EVENT-DRIVEN PHP

REACTPHP ДЛЯ НАЧИНАЮЩИХ

ЖУК СЕРГЕЙ

Руководство для новичков по созданию асинхронных приложений

ReactPHP для начинающих

Sergey Zhuk

Эта книга предназначена для продажи на http://leanpub.com/reactphp-for-beginners-ru

Эта версия была опубликована на 2020-07-31

Это книга с Leanpub book. Leanpub позволяет авторам и издателям участвовать в так называемом Lean Publishing - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

Оглавление

Обработка POST-запросов

На нашей главной странице сейчас есть совсем простая форма: текстовое поле, которое можно заполнить и отправить на сервер с POST-запросом. Вот HTML-код этой формы:

```
<form action="/upload"
      method="POST"
      class="justify-content-center">
  <div class="form-group">
    <label for="text">Текстовое сообщение</label>
    <textarea name="text"
              id="text"
              rows="3"
              cols="60"
              class="form-control">
    </textarea>
  </div>
  <button type="submit" class="btn btn-primary">
    Отправить
  </button>
</form>
```

Когда запрос приходит на сервер, он сначала обрабатывается им самим, а потом уже попадает в колбэк сервера. Сервер парсит тело запроса, поэтому у нас есть доступ к отдельным полям POST-данных, в нашем случае, к полю `text`.

Давайте откроем файл `routes.php` и изменим обработчик маршрута `/upload`:

```
// routes.php

'/upload' => function (ServerRequestInterface $request) {
  $text = $request->getParsedBody()['text'];

  return new Response(
    200,
    ['Content-Type' => 'text/plain'],
    "Введённое значение: $text"
  );
},
```

Распарсенные данные полей POST можно получить с помощью метода `getParsedBody()`. Он возвращает ассоциативный массив, ключами которого являются имена полей в отправленной форме. Давайте получим значение поля `text` и вернем ответ, который выведет то, что мы отправили вместе с запросом.

Перезапускаем сервер и попробуем что-нибудь отправить. Довольно просто, не так ли?

Загрузка файлов

Теперь давайте сделаем что-нибудь поинтереснее. Главная возможность нашего приложения — это загрузка файлов на сервер, а не отправка сообщений. Так что давайте поправим нашу форму и заменим текстовое поле на поле для загрузки файлов. Мы разрешим загрузку только определённых изображений, заодно обновим текст метки. И не забудьте добавить `enctype="multipart/form-data"` к тегу `form`, иначе форма не будет отправлять файлы:

```
<form action="/upload"
      method="post"
      class="justify-content-center"
      enctype="multipart/form-data">
  <div class="form-group">
    <label for="file">Загрузить файл:</label>
    <input name="file"
           id="file"
           type="file"
           accept="image/x-png,image/jpeg"/>
  </div>
  <button type="submit" class="btn btn-primary">
    Отправить
  </button>
</form>
```

Форма готова, теперь перейдем к обработчику запроса. Откроем `router.php` и для начала удалим всё внутри обработчика маршрута `/upload` — мы напишем новый с нуля:

```
// routes.php

return [
  // ...
  '/upload' => function (ServerRequestInterface $request) {

  },
];
```

Помните, что я говорил вам, что сервер сам делает некоторую обработку запросов? Так вот он не только парсит тело POST-запроса, но ещё также определяет были ли сделаны загрузки файлов. Метод `getUploadedFiles()` вернет все файлы, отправленные вместе с запросом, в виде ассоциативного массива, ключами которого будут имена полей из отправляемой формы. Нам нужно поле с именем `file`. Чтобы понять, куда двигаться дальше, давайте выведем переменную `$file` на консоль сервера и посмотрим, что в ней содержится. И не забудьте вернуть ответ из обработчика запроса. Просто любой ответ `200`, можно даже пустую строку, сейчас это не важно:

```
'/upload' => function (
  ServerRequestInterface $request,
  LoopInterface $loop
) {
  $files = $request->getUploadedFiles();
  $file = $files['file'];
  print_r($file);

  return new Response(
    200, ['Content-Type' => 'text/plain'], ''
  );
}
```

