Создание приложений для СУБД Firebird с использованием различных компонент и драйверов:

PHP и Laravel

Автор: Денис Симонов
12.12.2016

Оглавление

Создание web приложений на PHP

Привет. На этот раз мы рассмотрим процесс создания web приложения с использованием СУБД Firebird на языке PHP.

Обзор драйверов для работы с Firebird

В PHP есть два драйвера для работы с СУБД Firebird:

- Расширение Firebird/Interbase (ibase_ функции);
- PDO драйвер для Firebird.

Обзор расширения Firebird/Interbase

Расширение Firebird/Interbase появилось раньше и является наиболее проверенным.

Для установки расширения Firebird/Interbase в конфигурационном файле php.ini необходимо раскомментировать строку

```
extension=php_interbase.dll
```

или для UNIX подобных систем строку

```
extension=php_interbase.so
```

Это расширение требует, чтобы у вас была установлена клиентская библиотека fbclient.dll/gds32.dll (для UNIX подобных систем fbclient.so) соответствующей разрядности.

Замечание для пользователей Win32/Win64

Для работы этого расширения системной переменной Windows PATH должны быть доступны DLL-файлы fbclient.dll или gds32.dll. Хотя копирование DLL-файлов из директории PHP в системную папку Windows также решает проблему (потому что системная директория по умолчанию находится в переменной PATH), это не рекомендуется. Этому расширению требуются следующие файлы в переменной PATH: fbclient.dll или gds32.dll.

В Linux это расширение в зависимости от дистрибутива можно установить одной из следующих команд (необходимо уточнить поддерживаемые версии, возможно, необходимо подключить сторонний репозиторий):

```
apt-get install php5-firebird

rpm -ihv php5-firebird

yum install php70w-interbase

zypper install php5-firebird
```

Это расширение использует процедурный подход к написанию программ. Функции с префиксом ibase_ могут возвращать или принимать в качестве одного из параметров идентификатор соединения, транзакции, подготовленного запроса или курсора (результат SELECT запроса). Этот идентификатор имеет тип resource. Все выделенные ресурсы необходимо освобождать, как только они больше не требуются. Я не буду описывать каждую из функций подробно, вы можете посмотреть их описание по ссылке, вместо этого приведу несколько небольших примеров с комментариями.

```php
<?php

$db = 'localhost:example';
$username = 'SYSDBA';
$password = 'masterkey';

// Подключение к БД
$dbh = ibase_connect($db, $username, $password);
$sql = 'SELECT login, email FROM users';
// Выполняем запрос
$rc = ibase_query($dbh, $sql);
// Получаем результат построчно в виде объекта
while ($row = ibase_fetch_object($rc)) {
    echo $row->email, "\n";
}
// Освобождаем хэндл связанный с результатом запроса
ibase_free_result($rc);
// Освобождаем хэндл связанный с подключением
ibase_close($dbh);
```

Вместо функции ibase_connect вы можете применять функцию ibase_pconnect, которая создаёт так называемые постоянные соединения. В этом случае при вызове ibase_close соединение не закрывается, все связанные с ней ресурсы освобождаются, транзакция по умолчанию подтверждается, другие виды транзакций откатываются. Такое соединение может быть использовано повторно в другой сессии, если параметры подключения совпадают. В некоторых случаях постоянные соединения могут значительно повысить эффективность вашего веб приложения. Это особенно заметно, если затраты на установку соединения велики. Они позволяют дочернему процессу на протяжении всего жизненного цикла использовать одно и то же соединение вместо того, чтобы создавать его при обработке каждой страницы, которая взаимодействует с SQL-сервером. Этим постоянные соединения напоминают работу с пулом соединений. Подробнее о постоянных соединениях вы может прочитать по ссылке http://php.net/persistent-connections.

Внимание!

Многие ibase функции позволяют не передавать в них идентификатор соединения (транзакции, подготовленного запроса). В этом случае эти функции используют идентификатор последнего установленного соединения (начатой транзакции). Я не рекомендую так делать, в особенности, если ваше веб приложение может использовать более одного подключения.

Функция ibase_query выполняет SQL запрос и возвращает идентификатор результата или true, если запрос не возвращает набор данных. Эта функция помимо идентификатора подключения (транзакции) и текста SQL запроса может принимать переменное число аргументов в качестве значений параметров SQL запроса. В этом случае наш пример выглядит следующим образом:

```
…

$sql = 'SELECT login, email FROM users WHERE id=?';
$id = 1;
// Выполняем запрос
$rc = ibase_query($dbh, $sql, $id);
// Получаем результат построчно в виде объекта
if ($row = ibase_fetch_object($rc)) {
    echo $row->email, "\n";
}
// Освобождаем хэндл связанный с результатом запроса
ibase_free_result($rc);
…
```

Очень часто параметризованные запросы используются многократно с различным набором значений параметров, в этом случае для повышения производительности рекомендуется использовать подготовленные запросы. В этом случае сначала необходимо сначала получить идентификатор подготовленного запроса с помощью функции ibase_prepare, а затем выполнять подготовленный запрос с помощью функции ibase_execute.

```
…

$sql = 'SELECT login, email FROM users WHERE id=?';
// Подготавливаем запрос
$sth = ibase_prepare($dbh, $sql);
$id = 1;
// Выполняем запрос
$rc = ibase_execute($sth, $id);
// Получаем результат построчно в виде объекта
if ($row = ibase_fetch_object($rc)) {
    echo $row->email, "\n";
}
// Освобождаем хэндл связанный с результатом запроса
ibase_free_result($rc);
// Освобождаем подготовленный запрос
ibase_free_query($sth);
…
```

Подготовленные запросы гораздо чаще используются, когда необходима массовая заливка данных.

```
…

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
// Подготавливаем запрос
$sth = ibase_prepare($dbh, $sql);
$users = [["user1", "user1@gmail.com"], ["user2", "user2@gmail.com"]];
```

```
// Выполняем запрос
foreach ($users as $user)) {
  ibase_execute($sth, $user[0], $user[1]);
}
// Освобождаем подготовленный запрос
ibase_free_query($sth);
…
```

По последнему примеру можно увидеть один из недостатков этого расширения, а именно, функции с переменным числом аргументов не очень удобны для параметризованных запросов. Этот недостаток проявляется особенно ярко, если вы пытаетесь написать универсальный класс для исполнения любых запросов. Гораздо удобнее было бы, если параметры можно было передавать одним массивом. Конечно, существуют обходные пути вроде вот такого:

```
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;
}
```

Расширение Firebird/Interbase не работает с именованными параметрами запроса.

По умолчанию расширение Firebird/Interbase автоматически подтверждает транзакцию после выполнения каждого SQL запроса, если вам необходимо явное управление транзакциями, то необходимо стартовать транзакцию с помощью функции ibase_trans. Если параметры транзакции не указаны, то транзакция будет стартована с параметрами IBASE_WRITE | IBASE_CONCURRENCY | IBASE_WAIT. Описание констант для задания параметров транзакции можно найти здесь. Транзакцию необходимо завершать с помощью метода ibase_commit или ibase_rollback. Если вместо этих функций использовать функции ibase_commit_ret или ibase_rollback_ret, то транзакция будет завершаться как COMMIT RETAIN или ROLLBACK RETAIN.

Замечание.

Умолчательные параметры транзакции подходят для большинства случаев, и менять их параметры требуется очень редко. Дело в том что соединение с базой данных, как и все связанные с ним ресурсы существуют максимум до конца работы PHP скрипта. Даже если вы используете постоянные соединения, то все связанные ресурсы будут освобождены после вызова функции ibase_close. Несмотря на сказанное, настоятельно рекомендую завершать все выделенные ресурсы явно, вызывая соответствующие ibase_ функции.

