CONTINUE PART 2 : USING REMOTE APIS

Fetch Remote Data Using Guzzle

With Ashraff Hathibelagal

@hathibel

2- GUZZLE

- Guzzle is a PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services.
- Secondly, Guzzle provides a very clean API to work with. Documentation is very important when working with a library.Guzzle has done a very good job by providing a comprehensive documentation

WHAT IS GUZZLE ?

- Wait a minute what is composer ????

INSTALLING GUZZLE USING COMPOSER

- **It's a dependancy manager tool it is used everywhere in the PHP World. All large and well-known website components in other ward it will help you do the following :**
- **1- Install 3rd party php tools and framewords**
- **2- Update version of the tools you use**
- **3- Auto load your own code (so in the future you will only use composer's autoloader!!! Cool ?**
-

- Composer has two separate elements :
  - ✓ *The first is Composer itself which is a command line tool for grabbing and installing what you want to install.*
  - ✓ *The second is Packagist - the main composer repository. This is where the packages you may want to use are stored.*
  - ✓ *Every thing Composer installs does that in a directory names vendor which shouldn't has any of your code*

OVERVIEW OF COMPOSER

- Working with composer will has only 3 task types :
- 1- Define what you want to install (frame work or tool) in json format file
- 2- Define where your classes files and configuration file or in the future name spaces which you want to load in json file
- 3- Some command files to make composer run the settings you define
- 4- Require the autoloader of composer in your code

WORKING WITH COMPOSER

1

- First Step
- Prepare composer.JSON File to tell composer what you want to install

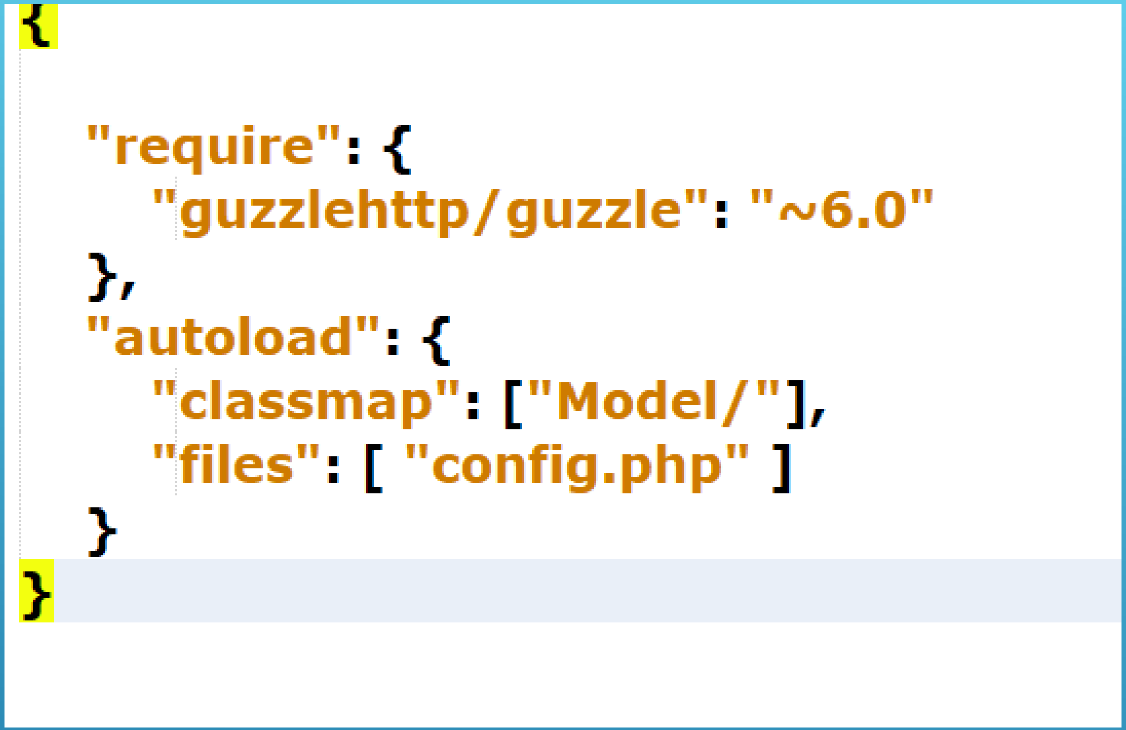
2

- Second Step
- Prepare composer.json file about your autoload options

3

- Third step
- Run composer install command

WORK SEQUENCE

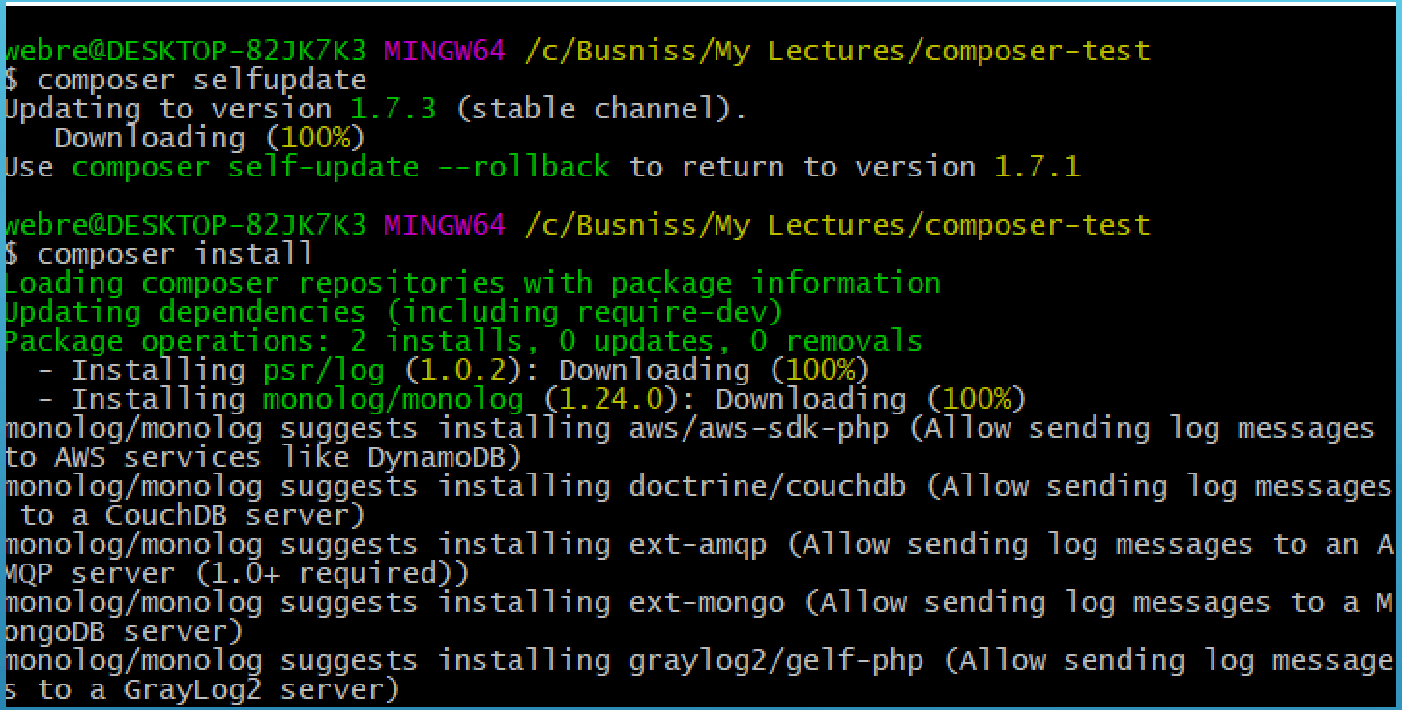


```
{

  "require": {
    "guzzlehttp/guzzle": "~6.0"
  },
  "autoload": {
    "classmap": ["Model/"],
    "files": [ "config.php" ]
  }
}
```

