March 8, 2019

Build a Simple REST API in PHP

Krasimir Hristozov

REST APIs are the backbone of modern web development. Most web applications these days are developed as single-page applications on the frontend, connected to backend APIs written in various languages. There are many great frameworks that can help you build REST APIs quickly. Laravel/Lumen and Symfony's API platform are the most often used examples in the PHP ecosystem. They provide great tools to process requests and generate JSON responses with the correct HTTP status codes. They also make it easy to handle common issues like authentication/authorization, request validation, data transformation, pagination, filters, rate throttling, complex endpoints with sub-resources, and API documentation.

You certainly don't need a complex framework to build a simple but secure API though. In this article, I'll show you how to build a simple REST API in PHP from scratch. We'll make the API secure by using Okta as our authorization provider and implementing the Client Credentials Flow.

Okta is an API service that allows you to create, edit, and securely store user accounts and user account data, and connect them with one or more applications. Register for a forever-free developer account, and when you're done, come back to learn more about building a simple REST API in PHP.

There are different authentication flows in OAuth 2.0, depending on if the client application is public or private and if there is a user involved or the communication is machine-to-machine only. The Client Credentials Flow is best suited for machine-to-machine communication where the client application is private (and can be trusted to hold a secret). At the end of the post, I'll show you how to build a test client application as well.

Create the PHP Project Skeleton for Your REST API

We'll start by creating a `/src` directory and a simple `composer.json` file in the top directory with just one dependency (for now): the DotEnv library which will allow us to keep our Okta authentication details in a `.env` file outside our code repository:

`composer.json`

```
{
    "require": {
        "vlucas/phpdotenv": "^2.4"
    },
    "autoload": {
        "psr-4": {
            "Src\\": "src/"
        }
    }
}
```

We've also configured a PSR-4 autoloader which will automatically look for PHP classes in the `/src` directory.

We can install our dependencies now:

```
composer install
```

We now have a `/vendor` directory, and the DotEnv dependency is installed (we can also use our autoloader to load our classes from `/src` with no `include()` calls).

Let's create a `.gitignore` file for our project with two lines in it, so the `/vendor` directory and our local `.env` file will be ignored:

Next we'll create a `.env.example` file for our Okta authentication variables:

`.env.example`

```
OKTAAUDIENCE=api://default
OKTAISSUER=
SCOPE=
OKTACLIENTID=
OKTASECRET=
```

and a `.env` file where we'll fill in our actual details from our Okta account later (it will be ignored by Git so it won't end up in our repository).

We'll need a `bootstrap.php` file which loads our environment variables (later it will also do some additional bootstrapping for our project).

`bootstrap.php`

```
<?php
require 'vendor/autoload.php';
use Dotenv\Dotenv;

$dotenv = new DotEnv(__DIR__);
$dotenv->load();

// test code, should output:
// api://default
// when you run $ php bootstrap.php
echo getenv('OKTAAUDIENCE');
```

Configure a Database for Your PHP REST API

We will use MySQL to power our simple API. We'll create a new database and user for our app:

```
mysql -uroot -p
CREATE DATABASE api_example CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
CREATE USER 'api_user'@'localhost' identified by 'api_password';
GRANT ALL on api_example.* to 'api_user'@'localhost';
quit
```

Our rest API will deal with just a single entity: Person, with the following fields: `id`, `firstname`, `lastname`, `firstparent_id`, `secondparent_id`. It will allow us to define people and up to two parents for each person (linking to other records in our database). Let's create the database table in MySQL:

```
mysql -uapi_user -papi_password api_example
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT,
    firstname VARCHAR(100) NOT NULL,
    lastname VARCHAR(100) NOT NULL,
    firstparent_id INT DEFAULT NULL,
    secondparent_id INT DEFAULT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (firstparent_id)
        REFERENCES person(id)
        ON DELETE SET NULL,
    FOREIGN KEY (secondparent_id)
        REFERENCES person(id)
        ON DELETE SET NULL
) ENGINE=INNODB;
```

We'll add the database connection variables to our `.env.example` file:

`.env.example`

```
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=
DB_USERNAME=
DB_PASSWORD=
```

