API's

ONTWIKKELEN IN VANILLA PHP

INHOUDSOPGAVE

INLEIDING

In dit document beschrijven we in de basis waar we als API developer rekening mee moeten houden tijdens de ontwikkeling van een API. We beschrijven de voorbereiding, na de voorbereiding gaan we pas coderen.

STAP 1 – DATA

In deze stap gaan we eerst inventariseren met welke data en welke structuur we te maken hebben.

In ons geval gebruiken we een database die we al in klas 1 hebben gebruikt voor het bouwen van een Forum in vanilla PHP.

De database bestaat uit 4 tabellen:

Tabel	Actie	Rijen	Type	Collatie	Grootte	Overhead
replies	Verkennen Structuur Zoeken Invoegen Legen Verwijderen	100	InnoDB	utf8mb4_unicode_ci	96,0 KiB	-
threads	Verkennen Structuur Zoeken Invoegen Legen Verwijderen	27	InnoDB	utf8mb4_unicode_ci	16,0 KiB	-
topics	Verkennen Structuur Zoeken Invoegen Legen Verwijderen	50	InnoDB	utf8mb4_unicode_ci	64,0 KiB	-
users	Verkennen Structuur Zoeken Invoegen Legen Verwijderen	6	InnoDB	utf8mb4_unicode_ci	32,0 KiB	-
4 tabellen	**Som**	**183**	**InnoDB**	**utf8_general_ci**	**208,0 KiB**	**0 B**

Figuur 1 - De database

STAP 2 – DATA BESCHIKBAARHEID

TABELLEN

In deze stap gaan we eerst bepalen welke tabellen we toegankelijk willen maken voor client apps.

We bepalen dat uitsluitend de volgende tabellen toegankelijk zijn voor client apps:

- replies
- threads
- topics

De tabel users is dus niet toegankelijk via de API.

KOLOMMEN

Per tabel bepalen we vervolgens welke beschikbaar zijn voor client apps.

threads

Kolom	Read	Write	Opmerkingen
id	✓		Wordt door de DB server bijgehouden
title	✓	✓	
description	✓	✓	
user_id	✓	✓	
created_at	✓		Wordt door de API ingevuld en bijgehouden
updated_at	✓		Wordt door de API ingevuld en bijgehouden

topics

Kolom	Read	Write	Opmerkingen
id	✓		Wordt door de DB server bijgehouden
title	✓	✓	
body	✓	✓	
user_id	✓	✓	
thread_id	✓	✓	
created_at	✓		Wordt door de API ingevuld en bijgehouden
updated_at	✓		Wordt door de API ingevuld en bijgehouden

replies

Kolom	Read	Write	Opmerkingen
id	✓		Wordt door de DB server bijgehouden
body	✓	✓	
user_id	✓	✓	
topic_id	✓	✓	
created_at	✓		Wordt door de API ingevuld en bijgehouden
updated_at	✓		Wordt door de API ingevuld en bijgehouden

STAP 3 – FUNCTIONALITEIT

In deze stap bepalen welke functionaliteit en via welk type request we de client apps bieden.

GET-REQUESTS

Met een GET-request stellen we de client apps in staat om data uit de database op te vragen. Dit doen we d.m.v. een aantal URL's.

De domeinnaam die we in de lessen gebruiken is: http://forum-php-api.test

URL	Beschrijving
http://forum-php-api.test/	Levert een JSON-response op met alle data van de threads in de database.
http://forum-php-api.test/threads	Levert een JSON-response op met alle data van de threads in de database.
http://forum-php-api.test/topics	Levert een JSON-response op met alle data van de topics in de database.
http://forum-php-api.test/replies	Levert een JSON-response op met alle data van de replies in de database.
http://forum-php-api.test/thread/{id}	Levert een JSON-response op met de data van één thread. Het laatste deel in de URL is een ID van de thread die opgevraagd wordt.
http://forum-php-api.test/topic/{id}	Levert een JSON-response op met de data van één topic. Het laatste deel in de URL is een ID van de topic die opgevraagd wordt.
http://forum-php-api.test/reply/{id}	Levert een JSON-response op met de data van één reply. Het laatste deel in de URL is een ID van de reply die opgevraagd wordt.

