LEARNING EVENT-DRIVEN PHP WITH

REACTPHP FOR BEGINNERS

BY ZHUK SERGEY

A beginners guide to building asynchronous applications

ReactPHP for Beginners

Sergey Zhuk

This book is for sale at http://leanpub.com/reactphp-for-beginners

This version was published on 2020-12-20

Contents

Handle POST requests

Our main web page has a pretty simple form: a textarea that can be filled and then submitted to the server via a POST request. This is HTML code of the form:

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

When a request arrives at our request handler it already has been processed by the server. The server has parsed its body, so we can consume individual fields of the POST data, in our case it is the `text` field.

Open `routes.php` file and modify the handler for `/upload` route:

```
// routes.php

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

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

Parsed POST data fields can be accessed via `getParsedBody()` method. It returns an associative array, where keys are names of fields from the submitted form. We get the value for field `text` and then return a response which prints what we have submitted.

As always restart the server and try to submit something. Banally simple, isn't it?

Uploading files

Now, let's code something really interesting. The main use case of our application is to upload files on a server and not the text. Let's modify our form and replace textarea with a file field. Only images are allowed to being uploaded. Also, update a label's text. And don't forget to add `enctype="multipart/form-data"` to a `form` tag or no files will be submitted:

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

The form is ready for uploads, let's move to the request handler. Open `routes.php` and remove everything from a handler from `/upload` route. We are going to write it from scratch:

```
// routes.php

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

  },
];
```

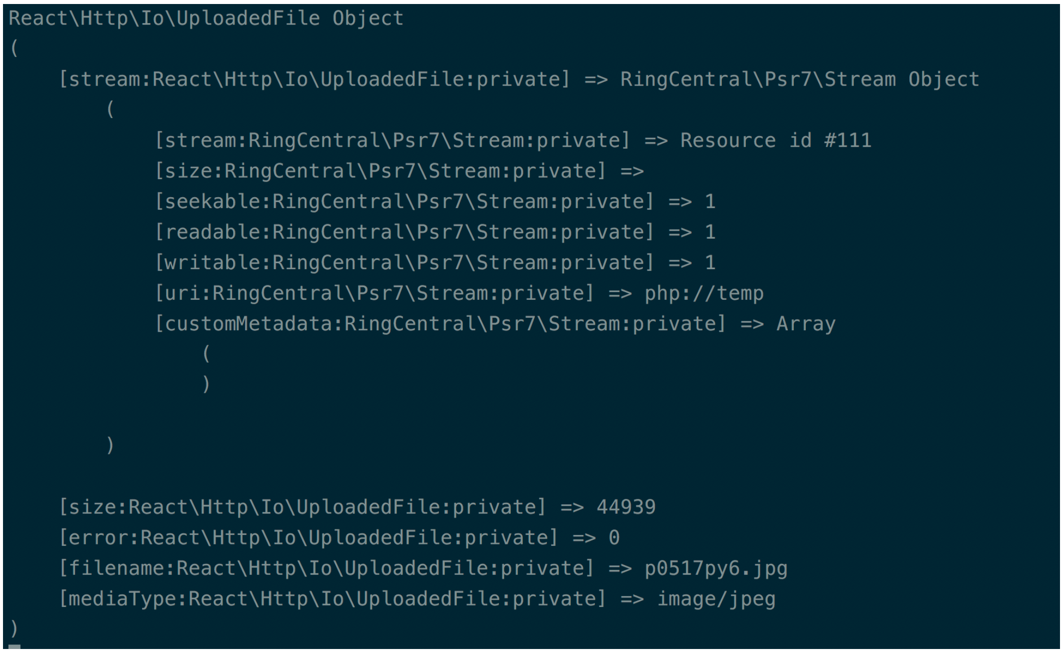
Remember that I've told you that our server does some request processing for us? Not only it parses the request body it also detects any uploads. Call method `getUploadedFiles()` to get all files that have been sent within the request. This method returns an associative array, where keys are names of fields from the submitted form. We are interested in a field with name `file`. To understand what to do next, let's output this variable `$file` on the server terminal. Oh, and don't forget to return a response from the handler. Just any `200` response, even an empty string will be OK:

```
// routes.php

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

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

Restart the server, open `http://127.0.0.1:8080/` in your browser. Select any image and click "submit", then you will be redirected to a blank page. Nothing interesting here. We need server logs. Move back to your terminal and examine what has been printed. In my case I have such output:



```
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
)
```

There will be a lot of strange text, but the main thing is at the top. Look, we have printed `React\Http\Io\UploadedFile` object. That means that our `$file` variable is an instance of `React\Http\Io\UploadedFile`. Nice, and what shall we do with this object? How can we store it on disk?

Method `getUploadedFiles()` returns a list of uploaded files in a normalized structure, where each element is an instance of `Psr\Http\Message\UploadedFileInterface`. This behavior is described[1] in PSR-7 which is a standard that describes common interfaces for representing HTTP messages. Class `React\Http\Io\UploadedFile` is an internal ReactPHP implementation of this interface. Your own code shouldn't depend on this class, instead, when working with uploads use `UploadedFileInterface`.

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

Storing uploads on disk

As you have already guessed we can't use `file_put_contents()` and other similar blocking stuff. In cases where we have to deal with a filesystem, we will definitely need child processes. Therefore, let's create one. But what shell command should be specified for it?