Then we'll input our local credentials in the .env file (which is not stored in the repo, remember?):

```
_
DB_PORT=3306
DB_DATABASE=api_example
DB_USERNAME=api_user
DB_PASSWORD=api_password
```

We can now create a class to hold our database connection and add the initialization of the connection to our bootstrap.php file:

src/System/DatabaseConnector.php

```
<?php
namespace Src\System;

class DatabaseConnector {

    private $dbConnection = null;

    public function __construct()
    {
        $host = getenv('DB_HOST');
        $port = getenv('DB_PORT');
        $db   = getenv('DB_DATABASE');
        $user = getenv('DB_USERNAME');
        $pass = getenv('DB_PASSWORD');

        try {
            $this->dbConnection = new \PDO(
                "mysql:host=$host;port=$port;charset=utf8mb4;dbname=$db",
                $user,
                $pass
            );
        } catch (\PDOException $e) {
            exit($e->getMessage());
        }
    }

    public function getConnection()
    {
        return $this->dbConnection;
    }
}
```

bootstrap.php (full version)

```
<?php
require 'vendor/autoload.php';
use Dotenv\Dotenv;

use Src\System\DatabaseConnector;

$dotenv = new DotEnv(__DIR__);
$dotenv->load();

$dbConnection = (new DatabaseConnector())->getConnection();
```

Let's create a `dbseed.php` file which creates our `Person` table and inserts some records in it for testing:

dbseed.php

```
$statement = <<<EOS
    CREATE TABLE IF NOT EXISTS person (
        id INT NOT NULL AUTO_INCREMENT,
        firstname VARCHAR(100) NOT NULL,
        lastname VARCHAR(100) NOT NULL,
        firstparent_id INT DEFAULT NULL,
        secondparent_id INT DEFAULT NULL,
        PRIMARY KEY (id),
        FOREIGN KEY (firstparent_id)
            REFERENCES person(id)
            ON DELETE SET NULL,
        FOREIGN KEY (secondparent_id)
            REFERENCES person(id)
            ON DELETE SET NULL
    ) ENGINE=INNODB;

    INSERT INTO person
        (id, firstname, lastname, firstparent_id, secondparent_id)
    VALUES
        (1, 'Krasimir', 'Hristozov', null, null),
        (2, 'Maria', 'Hristozova', null, null),
        (3, 'Masha', 'Hristozova', 1, 2),
        (4, 'Jane', 'Smith', null, null),
        (5, 'John', 'Smith', null, null),
        (6, 'Richard', 'Smith', 4, 5),
        (7, 'Donna', 'Smith', 4, 5),
        (8, 'Josh', 'Harrelson', null, null),
        (9, 'Anna', 'Harrelson', 7, 8);
EOS;

try {
    $createTable = $dbConnection->exec($statement);
    echo "Success!\n";
} catch (\PDOException $e) {
    exit($e->getMessage());
}
```

Our database is all set! If you want to reset it, just drop the `person` table in MySQL and then run `php dbseed.php` (I didn't add the drop statement to the seeder as a precaution against running it by mistake).

Add a Gateway Class for the Person Table

There are many patterns for working with databases in an object-oriented context, ranging from simple execution of direct SQL statements when needed (in a procedural way) to complex ORM systems (two of the most popular ORM choices in PHP are Eloquent and Doctrine). For our simple API, it makes sense to use a simple pattern as well so we'll go with a Table Gateway. We'll even skip creating a `Person` class (as the classical pattern would require) and just go with the `PersonGateway` class. We'll implement methods to return all records, return a specific person and add/update/delete a person.

