PSR-7とPSR-15による Webアプリケーション実装パターン

Web application implementation pattern based on PSR-7 and15

pixiv Inc.
USAMI Kenta

pixiv

2022-04-11
PHPerKaigi Day 2

お前誰よ

- うさみけんた（@tadsan）/ Zonu.EXE / にゃんだーすわん
- ピクシブ株式会社 pixiv事業本部 エンジニア
  - 最近はピクシブ百科事典(dic.pixiv.net)を開発しています
- Emacs Lisper, PHPer
  - Emacs PHP Modeを開発しています（2017年-）
- PHPerKaigi コアスタッフ、PHPカンファレンス実行委員

tadsanのあれこれが読める場所

- https://tadsan.fanbox.cc/
- https://scrapbox.io/php/
- https://www.phper.ninja/
- https://zenn.dev/tadsan
- https://qiita.com/tadsan
- https://github.com/bag2php

前回のPHPerKaigi

採択 2021/03/28 13:10〜 Track A レギュラートーク（40分）

作って理解するDIコンテナ PHPerKaigi 2021

☆ 30 ビデオ

 うさみけんた tadsan

人は言います——モジュールを疎結合にせよ、と。
人は言います——具象ではなく抽象に依存して実装せよ、と。

ソフトウェア設計におけるベストプラクティスは異口同音に関心の分離の重要性を説きます。SOLID原則の「D」ことThe Dependency Inversion Principle / 依存性逆転の原則を実装するための道具立てとしてDIコンテナ、あるいはIoCコンテナと呼ばれる仕組みが利用されることがあります。

「PHP DI」などで検索すると、さまざまな説明が読めることでしょう。.....御託はわかった。しかし昔の私には問題意識も活用方法も理解できず時間は過ぎ去っていきました。フレームワーク嫌いの私も数年も経験を積み、いくつかのフレームワークに触れてそれぞれの長短がわかってくると「Laravelのこれだけ使いたい」とか「CakePHPのここだけ羨ましい」などと考えはじめるものです。

このトークではPHP-DIに触発された簡易的なDIコンテナの実装方法を紹介し、PSR-18のような外部と通信するインターフェイスを分離しオブジェクトの生成をDIコンテナのオートワイヤリングに任せることで、どのように実装を簡潔にできるかを説明します。発表内容は実装にフォーカスしており、設計原則やソフトウェアアーキテクチャについては言及しません。

去年のPHPカンファレンス

採択 2021/10/03 10:00〜 Track2 Long session (60 mins)

配列、ジェネリクス、PHPで書けない型 PHP Conference Japan 2021

☆ 6

 うさみけんた tadsan

近年、PHPの機能強化により型宣言だけで安全に書ける範囲が広まっています。

その一方でPHPStanやPsalmといった静的解析ツールはPHPでは表現できない強力な型を提供することでコード品質向上の価値を高めることが可能です。
PhpStormはPHPプログラマに静的型の恩恵をもたらした一方、先述のツールと比べサポートする型について見劣りする点がありました。
ところが最近リリースされた待望の新バージョンである2021.2はこれまで静的解析ツールの専売特許だった型のいくつかがサポートされるようになり、型検査だけでなく入力補完などの恩恵を受けられるようになりました。そこで追加された型のひとつがジェネリクス（総称型）です。

本発表では、PHPの型についての基礎知識（今日からできる安心型付け入門）があることを前提として、PHPにジェネリクスは入るのか？の内容を軸に、PHPの型宣言では現時点で賄えない部分、特に配列の型およびジェネリクスの概念、class-string型の概念、そしてジェネリクスを実際に活用するためのテクニックを説明します。

Discord Channel: #track2-4-php-type

pixiv

一昨年のPHPカンファレンス

採択 2020/12/12 14:00〜 Track2 Long session (60 mins)

PSRで学ぶHTTP Webアプリケーションの実践 PHP Conference Japan 2020

☆ 20 ビデオ

 うさみけんた tadsan

PSRはPHP-FIG(PHPフレームワーク相互運用グループ)が発表する勧告群です。
その中ではHTTPについての勧告としてPSR-7, PSR-15, PSR-17そしてPSR-18が発表されており、これらはWebアプリケーションのモジュール間のインターフェイスとして活用できます。

今回は社内独自のフレームワークをPSR-7/15/17実装として置き換えた事例をとって、PHPとHTTPリクエストの関係およびPSR-17を実装したWebアプリケーションについて説明します。

このトークを見るにあたって、過去のPHPカンファレンスでの発表を含む「PSR-HTTPシリーズを理解するための情報源」を読むことを強く推奨します。
https://scrapbox.io/php/PSR-HTTPシリーズを理解するための情報源

Track ID: Track2-3
Discord Channel: #track2-3-psr-http-web

pixiv

Software Design 11月号

この記事は執筆時点で10月2～3日に開催を予定しているPHPカンファレンス2021と連動した短期集中連載の最終回です。つまり、この記事が掲載されたSoftware Design 2021年11月号をみなさんが手にとられるころにはすでに開催されています。カンファレンスで発表されたセッションはYouTubeチャンネル[注1]にアーカイブが公開されているはずですので、興味のある方はぜひご覧ください。筆者はPHPカンファレンス2021で「**配列、ジェネリクス、PHPで書けない型**」と題したトークを発表します。

最終回では近年の動的言語の型検査の動向についての概観に触れたあと、PHPの動向について説明します。

静的型付き(statically typed)の言語と呼びます。事前にデータの種類を固定せずに処理することを**動的型検査**(dynamic type checking)と呼び、実行時にデータに応じた処理をすることが基本のプログラミング言語を**動的プログラミング言語**(dynamic programming language)と呼びます。

メジャーなものでは、C言語、Java、C++、C#、Go、Swift、Rust、Scala、Kotlin、Haskell、OCamlなどは静的型付きの言語です。また、Python、JavaScript、Ruby、シェルスクリプト、Clojureなど Lisp系の言語、Smalltalk、そしてPHPは動的言語と呼ばれます。俗に実行ファイルを生成するものをコンパイル言語、ソースコードを直接実行するものをインタプリタ言語として

pixiv

WEB+DB PRESS総集編

pixiv

WEB+DB PRESS Vol.96(2016年)

第13回 PHP大規模開発入門

PHPからのHTTPリクエスト
ファイル操作関数、curl関数、Guzzleライブラリ

ピクシブ㈱
うさみ けんた　USAMI Kenta
mail tadsan@pixiv.com　GitHub zonuexe　Twitter @zonu_exe

PHPはWebと親和性の高いプログラミング環境です。HTTP(*Hypertext Transfer Protocol*)はWebの標準的な通信プロトコルであり、Webアプリケーションを実装するということは、利用者からのHTTPリクエストを受け付けてHTTPの仕様に沿った応答を返すプログラムを書くことにほかなりません。

実際にはWebサーバやPHP、あるいはWebアプリケーションフレームワークと呼ばれる層が巧妙に隠蔽(いんぺい)してくれるので、HTTPについてあまり意識しなくともWebサービスを構築することが可能ですが、今回はPHPから「Webサービスを利用する」という観点でHTTPについて解説します[注1]。

HTTPの基礎

ます。最も目にする機会が多いのは、`http://wdpress.gihyo.jp/`のようなURL[注4]の一部分でしょう。このようなURLをブラウザのアドレスバーに入力することでWebサイトを閲覧できます。

Webサイトにも多様な形態があります。たとえばイラスト(画像)を投稿したりほかの利用者が投稿したイラストを閲覧できるpixiv、動画を投稿またはライブ配信したり、それを見ることができるYouTubeなどです。これらのサービスは不特定多数の利用者にアクセスされることを想定していますが、そうではない領域にもHTTPは利用されています。たとえば全文検索エンジンのApache Solr(アパッチ ソーラー)はApache Lucene(アパッチ ルシーン)というJavaのライブラリをHTTPを介して操作できるようにしたフロントエンドです。

LuceneはJavaのライブラリなので、当然Java(も

pixiv

WEB+DB PRESS Vol.96(2017年)

第17回 PHP大規模開発入門

レガシーなプロダクトの改善
フレームワークを利用できない環境でのライブラリ活用

ピクシブ㈱
うさみ けんた　USAMI Kenta
mail tadsan@pixiv.com　GitHub zonuexe　Twitter @zonu_exe

PHPはブログなどのメディアや業務システム、モバイルアプリのバックエンド(API)など、その場所を選ばないWebアプリケーションの開発環境・プログラミング言語として、多くの現場で利用されています。

ひとまとめにPHPと言っても、有名フレームワークを採用したプロジェクト、フレームワークなしで開発されたプロジェクト、自社独自のフレームワーク基盤のプロジェクトなど、現場によって多種多様です。

今回は、最新のフレームワークを採用できない環境であっても、ライブラリを組み合わせて開発効率の向上を図るためのノウハウを紹介します。

Webフレームワークは必要か？

みなさんご存じのとおり、PHPはもともとWebに位でモジュールを分割するのか(あるいは分割しないのか)、どのディレクトリにファイルを配置すべきかなどをゼロから考えなければいけません。

機能の統合

「Batteries Included Philosophy」(電池同梱の哲学)という考え方があります。これはPythonで標榜される哲学で、「最初から電池が入っているので、箱から出してすぐ使えますよ」という意味です。

Webフレームワークは開発・運用しやすくするために、次のようなさまざまな機能の組み合わせからなります。それぞれの機能を詳しく知らなくても使い始められるよう構成されています。

◉ルーティング

pixiv

仕事の話

Powered by PSR-15

pixiv

PIXIV SPRING BOOT CAMP

pixiv

たぶん近日中に夏インターンも募集

COURSES

スペシャリスト

技術基盤（Webフレームワーク）

内容	オンライン百科事典「ピクシブ百科事典」の機能開発およびコア機能・Webアプリケーションフレームワークの改善を行うコースです。フレームワークの開発を通じて、Webサービスが提供すべき基本機能の理解や、開発者が利用しやすい設計について学ぶことが目的です。
使用技術・条件	PHP, HTTP, PHPStan ◯必須条件 - Webアプリケーションの開発経験があること（PHPである必要はありません） - IPA 「安全なウェブサイトの作り方」 https://www.ipa.go.jp/security/vuln/websecurity.html 相当のWebセキュリティについての事前知識があること

pixiv

それそろフレームワークとしての
~~突っ込みどころ~~改善ポイントが
枯渇しつつあるので
参加したい学生はお早めに

今回のお題

pixiv

プロポーザル

採択 2022/04/11 16:05〜 Track B　レギュラートーク（40分）

PSR-7とPSR-15によるWebアプリケーション実装パターン　PHPerKaigi 2022

☆ 13

 うさみけんた　tadsan

PSR-7がPHPにおけるHTTPの標準的なインターフェイスとして採択されて久しく、それに対応するHTTPハンドラ／ミドルウェアの仕様であるPSR-15を使えば特定のフレームワークに依存せずWebアプリケーションを実装できます。ピクシブ百科事典はPSR-7/PSR-18ベースのWebアプリケーションで実装されており、このトークではその経験を踏まえたアプリケーション実装パターンについて紹介します。

この発表ではPSRで学ぶHTTP Webアプリケーションの実践などで説明しているHTTPとPSR-7の関係について知っていることを前提に、よりアプリケーション実装やユニットテストなどの実践にフォーカスして説明します。

日本語のPSR-HTTP関連資料集

pixiv

Webフレームワークを作りたい人や内部構造を理解したい人向け

前半はフレームワーク
一般の概念の話もします

後半もPSR-15入門とテスト方法みたいな話に終始します

最後のスライド
(予告)

俺たちのPSR-15道はこれからだ！

(ご静聴ありがとうございました。
tadsanの次回作にご期待ください)

pixiv

さて

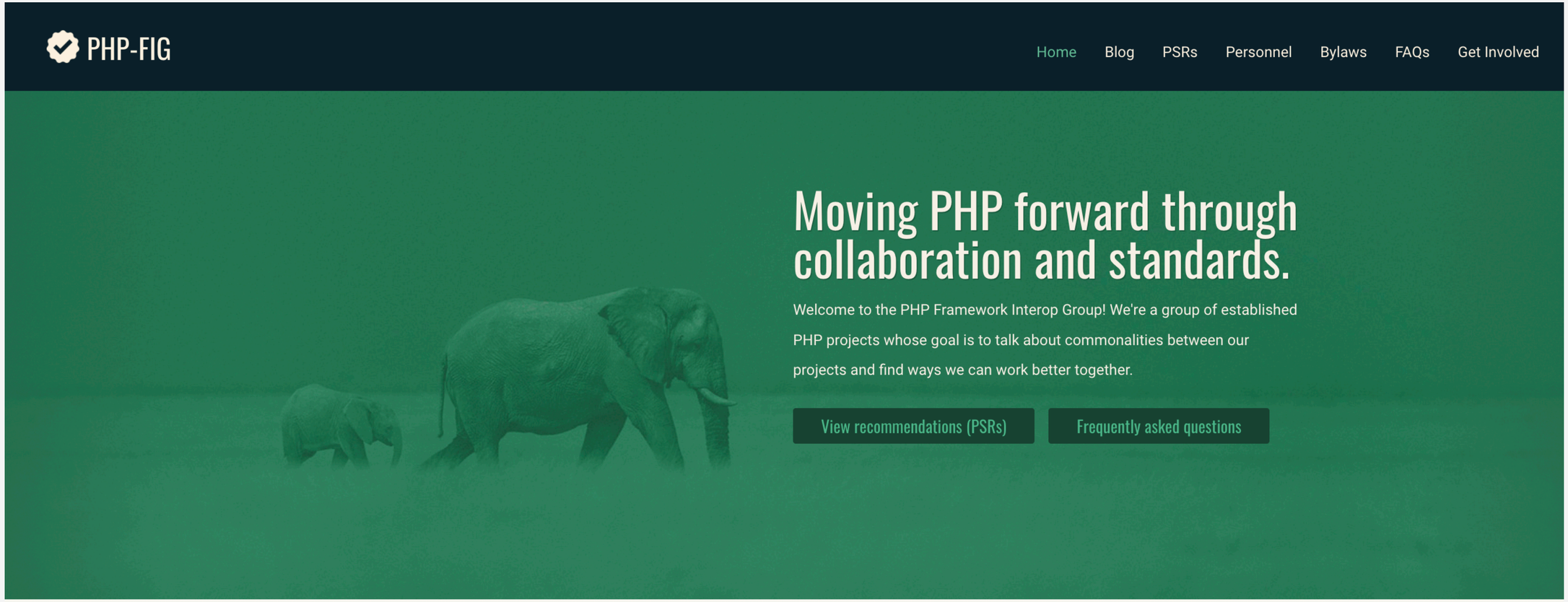
誰がPSR-15を使うのか

- PSR標準に準拠したフレームワークを開発したい人
- CakePHPのようなPSR-15準拠フレームワークでミドルウェアを書く必要に迫られた人
- シンプルなライブラリを組み合わせだけで、リーンでクリーンなWebアプリを開発したい人

PSRとは

pixiv

PHP-FIG



AUTOLOADING

Autoloaders remove the complexity of including files by mapping namespaces to file system paths.

PSR-4: Improved Autoloading

```php
<?php

use Vendor\Package\ClassName;

$object = new ClassName();
```


pixiv

PSRs

INTERFACES

Interfaces simplify the sharing of code between projects by following expected contracts.

PSR-3: Logger Interface | PSR-6: Caching Interface | PSR-11: Container Interface
PSR-13: Hypermedia Links | PSR-14: Event Dispatcher | PSR-16: Simple Cache

```php
<?php

namespace Psr\Log;

/**
 * Describes a logger instance
 */
interface LoggerInterface
{
```

HTTP

Interoperable standards and interfaces to have an agnostic approach to handling HTTP requests and responses, both on client and server side.

PSR-7: HTTP Message Interfaces | PSR-15: HTTP Handlers | PSR-17: HTTP Factories
PSR-18: HTTP Client

```php
<?php

namespace Psr\Http\Message;

/**
 * Representation of an outgoing, client-side request.
 */
interface RequestInterface extends MessageInterface
{
```

CODING STYLES

Standardized formatting reduces the cognitive friction when reading code from other authors.

PSR-1: Basic Coding Standard | PSR-12: Extended Coding Style Guide

```php
<?php

namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

Follow us on Twitter | Discuss on Discord | Contribute via GitHub | Join our mailing list

pixiv

PSR (PHP Standard Recommendation)

- PHP-FIG(PHP相互運用グループ)という団体が策定する標準勧告
- コーディング規約やインターフェイス定義などが定められている
- フレームワークやCMSの開発団体間で共通仕様を策定して
  コンポーネントを相互運用できるようにすることが目的
  - 本来はメンバー間での相互運用が目的なのだが、
    われわれ下々の一般PHPerもその成果物の恩恵に与っている恰好

よく知られたPSR

- **PSR-4 オートローディング**
  - 名前空間とディレクトリをマッピングするルール。Composerで利用可能
- **PSR-1 基本コーディング標準**
  - ファイル名はUTF-8で、定義とスクリプトのファイル分離など基本ルール
- **PSR-12 拡張コーディングスタイルガイド**
  - 改行やスペース位置などのスタイル策定のベースになるガイド

PSR interfaces

- PSR-3 ロガー
  - syslog風の汎用ロギングインターフェイス
- PSR-11 コンテナ
  - DIコンテナと例外送出のインターフェイス
- PSR-6 キャッシュ / PSR-16 単純なキャッシュ(SimpleCache)
  - TTL(寿命)付きで値を保持するクラスのインターフェイス

インターフェイス(interface) 概要

- class定義に似ているが、クラスが備えるべきメソッドの型だけ宣言したもの
- interface名は型宣言（パラメータや戻り値、プロパティなど）に利用できる
- クラスは複数のインターフェイスを実装（implements）できる
  - PHPのクラスは一つだけ継承（extends）できるのと対照的
- class定義と分けることで、具体的な実装と抽象を分離できる
  - 同じインターフェイスを実装したクラスに交換可能になる

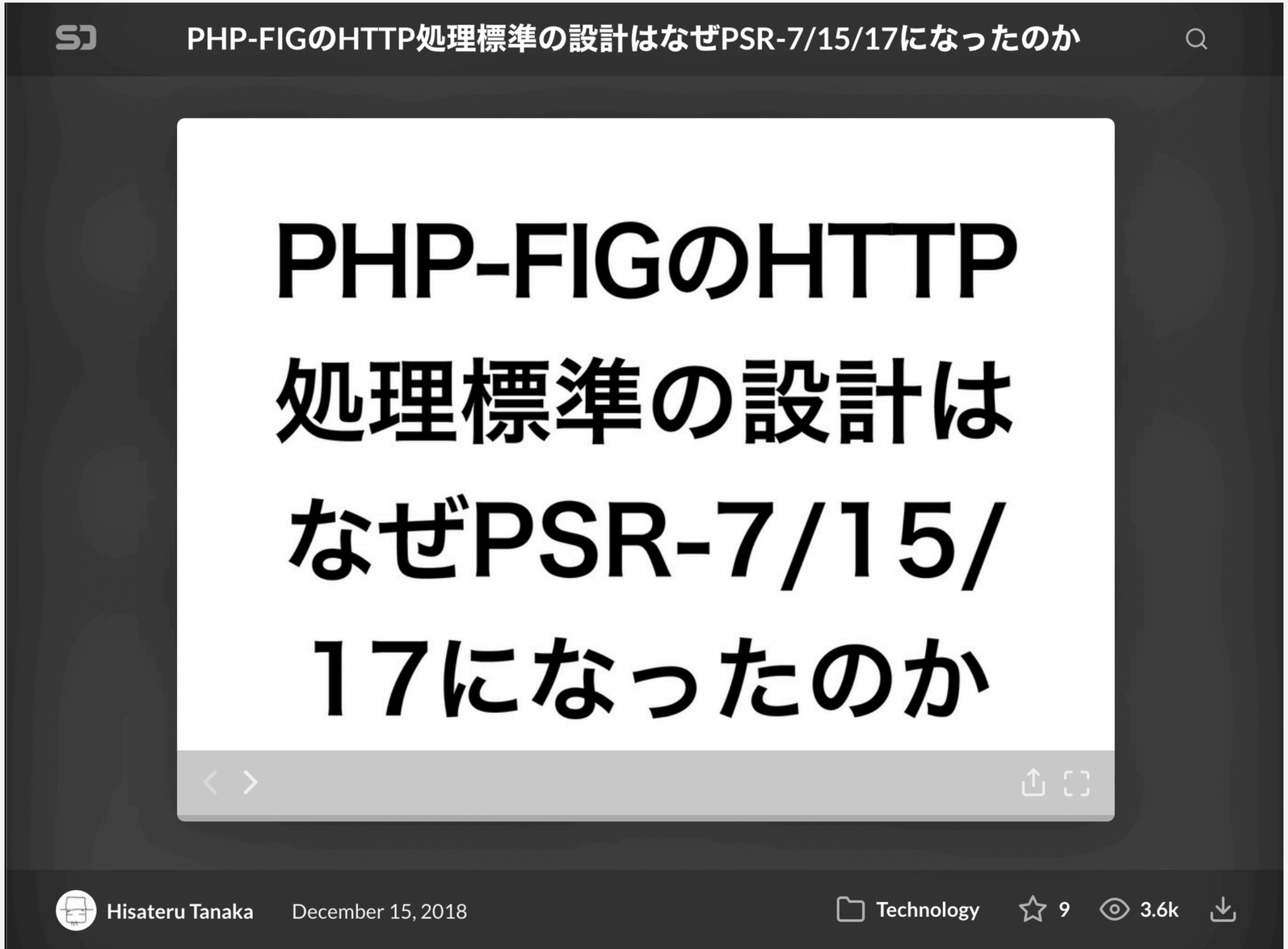
余談：PSRへの誤解

- PSRは全PHP開発者が守らなければいけない標準という性質ではない
  - PHP-FIG参加メンバーも勧告に従うかどうかは任意
- PSR-2/PSR-12（スタイルガイド）は規約ではなく、
  プロジェクトごとにスタイルを策定するためのガイドラインです
  - カスタマイズする場合でも、PHP-CS-FixerやPHP_CodeSnifferのようなツールでPSR-12からの差分で定義できるというのがメリット

PSR vs HTTP

- PSRではHTTPに関連するオブジェクトのインターフェイスを定義している
- **PSR-7 HTTPメッセージ**
  - リクエスト、レスポンス、URI、ストリームなどのインターフェイスを定義
  - PSR-17でこれらメッセージオブジェクトのファクトリが定義されている
- **PSR-15 HTTPハンドラ(ミドルウェア)**
  - リクエストハンドラとミドルウェアのインターフェイスを定義

PSRのコンセプトについて最重要資料



pixiv

PSR-7のおさらい

pixiv

PSR-7 HTTP Messages

- 不変(イミュータブル)が基本 (Streamを除く)
  - setXXXのようなメソッドは存在しない
- 継承関係がある
  - Psr\Http\MessageInterface (レスポンスとリクエストの共通集合)

```
├── RequestInterface          (HTTPリクエスト)
│   └── ServerRequestInterface (↑+追加情報)
└── ResponseInterface         (HTTPレスポンス)
```


pixiv

不変性(immutability)

- 一度作ったオブジェクト内部の状態が変わることはない
- 状態を付加したいときは $res->setHeader()のような操作はできず、
  $res = $res->withHeader() のように追記する
- この性質により、一度作ったインスタンスオブジェクトが別の場所で
  意図せず書き換えられる可能性を避けられるので非常に安全

PSR-7 Packages

- PSR-7を実装するパッケージは複数あり、基本的には交換可能
  https://packagist.org/providers/psr/http-message-implementation
- 基本的にはWebフレームワークとHTTPクライアントどちらでも使える
  - guzzlehttp/psr7, nyholm/psr7,
    laminas/laminas-diactoros, slim/psr7などがある
  - Guzzleのstreamは機能豊富、Nyholmは軽量などの特徴がある

ServerRequestInterface

- RequestInterfaceを継承したインターフェイス
- $_SERVER, $_GET, $_POST, $_COOKIE, $_FILES相当のデータを扱うメソッドが追加されている
  - withCookieParams()などは$_COOKIE相当のデータを付加するが、内部で保持するHTTPヘッダを書き換えてはならない(MUST NOT)と仕様で明言されている
  - withAttribute()/getAttribute()という、事実上なんでも入れていい収納スペースがある

ServerRequest::withAttribute()

- アトリビュート(属性)という名前が付いているが、PHP 8で追加された #[Attributes] とは何の関係もなく、単に付加的な情報を持たせる
- 型的には文字列をキーにして、本当にあらゆるものが付与できてしまう
  - 原則何を入れてもいいのだが、ユーザーのリクエストと関連のあるものに限って含めた方がとり回しがよくなるはず
  - キーにも任意の値をセットできるが、クラス名文字列を活用すると明示的
    - $req->withAttribute(Session::class, new ConcreteSession())
    - 個人的には単なるスカラー値よりオブジェクトで統一するのが安全と思う

ResponseInterface

- RequestInterfaceとは異なり、Server…版は存在しない
- レスポンスボディは文字列ではなく、StreamInterfaceを実装したオブジェクトで表現する
  - ボディをセットするには $response->withBody()を使う方法と $response->getBody()->write()する方法がある
    - 後者は追記になってしまうので、前者の方が確実

PSR-17 HTTP Factories

- PSR-7オブジェクトを生成する役割のインターフェイス
- $res = new GuzzleHttp\Psr7\Response()と書く代りに

  $res = $response_factory->createResponse()でオブジェクトを生成できる
- これがなぜ重要なのかは田中ひさてるさんの発表を見てください
- 個別の実装が具体的なライブラリに依存しなくなるようにできます

pixiv

不変性(immutability)

- 一度作ったオブジェクト内部の状態が変わることはない
- 状態を付加したいときは $res->setHeader()のような操作はできず、
  $res = $res->withHeader() のように追記する
- この性質により、一度作ったインスタンスオブジェクトが別の場所で
  意図せず書き換えられる可能性を避けられるので非常に安全

Nyholm Psr17Factory

- PSR-7オブジェクトを生成する役割のインターフェイス
- PSR-17の全部のinterfaceをオールインワンにまとめている
- なんちゃらインターフェイスを要求する引数にこのオブジェクト渡せばいい

Nyholm/psr7のREADMEより

Create server requests

The `nyholm/psr7-server` package can be used to create server requests from PHP superglobals.

```
composer require nyholm/psr7-server
```

```
$psr17Factory = new \Nyholm\Psr7\Factory\Psr17Factory();

$creator = new \Nyholm\Psr7Server\ServerRequestCreator(
    $psr17Factory, // ServerRequestFactory
    $psr17Factory, // UriFactory
    $psr17Factory, // UploadedFileFactory
    $psr17Factory  // StreamFactory
);

$serverRequest = $creator->fromGlobals();
```


pixiv

PSR-7とフレームワークの関係

- PHP-FIGメンバーだからと言って全て従うわけではない
- SlimやLaminas mezzioはPSR-7を採用している
- CakePHP 4.xのHTTPメッセージ（cakephp/http）は
  PSR-7とCakePHP 3互換のインターフェイス両方を持っている
- symfony/http-foundationはPSR-7ではないが、symfony/psr-http-message-bridgeというアダプターを噛ますとLaravelでも使える

PHPの実行環境

pixiv

従来型のPHP実行環境

- HTTPリクエストを受け付けるたびに全てがリセットされる ⇒ 超短命
  - クラス定義・関数定義・変数・定数… あらゆるものがまっさらになる
- Apache(mod_php), PHP-FPM, ビルトインサーバ(php -S), CGIなど、既存のPHPが動いてきた環境のほとんどがこの動作をする
- どんな環境でもCGIと互換性のあるスクリプトが動いてきたのがPHPが長年しぶとく生きのびた秘訣。
  - Perl、RubyやPythonはCGIとの断絶を経験している

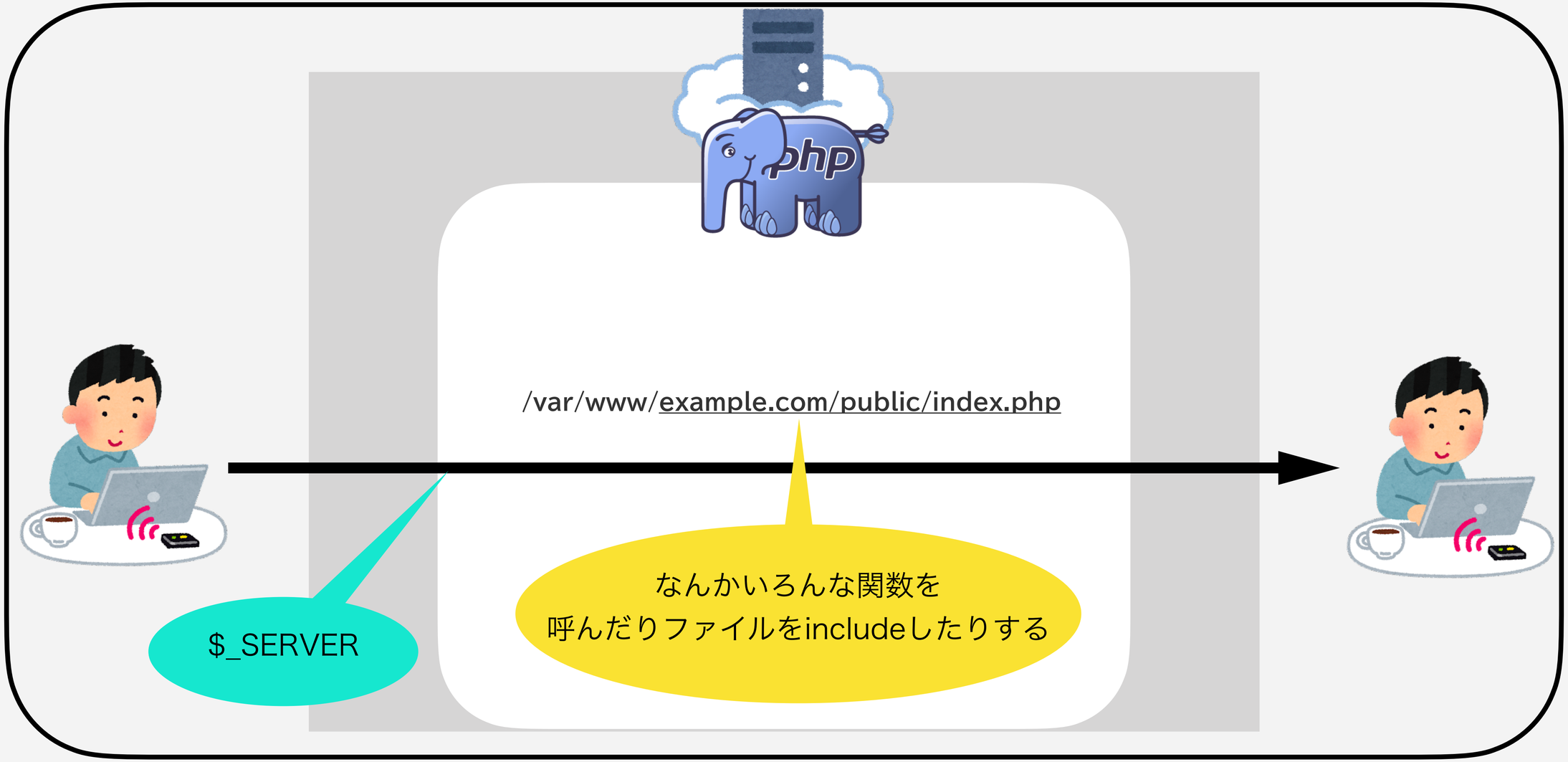
フレームワークを使わないPHP

- ユーザーからのHTTPリクエスト情報はスーパーグローバル変数に格納
  - 宣言なしでどこからでもアクセスできる特殊なグローバル変数
  - $_SERVER $_GET $_POST $_COOKIE $_FILES $_SESSION
  - リクエストボディはfile_get_contents('php://input')で取得可能
- header()関数などでセットしたものがHTTPレスポンスヘッダとして、echoしたものがHTTPレスポンスボディとして出力される

フレームワークを使わないPHP

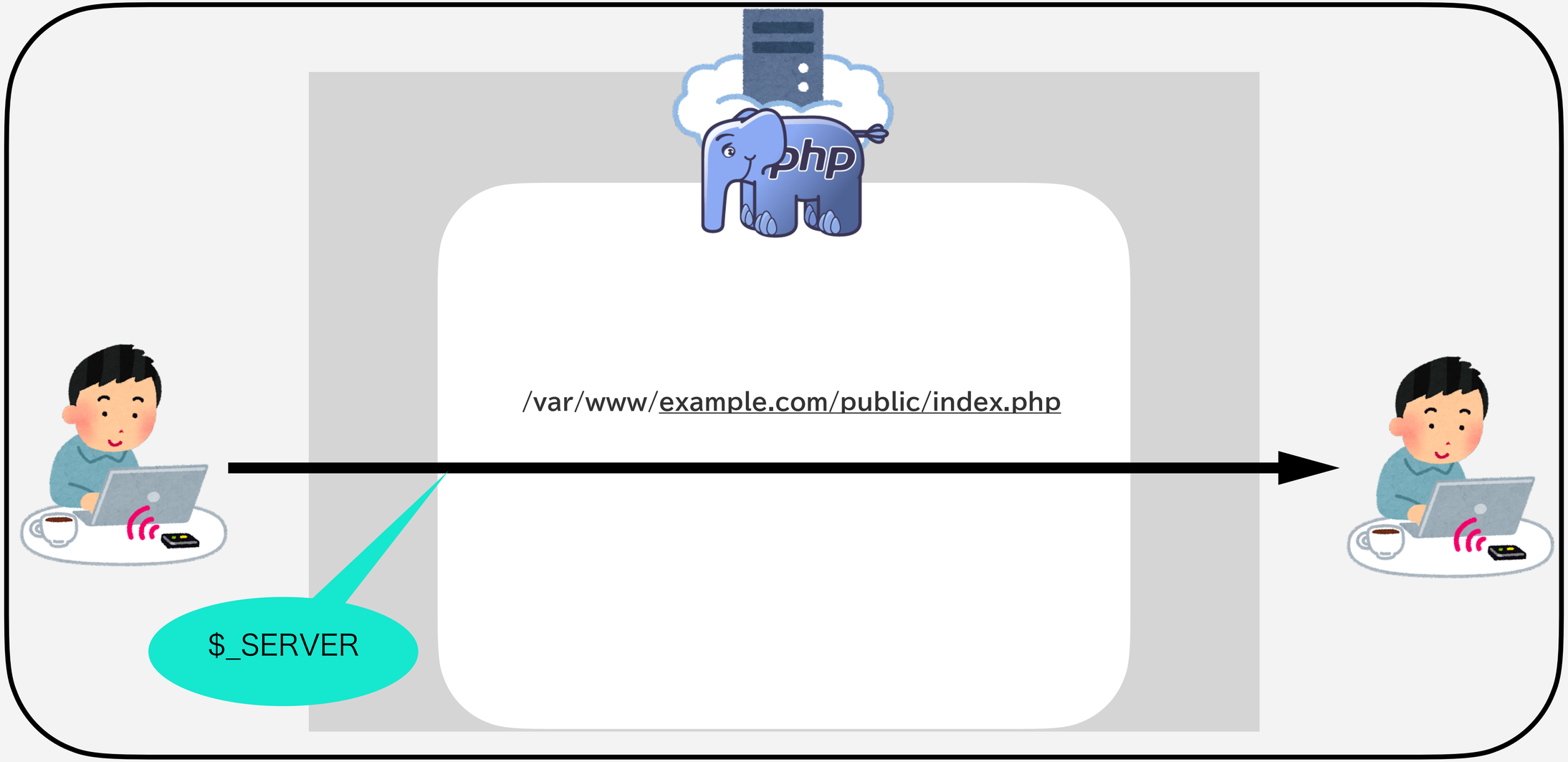
```
<?php
$name = $_GET['name'];
$title = "Hello, {$name}!";
header('Content-Type: application/html; charset=UTF-8');
?>
<title><?= htmlspecialchars($title) ?></title>
<h1><?= htmlspecialchars($title) ?></h1>

