# Creating Web Applications with PHP and Firebird

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

If your server supports PHP, you just create your .php files, put them in your web directory and the server will automatically parse them for you. There is no need to compile anything nor do you need to install any extra tools. PHP-enabled files are simply HTML files with a whole language of custom tags embedded in them. There is nothing to compile.

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

## PHP Drivers for Firebird

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

* The [Firebird/Interbase](http://php.net/ibase) extension (ibase\_ functions)
* [The PDO driver](http://php.net/manual/en/ref.pdo-firebird.php) for Firebird

### Firebird Client Library

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

**Note to Win32/Win64 users**

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

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

### The Firebird/InterBase Extension

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

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

**extension**=php\_interbase.dll

or this line on Linux and other POSIX systems:

**extension**=php\_interbase.so

#### Installing the Fb/IB Extension on Linux

Installing the Fb/IB Extension on Linux, one of the following commands should work. The one you use depends on the distribution package and the versions it supports:

apt-get install php5-firebird

rpm –ihv php5-firebird

yum install php70w-interbase

zypper install php5-firebird

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

#### Programming Style

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

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

<?php

$db = 'localhost:example';

$username = 'SYSDBA';

$password = 'masterkey';

*// Connect to database*

$dbh = ibase\_connect($db, $username, $password);

$sql = 'SELECT login, email FROM users';

*// Execute query*

$rc = ibase\_query($dbh, $sql);

*// Get the result row by row as object*

**while** ($row = ibase\_fetch\_object($rc)) {

**echo** $row->email, "\n";

}

*// Release the handle associated with the result of the query*

ibase\_free\_result($rc);

*// Release the handle associated with the connection*

ibase\_close($dbh);

#### The ibase\_ Connect Functions

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

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

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

Need to know

Many ibase\_ functions cannot accommodate the identifier of a connection, transaction or prepared query. Those functions use the identifier of the last established connection or last started transaction instead of the relevant identifier. It is not a recommended practice, especially if your web application can use more than one connection.

#### ibase\_query

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

*// …*

$sql = 'SELECT login, email FROM users WHERE id=?';

$id = 1;

*// Execute query*

$rc = ibase\_query($dbh, $sql, $id);

*// Get the result row by row as object*

**if** ($row = ibase\_fetch\_object($rc)) {

**echo** $row->email, "\n";

}

*// Release the handle associated with the result of the query*

ibase\_free\_result($rc);

*// …*

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

*// …*

$sql = 'SELECT login, email FROM users WHERE id=?';

*// Prepare statement*

$sth = ibase\_prepare($dbh, $sql);

$id = 1;

*// Execute statement*

$rc = ibase\_execute($sth, $id);

*// Get the result row by row as object*

**if** ($row = ibase\_fetch\_object($rc)) {

**echo** $row->email, "\n";

}

*// Release the handle associated with the result of the query*

ibase\_free\_result($rc);

*// Release the prepared statement*

ibase\_free\_query($sth);

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

*// …*

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';

*// Prepare statement*

$sth = ibase\_prepare($dbh, $sql);

$users = [["user1", "user1@gmail.com"], ["user2", "user2@gmail.com"]];

*// Execute statement*

**foreach** ($users **as** $user)) {

ibase\_execute($sth, $user[0], $user[1]);

}

*// Release the prepared statement*

ibase\_free\_query($sth);

*// …*

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

**function** fb\_execute ($stmt, $data)

{

**if** (!is\_array($data))

**return** ibase\_execute($stmt, $data);

array\_unshift($data, $stmt);

$rc = call\_user\_func\_array('ibase\_execute', $data);

**return** $rc;

}

The Firebird/Interbase extension does not support named parameters in queries.

#### ibase\_trans

By default, the Fb/IB extension commits the transaction automatically after executing each SQL query, making it necessary to start a transaction with the function [ibase\_trans](http://php.net/manual/ru/function.ibase-trans.php) if you need to control transactions explicitly. An explicit transaction is started with the following parameters if none are provided: IBASE\_WRITE | IBASE\_CONCURRENCY | IBASE\_WAIT. You can find the description of predefined constants for specifying the parameters of a transaction [here](http://php.net/manual/en/ibase.constants.php). A transaction must be completed by either ibase\_commit or ibase\_rollback. This extension supports the COMMIT RETAIN and ROLLBACK RETAIN parameters directly if you use the functions ibase\_commit\_ret or ibase\_rollback\_ret, respectively, instead.

**Note**

The default transaction parameters are good for most cases and it is really rarely that you need to change them. A connection to the database, along with all resources allocated to it, exists for no longer than it takes for the PHP script to complete. Even if you use persistent connections, all allocated resources will be released after the ibase\_close function is called. Even so, I strongly recommend releasing all allocated resources explicitly, by calling the corresponding ibase\_ functions.

I advise strongly against using the ibase\_commit\_ret and ibase\_rollback\_ret functions because they have no place in a web application. The purpose of COMMIT RETAIN and ROLLBACK RETAIN is to keep cursors open in desktop applications when a transaction ends.

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';

*// Prepare statement*

$sth = ibase\_prepare($dbh, $sql);

$users = [["user1", "user1@gmail.com"], ["user2", "user2@gmail.com"]];

$trh = ibase\_trans($dbh, IBASE\_WRITE | IBASE\_CONCURRENCY | IBASE\_WAIT);

**try** {

*// Execute statement*

**foreach** ($users **as** $user)) {

$r = ibase\_execute($sth, $user[0], $user[1]);

*// If an error occurs, throw an exception*

**if** ($r === **false**)

**throw new** \**Exception**(ibase\_errmsg());

}

ibase\_commit($trh);

}

**catch**(\**Exception** $e) {

ibase\_rollback($trh);

**echo** $e->getMessage();

}

*// Release the prepared statement*

ibase\_free\_query($sth);

**Warning!**

ibase functions raise no exception if an error occurs, although an error will cause some to return False. Note that it is essential to use the === strict relational operator to compare the result to False. Calling any ibase function could result in an error.

The function ibase\_errmsg is available to discover an error message and the function ibase\_errcode can provide the error code.

#### Services API Calls

The Fb/IB extension can interact with the Firebird server by way of functions that wrap calls to the Services API: ibase\_service\_attach, ibase\_service\_detach, ibase\_server\_info, **ibase\_maintain\_db, ibase\_db\_info, ibase\_backup, ibase\_restore**. They can return information about the Firebird server, initiate a backup or restore or get statistics. We are not examining them in detail, since they are required mainly to administer a database, a topic that is outside the scope of this project.

#### Firebird Events

The Fb/IB extension also supports working with Firebird events by means of a set of functions: **ibase\_set\_event\_handler**, **ibase\_free\_event\_handler, ibase\_wait\_event**.

