GeekBrains

Laravel

Урок 5. Работа с базой данных. Запросы. Миграции

Структура базы данных, соединение с БД, создание и наполнение таблиц. Миграции.

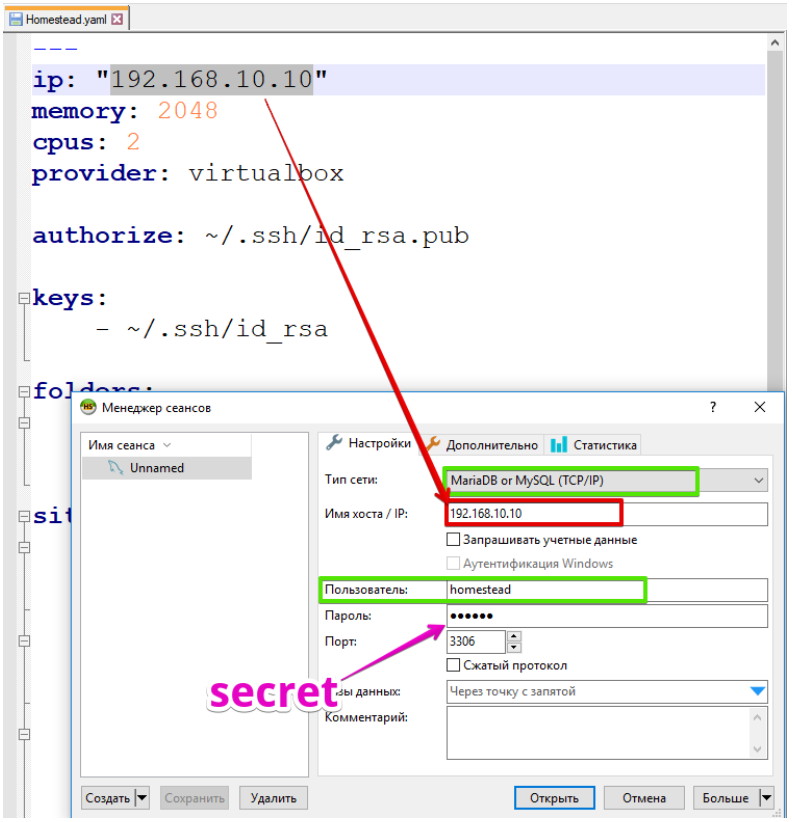
Оглавление

Теория

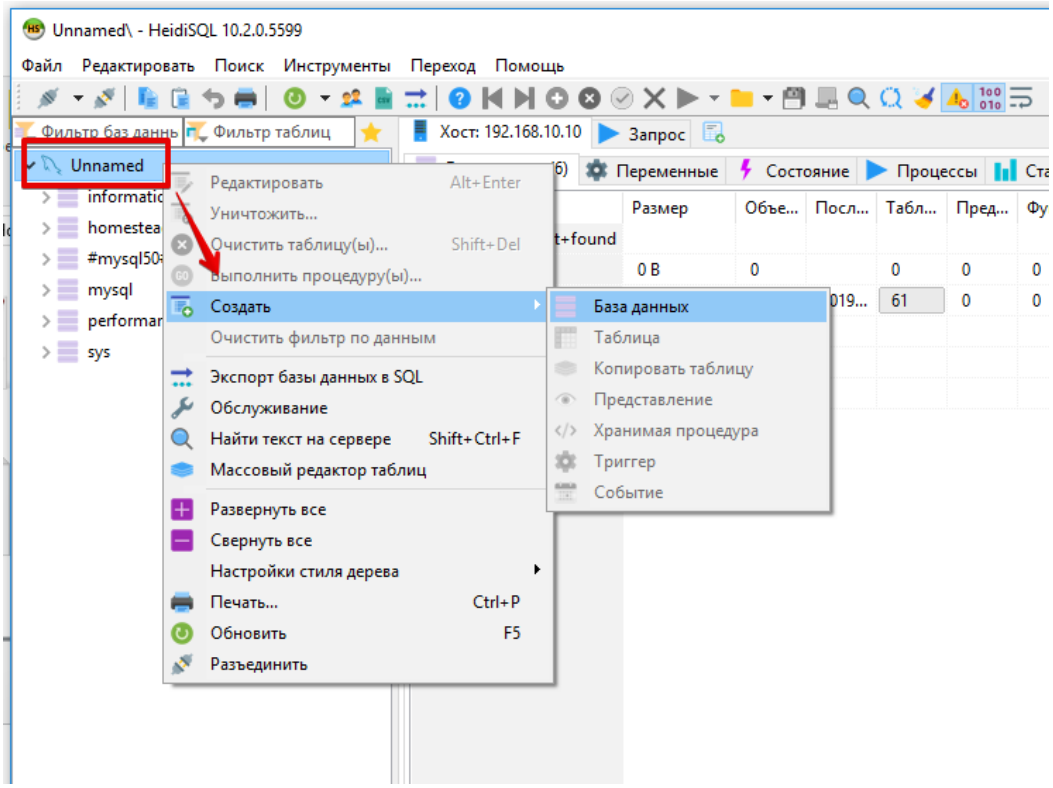
Установка окружения

Для работы с базой данных потребуется визуальный интерфейс HeidiSQL. Скачать эту программу можно с сайта https://www.heidisql.com/download.php

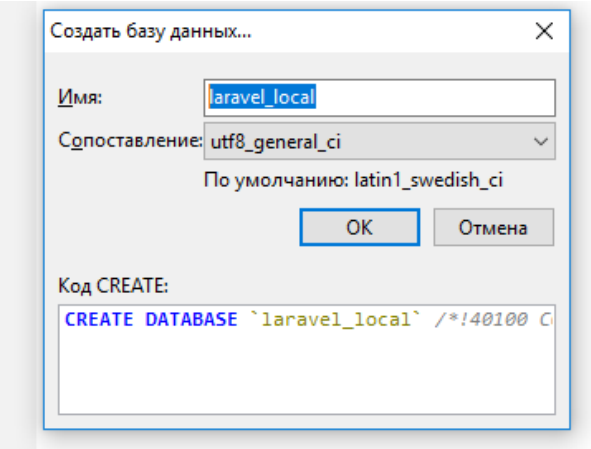
Установим и откроем программу. Выберем кнопку «Создать». Тип соединения — MySQL. IP-адрес укажем тот, что записан в настройках нашей виртуальной машины — **homestead**. Имя пользователя по умолчанию — **homestead**, введем его. Пароль — **secret**. Далее нажмем «Открыть».



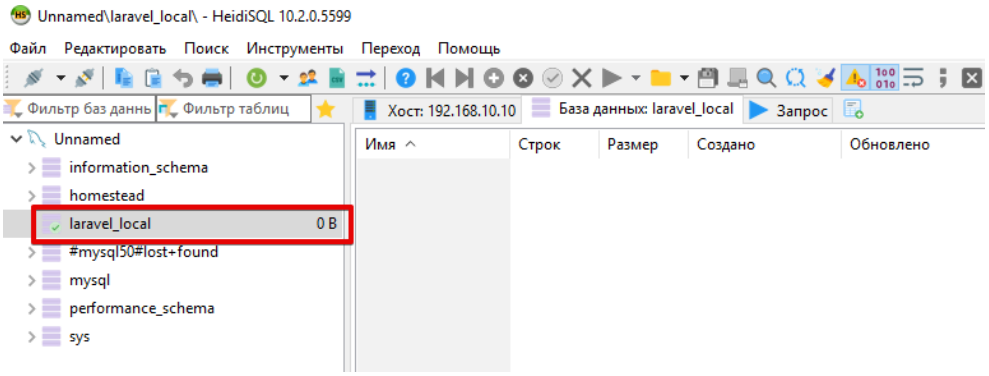
Теперь создаем базу данных. Для этого щелкаем правой кнопкой мыши по нашему соединению и выбираем «Создать» — «База данных».



В открывшемся окне вводим имя базы данных **laravel_local**. В поле «Сопоставление» выберем кодировку **utf8_general_ci**. Данные действия создадут команды SQL. Выполним их, нажав OK.



Таким образом мы создали новую базу данных на нашем сервере.

