GeekBrains

Laravel

Урок 10. Добавление HTML-редактора. Хранение файлов. Очереди в Laravel

Изменение полученных агрегатором данных и создание пользовательских. Добавление интерфейса для редактирования html-страниц с возможностью добавления изображений. Очереди для выполнения параллельных и последовательных задач.

Оглавление

Практика

Установка HTML-редактора

Чтобы получить информацию со стороннего сервиса или ввести ее на сайте, иногда требуется добавлять в нее html-разметку и изображения, менять шрифт или отступы, вводить нумерованные списки. Конечно, всегда можно внести разметку в обычное текстовое поле, а изображения загрузить, используя **ftp**, но это неудобно.

Чтобы пользователю было легко работать с вашим сервисом, можно установить html-редактор и специальный файловый менеджер. Он позволит загружать файлы на сервис и добавлять их в указанную часть редактируемого материала в виде ссылок.

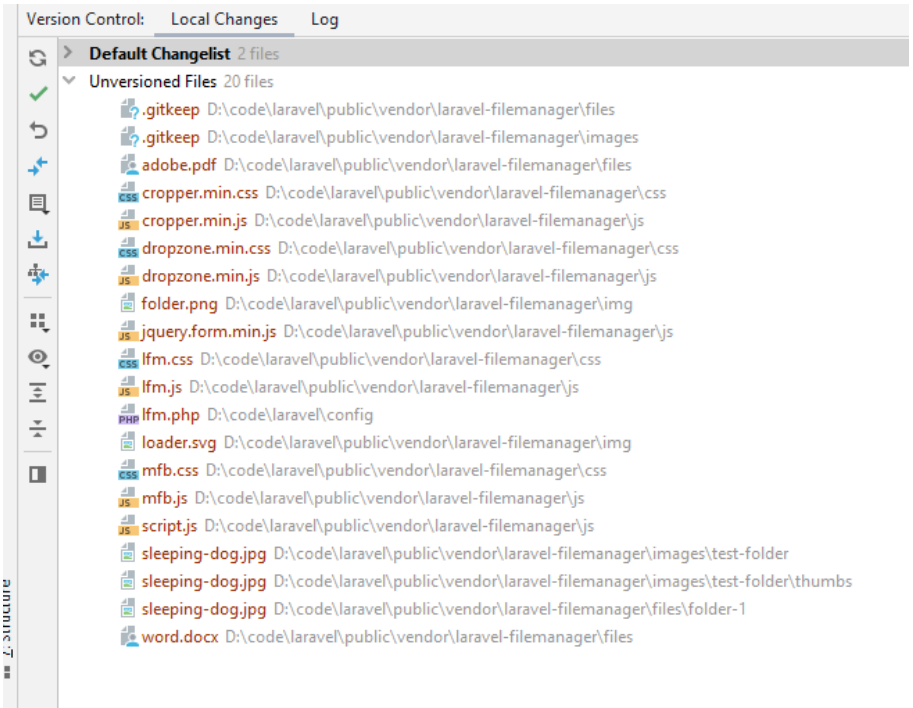
В качестве менеджера файлов рассмотрим пакет **laravel-filemanage**r. Подробная документация по нему доступна по адресу https://unisharp.github.io/laravel-filemanager/installation.

Для начала установим его в наше приложение, выполнив команду **composer require unisharp/laravel-filemanager:~1.8**

Добавим необходимые зависимости в папку **vendor**. Затем надо перенести из нее некоторые данные в основное приложение. Для этого выполним следующие команды:

- **php artisan vendor:publish --tag=lfm_config**
- **php artisan vendor:publish --tag=lfm_public**

Обратим внимание на локальные изменения в проекте:



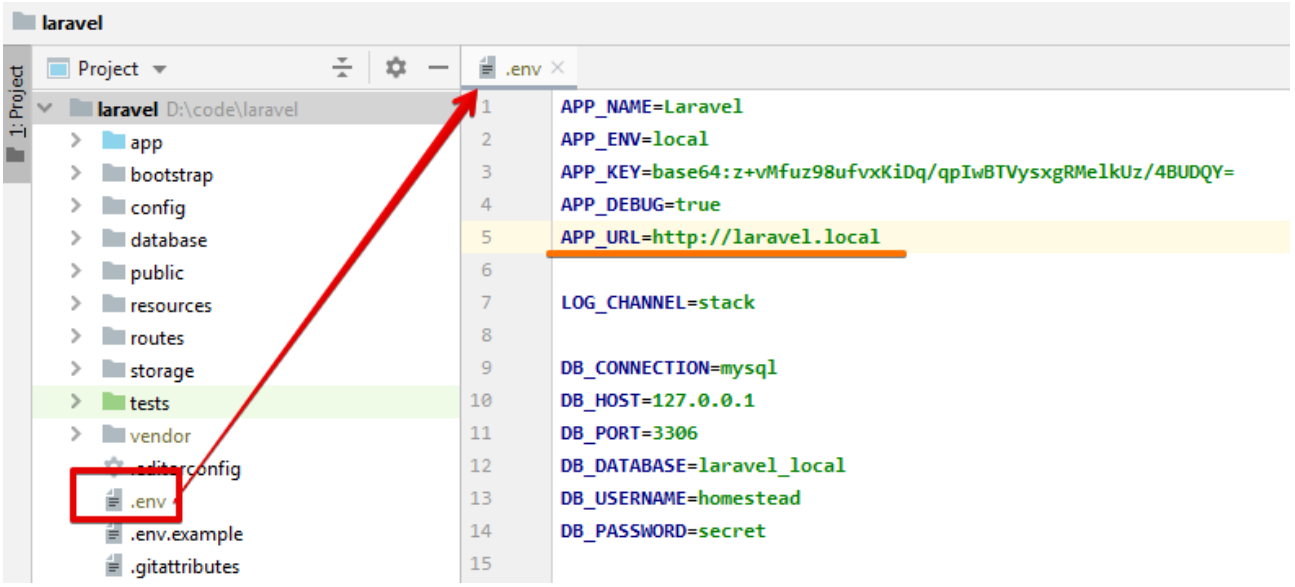
Теперь сбросим кеш нашего приложения, выполнив команды:

- **php artisan route:clear**

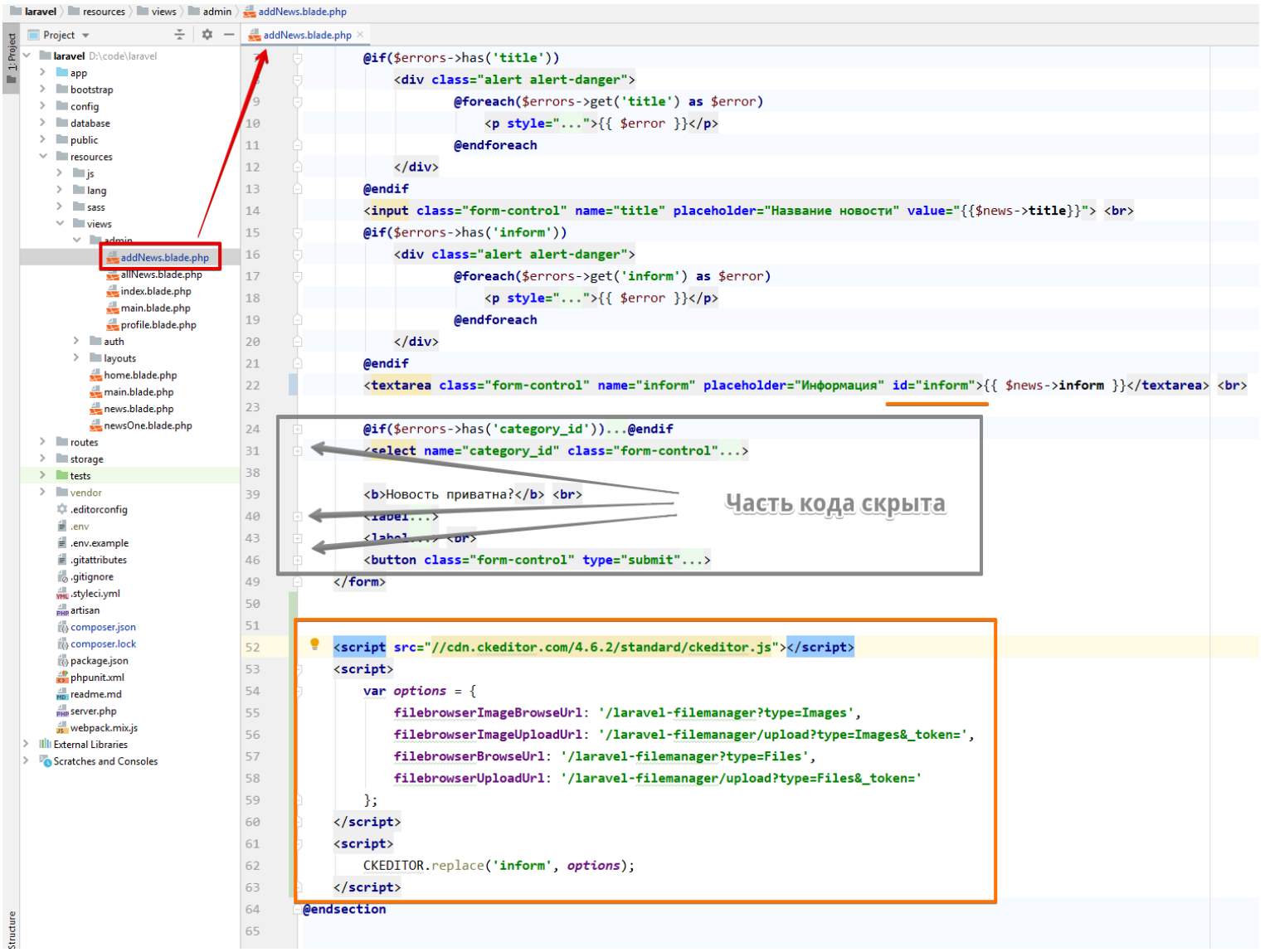
- **php artisan config:clear**

Также создадим симлинк из **public/storage** в **storage/app/public**, выполнив команду **php artisan storage:link**

Еще одно условие работы менеджера — правильное указание параметра **APP_URL**. В его значении должен стоять верный url вашего приложения.



Все приготовления выполнены. Теперь установим html-редактор **ckeditor** на страницу добавления и редактирования новостей. В качестве настроек укажем маршруты нашего файлового менеджера.

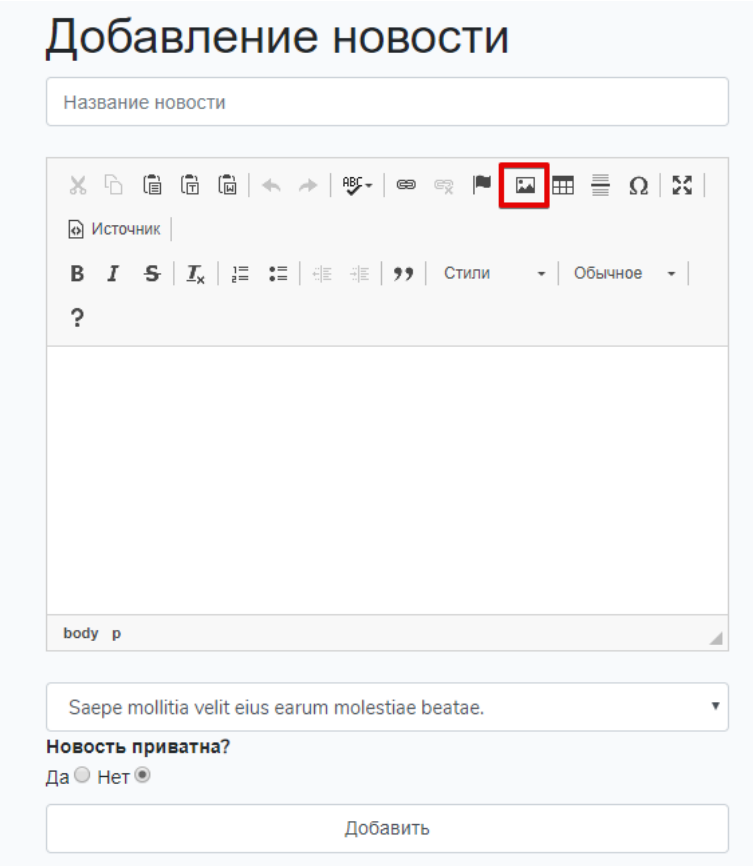


Это не самый удачный вариант, так как данные html-редактора будут тянуться из стороннего сервиса. «Портянка» из наборов параметров, которые мы передаем при вызове метода **replace**, тоже выглядит не очень.

Как быть, подскажет документация: https://ckeditor.com/docs/ckeditor5/latest/index.html.

А мы движемся дальше.