`src/TableGateways/PersonGateway.php`

```
<?php
namespace Src\TableGateways;

class PersonGateway {

    private $db = null;

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

    public function findAll()
    {
        $statement = "
            SELECT
                id, firstname, lastname, firstparent_id, secondparent_id
            FROM
                person;
        ";

        try {
            $statement = $this->db->query($statement);
            $result = $statement->fetchAll(\PDO::FETCH_ASSOC);
            return $result;
        } catch (\PDOException $e) {
            exit($e->getMessage());
        }
    }

    public function find($id)
    {
```

```
            person
        WHERE id = ?;
    ";

    try {
        $statement = $this->db->prepare($statement);
        $statement->execute(array($id));
        $result = $statement->fetchAll(\PDO::FETCH_ASSOC);
        return $result;
    } catch (\PDOException $e) {
        exit($e->getMessage());
    }
}

public function insert(Array $input)
{
    $statement = "
        INSERT INTO person
            (firstname, lastname, firstparent_id, secondparent_id)
        VALUES
            (:firstname, :lastname, :firstparent_id, :secondparent_id);
    ";

    try {
        $statement = $this->db->prepare($statement);
        $statement->execute(array(
            'firstname' => $input['firstname'],
            'lastname'  => $input['lastname'],
            'firstparent_id' => $input['firstparent_id'] ?? null,
            'secondparent_id' => $input['secondparent_id'] ?? null,
        ));
        return $statement->rowCount();
    } catch (\PDOException $e) {
        exit($e->getMessage());
    }
}

public function update($id, Array $input)
{
    $statement = "
        UPDATE person
        SET
            firstname = :firstname,
            lastname  = :lastname,
            firstparent_id = :firstparent_id,
            secondparent_id = :secondparent_id
        WHERE id = :id;
    ";

    try {
        $statement = $this->db->prepare($statement);
        $statement->execute(array(
            'id' => (int) $id,
            'firstname' => $input['firstname'],
            'lastname'  => $input['lastname'],
            'firstparent_id' => $input['firstparent_id'] ?? null,
            'secondparent_id' => $input['secondparent_id'] ?? null,
        ));
        return $statement->rowCount();
    } catch (\PDOException $e) {
        exit($e->getMessage());
        }
    }

    public function delete($id)
    {
        $statement = "
            DELETE FROM person
            WHERE id = :id;
        ";

        try {
            $statement = $this->db->prepare($statement);
            $statement->execute(array('id' => $id));
            return $statement->rowCount();
        } catch (\PDOException $e) {
            exit($e->getMessage());
        }
    }
}
```

Obviously, in a production system, you would want to handle the exceptions more gracefully instead of just exiting with an error message.

Here are some examples of using the gateway:

```php
// return all records
$result = $personGateway->findAll();

// return the record with id = 1
$result = $personGateway->find(1);

// insert a new record
$result = $personGateway->insert([
    'firstname' => 'Doug',
    'lastname' => 'Ellis'
]);

// update the record with id = 10
$result = $personGateway->update(10, [
    'firstname' => 'Doug',
    'lastname' => 'Ellis',
    'secondparent_id' => 1
]);

// delete the record with id = 10
$result = $personGateway->delete(10);
```

Implement the PHP REST API

We will implement a REST API now with the following endpoints:

```
// return all records
GET /person

// return a specific record
GET /person/{id}

// create a new record
POST /person

// update an existing record
PUT /person/{id}

// delete an existing record
DELETE /person/{id}
```

We'll create a `/public/index.php` file to serve as our front controller and process the requests, and a `src/Controller/PersonController.php` to handle the API endpoints (called from the front controller after validating the URI).

`public/index.php`

```php
<?php
require "../bootstrap.php";
use Src\Controller\PersonController;

header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: OPTIONS,GET,POST,PUT,DELETE");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With");

$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$uri = explode( '/', $uri );

// all of our endpoints start with /person
// everything else results in a 404 Not Found
if ($uri[1] !== 'person') {
    header("HTTP/1.1 404 Not Found");
    exit();
}

// the user id is, of course, optional and must be a number:
$userId = null;
if (isset($uri[2])) {
    $userId = (int) $uri[2];
}

$requestMethod = $_SERVER["REQUEST_METHOD"];

// pass the request method and user ID to the PersonController and process the HTTP request:
$controller = new PersonController($dbConnection, $requestMethod, $userId);
$controller->processRequest();
```