<p>現在は
<time><?= htmlspecialchars(date('Y年m月d日 H時i分s秒')) ?></time>
です</p>
```

Good old PHP



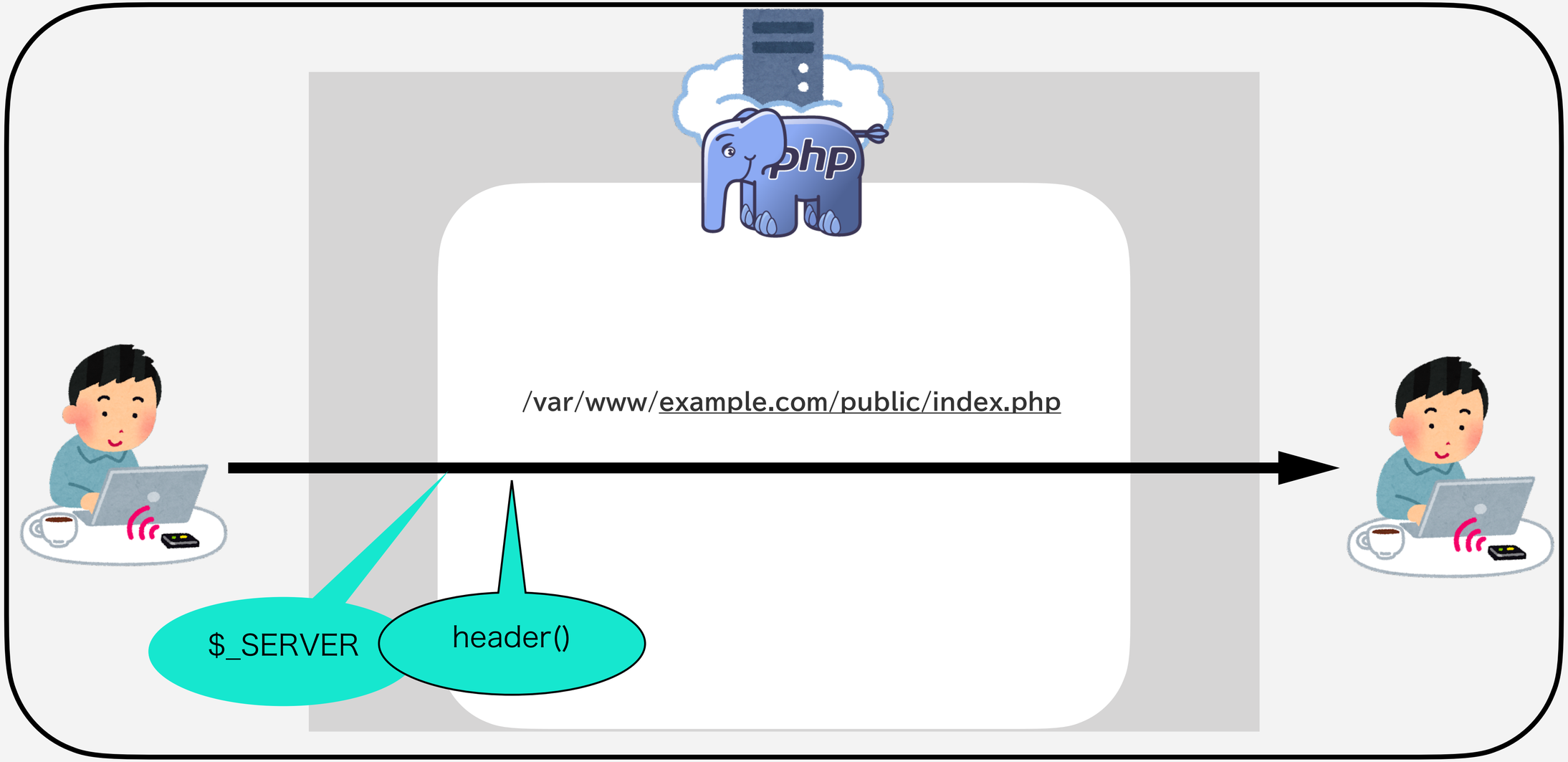
pixiv

Good old PHP



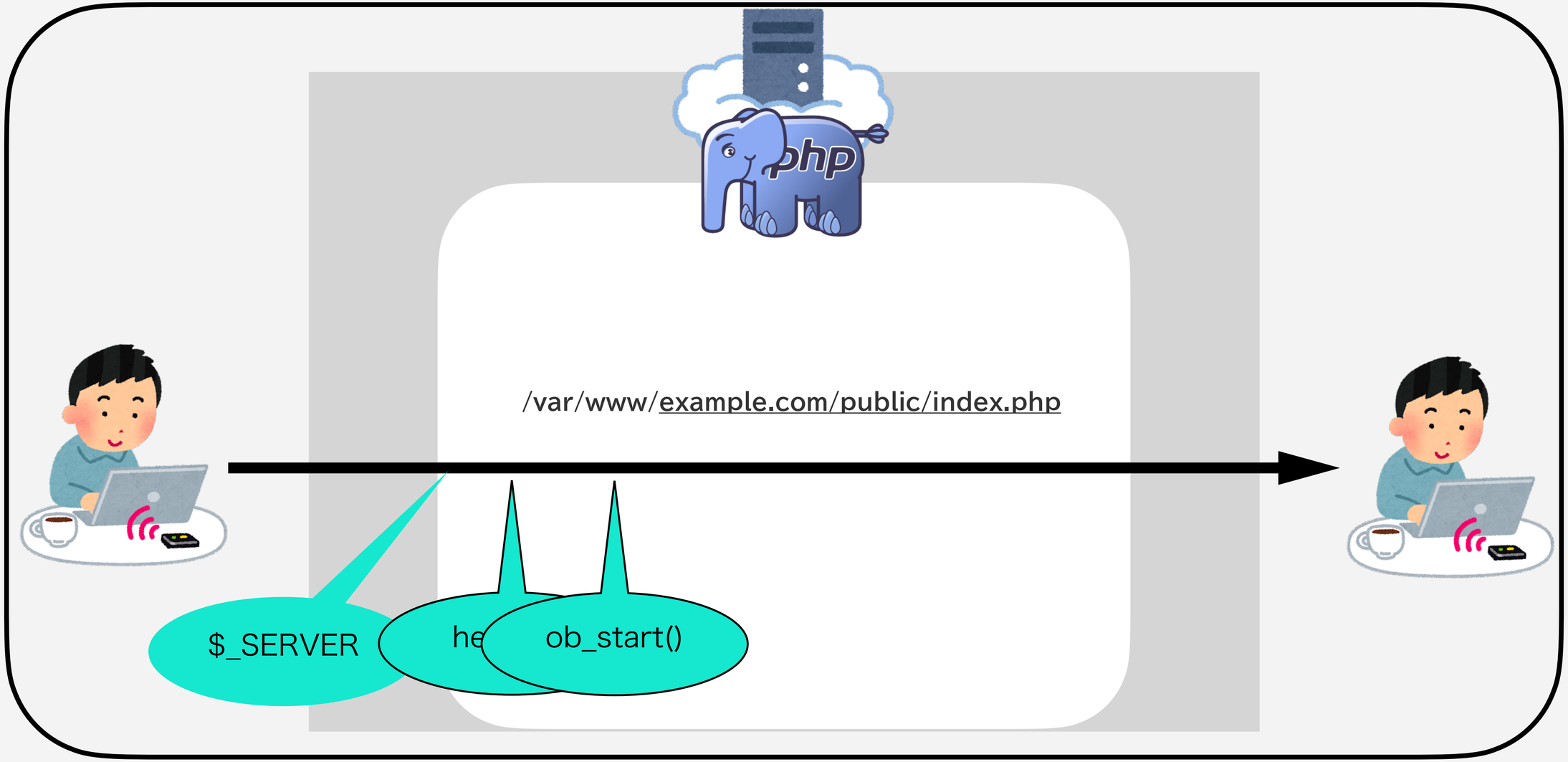
pixiv

Good old PHP



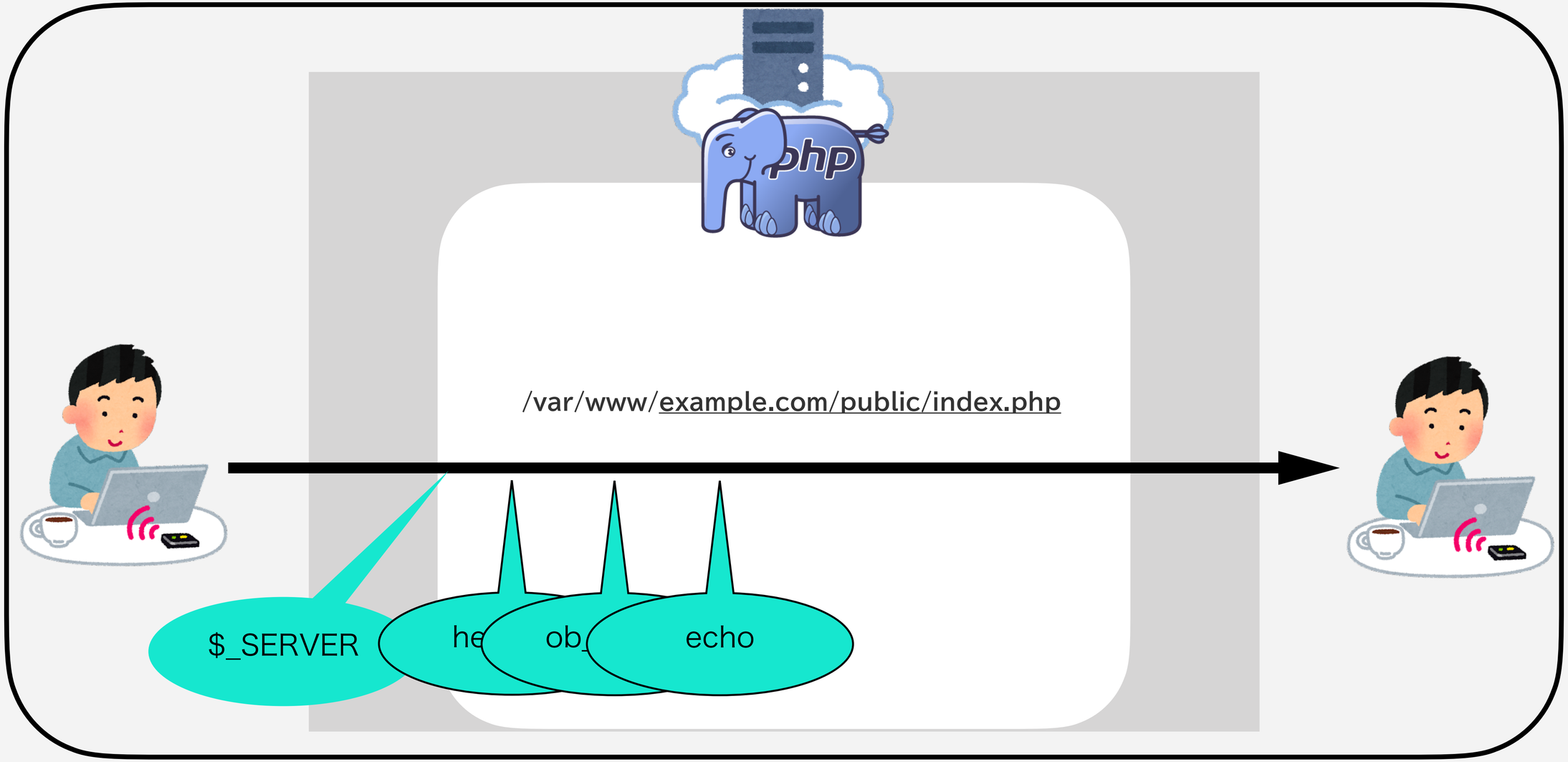
pixiv

Good old PHP



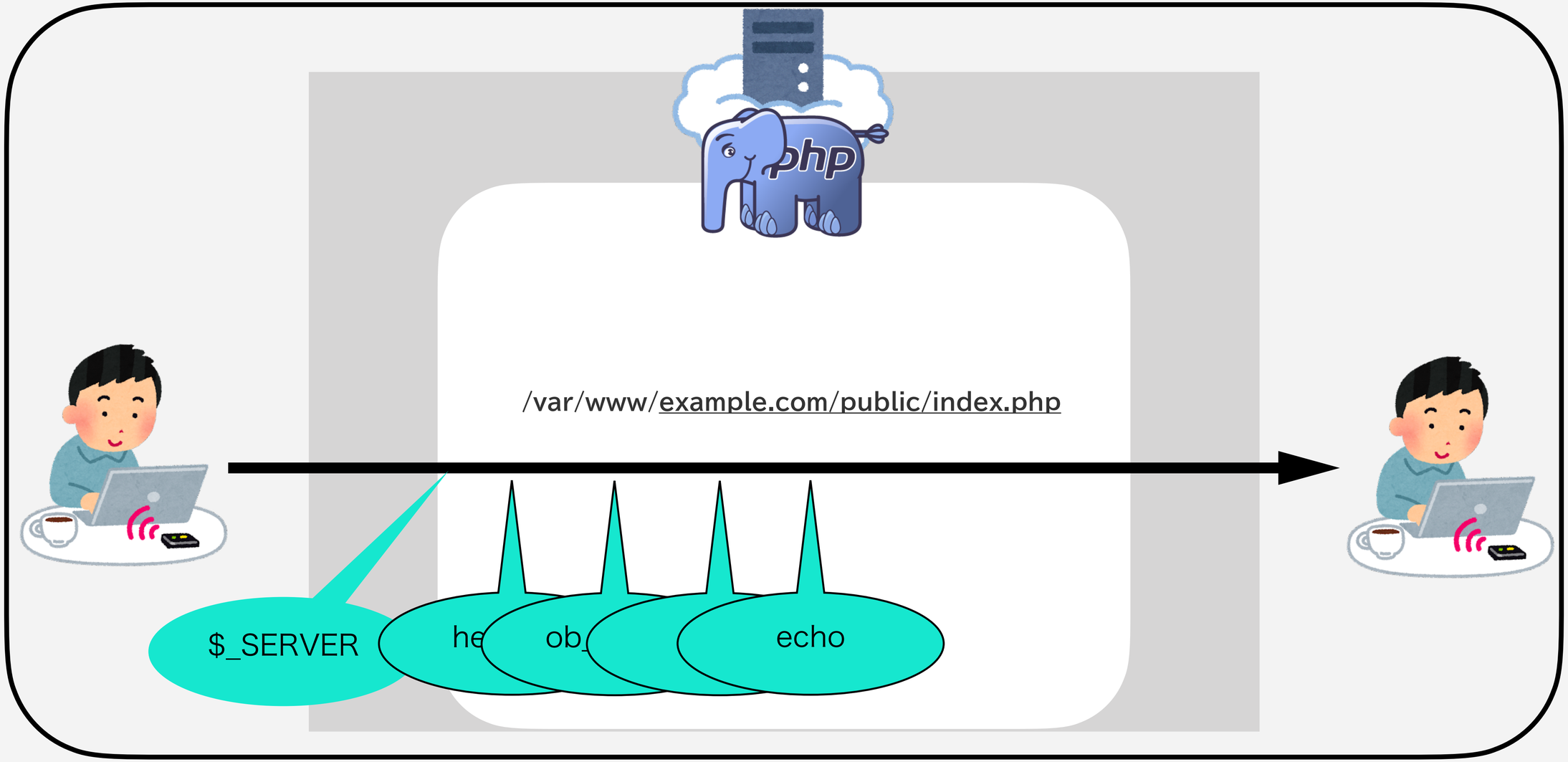
pixiv

Good old PHP



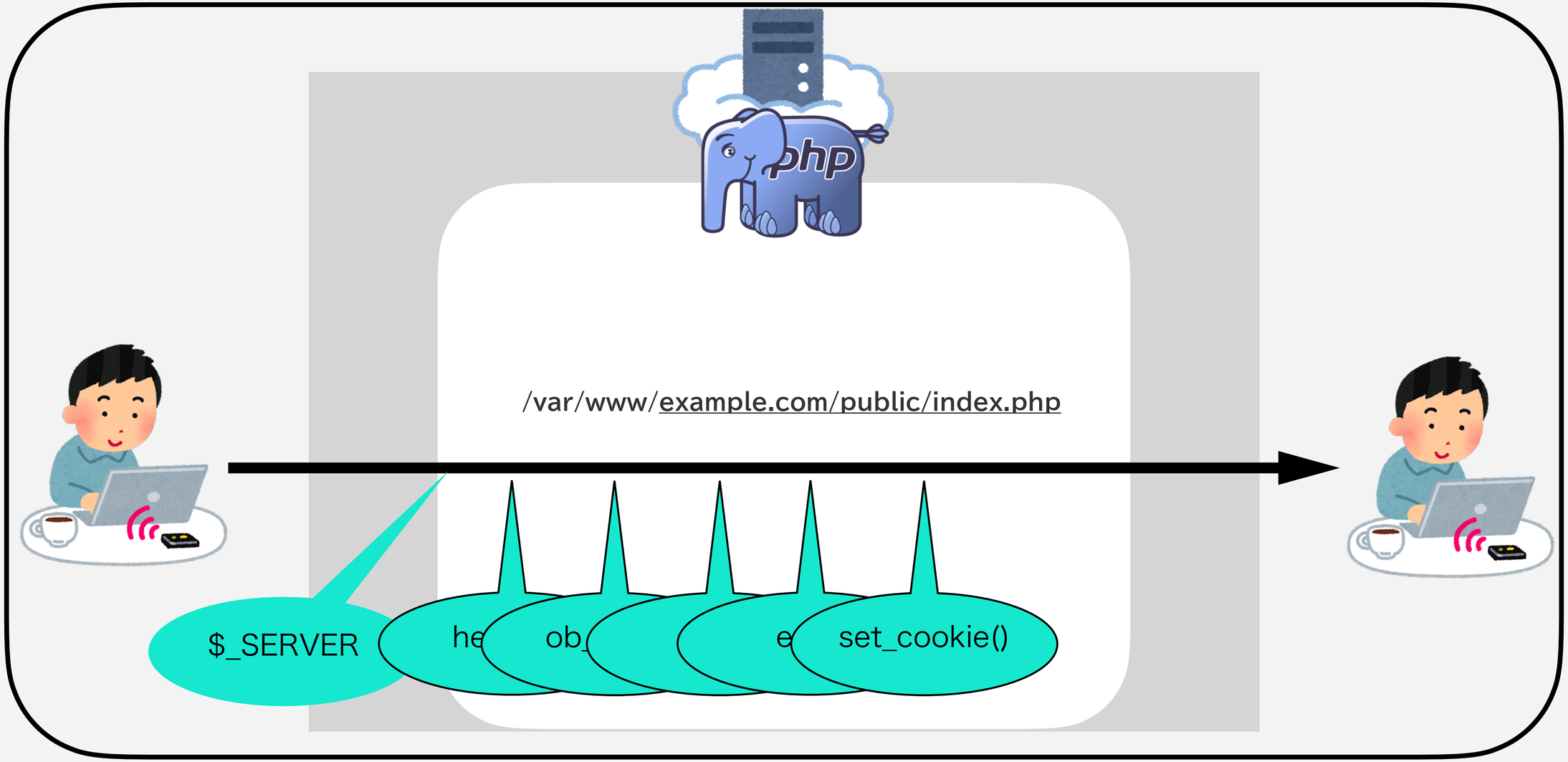
pixiv

Good old PHP



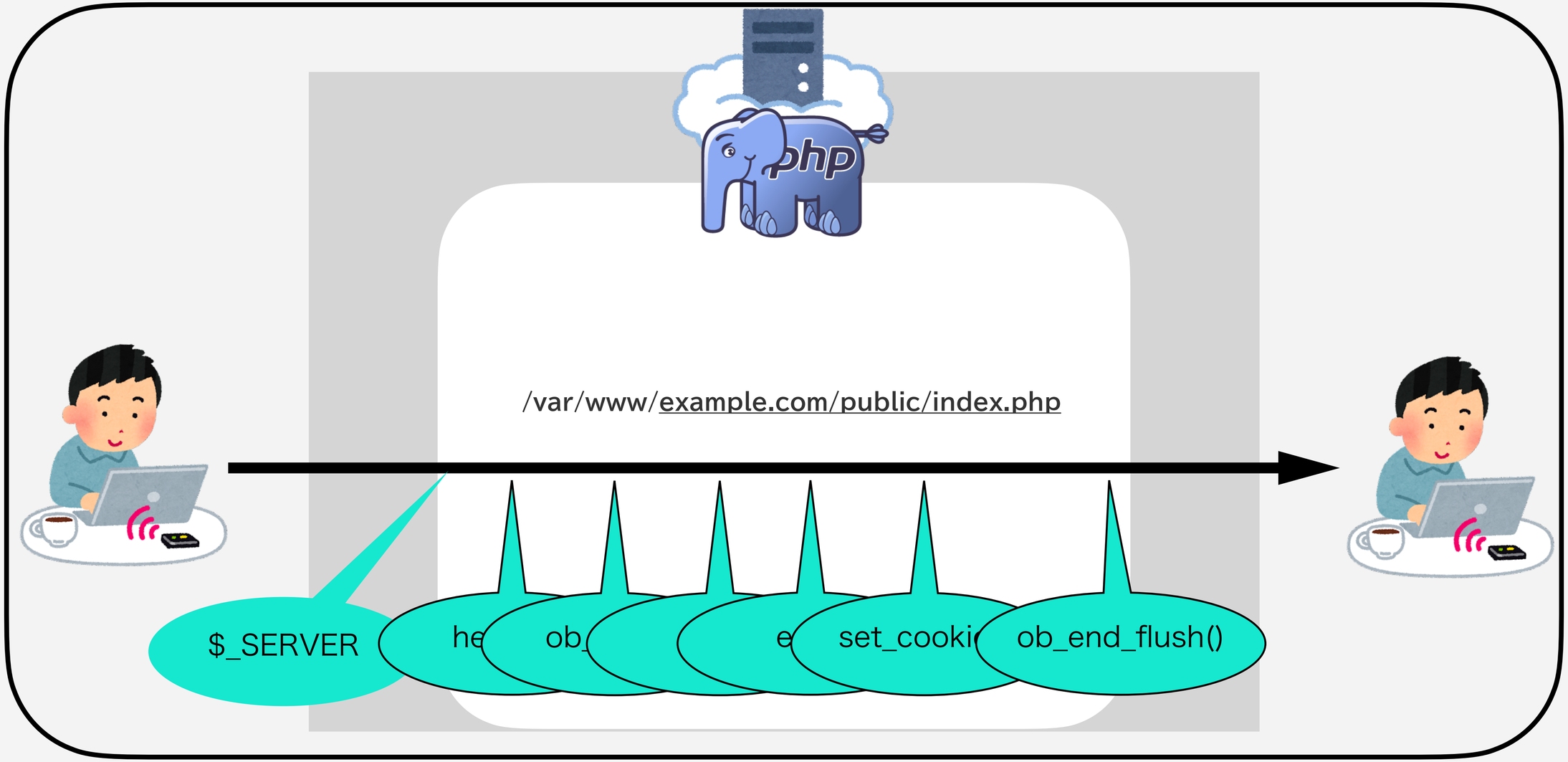
pixiv

Good old PHP



pixiv

Good old PHP



pixiv

よくある従来のPHPスクリプト(CGI型)

- どこでもユーザー入力をグローバル変数から参照する
  ($_GET, $_SERVER, $_COOKIEなど)
- どこでもheader()関数でHTTPヘッダを設定したり、
  どこでもechoしてHTMLを出力したりすることができる
  - 呼ばれた関数の処理の奥底で何気なくheader()や
    setcookie()関数を呼んでセットされてたりするやつ
  - 処理を全部追わないと最終的な出力 (レスポンスヘッダ・ボディ)がわからん

こういう世界と
どうやって整合性を
合わせるのか

PSR-7の出力 (emit)

- ResponseInterfaceを出力する役割（PSR-7の仕様外）
- ApacheやPHP-FPMで動かしているなら自作することも
  割と簡単にできるが、できればライブラリに任せた方がいい
- 従来型環境での定番は laminas/laminas-httphandlerrunner
  - SapiEmitterとSapiStreamEmitterの2種類がある
  - 一回で出力するか、リソースから読み込めた都度出力するかの違い

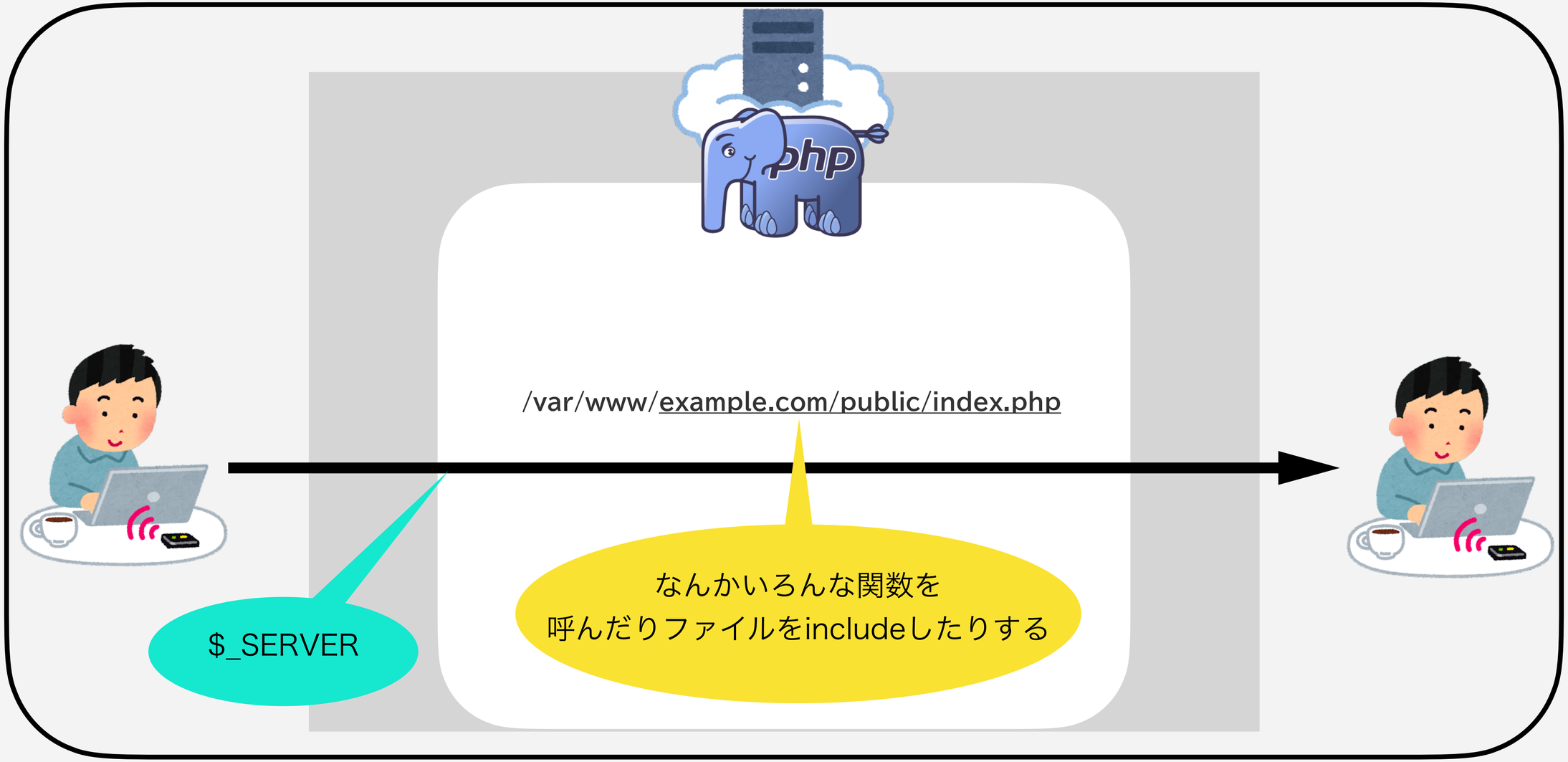
PSR-7の出力 (emit)