The Composer.JSON has two sections :

- Require : The tools that you want Composer to install
- Autoload : where are your classes and configuration files for composer to load , in the future that will be using name space not classmap.

1 & 2 COMPOSER.JSON FILE PREPARATION

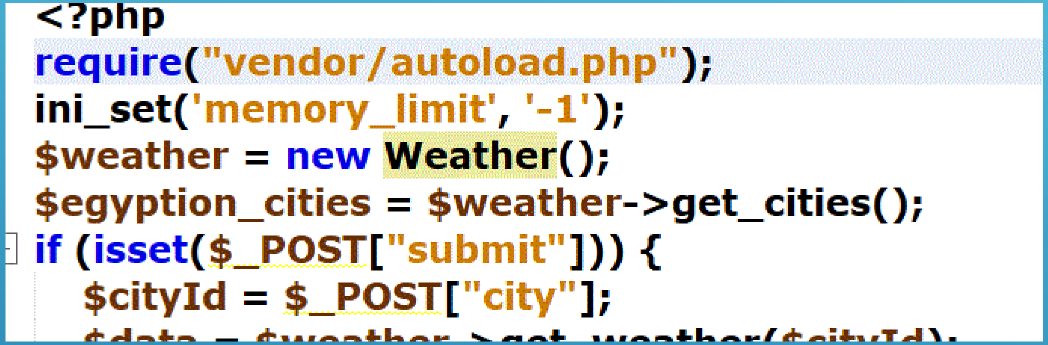


```
webre@DESKTOP-82JK7K3 MINGW64 /c/Busniss/My Lectures/composer-test
$ composer selfupdate
Updating to version 1.7.3 (stable channel).
   Downloading (100%)
Use composer self-update --rollback to return to version 1.7.1

webre@DESKTOP-82JK7K3 MINGW64 /c/Busniss/My Lectures/composer-test
$ composer install
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing psr/log (1.0.2): Downloading (100%)
  - Installing monolog/monolog (1.24.0): Downloading (100%)
monolog/monolog suggests installing aws/aws-sdk-php (Allow sending log messages
to AWS services like DynamoDB)
monolog/monolog suggests installing doctrine/couchdb (Allow sending log messages
 to a CouchDB server)
monolog/monolog suggests installing ext-amqp (Allow sending log messages to an A
MQP server (1.0+ required))
monolog/monolog suggests installing ext-mongo (Allow sending log messages to a M
ongoDB server)
monolog/monolog suggests installing graylog2/gelf-php (Allow sending log message
s to a GrayLog2 server)
```

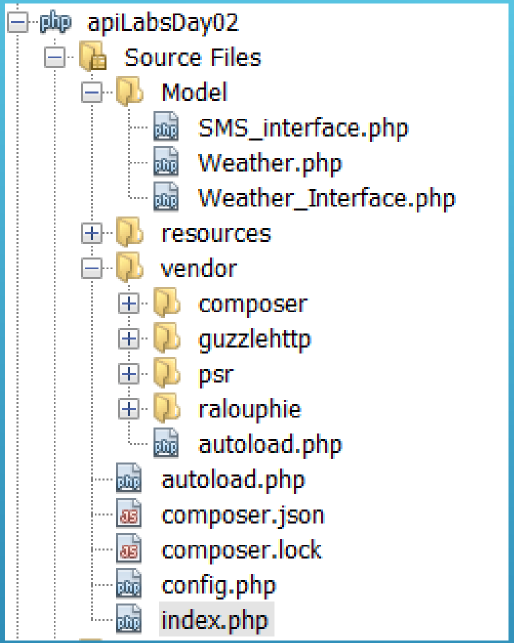
- Composer selfupdate
- Composer install
- Composer dump-autoload
- Note :
- When call composer install composer will add some files and a directory with name vendor in which all 3rd party tools are