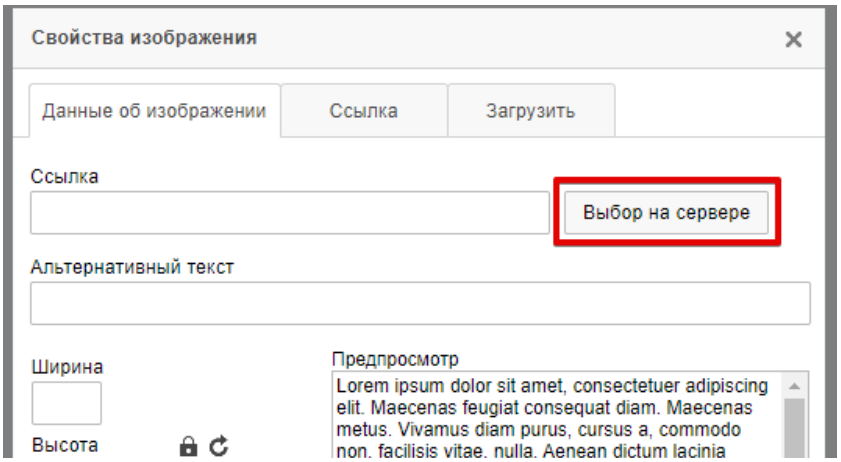
Перейдем в админ-панель и попробуем создать новую новость.



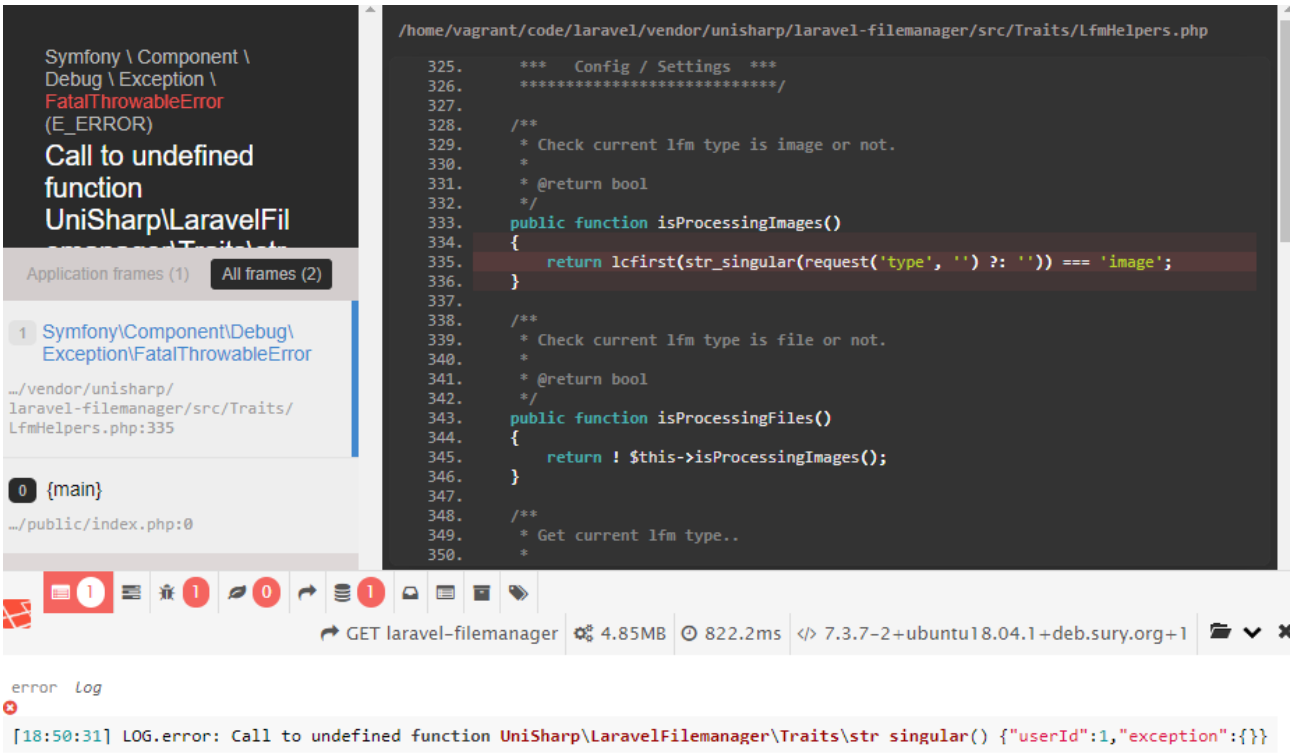
Редактор HTML на месте.

Теперь попробуем воспользоваться файловым менеджером. Для этого нажмем на кнопку «Изображение» (скриншот выше).

Должно появиться модальное окно. В нем нас интересует кнопка «Выбор на сервере»:



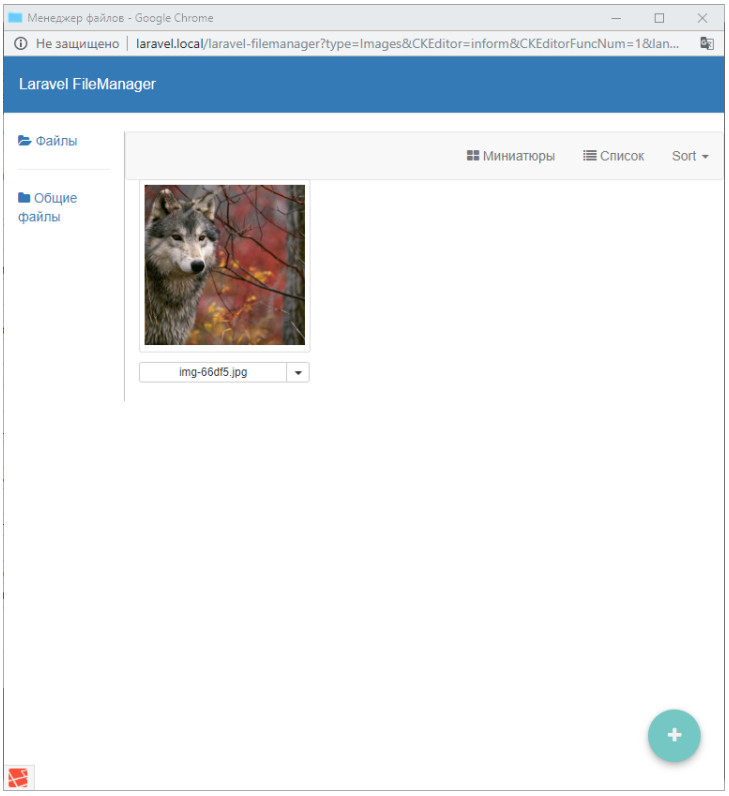
Нажмем на нее. Возможно, все будет хорошо. Если нет, увидим такой результат:



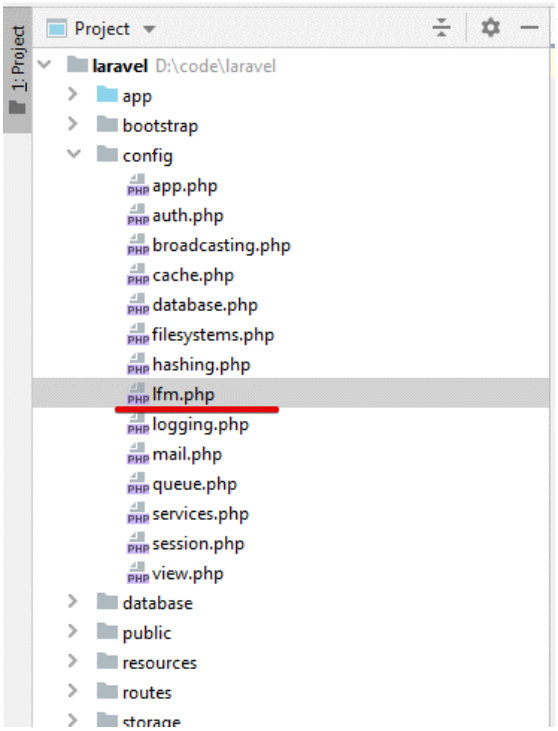
Из сообщения понятно, что чего-то не хватает. Добавим недостающий пакет командой **composer require laravel/helpers**

Затем нажмем еще раз на кнопку «Выбор на сервере».

Теперь менеджер файлов открылся.



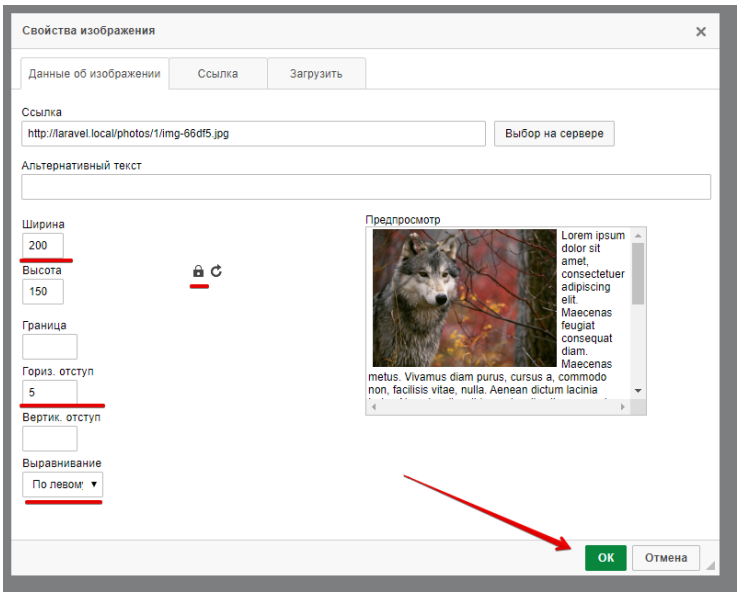
Настройки работы этого файлового менеджера формируются в файле **lfm.php.** Он появился у нас в проекте после выполнения команды публикации данных пакета. К нему вернемся позже.



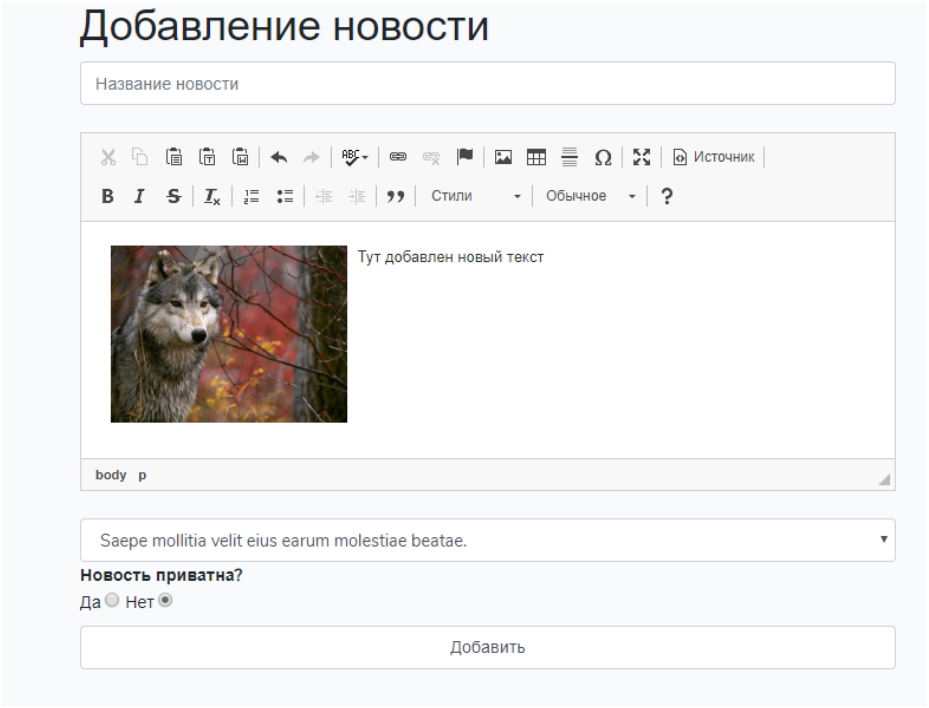
Добавим какие-нибудь файлы в нашу систему, используя файловый менеджер. После двойного щелчка по загруженной картинке окно с менеджером закроется — и пользователь вновь увидит модальное окно «Свойства изображения». В нем уже появилась выбранная нами картинка.

Это окно позволяет настроить параметры отображения изображения: ширину, высоту, отступы и другое. Меняя значения параметров здесь, мы корректируем значения атрибутов в html-разметке.

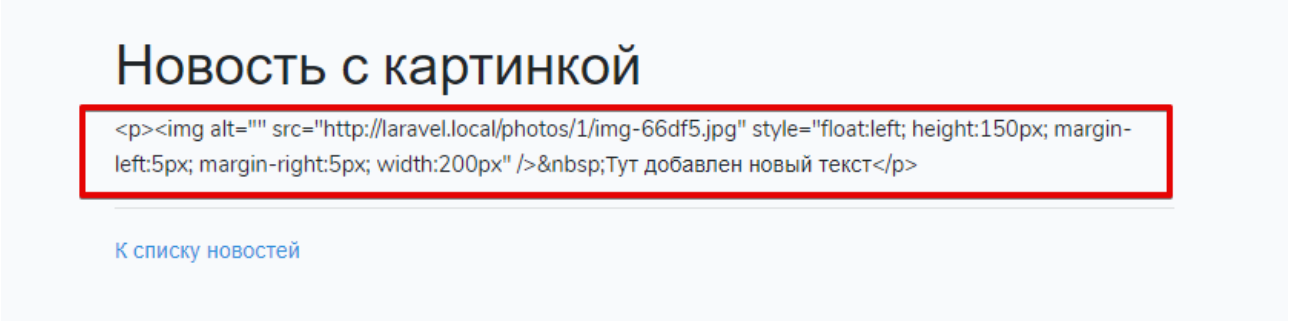
Указав необходимые значения, жмем на OK.