```php
// 概念的なざっくり実装。ほんとはもうちょっと考慮事項がある
function emitResponse(ResponseInterface $response): void
{
    http_response_code($response->getStatusCode());
    foreach ($response->getHeaders() as $name => $values) {
        foreach ((array)$values as $value) {
            header("{$name}: {$value}");
        }
    }

    echo $response->getBody();
}
```

PSR-7の入力 (ServerRequestの錬成)

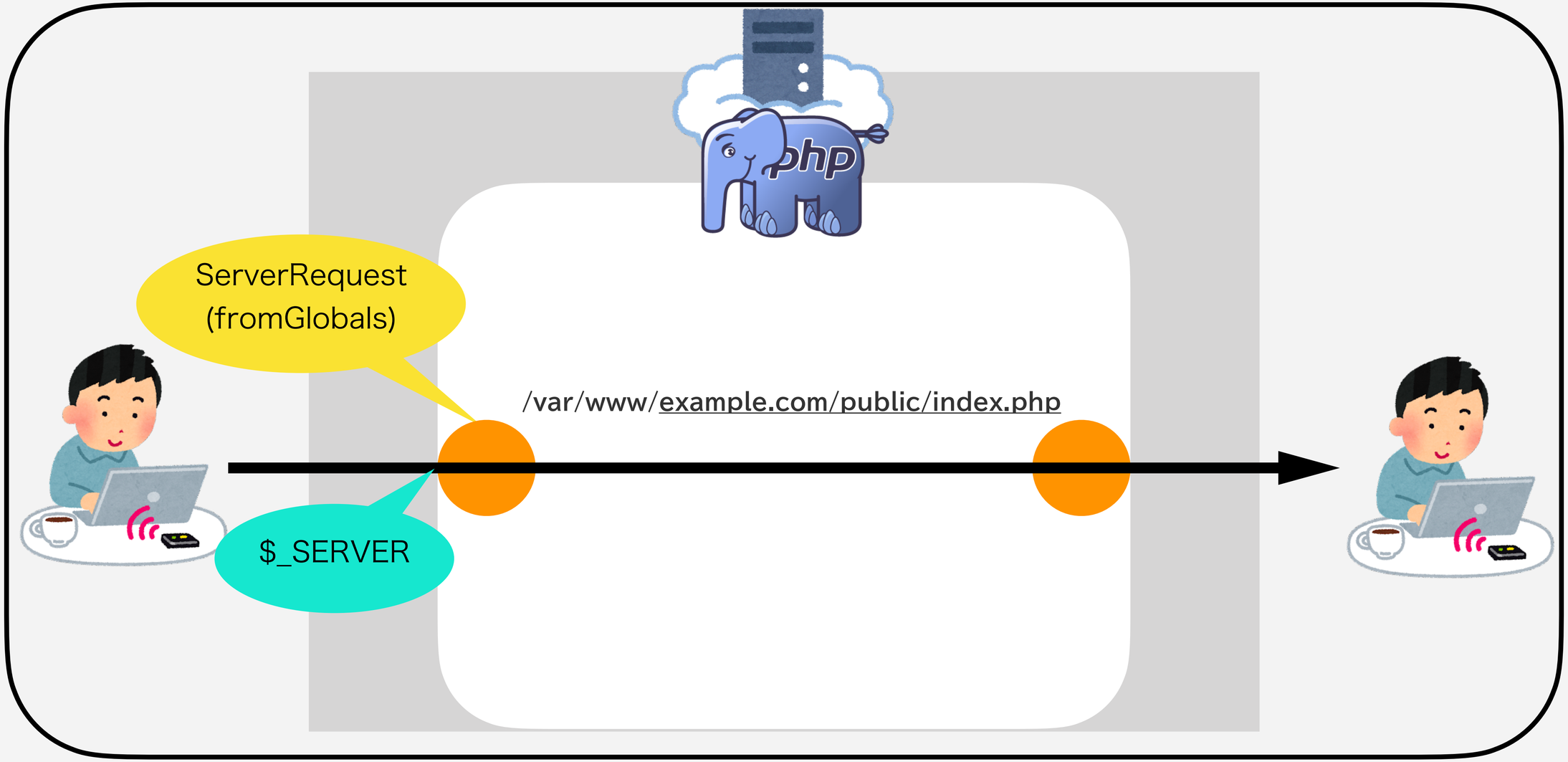
- ServerRequestはどこから来るの？
- 各PSR-7ライブラリとかがグローバル変数から情報をひっこぬいてServerRequestを作ってくれる便利クラスを提供してる（fromGlobals）
  - Guzzleの場合はServerRequest::fromGlobals()と呼ぶだけ
  - Nyholmはnyholm/psr7-serverという別パッケージに分離されてる
- 基本的に一箇所で一回だけ生成するようにした方がいい

Good old PHP



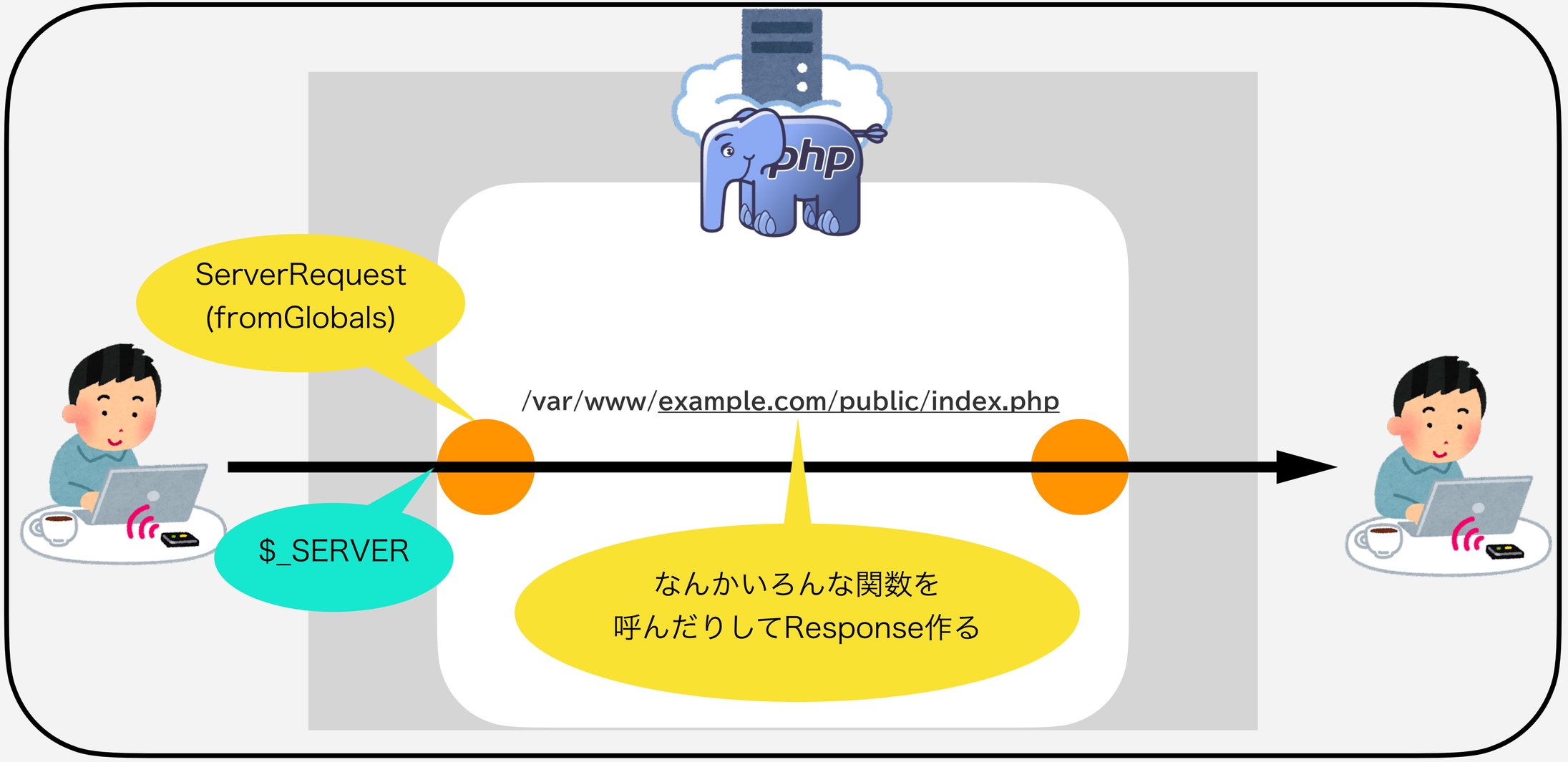
pixiv

従来型環境でPSR-7を使う概念



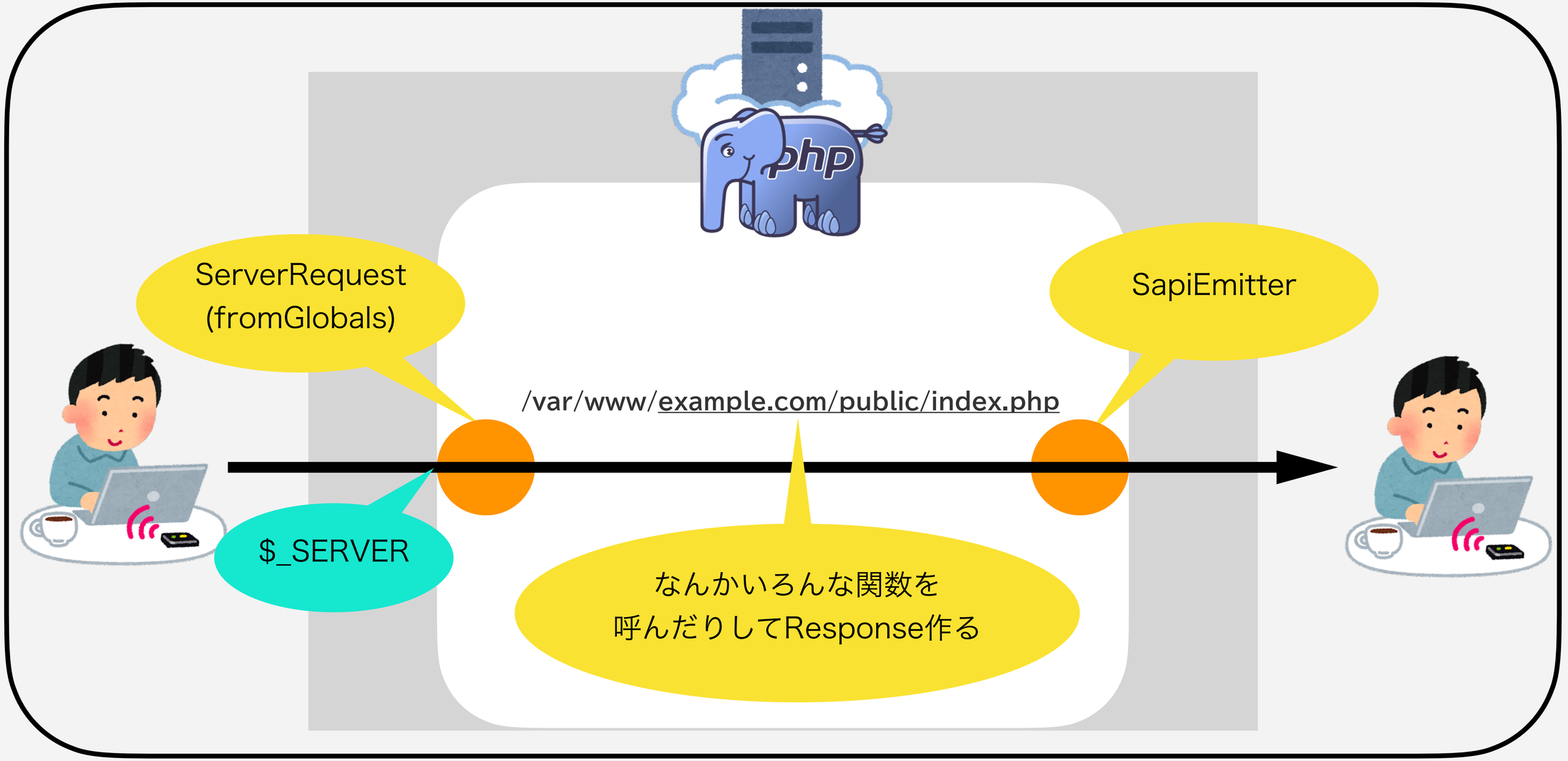
pixiv

従来型環境でPSR-7を使う概念



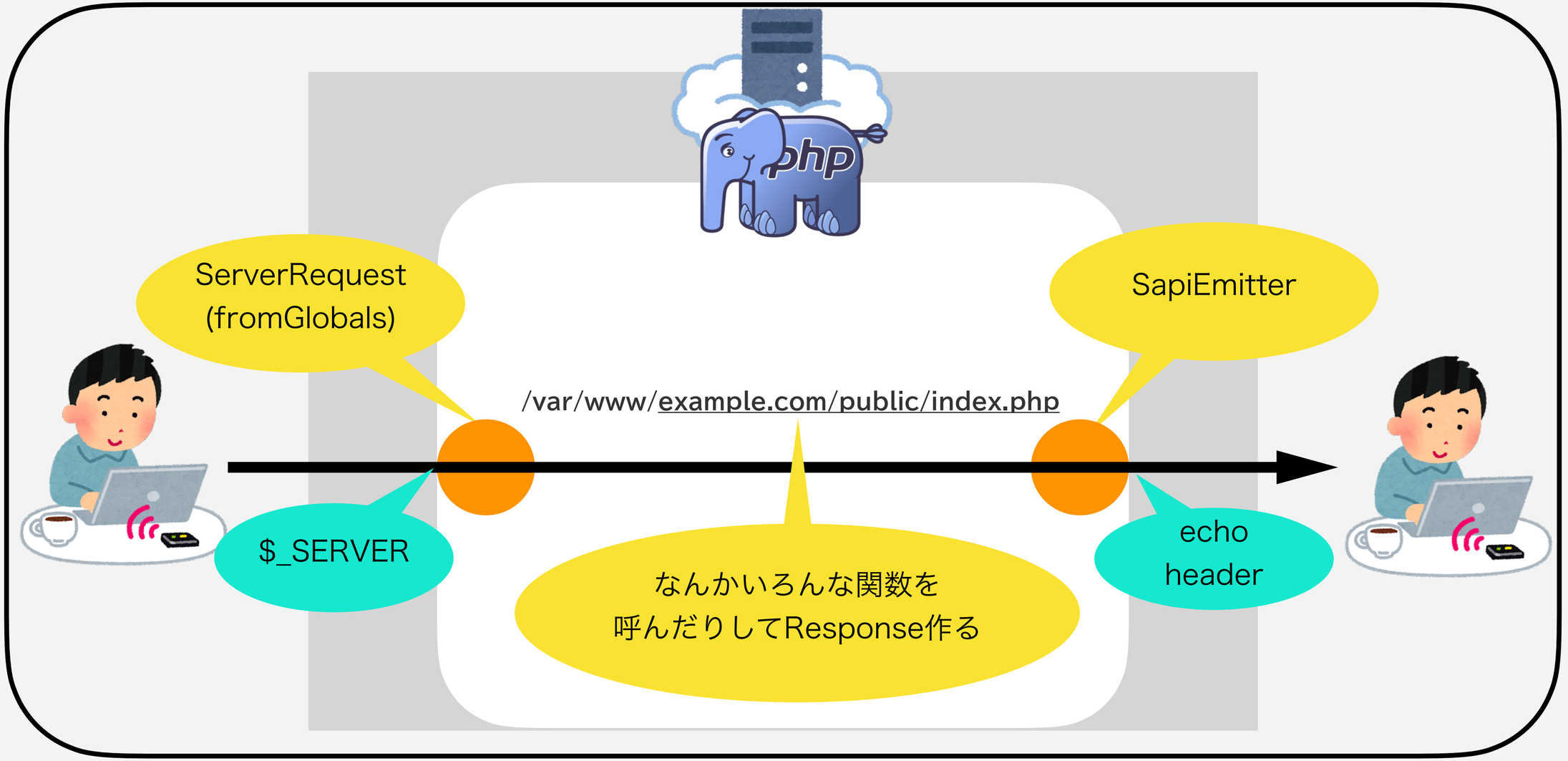
pixiv

従来型環境でPSR-7を使う概念



pixiv

従来型環境でPSR-7を使う概念



pixiv

フレームワークの光と影

- そもそもPHPはフレームワークがなくてもWebアプリが書ける

フレームワークの光と影

- そもそもPHPはフレームワークがなくてもWebアプリが書ける
- PHPにとってのWebフレームワークには二つの性質がある

フレームワークの光と影

- そもそもPHPはフレームワークがなくてもWebアプリが書ける
- PHPにとってのWebフレームワークには二つの性質がある
  - PHP標準ではめんどくさい機能を実現する便利ライブラリの集合体

フレームワークの光と影

- そもそもPHPはフレームワークがなくてもWebアプリが書ける
- PHPにとってのWebフレームワークには二つの性質がある
  - PHP標準ではめんどくさい機能を実現する便利ライブラリの集合体
  - フリーダムにどこでもなんでも書けてしまうPHPの機能に規則を与えることで秩序をもたらす拘束具

フレームワークの光と影

- そもそもPHPはフレームワークがなくてもWebアプリが書ける
- PHPにとってのWebフレームワークには二つの性質がある
  - PHP標準ではめんどくさい機能を実現する便利ライブラリの集合体
  - フリーダムにどこでもなんでも書けてしまうPHPの機能に規則を与えることで秩序をもたらす拘束具
- PSR-7/15は従来型のPHPにとって縛りプレイのための異物に過ぎない

縛ることによって得られるもの

