PHP Language Server Platform

Release 1

Sep 22, 2023

Contents

The Phpactor Language Server Platform is a framework for creating language servers implementing the Language Server Protocol.

CHAPTER 1

Tutorial

1.1 Getting Started

Below is an example which will run a language server which will respond to any request with a response "Hello world!":

```
#!/usr/bin/env php
<?php

use Amp\Success;
use Phpactor\LanguageServer\Core\Middleware\RequestHandler;
use Phpactor\LanguageServer\Core\Rpc\Message;
use Phpactor\LanguageServer\Core\Rpc\RequestMessage;
use Phpactor\LanguageServer\Core\Rpc\ResponseMessage;
use Phpactor\LanguageServer\Middleware\ClosureMiddleware;
use Phpactor\LanguageServer\Core\Dispatcher\Dispatcher\MiddlewareDispatcher;
use Phpactor\LanguageServerProtocol\InitializeParams;
use Phpactor\LanguageServer\Core\Server\Transmitter\MessageTransmitter;
use Phpactor\LanguageServer\Core\Dispatcher\Factory\ClosureDispatcherFactory;
use Phpactor\LanguageServer\LanguageServerBuilder;

require __DIR__ . '/../../vendor/autoload.php';

$builder = LanguageServerBuilder::create(new ClosureDispatcherFactory(
    function (MessageTransmitter $transmitter, InitializeParams $params) {
        return new MiddlewareDispatcher(
            new ClosureMiddleware(function (Message $message, RequestHandler
↪$handler) {
                if (!$message instanceof RequestMessage) {
                    return $handler->handle($message);
                }

                return new Success(new ResponseMessage($message->id, 'Hello World!'));
            })
```

(continues on next page)

(continued from previous page)

```
        );
    }
));

$builder
    ->build()
    ->run();
```

- `LanguageServerBuilder` abstracts the creation of streams and *builds the Language Server*. It accepts an instance of `DispatcherFactory` - `ClosureDispatcherFactory` is a `DispatcherFactory`. This class has the responsibility initializing the session. It is invoked when the Language Server client sends `initialize` method, providing its capabilities.
- `MessageTransmitter` is how your session can communicate with the client - you wouldn't normally use this directly, but more on this later. The `InitializeParams` is a class containing the initialization information from the client, including the `ClientCapabilities`.
- `MiddlewareDispatcher` Is a `Dispatcher` which uses the Middleware concept - this is the pipeline for incoming requests. Requests go in, and `ResponseMessage` classes come out (or `null` if no response is necessary).
- `ClosureMiddleware` is a `Middleware` which allows you to specific a `\Closure` instead of implementing a new class (which is what you'd normally do). The `Message` is the incoming message (`Request`, `Notification` or `Response`) from the client, the `RequestHandler` is used to delegate to the *next* `Middleware`.
- We return a `ResponseMessage` wrapped in a `Promise`. We only return a `Response` for `Request` messages, and the `Response` must reference the request's ID.
- The `Success` class is a `Promise` which resolves immediately. Returning a `Promise` allows us to run non-blocking co-routines.
- Then finally build and run the server. It will listen on STDIO by default.

If you run this example, you should be able to connect to the language server and it should respond (incorrectly) to all requests with "Hello World!".

Let's try it out.

```
$ echo '{"id":1,"method":"foobar","params":[]}' | ./bin/proxy request | php example/
↪server/minimal.php
```

The `proxy` binary file is used only for this demonstration, it adds the necessary formatting to the message before passing it to our new language server (running on STDIO by default).

It should show *something* like:

At this point you could connect an IDE to your new Language Server, but it wouldn't do very much.

In the next chapter we'll try and introduce some more concepts and add some language server functionality.

1.2 Creating a Language Server

In the previous tutorial we used the `ClosureDispatcherFactory`. This is fine, but let's now implement our own application - `AcmeLS` and give it a dedicated dispatcher factory `AcmeLsDispatcherFactory`. This will be the ingress for a new session:

```
#!/usr/bin/env php
<?php

require __DIR__ . '/../../vendor/autoload.php';

use AcmeLs\AcmeLsDispatcherFactory;
use Phpactor\LanguageServer\LanguageServerBuilder;
use Psr\Log\NullLogger;

$logger = new NullLogger();
LanguageServerBuilder::create(new AcmeLsDispatcherFactory($logger))
    ->build()
    ->run();
```

The dispatcher is responsible for bootstrapping your language server session and creating all the necessary classes that you will need. You might, for example, instantiate a container here using some initialization options form the client.

