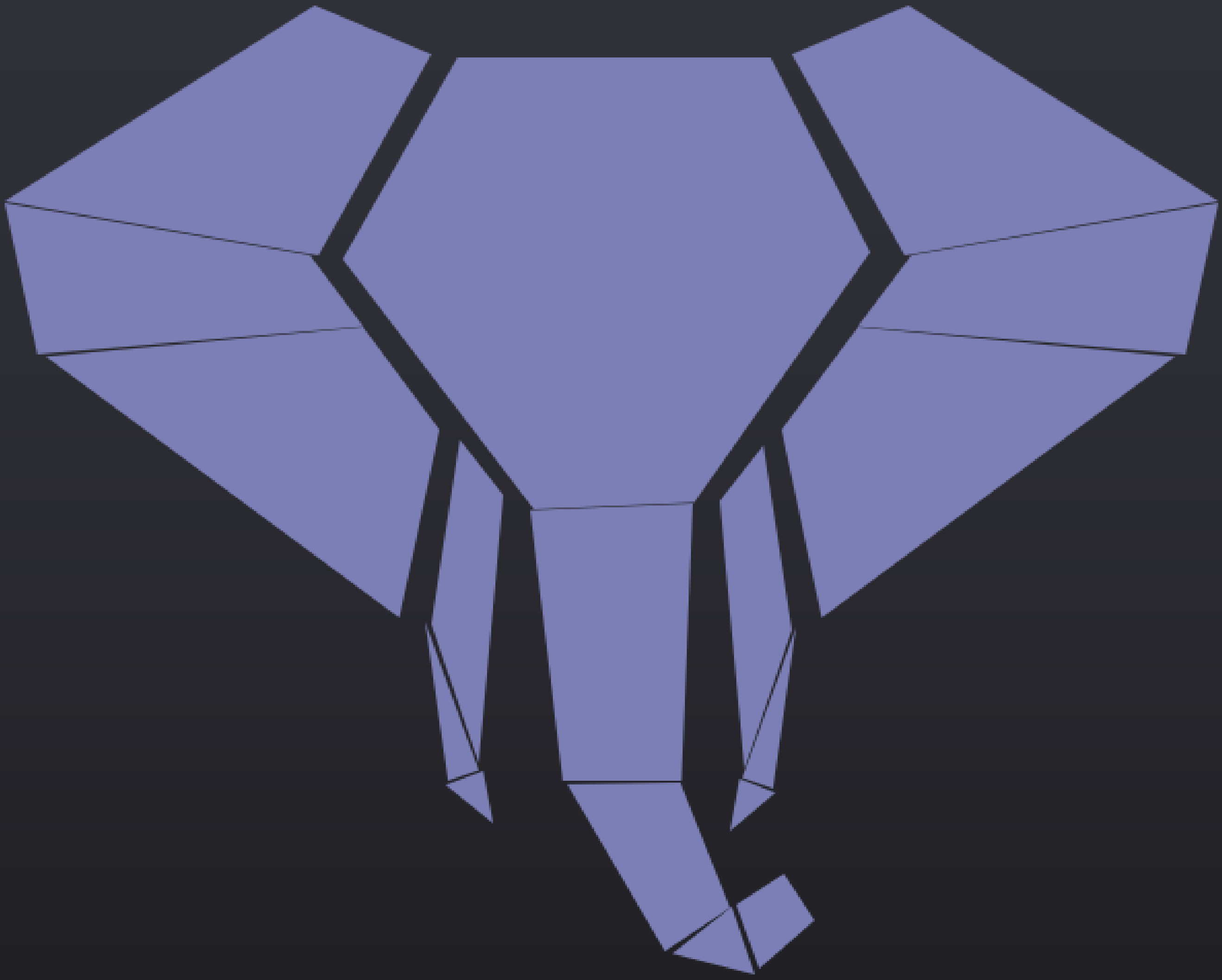
MEDIOR PHP

The Next Step



JOSEPH KANYO

Chapter 12.3.

Test a Repository

A `Repository` has methods that talk to the database, so in this case, we have some choices: if you are using a **Cloud Provider**, you can pay for a test database with much smaller power capabilities and use that in your `PDO` configuration.

You can set up **another database locally**, or you can **use Docker** to create a container with postgres in it and run your integration tests against that.