- $_GETのようなグローバル変数やheader()のような関数を排斥することで入出力が明確に分離されてコードの見通しがよくなり、テストしやすくなる
- Long-livingなPHPとの親和性
  - 従来のPHPはリクエストごとに全ての状態がリセットされるのに対して、RoadRunnerやSwooleでHTTPサーバを立てると毎回リセットされない
  - そのような環境ではCLI SAPIで動作するので、$_GETや関数からリクエスト情報が得られないし、echoしてもHTTPレスポンスにならない

Long-living PHPとの互換性

- 入口と出口、つまりServerRequestを作る部分とエミッタを差し替えればApacheやPHP-FPMなど従来型のPHP環境でも動かせるようになる
- 最近はフレームワークが実行環境の差異を吸収してくれるようになった
  - Symfony Runtime + https://github.com/php-runtime
  - Laravel Octane

CyberAgent白井さんの発表

Developer Conference2022　Concept　Session　Timetable　YouTube　#CADC2022

14:00〜 40分 + ライブQ&A | バックエンド

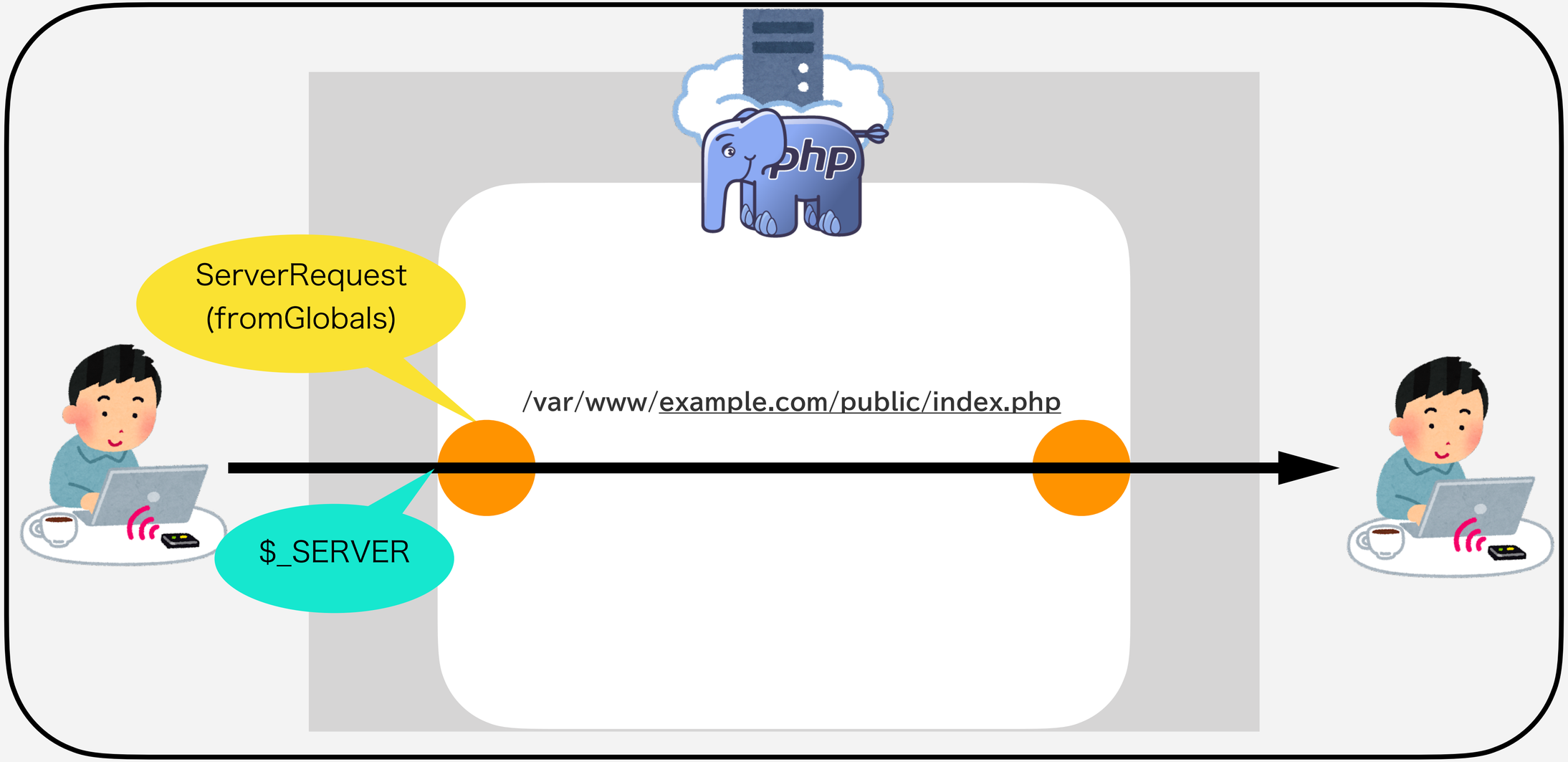
スマホゲーム業界におけるPHPの歴史とLaravel Octaneで広がるこれからのPHP

pixiv

Long-living PHPの世界

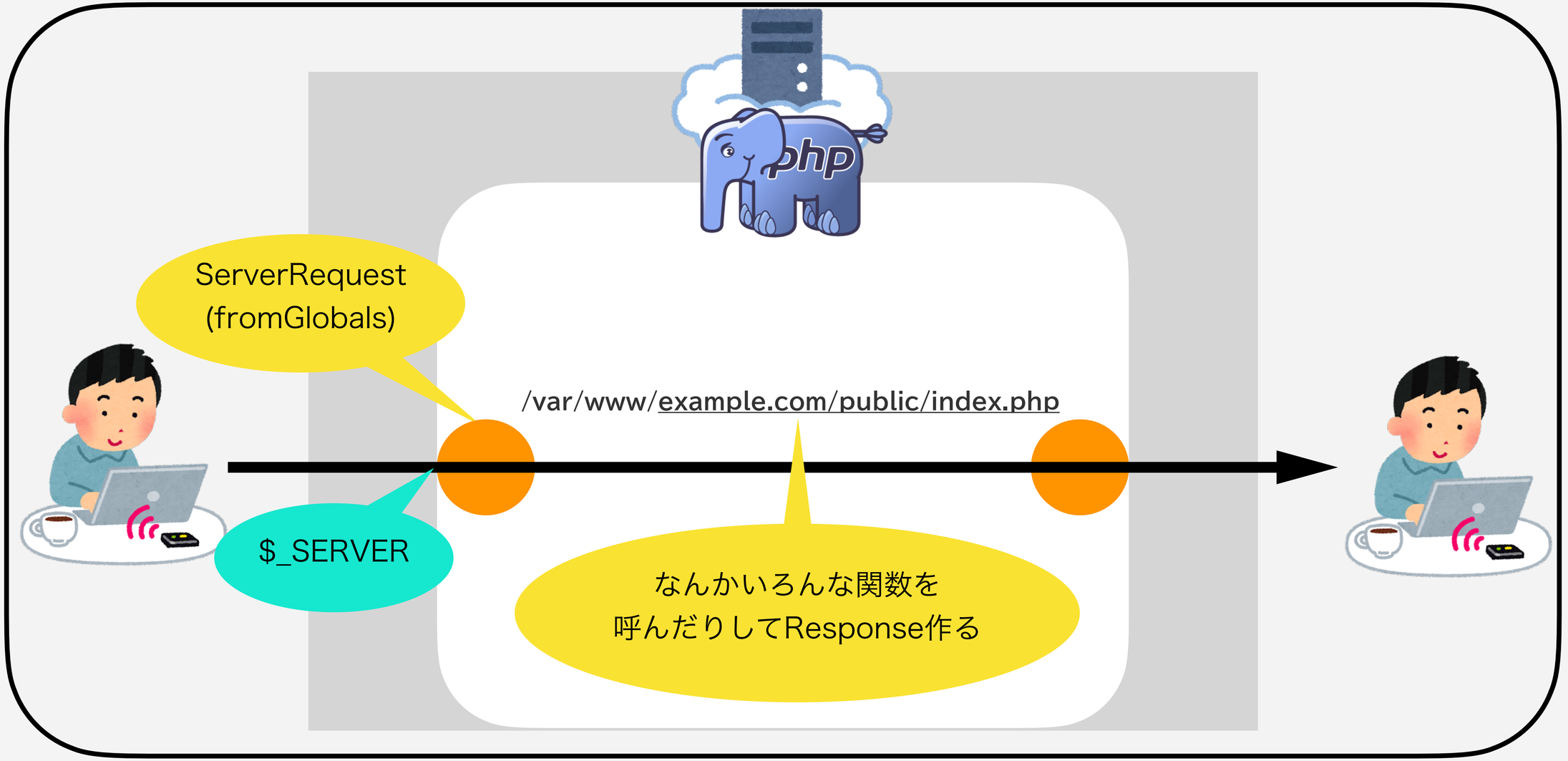
- 従来環境とはオブジェクトの初期化や生存期間の概念が変わるので注意
- $_SERVER, $_GETのようなグローバル変数や
  header()、setcookie()のような関数が今まで通りの動きをしない
  - Laravelでもこれらの機能を使うことはできたが、動かなくなる
- 定数、グローバル変数、静的プロパティなどもリセットされなくなる
- グローバルな状態を持たず引数と戻り値に閉じていれば問題ない

従来型環境でPSR-7を使う概念



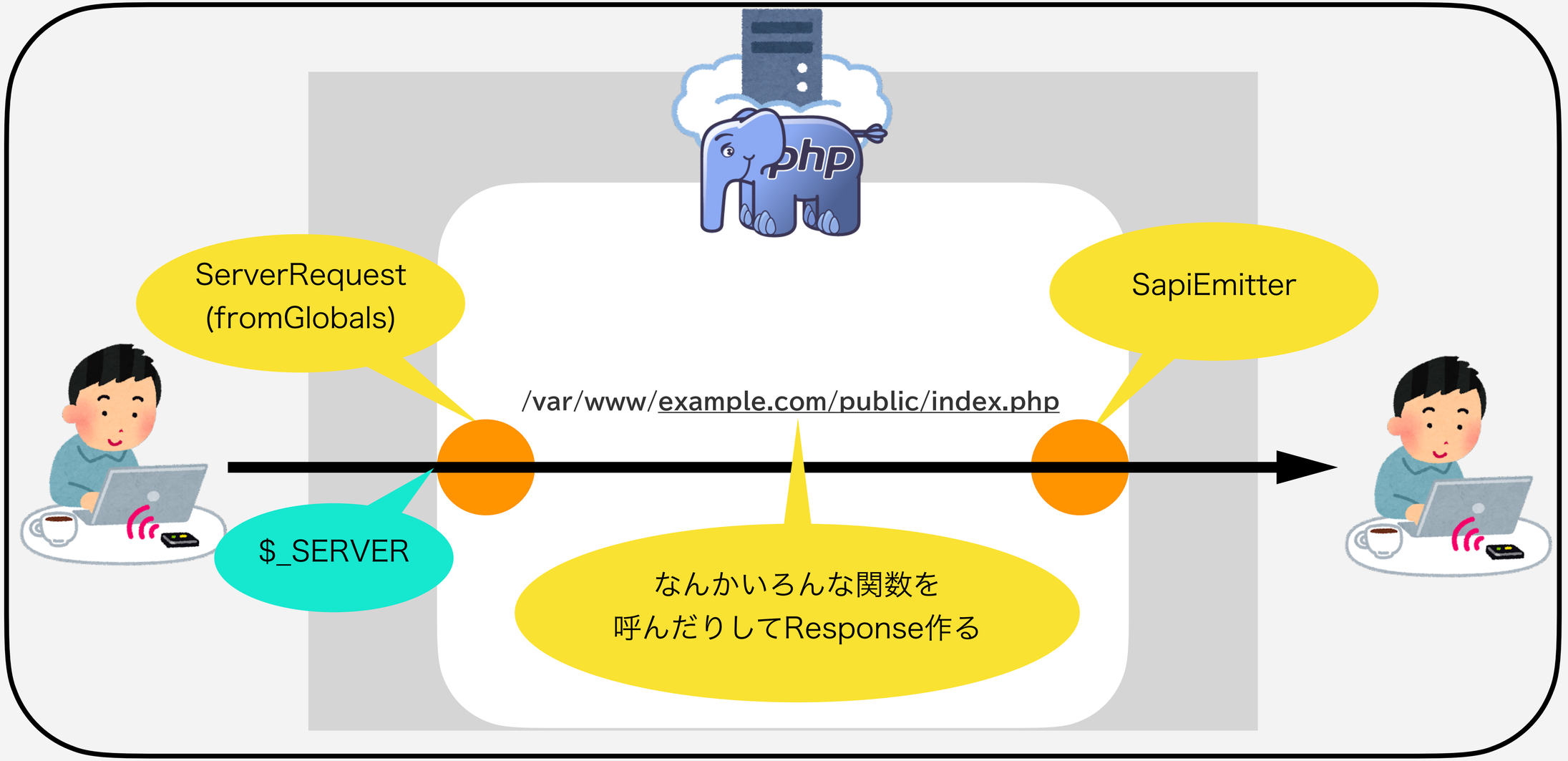
pixiv

従来型環境でPSR-7を使う概念



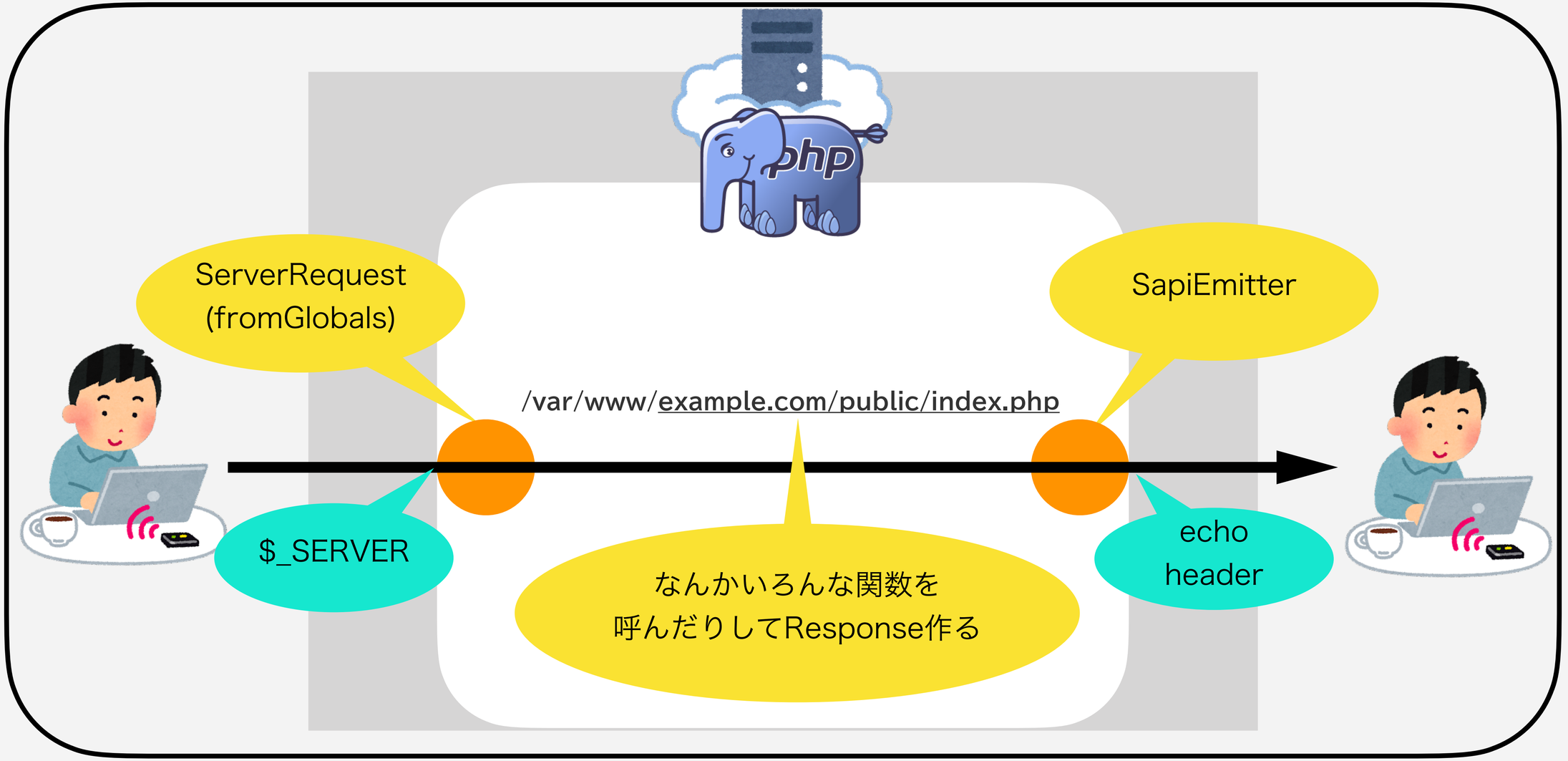
pixiv

従来型環境でPSR-7を使う概念



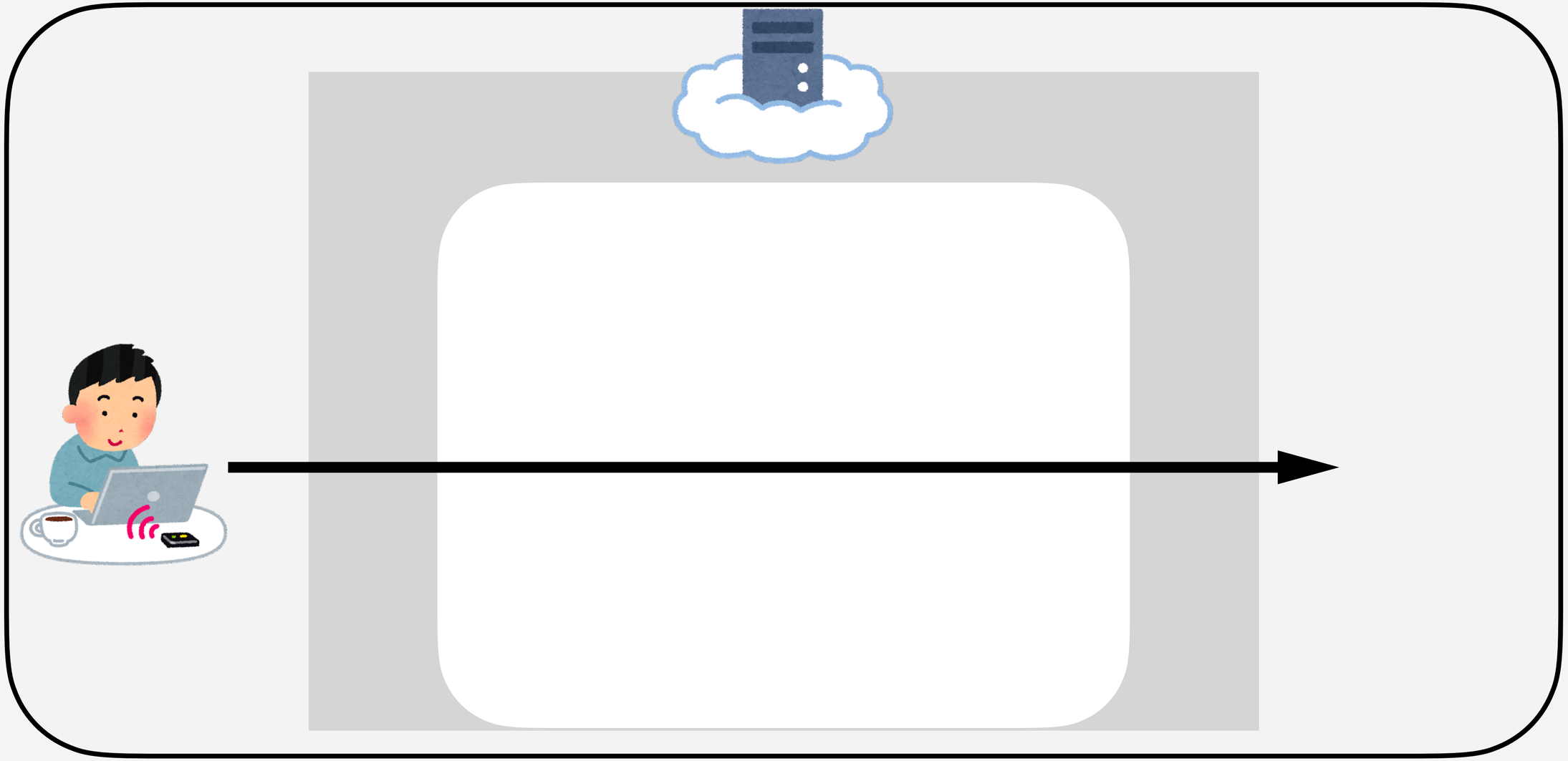
pixiv

従来型環境でPSR-7を使う概念



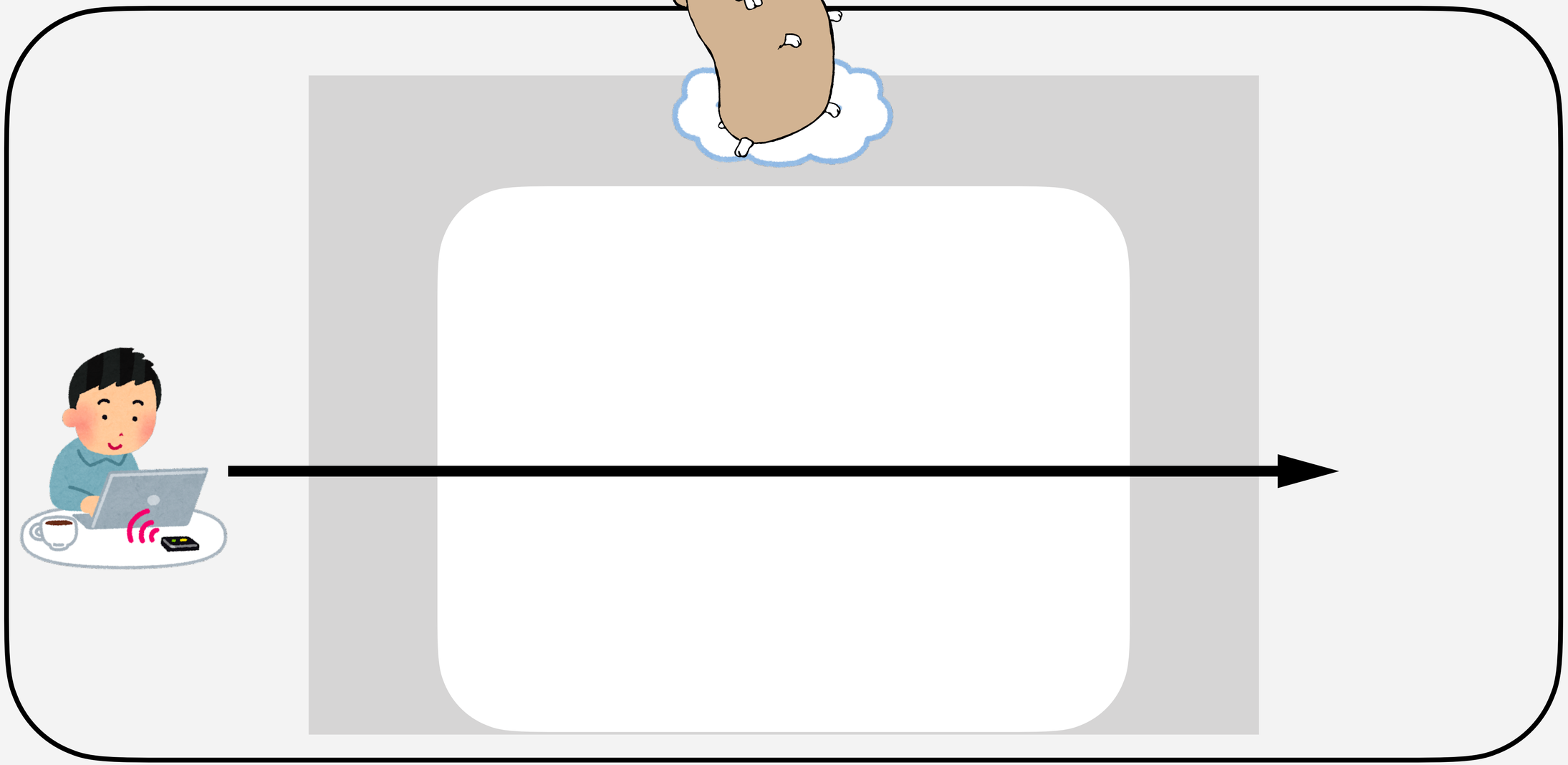
pixiv

RoadRunner



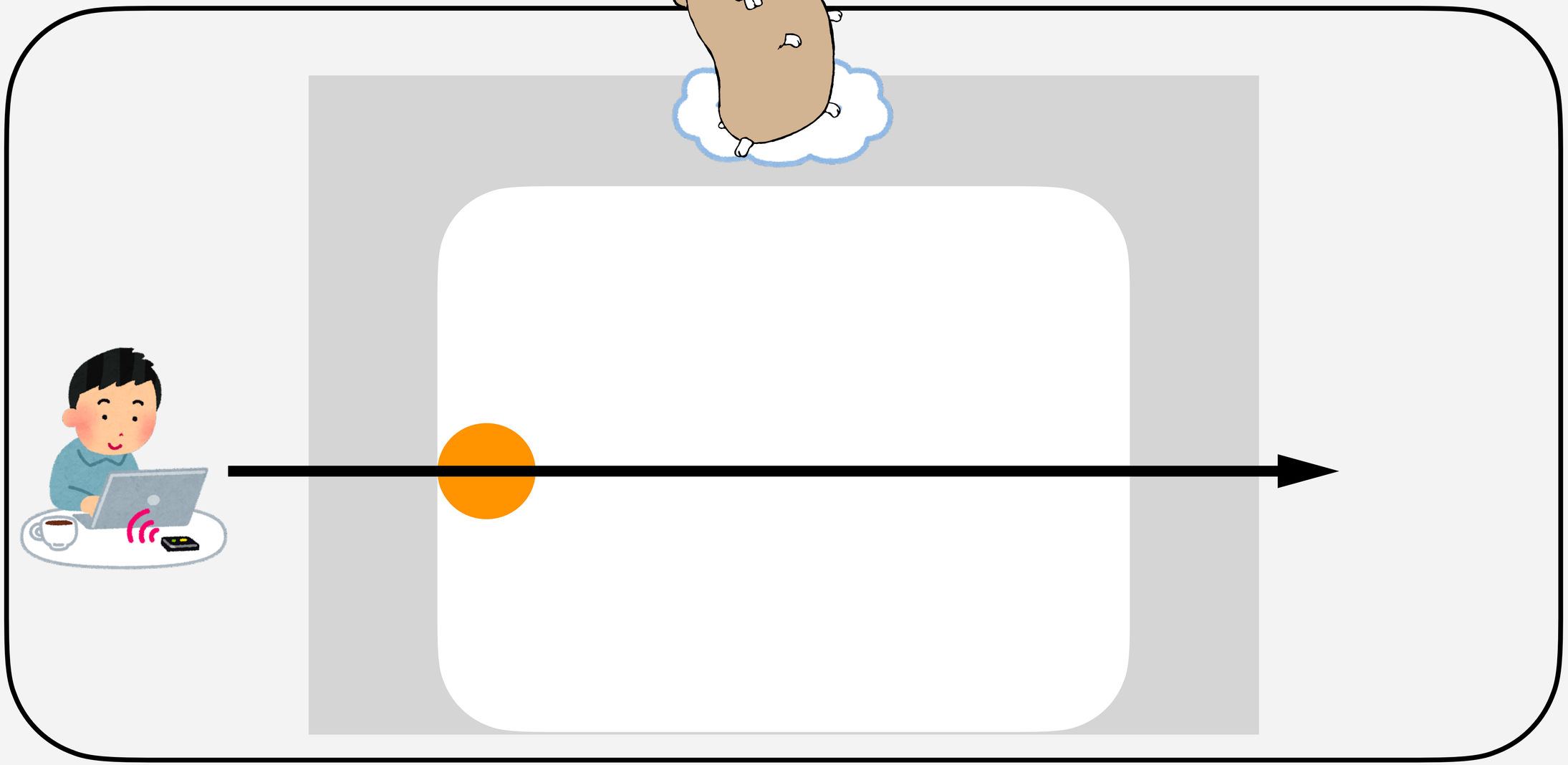
pixiv

RoadRunner



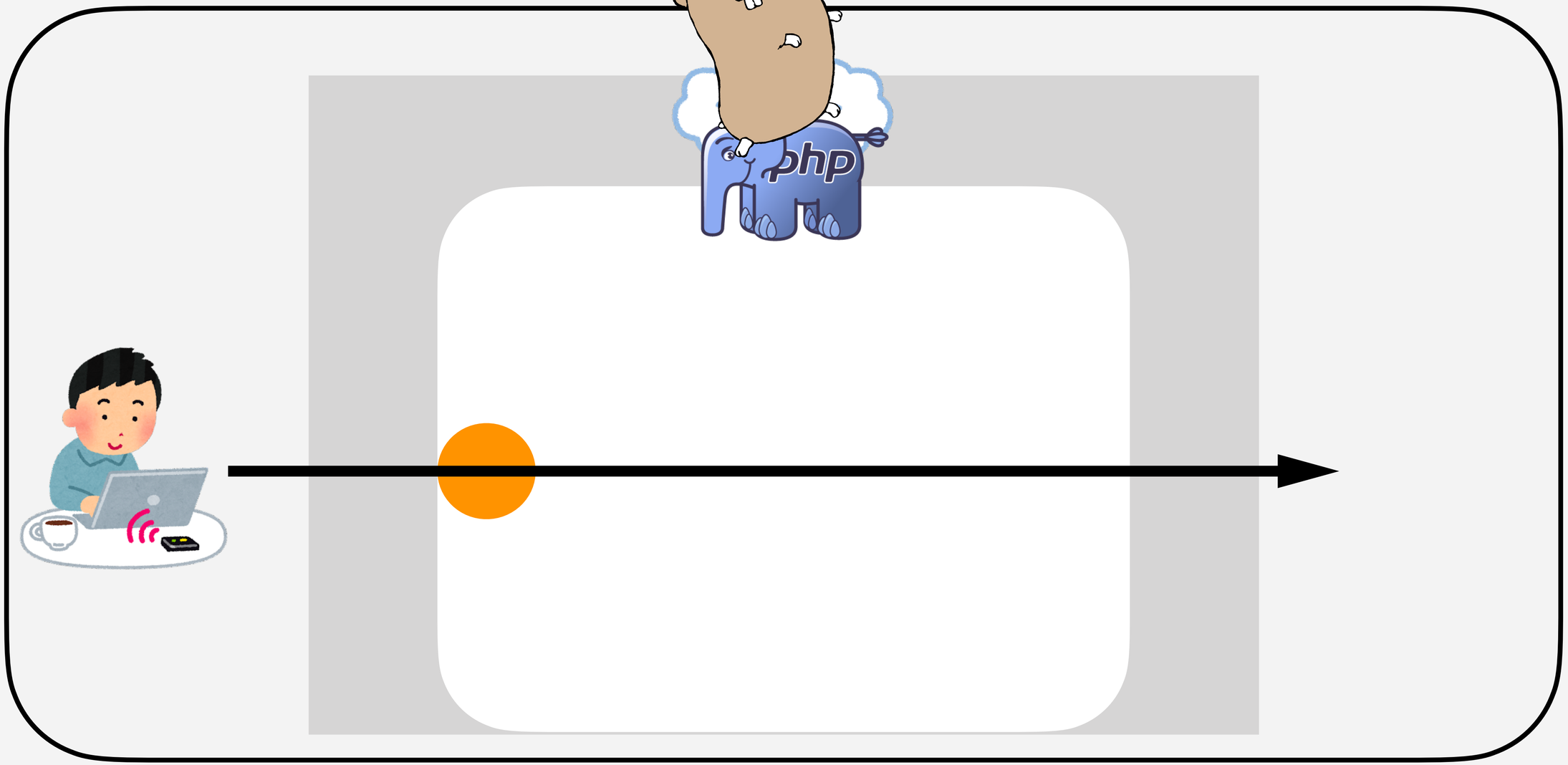
pixiv

RoadRunner



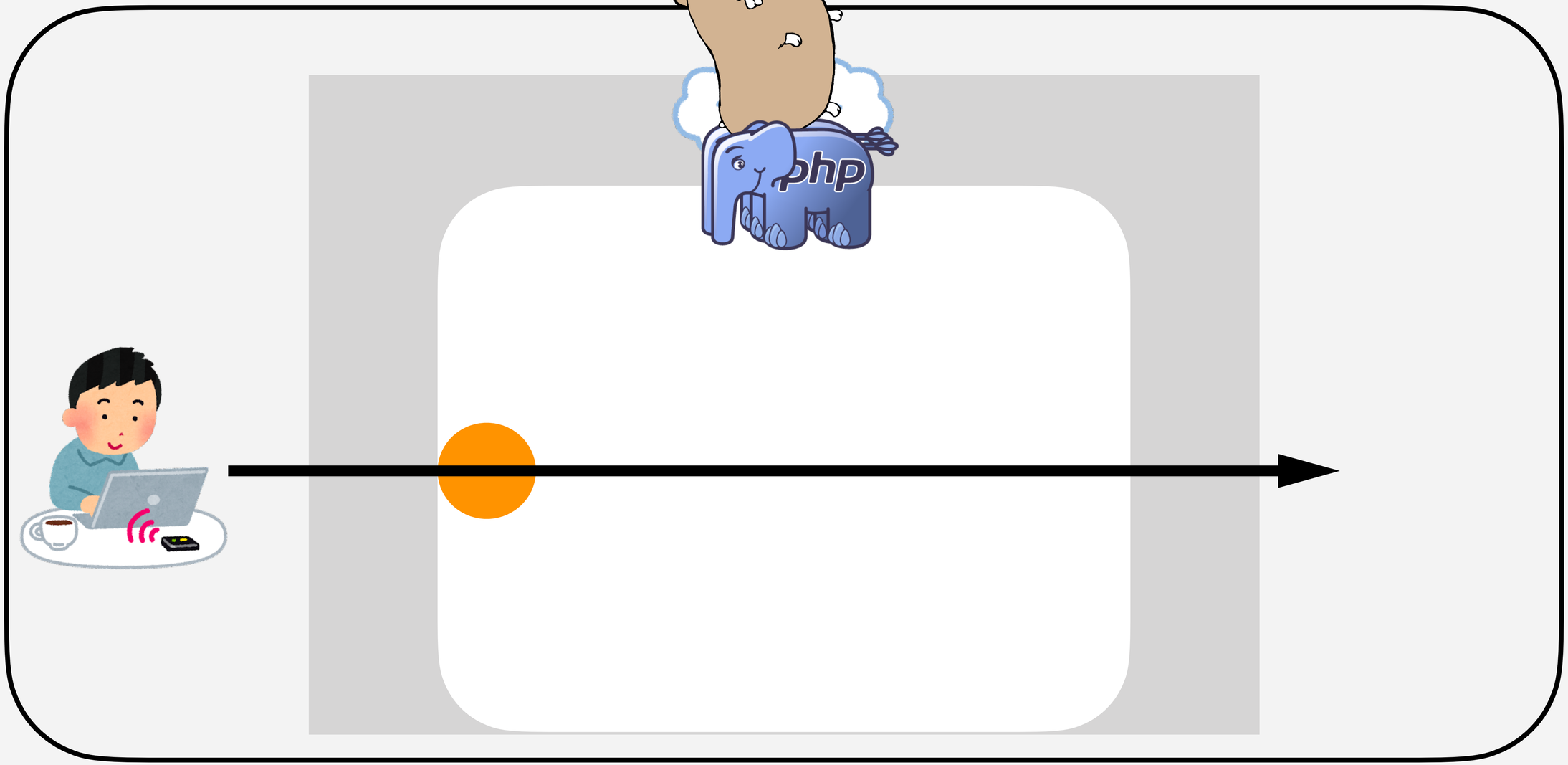
pixiv

RoadRunner



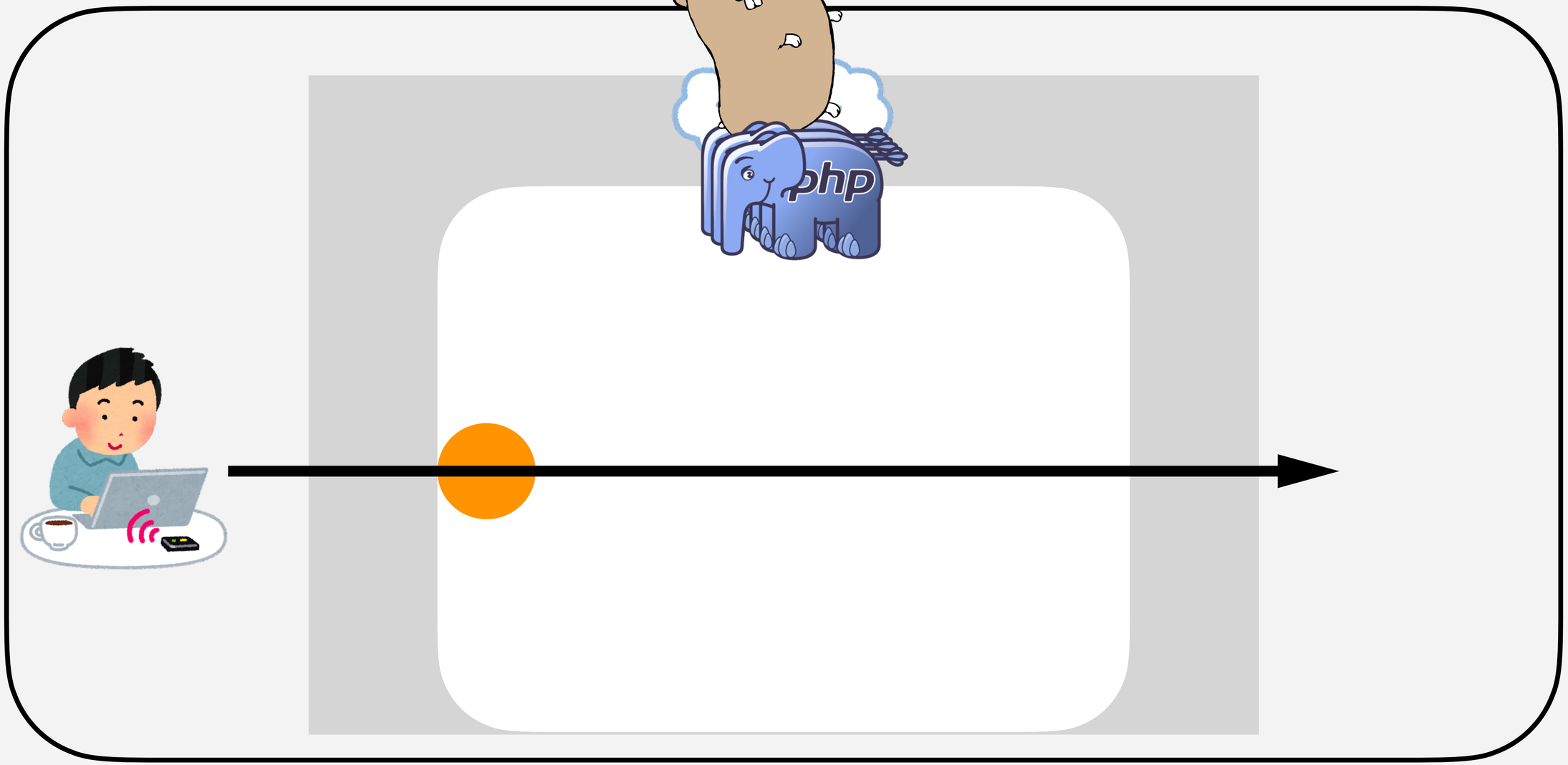
pixiv

RoadRunner



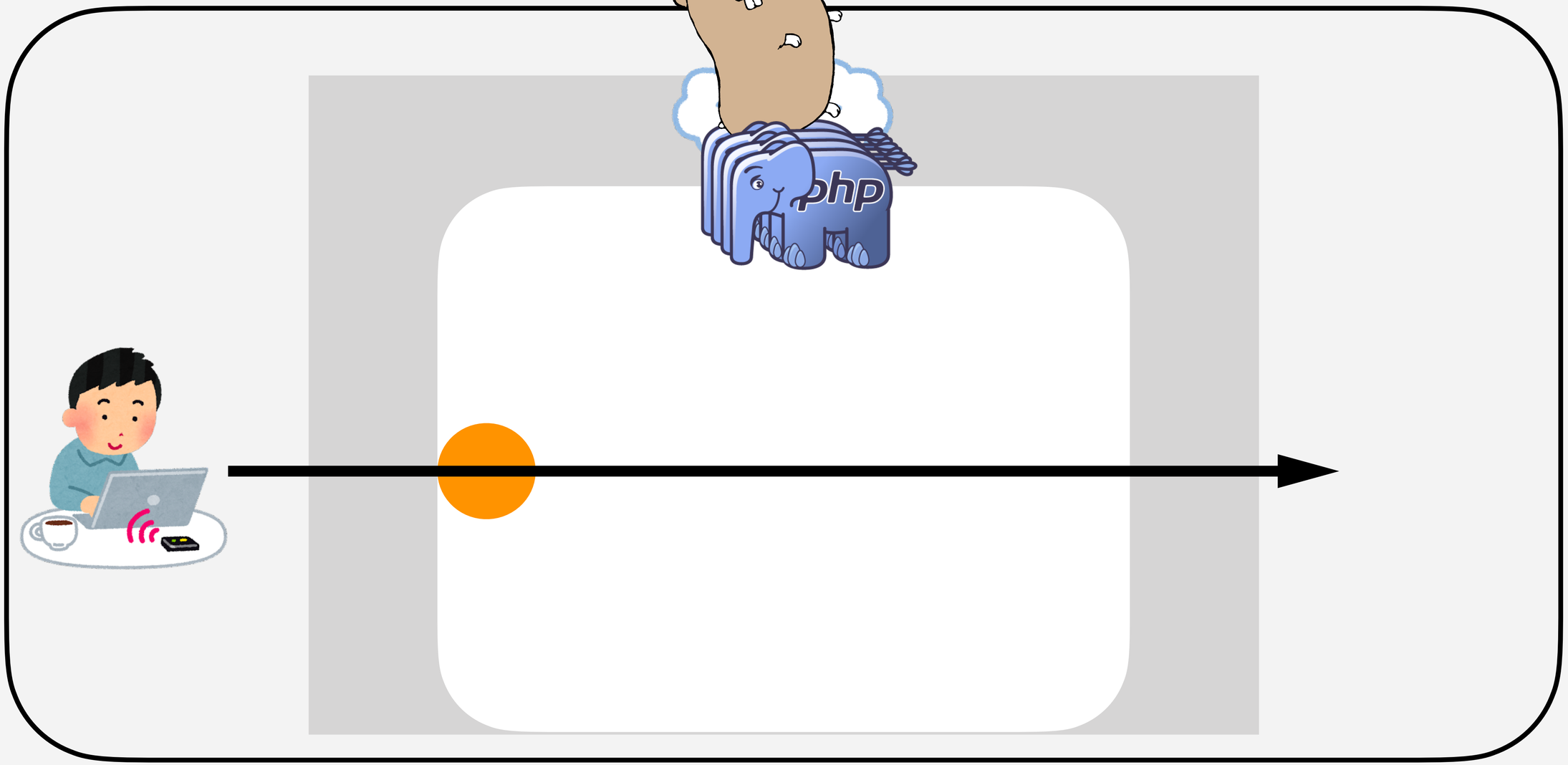
pixiv

RoadRunner



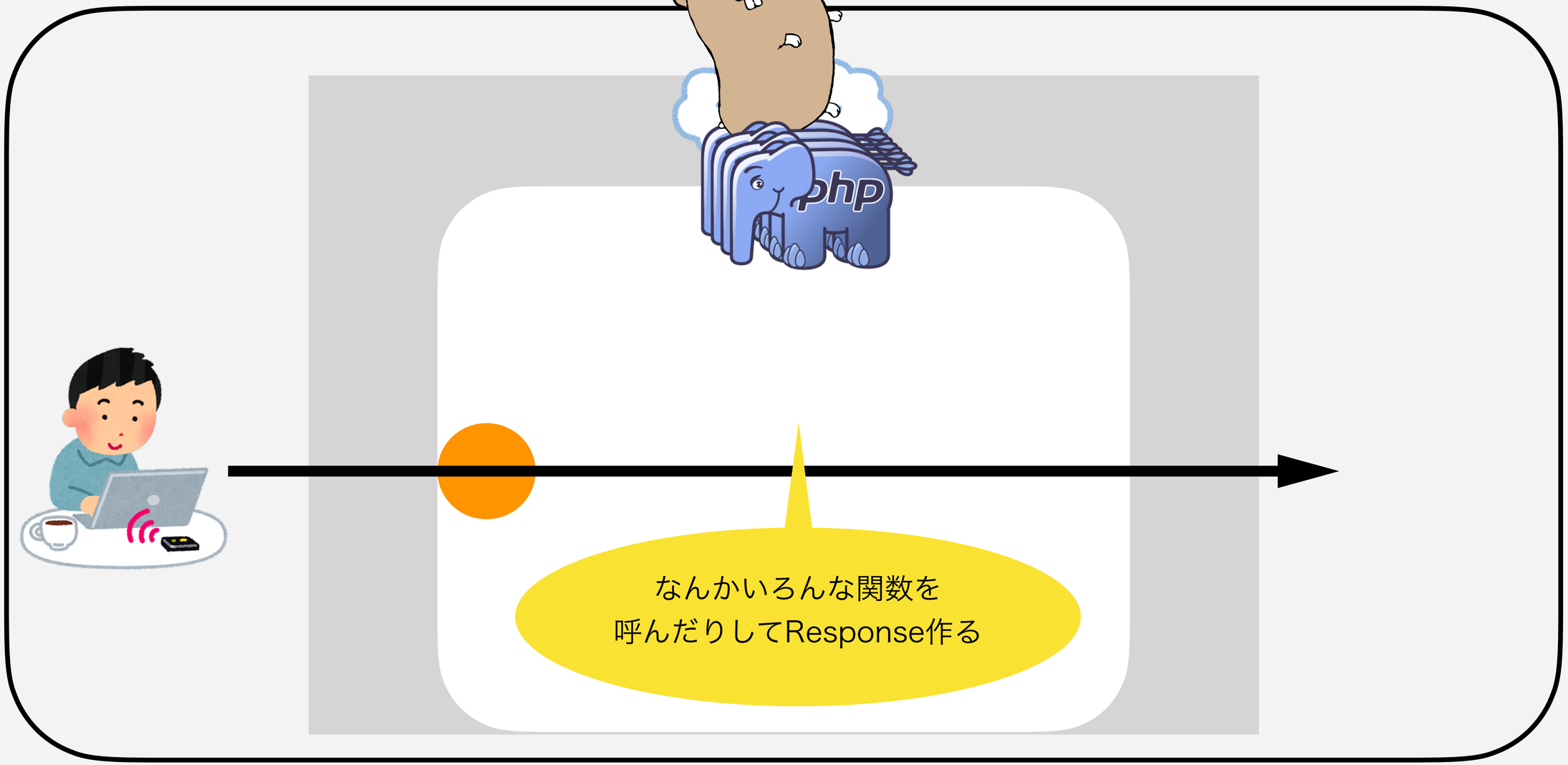
pixiv

