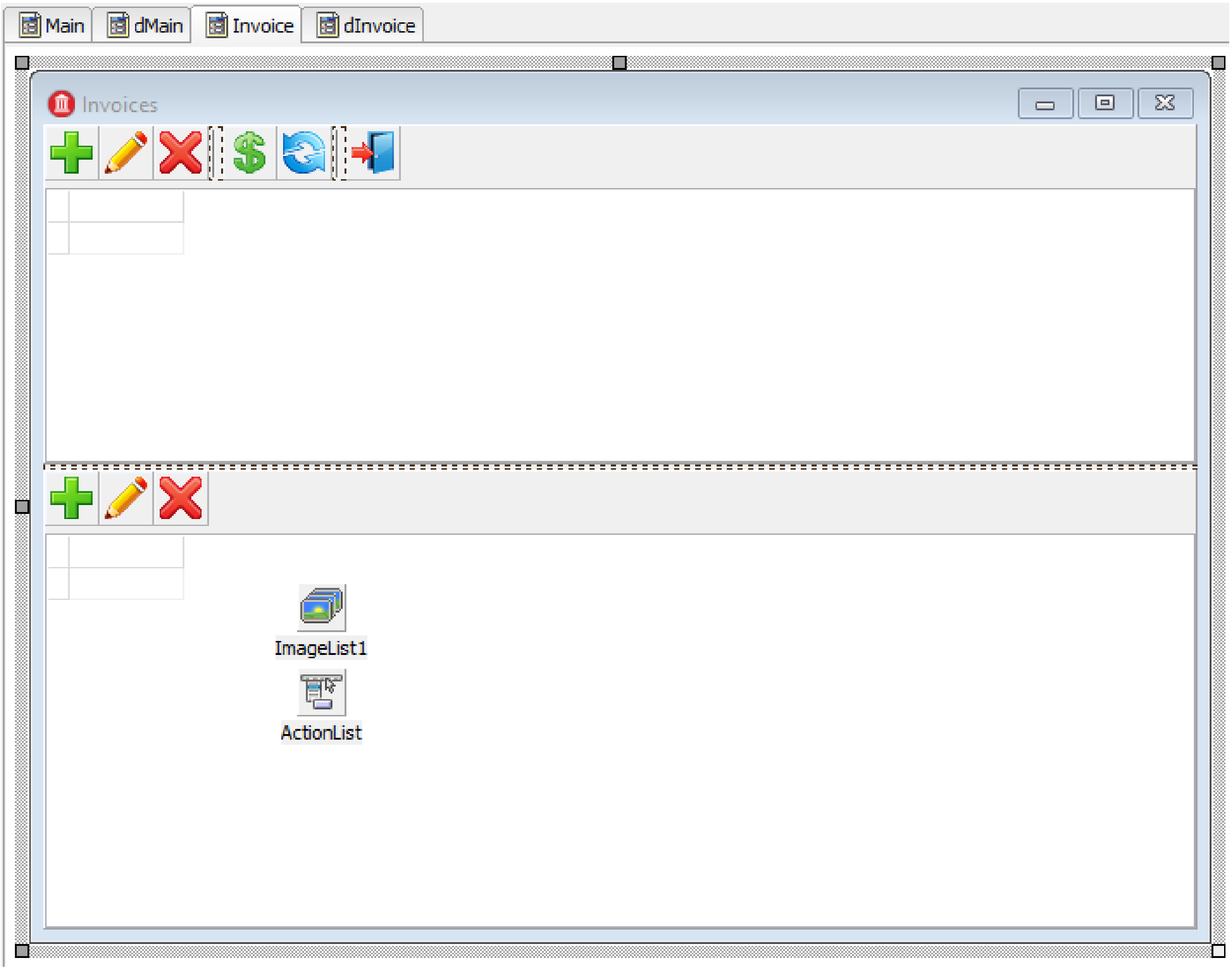
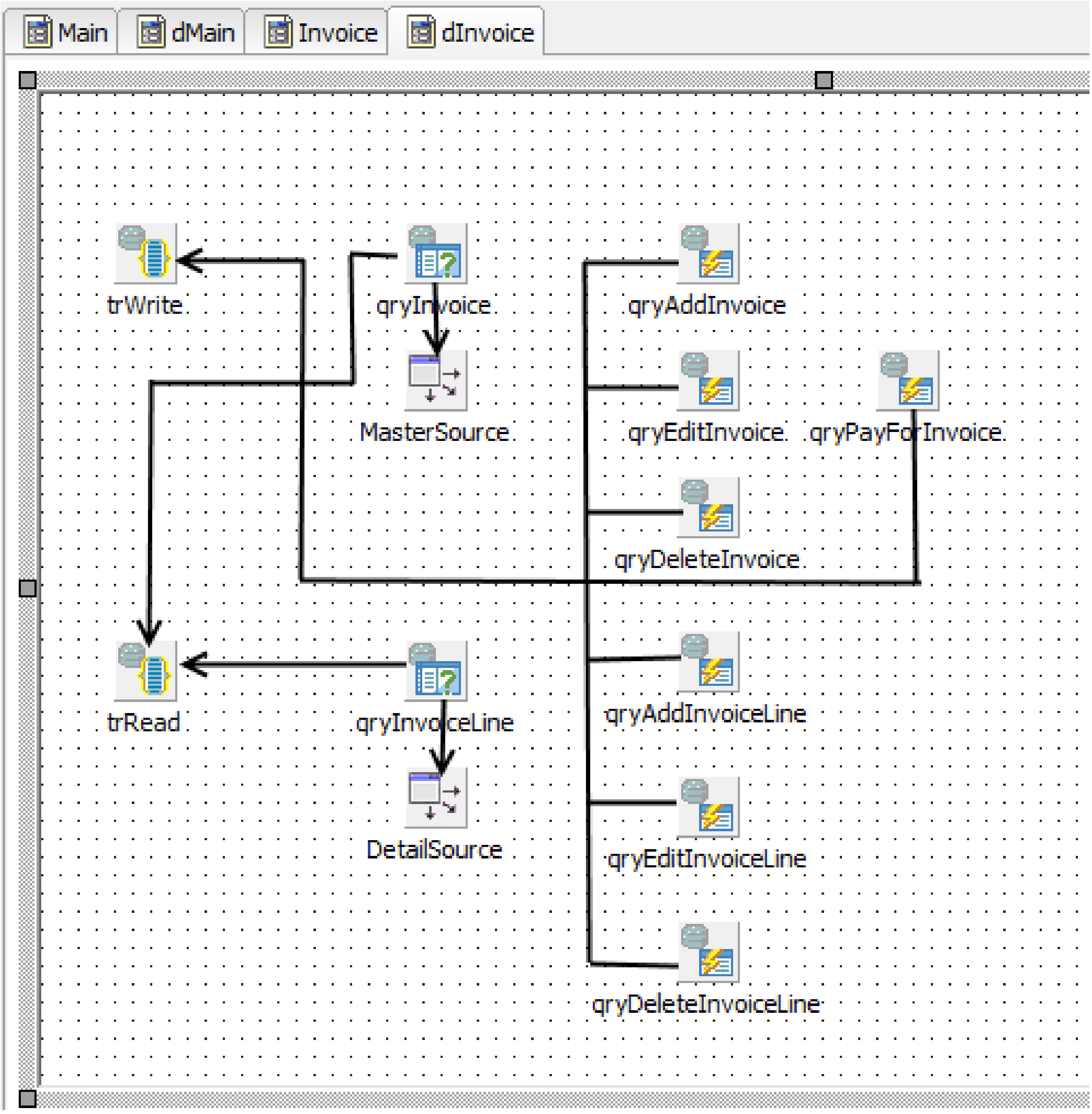


Figure 3.7. The Invoice data module tab



Since the latest invoices are the most requested ones, it makes sense to sort them by date in reverse order. The query will look like this in the SQL property of the qryInvoice dataset:

```
SELECT
  invoice.invoice_id AS invoice_id,
  invoice.customer_id AS customer_id,
  customer.NAME AS customer_name,
  invoice.invoice_date AS invoice_date,
  invoice.total_sale AS total_sale,
  IIF(invoice.payed=1, 'Yes', 'No') AS payed
FROM
  invoice
  JOIN customer ON customer.customer_id = invoice.customer_id
```

```
WHERE invoice.invoice_date BETWEEN :date_begin AND :date_end
ORDER BY invoice.invoice_date DESC
```

To open this dataset, it will be necessary to initialise the query parameters:

```
qryInvoice.ParamByName('date_begin').AsSqlTimeStamp := dmMain.BeginDateSt;
qryInvoice.ParamByName('date_end').AsSqlTimeStamp := dmMain.EndDateSt;
qryInvoice.Open;
```

For the purpose of illustration, we will use stored procedures to perform all operations on an invoice. Regular INSERT/UPDATE/DELETE queries can be used when operations are simple and involve writing to only one table in the database. We will execute each stored procedure as a separate query in TFDCommand objects. This component is not descended from *TFDRdbmsDataSet*, does not buffer data and returns not more than one result row. We are using it because it consumes fewer resources for queries that do not return data.

Since our stored procedures modify data, it is necessary to point the *Transaction* property of each TFDCommand object to the trWrite transaction.

> **Tip**
>
> Another alternative is to place the stored procedure calls for inserting, editing and adding a record in the corresponding properties of a *TFDUpdateSQL* object.

Doing the Work

Four operations are provided for working with the invoice header: adding, editing, deleting and setting the “paid” attribute. Once an invoice is paid, we prevent any modifications to either the header or the lines. The rule is implemented at stored procedure level. Let's examine the query strings in the *CommandText* property for calling the stored procedures.

qryAddInvoice.CommandText

```
EXECUTE PROCEDURE sp_add_invoice(
  NEXT VALUE FOR gen_invoice_id,
  :CUSTOMER_ID,
  :INVOICE_DATE
)
```

qryEditInvoice.CommandText

```
EXECUTE PROCEDURE sp_edit_invoice(
  :INVOICE_ID,
  :CUSTOMER_ID,
  :INVOICE_DATE
)
```

qryDeleteInvoice.CommandText

```
EXECUTE PROCEDURE sp_delete_invoice(:INVOICE_ID)
```

qryPayForInvoice.CommandText

```
EXECUTE PROCEDURE sp_pay_for_invoice(:invoice_id)
```

Since our stored procedures are not called from a *TFDUpdateSQL* object, we need to call *qryInvoice.Refresh* after they are executed, in order to update the data in the grid.

Stored procedures that do not require input data from the user are called as follows:

```
procedure TdmInvoice.DeleteInvoice;
begin
  // We do everything in a short transaction
  trWrite.StartTransaction;
  try
    qryDeleteInvoice.ParamByName('INVOICE_ID').AsInteger :=
      Invoice.INVOICE_ID.Value;
    qryDeleteInvoice.Execute;
    trWrite.Commit;
    qryInvoice.Refresh;
  except
    on E: Exception do
    begin
      if trWrite.Active then
        trWrite.Rollback;
      raise;
    end;
  end;
end;
```

Getting User Confirmation

Before performing some operations, such as deleting an invoice, we want to get confirmation from the user:

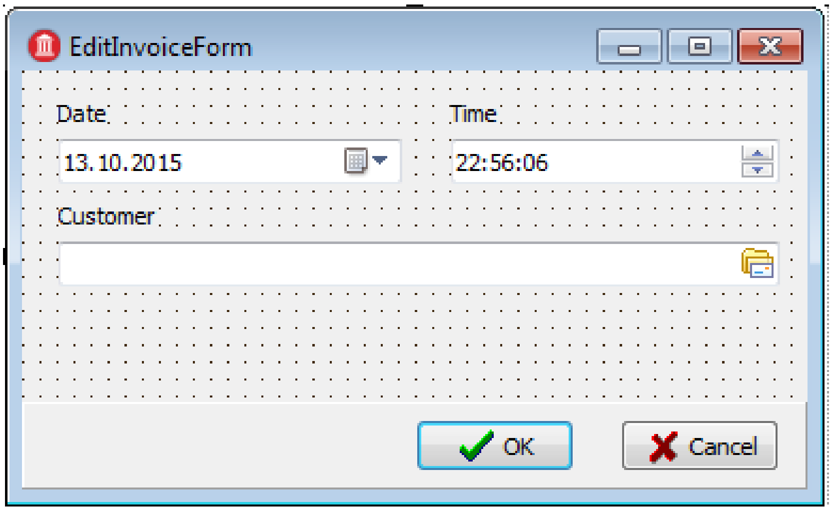
```
procedure TInvoiceForm.actDeleteInvoiceExecute(Sender: TObject);
begin
  if MessageDlg('Are you sure you want to delete an invoice?',
                mtConfirmation,
                [mbYes, mbNo], 0) = mrYes then
  begin
    Invoices.DeleteInvoice;
```

```
  end;
end;
```

Adding or Editing Records

As with the primary modules, we will use modal forms to add a new record or edit an existing one. We will not use data-aware visual components in this implementation. As another variation, we will use a *TButtonedEdit* component to select a customer. It will display the name of the current customer and open a modal form with a grid for selecting a customer on the click of the embedded button. We could use something like *TDBLookup-Combobox*, of course, but it has drawbacks: first, the customer list may be too large for scrolling comfortably through the drop-down list; secondly, the name alone may not be enough to find the customer you want.

Figure 3.8. The Customer input form



As the window for selecting a customer, we will use the same modal form that was created for adding customers. The code for the button click handler for the *TButtonedEdit* component is as follows:

```
procedure TEditInvoiceForm.edtCustomerRightButtonClick(Sender: TObject);
var
  xSelectForm: TCustomerForm;
begin
  xSelectForm := TCustomerForm.Create(Self);
  try
    xSelectForm.Visible := False;
    if xSelectForm.ShowModal = mrOK then
    begin
      FCustomerId := xSelectForm.Customers.Customer.CUSTOMER_ID.Value;
      edtCustomer.Text := xSelectForm.Customers.Customer.NAME.Value;
    end;
  finally
    xSelectForm.Free;
  end;
end;
```

Since we are not using data-aware visual components, we need to initialize the customer code and name for displaying during the call to the edit form:

```
procedure TInvoiceForm.actEditInvoiceExecute(Sender: TObject);
var
  xEditorForm: TEditInvoiceForm;
begin
  xEditorForm := TEditInvoiceForm.Create(Self);
  try
    xEditorForm.OnClose := EditInvoiceEditorClose;
    xEditorForm.Caption := 'Edit invoice';
    xEditorForm.InvoiceId := Invoices.Invoice.INVOICE_ID.Value;
    xEditorForm.SetCustomer(
      Invoices.Invoice.CUSTOMER_ID.Value,
      Invoices.Invoice.CUSTOMER_NAME.Value);
    xEditorForm.InvoiceDate := Invoices.Invoice.INVOICE_DATE.AsDateTime;
    xEditorForm.ShowModal;
  finally
    xEditorForm.Free;
  end;
end;

procedure TEditInvoiceForm.SetCustomer(ACustomerId: Integer;
  const ACustomerName: string);
begin
  FCustomerId := ACustomerId;
  edtCustomer.Text := ACustomerName;
end;
```

Adding a new invoice and editing an existing one will be handled in the *Close* event of the modal form as it is for the primary modules. However, we will not switch the dataset to *CachedUpdates* mode for these because the updates carried out by stored procedures and we are not using data-aware visual components to capture input.

```
procedure TInvoiceForm.actAddInvoiceExecute(Sender: TObject);
var
  xEditorForm: TEditInvoiceForm;
begin
  xEditorForm := TEditInvoiceForm.Create(Self);
  try
    xEditorForm.Caption := 'Add invoice';
    xEditorForm.OnClose := AddInvoiceEditorClose;
    xEditorForm.InvoiceDate := Now;
    xEditorForm.ShowModal;
  finally
    xEditorForm.Free;
  end;
end;

procedure TInvoiceForm.AddInvoiceEditorClose(Sender: TObject;
  var Action: TCloseAction);
var
  xEditorForm: TEditInvoiceForm;
begin
  xEditorForm := TEditInvoiceForm(Sender);
  if xEditorForm.ModalResult <> mrOK then
  begin
    Action := caFree;
    Exit;
```

```
  end;
  try
    Invoices.AddInvoice(xEditorForm.CustomerId, xEditorForm.InvoiceDate);
    Action := caFree;
  except
    on E: Exception do
    begin
      Application.ShowException(E);
// It does not close the window give the user correct the error
      Action := caNone;
    end;
  end;
end;

procedure TdmInvoice.AddInvoice(ACustomerId: Integer; AInvoiceDate: TDateTime);
begin
  // We do everything in a short transaction
  trWrite.StartTransaction;
  try
    qryAddInvoice.ParamByName('CUSTOMER_ID').AsInteger := ACustomerId;
    qryAddInvoice.ParamByName('INVOICE_DATE').AsSqlTimeStamp :=
    DateTimeToSQLTimeStamp(AInvoiceDate);
    qryAddInvoice.Execute();
    trWrite.Commit;
    qryInvoice.Refresh;
  except
    on E: Exception do
    begin
      if trWrite.Active then
        trWrite.Rollback;
      raise;
    end;
  end;
end;
```

The Invoice Details

Next, we move on to the details of an invoice. For the qryInvoiceLine dataset, we set the *MasterSource* property to the datasource that is linked to qryInvoice and the *MasterFields* property to INVOICE_ID. We specify the following query in the *SQL* property:

```
SELECT
  invoice_line.invoice_line_id AS invoice_line_id,
  invoice_line.invoice_id AS invoice_id,
  invoice_line.product_id AS product_id,
  product.name AS productname,
  invoice_line.quantity AS quantity,
  invoice_line.sale_price AS sale_price,
  invoice_line.quantity * invoice_line.sale_price AS total
FROM
  invoice_line
  JOIN product ON product.product_id = invoice_line.product_id
WHERE invoice_line.invoice_id = :invoice_id
```

As with the invoice header, we will use stored procedures to perform all modifications. Let's examine the query strings in the *CommandText* property of the commands that call the stored procedures.

qryAddInvoiceLine.CommandText

```
EXECUTE PROCEDURE sp_add_invoice_line(
  :invoice_id,
  :product_id,
  :quantity
)
```

qryEditInvoiceLine.CommandText

```
EXECUTE PROCEDURE sp_edit_invoice_line(
  :invoice_line_id,
  :quantity
)
```

qryDeleteInvoiceLine.CommandText

```
EXECUTE PROCEDURE sp_delete_invoice_line(
  :invoice_line_id
)
```

As with the header, the form for adding a new record and editing an existing one does not use data-aware visual components. To select a product, we use the *TButtonedEdit* component again. The code for the on-click handler for the button on the *TButtonedEdit* object is as follows:

```
procedure TEditInvoiceLineForm.edtProductRightButtonClick(Sender: TObject);
var
  xSelectForm: TGoodsForm;
begin
  if FEditMode = emInvoiceLineEdit then
    Exit;
  xSelectForm := TGoodsForm.Create(Self);
  try
    xSelectForm.Visible := False;
    if xSelectForm.ShowModal = mrOK then
    begin
      FProductId := xSelectForm.Goods.Product.PRODUCT_ID.Value;
      edtProduct.Text := xSelectForm.Goods.Product.NAME.Value;
      edtPrice.Text := xSelectForm.Goods.Product.PRICE.AsString;
    end;
  finally
    xSelectForm.Free;
  end;
end;
```

Since we are not using data-aware visual components, again we will need to initialize the product code and name and its price for displaying on the edit form.

```
procedure TInvoiceForm.actEditInvoiceLineExecute(Sender: TObject);
var
  xEditorForm: TEditInvoiceLineForm;
begin
  xEditorForm := TEditInvoiceLineForm.Create(Self);
  try
    xEditorForm.EditMode := emInvoiceLineEdit;
    xEditorForm.OnClose := EditInvoiceLineEditorClose;
    xEditorForm.Caption := 'Edit invoice line';
    xEditorForm.InvoiceLineId := Invoices.InvoiceLine.INVOICE_LINE_ID.Value;
    xEditorForm.SetProduct(
      Invoices.InvoiceLine.PRODUCT_ID.Value,
      Invoices.InvoiceLine.PRODUCTNAME.Value,
      Invoices.InvoiceLine.SALE_PRICE.AsCurrency);
    xEditorForm.Quantity := Invoices.InvoiceLine.QUANTITY.Value;
    xEditorForm.ShowModal;
  finally
    xEditorForm.Free;
  end;
end;

procedure TEditInvoiceLineForm.SetProduct(AProductId: Integer;
  AProductName: string; APrice: Currency);
begin
  FProductId := AProductId;
  edtProduct.Text := AProductName;
  edtPrice.Text := CurrToStr(APrice);
end;
```

We handle adding a new item and editing an existing one in the Close event of the modal form.

```
procedure TInvoiceForm.actAddInvoiceLineExecute(Sender: TObject);
var
  xEditorForm: TEditInvoiceLineForm;
begin
  xEditorForm := TEditInvoiceLineForm.Create(Self);
  try
xEditorForm.EditMode := emInvoiceLineAdd;
xEditorForm.OnClose := AddInvoiceLineEditorClose;
    xEditorForm.Caption := 'Add invoice line';
    xEditorForm.Quantity := 1;
    xEditorForm.InvoiceId := Invoices.Invoice.INVOICE_ID.Value;
    xEditorForm.ShowModal;
  finally
    xEditorForm.Free;
  end;
end;

procedure TInvoiceForm.actEditInvoiceLineExecute(Sender: TObject);
var
```

```
  xEditorForm: TEditInvoiceLineForm;
begin
  xEditorForm := TEditInvoiceLineForm.Create(Self);
  try
    xEditorForm.EditMode := emInvoiceLineEdit;
    xEditorForm.OnClose := EditInvoiceLineEditorClose;
    xEditorForm.Caption := 'Edit invoice line';
    xEditorForm.InvoiceLineId := Invoices.InvoiceLine.INVOICE_LINE_ID.Value;
    xEditorForm.SetProduct(
      Invoices.InvoiceLine.PRODUCT_ID.Value,
      Invoices.InvoiceLine.PRODUCTNAME.Value,
      Invoices.InvoiceLine.SALE_PRICE.AsCurrency);
    xEditorForm.Quantity := Invoices.InvoiceLine.QUANTITY.Value;
    xEditorForm.ShowModal;
  finally
    xEditorForm.Free;
  end;
end;

procedure TInvoiceForm.AddInvoiceLineEditorClose(Sender: TObject;
  var Action: TCloseAction);
var
  xEditorForm: TEditInvoiceLineForm;
  xCustomerId: Integer;
begin
  xEditorForm := TEditInvoiceLineForm(Sender);
  if xEditorForm.ModalResult <> mrOK then
  begin
    Action := caFree;
    Exit;
  end;
  try
    Invoices.AddInvoiceLine(xEditorForm.ProductId, xEditorForm.Quantity);
    Action := caFree;
  except
    on E: Exception do
    begin
      Application.ShowException(E);
      // It does not close the window give the user correct the error
      Action := caNone;
    end;
  end;
end;

procedure TInvoiceForm.EditInvoiceLineEditorClose(Sender: TObject;
  var Action: TCloseAction);
var
  xCustomerId: Integer;
  xEditorForm: TEditInvoiceLineForm;
begin
  xEditorForm := TEditInvoiceLineForm(Sender);
  if xEditorForm.ModalResult <> mrOK then
  begin
    Action := caFree;
    Exit;
  end;
  try
    Invoices.EditInvoiceLine(xEditorForm.Quantity);
```

```
    Action := caFree;
  except
    on E: Exception do
    begin
      Application.ShowException(E);
      // It does not close the window give the user correct the error
      Action := caNone;
    end;
  end;
end;
```

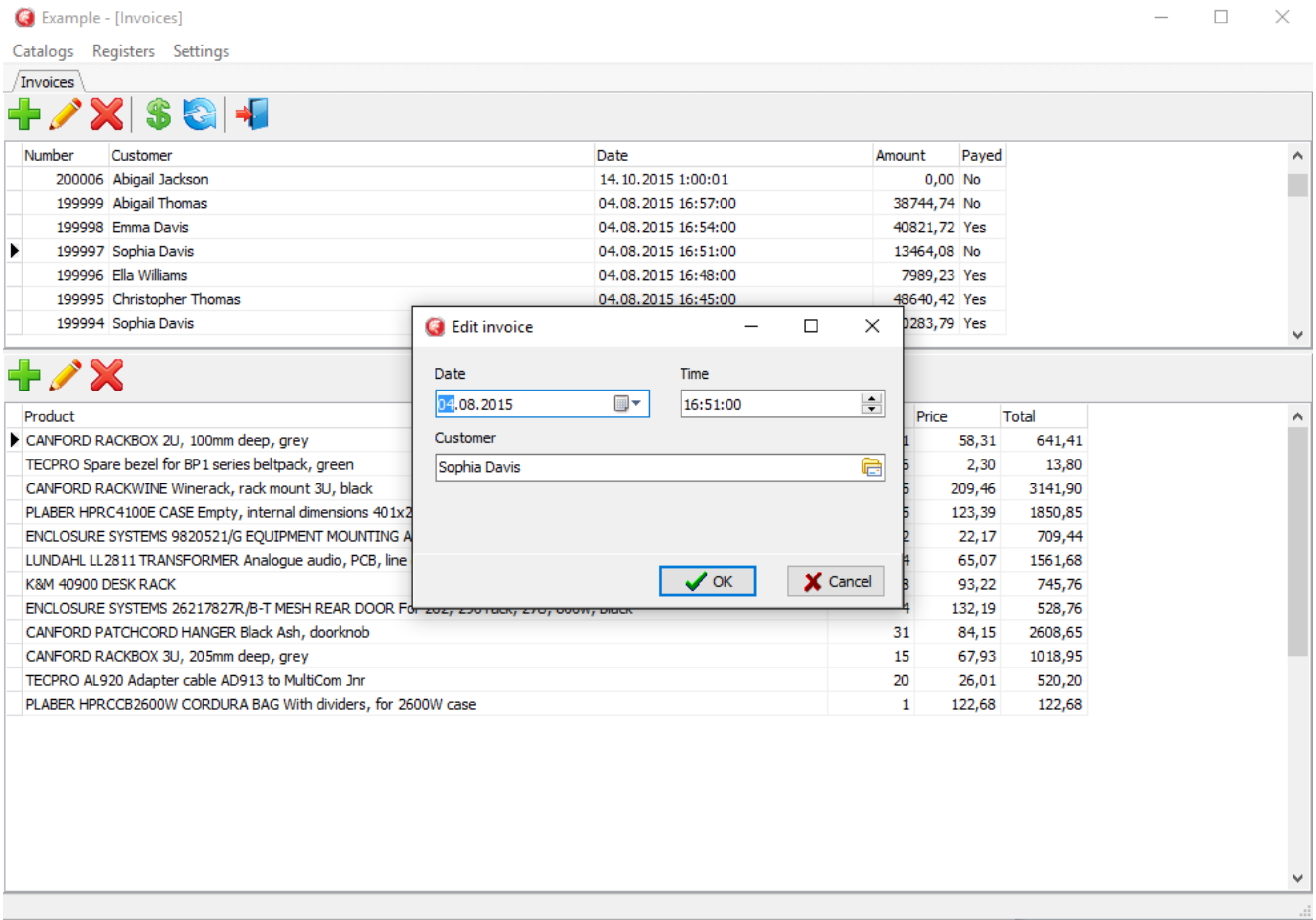
Now let's take a look at the code for the AddInvoiceLine and EditInvoiceLine procedures of the dmInvoice data module:

```
procedure TdmInvoice.AddInvoiceLine(AProductId: Integer; AQuantity: Integer);
begin
  // We do everything in a short transaction
  trWrite.StartTransaction;
  try
    qryAddInvoiceLine.ParamByName('INVOICE_ID').AsInteger :=
      Invoice.INVOICE_ID.Value;
    if AProductId = 0 then
      raise Exception.Create('Not selected product');
    qryAddInvoiceLine.ParamByName('PRODUCT_ID').AsInteger := AProductId;
    qryAddInvoiceLine.ParamByName('QUANTITY').AsInteger := AQuantity;
    qryAddInvoiceLine.Execute();
    trWrite.Commit;
    qryInvoice.Refresh;
    qryInvoiceLine.Refresh;
  except
    on E: Exception do
    begin
      if trWrite.Active then
        trWrite.Rollback;
      raise;
    end;
  end;
end;

procedure TdmInvoice.EditInvoiceLine(AQuantity: Integer);
begin
  // We do everything in a short transaction
  trWrite.StartTransaction;
  try
    qryEditInvoiceLine.ParamByName('INVOICE_LINE_ID').AsInteger :=
      InvoiceLine.INVOICE_LINE_ID.Value;
    qryEditInvoiceLine.ParamByName('QUANTITY').AsInteger := AQuantity;
    qryEditInvoiceLine.Execute();
    trWrite.Commit;
    qryInvoice.Refresh;
    qryInvoiceLine.Refresh;
  except
    on E: Exception do
    begin
      if trWrite.Active then
        trWrite.Rollback;
```

```
      raise;
    end;
  end;
end;
```

The Result

Figure 3.9. Screenshot of the sample application



As a result, we have an application that looks like this.

Conclusion

FireDac™ is a standard set of data-access and data-aware visual components for developing with various database systems, including Firebird, starting from Delphi™ XE3. FireDac™ ships with the higher-end versions of Delphi. Many independent sets of data access and data-aware visual components are available for working with Firebird, some commercial, others distributed under a variety of licences, including open source and freeware. They include FibPlus, IBObjects, UIB, UniDAC, IBDac, Interbase Express (IBX) and more. The principles for developing Firebird applications in Delphi™ are the same, regardless of the components you choose.

All queries to a database are executed within a transaction. To guarantee that applications will work correctly and efficiently with Firebird databases, it is advisable to manage transactions manually, by explicit calls to the *StartTransaction*, *Commit* and *Rollback* methods of the *TFDTransaction* component. Transactions should be as short as possible and you can use as many as the logic of your application requires.

The recommended configuration for a long-running, read-only transaction to view datasets is to use READ_COMMITTED isolation with REC_VERSION for conflict resolution. An application can run many datasets in one such transaction or one for each dataset, according to the requirements of the design.

To avoid holding an uncommitted transaction during an editing session, either use visual components that are not data-aware or use the *CachedUpdates* mode. With *CachedUpdates* you can restrict writes to short bursts of activity, keeping the read/write transaction active only for as long as it takes to post the most recent changes to the database.

The *TFDUpdateSQL* component is necessary for editing most datasets. Update queries are governed by its *InsertSQL*, *ModifySQL*, *DeleteSQL* and *FetchRowSQL* properties. The queries for those properties can be generated automatically by a wizard but manual corrections or adjustments are often required.

Acquiring values for auto-incrementing primary keys can be handled in one of two ways:

- Getting the value from the generator beforehand by specifying the *UpdateOptions.GeneratorName* and *UpdateOptions.AutoIncFields* properties for the *TFDQuery* component. This method cannot be used for auto-incrementing fields of the IDENTITY type that was introduced in Firebird 3.
- Getting the value by adding a RETURNING clause to the InsertSQL query. For this method you need to specify *Required=False* and *ReadOnly=True* for the field because the value is not entered directly.

It is convenient and sometimes necessary to implement more complex business logic with stored procedures. Using the *TFDCommand* component to execute stored procedures that do not return data reduces resource consumption
.

Source Code

ObjectPascal source code for the sample project is available for download using the following link: https://github.com/sim1984/FireDacEx.

For links to the database scripts and ready-to-use databases, refer to the final sections of the database chapter.

Chapter 4

Developing Firebird Applications with Microsoft Entity Framework

This chapter will describe the process of creating applications with a Firebird database using the Microsoft™ Entity Framework™ access components in the Visual Studio 2015™ environment.

ADO.NET Entity Framework (EF) combines an object-oriented data access technology with an object-relational mapping (ORM) solution for the Microsoft .NET Framework. It enables interaction with objects by means of both LINQ in the form of **LINQ to Entities** and with Entity SQL.

Methods of Interacting with a Database

Entity Framework assumes three possible methods for interacting with a database:

Database first:
: Entity Framework creates a set of classes that reflect the model of an existing database.

Model first:
: the developer creates a database model that Entity Framework later uses to create an actual database on the server.

Code first:
: the developer creates a class for the model of the data that will be stored in a database and then Entity Framework uses this model to generate the database and its tables

Our sample application will use the **Code first** approach, but you could use one of the others just as easily.

> **Note**
>
> As we already have a database, we will just write the code that would result in creating that database.

Setting Up for Firebird in Visual Studio 2015

To prepare for working with Firebird, you will need to install the following:

- FirebirdSql.Data.FirebirdClient.dll
- the Firebird DDEX Provider for Visual Studio

- EntityFramework.Firebird.dll

There is nothing difficult in installing the first two. They are currently distributed and installed into a project by means of the NuGet package manager. The DDEX Provider library, designed for operating Visual Studio wizards, is not so easy to install and may take more time and effort.

Efforts have been made to automate the installation process and include all components in a single installer package. However, you might need to install all of the components manually under some conditions. If so, you can download the following:

- FirebirdSql.Data.FirebirdClient-4.10.0.0.msi
- EntityFramework.Firebird-4.10.0.0-NET45.7z
- DDEXProvider-3.0.2.0.7z
- DDEXProvider-3.0.2.0-src.7z

The Installation Process

> **Important!**
>
> Because the installation involves operations in protected directories, you will need administrator privileges to do it.

Steps

1. Install FirebirdSql.Data.FirebirdClient-4.10.0.0.msi
2. Unpack EntityFramework.Firebird-4.10.0.0-NET45.7z to the folder with the installed Firebird client. In my case, it is the folder `c:\Program Files (x86)\FirebirdClient\`.
3. You need to install a Firebird build into the GAC. For your convenience, specify the path to the *gacutil* utility for .NET Framework 4.5 in the environment variable *%PATH%*. In my case, the path is `c:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\`
4. Run the command shell cmd.exe as administrator and go to the directory with the installed client, e.g.,

   ```
   chdir "c:\Program Files (x86)\FirebirdClient"
   ```

5. Now make sure that `FirebirdSql.Data.FirebirdClient` is installed into the GAC by typing the following command:

   ```
   gacutil /l FirebirdSql.Data.FirebirdClient
   ```

   If FirebirdSql.Data.FirebirdClient has not been installed into the GAC, use the following command to do it now:

   ```
   gacutil /i FirebirdSql.Data.FirebirdClient.dll
   ```

6. Now install EntityFramework.Firebird into the GAC

```
gacutil /i EntityFramework.Firebird.dll
```

7. Unpack DDEXProvider-3.0.2.0.7z to a directory convenient for you. Mine was unpacked to `c:\Program Files (x86)\FirebirdDDEX\`.

8. Unpack the contents of the `/reg_files/VS2015` subdirectory from the archive `DDEXProvider-3.0.2.0-src.7z` there as well.

> **Author's remark**
>
> For some strange reason these files are absent from the archive with the compiled dll libraries, but they are present in the source code archive.

9. Open the `FirebirdDDEXProvider64.reg` file in Notepad. Find the line that contains *%path%* and change it to the full path to the file `FirebirdSql.VisualStudio.DataTools.dll`, e.g.,

```
"CodeBase"="c:\\Program Files (x86)\\FirebirdDDEX\\FirebirdSql.VisualStudio.DataTools.dl
```

10. Save this Registry file and run it. Click YES to the question about adding the information to the Registry.

11. Now you need to edit the `machine.config` file. In my installation, the path is as follows:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config
```

Open this file in Notepad. Find the following section:

```
<system.data>
    <DbProviderFactories>
```

Add the following lines to this section:

```
<add name="FirebirdClient Data Provider"
     invariant="FirebirdSql.Data.FirebirdClient"
     description=".Net Framework Data Provider for Firebird"
     type="FirebirdSql.Data.FirebirdClient.FirebirdClientFactory,
           FirebirdSql.Data.FirebirdClient, Version=4.10.0.0, Culture=neutral,
           PublicKeyToken=3750abcc3150b00c" />
```

> **Note**
>
> The settings we have configured here are valid for version 4.10.0.

Do the same for `machine.config` located at `c:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\`

This completes the installation.

Testing the Installation

To make sure that everything has been installed successfully, start Visual Studio 2015. Find the Server Explorer and try to connect to an existing Firebird database.

Figure 4.1. Choose data source for testing installation

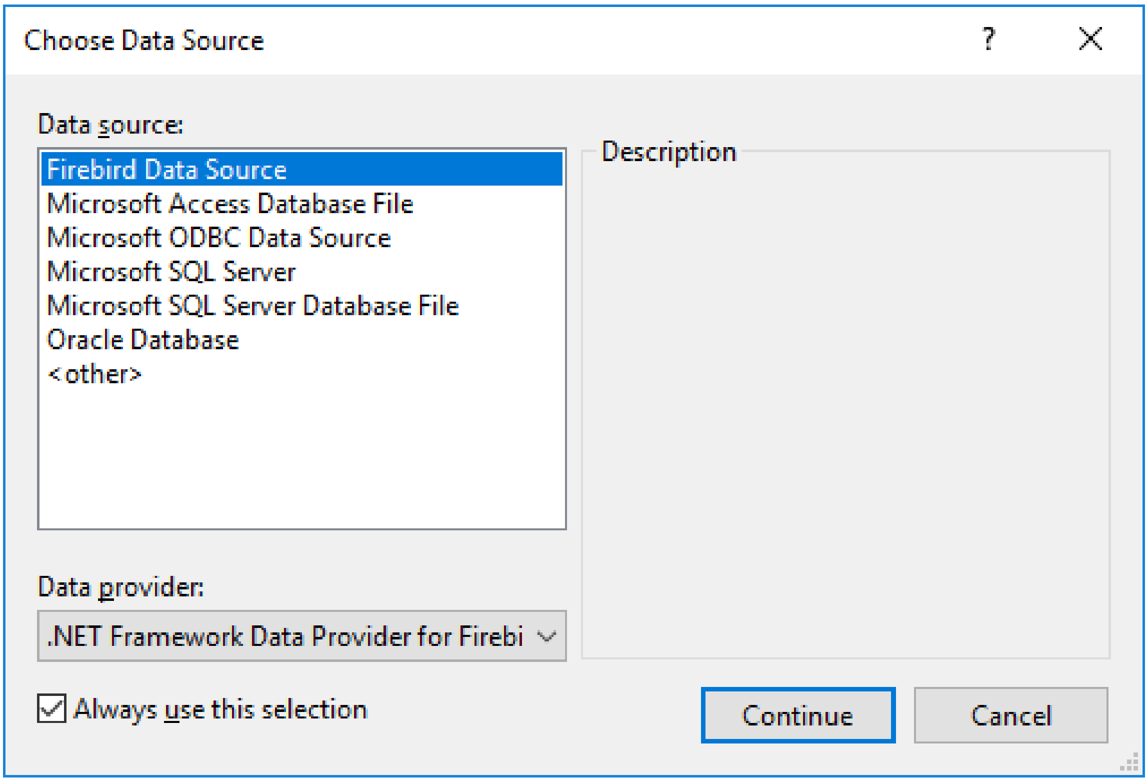


Figure 4.2. Locate a database

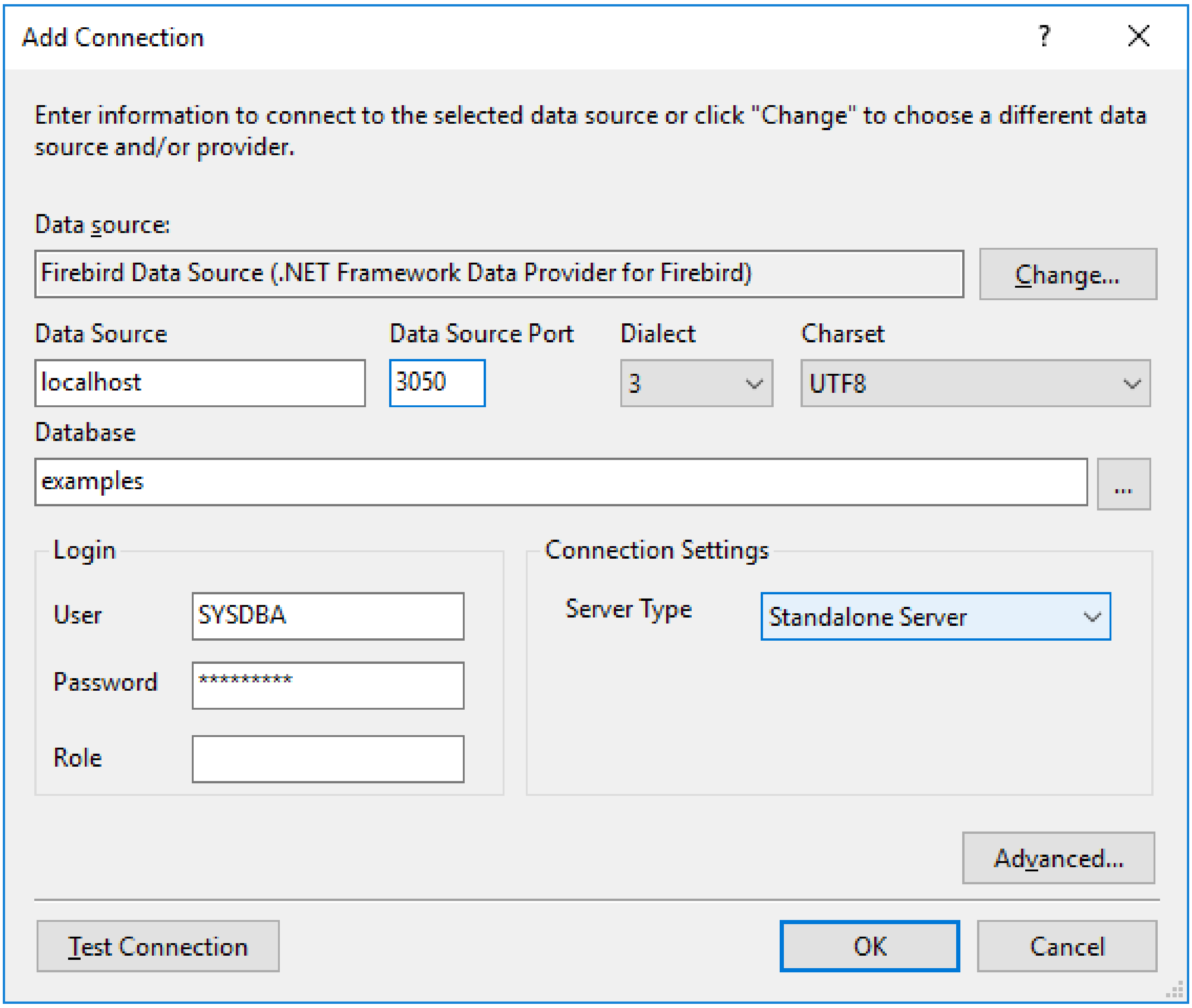
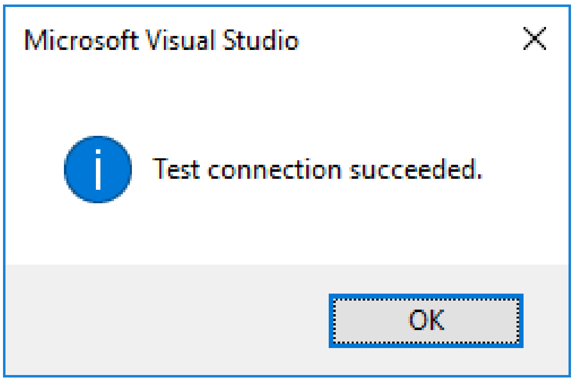


Figure 4.3. Test and confirm the connection



Creating a Project

For our example in this chapter, we will create a *Windows Forms* application. Other types of applications differ from it, but the principles of working with Firebird via Entity Framework remain the same.

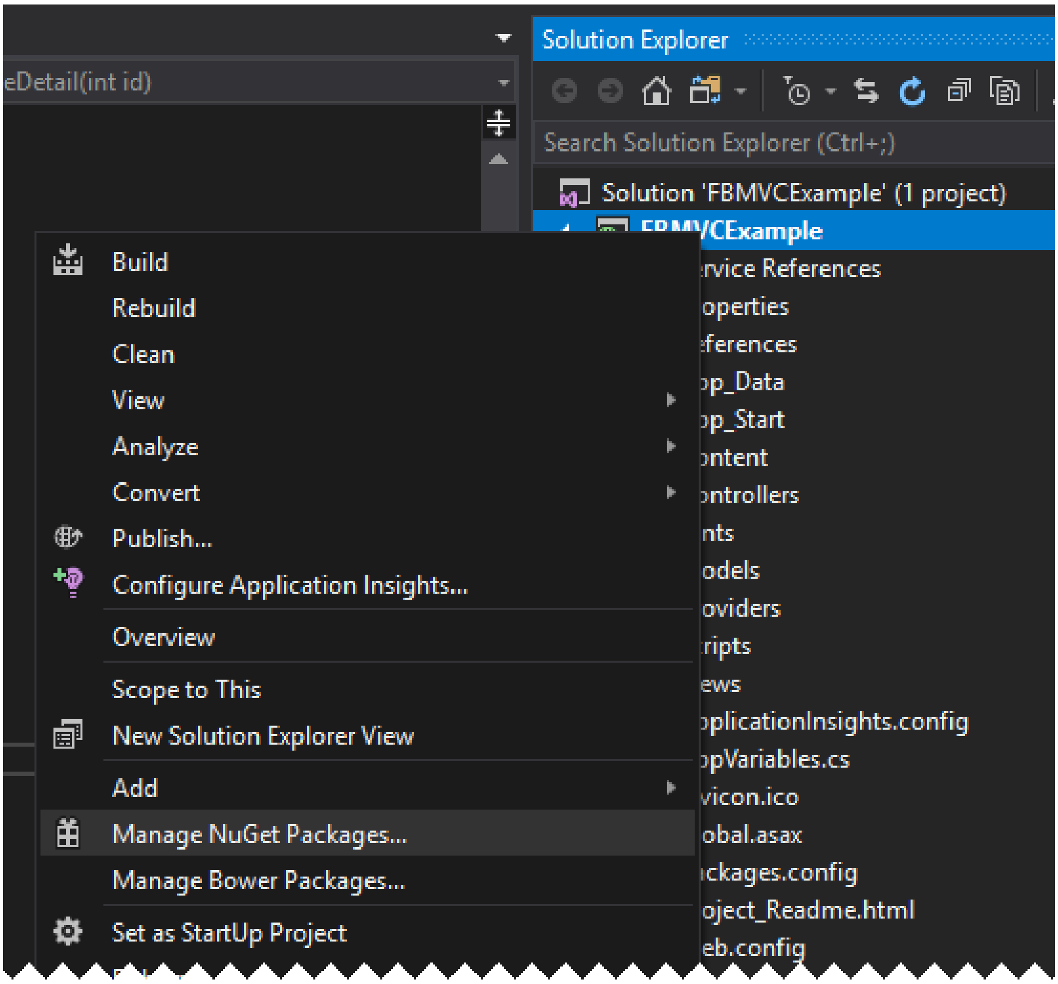
Adding Packages to the Project

The first task after creating a Windows Forms project is to add the following packages to it, using the NuGet package manager:

- `FirebirdSql.Data.FirebirdClient`
- `EntityFramework`
- `EntityFramework.Firebird`

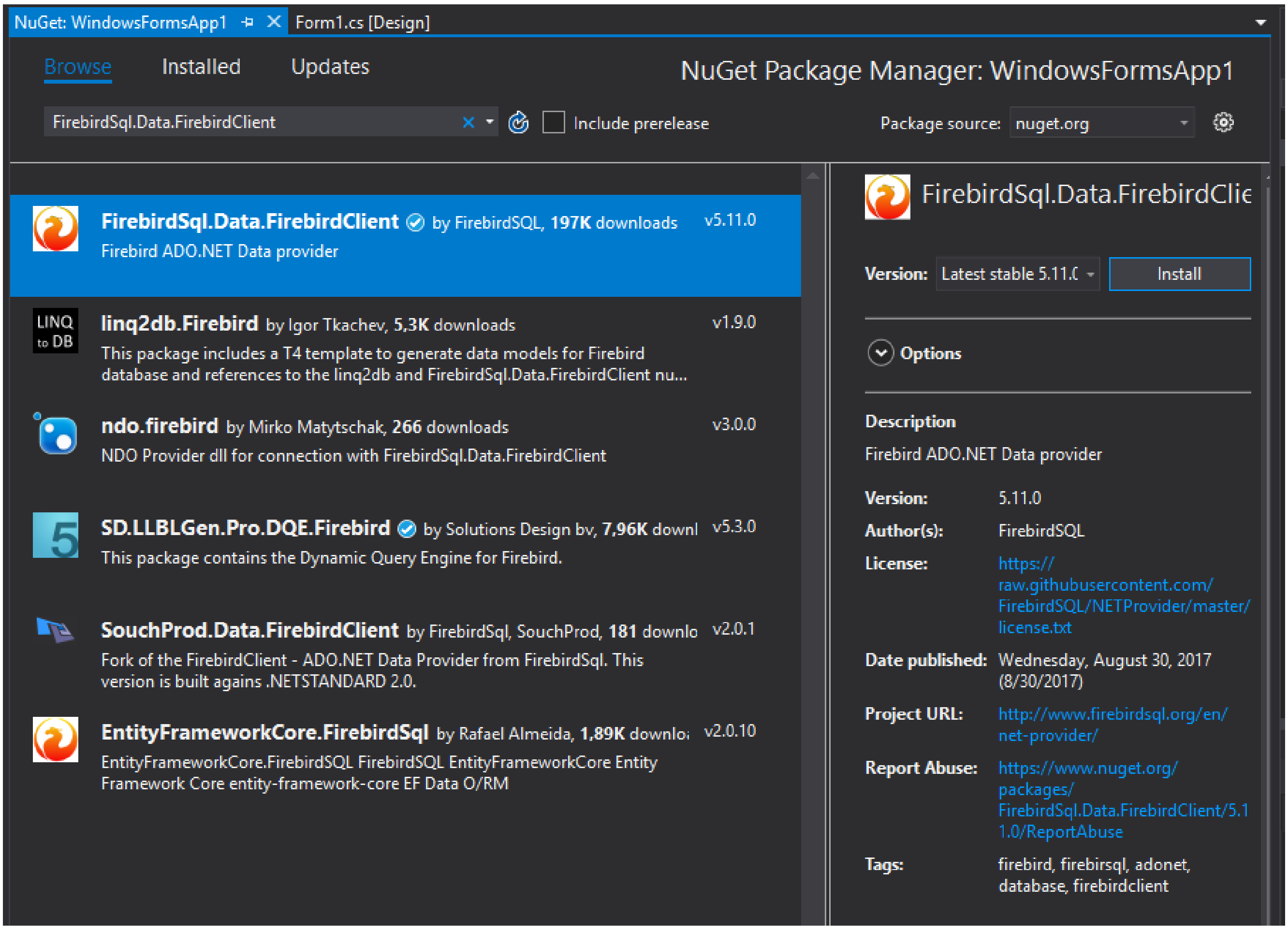
Right-click the project name in Solution Explorer and select *Manage NuGet Packages* from the drop-down list.

Figure 4.4. Solution Explorer—>select NuGet packages



Find the packages listed above in the Nuget catalogue and install them in the package manager.

Figure 4.5. Select and install packages from NuGet catalogue

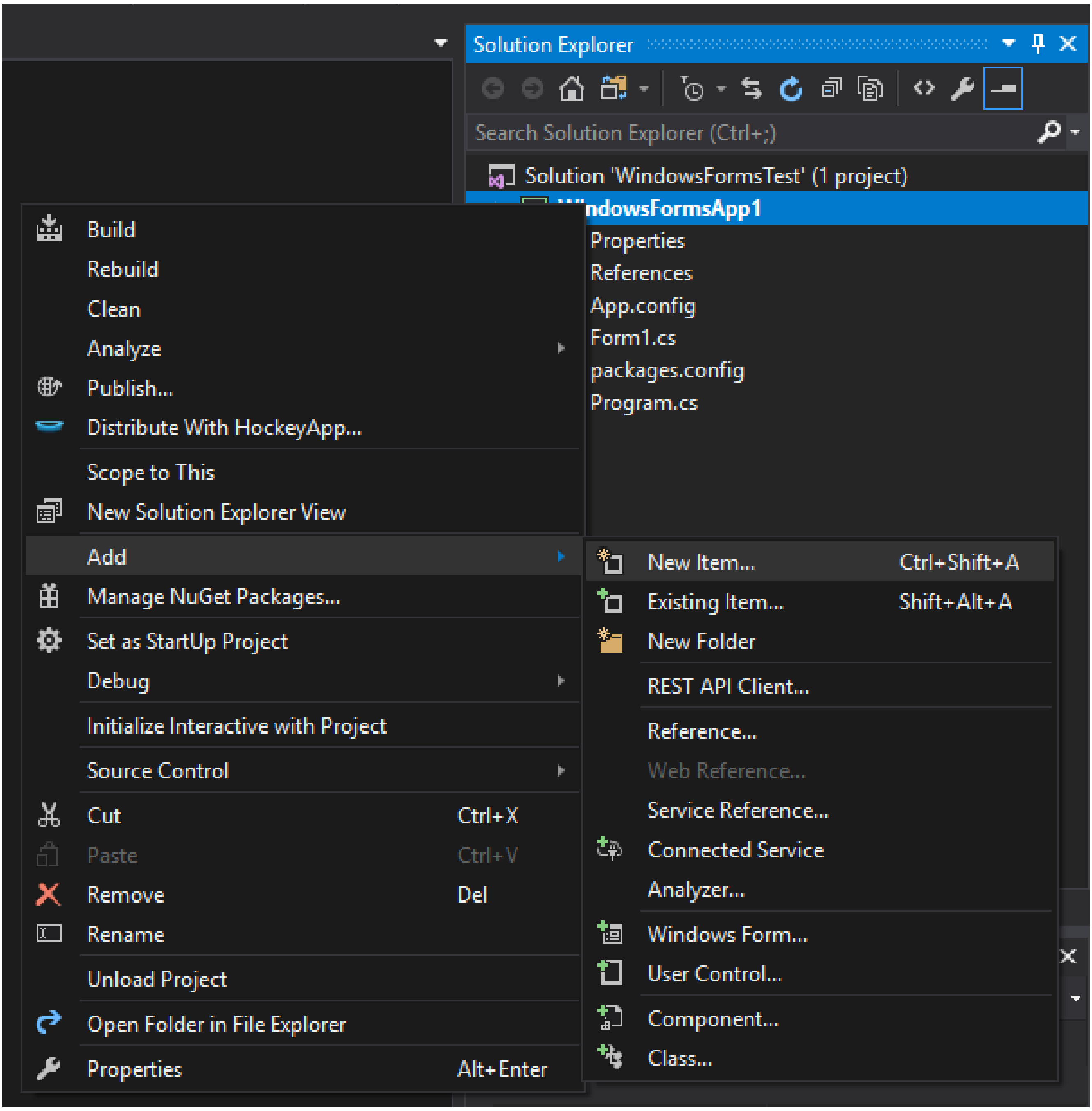


Creating an Entity Data Model (EDM)

In our application, we will use the *Code First* approach.

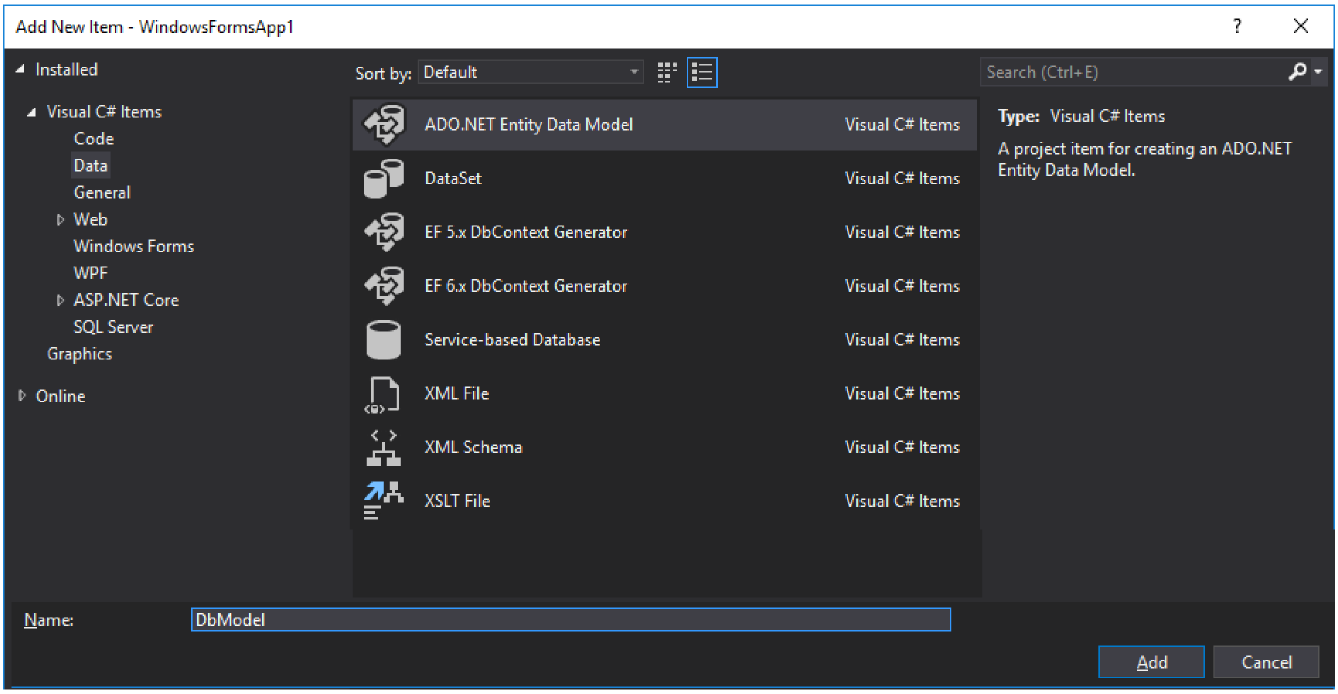
To create an EDM, right-click the project name in Solution Explorer and select *Add—>New Item* from the menu.

Figure 4.6. Solution Explorer - Add—>New Item



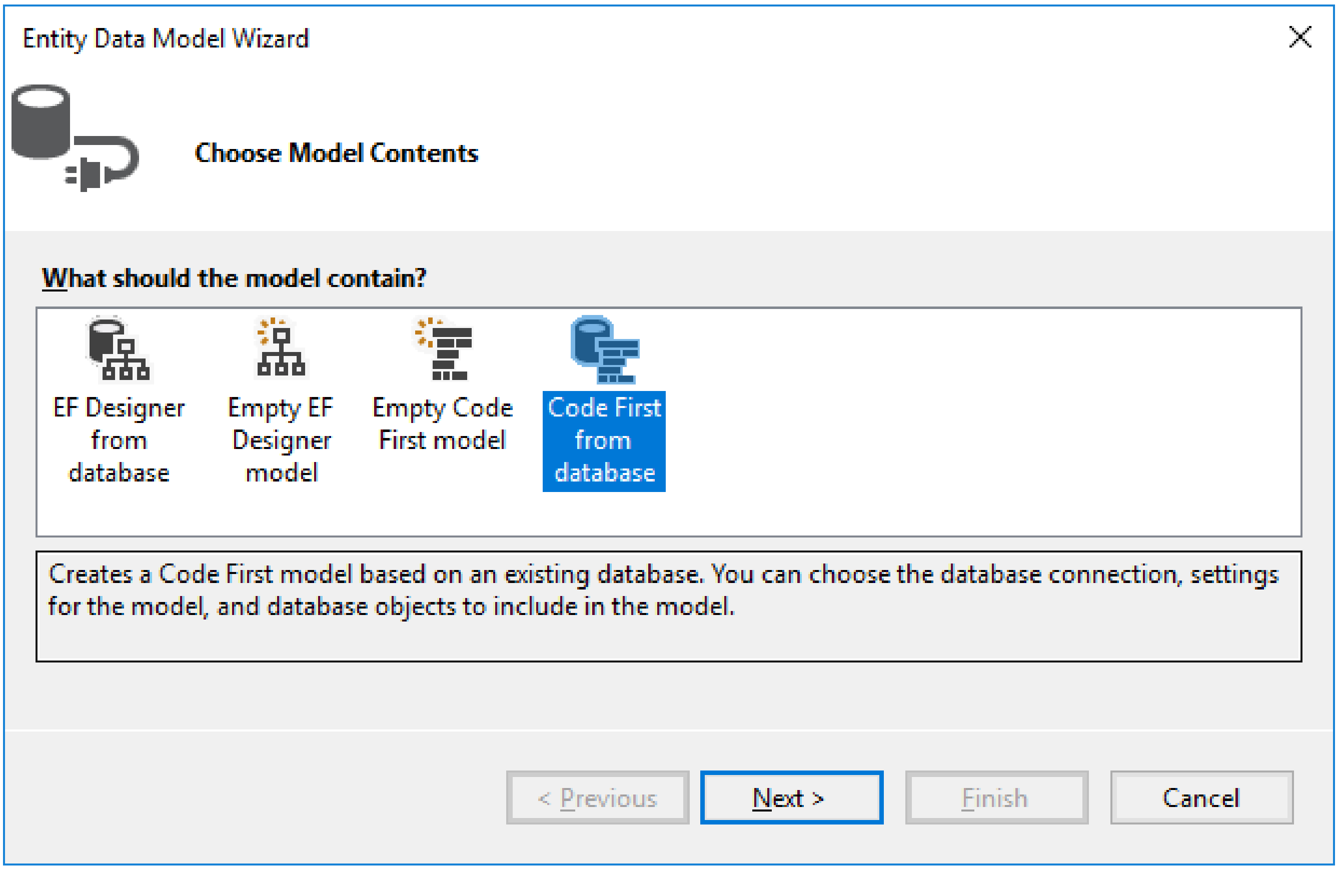
Next, in the *Add New Item* wizard, select *ADO.NET Entity Data Model*.

Figure 4.7. Add New Item wizard - select ADO.NET Entity Data Model



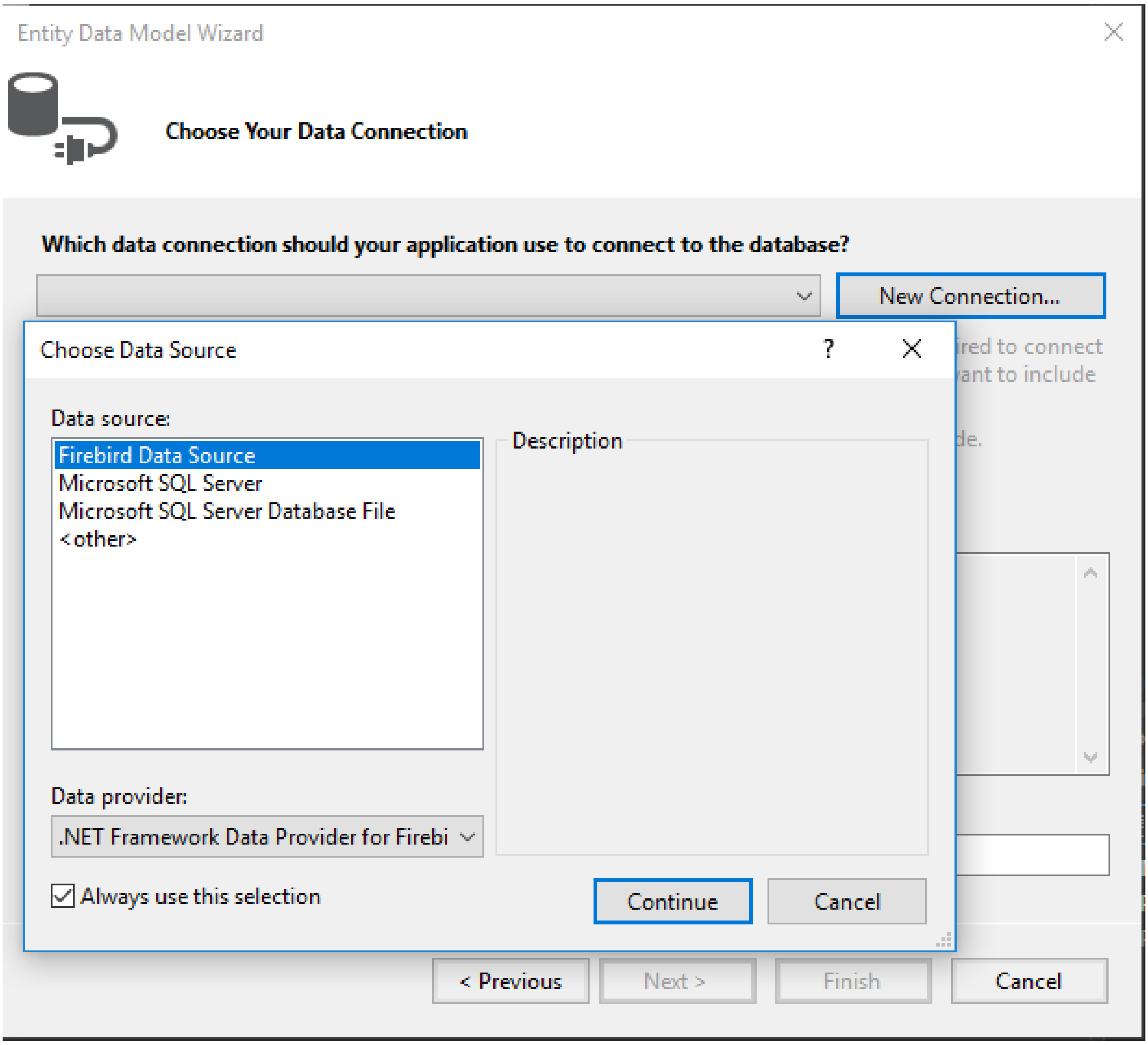
Since we already have a database, we will generate the EDM from the database. Select the icon captioned *Code First from database*.

Figure 4.8. Add New Item wizard - select 'Code First from database'



Now we need to to select the connection the model will be created from. If the connection does not exist, it will have to be created.

Figure 4.9. Add New Item wizard - choose Connection



You might need to specify some advanced properties in addition to the main connection properties. You might want to set the transaction isolation, for example, to a level different from the default *Read Committed*, or to specify connection pooling, or something else that differs from defaults.

Figure 4.10. Add Connection wizard - Connection properties

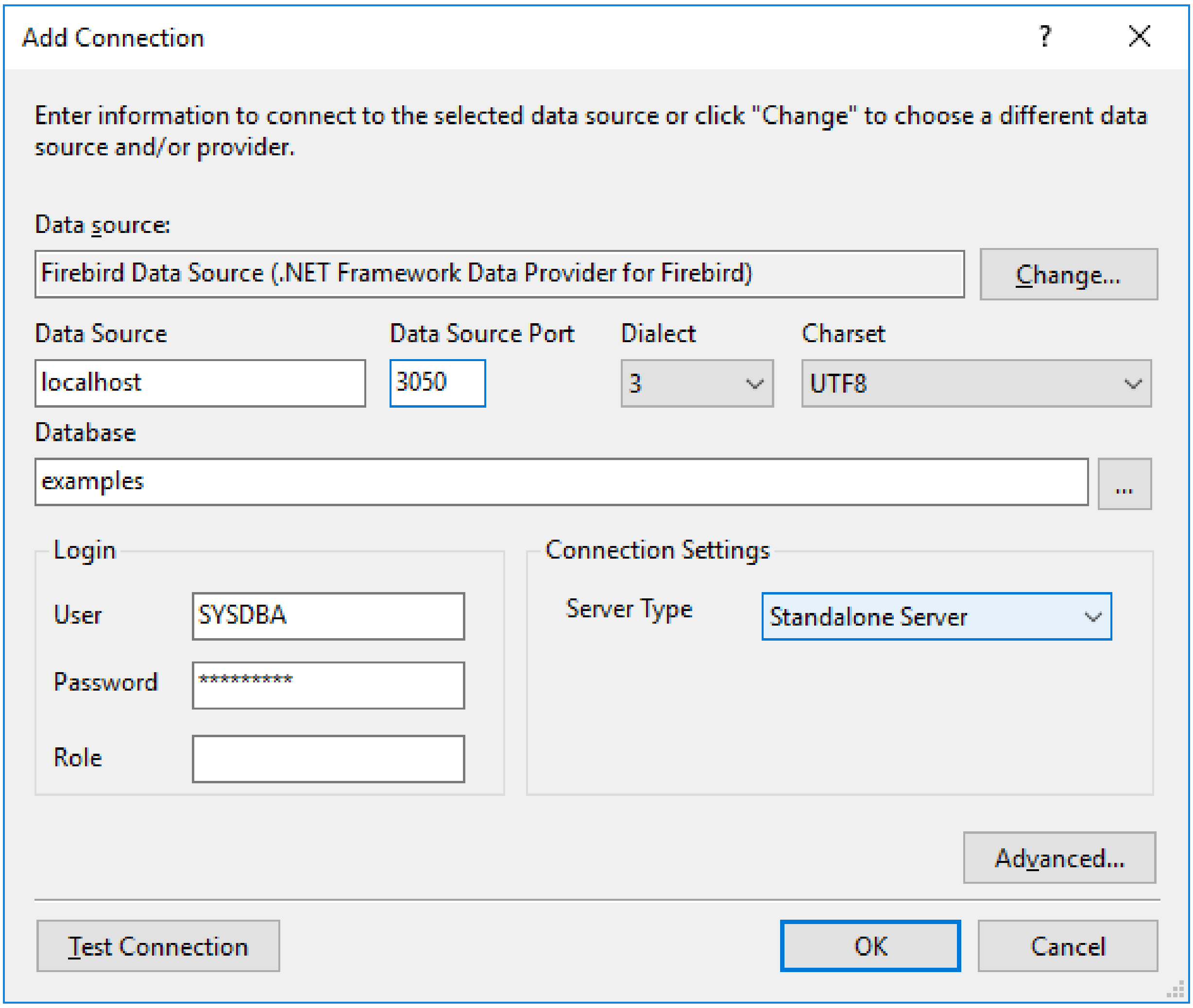
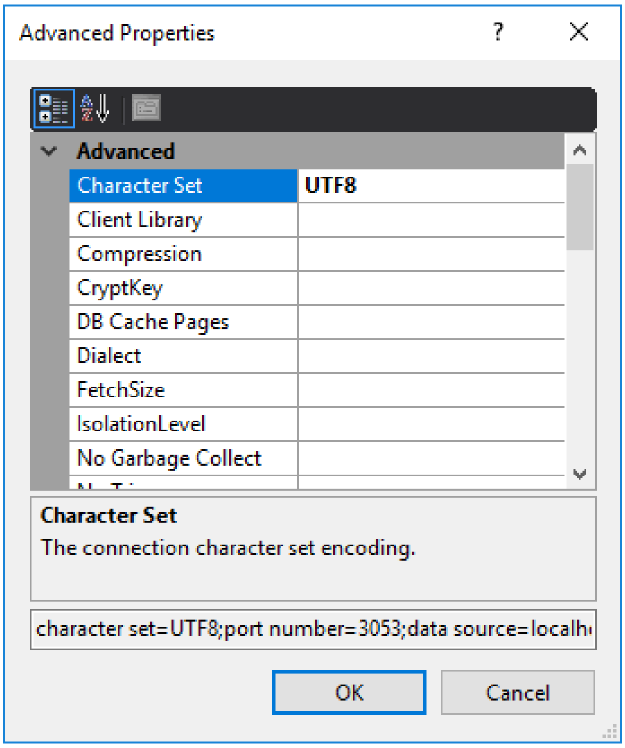


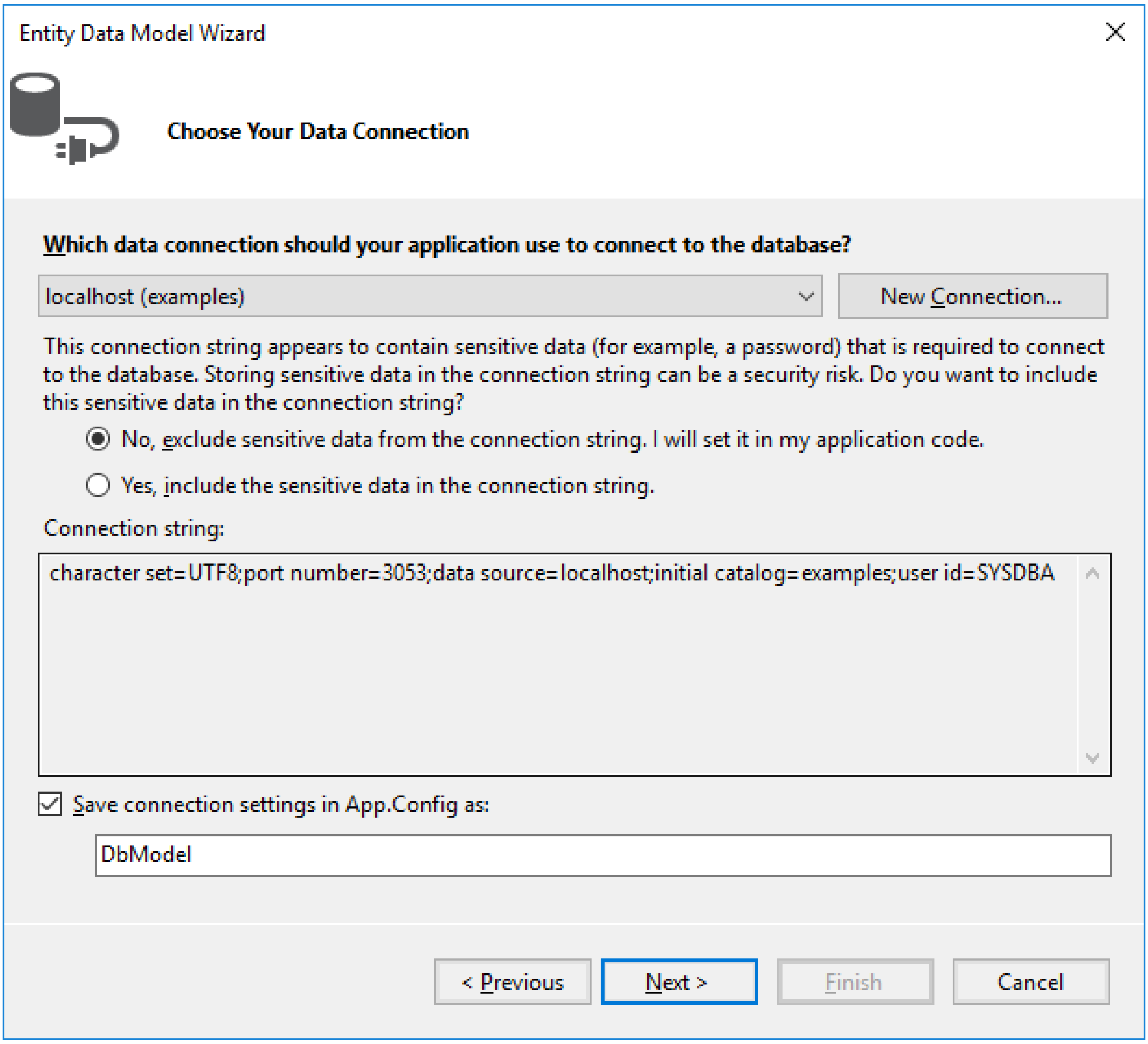
Figure 4.11. Add Connection wizard - Advanced connection properties



> **Tip**
>
> Snapshot is the recommended isolation level because Entity Framework and ADO.NET both use disconnected data access—where each connection and each transaction is active only for a very short time.

Next, the Entity Data Model wizard will ask you how to store the connection string.

Figure 4.12. EDM wizard - connection string storage



For a web application or another three-tier architecture, where all users will be working with the database using a single account, select **Yes**. If your application is going to request authentication for connecting to the database, select **No**.

Tip

It is much more convenient to work with wizards if you select Yes for each property. You can always change the isolation level 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 like this:

```
<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" />
```

For the configuration file to stop storing the confidential information, just delete this parameter from the connection string: `password=masterkey;`

Firebird 3.0 Notes

Unfortunately, the current ADO.Net provider for Firebird (version 5.9.0.0) does not support network traffic encryption, which is enabled by default in Firebird 3.0 and higher versions. If you want to work with Firebird 3.0, you need to change some settings in `firebird.conf` (or in `databases.conf` for a specific database) to make Firebird to work without trying to use network encryption.

To do it, change the setting from the default

```
# WireCrypt = Enabled
```

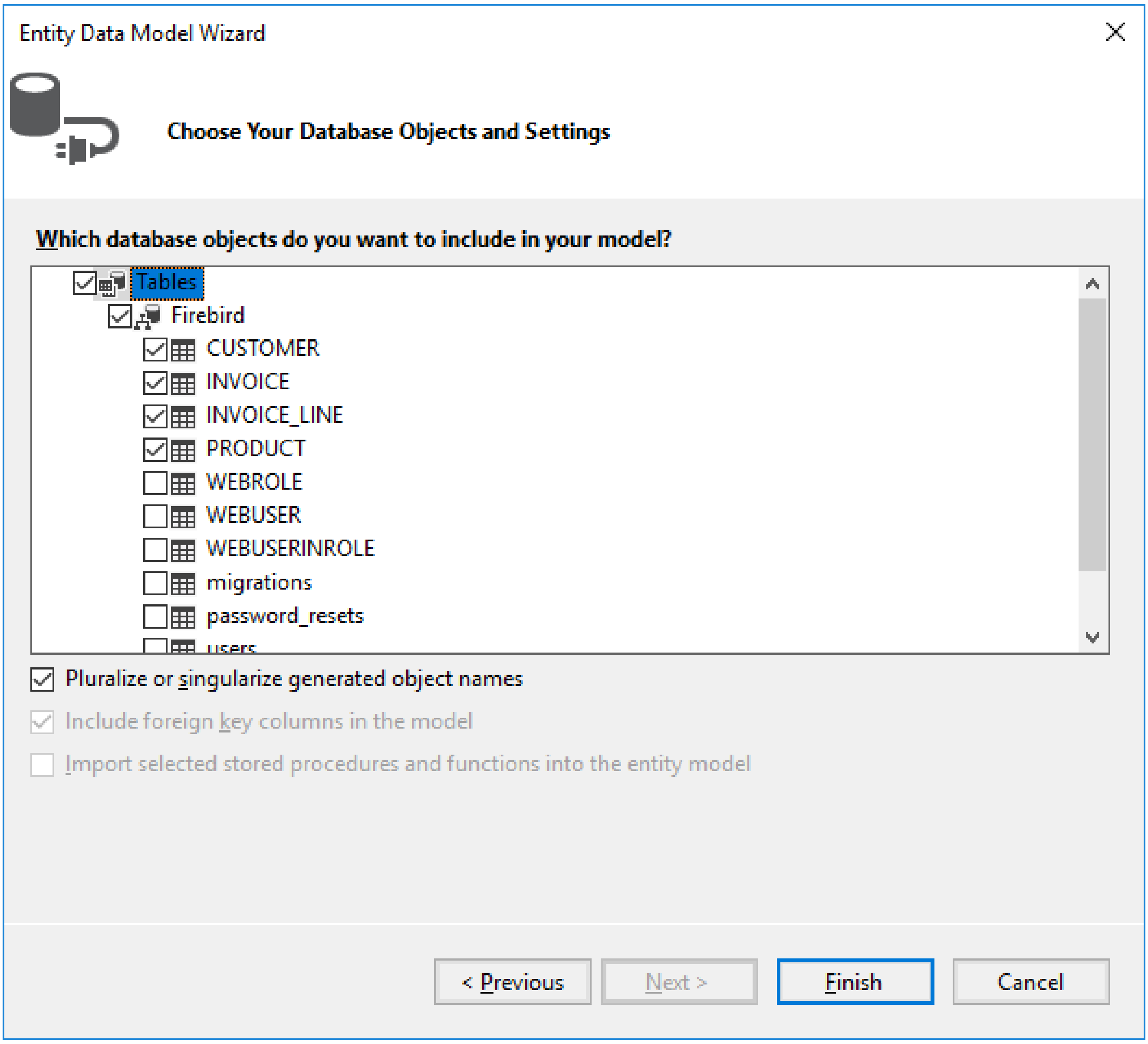
to

```
WireCrypt = Disabled
```

making sure to delete the '#' comment marker. Remember that you must restart the server for configuration changes to take effect.

Next, you will be asked which tables and views should be included in the model.

Figure 4.13. EDM wizard - select tables and views



For our project, select the four tables that are checked in the screenshot.

The basic EDM is now ready.

The EDM Files

When the wizard's work is finished, you should have five new files: a model file and four files each describing an entity in the model.

An Entity File

Let's take a look at the generated file describing the INVOICE entity:

```
[Table("Firebird.INVOICE")]
public partial class INVOICE
{
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2214:DoNotCallOverridableMethodsInConstructors")]
    public INVOICE()
    {
      INVOICE_LINES = new HashSet<INVOICE_LINE>();
    }

    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int INVOICE_ID { get; set; }

    public int CUSTOMER_ID { get; set; }

    public DateTime? INVOICE_DATE { get; set; }

    public decimal? TOTAL_SALE { get; set; }

    public short PAYED { get; set; }

    public virtual CUSTOMER CUSTOMER { get; set; }

    [System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",
"CA2227:CollectionPropertiesShouldBeReadOnly")]
    public virtual ICollection<INVOICE_LINE> INVOICE_LINES { get; set; }
}
```

The class contains properties for each field of the INVOICE table. Each of these properties has attributes that describe constraints. You can study the details of the various attributes in the Microsoft document, Code First Data Annotations.

Navigation Properties and "Lazy Loading"

Two navigation properties are generated: CUSTOMER and INVOICE_LINES. The first one contains a reference to the customer entity. The second contains a collection of invoice lines. It is generated because the INVOICE_LINE table has a foreign key to the INVOICE table. Of course, you can remove this property from the INVOICE entity, but it is not really necessary. The CUSTOMER and INVOICE_LINES properties use "lazy loading" which means that loading is not performed until the first access to an object. That way, the loading of related data is avoided unless it is actually needed. Once the data are accessed via the navigation property, they will be loaded from the database automatically.

> **Important**
>
> If lazy loading is in effect, classes that use it must be public and their properties must have the keywords `public` and `virtual`.

The DbModel File

Next, we examine the `DbModel.cs` file that describes the overall model.

```
    public partial class DbModel : DbContext
```

```
    {
        public DbModel()
            : base("name=DbModel")
        {
        }

        public virtual DbSet<CUSTOMER> CUSTOMERS { get; set; }
        public virtual DbSet<INVOICE> INVOICES { get; set; }
        public virtual DbSet<INVOICE_LINE> INVOICE_LINES { get; set; }
        public virtual DbSet<PRODUCT> PRODUCTS { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<CUSTOMER>()
                .Property(e => e.ZIPCODE)
                .IsFixedLength();

            modelBuilder.Entity<CUSTOMER>()
                .HasMany(e => e.INVOICES)
                .WithRequired(e => e.CUSTOMER)
                .WillCascadeOnDelete(false);

            modelBuilder.Entity<PRODUCT>()
                .HasMany(e => e.INVOICE_LINES)
                .WithRequired(e => e.PRODUCT)
                .WillCascadeOnDelete(false);

            modelBuilder.Entity<INVOICE>()
                .HasMany(e => e.INVOICE_LINES)
                .WithRequired(e => e.INVOICE)
                .WillCascadeOnDelete(false);

        }
    }
```

The properties coded here describe a dataset for each entity, along with advanced properties that are specified for creating a model with Fluent API. A complete description of the Fluent API can be found in the Microsoft document entitled [Configuring/Mapping Properties and Types with the Fluent API](#).

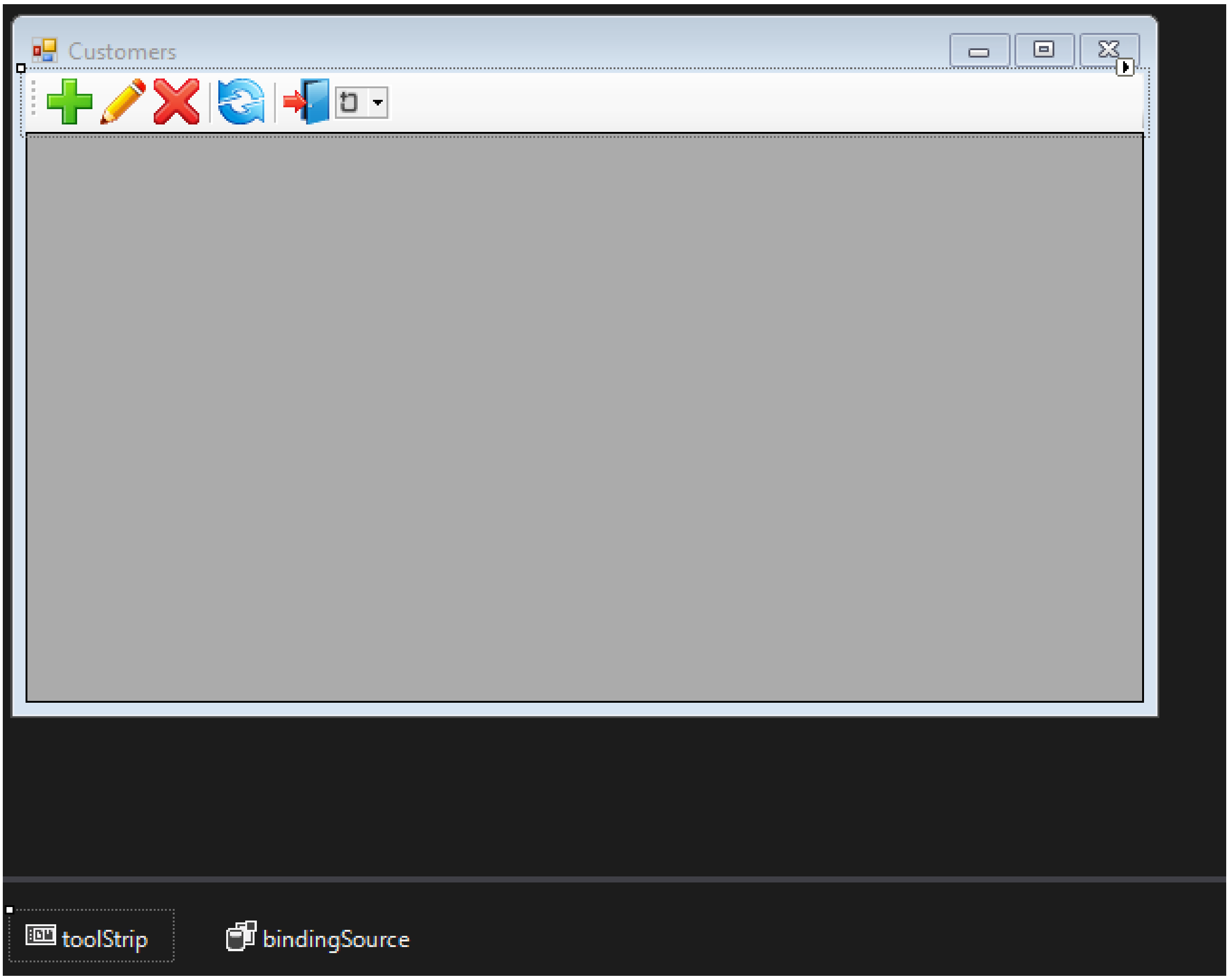
We will use the Fluent API to specify precision and scale for properties of type DECIMAL in the *OnModelCreating* method, by adding the following lines:

```
            modelBuilder.Entity<PRODUCT>()
                .Property(p => p.PRICE)
                .HasPrecision(15, 2);
            modelBuilder.Entity<INVOICE>()
                .Property(p => p.TOTAL_SALE)
                .HasPrecision(15, 2);

            modelBuilder.Entity<INVOICE_LINE>()
                .Property(p => p.SALE_PRICE)
                .HasPrecision(15, 2);

            modelBuilder.Entity<INVOICE_LINE>()
                .Property(p => p.QUANTITY)
                .HasPrecision(15, 0);
```

Creating a User Interface

In our application, we will create interfaces for two primary entities: a form each for the product and the customer entities. Each form contains a DataGridView grid, a ToolStrip toolbar with buttons and also a BindingSource component that is used to bind data to the controls on the form.

Figure 4.14. A form for the Customer entity



Since both forms are similar in function and implementation, we will describe just one.

Getting a Context

To work with our model, we will need the method for getting a context (or a model). The following statement is sufficient for that purpose:

```
DbModel dbContext = new DbModel();
```

If no confidential data are stored in the connection string—for example, the password is absent because it will be captured during the authentication process when the application is started—we will need a special method for storing and recovering the connection string or for storing the previously created context. For that, we will create a special class containing some application-level global variables, along with a method for getting a context.

A context might be the start and end dates of a work period, for example.

```
static class AppVariables
{
    private static DbModel dbContext = null;

    /// <summary>
    /// Start date of the working period
    /// </summary>
    public static DateTime StartDate { get; set; }

    /// <summary>
    /// End date of the working period
    /// </summary>
    public static DateTime FinishDate { get; set; }

    /// <summary>
    /// Returns an instance of the model (context)
    /// </summary>
    /// <returns>Model</returns>
    public static DbModel CreateDbContext() {
        dbContext = dbContext ?? new DbModel();
        return dbContext;
    }
}
```

The connection string itself is applied after the authentication process completes successfully during the application launch. We will add the following code to the *Load* event handler of the main form for that.

