GeekBrains

Laravel

Урок 9. События в Laravel. Провайдеры. Работа со сторонними API

Использование провайдеров в Laravel. Добавление провайдера для работы с API VK. Авторизация пользователя через ВКонтакте.

Оглавление

Теория

Паттерн «Наблюдатель»

Паттерн — это общепринятый подход в решении задач программирования. Знание паттернов позволяет программисту избегать костылей и велосипедов, допускать меньше ошибок и писать поддерживаемый, красивый и чистый код.

Паттерны условно можно разделить на **порождающие**, **структурные** и **поведенческие**.

«Наблюдатель» — это поведенческий паттерн проектирования, который создает механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах. Этот паттерн позволяет, не меняя структуру одного конкретного класса, выполнять действия в других классах, которые «подписаны на события». Привязка новых классов-подписчиков и их отвязка не затрагивает класс, за которым они наблюдают.

Подробнее об этом и многих других паттернах: https://refactoring.guru/ru/design-patterns/observer

XML-парсеры

XML — формат передачи и хранения данных. По сути он является аналогом HTML: его задача — выполнять разметку документа. В отличие от HTML, он является строгим языком разметки. Если, создавая html-страницу, разработчик не укажет корневой контейнер HTML или где-то не закроет открывающийся тег — документ все равно будет нормально разбираться браузером, и пользователь увидит материал. С XML такие фокусы не проходят. Документ XML с нарушением любого правила его составления будет невалидным.

Документы XML часто используются для хранения информации и ее передачи по сети. Самый часто встречающийся случай — сохранение документов MS Office. Если, к примеру, открыть документ Word, как архив, то внутри увидим набор xml-документов.

Для создания и чтения xml-документов в PHP есть встроенные расширения, такие как SimpleXML и DOM. Ссылки на данные расширения:

- https://www.php.net/manual/ru/book.dom.php
- https://www.php.net/manual/ru/book.simplexml.php

Данные расширения, как правило, работают «из коробки» PHP. Но можно столкнуться с нюансами, которые не видны на старте.

Как быть?

Нужно взять за правило — прежде чем приступать к задаче, всегда думать, не решал ли ее кто-то прежде. В нашем примере таким решением будет библиотека **orchestral/parser**. Документация — по адресу https://github.com/orchestral/parser

Данная библиотека позволяет получить данные из xml-документа в виде массива.

Сервис-провайдеры

Назначение сервис-провайдеров (провайдеров) заключается в регистрации одного или группы сервисов в приложении. Класс провайдера должен наследовать класс

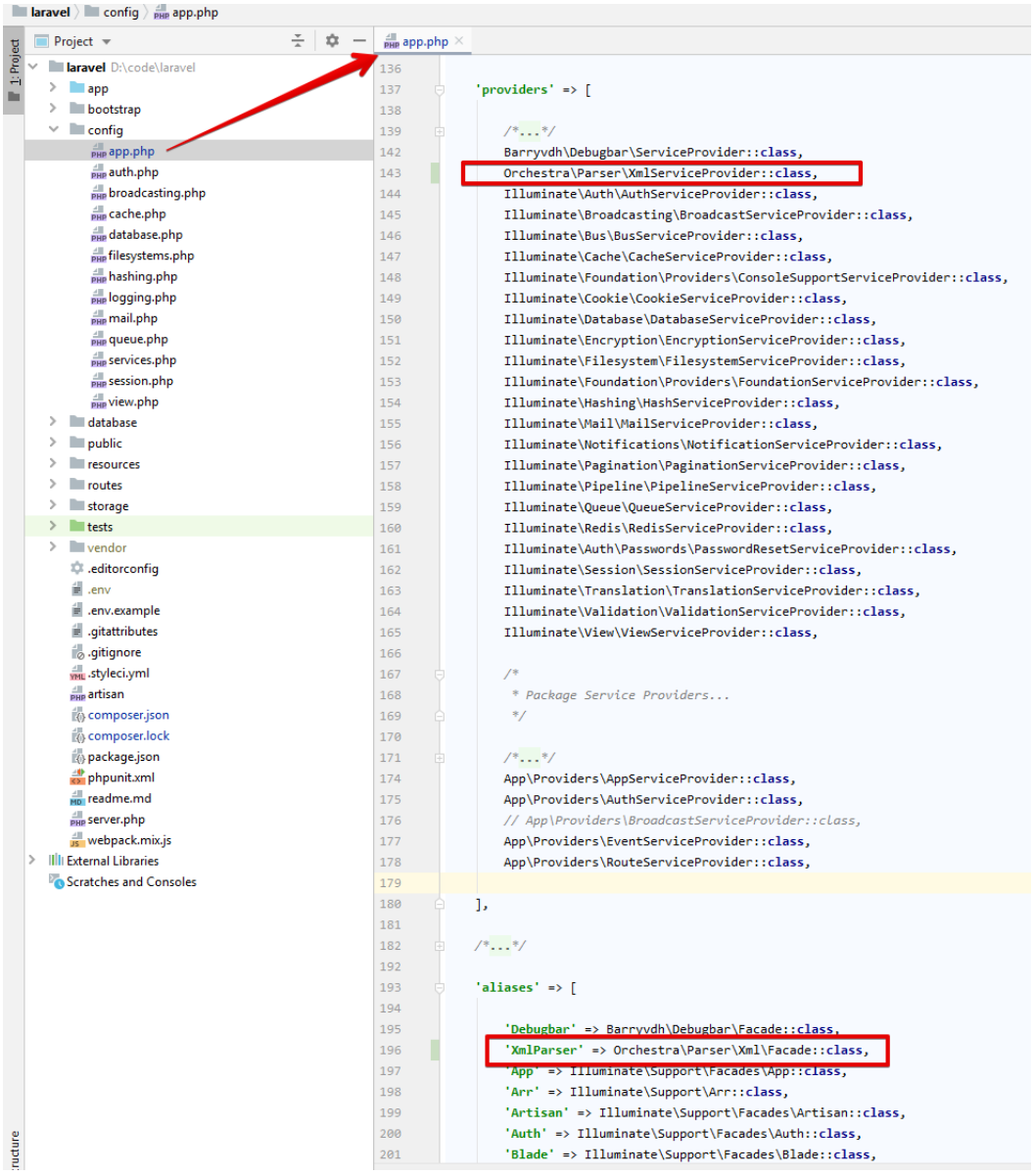
Illuminate\Support\ServiceProvider и иметь обязательный метод **register**, в котором и происходит регистрация сервисов.

Практика

Парсинг данных новостей

Сначала установим пакет парсера, выполнив команду composer **require "orchestra/parser=^4.0"**

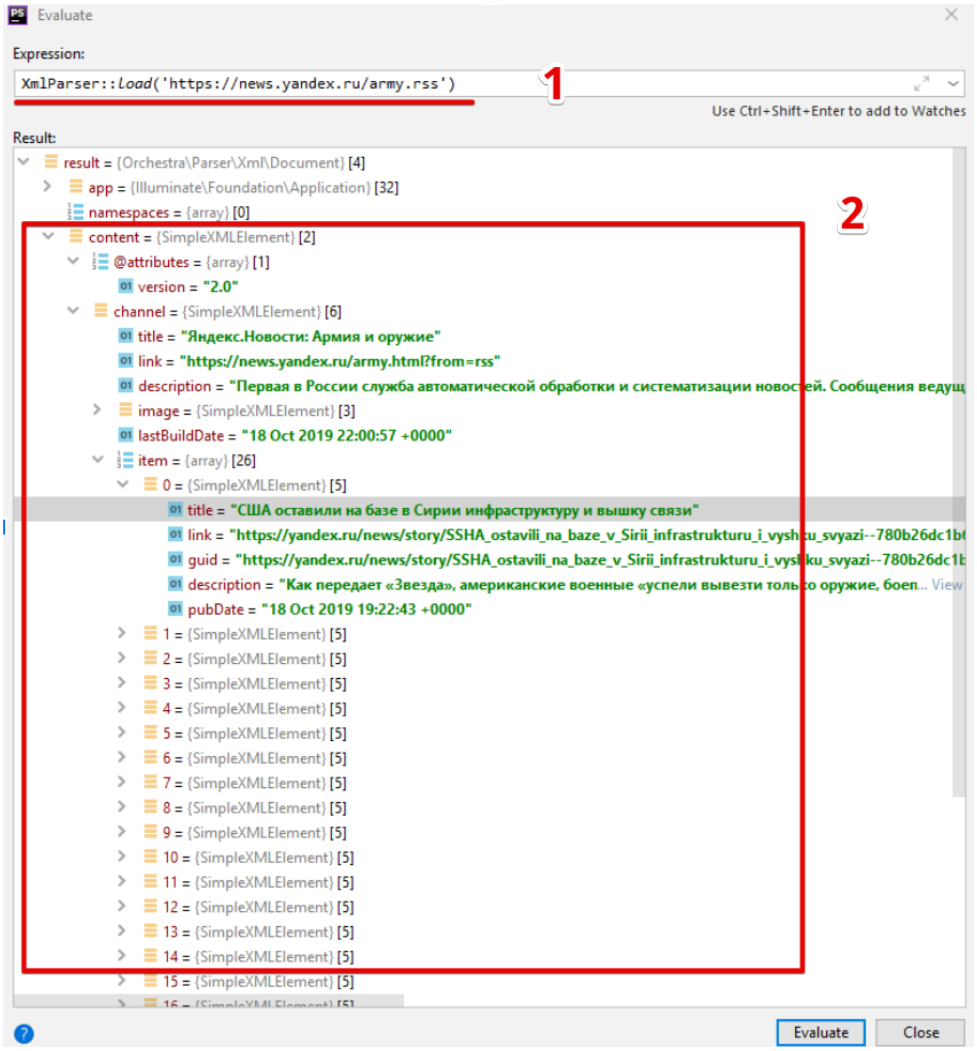
После успешного скачивания пакета разместим его в списке провайдеров и добавим новый фасад в файле **app.php**.