Пользоваться функциями ibase_commit_ret и ibase_rollback_ret настоятельно не рекомендую, так как это не имеет смысла. COMMIT RETAIN и ROLLBACK RETAIN были введены для того, чтобы в настольных приложениях сохранять открытыми курсоры при завершении транзакции.

```
…

$sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
// Подготавливаем запрос
$sth = ibase_prepare($dbh, $sql);
$users = [["user1", "user1@gmail.com"], ["user2", "user2@gmail.com"]];
$trh = ibase_trans($dbh, IBASE_WRITE | IBASE_CONCURRENCY | IBASE_WAIT);
try {
  // Выполняем запрос
  foreach ($users as $user)) {
    ibase_execute($sth, $user[0], $user[1]);
    // Если произошла ошибка, бросаем исключение
    $err_msg = ibase_errmsg();
    if ($err_msg)
      throw new \Exception($err_msg);
  }
  ibase_commit($trh);
}
catch(\Exception $e) {
  ibase_rollback($trh);
     echo $e->getMessage();
}
// Освобождаем подготовленный запрос
ibase_free_query($sth);
…
```

Внимание!

ibase функции не бросают исключение в случае возникновения ошибки. Потенциально ошибка может возникнуть поле вызова любой ibase функции. Текст ошибки можно узнать с помощью функции ibase_errmsg. Код ошибки можно получить с помощью функции ibase_errcode.

Расширение Firebird/Interbase позволяет взаимодействовать с сервером Firebird не только посредством SQL запросов, но и используя Service API (см. функции `ibase_service_attach`, `ibase_service_detach`, `ibase_server_info`, `ibase_maintain_db`, `ibase_db_info`, `ibase_backup`, `ibase_restore`). Эти функции позволяют получить информацию о сервере Firebird, сделать резервное копирование, восстановление или получить статистику. Эта функциональность требуется в основном для администрирования БД, поэтому мы не будем рассматривать её подробно.

Расширение Firebird/Interbase так же поддерживает работу с событиями Firebird (см. функции `ibase_set_event_handler`, `ibase_free_event_handler`, `ibase_wait_event`).

Обзор расширения PDO (драйвер Firebird)

Расширение PDO предоставляет обобщённый интерфейс для доступа к различным типам БД. Каждый драйвер базы данных, в котором реализован этот интерфейс, может представить специфичный для базы данных функционал в виде стандартных функций расширения.

PDO и все основные драйверы внедрены в PHP как загружаемые модули. Чтобы их использовать, требуется их просто включить, отредактировав файл php.ini следующим образом:

```
extension=php_pdo.dll
```

Замечание

Этот шаг необязателен для версий PHP 5.3 и выше, так как для работы PDO больше не требуются DLL.

Далее нужно выбрать DLL конкретных баз данных и либо загружать их во время выполнения функцией dl(), либо включить их в php.ini после php_pdo.dll. Например:

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

Эти DLL должны лежать в директории extension_dir.

Драйвер pdo_firebird требует, чтобы у вас была установлена клиентская библиотека fbclient.dll/gds32.dll (для UNIX подобных систем fbclient.so) соответствующей разрядности.

В Linux это расширение в зависимости от дистрибутива можно установить одной из следующих команд (необходимо уточнить поддерживаемые версии, возможно, необходимо подключить сторонний репозиторий):

```
apt-get install php5-firebird

rpm –ihv php5-firebird

yum install php70w-firebird

zypper install php5-firebird
```

PDO использует объектно-ориентированный подход к написанию программ. Какой именно драйвер будет использоваться в PDO, зависит от строки подключения, называемой так же DSN (Data Source Name). DSN состоит из префикса, который и определяет тип базы данных, и набора параметров в виде `<ключ>=<значение>`, разделённых точкой с запятой «;». Допустимый набор параметров зависит от типа базы данных. Для работы с Firebird строка подключения должна начинаться с префикса *firebird:* и иметь вид, описанный в документации в разделе PDO_FIREBIRD DSN.

Соединения устанавливаются автоматически при создании объекта PDO от его базового класса. Конструктор класса принимает аргументы для задания источника данных (DSN), а также необязательные имя пользователя и пароль (если есть). Четвёртым аргументом можно передать массив специфичных для драйвера настроек подключения в формате ключ=>значение.

```
<?php
```

```php
$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Подключение к БД
  $dbh = new \PDO($dsn, $username, $password, [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  $sql = 'SELECT login, email FROM users';
  // Выполняем запрос
  $query = $dbh->query($sql);
  // Получаем результат построчно в виде объекта
  while ($row = $query->fetch(\PDO::FETCH_OBJ)) {
    echo $row->email, "\n";
  }
    $query->closeCursor(); // Закрываем курсор
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

Установив свойство `\PDO::ATTR_ERRMODE` в значение `\PDO::ERRMODE_EXCEPTION`, мы установили режим, при котором любая ошибка, в том числе и ошибка при подключении к БД, будет возбуждать исключение \PDOException. Работать в таком режиме гораздо удобнее, чем проверять наличие ошибки после каждого вызова ibase_ функций.

Замечание

Для того чтобы PDO использовал постоянные соединения необходимо в конструктор PDO в массиве свойств передать `PDO::ATTR_PERSISTENT => true`.

Метод query выполняет SQL запрос и возвращает результирующий набор в виде объекта \PDOStatement. В этот метод помимо SQL запросы вы можете передать способ возвращения значений при фетче. Это может быть столбец, экземпляр заданного класса, объект. Различные способы вызова вы можете посмотреть в документации.

Если необходимо выполнить SQL запрос, не возвращающий набор данных, то вы можете воспользоваться методом exec, который возвращает количество задействованных строк. Этот метод не обеспечивает выполнение подготовленных запросов.

Если в запросе используются параметры, то необходимо пользоваться подготовленными запросами. В этом случае вместо метода query необходимо вызвать метод prepare. Этот метод возвращает объект класса \PDOStatement, который инкапсулирует в себе методы для работы с подготовленными запросами и их результатами. Для выполнения запроса необходимо вызвать метод execute, который может принимать в качестве аргумента массив с именованными или неименованными параметрами. Результат выполнения селективного запроса можно получить с помощью методов fetch, fetchAll, fetchColumn, fetchObject. Методы fetch и fetchAll могут возвращать результаты в различном виде:

ассоциативный массив, объект или экземпляр определённого класса. Последнее довольно часто используется в MVC паттерне при работе с моделями.

```
<?php

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Подключение к БД
  $dbh = new \PDO($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"]
  ];

  // Подготавливаем запрос
  $query = $dbh->prepare($sql);
  // Выполняем запрос
  foreach ($users as $user)) {
      $query->execute($user);
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

Пример использования именованных параметров.

```
<?php

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Подключение к БД
  $dbh = new \PDO($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"]
  ];
  // Подготавливаем запрос
  $query = $dbh->prepare($sql);
  // Выполняем запрос
  foreach ($users as $user)) {
      $query->execute($user);
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

Замечание

Для поддержки именованных параметров PDO производит предобработку запроса и заменяет параметры вида :paramname на «?», сохраняя при этом массив соответствия между именем параметра и номерами его позиций в запросе. По этой причине оператор EXECUTE BLOCK не будет работать, если внутри него используются переменные маркированные двоеточием. На данный момент нет никакой возможности заставить работать PDO с оператором EXECUTE BLOCK иначе, например, задать альтернативный префикс параметров, как это сделано в некоторых компонентах доступа.

Передать параметры в запрос можно и другим способом, используя так называемое связывание. Метод bindValue привязывает значение к именованному или неименованному параметру. Метод bindParam привязывает переменную к именованному или неименованному параметру. Последний метод особенно полезен для хранимых процедур, которые возвращают значение через OUT или IN OUT параметр (в Firebird механизм возврата значений из хранимых процедур другой).