### The PDO (Firebird Driver)

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

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

**extension**=php\_pdo.dll

**Note**

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

#### Firebird-specific Library

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

**extension**=php\_pdo.dll

**extension**=php\_pdo\_firebird.dll

These DLLs must be in the directory extension\_dir.

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

apt-get install php5-firebird

rpm –ihv php5-firebird

yum install php70w-firebird

zypper install php5-firebird

#### Programming Style

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

To be able to work with Firebird, the connection string must start with the *firebird:* prefix and conform to the format described in the [PDO\_FIREBIRD DSN](http://php.net/manual/en/ref.pdo-firebird.connection.php) section of the documentation.

#### Making Connections

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

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';

$username = 'SYSDBA';

$password = 'masterkey';

**try** {

*// Connect to database*

$dbh = **new** \PDO($dsn, $username, $password,

[\PDO::ATTR\_ERRMODE => \PDO::ERRMODE\_EXCEPTION]);

$sql = 'SELECT login, email FROM users';

*// Execute query*

$query = $dbh->query($sql);

*// Get the result row by row as object*

**while** ($row = $query->fetch(\PDO::FETCH\_OBJ)) {

**echo** $row->email, "\n";

}

$query->closeCursor();

} **catch** (\PDOException $e) {

**echo** $e->getMessage();

}

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

**Note**

For PDO to use [persistent connections](http://php.net/persistent-connections), the array of attributes must be passed to the PDO constructor with PDO::ATTR\_PERSISTENT => true.

#### Exception Handling

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

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

#### Querying

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

The various ways of calling query can be found in the [documentation](http://php.net/manual/en/pdo.query.php).

##### Queries with No Data Set

For executing an SQL query that returns no data set, you can use the [exec](http://php.net/manual/en/pdo.exec.php) method that returns the number of affected rows. Executing prepared queries is not supported by exec.

##### Parameterized Queries

If there are parameters in the query, prepared queries must be used. For this, the [prepare](http://php.net/manual/ru/pdo.prepare.php) method is called instead of the query method. The prepare method returns an object of the \PDOStatement class that encapsulates methods for working with prepared queries and their results. Executing the query requires calling the [execute](http://php.net/manual/ru/pdostatement.execute.php) method that can accept as its parameter an array of named or unnamed parameters.

The result of executing a SELECT query can be obtained with one the following methods: [fetch](http://php.net/manual/ru/pdostatement.fetch.php), [fetchAll](http://php.net/manual/ru/pdostatement.fetchall.php), [fetchColumn](http://php.net/manual/ru/pdostatement.fetchcolumn.php), [fetchObject](http://php.net/manual/ru/pdostatement.fetchobject.php). The fetch and fetchAll methods can return results in various forms: an associative array, an object or an instance of a particular class. The class instance option is quite often used in the MVC pattern during work with models.

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';

$username = 'SYSDBA';

$password = 'masterkey';

**try** {

*// Connect to database*

$dbh = **new** \PDO($dsn, $username, $password,

[\PDO::ATTR\_ERRMODE => \PDO::ERRMODE\_EXCEPTION]);

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';

$users = [

["user1", "user1@gmail.com"],

["user2", "user2@gmail.com"]

];

*// Prepare statement*

$query = $dbh->prepare($sql);

*// Execute statement*

**foreach** ($users **as** $user)) {

$query->execute($user);

}

} **catch** (\PDOException $e) {

**echo** $e->getMessage();

}

An example using named parameters:

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';

$username = 'SYSDBA';

$password = 'masterkey';

**try** {

*// Connect to database*

$dbh = **new** \PDO($dsn, $username, $password,

[\PDO::ATTR\_ERRMODE => \PDO::ERRMODE\_EXCEPTION]);

$sql = 'INSERT INTO users(login, email) VALUES(:login, :email)';

$users = [

[":login" => "user1", ":email" => "user1@gmail.com"],

[":login" => "user2", ":email" => "user2@gmail.com"]

];

*// Prepare statement*

$query = $dbh->prepare($sql);

*// Execute statement*

**foreach** ($users **as** $user)) {

$query->execute($user);

}

} **catch** (\PDOException $e) {

**echo** $e->getMessage();

}

**Note**

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

An alternative way to pass parameters to a query is by using "binding". The [bindValue](http://php.net/manual/ru/pdostatement.bindvalue.php) method binds a value to a named or unnamed parameter. The [bindParam](http://php.net/manual/ru/pdostatement.bindparam.php) method binds a variable to a named or unnamed parameter. The bindParam method is especially useful for stored procedures that return a value via the OUT or IN OUT parameter, which is different to the mechanism for returning values from stored procedures in Firebird.

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';

$username = 'SYSDBA';

$password = 'masterkey';

**try** {

*// Connect to database*

$dbh = **new** \PDO($dsn, $username, $password,

[\PDO::ATTR\_ERRMODE => \PDO::ERRMODE\_EXCEPTION]);

$sql = 'INSERT INTO users(login, email) VALUES(:login, :email)';

$users = [

["user1", "user1@gmail.com"],

["user2", "user2@gmail.com"]

];

*// Prepare statement*

$query = $dbh->prepare($sql);

*// Execute statement*

**foreach** ($users **as** $user)) {

$query->bindValue(":login", $user[0]);

$query->bindValue(":email", $user[1]);

$query->execute();

}

} **catch** (\PDOException $e) {

**echo** $e->getMessage();

}

**Caution**

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

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';

$username = 'SYSDBA';

$password = 'masterkey';

**try** {

*// Connect to database*

$dbh = **new** \PDO($dsn, $username, $password,

[\PDO::ATTR\_ERRMODE => \PDO::ERRMODE\_EXCEPTION]);

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';

$users = [

["user1", "user1@gmail.com"],

["user2", "user2@gmail.com"]

];

*// Prepare statement*

$query = $dbh->prepare($sql);

*// Execute statement*

**foreach** ($users **as** $user)) {

$query->bindValue(1, $user[0]);

$query->bindValue(2, $user[1]);

$query->execute();

}

} **catch** (\PDOException $e) {

**echo** $e->getMessage();

}

#### Transactions

By default, PDO commits the transaction automatically after executing each SQL query. If you want to control transactions explicitly, you need to start a transaction with the method [\PDO::beginTransaction](http://php.net/manual/ru/pdo.begintransaction.php) . By default, a transaction is started with the following parameters: CONCURRENCY | WAIT | READ\_WRITE. A transaction can be ended with the [\PDO::commit](http://php.net/manual/ru/pdo.commit.php) or [\PDO::rollback](http://php.net/manual/ru/pdo.commit.php) method.