```
private void MainForm_Load(object sender, EventArgs e) {
    var dialog = new LoginForm();
    if (dialog.ShowDialog() == DialogResult.OK)
    {
        var dbContext = AppVariables.getDbContext();
        try
        {
            string s = dbContext.Database.Connection.ConnectionString;
            var builder = new FbConnectionStringBuilder(s);
            builder.UserID = dialog.UserName;
            builder.Password = dialog.Password;
            dbContext.Database.Connection.ConnectionString = builder.ConnectionString;
            // try connect
            dbContext.Database.Connection.Open();
        }
        catch (Exception ex)
        {
           // display error
           MessageBox.Show(ex.Message, "Error");
           Application.Exit();
```

```
        }
     }
     else
        Application.Exit();
}
```

Now, to get a context, we use the static *CreateDbContext* method:

```
var dbContext = AppVariables.getDbContext();
```

Working with Data

The entities in the model definition contain no data. The easiest way to to load data is to call the *Load* method. For example,

```
private void LoadCustomersData()
{
    dbContext.CUSTOMERS.Load();
    var customers = dbContext.CUSTOMERS.Local;
    bindingSource.DataSource = customers.ToBindingList();
}

private void CustomerForm_Load(object sender, EventArgs e)
{
    LoadCustomersData();
    dataGridView.DataSource = bindingSource;
    dataGridView.Columns["CUSTOMER_ID"].Visible = false;
}
```

However, this approach has a few drawbacks:

1. The *Load* method loads all data from the CUSTOMER table to memory at once

2. Although lazy properties (INVOICES) are not loaded immediately, but only once they are accessed, they will be loaded anyway when the records are shown in the grid and it will happen each time a group of records is shown

3. Record ordering is not defined

To get around these drawbacks, we will use a feature of the LINQ (Language Integrated Query) technology, *LINQ to Entities*. LINQ to Entities offers a simple and intuitive approach to getting data using C# statements that are syntactically similar to SQL query statements. You can read about the LINQ syntax in LINQ to Entities.

LINQ Extension Methods

The LINQ extension methods can return two objects: *IEnumerable* and *IQueryable*. The *IQueryable* interface is inherited from *IEnumerable* so, theoretically, an *IQueryable* object is also an *IEnumerable*. In reality, they are distinctly different.

The *IEnumerable* interface is in the *System.Collections* namespace. An *IEnumerable* object is a collection of data in memory that can be addressed only in a forward direction. During the query execution, *IEnumerable* loads all data. Filtering, if required, is done on the client side.

The *IQueryable* interface is in the *System.Linq* namespace. It provides remote access to the database and movement through the data can be bi-directional. During the process of creating a query that returns an *IQueryable* object, the query is optimized to minimise memory usage and network bandwidth.

The *Local* property returns the IEnumerable interface, through which we can create LINQ queries.

```
private void LoadCustomersData()
{
    var dbContext = AppVariables.getDbContext();
    dbContext.CUSTOMERS.Load();
    var customers =
        from customer in dbContext.CUSTOMERS.Local
        orderby customer.NAME
        select new customer;
    bindingSource.DataSource = customers.ToBindingList();
}
```

However, as this query will be executed on the data in memory, it is really useful only for small tables that do not need to be filtered beforehand.

For a LINQ query to be converted into SQL and executed on the server, we need to access the dbContext.CUSTOMERS directly instead of accessing the *dbContext.CUSTOMERS.Local* property in the LINQ query. The prior call to *dbContext.CUSTOMERS.Load();* to load the collection to memory is not required.

IQueryable and BindingList

IQueryable objects present a small problem: they cannot return BindingList. *BindingList* is a base class for creating a two-way data-binding mechanism. We can use the *IQueryable* interface to get a regular list by calling *ToList* but, this way, we lose handy features such as sorting in the grid and several more. The deficiency was fixed in .NET Framework 5 by creating a special extension. To do the same thing in FW4, we will create our own solution.

```
public static class DbExtensions
{
    // Internal class for map generator values to it
    private class IdResult
    {
        public int Id { get; set; }
    }

    // Cast IQueryable to BindingList
    public static BindingList<T> ToBindingList<T>
       (this IQueryable<T> source) where T : class
    {
        return (new ObservableCollection<T>(source)).ToBindingList();
    }
```

```
    // Get the next value of the sequence
    public static int NextValueFor(this DbModel dbContext, string genName)
    {
        string sql = String.Format(
          "SELECT NEXT VALUE FOR {0} AS Id FROM RDB$DATABASE", genName);
        return dbContext.Database.SqlQuery<IdResult>(sql).First().Id;
    }

    // Disconnect all objects from the DbSet collection from the context
    // Useful for updating the cache
    public static void DetachAll<T>(this DbModel dbContext, DbSet<T> dbSet)
       where T : class
    {
        foreach (var obj in dbSet.Local.ToList())
        {
            dbContext.Entry(obj).State = EntityState.Detached;
        }
    }

    // Update all changed objects in the collection
    public static void Refresh(this DbModel dbContext, RefreshMode mode,
        IEnumerable collection)
    {
        var objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;
        objectContext.Refresh(mode, collection);
    }

    // Update the object
    public static void Refresh(this DbModel dbContext, RefreshMode mode,
        object entity)
    {
        var objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;
        objectContext.Refresh(mode, entity);
    }
}
```

Other Extensions

There are several more extensions in the *iQueryable* interface:

NextValueFor
: is used to get the next value from the generator.

dbContext.Database.SqlQuery
: allows SQL queries to be executed directly and their results to be displayed on some entity (projection).

DetachAll
: is used to detach all objects of the DBSet collection from the context. It is necessary to update the internal cache, because all retrieved data are cached and are not retrieved from the database again. However, that is not always useful because it makes it more difficult to get the latest version of records that were modified in another context.

> **Note**
>
> In web applications, a context usually exists for a very short period. A new context has an empty cache.

Refresh
: is used to update the properties of an entity object. It is useful for updating the properties of an object after it has been edited or added.

Code for Loading the Data

Our code for loading data will look like this:

```
private void LoadCustomersData()
{
    var dbContext = AppVariables.getDbContext();
    // disconnect all loaded objects
    // this is necessary to update the internal cache
    // for the second and subsequent calls of this method
    dbContext.DetachAll(dbContext.CUSTOMERS);
    var customers =
        from customer in dbContext.CUSTOMERS
        orderby customer.NAME
        select customer;
    bindingSource.DataSource = customers.ToBindingList();
}

private void CustomerForm_Load(object sender, EventArgs e)
{
    LoadCustomersData();
    dataGridView.DataSource = bindingSource;
    dataGridView.Columns["INVOICES"].Visible = false;
    dataGridView.Columns["CUSTOMER_ID"].Visible = false;
    dataGridView.Columns["NAME"].HeaderText = "Name";
    dataGridView.Columns["ADDRESS"].HeaderText = "Address";
    dataGridView.Columns["ZIPCODE"].HeaderText = "ZipCode";
    dataGridView.Columns["PHONE"].HeaderText = "Phone";
}
```

Adding a Customer

This is the code of the event handler for clicking the Add button:

```
private void btnAdd_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  // creating a new entity instance
  var customer = (CUSTOMER)bindingSource.AddNew();
  // create an editing form
  using (CustomerEditorForm editor = new CustomerEditorForm()) {
    editor.Text = "Add customer";
    editor.Customer = customer;
    // Form Close Handler
```

```
    editor.FormClosing += delegate (object fSender,
      FormClosingEventArgs fe) {
      if (editor.DialogResult == DialogResult.OK) {
        try {
          // get next sequence value
          // and assign it
          customer.CUSTOMER_ID = dbContext.NextValueFor("GEN_CUSTOMER_ID");
          // add a new customer
          dbContext.CUSTOMERS.Add(customer);
          // trying to save the changes
          dbContext.SaveChanges();
          // and update the current record
          dbContext.Refresh(RefreshMode.StoreWins, customer);
        }
        catch (Exception ex) {
          // display error
          MessageBox.Show(ex.Message, "Error");
          // Do not close the form to correct the error
          fe.Cancel = true;
        }
      }
      else
        bindingSource.CancelEdit();
    };
    // show the modal form
    editor.ShowDialog(this);
  }
}
```

While adding the new record, we used the generator to get the value of the next identifier. We could have done it without applying the value of the identifier, leaving the BEFORE INSERT trigger to fetch the next value of the generator and apply it. However, that would leave us unable to update the added record.

Editing a Customer

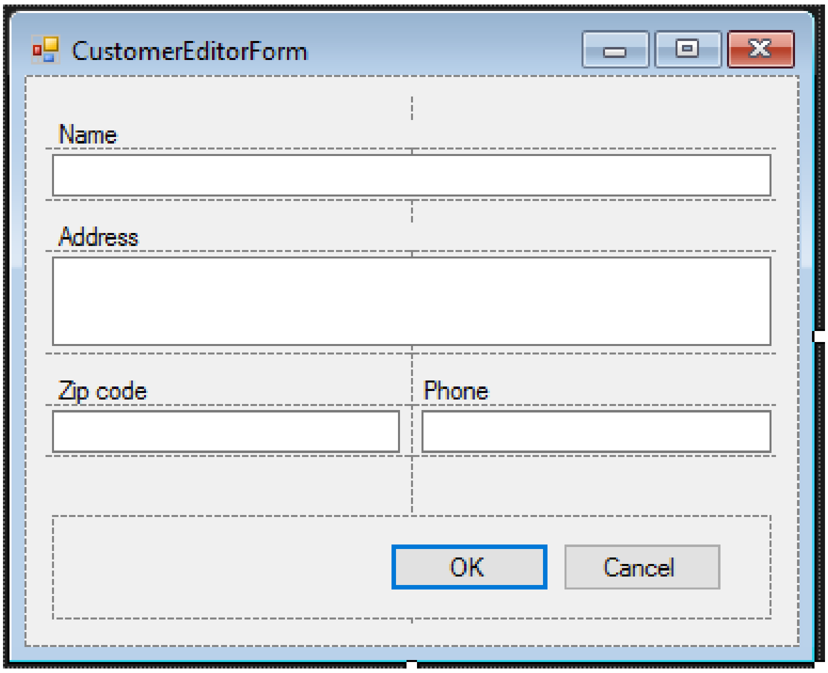
The code of the event handler for clicking the Edit button is as follows:

```
private void btnEdit_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  // get instance
  var customer = (CUSTOMER)bindingSource.Current;
  // create an editing form
  using (CustomerEditorForm editor = new CustomerEditorForm()) {
    editor.Text = "Edit customer";
    editor.Customer = customer;
    // Form Close Handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
      if (editor.DialogResult == DialogResult.OK) {
        try {
          // trying to save the changes
          dbContext.SaveChanges();
          dbContext.Refresh(RefreshMode.StoreWins, customer);
          // update all related controls
          bindingSource.ResetCurrentItem();
        }
        catch (Exception ex) {
```

```
          // display error
          MessageBox.Show(ex.Message, "Error");
          // Do not close the form to correct the error
          fe.Cancel = true;
        }
      }
      else
        bindingSource.CancelEdit();
    };
    // show the modal form
    editor.ShowDialog(this);
  }
}
```

The form for editing the customer looks like this:

Figure 4.15. Customer edit form



The code for binding to data is very simple.

```
public CUSTOMER Customer { get; set; }

private void CustomerEditorForm_Load(object sender, EventArgs e)
{
    edtName.DataBindings.Add("Text", this.Customer, "NAME");
    edtAddress.DataBindings.Add("Text", this.Customer, "ADDRESS");
    edtZipCode.DataBindings.Add("Text", this.Customer, "ZIPCODE");
    edtPhone.DataBindings.Add("Text", this.Customer, "PHONE");
}
```

Deleting a Customer

The code of the event handler for clicking the Delete button is as follows:

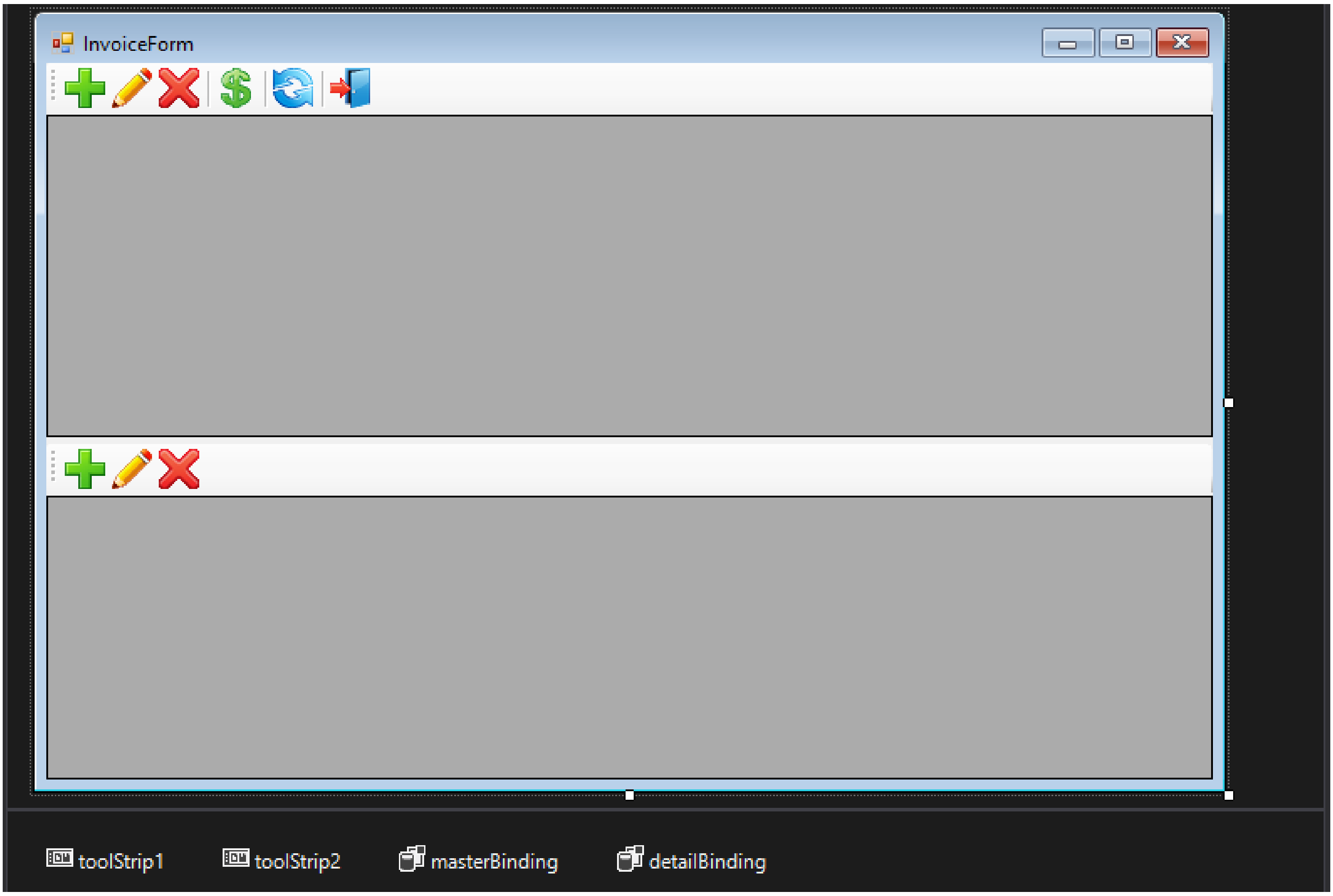
```
private void btnDelete_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  var result = MessageBox.Show("Are you sure you want to delete the customer?",
    "Confirmation",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question);
  if (result == DialogResult.Yes) {
    // get the entity
    var customer = (CUSTOMER)bindingSource.Current;
    try {
      dbContext.CUSTOMERS.Remove(customer);
      // trying to save the changes
      dbContext.SaveChanges();
      // remove from the linked list
      bindingSource.RemoveCurrent();
    }
    catch (Exception ex) {
      // display error
      MessageBox.Show(ex.Message, "Error");
    }
  }
}
```

Secondary Modules

Our application will have only one secondary module, named “Invoices”. Secondary modules typically contain larger numbers of records than primary ones and new records are added to them frequently.

An invoice consists of a title where some general attributes are described (number, date, customer …) and invoice lines with the list of products, their quantities, prices, etc. It is convenient to have two grids for such documents: the main one showing the invoice header data and the detail one for the list of products sold. We will need one DataGridView component for each entity on the document form, binding the appropriate BindingSource to each.

Figure 4.16. Invoice form



Filtering Data

Most secondary entities contain a field with the document creation date. To reduce the amount of retrieved data, the concept of a work period is usually introduced to filter the data sent to the client. A work period is a range of dates for which the records are required. Since the application can have more than one secondary entity, it makes sense to add variables containing the start and end dates of a work period to the global *AppVariables* data module (see Getting a Context that is used by all modules working with the database in one way or another. Once the application is started, the work period is usually defined by the dates when the current quarter starts and ends, although of course, other options are possible. While working with the application, the user can change the work period.

Since the most recent records are the most requested, it makes sense to sort them by date in reverse order. As with the primary modules, we will use LINQ to retrieve data.

Loading the Invoice Data

The following method loads the invoice headers:

```
public void LoadInvoicesData() {
    var dbContext = AppVariables.getDbContext();

    var invoices =
        from invoice in dbContext.INVOICES
        where (invoice.INVOICE_DATE >= AppVariables.StartDate) &&
              (invoice.INVOICE_DATE <= AppVariables.FinishDate)
        orderby invoice.INVOICE_DATE descending
        select new InvoiceView
        {
            Id = invoice.INVOICE_ID,
            Cusomer_Id = invoice.CUSTOMER_ID,
            Customer = invoice.CUSTOMER.NAME,
            Date = invoice.INVOICE_DATE,
            Amount = invoice.TOTAL_SALE,
            Payed = (invoice.PAYED == 1) ? "Yes" : "No"
        };
    masterBinding.DataSource = invoices.ToBindingList();
}
```

To simplify type casting, we define an *InvoiceView* class, rather than use some anonymous type. The definition is as follows:

```
public class InvoiceView {
    public int Id { get; set; }

    public int Cusomer_Id { get; set; }

    public string Customer { get; set; }

    public DateTime? Date { get; set; }

    public decimal? Amount { get; set; }

    public string Payed { get; set; }

    public void Load(int Id) {
        var dbContext = AppVariables.getDbContext();
        var invoices =
            from invoice in dbContext.INVOICES
            where invoice.INVOICE_ID == Id
            select new InvoiceView
            {
                Id = invoice.INVOICE_ID,
                Cusomer_Id = invoice.CUSTOMER_ID,
                Customer = invoice.CUSTOMER.NAME,
                Date = invoice.INVOICE_DATE,
                Amount = invoice.TOTAL_SALE,
                Payed = (invoice.PAYED == 1) ? "Yes" : "No"
            };

        InvoiceView invoiceView = invoices.ToList().First();
        this.Id = invoiceView.Id;
        this.Cusomer_Id = invoiceView.Cusomer_Id;
        this.Customer = invoiceView.Customer;
```

```
        this.Date = invoiceView.Date;
        this.Amount = invoiceView.Amount;
        this.Payed = invoiceView.Payed;
    }
}
```

The *Load* method allows us to update one added or updated record in the grid quickly, instead of completely reloading all records. Here is the code of the event handler for clicking the Add button:

```
private void btnAddInvoice_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  var invoice = dbContext.INVOICES.Create();
  using (InvoiceEditorForm editor = new InvoiceEditorForm()) {
    editor.Text = "Add invoice";
    editor.Invoice = invoice;
    // Form Close Handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
      if (editor.DialogResult == DialogResult.OK) {
        try {
          // get next sequence value
          invoice.INVOICE_ID = dbContext.NextValueFor("GEN_INVOICE_ID");
          // add a record
          dbContext.INVOICES.Add(invoice);
          // trying to save the changes
          dbContext.SaveChanges();
          // add the projection to the grid list
          ((InvoiceView)masterBinding.AddNew()).Load(invoice.INVOICE_ID);
        }
        catch (Exception ex) {
          // display error
          MessageBox.Show(ex.Message, "Error");
          // Do not close the form to correct the error
          fe.Cancel = true;
        }
      }
    };
    // show the modal form
    editor.ShowDialog(this);
  }
}
```

In our primary modules, the similarly-named method called dbContext.Refresh but, here, a record is updated by by calling the *Load* method of the *InvoiceView* class. The reason for the difference is that *dbContext.Refresh* is used to update entity objects, not the objects that can be produced by complex LINQ queries.

The code of the event handler for clicking the Edit button:

```
private void btnEditInvoice_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  // find entity by id
  var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
  if (invoice.PAYED == 1) {
    MessageBox.Show("The change is not possible, the invoice has already been paid.",
                    "Error");
    return;
```

```
  }
  using (InvoiceEditorForm editor = new InvoiceEditorForm()) {
    editor.Text = "Edit invoice";
    editor.Invoice = invoice;
    // Form Close Handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
      if (editor.DialogResult == DialogResult.OK) {
        try {
          // trying to save the changes
          dbContext.SaveChanges();
          // refresh
          CurrentInvoice.Load(invoice.INVOICE_ID);
          masterBinding.ResetCurrentItem();
        }
        catch (Exception ex) {
          // display error
          MessageBox.Show(ex.Message, "Error");
          // Do not close the form to correct the error
          fe.Cancel = true;
        }
      }
    };
    editor.ShowDialog(this);
  }
}
```

Here we needed to find an entity by the identifier provided in the current record. The CurrentInvoice is used to retrieve the invoice selected in the grid. This is how we code it:

```
public InvoiceView CurrentInvoice {
    get {
        return (InvoiceView)masterBinding.Current;
    }
}
```

Using the same approach, you can implement deleting the invoice header yourself.

Paying an Invoice

Besides adding, editing and deleting, we want one more operation for invoices: payment. Here is code for a method implementing this operation:

```
private void btnInvoicePay_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
  try {
    if (invoice.PAYED == 1)
      throw new Exception("The change is not possible, the invoice has already been paid.")
    invoice.PAYED = 1;
    // trying to save the changes
    dbContext.SaveChanges();
    // refresh record
    CurrentInvoice.Load(invoice.INVOICE_ID);
```

```
    masterBinding.ResetCurrentItem();
  }
  catch (Exception ex) {
    // display error
    MessageBox.Show(ex.Message, "Error");
  }
}
```

Showing the Invoice Lines

We have two choices for displaying the invoice lines:

1. Getting data for each invoice from the INVOICE_LINE navigation property and displaying the contents of this complex property in the detail grid, probably with LINQ transformations

2. Getting the data for each invoice with a separate LINQ query that will be re-executed when the cursor moves to another record in the master grid

Either way has its advantages and drawbacks.

The first one assumes that we want to retrieve all invoices at once for the specified period together with the bound data from the invoice lines when the invoice form is opened. Although it is done with one SQL query, it may take quite a while and requires a large amount of random-access memory. It is better suited to web applications where records are usually displayed page by page.

The second one is a bit more difficult to implement, but it allows the invoice form to be opened quickly and requires less resource. However, each time the cursor in the master grid moves, an SQL query will be executed, generating network traffic, albeit with only a small volume of data.

For our application we will use the second approach. We need an event handler for the BindingSource component for editing the current record:

```
private void masterBinding_CurrentChanged(object sender, EventArgs e) {
    LoadInvoiceLineData(this.CurrentInvoice.Id);
    detailGridView.DataSource = detailBinding;
}
```

Now, the method for loading the invoice data:

```
private void LoadInvoiceLineData(int? id) {
  var dbContext = AppVariables.getDbContext();
  var lines =
      from line in dbContext.INVOICE_LINES
      where line.INVOICE_ID == id
      select new InvoiceLineView
      {
          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 = Math.Round(line.QUANTITY * line.SALE_PRICE, 2)
      };
  detailBinding.DataSource = lines.ToBindingList();
}
```

We use the InvoiceLineView class as an extension:

```
public class InvoiceLineView {
    public int Id { get; set; }
    public int Invoice_Id { get; set; }
    public int Product_Id { get; set; }
    public string Product { get; set; }
    public decimal Quantity { get; set; }
    public decimal Price { get; set; }
    public decimal Total { get; set; }
}
```

> **Note**
>
> Unlike the InvoiceView class, this one has no method for loading one current record. In our example, the speed of reloading the detail grid it is not crucial, because one document does not contain thousands of items. Implementing this method is optional.

Now we will add a special property for retrieving the current line of the document selected in the detail grid.

```
public InvoiceLineView CurrentInvoiceLine {
    get {
        return (InvoiceLineView)detailBinding.Current;
    }
}
```

Working with Stored Procedures

The methods we will use for adding, editing and deleting illustrate how to work with stored procedures in Entity Framework. As an example, this is the method for adding a new record:

```
private void btnAddInvoiceLine_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  // get current invoice
  var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
  if (invoice.PAYED == 1) {
    MessageBox.Show("The change is not possible, the invoice has already been paid.", "Erro
    return;
  }
  // create invoice position
  var invoiceLine = dbContext.INVOICE_LINES.Create();
  invoiceLine.INVOICE_ID = invoice.INVOICE_ID;
  // create the position editor of the invoice
```

```
  using (InvoiceLineEditorForm editor = new InvoiceLineEditorForm()) {
    editor.Text = "Add invoice line";
    editor.InvoiceLine = invoiceLine;
    // Form Close Handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
      if (editor.DialogResult == DialogResult.OK) {
        try {
          // create SP parameters
          var invoiceIdParam = new FbParameter("INVOICE_ID",
                                               FbDbType.Integer);
          var productIdParam = new FbParameter("PRODUCT_ID",
                                               FbDbType.Integer);
          var quantityParam = new FbParameter("QUANTITY", FbDbType.Integer);
          // initial parameters values
          invoiceIdParam.Value = invoiceLine.INVOICE_ID;
          productIdParam.Value = invoiceLine.PRODUCT_ID;
          quantityParam.Value = invoiceLine.QUANTITY;
          // execute stored procedure
          dbContext.Database.ExecuteSqlCommand(
            "EXECUTE PROCEDURE SP_ADD_INVOICE_LINE("
          + "@INVOICE_ID, @PRODUCT_ID, @QUANTITY)",
             invoiceIdParam,
             productIdParam,
             quantityParam);
          // refresh grids
          // reload current invoice record
          CurrentInvoice.Load(invoice.INVOICE_ID);
          // reload all record in detail grid
          LoadInvoiceLineData(invoice.INVOICE_ID);
          // refresh all related data
          masterBinding.ResetCurrentItem();
        }
        catch (Exception ex) {
          // display error
          MessageBox.Show(ex.Message, "Error");
          // Do not close the form to correct the error
          fe.Cancel = true;
        }
      }
    };
    editor.ShowDialog(this);
  }
}
```

With our example, an update of the master grid record will be needed because one of its fields (TotalSale) contains aggregated information derived from the detail lines of the document. This is how we do that:

```
private void btnEditInvoiceLine_Click(object sender, EventArgs e) {
  var dbContext = AppVariables.getDbContext();
  // get current invoice
  var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
  if (invoice.PAYED == 1) {
    MessageBox.Show("The change is not possible, the invoice has already been paid.",
                    "Error");
    return;
  }
```

```
  // get current invoice position
  var invoiceLine = invoice.INVOICE_LINES
         .Where(p => p.INVOICE_LINE_ID == this.CurrentInvoiceLine.Id)
         .First();
  // create invoice position editor
  using (InvoiceLineEditorForm editor = new InvoiceLineEditorForm()) {
    editor.Text = "Edit invoice line";
    editor.InvoiceLine = invoiceLine;
    // form close handler
    editor.FormClosing += delegate (object fSender, FormClosingEventArgs fe) {
      if (editor.DialogResult == DialogResult.OK) {
        try {
          // create parameters
          var idParam = new FbParameter("INVOICE_LINE_ID", FbDbType.Integer);
          var quantityParam = new FbParameter("QUANTITY", FbDbType.Integer);
          // initial parameters values
          idParam.Value = invoiceLine.INVOICE_LINE_ID;
          quantityParam.Value = invoiceLine.QUANTITY;
          // execute stored procedure
          dbContext.Database.ExecuteSqlCommand(
              "EXECUTE PROCEDURE SP_EDIT_INVOICE_LINE("
            + "@INVOICE_LINE_ID, @QUANTITY)",
              idParam,
              quantityParam);
          // refresh grids
          // reload current invoice record
          CurrentInvoice.Load(invoice.INVOICE_ID);
          // reload all records in detail grid
          LoadInvoiceLineData(invoice.INVOICE_ID);
          // refresh all related controls
          masterBinding.ResetCurrentItem();
        }
        catch (Exception ex) {
          // display error
          MessageBox.Show(ex.Message, "Error");
          // Do not close the form to correct the error
          fe.Cancel = true;
        }
      }
    };
    editor.ShowDialog(this);
  }
}
```

Deleting an Invoice Detail Line

The method for deleting a detail record is implemented as follows:

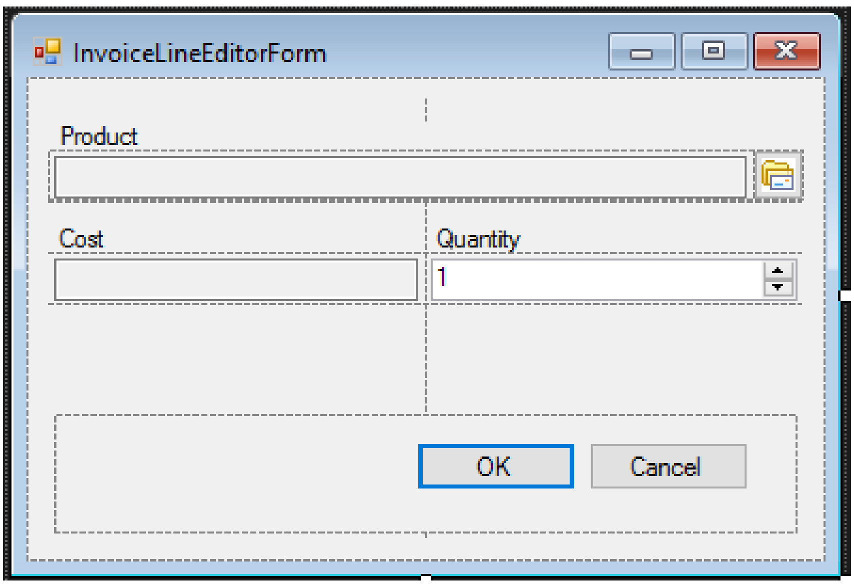
```
private void btnDeleteInvoiceLine_Click(object sender, EventArgs e) {
  var result = MessageBox.Show(
    " Are you sure you want to delete the invoice item?",
    "Confirmation",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question);
  if (result == DialogResult.Yes) {
```

```
    var dbContext = AppVariables.getDbContext();
    // get current invoice
    var invoice = dbContext.INVOICES.Find(this.CurrentInvoice.Id);
    try {
      if (invoice.PAYED == 1)
        throw new Exception("It is not possible to delete the entry, the invoice is paid.");
      // create parameters
      var idParam = new FbParameter("INVOICE_LINE_ID", FbDbType.Integer);
      // initialize parameters values
      idParam.Value = this.CurrentInvoiceLine.Id;
      // execute stored procedure
      dbContext.Database.ExecuteSqlCommand(
        "EXECUTE PROCEDURE SP_DELETE_INVOICE_LINE(@INVOICE_LINE_ID)",
        idParam);
      // update grids
      // reload current invoice
      CurrentInvoice.Load(invoice.INVOICE_ID);
      // reload all records in detail grids
      LoadInvoiceLineData(invoice.INVOICE_ID);
      // refresh related controls
      masterBinding.ResetCurrentItem();
    }
    catch (Exception ex) {
      // display error
      MessageBox.Show(ex.Message, "Error");
    }
  }
}
```

Showing Products for Selection

In the methods for adding and editing invoice lines we used the form. For displaying products, we will use a TextBox control.

Figure 4.17. Product form



A click on the button next to the TextBox will open a modal form with a grid for selecting products. The same modal form created for displaying the products is used for selecting them. The click handler code for the embedded button that initiates the form is:

```
public partial class InvoiceLineEditorForm : Form {
    public InvoiceLineEditorForm() {
        InitializeComponent();
    }

    public INVOICE_LINE InvoiceLine { get; set; }

    private void InvoiceLineEditorForm_Load(object sender, EventArgs e) {
        if (this.InvoiceLine.PRODUCT != null) {
            edtProduct.Text = this.InvoiceLine.PRODUCT.NAME;
            edtPrice.Text = this.InvoiceLine.PRODUCT.PRICE.ToString("F2");
            btnChooseProduct.Click -= this.btnChooseProduct_Click;
        }
        if (this.InvoiceLine.QUANTITY == 0)
            this.InvoiceLine.QUANTITY = 1;
        edtQuantity.DataBindings.Add("Value", this.InvoiceLine, "QUANTITY");
    }

    private void btnChooseProduct_Click(object sender, EventArgs e) {
        GoodsForm goodsForm = new GoodsForm();
        if (goodsForm.ShowDialog() == DialogResult.OK) {
            InvoiceLine.PRODUCT_ID = goodsForm.CurrentProduct.Id;
            edtProduct.Text = goodsForm.CurrentProduct.Name;
            edtPrice.Text = goodsForm.CurrentProduct.Price.ToString("F2");
        }
    }
}
```

Working with Transactions

Whenever we call the *SaveChanges()* method while adding, updating or deleting, Entity Framework starts and ends an implicit transaction. Since we use disconnected data access, all operations are carried out within one transaction. Entity Framework starts and ends a transaction automatically for each data retrieval. We will take the following example to illustrate how automatic transactions work.

Suppose we need to make a discount on goods selected in the grid. Without explicit transaction management, the code would be as follows:

```
var dbContext = AppVariables.getDbContext();
foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
    int id = (int)gridRows.Cells["Id"].Value;
    // here there is an implicit start and the completion of the transaction
    var product = dbContext.PRODUCTS.Find(id);
    // discount 10%
    decimal discount = 10.0m;
    product.PRICE = product.PRICE * (100 - discount) /100;
}
```

```
// here there is an implicit start and the completion of the transaction
// all changes occur in one transaction
dbContext.SaveChanges();
```

Let's say we select 10 products. Ten implicit transactions will be used for finding the products by their identifiers. One more transaction will be used to save the changes.

If we control transactions explicitly, we can use just one transaction for the same piece of work. For example:

```
var dbContext = AppVariables.getDbContext();
// explicit start of a default transaction
using (var dbTransaction = dbContext.Database.BeginTransaction()) {
  string sql =
    "UPDATE PRODUCT " +
    "SET PRICE = PRICE * ROUND((100 - @DISCOUNT)/100, 2) " +
    "WHERE PRODUCT_ID = @PRODUCT_ID";
  try {
    // create query parameters
    var idParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
    var discountParam = new FbParameter("DISCOUNT", FbDbType.Decimal);
    // create a SQL command to update records
    var sqlCommand = dbContext.Database.Connection.CreateCommand();
    sqlCommand.CommandText = sql;
    // specify which transaction to use
    sqlCommand.Transaction = dbTransaction.UnderlyingTransaction;
    sqlCommand.Parameters.Add(discountParam);
    sqlCommand.Parameters.Add(idParam);
    // prepare query
    sqlCommand.Prepare();
    // for all selected records in the grid
    foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
      int id = (int)gridRows.Cells["Id"].Value;
      // initialize query parameters
      idParam.Value = id;
      discountParam.Value = 10.0m; // discount 10%
      // execute sql statement
      sqlCommand.ExecuteNonQuery();
    }
    dbTransaction.Commit();
  }
  catch (Exception ex) {
    dbTransaction.Rollback();
    MessageBox.Show(ex.Message, "error");
  }
}
```

Our code starts the transaction with the default parameters. To specify your own parameters for a transaction, you should use the *UseTransaction* method.