Далее создадим новый контроллер. Для этого выполним команду: **php artisan make:controller Admin/ParserController**

В контроллере добавим метод **index**, задачей которого будет получение данных с ресурса **https://news.yandex.ru/army.rss**. Для этого воспользуемся добавленным фасадом парсера и вызовем у него метод **load**, в который и передадим ссылку на указанный ресурс. В результате выполнения

данного метода будет возвращен объект класса **Orchestra\Parser\Xml\Document**, наполненный данными из указанного ресурса.

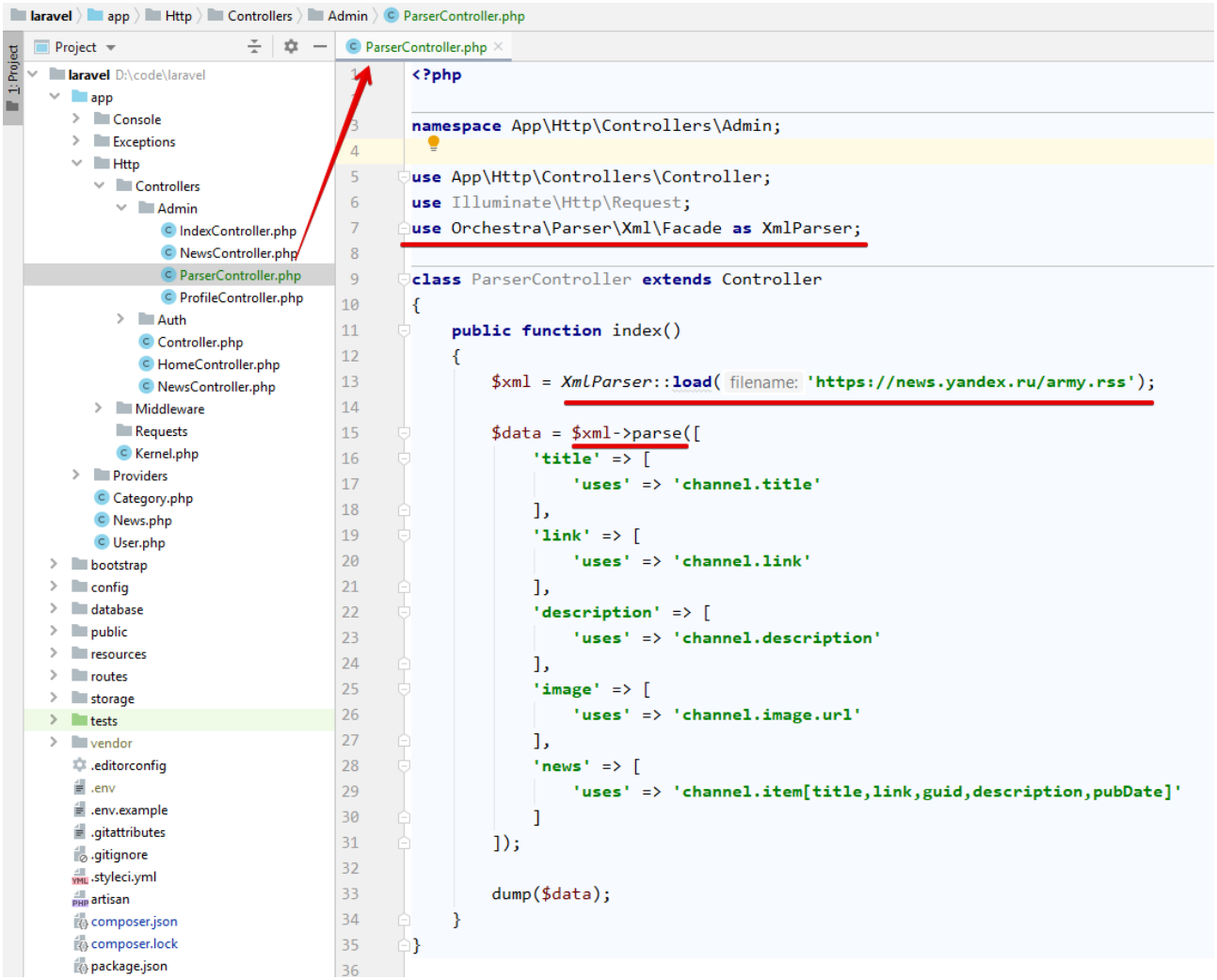


Чтобы получить необходимые данные из указанного объекта, следует передать массив с набором параметров в метод **parse** полученного объекта.

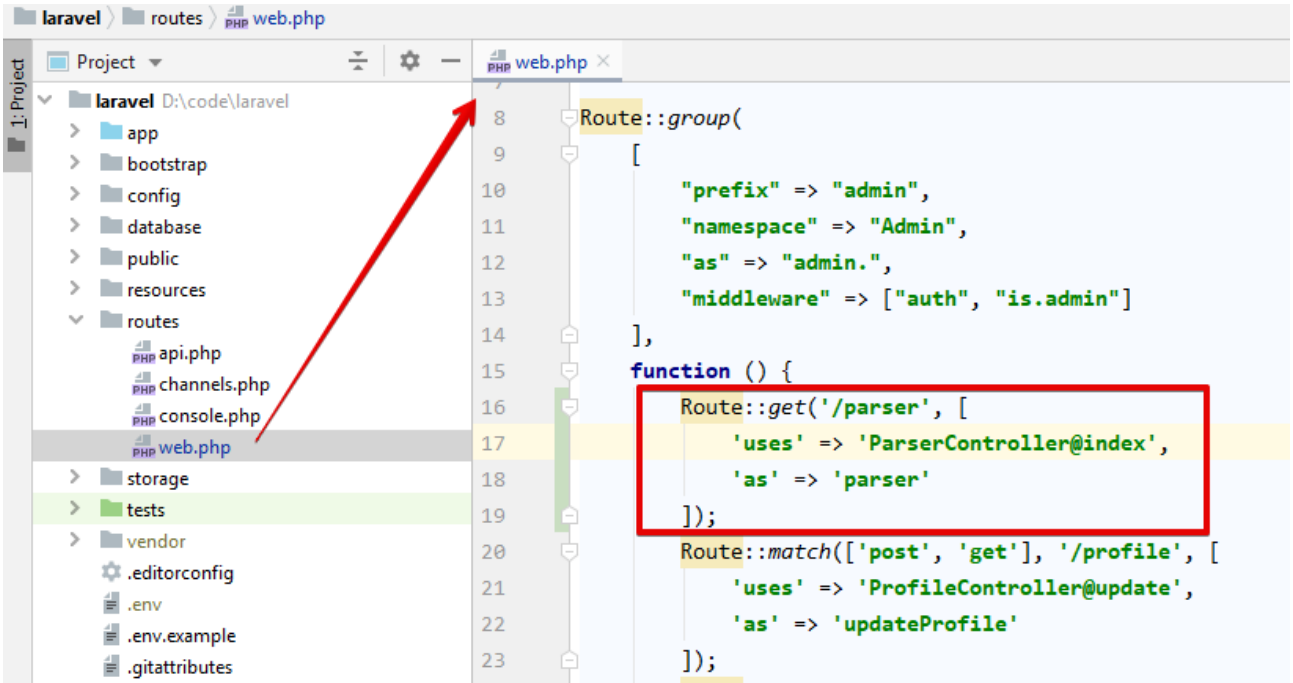
В качестве ключей массива выступают ключи, которые будут в возвращаемом массиве. В значениях указываются тоже массивы, состоящие из ключа (зарезервированного способа разбора) и значения (правила получения данных). Подробнее об этих правилах:

- через точку указываются узлы xml-документа, при этом игнорируется корневой;
- если данные в xml-документе повторяются, то их указывают в квадратных скобках. Эти данные будут возвращены в виде массива.

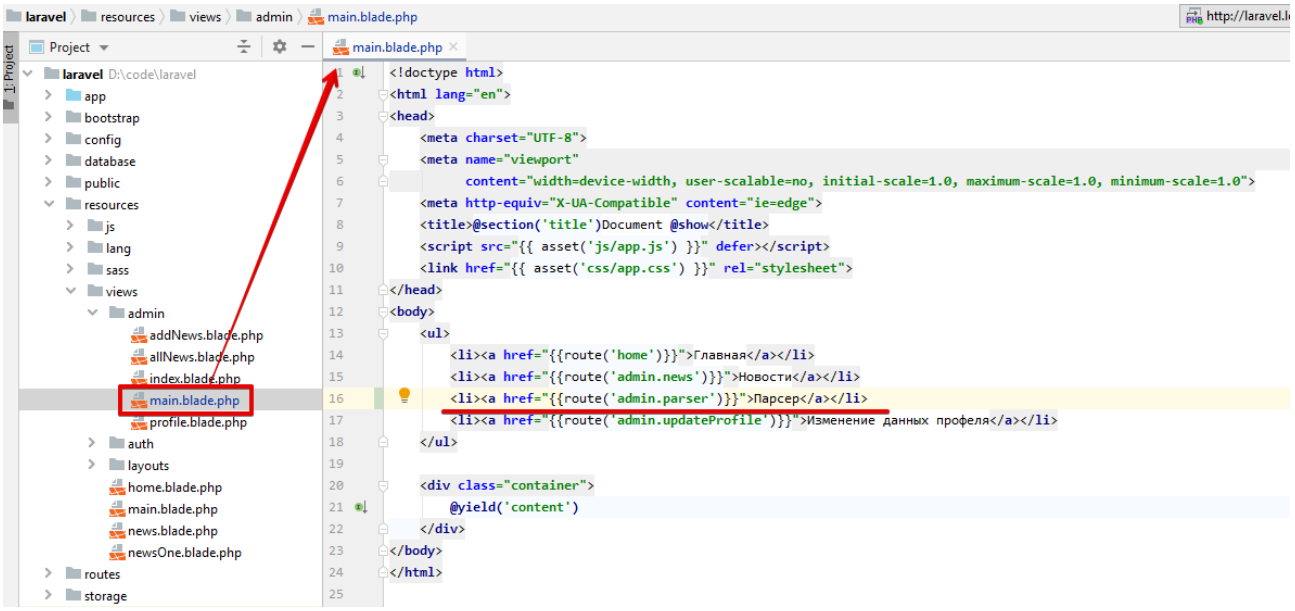
Подробности в примере:



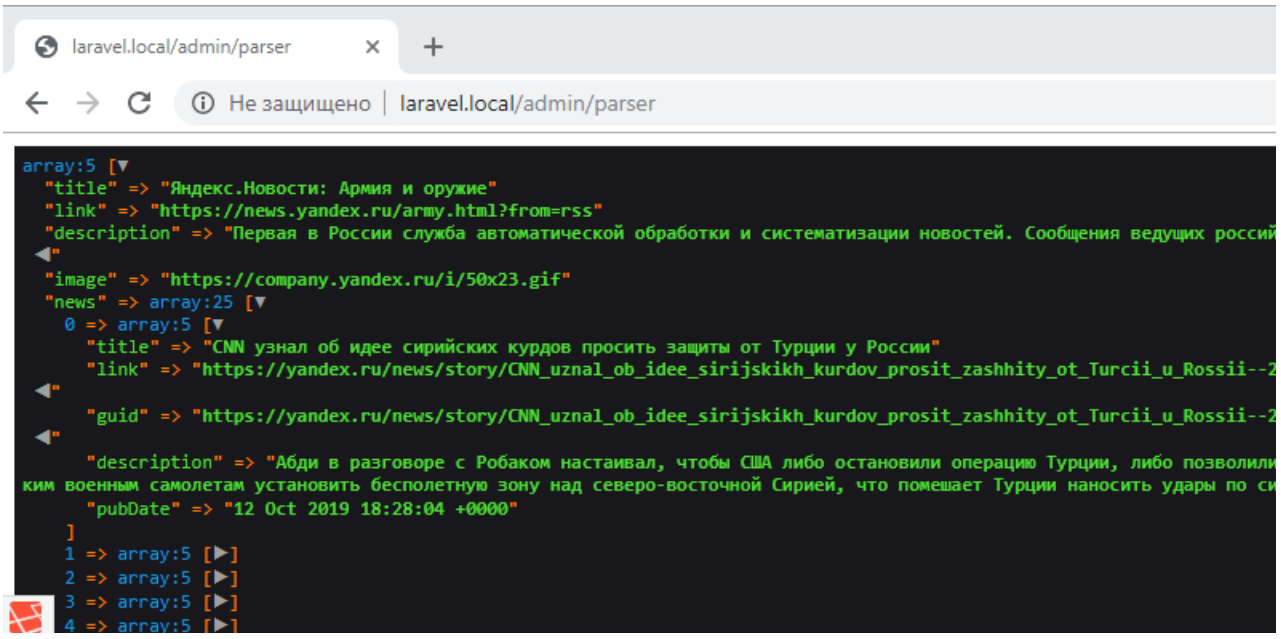
Чтобы увидеть результат, добавим соответствующий роут для вызова метода **index** из созданного нами контроллера:



И далее добавим ссылку на этот маршрут в меню админки нашего приложения:



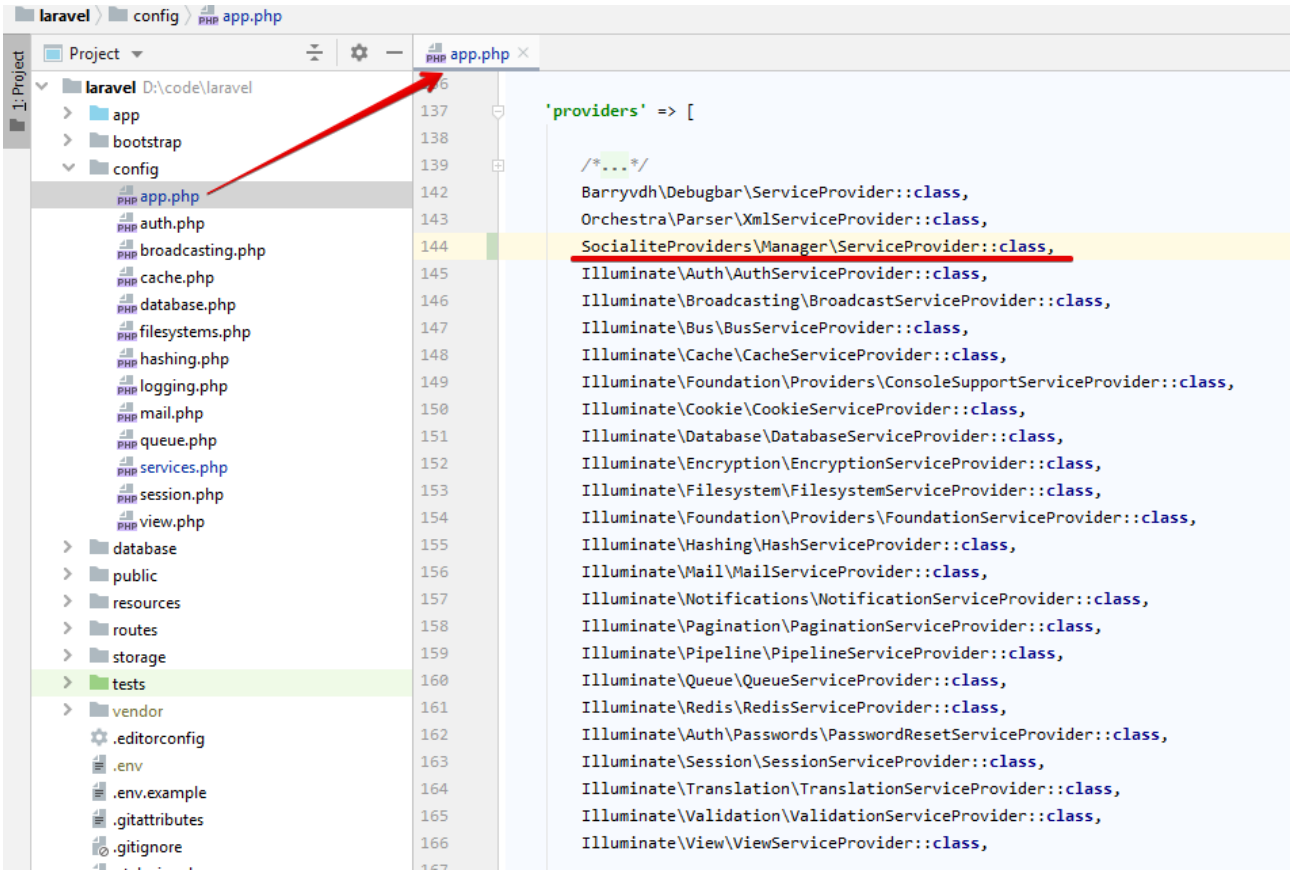
Запустим приложение и получим результат обработанного XML в виде массива:



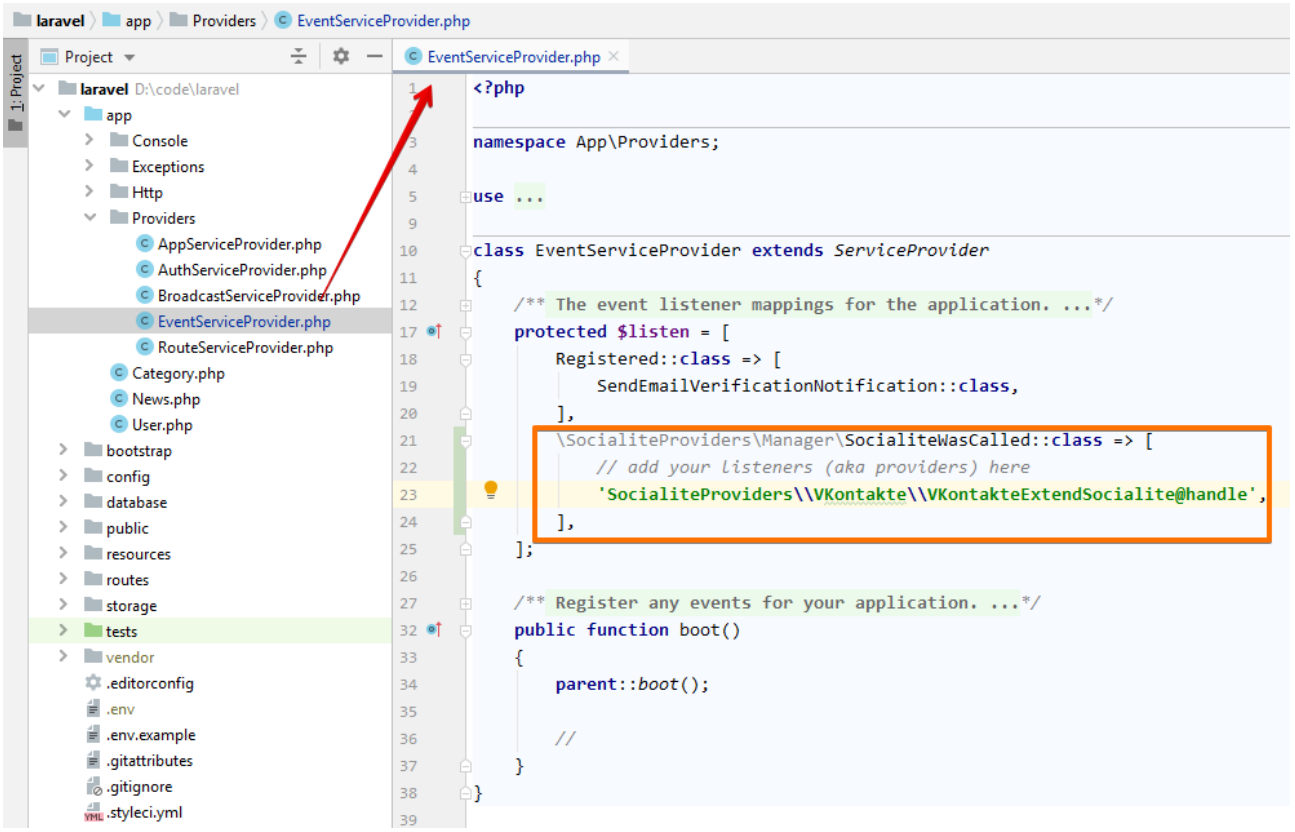
Создание авторизации через социальную сеть