Перезапускаем сервер и в браузере открываем страницу `http://127.0.0.1:8080/`. Выбираем любой локальное изображение и отправляем форму. После чего мы будем перенаправлены на пустую страницу, где нет ничего интересного, поэтому нужно смотреть логи сервера. Возвращаемся к терминалу и изучаем вывод сервера. В моём случае получился вот такой вывод:

```
React\Http\Io\UploadedFile Object
(
    [stream:React\Http\Io\UploadedFile:private] => RingCentral\Psr7\Stream Object
        (
            [stream:RingCentral\Psr7\Stream:private] => Resource id #111
            [size:RingCentral\Psr7\Stream:private] =>
            [seekable:RingCentral\Psr7\Stream:private] => 1
            [readable:RingCentral\Psr7\Stream:private] => 1
            [writable:RingCentral\Psr7\Stream:private] => 1
            [uri:RingCentral\Psr7\Stream:private] => php://temp
            [customMetadata:RingCentral\Psr7\Stream:private] => Array
                (
                )

        )

    [size:React\Http\Io\UploadedFile:private] => 44939
    [error:React\Http\Io\UploadedFile:private] => 0
    [filename:React\Http\Io\UploadedFile:private] => p0517py6.jpg
    [mediaType:React\Http\Io\UploadedFile:private] => image/jpeg
)
```

Тут много всего, но самое главное будет наверху. Обратите внимание, мы вывели объект `React\Http\Io\UploadedFile`. Это означает что переменная `$file` это объект класса `React\Http\Io\UploadedFi`
Так, а что можно делать с этим объектом? Как его теперь сохранить на диск?

Метод `getUploadedFiles()` возвращает список загружаемых файлов в нормализованном виде, где каждый элемент списка это объект, реализующий `Psr\Http\Message\UploadedFileInterface`. Такое поведение описано[1] в PSR-7, стандарте, описывающий общие интерфейсы для представления HTTP-сообщений. Класс `React\Http\Io\UploadedFile` — это внутренняя реализация ReactPHP этого интерфейса. Ваш код не должен зависеть от этого класса, вместо него при работе с загрузками ориентируйтесь на `UploadedFileInterface`.

Сохранение загруженных файлов на диск

Как вы уже наверное догадались, мы не можем использовать функцию `file_put_contents()` и другие подобные блокирующие функции. В ситуациях, когда мы имеем дело с файловой системой, нам определенно нужен дочерний процесс, поэтому давайте его создадим. Но какую команду оболочки использовать?

В предыдущей главе мы уже познакомились с командой `cat`. Оказывается, что она и на этот раз нам подойдет. Почему? Помните что `cat` это сокращение от *конкатенация*

> *Конкатенация файлов или стандартного потока ввода на стандартный поток вывода.*

Это означает, что команда, по сути, даёт возможность получать данные из одного потока и записывать их в другой. В своем терминале попробуйте выполнить следующую команду:

```
cat > test.txt
```

Затем что-нибудь напечатайте, можно даже пару строк, как закончите, нажмите `Ctrl + C`, чтобы завершить команду. А теперь посмотрите в своей текущей директории и вы обнаружите файл с именем `test.txt`, внутри которого будет то, что вы только что вводили в терминале. Команда `cat` прочитала данные из потока стандартного ввода, а затем записала эти данные в файл `test.txt`. Удаляем этот файл, он нам больше не нужен.

Это и будет командой для нашего дочернего процесса. Единственное отличие только в том, что нам нужно указать реальное имя файла. Создайте папку `uploads` в корне вашего проекта. Все наши загруженные файлы будут сохраняться здесь. Для получения имени загруженного файла нужно вызвать метод `getClientFilename()` у объекта `Psr\Http\Message\UploadedFileInterface`.

[1] https://www.php-fig.org/psr/psr-7/#16-uploaded-files

```
// routes.php

'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  /** @var \Psr\Http\Message\UploadedFileInterface $file */
  $file = $request->getUploadedFiles()['file'];
  $process = new Process(
    "cat > uploads/{$file->getClientFilename()}", __DIR__
  );

  return new Response(
    200, ['Content-Type' => 'text/plain'], ''
  );
}
```

Помните, что нужно явно указать `__DIR__` в качестве рабочей директории дочернего процесса.

