# 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.

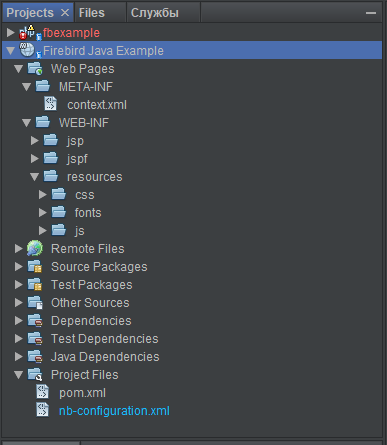
For testing and debugging purposes, we will also need to install one of the web servers or application servers (Apache Tomcat or GlassFish). We are basing our project on the Maven web application templates.

## 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:



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>

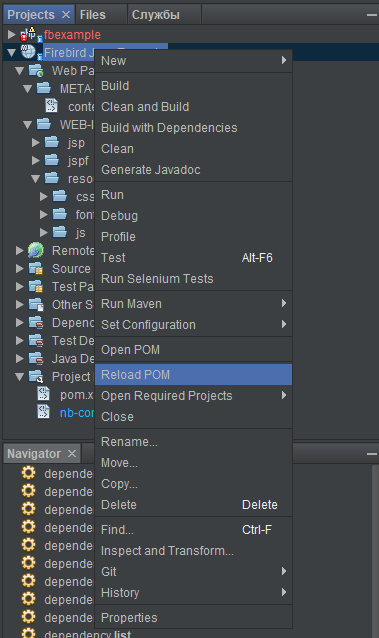
</plugins>

</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:



## Coding the Configuration

We use this configuration class to specify the locations of web resources and JSP views. The configureMessageConverters method directs that dates must be serialized to strings, overriding the default, which 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](https://www.jooq.org) 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.jar**

The 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](https://github.com/FirebirdSQL/jaybird/releases/download/v3.0.0/Jaybird-3.0.0-JDK_1.8.zip).

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](https://www.jooq.org/doc/3.9/manual-single-page/%23code-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 initializing 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 that translates 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 and
* 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();

// определяем конфигурацию подключения

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();

// используем диалект SQL СУБД 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](https://www.jooq.org/doc/3.9/manual-single-page/%23sql-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 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 above.

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](https://www.jooq.org/doc/3.9/manual-single-page/#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 tor 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](https://www.jooq.org/doc/3.9/manual-single-page/#merge-statement) 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--seBasicDataSource.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 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](https://www.firebirdsql.org/file/Jaybird_2_1_JDBC_driver_manual.pdf). 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](https://github.com/tonytomov/jqGrid). Currently, jqGrid is distributed under a commercial licence, but it is free for non-commercial purposes. You can use the [free-jqGrid](https://github.com/free-jqgrid/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.

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 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 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

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.

editCustomer

The editCustomer method is connected with the /customer/edit pathThe 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>&copy; 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](http://www.trirand.com/jqgridwiki/doku.php?id=wiki%3Acolmodel_options) 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](http://www.trirand.com/jqgridwiki/doku.php?id=wiki%3Anavigator) section of the jqGrid documentation.

##### Functions 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

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 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>&copy; 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('Ошибка',

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("&times;")

.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("&times;")

.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("&times;")

.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 line 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.