I will do the second one this time, if you want to follow along, you should create another database **inside DBngin**: **Posgres, version 16.2**, named **ani-merged-test**.

Of course, we need to **run our migrations against that database**, or if your project is really small you can just write down the SQL statements from your migration files and execute it manually against your test db. I will do this because this is a demo, but this is not good practice in the real world.

After this, we will have a setup we can run our tests against. Never run your integration tests against your development database.

We want to have our `PDO` configured before each test: we can do this with the `setUp()` method of the `TestCase` class. That code will run before each of the tests. And we can use the `tearDown()` to run some code after each of the tests. In this method, we want to execute a `DELETE FROM` statement to get a clean slate every time.

File: ./tests/Repository/AnimationPostgreSQLRepositoryTest.php

```php
class AnimationPostgreSQLRepositoryTest extends TestCase
{
    private PDO $pdo;
    private AnimationRepositoryInterface $animations;

    protected function setUp(): void
    {
        $dsn = sprintf(
        "pgsql:host=%s;port=%s;dbname=%s;user=%s;password=%s;sslmode=%s",
            "localhost",
            "5432",
            "ani-merged-test",
            "postgres",
            "postgres",
            "disable",
        );

        $options = [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
            PDO::ATTR_EMULATE_PREPARES => false,
        ];

        $this->pdo = new PDO($dsn, options: $options);
        $this->animations = new AnimationPostgreSQLRepository($this->pdo);
    }

    protected function tearDown(): void
    {
        $this->pdo->query("DELETE FROM animations;");
    }

    public function testInsertAndGet(): void
    {
        $title = 'Naruto';
        $year = 2002;
        $season = 'Autumn';
        $genres = ['shonen', 'action', 'ninja'];

        $animation = Animation::fromRequest([
            'title' => $title,
            'year' => $year,
            'season' => $season,
            'genres' => $genres,
        ]);

        $animation = $this->animations->insert($animation);

        $returnedAnimation = $this->animations->get($animation->getId());
        $this->assertSame($title, $returnedAnimation->getTitle());
        $this->assertSame($year, $returnedAnimation->getYear());
        $this->assertSame(Season::AU, $returnedAnimation->getSeason());
        $this->assertSame($genres, $returnedAnimation->getGenres());
    }
}
```

Here you can see how I structured this test:

- The fields that you would need in every test (`PDO` and the `AnimationRepository` implementation) should be a private property of your `Test` class
- We can initialize these in the `setUp()` method
- We don't forget to delete the records from the `animations` table of our `ani-merged-test` database in the `tearDown()` method
- Then I test both the `insert()` and the `get()` method of the `interface` here
- I set up the variables so that I can test against them. I use the `fromRequest()` helper as if they were **coming from the client**
- Then I use the id generated by the database and given back by the `insert()` method to pass into the `get()` method
- Lastly, I can just assert as many cases as I want

Of course in a real-world situation, you would write more tests for the `insert()` and `get()` as well, for example, you could write a test to see if the Exceptions are properly thrown or not. And of course, you would cover the other methods as well.

Optimistic Locking

We are going to prevent the data race with the version field. We need to be cautious in the `update()` Controller action because we are first retrieving the resource, then updating it, making a data race possible.

But we also made sure to always update the **version to be equal to version + 1** each time we call the (now) `PATCH` route.

So if 2 users retrieve the resource at the same time, one of them has to do the update first. This means that the other user just can check the version column in the database and if it's already incremented, we refuse the update operation from this user.