The Language Server invokes the factory method of this class with two necessary dependencies: `MessageTransmitter` and the `InitializeParams`.

Let's just jump in at the deep end:

```
<?php

namespace AcmeLs;

use Phpactor\LanguageServer\Adapter\Psr\AggregateEventDispatcher;
use␣
↪Phpactor\LanguageServer\Core\Dispatcher\ArgumentResolver\PassThroughArgumentResolver;
↪
use␣
↪Phpactor\LanguageServer\Core\Dispatcher\ArgumentResolver\LanguageSeverProtocolParamsResolver;
↪
use Phpactor\LanguageServer\Core\Dispatcher\ArgumentResolver\ChainArgumentResolver;
use Phpactor\LanguageServer\Core\Workspace\Workspace;
use Phpactor\LanguageServer\Listener\WorkspaceListener;
use Phpactor\LanguageServer\Middleware\CancellationMiddleware;
use Phpactor\LanguageServer\Middleware\ErrorHandlingMiddleware;
use Phpactor\LanguageServer\Middleware\InitializeMiddleware;
use Phpactor\LanguageServerProtocol\InitializeParams;
use Phpactor\LanguageServer\Core\Dispatcher\Dispatcher;
use Phpactor\LanguageServer\Core\Handler\HandlerMethodRunner;
use Phpactor\LanguageServer\Core\Dispatcher\DispatcherFactory;
use Phpactor\LanguageServer\Handler\System\ExitHandler;
use Phpactor\LanguageServer\Handler\Workspace\CommandHandler;
use Phpactor\LanguageServer\Middleware\ResponseHandlingMiddleware;
use Phpactor\LanguageServer\Core\Command\CommandDispatcher;
use Phpactor\LanguageServer\Handler\System\ServiceHandler;
use Phpactor\LanguageServer\Core\Handler\Handlers;
use Phpactor\LanguageServer\Handler\TextDocument\TextDocumentHandler;
use Phpactor\LanguageServer\Core\Dispatcher\Dispatcher\MiddlewareDispatcher;
use Phpactor\LanguageServer\Listener\ServiceListener;
use Phpactor\LanguageServer\Core\Server\RpcClient\JsonRpcClient;
use Phpactor\LanguageServer\Core\Service\ServiceManager;
use Phpactor\LanguageServer\Core\Service\ServiceProviders;
use Phpactor\LanguageServer\Example\Service\PingProvider;
use Phpactor\LanguageServer\Core\Server\ClientApi;
use Phpactor\LanguageServer\Core\Server\ResponseWatcher\DeferredResponseWatcher;
```

(continues on next page)

(continued from previous page)

```php
use Phpactor\LanguageServer\Core\Server\Transmitter\MessageTransmitter;
use Phpactor\LanguageServer\Middleware\HandlerMiddleware;
use Phpactor\LanguageServer\Middleware\ShutdownMiddleware;
use Psr\Log\LoggerInterface;

class AcmeLsDispatcherFactory implements DispatcherFactory
{
    /**
     * @var LoggerInterface
     */
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function create(MessageTransmitter $transmitter, InitializeParams $initializeParams): Dispatcher
    {
        $responseWatcher = new DeferredResponseWatcher();
        $clientApi = new ClientApi(new JsonRpcClient($transmitter, $responseWatcher));

        $serviceProviders = new ServiceProviders(
            new PingProvider($clientApi)
        );

        $serviceManager = new ServiceManager($serviceProviders, $this->logger);
        $workspace = new Workspace();

        $eventDispatcher = new AggregateEventDispatcher(
            new ServiceListener($serviceManager),
            new WorkspaceListener($workspace)
        );

        $handlers = new Handlers(
            new TextDocumentHandler($eventDispatcher),
            new ServiceHandler($serviceManager, $clientApi),
            new CommandHandler(new CommandDispatcher([])),
        );

        $runner = new HandlerMethodRunner(
            $handlers,
            new ChainArgumentResolver(
                new LanguageSeverProtocolParamsResolver(),
                new PassThroughArgumentResolver()
            ),
        );

        return new MiddlewareDispatcher(
            new ErrorHandlingMiddleware($this->logger),
            new InitializeMiddleware($handlers, $eventDispatcher, [
                'name' => 'acme',
                'version' => '1',
            ]),
            new ShutdownMiddleware($eventDispatcher),
            new ResponseHandlingMiddleware($responseWatcher),
```