POST-REQUESTS

We staan client apps toe om threads, topics of replies toe te voegen aan de database. Daarvoor geven we aan dat een client app de data van een nieuwe thread, topic of reply op dezelfde manier moet worden meegestuurd in de request als browser ook doen bij het opsturen van de data in een formulier. In JavaScript is dat FormData.

LET OP!!!!!

In onze API doen we niets aan **authenticatie**. En POST-requests wil je normaal gesproken niet zonder authenticatie toestaan. Zoals wij het nu programmeren betekent dit dat iedereen zonder restricties iets kan toevoegen aan onze database. Dit is iets wat je **NOOIT** wil in reële projecten.

URL	Beschrijving
http://forum-php-api.test/thread	Staat de client app toe een thread toe te voegen aan de database. De data dienen als FormData meegestuurd te worden in de request.
http://forum-php-api.test/topic	Staat de client app toe een topic toe te voegen aan de database. De data dienen als FormData meegestuurd te worden in de request. Een ID van een user en een thread moet eveneens in de FormData meegegeven worden.
http://forum-php-api.test/reply	Staat de client app toe een reply toe te voegen aan de database. De data dienen als FormData meegestuurd te worden in de request. Een ID van een user en een thread moet eveneens in de FormData meegegeven worden.

PUT/PATCH-REQUESTS

In onze API zien we een PACTH-request als een PUT-request. We maken dus geen onderscheid tussen deze twee. Met een PUT-request staan we een client app toe data in de database te wijzigen.

URL	Beschrijving
http://forum-php-api.test/thread/{id}	De thread met de gegeven ID (in de URL) zal worden gewijzigd. De nieuwe data worden meegestuurd in de request in de vorm van FormData.
http://forum-php-api.test/topic/{id}	De topic met de gegeven ID (in de URL) zal worden gewijzigd. De nieuwe data worden meegestuurd in de request in de vorm van FormData.
http://forum-php-api.test/reply/{id}	De reply met de gegeven ID (in de URL) zal worden gewijzigd. De nieuwe data worden meegestuurd in de request in de vorm van FormData.

LET OP!!!!!

In onze API doen we niets aan **authenticatie**. En PUT-requests wil je normaal gesproken niet zonder authenticatie toestaan. Zoals wij het nu programmeren betekent dit dat iedereen zonder restricties iets kan wijzigen in onze database. Dit is iets wat je **NOOIT** wil in reële projecten.

DELETE-REQUEST

We staan de client app eveneens toe om data uit de database te verwijderen.

URL	Beschrijving
http://forum-php-api.test/thread/{id}	De thread met de gegeven ID (in de URL) zal worden verwijderd.
http://forum-php-api.test/topic/{id}	De topic met de gegeven ID (in de URL) zal worden verwijderd.
http://forum-php-api.test/reply/{id}	De reply met de gegeven ID (in de URL) zal worden verwijderd.