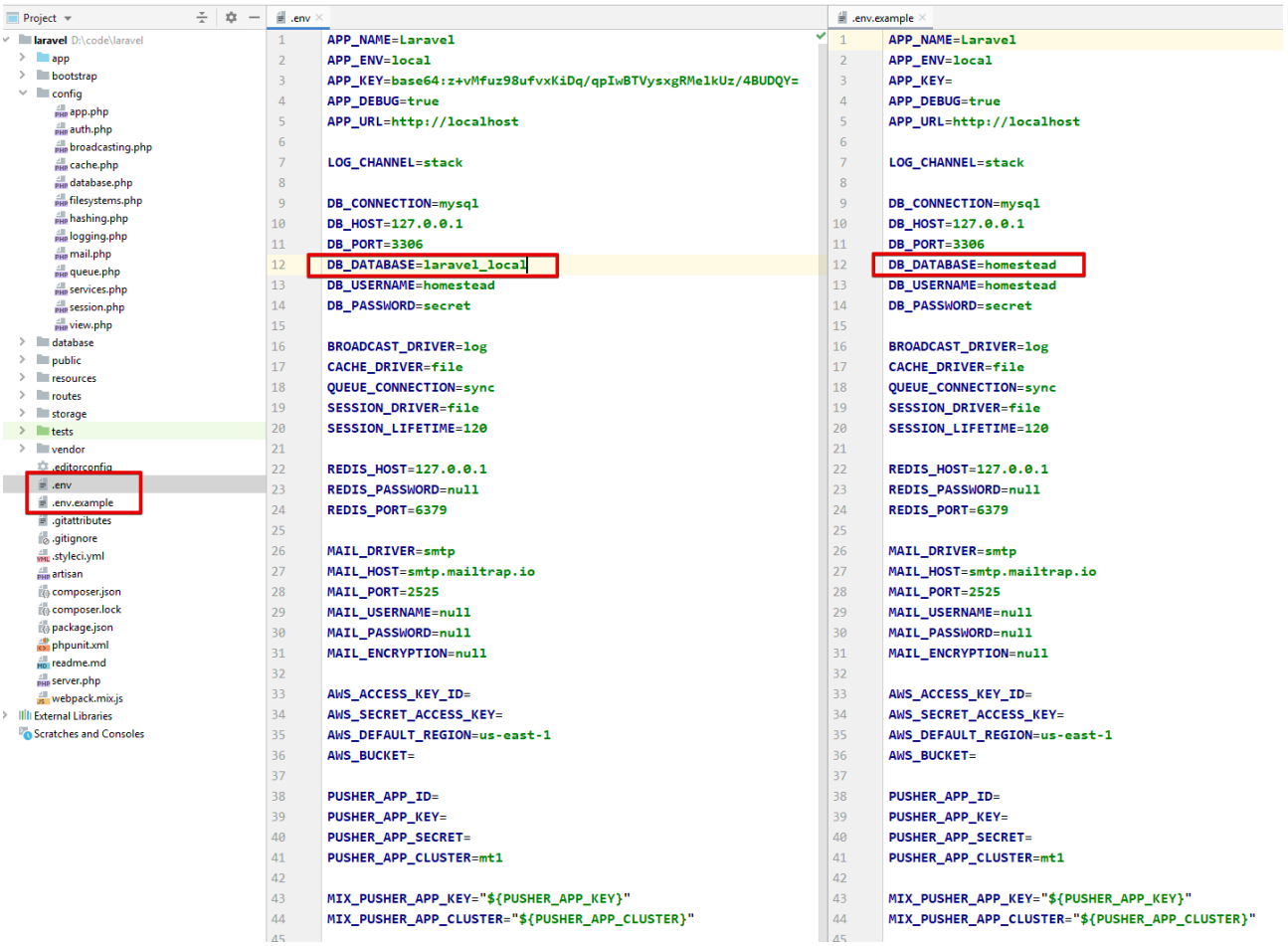


.env-файл

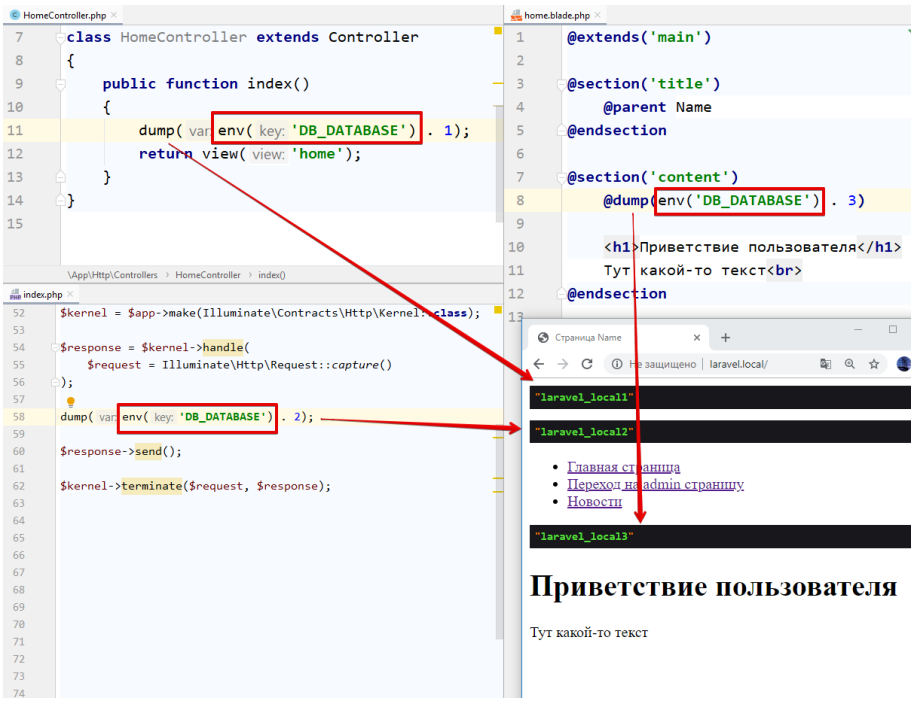
Теперь, когда мы создали базу данных, самое время поговорить о переменных окружения Laravel. Они хранятся в файле **.env** в корневом каталоге Laravel. Но если мы зайдем в файл **.gitignore** в этом же каталоге, то обнаружим, что **.env** — игнорируемый файл и не будет записан в репозиторий.

Это нужно для безопасности. В репозитории не должны храниться эти данные. Однако мы должны указать, какие переменные будут использоваться в нашем приложении. Для этого в корневом каталоге Laravel используется файл **.env.example**.

Добавим в **.env** информацию о созданной базе данных. Файл **.env.example** оставим без изменений.