$username = 'SYSDBA';

$password = 'masterkey';

**try** {

*// Connect to database*

$dbh = **new** \PDO($dsn, $username, $password,

[\PDO::ATTR\_ERRMODE => \PDO::ERRMODE\_EXCEPTION]);

*// Start the transaction to ensure consistency between statements*

$dbh->beginTransaction();

*// Get users from one table*

$users\_stmt = $dbh->prepare('SELECT login, email FROM old\_users');

$users\_stmt->execute();

$users = $users\_stmt->fetchAll(\PDO::FETCH\_OBJECT);

$users\_stmt->closeCursor();

*// And insert into another table*

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';

*// Prepapre statemenet*

$query = $dbh->prepare($sql);

*// Execute statememt*

**foreach** ($users **as** $user)) {

$query->bindValue(1, $user->LOGIN);

$query->bindValue(2, $user->EMAIL]);

$query->execute();

}

*// Commit transaction*

$dbh->commit();

} **catch** (\PDOException $e) {

*// Rollback transaction*

**if** ($dbh && $dbh->inTransaction())

$dbh->rollback();

**echo** $e->getMessage();

}

##### Changing the Transaction Parameters

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

$dbh = **new** \PDO($dsn, $username, $password);

$dbh->setAttribute(\PDO::ATTR\_AUTOCOMMIT, **false**);

$dbh->exec("SET TRANSACTION READ ONLY ISOLATION LEVEL READ COMMITTED NO WAIT");

*// Perform actions in the transaction*

*// …*

$dbh->exec("COMMIT");

$dbh->setAttribute(\PDO::ATTR\_AUTOCOMMIT, **true**);

#### Comparing the Drivers

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

|  |  |  |
| --- | --- | --- |
| **Capability** | **Firebird/Interbase extension** | **PDO** |
| Programming paradigm | Procedural | Object-oriented |
| Supported database engines | Firebird and InterBase; clones of either. | Any database for which there is a PDO driver, including Firebird |
| Handling query parameters | Only unnamed parameters, not very convenient because the functions used allow the number of parameters to be variable | Can work with both named and unnamed parameters. Very convenient although some Firebird features (the EXECUTE BLOCK statement) do not work. |
| Error handling | Requires checking the results of the ibase\_errmsg, ibase\_errcode functions. An error may occur after any ibase\_ function call without raising any exception. | An optional mode is provided to raise exceptions on any error |
| Transaction management | Allows transaction parameters to be specified | Does not allow transaction parameters to be specified. Workaround: execute the SQL statement SET TRANSACTION. |
| Firebird-specific features | Supports work with the Services API (backup, restore, statistics, etc.) and with database events. | Does not support any database-specific feature that cannot be implemented directly using an SQL statement. |

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

## Choosing a Framework for Building a Web Application

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

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

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

## Installing Laravel

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

* PHP >= 5.5.9
* PDO extension
* MCrypt extension
* OpenSSL extension
* Mbstring extension
* Tokenizer extension

### Installing Composer

Laravel uses [Composer](https://getcomposer.org) to manage dependencies. Install Composer first and then install Laravel.

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

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

php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"

php -r "if (hash\_file('SHA384', 'composer-setup.php') ===

'aa96f26c2b67226a324c27919f1eb05f21c248b987e6195cad9690d5c1ff713d53020a02

ac8c217dbf90a7eacc9d141d') { echo 'Installer verified'; } else { echo

'Installer corrupt'; unlink('composer-setup.php'); } echo PHP\_EOL;"

php composer-setup.php

php -r "unlink('composer-setup.php');"

This script does the following:

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

|  |
| --- |
| Caution  Because this script changes with each new version of the installer, it would be safer not to use it, but to go to <https://getcomposer.org/download/> and pick up a fresh installation script. |

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

### Windows

If you are working in Windows, you can make it easier to work with Composer by creating the composer.bat file Run the following command:

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

Then set up your PATH so that you can just call composer from any directory in your commandshell.

More details about installing Composer are available [here](https://getcomposer.org/doc/00-intro.md).

### Installing Laravel

Now, to install Laravel:

composer global require "laravel/installer"

## Creating a Project

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

laravel new fbexample

Wait until it finishes creating the project framework. A description of the directory structure can be found in the Laravel [documentation](http://laravel.su/docs/5.2/structure).

### Our Project's Structure

These are the directories we are most interested in:

app – the main directory of our application. Models will be located in the root directory. The Http subdirectory contains everything that is related to working with the browser. The Http/Controllers subdirectory contains our controllers.

config – the directory with configuration files. You will discover more details about the configuration process later.

public – the root directory of the web application (DocumentRoot). It contains static files: css, js, images, etc.

resources - contains views, localization files and, if any, LESS files, SASS and js applications on such frameworks as ReactJS, AngularJS or Ember that are later put together into the public folder with an external tool.

The root directory of our application contains the composer.json file that describes the packages our application will need besides those that are already present in Laravel. We will need two such packages: [zofe/rapyd](https://github.com/zofe/rapyd-laravel)-laravel for building a quick interface with grids and edit dialog boxes, and [sim1984/laravel-firebird](https://github.com/sim1984/laravel-firebird), an extension for working with Firebird databases.

The sim1984/laravel-firebird package is the author's fork of the [jacquestvanzuydam/laravel-firebird](https://github.com/jacquestvanzuydam/laravel-firebird) package. Its installation is a bit different--the description of how the package differs from the original is available in the article [Package for working with the Firebird DBMS in Laravel](https://habrahabr.ru/post/312874/), if you can read Russian. An English-language description of the packages and the changes from the original can be found in the readme.md document at this URL: <https://github.com/sim1984/laravel-firebird>

|  |
| --- |
| Caution  Remember to set the minimum-stability parameter to 'dev' because the package is not stable enough to publish at https://packagist.org. You will need to modify the composer.json file (see below) to add a reference to the gitHub repository. |

In the file composer.json:

"repositories": [

{

"type": "package",

"package": {

"version": "dev-master",

"name": "sim1984/laravel-firebird",

"source": {

"url": "https://github.com/sim1984/laravel-firebird",

"type": "git",

"reference": "master"

},

"autoload": {

"classmap": [""]

}

}

}

],

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

"zofe/rapyd": "2.2.\*",

"sim1984/laravel-firebird": "dev-master"