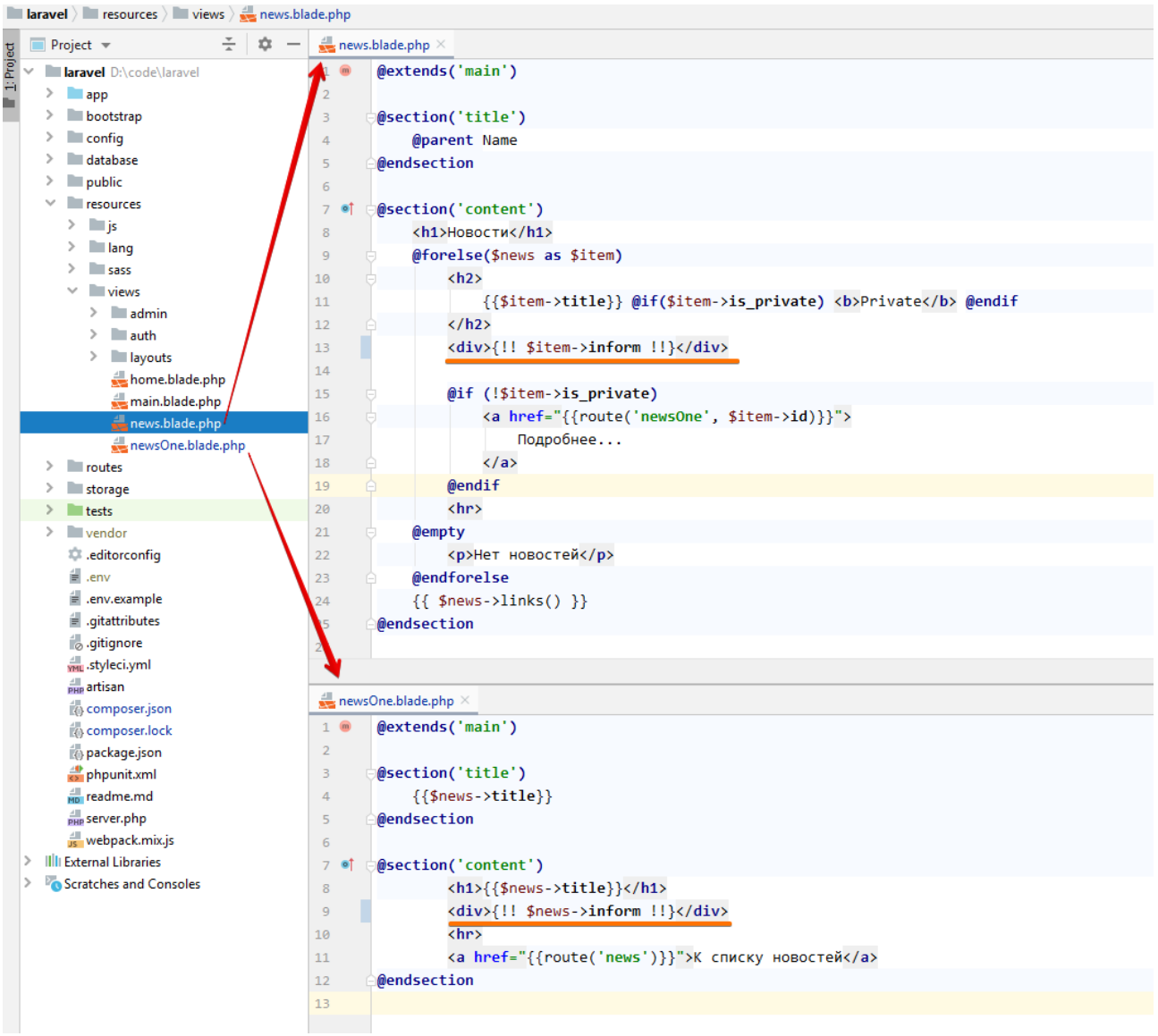
Далее в редакторе можно изменить текст, установить картинку в нужное место. Впишем название новости и укажем нужную категорию. Затем нажмем «Добавить».



Перейдем на страницу, где выводится новость.



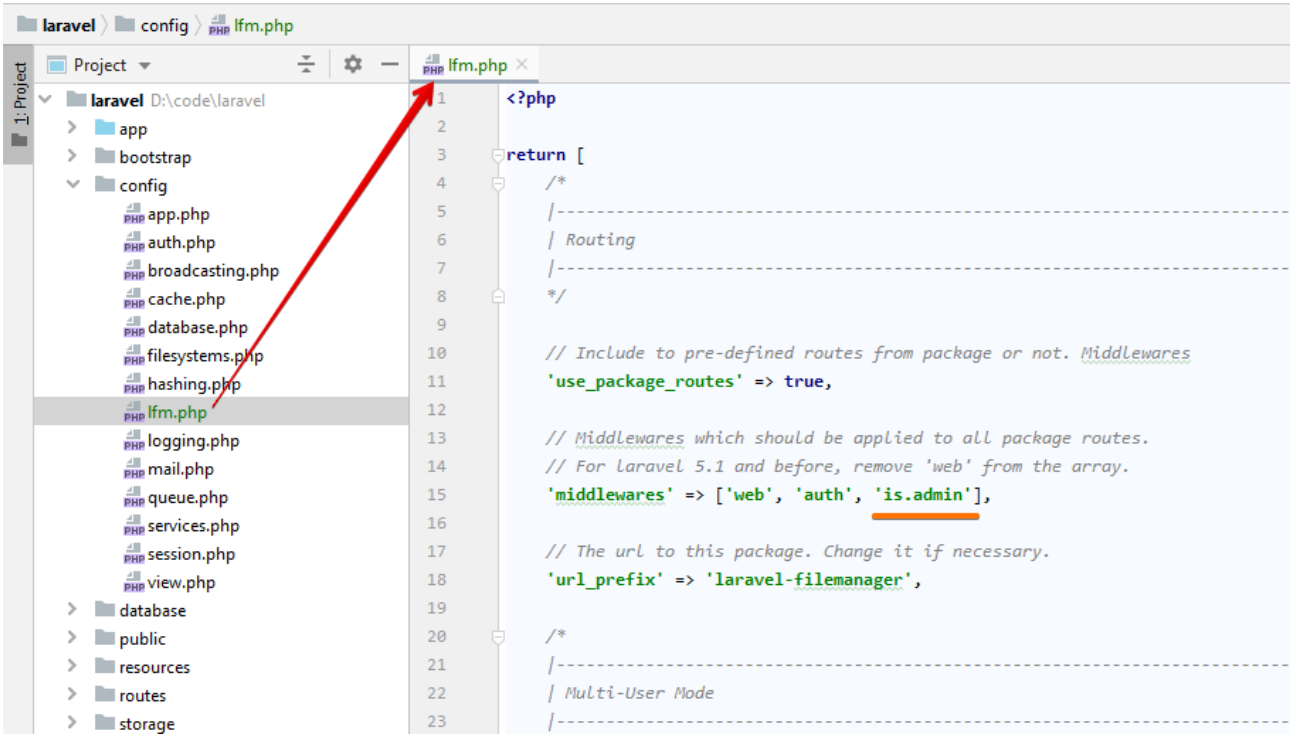
Что это? Все просто — по умолчанию при выводе контента шаблонизатор показывает все содержимое как есть. Чтобы изменить вывод, внесем корректировку в шаблоны.



Есть еще одна проблема. Если вы как администратор, находясь в системе, скопируете ссылку, которая указывается в окне с файловым менеджером, а затем залогинитесь через социальную сеть — то обнаружите, что эта ссылка по-прежнему открывает доступ к менеджеру файлов. Это происходит потому, что для роутов менеджера не установлен посредник **is.admin**.

Как его установить?

В файле **lfm.php** хранятся настройки файлового менеджера. Откроем этот файл — и сразу поймем, куда и что надо добавить.



Очереди

Пользователь, открывая страницу сайта, получает информацию, ранее сохраненную для него из базы данных. Эта операция происходит достаточно быстро, и все остаются довольны.

Но данные могут очень быстро меняться — например, данные о свободных местах в самолете. Сохранить заранее и поддерживать их актуальность трудно: авиакомпаний, рейсов и самолетов много.

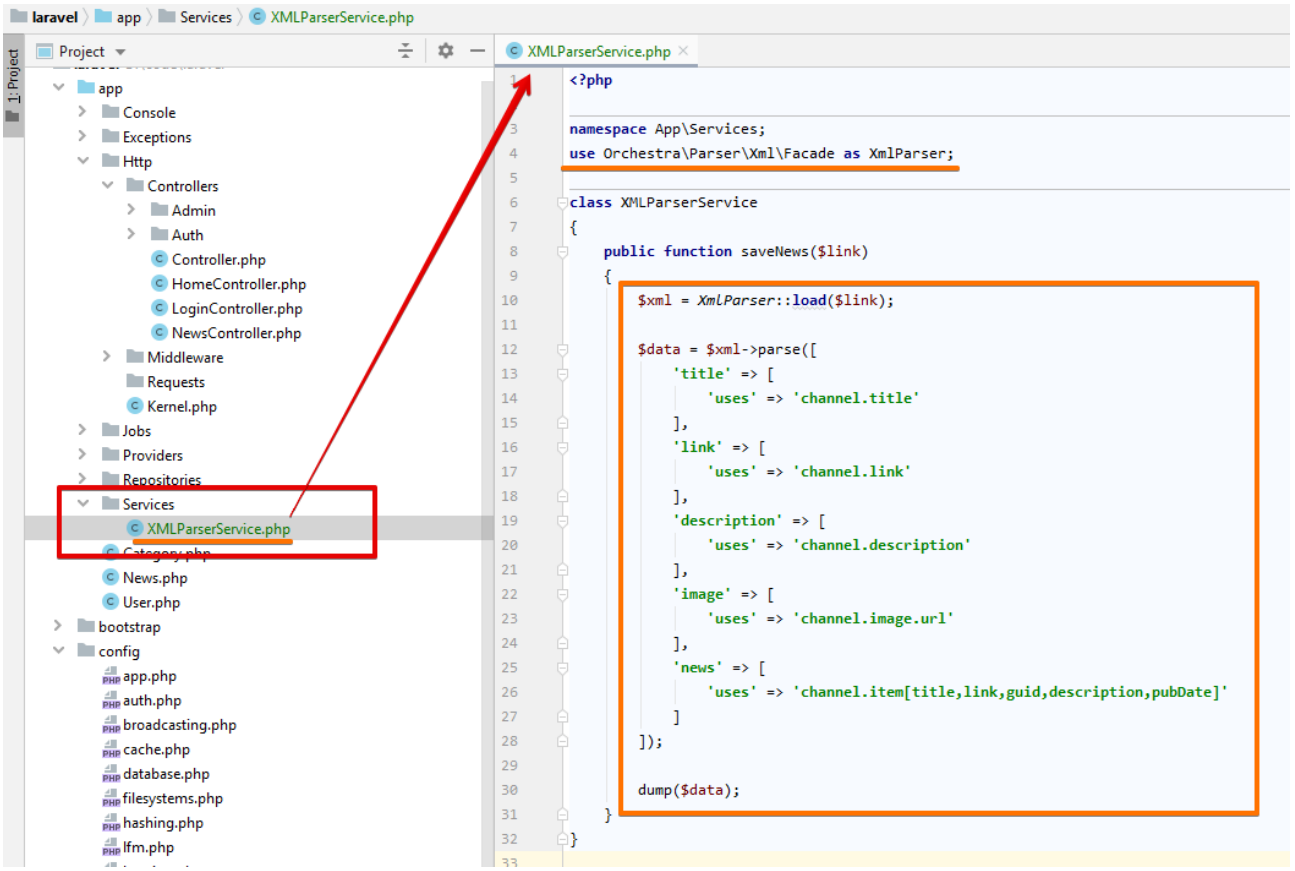
Такие данные обычно получают из сервисов — агрегаторов, которые тоже опрашивают другие ресурсы. Все это требует времени. Один такой запрос проходит относительно быстро. А если пользователь захочет получить информацию сразу из нескольких источников?