```php
<?php

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Подключение к БД
  $dbh = new \PDO($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"]
  ];

  // Подготавливаем запрос
  $query = $dbh->prepare($sql);
  // Выполняем запрос
  foreach ($users as $user)) {
    $query->bindValue(":login", $user[0]);
    $query->bindValue(":email", $user[1]);
    $query->execute();
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

Внимание

Нумерация неименованных параметров в методах bindParam и bindValue начинается с 1.

```php
<?php

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Подключение к БД
  $dbh = new \PDO($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"]
  ];

  // Подготавливаем запрос
  $query = $dbh->prepare($sql);
  // Выполняем запрос
  foreach ($users as $user)) {
    $query->bindValue(1, $user[0]);
    $query->bindValue(2, $user[1]);
    $query->execute();
  }
} catch (\PDOException $e) {
  echo $e->getMessage();
}
```

По умолчанию PDO автоматически подтверждает транзакцию после выполнения каждого SQL запроса, если вам необходимо явное управление транзакциями, то необходимо стартовать транзакцию с помощью метода \PDO::beginTransaction. По умолчанию транзакция стартует с параметрами CONCURRENCY | WAIT | READ_WRITE. Завершить транзакцию можно методом \PDO::commit или \PDO::rollback.

```php
<?php

$dsn = 'firebird:dbname=localhost:example;charset=utf8;';
$username = 'SYSDBA';
$password = 'masterkey';
try {
  // Подключение к БД
  $dbh = new \PDO($dsn, $username, $password, [\PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION]);
  // Стартуем транзакцию для обеспечения согласованности между запросами
  $dbh->beginTransaction();
  // Получаем пользователей из одной таблицы
  $users_stmt = $dbh->prepare('SELECT login, email FROM old_users');
  $users_stmt->execute();
  $users = $users_stmt->fetchAll(\PDO::FETCH_OBJECT);
  $users_stmt->closeCursor();

  // И переносим их в другую
  $sql = 'INSERT INTO users(login, email) VALUES(?, ?)';
  // Подготавливаем запрос
  $query = $dbh->prepare($sql);
  // Выполняем запрос
  foreach ($users as $user)) {
    $query->bindValue(1, $user->LOGIN);
    $query->bindValue(2, $user->EMAIL]);
    $query->execute();
  }
  // Подтверждаем транзакцию
  $dbh->commit();
} catch (\PDOException $e) {
  // Если соединение произошло и транзакция стартовала, откатываем её
  if ($dbh &&  $dbh->inTransaction())
    $dbh->rollback();
  echo $e->getMessage();
}
```

К сожалению метод beginTransaction не предоставляет возможности изменить параметры транзакции, однако вы можете сделать хитрый трюк, задав параметры транзакции оператором SET TRANSACTION.

```php
$dbh = new \PDO($dsn, $username, $password);
$dbh->setAttribute(\PDO::ATTR_AUTOCOMMIT, false);
$dbh->exec("SET TRANSACTION READ ONLY ISOLATION LEVEL READ COMMITTED NO WAIT");
// Выполняем действия в транзакции
// ….
$dbh->exec("COMMIT");
$dbh->setAttribute(\PDO::ATTR_AUTOCOMMIT, true);
```

Ниже представлена сводная таблица возможностей различных драйверов для работы с Firebird.

Возможность	**Расширение Firebird/Interbase**	**PDO**

Парадигма программирования	Функциональная	Объектно-ориентированная
Поддерживаемые БД	Firebird, Interbase, Yaffil и другие клоны Interbase.	Любая БД, для которой существует PDO драйвер, в том числе Firebird.
Работа с параметрами запросов	Только неименованные параметры, работать не очень удобно, поскольку используется функция с переменным числом аргументов.	Есть возможность работать как с именованными, так и неименованными параметрами. Работать очень удобно, однако некоторые возможности Firebird (оператор EXECUTE BLOCK) не работают.
Обработка ошибок	Проверка результата функций ibase_errmsg, ibase_errcode. Ошибка может произойти после вызова любой ibase функции при этом исключение не будет возбуждено.	Есть возможность установить режим, при котором любая ошибка приведёт к возбуждению исключения.
Управление транзакциями	Даёт возможность задать параметры транзакции.	Не даёт возможность задать параметры транзакции. Есть обходной путь через выполнение оператора SET TRANSACTION.
Специфичные возможности Interbase/Firebird	Есть возможность работать с расширениями Service API (backup, restore, получение статистики и т.д.), а также с событиями базы данных.	Не позволяет использовать специфичные возможности, с которыми невозможно работать, используя SQL.

Из приведённой таблицы видно, что большинству фреймворков гораздо удобнее пользоваться PDO.

Выбор фреймворка для построения WEB приложения

Небольшие web сайты можно писать, не используя паттерн MVC. Однако чем больше становится ваш сайт, тем сложнее его поддерживать, особенно если над ним работает не один человек. Поэтому при разработке нашего web приложения сразу договоримся об использовании этого паттерна.

Итак, мы решили использовать паттерн MVC. Однако написание приложение с использованием этого паттерна не такая простая задача как кажется, особенно если мы не пользуемся сторонними библиотеками. Если всё писать самому, то необходимо решить множество задач: автозагрузка файлов .php, включающих определение классов, маршрутизация и др. Для решения этих задач было создано большое количество фреймворков, например Yii, Laravel, Symphony, Kohana и многие другие. Лично мне нравится Laravel, поэтому далее я буду описывать создание приложения с использованием этого фреймворка.

Установка Laravel и создание проекта

Прежде чем устанавливать Laravel вам необходимо убедится, что ваше системное окружение соответствует требованиям.

- PHP >= 5.5.9
- PDO расширение для PHP (для версии 5.1+)
- MCrypt расширение для PHP (для версии 5.0)
- OpenSSL (расширение для PHP)
- Mbstring (расширение для PHP)
- Tokenizer (расширение для PHP)

Laravel использует Composer для управления зависимостями. Поэтому сначала установите Composer, а затем Laravel.

Самый простой способ установить composer под windows – это скачать и запустить инсталлятор Composer-Setup.exe. Инсталлятор установит Composer и настроит PATH, так что вы можете вызвать Composer из любой директории в командной строке.

Если необходимо установить Composer вручную, то необходимо запустить скрипт

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'aa96f26c2b67226a324c27919f1eb05f21c248b987e6195cad9690d5c1ff713d53020a02ac8c217dbf90a7eacc9d141d
') { echo 'Installer verified'; } else { echo 'Installer corrupt'; unlink('composer-setup.php');
} echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Этот скрипт осуществляет следующие действия:

- Скачивает инсталлятор в текущую директорию
- Проверяет инсталлятор с помощью SHA-384
- Запускает скрипт инсталляции
- Удаляет скрипт инсталляции

После запуска этого скрипта у вас появится файл composer.phar (phar – это архив) — по сути это PHP скрипт, который может принимать несколько команд (install, update, ...) и умеет скачивать и распаковывать библиотеки. Если вы работаете под windows, то вы можете облегчить себе задачу, создав файл composer.bat и поместив его в PATH. Для этого необходимо выполнить команду

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

Подробнее об установке composer смотри здесь.

Теперь устанавливаем сам Laravel

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

Если установка прошла успешно, то приступаем к созданию каркаса проекта.

```
laravel new fbexample
```

Ждём завершения, после чего у нас будет создан каркас проекта. Описание структуры каталогов можно найти в документации по Laravel. Нас будут интересовать следующие каталоги:

app – основной каталог нашего приложения. В корне каталога будут размещены модели. В подкаталоге Http находится все, что касается работы с браузером. В подкаталоге Http/Controllers – наши контроллеры.

config – каталог файлов конфигурации. Подробней о конфигурировании будет написано ниже.

public – корневой каталог web приложения (DocumentRoot). Содержит статические файлы - css, js, изображения и т.п.

resources - здесь находятся шаблоны (Views), файлы локализации и, если таковые имеются, рабочие файлы LESS, SASS и js-приложения на фреймворках типа ReactJS, AngularJS или Ember, которые потом собираются внешним инструментом в папку public.

В корне папки нашего приложения есть файл composer.json, который описывает, какие пакеты, потребуются нашему приложению помимо тех, что уже есть в Laravel. Нам потребуется два таких пакета: «zofe/rapyd-laravel» - для быстрого построения интерфейса с сетками (grids) и диалогами редактирования, и «sim1984/laravel-firebird» - расширение для работы с СУБД Firebird. Пакет «sim1984/laravel-firebird» является форком пакета «jacquestvanzuydam/laravel-firebird» поэтому его установка несколько отличается (описание отличий пакета от оригинального вы можете найти в статье «Пакет для работы с СУБД Firebird в Laravel»). Не забудьте установить параметр minimum-stability равный dev, так как пакет не является стабильным, а так же добавить ссылки на репозиторий.

```
…
"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": [""]
                }
            }
        }
    ],
    …
```

В секции require добавьте требуемые пакеты следующим образом:

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

Теперь можно запустить обновление пакетов командой (запускать надо в корне веб приложения)

```
composer update
```

После выполнения этой команды новые пакеты будут установлены в ваше приложение. Теперь можно приступить к настройке. Для начала выполним команду

```
php artisan vendor:publish
```

которая создаст дополнительные файлы конфигурации для пакета zofe/rapyd.