`src/Controller/PersonController.php`

```
use Src\TableGateways\PersonGateway;

class PersonController {

    private $db;
    private $requestMethod;
    private $userId;

    private $personGateway;

    public function __construct($db, $requestMethod, $userId)
    {
        $this->db = $db;
        $this->requestMethod = $requestMethod;
        $this->userId = $userId;

        $this->personGateway = new PersonGateway($db);
    }

    public function processRequest()
    {
        switch ($this->requestMethod) {
            case 'GET':
                if ($this->userId) {
                    $response = $this->getUser($this->userId);
                } else {
                    $response = $this->getAllUsers();
                };
                break;
            case 'POST':
                $response = $this->createUserFromRequest();
                break;
            case 'PUT':
                $response = $this->updateUserFromRequest($this->userId);
                break;
            case 'DELETE':
                $response = $this->deleteUser($this->userId);
                break;
            default:
                $response = $this->notFoundResponse();
                break;
        }
        header($response['status_code_header']);
        if ($response['body']) {
            echo $response['body'];
        }
    }

    private function getAllUsers()
    {
        $result = $this->personGateway->findAll();
        $response['status_code_header'] = 'HTTP/1.1 200 OK';
        $response['body'] = json_encode($result);
        return $response;
    }

    private function getUser($id)
    {
        $result = $this->personGateway->find($id);
        if (! $result) {
            return $this->notFoundResponse();
        }
        $response['status_code_header'] = 'HTTP/1.1 200 OK';
        $response['body'] = json_encode($result);
        return $response;
    }

    private function createUserFromRequest()
    {
        $input = (array) json_decode(file_get_contents('php://input'), TRUE);
        if (! $this->validatePerson($input)) {
            return $this->unprocessableEntityResponse();
        }
        $this->personGateway->insert($input);
        $response['status_code_header'] = 'HTTP/1.1 201 Created';
        $response['body'] = null;
        return $response;
    }

    private function updateUserFromRequest($id)
    {
        $result = $this->personGateway->find($id);
        if (! $result) {
            return $this->notFoundResponse();
        }
        $input = (array) json_decode(file_get_contents('php://input'), TRUE);
        if (! $this->validatePerson($input)) {
            return $this->unprocessableEntityResponse();
        }
        $this->personGateway->update($id, $input);
        $response['status_code_header'] = 'HTTP/1.1 200 OK';
        $response['body'] = null;
```

```
        $result = $this->personGateway->find($id);
        if (! $result) {
            return $this->notFoundResponse();
        }
        $this->personGateway->delete($id);
        $response['status_code_header'] = 'HTTP/1.1 200 OK';
        $response['body'] = null;
        return $response;
    }

    private function validatePerson($input)
    {
        if (! isset($input['firstname'])) {
            return false;
        }
        if (! isset($input['lastname'])) {
            return false;
        }
        return true;
    }

    private function unprocessableEntityResponse()
    {
        $response['status_code_header'] = 'HTTP/1.1 422 Unprocessable Entity';
        $response['body'] = json_encode([
            'error' => 'Invalid input'
        ]);
        return $response;
    }

    private function notFoundResponse()
    {
        $response['status_code_header'] = 'HTTP/1.1 404 Not Found';
        $response['body'] = null;
        return $response;
    }
}
```

You can test the API with a tool like Postman. First, go to the project directory and start the PHP server:

```
php -S 127.0.0.1:8000 -t public
```

Then connect to `127.0.0.1:8000` with Postman and send http requests. Note: when making PUT and POST requests, make sure to set the Body type to `raw`, then paste the payload in JSON format and set the content type to JSON (application/json).