File: ./src/Repository/AnimationPostgreSQLRepository.php

```
/**
 * @throws DatabaseException
 * @throws ReturningException
 * @throws EditConflictException
 */
#[\Override]
public function update(Animation $animation): Animation
{
    // We allow the UPDATE if the version is not changed
    $sql = 'UPDATE animations
        SET title = :title, year = :year, season = :season,
            genres = :genres, version = version + 1
        WHERE id = :id AND version = :version
        RETURNING version';

    $genres = Arr::toStr($animation->getGenres());

    try {
        $this->pdo->beginTransaction();

        // Add new bound value here
        $stmt = $this->pdo->prepare($sql);
        $stmt->execute([
            'title' => $animation->getTitle(),
            'year' => $animation->getYear(),
            'season' => $animation->getSeason()->value,
            'genres' => $genres,
            'id' => $animation->getId(),
            'version' => $animation->getVersion(),
        ]);
```

```php
            // if 0 rows were updated, there was a conflict with version
            // throw new custom Exception
            $rowsAffected = $stmt->rowCount();
            if ($rowsAffected === 0) {
                throw new EditConflictException('edit conflict for animation');
            }

            $data = $stmt->fetch();
            if ($data === false) {
                throw new ReturningException('RETURNING version SQL failed');
            }

            $this->pdo->commit();
        } catch (ReturningException $ex) {
            $this->pdo->rollBack();
            throw $ex;
        } catch (Throwable $ex) {
            $this->pdo->rollBack();

            throw new DatabaseException('animation update failed()',
                    previous: $ex);
        }

        $animation->setVersion($data['version']);

        return $animation;
    }
```

Now we just need to add one more catch branch to the in the **`Controller`**:

File: ./src/Controller/AnimationController.php

```php
public function update(
    Request $request,
    Response $response,
    array $args
): Response
{
    // …
    try {
        $animation = $this->animations->update($animation);
    } catch (EditConflictException) {
        return $this->formatter->editConflict($response);
    } catch (ReturningException|DatabaseException $e) {
        return $this->formatter->serverError($response, $request, $e);
    }

    return $this->formatter
        ->writeJSON($response, $animation->asJson(), envelope: 'anime');
}
```

Redis Integration

Why do we need `Redis`? It is an **in-memory** database that is fast but also writes to disc and it has built-in support for **forgetting** records after a set amount of time, which makes it optimal for using it as a cache, and in our case for a rate limiter.

We use it because our different **`child processes`** don't share their memory, since each HTTP request runs against a copy of our **`index.php`** on a newly created thread by the **PHP-FPM**, so if a client uses one of our routes for example 4 times we want to remember that he made 4 requests quickly after each other.

If the client makes too many requests within a certain timeframe, we don't want to allow the **`Request`** to be processed. But after some time we want to delete this info from the **`Redis`** database so the client can try again (Redis forgetting capabilities). This way we defend ourselves from request timing attacks and botnet attacks.

We want to use some **`env variables`** so we can make these time values configurable from the **`.env`** file.

You can make use of different types of logic when it comes to rate limiting, you can make:

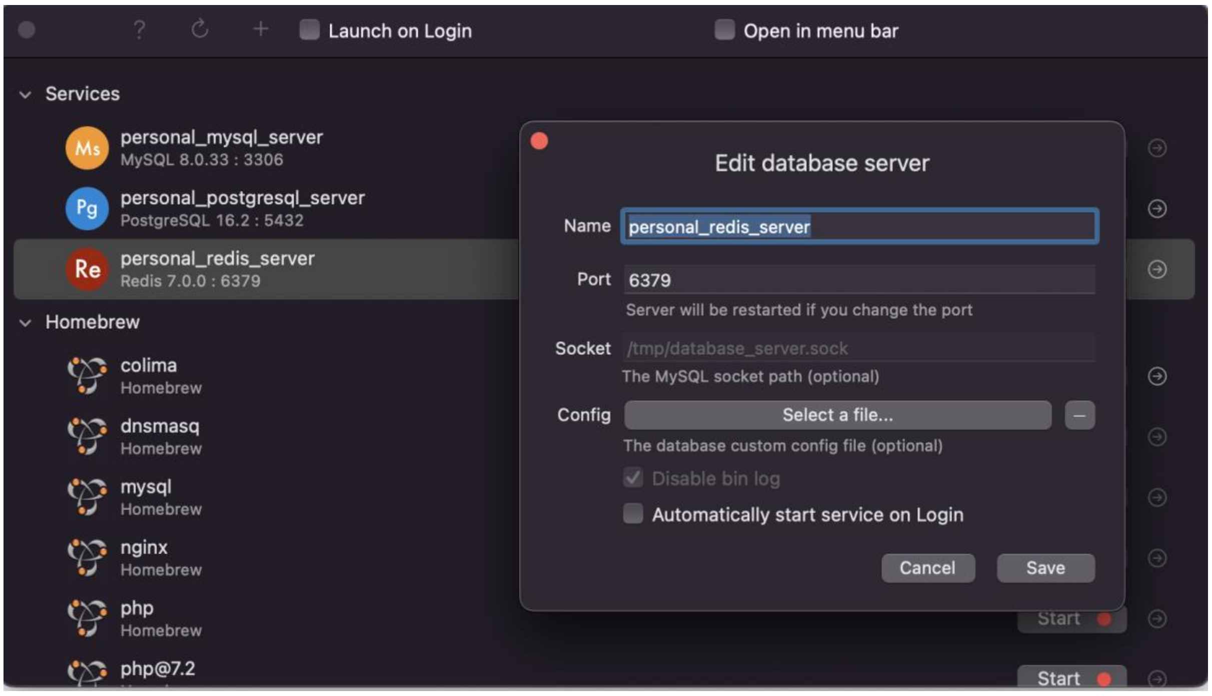
- A **global rate limiter**, that only counts the number of requests made to **any** of your application's routes in a given timeframe
- a **local rate limiter**, that counts requests per client basis, but it will summarize all of the requests of a given client, so if the client makes 4 requests to 4 different routes, we will remember the number 4 only
- a **local rate limiter**, that counts requests per client basis again, but counts these per route basis as well, so we remember how many requests each client made to each route