COMMAND LINE

Require auto uploader

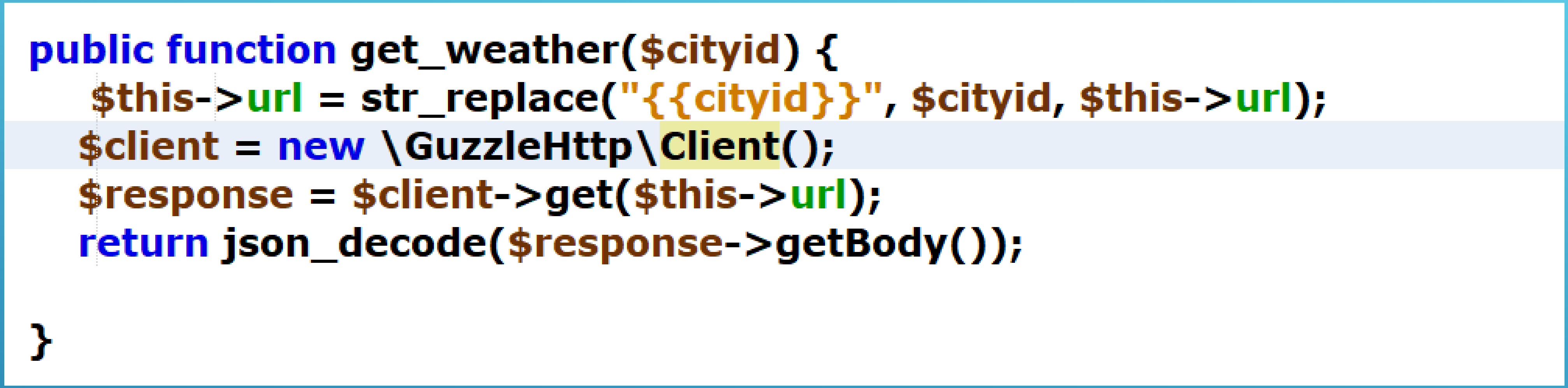


```php
<?php
require("vendor/autoload.php");
ini_set('memory_limit', '-1');
$weather = new Weather();
$egyption_cities = $weather->get_cities();
if (isset($_POST["submit"])) {
    $cityId = $_POST["city"];
    $data = $weather->get_weather($cityId);
```



```
apiLabsDay02
    Source Files
        Model
            SMS_interface.php
            Weather.php
            Weather_Interface.php
        resources
        vendor
            composer
            guzzlehttp
            psr
            ralouphie
            autoload.php
        autoload.php
        composer.json
        composer.lock
        config.php
        index.php
```