В файле config/app.php добавим два новых провайдера. Для этого добавим две новых записи в ключ providers

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

Теперь перейдём к файлу config/databases.conf, который содержит настройки подключения к базе данных. Добавим в ключ connections следующие строки

```
       …
       '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',
        ],
        …
```

Поскольку мы будем использовать наше подключение в качестве подключения по умолчанию, установим следующее

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

Обратите внимание на функцию env, которая используется для чтения переменных окружения приложения из специального файла .env, находящегося в корне проекта. Исправим в этом файле .env следующие строки

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

В файле конфигурации config/rapyd.php изменим отображение дат так, чтобы они были в формате принятом в России:

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

Первоначальная настройка закончена, теперь мы можем приступить непосредственно к написанию логики web приложения.

Создание моделей

Фреймворк Laravel поддерживает ORM Eloquent. ORM Eloquent - красивая и простая реализация паттерна ActiveRecord для работы с базой данных. Каждая таблица имеет соответствующий класс-модель, который используется для работы с этой таблицей. Модели позволяют читать данные из таблиц и записывать данные в таблицу.

Создадим модель заказчиков, для упрощения этого процесса в Laravel есть artisan команда.

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

Этой командой мы создаём шаблон модели. Теперь изменим нашу модель так, чтобы она выглядела следующим образом:

```
namespace App;

use Firebird\Eloquent\Model;

class Customer extends Model
{
    /**
     * Таблица, связанная с моделью
     *
     * @var string
     */
    protected $table = 'CUSTOMER';

    /**
     * Первичный ключ модели
     *
     * @var string
     */
    protected $primaryKey = 'CUSTOMER_ID';

    /**
     * Наша модель не имеет временной метки
     *
     * @var bool
     */
    public $timestamps = false;

    /**
     * Имя последовательности для генерации первичного ключа

     * @var string
     */
    protected $sequence = 'GEN_CUSTOMER_ID';
}
```

Обратите внимание, мы используем модифицированную модель Firebird\Eloquent\Model из пакета sim1984/laravel-firebird в качестве базовой. Она позволяет воспользоваться последовательностью, указанной в свойстве $sequence, для генерирования значения идентификатора первичного ключа.

По аналогии создадим модель товаров – Product.

```
namespace App;

use Firebird\Eloquent\Model;

class Product extends Model
{
    /**
     * Таблица, связанная с моделью
     *
     * @var string
     */
    protected $table = 'PRODUCT';

    /**
     * Первичный ключ модели
     *
     * @var string
     */
    protected $primaryKey = 'PRODUCT_ID';

    /**
     * Наша модель не имеет временной метки
```

```
     *
     * @var bool
     */
    public $timestamps = false;

    /**
     * Имя последовательности для генерации первичного ключа

     * @var string
     */
    protected $sequence = 'GEN_PRODUCT_ID';
}
```

Теперь создадим модель для шапки счёт-фактуры.

```
namespace App;

use Firebird\Eloquent\Model;

class Invoice extends Model {

    /**
     * Таблица, связанная с моделью
     *
     * @var string
     */
    protected $table = 'INVOICE';

    /**
     * Первичный ключ модели
     *
     * @var string
     */
    protected $primaryKey = 'INVOICE_ID';

    /**
     * Наша модель не имеет временной метки
     *
     * @var bool
     */
    public $timestamps = false;

    /**
     * Имя последовательности для генерации первичного ключа
     *
     * @var string
     */
    protected $sequence = 'GEN_INVOICE_ID';

    /**
     * Заказчик
     *
     * @return \App\Customer
     */
    public function customer() {
        return $this->belongsTo('App\Customer', 'CUSTOMER_ID');
    }

    /**
     * Позиции счёт фактуры

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

    /**
     * Оплата
     */
    public function pay() {
        $connection = $this->getConnection();

        $attributes = $this->attributes;

        $connection->executeProcedure('SP_PAY_FOR_INOVICE', [$attributes['INVOICE_ID']]);
    }
```

```
}
```

В этой модели можно заметить несколько дополнительных функций. Функция customer возвращает заказчика связанного со счёт фактурой через поле CUSTOMER_ID. Для осуществления такой связи используется метод belongsTo, в который передаются имя класса модели и имя поле связи. Функция lines возвращают позиции счёт-фактуры, которые представлены коллекцией моделей InvoiceLine (будет описана далее). Для осуществления связи один ко многим в функции lines используется метод hasMany, в который передаётся имя класса модели и поле связи. Подробнее о задании отношений между сущностями вы можете почитать в разделе Отношения документации Laravel.

Функция pay осуществляет оплату счёт фактуры. Для этого вызывается хранимая процедура SP_PAY_FOR_INVOICE. В неё передаётся идентификатор счёт фактуры. Значение любого поля (атрибута модели) можно получить из свойства attributes. Вызов хранимой процедуры осуществляется с помощью метода executeProcedure. Этот метод доступен только при использовании расширения sim1984/laravel-firebird.

Теперь создадим модель для позиций счёт фактуры.

```php
namespace App;

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

class InvoiceLine extends Model {

    /**
     * Таблица, связанная с моделью
     *
     * @var string
     */
    protected $table = 'INVOICE_LINE';

    /**
     * Первичный ключ модели
     *
     * @var string
     */
    protected $primaryKey = 'INVOICE_LINE_ID';

    /**
     * Наша модель не имеет временной метки
     *
     * @var bool
     */
    public $timestamps = false;

    /**
     * Имя последовательности для генерации первичного ключа
     *
     * @var string
     */
    protected $sequence = 'GEN_INVOICE_LINE_ID';

    /**
      * Массив имён вычисляемых полей
      *
      * @var array
      */
    protected $appends = ['SUM_PRICE'];

    /**
     * Товар
```

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

    /**
     * Сумма по позиции
     *
     * @return double
     */
    public function getSumPriceAttribute() {
        return $this->SALE_PRICE * $this->QUANTITY;
    }

    /**
     * Добавление объекта модели в БД
     * Переопределяем этот метод, т.к. в данном случаем мы работаем с помощью ХП
     *
     * @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;
    }

    /**
     * Сохранение изменений текущего экземпляра модели в БД
     * Переопределяем этот метод, т.к. в данном случаем мы работаем с помощью ХП
     *
     * @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);
        }
    }

    /**
     * Удаление текущего экземпляра модели в БД
     * Переопределяем этот метод, т.к. в данном случаем мы работаем с помощью ХП
     *
     * @return void
     */
    protected function performDeleteOnModel() {

        $connection = $this->getConnection();

        $attributes = $this->attributes;

        $connection->executeProcedure('SP_DELETE_INVOICE_LINE',
            [$attributes['INVOICE_LINE_ID']]);

    }

}
```

В этой модели есть функция product, которая возвращает продукт (модель App/Product), указанный в позиции счёт фактуры. Связь осуществляется по полю PRODUCT_ID с помощью метода belongsTo.

Вычисляемое поле SumPrice вычисляется с помощью функции getSumPriceAttribute. Для того чтобы это вычисляемое поле было доступно в модели, его имя должно быть указано в массиве имён вычисляемых полей $appends.

В этой модели мы переопределили операции insert, update и delete так, чтобы они выполнялись, используя хранимые процедуры. Эти хранимые процедуры помимо собственно операций вставки, редактирования и удаления пересчитывают сумму в шапке накладной. Этого можно было бы и не делать, но тогда пришлось бы выполнять в одной транзакции модификацию нескольких моделей. Как это сделать будет показано далее.

Теперь немного поговорим о том, как работать с моделями в Laravel для выборки, вставки, редактирования и удаления данных. Laravel оперирует данными с помощью конструктора запросов. Полное описание синтаксиса и возможностей этого конструктора вы можете найти по ссылке https://laravel.ru/docs/v5/queries. Например, для получения всех строк поставщиков вы можете выполнить следующий запрос

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

Этот конструктор запросов является довольно мощным средством для построения и выполнения SQL запросов. Вы можете выполнять также фильтрация, сортировку и соединения таблиц, например

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

Однако гораздо удобнее работать с использованием моделей. Описание моделей Eloquent ORM и синтаксиса запроса к ним можно найти по ссылке https://laravel.ru/docs/v5/eloquent. Так для получения всех элементов коллекции поставщиков необходимо выполнить следующий запрос

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

Следующий запрос вернёт первые 20 поставщиков отсортированных по алфавиту.

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

Для сложных моделей связанные отношения или коллекции отношений могут быть получены через динамические атрибуты. Например, следующий запрос вернёт позиции счёт-фактуры с идентификатором 1.

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

Добавление записей осуществляется через создание экземпляра модели, инициализации его свойств и сохранение модели с помощью метода save.