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

composer update

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

### Configuration

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

php artisan vendor:publish

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

Zofe\Rapyd\RapydServiceProvider::**class**,

Firebird\FirebirdServiceProvider::**class**,

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

'firebird' => [

'driver' => 'firebird',

'host' => env('DB\_HOST', 'localhost'),

'port' => env('DB\_PORT', '3050'),

'database' => env('DB\_DATABASE', 'examples'),

'username' => env('DB\_USERNAME', 'SYSDBA'),

'password' => env('DB\_PASSWORD', 'masterkey'),

'charset' => env('DB\_CHARSET', 'UTF8'),

'engine\_version' => '3.0.0',

],

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

'default' => env('DB\_CONNECTION', 'firebird'),

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

DB\_CONNECTION=firebird

DB\_HOST=localhost

DB\_PORT=3050

DB\_DATABASE=examples

DB\_USERNAME=SYSDBA

DB\_PASSWORD=masterkey

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

'fields' => [

'attributes' => ['class' => 'form-control'],

'date' => [

'format' => 'Y-m-d',

],

'datetime' => [

'format' => 'Y-m-d H:i:s',

'store\_as' => 'Y-m-d H:i:s',

],

],

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

## Creating Models

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

### A Tool for Model-making

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

php artisan make:model Customer

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

**namespace** App;

**use** Firebird\Eloquent\Model;

**class** Customer **extends** Model

{

/\*\*

\* Table associated with the model

\*

\* @var string

\*/

**protected** $table = 'CUSTOMER';

/\*\*

\* Primary key of the model

\*

\* @var string

\*/

**protected** $primaryKey = 'CUSTOMER\_ID';

/\*\*

\* Our model does not have a timestamp

\*

\* @var bool

\*/

**public** $timestamps = **false**;

/\*\*

\* The name of the sequence for generating the primary key

\*

\* @var string

\*/

**protected** $sequence = 'GEN\_CUSTOMER\_ID';

}

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

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

**namespace** App;

**use** Firebird\Eloquent\Model;

**class** Product **extends** Model

{

/\*\*

\* Table associated with the model

\*

\* @var string

\*/

**protected** $table = 'PRODUCT';

/\*\*

\* Primary key of the model

\*

\* @var string

\*/

**protected** $primaryKey = 'PRODUCT\_ID';

/\*\*

\* Our model does not have a timestamp

\*

\* @var bool

\*/

**public** $timestamps = **false**;

/\*\*

\* The name of the sequence for generating the primary key

\*

\* @var string

\*/

**protected** $sequence = 'GEN\_PRODUCT\_ID';

}

Now, a model for the invoice header:

**namespace** App;

**use** Firebird\Eloquent\Model;

**class** Invoice **extends** Model {

/\*\*

\* Table associated with the model

\*

\* @var string

\*/

**protected** $table = 'INVOICE';

/\*\*

\* Primary key of the model

\*

\* @var string

\*/

**protected** $primaryKey = 'INVOICE\_ID';

/\*\*

\* Our model does not have a timestamp

\*

\* @var bool

\*/

**public** $timestamps = **false**;

/\*\*

\* The name of the sequence for generating the primary key

\*

\* @var string

\*/

**protected** $sequence = 'GEN\_INVOICE\_ID';

/\*\*

\* Customer

\*

\* @return \App\Customer

\*/

**public function** customer() {

**return** $this->belongsTo('App\Customer', 'CUSTOMER\_ID');

}

/\*\*

\* Invoice lines

\* @return \App\InvoiceLine[]

\*/

**public function** lines() {

**return** $this->hasMany('App\InvoiceLine', 'INVOICE\_ID');

}

/\*\*

\* Payed

\*/

**public function** pay() {

$connection = $this->getConnection();

$attributes = $this->attributes;

$connection->executeProcedure('SP\_PAY\_FOR\_INOVICE',

[$attributes['INVOICE\_ID']]);

}

}

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

The function lines returns items from the invoice that are represented by a collection of InvoiceLine models, described later. To establish the one-to-many relation in the lines function, the name of the class model and the relation field are passed to the hasMany method. You can find more details about specifying relations between entities in the [Relationships](https://laravel.com/docs/5.2/eloquent-relationships) section of the Laravel documentation.

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

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

### Invoice Items Model

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

**namespace** App;

**use** Firebird\Eloquent\Model;

**use** Illuminate\Database\Eloquent\Builder;

**class** InvoiceLine **extends** Model {

/\*\*

\* Table associated with the model

\*

\* @var string

\*/

**protected** $table = 'INVOICE\_LINE';

/\*\*

\* Primary key of the model

\*

\* @var string

\*/

**protected** $primaryKey = 'INVOICE\_LINE\_ID';

/\*\*

\* Our model does not have a timestamp

\*

\* @var bool

\*/

**public** $timestamps = **false**;

/\*\*

\* The name of the sequence for generating the primary key

\*

\* @var string

\*/

**protected** $sequence = 'GEN\_INVOICE\_LINE\_ID';

/\*\*

\* Array of names of computed fields

\*

\* @var array

\*/

**protected** $appends = ['SUM\_PRICE'];

/\*\*

\* Product

\*

\* @return \App\Product

\*/

**public function** product() {

**return** $this->belongsTo('App\Product', 'PRODUCT\_ID');

}

/\*\*

\* Amount by item

\*

\* @return double

\*/

**public function** getSumPriceAttribute() {

**return** $this->SALE\_PRICE \* $this->QUANTITY;

}

/\*\*

\* Adding a model object to the database

\* Override this method, because in this case, we work with a stored procedure

\*

\* @param \Illuminate\Database\Eloquent\Builder $query

\* @param array $options

\* @return bool

\*/

**protected function** performInsert(Builder $query, **array** $options = []) {

**if** ($this->fireModelEvent('creating') === **false**) {

**return false**;

}

$connection = $this->getConnection();

$attributes = $this->attributes;

$connection->executeProcedure('SP\_ADD\_INVOICE\_LINE', [

$attributes['INVOICE\_ID'],

$attributes['PRODUCT\_ID'],

$attributes['QUANTITY']

]);

*// We will go ahead and set the exists property to true,*

*// so that it is set when the created event is fired, just in case*

*// the developer tries to update it during the event. This will allow*

*// them to do so and run an update here.*

$this->exists = **true**;

$this->wasRecentlyCreated = **true**;

$this->fireModelEvent('created', **false**);

**return true**;

}

/\*\*

\* Saving changes to the current model instance in the database

\* Override this method, because in this case, we work with a stored procedure

\*

\* @param \Illuminate\Database\Eloquent\Builder $query

\* @param array $options

\* @return bool

\*/

**protected function** performUpdate(Builder $query, **array** $options = []) {

$dirty = $this->getDirty();

**if** (count($dirty) > 0) {

*// If the updating event returns false, we will cancel*

*// the update operation so developers can hook Validation systems*

*// into their models and cancel this operation if the model does*

*// not pass validation. Otherwise, we update.*

**if** ($this->fireModelEvent('updating') === **false**) {

**return false**;

}

$connection = $this->getConnection();

$attributes = $this->attributes;

$connection->executeProcedure('SP\_EDIT\_INVOICE\_LINE', [

$attributes['INVOICE\_LINE\_ID'],

$attributes['QUANTITY']

]);

$this->fireModelEvent('updated', **false**);

}

}

/\*\*

\* Deleting the current model instance from the database

\* Override this method, because in this case, we work with a stored procedure

\*

\* @return void

\*/

**protected function** performDeleteOnModel() {

$connection = $this->getConnection();

$attributes = $this->attributes;

$connection->executeProcedure('SP\_DELETE\_INVOICE\_LINE',

[$attributes['INVOICE\_LINE\_ID']]);

}

}

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

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

#### Operations

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

### How Laravel Manages Data

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

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

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

DB::table('users')

->join('contacts', **function** ($join) {

$join->on('users.id', '=', 'contacts.user\_id')->orOn(...);

})

->get()

Nevertheless, models are more convenient to work with. You can find the description of Eloquent ORM models and the syntax for querying them at <https://laravel.com/docs/5.2/eloquent>. As an example, to retrieve all elements from the collection of customers would require executing the following query:

$customers = Customer::all();

This query will return the first 20 customers sorted alphabetically:

$customers = App\Customer::select()

->orderBy('name')

->take(20)

->get();

#### Complex Models

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

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

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

$flight = **new** Flight;

$flight->name = $request->name;

$flight->save();

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

$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();

Deleting a record involves finding it and calling the delete method.