```
private void btnDiscount_Click(object sender, EventArgs e) {
  DiscountEditorForm editor = new DiscountEditorForm();
  editor.Text = "Enter discount";
  if (editor.ShowDialog() != DialogResult.OK)
    return;
```

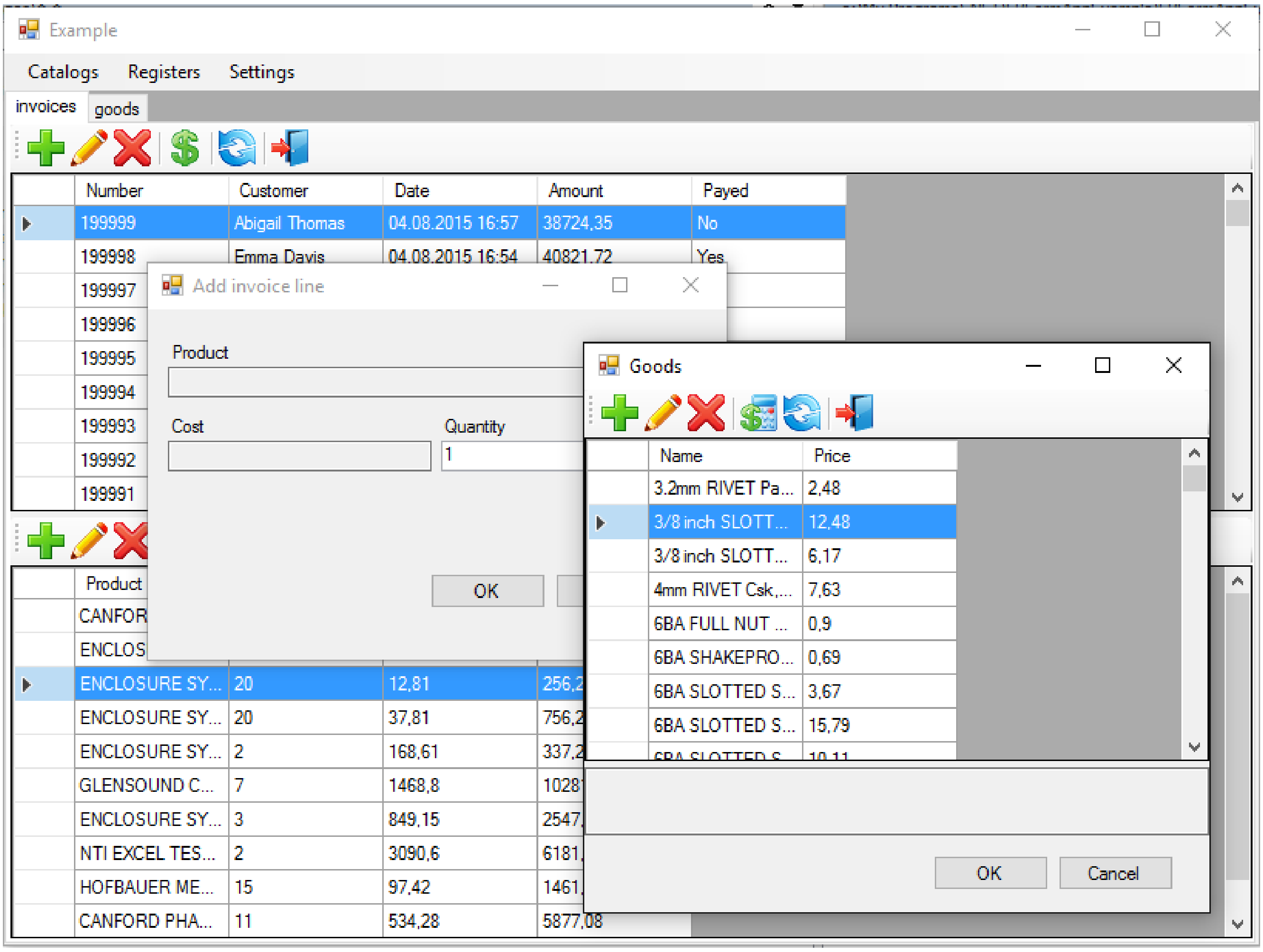
```
  bool needUpdate = false;
  var dbContext = AppVariables.getDbContext();
  var connection = dbContext.Database.Connection;
  // explicit start of transaction
  using (var dbTransaction = connection.BeginTransaction(IsolationLevel.Snapshot)) {
    dbContext.Database.UseTransaction(dbTransaction);
    string sql =
      "UPDATE PRODUCT " +
      "SET PRICE = ROUND(PRICE * (100 - @DISCOUNT)/100, 2) " +
      "WHERE PRODUCT_ID = @PRODUCT_ID";
    try {
      // create query parameters
      var idParam = new FbParameter("PRODUCT_ID", FbDbType.Integer);
      var discountParam = new FbParameter("DISCOUNT", FbDbType.Decimal);
      // create a SQL command to update records
      var sqlCommand = connection.CreateCommand();
      sqlCommand.CommandText = sql;
      // specify which transaction to use
      sqlCommand.Transaction = dbTransaction;
      sqlCommand.Parameters.Add(discountParam);
      sqlCommand.Parameters.Add(idParam);
      // prepare statement
      sqlCommand.Prepare();
      // for all selected records in the grid
      foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
        int id = (int)gridRows.Cells["PRODUCT_ID"].Value;
        // initialize query parameters
        idParam.Value = id;
        discountParam.Value = editor.Discount;
        // execute SQL statement
        needUpdate = (sqlCommand.ExecuteNonQuery() > 0) || needUpdate;
      }
      dbTransaction.Commit();
    }
    catch (Exception ex) {
      dbTransaction.Rollback();
      MessageBox.Show(ex.Message, "error");
      needUpdate = false;
    }
  }
  // refresh grid
  if (needUpdate) {
    // for all selected records in the grid
    foreach (DataGridViewRow gridRows in dataGridView.SelectedRows) {
      var product = (PRODUCT)bindingSource.List[gridRows.Index];
      dbContext.Refresh(RefreshMode.StoreWins, product);
    }
    bindingSource.ResetBindings(false);
  }
}
```

That's it. Now only one transaction is used for the entire set of updates and there are no unnecessary commands for finding data.

All that is left to do is to add a dialog box for entering the value of the discount and code to update data in the grid. Try to do it on your own.

The Result

Figure 4.18. The result of the Entity Framework project



Source Code

You can get the source code for the sample application using this link: https://github.com/sim1984/FB-FormAppExample.

Chapter 5

Creating Web Applications in Entity Framework with MVC

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.

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 Microsoft Entity Framework.

The .NET Frameworks

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. As I prefer using the MVC pattern, it is this technology that we will be examining.

The ASP.NET MVC Platform

The ASP.NET MVC platform is a framework for creating websites and web applications on the model-view-controller (MVC) pattern. The concept underlying 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 string values in a query and send them to the model, which can use these values to send a query to the database.

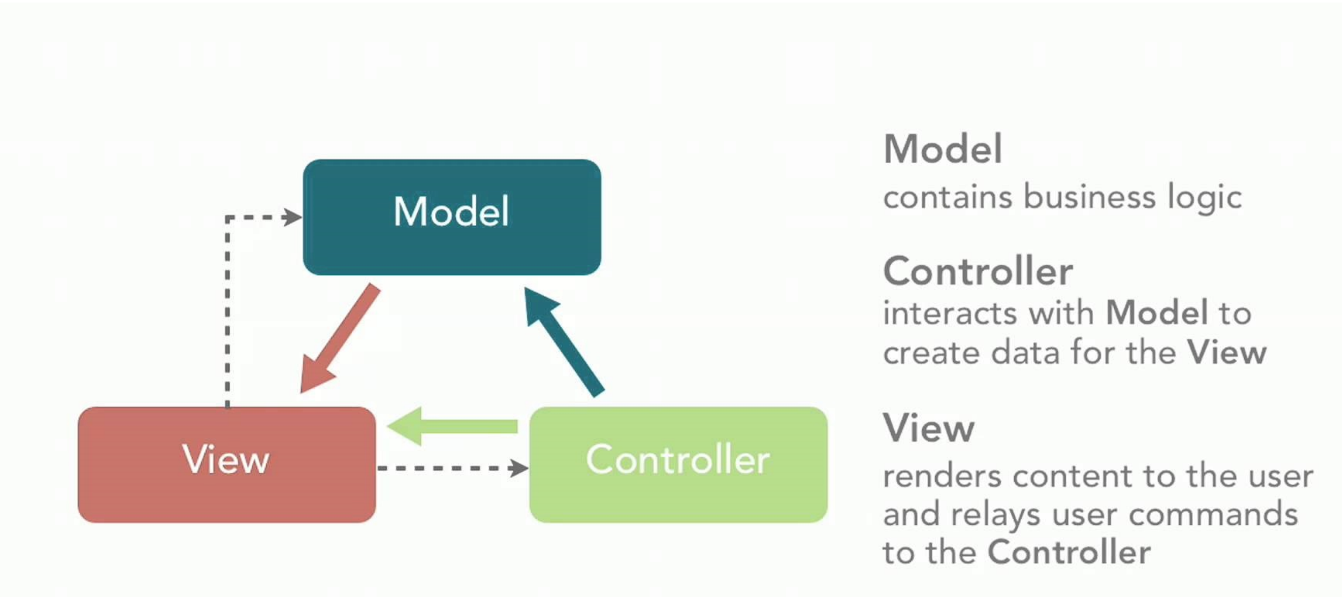
View
: the visual part of application's user interface. The user interface is usually created to reflect the data from the model.

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.

Model-View-Controller Interaction

Interaction between these components is illustrated in the following general diagram:

Figure 5.1. Interaction between M-V-C parts



The MVC pattern supports 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 in 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.

Software Stack

Along with the libraries for working with Firebird, Entity Framework and MVC.NET, you 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 Firebird Work

Some essential steps are needed before you can start working in Visual Studio with Firebird. The preparation process is described in detail in the previous chapter, under the topic Setting Up for Firebird in Visual Studio 2015.

Creating a Project

The Following topics will show how to 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*.

Figure 5.2. Create the FBMVCExample project

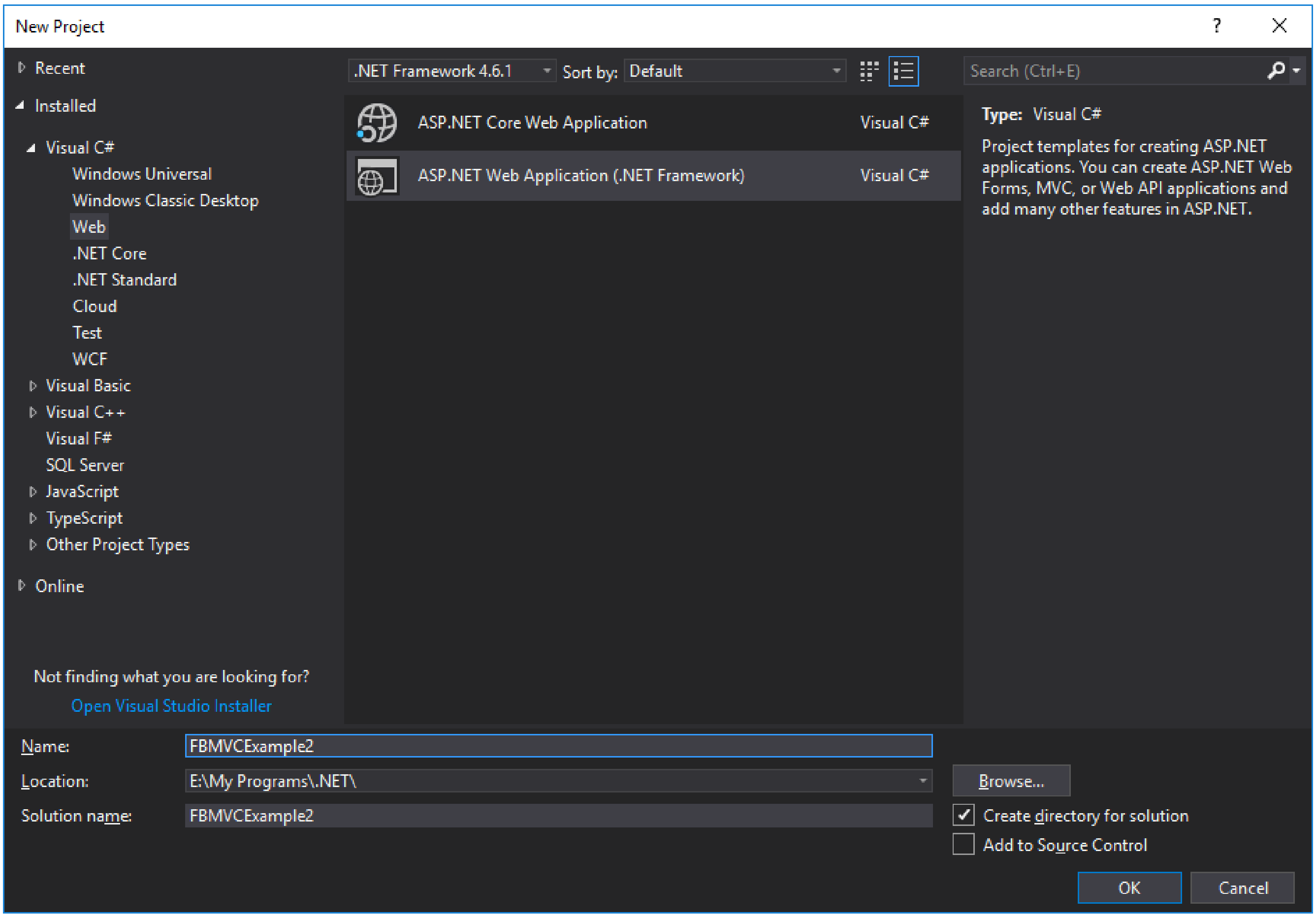
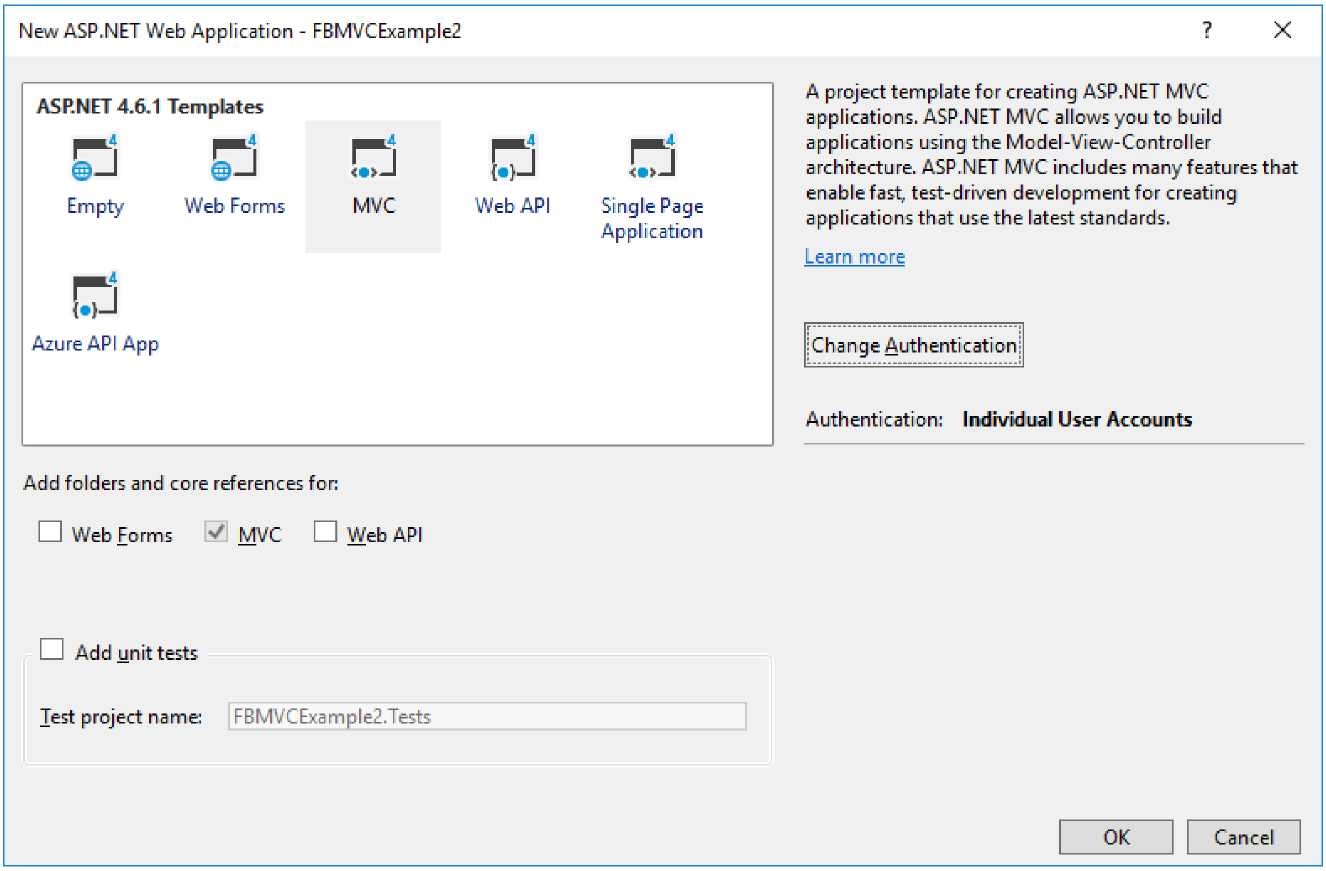
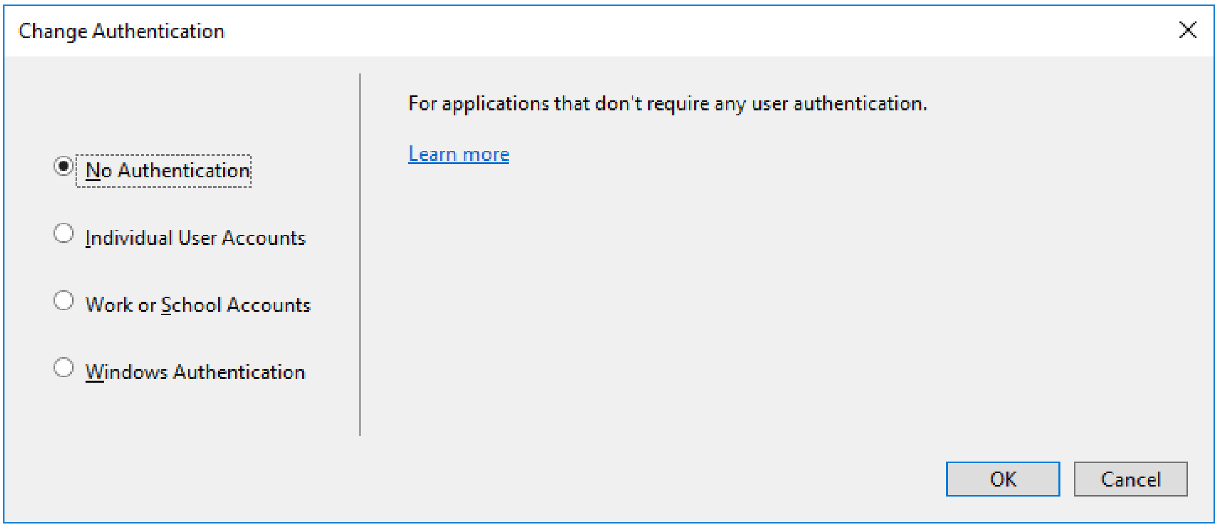


Figure 5.3. Change authentication setting



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

Figure 5.4. Disable authentication for now



Structure of the Project

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

Table 5.1. Basic Structure of the MVC Project

Folder or File	Purpose
/App_Data folder	Where private web application data, such as XML files or database files, are located.
/App_Start folder	Contains some main configuration settings for the project, including the definitions of routes and filters.
/Content folder	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 folder	Controller classes are saved here. It is an optional convention. You can store controller classes anywhere.
/Models folder	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 folder	Stores 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 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 subfolder	Stores layouts and views not specific to one controller.
/Views/Web.config file	Contains 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 file	The global class of an ASP.NET application. A configuration for a route is registered in the file with its code (Global.asax.cs). Also 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 file	The configuration file for the application.

Adding the Missing Packages

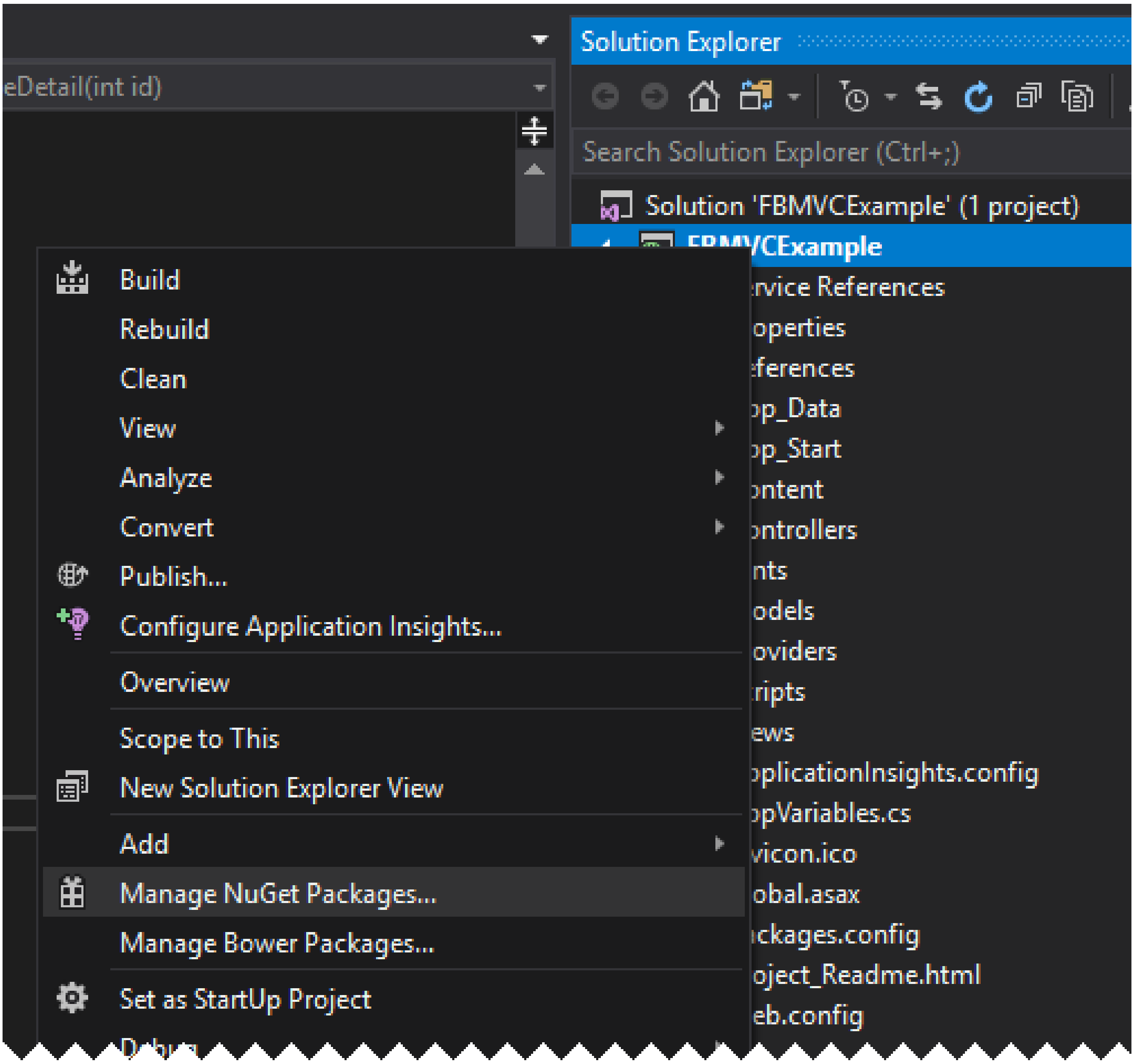
We will use the NuGet package manager to add the missing packages:

- 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.

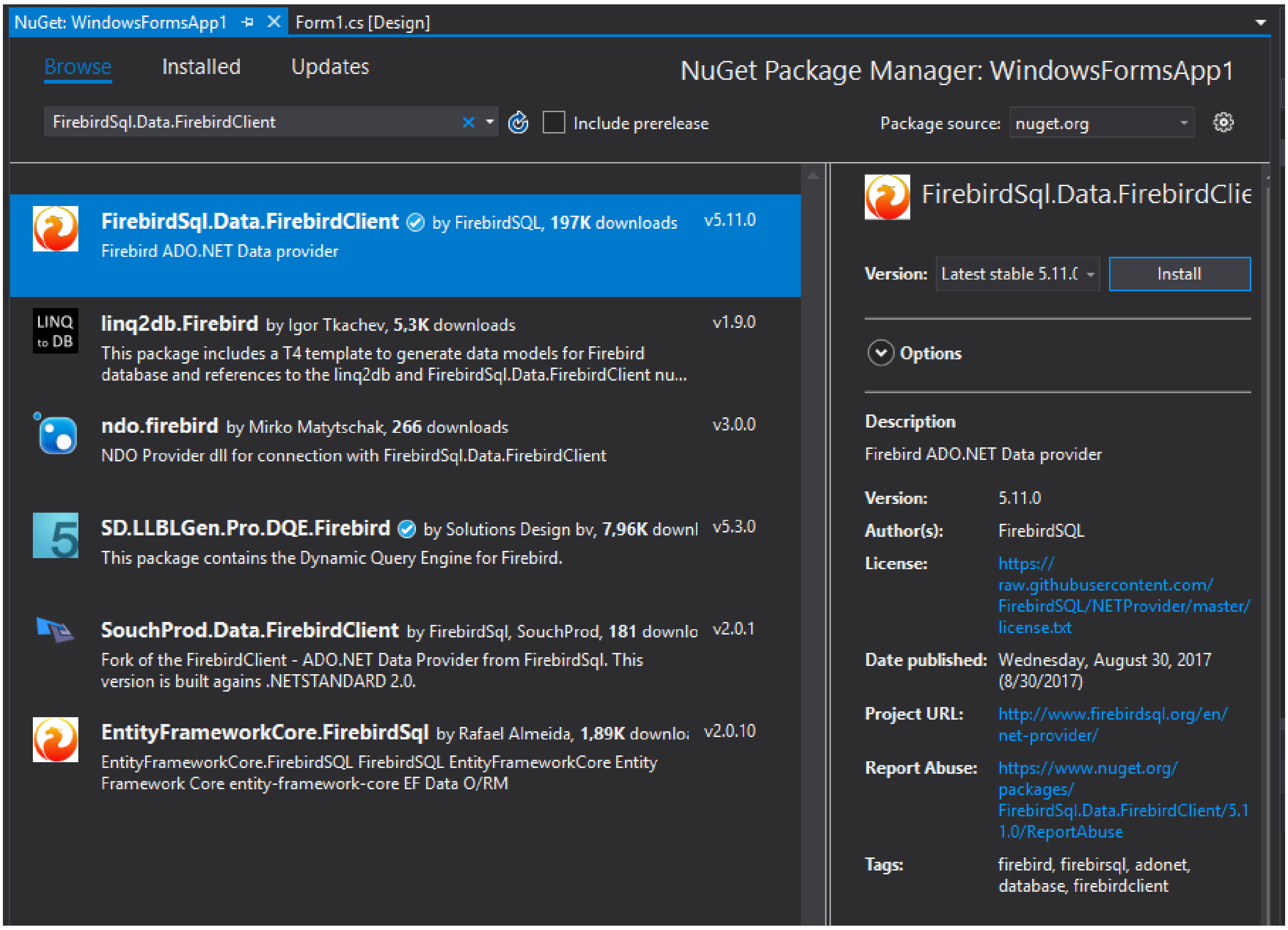
Right-click the project name in Solution Explorer and select the Manage NuGet Packages item in the drop-down menu.

Figure 5.5. Select Manage NuGet Packages



Find and install the necessary packages in the package manager.

Figure 5.6. Select packages for installing

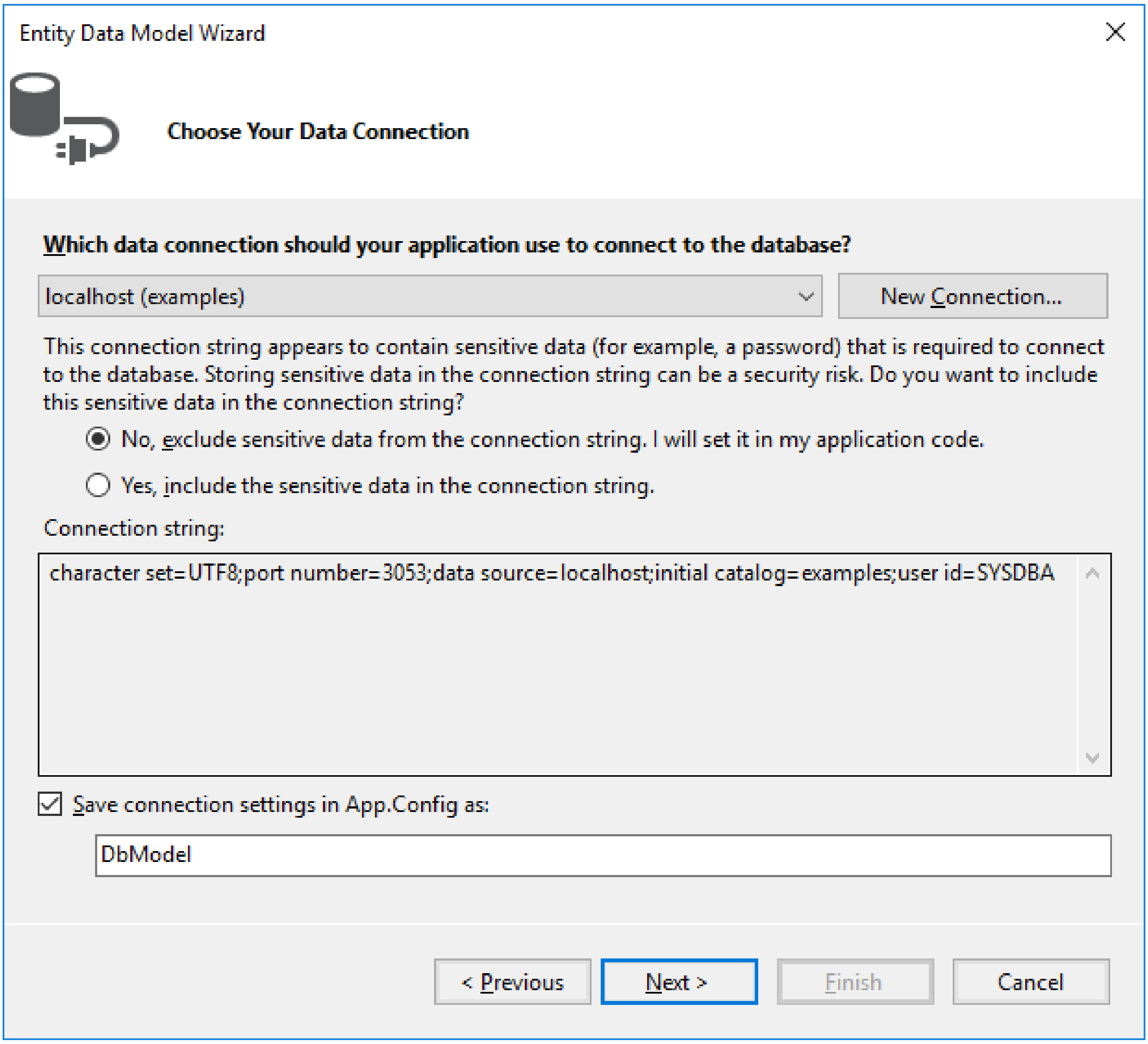


Creating an EDM

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 in the topic Creating an Entity Data Model (EDM).

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

Figure 5.7. Configuring connection string storage



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" />
```

Creating a User Interface

Our first controller will be used to display customer data and accept input for searches, inserts, edits and deletes.

Creating the Controller for the Customer Interface

Figure 5.8. Select Add—>Controller

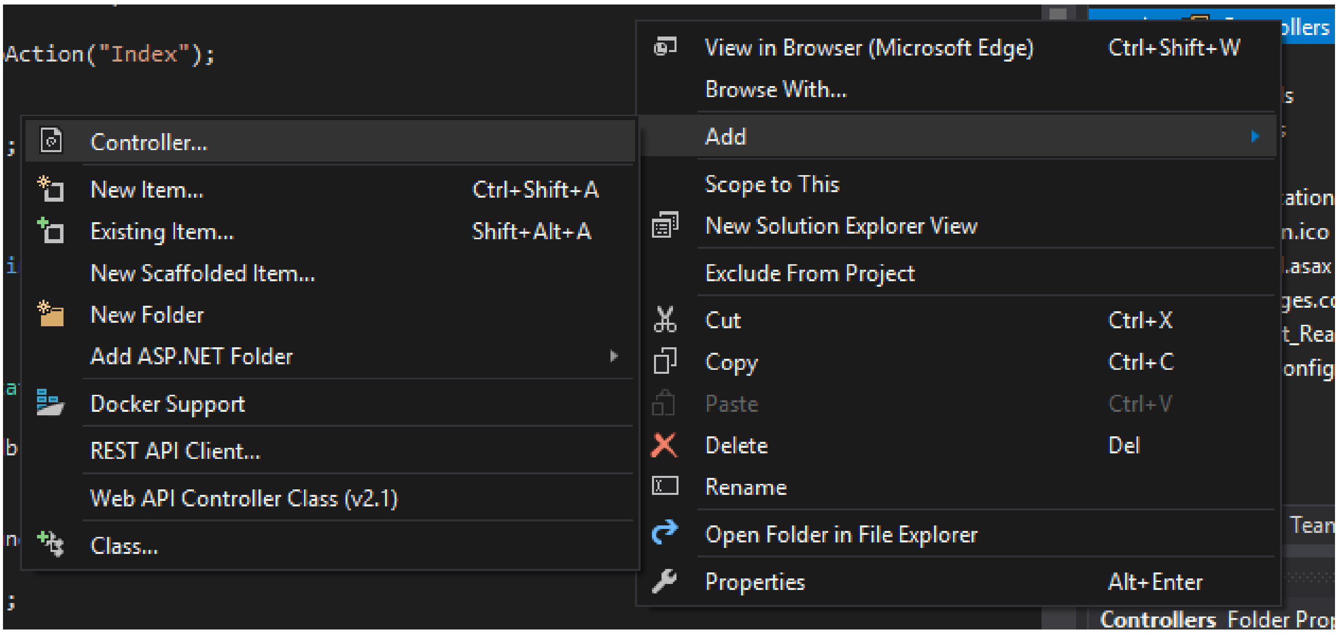


Figure 5.9. Creating a controller (1)

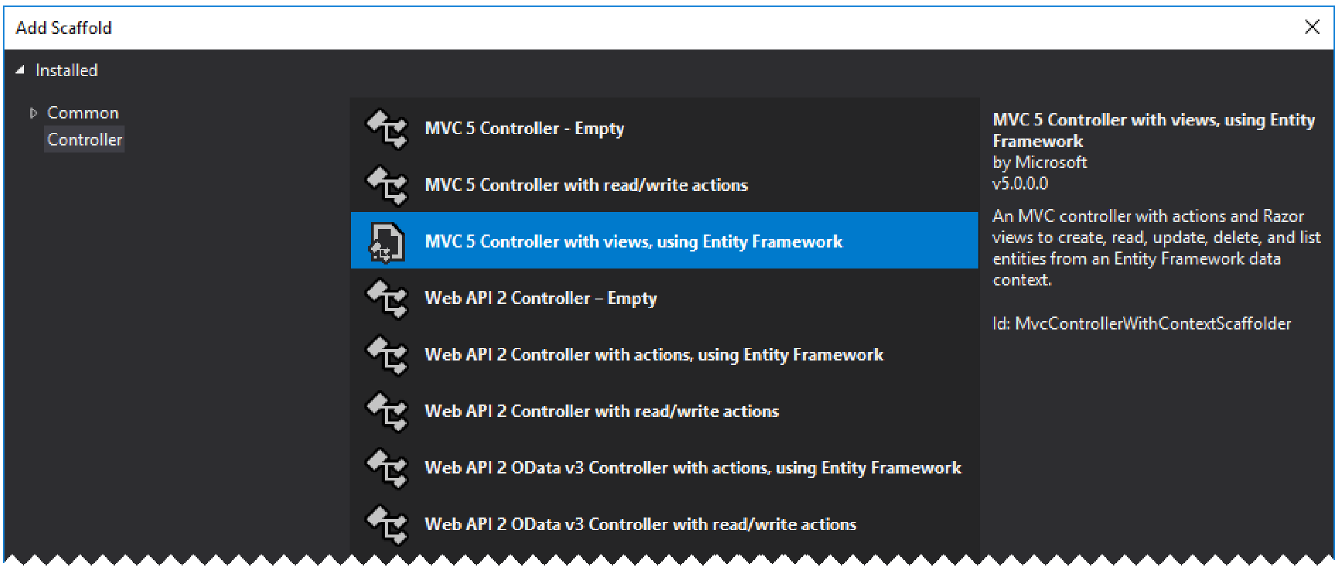
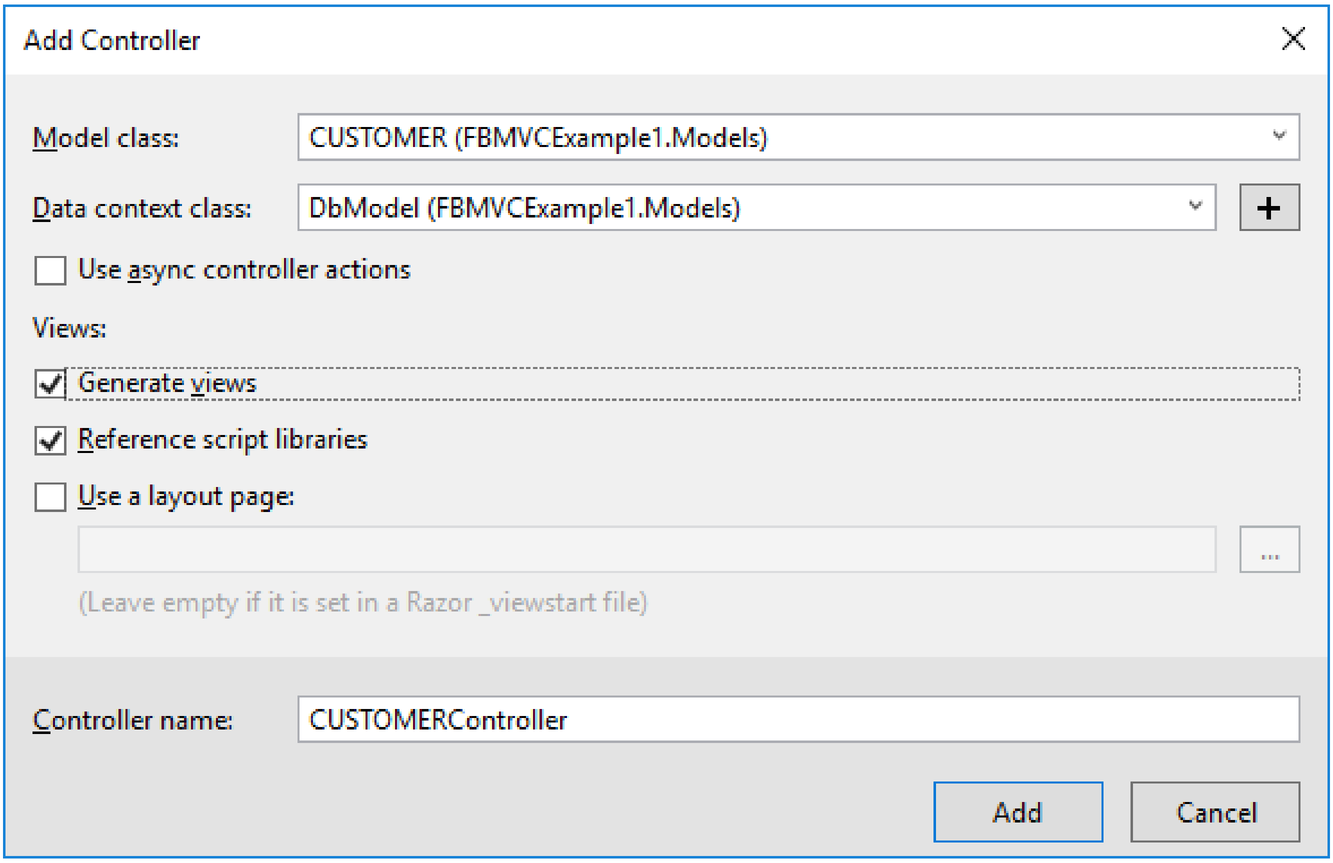


Figure 5.10. Creating a controller (2)



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

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

Since the Ajax technology and the jqGrid library will be used extensively in our project, the first 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.

Limiting Overhead

We want to be aware of ways to limit the overhead involved in passing data and connections back and forth over the wide-area network. There are techniques that can help us with this.

Limiting Returned Data

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 project.

Limiting Connections

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 to make a connection to the database only when it is really necessary.

Let the Browser Help You!

One of the ways to reduce the amount of interaction with the database is to do the correctness checking on the user input in the browser. Fortunately, modern HTML5 and JavaScript libraries can do just that. For example, you can make the browser check for the presence of a required field or the maximum length of a string field in the input form.

Adapting the Controller to jqGrid

Now, we are going to 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/Customer/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 method has the `[HttpPost]` attribute specified for it to indicate that the parameters of the `HTTP POST request ()` are to be displayed on the Customer model. Examine 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.

The Attribute ValidateAntiforgeryToken

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 rather than previously created web forms. To fix that, 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"7gt;</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 using 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 handy 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 fills out the rest of the string with the asterisk character as a wildcard. For example, the expression will include two files at once in the bundle: `jquery.validate.js` and `jquery.validate.unobtrusive.js`, along with their minimized versions, because their names both start with "`jquery.validate`".

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

> **Important**
>
> The full versions of the scripts and cascading style sheets should be used 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.

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.

```
@{
  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>
```

It is important to configure the model properties correctly in order to display the grid properly, 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. In the comments I have tried to describe the parameters being used. The full description of the model parameters can be found in the documentation for the jqGrid library in the ColModel API 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 list of customers will look like this:

Figure 5.11. Customer list view

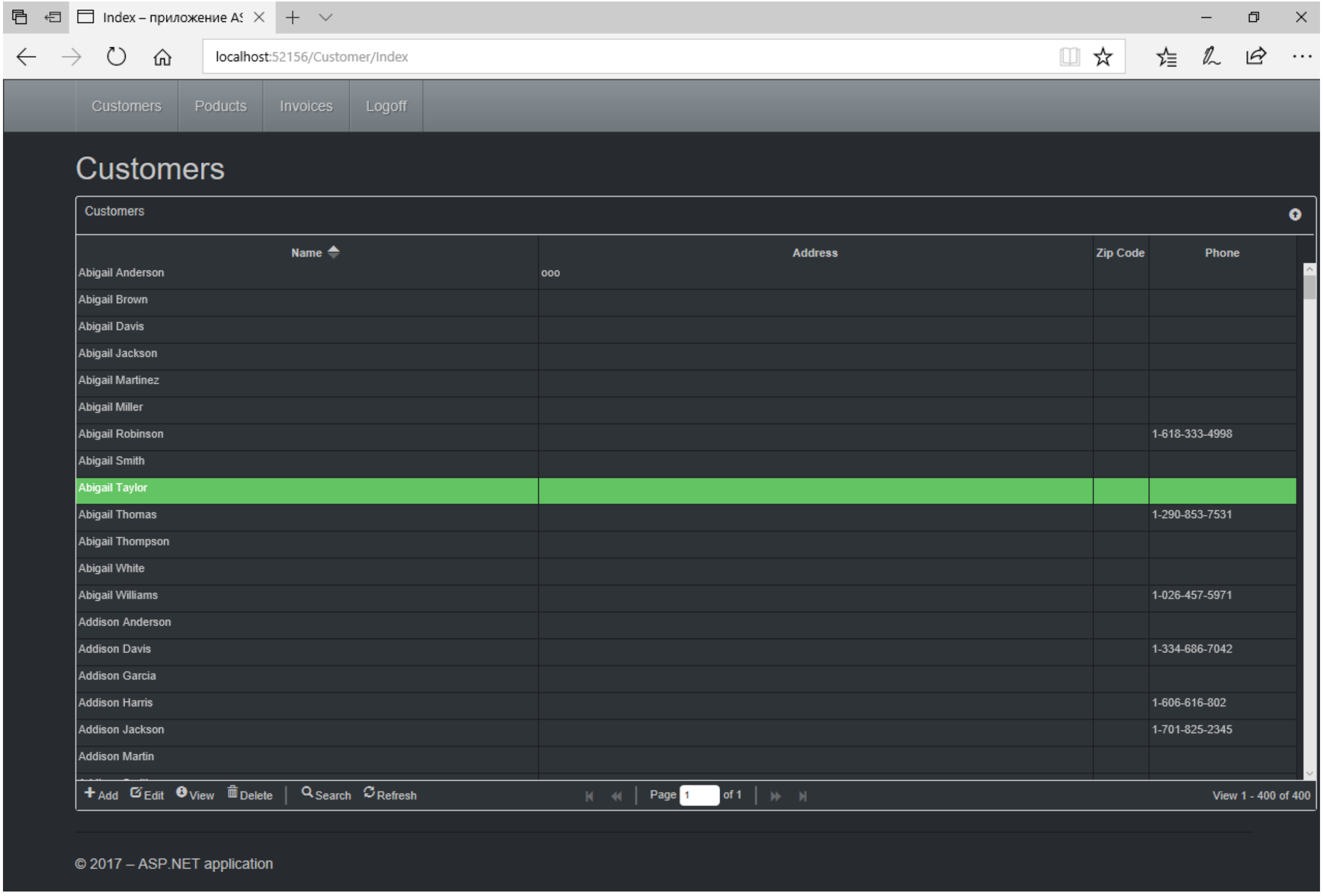
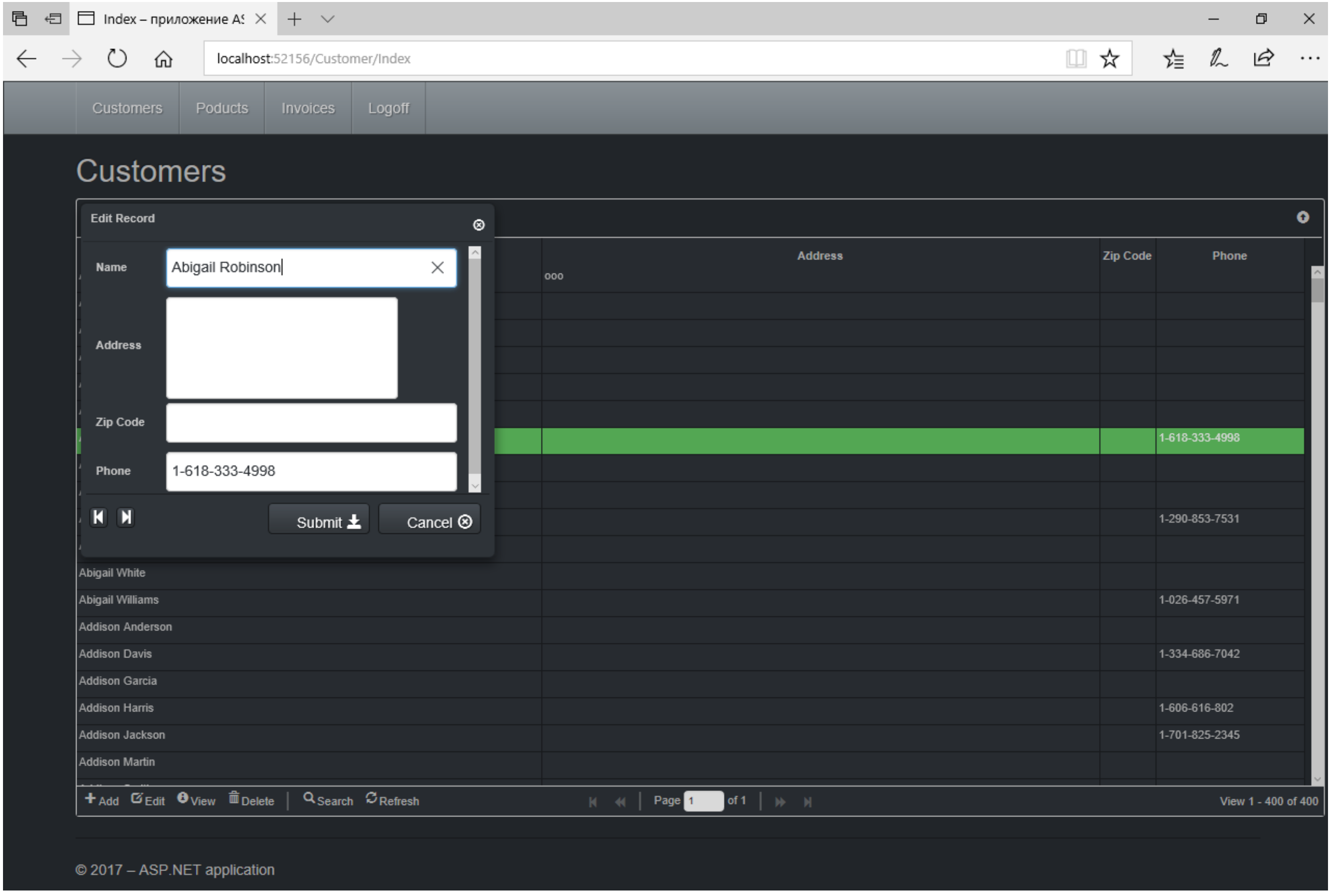


Figure 5.12. A customer selected for editing



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 linked at the end of this chapter.

Creating a UI 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, our detail grid will be embedded in the main one.

Controllers for Invoices

The controller of the invoice module must be able to return data for both invoice headers and the associated 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 project, all data modification operations are performed in stored procedures, but you could do the same work using Entity Framework. DDL code for the stored procedures can be found in the database creation script in an earlier chapter and also in the .zip archives of all the DDL scripts:

https://github.com/sim1984/example-db_2_5/archive/1.0.zip
or https://github.com/sim1984/example-db_3_0/archive/1.0.zip

Views for Invoices

As 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');
          }
        }
      });
    }
  }
});
```

Dialog Boxes for Invoices

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 that invokes 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 that 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.

Loading data for the lines requires passing the primary key from 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 += " ???????</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 list 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, et al.

The description of the ASP.NET Identity technology is quite comprehensive and goes beyond the scope of this publication but you can read about it at http://www.asp.net/identity.

For our application, we will take a less complicated approach based on form authentication. Enabling form authentication entails some 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, regardless 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 for this setting.
- *UseDeviceProfile*—if the device supports cookie sets, they are used, otherwise, they are not used; no detection test is run. Used by default.

defaultUrl
: specifies the URL to redirect to after authentication

domain
: specifies cookie sets for the entire domain, allowing for the same cookie sets to be used for the main domain and its sub-domains. 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 to avoid the need for an additional connection string.

Infrastructure for Authentication

Now we need to create all the infrastructure required for authentication—models, controllers and views. The *WebUser* model describes 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 b
[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 Db
…
  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 here is a 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, rather than the actual password, is stored in an open form, using the md5 algorithm, for example. For our example, we have simplified authentication somewhat.

Our code 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>???????????</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.

Adding a New User

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 proprty *HttpContext.User* 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 property *IIdentity.IsAuthenticated* is set to true and the *Name* property gets the username as its value.

Next, we will add authentication items using the *universal providers* mechanism.

Universal Providers

Universal providers offer a ready-made authentication functionality. 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. That is handy if we do not need all of the fancy ASP.NET Identity features, but just a very simple authentication system.

So, our next step is to redefine the role provider. To do this, we need to add the Microsoft.AspNet.Providers package using NuGet.

Defining the Role Provider

To define the role provider, 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:

- GetRolesForUser—for obtaining a set of roles for a specified user
- CreateRole—for creating a role
- IsUserInRole—determines whether the user has a specified role in the system

Configuring the Role Provider for Use

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>
```

Authorizing Access to Controller Methods

Now we can limit (filter) 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 project, 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.

Source Code

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

Chapter 6

Developing Web Applications with PHP and Firebird

In this chapter, we are going to create a web application using the PHP language with Firebird as the back-end. It is assumed that you have a web server, such as Apache HTTP Server or Nginx with PHP installed and active and a Firebird server available in the stack and running. A lightweight, stand-alone package such as QuickPHP would be enough for testing and debugging your project locally.

If your server supports PHP, you just create your .php files, put them in your web directory and the server will automatically parse them for you. PHP-enabled files are simply HTML files with a whole language of custom tags embedded in them. There is nothing to compile.

Interfacing PHP and Firebird

To communicate with a Firebird database, you will need a driver.

PHP Drivers for Firebird

Two free, downloadable drivers are available for interfacing with Firebird:

- The Firebird/Interbase extension (`ibase_` functions)
- The PDO driver for Firebird

Firebird Client Library

Both drivers require that you have the `fbclient.dll` client library installed (`fbclient.so` for POSIX systems). Make sure it is for the correct CPU register width (32-bit or 64-bit) to match that of your web server/PHP installation. For example, if you have a 64-bit machine running 64-bit Firebird and 32-bit Apache/PHP then you need the 32-bit driver.

> **Note to Win32/Win64 users**
>
> For the drivers to work with the Windows PATH system variable, the `fbclient.dll` DLL file must be available. Copying the DLL file from the PHP directory or a Firebird installation to the Windows system folder would work, because the system directory is in the PATH variable by default. However, it is not recommended. The more robust way to do it is to prepend the file path to the PATH variable explicitly yourself, using the Windows advanced administration tool.

Make sure you have the matching release version of the Firebird client for your Firebird server.