```
$flight = new Flight;

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

$flight->save();
```

Для изменения запись её необходимо найти, изменить необходимые атрибуты и сохранить методом save.

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

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

$flight->save();
```

Для удаления записи её необходимо найти и вызвать метод delete.

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

$flight->delete();
```

Удалить запись по ключу можно и гораздо быстрее с помощью метода destroy. В этом случае можно удалить модель не получая её экземпляр.

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

Существуют и другие способы удаления записей, например «мягкое» удаление. Подробно о способах удаления вы можете прочитать по ссылке https://laravel.ru/docs/v5/eloquent#удаление.

Теперь поговорим немного о транзакциях. Что это такое я рассказывать не буду, а лишь покажу, как их можно использовать совместно с Eloquent ORM.

```
DB::transaction(function () {
  // Создаём новую позицию в счёт фактуре
  $line = new App\InvoiceLine();
  $line->CUSTOMER_ID = 45;
  $line->PRODUCT_ID = 342;
  $line->QUANTITY = 10;
  $line->COST = 12.45;
  $line->save();

  // добавляем сумму позиции по строке к сумме накладной
  $invoice = App\Invoice::find($line->CUSTOMER_ID);
  $invoice->INVOICE_SUM += $line->SUM_PRICE;
  $invoice->save();

});
```

Всё что находится в функции обратного вызова, которая является аргументом метода transaction, выполняется в рамках одной транзакции.

Создание контроллеров и настройка маршрутизации

Фреймворк Laravel имеет мощную подсистему маршрутизации. Вы можете отображать ваши маршруты, как на простые функции обратного вызова, так и на методы контроллеров. Простейшие примеры маршрутов выглядят вот так

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

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

В первом случае мы регистрируем обработчик GET запроса для корня сайта, во втором – для POST запроса с маршрутом /foo/bar.

Вы можете зарегистрировать маршрут сразу на несколько типов HTTP запросов, например

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

Из маршрута можно извлекать часть адреса и использовать его в качестве параметров функции-обработчика

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

Параметры маршрута всегда заключаются в фигурные скобки.

Подробнее о возможности настройки маршрутизации вы можете посмотреть в документации глава «Маршрутизация». Маршруты настраиваются в файле app/Http/routes.php в Laravel 5.2 и routes/wep.php в Laravel 5.3.

Вместо того чтобы описывать обработку всех запросов в едином файле маршрутизации, мы можем организовать её использую классы Controller, которые позволяют группировать связанные обработчики запросов в отдельные классы. Контроллеры хранятся в папке app/Http/Controllers.

Все Laravel контроллеры должны расширять базовый класс контроллера App\Http\Controllers\Controller, присутствующий в Laravel по умолчанию. Подробнее о написании контроллеров вы можете почитать по ссылке http://laravel.su/docs/5.2/controllers.

Напишем наш первый контроллера заказчиков.

```
<?php

/*
 * Контроллер заказчиков
 */

namespace App\Http\Controllers;

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


class CustomerController extends Controller
{

    /**
     * Отображает список заказчиков
     *
     * @return Response
     */
    public function showCustomers()
    {
        // запрашиваем из модели первые 20 заказчиков
        // отсортированных по алфавиту
        $customers = Customer::select()
                    ->orderBy('NAME')
                     ->take(20)
                    ->get();
         var_dump($customers);
    }

}
```

Теперь необходимо связать методы контроллера с маршрутом. Для этого в routes.php (web.php) необходимо внести строку

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

Здесь имя контроллера отделено от имени метода символом @.

Для быстрого построения интерфейса с сетками и диалогами редактирования будем использовать пакет «zofe/rapyd». Мы его уже подключили ранее. Классы пакета zofe/rapyd берут на себя построение типичных запросов к моделям Eloquent ORM. Изменим контроллер заказчиков так, чтобы он выводил данные в сетку (grid), позволял производить их фильтрацию, а также добавлять, редактировать и удалять записи через диалоги редактирования.

```php
<?php

/*
 * Контроллер заказчиков
 */

namespace App\Http\Controllers;

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


class CustomerController extends Controller {

    /**
     * Отображает список заказчиков
     *
     * @return Response
     */
    public function showCustomers() {
         // Подключаем виджет для поиска
        $filter = \DataFilter::source(new Customer);
        // Поиск будет по наименованию поставщика
        $filter->add('NAME', 'Наименование', 'text');
         // Задаём подпись кнопке поиска
        $filter->submit('Поиск');
         // Добавляем кнопку сброса фильтра и задаём её подпись
        $filter->reset('Сброс');

         // Создам сетку для отображения отфильтрованных данных
        $grid = \DataGrid::source($filter);

        // выводимые столбцы
        // Поле, подпись, сортируемый
        $grid->add('NAME', 'Наименование', true);
        $grid->add('ADDRESS', 'Адрес');
        $grid->add('ZIPCODE', 'Индекс');
        $grid->add('PHONE', 'Телефон');

        // Добавляем кнопки для просмотра, редактирования и удаления записи
        $grid->edit('/customer/edit', 'Редактирование', 'show|modify|delete');
        // Добавляем кнопку добавления заказчика
        $grid->link('/customer/edit', "Добавление заказчика", "TR");
        // задаём сортировку
        $grid->orderBy('NAME', 'asc');
        // задаём количество записей на страницу
        $grid->paginate(10);
        // отображаем шаблон customer и передаём в него фильтр и грид
        return view('customer', compact('filter', 'grid'));
    }

    /**
     * Добавление, редактирование и удаление заказчика
     *
     * @return Response
     */
    public function editCustomer() {
        if (\Input::get('do_delete') == 1)
            return "not the first";
        // создаём редактор
        $edit = \DataEdit::source(new Customer());
        // задаём подпись диалога в зависимости от типа операции
        switch ($edit->status) {
            case 'create':
                $edit->label('Добавление заказчика');
                break;
            case 'modify':
                $edit->label('Редактирование заказчика');
                break;
            case 'do_delete':
                $edit->label('Удаление заказчика');
                break;
            case 'show':
                $edit->label('Карточка заказчика');
                   // добавляем ссылку для возврата назад на список заказчиков
                $edit->link('customers', 'Назад', 'TR');
                break;
        }
        // задаём что после операций добавления, редактирования и удаления
        // возвращаемся к списку заказчиков
```

```
        $edit->back('insert|update|do_delete', 'customers');
        // Добавляем редакторы определённого типа, задаём им подпись
        // и связываем их с атрибутами модели
        $edit->add('NAME', 'Наименование', 'text')->rule('required|max:60');
        $edit->add('ADDRESS', 'Адрес', 'textarea')
             ->attributes(['rows' => 3])
             ->rule('max:250');
        $edit->add('ZIPCODE', 'Индекс', 'text')->rule('max:10');
        $edit->add('PHONE', 'Телефон', 'text')->rule('max:14');
        // отображаем шаблон customer_edit и передаём в него редактор
        return $edit->view('customer_edit', compact('edit'));
    }
}
```

Laravel по умолчанию использует шаблонизатор blade. Функция view находит необходимый шаблон в директории resources/views, делает необходимые замены в нём и возвращает текст HTML страницы. Кроме того, она передаёт в него переменные, которые становятся доступными в шаблоне. Описание синтаксиса шаблонов blade вы можете найти по адресу http://laravel.su/docs/5.3/blade.

Шаблон для отображения заказчиков выглядит следующим образом:

```
@extends('example')

@section('title','Заказчики')

@section('body')

    <h1>Заказчики</h1>
    <p>
        {!! $filter !!}
        {!! $grid !!}
    </p>
@stop
```

Данный шаблон унаследован от шаблона example и переопределяет его секцию body. Переменные $filter и $grid содержат HTML код для осуществления фильтрации и отображения данных в сетке. Шаблон example является общим для всех страниц.