Следующим шагом будет запуск процесса с помощью метода `start()`, но как мы помним, этот метод требует объект цикла событий. Откуда нам его здесь взять? А поступим точно так же, как мы делали в обработчики для маршрута `/` — воспользуемся инъекцией зависимостей. Обновим объявление колбэка, добавив второй параметр `LoopInterface $loop`. Теперь имея доступ к объекту цикла событий, мы можем вызвать метод `start()` и запустить дочерний процесс:

```
'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  /** @var \Psr\Http\Message\UploadedFileInterface $file */
  $file = $request->getUploadedFiles()['file'];
  $process = new Process(
    "cat > uploads/{$file->getClientFilename()}", __DIR__
  );
  $process->start($loop);

  return new Response(
    200, ['Content-Type' => 'text/plain'], ''
  );
}
```

Обратите внимание, что в классе `Router` мы уже предоставляем всем обработчикам запросов объект цикла событий:

```
// src/Router.php

class Router
{
  // ...

  public function __invoke(ServerRequestInterface $request)
  {
    $path = $request->getUri()->getPath();
    echo "Запрос: $path\n";

    $handler = $this->routes[$path] ?? $this->notFound($path);

    return $handler($request, $this->loop);
  }
}
```

Последним шагом будет получение содержимого загруженного файла и отправка его в поток стандартного ввода нашего дочернего процесса. Помните, когда нам нужно было прочитать файл с диска и отправить его содержимое в качестве тела ответа, мы использовали поток STDOUT дочернего процесса? В этот раз наоборот нам нужно записать данные внутри дочернего процесса. Это означает, что нам нужен STDIN потом и соответствующее свойство дочернего процесса `stdin`.

У потока `stdin`, поскольку это поток записи данных, есть метод `write()`. Всё, что будет записано в него, будет отправлено прямо на стандартный поток ввода дочернего процесса. Как нам получить содержимое загруженного файла? Это очень просто, нужно просто вызвать следующую цепочку методов `$file->getStream()->getContents()`. Первый метод вернет поток, который представляет собой данные загруженного файла. Затем, чтобы получить эти данные нужно вызвать у потока метод `getContents()`:

```
// routes.php

'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  /** @var \Psr\Http\Message\UploadedFileInterface $file */
  $file = $request->getUploadedFiles()['file'];

  $process = new Process(
    "cat > uploads/{$file->getClientFilename()}", __DIR__
  );
  $process->start($loop);
  $process->stdin->write($file->getStream()->getContents());

  return new Response(
```

```
    200,
    ['Content-Type' => 'text/plain'],
    'Загрузка завершена'
  );
}
```

Вот и всё, обработчик маршрута `/upload` готов. Обратите внимание, что в ответе я добавил строку `'Загрузка завершена'` просто как индикатор того, что загрузка прошла успешно.

Наконец, перезагружаем сервер и теперь нам доступен функционал загрузки файлов. Попробуйте загрузить какое-нибудь изображение с диска и оно появится в директории `uploads` в корне проекта.

Баг с *зависшими* процессами

Взгляните ещё раз на обработчик маршрута `/upload`:

```
// routes.php

'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  /** @var \Psr\Http\Message\UploadedFileInterface $file */
  $file = $request->getUploadedFiles()['file'];
  $process = new Process(
    "cat > uploads/{$file->getClientFilename()}", __DIR__
  );
  $process->start($loop);
  $process->stdin->write($file->getStream()->getContents());

  return new Response(
    200,
    ['Content-Type' => 'text/plain'],
    'Загрузка завершена'
  );
},
```

Здесь скрывается одна проблема — мы создаем и запускаем дочерний процесс, используя следующую команду:

```
"cat > uploads/{$file->getClientFilename()}"
```

Затем мы записываем содержимое загруженного файла в этот процесс, и сразу же после этого возвращаем ответ. А теперь давайте посмотрим, что происходит с нашим дочерним процессом. Попробуйте выполнить эту команду в своем терминале:

```
cat > test.txt
```

Команда по-прежнему слушает входящие данные и сохраняет в файле `test.txt`. Как раз только после того, как мы прервем её с помощью `Ctrl + C`, она завершится.

Сейчас внимательно посмотрите на наш дочерний процесс — мы нигде его не завершаем. Создаём процесс, запускаем его и затем возвращаем ответ. Возможно, вы предполагаете, что после того как завершится колбэк, процесс также будет завершен. Что ж, на самом деле это не так, и я могу вам это показать.