The Firebird/InterBase Extension

The Firebird/Interbase ("Fb/IB") extension predates the PDO driver and is regarded as the more proven solution of the two.

To install the extension, uncomment this line in the php.ini configuration file:

```
extension=php_interbase.dll
```

or this line on Linux and other POSIX systems:

```
extension=php_interbase.so
```

Installing the Fb/IB Extension on Linux

In Linux, one of the following commands should work. The one you use depends on the distribution package and the versions it supports:

```
apt-get install php5-firebird

rpm -ihv php5-firebird

yum install php70w-interbase

zypper install php5-firebird
```

> **Tip**
>
> You might need to enable third party repositories if you find you have unresolvable dependency problems.

Programming Style

The Firebird/InterBase extension uses a procedural approach to developing programs. Functions with the *ibase_* prefix can return or accept the identifier (ID) of a connection, transaction, prepared query or cursor (the result of the SELECT query) as one of their parameters. This identifier is a server-allocated resource which, like all allocated resources, should be released immediately it is no longer needed.

The PHP functions will not be described in detail here. You can study their descriptions at http://php.net/ibase. Several small examples with comments will be provided instead.

```
<?php
$db = 'localhost:example';
$username = 'SYSDBA';
$password = 'masterkey';
// Connect to database
$dbh = ibase_connect($db, $username, $password);
```

```
$sql = 'SELECT login, email FROM users';
// Execute query
$rc = ibase_query($dbh, $sql);
// Get the result row by row as object
while ($row = ibase_fetch_object($rc)) {
  echo $row->email, "\n";
}
// Release the handle associated with the result of the query
ibase_free_result($rc);
// Release the handle associated with the connection
ibase_close($dbh);
```

The ibase_ Connect Functions

The *ibase_pconnect* function, that creates so-called “persistent connections”, could be used instead of *ibase_connect*. A call to *ibase_close* on this style of connection does not close it but all resources allocated to it will be released. The default transaction is committed, while any others are rolled back. This type of connection can be re-used in another session if the connection parameters match.

Persistent connections can increase the performance of a web application, sometimes considerably. It is especially noticeable if establishing a connection involves a lot of traffic. They allow a child process to use the same connection throughout its entire lifetime instead of creating a connection every time a page interacts with the Firebird server. Persistent connections are not unlike working with a connection pool.

You can find more details about persistent connections at http://php.net/persistent-connections.

> **Need to know**
>
> Many ibase_ functions cannot accommodate the identifier of a connection, transaction or prepared query. Those functions use the identifier of the last established connection or last started transaction instead of the relevant identifier. Ii is not a recommended practice, especially if your web application can use more than one connection.

ibase_query

The ibase_query function executes an SQL query and returns the identifier of the result or True if the query returns no data set. Along with the connection or transaction ID and the text of the SQL query, this function can accept a variable number of parameters to populate the SQL query parameters. For example,

```
// …
$sql = 'SELECT login, email FROM users WHERE id=?';
$id = 1;
// Execute query
$rc = ibase_query($dbh, $sql, $id);
// Get the result row by row as object
if ($row = ibase_fetch_object($rc)) {
  echo $row->email, "\n";
}
// Release the handle associated with the result of the query
ibase_free_result($rc);
// …
```

Parameterized queries are typically used multiple times with fresh sets of parameter values each time. Prepared queries are recommended for this style of usage. The identifier of a query is returned by the function *ibase_prepare* and then the prepared query is executed using the function ibase_execute.

```
// …
$sql = 'SELECT login, email FROM users WHERE id=?';
// Prepare statement
$sth = ibase_prepare($dbh, $sql);
$id = 1;
// Execute statement
$rc = ibase_execute($sth, $id);
// Get the result row by row as object
if ($row = ibase_fetch_object($rc)) {
  echo $row->email, "\n";
}
// Release the handle associated with the result of the query
ibase_free_result($rc);
// Release the prepared statement
ibase_free_query($sth);
```

Prepared queries are very often used when a large amount of data input is anticipated.

```
// …
$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
// Prepare statement
$sth = ibase_prepare($dbh, $sql);
$users = [["user1", "user1@gmail.com"], ["user2", "user2@gmail.com"]];
// Execute statement
foreach ($users as $user)) {
  ibase_execute($sth, $user[0], $user[1]);
}
// Release the prepared statement
ibase_free_query($sth);
// …
```

It is actually a disadvantage of this extension that functions can take a variable number of parameters. It less than ideal for parameterized queries, as the last example demonstrates. It is especially noticeable if you try to write a universal class for executing any query. It would be much more useful to be able to send parameters in one array.

This would be one way to get around it:

```
function fb_execute ($stmt, $data)
{
  if (!is_array($data))
    return ibase_execute($stmt, $data);
  array_unshift($data, $stmt);
  $rc = call_user_func_array('ibase_execute', $data);
  return $rc;
}
```

The Fb/IB extension does not support named parameters in queries.

ibase_trans

By default, the Fb/IB extension commits the transaction automatically after executing each SQL query, making it necessary to start a transaction with the function *ibase_trans* if you need to control transactions explicitly. An explicit transaction is started with the following parameters if none are provided: IBASE_WRITE | IBASE_CONCURRENCY | IBASE_WAIT. You can find the description of predefined constants for specifying the parameters of a transaction here. A transaction must be completed by either *ibase_commit* or *ibase_rollback*.

This extension supports the COMMIT RETAIN and ROLLBACK RETAIN parameters directly if you use the functions *ibase_commit*_ret or *ibase_rollback_ret*, respectively, instead.

> **Note**
>
> The default transaction parameters are good for most cases and it is really rarely that you need to change them. A connection to the database, along with all resources allocated to it, exists for no longer than it takes for the PHP script to complete. Even if you use persistent connections, all allocated resources will be released after the *ibase_close* function is called. Even so, I strongly recommend releasing all allocated resources explicitly by calling the corresponding ibase_ functions.
>
> I advise strongly against using the *ibase_commit_ret* and *ibase_rollback_ret* functions because they have no place in a web application. The purpose of COMMIT RETAIN and ROLLBACK RETAIN is to keep cursors open in desktop applications when a transaction ends.

```
$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
// Prepare statement
$sth = ibase_prepare($dbh, $sql);
$users = [["user1", "user1@gmail.com"], ["user2", "user2@gmail.com"]];
$trh = ibase_trans($dbh, IBASE_WRITE | IBASE_CONCURRENCY | IBASE_WAIT);
try {
  // Execute statement
  foreach ($users as $user)) {
    $r = ibase_execute($sth, $user[0], $user[1]);
    // If an error occurs, throw an exception
    if ($r === false)
      throw new \Exception(ibase_errmsg());
  }
  ibase_commit($trh);
}
catch(\Exception $e) {
  ibase_rollback($trh);
  echo $e->getMessage();
}
// Release the prepared statement
ibase_free_query($sth);
```

> **Warning**
>
> ibase_ functions raise no exception if an error occurs, although an error will cause some to return False. Note that it is essential to use the === strict relational operator to compare the result to False. Calling any ibase function could result in an error.
>
> The function *ibase_errmsg* is available to discover an error message and the function ibase_errcode can provide the error code.

Services API Calls

The Fb/IB extension can interact with the Firebird server by way of functions that wrap calls to the Services API: *ibase_service_attach, ibase_service_detach, ibase_server_info, ibase_maintain_db, ibase_db_info, ibase_backup, ibase_restore*. They can return information about the Firebird server, initiate a backup or restore or get statistics. We are not examining them in detail, since they are required mainly to administer a database, a topic that is outside the scope of this project.

Firebird Events

The Firebird/Interbase extension also supports working with Firebird events by means of a set of functions: *ibase_set_event_handler, ibase_free_event_handler, ibase_wait_event*.

The PDO (Firebird Driver)

The PDO extension is a common interface for accessing various types of databases. Each database driver that implements this interface can provide database-specific features in the form of standard extension functions.

PDO and all basic drivers are built into PHP as extensions. To use them, just enable them by editing the php.ini file as follows:

```
extension=php_pdo.dll
```

> **Note**
>
> This step is optional for PHP versions 5.3 and higher because DLLs are no longer needed for PDO to work.

Firebird-specific Library

The other requirement is for database-specific DLLs to be configured; or else loaded during execution by means of the *dl()* function; or else included in `php.ini` following php_pdo.dll. For example:

```
extension=php_pdo.dll
extension=php_pdo_firebird.dll
```

These DLLs must be in the directory `extension_dir`.

In Linux, one of the following commands should work. The one you use depends on the distribution package and the versions it supports:

```
apt-get install php5-firebird

rpm -ihv php5-firebird

yum install php70w-firebird
```

```
zypper install php5-firebird
```

Programming Style

PDO uses an object-oriented approach to developing programs. The DSN (Data Source Name), a.k.a. connection string, determines which specific driver will be used in PDO. The DSN consists of a prefix that determines the database type and a set of parameters in the form of `<key>=<value>` separated by semicolons. The valid set of parameters depends on the database type.

To be able to work with Firebird, the connection string must start with the `firebird:` prefix and conform to the format described in the PDO_FIREBIRD DSN section of the documentation.

Making Connections

Connections are established automatically during creation of the PDO from its abstract class. The class constructor accepts parameters to specify the data source (DSN) and also the optional username and password, if any. A fourth parameter can be used to pass an array of driver-specific connection settings in the `key=value` format.

```
$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Connect to database
  $dbh = new \PDO($dsn, $username, $password,
                  [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  $sql = 'SELECT login, email FROM users';
  // Execute query
  $query = $dbh->query($sql);
  // Get the result row by row as object
  while ($row = $query->fetch(\PDO::FETCH_OBJ)) {
    echo $row->email, "\n";
  }
  $query->closeCursor();
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

> **Persistent connections**
>
> For PDO to use persistent connections, the array of attributes must be passed to the PDO constructor with `PDO::ATTR_PERSISTENT => true`.

Exception Handling

The PDO driver is much more friendly than the Firebird/InterBase extension with respect to exception handling. Setting the *\PDO::ATTR_ERRMODE* attribute to the value *\PDO::ERRMODE_EXCEPTION* specifies a mode in which any error, including a database connection error, will raise the exception *\PDOException*.

This is superior to the laborious procedure of checking whether an error has occurred each time an ibase_ function is called.

Querying

The *query* method executes an SQL query and returns the result set in the form of a *\PDOStatement* object. A fetch to this method can return the result in more than one form: it could be a column, an instance of the specified class, an object.

The various ways of calling query can be found in the documentation.

Queries with No Data Set

For executing an SQL query that returns no data set, you can use the exec method that returns the number of affected rows.

Executing prepared queries is not supported by *exec*.

Parameterized Queries

If there are parameters in the query, prepared queries must be used. For this, the *prepare* method is called instead of the *query* method. The *prepare* method returns an object of the *\PDOStatement* class that encapsulates methods for working with prepared queries and their results. Executing the query requires calling the *execute* method that can accept as its parameter an array of named or unnamed parameters.

The result of executing a SELECT query can be obtained with one the following methods: *fetch, fetchAll, fetchColumn, fetchObject*. The *fetch* and *fetchAll* methods can return results in various forms: an associative array, an object or an instance of a particular class. The class instance option is quite often used in the MVC pattern during work with models.

```
$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Connect to database
  $dbh = new \PDO($dsn, $username, $password,
                  [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  $sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
  $users = [
    ["user1", "user1@gmail.com"],
    ["user2", "user2@gmail.com"]
  ];
  // Prepare statement
  $query = $dbh->prepare($sql);
  // Execute statement
  foreach ($users as $user)) {
    $query->execute($user);
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

An example using named parameters:

```
$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Connect to database
  $dbh = new \PDO($dsn, $username, $password,
                  [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  $sql = 'INSERT INTO users(login, email) VALUES(:login, :email)';
  $users = [
    [":login" => "user1", ":email" => "user1@gmail.com"],
    [":login" => "user2", ":email" => "user2@gmail.com"]
  ];
  // Prepare statement
  $query = $dbh->prepare($sql);
  // Execute statement
  foreach ($users as $user)) {
    $query->execute($user);
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

> **Note**
>
> In order to support named parameters, PDO preprocesses the query and replaces parameters of the :paramname type with "?", retaining the array of correspondence between the parameter names and their left-to-right positions in the query. For that reason, the EXECUTE BLOCK statement will not work if there are colon-prefixed variables. Currently, PDO offers no workaround to support a parameterized EXECUTE BLOCK statement, such as by specifying an alternative prefix for parameters as has been implemented in some access components.

Another Way to Do It

An alternative way to pass parameters to a query is by using “binding”. The *bindValue* method binds a value to a named or unnamed parameter. The *bindParam* method binds a variable to a named or unnamed parameter. The *bindParam* method is especially useful for stored procedures that return a value via the OUT or IN OUT parameter, which is different to the mechanism for returning values from stored procedures in Firebird.

```
$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Connect to database
  $dbh = new \PDO($dsn, $username, $password,
                  [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  $sql = 'INSERT INTO users(login, email) VALUES(:login, :email)';
  $users = [
    ["user1", "user1@gmail.com"],
    ["user2", "user2@gmail.com"]
  ];
  // Prepare statement
  $query = $dbh->prepare($sql);
  // Execute statement
  foreach ($users as $user)) {
    $query->bindValue(":login", $user[0]);
```

```
    $query->bindValue(":email", $user[1]);
    $query->execute();
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

> **Caution**
>
> The numbers associated with unnamed parameters for the bindParam and bindValue methods start from 1.

```
$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Connect to database
  $dbh = new \PDO($dsn, $username, $password,
                  [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  $sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
  $users = [
    ["user1", "user1@gmail.com"],
    ["user2", "user2@gmail.com"]
  ];
  // Prepare statement
  $query = $dbh->prepare($sql);
  // Execute statement
  foreach ($users as $user)) {
    $query->bindValue(1, $user[0]);
    $query->bindValue(2, $user[1]);
    $query->execute();
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

Transactions

By default, PDO commits the transaction automatically after executing each SQL query. If you want to control transactions explicitly, you need to start a transaction with the method *\PDO::beginTransaction*. By default, a transaction is started with the following parameters: CONCURRENCY | WAIT | READ_WRITE. A transaction can be ended with the *\PDO::commit* or *\PDO::rollback* method.

```
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Connect to database
  $dbh = new \PDO($dsn, $username, $password,
                  [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  // Start the transaction to ensure consistency between statements
  $dbh->beginTransaction();
  // Get users from one table
  $users_stmt = $dbh->prepare('SELECT login, email FROM old_users');
```

```
  $users_stmt->execute();
  $users = $users_stmt->fetchAll(\PDO::FETCH_OBJECT);
  $users_stmt->closeCursor();
  // And insert into another table
  $sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
  // Prepapre statemenet
  $query = $dbh->prepare($sql);
  // Execute statememt
  foreach ($users as $user)) {
    $query->bindValue(1, $user->LOGIN);
    $query->bindValue(2, $user->EMAIL]);
    $query->execute();
  }
  // Commit transaction
  $dbh->commit();
} catch (\PDOException $e) {
  // Rollback transaction
  if ($dbh && $dbh->inTransaction())
    $dbh->rollback();
  echo $e->getMessage();
}
```

Changing Transaction Parameters

Unfortunately, the *beginTransaction* method does not permit transaction parameters to be changed, but you can do the trick by specifying transaction parameters in the SQL statement SET TRANSACTION.

```
$dbh = new \PDO($dsn, $username, $password);
$dbh->setAttribute(\PDO::ATTR_AUTOCOMMIT, false);
$dbh->exec("SET TRANSACTION READ ONLY ISOLATION LEVEL READ COMMITTED NO WAIT");
// Perform actions in the transaction
// …
$dbh->exec("COMMIT");
$dbh->setAttribute(\PDO::ATTR_AUTOCOMMIT, true);
```

Comparing the Drivers

The following table summarises the capabilities offered by the two drivers for working with Firebird.

Table 6.1. Comparing the Firebird/InterBase and PDO Drivers

Capability	Fb/IB Extension	PDO Extension
Programming paradigm	Procedural	Object-oriented
Supported database engines	Firebird and InterBase; or clones of either	Any database engine for which there is a PDO driver, including Firebird
Handling query parameters	Only unnamed parameters, not very convenient because the func-	Can work with both named and unnamed parameters. Very con-

Capability	Fb/IB Extension	PDO Extension
	tions used allow the number of parameters to be variable	venient although some Firebird features (the EXECUTE BLOCK statement) do not work.
Error handling	Requires checking the results of the ibase_errmsg, ibase_errcode functions. An error may occur after any ibase_ function call without raising any exception.	An optional mode is provided to raise exceptions on any error
Transaction management	Allows transaction parameters to be specified	Does not allow transaction parameters to be specified. Workaround: execute the SQL statement SET TRANSACTION.
Firebird-specific features	Supports work with the Services API (backup, restore, statistics, etc.) and with database events	Does not support any database-specific feature that cannot be implemented directly using an SQL statement

From these comparisons we can conclude that PDO is better equipped than the FB/IB extension for most frameworks.

Choosing a Framework for Building a Web Application

Small websites can be developed without using the MVC pattern. However, the larger your website gets, the more complicated it becomes to maintain, especially if more than one person is working on it. Hence, this is the pattern we are going to use for developing our web application.

Having decided to use the MVC pattern, we do have a few issues to think about. Development of an application modeled on this pattern is not so easy as it may seem, especially if we do not use third-party libraries. If you write everything on your own, you will have to solve a lot of problems: automatically loading `.php` files enabling the definition of classes, routing, and so on.

Several frameworks have been created for solving these problems, such as Yii, Laravel, Symphony, Kohana and many more. My personal preference is Laravel, so the development of the application described here is going to use this framework.

Installing Laravel

Before installing Laravel, make sure that your system environment meets the requirements.

MISSING TEXT

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'aa96f26c2b67226a324c27919f1eb05f21c248b987e6195cad9690d5c1ff713d53020a02
ac8c217dbf90a7eacc9d141d') { echo 'Installer verified'; } else { echo
'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Installing Composer

Laravel uses Composer to manage dependencies. Install Composer first and then install Laravel.

The easiest way to install Composer on Windows is by downloading and running the installation file: Composer-Setup.exe. The installation wizard will install Composer and configure PATH so that you can run Composer from the command line in any directory.

If you need to install Composer manually, run the following script:

```
echo @php "%~dp0composer.phar" %*>composer.bat
```

This script does the following:

- Downloads the installation file to the current directory
- Checks the installation file using SHA-384
- Runs the installation script
- Removes the installation script

After you run the script, the `composer.phar` file will appear. The `.phar` extension marks an archive but, actually, it is a PHP script that can understand only a few commands (install, update, ...) and can download and unpack libraries.

> **Windows**
>
> If you are working in Windows, you can make the task easier by creating the `composer.bat` file and saving it to PATH. Run the following command:
>
> ```
> composer global require "laravel/installer"
> ```

More details about installing Composer are available here.

Installing Laravel

Now, to install Laravel:

```
composer global require "laravel/installer"
```

Creating a Project

If the installation is successful, we can carry on with creating the project framework. Enter:

```
laravel new fbexample
```

Wait until it finishes creating the project framework. A description of the directory structure can be found in the Laravel documentation.

Our Project's Structure

These are the directories we are most interested in:

- `app`—the main directory of our application. Models will be located in the root directory. The `Http` subdirectory contains everything that is related to working with the browser. The `Http/Controllers` subdirectory contains our controllers.
- `config`—the directory with configuration files. You will discover more details about the configuration process later.
- `public`—the root directory of the web application (DocumentRoot). It contains static files: css, js, images, etc.
- `resources`—contains views, localization files and, if any, LESS files, SASS and js applications on such frameworks as ReactJS, AngularJS or Ember that are later put together into the public folder with an external tool.
- The root directory of our application contains the `composer.json` file that describes the packages our application will need besides those that are already present in Laravel.

  We will need two such packages: zofe/rapyd-laravel for building a quick interface with grids and edit dialog boxes, and sim1984/laravel-firebird, an extension for working with Firebird databases.

The `sim1984/laravel-firebird` package is the author's fork of the `jacquestvanzuydam/laravel-firebird` package. Its installation is a bit different.

> **Caution**
>
> Remember to set the minimum-stability parameter to 'dev' because the package is not stable enough to add links to the repository.

In the file `composer.json`:

```
"repositories": [
  {
    "type": "package",
    "package": {
      "version": "dev-master",
```

```
        "name": "sim1984/laravel-firebird",
        "source": {
          "url": "https://github.com/sim1984/laravel-firebird",
          "type": "git",
          "reference": "master"
        },
        "autoload": {
          "classmap": [""]
        }
      }
    }
  ],
```

Use the `require` section to add the required packages in the following way:

```
  "zofe/rapyd": "2.2.*",
  "sim1984/laravel-firebird": "dev-master"
```

Now you can start updating the packages with the following command, which must be started in the root directory of the web application:

```
  composer update
```

On completion of that command, the new packages will be installed in your application.

Configuration

Now we can get down to configuration. To get it started, execute the following command to create additional configuration files for the zofe/rapyd package:

```
  php artisan vendor:publish
```

We add two new providers to the file `config/app.php` by adding two new entries to the `providers` key:

```
  Zofe\Rapyd\RapydServiceProvider::class,
  Firebird\FirebirdServiceProvider::class,
```

We proceed to the file `config/databases.conf` (not to be confused with databases.conf in your Firebird server root!) that contains the database connection settings. Add the following lines to the `connections` key:

```
  'firebird' => [
    'driver' => 'firebird',
    'host' => env('DB_HOST', 'localhost'),
    'port' => env('DB_PORT', '3050'),
    'database' => env('DB_DATABASE', 'examples'),
    'username' => env('DB_USERNAME', 'SYSDBA'),
    'password' => env('DB_PASSWORD', 'masterkey'),
    'charset' => env('DB_CHARSET', 'UTF8'),
    'engine_version' => '3.0.0',
```

```
],
```

Since we will use our connection as the default connection, specify the following:

```
'default' => env('DB_CONNECTION', 'firebird'),
```

Pay attention to the `env` function that is used to read the environment variables of the application from the special `.env` file located in the root directory of the project. Correct the following lines in the .env file:

```
DB_CONNECTION=firebird
DB_HOST=localhost
DB_PORT=3050
DB_DATABASE=examples
DB_USERNAME=SYSDBA
DB_PASSWORD=masterkey
```

Edit the `config/rapyd.php` configuration file to change the date and time formats to match those used in your locale:

```
'fields' => [
  'attributes' => ['class' => 'form-control'],
  'date' => [
    'format' => 'Y-m-d',
  ],
  'datetime' => [
    'format'   => 'Y-m-d H:i:s',
    'store_as' => 'Y-m-d H:i:s',
  ],
],
```

That completes the initial configuration. Now we can start building the logic of the web application.

Creating Models

The Laravel framework supports the Eloquent ORM, an elegant and simple implementation of the ActiveRecord pattern for working with a database. Each table has a corresponding class model that works with it. Models enable the application to read data from tables and write data to a table. The model we are going to work with complies fully with the one illustrated earlier, at the beginning of the Database chapter.

A Tool for Model-making

To create a model for our customer entity, Laravel offers the `artisan` command that makes it relatively easy. This is the command for creating a model template:

```
php artisan make:model Customer
```

We want to change the model so that it looks like this:

```
namespace App;

use Firebird\Eloquent\Model;

class Customer extends Model
{
  /**
   * Table associated with the model
   *
   * @var string
   */
  protected $table = 'CUSTOMER';

  /**
   * Primary key of the model
   *
   * @var string
   */
  protected $primaryKey = 'CUSTOMER_ID';
  /**
   * Our model does not have a timestamp
   *
   * @var bool
   */
  public $timestamps = false;

  /**
   * The name of the sequence for generating the primary key
   *
   * @var string
   */
  protected $sequence = 'GEN_CUSTOMER_ID';
}
```

Notice that we use the modified Firebird\Eloquent\Model model from the `sim1984/laravel-firebird` package as the basis. It allows us to use the sequence specified in the *$sequence* attribute to generate values for the primary key ID.

We create a model for products—Product—in the same way.

```
namespace App;

use Firebird\Eloquent\Model;

class Product extends Model
{
  /**
   * Table associated with the model
   *
   * @var string
   */
  protected $table = 'PRODUCT';
```

```
  /**
   * Primary key of the model
   *
   * @var string
   */
  protected $primaryKey = 'PRODUCT_ID';

  /**
   * Our model does not have a timestamp
   *
   * @var bool
   */
  public $timestamps = false;

  /**
   * The name of the sequence for generating the primary key
   *
   * @var string
   */
  protected $sequence = 'GEN_PRODUCT_ID';
}
```

Now, a model for the invoice header:

```
namespace App;

use Firebird\Eloquent\Model;

class Invoice extends Model {

  /**
   * Table associated with the model
   *
   * @var string
   */
  protected $table = 'INVOICE';

  /**
   * Primary key of the model
   *
   * @var string
   */
  protected $primaryKey = 'INVOICE_ID';

  /**
   * Our model does not have a timestamp
   *
   * @var bool
   */
  public $timestamps = false;

  /**
   * The name of the sequence for generating the primary key
   *
   * @var string
   */
```

```
  protected $sequence = 'GEN_INVOICE_ID';

  /**
   * Customer
   *
   * @return \App\Customer
   */
  public function customer() {
    return $this->belongsTo('App\Customer', 'CUSTOMER_ID');
  }
  /**
   * Invoice lines
   * @return \App\InvoiceLine[]
   */
  public function lines() {
    return $this->hasMany('App\InvoiceLine', 'INVOICE_ID');
  }

  /**
   * Payed
   */
  public function pay() {
    $connection = $this->getConnection();
    $attributes = $this->attributes;
    $connection->executeProcedure('SP_PAY_FOR_INOVICE',
                              [$attributes['INVOICE_ID']]);
  }
}
```

You'll observe some additional functions in this model. The *customer* function returns the customer that relates to the invoice header via the CUSTOMER_ID field. The *belongsTo* method is used for establishing this relation. The name of the model class and the name of the relation field are passed to this method.

The function *lines* returns items from the invoice that are represented by a collection of *InvoiceLine* models, described later. To establish the one-to-many relation in the *lines* function, the name of the class model and the relation field are passed to the *hasMany* method.

You can find more details about specifying relations between entities in the Relationships section of the Laravel documentation.

The *pay* function performs payment of an invoice by calling the stored procedure SP_PAY_FOR_INVOICE, passing the identifier of the invoice header. The value of any field (model attribute) can be obtained from the attribute *attribute*. The *executeProcedure* method calls the stored procedure.

> **Note**
>
> This method is available only when the `sim1984/laravel-firebird` extension is used.

Invoice Items Model

Now we are going to create a model for items in an invoice:

```
namespace App;
```

```
use Firebird\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class InvoiceLine extends Model {

  /**
   * Table associated with the model
   *
   * @var string
   */
  protected $table = 'INVOICE_LINE';

  /**
   * Primary key of the model
   *
   * @var string
   */
  protected $primaryKey = 'INVOICE_LINE_ID';

  /**
   * Our model does not have a timestamp
   *
   * @var bool
   */
  public $timestamps = false;

  /**
   * The name of the sequence for generating the primary key
   *
   * @var string
   */
  protected $sequence = 'GEN_INVOICE_LINE_ID';

  /**
   * Array of names of computed fields
   *
   * @var array
   */
  protected $appends = ['SUM_PRICE'];

  /**
   * Product
   *
   * @return \App\Product
   */
  public function product() {
    return $this->belongsTo('App\Product', 'PRODUCT_ID');
  }

  /**
   * Amount by item
   *
   * @return double
   */
  public function getSumPriceAttribute() {
    return $this->SALE_PRICE * $this->QUANTITY;
  }
```

```
/**
 * Adding a model object to the database
 * Override this method, because in this case, we work with a stored procedure
 *
 * @param \Illuminate\Database\Eloquent\Builder $query
 * @param array $options
 * @return bool
 */
protected function performInsert(Builder $query, array $options = []) {
  if ($this->fireModelEvent('creating') === false) {
    return false;
  }
  $connection = $this->getConnection();
  $attributes = $this->attributes;
  $connection->executeProcedure('SP_ADD_INVOICE_LINE', [
    $attributes['INVOICE_ID'],
    $attributes['PRODUCT_ID'],
    $attributes['QUANTITY']
  ]);
  // We will go ahead and set the exists property to true,
  // so that it is set when the created event is fired, just in case
  // the developer tries to update it during the event. This will allow
  // them to do so and run an update here.
  $this->exists = true;
  $this->wasRecentlyCreated = true;
  $this->fireModelEvent('created', false);
  return true;
}

/**
 * Saving changes to the current model instance in the database
 * Override this method, because in this case, we work with a stored procedure
 *
 * @param \Illuminate\Database\Eloquent\Builder $query
 * @param array $options
 * @return bool
 */
protected function performUpdate(Builder $query, array $options = []) {
  $dirty = $this->getDirty();
  if (count($dirty) > 0) {
    // If the updating event returns false, we will cancel
    // the update operation so developers can hook Validation systems
    // into their models and cancel this operation if the model does
    // not pass validation. Otherwise, we update.
    if ($this->fireModelEvent('updating') === false) {
      return false;
    }
    $connection = $this->getConnection();
    $attributes = $this->attributes;
    $connection->executeProcedure('SP_EDIT_INVOICE_LINE', [
      $attributes['INVOICE_LINE_ID'],
      $attributes['QUANTITY']
    ]);
    $this->fireModelEvent('updated', false);
  }
}
```

```
  /**
   * Deleting the current model instance from the database
   * Override this method, because in this case, we work with a stored procedure
   *
   * @return void
   */
  protected function performDeleteOnModel() {
    $connection = $this->getConnection();
    $attributes = $this->attributes;
    $connection->executeProcedure('SP_DELETE_INVOICE_LINE',
      [$attributes['INVOICE_LINE_ID']]);
  }
}
```

The *product* function in this model returns the product, actually the *App/Product* model that was specified as the invoice item. The relation is established through the PRODUCT_ID field by the *belongsTo* method.

The *SumPrice* is a calculated field, calculated by the function *getSumPriceAttribute*. For a calculated field to be available in the model, its name must be specified in the *$appends* array that stores the names of calculated fields.

Operations

In this model, we redefined the insert, update and delete operations so that they are performed through stored procedures. Along with performing the insert, update and delete operations, these stored procedures recalculate the total in the invoice header. We could have avoided doing that, but then we would have had to modify several models in one transaction. Later, we will examine how to do it that way.

How Laravel Manages Data

Now let us talk a bit about how to work in Laravel with models for retrieving, inserting, updating and deleting data. Laravel uses the *query* constructor to manage data. The full description of the syntax and capabilities of this constructor is available at https://laravel.com/docs/5.2/queries. For example, you can execute the following query to retrieve all supplier rows:

```
$customers = DB::table('CUSTOMER')->get();
```

This query constructor is quite a powerful tool for building and executing SQL queries. You can also direct it to filter, sort and merge tables. For example:

```
DB::table('users')
  ->join('contacts', function ($join) {
    $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
  })
  ->get()
```

Nevertheless, models are more convenient to work with. You can find the description of Eloquent ORM models and the syntax for querying them at https://laravel.com/docs/5.2/eloquent.

As an example, to retrieve all elements from the collection of customers would require executing the following query:

```
$customers = Customer::all();
```

This query will return the first 20 customers sorted alphabetically:

```
$customers = App\Customer::select()
           ->orderBy('name')
           ->take(20)
           ->get();
```

Complex Models

When a model is more complex, its relationships or relationship collections can be retrieved via *dynamic attributes*. The following query, for example, returns the items of the invoice that has the identifier 1:

```
$lines = Invoice::find(1)->lines;
```

Records are added by creating an instance of the model, initiating its attributes and saving the model using the *save* method:

```
$flight = new Flight;
$flight->name = $request->name;
$flight->save();
```

Updating a record involves finding it, accepting changes to the appropriate attributes and saving it with the *save* method:

```
$flight = App\Flight::find(1);
$flight->name = 'New Flight Name';
$flight->save();
To delete a record, involves finding it and calling the delete method.
$flight = App\Flight::find(1);
$flight->delete();
```

The *destroy* method allows a record to be deleted more rapidly by its key value, without needing to retrieve its instance:

```
App\Flight::destroy(1);
```

There are other ways of deleting records, for instance, "soft" deletion. You can read more about deletion methods at https://laravel.com/docs/5.2/eloquent#deleting-models.

Transactions

Now let us talk a little about transactions. Without going into the fine detail, I will demonstrate how transactions and the Eloquent ORM can be used together.

```
DB::transaction(function () {
  // Create a new position in the invoice
  $line = new App\InvoiceLine();
  $line->CUSTOMER_ID = 45;
  $line->PRODUCT_ID = 342;
  $line->QUANTITY = 10;
  $line->COST = 12.45;
  $line->save();
  // add the sum of the line item to the amount of the invoice
  $invoice = App\Invoice::find($line->CUSTOMER_ID);
  $invoice->INVOICE_SUM += $line->SUM_PRICE;
  $invoice->save();
});
```

Every parameter of the *transaction* method that is located inside the callback function is executed within one transaction.

Creating Controllers and Configuring Routing

The Laravel framework has a powerful routing subsystem. You can display your routes both for simple callback functions and for the controller methods. The simplest sample routes look like this:

```
Route::get('/', function () {
  return 'Hello World';
});

Route::post('foo/bar', function () {
  return 'Hello World';
});
```

In the first example, we register the handler of the GET request for the website root for the POST request with the route `/foo/bar` in the second.

You can register a route for several types of HTTP requests. For example:

```
Route::match(['get', 'post'], 'foo/bar', function () {
  return 'Hello World';
});
```

You can extract some part of the URL from the route for use as a parameter in the handling function:

```
Route::get('posts/{post}/comments/{comment}', function ($postId, $commentId) {
  //
});
```

The parameters of a route are always enclosed in braces.

You can find more details about routing configuration in the Routing chapter of the documentation. Routes are configured in the `app/Http/routes.php` file in Laravel 5.2 and in the `routes/wep.php` file in Laravel 5.3.

Using Controllers to Route Requests

Instead of directing the processing of all requests from a single routing file, we can use Controller classes to group related request handlers into separate classes. Controllers are stored in the `app/Http/Controllers` folder.

All Laravel controllers must extend the basic class of the controller `App\Http\Controllers\Controller` that exists in Laravel by default. You can read more details about writing controllers at https://laravel.com/docs/5.2/controllers.

A Customer Controller

First, we'll write our Customer controller.

```
<?php

/*
* Customer controller
*/
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Customer;

class CustomerController extends Controller
{
  /**
   * Show customer list
   *
   * @return Response
   */
  public function showCustomers()
  {
    // get the first 20 customers
    // sorted alphabetically
    $customers = Customer::select()
                ->orderBy('NAME')
                ->take(20)
                ->get();
    var_dump($customers);
  }
}
```

Now we have to link the controller methods to the route. For this, add the following line to `routes.php` (web.php):

```
Route::get('/customers', 'CustomerController@showCustomers');
```

The controller name is separated from the method name with the @ character.

To build a quick interface with grids and edit dialog boxes, we will use the `zofe/rapyd` package that was enabled earlier. Classes from the `zofe/rapyd` package take up the role of building standard queries to Eloquent ORM models. We will change the customer controller so that it shows data on the grid, allows filtering and record insertions, updates and deletes by way of the edit dialog boxes.

```
<?php
/*
 * Customer Controller
 */
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Customer;

class CustomerController extends Controller {

  /**
   * Displays the list of customers
   *
   * @return Response
   */
  public function showCustomers() {
    // Connect widget for search
    $filter = \DataFilter::source(new Customer);
    // Search will be by the name of the supplier
    $filter->add('NAME', 'Name', 'text');
    // Set capture for search button
    $filter->submit('Search');
    // Add the filter reset button and assign it caption
    $filter->reset('Reset');
    // Create a grid to display the filtered data
    $grid = \DataGrid::source($filter);
    // output columns
    // Field, label, sorted
    $grid->add('NAME', 'Name', true);
    $grid->add('ADDRESS', 'Address');
    $grid->add('ZIPCODE', 'Zip Code');
    $grid->add('PHONE', 'Phone');
    // Add buttons to view, edit and delete records
    $grid->edit('/customer/edit', 'Edit', 'show|modify|delete');
    // Add the Add Customer button
    $grid->link('/customer/edit', "Add customer", "TR");
    $grid->orderBy('NAME', 'asc');
    // set the number of records per page
    $grid->paginate(10);
    // display the customer template and pass the filter and grid to it
    return view('customer', compact('filter', 'grid'));
  }
```

```
  /**
   * Add, edit and delete a customer
   *
   * @return Response
   */
  public function editCustomer() {
    if (\Input::get('do_delete') == 1)
      return "not the first";
    // create an editor
    $edit = \DataEdit::source(new Customer());
    // Set title of the dialog, depending on the type of operation
    switch ($edit->status) {
      case 'create':
        $edit->label('Add customer');
        break;
      case 'modify':
        $edit->label('Edit customer');
        break;
      case 'do_delete':
        $edit->label('Delete customer');
        break;
      case 'show':
        $edit->label("Customer's card");
        // add a link to go back to the list of customers
        $edit->link('customers', 'Back', 'TR');
        break;
    }
    // set that after the operations of adding, editing and deleting,
    // you need to return to the list of customers
    $edit->back('insert|update|do_delete', 'customers');
    // We add editors of a certain type, assign them a label and
    // associate them with the attributes of the model
    $edit->add('NAME', 'Name', 'text')->rule('required|max:60');
    $edit->add('ADDRESS', 'Address', 'textarea')
         ->attributes(['rows' => 3])
         ->rule('max:250');
    $edit->add('ZIPCODE', 'Zip code', 'text')->rule('max:10');
    $edit->add('PHONE', 'Phone', 'text')->rule('max:14');
    // display the template customer_edit and pass it to the editor
    return $edit->view('customer_edit', compact('edit'));
  }
}
```

blade Templates

By default, Laravel uses the *blade template engine*. The *view* function finds the necessary template in the `resources/views` directory, makes the necessary changes to it and returns the text of the HTML page, at the same time passing to it any variables that are supplied in the template. You can find the description of the *blade* template syntax at https://laravel.com/docs/5.2/blade.

The Template for Displaying Customers

The template for displaying customers looks like this:

```
@extends('example')

@section('title', 'Customers')

@section('body')
<h1>Customers</h1>
<p>
  {!! $filter !!}
  {!! $grid !!}
</p>
@stop
```

This template is inherited from the example template and redefines its body section. The $filter and $grid variables contain the HTML code for filtering and displaying data on the grid. The example template is common for all pages.

```
@extends('master')
@section('title', 'Example of working with Firebird')

@section('body')
<h1>??????</h1>
  @if(Session::has('message'))
  <div class="alert alert-success">
    {!! Session::get('message') !!}
  </div>
  @endif
  <p>Example of working with Firebird.<br/>
  </p>
@stop

@section('content')
@include('menu')
@yield('body')

@stop
```

This template is itself inherited from the master template and also enables the menu template. The menu is quite simple and consists of three items: Customers, Products and Invoices.