Один из комфортных для пользователя вариантов может проходить по такому алгоритму:

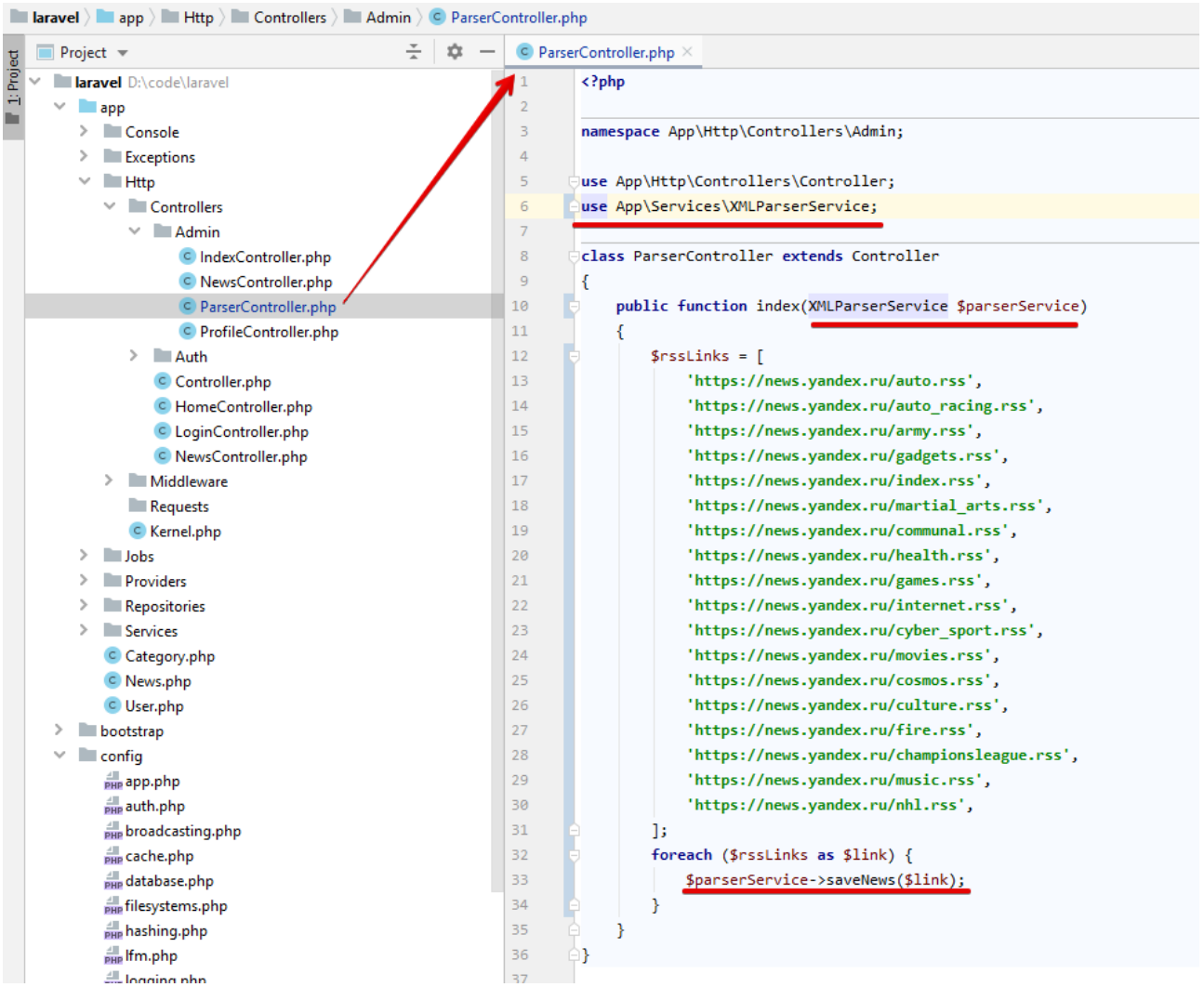
1. пользователь делает в браузере запрос на поиск билетов, указывая необходимые для него сервисы.
2. Запрос на сервере обрабатывается. В базу (в очередь) записывается информация о ресурсах, с которых надо получить ответ и, возможно, дополнительные параметры.
3. Пользователю выводится страница с информацией о том, что происходит с запросами к сервисам.
4. В это же время фоновые программы (воркеры) видят, что в очереди (в базе) появились новые задачи. Они начинают их разбирать, запуская парсинги с указанными параметрами.
5. По мере получения ответов из сторонних сервисов данные передаются пользователю.

На прошлом уроке мы создали парсер, собирающий информацию из «Яндекс.Новостей». Алгоритм парсера скрыт за фасадом **Orchestra\Parser\Xml\Facade**, которому задан псевдоним **XmlParser**. Тем

не менее кода у нас в контроллере много. Вынесем его в отдельный метод сервиса **XMLParserService**.



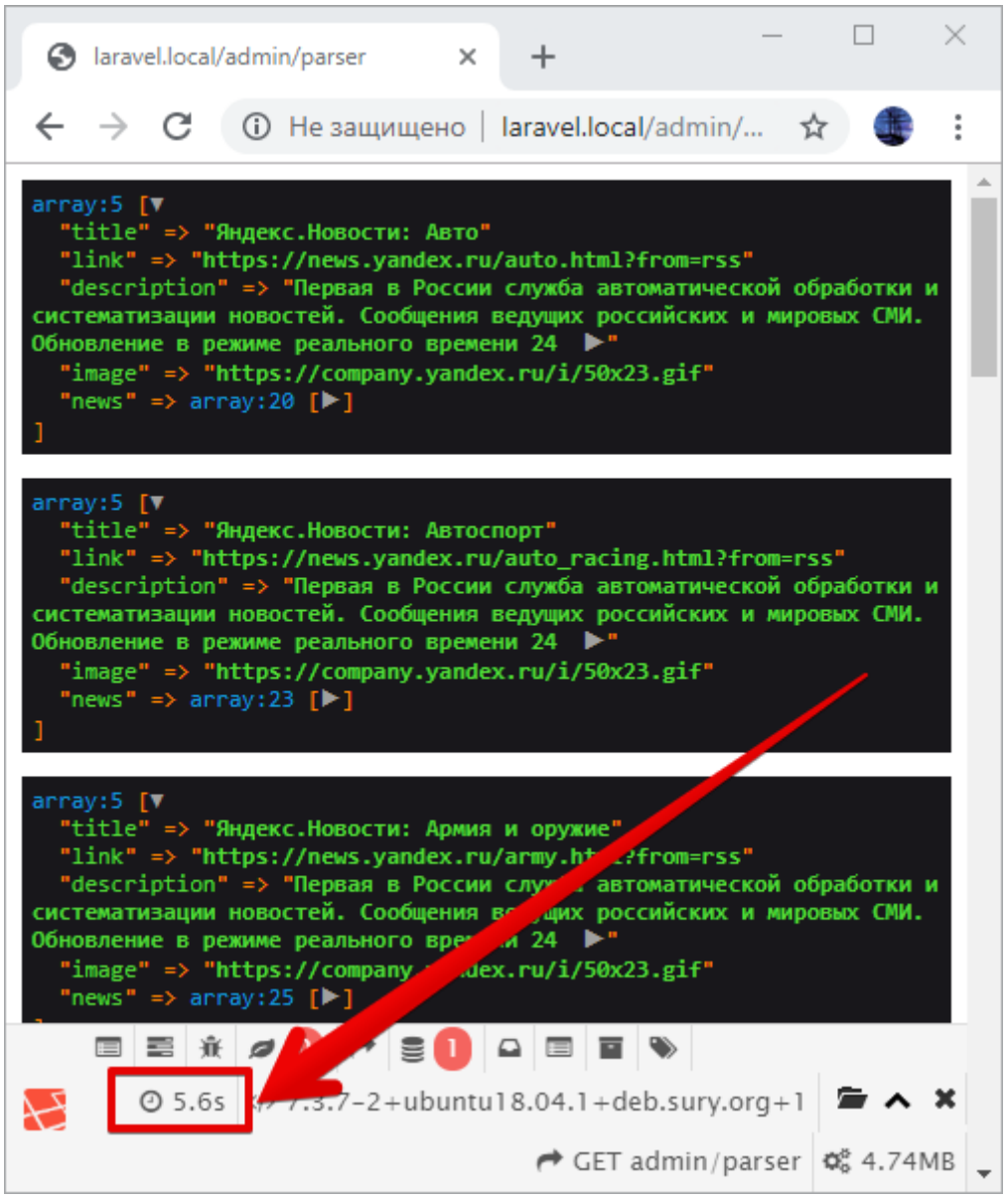
Укажем, что первым параметром в методе **index** класса **ParserController** ожидаем экземпляр данного сервиса — и Laravel сам будет его создавать. В контроллере добавим массив с ссылками на другие категории новостей и при помощи цикла получим данные.



Вот список адресов:

```
'https://news.yandex.ru/auto.rss',
'https://news.yandex.ru/auto_racing.rss',
'https://news.yandex.ru/army.rss',
'https://news.yandex.ru/gadgets.rss',
'https://news.yandex.ru/index.rss',
'https://news.yandex.ru/martial_arts.rss',
'https://news.yandex.ru/communal.rss',
'https://news.yandex.ru/health.rss',
'https://news.yandex.ru/games.rss',
'https://news.yandex.ru/internet.rss',
'https://news.yandex.ru/cyber_sport.rss',
'https://news.yandex.ru/movies.rss',
'https://news.yandex.ru/cosmos.rss',
'https://news.yandex.ru/culture.rss',
'https://news.yandex.ru/fire.rss',
'https://news.yandex.ru/championsleague.rss',
'https://news.yandex.ru/music.rss',
'https://news.yandex.ru/nhl.rss',
```

Результат выполнения:



Все работает, но есть нюанс.

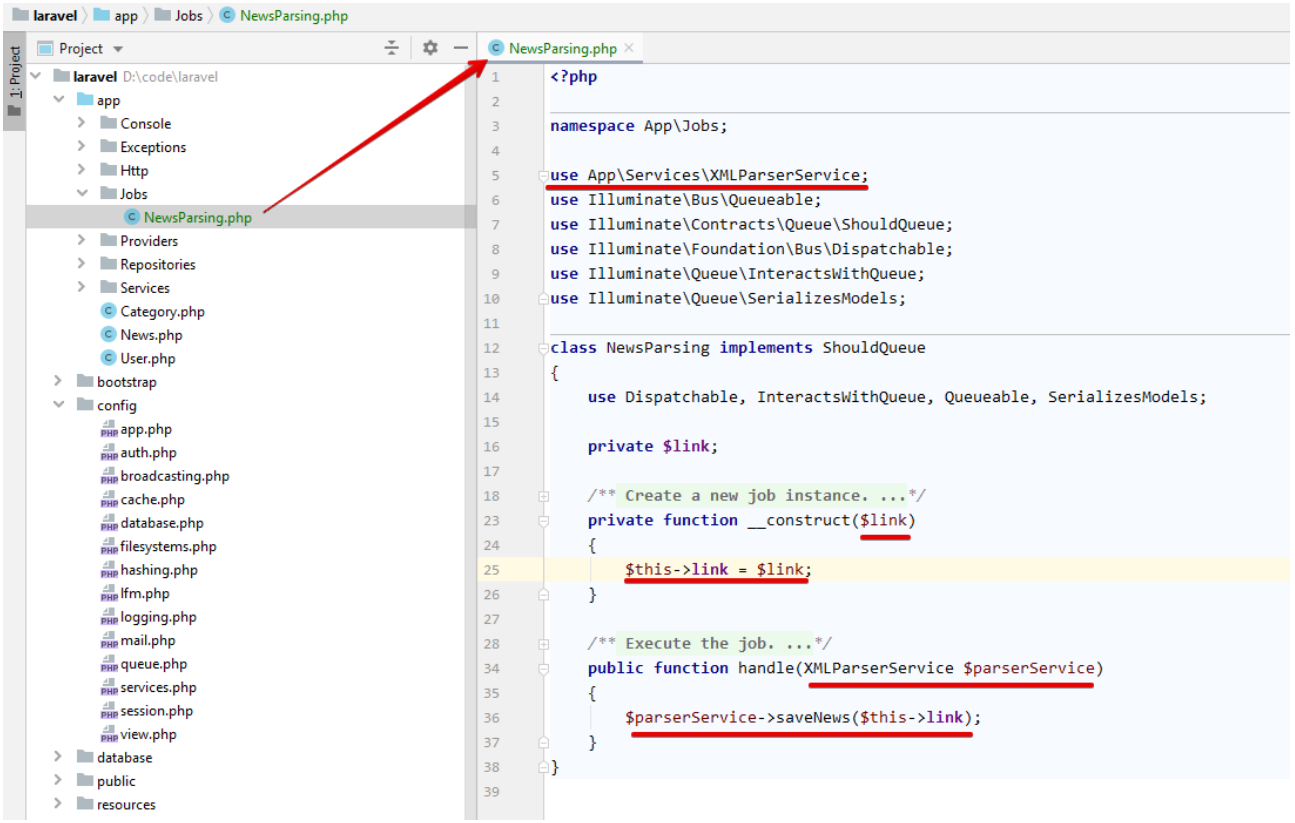
А если данные с сервиса будут возвращаться также долго, как из сервиса VK?

А если запросов будет больше сотни?