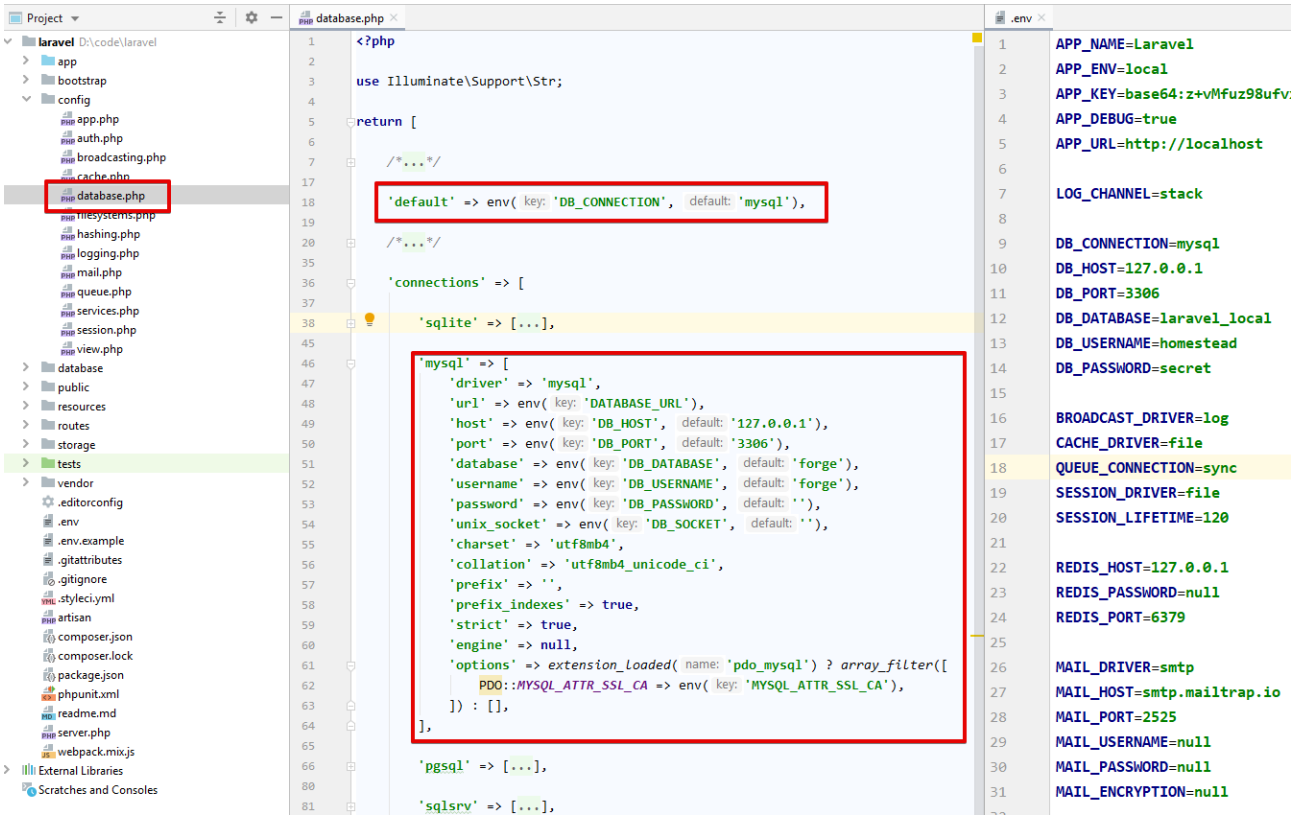


Как получить доступ к переменным окружения? Для этого в Laravel используется специальный хелпер **env**, который доступен в любой части приложения. Он принимает несколько параметров. Первый (обязательный) — имя переменной окружения. Второй — значение по умолчанию, если переменная не определена.



Конфигурация базы данных

Для конфигурации базы данных используется файл **database.php**, находящийся в каталоге **config**. Открыв этот файл, увидим значение по умолчанию используемой базы данных — **mysql**. Ниже есть настройки, содержащиеся в массиве **connections**. Данный массив собирает значение из файла **.env** при помощи хелпера **env**.

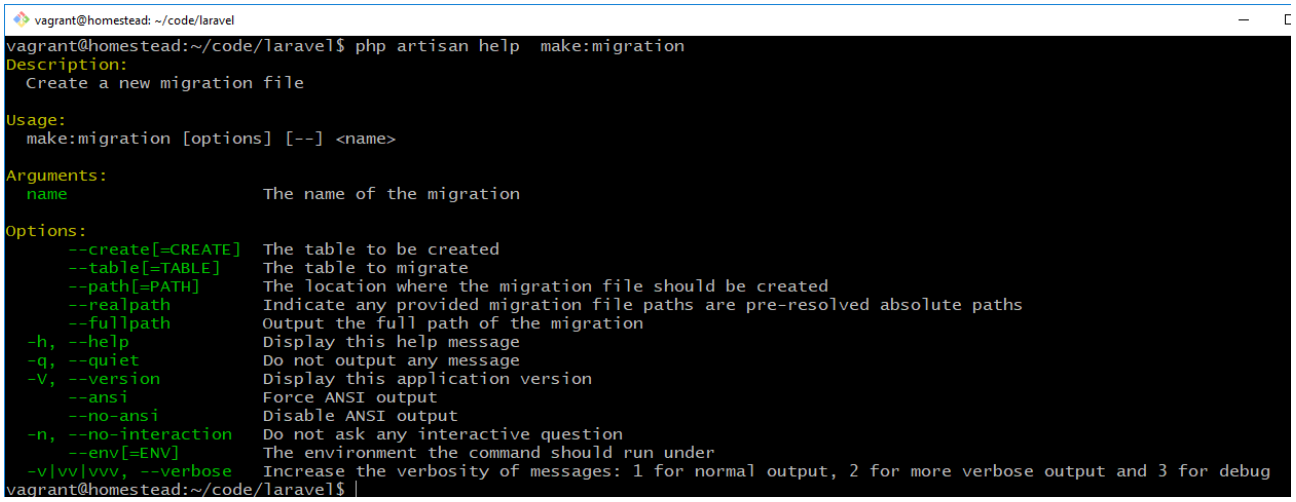


Практика

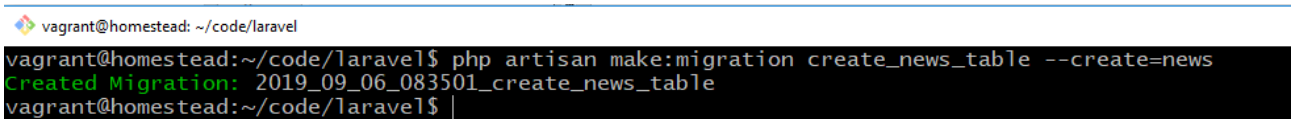
Создание миграций

У приложения есть жизненный цикл, во время которого производятся дополнения и правки функциональности. Поэтому в приложении часто требуется изменение базы данных. Прямое изменение — не лучший вариант, поскольку оно не исключает человеческий фактор: есть риск ошибки. Поэтому чаще всего используются миграции, чтобы гарантировать одинаковые изменения на тестовом сервере и на боевом, который используется в продакшене. Еще один плюс миграции: как правило, их можно отменить к предыдущей версии базы. Это удобно при обнаружении ошибок, не выявленных на этапе разработки и тестирования.

Создавать миграции мы будем при помощи консоли. Сначала проверим список опций команды по созданию миграции. Для этого в консоли вводим **php artisan help make:migration**:



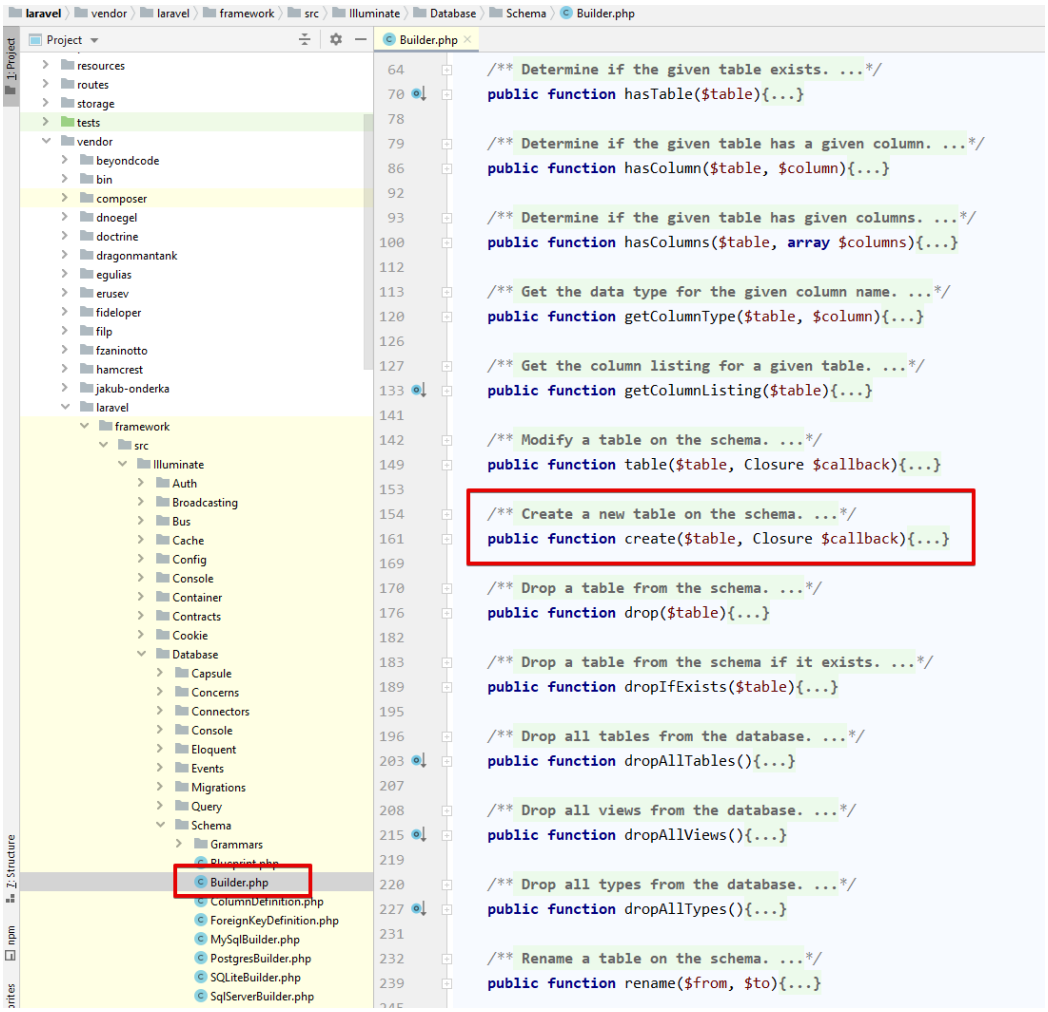
Создадим новую таблицу для сохранения новостей. Для этого введем команду: **php artisan make:migration create_news_table --create=news**:



Создался новый файл в директории **~/database/migrations**. Открыв этот файл, увидим два метода: **up**, который будет вызываться во время наката (выполнения) миграций, и **down**, который будет выполняться, если придется откатить миграцию.

Стоит отметить, что создание файла миграций не означает его выполнение. Как выполнять миграции, посмотрим позже.

Теперь подробнее рассмотрим, что содержится в методе **up**. Видим фасад **Schema** — обертку для класса **\Illuminate\Database\Schema\Builder**. В этом методе происходит обращение к **create** — статическому методу фасада. А фактически выполняется метод у объекта класса **\Illuminate\Database\Schema\Builder**. Посмотрим внимательнее на этот и другие методы объекта данного класса.



Первым параметром метода **create** передается строка — название будущей таблицы. Вторым — анонимная функция. Она в качестве параметра принимает объект класса **Illuminate\Database\Schema\Blueprint**, который будет подставлен в переменную **$table** при помощи **Dependency Injection(DI)**. Благодаря методам этого объекта мы и создаем нужную структуру таблицы внутри самой функции,.

У этого класса много методов, подробнее о них: https://laravel.com/docs/5.8/migrations#columns.

Разберем некоторые из них:

- **bigIncrements** — создает столбец первичного ключа с автоинкрементом. Принимает в качестве параметра строку, которая и служит названием столбца;
- **timestamps** — добавляет в таблицу два столбца: **created_at** и **updated_at**;
- **string** — добавляет в таблицу столбец с типом данных **VARCHAR**. Первым параметром принимает строку, которая будет именем столбца. Второй параметр необязательный — длина строки;
- **text** — добавляет в таблицу столбец с типом данных **TEXT**. Первым параметром принимает строку, которая будет именем столбца;
- **json** — добавляет в таблицу столбец с типом данных **JSON**. Первым параметром принимает строку, которая будет именем столбца;

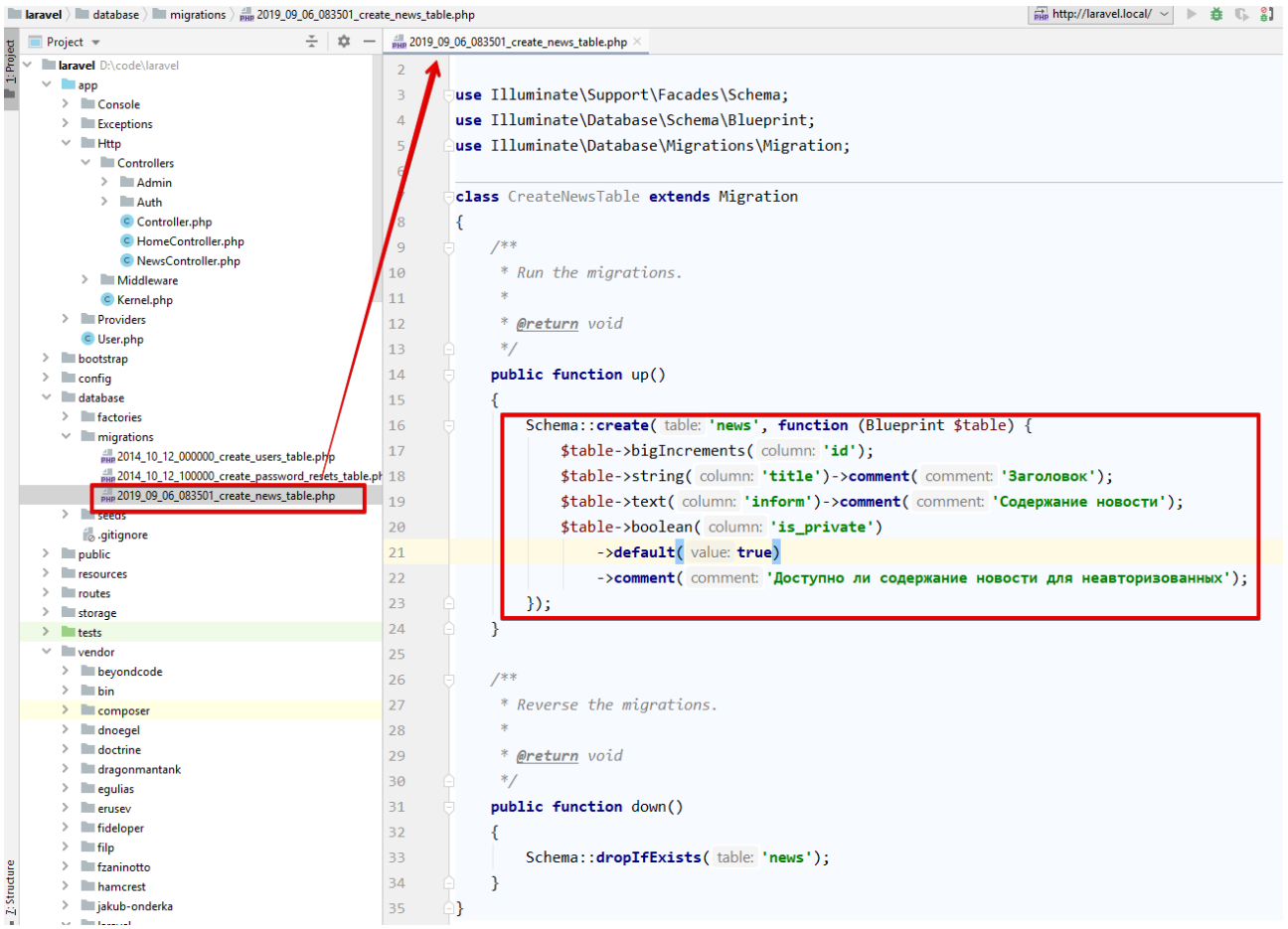
- **bigInteger** — добавляет в таблицу столбец с типом данных **BIGINT**. Первым параметром принимает строку, которая будет именем столбца;
- **dateTime** — добавляет в таблицу столбец с типом данных **DATETIME**. Первым параметром принимает строку, которая будет именем столбца;
- **boolean** — добавляет в таблицу столбец с типом данных **BOOLEAN**. Первым параметром принимает строку, которая будет именем столбца.

Помимо основных методов, которые определяют тип данных столбца в таблице, есть модификаторы колонок, которые позволяют установить дополнительные параметры столбца:

- **default** — метод указывает на то значение, которое будет установлено для колонки по умолчанию;
- **nullable** — метод принимает **true** || **false** в качестве параметра и указывает, может ли быть значение в данной колонке **null**.
- **comment** — принимает в качестве параметра строку, которая станет комментарием для указанного столбца.

Данные модификаторы вызываются после определения типа столбца.