```
<nav class="navbar main">
  <div class="navbar-header">
    <button type="button" class="navbar-toggle"
            data-toggle="collapse" data-target=".main-collapse">
      <span class="sr-only"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
  </div>
  <div class="collapse navbar-collapse main-collapse">
    <ul class="nav nav-tabs">
      <li @if (Request::is('customer*'))
          class="active"@endif>{!! link_to("customers", "Customers") !!}</li>
```

```
      <li @if (Request::is('product*'))
          class="active"@endif>{!! link_to("products", "Products") !!}</li>
      <li @if (Request::is('invoice*'))
          class="active"@endif>{!! link_to("invoices", "Invoices") !!}</li>
    </ul>
  </div>
</nav>
```

The master template enables css styles and JavaScript files with libraries.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>@yield('title', 'An example of a Web application on Firebird')</title>
  <meta name="description" content="@yield('description',
        'An example of a Web application on Firebird')" />
  @section('meta', '')
  <link href="http://fonts.googleapis.com/css?family=Bitter" rel="stylesheet"
        type="text/css" />
  <link href="//netdna.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css"
        rel="stylesheet">
  <link href="//maxcdn.bootstrapcdn.com/font-awesome/4.1.0/css/font-awesome.min.css"
        rel="stylesheet">
  {!! Rapyd::styles(true) !!}
</head>
<body>
  <div id="wrap">
    <div class="container">
      <br />
      <div class="row">
        <div class="col-sm-12">
          @yield('content')
        </div>
      </div>
    </div>
  </div>
  <div id="footer">
  </div>
  <script src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
  </script>
  <script src="//netdna.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js">
  </script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery.pjax/1.9.6/jquery.pjax.min.js"
  <script src="https://cdnjs.cloudflare.com/ajax/libs/riot/2.2.4/riot+compiler.min.js"></sc
  {!! Rapyd::scripts() !!}
</body>
</html>
```

The customer_edit template:

```
@extends('example')
@section('title', 'Edit customer')
@section('body')
```

```
  <p>
    {!! $edit !!}
  </p>
@stop
```

A Product Controller

Implementation of the product controller is similar to what we did for the customer controller:

```
<?php

/*
 * Product Controller
 */
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Product;

class ProductController extends Controller {

  /**
   * Displays a list of products
   *
   * @return Response
   */
  public function showProducts() {
    // Connect widget for search
    $filter = \DataFilter::source(new Product);
    // The search will be by product name
    $filter->add('NAME', 'Name', 'text');
    $filter->submit('Search');
    $filter->reset('Reset');
    // Create a grid to display the filtered data
    $grid = \DataGrid::source($filter);
    // output grid columns
    // Field, label, sorting
    $grid->add('NAME', 'Name', true);
    // Set the format with 2 decimal places
    $grid->add('PRICE|number_format[2,., ]', 'Price');
    $grid->row(function($row) {
      // Press the money values to the right
      $row->cell('PRICE')->style("text-align: right");
    });
    // Add buttons to view, edit and delete records
    $grid->edit('/product/edit', 'Edit', 'show|modify|delete');
    // Add the Add product button
    $grid->link('/product/edit', "?????????? ??????", "TR");
    // set sorting
    $grid->orderBy('NAME', 'asc');
    // set the number of records per page
    $grid->paginate(10);
    // display the customer template and pass the filter and grid to it
    return view('product', compact('filter', 'grid'));
```

```
  }

  /**
   * Add, edit and delete products
   *
   * @return Response
   */
  public function editProduct() {
    if (\Input::get('do_delete') == 1)
      return "not the first";
    // create editor
    $edit = \DataEdit::source(new Product());
    // Set the title of the dialog, depending on the type of operation
    switch ($edit->status) {
      case 'create':
        $edit->label('Add product');
        break;
      case 'modify':
        $edit->label('Edit product');
        break;
      case 'do_delete':
        $edit->label('Delete product');
        break;
      case 'show':
        $edit->label("Product's card");
        $edit->link('products', 'Back', 'TR');
        break;
    }
    // set that after the operations of adding, editing and deleting,
    // you need to return to the list of products
    $edit->back('insert|update|do_delete', 'products');
    // We add editors of a certain type, assign them a label and
    // associate them with the attributes of the model
    $edit->add('NAME', 'Name', 'text')->rule('required|max:100');
    $edit->add('PRICE', 'Price', 'text')->rule('max:19');
    $edit->add('DESCRIPTION', 'Description', 'textarea')
         ->attributes(['rows' => 8])
         ->rule('max:8192');
    // display the template product_edit and pass it to the editor
    return $edit->view('product_edit', compact('edit'));
  }
}
```

A Controller for Invoices

The invoice controller is more complex and includes an additional function to pay an invoice. Paid invoices are highlighted in a different color. While viewing an invoice, you can also see its items. While editing an invoice, you can edit its items as well. Here is the code for the controller with detailed comments.

```
<?php
/*
 * Invoice controller
 */
namespace App\Http\Controllers;
```

```
use App\Http\Controllers\Controller;
use App\Invoice;
use App\Customer;
use App\Product;
use App\InvoiceLine;

class InvoiceController extends Controller {

  /**
   * Show invoice list
   *
   * @return Response
   */
  public function showInvoices() {
    // The invoice model will also select the related suppliers
    $invoices = Invoice::with('customer');
    // Add a widget for search.
    $filter = \DataFilter::source($invoices);
    // Let's filter by date range
    $filter->add('INVOICE_DATE', 'Date', 'daterange');
    // and filter by customer name
    $filter->add('customer.NAME', 'Customer', 'text');
    $filter->submit('Search');
    $filter->reset('Reset');
    // Create a grid to display the filtered data
    $grid = \DataGrid::source($filter);
    // output grid columns
    // Field, caption, sorted
    // For the date we set an additional function that converts
    // the date into a string
    $grid->add('INVOICE_DATE|strtotime|date[Y-m-d H:i:s]', 'Date', true);
    // for money we will set a format with two decimal places
    $grid->add('TOTAL_SALE|number_format[2,., ]', 'Amount');
    $grid->add('customer.NAME', 'Customer');
    // Boolean printed as Yes/No
    $grid->add('PAID', 'Paid')
         ->cell(function( $value, $row) {
                  return $value ? 'Yes' : 'No';
                });
    // set the function of processing each row
    $grid->row(function($row) {
      // The monetary values are pressed to the right
      $row->cell('TOTAL_SALE')->style("text-align: right");
      // paint the paid waybills in a different color
      if ($row->cell('PAID')->value == 'Yes') {
        $row->style("background-color: #ddffee;");
      }
    });
    // Add buttons to view, edit and delete records
    $grid->edit('/invoice/edit', '??????????????', 'show|modify|delete');
    // Add the button for adding invoices
    $grid->link('/invoice/edit', "?????????? ?????", "TR");

    $grid->orderBy('INVOICE_DATE', 'desc');
    // set the number of records per page
    $grid->paginate(10);
    // display the customer template and pass the filter and grid to it
```

```
    return view('invoice', compact('filter', 'grid'));
  }

  /**
   * Add, edit and delete invoice
   *
   * @return Response
   */
  public function editInvoice() {
    // get the text of the saved error, if it was
    $error_msg = \Request::old('error_msg');
    // create an invoice invoice editor
    $edit = \DataEdit::source(new Invoice());
    // if the invoice is paid, then we generate an error when trying to edit it
    if (($edit->model->PAID) && ($edit->status === 'modify')) {
      $edit->status = 'show';
      $error_msg = 'Editing is not possible. The account has already been paid.';
    }
    // if the invoice is paid, then we generate an error when trying to delete it
    if (($edit->model->PAID) && ($edit->status === 'delete')) {
      $edit->status = 'show';
      $error_msg = 'Deleting is not possible. The account has already been paid.';
    }
    // Set the label of the dialog, depending on the type of operation
    switch ($edit->status) {
      case 'create':
        $edit->label('Add invoice');
        break;

      case 'modify':
        $edit->label('Edit invoice');
        break;

      case 'do_delete':
        $edit->label('Delete invoice');
        break;

      case 'show':
        $edit->label('Invoice');
        $edit->link('invoices', 'Back', 'TR');
        // If the invoice is not paid, we show the pay button
        if (!$edit->model->PAID)
          $edit->link('invoice/pay/' . $edit->model->INVOICE_ID,
                      'Pay', 'BL');
        break;
    }
    // set that after the operations of adding, editing and deleting,
    // we return to the list of invoices
    $edit->back('insert|update|do_delete', 'invoices');
    // set the "date" field, that it is mandatory
    // The default is the current date
    $edit->add('INVOICE_DATE', '????', 'datetime')
         ->rule('required')
         ->insertValue(date('Y-m-d H:i:s'));
    // add a field for entering the customer. When typing a customer name,
    // a list of prompts will be displayed
    $edit->add('customer.NAME', 'Customer', 'autocomplete')
         ->rule('required')
```

```
        ->options(Customer::lists('NAME', 'CUSTOMER_ID')
                          ->all());
    // add a field that will display the invoice amount, read-only
    $edit->add('TOTAL_SALE', 'Amount', 'text')
         ->mode('readonly')
         ->insertValue('0.00');
    // add paid checkbox
    $paidCheckbox = $edit->add('PAID', 'Paid', 'checkbox')
                        ->insertValue('0')
                        ->mode('readonly');
    $paidCheckbox->checked_output = 'Yes';
    $paidCheckbox->unchecked_output = 'No';
    // create a grid to display the invoice line rows
    $grid = $this->getInvoiceLineGrid($edit->model, $edit->status);
    // we display the invoice_edit template and pass the editor and grid to
    // it to display the invoice invoice items
    return $edit->view('invoice_edit', compact('edit', 'grid', 'error_msg'));
  }

  /**
   * Payment of invoice
   *
   * @return Response
   */
  public function payInvoice($id) {
    try {
      // find the invoice by ID
      $invoice = Invoice::findOrFail($id);
      // call the payment procedure
      $invoice->pay();
    } catch (\Illuminate\Database\QueryException $e) {
      // if an error occurs, select the exclusion text
      $pos = strpos($e->getMessage(), 'E_INVOICE_ALREADY_PAYED');
      if ($pos !== false) {
        // redirect to the editor page and display the error there
        return redirect('invoice/edit?show=' . $id)
             ->withInput(['error_msg' => 'Invoice already paid']);
      } else
        throw $e;
    }
    // redirect to the editor page
    return redirect('invoice/edit?show=' . $id);
  }

  /**
   * Returns the grid for the invoice item
   * @param \App\Invoice $invoice
   * @param string $mode
   * @return \DataGrid
   */
  private function getInvoiceLineGrid(Invoice $invoice, $mode) {
    // Get invoice items
    // For each ivoice item, the associated product will be initialized
    $lines = InvoiceLine::with('product')
               ->where('INVOICE_ID', $invoice->INVOICE_ID);
    // Create a grid for displaying invoice items
    $grid = \DataGrid::source($lines);
    // output grid columns
```

```
    // Field, caption, sorted
    $grid->add('product.NAME', 'Name');
    $grid->add('QUANTITY', 'Quantity');
    $grid->add('SALE_PRICE|number_format[2,., ]', 'Price')
         ->style('min-width: 8em;');
    $grid->add('SUM_PRICE|number_format[2,., ]', 'Amount')
         ->style('min-width: 8em;');
    // set the function of processing each row
    $grid->row(function($row) {
      $row->cell('QUANTITY')->style("text-align: right");
      // The monetary values are pressed to the right
      $row->cell('SALE_PRICE')->style("text-align: right");
      $row->cell('SUM_PRICE')->style("text-align: right");
    });
    if ($mode == 'modify') {
      // Add buttons to view, edit and delete records
      $grid->edit('/invoice/editline', '??????????????', 'modify|delete');
      // Add a button to add an invoice item
      $grid->link('/invoice/editline?invoice_id=' . $invoice->INVOICE_ID,
                  "Add item", "TR");
    }
    return $grid;
  }

  /**
   * Add, edit and delete invoice items
   *
   * @return Response
   */
  public function editInvoiceLine() {
    if (\Input::get('do_delete') == 1)
      return "not the first";
    $invoice_id = null;
    // create the editor of the invoice item
    $edit = \DataEdit::source(new InvoiceLine());
    // Set the label of the dialog, depending on the type of operation
    switch ($edit->status) {
      case 'create':
        $edit->label('Add invoice item');
        $invoice_id = \Input::get('invoice_id');
        break;

      case 'modify':
        $edit->label('Edit invoice item');
        $invoice_id = $edit->model->INVOICE_ID;
        break;

      case 'delete':
        $invoice_id = $edit->model->INVOICE_ID;
        break;

      case 'do_delete':
        $edit->label('Delete invoice item');
        $invoice_id = $edit->model->INVOICE_ID;
        break;
    }
    // make url to go back
    $base = str_replace(\Request::path(), '', strtok(\Request::fullUrl(), '?'));
```

```
    $back_url = $base . 'invoice/edit?modify=' . $invoice_id;
    // set the page to go back
    $edit->back('insert|update|do_delete', $back_url);
    $edit->back_url = $back_url;
    // add a hidden field with an invoice code
    $edit->add('INVOICE_ID', '', 'hidden')
         ->rule('required')
         ->insertValue($invoice_id)
         ->updateValue($invoice_id);
    // Add a field for entering the goods. When you type the product name,
    // a list of prompts is displayed.
    $edit->add('product.NAME', 'Name', 'autocomplete')
         ->rule('required')
         ->options(Product::lists('NAME', 'PRODUCT_ID')->all());
    // Field for input quantity
    $edit->add('QUANTITY', 'Quantity', 'text')
         ->rule('required');
    // display the template invoice_line_edit and pass it to the editor
    return $edit->view('invoice_line_edit', compact('edit'));
  }
}
```

The Invoice Editor

The invoice editor has a view that is not standard for `zofe/rapyd` because we want to display a grid with invoice items. To do that, we change the invoice_edit template as follows:

```
@extends('example')

@section('title','Edit invoice')

@section('body')
  <div class="container">
    {!! $edit->header !!}
    @if($error_msg)
      <div class="alert alert-danger">
        <strong>??????!</strong> {{ $error_msg }}
      </div>
    @endif
    {!! $edit->message !!}
    @if(!$edit->message)
      <div class="row">
        <div class="col-sm-4">
          {!! $edit->render('INVOICE_DATE') !!}
          {!! $edit->render('customer.NAME') !!}
          {!! $edit->render('TOTAL_SALE') !!}
          {!! $edit->render('PAID') !!}
        </div>
      </div>
      {!! $grid !!}
    @endif
    {!! $edit->footer !!}
  </div>
@stop
```

Changing the Routes

Now that all controllers are written, we are going to change the routes so that our website opens the list of invoices on the start page. Be aware that routes are configured in the file `app/Http/routes.php` in Laravel 5.2 and in `routes/wep.php` in Laravel 5.3.

```
Route::get('/', 'InvoiceController@showInvoices');
Route::get('/customers', 'CustomerController@showCustomers');
Route::any('/customer/edit', 'CustomerController@editCustomer');
Route::get('/products', 'ProductController@showProducts');
Route::any('/product/edit', 'ProductController@editProduct');
Route::get('/invoices', 'InvoiceController@showInvoices');
Route::any('/invoice/edit', 'InvoiceController@editInvoice');
Route::any('/invoice/pay/{id}', 'InvoiceController@payInvoice');
Route::any('/invoice/editline', 'InvoiceController@editInvoiceLine');
```

Here the `/invoice/pay/{id}` route picks up the invoice identifier from the URL and sends it to the *payInvoice* method. The rest of the routes should be self-explanatory.

The Result

Some screenshots from the web application we developed in this project.

Figure 6.1. Screenshot 1

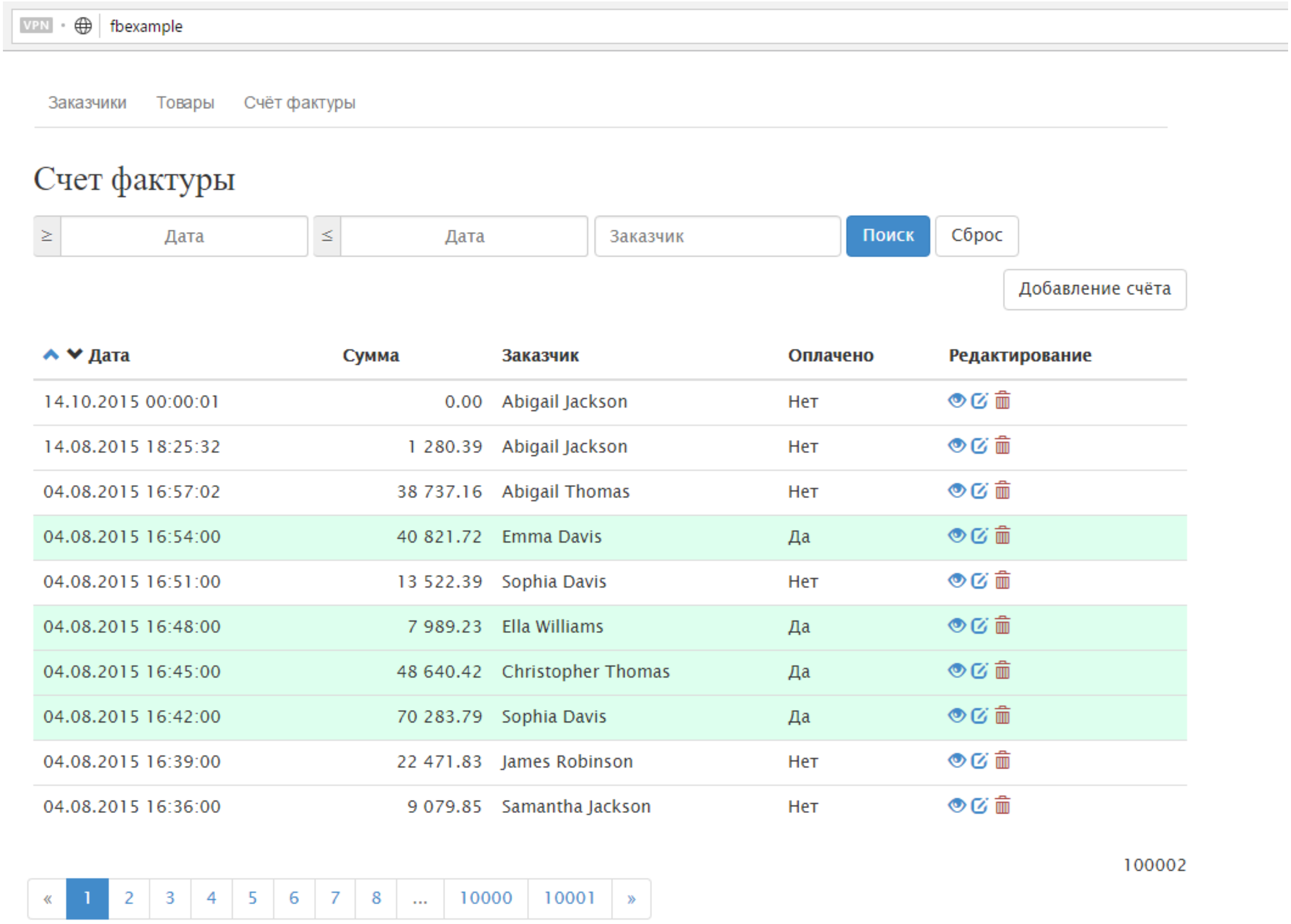
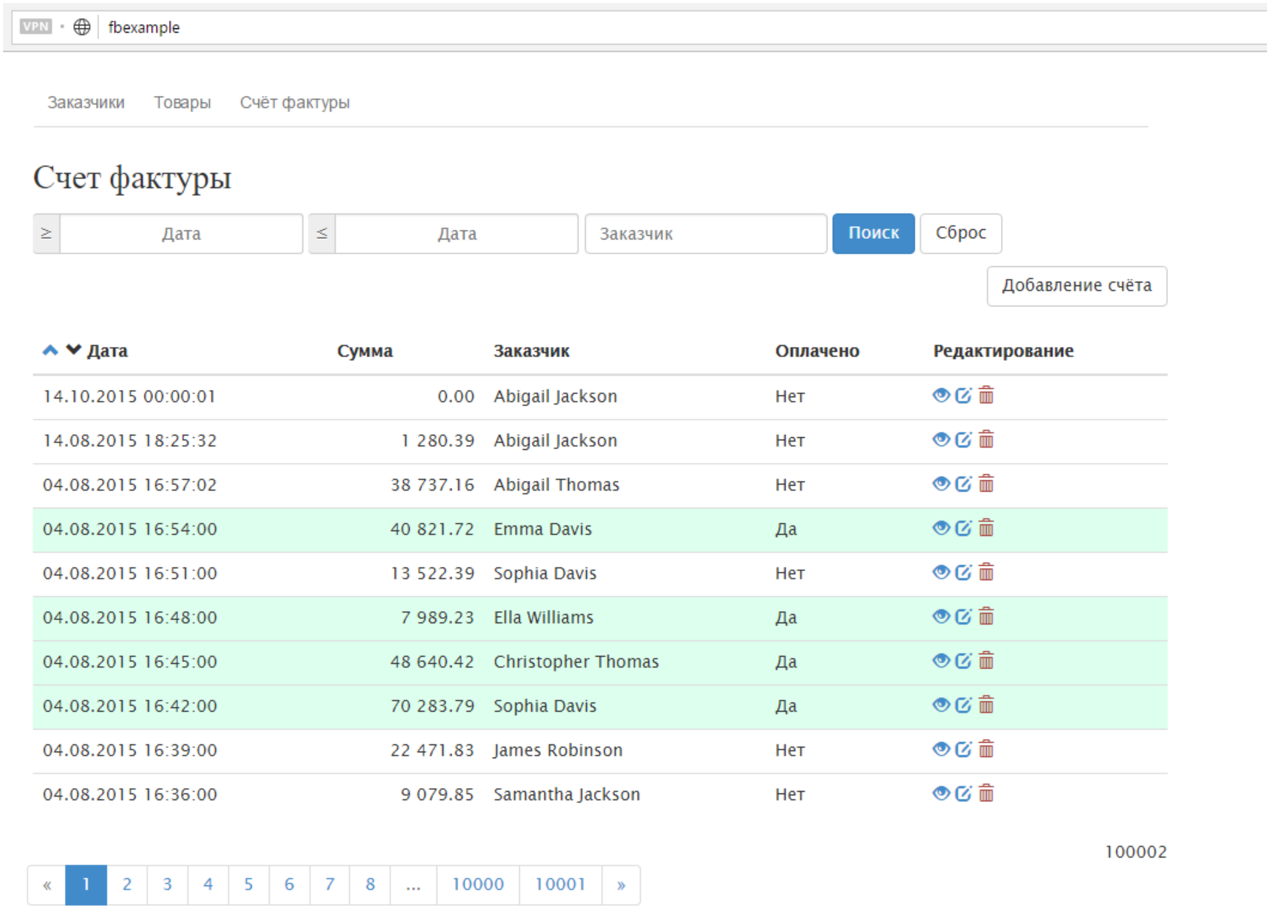


Figure 6.2. Screenshot 2



Source Code

You can download the source code for this project from https://github.com/sim1984/phpfbexample

Chapter 7

Creating an Application with jOOQ and Spring MVC

This chapter will describe how to create a web application in the Java language using the Spring MVC framework, the jOOQ library and a Firebird sample database.

To make development easier, you can use one of the popular IDEs for Java (NetBeans, IntelliJ IDEA, Eclipse, JDeveloper and others). I used NetBeans.

For testing and debugging purposes, we will also need to install one of the web servers or application servers (Apache Tomcat or GlassFish). We are basing our project on the Maven web application templates.

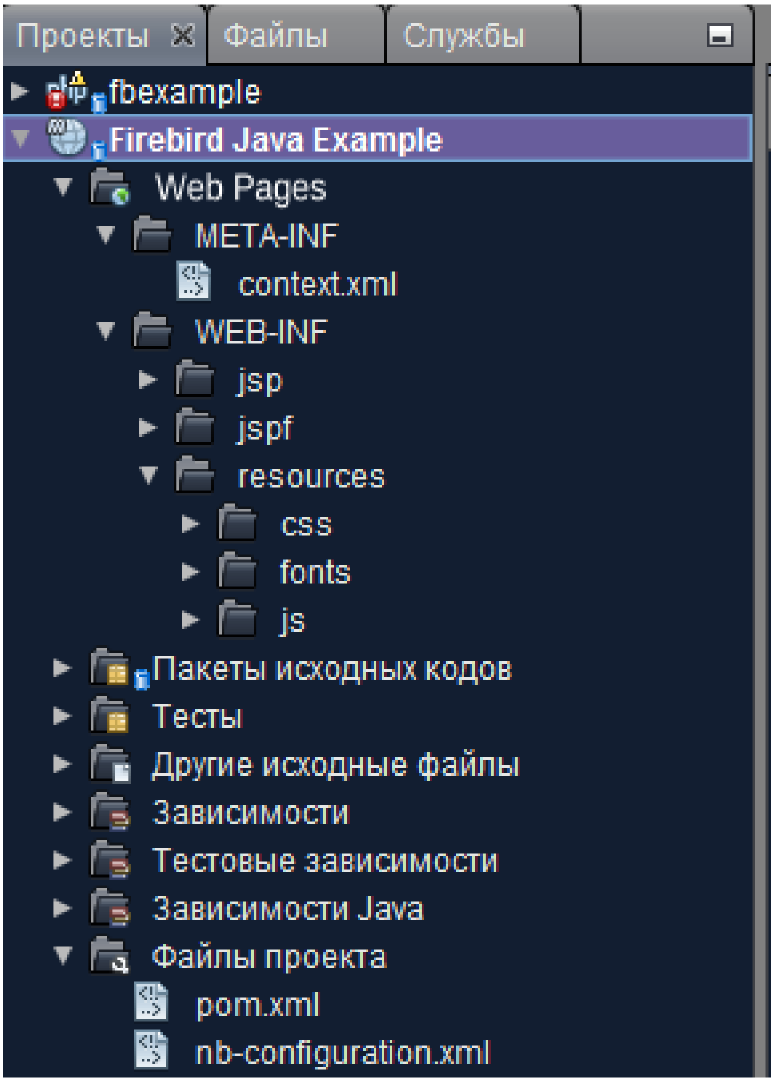
Organising the Folder Structure

After a template-based project has been created, its folder structure will need to be rearranged to suit Spring 4. In the NetBeans 8.2 environment, the steps would be as follows:

1. Delete the `index.html` file
2. Create the `WEB-INF` folder inside the `Web Pages` folder
3. Create the `jsp`, `jspf` and `resources` folders inside the `WEB-INF` folder
4. Create the `js` and `CSS` folders inside the `resources` folder
5. Create the `index.jsp` file inside the `jsp` folder

The new structure of the folders should look like this:

Figure 7.1. Folder structure for the template-based project



The `WEB-INF/jsp` folder will contain jsp pages and the `jspf` folder will contain page fragments that will be added to other pages using the following directive:

```
<%@ include file ="<filename>" %>
```

The `resources` folder is used to store static web resources—the `WEB-INF/resources/css` folder for cascading style sheet files, the `WEB-INF/resources/fonts` folder for font files, the `WEB-INF/resources/js` folder for JavaScript files and third-party JavaScript libraries.

Now, we modify the `pom.xml` file and add the general properties of the application, dependencies on library packages (Spring MVC, Jaybird, JDBC pool, JOOQ) and the properties of the JDBC connection.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
<groupId>ru.ibase</groupId>
<artifactId>fbjavaex</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<name>Firebird Java Example</name>

<properties>
    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spring.version>4.3.4.RELEASE</spring.version>
    <jstl.version>1.2</jstl.version>
    <javax.servlet.version>3.0.1</javax.servlet.version>
    <db.url>jdbc:firebirdsql://localhost:3050/examples</db.url>
    <db.driver>org.firebirdsql.jdbc.FBDriver</db.driver>
    <db.username>SYSDBA</db.username>
    <db.password>masterkey</db.password>
</properties>


<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-web-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>${javax.servlet.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>${jstl.version}</version>
    </dependency>

    <!-- Work with JSON -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.8.5</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
        <version>2.8.5</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
```

```
        <version>2.8.5</version>
    </dependency>


    <!-- Spring -->

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <!-- JDBC -->

    <dependency>
        <groupId>org.firebirdsql.jdbc</groupId>
        <artifactId>jaybird-jdk18</artifactId>
        <version>3.0.0</version>
    </dependency>

    <!-- Connection pool -->

    <dependency>
        <groupId>commons-dbcp</groupId>
        <artifactId>commons-dbcp</artifactId>
        <version>1.4</version>
    </dependency>

    <!-- jOOQ -->

    <dependency>
        <groupId>org.jooq</groupId>
        <artifactId>jooq</artifactId>
        <version>3.9.2</version>
```

```
        </dependency>

        <dependency>
            <groupId>org.jooq</groupId>
            <artifactId>jooq-meta</artifactId>
            <version>3.9.2</version>
        </dependency>

        <dependency>
            <groupId>org.jooq</groupId>
            <artifactId>jooq-codegen</artifactId>
            <version>3.9.2</version>
        </dependency>

        <!-- Testing -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <type>jar</type>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring.version}</version>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <build>

        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                    <compilerArguments>
                        <endorseddirs>${endorsed.dir}</endorseddirs>
                    </compilerArguments>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
```

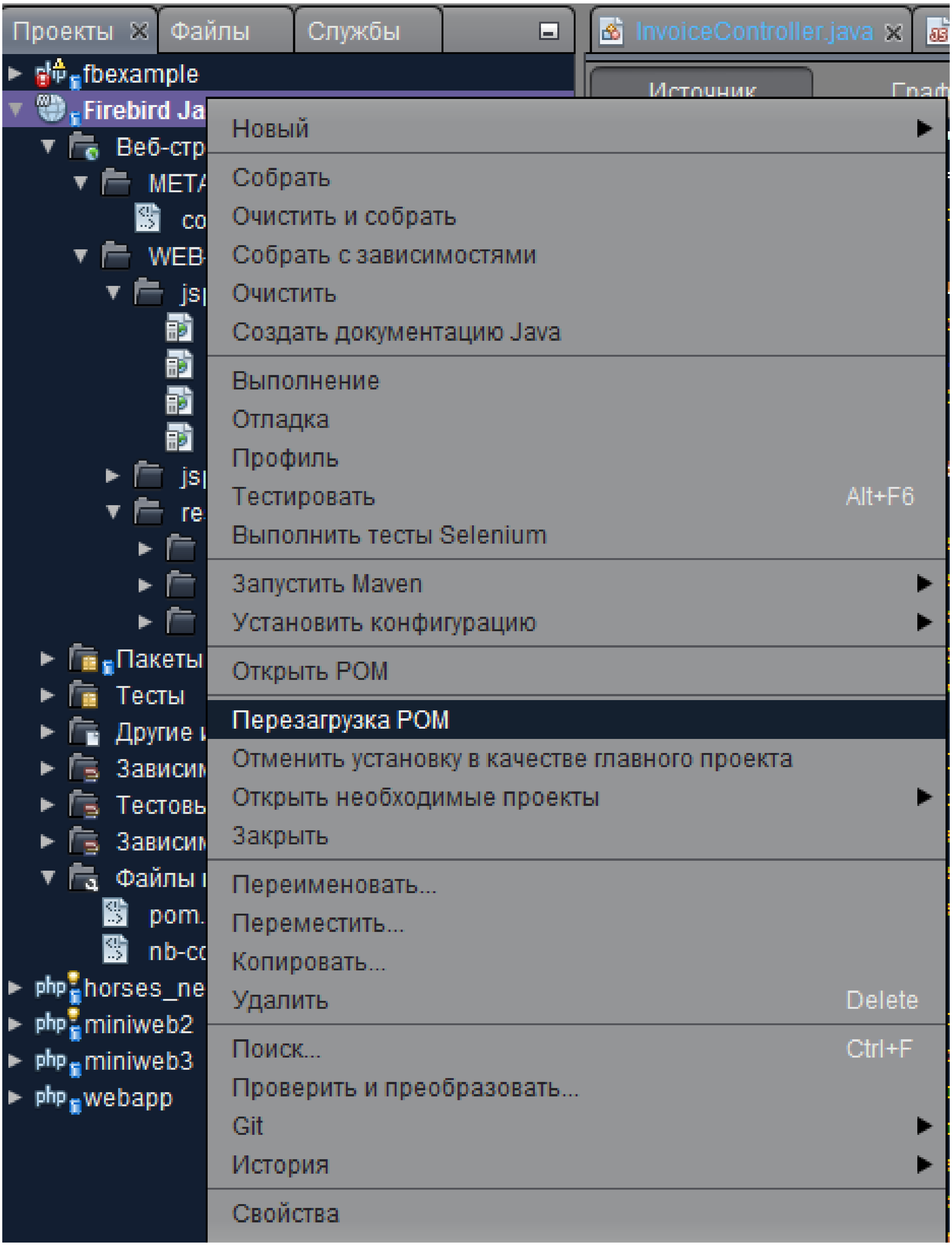
```
                <version>2.6</version>
                <executions>
                    <execution>
                        <phase>validate</phase>
                        <goals>
                            <goal>copy</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>${endorsed.dir}</outputDirectory>
                            <silent>true</silent>
                            <artifactItems>
                                <artifactItem>
                                    <groupId>javax</groupId>
                                    <artifactId>javaee-endorsed-api</artifactId>
                                    <version>7.0</version>
                                    <type>jar</type>
                                </artifactItem>
                            </artifactItems>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

What is a POM?

A *Project Object Model* or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. More details can be found at http://maven.apache.org/guides/introduction/introduction-to-the-pom.

After all the necessary dependencies have been fulfilled, a restart of the POM is recommended, to load all the necessary libraries and avoid errors that might otherwise occur while you are working on the project. This is how it is done in NetBeans:

Figure 7.2. Restarting the POM from NetBeans



Coding the Configuration

We use this configuration class to specify the locations of web resources and JSP views. The *configureMessage-Converters* method directs that dates must be serialized to strings, overriding the default that serializes them to a numeric representation of a timestamp.

I am creating Java configuration classes here as I am not a big fan of doing configuration in XML.

```
package ru.ibase.fbjavaex.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.JstlView;
import org.springframework.web.servlet.view.UrlBasedViewResolver;
import org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;
import org.springframework.http.converter.HttpMessageConverter;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import java.util.List;

@Configuration
@ComponentScan("ru.ibase.fbjavaex")
@EnableWebMvc
public class WebAppConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureMessageConverters(
      List<HttpMessageConverter<?>> httpMessageConverters) {
        MappingJackson2HttpMessageConverter jsonConverter =
            new MappingJackson2HttpMessageConverter();
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS,
                               false);
        jsonConverter.setObjectMapper(objectMapper);
        httpMessageConverters.add(jsonConverter);
    }

    @Bean
    public UrlBasedViewResolver setupViewResolver() {
        UrlBasedViewResolver resolver = new UrlBasedViewResolver();
        resolver.setPrefix("/WEB-INF/jsp/");
        resolver.setSuffix(".jsp");
        resolver.setViewClass(JstlView.class);
        return resolver;
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
```

```
                .addResourceLocations("/WEB-INF/resources/");
    }
}
```

Start-up Code—WebInitializer

Now we'll get rid of the `Web.xml` file and create the WebInitializer.java class in its place:

```
package ru.ibase.fbjavaex.config;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration.Dynamic;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class WebInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext ctx =
            new AnnotationConfigWebApplicationContext();
        ctx.register(WebAppConfig.class);
        ctx.setServletContext(servletContext);
        Dynamic servlet = servletContext.addServlet("dispatcher",
                                                    new DispatcherServlet(ctx));
        servlet.addMapping("/");
        servlet.setLoadOnStartup(1);
    }

}
```

All that is left to configure is IoC containers for injecting dependencies, a step we will return to later. We proceed next to generating classes for working with the database via Java Object-Oriented Querying (jOOQ).

Generating classes for jOOQ

Work with the database will be carried out using the jOOQ library. jOOQ builds SQL queries from jOOQ objects and code (similarly to LINQ). jOOQ is more closely integrated with the database than ORM, enabling more database features to be utilized, rather than just the simple CRUD SQL queries used in Active Record. jOOQ can work with stored procedures and functions, sequences, and use window functions and other Firebird-specific features.

You can find the full documentation for jOOQ at http://www.jooq.org/doc/3.9/manual-single-page/.

jOOQ Classes

jOOQ classes for working with the database are generated on the basis of the database schema described in the earlier chapter, The examples.fdb Database.

To generate jOOQ classes for working with our database, you will need to download these binary files at http://www.jooq.org/download or via the maven repository:

- jooq-3.9.2.jarThe main library included in our application for working with jOOQ
- jooq-meta-3.9.2.jar—The tool included in your build for navigating the database schema via generated objects
- jooq-codegen-3.9.2.jar—The tool included in your build for generating the database schema

Along with those, of course, you will need to download the Jaybird driver for connecting to the Firebird database via JDBC: `jaybird-full-3.0.0.jar`.

Configuration for Database Schema Classes

For generating the classes for the database schema, we create the configuration file example.xml:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-3.8.0.xsd">
  <!-- Configuration of connection to the database -->
  <jdbc>
    <driver>org.firebirdsql.jdbc.FBDriver</driver>
    <url>jdbc:firebirdsql://localhost:3050/examples</url>
    <user>SYSDBA</user>
    <password>masterkey</password>
    <properties>
      <property>
        <key>charSet</key>
        <value>utf-8</value>
      </property>
    </properties>
  </jdbc>

  <generator>
    <name>org.jooq.util.JavaGenerator</name>

    <database>
      <!-- The type of the database. Format:
           org.util.[database].[database]Database -->
      <name>org.jooq.util.firebird.FirebirdDatabase</name>

      <inputSchema></inputSchema>

      <!-- All objects that are generated from your schema
           (Java regular expression. Use filters to limit number of objects).
           Watch for sensitivity to the register. Depending on your database,
           this can be important!
       -->
      <includes>.*</includes>

      <!-- Objects that are excluded when generating from your schema.
```

```
           (Java regular expression).
           In this case, we exclude system tables RDB$, monitoring tables MON$
           and security pseudo-tables SEC$.
      -->
      <excludes>
          RDB\$.*
        | MON\$.*
        | SEC\$.*
      </excludes>
    </database>

    <target>
      <!-- The name of the package to which the generated -->
      <packageName>ru.ibase.fbjavaex.exampledb</packageName>

      <!-- Directory for posting the generated classes.
           Here, the Maven directory structure is used. -->
      <directory>e:/OpenServer/domains/localhost/fbjavaex/src/main/java/</directory>
    </target>
  </generator>
</configuration>
```

Generating the Schema Classes

In the the command shell, execute the following command to create the classes needed for writing queries to database objects in Java:

```
java -cp jooq-3.9.2.jar;jooq-meta-3.9.2.jar;jooq-codegen-3.9.2.jar;
jaybird-full-3.0.0.jar;. org.jooq.util.GenerationTool example.xml
```

You can find more details about the process of generating classes at https://www.jooq.org/doc/3.9/manual-single-page/#code-generation.

Dependency Injection

Dependency injection is a process whereby objects define their dependencies, that is, the other objects they work with. It is done only through constructor arguments, arguments to a factory method, or properties set or returned using a factory method. The container then injects those dependencies when it creates the bean. You can find more details about dependency injection at http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans.

Configuring IoC Containers for Dependency Injection

In Spring, dependency injection (DI) is carried out through the Spring IoC (Inversion of Control) container.

As before, we will avoid xml configuration and base our approach on annotations and Java configuration.

The main attributes and parts of the Java configuration of an IoC container are classes with the @Configuration annotation and methods with the @Bean annotation.

The @Bean Annotation

The @Bean annotation is used to define a method's activity in creating, configuring and initiating a new object controlled by the Spring IoC container. Methods so defined can be used the same way as classes with the @Configuration annotation.

Our IoC container will return

- the connection pool
- the transaction manager
- the exception translator thattranslates SQLException exceptions into Spring-specific DataAccessException exceptions
- the DSL context that is the starting point for building all queries using the Fluent API
- managers for implementing the business logic
- grids for displaying data

```
/**
 * IoC container configuration
 * to implement dependency injection.
 */

package ru.ibase.fbjavaex.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;
import org.apache.commons.dbcp.BasicDataSource;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy;
import org.jooq.impl.DataSourceConnectionProvider;
import org.jooq.DSLContext;
import org.jooq.impl.DefaultDSLContext;
import org.jooq.impl.DefaultConfiguration;
import org.jooq.SQLDialect;
import org.jooq.impl.DefaultExecuteListenerProvider;

import ru.ibase.fbjavaex.exception.ExceptionTranslator;

import ru.ibase.fbjavaex.managers.*;
import ru.ibase.fbjavaex.jqgrid.*;

/**
 * The Spring IoC configuration class of the container
 */
@Configuration
public class JooqConfig {


    /**
     * Return connection pool
     *
     * @return
     */
```

```
    @Bean(name = "dataSource")
    public DataSource getDataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        // ?????????? ???????????? ???????????
        dataSource.setUrl("jdbc:firebirdsql://localhost:3050/examples");
        dataSource.setDriverClassName("org.firebirdsql.jdbc.FBDriver");
        dataSource.setUsername("SYSDBA");
        dataSource.setPassword("masterkey");
        dataSource.setConnectionProperties("charSet=utf-8");
        return dataSource;
    }

    /**
     * Return transaction manager
     *
     * @return
     */
    @Bean(name = "transactionManager")
    public DataSourceTransactionManager getTransactionManager() {
        return new DataSourceTransactionManager(getDataSource());
    }

    @Bean(name = "transactionAwareDataSource")
    public TransactionAwareDataSourceProxy getTransactionAwareDataSource() {
        return new TransactionAwareDataSourceProxy(getDataSource());
    }

    /**
     * Return connection provider
     *
     * @return
     */
    @Bean(name = "connectionProvider")
    public DataSourceConnectionProvider getConnectionProvider() {
        return new DataSourceConnectionProvider(getTransactionAwareDataSource());
    }

    /**
     * Return exception translator
     *
     * @return
     */
    @Bean(name = "exceptionTranslator")
    public ExceptionTranslator getExceptionTranslator() {
        return new ExceptionTranslator();
    }

    /**
     * Returns the DSL context configuration
     *
     * @return
     */
    @Bean(name = "dslConfig")
    public org.jooq.Configuration getDslConfig() {
        DefaultConfiguration config = new DefaultConfiguration();
        // ?????????? ??????? SQL ???? Firebird
        config.setSQLDialect(SQLDialect.FIREBIRD);
        config.setConnectionProvider(getConnectionProvider());
```

```
        DefaultExecuteListenerProvider listenerProvider =
          new DefaultExecuteListenerProvider(getExceptionTranslator());
        config.setExecuteListenerProvider(listenerProvider);
        return config;
    }

    /**
     * Return DSL context
     *
     * @return
     */
    @Bean(name = "dsl")
    public DSLContext getDsl() {
        org.jooq.Configuration config = this.getDslConfig();
        return new DefaultDSLContext(config);
    }

    /**
     * Return customer manager
     *
     * @return
     */
    @Bean(name = "customerManager")
    public CustomerManager getCustomerManager() {
        return new CustomerManager();
    }

    /**
     * Return customer grid
     *
     * @return
     */
    @Bean(name = "customerGrid")
    public JqGridCustomer getCustomerGrid() {
        return new JqGridCustomer();
    }

    /**
     * Return product manager
     *
     * @return
     */
    @Bean(name = "productManager")
    public ProductManager getProductManager() {
        return new ProductManager();
    }

    /**
     * Return product grid
     *
     * @return
     */
    @Bean(name = "productGrid")
    public JqGridProduct getProductGrid() {
        return new JqGridProduct();
    }

    /**
```

```
     * Return invoice manager
     *
     * @return
     */
    @Bean(name = "invoiceManager")
    public InvoiceManager getInvoiceManager() {
        return new InvoiceManager();
    }

    /**
     * Return invoice grid
     *
     * @return
     */
    @Bean(name = "invoiceGrid")
    public JqGridInvoice getInvoiceGrid() {
        return new JqGridInvoice();
    }

    /**
     * Return invoice items grid
     *
     * @return
     */
    @Bean(name = "invoiceLineGrid")
    public JqGridInvoiceLine getInvoiceLineGrid() {
        return new JqGridInvoiceLine();
    }

    /**
     * Return working period
     *
     * @return
     */
    @Bean(name = "workingPeriod")
    public WorkingPeriod getWorkingPeriod() {
        return new WorkingPeriod();
    }

}
```

Creating SQL Queries Using jOOQ

Before we move on to the implementation of managers and grids, we will examine briefly how to work with the database via jOOQ. You can find the full documentation on this issue in the SQL-building section of the jOOQ documentation.

The *org.jooq.impl.DSL* class is the main one from which jOOQ objects are created. It acts as a static factory for table expressions, column (or field) expressions, conditional expressions and many other parts of a query.

DSLContext references the *org.jooq.Configuration* object that configures the behavior of jOOQ during the execution of queries. Unlike with static DSL, with DSLContext you can to create SQL statements that are already “configured” and ready for execution.

In our application, *DSLContext* is created in the *getDsl* method of the *JooqConfig* configuration class. Configuration for *DSLContext* is returned by the *getDslConfig* method. In this method we specify the Firebird dialect that we will use, the connection provider that determines how we get a connection via JDBC and the SQL query execution listener.

The jOOQ DSL

jOOQ comes with its own DSL (for *Domain Specific Language*) that emulates SQL in Java. It allows you to write SQL statements almost as though Java actually supported them. Its effect is similar to what .NET in C# does with LINQ to SQL.

jOOQ uses an informal BNF notation modelling a unified SQL dialect suitable for most database engines. Unlike other, simpler frameworks that use the Fluent API or the chain method, the jOOQ-based BNF interface does not permit bad query syntax.

```
A simple SQL query:
SELECT *
  FROM author a
  JOIN book b ON a.id = b.author_id
 WHERE a.year_of_birth > 1920
   AND a.first_name = 'Paulo'
 ORDER BY b.title
```

In jOOQ it looks like this:

```
Result<Record> result =
dsl.select()
   .from(AUTHOR.as("a"))
   .join(BOOK.as("b")).on(a.ID.equal(b.AUTHOR_ID))
   .where(a.YEAR_OF_BIRTH.greaterThan(1920)
   .and(a.FIRST_NAME.equal("Paulo")))
   .orderBy(b.TITLE)
   .fetch();
```

The *AUTHOR* and *BOOK* classes describing the corresponding tables must be generated beforehand. The process of generating jOOQ classes according to the specified database schema was described earlier.

We specified table aliases for the AUTHOR and BOOK tables using the AS clause. Here is the same query in DSL without aliases:

```
Result<Record> result =
dsl.select()
   .from(AUTHOR)
   .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
   .where(AUTHOR.YEAR_OF_BIRTH.greaterThan(1920)
   .and(AUTHOR.FIRST_NAME.equal("Paulo")))
   .orderBy(BOOK.TITLE)
   .fetch();
```

Now we take a more complex query with aggregate functions and grouping:

```
SELECT AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, COUNT(*)
FROM AUTHOR
  JOIN BOOK ON AUTHOR.ID = BOOK.AUTHOR_ID
WHERE BOOK.LANGUAGE = 'DE'
  AND BOOK.PUBLISHED > '2008-01-01'
GROUP BY AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME
  HAVING COUNT(*) > 5
ORDER BY AUTHOR.LAST_NAME ASC NULLS FIRST
  OFFSET 1 ROWS
  FETCH FIRST 2 ROWS ONLY
```

In jOOQ:

```
dsl.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
   .from(AUTHOR)
   .join(BOOK).on(BOOK.AUTHOR_ID.equal(AUTHOR.ID))
   .where(BOOK.LANGUAGE.equal("DE"))
   .and(BOOK.PUBLISHED.greaterThan("2008-01-01"))
   .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
   .having(count().greaterThan(5))
   .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
   .limit(2)
   .offset(1)
   .fetch();
```

> **Note**
>
> 'Dialect' in the jOOQ context represents not just the SQL dialect of the database but also the major version number of the database engine. The field 'limit', limiting the number of records returned, will be generated according to the SQL syntax available to the database engine. The example above used FIREBIRD_3_0, which supports OFFSET … FETCH. If we had specified the FIREBIRD_2_5 or just the FIREBIRD dialect, the ROWS clause would have been used instead.

You can build a query in parts. This will allow you to change it dynamically, to change the sort order or to add additional filter conditions.

```
SelectFinalStep<?> select
    = dsl.select()
         .from(PRODUCT);

SelectQuery<?> query = select.getQuery();
switch (searchOper) {
    case "eq":
         query.addConditions(PRODUCT.NAME.eq(searchString));
         break;
    case "bw":
         query.addConditions(PRODUCT.NAME.startsWith(searchString));
         break;
    case "cn":
         query.addConditions(PRODUCT.NAME.contains(searchString));
         break;
}
switch (sOrd) {
```

```
    case "asc":
         query.addOrderBy(PRODUCT.NAME.asc());
         break;
    case "desc":
         query.addOrderBy(PRODUCT.NAME.desc());
         break;
}
return query.fetchMaps();
```

Named and Unnamed Parameters

By default, any time you present a query containing a parameter that is string literal, a date, a number literal or an external variable, jOOQ uses unnamed parameters to bind that variable or literal. To illustrate, the following expression in Java:

```
dsl.select()
   .from(BOOK)
   .where(BOOK.ID.equal(5))
   .and(BOOK.TITLE.equal("Animal Farm"))
   .fetch();
```

is equivalent to the full form:

```
dsl.select()
   .from(BOOK)
   .where(BOOK.ID.equal(val(5)))
   .and(BOOK.TITLE.equal(val("Animal Farm")))
   .fetch();
```

and is converted into the SQL query:

```
SELECT *
FROM BOOK
WHERE BOOK.ID = ?
  AND BOOK.TITLE = ?
```

You need not concern yourself with the index position of the field value that corresponds to a parameter, as the values will be bound to the appropriate parameter automatically. The index of the parameter list is 1-based. If you need to change the value of a parameter, you just select it by its index number.