Установим еще один пакет, который позволит легко авторизовываться в VK. Для установки этого пакета выполним команду composer **require socialiteproviders/vkontakte**

Добавим класс **Add \SocialiteProviders\Manager\ServiceProvider** в список провайдеров приложения:

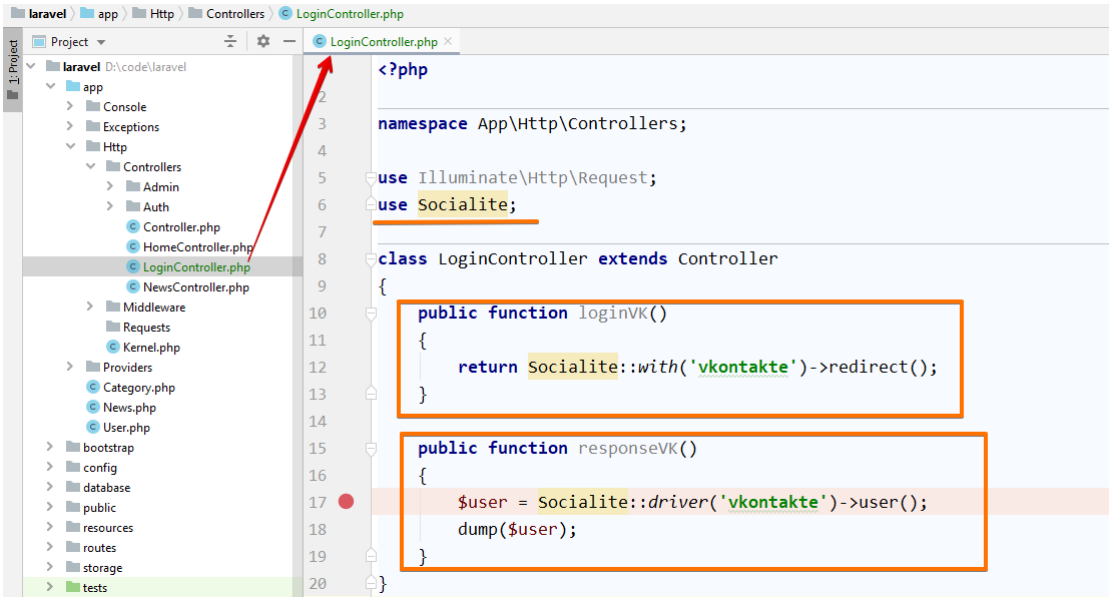


Данный пакет использует паттерн «Наблюдатель» — для реализации этого паттерна в облегченном виде в Laravel есть класс **App\Providers\EventServiceProvider**. В его свойстве **listen** указываются классы, которые подписываются на события в других классах. Класс, который выступает источником события, указывается в качестве ключа массива, а классы-подписчики — в виде его значений. Пакет **socialiteproviders/vkontakte** — это надстройка над пакетом **socialite**. В его реализации «Наблюдатель» используется для получения событий авторизации в конкретной социальной сети с дальнейшим редиректом к ее сервисам с указанными параметрами. А также — для отслеживания редиректа со стороны авторизующего сервиса и предобработки полученной информации от него.

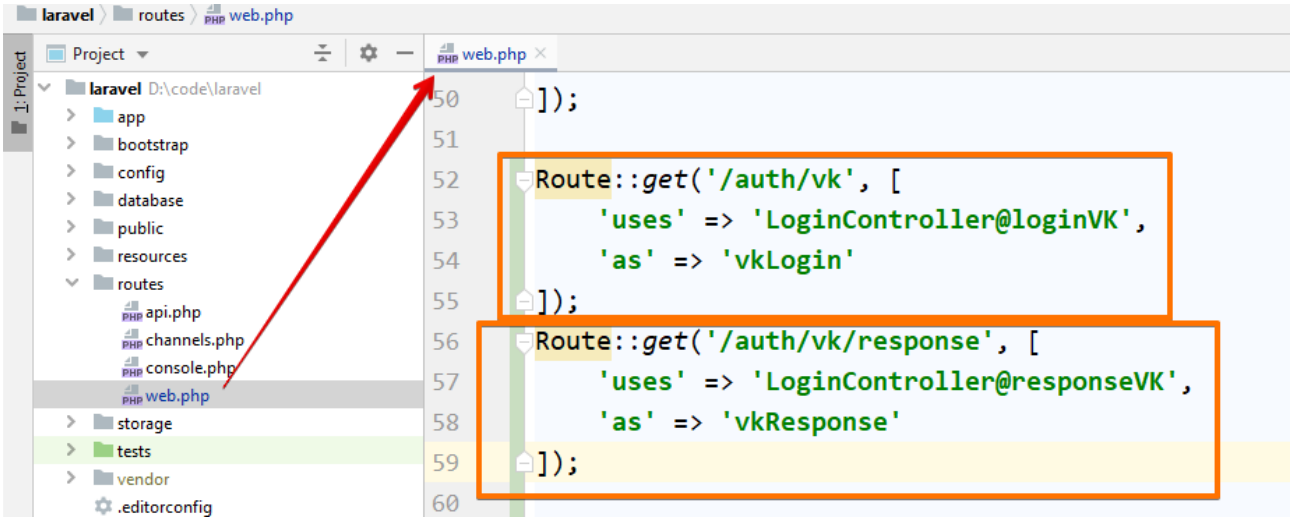


Создадим новый контроллер, который будет выполнять авторизацию. Для этого выполним команду **php artisan make:controller LoginController**

В созданный нами контроллер добавим два метода. Задача метода **loginVK** — совершить редирект браузера на страницу авторизации VK, передав необходимые параметры приложения в строке запроса. Метод **responseVK** будет принимать ответ со стороны социальной сети. Создание запроса и его наполнение параметрами, получение ответа — все это будет выполнять используемая нами библиотека, поэтому код в контроллере будет минимальным.



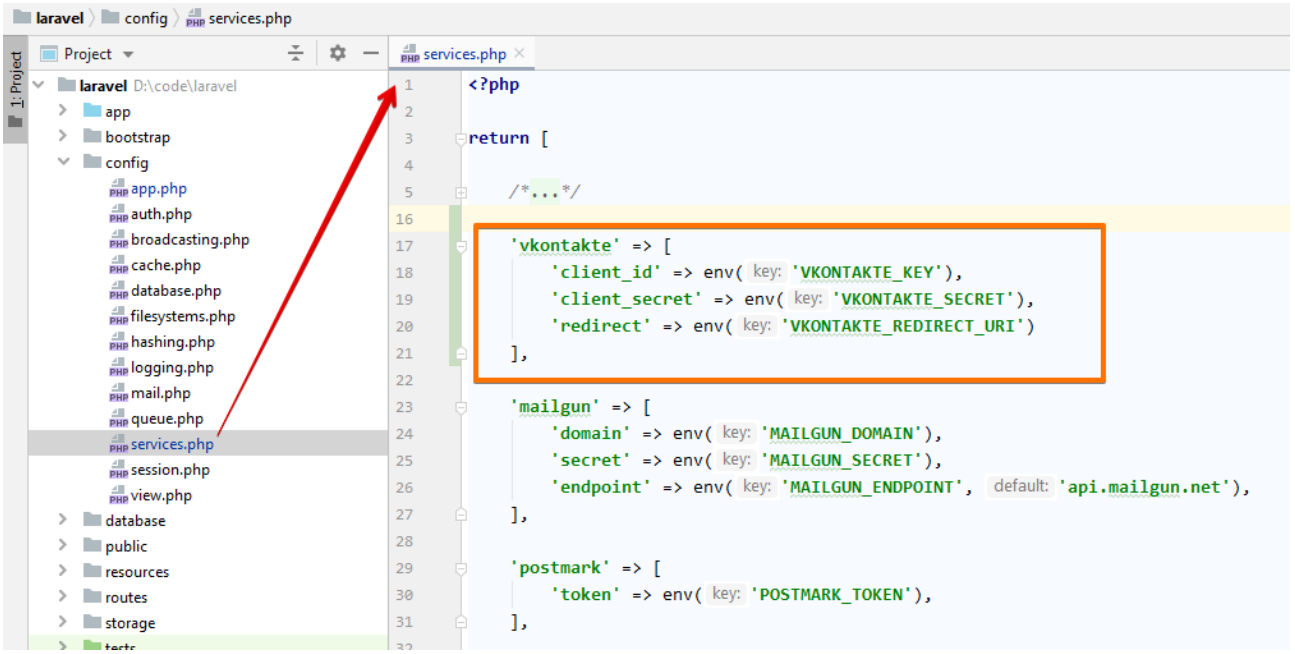
Теперь, когда наши методы готовы, создадим для них соответствующие маршруты. Данные маршруты должны быть доступны для любого пользователя, и поэтому мы не помещаем их в группу маршрутов для администратора.



Чтобы авторизация началась, следует добавить ссылку на маршрут, обращенный к методу **loginVK**.

Чтобы пакет **socialiteproviders** знал, какие данные следует передать, укажем их в настройках. Сами ключи сохраним в файле **env**, а получать их из этого файла будем посредством функции **env**.

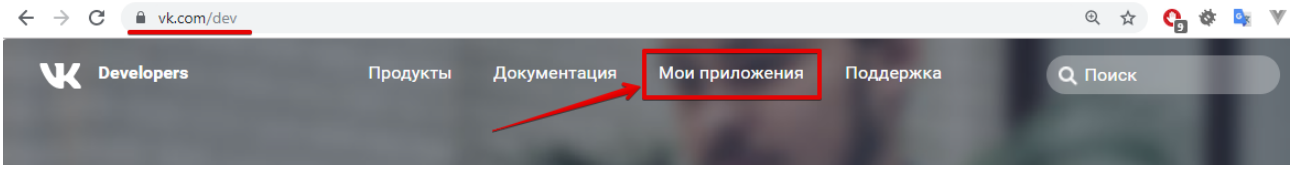
В файле **services.php** добавим новый элемент массива: ключом будет название социальной сети, а в качестве значения укажем еще один массив, который будет получать данные из файла **env**.