$flight = App\Flight::find(1);

$flight->delete();

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

App\Flight::destroy(1);

There are other ways of deleting records, for instance, "soft" deletion. You can read more about deletion methods at [https://laravel.com/docs/5.2/eloquent#deleting-models](https://laravel.com/docs/5.2/eloquent%23deleting-models).

### Transactions

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

DB::transaction(**function** () {

*// Create a new position in the invoice*

$line = **new** App\InvoiceLine();

$line->CUSTOMER\_ID = 45;

$line->PRODUCT\_ID = 342;

$line->QUANTITY = 10;

$line->COST = 12.45;

$line->save();

*// add the sum of the line item to the amount of the invoice*

$invoice = App\Invoice::find($line->CUSTOMER\_ID);

$invoice->INVOICE\_SUM += $line->SUM\_PRICE;

$invoice->save();

});

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

## Creating Controllers and Configuring Routing

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

Route::get('/', **function** () {

**return** 'Hello World';

});

Route::post('foo/bar', **function** () {

**return** 'Hello World';

});

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

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

Route::match(['get', 'post'], 'foo/bar', **function** () {

**return** 'Hello World';

});

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

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

*//*

});

The parameters of a route are always enclosed in braces.

You can find more details about routing configuration in the [Routing](https://laravel.ru/docs/v5/routing) chapter of the documentation. Routes are configured in app/Http/routes.php file in Laravel 5.2 and in the routes/wep.php file in Laravel 5.3.

### Using Controllers to Route Requests

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

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

#### A Customer Controller

First we'll write our customer controller:

<?php

*/\**

*\* Customer controller*

*\*/*

**namespace** App\Http\Controllers;

**use** App\Http\Controllers\Controller;

**use** App\Customer;

**class** CustomerController **extends** Controller

{

/\*\*

\* Show customer list

\*

\* @return Response

\*/

**public function** showCustomers()

{

*// get the first 20 customers*

*// sorted alphabetically*

$customers = Customer::select()

->orderBy('NAME')

->take(20)

->get();

var\_dump($customers);

}

}

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

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

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

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

<?php

*/\**

*\* Customer Controller*

*\*/*

**namespace** App\Http\Controllers;

**use** App\Http\Controllers\Controller;

**use** App\Customer;

**class** CustomerController **extends** Controller {

/\*\*

\* Displays the list of customers

\*

\* @return Response

\*/

**public function** showCustomers() {

*// Connect widget for search*

$filter = \DataFilter::source(**new** Customer);

*// Search will be by the name of the supplier*

$filter->add('NAME', 'Name', 'text');

*// Set capture for search button*

$filter->submit('Search');

*// Add the filter reset button and assign it caption*

$filter->reset('Reset');

*// Create a grid to display the filtered data*

$grid = \DataGrid::source($filter);

*// output columns*

*// Field, label, sorted*

$grid->add('NAME', 'Name', **true**);

$grid->add('ADDRESS', 'Address');

$grid->add('ZIPCODE', 'Zip Code');

$grid->add('PHONE', 'Phone');

*// Add buttons to view, edit and delete records*

$grid->edit('/customer/edit', 'Edit', 'show|modify|delete');

*// Add the Add Customer button*

$grid->link('/customer/edit', "Add customer", "TR");

$grid->orderBy('NAME', 'asc');

*// set the number of records per page*

$grid->paginate(10);

*// display the customer template and pass the filter and grid to it*

**return** view('customer', compact('filter', 'grid'));

}

/\*\*

\* Add, edit and delete a customer

\*

\* @return Response

\*/

**public function** editCustomer() {

**if** (\Input::get('do\_delete') == 1)

**return** "not the first";

*// create an editor*

$edit = \DataEdit::source(**new** Customer());

*// Set title of the dialog, depending on the type of operation*

**switch** ($edit->status) {

**case** 'create':

$edit->label('Add customer');

**break**;

**case** 'modify':

$edit->label('Edit customer');

**break**;

**case** 'do\_delete':

$edit->label('Delete customer');

**break**;

**case** 'show':

$edit->label("Customer's card");

*// add a link to go back to the list of customers*

$edit->link('customers', 'Back', 'TR');

**break**;

}

*// set that after the operations of adding, editing and deleting,*

*// you need to return to the list of customers*

$edit->back('insert|update|do\_delete', 'customers');

*// We add editors of a certain type, assign them a label and*

*// associate them with the attributes of the model*

$edit->add('NAME', 'Name', 'text')->rule('required|max:60');

$edit->add('ADDRESS', 'Address', 'textarea')

->attributes(['rows' => 3])

->rule('max:250');

$edit->add('ZIPCODE', 'Zip code', 'text')->rule('max:10');

$edit->add('PHONE', 'Phone', 'text')->rule('max:14');

*// display the template customer\_edit and pass it to the editor*

**return** $edit->view('customer\_edit', compact('edit'));

}

}

##### Blade Templates

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

##### The Template for Displaying Customers

The template for displaying customers looks like this:

@**extends**('example')

@section('title', 'Customers')

@section('body')

<h1>Customers</h1>

<p>

{!! $filter !!}

{!! $grid !!}

</p>

@stop

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

@**extends**('master')

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

@section('body')

<h1>Пример</h1>

@**if**(Session::has('message'))

<div **class**="alert alert-success">

{!! Session::get('message') !!}

</div>

@endif

<p>Example of working with Firebird.<br/>

</p>

@stop

@section('content')

@include('menu')

@yield('body')

@stop

This template is itself inherited from the master template and also enables the menu template.

The menu is quite simple and consists of three items: Customers, Products and Invoices.