```
@extends('master')
@section('title', 'Пример работы с Firebird')

@section('body')

  <h1>Пример</h1>

  @if(Session::has('message'))
    <div class="alert alert-success">
        {!! Session::get('message') !!}
    </div>
  @endif

  <p>Пример работы с Firebird.<br/>
  </p>
@stop


@section('content')

    @include('menu')

    @yield('body')

@stop
```

Этот шаблон сам унаследован от шаблона master, кроме того он подключает шаблон menu.

Меню довольно простое, состоит из трёх пунктов Заказчики, Продукты и Счёт фактуры.

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

В шаблоне master подключаются css стили и JavaScript файлы с библиотеками.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>@yield('title', 'Пример Web приложения на Firebird')</title>
        <meta name="description" content="@yield('description', 'Пример Web приложения на
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>
```

Шаблон редактора заказчика customer_edit выглядит следующим образом

```
@extends('example')

@section('title', 'Редактирование заказчика')

@section('body')
    <p>
        {!! $edit !!}
    </p>
@stop
```

Контроллер товаров сделан аналогично контроллеру поставщиков.

```
<?php

/*
 * Контроллер товаров
 */

namespace App\Http\Controllers;

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


class ProductController extends Controller {
    /**
     * Отображает список продуктов
     *
     * @return Response
     */
    public function showProducts() {
        // Подключаем виджет для поиска
        $filter = \DataFilter::source(new Product);
        // Поиск будет по наименованию продукта
        $filter->add('NAME', 'Наименование', 'text');
        $filter->submit('Поиск');
        $filter->reset('Сброс');

       // Создам сетку для отображения отфильтрованных данных
        $grid = \DataGrid::source($filter);

        // выводимые столбцы сетки
        // Поле, подпись, сортируемый
        $grid->add('NAME', 'Наименование', true);
       // задаём формат с 2 знаками после запятой
        $grid->add('PRICE|number_format[2,., ]', 'Стоимость');

        $grid->row(function($row) {
            // Денежные величины прижимаем вправо
            $row->cell('PRICE')->style("text-align: right");
        });
        // Добавляем кнопки для просмотра, редактирования и удаления записи
        $grid->edit('/product/edit', 'Редактирование', 'show|modify|delete');
       // Добавляем кнопку добавления заказчика
        $grid->link('/product/edit', "Добавление товара", "TR");
        // задаём сортировку
        $grid->orderBy('NAME', 'asc');
         // задаём количество записей на страницу
        $grid->paginate(10);
        // отображаем шаблон customer и передаём в него фильтр и грид
        return view('product', compact('filter', 'grid'));
    }

    /**
     * Добавление, редактирование и удаление продуктов
     *
     * @return Response
```

```
     */
    public function editProduct() {
        if (\Input::get('do_delete') == 1)
            return "not the first";
        // создаём редактор
        $edit = \DataEdit::source(new Product());
        // задаём подпись диалога в зависимости от типа операции
        switch ($edit->status) {
            case 'create':
                $edit->label('Добавление товара');
                break;
            case 'modify':
                $edit->label('Редактирование товара');
                break;
            case 'do_delete':
                $edit->label('Удаление товара');
                break;
            case 'show':
                $edit->label('Карточка товара');
                $edit->link('products', 'Назад', 'TR');
                break;
        }
        // задаём что после операций добавления, редактирования и удаления
        // возвращаемся к списку заказчиков
        $edit->back('insert|update|do_delete', 'products');
        // Добавляем редакторы определённого типа, задаём им подпись
        // и связываем их с атрибутами модели
        $edit->add('NAME', 'Наименование', 'text')->rule('required|max:100');
        $edit->add('PRICE', 'Стоимость', 'text')->rule('max:19');
        $edit->add('DESCRIPTION', 'Описание', 'textarea')
             ->attributes(['rows' => 8])
             ->rule('max:8192');
        // отображаем шаблон product_edit и передаём в него редактор
        return $edit->view('product_edit', compact('edit'));
    }
}
```