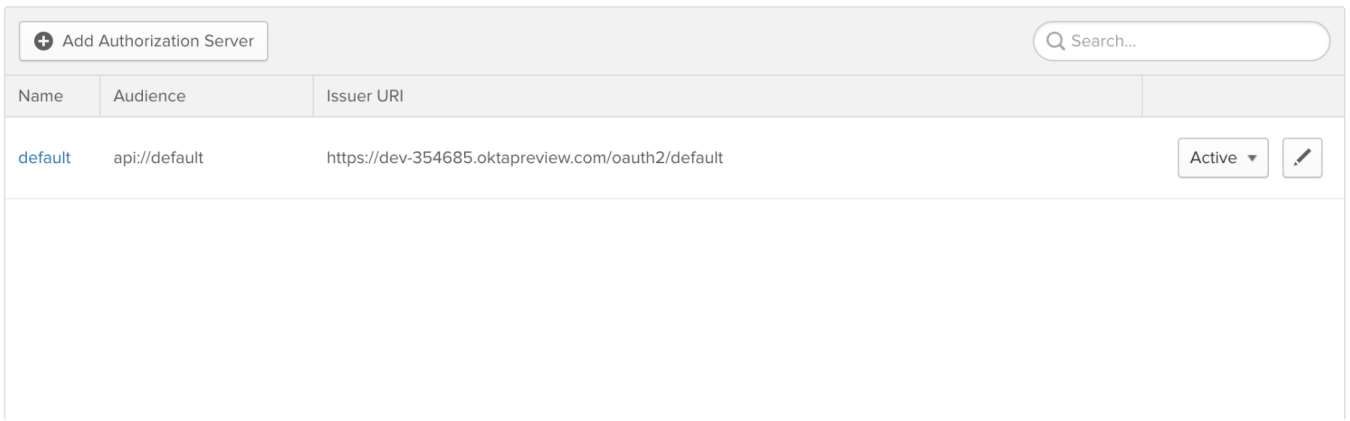
Secure Your PHP REST API with OAuth 2.0

We'll use Okta as our authorization server and we'll implement the Client Credentials Flow. The flow is recommended for machine-to-machine authentication when the client is private and works like this: The client application holds a Client ID and a Secret; The client passes these credentials to Okta and obtains an access token; The client sends the access token to the REST API server; The server asks Okta for some metadata that allows it to verify tokens and validates the token (alternatively, it can just ask Okta to verify the token); The server then provides the API resource if the token is valid, or responds with a 401 Unauthorized status code if the token is missing, expired or invalid.

Before you proceed, you need to log into your Okta account (or create a new one for free), create your authorization server and set up your client application.

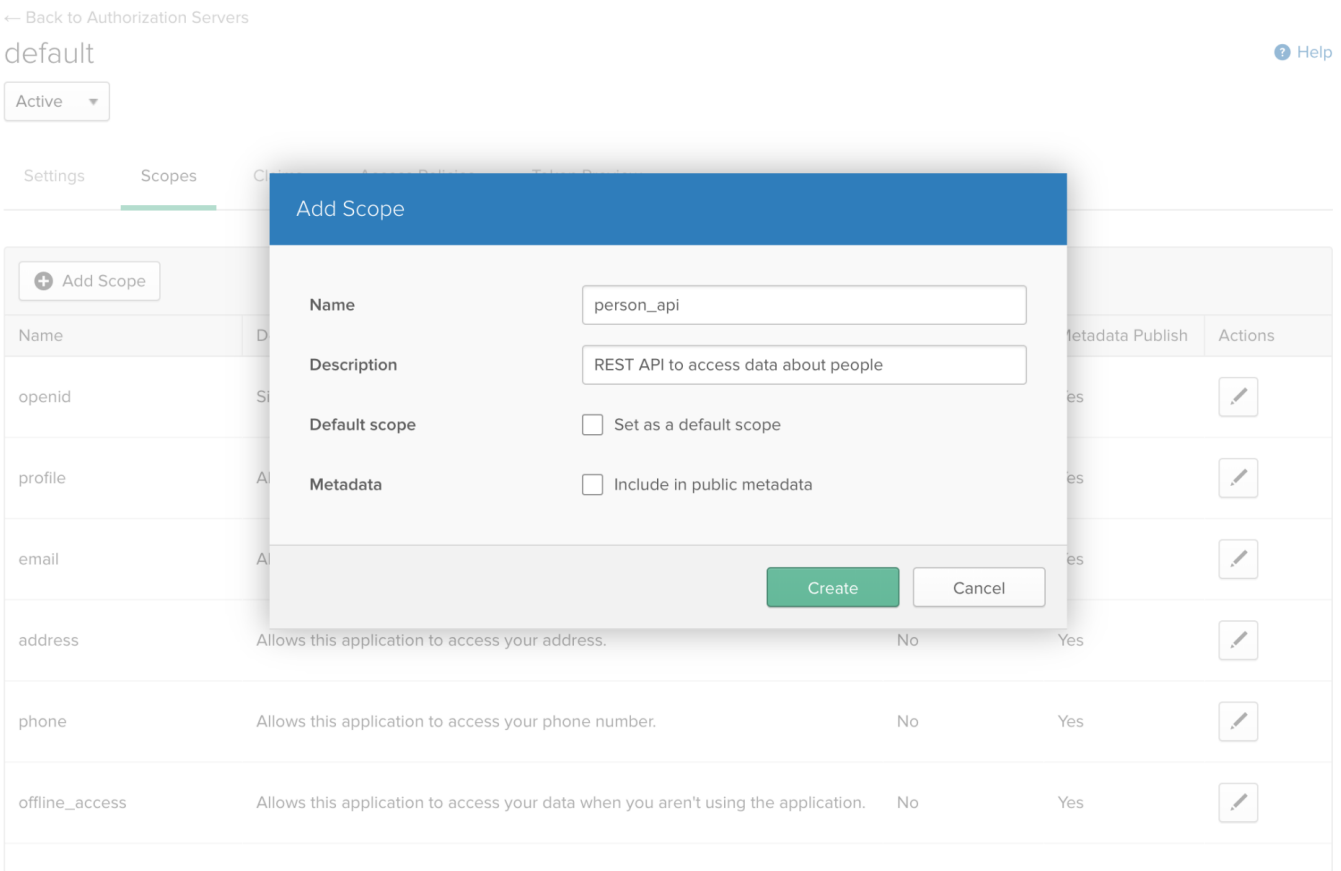
Log in to your developer console, navigate to API, then to the Authorization Servers tab. Click on the link to your default server. We will copy the Issuer Uri field from this Settings tab and add it to our .env file:

```
OKTAISSUER=https://{yourOktaDomain}/oauth2/default
```