<nav **class**="navbar main">

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

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

data-toggle="collapse" data-target=".main-collapse">

<span **class**="sr-only"></span>

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

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

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

</button>

</div>

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

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

<li @**if** (Request::is('customer\*'))

**class**="active"@endif>{!! link\_to("customers", "Customers") !!}</li>

<li @**if** (Request::is('product\*'))

**class**="active"@endif>{!! link\_to("products", "Products") !!}</li>

<li @**if** (Request::is('invoice\*'))

**class**="active"@endif>{!! link\_to("invoices", "Invoices") !!}</li>

</ul>

</div>

</nav>

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

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

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

<title>@yield('title', 'An example of a Web application on Firebird')</title>

<meta name="description" content="@yield('description',

'An example of a Web application on Firebird')" />

@section('meta', '')

<link href="http://fonts.googleapis.com/css?family=Bitter" rel="stylesheet"

type="text/css" />

<link href="//netdna.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css"

rel="stylesheet">

<link href="//maxcdn.bootstrapcdn.com/font-awesome/4.1.0/css/font-awesome.min.css"

rel="stylesheet">

{!! Rapyd::styles(true) !!}

</head>

<body>

<div id="wrap">

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

<br />

<div **class**="row">

<div **class**="col-sm-12">

@yield('content')

</div>

</div>

</div>

</div>

<div id="footer">

</div>

<script src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">

</script>

<script src="//netdna.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js">

</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery.pjax/1.9.6/jquery.pjax.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/riot/2.2.4/riot+compiler.min.js"></script>

{!! Rapyd::scripts() !!}

</body>

</html>

The customer\_edit template:

@**extends**('example')

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

@section('body')

<p>

{!! $edit !!}

</p>

@stop

#### A Product Controller

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

<?php

*/\**

*\* Product Controller*

*\*/*

**namespace** App\Http\Controllers;

**use** App\Http\Controllers\Controller;

**use** App\Product;

**class** ProductController **extends** Controller {

/\*\*

\* Displays a list of products

\*

\* @return Response

\*/

**public function** showProducts() {

*// Connect widget for search*

$filter = \DataFilter::source(**new** Product);

*// The search will be by product name*

$filter->add('NAME', 'Name', 'text');

$filter->submit('Search');

$filter->reset('Reset');

*// Create a grid to display the filtered data*

$grid = \DataGrid::source($filter);

*// output grid columns*

*// Field, label, sorting*

$grid->add('NAME', 'Name', **true**);

*// Set the format with 2 decimal places*

$grid->add('PRICE|number\_format[2,., ]', 'Price');

$grid->row(**function**($row) {

*// Press the money values to the right*

$row->cell('PRICE')->style("text-align: right");

});

*// Add buttons to view, edit and delete records*

$grid->edit('/product/edit', 'Edit', 'show|modify|delete');

*// Add the Add product button*

$grid->link('/product/edit', "Add Product", "TR");

*// set sorting*

$grid->orderBy('NAME', 'asc');

*// set the number of records per page*

$grid->paginate(10);

*// display the customer template and pass the filter and grid to it*

**return** view('product', compact('filter', 'grid'));

}

/\*\*

\* Add, edit and delete products

\*

\* @return Response

\*/

**public function** editProduct() {

**if** (\Input::get('do\_delete') == 1)

**return** "not the first";

*// create editor*

$edit = \DataEdit::source(**new** Product());

*// Set the title of the dialog, depending on the type of operation*

**switch** ($edit->status) {

**case** 'create':

$edit->label('Add product');

**break**;

**case** 'modify':

$edit->label('Edit product');

**break**;

**case** 'do\_delete':

$edit->label('Delete product');

**break**;

**case** 'show':

$edit->label("Product's card");

$edit->link('products', 'Back', 'TR');

**break**;

}

*// set that after the operations of adding, editing and deleting,*

*// you need to return to the list of products*

$edit->back('insert|update|do\_delete', 'products');

*// We add editors of a certain type, assign them a label and*

*// associate them with the attributes of the model*

$edit->add('NAME', 'Name', 'text')->rule('required|max:100');

$edit->add('PRICE', 'Price', 'text')->rule('max:19');

$edit->add('DESCRIPTION', 'Description', 'textarea')

->attributes(['rows' => 8])

->rule('max:8192');

*// display the template product\_edit and pass it to the editor*

**return** $edit->view('product\_edit', compact('edit'));

}

}

#### A Controller for Invoices

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

<?php

*/\**

*\* Invoice controller*

*\*/*

**namespace** App\Http\Controllers;

**use** App\Http\Controllers\Controller;

**use** App\Invoice;

**use** App\Customer;

**use** App\Product;

**use** App\InvoiceLine;

**class** InvoiceController **extends** Controller {

/\*\*

\* Show invoice list

\*

\* @return Response

\*/

**public function** showInvoices() {

*// The invoice model will also select the related suppliers*

$invoices = Invoice::with('customer');

*// Add a widget for search.*

$filter = \DataFilter::source($invoices);

*// Let's filter by date range*

$filter->add('INVOICE\_DATE', 'Date', 'daterange');

*// and filter by customer name*

$filter->add('customer.NAME', 'Customer', 'text');

$filter->submit('Search');

$filter->reset('Reset');

*// Create a grid to display the filtered data*

$grid = \DataGrid::source($filter);

*// output grid columns*

*// Field, caption, sorted*

*// For the date we set an additional function that converts*

*// the date into a string*

$grid->add('INVOICE\_DATE|strtotime|date[Y-m-d H:i:s]', 'Date', **true**);

*// for money we will set a format with two decimal places*

$grid->add('TOTAL\_SALE|number\_format[2,., ]', 'Amount');

$grid->add('customer.NAME', 'Customer');

*// Boolean printed as Yes/No*

$grid->add('PAID', 'Paid')

->cell(**function**( $value, $row) {

**return** $value ? 'Yes' : 'No';

});

*// set the function of processing each row*

$grid->row(**function**($row) {

*// The monetary values are pressed to the right*

$row->cell('TOTAL\_SALE')->style("text-align: right");

*// paint the paid waybills in a different color*

**if** ($row->cell('PAID')->value == 'Yes') {

$row->style("background-color: #ddffee;");

}

});

*// Add buttons to view, edit and delete records*

$grid->edit('/invoice/edit', 'Edit', 'show|modify|delete');

*// Add the button for adding invoices*

$grid->link('/invoice/edit', "Add invoice", "TR");

$grid->orderBy('INVOICE\_DATE', 'desc');

*// set the number of records per page*

$grid->paginate(10);

*// display the customer template and pass the filter and grid to it*

**return** view('invoice', compact('filter', 'grid'));

}

/\*\*

\* Add, edit and delete invoice

\*

\* @return Response

\*/

**public function** editInvoice() {

*// get the text of the saved error, if it was*

$error\_msg = \Request::old('error\_msg');

*// create an invoice invoice editor*

$edit = \DataEdit::source(**new** Invoice());

*// if the invoice is paid, then we generate an error when trying to edit it*

**if** (($edit->model->PAID) && ($edit->status === 'modify')) {

$edit->status = 'show';

$error\_msg = 'Editing is not possible. The account has already been paid.';

}

*// if the invoice is paid, then we generate an error when trying to delete it*

**if** (($edit->model->PAID) && ($edit->status === 'delete')) {

$edit->status = 'show';

$error\_msg = 'Deleting is not possible. The account has already been paid.';