RoadRunner



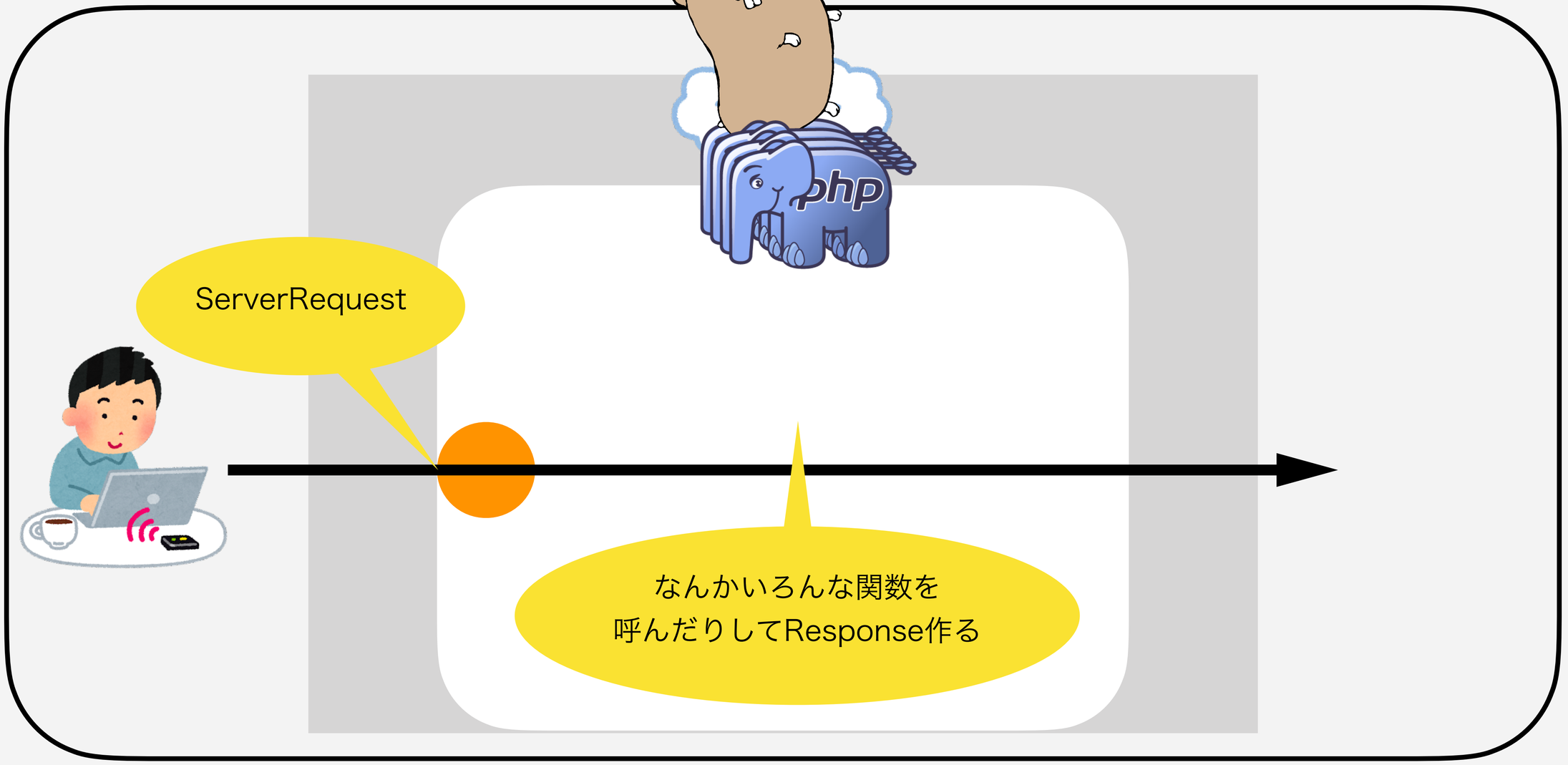
pixiv

RoadRunner



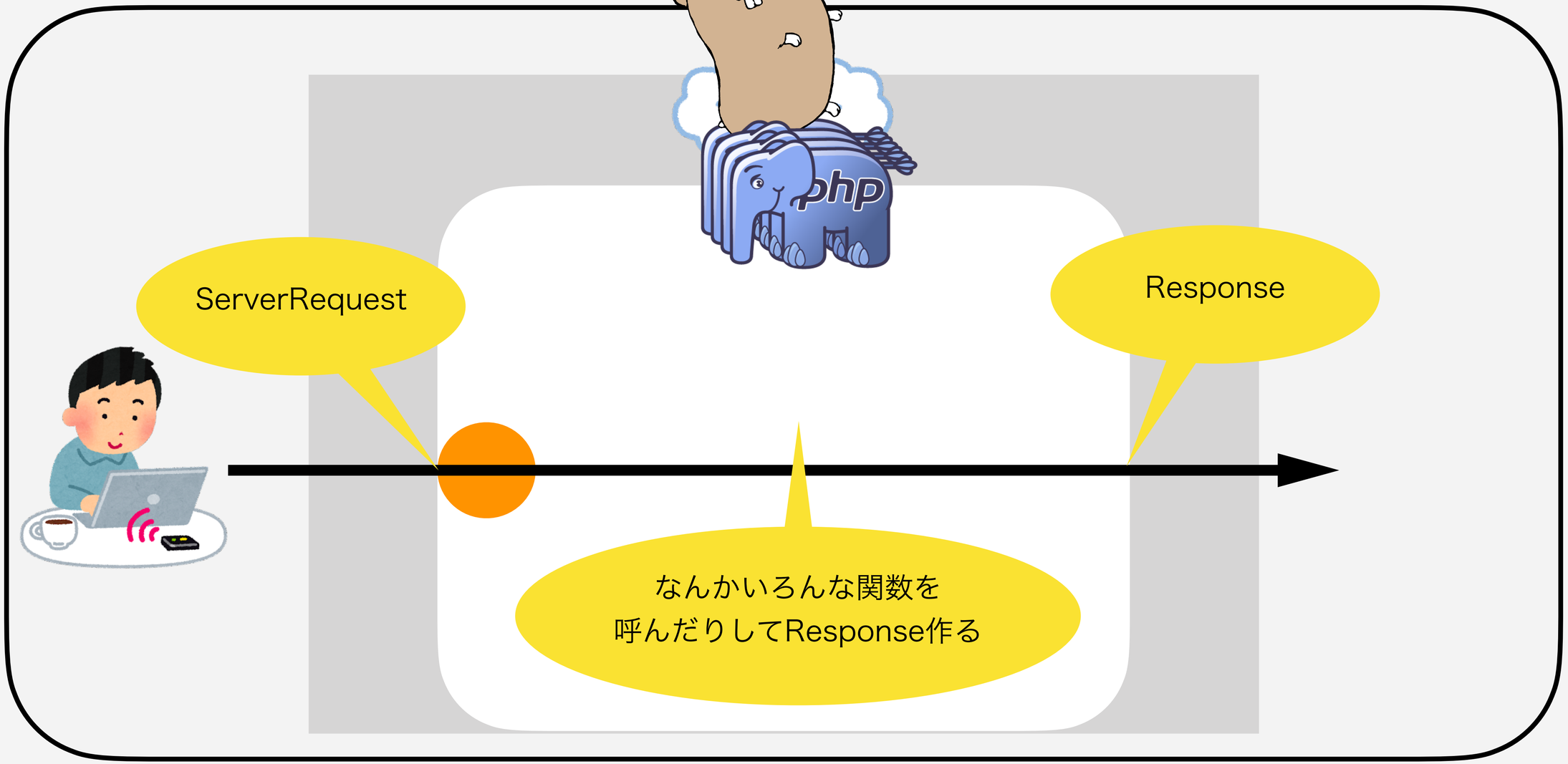
pixiv

RoadRunner



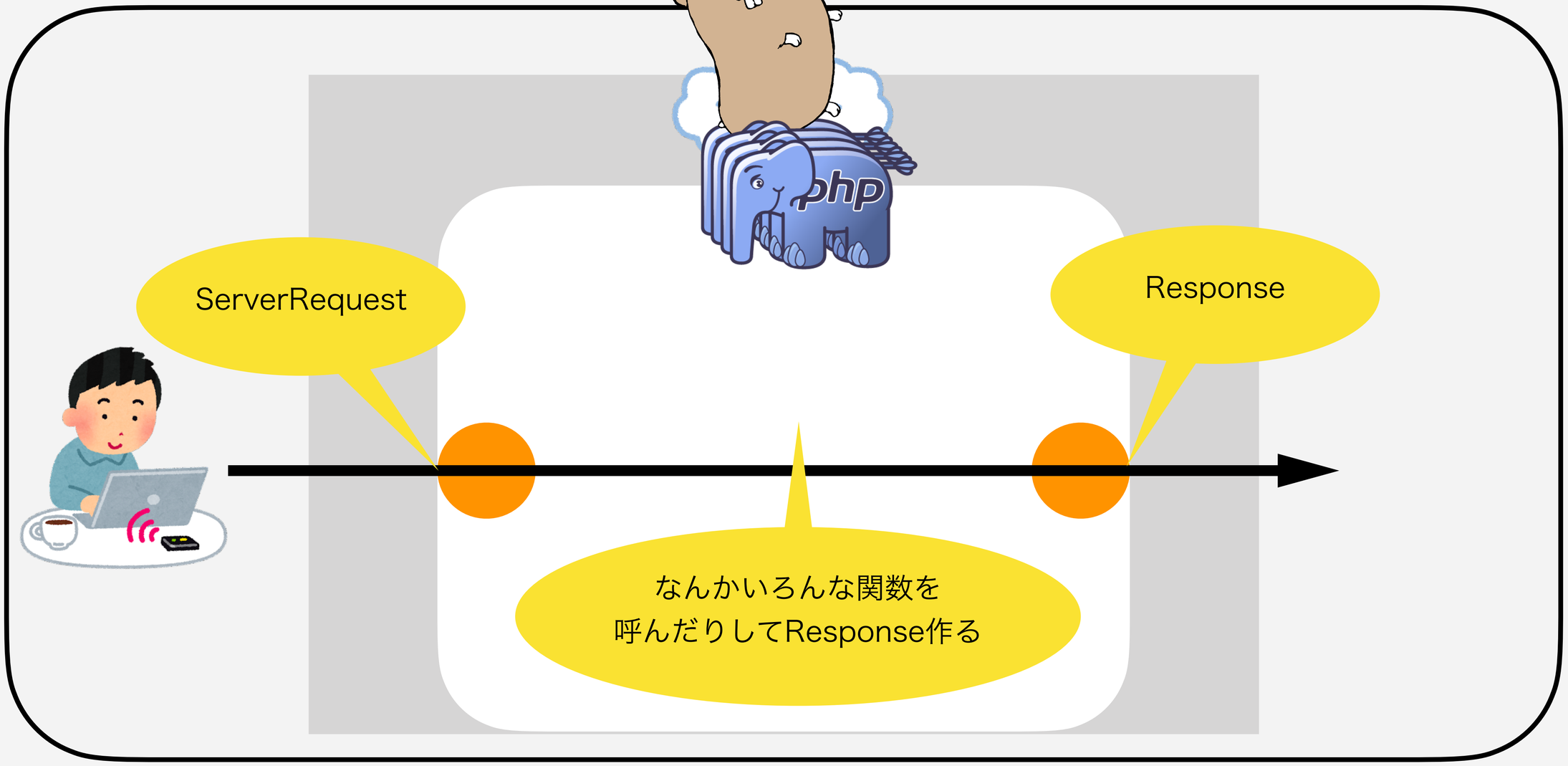
pixiv

RoadRunner



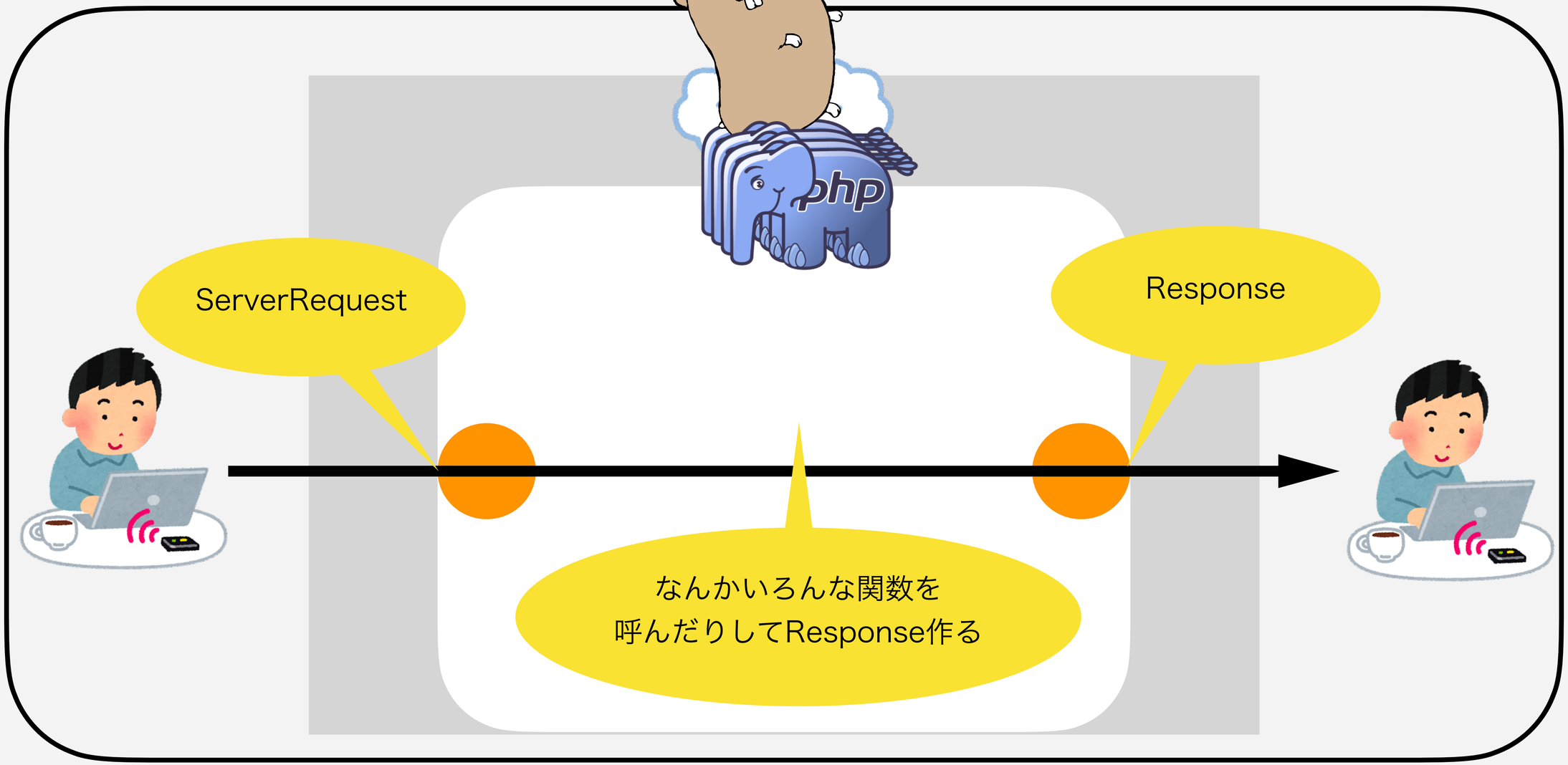
pixiv

RoadRunner



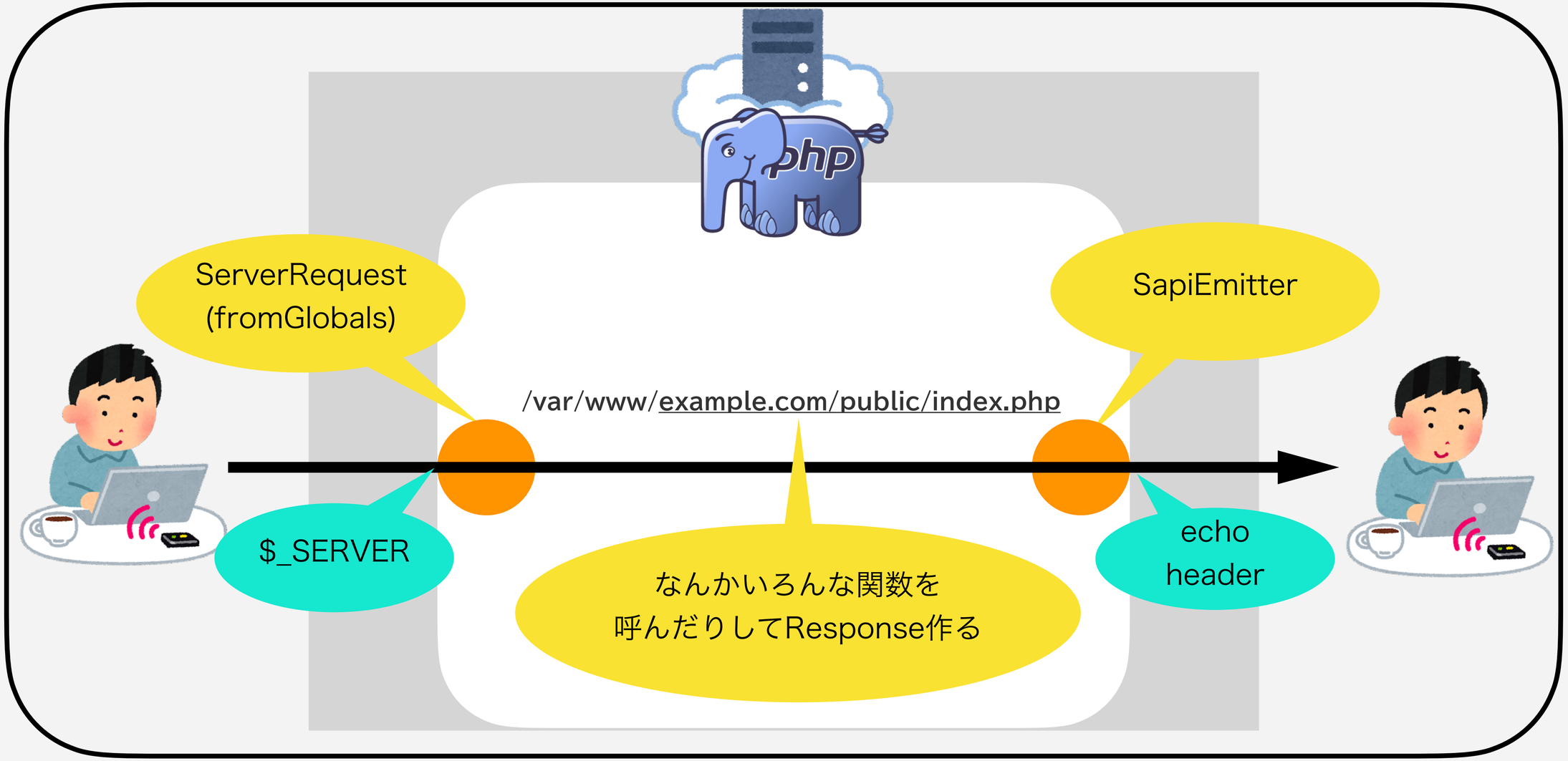
pixiv

RoadRunner



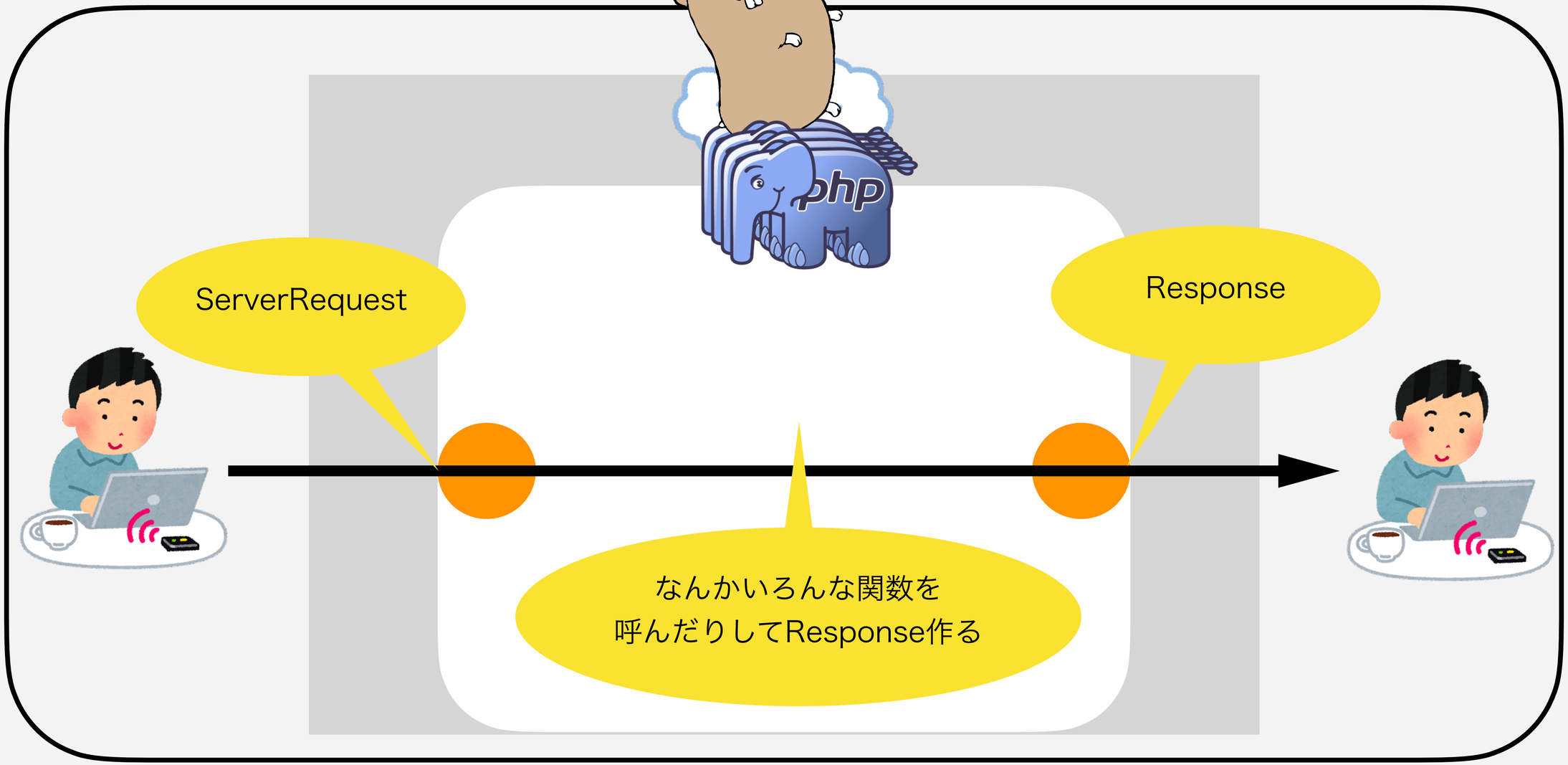
pixiv

従来型環境


ServerRequest
(fromGlobals)
SapiEmitter
/var/www/example.com/public/index.php
$_SERVER
なんかいろんな関数を
呼んだりしてResponse作る
echo
header

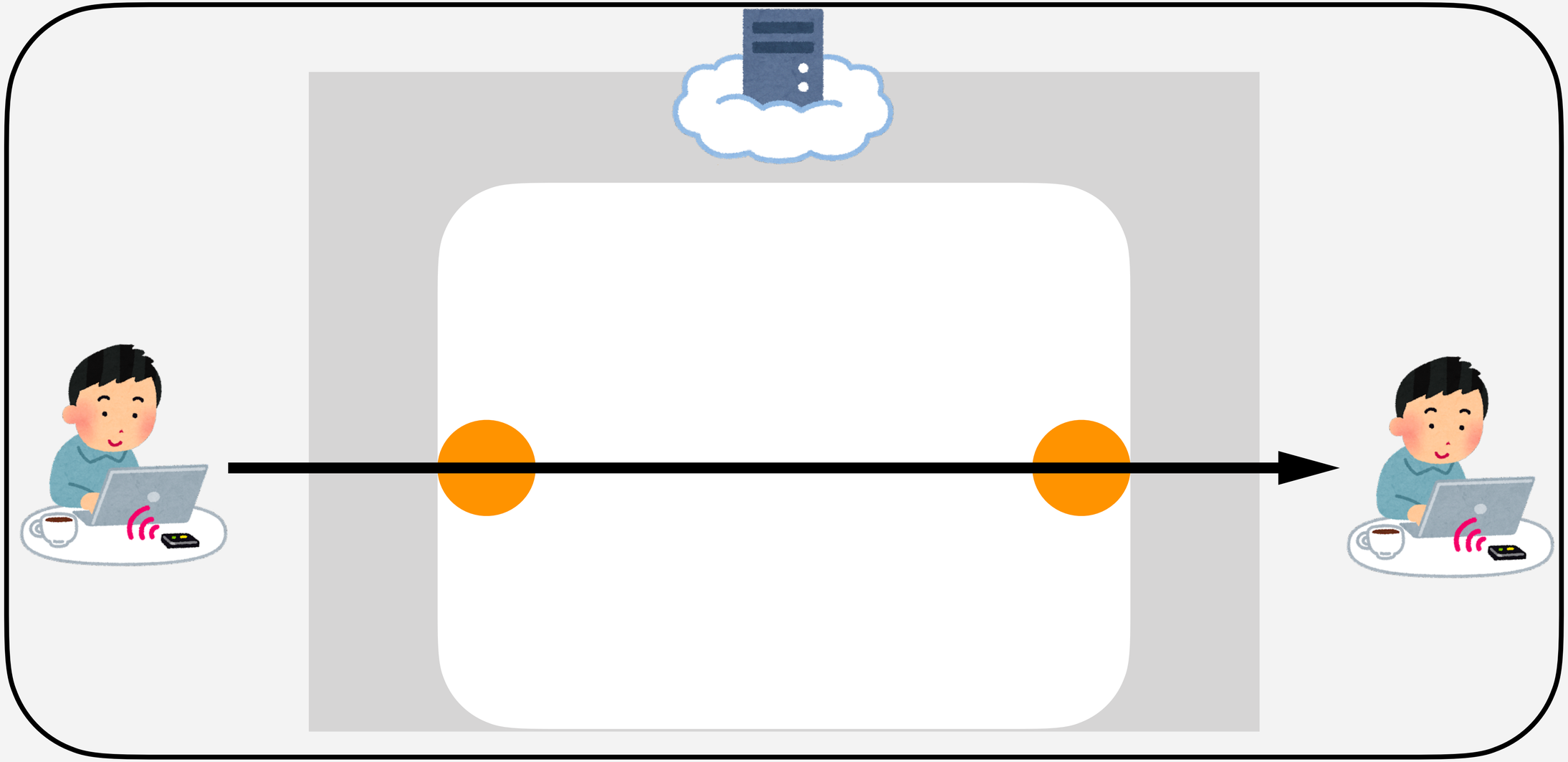
pixiv

RoadRunner



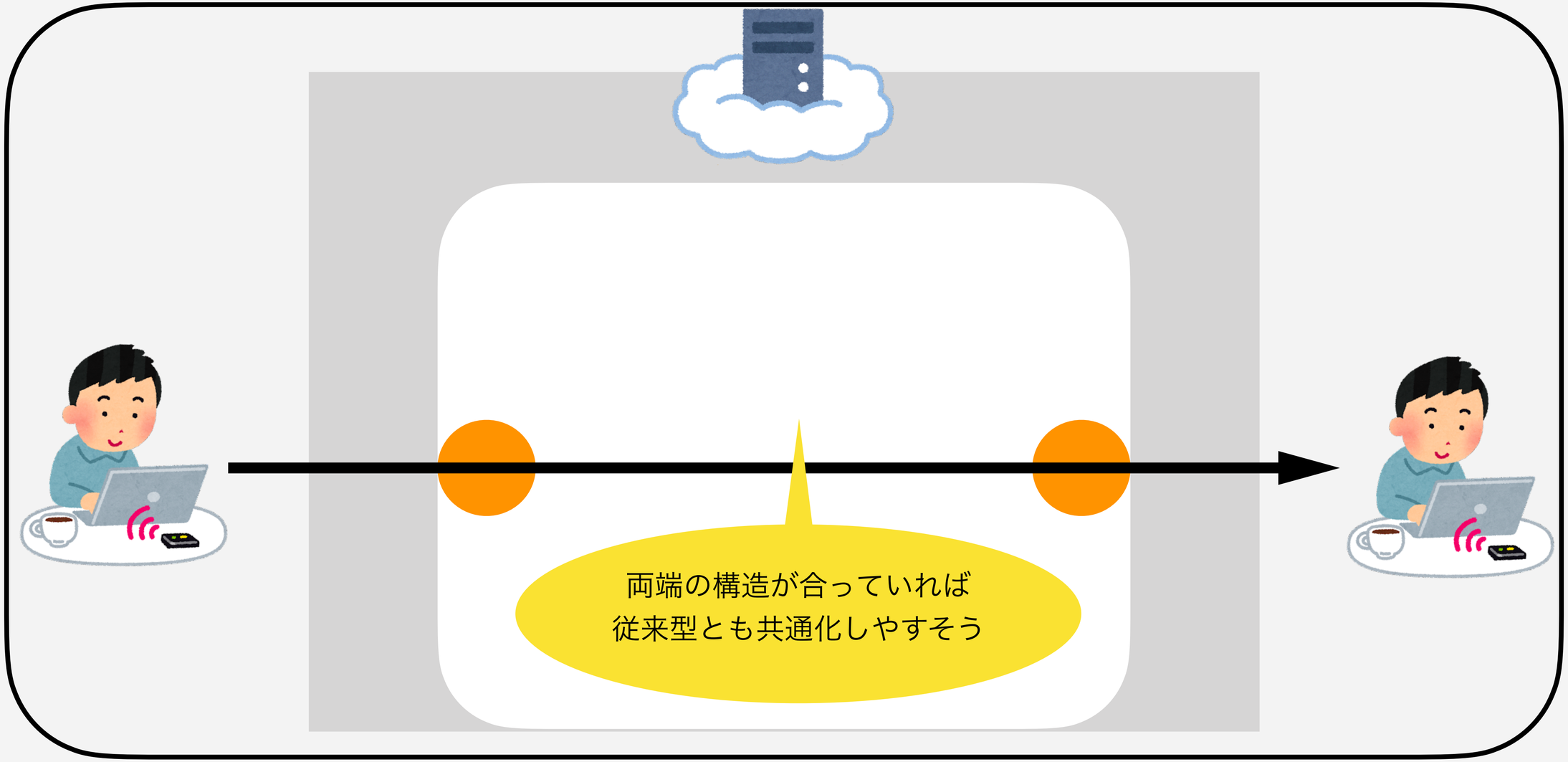
pixiv

入出力を端に揃えると交換可能に



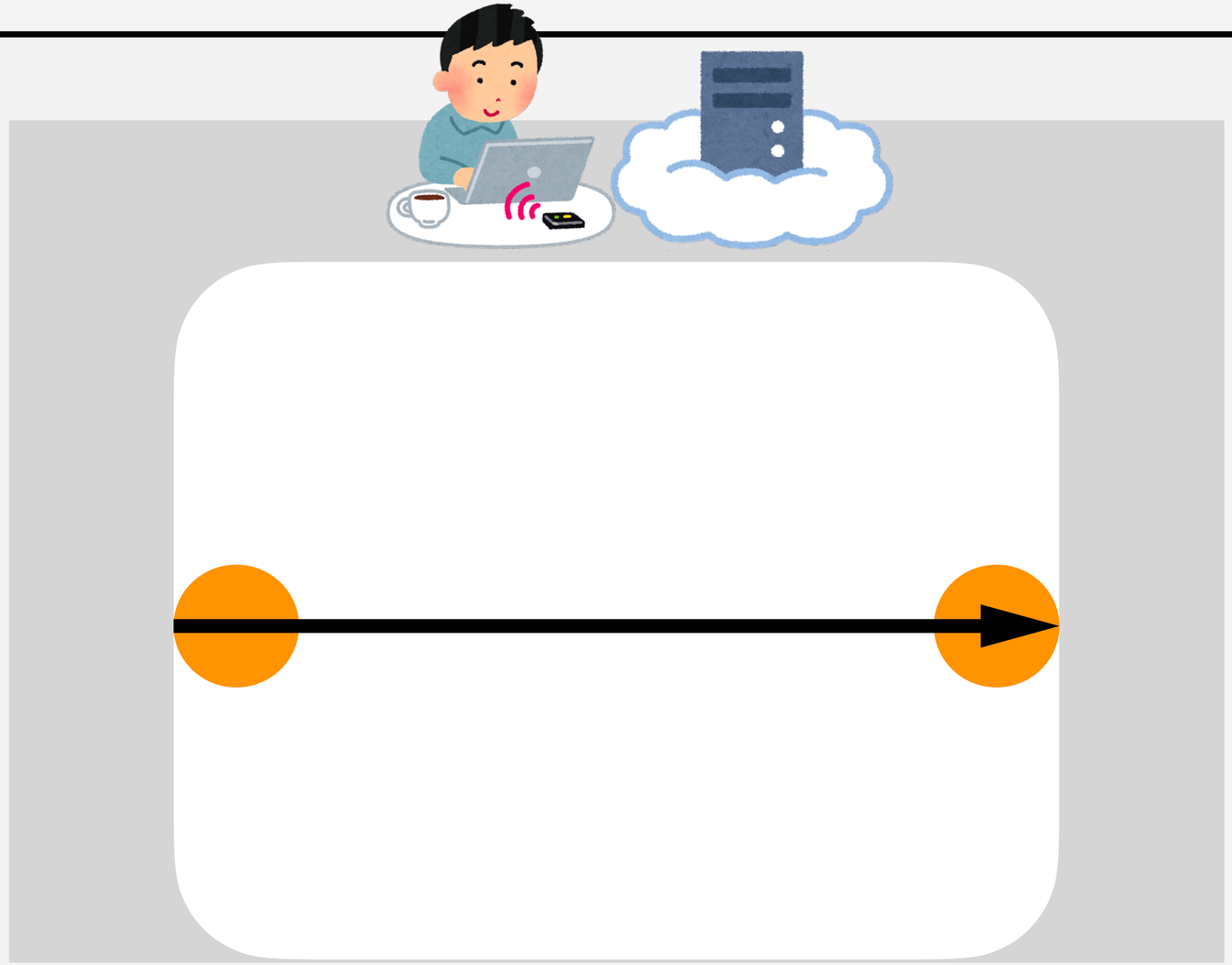
pixiv

入出力を端に揃えると交換可能に

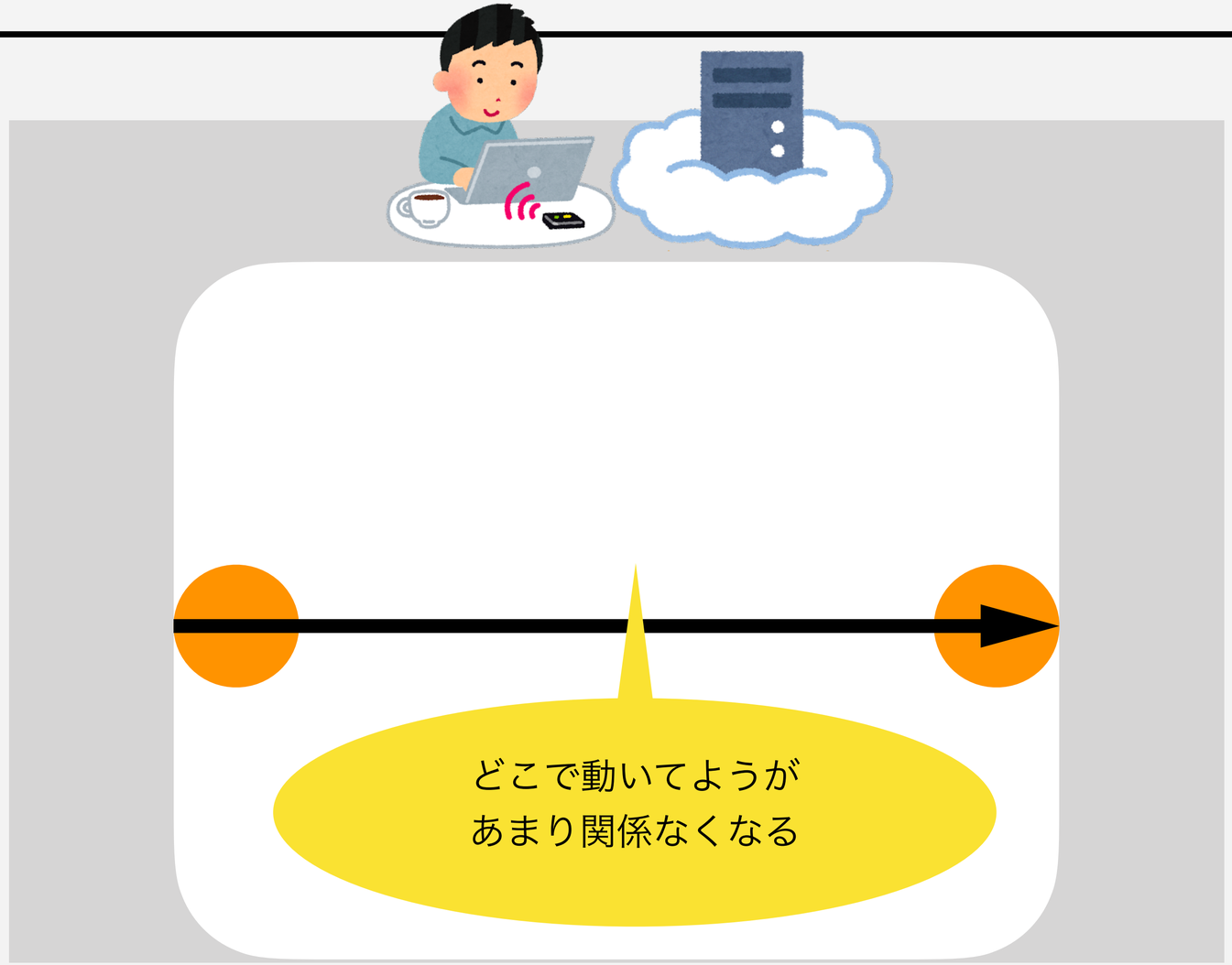


pixiv

外の世界の事情を無視できる

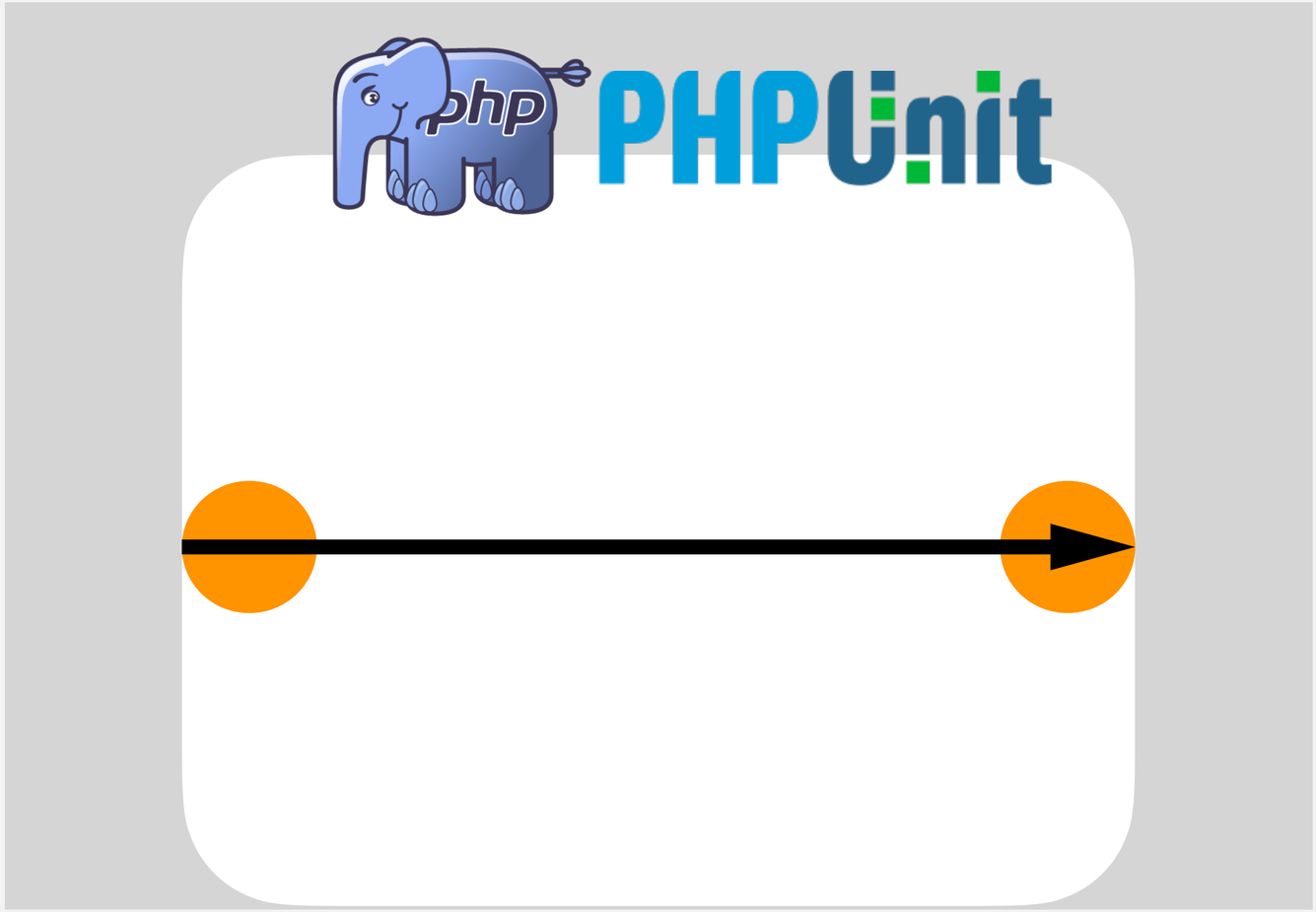


外の世界の事情を無視できる



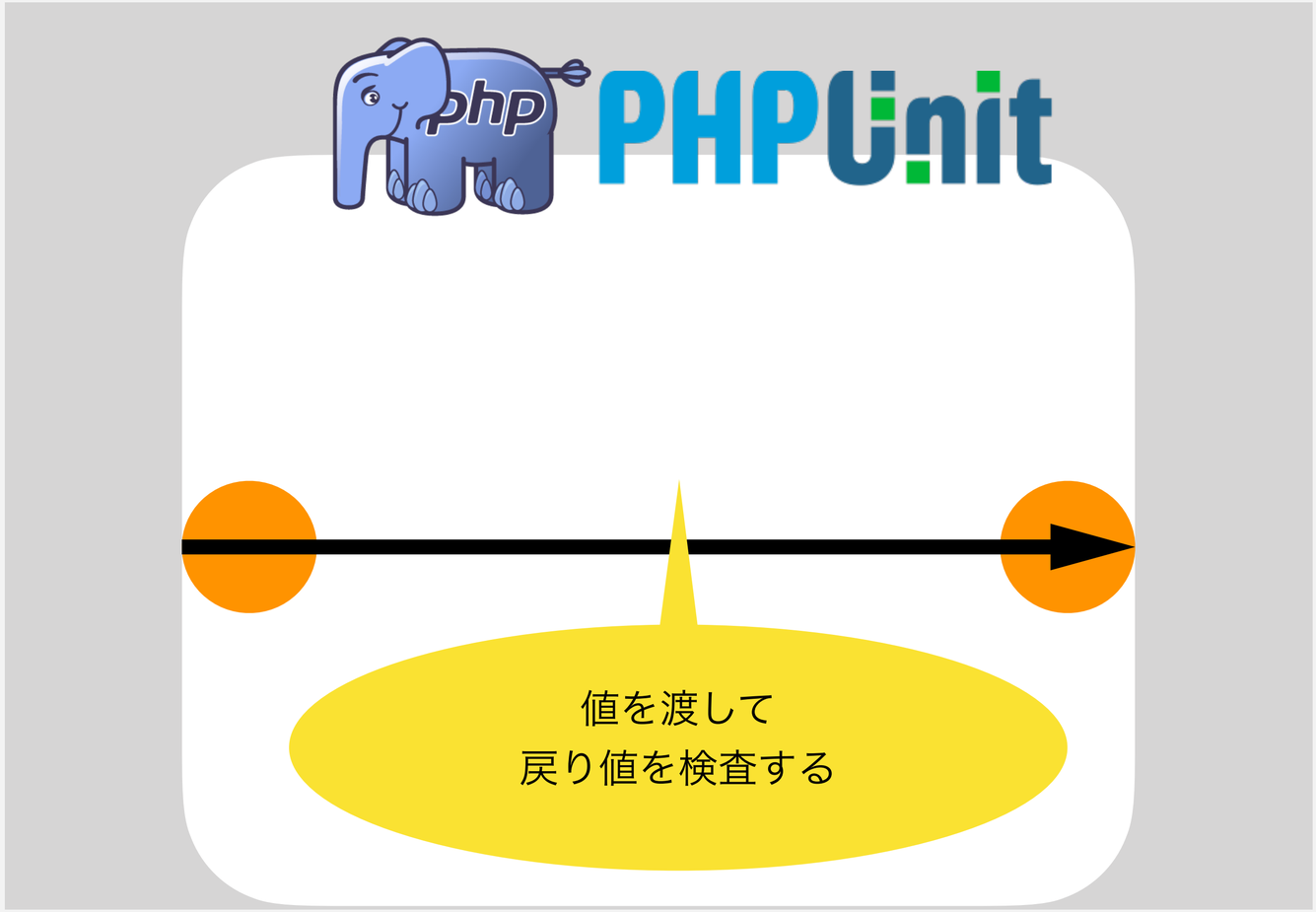
pixiv

ユニットテストで実行もできる



pixiv

ユニットテストで実行もできる



pixiv

もうちょっと
具体的に

PSR-15

pixiv

PSR-15 HTTP Handlers

- Psr\Http\Server\名前空間に2種類のインターフェイスが定義されている
- RequestHandlerInterface
  - レスポンスを生成する（例外送出するかもしれない）
- MiddlewareInterface
  - リクエストハンドラを呼び出すか、呼び出さずにレスポンスを生成してもよい

「ミドルウェア」という語について

- コンピュータ用語の中でも意味が一貫していないことで有名
  - なんかの中間で何かを処理するという非常に漠然としたニュアンス
  - OSでもハードウェアでもない、ソフトウェアとやりとりする何者かがミドルウェアと呼ばれることがある（Webサーバとかファイルシステムとか）
- Webフレームワークでは最終的に呼び出されるやつ（RequestHandler）の間に入って何か処理するものをミドルウェアと呼ぶ

ほかの世界のミドルウェア

- 結構いろんな言語のHTTP仕様に入っている
  - Ruby(Rails含む)ではRack、PythonではWSGIなど
- PHPでもStackPHPという仕様が提案されていたこともあった
- CakePHP 3では独自インターフェイスのミドルウェアがあったが、CakePHP 4でPSR-15互換に移行された

RequestHandlerInterface