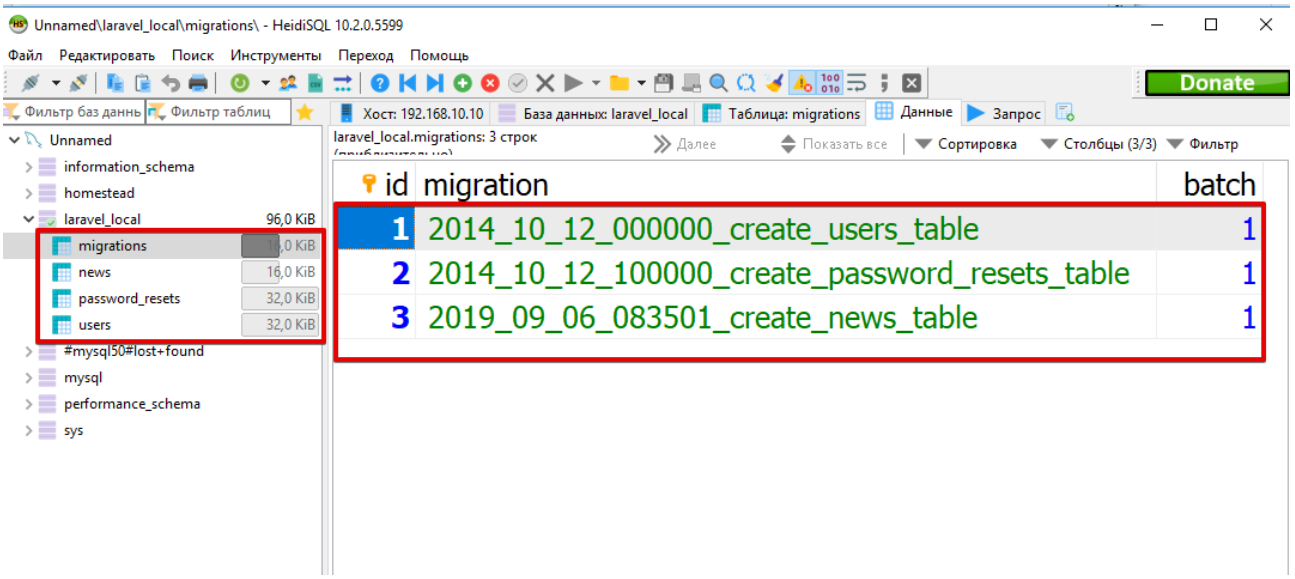
Добавим в метод **up** — код, который создаст нашу таблицу:



Для выполнения миграций выполним команду **php artisan migrate**:

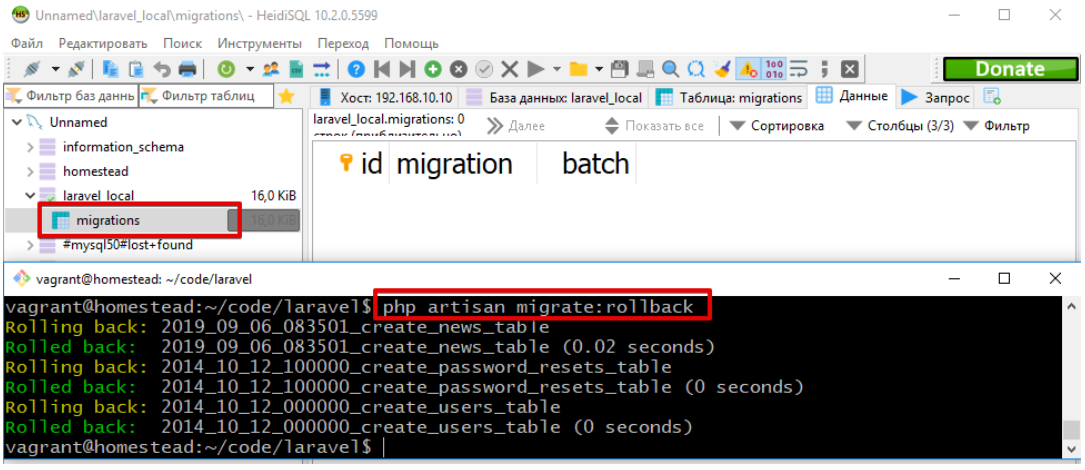
```
vagrant@homestead: ~/code/laravel
vagrant@homestead:~/code/laravel$ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (0.08 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (0.11 seconds)
Migrating: 2019_09_06_083501_create_news_table
Migrated:  2019_09_06_083501_create_news_table (0.06 seconds)
vagrant@homestead:~/code/laravel$
```

Обратим внимание: мы выполнили все миграций, которые были в папке **migrations**. Перейдем в программу HeidiSQL и посмотрим, какие таблицы были созданы.

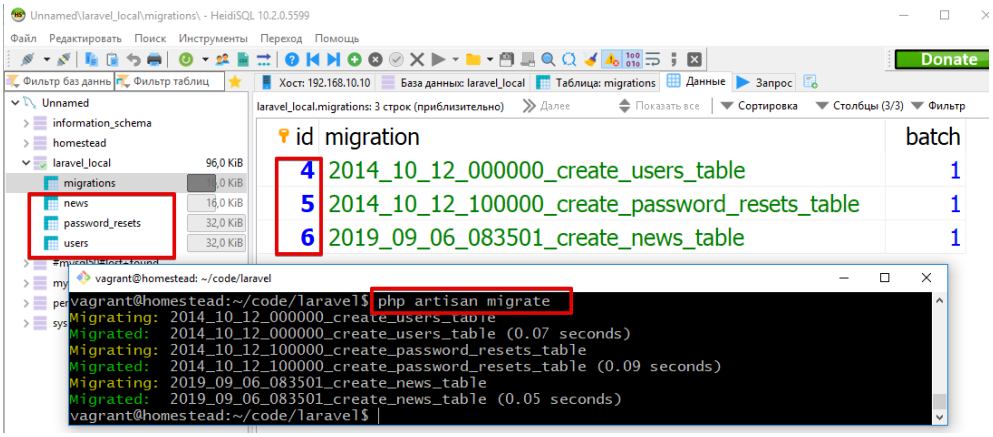


В нашей базе данных появилось четыре таблицы. Рассмотрим подробнее таблицу **migrations**. В этой таблице хранится информация о тех миграциях, которые были выполнены. Глядя на данные в ней, можем заметить, что сначала была создана таблицы **users**, затем таблица **password_resets** и после этого наша таблица для хранения новостей.

Если запустим миграцию еще раз предыдущей командой, то миграции, которые уже были выполнены, выполняться не будут. Но мы можем откатить эти миграции, выполнив команду **php artisan migrate:rollback**:



Данная команда откатит все миграции, которые были выполнены в данном предложении. На скриншоте видно, что все таблицы были удалены, но таблица **migrations** осталась. Повторим миграции.



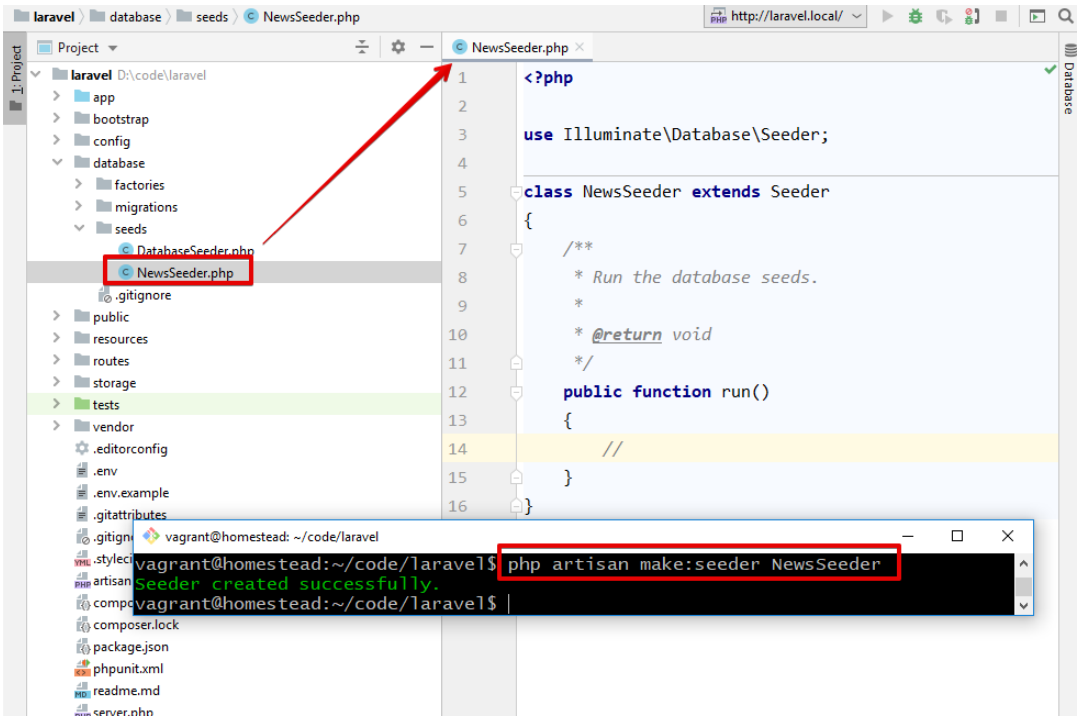
В том случае, если нам нужно откатить только последнюю миграцию или две последних миграции, то можем воспользоваться командой: `php artisan migrate:rollback --step=2`. Количество шагов в данной команде указывает какое количество миграций следует откатить.