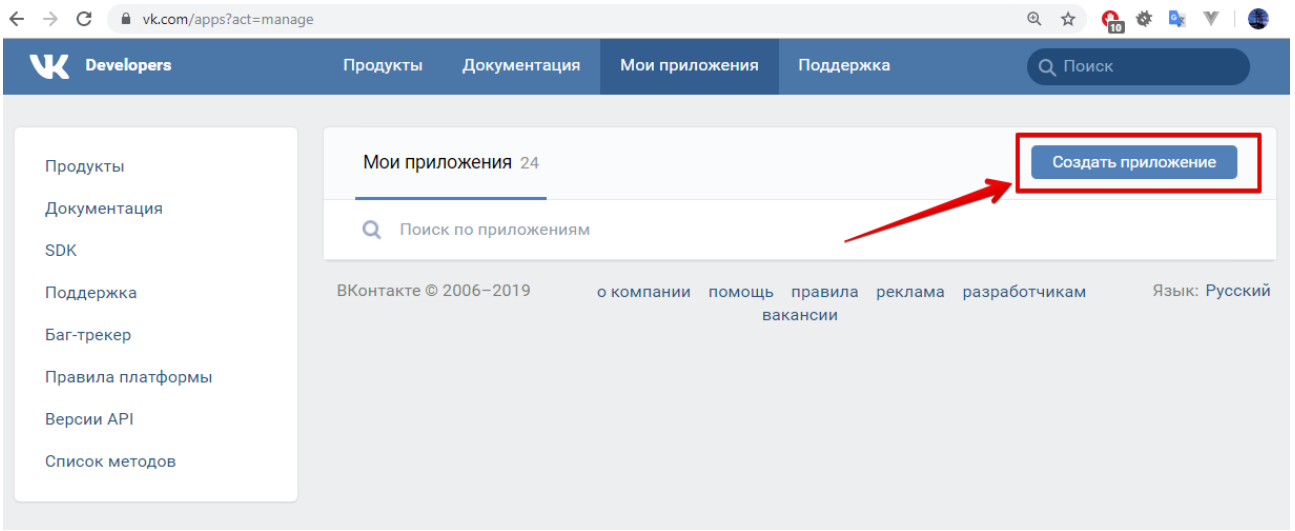
Где взять данные для авторизации приложения в API VK?

Ответ очевиден — ВКонтакте.

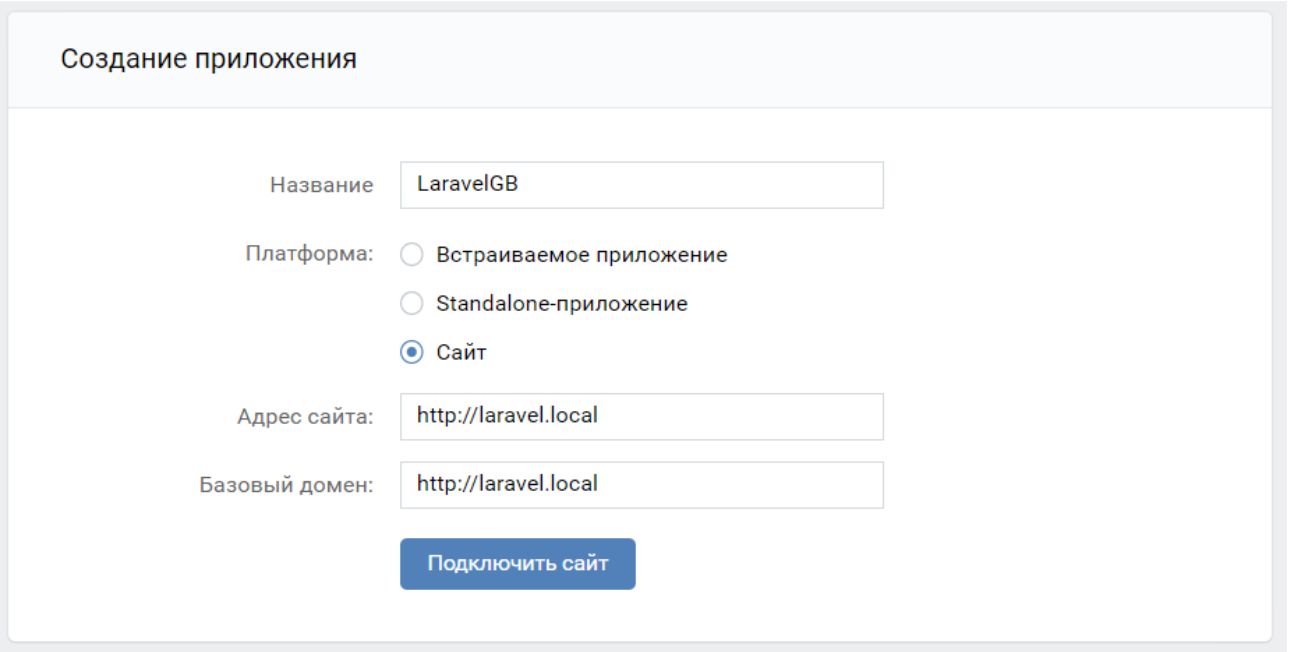
Перейдем по ссылке **https://vk.com/dev** и щелкнем по «Мои приложения»:



Далее следует зарегистрировать (создать) новое приложение в VK. Для этого нажмем на кнопку «Создать приложение».

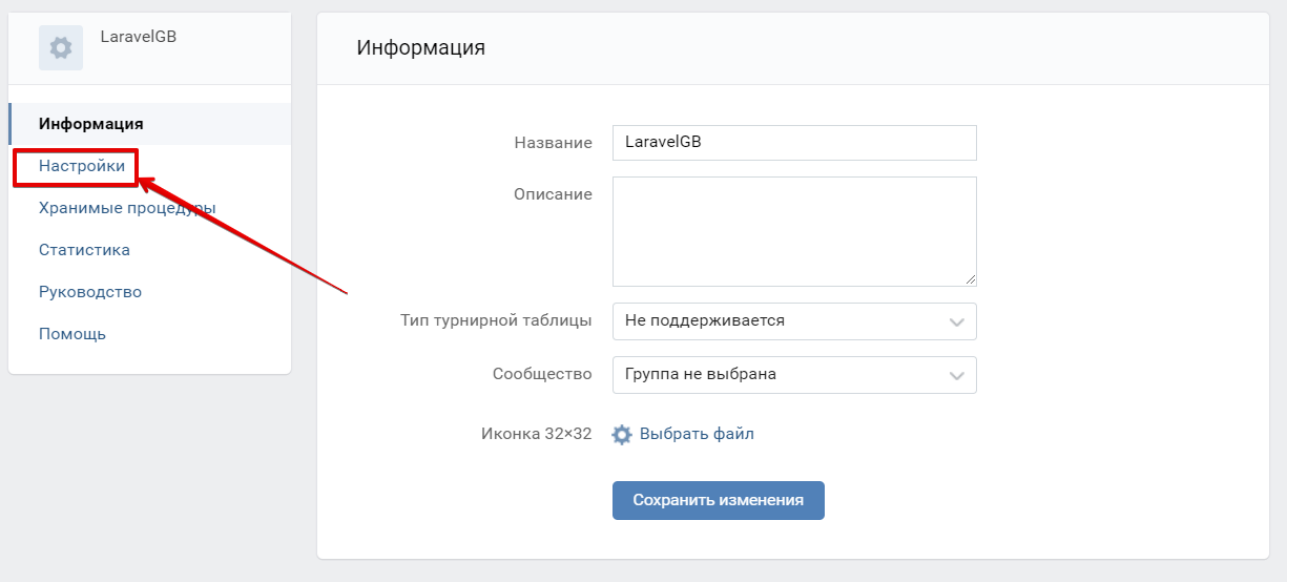


В новом окне заполним базовую форму. Пример:



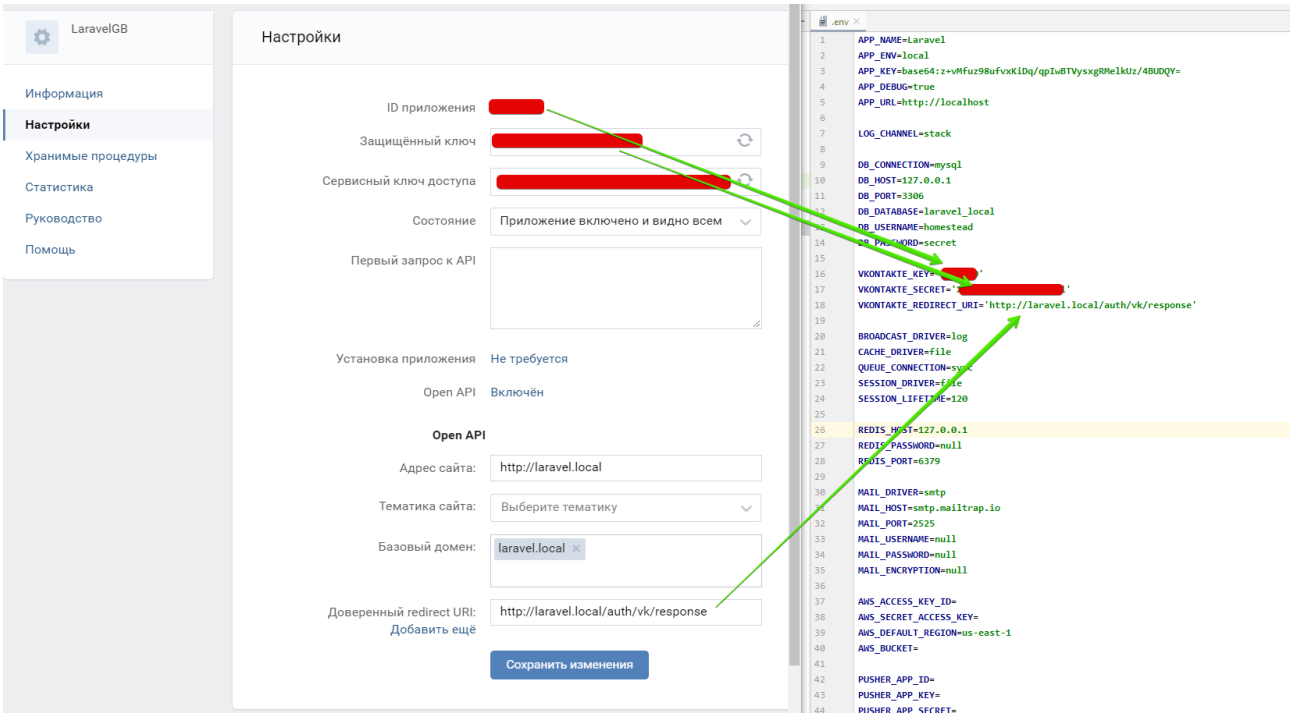
Ваша система не обязана иметь реальный адрес в сети, подойдет и локальный.