}

*// Set the label of the dialog, depending on the type of operation*

**switch** ($edit->status) {

**case** 'create':

$edit->label('Add invoice');

**break**;

**case** 'modify':

$edit->label('Edit invoice');

**break**;

**case** 'do\_delete':

$edit->label('Delete invoice');

**break**;

**case** 'show':

$edit->label('Invoice');

$edit->link('invoices', 'Back', 'TR');

*// If the invoice is not paid, we show the pay button*

**if** (!$edit->model->PAID)

$edit->link('invoice/pay/' . $edit->model->INVOICE\_ID,

'Pay', 'BL');

**break**;

}

*// set that after the operations of adding, editing and deleting,*

*// we return to the list of invoices*

$edit->back('insert|update|do\_delete', 'invoices');

*// set the "date" field, that it is mandatory*

*// The default is the current date*

$edit->add('INVOICE\_DATE', 'Дата', 'datetime')

->rule('required')

->insertValue(date('Y-m-d H:i:s'));

*// add a field for entering the customer. When typing a customer name,*

*// a list of prompts will be displayed*

$edit->add('customer.NAME', 'Customer', 'autocomplete')

->rule('required')

->options(Customer::lists('NAME', 'CUSTOMER\_ID')

->all());

*// add a field that will display the invoice amount, read-only*

$edit->add('TOTAL\_SALE', 'Amount', 'text')

->mode('readonly')

->insertValue('0.00');

*// add paid checkbox*

$paidCheckbox = $edit->add('PAID', 'Paid', 'checkbox')

->insertValue('0')

->mode('readonly');

$paidCheckbox->checked\_output = 'Yes';

$paidCheckbox->unchecked\_output = 'No';

*// create a grid to display the invoice line rows*

$grid = $this->getInvoiceLineGrid($edit->model, $edit->status);

*// we display the invoice\_edit template and pass the editor and grid to*

*// it to display the invoice invoice items*

**return** $edit->view('invoice\_edit', compact('edit', 'grid', 'error\_msg'));

}

/\*\*

\* Payment of invoice

\*

\* @return Response

\*/

**public function** payInvoice($id) {

**try** {

*// find the invoice by ID*

$invoice = Invoice::findOrFail($id);

*// call the payment procedure*

$invoice->pay();

} **catch** (\Illuminate\Database\QueryException $e) {

*// if an error occurs, select the exclusion text*

$pos = strpos($e->getMessage(), 'E\_INVOICE\_ALREADY\_PAYED');

**if** ($pos !== **false**) {

*// redirect to the editor page and display the error there*

**return** redirect('invoice/edit?show=' . $id)

->withInput(['error\_msg' => 'Invoice already paid']);

} **else**

**throw** $e;

}

*// redirect to the editor page*

**return** redirect('invoice/edit?show=' . $id);

}

/\*\*

\* Returns the grid for the invoice item

\* @param \App\Invoice $invoice

\* @param string $mode

\* @return \DataGrid

\*/

**private function** getInvoiceLineGrid(Invoice $invoice, $mode) {

*// Get invoice items*

*// For each ivoice item, the associated product will be initialized*

$lines = InvoiceLine::with('product')

->where('INVOICE\_ID', $invoice->INVOICE\_ID);

*// Create a grid for displaying invoice items*

$grid = \DataGrid::source($lines);

*// output grid columns*

*// Field, caption, sorted*

$grid->add('product.NAME', 'Name');

$grid->add('QUANTITY', 'Quantity');

$grid->add('SALE\_PRICE|number\_format[2,., ]', 'Price')

->style('min-width: 8em;');

$grid->add('SUM\_PRICE|number\_format[2,., ]', 'Amount')

->style('min-width: 8em;');

*// set the function of processing each row*

$grid->row(**function**($row) {

$row->cell('QUANTITY')->style("text-align: right");

*// The monetary values are pressed to the right*

$row->cell('SALE\_PRICE')->style("text-align: right");

$row->cell('SUM\_PRICE')->style("text-align: right");

});

**if** ($mode == 'modify') {

*// Add buttons to view, edit and delete records*

$grid->edit('/invoice/editline', 'Edit', 'modify|delete');

*// Add a button to add an invoice item*

$grid->link('/invoice/editline?invoice\_id=' . $invoice->INVOICE\_ID,

"Add item", "TR");

}

**return** $grid;

}

/\*\*

\* Add, edit and delete invoice items

\*

\* @return Response

\*/

**public function** editInvoiceLine() {

**if** (\Input::get('do\_delete') == 1)

**return** "not the first";

$invoice\_id = null;

*// create the editor of the invoice item*

$edit = \DataEdit::source(**new** InvoiceLine());

*// Set the label of the dialog, depending on the type of operation*

**switch** ($edit->status) {

**case** 'create':

$edit->label('Add invoice item');

$invoice\_id = \Input::get('invoice\_id');

**break**;

**case** 'modify':

$edit->label('Edit invoice item');

$invoice\_id = $edit->model->INVOICE\_ID;

**break**;

**case** 'delete':

$invoice\_id = $edit->model->INVOICE\_ID;

**break**;

**case** 'do\_delete':

$edit->label('Delete invoice item');

$invoice\_id = $edit->model->INVOICE\_ID;

**break**;

}

*// make url to go back*

$base = str\_replace(\Request::path(), '', strtok(\Request::fullUrl(), '?'));

$back\_url = $base . 'invoice/edit?modify=' . $invoice\_id;

*// set the page to go back*

$edit->back('insert|update|do\_delete', $back\_url);

$edit->back\_url = $back\_url;

*// add a hidden field with an invoice code*

$edit->add('INVOICE\_ID', '', 'hidden')

->rule('required')

->insertValue($invoice\_id)

->updateValue($invoice\_id);

*// Add a field for entering the goods. When you type the product name,*

*// a list of prompts is displayed.*

$edit->add('product.NAME', 'Name', 'autocomplete')

->rule('required')

->options(Product::lists('NAME', 'PRODUCT\_ID')->all());

*// Field for input quantity*

$edit->add('QUANTITY', 'Quantity', 'text')

->rule('required');

*// display the template invoice\_line\_edit and pass it to the editor*

**return** $edit->view('invoice\_line\_edit', compact('edit'));

}

}

##### The Invoice Editor

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

@**extends**('example')

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

@section('body')

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

{!! $edit->header !!}

@**if**($error\_msg)

<div **class**="alert alert-danger">

<strong>Error!</strong> {{ $error\_msg }}

</div>

@endif

{!! $edit->message !!}

@**if**(!$edit->message)

<div **class**="row">

<div **class**="col-sm-4">

{!! $edit->render('INVOICE\_DATE') !!}

{!! $edit->render('customer.NAME') !!}

{!! $edit->render('TOTAL\_SALE') !!}

{!! $edit->render('PAID') !!}

</div>

</div>

{!! $grid !!}

@endif

{!! $edit->footer !!}

</div>

@stop

#### Changing the Routes

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

Route::get('/', 'InvoiceController@showInvoices');