So you can make it as configurable your business logic as you want. In this book, I am going to make the second version, so we will remember each client by their **IP Address**, but we won't differentiate between what route the request is made to, and if the number for a given client exceeds our limit value, we send back a **`429 Response`**.

Redis Database

You can use `docker` if you want, I am going to make use of the tools I have been using so far: `DBngin`, and `TablePlus`. I will create a Redis server locally and I name it **personal_redis_server**, it will run on port `6379` which is the standard port for `Redis`.

Image: Dbngin UI



personal_mysql_server
MySQL 8.0.33 : 3306
personal_postgresql_server
PostgreSQL 16.2 : 5432
personal_redis_server
Redis 7.0.0 : 6379
colima
Homebrew
dnsmasq
Homebrew
mysql
Homebrew
nginx
Homebrew
php
Homebrew
php@7.2

Edit database server

Name: personal_redis_server
Port: 6379
Server will be restarted if you change the port
Socket: /tmp/database_server.sock
The MySQL socket path (optional)
Config: Select a file...
The database custom config file (optional)
Disable bin log
Automatically start service on Login
Cancel Save

That is it, at its most basic use we don't need a **user** and a **password** right now.

Predis

The next thing we need is a client so we can use `Redis` commands from the **PHP code**. We will use the predis/predis library for this use case, which is a well-written adapter for `Redis` in PHP.

```
$ valet composer require predis/predis
```

The next thing we need is to set some environmental variables:

- **scheme**: we will use **TCP** in this case, you can read the **GitHub** page for different options and their use
- **host**: of course, where the `Redis` database is located, we will use localhost
- **port**: again, of course, we configured our server to run on port `6379`

- **requests**: I am going to use this to configure the value that how many requests a client can make in the given timeframe
- **expiration**: we use to define the timeframe we just mentioned
- **storage_key_format**: `Redis` is a **key-value** database, so we want to have unique keys, I will configure a `sprintf()` string here that expects one `%s` value in it, which is the IP address of the client

Let's make these variables available in the `.env` and `.env.example` files, and set the rules for them in the `dotenv_rules.php` file. Lastly let's also add them to our `app.php` file so we can retrieve them with the `Configuration` object:

File: ./.env

```
# ...
# Redis
REDIS_SCHEME=tcp
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_REQUESTS=30
# We let 30 requests/user
REDIS_EXPIRATION=60
# For 60 seconds (1 min)
REDIS_STORAGE_KEY_FORMAT=rate:%s:requests
# For me the key would look like rate:127.0.0.1:requests
```

File: ./.env.example

```
# ...
# Redis, same as in .env
REDIS_SCHEME=tcp
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_REQUESTS=30
REDIS_EXPIRATION=60
REDIS_STORAGE_KEY_FORMAT=rate:%s:requests
```

File: ./config/dotenv_rules.php

```php
return function (Dotenv $dotenv) {
    // ...
    // Redis
    $dotenv->required('REDIS_HOST');
    $dotenv->required('REDIS_PORT');
    $dotenv->required('REDIS_SCHEME')
        ->allowedValues(['tcp', 'tls']);
    $dotenv->required('REDIS_REQUESTS');
    $dotenv->required('REDIS_EXPIRATION');
    $dotenv->required('REDIS_STORAGE_KEY_FORMAT');
};
```