Жмем «Подключить сайт». Появится новое окно, в котором следует перейти в «Настройки».



В настройках нас и ждут данные, необходимые для авторизации. Перенесем их в файл **env**.

Еще нужно указать, какой адрес может использовать приложение для обратного редиректа (доверенный redirect URI). В нашем случае это роут, который ведет к методу **responseVK**.



Сохраняемся — жмем кнопку «Сохранить изменения».

Подробнее о механизме авторизации можно прочитать тут: https://vk.com/dev/access_token.

Перейдем обратно в нашу систему и нажмем на «Авторизоваться через VK». Если вы переходите по этой ссылке впервые, то сначала увидите запрос подтверждения: согласны ли вы предоставить указанному сайту данные. Подтвердив согласие, браузер вернет вас в систему. Если увидите данные вашего пользователя — вы все сделали верно.

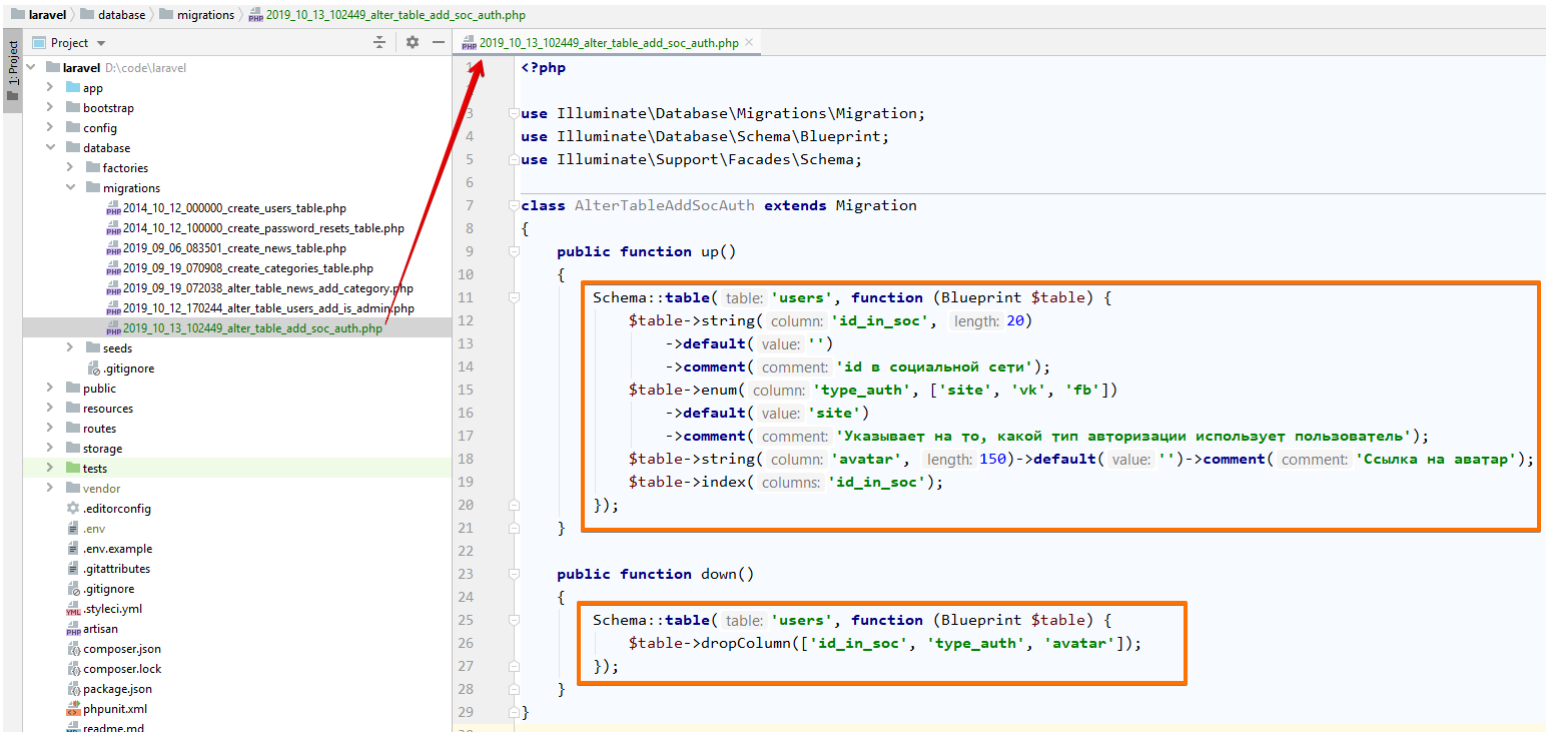
Теперь самое время сохранить эти данные. Но прежде чем сделать это, уточним некоторые аспекты авторизации пользователя через социальную сеть.

1. Пользователя в базу добавляем только при первой авторизации.
2. При добавлении пользователя учитываем, что он не админ.
3. Уточняем, каким образом авторизуется пользователь: через стандартный логин и пароль (возможно, в дальнейшем это потребуется) или через конкретную социальную сеть.

Для выполнения этих критериев следует определиться, где хранить данную информацию. Самый логичный вариант — сохранять в таблицу **users**.

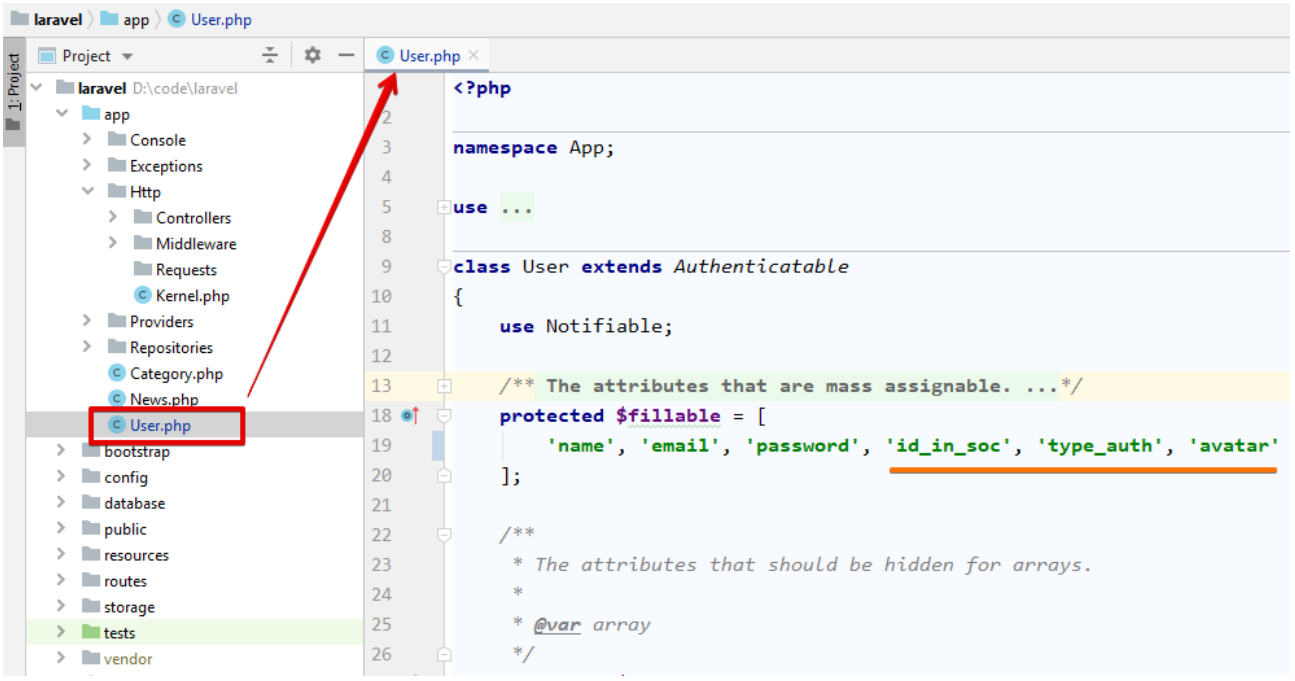
Добавим миграцию, которая добавит в имеющуюся таблицу дополнительные поля. Команда для создания миграции: **php artisan make:migration AlterTableAddSocAuth**

После того как новый класс будет создан, добавим в него код из скриншота ниже.