```
Select<?> select =
  dsl.select()
     .from(BOOK)
     .where(BOOK.ID.equal(5))
     .and(BOOK.TITLE.equal("Animal Farm"));
Param<?> param = select.getParam("2");
Param.setValue("Animals as Leaders");
```

Another way to assign a new value to a parameter is to call the *bind* method:

```
Query query1 =
  dsl.select()
     .from(AUTHOR)
     .where(LAST_NAME.equal("Poe"));
query1.bind(1, "Orwell");
```

jOOQ supports **named parameters**, too. They need to be created explicitly using *org.jooq.Param*:

```
// Create a query with a named parameter. You can then use that name for
// accessing the parameter again
Query query1 =
  dsl.select()
     .from(AUTHOR)
     .where(LAST_NAME.equal(param("lastName", "Poe")));
Param<?> param1 = query.getParam("lastName");

// Or, keep a reference to the typed parameter in order
// not to lose the <T> type information:
Param<String> param2 = param("lastName", "Poe");
Query query2 =
  dsl.select()
     .from(AUTHOR)
     .where(LAST_NAME.equal(param2));

// You can now change the bind value directly on the Param reference:
param2.setValue("Orwell");

Another way to assign a new value to a parameter is to call the bind method:
// Or, with named parameters
Query query2 =
  dsl.select()
     .from(AUTHOR)
     .where(LAST_NAME.equal(param("lastName", "Poe")));
query2.bind("lastName", "Orwell");
```

Returning Values from SELECT Queries

jOOQ offers several methods for fetching data from SQL queries. We are not covering all of them here but you can find more details about them in the Fetching section of the jOOQ documentation.

For our example, we will return the data to a map list (the *fetchMaps* method) which is handy to use for serializing a result for JSON.

Other Types of Queries

We'll take a look at other types of queries. This query inserts a record:

```
INSERT INTO AUTHOR
       (ID, FIRST_NAME, LAST_NAME)
VALUES (100, 'Hermann', 'Hesse');
```

In jOOQ:

```
dsl.insertInto(AUTHOR,
        AUTHOR.ID, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
   .values(100, "Hermann", "Hesse")
   .execute();
```

A query to update a record:

```
UPDATE AUTHOR
   SET FIRST_NAME = 'Hermann',
       LAST_NAME = 'Hesse'
 WHERE ID = 3;
```

In jOOQ:

```
dsl.update(AUTHOR)
   .set(AUTHOR.FIRST_NAME, "Hermann")
   .set(AUTHOR.LAST_NAME, "Hesse")
   .where(AUTHOR.ID.equal(3))
   .execute();
```

A query to delete a record:

```
DELETE FROM AUTHOR
 WHERE ID = 100;
```

In jOOQ:

```
dsl.delete(AUTHOR)
   .where(AUTHOR.ID.equal(100))
   .execute();
```

More complex update queries can be built in jOOQ, such as a MERGE query, for example.

Stored Procedures with jOOQ

A great benefit of jOOQ is its support for working with stored procedures. Stored procedures are extracted to the **.Routines.** package. From there, you can work with them easily. For instance, the following code in Java:

```
int invoiceId = dsl.nextval(GEN_INVOICE_ID).intValue();
```

```
spAddInvoice(dsl.configuration(),
             invoiceId,
             customerId,
             invoiceDate);
```

is equivalent to getting the next value of the generator using the following SQL query:

```
SELECT NEXT VALUE FOR GEN_INVOICE_ID
  FROM RDB$DATABASE
```

and calling the stored procedure after that:

```
EXECUTE PROCEDURE SP_ADD_INVOICE   (
   :INVOICE_ID, :CUSTOMER_ID, :INVOICE_DATE );
```

jOOQ also provides tools to build simple DDL queries, but we do not cover them here.

Working with Transactions

By default, jOOQ runs in a mode that commits transactions automatically. It starts a new transaction for each SQL statement and commits the transaction if there are no errors in the execution of the statement. The transaction is rolled back if an error occurs.

The default transaction has the following parameters: READ_WRITE READ_COMMITTED REC_VERSION WAIT, the same parameters that are used by the JDBC driver. You can change the default isolation mode using the parameters of the connection pool—see *BasicDataSource.setDefaultTransactionIsolation* in the *getDataSource* method of the *JooqConfig* configuration class.

Explicit Transactions

In jOOQ you have several ways to control transactions explicitly. Since we are going to develop our application using the Spring Framework, we will use the transaction manager specified in the configuration (JooqConfig). You can get the transaction manager by declaring the *txMgr* property in the class as follows:

```
@Autowired
private DataSourceTransactionManager txMgr;
```

The standard scenario for using this technique with a transaction would be coded like this:

```
TransactionStatus tx = txMgr.getTransaction(new DefaultTransactionDefinition());
try {
    // actions in the context of a transaction
    for (int i = 0; i < 2; i++)
```

```
        dsl.insertInto(BOOK)
            .set(BOOK.ID, 5)
            .set(BOOK.AUTHOR_ID, 1)
            .set(BOOK.TITLE, "Book 5")
            .execute();
    // transaction commit
    txMgr.commit(tx);
}
catch (DataAccessException e) {
    // transaction rollback
    txMgr.rolback(tx);
}
```

However, Spring enables that scenario to be implemented much more easily using the *@Transactional* annotation specified before the method of the class. Thereby, all actions performed by the method will be wrapped in the transaction.

```
    /**
     * Delete customer
     *
     * @param customerId
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void delete(int customerId) {
        this.dsl.deleteFrom(CUSTOMER)
                .where(CUSTOMER.CUSTOMER_ID.eq(customerId))
                .execute();
    }
```

Transaction Parameters

Propagation

The propagation parameter defines how to work with transactions if our method is called from an external transaction.

- Propagation.REQUIRED—execute in the existing transaction if there is one. Otherwise, create a new one.
- Propagation.MANDATORY—execute in the existing transaction if there is one. Otherwise, raise an exception.
- Propagation.SUPPORTS—execute in the existing transaction if there is one. Otherwise, execute outside the transaction.
- Propagation.NOT_SUPPORTED—always execute outside the transaction. If there is an existing one, it will be suspended.
- Propagation.REQUIRES_NEW—always execute in a new independent transaction. If there is an existing one, it will be suspended until the new transaction is ended.
- Propagation.NESTED—if there is an existing transaction, execute in a new so-called “nested” transaction. If the nested transaction is rolled back, it will not affect the external transaction; if the external transaction is rolled back, the nested one will be rolled back as well. If there is no existing transaction, a new one is simply created.
- Propagation.NEVER—always execute outside the transaction. Raise an exception if there is an existing one.

Isolation Level

The isolation parameter defines the isolation level. Five values are supported: DEFAULT, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE. If the DEFAULT value of the *isolation* parameter is specified, that level will be used.

The other isolation levels are taken from the SQL standard, not all of them supported exactly by Firebird. Only the READ_COMMITED level corresponds in all of the criteria, so JDBC READ_COMMITTED is mapped into read_committed in Firebird. REPEATABLE_READ is mapped into concurrency (SNAPSHOT) and SERIALIZABLE is mapped into consistency (SNAPSHOT TABLE STABILITY).

Firebird supports additional transaction parameters besides isolation level, viz. NO RECORD_VERSION/ RECORD_VERSION (applicable only to a transaction with READ COMMITTED isolation) and WAIT/ NO WAIT. The standard isolation levels can be mapped to Firebird transaction parameters by specifying the properties of the JDBC connection (see more details in the *Using Transactions* chapter of Jaybird 2.1 JDBC driver Java Programmer's Manual.

If your transaction works with more than one query, it is recommended to use the REPEATABLE_READ isolation level to maintain data consistency.

Read Mode

By default, a transaction is in the read-write mode. The *readOnly* property in the *@Transactional* annotation can be used to specify that it is to be read-only.

Writing the Application Code

We will display the data of our application using the JavaScript component jqGrid. Currently, jqGrid is distributed under a commercial licence, but it is free for non-commercial purposes. You can use the free-jqGrid fork instead.

To display data and page-by-page navigation elements in this grid, we need to return data in the JSON format, the structure of which looks like this:

```
{
    total: 100,
    page: 3,
    records: 3000,
    rows: [
       {id: 1, name: "Ada"},
       {id: 2, name: "Smith"},
      …
    ]
}
```

where

- total—the total number of pages
- page—the number of the current page
- records—the total number of records

- rows—the count of records on the current page array

The following code creates a class demonstrating this structure:

```
package ru.ibase.fbjavaex.jqgrid;

import java.util.List;
import java.util.Map;

/**
 * A class describing the structure that is used in jqGrid
 * Designed for JSON serialization
 *
 * @author Simonov Denis
 */
public class JqGridData {

    /**
     * Total number of pages
     */
    private final int total;

    /**
     * The current page number
     */
    private final int page;

    /**
     * Total number of records
     */
    private final int records;

    /**
     * The actual data
     */
    private final List<Map<String, Object>> rows;

    /**
     * Constructor
     *
     * @param total
     * @param page
     * @param records
     * @param rows
     */
    public JqGridData(int total, int page, int records,
                      List<Map<String, Object>> rows) {
        this.total = total;
        this.page = page;
        this.records = records;
        this.rows = rows;
    }

    /**
     * Returns the total number of pages
     *
     * @return
```

```
     */
    public int getTotal() {
        return total;
    }

    /**
     * Returns the current page
     *
     * @return
     */
    public int getPage() {
        return page;
    }

    /**
     * Returns the total number of records
     *
     * @return
     */
    public int getRecords() {
        return records;
    }

    /**
     * Return list of map
     * This is an array of data to display in the grid
     *
     * @return
     */
    public List<Map<String, Object>> getRows() {
        return rows;
    }
}
```

Now we will write an abstract class that will return that structure depending on the search and sorting conditions. It will be a parent class for the entity-specific classes that return similar structures.

```
/*
 * Abstract class for working with JqGrid
 */
package ru.ibase.fbjavaex.jqgrid;

import java.util.Map;
import java.util.List;
import org.jooq.DSLContext;
import org.springframework.beans.factory.annotation.Autowired;

/**
 * Working with JqGrid
 *
 * @author Simonov Denis
 */
public abstract class JqGrid {


    @Autowired(required = true)
```

```
    protected DSLContext dsl;

    protected String searchField = "";
    protected String searchString = "";
    protected String searchOper = "eq";
    protected Boolean searchFlag = false;
    protected int pageNo = 0;
    protected int limit = 0;
    protected int offset = 0;
    protected String sIdx = "";
    protected String sOrd = "asc";

    /**
     * Returns the total number of records
     *
     * @return
     */
    public abstract int getCountRecord();

    /**
     * Returns the structure for JSON serialization
     *
     * @return
     */
    public JqGridData getJqGridData() {
        int recordCount = this.getCountRecord();
        List<Map<String, Object>> records = this.getRecords();

        int total = 0;
        if (this.limit > 0) {
            total = recordCount / this.limit + 1;
        }

        JqGridData jqGridData = new JqGridData(
            total,
            this.pageNo,
            recordCount,
            records);
        return jqGridData;
    }


    /**
     * Returns the number of records per page
     *
     * @return
     */
    public int getLimit() {
        return this.limit;
    }

    /**
     * Returns the offset to retrieve the first record on the page
     *
     * @return
     */
    public int getOffset() {
        return this.offset;
```

```
    }

    /**
     * Returns field name for sorting
     *
     * @return
     */
    public String getIdx() {
        return this.sIdx;
    }

    /**
     * Returns the sort order
     *
     * @return
     */
    public String getOrd() {
        return this.sOrd;
    }

    /**
     * Returns the current page number
     *
     * @return
     */
    public int getPageNo() {
        return this.pageNo;
    }

    /**
     * Returns an array of records as a list of maps
     *
     * @return
     */
    public abstract List<Map<String, Object>> getRecords();

    /**
     * Returns field name for search
     *
     * @return
     */
    public String getSearchField() {
        return this.searchField;
    }

    /**
     * Returns value for search
     *
     * @return
     */
    public String getSearchString() {
        return this.searchString;
    }

    /**
     * Returns the search operation
     *
     * @return
```

```
     */
    public String getSearchOper() {
        return this.searchOper;
    }

    /**
     * Sets the limit on the number of display records
     *
     * @param limit
     */
    public void setLimit(int limit) {
        this.limit = limit;
    }

    /**
     * Sets the number of records to skip
     *
     * @param offset
     */
    public void setOffset(int offset) {
        this.offset = offset;
    }

    /**
     * Sets the sorting
     *
     * @param sIdx
     * @param sOrd
     */
    public void setOrderBy(String sIdx, String sOrd) {
        this.sIdx = sIdx;
        this.sOrd = sOrd;
    }

    /**
     * Sets the current page number
     *
     * @param pageNo
     */
    public void setPageNo(int pageNo) {
        this.pageNo = pageNo;
        this.offset = (pageNo - 1) * this.limit;
    }

    /**
     * Sets the search condition
     *
     * @param searchField
     * @param searchString
     * @param searchOper
     */
    public void setSearchCondition(String searchField, String searchString,
                                   String searchOper) {
        this.searchFlag = true;
        this.searchField = searchField;
        this.searchString = searchString;
        this.searchOper = searchOper;
    }
```

```
}
```

> **Note**
>
> Notice that this class contains the DSLContext dsl property that will be used to build jOOQ queries for retrieving data.

Creating the Primary Modules

Now we can start creating modules. The process of creating modules is described here, using the customer module as an example. Creating the product module is similar and, if you are interested, you can examine its source code in the .zip download linked at the end of this chapter.

First, we implement a class for working with jqGrid, inheriting it from our abstract class *ru.ibase.fbjavaex.jqgrid.JqGrid*. It will be able to search and sort by the NAME field in reversing order. Track the source code below for explanatory comments.

```
package ru.ibase.fbjavaex.jqgrid;

import org.jooq.*;
import java.util.List;
import java.util.Map;

import static ru.ibase.fbjavaex.exampledb.Tables.CUSTOMER;

/**
 * Customer grid
 *
 * @author Simonov Denis
 */
public class JqGridCustomer extends JqGrid {

    /**
     * Adding a search condition
     *
     * @param query
     */
    private void makeSearchCondition(SelectQuery<?> query) {
        switch (this.searchOper) {
            case "eq":
                // CUSTOMER.NAME = ?
               query.addConditions(CUSTOMER.NAME.eq(this.searchString));
               break;
            case "bw":
                // CUSTOMER.NAME STARTING WITH ?
               query.addConditions(CUSTOMER.NAME.startsWith(this.searchString));
               break;
            case "cn":
                // CUSTOMER.NAME CONTAINING ?
               query.addConditions(CUSTOMER.NAME.contains(this.searchString));
               break;
        }
```

```
    }


    /**
     * Returns the total number of records
     *
     * @return
     */
    @Override
    public int getCountRecord() {
        // query that returns the number of records
        SelectFinalStep<?> select
            = dsl.selectCount()
                 .from(CUSTOMER);

        SelectQuery<?> query = select.getQuery();
        // if perform a search, then add the search condition
        if (this.searchFlag) {
            makeSearchCondition(query);
        }

        return (int) query.fetch().getValue(0, 0);
    }

    /**
     * Returns the grid records
     *
     * @return
     */
    @Override
    public List<Map<String, Object>> getRecords() {
        // Basic selection query
        SelectFinalStep<?> select =
            dsl.select()
               .from(CUSTOMER);

        SelectQuery<?> query = select.getQuery();
        // if perform a search, then add the search condition
        if (this.searchFlag) {
            makeSearchCondition(query);
        }
        // set the sort order
        switch (this.sOrd) {
            case "asc":
                query.addOrderBy(CUSTOMER.NAME.asc());
                break;
            case "desc":
                query.addOrderBy(CUSTOMER.NAME.desc());
                break;
        }
        // limit the number of records
        if (this.limit != 0) {
            query.addLimit(this.limit);
        }

        if (this.offset != 0) {
            query.addOffset(this.offset);
        }
```

```
        // return an array of maps
        return query.fetchMaps();
    }
}
```

CustomerManager Class

The CustomerManager class that is defined next is a kind of business layer between the corresponding controller and the database. We will use it for adding, editing and deleting a customer. All operations in this layer will be performed in a SNAPSHOT-level transaction.

```
package ru.ibase.fbjavaex.managers;

import org.jooq.DSLContext;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Isolation;

import static ru.ibase.fbjavaex.exampledb.Tables.CUSTOMER;
import static ru.ibase.fbjavaex.exampledb.Sequences.GEN_CUSTOMER_ID;

/**
 * Customer manager
 *
 * @author Simonov Denis
 */
public class CustomerManager {

    @Autowired(required = true)
    private DSLContext dsl;

    /**
     * Adding a customer
     *
     * @param name
     * @param address
     * @param zipcode
     * @param phone
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void create(String name, String address, String zipcode, String phone) {
        if (zipcode != null) {
            if (zipcode.trim().isEmpty()) {
                zipcode = null;
            }
        }

        int customerId = this.dsl.nextval(GEN_CUSTOMER_ID).intValue();

        this.dsl
                .insertInto(CUSTOMER,
                        CUSTOMER.CUSTOMER_ID,
```

```
                        CUSTOMER.NAME,
                        CUSTOMER.ADDRESS,
                        CUSTOMER.ZIPCODE,
                        CUSTOMER.PHONE)
                .values(
                        customerId,
                        name,
                        address,
                        zipcode,
                        phone
                )
                .execute();
    }

    /**
     * Editing a customer
     *
     * @param customerId
     * @param name
     * @param address
     * @param zipcode
     * @param phone
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void edit(int customerId, String name, String address,
                     String zipcode, String phone) {

        if (zipcode != null) {
            if (zipcode.trim().isEmpty()) {
                zipcode = null;
            }
        }

        this.dsl.update(CUSTOMER)
                .set(CUSTOMER.NAME, name)
                .set(CUSTOMER.ADDRESS, address)
                .set(CUSTOMER.ZIPCODE, zipcode)
                .set(CUSTOMER.PHONE, phone)
                .where(CUSTOMER.CUSTOMER_ID.eq(customerId))
                .execute();
    }

    /**
     * Deleting a customer
     *
     * @param customerId
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void delete(int customerId) {
        this.dsl.deleteFrom(CUSTOMER)
                .where(CUSTOMER.CUSTOMER_ID.eq(customerId))
                .execute();
    }
}
```

Customer Controller Class

Controller classes start with the *@Controller* annotation. The *@RequestMapping* annotation preceding the method is necessary for directing the actions of the controller, for specifying the path that will be used to call the action.

- The path is specified in the *value* attribute
- The *method* attribute specifies the HTTP request method (PUT, GET, POST, DELETE)
- The *index* method will be the input point of our controller. It is responsible for displaying the JSP page (view) that contains the layout for displaying the grid, the tool bar and the navigation bar.

Data for display are loaded asynchronously by the jqGrid component. The path is `/customer/getdata`, to which the *getData* method is connected.

getData Method

The *getData* method contains the additional *@ResponseBody* annotation for indicating that our method returns the object for serialization into a specific format. The annotation *@RequestMapping* contains the attribute *produces = MediaType.APPLICATION_JSON*, directing that the returned object be serialized into the JSON format.

It is in the *getData* method that we work with the *JqGridCustomer* class described earlier. The *@RequestParam* annotation enables the value of the parameter to be retrieved from the HTTP request. This class method works with GET requests.

- The *value* attribute in the *@RequestParam* annotation defines the name of the parameter to be retrieved from the HTTP request.
- The *Required* attribute can designate the HTTP request parameter as mandatory.
- The *defaultValue* attribute supplies the value that is to be used if the HTTP parameter is not specified.

Customer Action Methods

The *addCustomer* method is used to add a new customer. It is connected with the `/customer/create` path and, unlike the previous method, it works with the POST request. The method returns {success: true} if the customer is added successfully. If an error occurs, it returns an object with the error message. The *addCustomer* method works with the *CustomerManager* business layer method.

The *editCustomer* method is connected with the `/customer/edit` path. The *deleteCustomer* method is connected with the `/customer/delete` path. Both methods operate on existing customer records.

```
package ru.ibase.fbjavaex.controllers;

import java.util.HashMap;
import java.util.Map;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```

```
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestParam;
import javax.ws.rs.core.MediaType;

import org.springframework.beans.factory.annotation.Autowired;

import ru.ibase.fbjavaex.managers.CustomerManager;

import ru.ibase.fbjavaex.jqgrid.JqGridCustomer;
import ru.ibase.fbjavaex.jqgrid.JqGridData;


/**
 * Customer Controller
 *
 * @author Simonov Denis
 */
@Controller
public class CustomerController {


    @Autowired(required = true)
    private JqGridCustomer customerGrid;

    @Autowired(required = true)
    private CustomerManager customerManager;


    /**
     * Default action
     * Returns the JSP name of the page (view) to display
     *
     * @param map
     * @return name of JSP template
     */
    @RequestMapping(value = "/customer/", method = RequestMethod.GET)
    public String index(ModelMap map) {
        return "customer";
    }

    /**
     * Returns JSON data for jqGrid
     *
     * @param rows number of entries per page
     * @param page page number
     * @param sIdx sorting field
     * @param sOrd sorting order
     * @param search should the search be performed
     * @param searchField search field
     * @param searchString value for searching
     * @param searchOper search operation
     * @return JSON data for jqGrid
     */
    @RequestMapping(value = "/customer/getdata",
            method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public JqGridData getData(
```

```
            // number of entries per page
            @RequestParam(value = "rows", required = false,
                          defaultValue = "20") int rows,
            // page number
            @RequestParam(value = "page", required = false,
                          defaultValue = "1") int page,
            // sorting field
            @RequestParam(value = "sidx", required = false,
                          defaultValue = "") String sIdx,
            // sorting order
            @RequestParam(value = "sord", required = false,
                          defaultValue = "asc") String sOrd,
            // should the search be performed
            @RequestParam(value = "_search", required = false,
                          defaultValue = "false") Boolean search,
            // search field
            @RequestParam(value = "searchField", required = false,
                          defaultValue = "") String searchField,
            // value for searching
            @RequestParam(value = "searchString", required = false,
                          defaultValue = "") String searchString,
            // search operation
            @RequestParam(value = "searchOper", required = false,
                          defaultValue = "") String searchOper,
            // filters
            @RequestParam(value="filters", required=false,
                          defaultValue="") String filters) {
        customerGrid.setLimit(rows);
        customerGrid.setPageNo(page);
        customerGrid.setOrderBy(sIdx, sOrd);
        if (search) {
            customerGrid.setSearchCondition(searchField, searchString, searchOper);
        }

        return customerGrid.getJqGridData();
    }

    @RequestMapping(value = "/customer/create",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> addCustomer(
            @RequestParam(value = "NAME", required = true,
                          defaultValue = "") String name,
            @RequestParam(value = "ADDRESS", required = false,
                          defaultValue = "") String address,
            @RequestParam(value = "ZIPCODE", required = false,
                          defaultValue = "") String zipcode,
            @RequestParam(value = "PHONE", required = false,
                          defaultValue = "") String phone) {
        Map<String, Object> map = new HashMap<>();
        try {
            customerManager.create(name, address, zipcode, phone);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
```

```
    }

    @RequestMapping(value = "/customer/edit",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> editCustomer(
            @RequestParam(value = "CUSTOMER_ID", required = true,
                          defaultValue = "0") int customerId,
            @RequestParam(value = "NAME", required = true,
                          defaultValue = "") String name,
            @RequestParam(value = "ADDRESS", required = false,
                          defaultValue = "") String address,
            @RequestParam(value = "ZIPCODE", required = false,
                          defaultValue = "") String zipcode,
            @RequestParam(value = "PHONE", required = false,
                          defaultValue = "") String phone) {
        Map<String, Object> map = new HashMap<>();
        try {
            customerManager.edit(customerId, name, address, zipcode, phone);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    @RequestMapping(value = "/customer/delete",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> deleteCustomer(
            @RequestParam(value = "CUSTOMER_ID", required = true,
                          defaultValue = "0") int customerId) {
        Map<String, Object> map = new HashMap<>();
        try {
            customerManager.delete(customerId);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }
}
```

Customer Display

The JSP page for displaying the customer module contains nothing special: the layout with the main parts of the page, the table for displaying the grid and the block for displaying the navigation bar. JSP templates are fairly unsophisticated. If you wish, you can replace them with other template systems that support inheritance.

The `../jspf/head.jspf` file contains common scripts and styles for all website pages and the `../jspf/menu.jspf` file contains the website's main menu. Their code is not reproduced here: it is quite simple and you can examine it in the project's source if you are curious.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:set var="cp" value="${pageContext.request.servletContext.contextPath}"
       scope="request" />

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>An example of a Spring MVC application using Firebird
               and jOOQ</title>

        <!-- Scripts and styles -->
        <%@ include file="../jspf/head.jspf" %>
        <script src="${cp}/resources/js/jqGridCustomer.js"></script>
    </head>
    <body>
        <!-- Navigation menu -->
        <%@ include file="../jspf/menu.jspf" %>

        <div class="container body-content">

            <h2>Customers</h2>

            <table id="jqGridCustomer"></table>
            <div id="jqPagerCustomer"></div>

            <hr/>
            <footer>
                <p>© 2016 - An example of a Spring MVC application
                using Firebird and jOOQ</p>
            </footer>
        </div>

<script type="text/javascript">
    $(document).ready(function () {
        JqGridCustomer({
            baseAddress: '${cp}'
        });
    });
</script>

    </body>
</html>
```

The basic logic on the client side is concentrated in the `/resources/js/jqGridCustomer.js` JavaScript module.

```
var JqGridCustomer = (function ($) {

  return function (options) {
      var jqGridCustomer = {
          dbGrid: null,
          options: $.extend({
              baseAddress: null,
              showEditorPanel: true
```

```
        }, options),
        // return model description
        getColModel: function () {
            return [
                {
                    label: 'Id',
                    name: 'CUSTOMER_ID', // field name
                    key: true,
                    hidden: true
                },
                {
                    label: 'Name',
                    name: 'NAME',
                    width: 240,
                    sortable: true,
                    editable: true,
                    edittype: "text", // input field type in the editor
                    search: true,
                    searchoptions: {
                        // allowed search operators
                        sopt: ['eq', 'bw', 'cn']
                    },
                    // size and maximum length for the input field
                    editoptions: {size: 30, maxlength: 60},
                    editrules: {required: true}
                },
                {
                    label: 'Address',
                    name: 'ADDRESS',
                    width: 300,
                    sortable: false, // prohibit sorting
                    editable: true,
                    search: false, // prohibit search
                    edittype: "textarea", // Memo field
                    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}
                }
            ];
        },
```

```
        // grid initialization
        initGrid: function () {
            // url to retrieve data
            var url = jqGridCustomer.options.baseAddress
                    + '/customer/getdata';
            jqGridCustomer.dbGrid = $("#jqGridCustomer").jqGrid({
                url: url,
                datatype: "json", // data format
                mtype: "GET", // request type
                colModel: jqGridCustomer.getColModel(),
                rowNum: 500, // number of rows displayed
                loadonce: false, // load only once
                sortname: 'NAME', // Sorting by NAME by default
                sortorder: "asc",
                width: window.innerWidth - 80,
                height: 500,
                viewrecords: true, // display the number of records
                guiStyle: "bootstrap",
                iconSet: "fontAwesome",
                caption: "Customers",
                // navigation item
                pager: 'jqPagerCustomer'
            });
        },
        // editing options
        getEditOptions: function () {
          return {
            url: jqGridCustomer.options.baseAddress + '/customer/edit',
            reloadAfterSubmit: true,
            closeOnEscape: true,
            closeAfterEdit: true,
            drag: true,
            width: 400,
            afterSubmit: jqGridCustomer.afterSubmit,
            editData: {
              // In addition to the values from the form, pass the key field
              CUSTOMER_ID: function () {
                // get the current row
                var selectedRow = jqGridCustomer.dbGrid.getGridParam("selrow");
                // get the value of the field CUSTOMER_ID
                var value = jqGridCustomer.dbGrid.getCell(selectedRow,
                            'CUSTOMER_ID');
                return value;
              }
            }
          };
        },
        // Add options
        getAddOptions: function () {
          return {
            url: jqGridCustomer.options.baseAddress + '/customer/create',
            reloadAfterSubmit: true,
            closeOnEscape: true,
            closeAfterAdd: true,
            drag: true,
            width: 400,
            afterSubmit: jqGridCustomer.afterSubmit
          };
```

```
        },
        // Edit options
        getDeleteOptions: function () {
          return {
            url: jqGridCustomer.options.baseAddress + '/customer/delete',
            reloadAfterSubmit: true,
            closeOnEscape: true,
            closeAfterDelete: true,
            drag: true,
            msg: "Delete the selected customer?",
            afterSubmit: jqGridCustomer.afterSubmit,
            delData: {
              // pass the key field
              CUSTOMER_ID: function () {
                var selectedRow = jqGridCustomer.dbGrid.getGridParam("selrow");
                var value = jqGridCustomer.dbGrid.getCell(selectedRow,
                            'CUSTOMER_ID');
                return value;
              }
            }
          };
        },
        // initializing the navigation bar with editing dialogs
        initPagerWithEditors: function () {
            jqGridCustomer.dbGrid.jqGrid('navGrid', '#jqPagerCustomer',
                {
                     // buttons
                     search: true,
                     add: true,
                     edit: true,
                     del: true,
                     view: true,
                     refresh: true,
                     // button captions
                     searchtext: "Search",
                     addtext: "Add",
                     edittext: "Edit",
                     deltext: "Delete",
                     viewtext: "View",
                     viewtitle: "Selected record",
                     refreshtext: "Refresh"
                },
                jqGridCustomer.getEditOptions(),
                jqGridCustomer.getAddOptions(),
                jqGridCustomer.getDeleteOptions()
           );
        },
        // initialize the navigation bar without editing dialogs
        initPagerWithoutEditors: function () {
            jqGridCustomer.dbGrid.jqGrid('navGrid', '#jqPagerCustomer',
                {
                     // buttons
                     search: true,
                     add: false,
                     edit: false,
                     del: false,
                     view: false,
                     refresh: true,
```

```
                    // button captions
                    searchtext: "Search",
                    viewtext: "View",
                    viewtitle: "Selected record",
                    refreshtext: "Refresh"
                }
            );
        },
        // initialize the navigation bar
        initPager: function () {
            if (jqGridCustomer.options.showEditorPanel) {
                jqGridCustomer.initPagerWithEditors();
            } else {
                jqGridCustomer.initPagerWithoutEditors();
            }
        },
        // initialize
        init: function () {
            jqGridCustomer.initGrid();
            jqGridCustomer.initPager();
        },
        // processor of the results of processing 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 {
                // if an error was not returned, refresh the grid
                $(this).jqGrid(
                        'setGridParam',
                        {
                            datatype: 'json'
                        }
                ).trigger('reloadGrid');
            }
            return [true, "", 0];
        }
    };
    jqGridCustomer.init();
    return jqGridCustomer;
  };
})(jQuery);
```

Visual Elements

The jqGrid grid

is created in the *initGrid* method and is bound to the *html* element with the jqGridCustomer identifier. The grid column desciptions are returned by the *getColModel* method.

Each column in jqGrid has a number of properties available. The source code contains comments explaining column properties. You can read more details about configuring the model of jqGrid columns in the ColModel API section of the documentation for the jqGrid project.

The navigation bar
: can be created either with edit buttons or without them, using the *initPagerWithEditors* and *initPagerWithoutEditors* methods, respectively. The bar constructor binds it to the element with the jqPagerCustomer identifier. The options for creating the navigation bar are described in the Navigator section of the jqGrid documentation.

Functions and Settings for Options
: The *getEditOptions*, *getAddOptions*, *getDeleteOptions* functions return the options for the edit, add and delete dialog boxes, respectively.

  The *url* property defines the URL to which the data will be submitted after the OK button in clicked in the dialog box.

  The *afterSubmit* property marks the event that occurs after the data have been sent to the server and a response has been received back.

  The *afterSubmit* method checks whether the controller returns an error. The grid is updated if no error is returned; otherwise, the error is shown to the user.

> **Note**
>
> The editData property allows you to specify the values of additional fields that are not shown in the edit dialog box. Edit dialog boxes do not show the values of hidden fields and it is rather tedious if you want to display automatically generated keys.

Creating Secondary Modules

A secondary module typically contains many more records than a primary one and new records are added frequently. Most secondary tables contain a field with the record creation date. In order to reduce the amount of retrieved data, the notion of a *work period* is often incorporated to limit the range of data sent to the client. A work period is a range of dates for which the records are required. The work period is described by the *WorkingPeriod* class, defined via the workingPeriod bean in the *ru.ibase.fbjavaex.config.JooqConfig* configuration class.

```
package ru.ibase.fbjavaex.config;

import java.sql.Timestamp;
import java.time.LocalDateTime;

/**
 * Working period
 *
 * @author Simonov Denis
 */
public class WorkingPeriod {

    private Timestamp beginDate;
    private Timestamp endDate;

    /**
     * Constructor
     */
```

```
    WorkingPeriod() {
        // in real applications is calculated from the current date
        this.beginDate = Timestamp.valueOf("2015-06-01 00:00:00");
        this.endDate = Timestamp.valueOf(LocalDateTime.now().plusDays(1));
    }

    /**
     * Returns the start date of the work period
     *
     * @return
     */
    public Timestamp getBeginDate() {
        return this.beginDate;
    }

    /**
     * Returns the end date of the work period
     *
     * @return
     */
    public Timestamp getEndDate() {
        return this.endDate;
    }

    /**
     * Setting the start date of the work period
     *
     * @param value
     */
    public void setBeginDate(Timestamp value) {
        this.beginDate = value;
    }

    /**
     * Setting the end date of the work period
     *
     * @param value
     */
    public void setEndDate(Timestamp value) {
        this.endDate = value;
    }

    /**
     * Setting the working period
     *
     * @param beginDate
     * @param endDate
     */
    public void setRangeDate(Timestamp beginDate, Timestamp endDate) {
        this.beginDate = beginDate;
        this.endDate = endDate;
    }
}
```

In our project we have only one secondary module called "Invoices". An invoice consists of a header where some general attributes are described (number, date, customer …) and one or more invoice items (product name, quantity, price, etc.). The invoice header is displayed in the main grid while items can be viewed in a detail grid that is opened with a click on the "+" icon of the selected document.

We implement a class, inherited from the *ru.ibase.fbjavaex.jqgrid.JqGrid* abstract class described earlier, for viewing the invoice headers via jqGrid. Searching can be by customer name or invoice date and reversible date order is supported, too.

```
package ru.ibase.fbjavaex.jqgrid;

import java.sql.*;
import org.jooq.*;

import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import ru.ibase.fbjavaex.config.WorkingPeriod;

import static ru.ibase.fbjavaex.exampledb.Tables.INVOICE;
import static ru.ibase.fbjavaex.exampledb.Tables.CUSTOMER;

/**
 * Grid handler for the invoice journal
 *
 * @author Simonov Denis
 */
public class JqGridInvoice extends JqGrid {

    @Autowired(required = true)
    private WorkingPeriod workingPeriod;

    /**
     * Adding a search condition
     *
     * @param query
     */
    private void makeSearchCondition(SelectQuery<?> query) {
        // adding a search condition to the query,
        // if it is produced for different fields,
        // different comparison operators are available when searching.
        if (this.searchString.isEmpty()) {
            return;
        }

        if (this.searchField.equals("CUSTOMER_NAME")) {
            switch (this.searchOper) {
               case "eq": // equal
                 query.addConditions(CUSTOMER.NAME.eq(this.searchString));
                 break;
               case "bw": // starting with
                 query.addConditions(CUSTOMER.NAME.startsWith(this.searchString));
                 break;
               case "cn": // containing
                 query.addConditions(CUSTOMER.NAME.contains(this.searchString));
                 break;
            }
        }
        if (this.searchField.equals("INVOICE_DATE")) {
```

```
            Timestamp dateValue = Timestamp.valueOf(this.searchString);

            switch (this.searchOper) {
               case "eq": // =
                 query.addConditions(INVOICE.INVOICE_DATE.eq(dateValue));
                 break;
               case "lt": // <
                 query.addConditions(INVOICE.INVOICE_DATE.lt(dateValue));
                 break;
               case "le": // <=
                 query.addConditions(INVOICE.INVOICE_DATE.le(dateValue));
                 break;
               case "gt": // >
                 query.addConditions(INVOICE.INVOICE_DATE.gt(dateValue));
                 break;
               case "ge": // >=
                 query.addConditions(INVOICE.INVOICE_DATE.ge(dateValue));
                 break;
            }
        }
    }

    /**
     * Returns the total number of records
     *
     * @return
     */
    @Override
    public int getCountRecord() {
        SelectFinalStep<?> select
                = dsl.selectCount()
                     .from(INVOICE)
                     .where(INVOICE.INVOICE_DATE.between(
                                this.workingPeriod.getBeginDate(),
                                this.workingPeriod.getEndDate()));

        SelectQuery<?> query = select.getQuery();

        if (this.searchFlag) {
            makeSearchCondition(query);
        }

        return (int) query.fetch().getValue(0, 0);
    }



    /**
     * Returns the list of invoices
     *
     * @return
     */
    @Override
    public List<Map<String, Object>> getRecords() {
        SelectFinalStep<?> select = dsl.select(
                INVOICE.INVOICE_ID,
                INVOICE.CUSTOMER_ID,
                CUSTOMER.NAME.as("CUSTOMER_NAME"),
```

```
                INVOICE.INVOICE_DATE,
                INVOICE.PAID,
                INVOICE.TOTAL_SALE)
            .from(INVOICE)
            .innerJoin(CUSTOMER).on(CUSTOMER.CUSTOMER_ID.eq(INVOICE.CUSTOMER_ID))
            .where(INVOICE.INVOICE_DATE.between(
                       this.workingPeriod.getBeginDate(),
                       this.workingPeriod.getEndDate()));

        SelectQuery<?> query = select.getQuery();
        // add a search condition
        if (this.searchFlag) {
            makeSearchCondition(query);
        }
        // add sorting
        if (this.sIdx.equals("INVOICE_DATE")) {
            switch (this.sOrd) {
               case "asc":
                 query.addOrderBy(INVOICE.INVOICE_DATE.asc());
                 break;
               case "desc":
                 query.addOrderBy(INVOICE.INVOICE_DATE.desc());
                 break;
            }
        }
        // limit the number of records and add an offset
        if (this.limit != 0) {
            query.addLimit(this.limit);
        }
        if (this.offset != 0) {
            query.addOffset(this.offset);
        }

        return query.fetchMaps();
    }
}
```

Invoice Items

We make the class for viewing the invoice items via jqGrid a little simpler. Its records are filtered by invoice header code and user-driven search and sort options are not implemented.

```
package ru.ibase.fbjavaex.jqgrid;

import org.jooq.*;

import java.util.List;
import java.util.Map;

import static ru.ibase.fbjavaex.exampledb.Tables.INVOICE_LINE;
import static ru.ibase.fbjavaex.exampledb.Tables.PRODUCT;

/**
 * The grid handler for the invoice items
```

```
 *
 * @author Simonov Denis
 */
public class JqGridInvoiceLine extends JqGrid {

    private int invoiceId;


    public int getInvoiceId() {
        return this.invoiceId;
    }

    public void setInvoiceId(int invoiceId) {
        this.invoiceId = invoiceId;
    }

    /**
     * Returns the total number of records
     *
     * @return
     */
    @Override
    public int getCountRecord() {
        SelectFinalStep<?> select
            = dsl.selectCount()
                 .from(INVOICE_LINE)
                 .where(INVOICE_LINE.INVOICE_ID.eq(this.invoiceId));

        SelectQuery<?> query = select.getQuery();

        return (int) query.fetch().getValue(0, 0);
    }


    /**
     * Returns invoice items
     *
     * @return
     */
    @Override
    public List<Map<String, Object>> getRecords() {
        SelectFinalStep<?> select = dsl.select(
                INVOICE_LINE.INVOICE_LINE_ID,
                INVOICE_LINE.INVOICE_ID,
                INVOICE_LINE.PRODUCT_ID,
                PRODUCT.NAME.as("PRODUCT_NAME"),
                INVOICE_LINE.QUANTITY,
                INVOICE_LINE.SALE_PRICE,
                INVOICE_LINE.SALE_PRICE.mul(INVOICE_LINE.QUANTITY).as("TOTAL"))
            .from(INVOICE_LINE)
            .innerJoin(PRODUCT).on(PRODUCT.PRODUCT_ID.eq(INVOICE_LINE.PRODUCT_ID))
            .where(INVOICE_LINE.INVOICE_ID.eq(this.invoiceId));

        SelectQuery<?> query = select.getQuery();
        return query.fetchMaps();
    }
}
```

InvoiceManager Class

The *ru.ibase.fbjavaex.managers.InvoiceManager* class is a kind of business layer that will be used to direct adding, editing and deleting invoices and their items, along with invoice payment. All operations in this layer will be performed in a SNAPSHOT transaction. We have chosen to have our application perform all of the invoice management options in this class by calling stored procedures. It is not mandatory to do it this way, of course. It is just one option.

```
package ru.ibase.fbjavaex.managers;


import java.sql.Timestamp;
import org.jooq.DSLContext;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Isolation;

import static ru.ibase.fbjavaex.exampledb.Sequences.GEN_INVOICE_ID;
import static ru.ibase.fbjavaex.exampledb.Routines.spAddInvoice;
import static ru.ibase.fbjavaex.exampledb.Routines.spEditInvoice;
import static ru.ibase.fbjavaex.exampledb.Routines.spPayForInovice;
import static ru.ibase.fbjavaex.exampledb.Routines.spDeleteInvoice;
import static ru.ibase.fbjavaex.exampledb.Routines.spAddInvoiceLine;
import static ru.ibase.fbjavaex.exampledb.Routines.spEditInvoiceLine;
import static ru.ibase.fbjavaex.exampledb.Routines.spDeleteInvoiceLine;

/**
 * Invoice manager
 *
 * @author Simonov Denis
 */
public class InvoiceManager {

    @Autowired(required = true)
    private DSLContext dsl;

    /**
     * Add invoice
     *
     * @param customerId
     * @param invoiceDate
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void create(Integer customerId,
                       Timestamp invoiceDate) {
        int invoiceId = this.dsl.nextval(GEN_INVOICE_ID).intValue();

        spAddInvoice(this.dsl.configuration(),
            invoiceId,
            customerId,
            invoiceDate);
    }
```

```
    /**
     * Edit invoice
     *
     * @param invoiceId
     * @param customerId
     * @param invoiceDate
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void edit(Integer invoiceId,
                     Integer customerId,
                     Timestamp invoiceDate) {
        spEditInvoice(this.dsl.configuration(),
            invoiceId,
            customerId,
            invoiceDate);
    }

    /**
     * Payment of invoices
     *
     * @param invoiceId
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void pay(Integer invoiceId) {
        spPayForInovice(this.dsl.configuration(),
            invoiceId);
    }

    /**
     * Delete invoice
     *
     * @param invoiceId
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void delete(Integer invoiceId) {
        spDeleteInvoice(this.dsl.configuration(),
            invoiceId);
    }

    /**
     * Add invoice item
     *
     * @param invoiceId
     * @param productId
     * @param quantity
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void addInvoiceLine(Integer invoiceId,
                               Integer productId,
                               Integer quantity) {
        spAddInvoiceLine(this.dsl.configuration(),
            invoiceId,
            productId,
```

```
            quantity);
    }

    /**
     * Edit invoice item
     *
     * @param invoiceLineId
     * @param quantity
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void editInvoiceLine(Integer invoiceLineId,
                                Integer quantity) {
        spEditInvoiceLine(this.dsl.configuration(),
            invoiceLineId,
            quantity);
    }

    /**
     * Delete invoice item
     *
     * @param invoiceLineId
     */
    @Transactional(propagation = Propagation.REQUIRED,
                   isolation = Isolation.REPEATABLE_READ)
    public void deleteInvoiceLine(Integer invoiceLineId) {
        spDeleteInvoiceLine(this.dsl.configuration(),
            invoiceLineId);
    }
}
```

Invoice Controller Class

Now we move on to writing the controller. The input point of our controller will be the *index* method, that is responsible for displaying the JSP page (view). This page contains the layout for displaying the grid and the tool and navigation bars.

Data for displaying invoice headers are loaded asynchronously by the *jqGrid* component (the path is `/invoice/getdata`). The *getData* method is connected with this path, similarly to the primary modules. Invoice items are returned by the *getDetailData* method (the path is `/invoice/getdetaildata`). The primary key of the invoice whose detail grid is currently open is passed to this method.

The methods implemented are addInvoice, editInvoice, deleteInvoice, payInvoice for invoice headers and addInvoiceLine, editInvoiceLine, deleteInvoiceLine for invoice line items.