Таймеры

У цикла событий есть интересный функционал, называемый *таймеры*, позволяющий периодично выполнять определённый код с указанным временным интервалом. Допустим, нам нужно выполнять какое-то действие каждую секунду. Отлично, без проблем, добавим таймер:

```
$loop->addPeriodicTimer(
  1,
  function () {
    // код, который выполняется каждую секунду
  }
);
```

Метод `addPeriodicTimer()` первым аргументов принимает интервал в секундах, а вторым — выполняемый колбэк. Давайте добавим таймер и посмотрим, что происходит дальше с нашим процессом после того, как будет отправлен ответ от сервера. Обновим обработчик маршрута `/upload` и добавим следующий код:

```
// routes.php

'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  // ...

  $loop->addPeriodicTimer(
    1,
    function () use ($process) {
      echo 'Дочерний процесс ';
      echo $process->isRunning()
        ? 'выполняется'
        : 'остановлен';
      echo PHP_EOL;
    }
```

```
  );

  return new Response(
    200,
    ['Content-Type' => 'text/plain'],
    'Загрузка завершена'
  );
}
```

Перед тем как вернуть ответ, к циклу событий мы добавляем таймер. Теперь каждую секунду он будет логировать в консоль сервера статус нашего дочернего процесса: выполняется ли он или нет. Обратите внимание, что несмотря на то что ответ будет отправлен, таймер всё равно продолжит выполняться.

Перезапускаем сервер и попробуем что-нибудь загрузить. Обратите внимание на терминал, в котором запущен сервер, наш дочерний процесс по-прежнему выполняется, даже после того, как браузеру ответ был отправлен. Так быть не должно, нужно это исправить.

```
Работает на http://127.0.0.1:8080
Дочерний процесс выполняется
Дочерний процесс выполняется
Дочерний процесс выполняется
Дочерний процесс выполняется
Дочерний процесс выполняется
Дочерний процесс выполняется
Дочерний процесс выполняется
```

Обратно к потокам

Мы уже знаем, что свойство `stdin` дочернего процесса является потоком. Или, если быть более точным, потоком для записи. У этого потока есть метод `end()`. Когда мы *заканчиваем* поток, вызывая метод `end()`, то поток автоматически будет закрыт, после того как все данные, отправленные в него, будут записаны. И как только поток `stdin` будет закрыт, то команда `cat` также будет завершена и весь дочерний процесс остановится.

Таким образом, чтобы поправить баг с зависшими процессами, нам нужно всего-то добавить одну строку `$process->stdin->end()` сразу после вызова `write()`:

```
// routes.php

'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  // ...
  $process->stdin->write($file->getStream()->getContents());
  $process->stdin->end();

  $loop->addPeriodicTimer(
    1,
    function () use ($process) {
      echo 'Дочерний процесс ';
      echo $process->isRunning()
        ? 'выполняется'
        : 'остановлен';
      echo PHP_EOL;
    }
  );

  return new Response(
    200,
    ['Content-Type' => 'text/plain'],
    'Загрузка завершена'
  );
}
```

Мы записываем содержимое загруженного файла в поток, а затем завершаем его методом `end()`. Как только все данные будут записаны, поток будет закрыт. Перезапускаем сервер и снова пробуем что-нибудь загрузить. На этот раз дочерний процесс будет остановлен и больше никаких зависаний — баг исправлен.

```
Работает на http://127.0.0.1:8080
Дочерний процесс остановлен
Дочерний процесс остановлен
Дочерний процесс остановлен
```

Кстати наш код можно ещё немного улучшить. Метод потока для записи `end(mixed $data = null)` также опционально принимает данные, которые будут записаны перед тем, как поток будет закрыт. А это как раз то, что нам нужно: записать содержимое загруженного файла и сразу закрыть поток. Поэтому можно смело заменить эти две строки:

```
$process->stdin->write($file->getStream()->getContents());
$process->stdin->end();
```

на одну эту:

```
$process->stdin->end($file->getStream()->getContents());
```

и ничего не поменяется. Да, и не забудьте удалить таймер. Вот окончательная версия обработчика маршрута `/upload`:

```
// routes.php

'/upload' => function (
  ServerRequestInterface $request, LoopInterface $loop
) {
  /** @var \Psr\Http\Message\UploadedFileInterface $file */
  $file = $request->getUploadedFiles()['file'];
  $process = new Process(
    "cat > uploads/{$file->getClientFilename()}", __DIR__
  );
  $process->start($loop);

  $process->stdin->end($file->getStream()->getContents());

  return new Response(
    200,
    ['Content-Type' => 'text/plain'],
    'Загрузка завершена'
  );
},
```