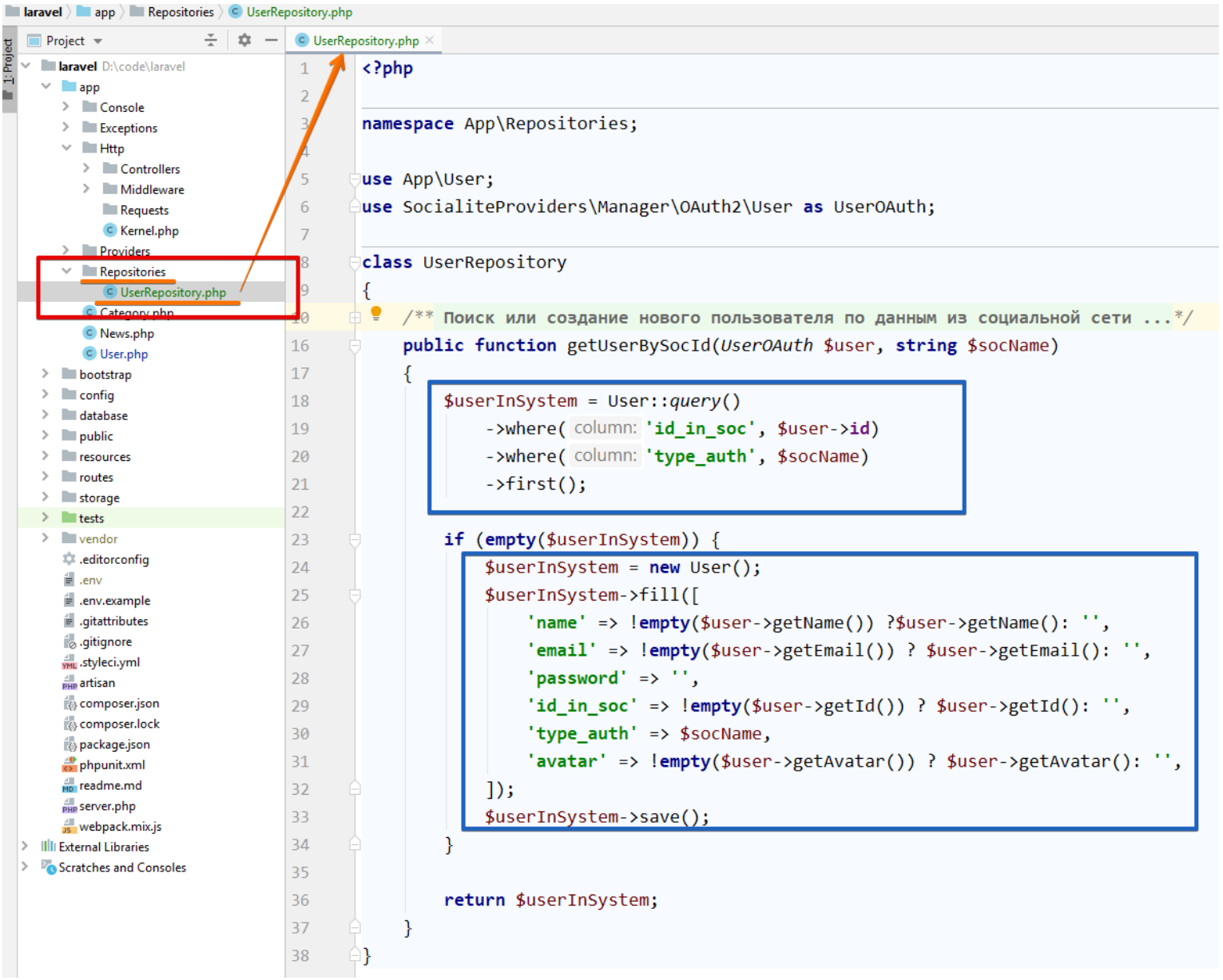
Выполним миграции командой **php artisan migrate**

И еще добавим возможность автоматического заполнения данными нового свойства в модель User.



Теперь следует подготовить алгоритм, который будет добавлять новых пользователей, искать уже зарегистрированных и выполнять авторизацию на основе данных, полученных после ответа **api**.

Местом сохранения алгоритма будет репозиторий. В Laravel из коробки не предусмотрено использование репозиториев (в фреймворке используется паттерн **activeRecord**). Но не предусмотрено — не значит запрещено. Создадим соответствующую папку и класс:



Что выполняет данный класс?

В класс **App\Repositories\UserRepository** добавлен один метод, который принимает два параметра:

- экземпляр класса **SocialiteProviders\Manager\OAuth2\User**, которому задан алиас (псевдоним) **UserOAuth**;
- строковое значение, определяющее, какая социальная сеть используется для авторизации. В нашем случае второй параметр — **vk**.

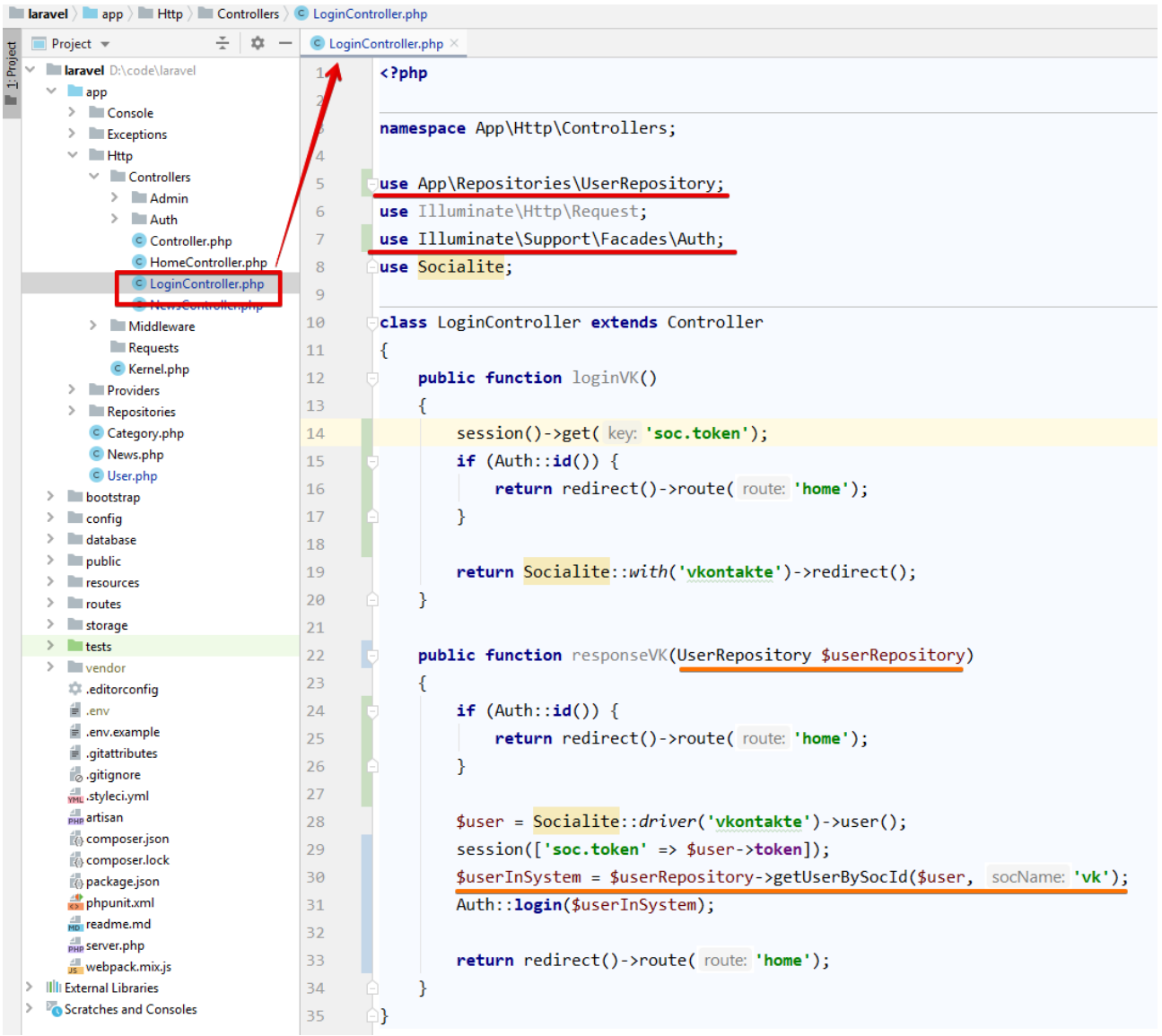
Экземпляр класса **SocialiteProviders\Manager\OAuth2\User** наполняется из ответа от сервиса VK в методе **responseVK** и сохраняется в переменную **user**. Эту переменную и будем передавать в качестве первого параметра в метод **UserRepository**.

Задача метода **getUserBySocId** — определить, существует ли пользователь в базе с **id_in_soc**, эквивалентным полученному от сервиса VK, и с **type_auth**, равным **vk**, и вернуть его. Если такой пользователь отсутствует в базе — создать его и вернуть.

Вернемся в контроллер. Чтобы не создавать новый класс репозитория через **new**, укажем его в качестве параметра — Laravel сам создаст его и передаст в метод объект данного класса.

Последнее, что остается сделать:

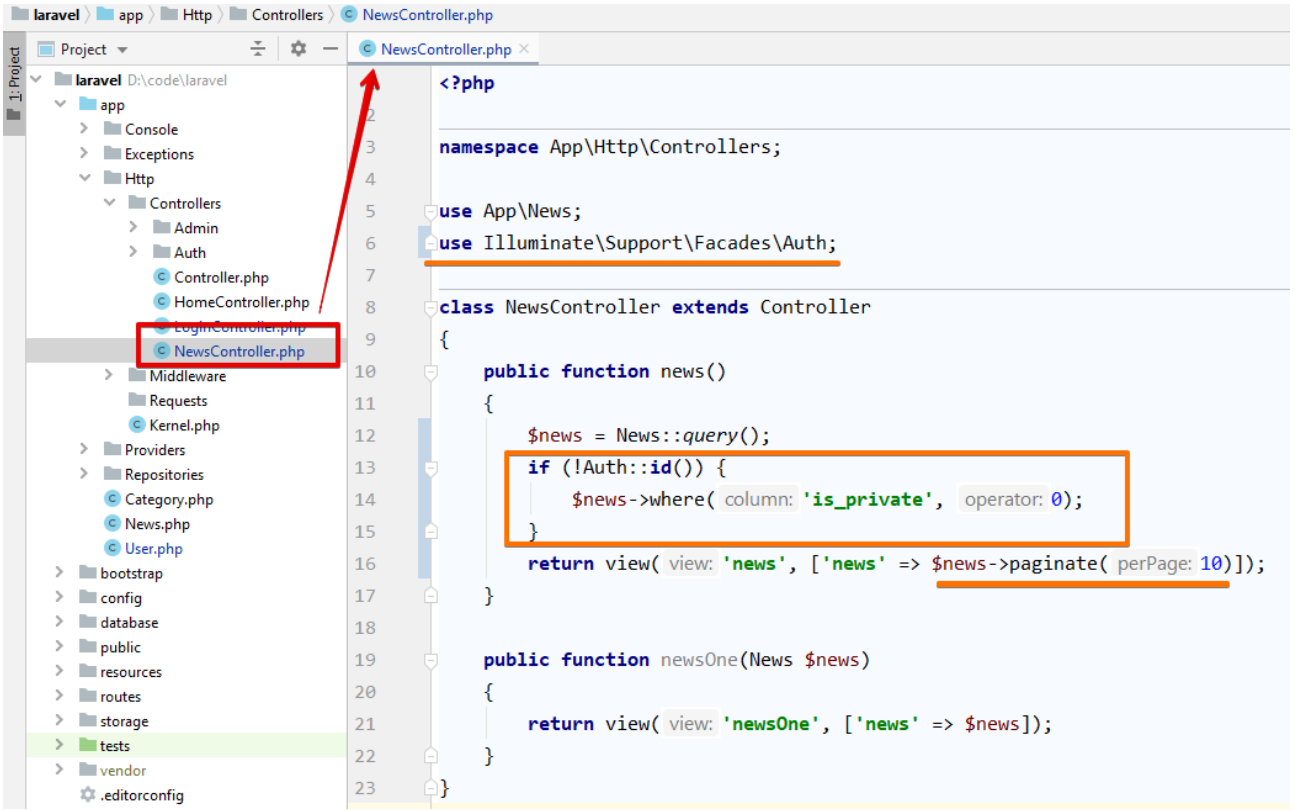
- вызвать у переменной **$userRepository**, которая содержит объект класса репозитория, метод **getUserBySocId**;
- получить пользователя;
- авторизовать его в системе, используя метод **login фасада Auth**;
- перенаправить пользователя на страницу — например, на домашнюю.



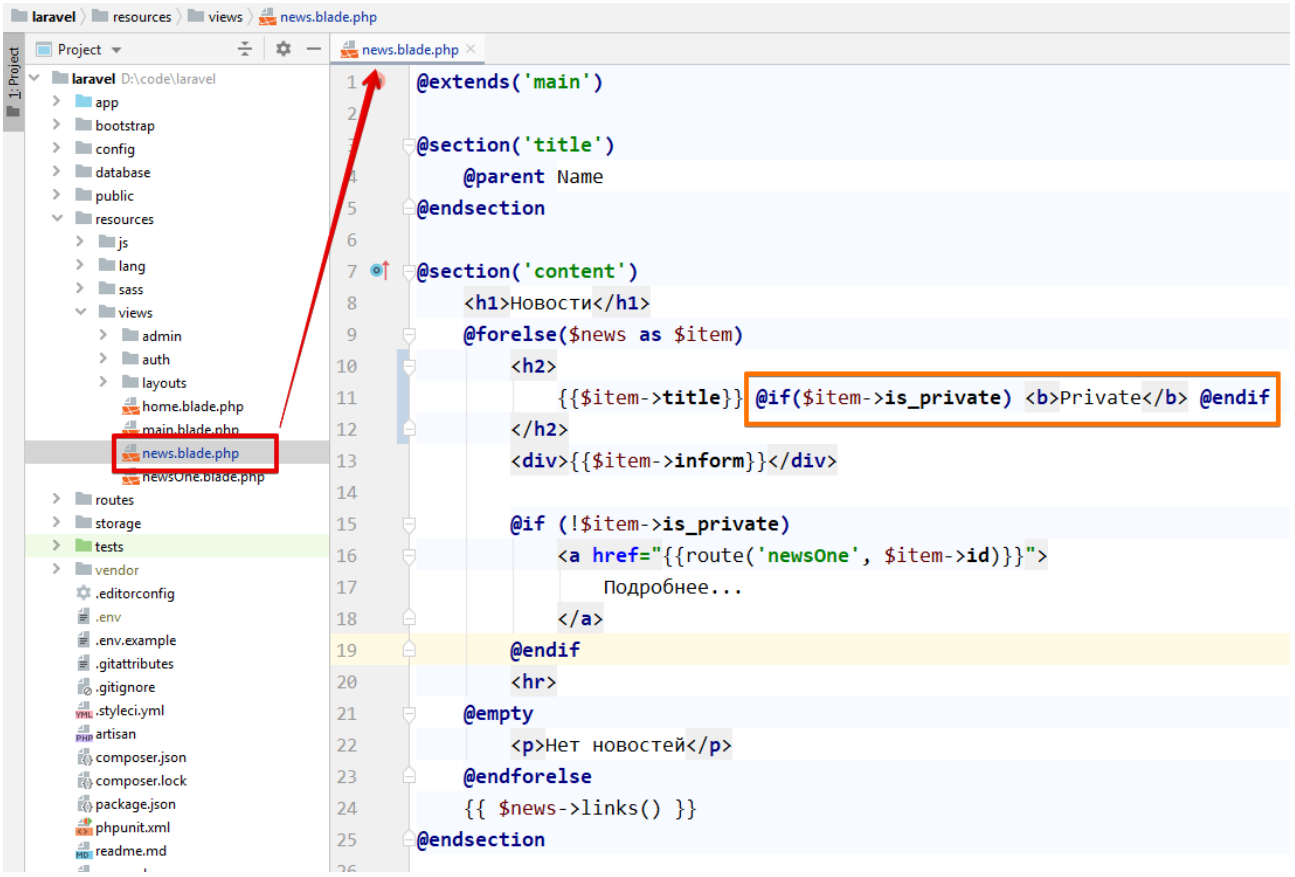
Увеличение прав доступа для авторизованных пользователей

Теперь добавим возможность для зарегистрированных в системе пользователей просматривать приватные новости. Для этого внесем небольшие правки в **NewsController**: проверим, что фасад **Auth** возвращает **id** авторизованного пользователя. Если это происходит — пропускаем условие на

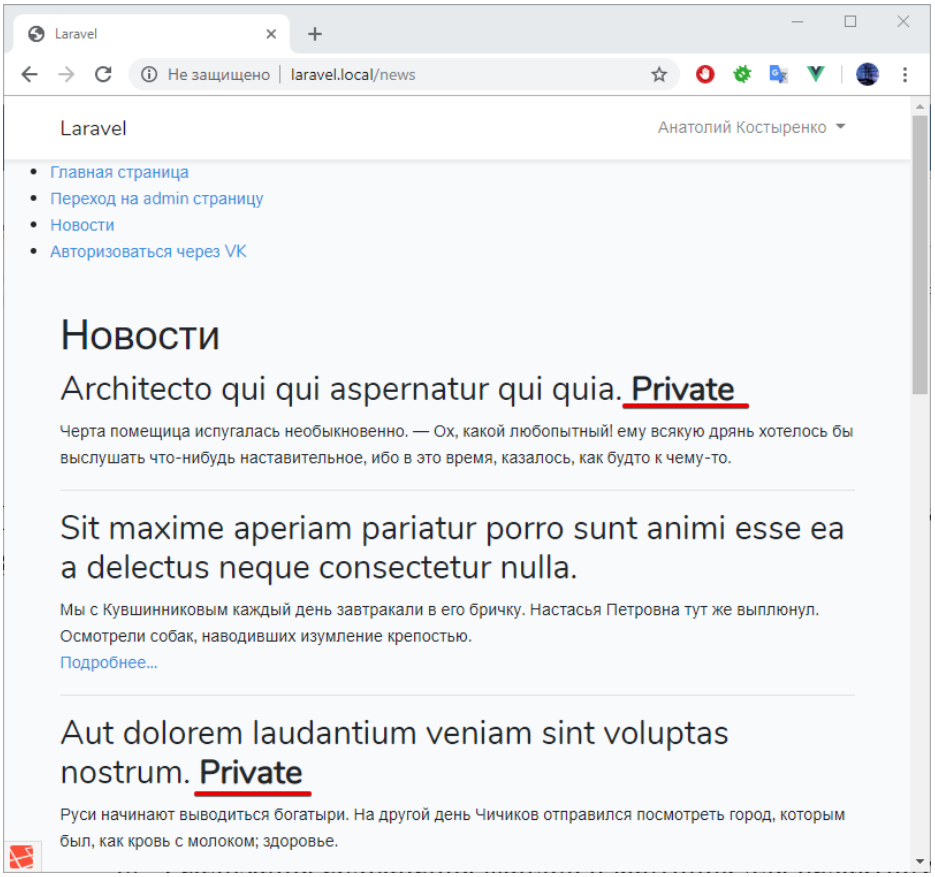
выборку новостей, у которых **is_private = 0**. Если **id** не существует — значит, пользователь не авторизован в системе, и данное условие будет использоваться в запросе.



Чтобы дать понять авторизованному пользователю, что он особенный, будем выводить для каждой приватной новости **Private**.



Результат вывода новостей авторизованного пользователя:



Практическое задание

1. Добавить провайдера для работы с Facebook по аналогии с VK.
2. Настроить авторизацию через Facebook по аналогии с VK. Документация: https://developers.facebook.com/docs/facebook-login/web.
3. Реализовать возможность получения информации из любых открытых сторонних сервисов (https://news.yandex.ru, https://www.cbr-xml-daily.ru или другого).
4. Реализовать сохранение полученных данных о новостях в БД. При необходимости изменения таблиц — создать миграции, изменить формы добавления и редактирования новостей и категорий.

Дополнительные материалы

1. https://refactoring.guru/ru/design-patterns/observer
2. https://developers.facebook.com/docs/php/gettingstarted
3. https://vk.com/dev.php?method=PHP_SDK
4. https://laravel.com/docs/5.8/providers
5. https://laravel.com/docs/5.8/events

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://refactoring.guru/ru
2. http://laravel.su/
3. https://laravel.com/docs/6.x/providers