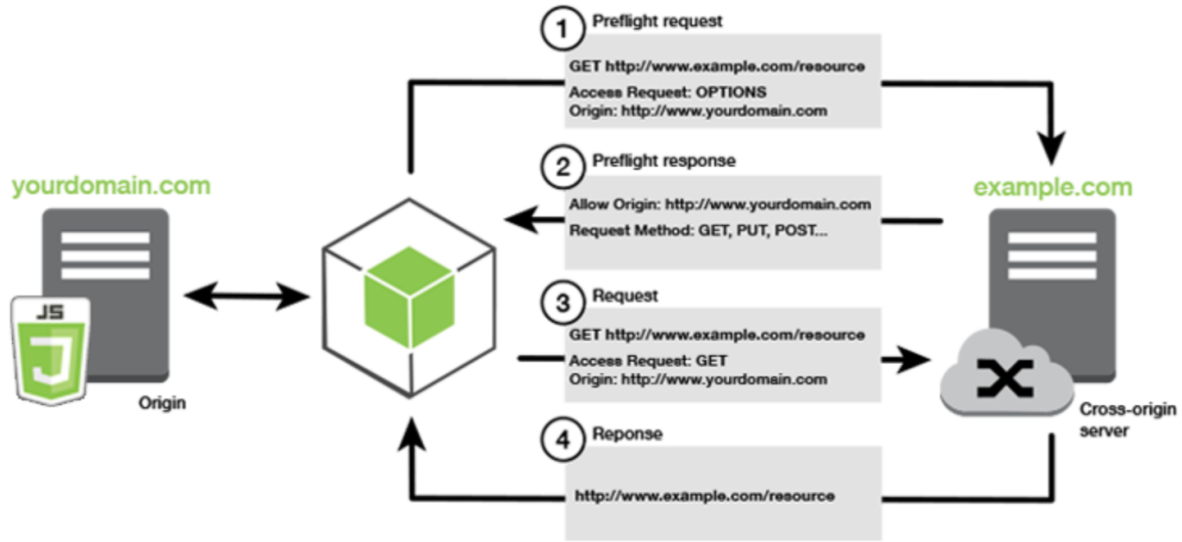
STAP 4 – CORS

CORS staat voor Cross-Origin-Resource-Sharing en biedt een beveiliging vanuit de browser tegen onjuist gebruik van externe resources. Wat houdt dit eigenlijk in?

VOORBEELD

Een client app draait op een server en onder domeinnaam yourdomain.com. De API echter draait op een andere server met een eigen domeinnaam, nl. example.com. CORS staat requests naar een andere server, zonder speciale stappen, niet toe uit veiligheidsoverwegingen. Dit betekent dat zonder deze speciale stappen uit te voeren de client app niet in staat is gebruik te maken van de API en diens resources (b.v. database).

Een browser voert in het geval van twee servers altijd eerst een zogenaamde preflight uit (hand shake) waarin de server van de API gevraagd wordt of de communicatie vanuit de client is toegestaan. Als de API daar in positieve zin op reageert door een speciale response te sturen kan een client app vervolgens wel een request doen. De API bepaald in de speciale response wat dan is toegestaan aan requests. (Zie figuur 2)



Figuur 2 - CORS afhandeling

WELKE REQUEST VERSTUURD EEN BROWSER?

De browser stuurt een OPTIONS-request, waarbij in de header de domeinnaam van de client app wordt meegestuurd.

HOE MOET ONZE API REAGEREN OP EEN OPTIONS-REQUEST?

Onze API dient een header te sturen op de onderstaande manier:

```
30          if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
31              header('Access-Control-Allow-Origin: *');
32              header('Access-Control-Allow-Methods: POST, GET, DELETE, PUT, PATCH, OPTIONS');
33              header('Access-Control-Allow-Headers: token, Content-Type, Accept, Access-Control-Allow-Origin');
34              header('Access-Control-Max-Age: 1728000');
35              header('Content-Length: 0');
36              header('Content-Type: text/plain');
37              if($stop_execution)
38                  die();
39          }
40      }
```

Figuur 3 - API response op CORS

Regel 31

Hiermee geeft onze API aan dat het niet uitmaakt vanuit welke domeinnaam de request komt.

Regel 32

Hiermee vertellen we de browser welke requests toegestaan zijn.

Regel 33

Hiermee vertellen we de browser welke headers toegestaan zijn.

NIET WEBBASED APPS

Let op, bovenstaande geldt alleen voor Web Applicaties en Websites. Mobile apps en apps als Postman kunnen vaak zonder CORS gebruik maken van een API.

STAP 5- STRUCTUUR VAN ONZE CODE

INLEIDING

We willen gebruik maken van clean URL's. Daarom moeten we, voor een Apache server, een **.htaccess file** aanmaken in de map waar het bestand **index.php** geplaatst is.

.HTACCESS

Met dit bestand vertellen we de Apache server hoe om te gaan met clean URL's. Ik ga niet verder in op hoe dit bestand tot stand komt.