File: ./config/app.php

```php
// ...
return [
    // ...
    'db' => [
        'pgsql' => [
            // ...
        ],
        'redis' => [
            'connection' => [
                'scheme' => $_ENV['REDIS_SCHEME'],
                'host' => $_ENV['REDIS_HOST'],
                'port' => intval($_ENV['REDIS_PORT']),
            ],
            'config' => [
                'requests' => intval($_ENV['REDIS_REQUESTS']),
                'expiration' => intval($_ENV['REDIS_EXPIRATION']),
                'storage_key_format' => $_ENV['REDIS_STORAGE_KEY_FORMAT'],
            ],
        ],
    ],
];
```

The last thing in this chapter is that we register a `ClientInterface` from the `predis` package so we can just inject it into our constructors:

File: ./config/bindings.php

```php
// ...
return [
    // ...
    // (Redis) Database
    ClientInterface::class => function(Configuration $config) {
        return new Client([
            'scheme' => $config->get('db.redis.connection.scheme'),
            'host' => $config->get('db.redis.connection.host'),
            'port' => $config->get('db.redis.connection.port'),
        ]);
    },
];
```

You can read the `predis` docs and also set up a password and a user to make it more safe, but for an example, this will do.

Different Authentication Techniques

Important: For all authentication methods that we're describing in this chapter, it's assumed that your API only communicates with clients over HTTPS.

Authorization: Basic

The HTTP protocol itself comes with a basic authentication system. We send in an `Authorization` header with the `Request` object. After the `Authorization:` you need to type the `Basic` word to denote the type of authentication you want to use.

Yes, in the **HTTP protocol itself, authorization and authentication are often confused**, for example, the `401` status code should be **Unauthenticated** instead of `Unauthorized`.

After the `Basic,` you need to have the **base64 encoded** string of your credentials, that are in the **username:password** format. So a valid example would be:

```
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

Its advantage lies in its simplicity, it is supported out-of-the-box in most programming languages, web browsers, and tools such as `curl` or `wget`.

Best used for APIs, where you don't have real users, but want to have an easy authentication setup. For APIs, where you have real users, it is a bad fit and will create lots of overhead. You can consider it if your API's traffic is low.

Authorization: Bearer

This is also called token-based authentication. The process is the following:

- The client sends in a `Request` containing their credentials, typically the email and the password
- The API verifies the credentials, then creates a token in a similar way we already did with the `User` activation and sends it back
- Then in subsequent requests, when the client wants to access a restricted resource, it sends in the `Authorization` Header in the following form:

```
Authorization: Bearer <token>
```

- Then we validate the token and reject or allow the `Request` based on the validation

Best used for APIs, where you store the client credentials, like ours. The downside is that managing tokens can be quite complicated.

We can break down token-based authentication into further categories:

Stateful token authentication

Stateful means, that we need to manage the state of the token ourselves. Typically this means, that you have a **tokens database table** where you manage the expiration of your token. If you think about it, we already do something similar in our API.

This is good because if some of the tokens get stolen, you can just delete the corresponding database record from the tokens table, and you just revoke the token, prohibiting further misuse. This approach gives us full control over the state of the token.

Of course, the downside is that you need to manage the tokens on the server side, but this is not an actual downside in most cases. We will use this approach for our authentication system.

Stateless token authentication

This time the state of the token is encoded into the token itself. So we don't store it in our database usually. This is often achieved with a **JWT token**.

The advantage is that the encoding and decoding of the tokens can be done in memory, but it is really hard to revoke the token once stolen. And these tokens are highly configurable, so there are lots of things to get wrong. You should always use a well-established library for these instead of writing your own.

This approach is best used for delegated authentication, so in a microservice architecture between your microservices, it can work great.

Token Types

JWT stands for JSON Web Token, which is the most common way for stateless token implementation. But here are other, more reliable methods, **like PASETO tokens**.

Resource: There is a really good paid video by Techschool on this topic. It tells you everything you need to know about JWT and PASETO tokens.

The other types of things you usually hear are **AccessTokens and IdTokens**. AccesToken is usually just a hash that you send in the Request header and it gives you access to different resources, hence the name.

IdTokens on the other hand are usually JWT tokens that also have the information of the User encoded in them and used for authentication, instead of authorization, like access tokens.

Some **RefreshTokens usually** come together with AccessTokens and they give you the ability to refresh the duration of the AccessToken once expired. They have a longer expiration date. And they are not a typical hash, nor a JWT, but a so-called opaque token, so a special kind of hash that the Authorization server, that gave you the access -and refers token pairs, understands. But another system won’t understand this type of token.

Resource: If you are curious about these tokens, I recommend this free video resource for watching.

Authorization: Key

We also have a so-called **API key authentication**. I like explaining it in a way that this is the same as the stateful token approach, but without expiration.

So you have a database table, you generate an API key and send it to the user after a successful registration. Then the user sends in this key in the `Authorization:` header every time it wants to access a restricted resource.

Of course, there are different ways of implementing it, usually the user can generate it themselves and even put different permissions inside such key, but the basics are the same. An example would be:

```
Authorization: Key <token>
```

Oauth 2.0 /OpenID Connect

Important: Oauth 1.0 is deprecated, don’t use it!

The main flow of this is the following:

- When you want to authenticate a request, you redirect the user to an ‘authentication and consent’ form hosted by the **identity provider**
- If the user consents, then the identity provider sends your **API an *authorization code***
- Your API then sends the authorization code to another endpoint provided by the identity provider. They verify the authorization code, and if it’s valid they will send you a **JSON response containing an *ID token***
- This ID token is itself a JWT. You need to validate and decode this JWT to get the actual user information, which includes things like their email address, name, birth date, timezone, etc
- Now that you know who the user is, you can then implement a stateful or stateless authentication token pattern so that you don’t have to go through the whole process for every subsequent request

This is the so-called **Authorization CodeFlow**, and there are more ways you can implement it, but this is the most general way.

The main advantage of this is that you don’t store user information, the identity provider does. You redirect the user to Google or Facebook, which is an identity provider as well, and then Google sends an authorization code to our API. We then sent it back to Google to see if the code was valid and we did not get it from a hacker posing as Google. Then Google sends us a JWT IdToken that has the information of the user and now we have an authenticated user.

The first thing to mention here is that *OAuth 2.0 is not an **authentication protocol***, and you shouldn’t use it for authenticating users. The oauth.net website has a great article explaining this, and I highly recommend reading it.

If you want to implement authentication checks against a third-party identity provider, you should use OpenID Connect (which is built directly on **top of OAuth 2.0).**

So Oauth 2.0 In itself is an authorization protocol, so it would give you an AccessToken that enables you to access resources. By combining it with OpenId Connect we can receive an IdToken with the User info instead and authenticate the user. Important distinction.

Resource: So here is a last resource for you if you want to take a deeper dive into this protocol.

As I mentioned, we will implement a token-based stateful authentication in the next chapters, because we already have most parts of this in place.

Chapter 21.1.

CORS Example

We can simulate a front-end app easily. For example, you can download Visual Studio Code, then the Live Server extension needs to be installed.

Then in the bottom right corner of your new code editor, you will see **a Go Live button** that will serve the current folder. So we can just create an `index.html` file in a simple directory and put this code in there:

File: ./index.html

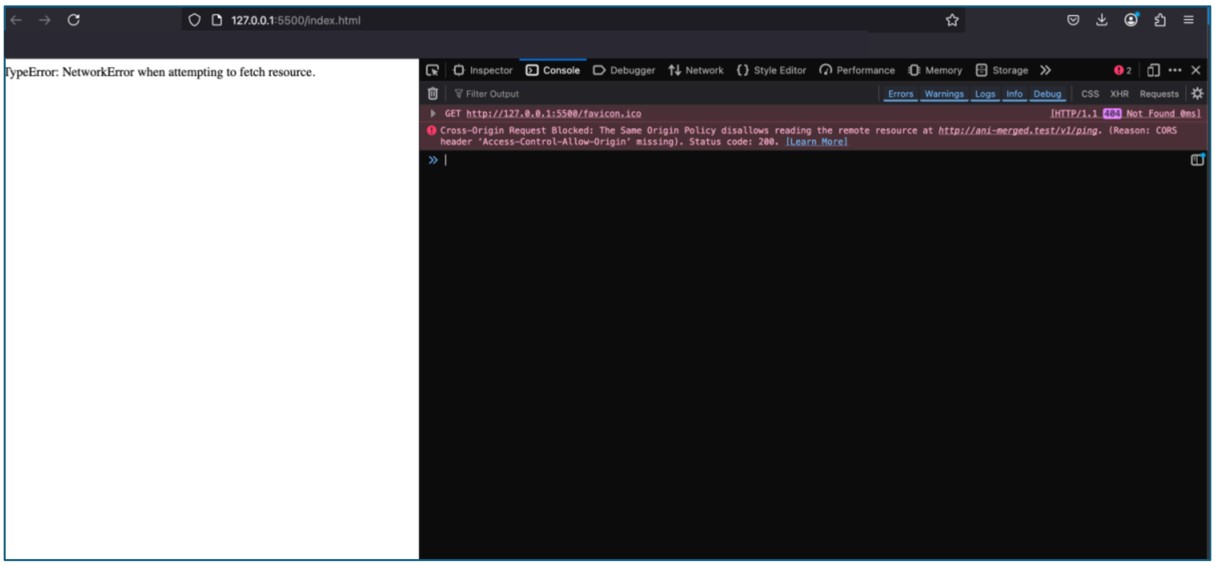
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>FE Test</title>
  </head>
  <body>
    <div id="output"></div>
  </body>



  <script>
    document.addEventListener('DOMContentLoaded', function() {
      fetch("http://ani-merged.test/v1/ping").then(
        function (response) {
          response.text().then(function (text) {
            document.getElementById("output").innerHTML = text;
          });
        },
        function(err) {
          document.getElementById("output").innerHTML = err;
        }
      );
    });
  </script>
</html>
```