```php
<?php

namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface RequestHandlerInterface
{
    public function handle(ServerRequestInterface $request): ResponseInterface;
}
```

MiddlewareInterface

```php
<?php

namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface MiddlewareInterface
{
    public function process(ServerRequestInterface $request,
                            RequestHandlerInterface $handler): ResponseInterface;
}
```

PSR Middlewares

- GitHubのmiddlewares Organizationに実装済みのミドルウェアがある
  - https://github.com/middlewares/psr15-middlewares
  - https://github.com/middlewares/awesome-psr15-middlewares
  - 私は音楽性の合わない部分があるので使ってませんが便利だと思います
    - PSR-17に対応してないからです

簡単なRequestHandlerを作ってみる

- 以後、スライド上に出すコードは簡潔にするために適宜省略しています
  - namespaceやuseは省略しています
  - RequestFactoryInterfaceなどの末尾もasで省略します
- 完全なソースコードはGitHub上で確認してください
  - https://github.com/zonuexe/phperkaigi-psr15

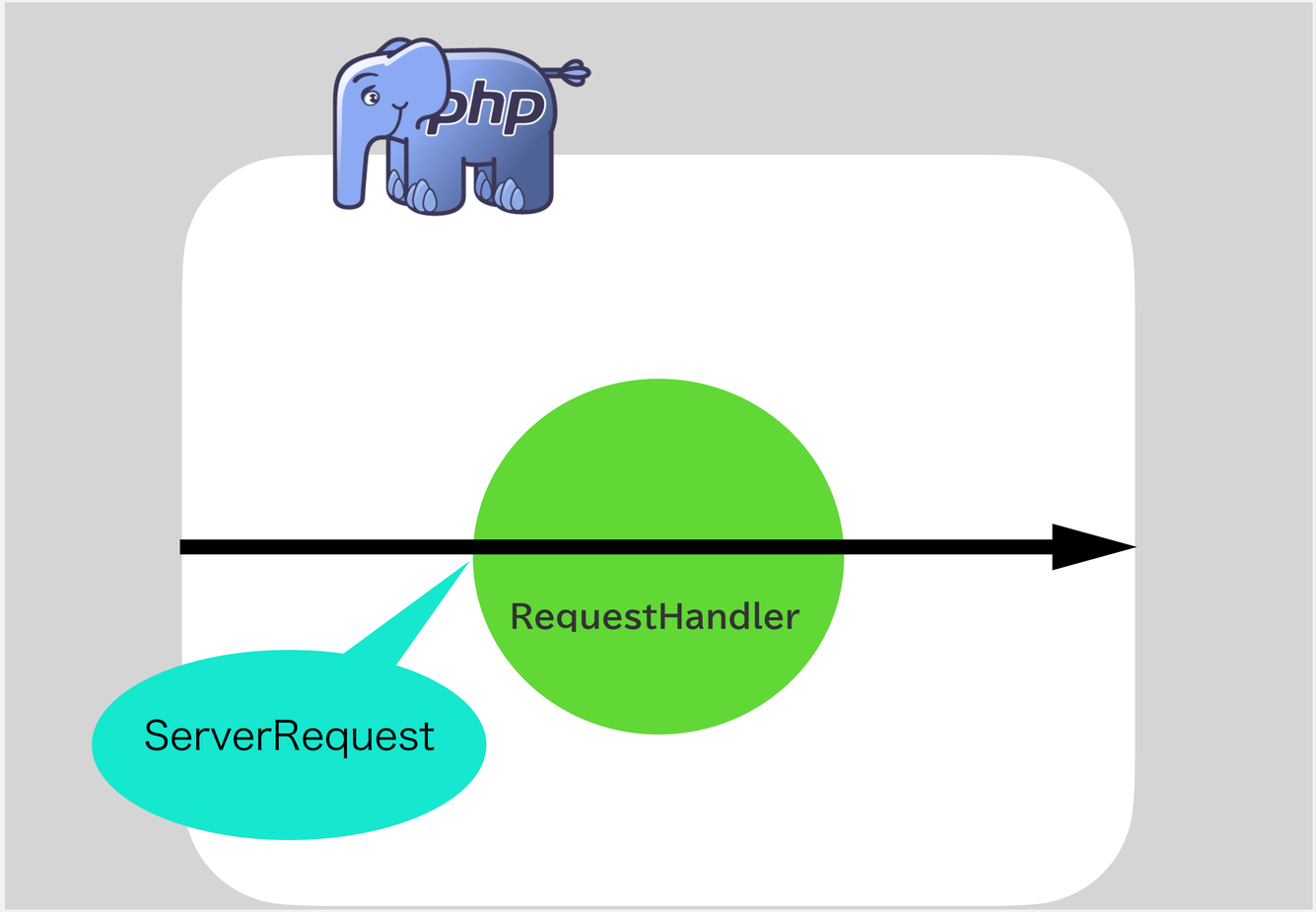
HelloWorldっぽいレスポンスを返す

```
class HelloJsonHandler implements RequestHandlerInterface {
  public function __construct(
    private StreamFactory $stream_factory,
    private ResponseFactory $response_factory
  ){}

  public function handle(ServerRequest $request): Response {
    $body = $this->stream_factory->createStream(json_encode(['Hello' => 'World']);
    return $this->response_factory->createResponse(200)
      ->withHeader('Content-Type', ['application/json'])
      ->withBody($body);
  }
}
```

GETのときにHelloWorldを返す

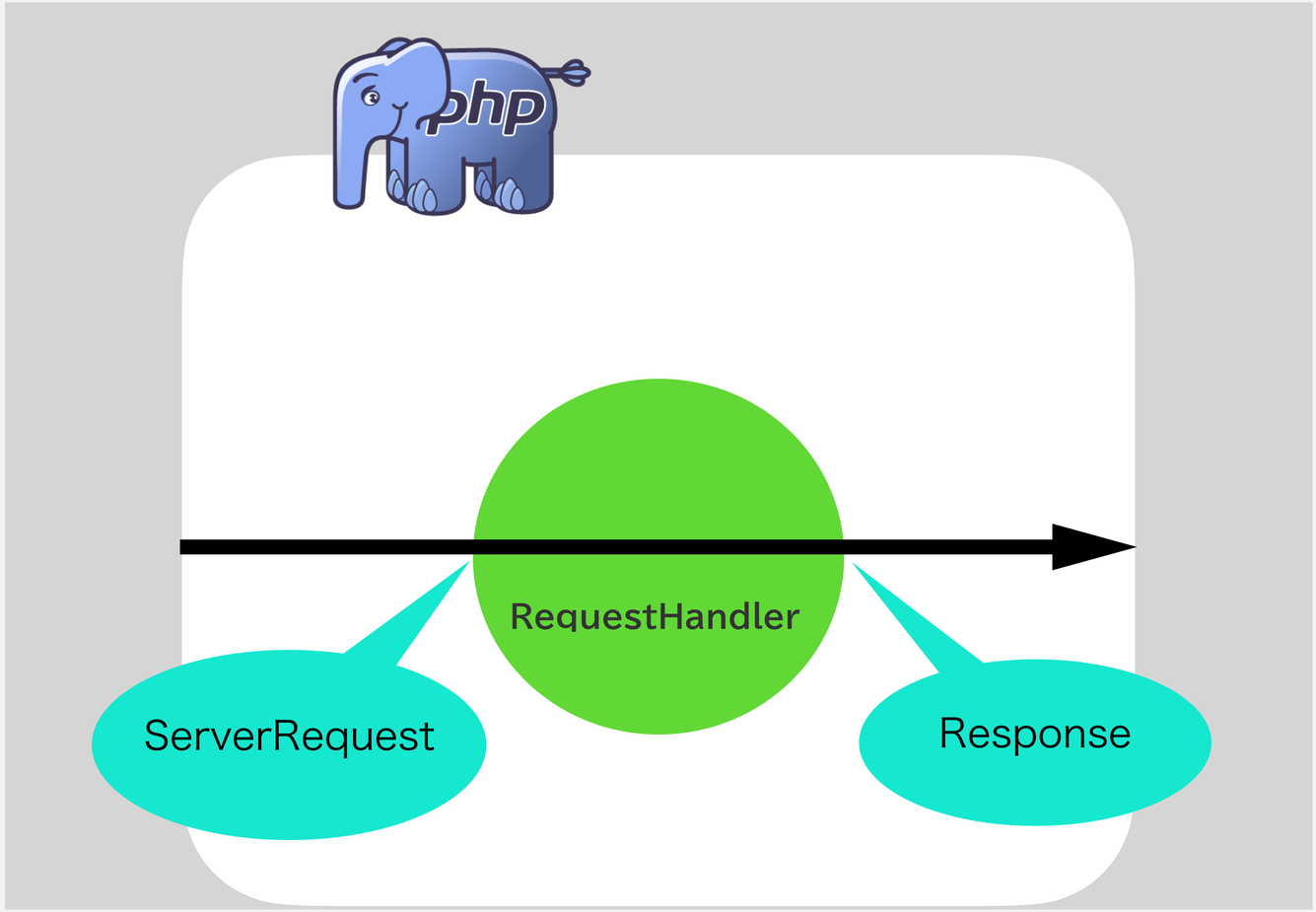
```
// 前略
public function handle(ServerRequestInterface $request): ResponseInterface {
    if ($request->getMethod() !== 'GET') {
        return $this->response_factory->createResponse(404);
    }

    // 後はさっきと同じ
    $body = $this->stream_factory->createStream(json_encode(['Hello' => 'World']);
    return $this->response_factory->createResponse(200)
      ->withHeader('Content-Type', ['application/json'])
      ->withBody($body);
 }
```

RequestHandler呼び出しイメージ



pixiv

RequestHandler呼び出しイメージ



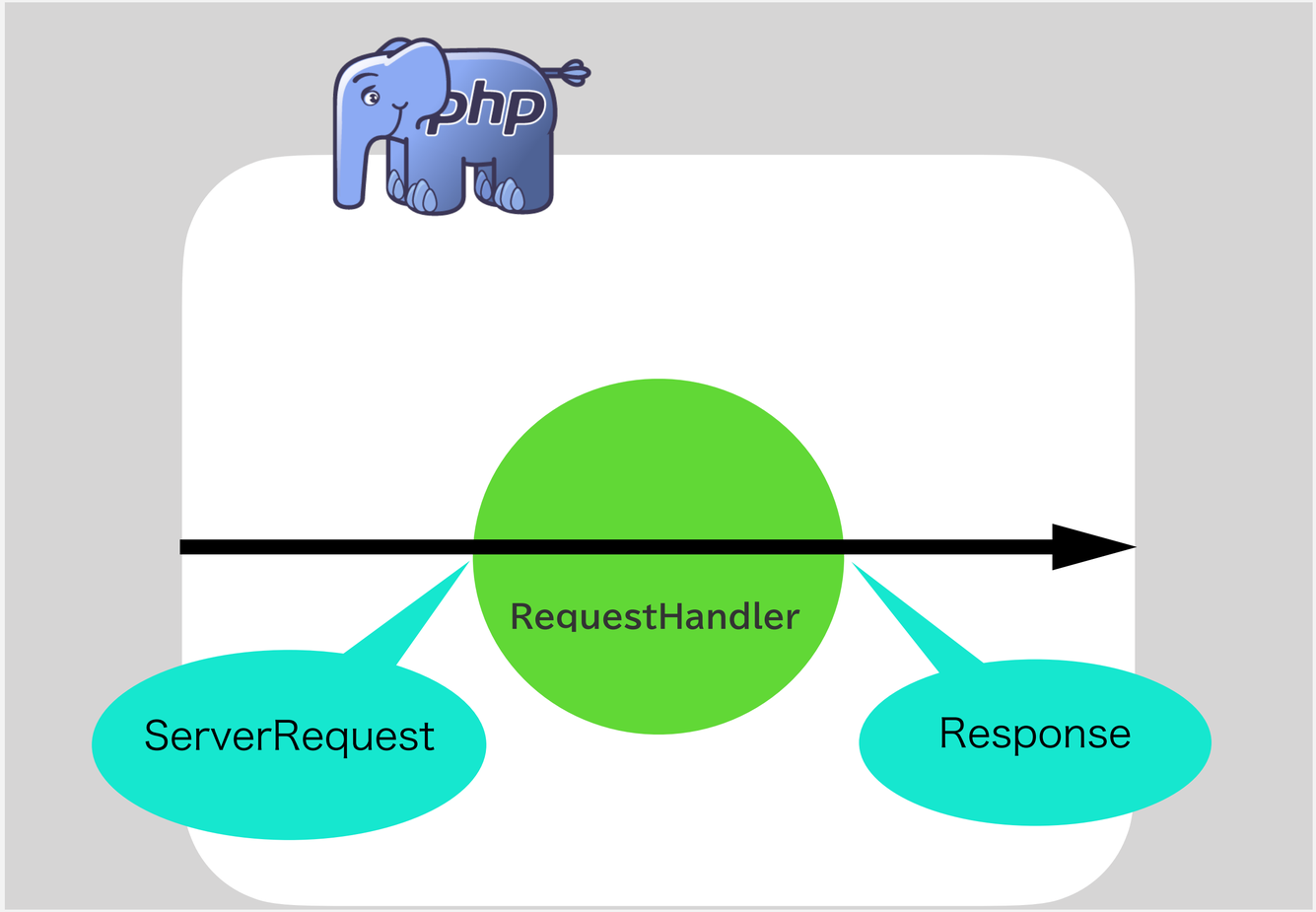
pixiv

テストしてみよう

テストケース自身が
Factoryの機能を持っている
と書き心地が良くなる

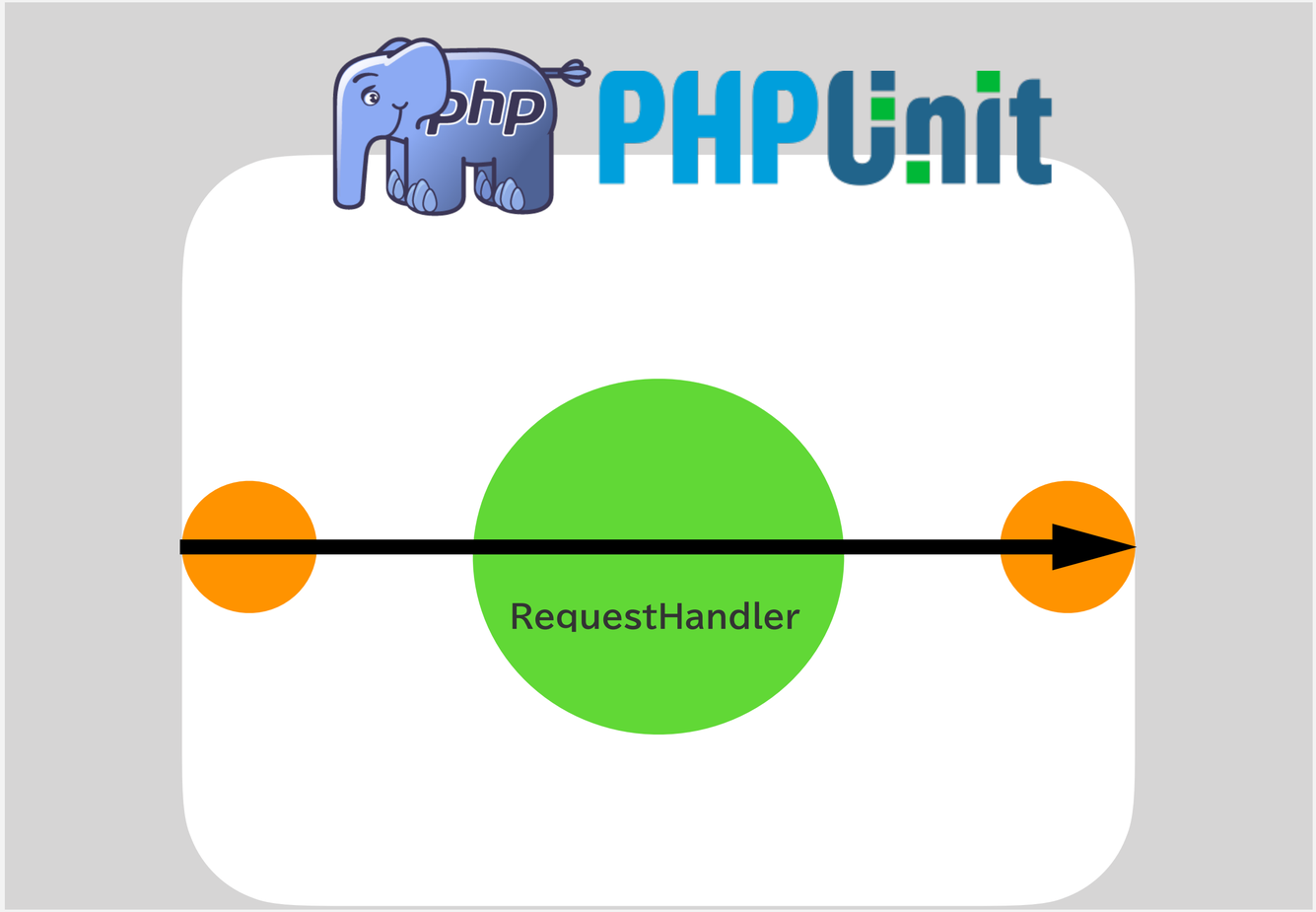
- シンプルにPHPUnitを使える
- ResponseHanlderのインスタンスを用意する
- ServerRequestのインスタンスを用意する
- $response = $handler->handler($request) のように呼び出す
- $responseからステータスコード、HTTPヘッダ、ボディなどを取り出して assertEquals()などで比較してみる

RequestHandler呼び出しイメージ



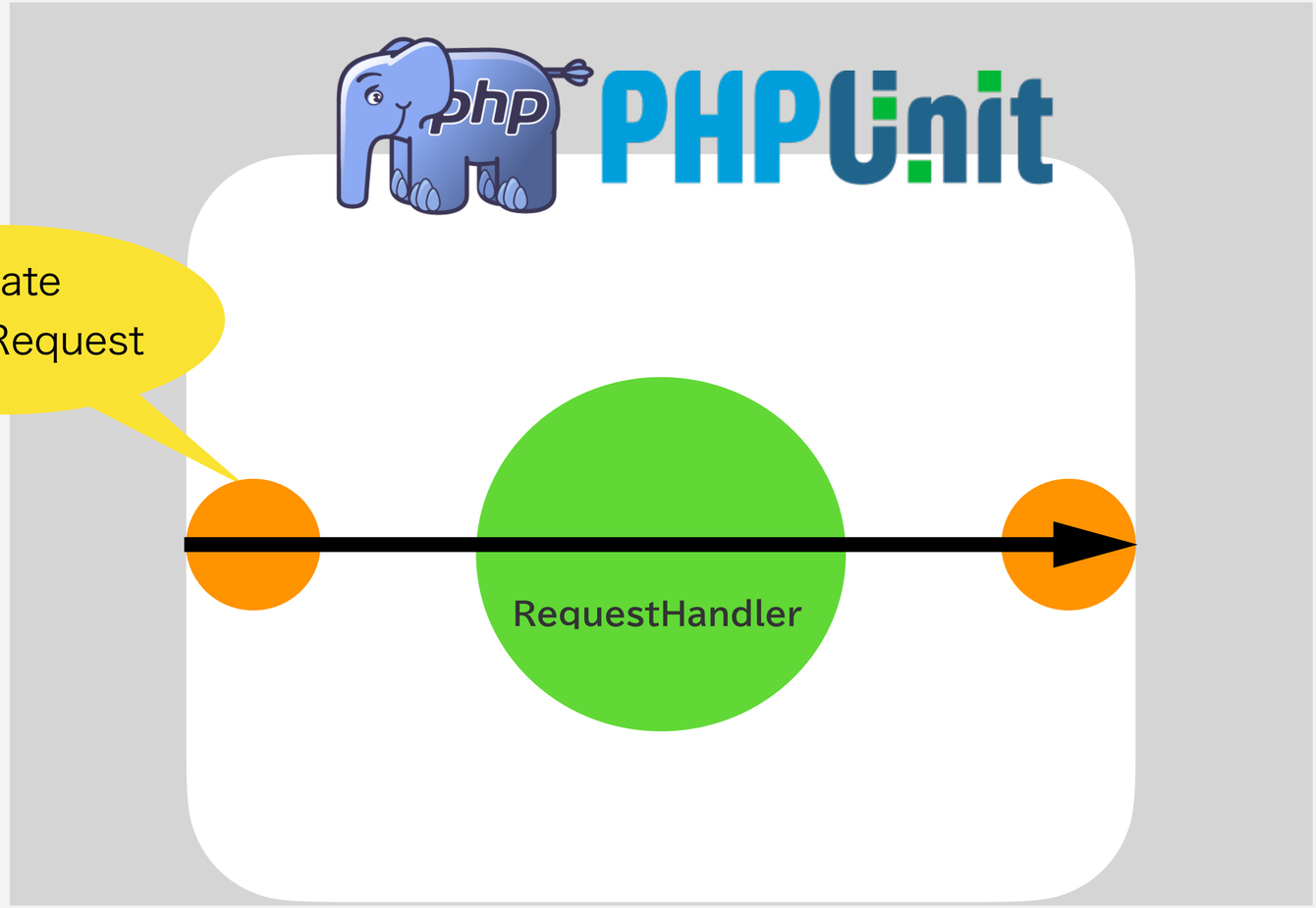
pixiv

RequestHandler呼び出しイメージ



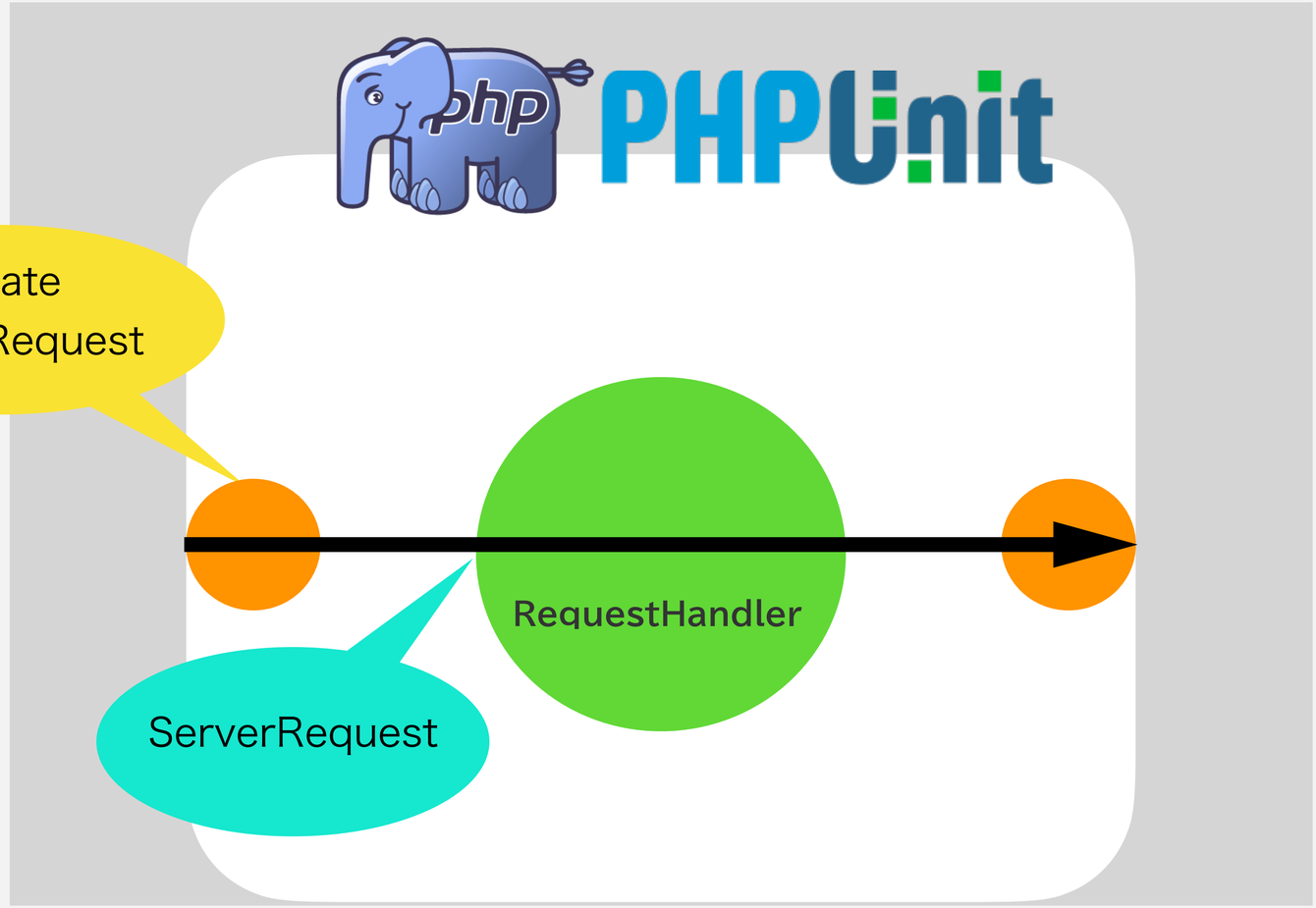
pixiv

RequestHandler呼び出しイメージ



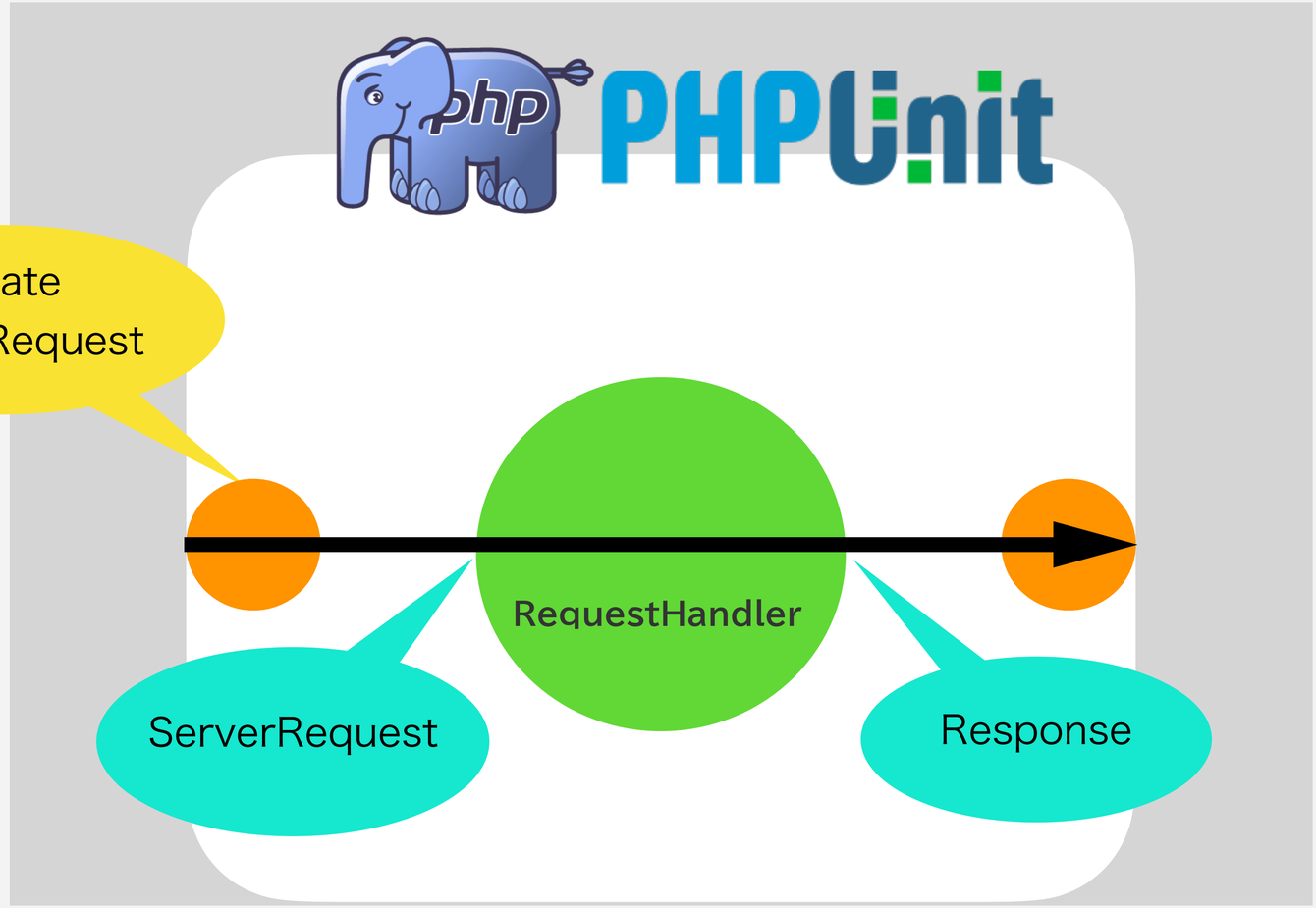
pixiv

RequestHandler呼び出しイメージ



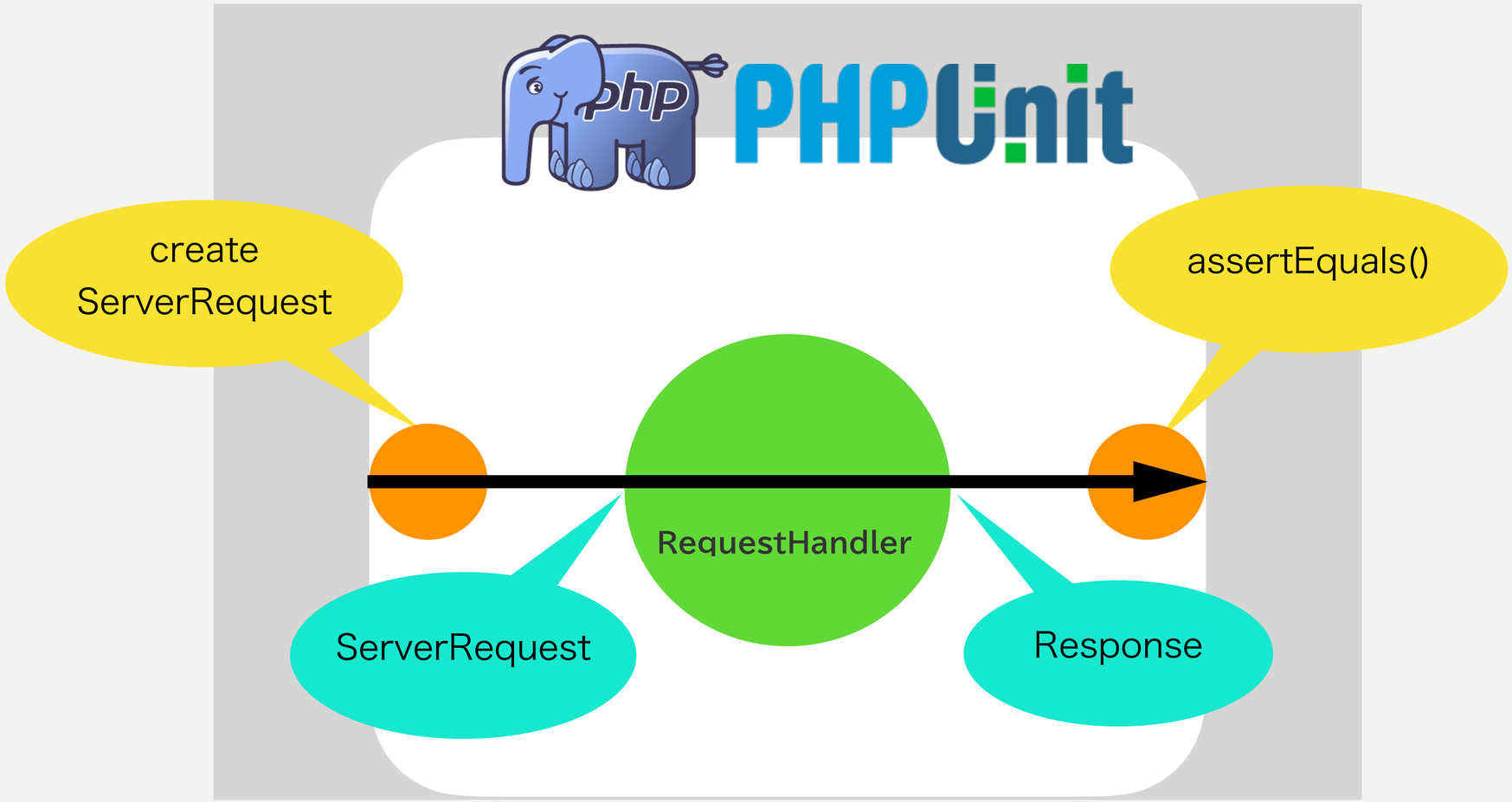
pixiv

RequestHandler呼び出しイメージ



pixiv

RequestHandler呼び出しイメージ



pixiv

テストの準備 (HttpFactoryTrait)