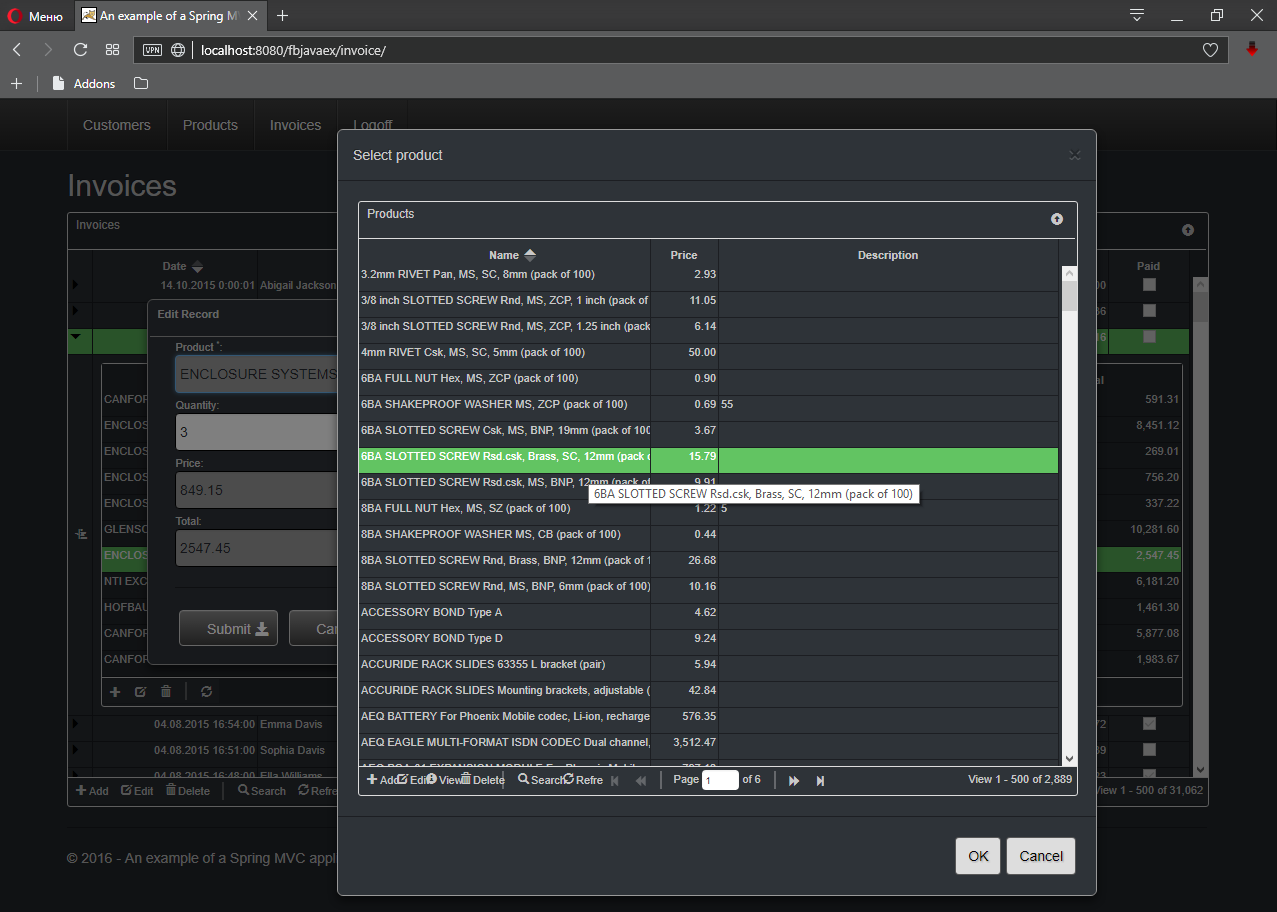
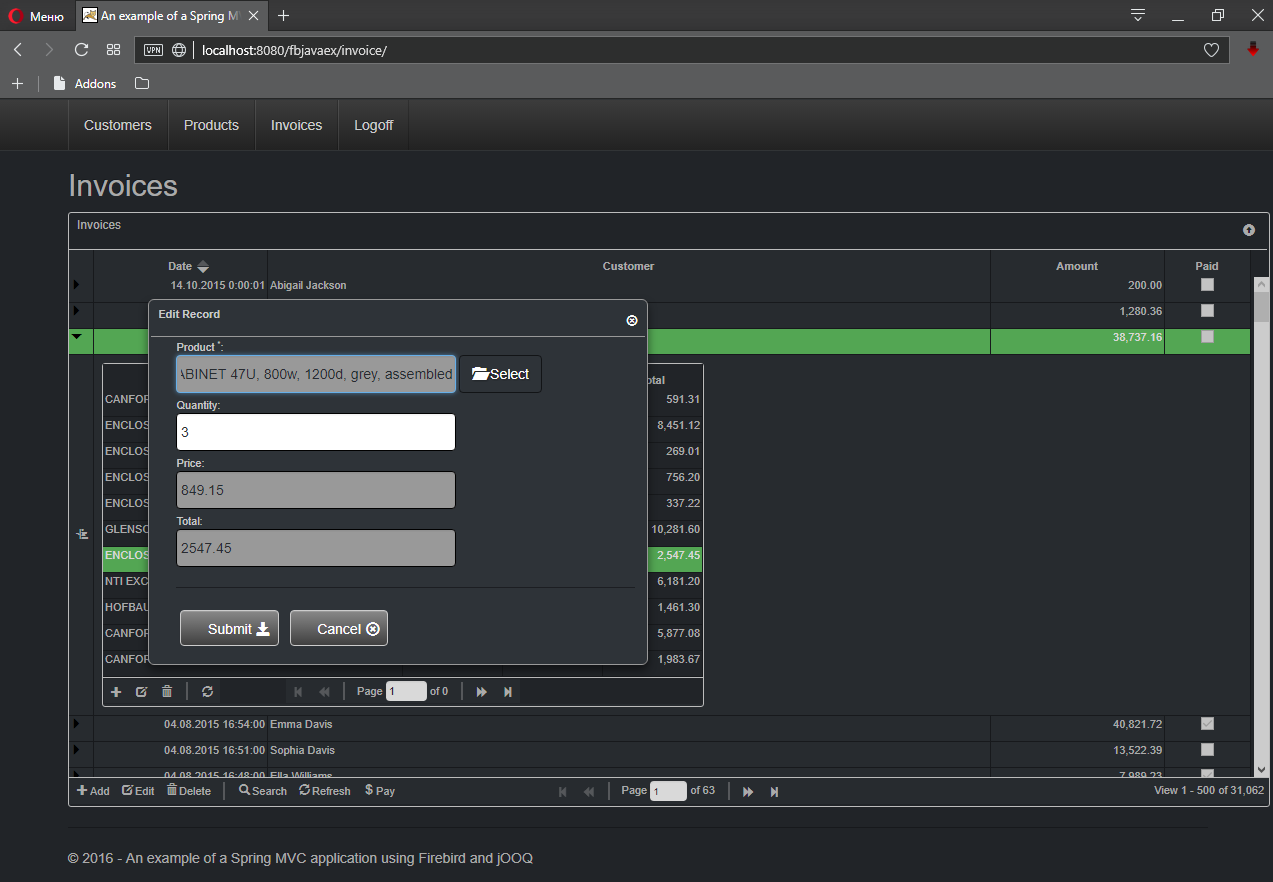
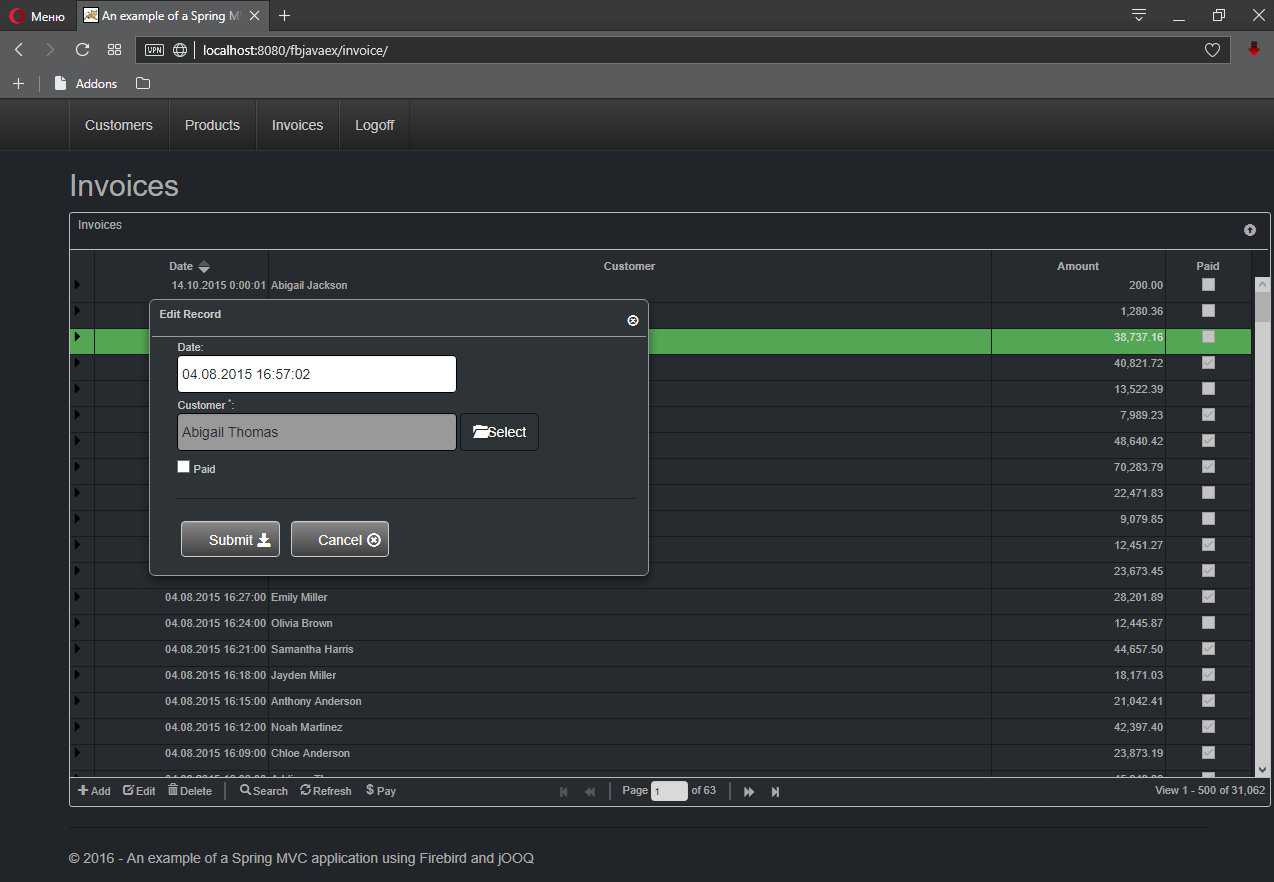
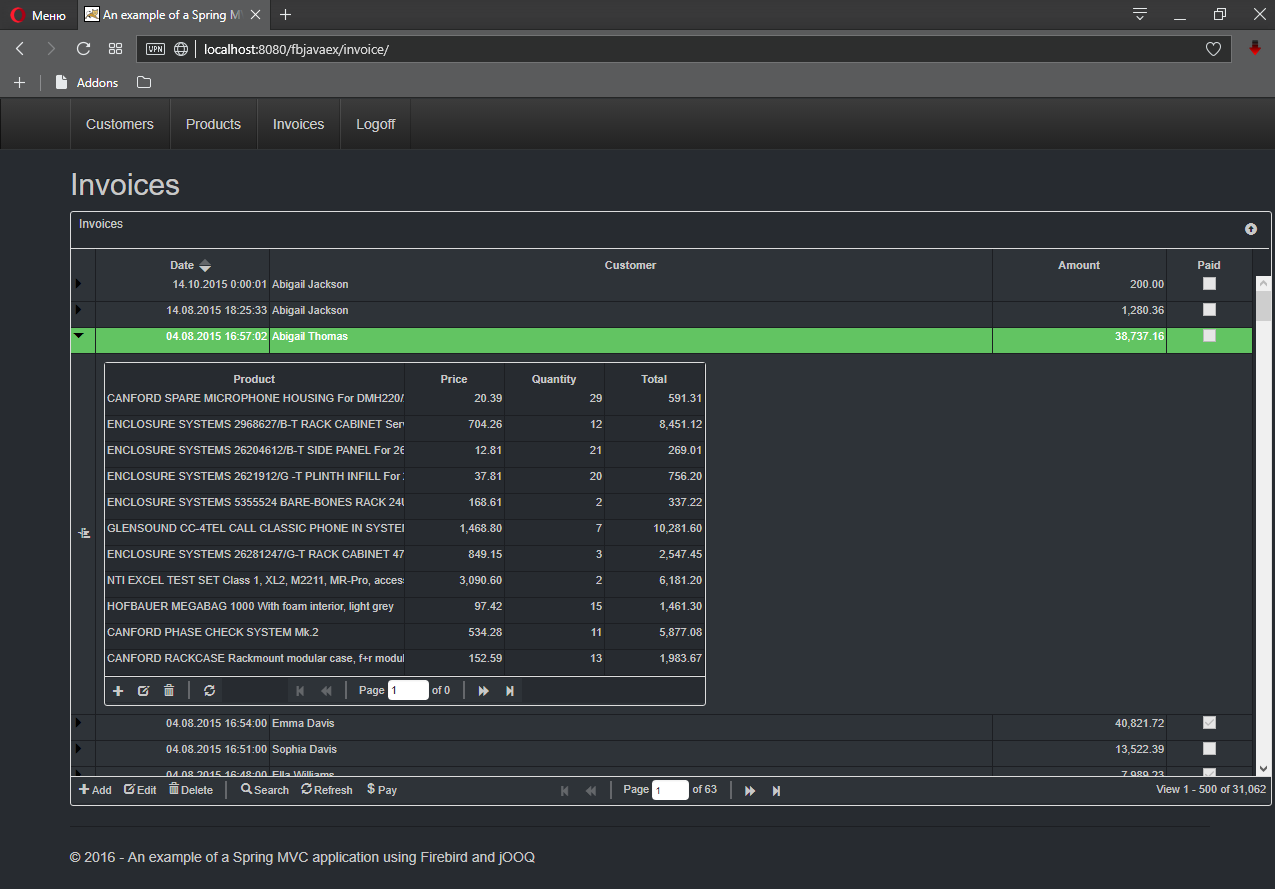
When sending data to the server, we need the reverse operation--convert time from the current time zone to UTC. The convertToUTC function is responsible for that.

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.



You can download the source code from the link <https://github.com/sim1984/fbjavaex>