Контроллер счёт фактур является более сложным. В него добавлена дополнительная функция оплаты счёта. Оплаченные счёт фактуры подсвечиваются другим цветом. При просмотре счёт фактуры отображаются так же её позиции. Во время редактирования счёт фактуры есть возможность редактировать и её позиции. Далее я приведу текст контроллера с подробными комментариями.

```
<?php

/*
 * Контроллер счёт фактур
 */

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 {

    /**
     * Отображает список счёт-фактур
     *
     * @return Response
     */
    public function showInvoices() {
        // Модель счёт фактур будет одновременно
        // выбирать связанных поставщиков
        $invoices = Invoice::with('customer');
        // Подключаем виджет для поиска
        $filter = \DataFilter::source($invoices);
        // Позволяем фильтровать по диапазону дат
        $filter->add('INVOICE_DATE', 'Дата', 'daterange');
```

```
        // и фильтровать по имени заказчика
        $filter->add('customer.NAME', 'Заказчик', 'text');
        $filter->submit('Поиск');
        $filter->reset('Сброс');

        // Создам сетку для отображения отфильтрованных данных
        $grid = \DataGrid::source($filter);

        // выводимые столбцы сетки
        // Поле, подпись, сортируемый
        // для даты задаём дополнительную функцию, которая преобразует дату в строку
        $grid->add('INVOICE_DATE|strtotime|date[d.m.Y H:i:s]', 'Дата', true);
        // для денег задам формат с двумя знаками после запятой
        $grid->add('TOTAL_SALE|number_format[2,., ]', 'Сумма');
        $grid->add('customer.NAME', 'Заказчик');
        // Значение boolean отображаем как Да/Нет
        $grid->add('PAID', 'Оплачено')
             ->cell(function( $value, $row) {
                    return $value ? 'Да' : 'Нет';
                });
        // задаём функцию обработки каждой строки
        $grid->row(function($row) {
            // Денежные величины прижимаем вправо
            $row->cell('TOTAL_SALE')->style("text-align: right");
            // окрашиваем оплаченные накладные в другой цвет
            if ($row->cell('PAID')->value == 'Да') {
                $row->style("background-color: #ddffee;");
            }
        });

        // Добавляем кнопки для просмотра, редактирования и удаления записи
        $grid->edit('/invoice/edit', 'Редактирование', 'show|modify|delete');
        // Добавляем кнопку добавления счёт-фактуры
        $grid->link('/invoice/edit', "Добавление счёта", "TR");
        // задаём сортировку
        $grid->orderBy('INVOICE_DATE', 'desc');
        // задаём количество записей на страницу
        $grid->paginate(10);
        // отображаем шаблон customer и передаём в него фильтр и грид
        return view('invoice', compact('filter', 'grid'));
    }

    /**
     * Добавление, редактирование и удаление счет фактуры
     *
     * @return Response
     */
    public function editInvoice() {
        // Получаем текст сохранённой ошибки, если она была
        $error_msg = \Request::old('error_msg');
        // создаём редактор счёт фактуры
        $edit = \DataEdit::source(new Invoice());
        // если счёт оплачен, то генерируем ошибку при попытке его редактирования
        if (($edit->model->PAID) && ($edit->status === 'modify')) {
            $edit->status = 'show';
            $error_msg = 'Редактирование не возможно. Счёт уже оплачен.';
        }
        // если счёт оплачен, то генерируем ошибку при попытке его удаления
        if (($edit->model->PAID) && ($edit->status === 'delete')) {
            $edit->status = 'show';
            $error_msg = 'Удаление не возможно. Счёт уже оплачен.';
        }
        // задаём подпись диалога в зависимости от типа операции
        switch ($edit->status) {
            case 'create':
                $edit->label('Добавление счета');
                break;
            case 'modify':
                $edit->label('Редактирование счета');
                break;
            case 'do_delete':
                $edit->label('Удаление счета');
                break;
            case 'show':
                $edit->label('Счет');
                $edit->link('invoices', 'Назад', 'TR');
                 // Если счёт фактуры не оплачена, показываем кнопку оплатить
                if (!$edit->model->PAID)
                    $edit->link('invoice/pay/' . $edit->model->INVOICE_ID, 'Оплатить', 'BL');
                break;
```

```
        }

        // задаём что после операций добавления, редактирования и удаления
        // возвращаемся к списку счет фактур
        $edit->back('insert|update|do_delete', 'invoices');

        // Задаём для поля дата, что оно обязательное
        // По умолчанию ставится текущая дата
        $edit->add('INVOICE_DATE', 'Дата', 'datetime')
                ->rule('required')
                ->insertValue(date('Y-m-d H:i:s'));

        // Добавляем поле для ввода заказчика. При наборе имени заказчика
        // будет отображаться список подсказок
        $edit->add('customer.NAME', 'Заказчик', 'autocomplete')
                ->rule('required')
                ->options(Customer::lists('NAME', 'CUSTOMER_ID')->all());
        // добавляем поле, которое будет отображать сумму накладной, только для чтения
        $edit->add('TOTAL_SALE', 'Сумма', 'text')
                ->mode('readonly')
                ->insertValue('0.00');
        // Добавляем галочку Оплачено
        $paidCheckbox = $edit->add('PAID', 'Оплачено', 'checkbox')
                ->insertValue('0')
                ->mode('readonly');
        $paidCheckbox->checked_output = 'Да';
        $paidCheckbox->unchecked_output = 'Нет';

        // создаём грид для отображения строк счет фактуры
        $grid = $this->getInvoiceLineGrid($edit->model, $edit->status);
        // отображаем шаблон invoice_edit и передаём в него редактор и
                    // грид для отображения позиций
        return $edit->view('invoice_edit', compact('edit', 'grid', 'error_msg'));
    }

    /**
     * Оплата счёт фактуры
     *
     *  @return Response
     */
    public function payInvoice($id) {
        try {
            // находим счёт фактуру по идентификатору
            $invoice = Invoice::findOrFail($id);
            // вызываем процедуру оплаты
            $invoice->pay();
        } catch (\Illuminate\Database\QueryException $e) {
            // если произошла ошибка, то
            // выделяем текст исключения
            $pos = strpos($e->getMessage(), 'E_INVOICE_ALREADY_PAYED');
            if ($pos !== false) {
                // перенаправляем на страницу редактора и отображаем там ошибку
                return redirect('invoice/edit?show=' . $id)
                        ->withInput(['error_msg' => 'Счёт уже оплачен']);
            } else
                throw $e;
        }
        // перенаправляем на страницу редактора
        return redirect('invoice/edit?show=' . $id);
    }

    /**
     * Получение сетки для строк счета фактуры
     * @param \App\Invoice $invoice
     * @param string $mode
     * @return \DataGrid
     */
    private function getInvoiceLineGrid(Invoice $invoice, $mode) {
        // Получаем строки счёт фактуры
        // Для каждой позиции счёта будет инициализироваться связанный с ним продукт
        $lines = InvoiceLine::with('product')->where('INVOICE_ID', $invoice->INVOICE_ID);

        // Создам сетку для отображения позиций накладной
        $grid = \DataGrid::source($lines);
        // выводимые столбцы сетки
        // Поле, подпись, сортируемый
        $grid->add('product.NAME', 'Наименование');
        $grid->add('QUANTITY', 'Количество');
        $grid->add('SALE_PRICE|number_format[2,., ]', 'Стоимость')->style('min-width: 8em;');
        $grid->add('SUM_PRICE|number_format[2,., ]', 'Сумма')->style('min-width: 8em;');
```

```
        // задаём функцию обработки каждой строки
        $grid->row(function($row) {
            $row->cell('QUANTITY')->style("text-align: right");
            // Денежные величины приживаем вправо
            $row->cell('SALE_PRICE')->style("text-align: right");
            $row->cell('SUM_PRICE')->style("text-align: right");
        });

        if ($mode == 'modify') {
            // Добавляем кнопки для просмотра, редактирования и удаления записи
            $grid->edit('/invoice/editline', 'Редактирование', 'modify|delete');
            // Добавляем кнопку добавления заказчика
            $grid->link('/invoice/editline?invoice_id=' . $invoice->INVOICE_ID, "Добавление
позиции", "TR");
        }

        return $grid;
    }

    /**
     * Добавление, редактирование и удаление позиций счет фактуры
     *
     * @return Response
     */
    public function editInvoiceLine() {
        if (\Input::get('do_delete') == 1)
            return "not the first";

        $invoice_id = null;
        // создаём редактор позиции счёт фактуры
        $edit = \DataEdit::source(new InvoiceLine());
        // задаём подпись диалога в зависимости от типа операции
        switch ($edit->status) {
            case 'create':
                $edit->label('Добавление позиции');
                $invoice_id = \Input::get('invoice_id');
                break;
            case 'modify':
                $edit->label('Редактирование позиции');
                $invoice_id = $edit->model->INVOICE_ID;
                break;
            case 'delete':
                $invoice_id = $edit->model->INVOICE_ID;
                break;
            case 'do_delete':
                $edit->label('Удаление позиции');
                $invoice_id = $edit->model->INVOICE_ID;
                break;
        }
        // формируем url для возврата
        $base = str_replace(\Request::path(), '', strtok(\Request::fullUrl(), '?'));
        $back_url = $base . 'invoice/edit?modify=' . $invoice_id;
        // устанавливаем страницу для возврата
        $edit->back('insert|update|do_delete', $back_url);
        $edit->back_url = $back_url;
        // добавляем скрытое поле с кодом счёт фактуры
        $edit->add('INVOICE_ID', '', 'hidden')
             ->rule('required')
             ->insertValue($invoice_id)
             ->updateValue($invoice_id);
        // Добавляем поле для ввода товара. При наборе имени товара
        // будет отображаться список подсказок
        $edit->add('product.NAME', 'Наименование', 'autocomplete')
             ->rule('required')
             ->options(Product::lists('NAME', 'PRODUCT_ID')->all());
        // поле для ввода количества
        $edit->add('QUANTITY', 'Количество', 'text')
             ->rule('required');
        // отображаем шаблон invoice_line_edit и передаём в него редактор
        return $edit->view('invoice_line_edit', compact('edit'));
    }
}
```

Редактор счёт фактур имеет не стандартный для zofe/rapyd вид, поскольку нам необходимо выводить сетку с позициями счёт фактур. Для этого мы изменили шаблон invoice_edit следующим образом.