Seeding — посев данных

Иногда для тестирования приложения необходимо какие-то входные данные. Их можно занести вручную, но это не совсем удобно. Удобнее если эти данные добавляются автоматически. Laravel предлагает для этой задачи решение, которое называется seeder. Их принцип действия похож на работу миграций — разработчик добавляет в проект специальный класс, в котором хранится логика для добавления данных.

У нас сейчас создана таблица News. Запомним ее данными, и для этого создадим новый seeder, выполнив команду: `php artisan make:seeder NewsSeeder`.

После выполнения данной команды в папке ~database/seeds будет создан новый файл NewsSeeder.php — файл одноименного класса, в котором содержится единственный метод run, который будет выполняться при запуске seeder.



Для заполнения базы данных фейковой информацией воспользуемся библиотекой **fzaninotto/Faker**. Предварительно установим ее через **composer** — команда для установки библиотеки: **composer require fzaninotto/faker**.

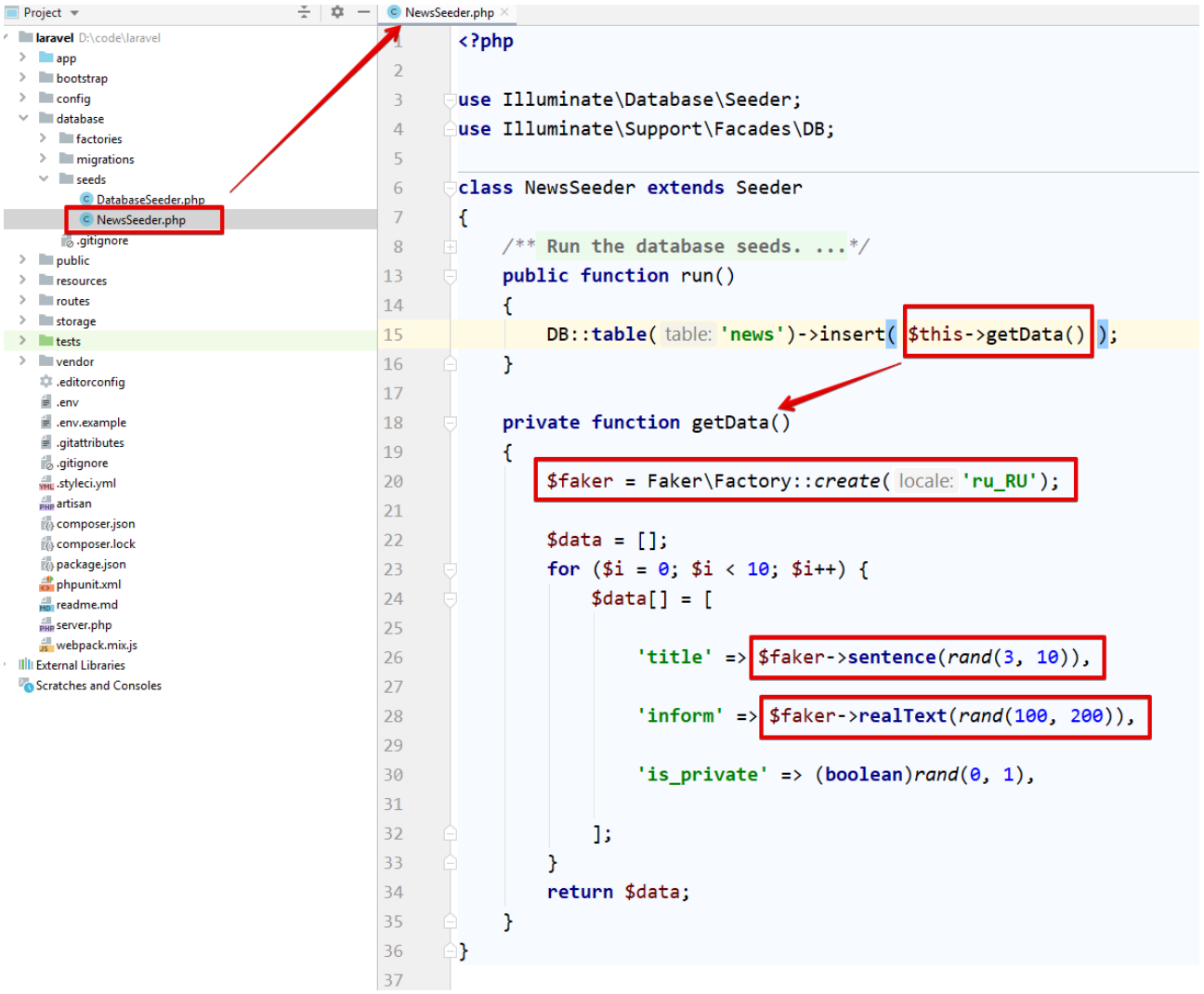
Подробная информация по данной библиотеке: https://github.com/fzaninotto/Faker

Добавим в **seeder** немного кода. На скриншоте ниже в методе **run** обращаемся к фасаду **DB**, который обратится к объекту класса **\Illuminate\Database\Query\Builder**. Таким образом мы получим возможность добавлять данные в указанную таблицу **news**. У полученного объекта вызываем метод **insert**, который и будет производить данную вставку. Метод **insert** ожидает массив данных, в котором ключи будут соответствовать полям таблицы.

Для генерации данных создаем отдельный метод **getData** и в нем обращаемся к библиотеке, которую установили. Обращаемся к статическому методу **create** и указываем локализацию. После этого набираем массив из 10 подмассивов. Каждый подмассив — отдельная строка в базе данных.

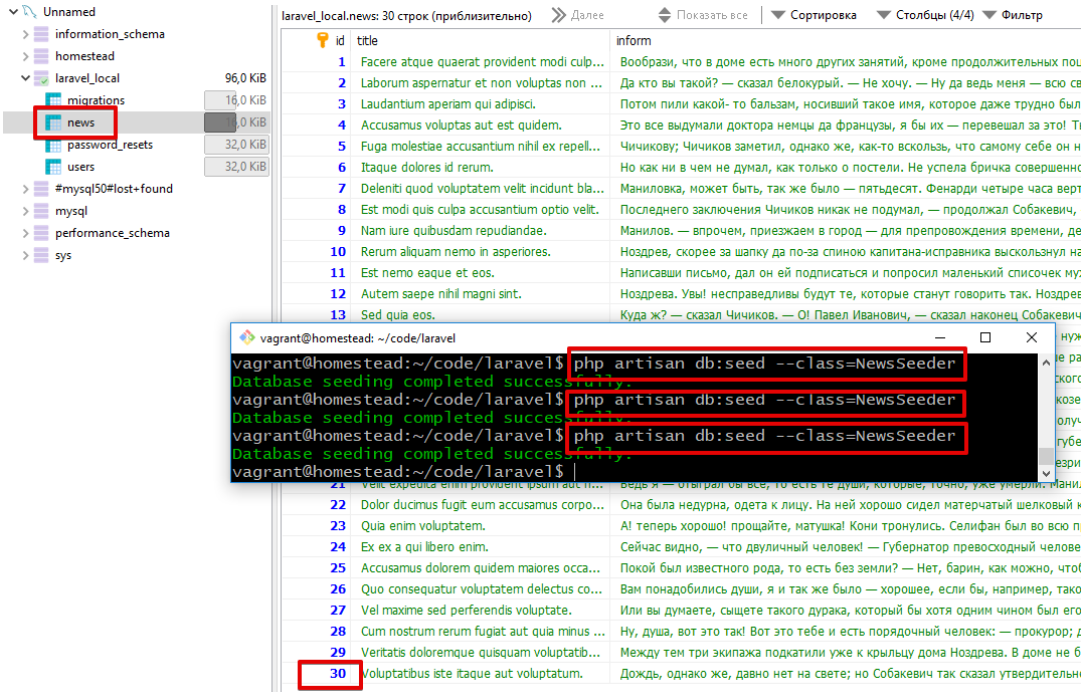
Для каждого подмассива собираем фейковые данные. Для этого используем объект, сохраненный в переменной **$faker**, и вызываем соответствующие метод:

- **sentence** — метод класса **Faker\Generator**, возвращающий предложение, состоящее из указанного количества строк. В данной библиотеке нет реализации данного метода на русском языке, поэтому результат на английском;
- **realText** — возвращает указанное количество слов из реального текста, минимум 10 и максимум 200.