- We use the fetch() function to request our API ping endpoint. By default, this ends a `GET` request, but it's also possible to configure this to use different HTTP methods and add custom headers. We'll explain how to do that later
- The `fetch()` method works *asynchronously* and returns a promise. We use the `then()` method on the promise to set up two callback functions: the first callback is executed if `fetch()` is successful, and the second is executed if there is a failure
- In our 'successful' callback we read the response body with the `response.text()` and used the `document.getElementById("output").innerHTML` to replace the contents of the `<div id="output"></div>` element with this response body
- In the 'failure' callback we replace the contents of the `<div id="output"></div>` element with the error message
- This logic is all wrapped up in the `document.addEventListener('DOMContentLoaded', function(){…}),` which essentially means that `fetch()` won't be called until the user's web browser has completely loaded the HTML document

Image: FE Test App

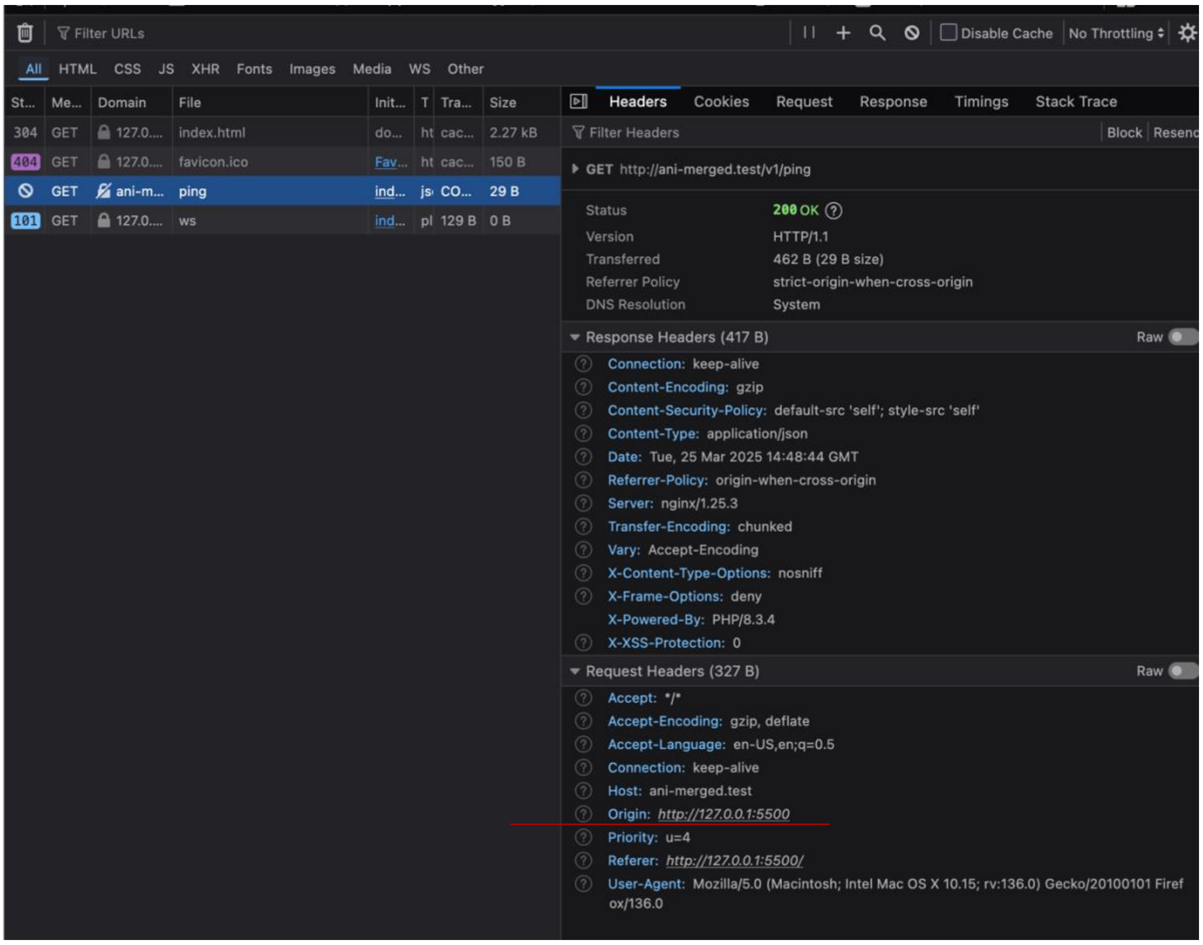


```
TypeError: NetworkError when attempting to fetch resource.
```

```
GET http://127.0.0.1:5500/favicon.ico                    [HTTP/1.1 404 Not Found 0ms]
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at http://ani-merged.test/v1/ping. (Reason: CORS
header 'Access-Control-Allow-Origin' missing). Status code: 200. [Learn More]
```

The first thing is that the headers demonstrate that the request was sent to our API, which processed the request and returned a successful `200 OK` response to the web browser containing all our standard response headers. To re-iterate: *the request itself was not prevented by the same-origin policy — it's just that the browser won't let JavaScript see the response*.