You can see the Issuer URI of my test Okta account in the screenshot above. Copy your own value and put it in your `.env` file.

Next click the **Edit** icon, go to the **Scopes** tab and click **Add Scope** to add a scope for the REST API. We'll title it `person_api`:



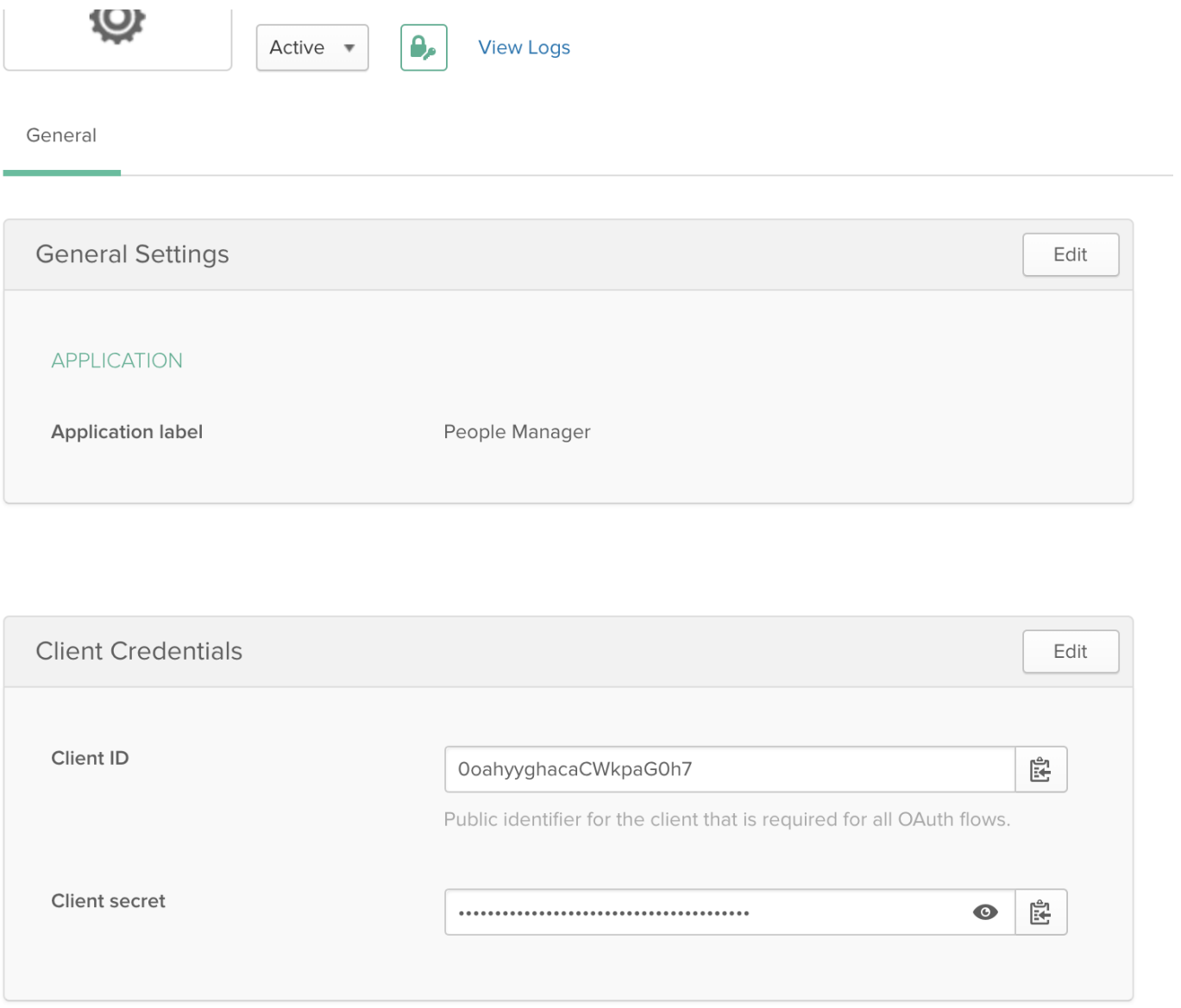
We need to add the scope to our `.env` file as well. We'll add the following to `.env.example`:

```
SCOPE=
```