```
package ru.ibase.fbjavaex.controllers;

import java.sql.Timestamp;
import java.util.HashMap;
import java.util.Map;
import java.util.Date;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.beans.PropertyEditorSupport;
```

```
import javax.ws.rs.core.MediaType;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.WebDataBinder;
import ru.ibase.fbjavaex.jqgrid.JqGridInvoice;
import ru.ibase.fbjavaex.jqgrid.JqGridInvoiceLine;

import ru.ibase.fbjavaex.managers.InvoiceManager;

import ru.ibase.fbjavaex.jqgrid.JqGridData;


/**
 * Invoice controller
 *
 * @author Simonov Denis
 */
@Controller
public class InvoiceController {

    @Autowired(required = true)
    private JqGridInvoice invoiceGrid;

    @Autowired(required = true)
    private JqGridInvoiceLine invoiceLineGrid;

    @Autowired(required = true)
    private InvoiceManager invoiceManager;

    /**
     * Describe how a string is converted to a date
     * from the input parameters of the HTTP request
     *
     * @param binder
     */
    @InitBinder
    public void initBinder(WebDataBinder binder)   {
        binder.registerCustomEditor(Timestamp.class,
                new PropertyEditorSupport() {
            @Override
            public void setAsText(String value) {
              try {
                if ((value == null) || (value.isEmpty())) {
                  setValue(null);
                } else {
                  Date parsedDate = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss")
                       .parse(value);
                  setValue(new Timestamp(parsedDate.getTime()));
                }
              } catch (ParseException e) {
                throw new java.lang.IllegalArgumentException(value);
```

```
            }
        }
    });
}

/**
 * Default action
 * Returns the JSP name of the page (view) to display
 *
 * @param map
 * @return JSP page name
 */
@RequestMapping(value = "/invoice/", method = RequestMethod.GET)
public String index(ModelMap map) {

    return "invoice";
}

/**
 * Returns a list of invoices in JSON format for jqGrid
 *
 * @param rows number of entries per page
 * @param page current page number
 * @param sIdx sort field
 * @param sOrd sorting order
 * @param search search flag
 * @param searchField search field
 * @param searchString search value
 * @param searchOper comparison operation
 * @param filters filter
 * @return
 */
@RequestMapping(value = "/invoice/getdata",
        method = RequestMethod.GET,
        produces = MediaType.APPLICATION_JSON)
@ResponseBody
public JqGridData getData(
        @RequestParam(value = "rows", required = false,
                      defaultValue = "20") int rows,
        @RequestParam(value = "page", required = false,
                      defaultValue = "1") int page,
        @RequestParam(value = "sidx", required = false,
                      defaultValue = "") String sIdx,
        @RequestParam(value = "sord", required = false,
                      defaultValue = "asc") String sOrd,
        @RequestParam(value = "_search", required = false,
                      defaultValue = "false") Boolean search,
        @RequestParam(value = "searchField", required = false,
                      defaultValue = "") String searchField,
        @RequestParam(value = "searchString", required = false,
                      defaultValue = "") String searchString,
        @RequestParam(value = "searchOper", required = false,
                      defaultValue = "") String searchOper,
        @RequestParam(value = "filters", required = false,
                      defaultValue = "") String filters) {

    if (search) {
        invoiceGrid.setSearchCondition(searchField, searchString, searchOper);
```

```
        }
        invoiceGrid.setLimit(rows);
        invoiceGrid.setPageNo(page);

        invoiceGrid.setOrderBy(sIdx, sOrd);

        return invoiceGrid.getJqGridData();
    }

    /**
     * Add invoice
     *
     * @param customerId customer id
     * @param invoiceDate invoice date
     * @return
     */
    @RequestMapping(value = "/invoice/create",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> addInvoice(
            @RequestParam(value = "CUSTOMER_ID", required = true,
                          defaultValue = "0") Integer customerId,
            @RequestParam(value = "INVOICE_DATE", required = false,
                          defaultValue = "") Timestamp invoiceDate) {
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.create(customerId, invoiceDate);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    /**
     * Edit invoice
     *
     * @param invoiceId invoice id
     * @param customerId customer id
     * @param invoiceDate invoice date
     * @return
     */
    @RequestMapping(value = "/invoice/edit",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> editInvoice(
            @RequestParam(value = "INVOICE_ID", required = true,
                          defaultValue = "0") Integer invoiceId,
            @RequestParam(value = "CUSTOMER_ID", required = true,
                          defaultValue = "0") Integer customerId,
            @RequestParam(value = "INVOICE_DATE", required = false,
                          defaultValue = "") Timestamp invoiceDate) {
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.edit(invoiceId, customerId, invoiceDate);
            map.put("success", true);
```

```
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    /**
     * Pays an invoice
     *
     * @param invoiceId invoice id
     * @return
     */
    @RequestMapping(value = "/invoice/pay",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> payInvoice(
            @RequestParam(value = "INVOICE_ID", required = true,
                          defaultValue = "0") Integer invoiceId) {
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.pay(invoiceId);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    /**
     * Delete invoice
     *
     * @param invoiceId invoice id
     * @return
     */
    @RequestMapping(value = "/invoice/delete",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> deleteInvoice(
            @RequestParam(value = "INVOICE_ID", required = true,
                          defaultValue = "0") Integer invoiceId) {
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.delete(invoiceId);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    /**
     * Returns invoice item
     *
     * @param invoice_id invoice id
     * @return
     */
```

```
    @RequestMapping(value = "/invoice/getdetaildata",
            method = RequestMethod.GET,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public JqGridData getDetailData(
            @RequestParam(value = "INVOICE_ID", required = true) int invoice_id) {

        invoiceLineGrid.setInvoiceId(invoice_id);

        return invoiceLineGrid.getJqGridData();

    }

    /**
     * Add invoice item
     *
     * @param invoiceId invoice id
     * @param productId product id
     * @param quantity quantity of products
     * @return
     */
    @RequestMapping(value = "/invoice/createdetail",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> addInvoiceLine(
            @RequestParam(value = "INVOICE_ID", required = true,
                          defaultValue = "0") Integer invoiceId,
            @RequestParam(value = "PRODUCT_ID", required = true,
                          defaultValue = "0") Integer productId,
            @RequestParam(value = "QUANTITY", required = true,
                          defaultValue = "0") Integer quantity) {
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.addInvoiceLine(invoiceId, productId, quantity);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    /**
     * Edit invoice item
     *
     * @param invoiceLineId invoice item id
     * @param quantity quantity of products
     * @return
     */
    @RequestMapping(value = "/invoice/editdetail",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> editInvoiceLine(
            @RequestParam(value = "INVOICE_LINE_ID", required = true,
                          defaultValue = "0") Integer invoiceLineId,
            @RequestParam(value = "QUANTITY", required = true,
                          defaultValue = "0") Integer quantity) {
```

```
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.editInvoiceLine(invoiceLineId, quantity);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }

    /**
     * Delete invoice item
     *
     * @param invoiceLineId invoice item id
     * @return
     */
    @RequestMapping(value = "/invoice/deletedetail",
            method = RequestMethod.POST,
            produces = MediaType.APPLICATION_JSON)
    @ResponseBody
    public Map<String, Object> deleteInvoiceLine(
            @RequestParam(value = "INVOICE_LINE_ID", required = true,
                          defaultValue = "0") Integer invoiceLineId) {
        Map<String, Object> map = new HashMap<>();
        try {
            invoiceManager.deleteInvoiceLine(invoiceLineId);
            map.put("success", true);
        } catch (Exception ex) {
            map.put("error", ex.getMessage());
        }
        return map;
    }
}
```

The invoice controller is very similar to the primary module controllers except for two things:

1. The controller displays and works with the data of both the main grid and the detail grid

2. Invoices are filtered by the date field so that only those invoices that are included in the work period are displayed

Working with Dates in Java

Working with dates in Java throws up a few quirks.

The *java.sql.Timestamp* type in Java supports precision up to nanoseconds whereas the maximum precision of the TIMESTAMP type in Firebird is one ten-thousandth of a second. That is not really a significant problem.

Date and time types in Java support working with time zones. Firebird does not currently support the TIMESTAMP WITH TIMEZONE type. Java works on the assumption that dates in the database are stored in the time zone of the server. However, time will be converted to UTC during serialization into JSON. It must be taken into account when processing time data in JavaScript.

Attention!

Java takes the time offset from its own time zone database, not from the operating system. This practice considerably increases the need to keep up with the latest version of JDK. If you have some old version of JDK installed, working with date and time may be incorrect.

By default, a date is serialized into JSON in as the number of nanoseconds since January 1, 1970, which is not always what is wanted. A date can be serialized into a text representation, by setting to False the date conversion configuration property *SerializationFeature.WRITE_DATES_AS_TIMESTAMPS* date conversion in the *configureMessageConverters* method of the *WebAppConfig* class.

We will return to date processing a little later.

```
@Configuration
@ComponentScan("ru.ibase.fbjavaex")
@EnableWebMvc
public class WebAppConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureMessageConverters(
      List<HttpMessageConverter<?>> httpMessageConverters) {
        MappingJackson2HttpMessageConverter jsonConverter =
            new MappingJackson2HttpMessageConverter();
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS,
                               false);
        jsonConverter.setObjectMapper(objectMapper);
        httpMessageConverters.add(jsonConverter);
    }
…
}
```

The *initBinder* method of the *InvoiceController* controller describes how the text representation of a date sent by the browser is converted into a value of type Timestamp.

Displaying the Invoices

The JSP page contains the layout for displaying the grid with invoice headers and the navigation bar. Invoice items are displayed as a drop-down grid when the header of the selected invoice is clicked.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:set var="cp" value="${pageContext.request.servletContext.contextPath}"
       scope="request" />

<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>An example of a Spring MVC application using Firebird and jOOQ</title>
```

```
        <!-- Scripts and styles -->
        <%@ include file="../jspf/head.jspf" %>
        <script src="${cp}/resources/js/jqGridProduct.js"></script>
        <script src="${cp}/resources/js/jqGridCustomer.js"></script>
        <script src="${cp}/resources/js/jqGridInvoice.js"></script>
    </head>
    <body>
        <!-- Navigation menu -->
        <%@ include file="../jspf/menu.jspf" %>

        <div class="container body-content">

            <h2>Invoices</h2>

            <table id="jqGridInvoice"></table>
            <div id="jqPagerInvoice"></div>

            <hr />
            <footer>
                <p>© 2016 - An example of a Spring MVC application using
                   Firebird and jOOQ</p>
            </footer>
        </div>

        <script type="text/javascript">
            var invoiceGrid = null;
            $(document).ready(function () {
                invoiceGrid = JqGridInvoice({
                    baseAddress: '${cp}'
                });
            });
        </script>

    </body>
</html>
```

The basic logic on the client side is concentrated in the `/resources/js/jqGridInvoice.js` JavaScript module.

```
var JqGridInvoice = (function ($, jqGridProductFactory, jqGridCustomerFactory) {

    return function (options) {
        var jqGridInvoice = {
            dbGrid: null,
            detailGrid: null,
            options: $.extend({
                baseAddress: null
            }, options),
            // return invoice model description
            getInvoiceColModel: function () {
                return [
                    {
                        label: 'Id',
                        name: 'INVOICE_ID', // field name
                        key: true,
                        hidden: true
```

```
                    },
                    {
                        label: 'Customer Id'
                        name: 'CUSTOMER_ID',
                        hidden: true,
                        editrules: {edithidden: true, required: true},
                        editable: true,
                        edittype: 'custom', // custom 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", // input type
                        align: "right",
                        // format as date
                        formatter: jqGridInvoice.dateTimeFormatter,
                        sorttype: 'date', // sort as date
                        formatoptions: {
                            srcformat: 'Y-m-d\TH:i:s', // input format
                            newformat: 'Y-m-d H:i:s'   // output format
                        },
                        editoptions: {
                            // initializing the form element for editing
                            dataInit: function (element) {
                                // creating 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: { // search 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",
            // foramt as currency
            formatter: '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"
```

```
                }
            }
        ];
    },
    initGrid: function () {
        // url to retrieve data
        var url = jqGridInvoice.options.baseAddress + '/invoice/getdata';
        jqGridInvoice.dbGrid = $("#jqGridInvoice").jqGrid({
            url: url,
            datatype: "json", // data format
            mtype: "GET", // http request type
            // model description
            colModel: jqGridInvoice.getInvoiceColModel(),
            rowNum: 500, // number of rows displayed
            loadonce: false, // load only once
            // default sort by INVOICE_DATE column
            sortname: 'INVOICE_DATE',
            sortorder: "desc", // sorting order
            width: window.innerWidth - 80,
            height: 500,
            viewrecords: true, // display the number of entries
            guiStyle: "bootstrap",
            iconSet: "fontAwesome",
            caption: "Invoices",
            // pagination element
            pager: '#jqPagerInvoice',
            subGrid: true, // show subGrid
            // javascript function to display the child grid
            subGridRowExpanded: jqGridInvoice.showChildGrid,
            subGridOptions: {
                // load only once
                reloadOnExpand: false,
                // load the subgrid string only when you click on the "+"
                selectOnExpand: true
            }
        });
    },
    // date format function
    dateTimeFormatter: function(cellvalue, options, rowObject) {
        var date = new Date(cellvalue);
        return date.toLocaleString().replace(",", "");
    },
    // returns a template for the editing dialog
    getTemplate: function () {
      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='invoiceGrid.showCustomerWindow(); ";
      template += "return false;'>";
      template += "<span class='glyphicon glyphicon-folder-open'>";
      template += "</span>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 template;
        },
        // date conversion in UTC
        convertToUTC: function(datetime) {
          if (datetime) {
              var dateParts = datetime.split('.');
              var date = dateParts[2].substring(0, 4) + '-' +
                         dateParts[1] + '-' + dateParts[0];
              var time = dateParts[2].substring(5);
              if (!time) {
                  time = '00:00:00';
              }
              var dt = Date.parse(date + 'T' + time);
              var s = dt.getUTCFullYear() + '-' +
                      dt.getUTCMonth() + '-' +
                      dt.getUTCDay() + 'T' +
                      dt.getUTCHour() + ':' +
                      dt.getUTCMinute() + ':' +
                      dt.getUTCSecond() + '  GMT';
              return s;
          } else
              return null;
        },
        // returns the options for editing invoices
        getEditInvoiceOptions: function () {
          return {
            url: jqGridInvoice.options.baseAddress + '/invoice/edit',
            reloadAfterSubmit: true,
            closeOnEscape: true,
            closeAfterEdit: true,
            drag: true,
            modal: true,
            top: $(".container.body-content").position().top + 150,
            left: $(".container.body-content").position().left + 150,
            template: jqGridInvoice.getTemplate(),
            afterSubmit: jqGridInvoice.afterSubmit,
            editData: {
              INVOICE_ID: function () {
                var selectedRow = jqGridInvoice.dbGrid.getGridParam("selrow");
                var value = jqGridInvoice.dbGrid
                          .getCell(selectedRow, 'INVOICE_ID');
                return value;
              },
              CUSTOMER_ID: function () {
                return $('#dlgEditInvoice input[name=CUSTOMER_ID]').val();
              },
              INVOICE_DATE: function () {
                var datetime = $('#dlgEditInvoice input[name=INVOICE_DATE]')
                              .val();
                return jqGridInvoice.convertToUTC(datetime);
              }
            }
          };
        },
```

```
            // returns options for adding invoices
            getAddInvoiceOptions: function () {
              return {
                url: jqGridInvoice.options.baseAddress + '/invoice/create',
                reloadAfterSubmit: true,
                closeOnEscape: true,
                closeAfterAdd: true,
                drag: true,
                modal: true,
                top: $(".container.body-content").position().top + 150,
                left: $(".container.body-content").position().left + 150,
                template: jqGridInvoice.getTemplate(),
                afterSubmit: jqGridInvoice.afterSubmit,
                editData: {
                  CUSTOMER_ID: function () {
                    return $('#dlgEditInvoice input[name=CUSTOMER_ID]').val();
                  },
                  INVOICE_DATE: function () {
                    var datetime = $('#dlgEditInvoice input[name=INVOICE_DATE]')
                                 .val();
                    return jqGridInvoice.convertToUTC(datetime);
                  }
                }
              };
            },
            // returns the options for deleting invoices
            getDeleteInvoiceOptions: function () {
              return {
                url: jqGridInvoice.options.baseAddress + '/invoice/delete',
                reloadAfterSubmit: true,
                closeOnEscape: true,
                closeAfterDelete: true,
                drag: true,
                msg: "Delete the selected invoice?",
                afterSubmit: jqGridInvoice.afterSubmit,
                delData: {
                  INVOICE_ID: function () {
                    var selectedRow = jqGridInvoice.dbGrid.getGridParam("selrow");
                    var value = jqGridInvoice.dbGrid
                              .getCell(selectedRow, 'INVOICE_ID');
                    return value;
                  }
                }
              };
            },
            initPager: function () {
                // display the navigation bar
                jqGridInvoice.dbGrid.jqGrid('navGrid', '#jqPagerInvoice',
                    {
                         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"
            },
            jqGridInvoice.getEditInvoiceOptions(),
            jqGridInvoice.getAddInvoiceOptions(),
            jqGridInvoice.getDeleteInvoiceOptions()
        );
        // Add a button to pay the invoice
        var urlPay = jqGridInvoice.options.baseAddress + '/invoice/pay';
        jqGridInvoice.dbGrid.navButtonAdd('#jqPagerInvoice',
            {
               buttonicon: "glyphicon-usd",
               title: "Pay",
               caption: "Pay",
               position: "last",
               onClickButton: function () {
                 // get the id of the current record
                 var id = jqGridInvoice.dbGrid.getGridParam("selrow");
                 if (id) {
                   $.ajax({
                       url: urlPay,
                       type: 'POST',
                       data: {INVOICE_ID: id},
                       success: function (data) {
                           // Check if an error has occurred
                           if (data.hasOwnProperty("error")) {
                               jqGridInvoice.alertDialog('??????',
                                                          data.error);
                           } else {
                                  // refresh grid
                               $("#jqGridInvoice").jqGrid(
                                   'setGridParam',
                                   {
                                       datatype: 'json'
                                   }
                               ).trigger('reloadGrid');
                           }
                       }
                   });
                 }
               }
            }
        );
    },
    init: function () {
        jqGridInvoice.initGrid();
        jqGridInvoice.initPager();
    },
    afterSubmit: function (response, postdata) {
        var responseData = response.responseJSON;
        // Check if an error has occurred
        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];
            },
            getInvoiceLineColModel: function (parentRowKey) {
              return [
                  {
                    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_NAME',
              width: 300,
              editable: true,
              edittype: "text",
              editoptions: {
                  size: 50,
                  maxlength: 60,
                  readonly: true
              },
              editrules: {required: true}
            },
            {
              label: 'Price',
              name: 'SALE_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=SALE_PRICE]').val()-0;
                    var total = quantity * price;
                    $('#dlgEditInvoiceLine input[name=TOTAL]').val(total);
                  }
                }],
                defaultValue: 1
              }
            },
            {
              label: 'Total',
              name: 'TOTAL',
              formatter: 'currency',
              align: "right",
              width: 100,
              editable: true,
              editoptions: {
                  readonly: true
              }
            }
        ];
```

```
        },
        // returns the options for editing the invoice item
        getEditInvoiceLineOptions: function () {
          return {
            url: jqGridInvoice.options.baseAddress + '/invoice/editdetail',
            reloadAfterSubmit: true,
            closeOnEscape: true,
            closeAfterEdit: true,
            drag: true,
            modal: true,
            top: $(".container.body-content").position().top + 150,
            left: $(".container.body-content").position().left + 150,
            template: jqGridInvoice.getTemplateDetail(),
            afterSubmit: jqGridInvoice.afterSubmit,
            editData: {
              INVOICE_LINE_ID: function () {
                var selectedRow = jqGridInvoice.detailGrid
                                 .getGridParam("selrow");
                var value = jqGridInvoice.detailGrid
                           .getCell(selectedRow, 'INVOICE_LINE_ID');
                return value;
              },
              QUANTITY: function () {
                return $('#dlgEditInvoiceLine input[name=QUANTITY]').val();
              }
            }
          };
        },
        // returns options for adding an invoice item
        getAddInvoiceLineOptions: function () {
          return {
            url: jqGridInvoice.options.baseAddress + '/invoice/createdetail',
            reloadAfterSubmit: true,
            closeOnEscape: true,
            closeAfterAdd: true,
            drag: true,
            modal: true,
            top: $(".container.body-content").position().top + 150,
            left: $(".container.body-content").position().left + 150,
            template: jqGridInvoice.getTemplateDetail(),
            afterSubmit: jqGridInvoice.afterSubmit,
            editData: {
              INVOICE_ID: function () {
                var selectedRow = jqGridInvoice.dbGrid.getGridParam("selrow");
                var value = jqGridInvoice.dbGrid
                           .getCell(selectedRow, 'INVOICE_ID');
                return value;
              },
              PRODUCT_ID: function () {
                return $('#dlgEditInvoiceLine input[name=PRODUCT_ID]').val();
              },
              QUANTITY: function () {
                return $('#dlgEditInvoiceLine input[name=QUANTITY]').val();
              }
            }
          };
        },
        // returns the option to delete the invoice item
```

```
            getDeleteInvoiceLineOptions: function () {
              return {
                url: jqGridInvoice.options.baseAddress + '/invoice/deletedetail',
                reloadAfterSubmit: true,
                closeOnEscape: true,
                closeAfterDelete: true,
                drag: true,
                msg: "Delete the selected item?",
                afterSubmit: jqGridInvoice.afterSubmit,
                delData: {
                  INVOICE_LINE_ID: function () {
                    var selectedRow = jqGridInvoice.detailGrid
                                     .getGridParam("selrow");
                    var value = jqGridInvoice.detailGrid
                               .getCell(selectedRow, 'INVOICE_LINE_ID');
                    return value;
                  }
                }
              };
            },
            // Event handler for the parent grid expansion event
            // takes two parameters: the parent record identifier
            // and the primary record key
            showChildGrid: function (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 = jqGridInvoice.options.baseAddress
                                 + '/invoice/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
                jqGridInvoice.detailGrid = $("#" + childGridID).jqGrid({
                    url: childGridURL,
                    mtype: "GET",
                    datatype: "json",
                    page: 1,
                    colModel: jqGridInvoice.getInvoiceLineColModel(parentRowKey),
                    loadonce: false,
                    width: '100%',
                    height: '100%',
                    guiStyle: "bootstrap",
                    iconSet: "fontAwesome",
                    pager: "#" + childGridPagerID
                });
                // displaying the toolbar
                $("#" + childGridID).jqGrid(
                    'navGrid', '#' + childGridPagerID,
```

```
            {
                search: false,
                add: true,
                edit: true,
                del: true,
                refresh: true
            },
            jqGridInvoice.getEditInvoiceLineOptions(),
            jqGridInvoice.getAddInvoiceLineOptions(),
            jqGridInvoice.getDeleteInvoiceLineOptions()
        );
    },
    // returns a template for the invoice item editor
    getTemplateDetail: function () {
      var template = "<div style='margin-left:15px;' ";
      template += "id='dlgEditInvoiceLine'>";
      template += "<div>{INVOICE_ID} </div>";
      template += "<div>{PRODUCT_ID} </div>";
      // input field with a button
      template += "<div> Product <sup>*</sup>:</div>";
      template += "<div>";
      template += "<div style='float: left;'>{PRODUCT_NAME}</div> ";
      template += "<a style='margin-left: 0.2em;' class='btn' ";
      template += "onclick='invoiceGrid.showProductWindow(); ";
      template += "return false;'>";
      template += "<span class='glyphicon glyphicon-folder-open'>";
      template += "</span> Select</a> ";
      template += "<div style='clear: both;'></div>";
      template += "</div>";
      template += "<div> Quantity: </div><div>{QUANTITY} </div>";
      template += "<div> Price: </div><div>{SALE_PRICE} </div>";
      template += "<div> Total: </div><div>{TOTAL} </div>";
      template += "<hr style='width: 100%;'/>";
      template += "<div> {sData} {cData}  </div>";
      template += "</div>";
      return template;
    },
    // Display selection window from the goods directory.
    showProductWindow: function () {
      var dlg = $('<div>')
                .attr('id', 'dlgChooseProduct')
                .attr('aria-hidden', 'true')
                .attr('role', 'dialog')
                .attr('data-backdrop', 'static')
                .css("z-index", '2000')
                .addClass('modal')
                .appendTo($('body'));

      var dlgContent = $("<div>")
                .addClass("modal-content")
                .css('width', '760px')
                .appendTo($('<div>')
                        .addClass('modal-dialog')
                        .appendTo(dlg));

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

```
          .addClass("close")
          .attr('type', 'button')
          .attr('aria-hidden', 'true')
          .attr('data-dismiss', 'modal')
          .html("×")
          .appendTo(dlgHeader);
$("<h5>").addClass("modal-title")
           .html("Select product")
           .appendTo(dlgHeader);
var dlgBody = $('<div>')
          .addClass("modal-body")
          .appendTo(dlgContent);
var dlgFooter = $('<div>').addClass("modal-footer")
                          .appendTo(dlgContent);
$("<button>")
  .attr('type', 'button')
  .addClass('btn')
  .html('OK')
  .on('click', function () {
      var rowId = $("#jqGridProduct")
                 .jqGrid("getGridParam", "selrow");
      var row = $("#jqGridProduct")
               .jqGrid("getRowData", rowId);
      $('#dlgEditInvoiceLine input[name=PRODUCT_ID]')
          .val(row["PRODUCT_ID"]);
      $('#dlgEditInvoiceLine input[name=PRODUCT_NAME]')
          .val(row["NAME"]);
      $('#dlgEditInvoiceLine input[name=SALE_PRICE]')
          .val(row["PRICE"]);
      var price = $('#dlgEditInvoiceLine input[name=SALE_PRICE]')
          .val()-0;
      var quantity = $('#dlgEditInvoiceLine input[name=QUANTITY]')
          .val()-0;
      var total = Math.round(price * quantity * 100) / 100;
      $('#dlgEditInvoiceLine input[name=TOTAL]').val(total);
      dlg.modal('hide');
  })
  .appendTo(dlgFooter);

$("<button>")
      .attr('type', 'button')
      .addClass('btn')
      .html('Cancel')
      .on('click', function () {
          dlg.modal('hide');
      })
      .appendTo(dlgFooter);

$('<table>')
      .attr('id', 'jqGridProduct')
      .appendTo(dlgBody);
$('<div>')
      .attr('id', 'jqPagerProduct')
      .appendTo(dlgBody);

dlg.on('hidden.bs.modal', function () {
      dlg.remove();
});
```

```
          dlg.modal();

          jqGridProductFactory({
              baseAddress: jqGridInvoice.options.baseAddress
          });
        },
        // Display the selection window from the customer's directory.
        showCustomerWindow: function () {
            // 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 dialog 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("×")
                    .appendTo(dlgHeader);
            // title of dialog
            $("<h5>").addClass("modal-title")
                     .html("Select customer")
                     .appendTo(dlgHeader);
            // body of dialog
            var dlgBody = $('<div>')
                    .addClass("modal-body")
                    .appendTo(dlgContent);
            // footer of dialog
            var dlgFooter = $('<div>').addClass("modal-footer")
                                      .appendTo(dlgContent);
            // "OK" button
            $("<button>")
                .attr('type', 'button')
                .addClass('btn')
                .html('OK')
                .on('click', function () {
                    var rowId = $("#jqGridCustomer")
                               .jqGrid("getGridParam", "selrow");
                    var row = $("#jqGridCustomer")
                             .jqGrid("getRowData", rowId);
                    // Keep the identifier and the name of the customer
                    // in 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);
        // "Cancel" button
        $("<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', 'jqGridCustomer')
                .appendTo(dlgBody);
        // add the navigation bar
        $('<div>')
                .attr('id', 'jqPagerCustomer')
                .appendTo(dlgBody);
        dlg.on('hidden.bs.modal', function () {
            dlg.remove();
        });
        // display dialog
        dlg.modal();
        jqGridCustomerFactory({
            baseAddress: jqGridInvoice.options.baseAddress
        });
    },
    // A window for displaying the error.
    alertDialog: function (title, error) {
        var alertDlg = $('<div>')
                .attr('aria-hidden', 'true')
                .attr('role', 'dialog')
                .attr('data-backdrop', 'static')
                .addClass('modal')
                .appendTo($('body'));
        var dlgContent = $("<div>")
                .addClass("modal-content")
                .appendTo($('<div>')
                        .addClass('modal-dialog')
                        .appendTo(alertDlg));
        var dlgHeader = $('<div>').addClass("modal-header")
                                  .appendTo(dlgContent);
        $("<button>")
                .addClass("close")
                .attr('type', 'button')
                .attr('aria-hidden', 'true')
                .attr('data-dismiss', 'modal')
                .html("×")
                .appendTo(dlgHeader);
        $("<h5>").addClass("modal-title")
                 .html(title)
                 .appendTo(dlgHeader);
```

```
                $('<div>')
                        .addClass("modal-body")
                        .appendTo(dlgContent)
                        .append(error);
                alertDlg.on('hidden.bs.modal', function () {
                    alertDlg.remove();
                });
                alertDlg.modal();
            }
        };
        jqGridInvoice.init();
        return jqGridInvoice;
    };
})(jQuery, JqGridProduct, JqGridCustomer);
```

Displaying and Editing Invoice Lines

In the invoice module, the main grid is used to display headers and the detail grid, opened with a click, is used to display invoice items. For the child grid to be displayed, the True value is assigned to the *subGrid* property. The child grid is displayed using the *subGridRowExpanded* event connected with the *showChildGrid* method.

The items are filtered by the primary key of the invoice. Along with the main buttons on the navigation bar, a custom button for paying for the invoice is added to the invoice header using the *jqGridInvoice.dbGrid.navButtonAdd* function (see the initPager method).

Dialog Boxes

Dialog boxes for editing secondary modules are much more complicated than their primary counterparts. They often use options selected from other modules. For that reason, these edit dialog boxes cannot be built automatically using jqGrid. However, this library has an option to build dialog boxes using templates, which we use.

The dialog box template is returned by the *getTemplate* function. The invoiceGrid.showCustomerWindow() function opens the customer module for selecting a customer. It uses the functions of the JqGridCustomer module described earlier. After the customer is selected in the modal window, its key is inserted into the CUSTOMER_ID field. Fields that are to be sent to the server using pre-processing or from hidden fields are described in the editData property of the Edit and Add options.

Processing Dates

To get back to processing dates: as we already know, the *InvoiceController* controller returns the date in UTC. Because we want to display it in the current time zone, we specify the *jqGridInvoice.dateTimeFormatter* date formatting function via the *formatter* property of the corresponding INVOICE_DATE field.

When sending data to the server, we need the reverse operation—convert time from the current time zone to UTC. The *convertToUTC* function is responsible for that.

The custom template returned by the *getTemplateDetail* function is also used for editing invoice items. The *invoiceGrid.showProductWindow()* function opens a window for selecting a product from the product list. This function uses the functions of the JqGridProduct module.

The code for the JqGridInvoice module contains detailed comments and more explanation so that you can understand the logic of its workings.

The Result

Some screenshots from the web application we have developed in our project.

Figure 7.3. Screenshot 1

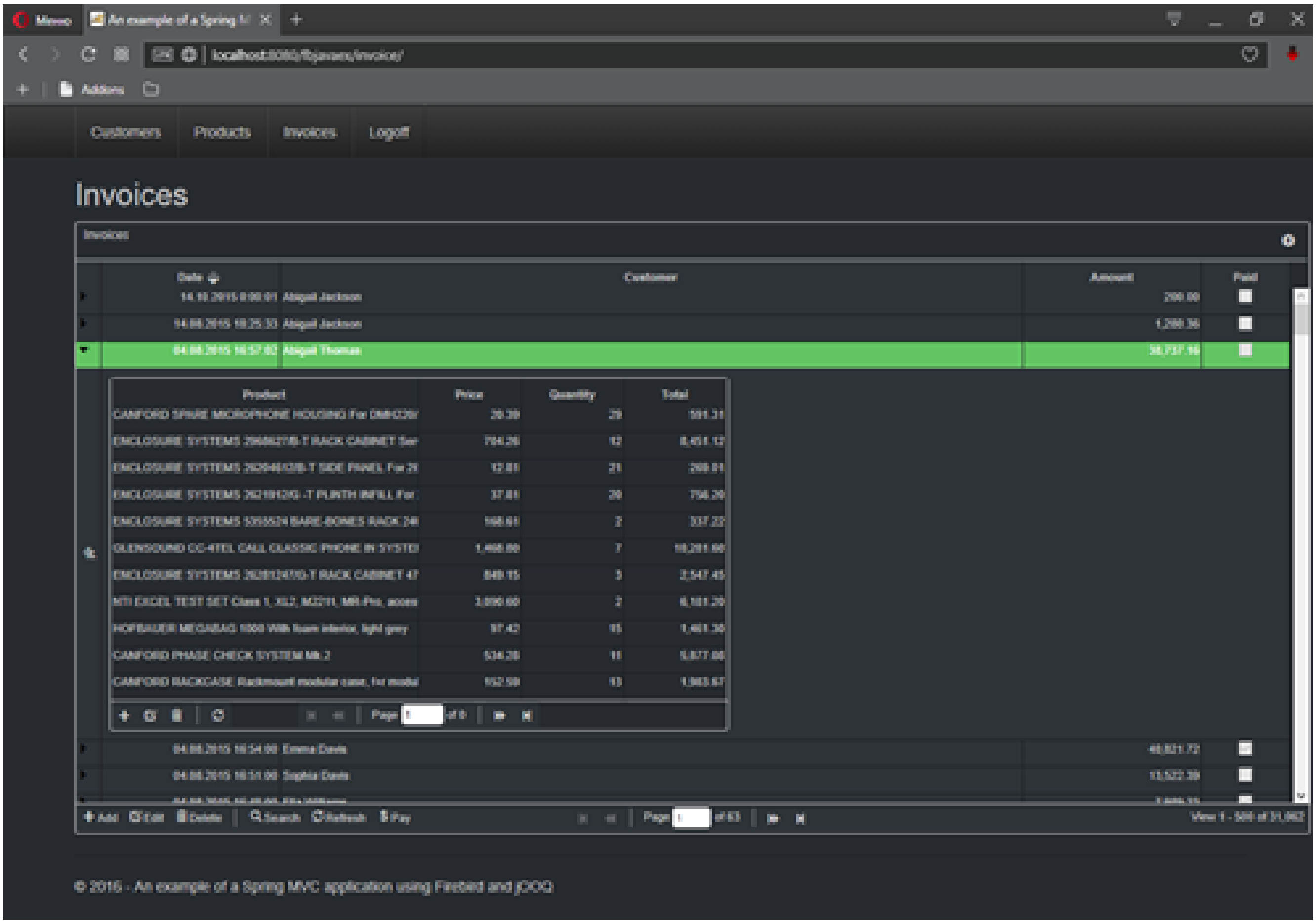


Figure 7.4. Screenshot 2

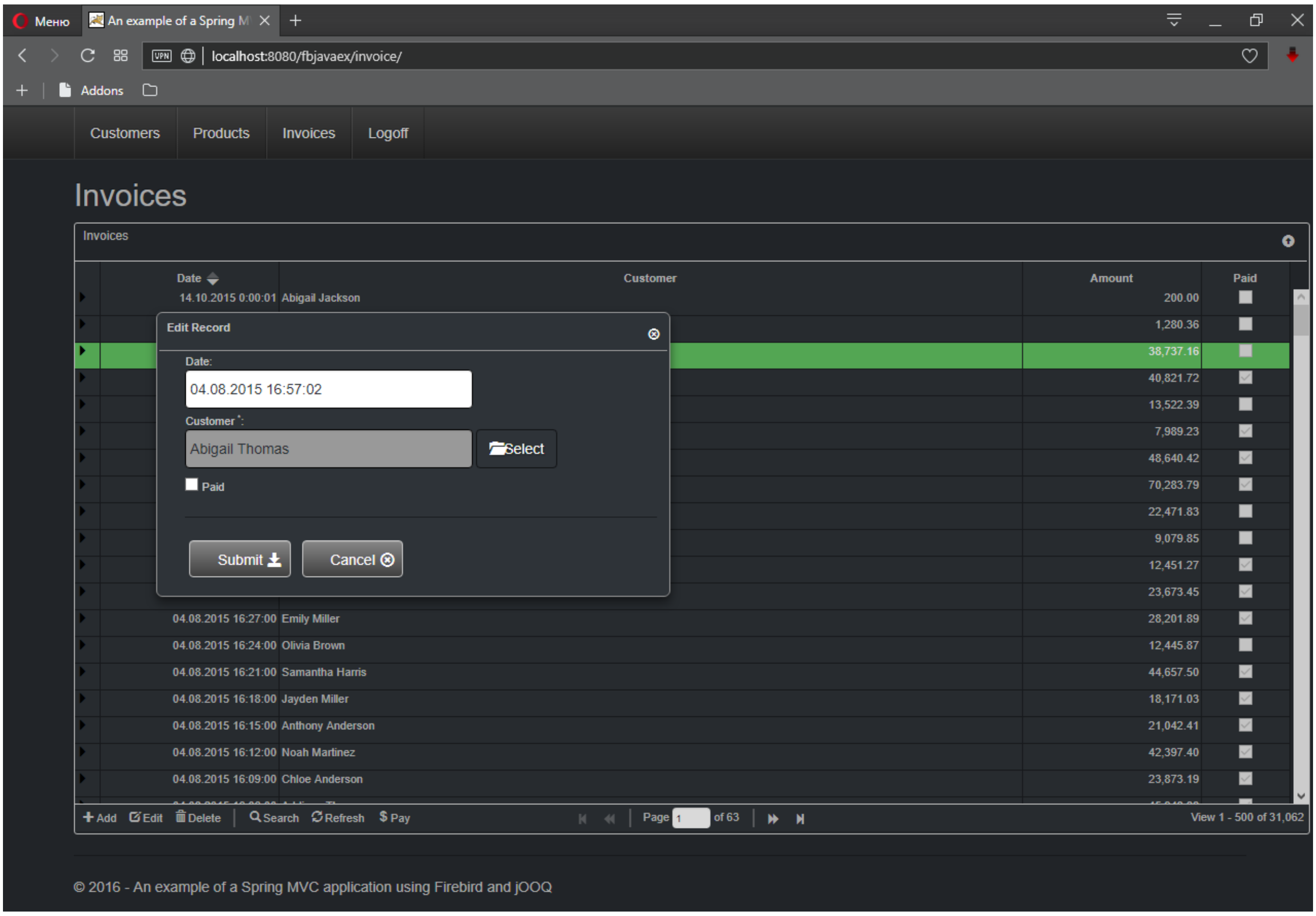


Figure 7.5. Screenshot 3

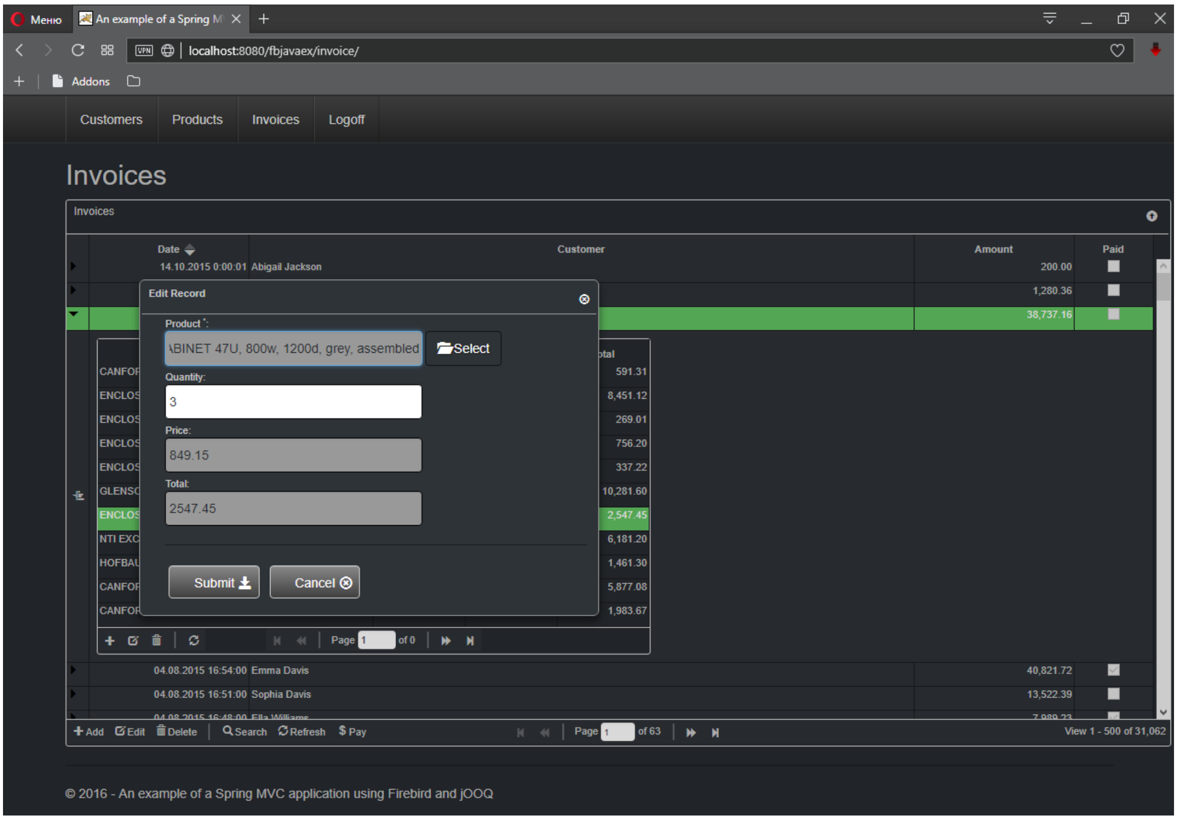
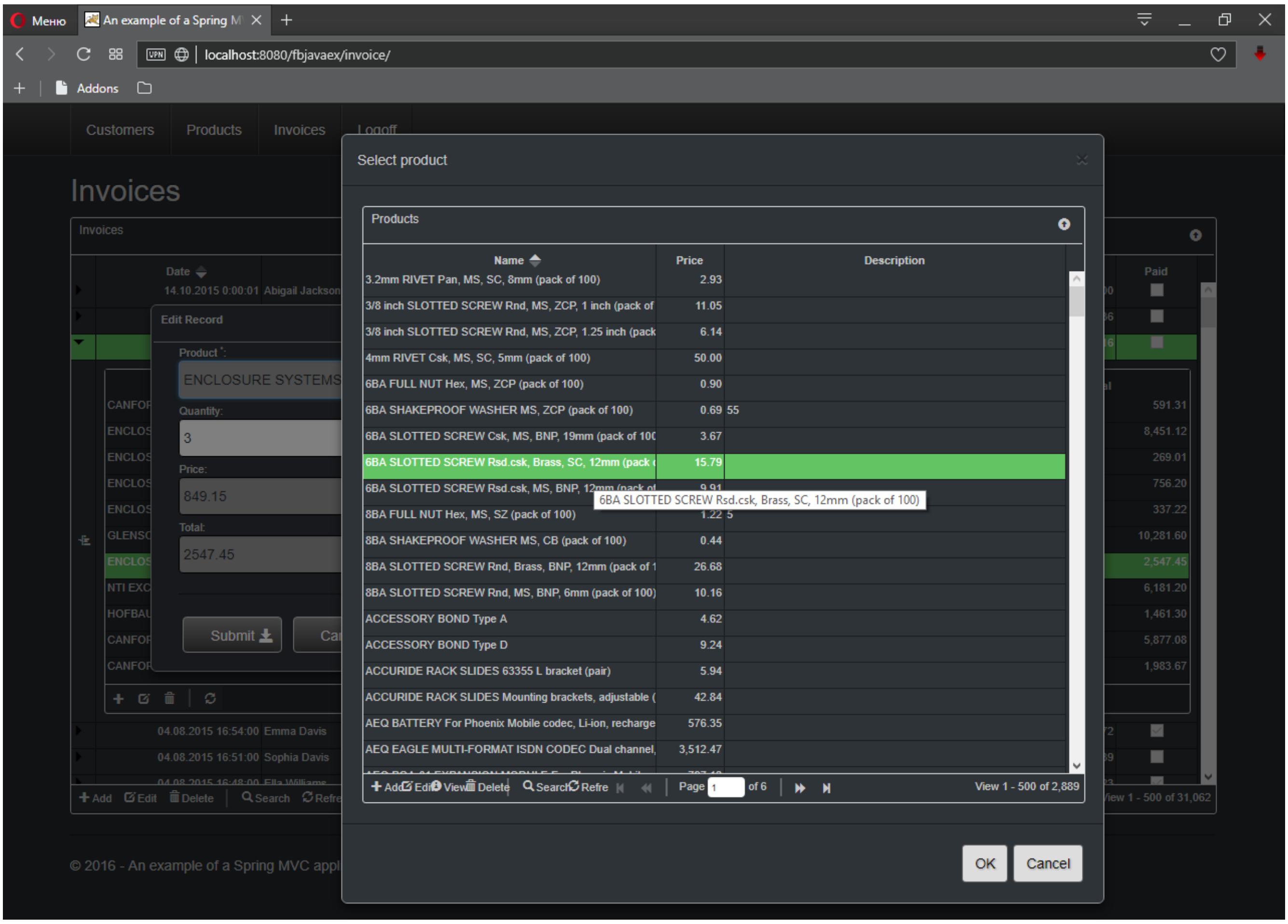


Figure 7.6. Screenshot 4



Source Code

You can download the source code from the link https://github.com/sim1984/fbjavaex.

Appendix A: License notice