Image: FE Test App



```
Filter URLs                                  Disable Cache   No Throttling
All  HTML  CSS  JS  XHR  Fonts  Images  Media  WS  Other

St...  Me...  Domain     File         Init...  T   Tra...  Size
304    GET    127.0....  index.html   do...    ht  cac...  2.27 kB
404    GET    127.0....  favicon.ico  Fav...   ht  cac...  150 B
       GET    ani-m...   ping         ind...   js  CO...   29 B
101    GET    127.0....  ws           ind...   pl  129 B   0 B

Headers   Cookies   Request   Response   Timings   Stack Trace
Filter Headers                                       Block  Resend

GET http://ani-merged.test/v1/ping

Status              200 OK
Version             HTTP/1.1
Transferred         462 B (29 B size)
Referrer Policy     strict-origin-when-cross-origin
DNS Resolution      System

Response Headers (417 B)                                     Raw
Connection: keep-alive
Content-Encoding: gzip
Content-Security-Policy: default-src 'self'; style-src 'self'
Content-Type: application/json
Date: Tue, 25 Mar 2025 14:48:44 GMT
Referrer-Policy: origin-when-cross-origin
Server: nginx/1.25.3
Transfer-Encoding: chunked
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-Powered-By: PHP/8.3.4
X-XSS-Protection: 0

Request Headers (327 B)                                      Raw
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Host: ani-merged.test
Origin: http://127.0.0.1:5500
Priority: u=4
Referer: http://127.0.0.1:5500/
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:136.0) Gecko/20100101 Firef
ox/136.0
```

The second thing is that the web browser automatically sets an `Origin` header on the request to show where the request originates from (highlighted by the red line above).