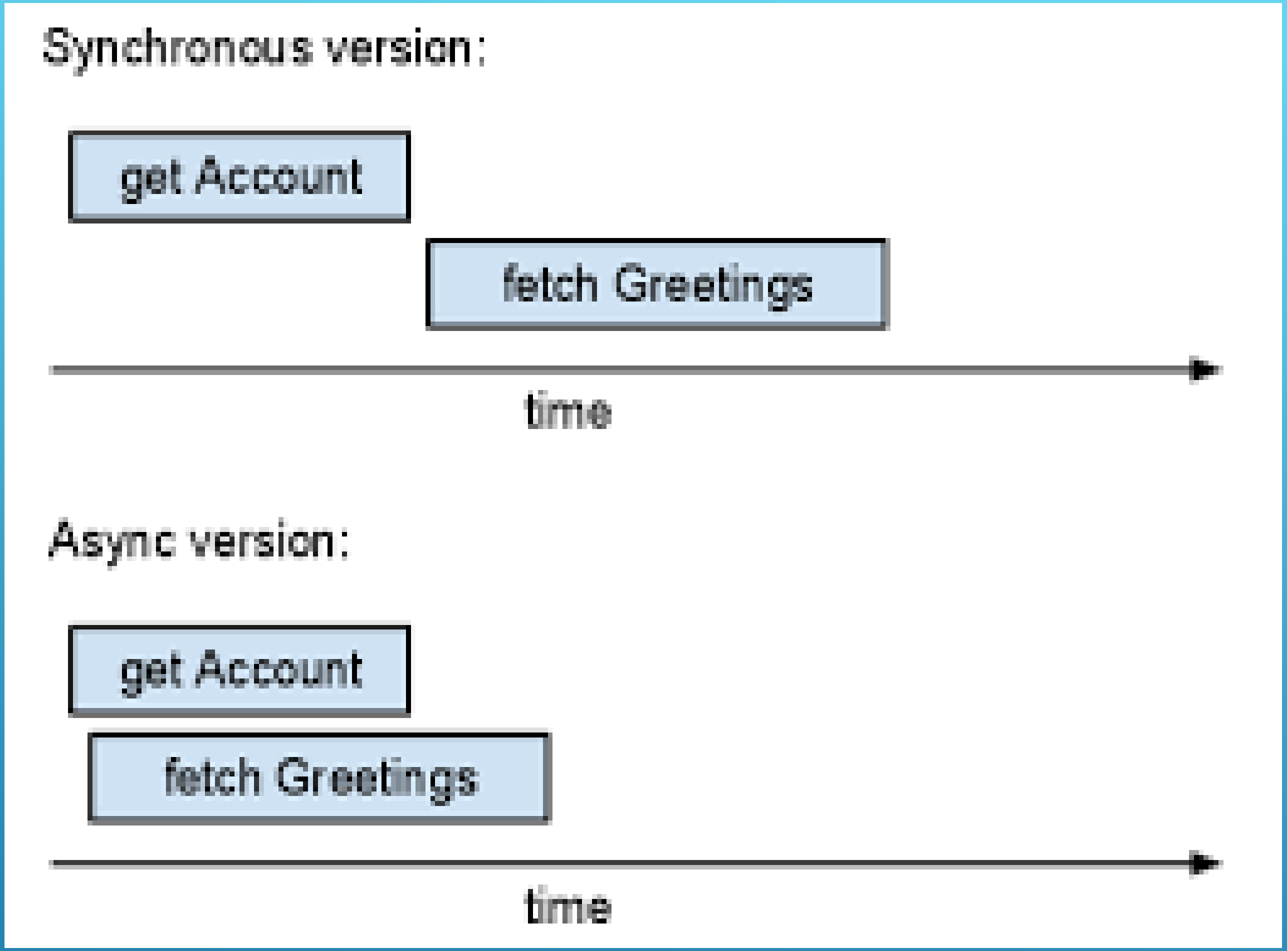
IN THE CODE



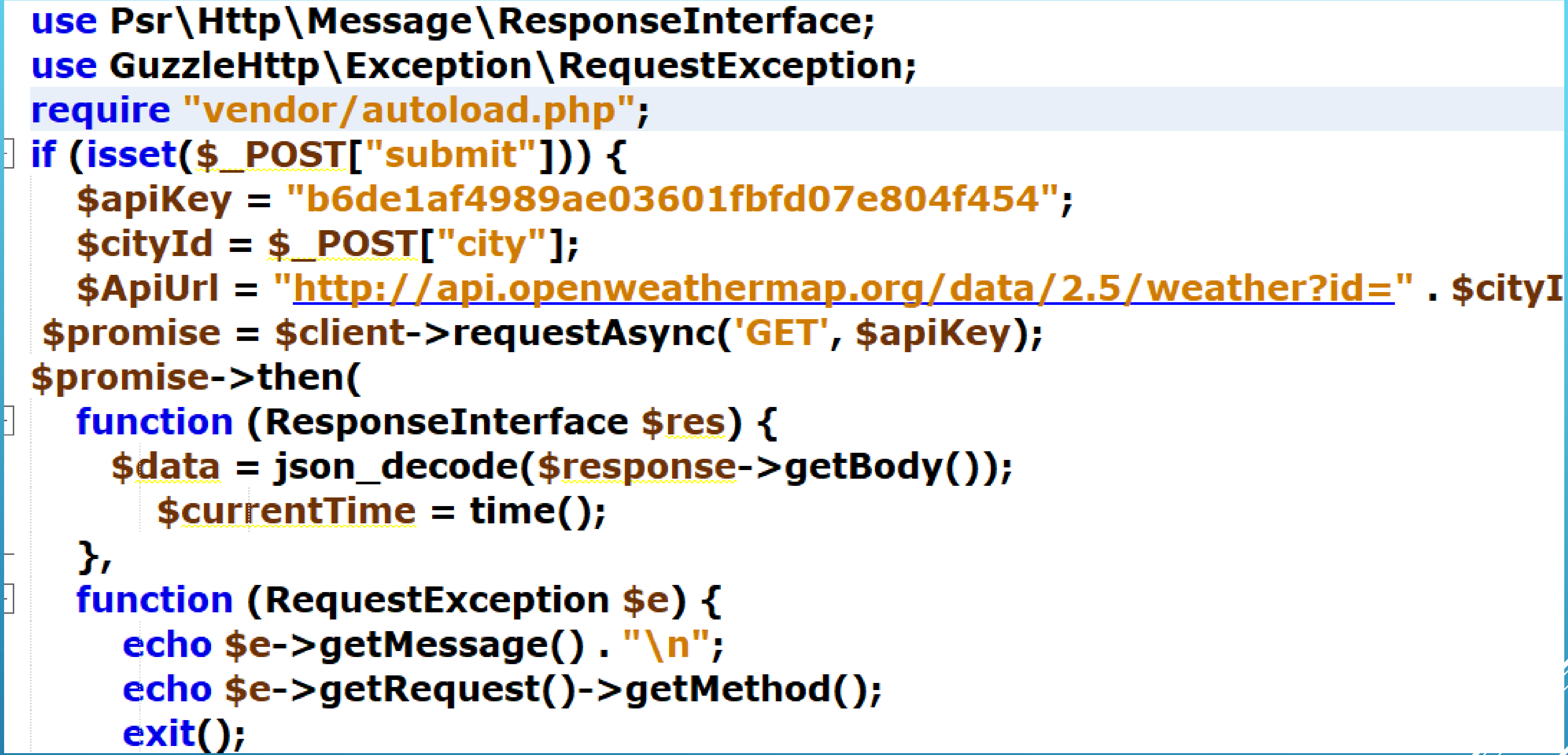
```
public function get_weather($cityid) {
     $this->url = str_replace("{{cityid}}", $cityid, $this->url);
    $client = new \GuzzleHttp\Client();
    $response = $client->get($this->url);
    return json_decode($response->getBody());

}
```