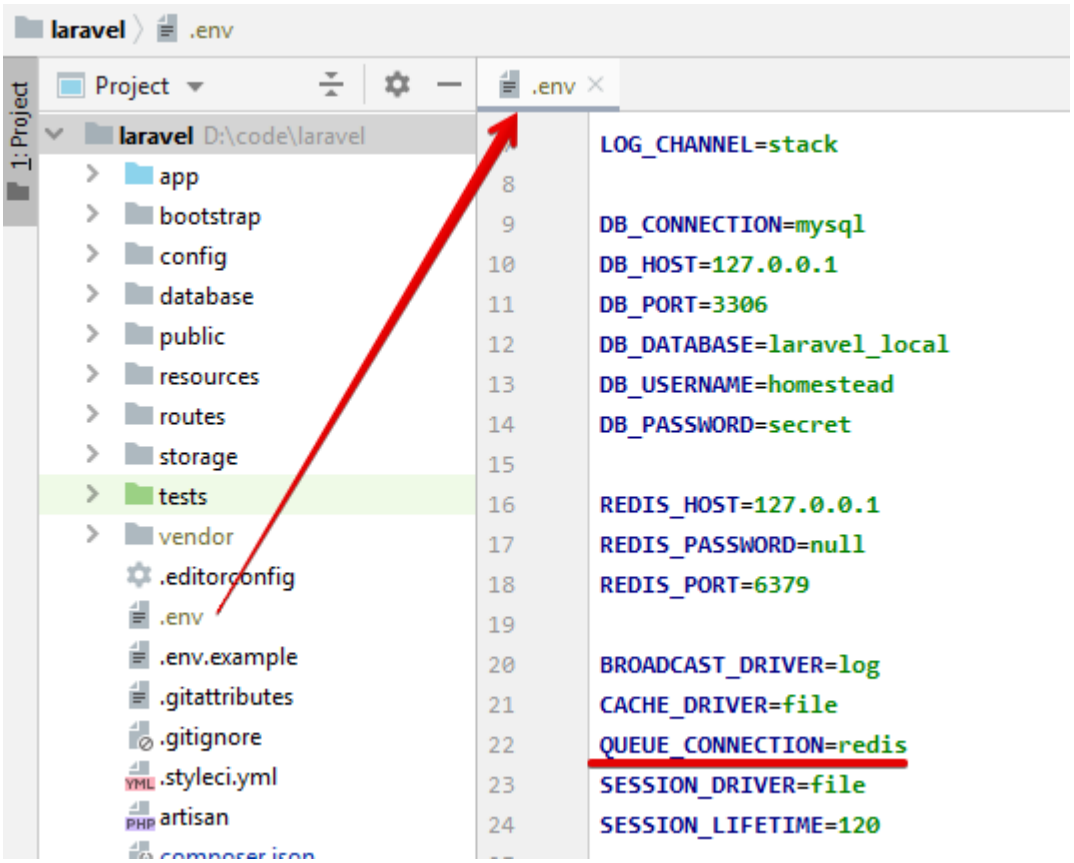
Пользователь уйдет на другой ресурс. Конечно, запускать парсинг можно в ночное время по расписанию. Но насколько новости будут актуальны?

Чтобы ускорить этот процесс, воспользуемся встроенной функциональностью Laravel — очередями. Сначала создадим шаблон задачи, которую будем отправлять в очередь на выполнение. Команда для этого: **php artisan make:job NewsParsing**

Этот шаблон будет получать единственный параметр — ссылку на ресурс, с которого следует получить информацию. После старта выполнения он будет обращаться к сервису **XMLParserService**, его методу **saveNews**, в который и будет передаваться ссылка.



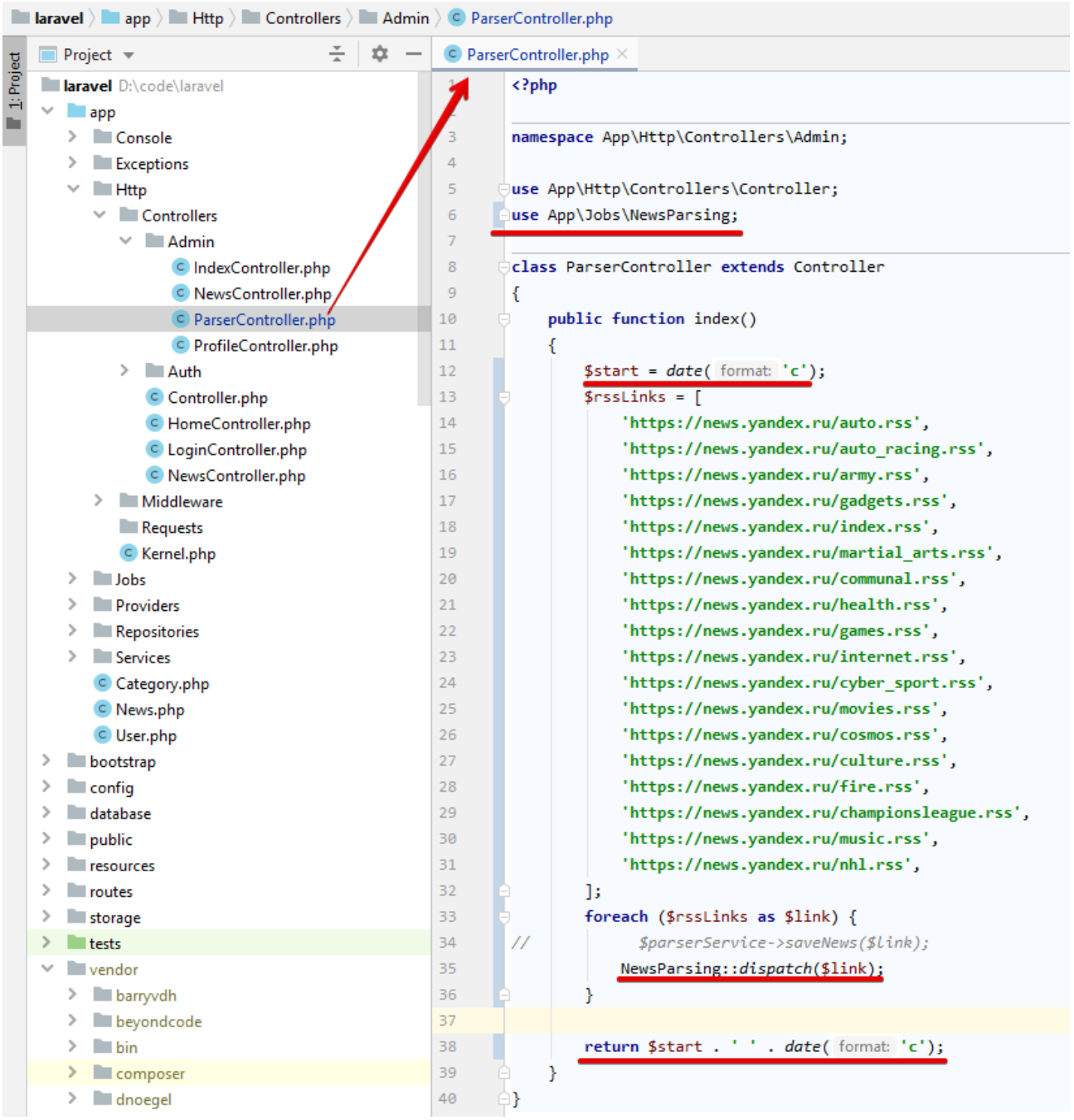
Данные очереди будем хранить в **Redis**. Для этого зайдем в файл **env** и для параметра **QUEUE_CONNECTION** установим значение **redis**.



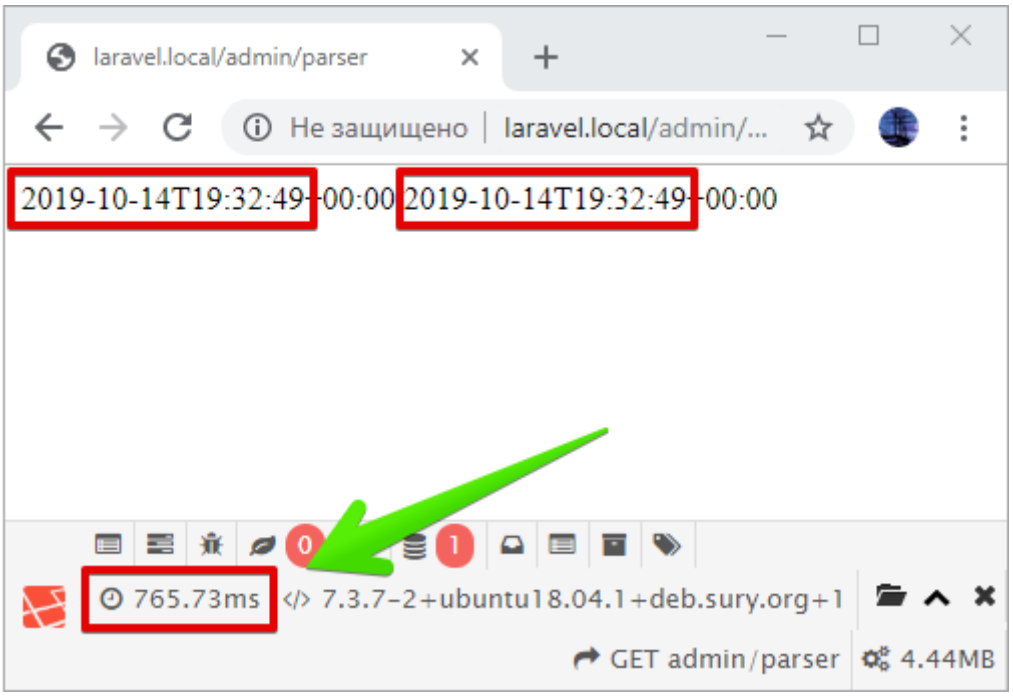
Затем скачаем специальный пакет, который поможет подружить Laravel с redis, установив для фреймворка клиент **redis**. Воспользуемся командой **composer require predis/predis**

Чтобы поставить задачу в очередь, заменим в контроллере обращение к сервису на вызов статического метода **dispatch** у **NewsParsing**. В качестве параметра передадим ссылку на ресурс.

А еще добавим вывод времени в начале и конце выполнения метода **index**, чтобы оценить продолжительность выполнения запроса.



Запускаем:

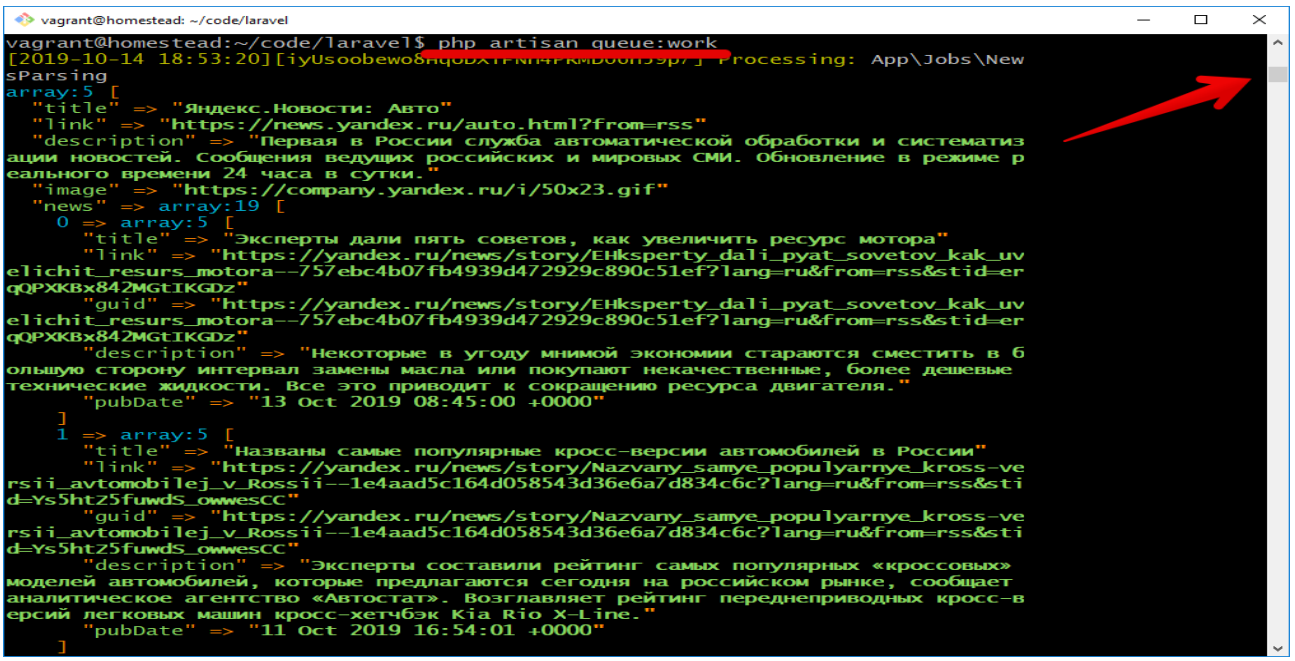


Запрос отработал меньше чем за секунду.