(continues on next page)

(continued from previous page)

```
            new CancellationMiddleware($runner),
            new HandlerMiddleware($runner)
        );
    }
}
```

- `MessageTransmitter`: This class is provided by the Language Server and allows you to send messages to the connected client. This is quite low-level, instead you should use the `ClientApi`.
- `InitializeParams`: The initialization parameters provided by the client.
- `ResponseWatcher`: Class which tracks requests made by the server *to* the client and can resolve responses, used as a dependency for...
- `ClientApi`: This class allows you to send (and receive) messages to the client. It provides a convenient API `$clientApi->window()->showMessage()->error('Foobar')`. In cases where the API doesn't provide what you need you can use the ...
- `RpcClient` which allows you to send *requests* and *notifications* to the client.
- `ServiceProviders`, `PingProvider`, `ServiceManager`: Ping provider is an annoying service which pings your client for no reason at all, it is an example background process. See *Service Providers* for more information on services.
- `Workspace`: This class can keeps track of LSP text documents.
- `EventDispatcher`: Required by some middlewares to transmit events which can be handled by `Psr\EventDispatcher\ListenerProviderInterface` classes. We use:
  - `ServiceListener`: responsible to start all the services when the server is `initialized`.
  - `WorkspaceListener`: will update the above mentioned `Workspace` based on events emitted by the `TextDocumentHandler`.
- `Handlers`: Method handlers are responsible for handling incoming method requests, this is the main extension point, see *Method Handlers*
- `HandlerMethodRunner`: This class is responsible for calling methods on your class and converting the array of parameters from the request to match the parameters on a handler's method. Find out more in *Method Handlers*.

The RPC method handlers:

- `TextDocumentHandler`: Handles all text document notifications from the client (i.e. text document synchronization). It emmits events.
- `ServiceHandler`: Non-protocol handler for starting/stopping monitoring services.
- `CommandHandler`: Clients can execute commands (e.g. refactor something) on the server, this class handlers that. See *Commands*.
- `ExitHandler`: Handles `shutdown` notifications from the client.

Finally we build the middleware dispatcher with the middlewares which will handle the request:

- `ErrorHandlingMiddleware`: Will catch any errors thrown by succeeding middlewares and log them. As a long running process we don't want to exit each time something goes wrong.
- `InitializeMiddleware`: This middleware *responds* to the initialize request. It also allows your handlers to inject capabiltities into the response, more in *Method Handlers*.
- `ResponseHandlingMiddleware`: Catch responses to requests made *by* the server, and resolves them using our `ResponseWatcher`.

- `CancellationMiddleware`: Often the client knows that a request is no longer required, and it request that that request be *cancelled* (imagine a long-running search). This middleware intercepts the `$/cancelRequest` notifications and tells the runner to cancel them.
- `HandlerMiddleware`: The final destination - will forward the request to the handler runner which will dispatch our *handlers*

In your application you might choose to connect all of this magic in a dependency injection container.

CHAPTER 2

Reference

2.1 Language Server Builder

The language server builder takes care of:

- Creating the necessary streams.
- Creating the *tester*.

It is optional, you can also have a look inside and instantiate the server yourself.

It accepts:

- `Phpactor\LanguageServer\Core\Dispatcher\DispatcherFactory`.
- An optional PSR `Psr\Log\LoggerInterface`.

```php
<?php

use Phpactor\LanguageServer\LanguageServerBuilder;

$server = Phpactor\LanguageServer\LanguageServerBuilder::create(
     new MyDispatcher(),
     new NullLogger()
)->build();

$server->run();
// or
$promise = $server->start();
```

2.1.1 Run or Start

The *run* method on the built language server will start the server and listen for connections. It will also register an error and signal handler.

The *start* method will simply return a promise, without doing anything extra.

2.2 Method Handlers

Method handlers handle the RPC calls from the client.