```php
use Nyholm\Psr7\Factory\Psr17Factory;

trait HttpFactoryTrait
{
  private function psr17factory(): Psr17Factory
  {
     return new Psr17Factory();
  }
```

テストの準備 (HttpFactoryTrait)

```
public function getRequestFactory(): RequestFactoryInterface
{
    return $this->psr17factory();
}

public function getResponseFactory(): ResponseFactoryInterface
{
    return $this->psr17factory();
}

public function getServerRequestFactory(): ServerRequestFactoryInterface
{
```

テストの準備 (HttpFactoryTrait)

```
/**
 * Create a new stream from a string.
 *
 * The stream SHOULD be created with a temporary resource.
 *
 * @param string $content String content with which to populate the stream.
 */
public function createStream(string $contents): StreamInterface
{
    return $this->getStreamFactory()->createStream($contents);
}
```

HttpFactoryTraitの特徴

- 特定のPSR-7ライブラリにあえて強く依存している
  - Nyholm Psr17Factoryは状態を持たないのでオブジェクトをキャッシュしてもいいが、そもそも状態をまったく持たない軽量なオブジェクトなので必要な都度新しいインスタンスを作ってもボトルネックにならない
  - 正攻法であればsetUpやsetUpBeforeClassで注入する形をとるが、dataProviderから依存できるようにあえて割り切っている

テストしてみよう (setup)

```
class HelloJsonHandlerTest extends \PHPUnit\Framework\TestCase
{
  use Helper\HttpFactoryTrait;

  private HelloJsonHandler $subject;

  public function setUp(): void {
    parent::setUp();

    $this->subject = new HelloJsonHandler(
       $this->getStreamFactory(), $this->getResponseFactory());
  }
```

テスト

```
    /**
     * @dataProvider requestProvider
     */
    public function test(ServerRequest $request, array $expected): void
    {
        $actual = $this->subject->handle($request);

        $this->assertSame($expected['status_code'], $actual->getStatusCode());
        $this->assertEquals($expected['headers'], $actual->getHeaders());
        $this->assertSame($expected['body'], (string)$actual->getBody());
    }
```

テスト (dataProvider)

```
public function requestProvider(): iterable
{
  yield 'GET' => [
    $this->createServerRequest('GET', '/dummy'),
    [
      'status_code' => 200,
      'headers' => [
        'Content-Type' => ['application/json'],
      ],
      'body' => '{"Hello":"World"}',
    ],
];
```

テスト (再掲)

```
    /**
     * @dataProvider requestProvider
     */
    public function test(ServerRequest $request, array $expected): void
    {
        $actual = $this->subject->handle($request);

        $this->assertSame($expected['status_code'], $actual->getStatusCode());
        $this->assertEquals($expected['headers'], $actual->getHeaders());
        $this->assertSame($expected['body'], (string)$actual->getBody());
    }
```

テスト (dataProviderを追加)

```
$default_expected = ['status_code' => 404, 'headers' => [], 'body' => ''];

$http_methods = ['POST', 'PUT', 'DELETE'];

foreach ($http_methods as $method) {
  yield $method => [
    $this->createServerRequest($method, '/dummy'),
    $default_expected,
  ];
}
```

テスト (dataProviderを追加)

```
$default_expected = ['status_code' => 404, 'headers' => [], 'body' => ''];

$http_methods = ['POST', 'PUT', 'DELETE'];

foreach ($http_methods as $method) {
  yield $method => [
    $this->createServerRequest($method, '/dummy'),
    $default_expected,
  ];
}
```

GET以外のメソッドは
すべて結果は同じになる

PHPUnit実行結果

```
megurine % ./vendor/bin/phpunit
PHPUnit 9.5.20 #StandWithUkraine

Warning:       No code coverage driver available

....                                                                4 / 4 (100%)

Time: 00:00.004, Memory: 10.00 MB

OK (4 tests, 12 assertions)
megurine %
```

RequestHandlerテストの要点

- テストケースの内容はごく薄くする
- 「この値を渡したときに必ずこの値を返す」という構造に落とし込む
  - そうなっていないなら依存の分離がうまくいっていない
- 戻り値となるResponseオブジェクトからは、特に getStatusCode(), getHeaders(), getBody()に注目する
  - bodyが巨大な場合は必ずしもassertSame()で比較する必要はない

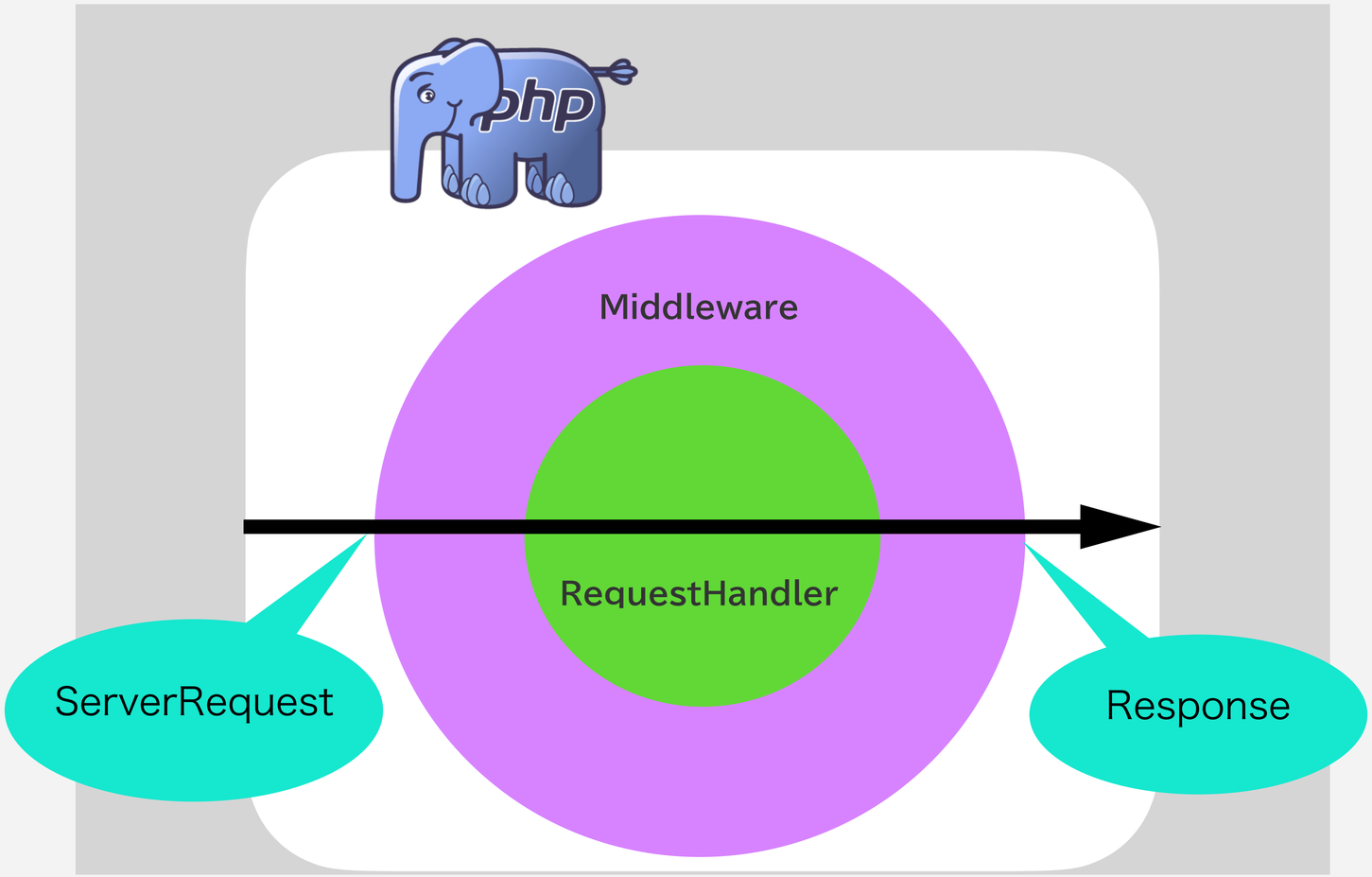
MiddlewareInterface

```php
<?php

namespace Psr\Http\Server;

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

interface MiddlewareInterface
{
    public function process(ServerRequestInterface $request,
                            RequestHandlerInterface $handler): ResponseInterface;
}
```

簡単なMiddlewareを作ってみる

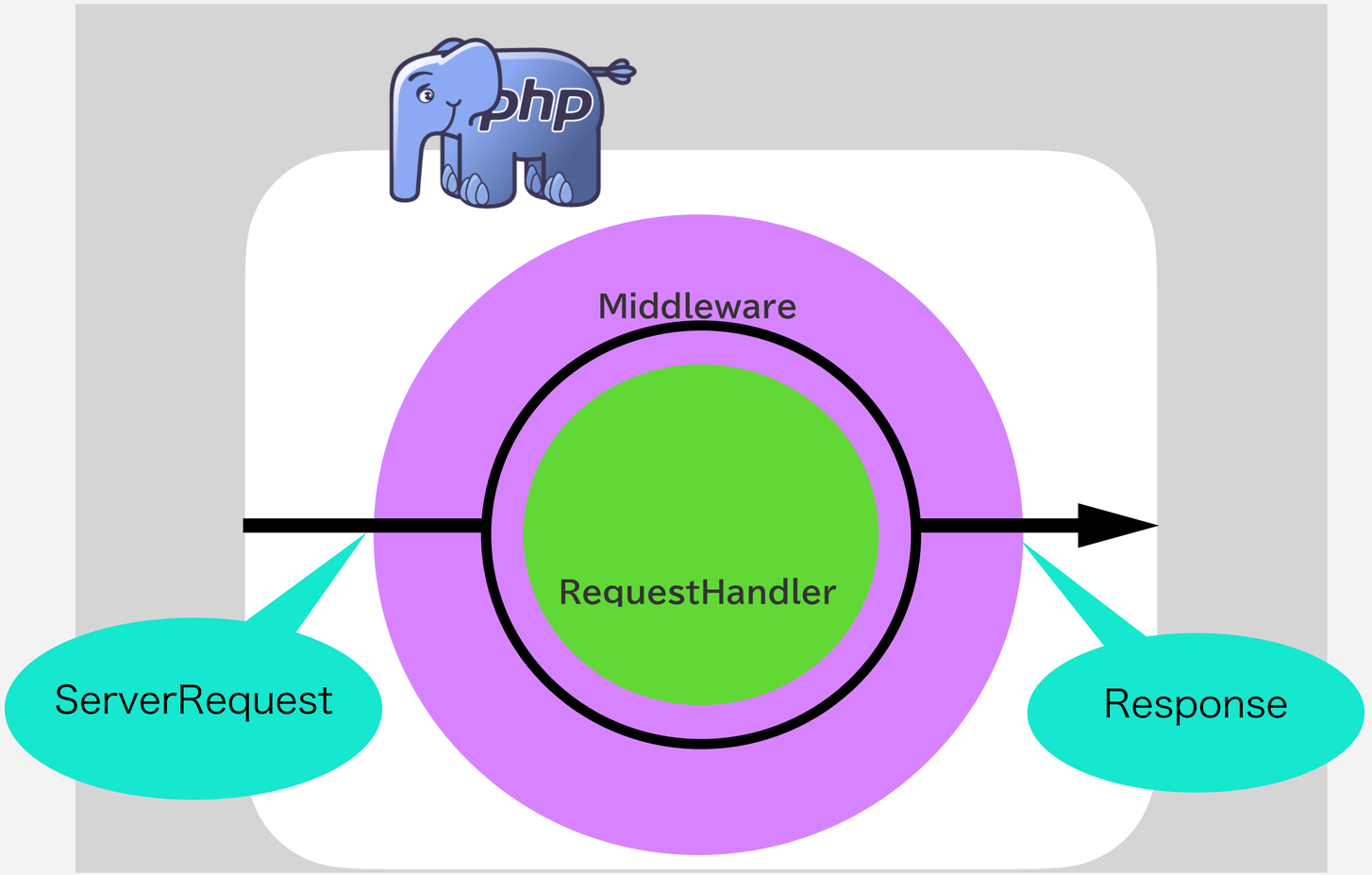
- Middlewareは結構なんでもできる
  - RequestHandlerに渡す前にServerRequestを加工する
  - RequestHandlerから戻ってきたResponseを加工する
  - RequestHandlerが発生したエラーをキャッチしてハンドリングする

Middleware呼び出しイメージ



pixiv

Middleware呼び出しイメージ



pixiv

何もしないミドルウェア(ストロー)

```
class StrawMiddleware implements MiddlewareInterface
{
  public function process(
    ServerRequest $request, RequestHandler $handler): Response
  {
    return $handler->handle($request);

  }
}
```

全レスポンスに規定のヘッダを付与

```
class StrawMiddleware implements MiddlewareInterface
{
  public function process(
    ServerRequest $request, RequestHandler $handler): Response
  {
    return $handler->handle($request)
      ->withHeader('X-Content-Type-Options', ['nosniff']);
  }
}
```

HTTPSにリダイレクト

```
class HttpsRedirectMiddleware implements MiddlewareInterface
{
  public function process(
    ServerRequest $request, RequestHandler $handler): Response
  {
    $uri = $request->getUri();
    if (strtolower($uri->getScheme()) !== 'https') {
      return $this->response_factory->createResponse(302)
        ->withHeader('Location', [(string)$uri->withScheme('https')]);
    }
    return $handler->handle($request);
  }
}
```

例外に対して標準エラーページ

```
class ErrorPageMiddleware implements MiddlewareInterface
{
  public function process(
    ServerRequest $request, RequestHandler $handler): Response
  {
    try {
      return $handler->handle;
    } catch (HttpException $e) {
      return $this->renderHtmlErrorPage($e);
    }
  }
}
```

RequestHandlerのテスト(再掲)

```
    /**
     * @dataProvider requestProvider
     */
    public function test(ServerRequest $request, array $expected): void
    {
        $actual = $this->subject->handle($request);

        $this->assertSame($expected['status_code'], $actual->getStatusCode());
        $this->assertEquals($expected['headers'], $actual->getHeaders());
        $this->assertSame($expected['body'], (string)$actual->getBody());
    }
```

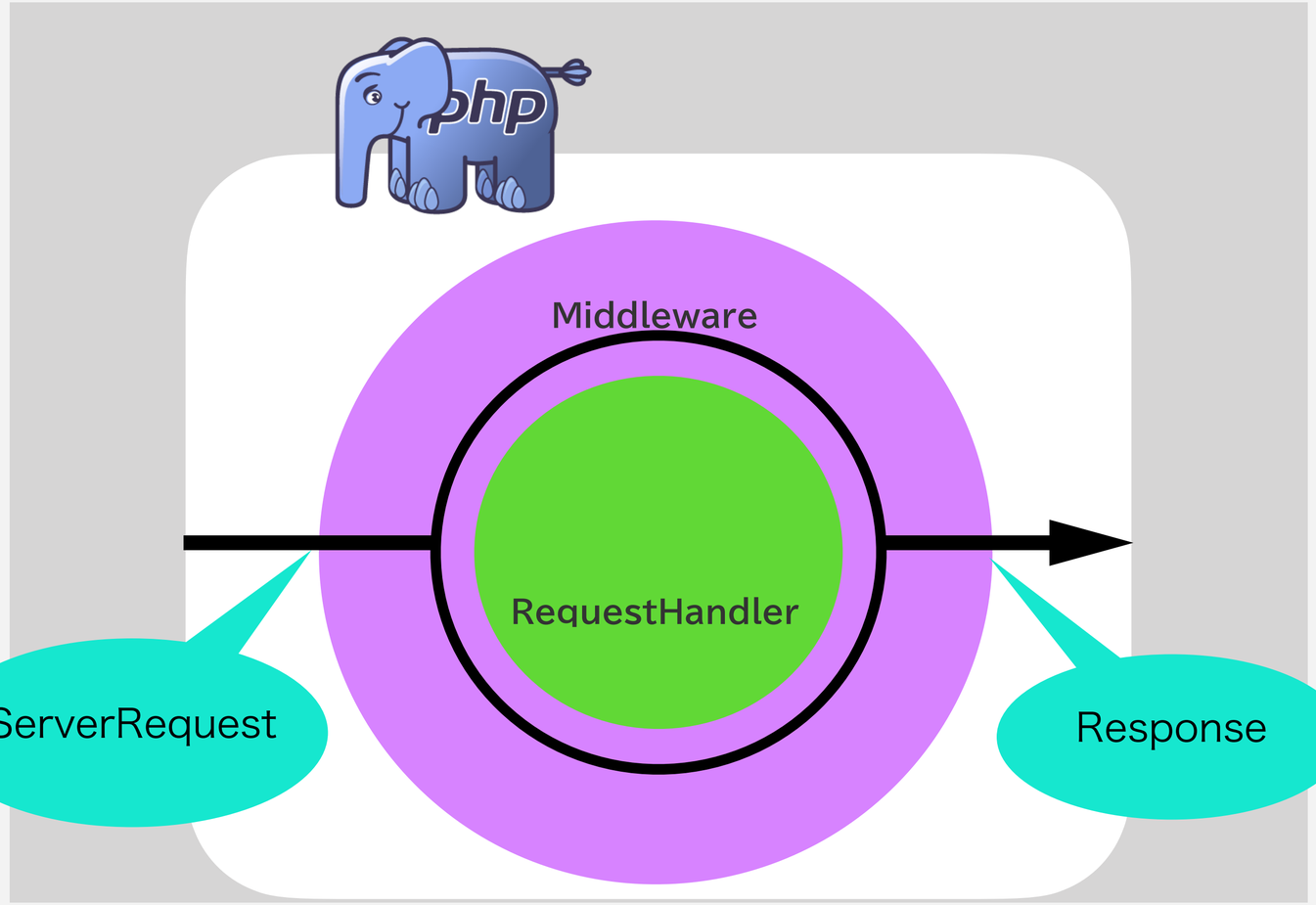
ResponseHandlerのテスト (再掲)

- シンプルにPHPUnitを使える
- ResponseHanlderのインスタンスを用意する
- ServerRequestのインスタンスを用意する
- $response = $handler->handler($request) のように呼び出す
- $responseからステータスコード、HTTPヘッダ、ボディなどを取り出して assertEquals()などで比較してみる

Middlewareのテスト

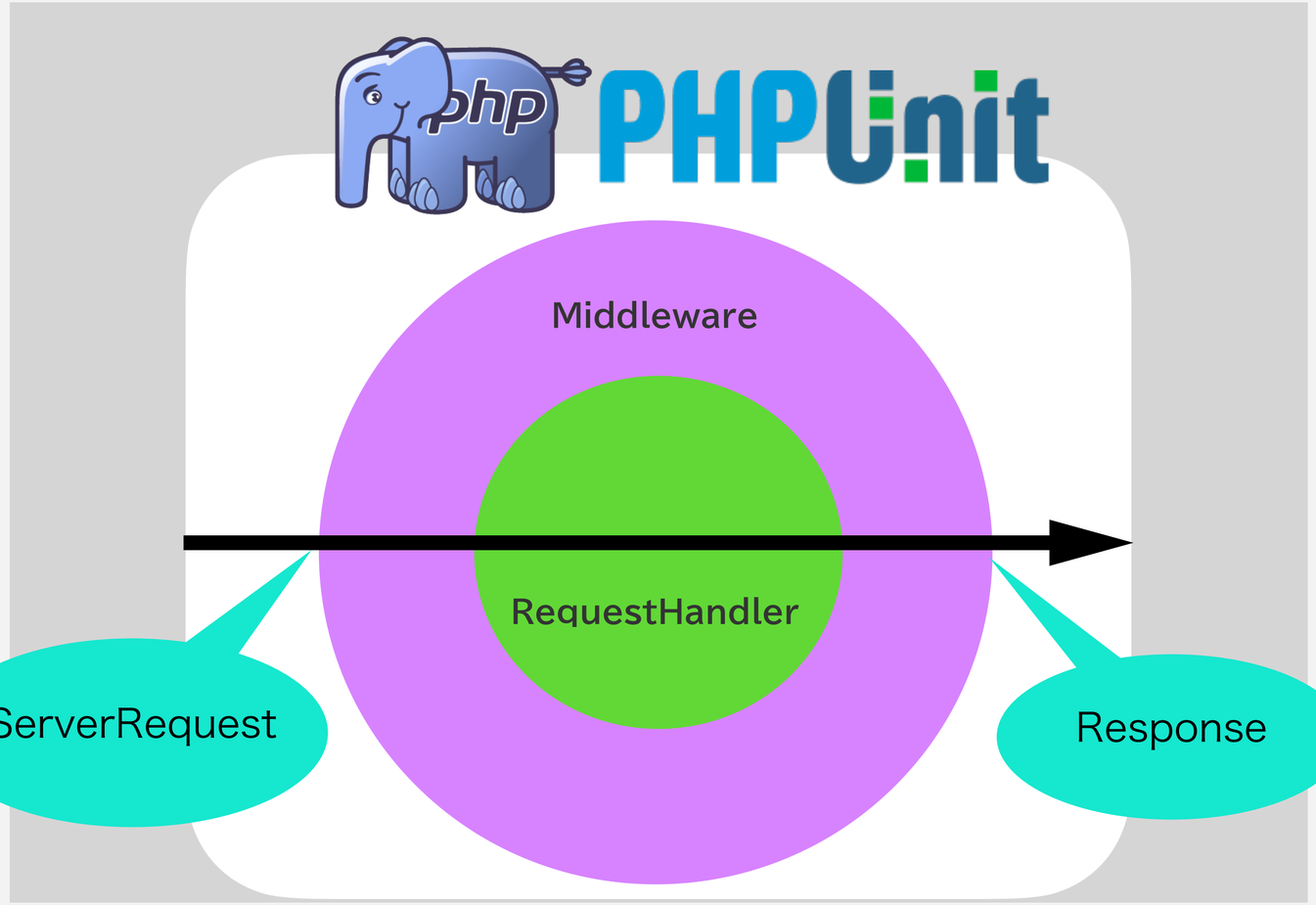
- もちろんPHPUnitを使える
- テスト対象のMiddlewareのインスタンスを用意する
- ServerRequestとResponseHanlderのインスタンスを用意する
- $response = $middleware->process($request, $handler)のように呼び出す
- $responseからステータスコード、HTTPヘッダ、ボディなどを取り出してassertEquals()などで比較してみる

Middleware呼び出しイメージ



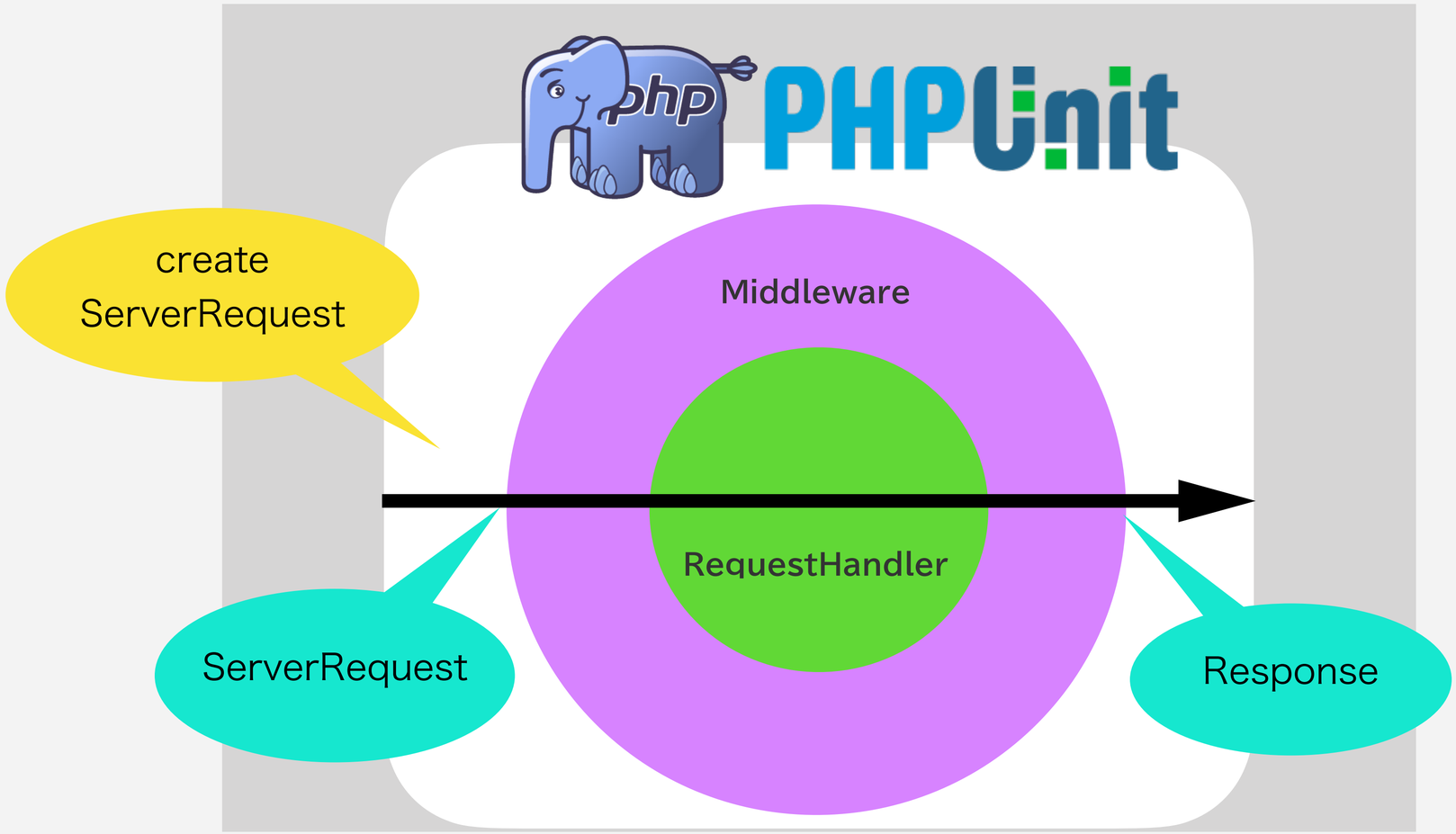
pixiv

Middleware呼び出しイメージ



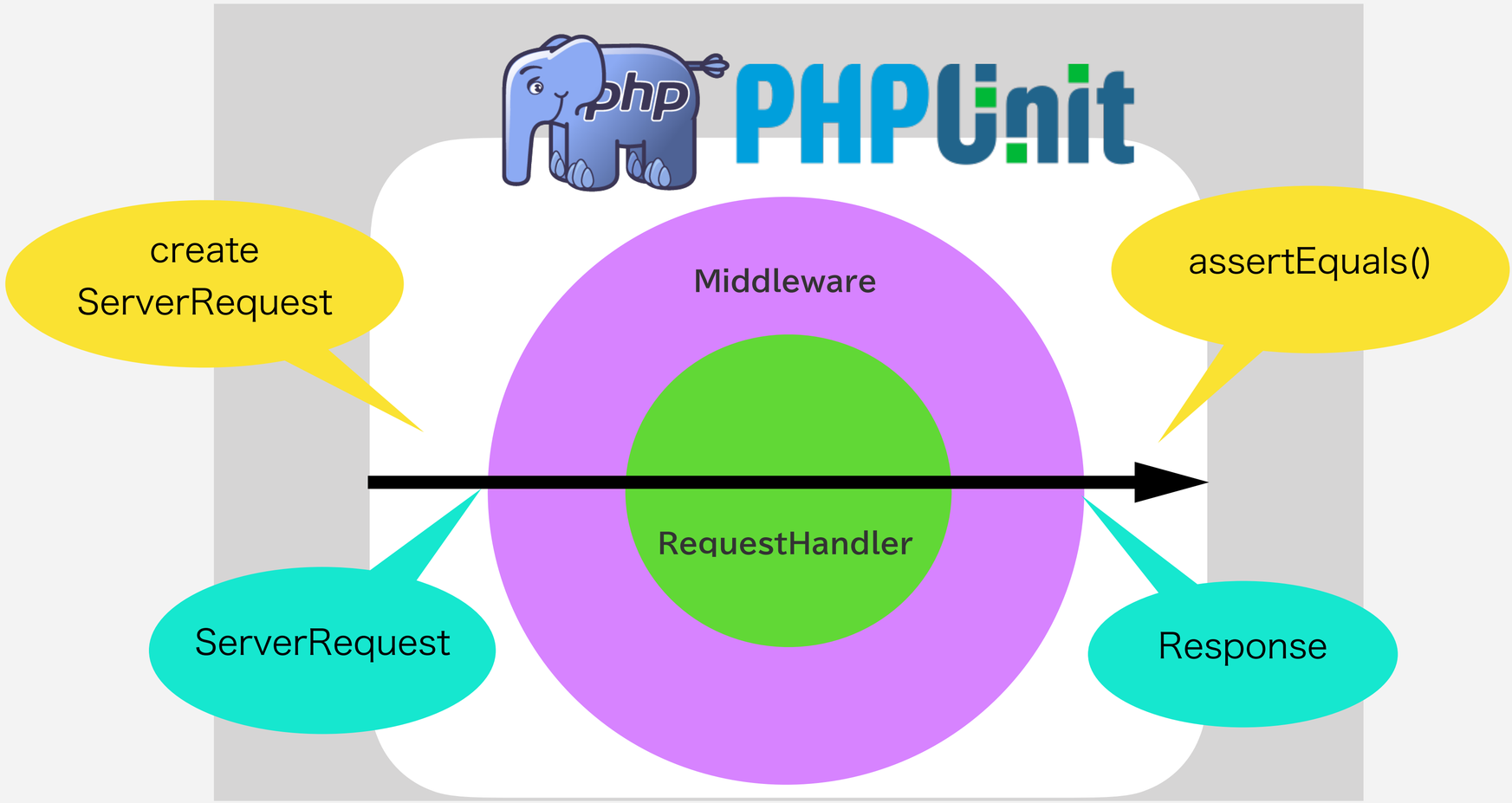
pixiv

Middleware呼び出しイメージ



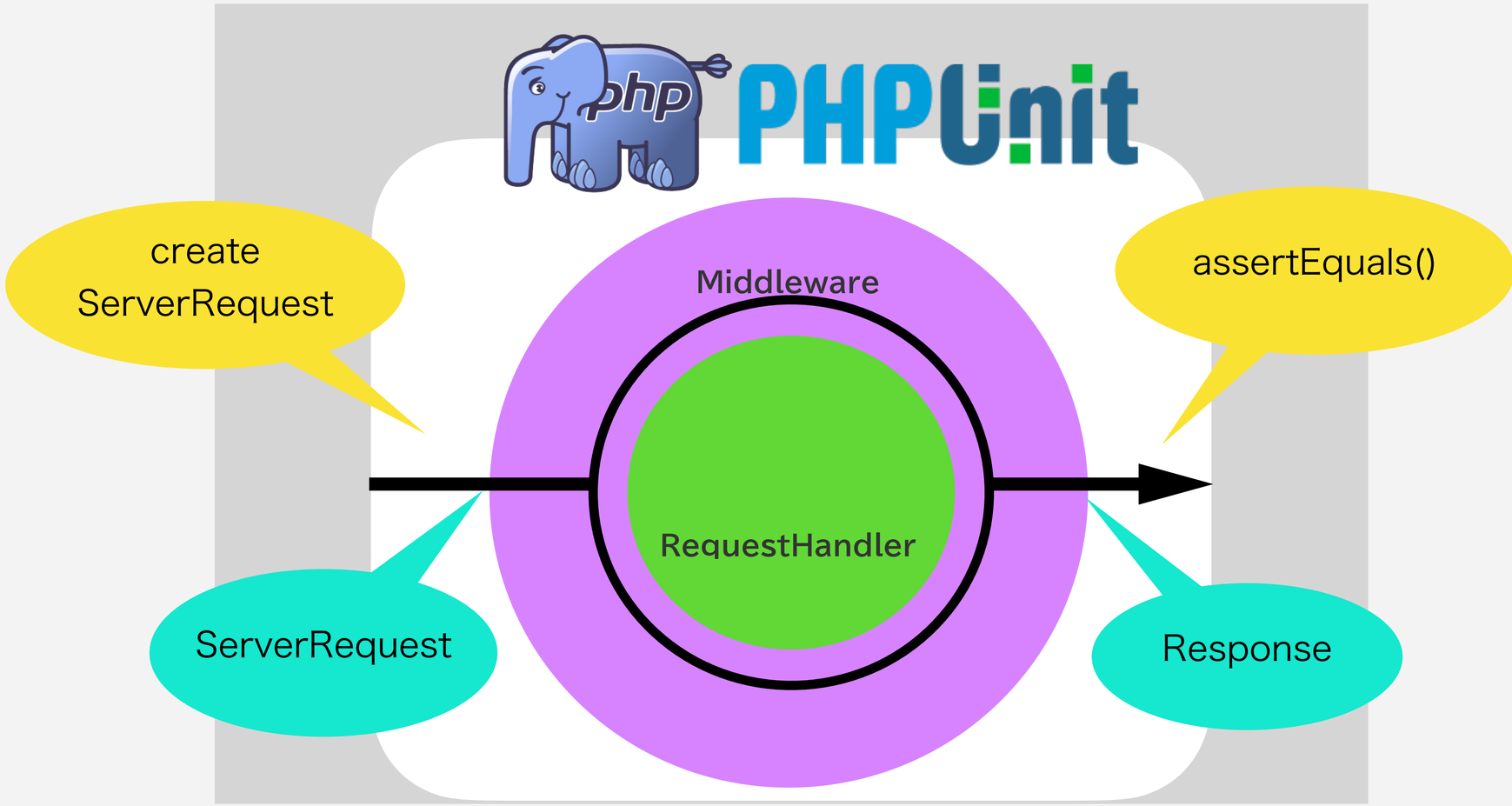
pixiv

Middleware呼び出しイメージ



pixiv

Middleware呼び出しイメージ



pixiv

Middlewareのテスト(?)