SAME WEATER EXAMPLE USING GUZZLE

USING GUZZLE YOU CAN DO ASYNC AND CONCURRENT REQUESTS <<ADVANCED BEGIN>>

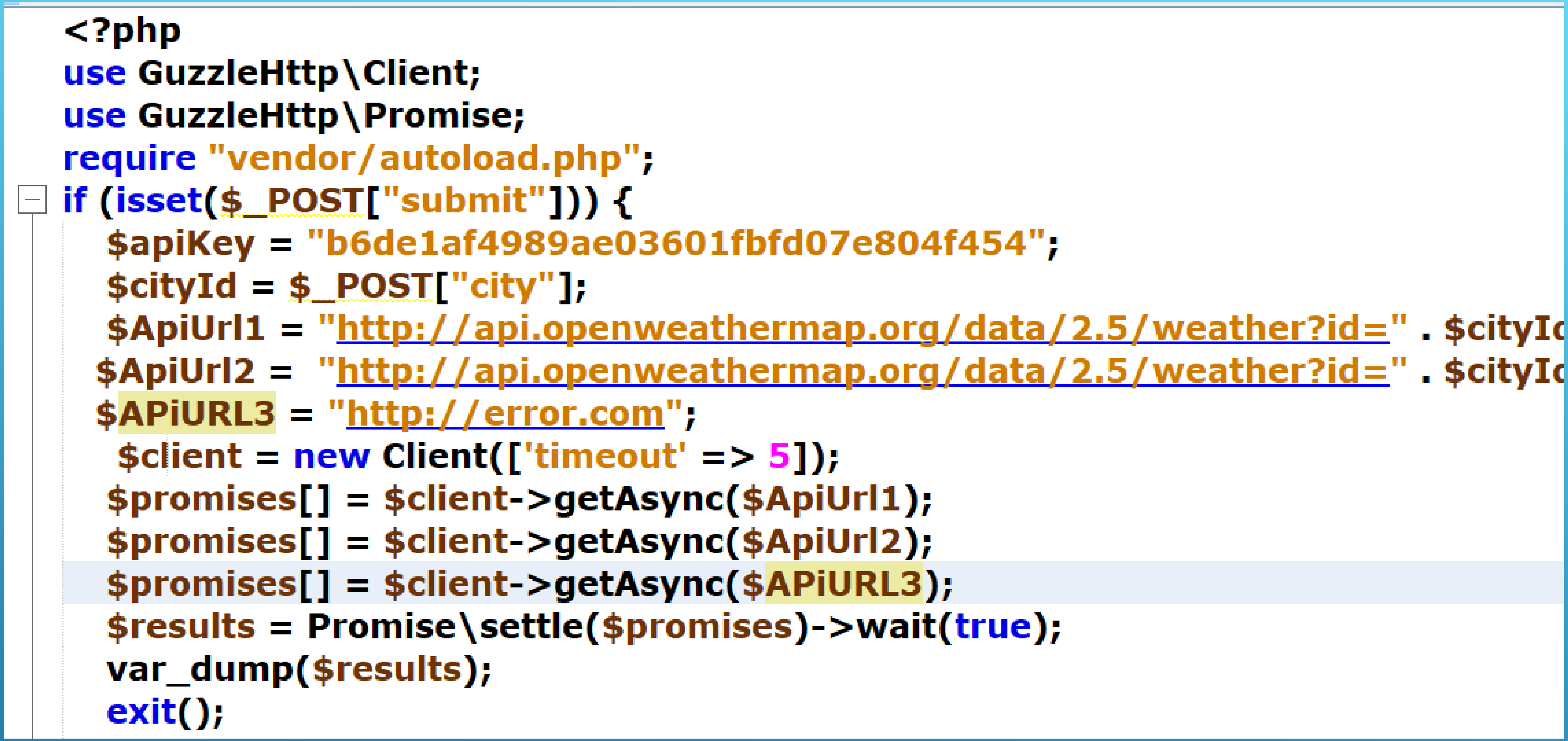


ASYNCH VS SYNCH



```
use Psr\Http\Message\ResponseInterface;
use GuzzleHttp\Exception\RequestException;
require "vendor/autoload.php";
if (isset($_POST["submit"])) {
    $apiKey = "b6de1af4989ae03601fbfd07e804f454";
    $cityId = $_POST["city"];
    $ApiUrl = "http://api.openweathermap.org/data/2.5/weather?id=" . $cityI
 $promise = $client->requestAsync('GET', $apiKey);
$promise->then(
    function (ResponseInterface $res) {
      $data = json_decode($response->getBody());
         $currentTime = time();
    },
    function (RequestException $e) {
       echo $e->getMessage() . "\n";
       echo $e->getRequest()->getMethod();
       exit();
```

CALL AN API ASYNCH



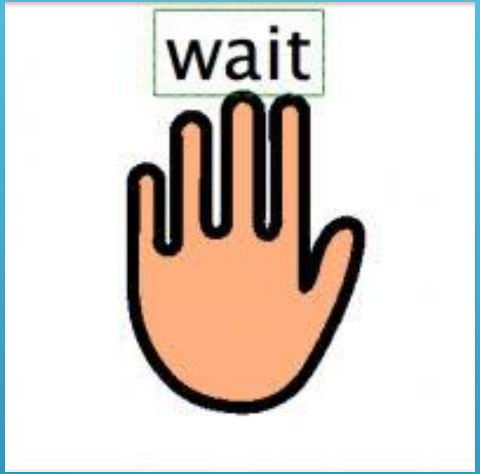
```
<?php
use GuzzleHttp\Client;
use GuzzleHttp\Promise;
require "vendor/autoload.php";
if (isset($_POST["submit"])) {
    $apiKey = "b6de1af4989ae03601fbfd07e804f454";
    $cityId = $_POST["city"];
    $ApiUrl1 = "http://api.openweathermap.org/data/2.5/weather?id=" . $cityIc
   $ApiUrl2 =  "http://api.openweathermap.org/data/2.5/weather?id=" . $cityIc
   $APiURL3 = "http://error.com";
     $client = new Client(['timeout' => 5]);
    $promises[] = $client->getAsync($ApiUrl1);
    $promises[] = $client->getAsync($ApiUrl2);
    $promises[] = $client->getAsync($APiURL3);
    $results = Promise\settle($promises)->wait(true);
    var_dump($results);
    exit();
```

CONCURRENT REQUESTS TO MULTIPLE APIS

USING GUZZLE YOU CAN DO ASYNC AND CONCURRENT REQUESTS FINISHED <<ADVANCED **FINISHED**>>

REST has become the default for most Web and mobile apps

PART 3 : REST APIS



CONCEPTS >> BIG PICTURE >> CODE

CODE IS SO EASY ☺ ☺ BUT UNDERSTAND THE BEST PRACTICES TO HELP CLIENTS (I.E MOBILE DEVELOPERS , FRONT END)

An HTTP request has the format shown in Figure 1:

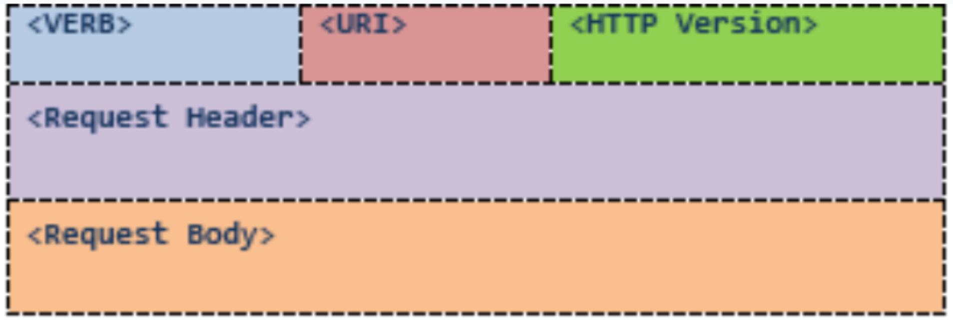


Figure 1: HTTP request format.

<VERB> is one of the HTTP methods like GET, PUT, POST, DELETE, OPTIONS, etc

<URI> is the URI of the resource on which the operation is going to be performed

<HTTP Version> is the version of HTTP, generally "HTTP v1.1" .