Для запуска посева необходимо в консоли выполнить команду **php artisan db:seed --class=NewsSeeder**.

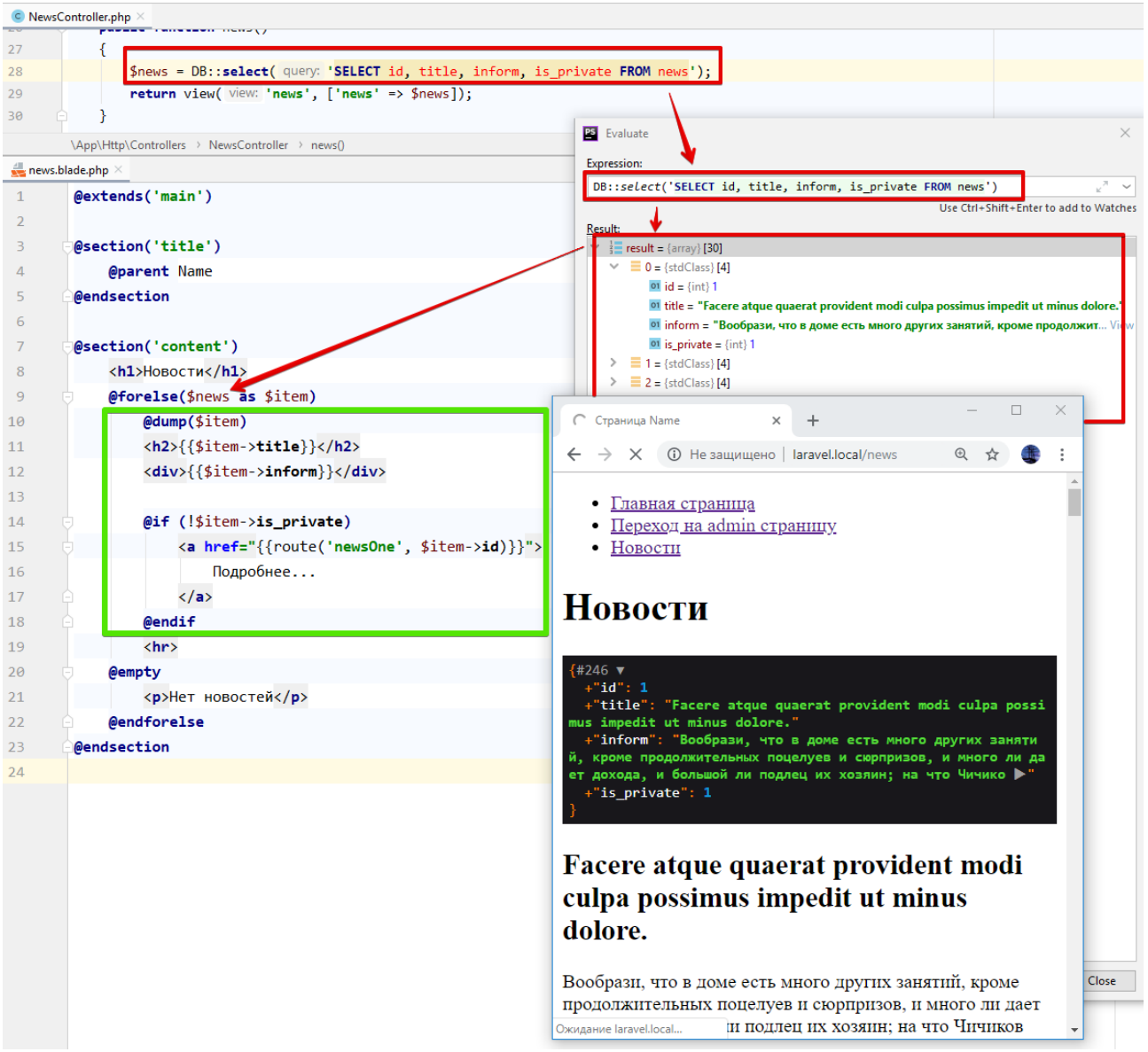
Seeders можно запускать любое количество раз.



Взаимодействие с базой данных

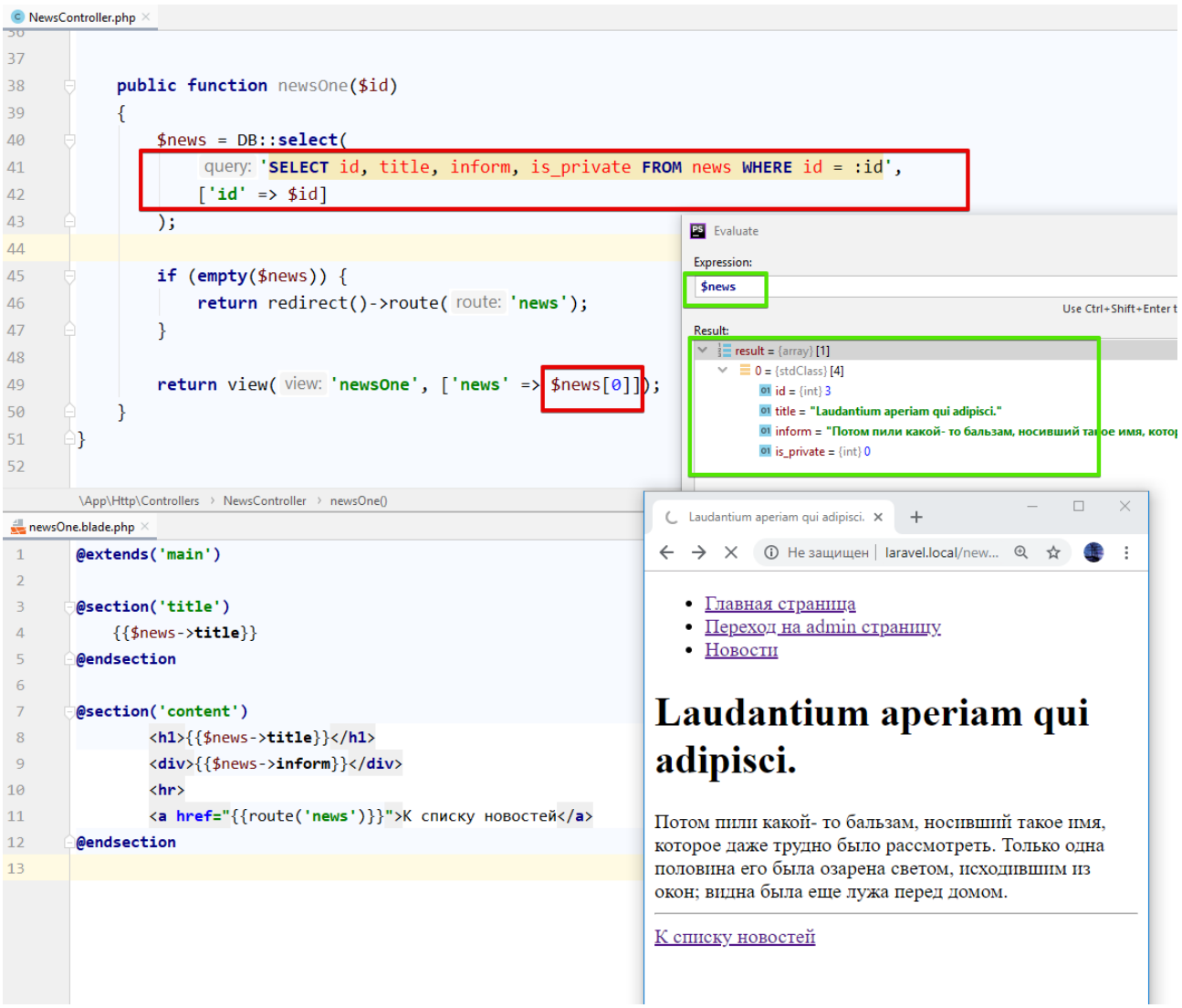
Теперь, когда у нас есть база данных, а в ней — необходимые нам данные, реализуем получение этих данных, их вставку и удаление.