They look like this:

```
<?php

use Phpactor\LanguageServer\Core\Handler\Handler;
use Amp\Promise;

class MyHandler implements Handler
{
    public function methods(): array
    {
        return [
            'method/name' => 'doSomething',
        ];
    }

    public function doSomething($args, CancellationToken $canellation): Promise
    {
        return new Success('hello!');
    }
}
```

Once registered this command will respond to an RPC request to `method/name` with `hello!`.

2.2.1 Argument Resolvers

The first arguments passed to the parameter will depend on the argument resolvers used by the `HandlerRunner`, the last argument is *always* a cancellation token more on this later.

```
<?php

$runner = new HandlerMethodRunner(
    new Handlers(new MyHandler()),
    new ChainArgumentResolver(
        new LanguageSeverProtocolParamsResolver(),
        new PassThroughArgumentResolver()
    ),
);
```

Here we use the `ChainArgumentResolver` to try two different stragies.

LanguageServerProtocolParamsResolver

This strategy will see if your method implements an LSP `*Params` class and automatically instantaite it for you:

```
<?php

class MyHandler implements Handler
{
    public function methods(): array
    {
```

(continues on next page)

(continued from previous page)

```
        return [
            'textDocument/completion' => 'complete',
        ];
    }

    public function doSomething(CompletionParams $completionParams, CancellationToken
↪$canellation): Promise
    {
        $uriToTextDocument = $completionParams->textDocument->uri;
        // ...
    }
}
```

You should be able to do this with *any method documented in the language server specification*.

DTLArgumentResolver

This argument resolver will try and match the parameters from the request to the parameters of your method.

PassThroughArgumentResolver

This is a fallback resolver which will simply pass the raw array of arguments.

2.2.2 Co-routines

Your method MUST return an `Amp\Promise`. If you return immediately you can use the `new Success($value)` promise, if you do any *interruptable** work which takes a significant amount of time you should use a co-routing. For example:

```
<?php

class MyHandler implements Handler
{
    //...

    public function doSomething(CompletionParams $params, CancellationToken
↪$canellation): Promise
    {
        return \Amp\call(function () {
            // ...
            $completionItems = [];

            foreach($this->magicCompletionProvider->provideCompletions($params) as
↪$completion) {
                $completionItems[] = $completion;
                yield Amp\delay(1);
            }

            return $completionItems;
        });

    }
}
```

The above will process a single completion item but then yield control back to the server for 1 millisecond before continuing. This allows the server to do other things (like for example **cancel this request**).

2.2.3 Cancellation

The `CancellationToken` passed to the method handler can throw an exception if the request is cancelled as follows:

```
<?php

class MyHandler implements Handler
{
    //...

    public function doSomething(CompletionParams $params, CancellationToken
↪$canellation): Promise
    {
        return \Amp\call(function () {
            // ...
            $completionItems = [];

            foreach($this->magicCompletionProvider->provideCompletions($params) as
↪$completion) {
                $completionItems[] = $completion;
                yield Amp\delay(1);
                try {
                    $cancellation->throwIfRequested();
                } catch (Amp\CancelledException $cancelled) {
                    break;
                }
            }

            return $completionItems;
        });
    }
}
```

In the above example, when the server cancels this request, the exception will be thrown and we will return early.

2.3 Service Providers

Service providers are background services which can should be started on on the `initialized` notification from the client.

A good example of a service is a code indexing service which watches the file system and indexes code when files change.

2.3.1 Example

A full example of a service provider:

```
1 <?php
2
```

(continues on next page)

(continued from previous page)

```php
namespace Phpactor\LanguageServer\Example\Service;

use Amp\CancellationToken;
use Amp\CancelledException;
use Amp\Delayed;
use Amp\Promise;
use Phpactor\LanguageServer\Core\Server\ClientApi;
use Phpactor\LanguageServer\Core\Service\ServiceProvider;

/**
 * Example service which shows a "ping" message every second.
 */
class PingProvider implements ServiceProvider
{
    /**
     * @var ClientApi
     */
    private $client;

    public function __construct(ClientApi $client)
    {
        $this->client = $client;
    }

    /**
     * {@inheritDoc}
     */
    public function services(): array
    {
        return [
            'ping'
        ];
    }

    /**
     * @return Promise<null>
     */
    public function ping(CancellationToken $cancel): Promise
    {
        return \Amp\call(function () use ($cancel) {
            while (true) {
                try {
                    $cancel->throwIfRequested();
                } catch (CancelledException $cancelled) {
                    break;
                }
                yield new Delayed(1000);
                $this->client->window()->showMessage()->info('ping');
            }
        });
    }
}
```

This is similar to *method handlers* with the exception that:

- The `services` method provides only an array of method names. The name doubles as both the method and service name.