We have already worked with `cat` command. It turns out that, and this time we can also use it. How? Remember that `cat` stands for *concatenate*:

> *Concatenate FILE(s), or standard input, to standard output.*

That means that it allows to grab data from one stream and write it to another. In your terminal try this command:

```
cat > test.txt
```

Then type anything, even several lines and then press `Ctrl + C` to exit the command. Now, look through your current directory and you will find a file named `test.txt` with the contents that you have just typed in the terminal. Command `cat` has read data from its standard input and then this data was written to file `test.txt`. Now remove this file, we don't need it anymore.

This will be the command for our child process. The only difference is that we need to provide a real name for a file. Create folder `uploads` in the root directory of your project. All our uploads will go there. To get a name of the uploaded file call `getClientFilename()` on the `Psr\Http\Message\UploadedFileInterface` object.

```
// 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'], ''
  );
}
```

Remember that you need to specify `__DIR__` as the working directory for a child process.

The next step is to `start()` the process. But method `start()` requires an instance of the running event loop. How can we get this instance here? Just like we did in `/` route handler, via dependency injection. Update our callback signature and add the second parameter `LoopInterface $loop` to it. Having access to loop we can `start` the process:

```
// 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);

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

Notice, that `Router` already provides a loop to all route handlers:

```
// src/Router.php

final class Router
{
  // ...

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

    $handler = $this->routes[$path] ?? $this->notFound($path);
    return $handler($request, $this->loop);
  }
}
```

The last step is to get the contents of the uploaded file and put these contents into the standard input of our child process. Previously for communication with a child process, we have used streams. Remember, when we wanted to read the file from disk and put its contents into the response we have used STDOUT stream. And on the opposite this time we want to write something into a child process. That means that now we need its STDIN stream and `stdin` property.

`stdin` stream has method `write()`. Everything we put into this method will be written right into a child process standard input. How can we get the contents of the uploaded file? It couldn't be easier, call this chain of methods: `$file->getStream()->getContents()`. The first method returns a stream that represents the data of the uploaded file. Then to retrieve the actual contents call method `getContents()` on this stream:

```
// 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'], 'Uploaded'
  );
}
```

Done. The "upload" handler is ready. Notice, that I've added `'Uploaded'` string as a response, as an indicator that an upload was successful.

And that's it. Restart the server and the use case "uploading" is available. Try to upload a local image to our application and it will be available in `uploads` folder in the root directory of the project.

Issue with hanging process

Now take a look at the handler for `/upload` route:

```
// 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'], "Uploaded"
  );
},
```

There is a hidden issue here. We create and start a child process with the following command:

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

Then we write the contents of the uploaded file to this process. And right after that, we return a response. Now, let's see what happens inside out child process. Try to launch the following command in terminal:

```
cat > test.txt
```

Then continue typing anything. You even may press "Enter" but `cat` command is still listening for incoming input to store it inside the `test.txt` file. And only when we interrupt the process by `Ctrl+C` the command stops.

Now, take a closer look at our child process. There is no termination here. We create a process, start it and then return a response. You might think that when we `return` from the callback the process terminates by itself. Well, it's not true. And I can prove it too you.

Timers

Event loop has an interesting feature called *timers* which allows you to periodically execute some code with a specified interval. Let's say that we want to execute some code every second. No problem, add a timer:

```
$loop->addPeriodicTimer(
  1,
  function () {
    // code to be executed every second
  }
);
```

Method `addPeriodicTimer()` accepts an interval in seconds and a callback to execute. Let's add a timer and see what is happening with our child process after we receive a response from the server. Update `/upload` route handler and add the following code:

```
// routes.php

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

  $loop->addPeriodicTimer(
    1,
    function () use ($process) {
      echo 'Child process ';
      echo $process->isRunning()
        ? ' is running'
        : 'has stopped';
      echo PHP_EOL;
    }
  );

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

Before returning a response we add a periodic timer to the loop. Now, every second it will log on the server's console the status of our child process, whether it is running or not. Notice that although the response is sent, the timer continues running.

Restart the server and try to upload something. You will see that the process is still alive, even if the response has been sent to the browser. That's not good. Let's fix it.

```
Listening on http://127.0.0.1:8080
Child process  is running
Child process  is running
Child process  is running
Child process  is running
Child process  is running
Child process  is running
Child process  is running
```

Back to streams

We already know that property `stdin` of a child process is a stream. Or, to be more specific it is a writable stream. And we can `end()` this stream. When the stream ends it will be closed once all data has been written. And once `stdin` is closed the `cat` command will be done and the whole process will be finished.

So, to fix the issue with a hanging process we need to add just one line of code `$process->stdin->end()` right after calling `write()`:

```php
// routes.php

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

  $loop->addPeriodicTimer(
    1,
    function () use ($process) {
      echo 'Child process ';
      echo $process->isRunning()
        ? ' is running'
        : 'has stopped';
      echo PHP_EOL;
    }
  );

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

We write the contents of the uploaded file to the stream and then `end` this stream. Once data is written the stream will be closed. Restart the server and try to upload something. At this time the process will be stopped. No more hanging child processes. The issue has been fixed.

```
Listening on http://127.0.0.1:8080
Child process has stopped
Child process has stopped
Child process has stopped
```

But the code still can be a bit improved. Method `end(mixed $data = null)` of the writable stream optionally accepts data, which can be written before the stream will be closed. And that is exactly what we need: write the contents of the upload and then close the stream. So, we can safely replace these two lines:

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

with this one:

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

and noting changes. And don't forget to remove the timer. This is the final version of the `/upload` handler:

```php
// 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'], `Uploaded`
  );
},
```