Для взаимодействия с базой данных воспользуемся фасадом **DB** и его методами. Первый метод, который мы применим, **select**. Он позволяет использовать обычные SQL-запросы с возможностью экранировать данные, полученные от пользователя.



Как видно на скриншоте, в **NewsController** используется выборка из базы данных. Статический метод **Select** принимает SQL-запрос, в результате которого получаем массив **std**-классов. **stdClass** в PHP является предопределенным и часто используется для заполнения динамическими свойствами. В нашем случае каждый класс в полученном массиве содержит поля в соответствии с указанными в SQL-запросе. Учитывая это, следует произвести изменения в шаблоне **news.blade.php**. Данные изменения также показаны на скриншоте выше.

Теперь в методе **newsOne** в **NewsController** внесем изменения, чтобы получить конкретную запись из базы данных, а также создадим новый шаблон для вывода этой информации.



На скриншоте видим, что в метод **Select** передаем теперь два параметра. Первым передается SQL. Обратим внимание на то, что в данном запросе используется **:id**. Логично было бы подставить в эту строку переменную **$id**, которая и содержит идентификатор нужной записи. Но надо помнить, что эта переменная заполняется из данных, переданных пользователем. А значит, данные не безопасны — они могут содержать SQL-инъекцию. Подробнее о ней можно прочитать в статье https://habr.com/ru/post/148151/. Если в двух словах, это атака на базу данных.

Для защиты SQL-инъекций в Laravel используются подготовленные запросы. Сначала базе говорится, какой запрос будет происходить, и указывается, куда в запросе нужно подставить данные. Затем передаются данные и выполняется запрос. Данные подставляются в строго отведенные им места, что исключает атаку.

Вторым параметром в **Select** передаем массив, ключом в котором указываем названия плейсхолдеров. В значении передаем данные для подстановки. В примере выше в SQL-запросе используется один плейсхолдер **:id**, поэтому в массиве содержится одна пара «ключ — значение»: **'id' => $id**.

Помимо статичного метода для получения данных из базы, у фасада **DB** есть и другие:

- **selectOne(string $query, array $bindings = [])** — метод для поиска одной записи;

- **select(string $query, array $bindings = [])** — метод для выборки более чем одной записи;
- **insert(string $query, array $bindings = [])** — метод для вставки;
- **update(string $query, array $bindings = [])** — метод для изменения;
- **delete(string $query, array $bindings = [])** — метод для удаления;
- **statement(string $query, array $bindings = [])** — метод для запросов, не возвращающих результат.

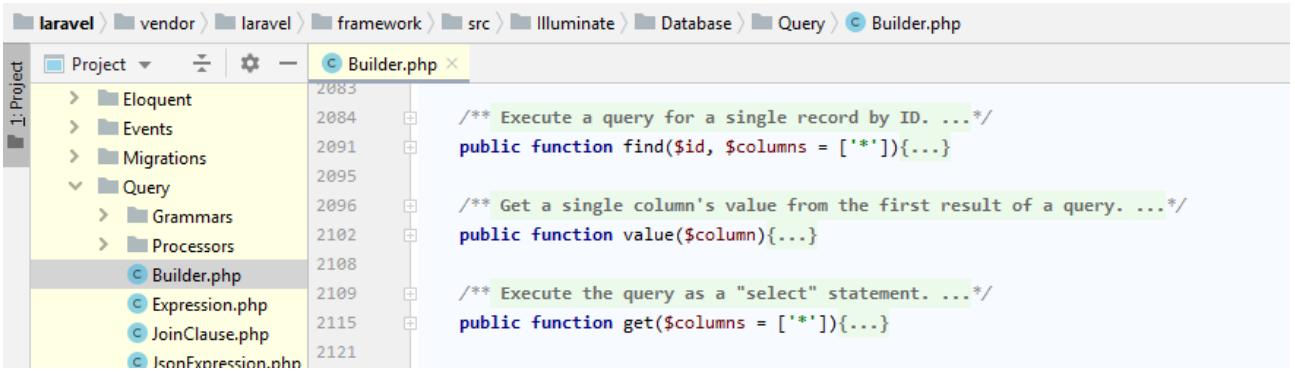
Конструктор запросов

В практике работы с базой данных использование подхода, описанного выше, допустимо. Но он используется реже, чем конструктор запросов. Рассмотрим следующий пример:

NewsController.php

```
31      public function news()
32      {
33          $news = DB::table( table: 'news')->get();
34          return view( view: 'news', ['news' => $news]);
35      }
36
37      public function newsOne($id)
38      {
39          $news = DB::table( table: 'news')->find($id);
40
41          if (empty($news)) {
42              return redirect()->route( route: 'news');
43          }
44
45          return view( view: 'newsOne', ['news' => $news]);
46      }
47  }
```

Здесь используется фасад DB, у которого вызывается метод **table**. Он принимает в качестве параметра название таблицы, к которой будет выполнен запрос. В данном случае обращаемся к таблице **news**. В результате метод **table** возвращает объект класса **Illuminate\Database\Query\Builder**. Он содержит большое количество методов, которые используются для создания запросов к базе данных.



В примере применяются методы **get** и **find**. Они позволяют выполнить SQL-запросы, которые мы писали ранее. Рассмотрим и другие примеры:

- агрегатные функции: **count**, **max**, **min**, **avg** и **sum**. Пример использования:
    - `DB::table('news')->count();`
    - `DB::table('news')->max('id');`
- объединение таблиц:
    - ```
      $users = DB::table('users')
          ->join('contacts', 'users.id', '=', 'contacts.user_id')
          ->join('orders', 'users.id', '=', 'orders.user_id')
          ->select('users.*', 'contacts.phone', 'orders.price')
          ->get();
      ```
    - ```
      $users = DB::table('users')
          ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
          ->get();
      ```
- условия: **where**, **orWhere**, **whereBetween**, **whereNotBetween**, **wherein**, **whereNotIn**, **whereNull**, **whereNotNull**, **whereDate**, **whereMonth**, **whereDay**, **whereYear**:
    - `$users = DB::table('users')->where('votes', '=', 100)->get();`
    - ```
      $users = DB::table('users')->where([
          ['status', '=', '1'],
          ['subscribed', '<>', '1'],
      ])->get();
      ```
    - ```
      $users = DB::table('users')
          ->where('votes', '>', 100)
          ->orWhere('name', 'John')
          ->get();
      ```
    - ```
      $users = DB::table('users')
          ->whereIn('id', [1, 2, 3])
          ->get();
      ```

Практическое задание

1. Продумать структуру сохранения данных следующих таблиц:
    a. Новости.
    b. Категории новостей.
      ```

c. Источники получения новостных данных.

Структура может быть реализована в виде рисунка, таблиц или специальных диаграмм.

2. Сделать миграции для таблиц из первого задания. Миграции должны иметь ролбеки.
3. С помощью сидинга реализовать заполнение созданных таблиц фейковыми данными. Таблица категорий — минимум 5 записей. Таблица новостей — минимум 10 записей на каждую категорию. Таблица источников — минимум 10 записей.
4. Удалить созданные методы в контроллере для хранения данных и реализовать вывод данных из базы.

Дополнительные материалы

1. https://laravel.com/docs/5.8/database.
2. https://laravel.com/docs/5.8/queries.
3. https://laravel.com/docs/5.8/migrations.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://laravel.com/docs/5.8/.
2. http://laravel.su/.