- The method is called when the Language Server is initialized (or when it is started via. the service manager).
- Services are passed only a `CancellationToken`.

2.3.2 Usage

```php
<?php

$serviceProviders = new ServiceProviders(
    new PingProvider($clientApi)
);

$serviceManager = new ServiceManager($serviceProviders, $logger);
$eventDispatcher = new EventDispatcher(
    new ServiceListener($serviceManager)
);

$handlers = new Handlers(
    // ...
    new ServiceHandler($serviceManager, $clientApi),
    // ...
);

return new MiddlewareDispatcher(
    // ...
    new InitializeMiddleware($handlers, $eventDispatcher)
    // ...
);
```

In the above code the `ServiceManager` is responsible for starting and stopping services, the `ServiceHandler` handles RPC methods to start/stop services, and we use the `ServiceListener` to start the services when the server is initialized (based on the `Initialized` event issued by the `InitializeMiddleware`.

2.4 Diagnostic Providers

Diagnostic providers are invoked when text documents are updated and are responsible to send diagnostics (e.g. actual or potential problems with the code) to the client.

2.4.1 Example

Example of a diagnostic provider:

```php
<?php

namespace Phpactor\LanguageServer\Example\Diagnostics;

use Amp\CancellationToken;
use Amp\Promise;
use Phpactor\LanguageServerProtocol\Diagnostic;
use Phpactor\LanguageServerProtocol\DiagnosticSeverity;
use Phpactor\LanguageServerProtocol\Position;
use Phpactor\LanguageServerProtocol\Range;
use Phpactor\LanguageServerProtocol\TextDocumentItem;
```

(continues on next page)

(continued from previous page)

```php
use Phpactor\LanguageServer\Core\Diagnostics\DiagnosticsProvider;
use function Amp\call;

class SayHelloDiagnosticsProvider implements DiagnosticsProvider
{
    /**
     * {@inheritDoc}
     */
    public function provideDiagnostics(TextDocumentItem $textDocument, CancellationToken $cancel): Promise
    {
        /** @phpstan-ignore-next-line */
        return call(function () {
            return [
                new Diagnostic(
                    new Range(
                        new Position(0, 0),
                        new Position(1, 0)
                    ),
                    'This is the first line, hello!',
                    DiagnosticSeverity::INFORMATION
                )
            ];
        });
    }

    public function name(): string
    {
        return 'say-hello';
    }
}
```

```
$diagnosticsService = new DiagnosticsService(
    new DiagnosticsEngine($clientApi, new AggregateDiagnosticsProvider(
        $logger,
        new SayHelloDiagnosticsProvider()
    ))
);
```

2.4.2 Integration

Diagnostics are facilitated through the "Diagnostics Service" which in turn requires the `DiagnosticsEngine` which accepts a `DiagnosticProvider` - below we use the `AggregateDiagnosticsProvider` which allows you to provide many diagnostic providers:

```php
<?php

$diagnosticsService = new DiagnosticsService(
    new DiagnosticsEngine($clientApi, new AggregateDiagnosticsProvider(
        $logger,
        new SayHelloDiagnosticsProvider()
    ))
);
```

2.5 Code Action Provider

Code action providers can be implemented to enable you to suggest *commands* which can be executed on a given text document and range.

2.5.1 Example

Example of a command:

```php
<?php

namespace Phpactor\LanguageServer\Example\CodeAction;

use Amp\CancellationToken;
use Amp\Promise;
use Phpactor\LanguageServerProtocol\CodeAction;
use Phpactor\LanguageServerProtocol\CodeActionKind;
use Phpactor\LanguageServerProtocol\Command;
use Phpactor\LanguageServerProtocol\Range;
use Phpactor\LanguageServerProtocol\TextDocumentItem;
use Phpactor\LanguageServer\Core\CodeAction\CodeActionProvider;
use function Amp\call;

class SayHelloCodeActionProvider implements CodeActionProvider
{
    public function provideActionsFor(TextDocumentItem $textDocument, Range $range, CancellationToken $cancel): Promise
    {
        /** @phpstan-ignore-next-line */
        return call(function (): array {
            return [
                CodeAction::fromArray([
                    'title' => 'Alice',
                    'command' => new Command('Hello Alice', 'phpactor.say_hello', [
                        'Alice',
                    ])
                ]),
                CodeAction::fromArray([
                    'title' => 'Bob',
                    'command' => new Command('Hello Bob', 'phpactor.say_hello', [
                        'Bob',
                    ])
                ])
            ];
        });
    }

    /**
     * {@inheritDoc}
     */
    public function kinds(): array
    {
        return [CodeActionKind::QUICK_FIX];
    }

    public function describe(): string
```