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

Route::any('/customer/edit', 'CustomerController@editCustomer');

Route::get('/products', 'ProductController@showProducts');

Route::any('/product/edit', 'ProductController@editProduct');

Route::get('/invoices', 'InvoiceController@showInvoices');

Route::any('/invoice/edit', 'InvoiceController@editInvoice');

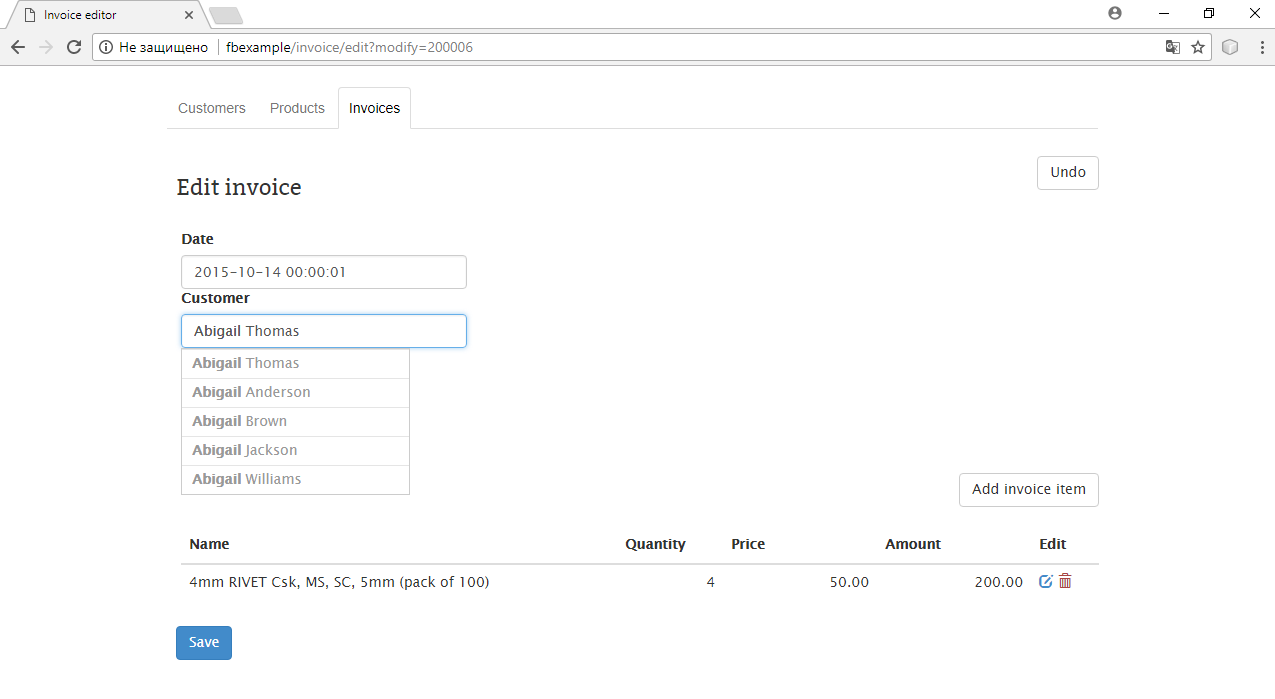
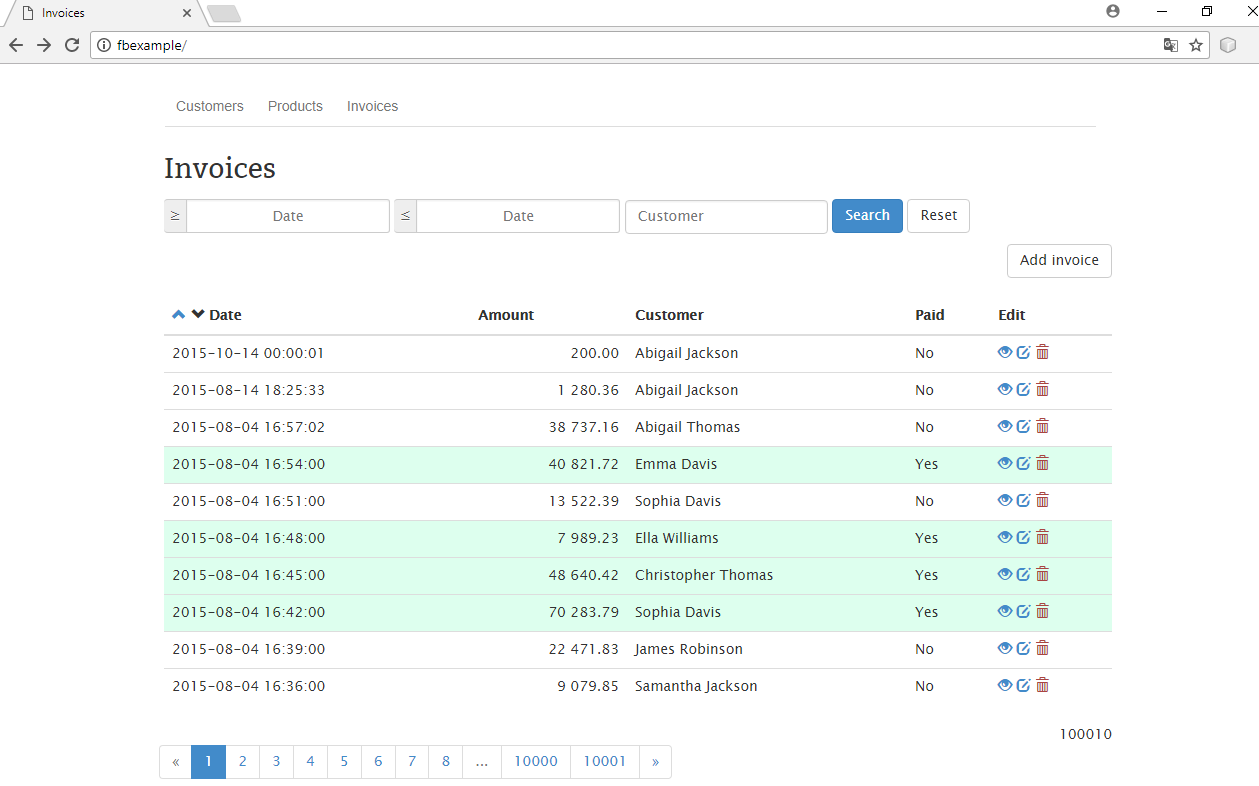
Route::any('/invoice/pay/{id}', 'InvoiceController@payInvoice');

Route::any('/invoice/editline', 'InvoiceController@editInvoiceLine');

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

## The Result

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



### Source Code

You can download the source code at <https://github.com/sim1984/phpfbexample>