```
    /**
     * @dataProvider requestProvider
     */
    public function test(ServerRequest $request, array $expected): void
    {
        $actual = $this->subject->process($request, $handler);

        $this->assertSame($expected['status_code'], $actual->getStatusCode());
        $this->assertEquals($expected['headers'], $actual->getHeaders());
        $this->assertSame($expected['body'], (string)$actual->getBody());
    }
```

Middlewareのテスト

```
    /**
     * @dataProvider requestProvider
     */
    public function test(ServerRequest $request, array $expected): void
    {
        $actual = $this->subject->process($request, $handler);

        $this->assertSame($expected['status_code'], $actual->getStatusCode());
        $this->assertEquals($expected['headers'], $actual->getHeaders());
        $this->assertSame($expected['body'], (string)$actual->getBody());
    }
```

Middlewareのテスト

```
/**
 * @dataProvider requestProvider
 */
public function test(ServerRequest $request, array $expected): void
{
  $ok = $this->createResponse(200);
  $actual = $this->subject->process($request, new class($ok) implements RequestHandler {
    public function __construct(private Response $response) {}
    public function handle(ServerRequest $request): Response {
      return $this->response;
    }
  });
```

Middlewareのテスト

固定のレスポンスを返すだけ

```
/**
 * @dataProvider requestProvider
 */
public function test(ServerRequest $request, array $expected): void
{
  $ok = $this->createResponse(200);
  $actual = $this->subject->process($request, new class($ok) implements RequestHandler {
    public function __construct(private Response $response) {}
    public function handle(ServerRequest $request): Response {
      return $this->response;
    }
  });
```

Middlewareのテスト

固定のレスポンスを返すだけ

```
/**
 * @dataProvider requestProvider
 */
public function test(ServerRequest $request, array $expected): void
{
  $ok = $this->createResponse(200);
  $actual = $this->subject->process($request, new class($ok) implements RequestHandler {
    public function __construct(private Response $response) {}
    public function handle(ServerRequest $request): Response {
      return $this->response;
    }
  });
```

3行で実装できるが…
まあまあだるい

大雑把な処理の流れ

- どうにかしてServerRequestオブジェクトを用意する
- 最終的に処理するRequestHandlerオブジェクトを用意する
- 間に処理するMiddlewareのオブジェクトを用意する
- Middleware::process($request, $handler)として呼び出すと最終的なResponseが帰ってくる
- エミッタがResponseをどうにか出力する

テストの準備 (RawHandler)

```
class RawHandler implements RequestHandlerInterface {
  /**
   * @psalm-var list<ServerRequest>
   * @psalm-readonly-allow-private-mutation
   */
  public $received_server_requests = [];
  public function __construct(private Response $response) {}

  public function handle(ServerRequest $request): Response {
    $this->received_server_requests[] = $request;
    return $this->response;
  }
```

Middlewareのテスト (無名クラス)

```
/**
 * @dataProvider requestProvider
 */
public function test(ServerRequest $request, array $expected): void
{
  $ok = $this->createResponse(200);
  $actual = $this->subject->process($request, new class($ok) implements RequestHandler {
    public function __construct(private Response $response) {}
    public function handle(ServerRequest $request): Response {
      return $this->response;
    }
  });
```

Middlewareのテスト (RawHandler)

```
    /**
     * @dataProvider requestProvider
     */
    public function test(ServerRequest $request, array $expected): void
    {
        $ok_handler = new RawHandler($this->createResponse(200));
        $actual = $this->subject->process($ok_handler);

        $this->assertSame($expected['status_code'], $actual->getStatusCode());
        $this->assertEquals($expected['headers'], $actual->getHeaders());
        $this->assertSame($expected['body'], (string)$actual->getBody());
    }
```

Middlewareのテスト (RawHandler)

```
  public function test(ServerRequest $request, array $expected): void
  {
    $ok_handler = new RawHandler($this->createResponse(200));
    $actual = $this->subject->process($ok_handler);

    $this->assertEquals($expected, [
      'status_code' => $actual->getStatusCode());
      'headers'=> $actual->getHeaders(),
      'body' => (string)$actual->getBody(),
      'not_redirected_count' => count($ok_handler->received_server_requests),
    );
  }
```

Middlewareのテスト (RawHandler)

```
public function test(ServerRequest $request, array $expected): void
{
  $ok_handler = new RawHandler($this->createResponse(200));
  $actual = $this->subject->process($ok_handler);

  $this->assertEquals($expected, [
    'status_code' => $actual->getStatusCode());
    'headers'=> $actual->getHeaders(),
    'body' => (string)$actual->getBody(),
    'not_redirected_count' => count($ok_handler->received_server_requests),
  );
}
```

$ok_handlerがコールされた回数

Middlewareテストの要点

- 基本はRequestHandlerと同様にシンプルかつ明瞭にする
- どのようなハンドラを渡せばMiddlewareが正常に動作しているか
  判断できるかを見極める
  - テスト用ハンドラはこだわらなくてもパターン化しやすい
  - リクエストに影響を及ぼすハンドラならリクエストの内容もチェックする
- 実アプリケーションで呼ばれる他のハンドラに依存すると
  何をテストしたいのかぶれるので基本的には避ける

その他のハンドラ (関数のラッパー)

```php
class CallbackHandler implements RequestHandlerInterface {
  /**
   * @psalm-var list<ServerRequest>
   * @psalm-readonly-allow-private-mutation
   */
  public $received_server_requests = [];
  public function __construct(private Closure $callback) {}

  public function handle(ServerRequest $request): Response {
      $this->received_server_requests[] = $request;
      return ($this->callback)($request);
  }
```

その他のハンドラ (例外を投げる君)

```
class ThrowingHandler implements RequestHandlerInterface {
  /**
   * @psalm-var list<ServerRequest>
   * @psalm-readonly-allow-private-mutation
   */
  Public $received_server_requests = [];
  public function __construct(private Throwable $e) {}

  public function handle(ServerRequest $request): Response {
      $this->received_server_requests[] = $request;
      throw $this->e;
  }
```

実アプリケーションへの適用を考える

pixiv

大雑把な処理の流れ

- どうにかしてServerRequestオブジェクトを用意する
- 最終的に処理するRequestHandlerオブジェクトを用意する
- 間に処理するMiddlewareのオブジェクトを用意する
- $middleware->process($request, $handler)として呼び出すと最終的なResponseが帰ってくる
- エミッタがResponseをどうにか出力する

大雑把な処理の流れ

nyholm/
psr7-server

- どうにかしてServerRequestオブジェクトを用意する
- 最終的に処理するRequestHandlerオブジェクトを用意する
- 間に処理するMiddlewareのオブジェクトを用意する
- $middleware->process($request, $handler)として呼び出すと最終的なResponseが帰ってくる
- エミッタがResponseをどうにか出力する

pixiv

大雑把な処理の流れ

nyholm/
psr7-server

ちくちく定義する

- どうにかしてServerRequestオブジェクトを用意する
- 最終的に処理するRequestHandlerオブジェクトを用意する
- 間に処理するMiddlewareのオブジェクトを用意する
- $middleware->process($request, $handler)として呼び出すと
  最終的なResponseが帰ってくる
- エミッタがResponseをどうにか出力する

pixiv

大雑把な処理の流れ

nyholm/
psr7-server

ちくちく定義する

- どうにかしてServerRequestオブジェクトを用意する
- 最終的に処理するRequestHandlerオブジェクトを用意する
- 間に処理するMiddlewareのオブジェクトを用意する
- $middleware->process($request, $handler)として呼び出すと
  最終的なResponseが帰ってくる
- エミッタがResponseをどうにか出力する

Laminasのエミッタを使う

pixiv

大雑把な処理の流れ

nyholm/
psr7-server

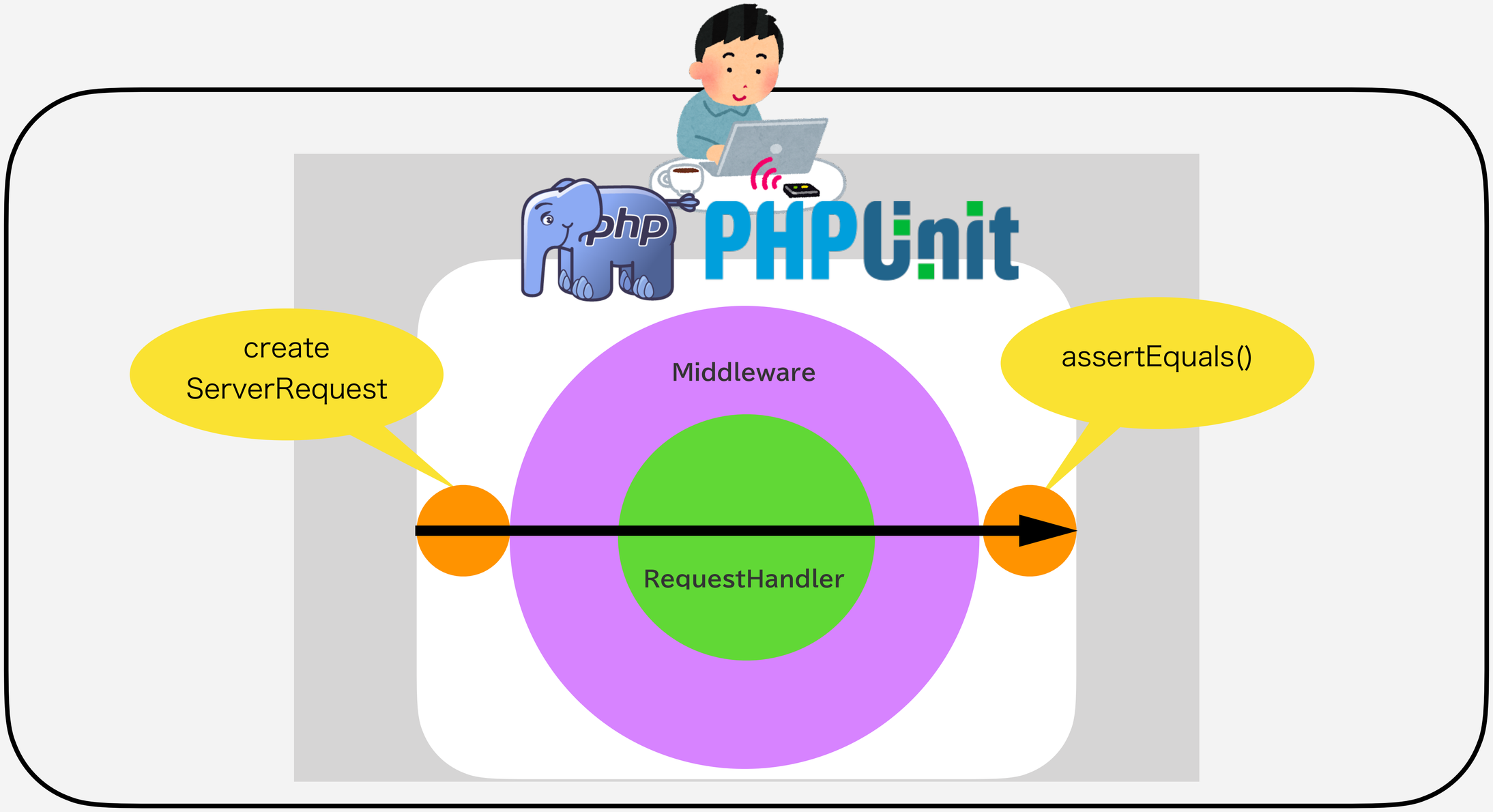
ちくちく定義する

- どうにかしてServerRequestオブジェクトを用意する
- 最終的に処理するRequestHandlerオブジェクトを用意する
- 間に処理するMiddlewareのオブジェクトを用意する
- $middleware->process($request, $handler)として呼び出すと
  最終的なResponseが帰ってくる
- エミッタがResponseをどうにか出力する

この$middleware何者

Laminasのエミッタを使う

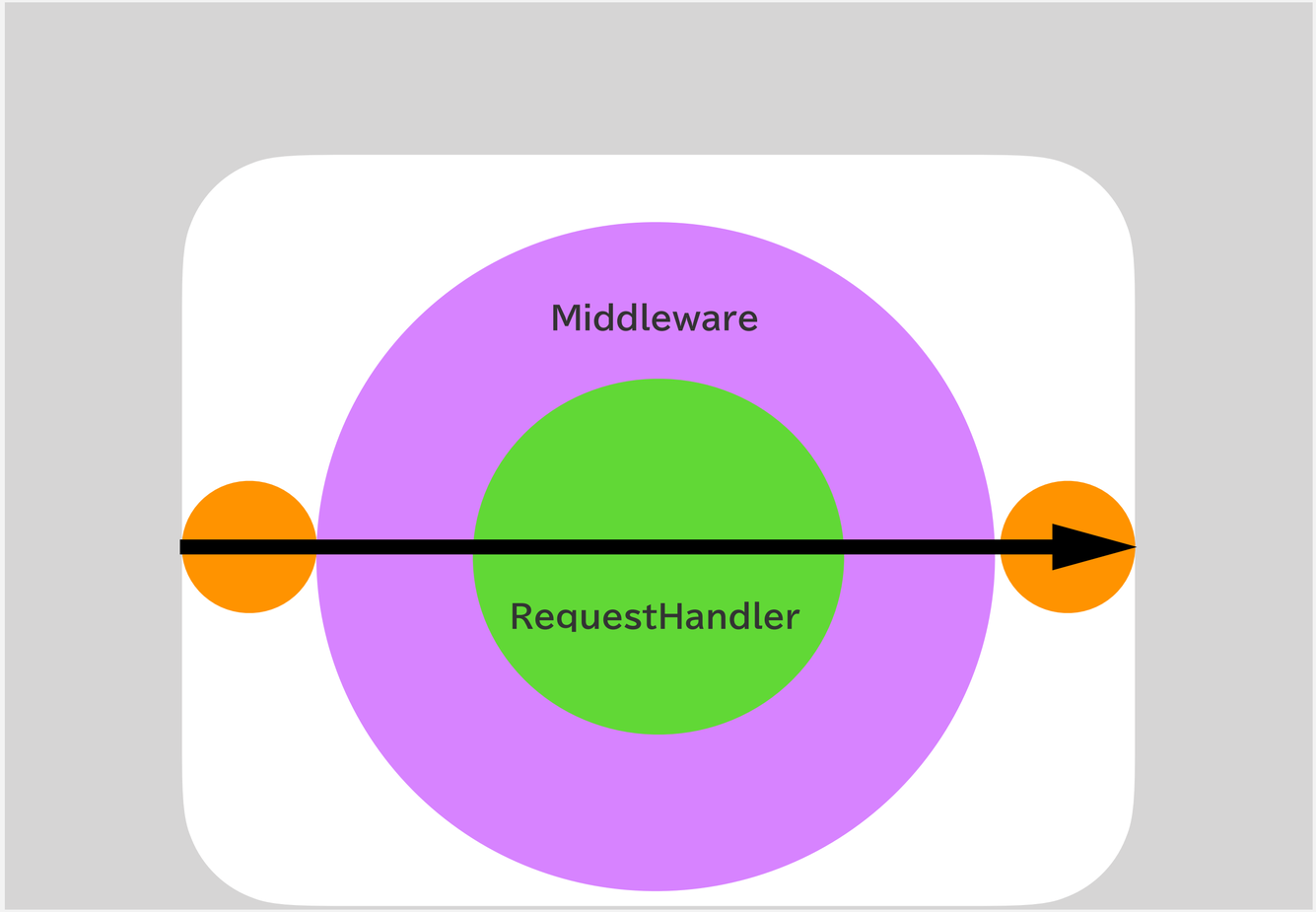
pixiv


PHPUnit
create
ServerRequest
Middleware
RequestHandler
assertEquals()

pixiv

これまでは
Middleware1個
Handler1個
しか考えてない

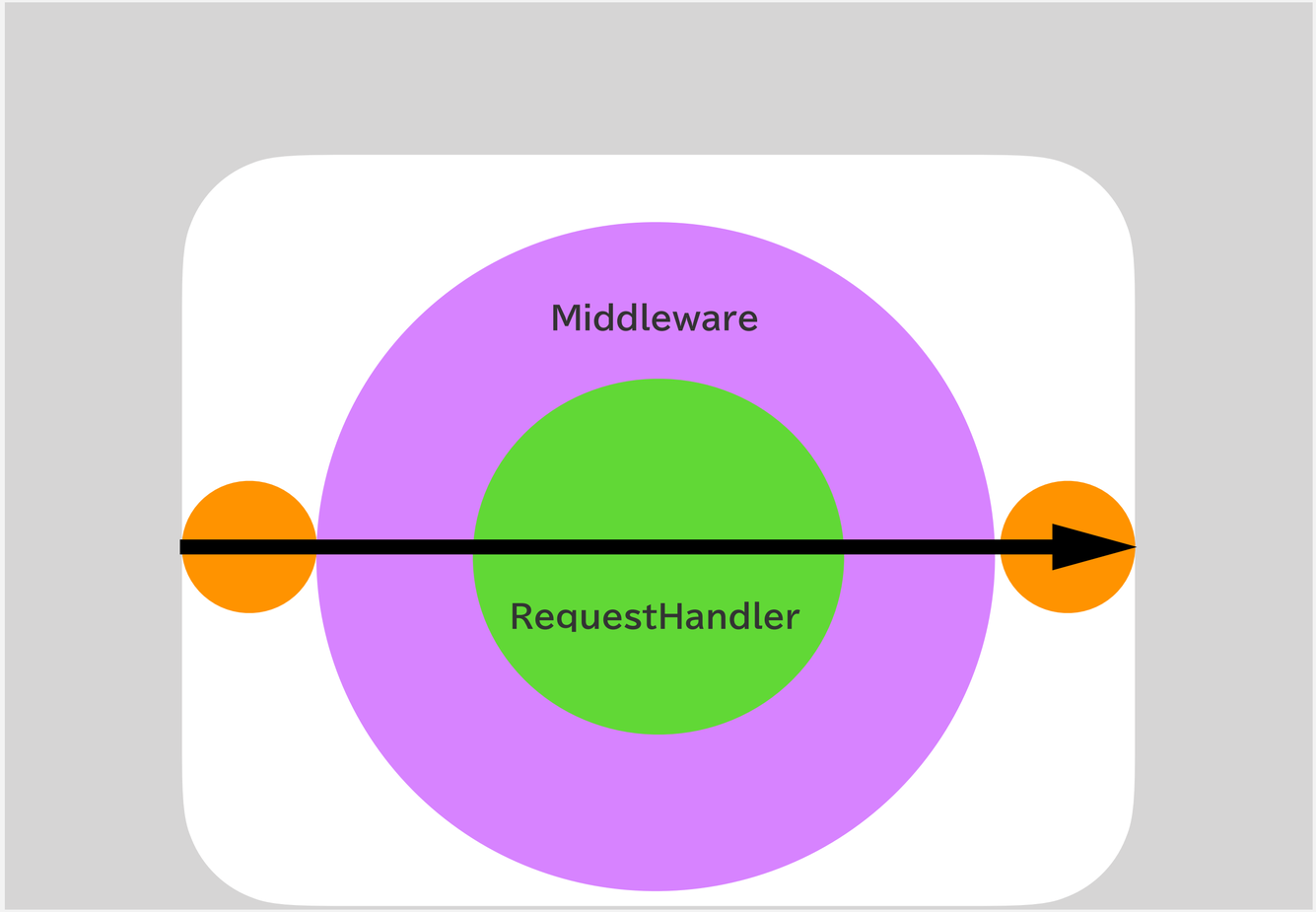
いままでのミドルウェア実行イメージ



pixiv

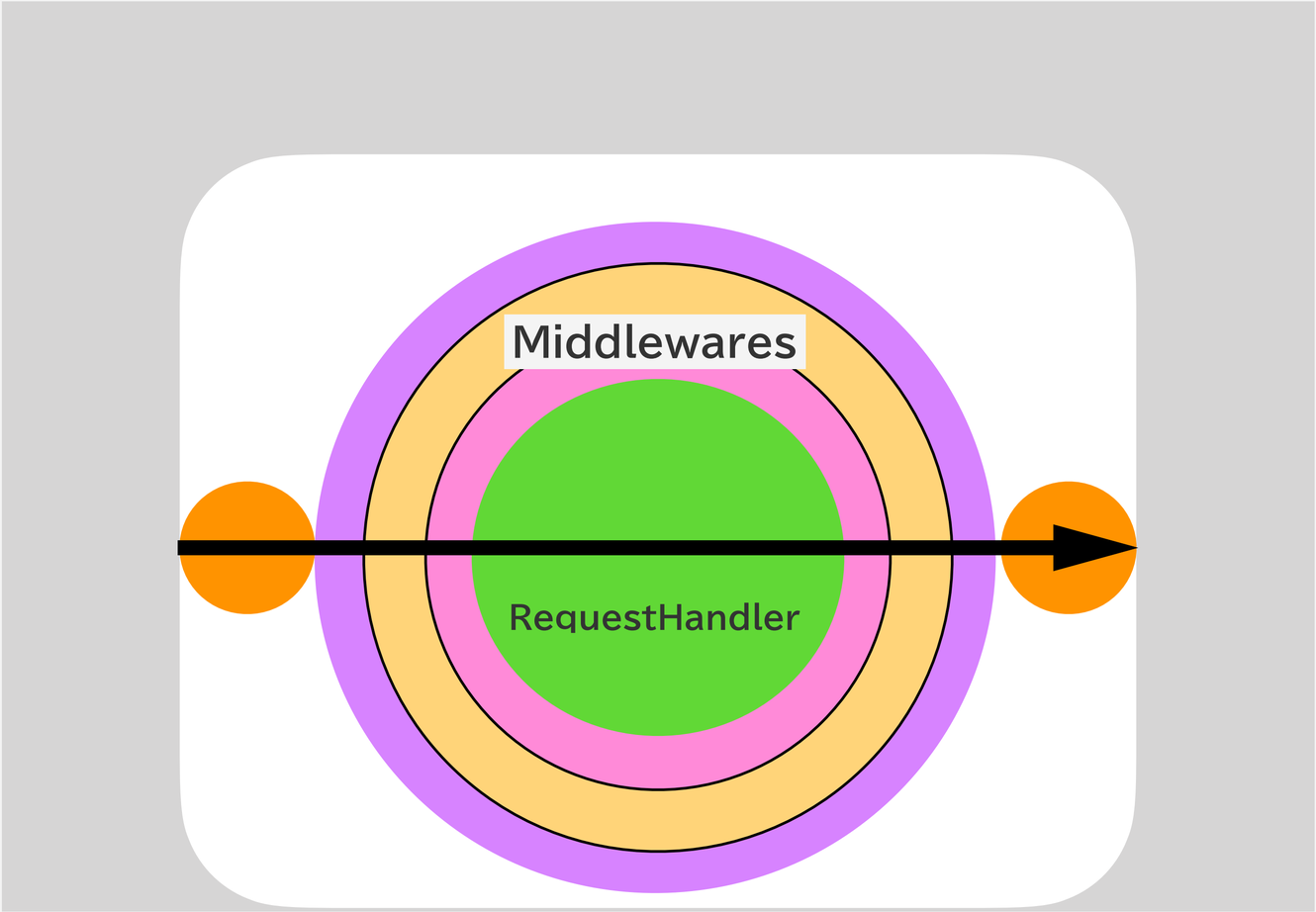
せっかくの既存
ミドルウェアたくさん
重ねたくないですか

いままでのミドルウェア実行イメージ



pixiv

ミドルウェアを重ねたい



pixiv

複数のミドルウェアを扱う

- MiddlewareはServerRequestとRequestHandlerを受け取り、$request_handler->handle($request)のように呼び出す
  - このままではミドルウェア1個ハンドラ1個しか受け付けない…？
- 「次のMiddleware」をラップしたRequestHandlerを用意する
  - PSR-15のメタドキュメントでは「キュー」と「デコレータ」の2種類が紹介
- 特に理由がなければRelayというやつを選ぶといい感じ
  - 自作しても全然よい

セッションってどうするの？

- SessionMiddleware的なものを作る
  - RequestのCookieなどのセッションIDからDBなどを引く
  - session_start()や$_SESSIONのようなグローバルを参照する部分はさらに別のクラスに分けるとテストしやすくできる
- $request->withAttribute()で付加することで後続のMiddlewareやRequestHandlerで $request->getAttribute()で取り出せる

セッションを参照する

```
// 前略
public function handle(ServerRequestInterface $request): ResponseInterface {
    $session = $request->getAttribute(Session::class);
    assert($session instanceof Session);

    // $session を使った処理を書く
    if ($session instanceof LoggedIn) {
        // ...
    }
}
```

どうやってユーザーに
コードを書かせるか

pixiv

「心に棚を作れ」(by 炎の転校生)

- 独自FWを作るとき、全体的な枠組みを作るのもアプリ実装するのも自分
  - 「この変な設計したやつは誰だ」「**俺だ**」
  - 「規約を守れず腑抜けたアプリケーションを書くのは誰だ」「**俺だ**」
- 「それはそれ、これはこれ」 by 逆境ナイン
  - 微妙におかしな設計をする自分と設計意図通りにアプリケーションを実装できない自分がいる
  - 妥協せずにバトルさせたあとに使いやすいフレームワークができる(気がする)

ここまでできれば
外の枠組みは
結構できてきた

アプリケーションってどこにあるの？

いわゆるコントローラやアクション

- URLごとに対応して起動される関数やメソッドをどうする？
- 基本はRequestHandlerから、どうにか関数決めて起動してやればいい
- RequestHandlerはLaravelのSingleActionControllerと
  概念的に近いものだと考えることもできる
  - ただし引数や戻り値のインターフェイスが厳密
- PHP-DIのようなライブラリを組み合せることでインスタンス生成や
  関数呼び出しを簡略化することもできる

phperkaigi-golf

- https://github.com/phppg/phperkaigi-golf
  - PHPerKaigi 2020のときに実装したWebアプリケーション
  - PSR-7/PSR-15ベース / Relayを使用
  - PHP-DIを使用
    - ルーティングに無名関数(クロージャ)を登録して実行するマイクロフレームワーク風の実装

Index.phpのルーティング定義(抜萃)

```
$router = new RouterContainer();
$map = $router->getMap();

$map->get('index', '/', fn (
    ResponseFactory $factory,
    StreamFactory $stream,
    View\HtmlFactory $html
): HttpResponse => $factory->createResponse()->withBody($stream->createStream($html('index', [
])));
```

Index.phpのルーティング定義(抜萃)

```php
$map->post('post_terms', '/terms', fn (Http\TermsAgreementAction $action, ServerRequest $request): HttpResponse => $action($request));

$map->post('post_login', '/login', fn (Http\LoginAction $action, ServerRequest $request): HttpResponse => $action($request));

$map->get('phpinfo', '/phpinfo.php', function (ResponseFactory $factory, StreamFactory $stream) {
    ob_start();
    phpinfo();

    return $factory->createResponse()->withBody($stream->createStream(ob_get_clean() ?: ''));
});
```

Index.phpのミドルウェア定義(1/2)

```
$queue[] = fn (ServerRequest $request, RequestHandler $handler): HttpResponse
    => $handler->handle($request)->withHeader('X-Powered-By', 'PHP/' . PHP_VERSION)->withHeader('X-Robots-Tag', 'noindex');
$queue[] = new IpAddress();
$queue[] = $container->get(Http\Dispatcher::class);
$queue[] = $container->get(Http\SessionSetter::class);
$queue[] = function (ServerRequest $request, RequestHandler $handler) use ($http, $router): HttpResponse {
    $session = $request->getAttribute('session');
    assert($session instanceof Session);

    $uri = $request->getUri()->getPath();
    if (!in_array($uri, ['/', '/terms', '/login', '/phpinfo.php'], true) && !$session->isLoggedIn()) {
        return $http->createResponse(302)->withHeader('Location', [
            $router->getGenerator()->generate('terms'),
        ]);
    }

    return $handler->handle($request);
};
```

Index.phpのミドルウェア定義(2/2)

```
$queue[] = fn (ServerRequest $request): HttpResponse
    => $container->call($router->getMatcher()->match($request)->handler ?? $_404, [
        'request' => $request,
    ]);

$relay = new Relay($queue);
$response = $relay->handle($server_request);

$container->get(Emitter::class)->emit($response);
~
~
```

ピクシブ百科事典

- http://dic.pixiv.net/
  - 2009年から運営されているWebサービス
  - PSR-7/PSR-15ベース / Relay / PHP-DIを使用
    - もともとCakePHP風のController/Modelの独自フレームワーク
    - 現在はPSR-15でコントローラを書けるようにしている
- 全体的な雰囲気は「百科事典 インターン PSR-7」とかで
  Google検索してもらうと概観がわかってもらえると思います

Cookieを扱う難しさ

- withCookieParams()などは$_COOKIE相当のデータを付加するが、内部で保持するHTTPヘッダを書き換えてはならない(MUST NOT)と仕様で明言されている
- プレーンなPHPではsetcookie()すればよかったが、PSR-7ベースではプリミティブすぎて、setHeaderで直接書かないといけない

PSR-7がプリミティブすぎる件

- 外部からの入力を型安全に受け取るには工夫が必要
  - filter_var()とか使うのが一番安全
  - 最近ではPSLというライブラリもいい感じ
  - ParamHelperにPSR-7とValueObject の力を授けた話
- マルチパートってどうやって扱えばいいの…？
  - いいライブラリあれば誰か教えてください

ParamHelperにPSR-7とValueObject の力を授けた話

ParamHelperにPSR-7とValueObject の力を授けた話

Article by ふじしゃん 2022-03-30

こんにちは、VTuberとPHP をこよなく愛しているふじしゃんです。
去年の7月からpixiv運営本部 Webエンジニアリングチームでアルバイトをしています。

今回は、pixivの `ParamHelper` にPSR-7とValueObjectの力を授けた `RequestParamFilter` をピクシブ百科事典に実装した話を書いていきます。

ParamHelper について

これまでピクシブ百科事典には、リクエストパラメータやリクエストボディを厳密に検証する仕組みがありませんでした。
Webアプリケーションにとって入力値検証は非常に重要なことです。

pixivでは、受け取った値を安全に扱うために `ParamHelper` という機能を実装し、必ず検証するようにしています。

以下のように書くことで値を検証しPHPStanで型付けを行うことができます。

pixiv

今回の発表の参照実装とか便利なライブラリとかそのうちいろいろ出します！！！！

俺たちのPSR-15道はこれからだ！

(ご静聴ありがとうございました。
tadsanの次回作にご期待ください)

pixiv