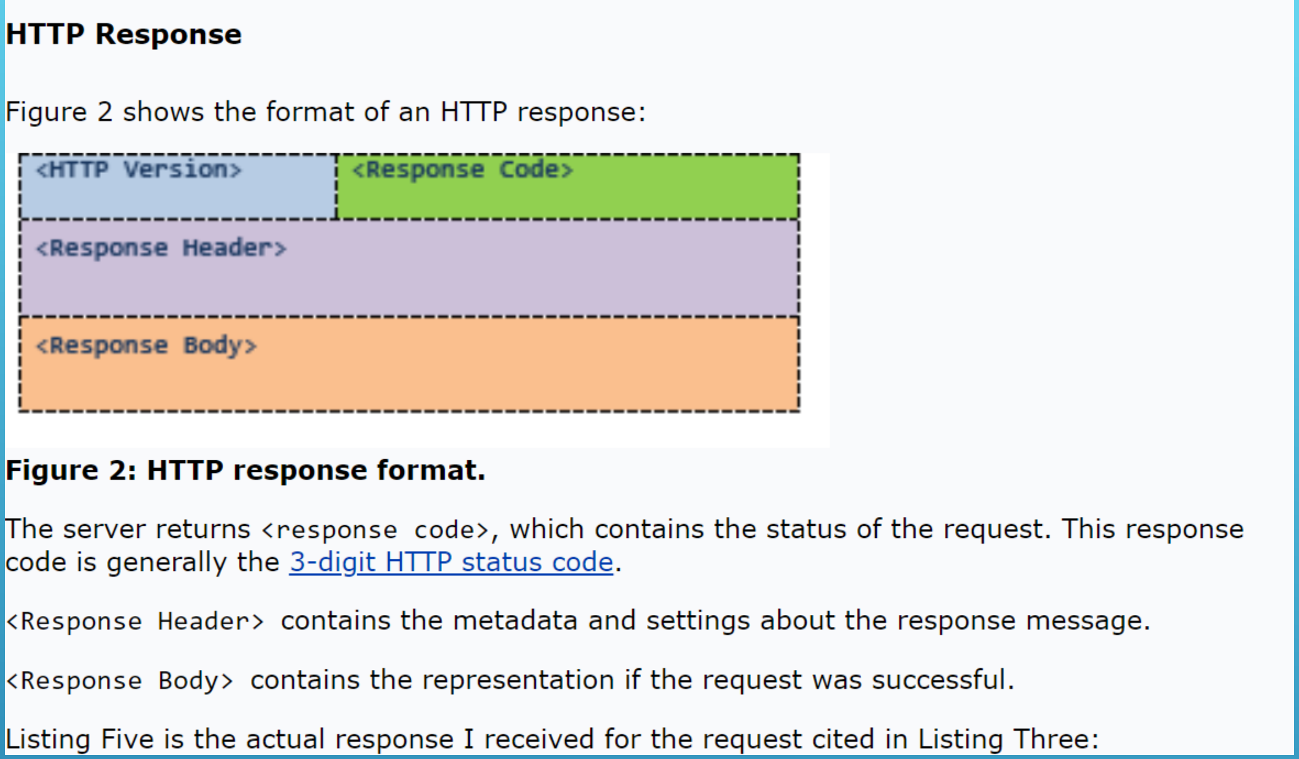
<Request Header> contains the metadata as a collection of key-value pairs of headers and their values. These settings contain information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and a lot more information.

<Request Body> is the actual message content. In a RESTful service, that's where the representations of resources sit in a message.

Listing Three: A sample POST request.

```
POST http://MyService/Person/
Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

HTTP REQUEST STRUCTURE



HTTP Response

Figure 2 shows the format of an HTTP response:

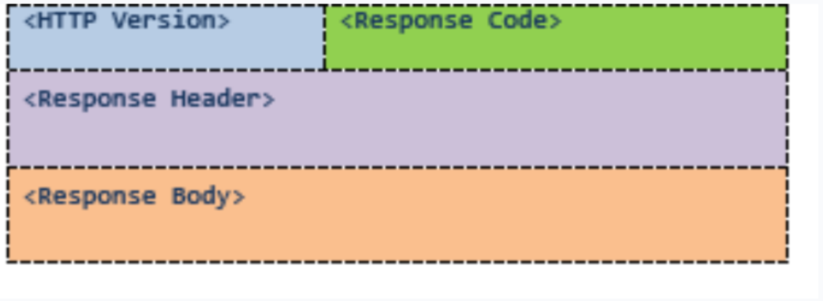


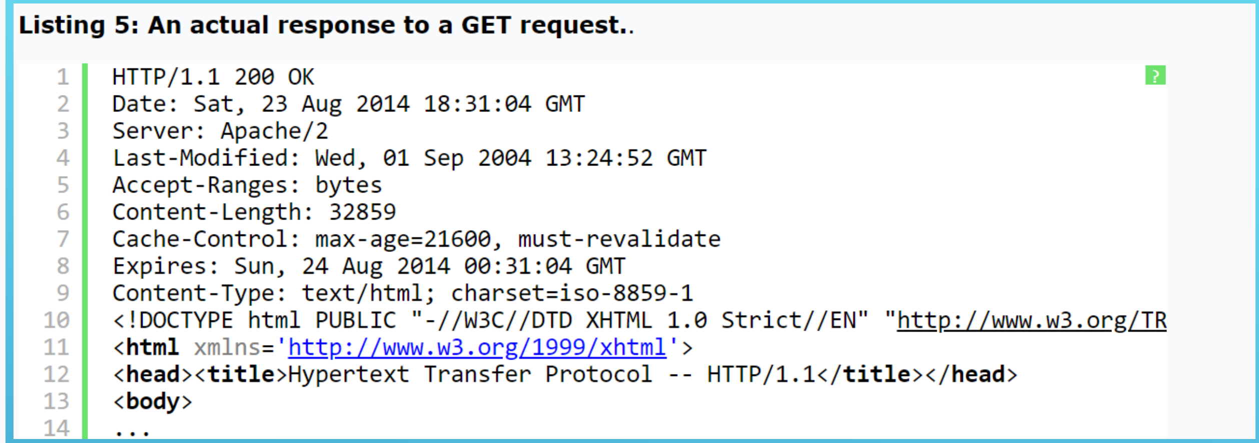
Figure 2: HTTP response format.

The server returns `<response code>`, which contains the status of the request. This response code is generally the 3-digit HTTP status code.

`<Response Header>` contains the metadata and settings about the response message.

`<Response Body>` contains the representation if the request was successful.

Listing Five is the actual response I received for the request cited in Listing Three:

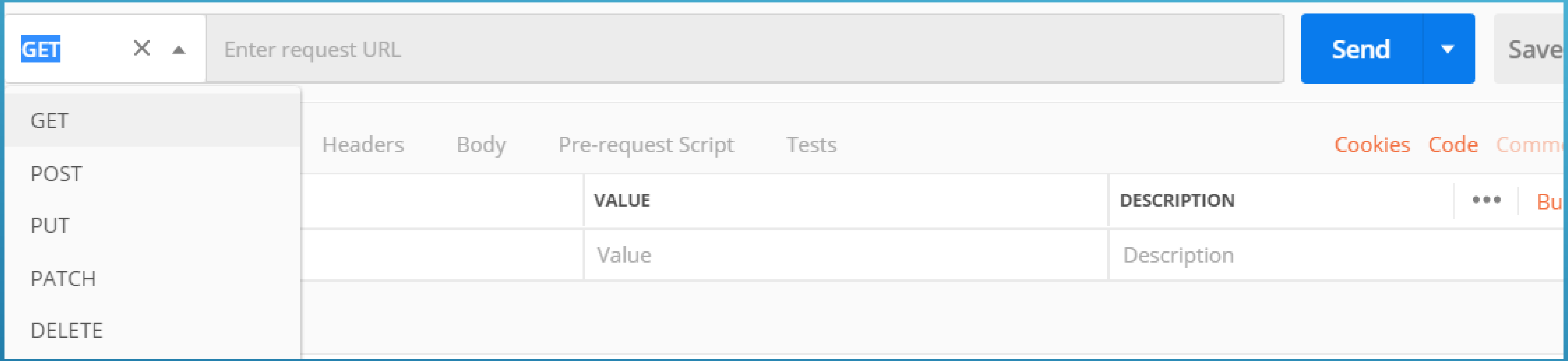


Listing 5: An actual response to a GET request..