(continues on next page)

(continued from previous page)

```
    {
        return 'says hello!';
    }
}
```

It unconditionally provides two code actions: `Alice` and `Bob`. It references a previously registered *commands* such as:

```
<?php

namespace Phpactor\LanguageServer\Example\Command;

use Phpactor\LanguageServer\Core\Command\Command;
use Phpactor\LanguageServer\Core\Server\ClientApi;

class SayHelloCommand implements Command
{
    /**
     * @var ClientApi
     */
    private $api;

    public function __construct(ClientApi $api)
    {
        $this->api = $api;
    }

    public function __invoke(string $name): void
    {
        $this->api->window()->showMessage()->info(sprintf('Hello %s!', $name));
    }
}
```

2.6 Commands

Commands are issued from the client to the server, they are similar in concept to RPC calls with the exception that they are expcitly registered with the server and executed via. an RPC method.

2.6.1 Usage

The command handler accepts a `CommandDispatcher` which in turn accepts a map of command *names* to invokable objects:

```
<?php

use Phpactor\LanguageServer\Handler\Workspace\CommandHandler;
use Phpactor\LanguageServer\Workspace\CommandDispatcher;

// ...
$handler = new CommandHandler(
    new CommandDispatcher([
        'my_command' => function (array $args) {
```

(continues on next page)

(continued from previous page)

```
                // do something
            }
        ])
));
```

Now, when the client connects, the server will signify (via. `ServerCapabilties`) that this command is available.

CHAPTER 3

Guide

3.1 Testing

This package includes some tools to make testing easier.

3.1.1 Protocol Factory

The `ProtocolFactory` is a utility class for creating LSP protocol objects:

```php
<?php

use Phpactor\LanguageServer\Test\ProtocolFactory;

$item = ProtocolFactory::textDocumentItem('uri', 'content');
$initializeParams = ProtocolFactory::initializeParams('/path/to/rootUri');
```

This is useful as the LSP objects can be complicated and we can assume some defaults using the factory.

3.1.2 Unit Testing Handlers, Services etc.

You can use the *Language Server Tester* to test your handlers, services, commands etc as follows:

```php
<?php

$tester = LanguageServerTesterBuilder::create()
    ->addHanlder($myHandler)
    ->addServiceProvider($myServiceProvider)
    ->addCommand($myCommand)
    ->build();

$result = $tester->requestAndWait('soMyThing', []);
```

Lean more about the *LanguageServerTester*

3.1.3 Integration Testing

If you are using the `LanguageServerBuilder` to manage the instantiation of your `LanguageServer` then, assuming you are using some kind of dependency injection container, you can use the `tester` method to get the *Language Server Tester*.

```
<?php

$builder = $container->get(LanguageServerBuilder::class);
assert($builder instanceof LanguageServerBuilder);
$tester = $builder->tester();
$response = $tester->requestAndWait('foobar', ['bar' => 'foo']);
$response = $tester->notifyAndWait('foobar', ['bar' => 'foo']);
```

This will provide the *Language Server Tester* with the "real" dispatcher.

3.1.4 Language Server Tester

The tester provides access to a test transmitter from which you can access any message sent by the server:

```
<?php

// ...
$messageOrNull = $tester->transmitter()->shift();
```

You can also use some convenience methods to control the server:

```
<?php

// ...
$messageOrNull = $tester->textDocument()->open('/uri/to/text.php', 'content');
$tester->services()->start('myservice');
```

The tester will automatically initialize the server, but you can also pass your own initialization parameters:

```
<?php

// ...
$tester = $builder->tester(ProtocolFactory::initializeParams('/uri/foobar.php'));
```