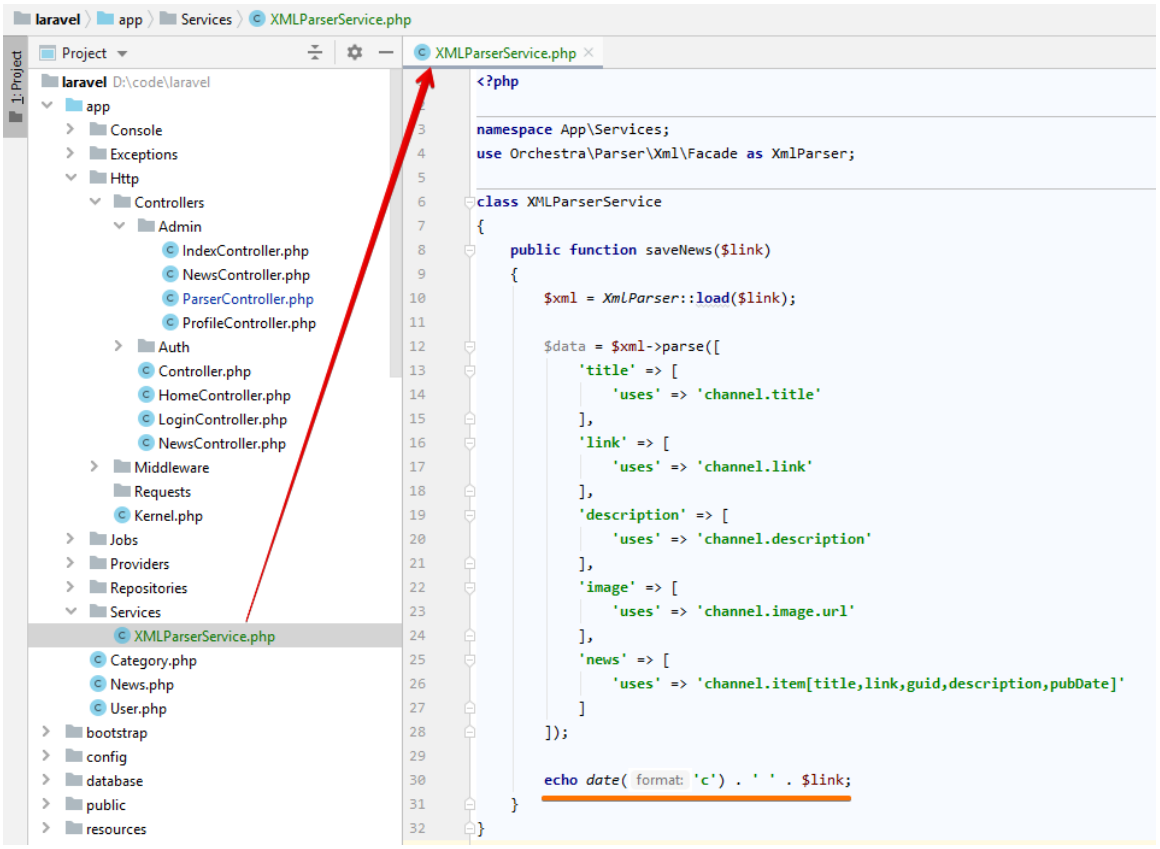
Почему такая разница по сравнению с предыдущим запуском? Все просто — данный запуск не выполнил ни одного парсинга, а только добавил записи в **redis**: название класса задачи, которую необходимо выполнить, и данные для него. То есть каждый раз, выполняя **NewsParsing::*dispatch*($link);**, Laravel добавлял новую запись в **redis**. В ней он указывал класс **NewsParsing** и значение параметра **$link**.

Как запустить выполнение задач из очереди?

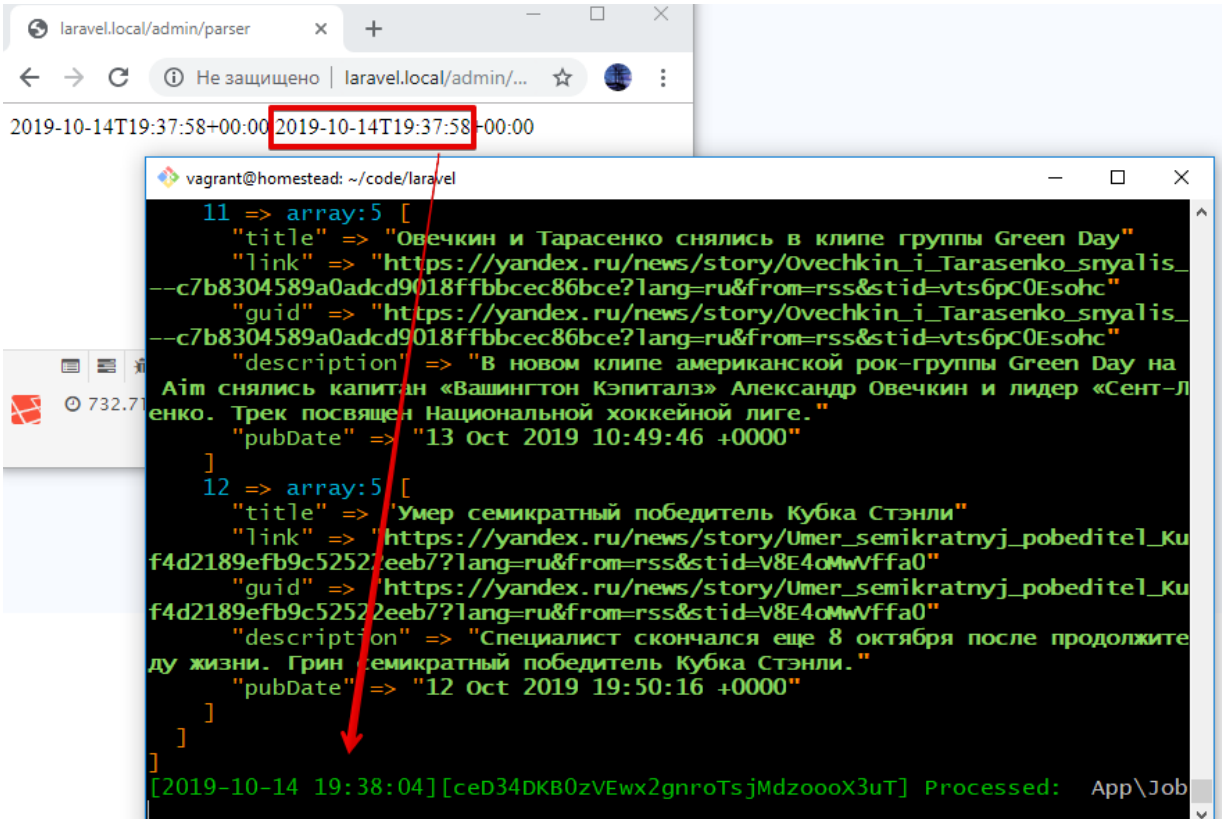
Потребуется демон или воркер — тот скрипт, который будет висеть в фоновом режиме и обрабатывать очередь. Создадим его командой **php artisan queue:work**



В консоли много информации — сократим ее:



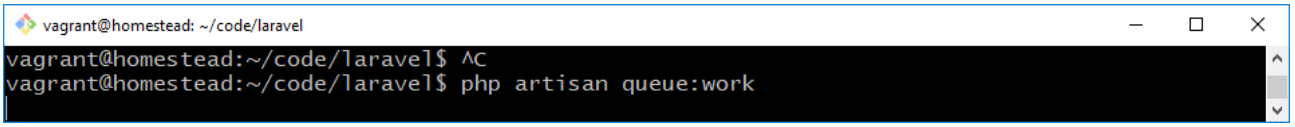
Еще раз вернемся в браузер и обновим страницу, а после этого откроем консоль.



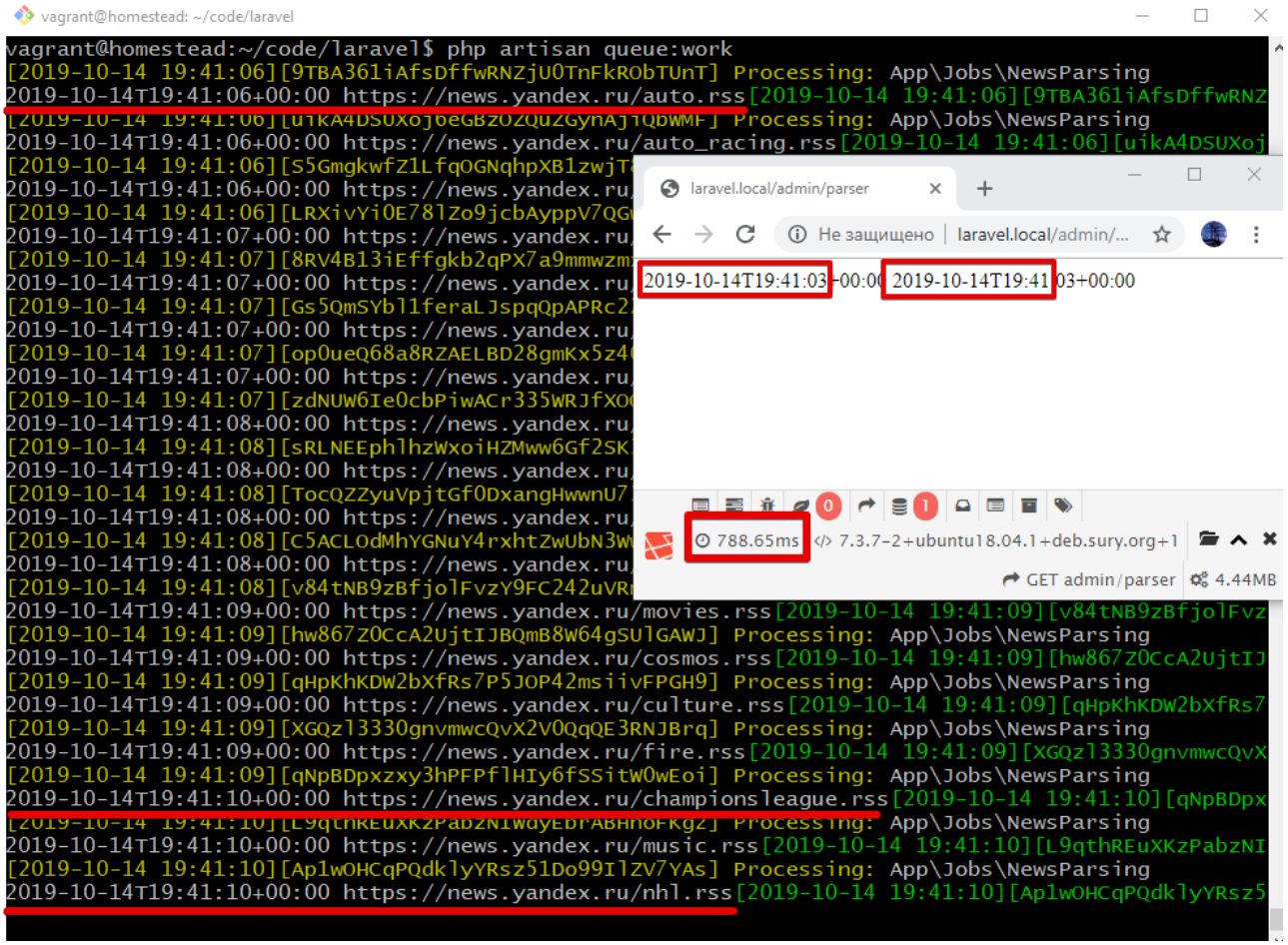
Воркер сработал — задачи выполнились. Но вывод не изменился.

Дело вот в чем: запуская воркер, программа сохраняется в оперативной памяти — и обращения к файлам не происходит. Чтобы воркер изменился, его надо перезапустить.

Сначала остановим его, нажав на клавиатуре сочетание клавиш Ctrl+C, а затем еще раз запустим.



И опять обновим страницу в браузере. Отметим, что вывод сообщений в консоли изменился.

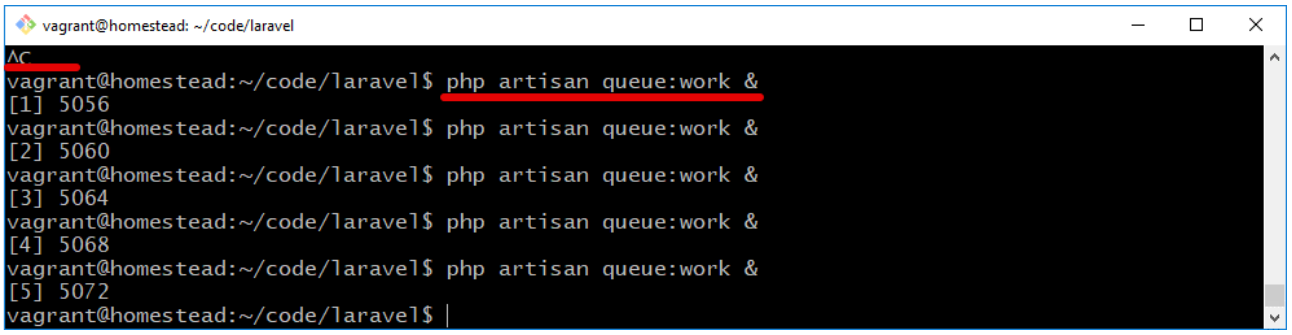


Был запущен только один воркер, поэтому все задачи в очереди выполняются последовательно. Такой вариант работы очереди удобен, например, при загрузке тяжелых файлов на сервер, отправке сообщений.

Для параллельного запуска задач надо запустить несколько воркеров. Их количество воркеров зависит от числа задач, которым предстоит запускаться одновременно. Если задач больше, чем запущенных воркеров, то воркеры, выполнившие задачу, будут забирать следующие, пока не кончатся задачи или их не остановят.