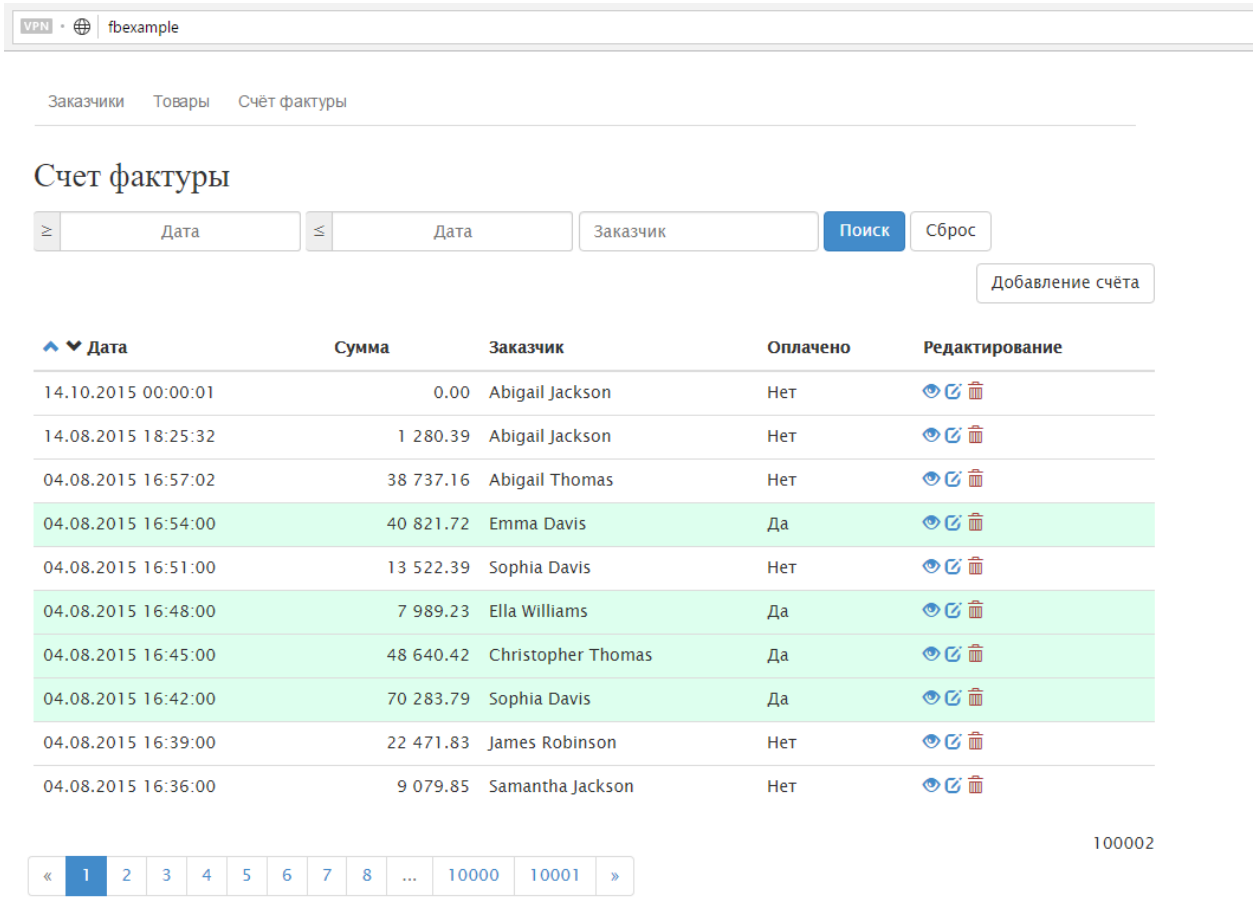
```
@extends('example')

@section('title','Редактирование счета')

@section('body')

    <div class="container">
        {!! $edit->header !!}

        @if($error_msg)
            <div class="alert alert-danger">
                <strong>Ошибка!</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
```

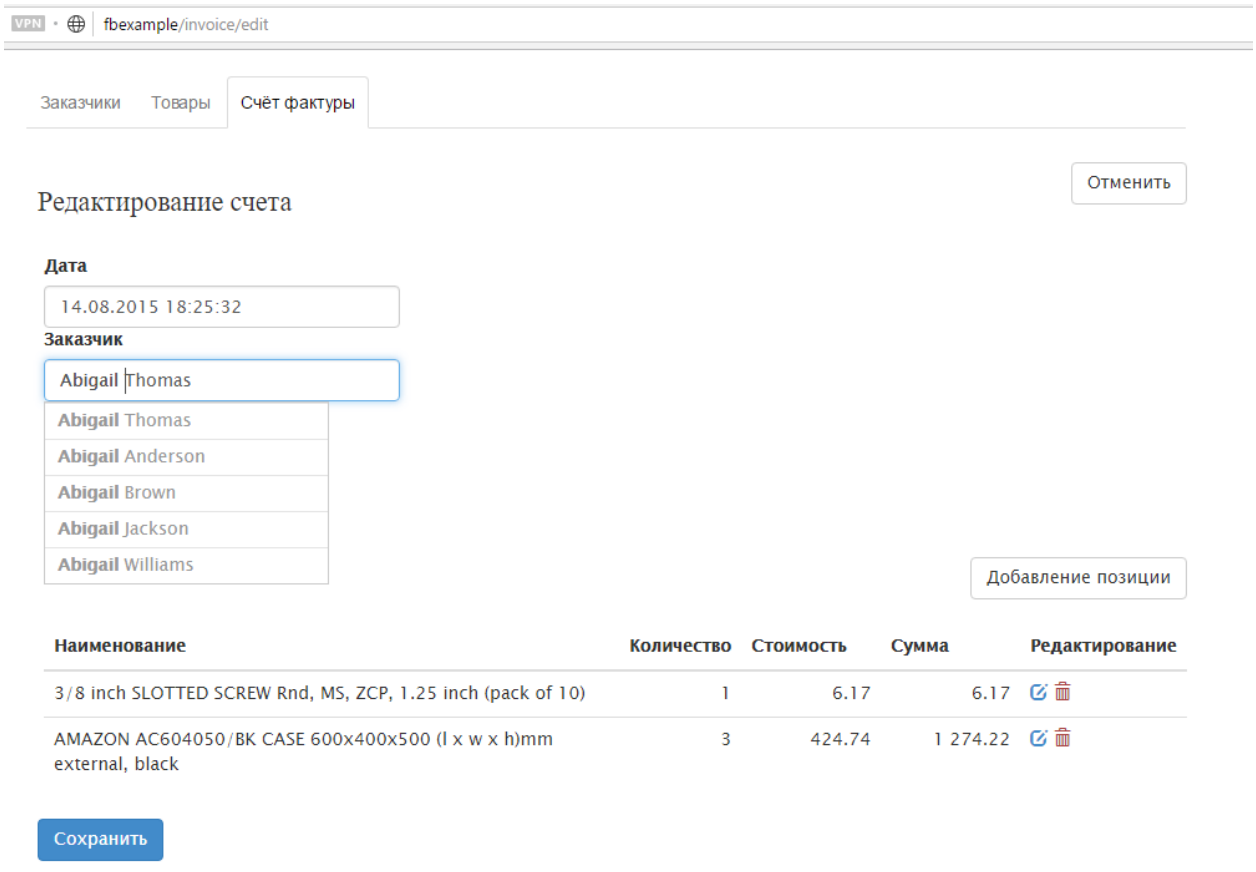
Теперь, когда все контроллеры написаны, изменим маршруты так, чтобы наш сайт на стартовой странице открывал список счёт фактур. Напоминаю, что маршруты настраиваются в файле app/Http/routes.php в Laravel 5.2 и routes/wep.php в Laravel 5.3.

```
// Корневой маршрут
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');
```

Здесь маршрут /invoice/pay/{id} выделяет идентификатор счёт фактуры из адреса и передаёт его в метод payInvoice. Остальные маршруты не требуют отдельного пояснения.

Напоследок приведу несколько скриншотов получившегося веб приложения.


VPN
fbexample
Заказчики
Товары
Счёт фактуры
Счет фактуры
≥
Дата
≤
Дата
Заказчик
Поиск
Сброс
Добавление счёта
Дата
Сумма
Заказчик
Оплачено
Редактирование
14.10.2015 00:00:01 0.00 Abigail Jackson Нет
14.08.2015 18:25:32 1 280.39 Abigail Jackson Нет
04.08.2015 16:57:02 38 737.16 Abigail Thomas Нет
04.08.2015 16:54:00 40 821.72 Emma Davis Да
04.08.2015 16:51:00 13 522.39 Sophia Davis Нет
04.08.2015 16:48:00 7 989.23 Ella Williams Да
04.08.2015 16:45:00 48 640.42 Christopher Thomas Да
04.08.2015 16:42:00 70 283.79 Sophia Davis Да
04.08.2015 16:39:00 22 471.83 James Robinson Нет
04.08.2015 16:36:00 9 079.85 Samantha Jackson Нет
100002
« 1 2 3 4 5 6 7 8 ... 10000 10001 »


VPN
fbexample/invoice/edit
Заказчики
Товары
Счёт фактуры
Отменить
Редактирование счета
Дата
14.08.2015 18:25:32
Заказчик
Abigail Thomas
Abigail Thomas
Abigail Anderson
Abigail Brown
Abigail Jackson
Abigail Williams
Добавление позиции
Наименование
Количество
Стоимость
Сумма
Редактирование
3/8 inch SLOTTED SCREW Rnd, MS, ZCP, 1.25 inch (pack of 10) 1 6.17 6.17
AMAZON AC604050/BK CASE 600x400x500 (l x w x h)mm external, black 3 424.74 1 274.22
Сохранить

На этом мой пример закончен. Исходные коды вы можете скачать по ссылке https://github.com/sim1984/phpfbexample.