ECPHP - API Gateway Authentication Bundle documentation

Release 1.0.0

Jul 05, 2021

CONTENTS

This bundle provides the necessary to authenticate a request based on a specific HTTP header.

It has been written to facilitate the authentication of requests from API Gateway.

This bundle relies on lexik/jwt-authentication-bundle and provide a specific KeyLoader.

The features it provides are:

- Provides default configuration to work with API Gateway,
- Has a failsafe mechanism for public key retrieval and embed the public keys of the default API Gateway in case of failure,
- Provides a default `UserProvider` service and `User` entity,

CHAPTER

ONE

API GATEWAY

The European Commission API Gateway service allows you to deploy microservices as APIs behind the Gateway.

The Gateway offers an added layer of security and multiple useful utilities such as:

- API protection with tokens
- API lifecycle management
- API versioning
- API traffic management & throttling
- API analytics
- API management automation
- No hassle API publication with swagger
- Store of APIs to reuse

This service is based on the open source project WSO2 API Gateway, in a distributed deployment with custom components.

1.1 Requirements

1.1.1 PHP

PHP greater or equal to 7.4.

1.1.2 Symfony

The minimal required version of Symfony is 5.

1.1.3 Extensions

These PHP extensions are required:

- openssl

1.2 Installation

This package has a Symfony Flex recipe that will install configuration files for you.

Default configuration files will be copied in the *dev* environment.

1.2.1 Step 1

The recommended way to install it is with Composer.

This package requires a PSR HTTP client and a PSR Message implementation.

We recommend using *symfony/http-client* and *nyholm/psr7*, but feel free to use the one you prefer.

```
composer require symfony/http-client
composer require nyholm/psr7
```

```
composer require ecphp/api-gw-authentication-bundle
```

This package has a Symfony recipe that will provides the minimum configuration files.

Warning: Be carefull, the recipe will create enable some routes in your `dev` environment only. Those routes might be considered as a security issue if they are enabled in the `production` environment. Those routes are `/api/token` and `/api/user`. Find the documentation related to those routes inside the classes themselves. To disable them completely, just delete the file `packages/config/routes/dev/api_gw_authentication.yaml` from your application.

1.2.2 Step 2

Edit the bundle configuration by editing the file `config/packages/dev/api_gw_authentication.yaml`.

```
api_gw_authentication:
    defaults:
        env: acceptance # Available values are: acceptance, intra, production, user
```

Optionally, to use your own public and private key, then you do not need this package. Simply enable the bundle lexik/jwt-authentication-bundle and follow their documentation.

However, if you still want this package and your own keys, edit the configuration as such

```
api_gw_authentication:
    defaults:
        env: user # Available values are: acceptance, intra, production, user
    envs:
        user:
```

(continues on next page)

(continued from previous page)

```
            public: <path-to-the-public-key>
            private: <path-to-the-private-key>
```

The environment `user` is the only custom environment that you can create. It has a very limited use. It was mostly created for the unit tests.

1.2.3 Step 3

This is the crucial part of your application's security configuration.

Edit the security settings of your application by edition the file *config/packages/security.yaml*.

```
security:
    firewalls:
        default:
            anonymous: ~
            stateless: true
            guard:
                provider: api_gw_authentication # This is provided by default by the␣
↪bundle.
                authenticators:
                    - lexik_jwt_authentication.jwt_token_authenticator
    access_control:
        - { path: ^/api/token, role: IS_ANONYMOUS } # Optional - See step 2, enable this␣
↪ONLY for dev environment
        - { path: ^/api, role: IS_AUTHENTICATED_FULLY }
```

This configuration example will trigger the authentication on paths starting with */api*, therefore make sure that at least such paths exists.

Feel free to change these configuration to fits your need. Have a look at the Symfony documentation about security and Guard authentication.

1.2.4 Step 4

Optionally, you can override the default HTTP client.

Edit your own *services.yaml* file as such:

```
services
    cachedHttpClient:
        class: 'Symfony\Component\HttpClient\CachingHttpClient'
        arguments:
            $store: '@http_cache.store'

    api_gw_authentication.http_client:
        class: 'Symfony\Component\HttpClient\Psr18Client'
        arguments:
            $client: '@cachedHttpClient'
```

1.3 Configuration

Hereunder an example of configuration for this bundle.

```yaml
api_gw_authentication:
    defaults:
        env: acceptance # Available values are: acceptance, intra, production, user
```

You may customize a specific configuration by doing:

```yaml
api_gw_authentication:
    defaults:
        env: user # Available values are: acceptance, intra, production, user
    envs:
        user:
            public: <path-to-public-key-in-pem>
            private: <path-to-private-key-in-pem>
```

However, it is impossible to override existing API Gateway environments.

1.4 Usage

1.4.1 Step 1

Follow the *Installation* procedure.

1.4.2 Step 2

Configure the configuration files accordingly and the security of your Symfony application.

1.4.3 Step 3

Get a valid token from API Gateway.

1.4.4 Step 4

- Make a request to `/api/user` with the `Authorization` header.

```
curl -X GET "http://127.0.0.1:8000/api/user" -H "Authorization:  Bearer
<insert-token-here>"
```

At this point, the `KeyLoader` will try to retrieve the public key from the API Gatewey environment in use.

If it fails, it will use a local copy of the key inside the bundle.

The `HttpClient` in use in this bundle is a `CachingHttpClient`, which means that the request to API Gateway is cached by default. So when you request the keys multiple times, only one http call will be made.

There is no lifespan configuration for a `CachingHttpClient`, it is forever cached until you clear the Symfony cache yourself.

1.5 Tests, code quality and code style

Every time changes are introduced into the library, Travis CI and Github Actions run the tests written with PHPSpec.

PHPInfection is also triggered used to ensure that your code is properly tested.

The code style is based on PSR-12 plus a set of custom rules. Find more about the code style in use in the package drupol/php-conventions.

A PHP quality tool, Grumphp, is used to orchestrate all these tasks at each commit on the local machine, but also on the continuous integration tools (Travis, Github actions)

To run the whole tests tasks locally, do

```
composer grumphp
```

or

```
./vendor/bin/grumphp run
```

Here's an example of output that shows all the tasks that are setup in Grumphp and that will check your code

```
./vendor/bin/grumphp run
GrumPHP is sniffing your code!
Running task  1/13: SecurityChecker... ✓
Running task  2/13: Composer... ✓
Running task  3/13: ComposerNormalize... ✓
Running task  4/13: YamlLint... ✓
Running task  5/13: JsonLint... ✓
Running task  6/13: PhpLint... ✓
Running task  7/13: TwigCs... ✓
Running task  8/13: PhpCsAutoFixerV2... ✓
Running task  9/13: PhpCsFixerV2... ✓
Running task 10/13: Phpcs... ✓
Running task 11/13: PhpStan... ✓
Running task 12/13: Phpspec... ✓
Running task 13/13: Infection... ✓
```

1.6 Contributing

See the file CONTRIBUTING.md but feel free to contribute to this project by sending Github pull requests.

1.7 Development

1.7.1 Maintainers

See the MAINTAINERS.txt file.

1.7.2 Documentation

Documentation can be built locally using Sphinx.

To render the documentation locally do the following steps:

- `docker-compose up`
- Navigate to http://127.0.0.1:8100/

1.7.3 Contributors

See the Github insights page.