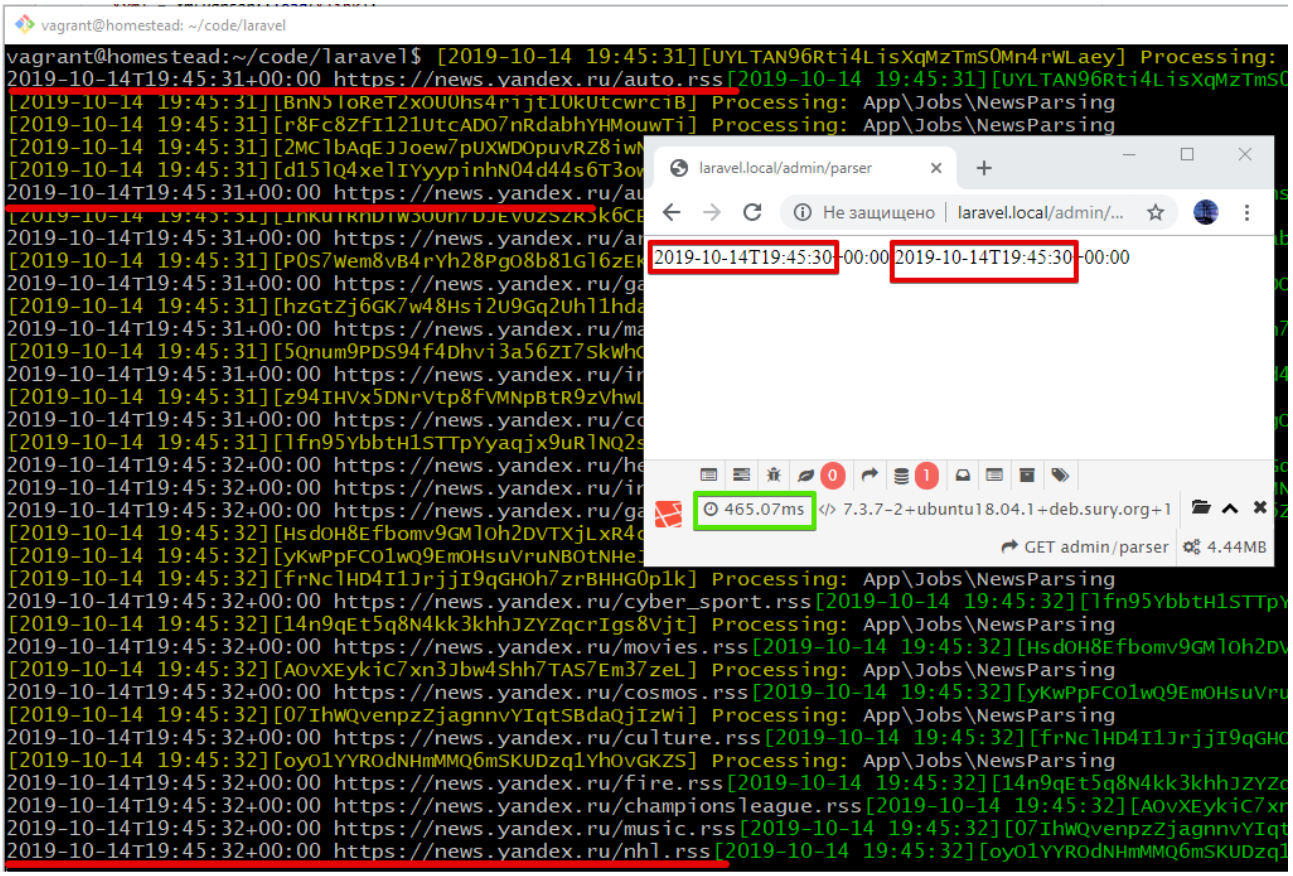
Остановим запущенный воркер.

Команда для запуска нескольких воркеров: **php artisan queue:work &**



Эта команда отличается от предыдущей одним символом, который запускает воркер в фоновом режиме. Значит, есть возможность запустить и другие. Повторим команду 20 раз. Когда начнется запись задач в очередь, воркеры будут принимать по одной и запустятся параллельно (линков у нас меньше 20).

Перейдем в браузер и обновим страницу.



Все задачи выполнились примерно за одну секунду — против первоначальных 5,6 секунд.

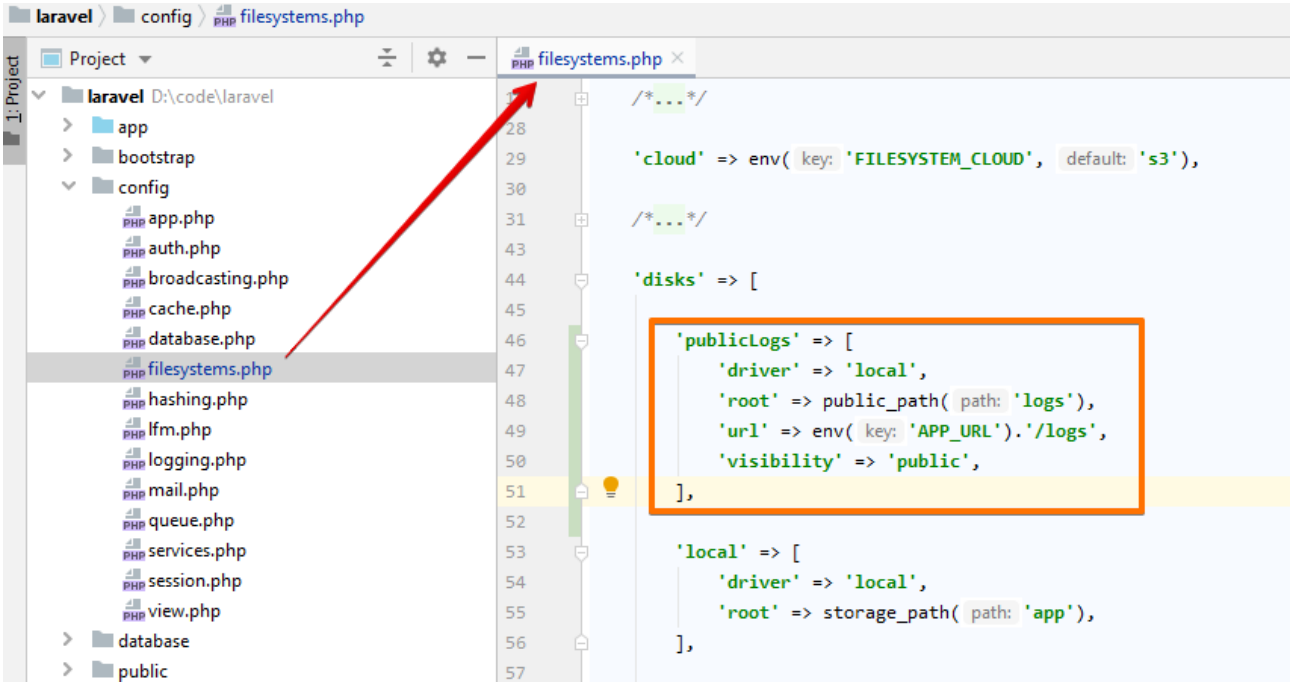
Сохранение файлов

Результат выполнения парсинга правильнее сохранять в базу, но мы сохраним его в файлы. Работая с Laravel, вы можете использовать стандартные функции PHP для работы с файлами, но удобнее пользоваться встроенной функциональностью фреймворка.

В Laravel предустановлен пакет **Flysystem**, который позволяет работать с файловой системой — как локальной, так и удаленной. Настройки файловой системы находятся в файле **filesystems.php**. В нем можно настроить файловые системы — диски. Каждый диск представляет определенный драйвер и

место хранения. В конфигурационном файле есть примеры для каждого поддерживаемого драйвера. Количество дисков ничем не ограничено — есть возможность применять несколько дисков, которые используют один драйвер.

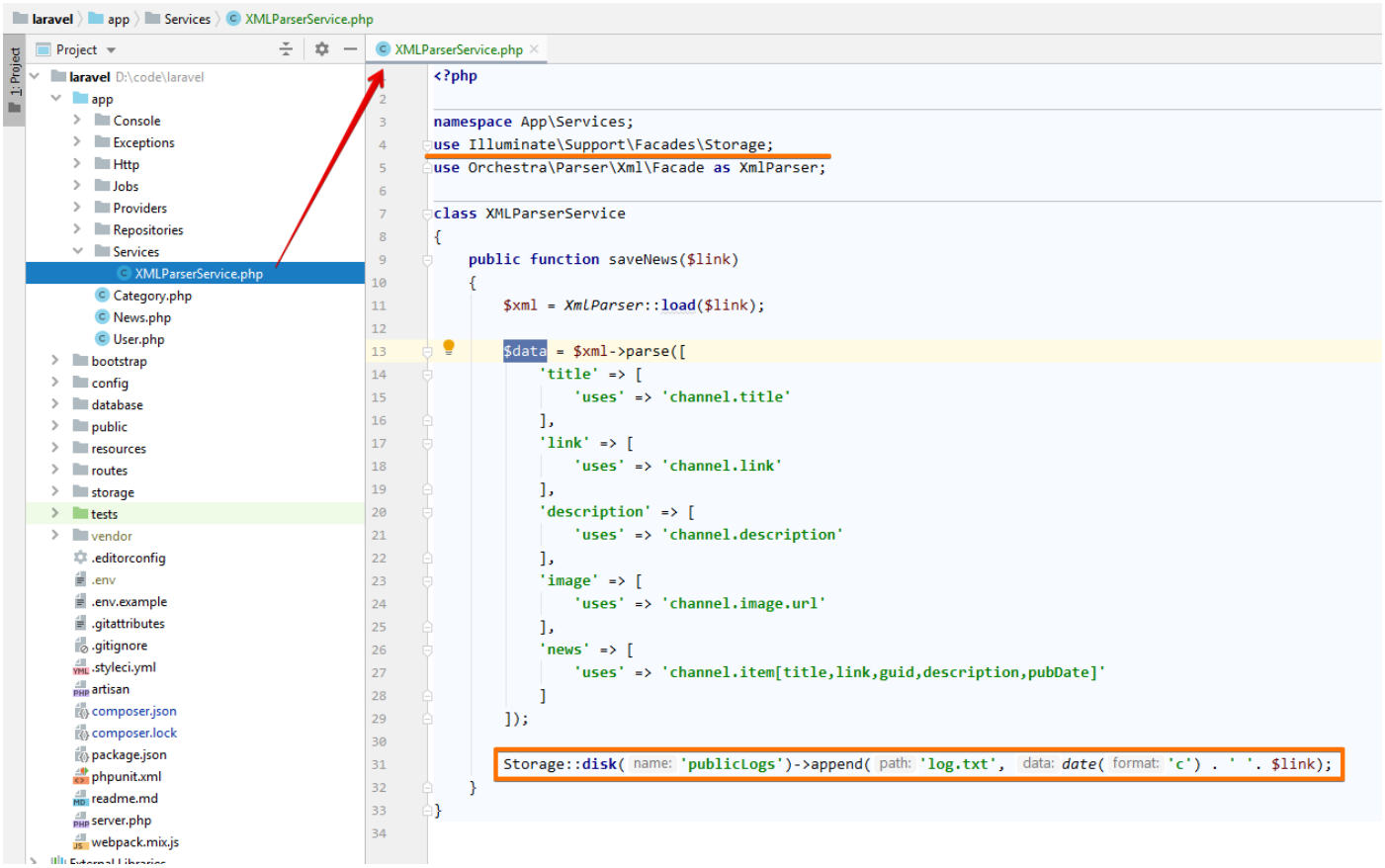
В файле **filesystems.php** хранятся настройки для работы с файловыми системами (подробнее — https://laravel.ru/docs/v5/filesystem). Добавим свою настройку.



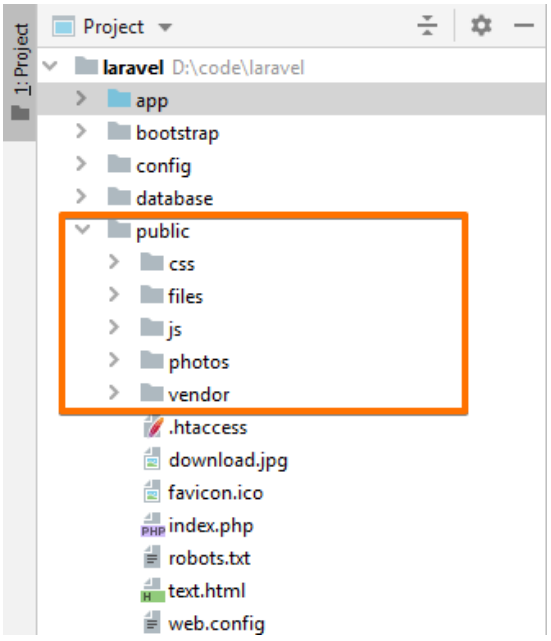
Эта настройка указывает, что в системе появился новый диск. А также показывает, какой драйвер для него следует использовать, как получить файлы из веб-интерфейса и где они должны физически сохраняться.

Для работы с файлами используется фасад **Illuminate\Support\Facades\Storage**. Статический метод **disk** данного фасада принимает необязательный параметр — название диска, с которым будет работать. Если он не указан — используется тот, который назначен по умолчанию в файле **filesystems.php**. В результате выполнения этого метода возвращается экземпляр класса **Illuminate\Filesystem\FilesystemAdapter**. Он содержит методы для работы с файлами: удаление, сохранение, запись, проверка на существование и другие.

Реализуем сохранение ссылки, переданной в сервис, в файле **log.txt** внутри созданного «диска».