```
HTTP/1.1 200 OK
Date: Sat, 23 Aug 2014 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
Cache-Control: max-age=21600, must-revalidate
Expires: Sun, 24 Aug 2014 00:31:04 GMT
Content-Type: text/html; charset=iso-8859-1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR
<html xmlns='http://www.w3.org/1999/xhtml'>
<head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head>
<body>
...
```

RESPONSE STRUCTURE

- **HTTP Verb**: the action counterpart to the noun-based resource. The primary or most-commonly-used **HTTP verbs** (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE.



I) HTTP VERBS

- GET : Just retrieve data from server, **no changes in the server**, so it could be repeated and bookmarked $_GET

GET /addresses.php?id=1
Retrieve the address which number is 1

- POST : causes changes in the server (most cases in create, so it shouldn't be repeated, it has a body $_POST

POST /addresses.php
Create new address with the data stored in the post body

- Are there other HTTP verbs ???

- PUT : The PUT method replaces all current representations of the target resource with the request payload..
- Access PUT using PHP
- **$_PUT= json_decode(file_get_contents("php://input"),true);**

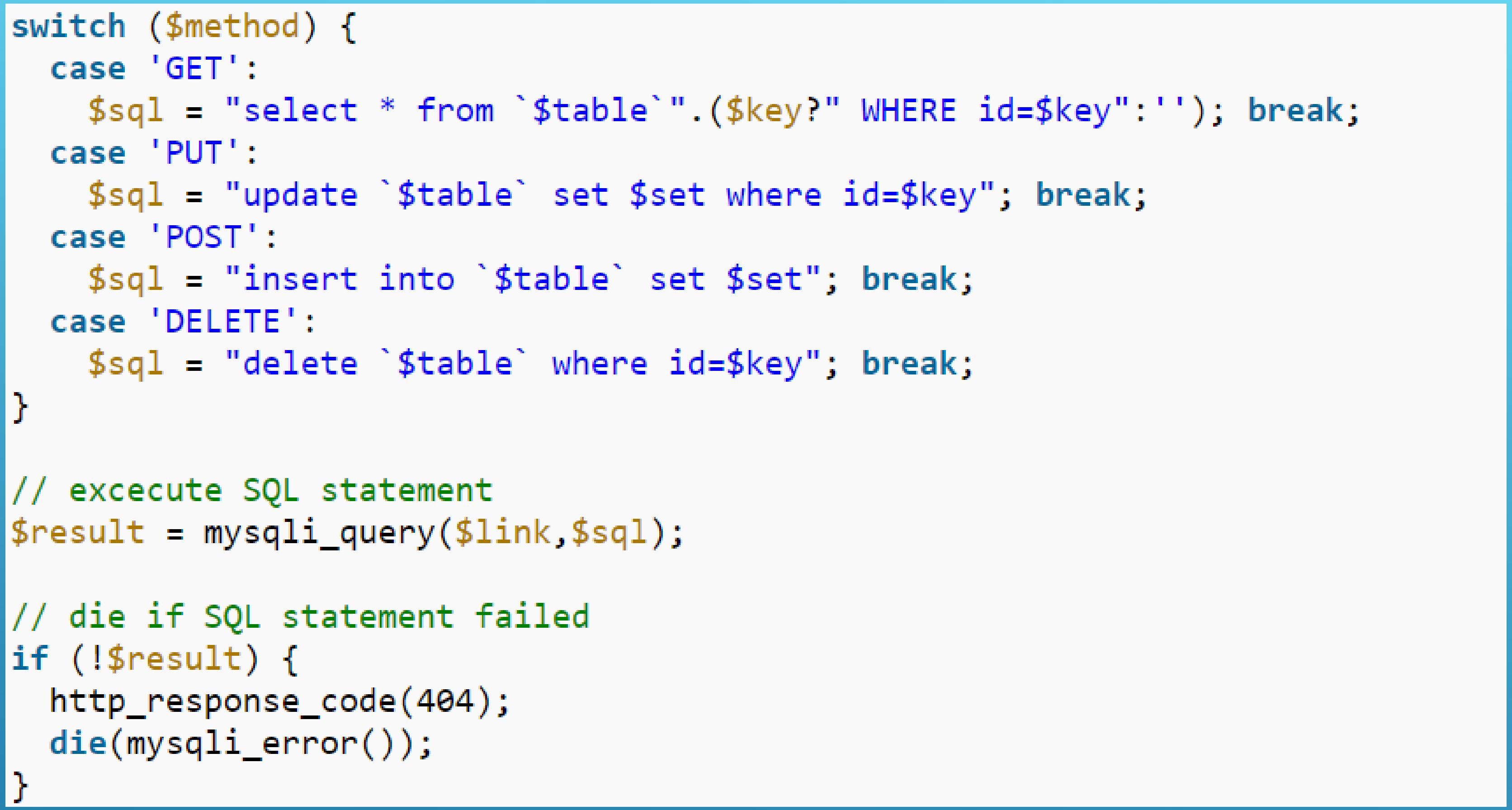
PUT /addresses.php?id=1
update the address which number is 1 with current payload

- $_SERVER["REQUEST_METHOD"] : gives which verb was used GET OR POST, Delete

- Delete : Request that a resource be removed; however, the resource does not have to be removed immediately. It could be an asynchronous or long-running request.

Delete /addresses.php?id=1
delete the address which number is 1

- $_SERVER[“REQUEST_METHOD”] : gives which verb was used GET OR POST, Delete



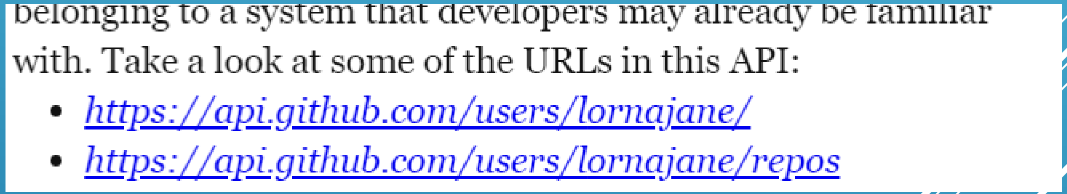
```
switch ($method) {
  case 'GET':
    $sql = "select * from `$table`".($key?" WHERE id=$key":''); break;
  case 'PUT':
    $sql = "update `$table` set $set where id=$key"; break;
  case 'POST':
    $sql = "insert into `$table` set $set"; break;
  case 'DELETE':
    $sql = "delete `$table` where id=$key"; break;
}