and the key with the value to .env:

```
SCOPE=person_api
```

The next step is to create a client. Navigate to **Applications**, then click **Add Application**. Select **Service**, then click **Next**. Enter a name for your service, (e.g. People Manager), then click **Done**. This will take you to a page that has your client credentials:



These are the credentials that your client application will need in order to authenticate. For this example, the client and server code will be in the same repository, so we will add these credentials to our `.env` file as well (make sure to replace `{yourClientId}` and `{yourClientSecret}` with the values from this page):

Add to .env.example:

```
OKTACLIENTID=
OKTASECRET=
```

Add to .env:

```
OKTACLIENTID={yourClientId}
OKTASECRET={yourClientSecret}
```

Add Authentication to Your PHP REST API

We'll use the Okta JWT Verifier library. It requires a JWT library (we'll use `spomky-labs/jose`) and a PSR-7 compliant library (we'll use `guzzlehttp/psr7`). We'll install everything through composer:

```
composer require okta/jwt-verifier spomky-labs/jose guzzlehttp/psr7
```

Now we can add the authorization code to our front controller (if using a framework, we'll do this in a middleware instead):

`public/index.php` (full version for clarity)

```php
header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");
header("Access-Control-Allow-Methods: OPTIONS,GET,POST,PUT,DELETE");
header("Access-Control-Max-Age: 3600");
header("Access-Control-Allow-Headers: Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With");

$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$uri = explode('/', $uri);

// all of our endpoints start with /person
// everything else results in a 404 Not Found
if ($uri[1] !== 'person') {
    header("HTTP/1.1 404 Not Found");
    exit();
}

// the user id is, of course, optional and must be a number:
$userId = null;
if (isset($uri[2])) {
    $userId = (int) $uri[2];
}

// authenticate the request with Okta:
if (! authenticate()) {
    header("HTTP/1.1 401 Unauthorized");
    exit('Unauthorized');
}

$requestMethod = $_SERVER["REQUEST_METHOD"];

// pass the request method and user ID to the PersonController:
$controller = new PersonController($dbConnection, $requestMethod, $userId);
$controller->processRequest();

function authenticate() {
    try {
        switch(true) {
            case array_key_exists('HTTP_AUTHORIZATION', $_SERVER) :
                $authHeader = $_SERVER['HTTP_AUTHORIZATION'];
                break;
            case array_key_exists('Authorization', $_SERVER) :
                $authHeader = $_SERVER['Authorization'];
                break;
            default :
                $authHeader = null;
                break;
        }
        preg_match('/Bearer\s(\S+)/', $authHeader, $matches);
        if(!isset($matches[1])) {
            throw new \Exception('No Bearer Token');
        }
        $jwtVerifier = (new \Okta\JwtVerifier\JwtVerifierBuilder())
            ->setIssuer(getenv('OKTAISSUER'))
            ->setAudience('api://default')
            ->setClientId(getenv('OKTACLIENTID'))
            ->build();
        return $jwtVerifier->verify($matches[1]);
    } catch (\Exception $e) {
        return false;
    }
}
```