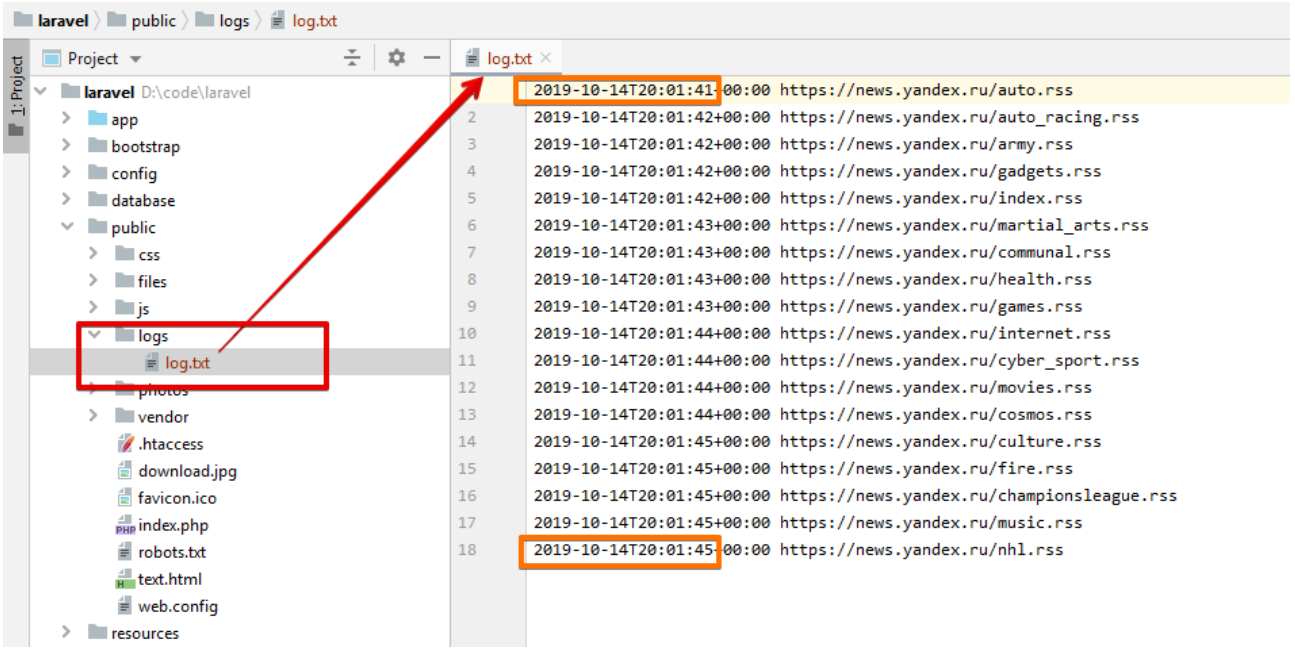
Перед запуском воркеров отметим структуру директории в папке **public:**



Остановим все воркеры и запустим только один.

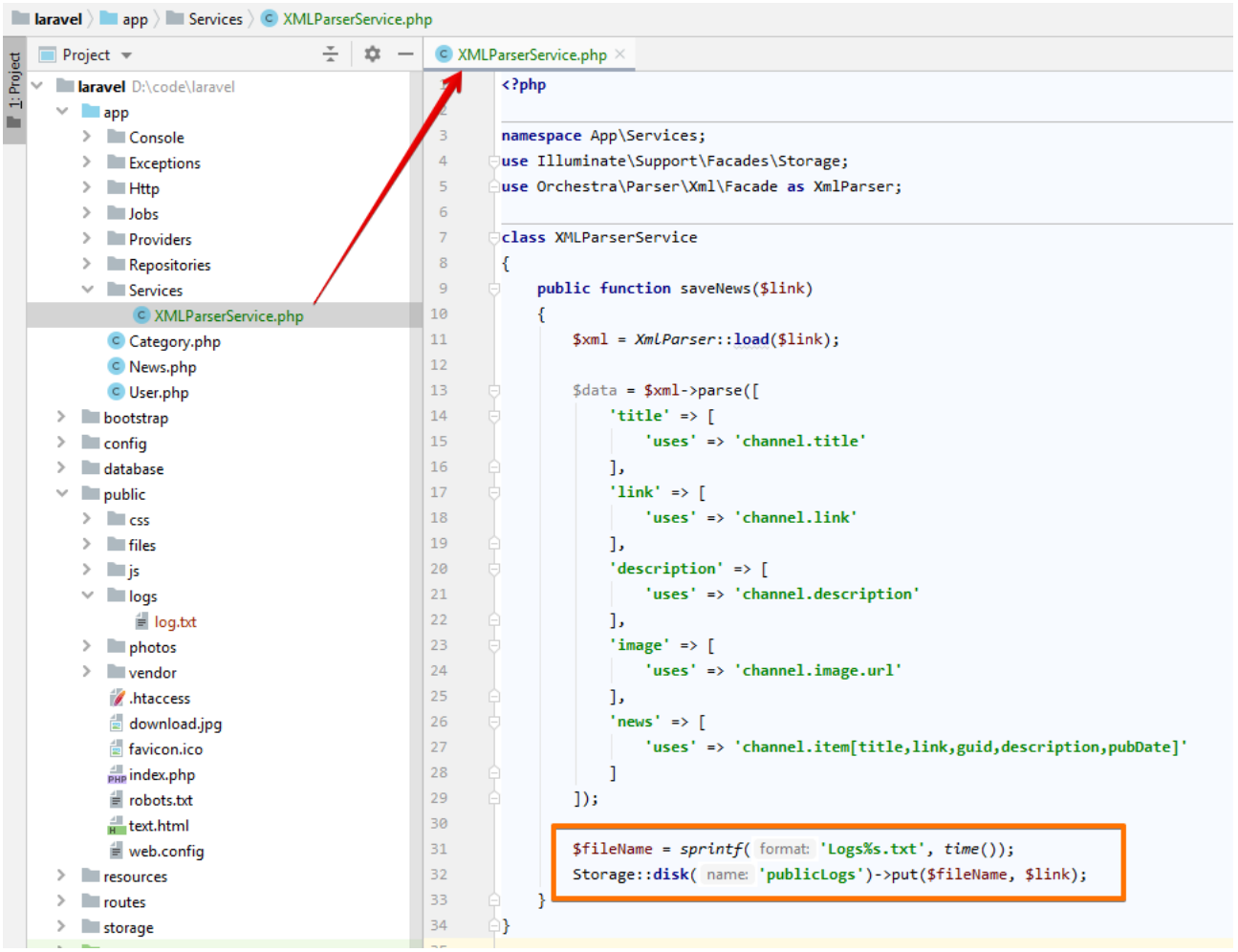
- **php artisan queue:restart**
- **php artisan queue:work**

Обновим страницу в браузере. Дождемся момента, когда воркер обработает все задачи, и еще раз заглянем в папку **public**.

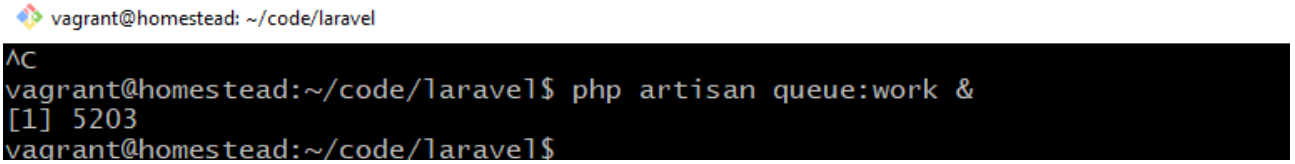


Отлично! Папка **logs** была создана, а в ней — добавлен необходимый файл, который содержит список ссылок.

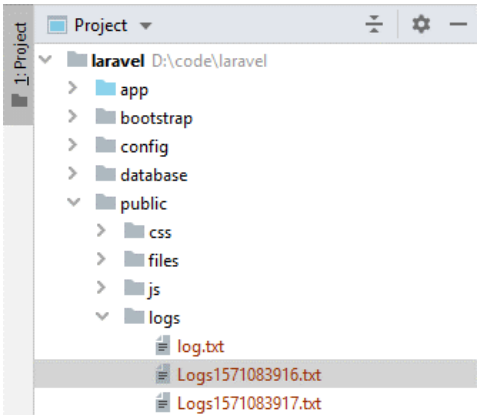
Теперь попробуем сохранять файлы с именем, содержащим в себе временную метку.



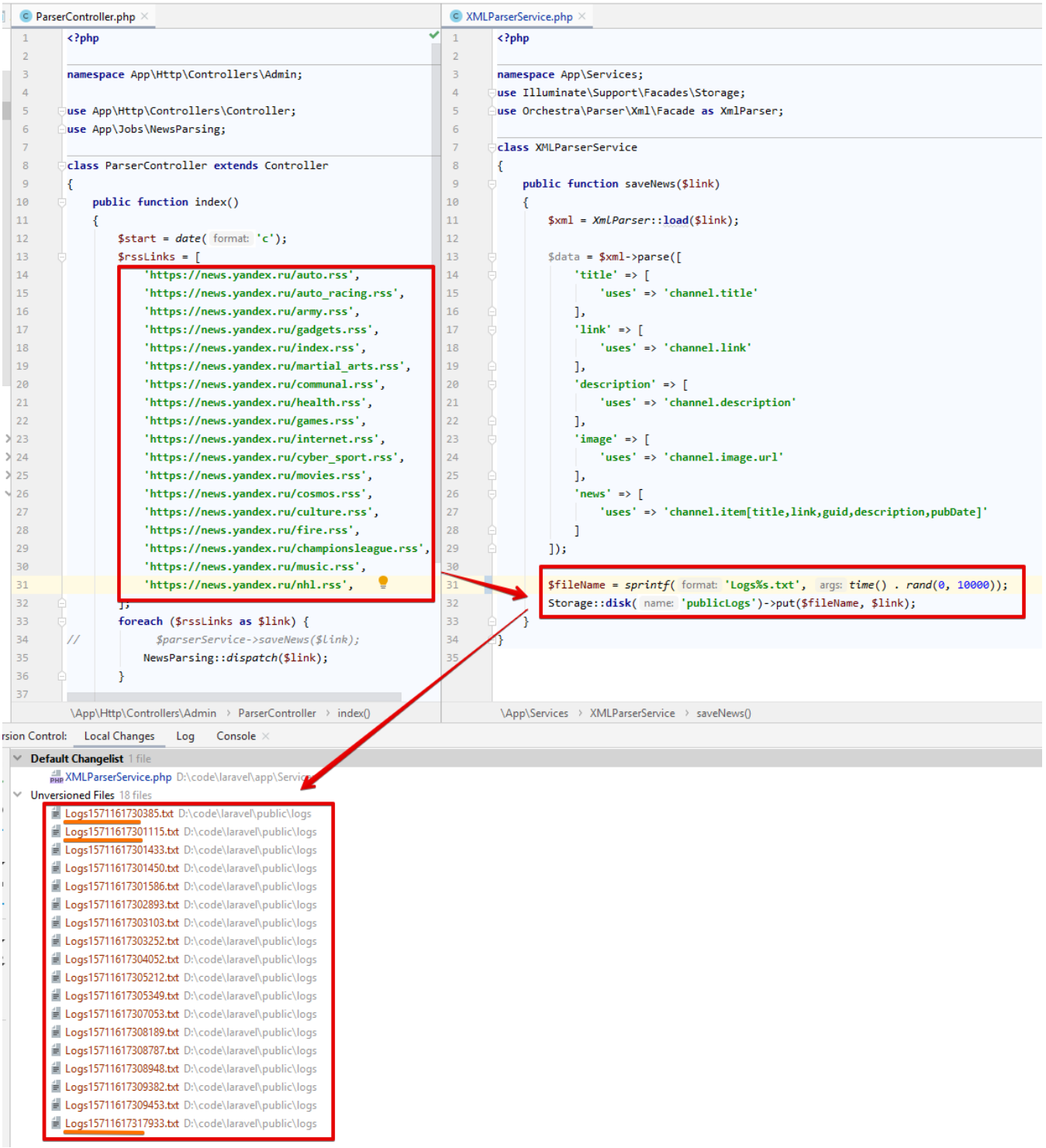
Перезапустим воркеры и запустим их в количестве 20 штук. Обновим страницу браузера.



Поскольку воркеры работали практически параллельно, у нас создалось всего два файла.



Внесем небольшой рандом в имена файлов и перезапустим воркеры. Обновим страницу браузера.



Практическое задание

1. Оптимизировать использование html-редактора, чтобы его исходники хранились в системе.

2. Создать миграцию для добавления в базу новой таблицы **resources**. Она будет хранить информацию о ресурсах, с которых необходимо забирать новости. Добавить интерфейс для редактирования и добавления данных в эту таблицу.

3. Реализовать алгоритм получения новых новостей из ресурсов, сохраненных в таблице **resources**, с добавлением нужной информации в таблицу **news**.
4. Используя очереди, реализовать алгоритм параллельного запроса информации из сторонних сервисов с выводом ее пользователю в браузер.

Дополнительные материалы

1. https://laravel.com/docs/5.8/queues
2. https://laravel.com/docs/5.8/horizon
3. https://laravel.com/docs/5.8/filesystem
4. https://ckeditor.com/ckeditor-4/

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. http://laravel.su/
2. https://laravel.com/docs/6.x/queues/