// excecute SQL statement
$result = mysqli_query($link,$sql);

// die if SQL statement failed
if (!$result) {
  http_response_code(404);
  die(mysqli_error());
}
```

HOW WE GET THE $VERB ?

HTTP STATUS CODES

- 2xx (Successful): The request was successfully received, understood, and accepted (200)
  - 201 Created successfully
  - 202 Deleted successfully
  - 204 Updated successfully
  - 200 request fulfilled
- 3xx (Redirection): Further action needs to be taken in order to complete the request (301)
- Setting a response code http_response_code(500)
- 4xx (Client Error): The request contains bad syntax or cannot be fulfilled (403 & 404)
  - 403 No access right Forbidden
  - 404 resource not found
  - 405 method not found
  - 402 payment required
  - 401 bad authentication
  - 400 bad request
  - 406 resource not acceptable
- 5xx (Server Error): The server failed to fulfill an apparently valid request (500)
  - 500 internal server error

- REPRESENTATIONAL STATE TRANSFER (transferring representation of resources)
- SET OF PRINCIPALES ON HOW DATA COULD BE TRANSFER ELEGANTLY VIA HTTP
- Each resource has at least one URL
- The focus of a RESTful service is on resources and how to provide access to these resources.
- A resource can easily be thought of as an object as in OOP. A resource can consist of other resources. While designing a system, the first thing to do is identify the resources and determine how they are related to each other. This is similar to the first step of designing a database: Identify entities and relations.
- A RESTful service uses a directory hierarchy like human readable URIs to address its resources
- http://MyService/Persons/1
- This URL has following format: Protocol://ServiceName/ResourceType/ResourceID
- REST SET OF PRETY URLs


belonging to a system that developers may already be familiar with. Take a look at some of the URLs in this API:

- https://api.github.com/users/lornajane/
- https://api.github.com/users/lornajane/repos

REST 101

- Avoid verbs for your resource unless the resource is a process such as search
  - DON'T USE : http://MyService/DeletePerson/1. For Delete a person
- RESTful systems should have a uniform interface. HTTP 1.1 provides a set of HTTP Verbs (GET, POST, DELETE and PUT)

Examples :

Delete	**http://MyService/Persons/1**
Get:	**http://MyService/Persons/1**
Update:	**http://MyService/Persons/1**
POST	**http://MyService/Persons/**

You should use these methods only for the purpose for which they are intended. For instance, never use GET to create or delete a resource on the server.

- Use plural nouns for naming your resources.
- Avoid using spaces

REST 101

http://MyService/Persons?id=1

OR http://MyService/Persons/1

The query parameter approach works just fine and REST does not stop you from using query parameters. However, this approach has a few disadvantages.

Increased complexity and reduced readability, which will increase if you have more parameters

Use http://localhost/rest01.php/items/10

Not http://localhost/rest01.php?Resources=items&&id=10

```
$url_piecies =  explode("/",$_SERVER["REQUEST_URI"]);
$resource =  (isset($url_piecies[2]))? $url_piecies[2] : "" ;
$resource_id =(isset($url_piecies[3]) && is_numeric($url_piecies[3]) ) ?$url_piecies[3] : 0;
```

Resource	**Methods**	**URI**	**Description**
Person	GET,POST,PUT, DELETE	http://MyService/Persons/{PersonID}	Contains information about a person, can create new person, update and delete persons. {PersonID} is optional **Format:** text/JSON
Club	GET,POST,PUT	http://MyService/Clubs/{ClubID}	Contains information about a club. can create new club & update existing clubs. {ClubID} is optional **Format:** text/JSON
Search	GET	http://MyService/Search?	Search a person or a club **Format:** text/xml **Query Parameters:** Name: String, Name of a person or a club Country: String, optional, Name of the country of a person or a club

EXAMPLE OF A REST SERVICE

- **Receiving the request & identifying it's parameters (VERB, RESOURCE, RESOURCE ID, PARAMETERS)**
- **Logging the request (why?)**
- **Validating the request**
- **If not valid request , send appropriate code and message**
- **If valid request , start dispatching (initialize data access and business logic objects which will handle the request)**
- **Prepare the response based on the verb (Handler for GET, POST, PUT and DELETE)**
- **If you have a valid response send it with appropriate response after logging it, why?**
- **If you don't have a valid response response is not of valid, send response code and error after logging it**
- **Try drawing a flow chart for it**

REST SERVICE STEPS

FINALLY CODE
CREARTING A RESTFUL API USING PHP

```
$method = $_SERVER['REQUEST_METHOD'];
```

Use http://localhost/rest01.php/items/10
Not http://localhost/rest01.php?Resources=items&&id=10

```
$url_piecies =  explode("/",$_SERVER["REQUEST_URI"]);
        $resource =  (isset($url_piecies[2]))? $url_piecies[2] : "" ;
        $resource_id =(isset($url_piecies[3]) && is_numeric($url_piecies[3]) ) ?$url_piecies[3] : 0;
header ("Content-Type: application/json");
$input = json_decode(file_get_contents('php://input'),true);
http_response_code(404)
```

REMEMBER BEFORE SEE THE WHOLE THING THAT