Build a Sample Client Application (Command Line Script) to Test the PHP REST API

In this section, we will add a simple client application (a command line script using curl) to test the REST API. We'll create a new php file 'public/clients.php' with a very simple flow: it will retrieve the Okta details (issuer, scope, client id and secret) from the .env file, then it will obtain an access token from Okta and it will run API calls to get all users and get a specific user (passing the Okta access token in the Authorization header).

`public/client.php`

```
$clientId     = getenv('OKTACLIENTID');
$clientSecret = getenv('OKTASECRET');
$scope        = getenv('SCOPE');
$issuer       = getenv('OKTAISSUER');

// obtain an access token
$token = obtainToken($issuer, $clientId, $clientSecret, $scope);

// test requests
getAllUsers($token);
getUser($token, 1);

// end of client.php flow

function obtainToken($issuer, $clientId, $clientSecret, $scope) {
    echo "Obtaining token...";

    // prepare the request
    $uri = $issuer . '/v1/token';
    $token = base64_encode("$clientId:$clientSecret");
    $payload = http_build_query([
        'grant_type' => 'client_credentials',
        'scope'      => $scope
    ]);

    // build the curl request
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $uri);
    curl_setopt( $ch, CURLOPT_HTTPHEADER, [
        'Content-Type: application/x-www-form-urlencoded',
        "Authorization: Basic $token"
    ]);
    curl_setopt($ch, CURLOPT_POST, 1);
    curl_setopt($ch, CURLOPT_POSTFIELDS, $payload);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

    // process and return the response
    $response = curl_exec($ch);
    $response = json_decode($response, true);
    if (! isset($response['access_token'])
        || ! isset($response['token_type'])) {
        exit('failed, exiting.');
    }

    echo "success!\n";
    // here's your token to use in API requests
    return $response['token_type'] . " " . $response['access_token'];
}

function getAllUsers($token) {
    echo "Getting all users...";
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, "http://127.0.0.1:8000/person");
    curl_setopt( $ch, CURLOPT_HTTPHEADER, [
        'Content-Type: application/json',
        "Authorization: $token"
    ]);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec($ch);

    var_dump($response);
}

function getUser($token, $id) {
    echo "Getting user with id#$id...";
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, "http://127.0.0.1:8000/person/" . $id);
    curl_setopt( $ch, CURLOPT_HTTPHEADER, [
        'Content-Type: application/json',
        "Authorization: $token"
    ]);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec($ch);

    var_dump($response);
}
```

You can run the application from the command line by going to the `/public` directory and running:

```
php client.php
```

(Don't forget to start the server if you haven't already!)

```
php -S 127.0.0.1:8000 -t public
```

Learn More About PHP, Secure REST APIs, and OAuth 2.0 Client Credentials Flow

You can find the whole code example here: GitHub link

If you would like to dig deeper into the topics covered in this article, the following resources are a great starting point:

- Our Vue/PHP Quickstart Guide
- Okta Authentication Overview
- Add Authentication to your PHP App in 5 Minutes
- Build Simple Login in PHP

Like what you learned today? Follow us on Twitter, and subscribe to our YouTube channel for more awesome content!

FORUM

RSS BLOG

YOUTUBE

More Info

INTEGRATE WITH OKTA
BLOG
CHANGE LOG
3RD PARTY NOTICES
COMMUNITY TOOLKIT

Contact & Legal

CONTACT OUR TEAM
CONTACT SALES
CONTACT SUPPORT
TERMS & CONDITIONS
PRIVACY POLICY