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /

    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d

    # http://forum-api-2021.test/thread/1
    RewriteRule ^([^/]+)/([^/]+)/?$ index.php?resource=$1&id=$2 [L,QSA]

    # http://forum-api-2021.test/thread
    RewriteRule ^([^/\.]+)/?$ index.php?resource=$1 [L]
</IfModule>
```

Figuur 4 - .htaccess voor clean URL's

STAP 6 – CODEREN

INLEIDING

Wat we absoluut willen is verschillende verantwoordelijkheden in functionaliteit en features delegeren. Dit betekent dat we code onderverdelen in verschillende bronbestanden op basis van de functionaliteit en de features en ook daar de verantwoordelijkheid laten. Maar dan moeten we wel van tevoren weten om welke functionaliteit en feature het gaat en wat de verantwoordelijkheden zijn die we delegeren. We willen zoveel mogelijk code volgens het principe van OOP schrijven.

FUNCTIONALITEIT, FEATURES EN VERANTWOORDELIJKHEDEN

We kunnen de volgende zaken onderscheiden in wat onze API moet doen:

1. **Entry point**
   Hier komen alle requests binnen. Het enige wat een entry point moet doen is het doorgeven van de requests aan het verantwoordelijke deel van de code. De code in een entry point is zo minimaal mogelijk.

2. **Request afhandelen**
   De code die we schrijven draagt alleen de zorg voor:
   a. Het analyseren van de request
   b. Het bepalen welk onderdeel van onze code de vervolgacties moet uitvoeren
   c. Het overdragen van de verantwoordelijkheid aan de code voor de vervolgactie.
   d. Het ontvangen van de resultaten vanuit die vervolgacties
   e. Het doorgeven van de resultaten aan code die verantwoordelijk is voor het juist vormgeven van de response naar de client app.
   f. Het terugsturen van juist vormgegeven response data naar een client.

3. **Response**
   De code die we hier gaan schrijven draagt alleen de verantwoordelijkheid van het juist vormgeven van de response. Het maakt in principe deze code niet uit wat binnenkomt aan resultaten. Het juist vormgeven houdt in dat deze code de juiste headers samenstelt, de juiste structuur van response data vormgeeft en uiteindelijk dit pakket weer teruggeeft aan de code die de requests afhandelt.

4. **Regisseren vervolgacties**
   We willen een regisseur voor benodigde vervolgacties ontwikkelen. In de design pattern MVC noemen we een dergelijke regisseur een **Controller**. We gaan daarom in ons ontwerp dit pattern volgen. Een controller regisseert dus welke detailstappen er moeten worden uitgevoerd om het gewenste resultaat te krijgen aan data. Ook de regisseur delegeert weer, want een Controller is bijvoorbeeld weer niet verantwoordelijk voor het benaderen van een database.

5. **Data verzamelen**
   De code die we hier gaan schrijven is verantwoordelijk voor het benaderen en manipuleren van de database en het verzamelen van de gewenste data. In de design pattern MVC noemen we dit een **Model**. Ook nu volgen we deze pattern.

6. **Raw database benadering**
   We gaan de code die op het laagste niveau met een database server communiceert ook apart nemen in een eigen bestand. We willen namelijk, wanneer we b.v. een andere type database server willen gebruiken niet de Models gaan aanpassen.

ENTRY IN ONZE API

FILE: index.php

Zoals aangegeven in de **.htaccess file** is de entry point voor onze API altijd de file **index.php**.

REQUESTS AFHANDELEN

FILE: RequestHandler.php

In een class (OOP) **RequestHandler** programmeren we de verantwoordelijkheid rondom het afhandelen van de requests en het terugsturen van een response.

RESPONSE

FILE: ApiResponse.php

In een class **ApiResponse** programmeren we de verantwoordelijkheid om een response op de juiste manier vorm te geven en terug te geven aan de **RequestHandler**.

REGISTEREN VERVOLGACTIES

FILE: xxxxController.php

Voor iedere resource (tabel in de database b.v.), in overeenstemming met MVC, maken we een aparte Controller. Daarmee verdelen we de verantwoordelijkheid onder de Controllers ook nog eens m.b.t. de resources.

DATA VERZAMELEN

FILE: xxxxModel.php

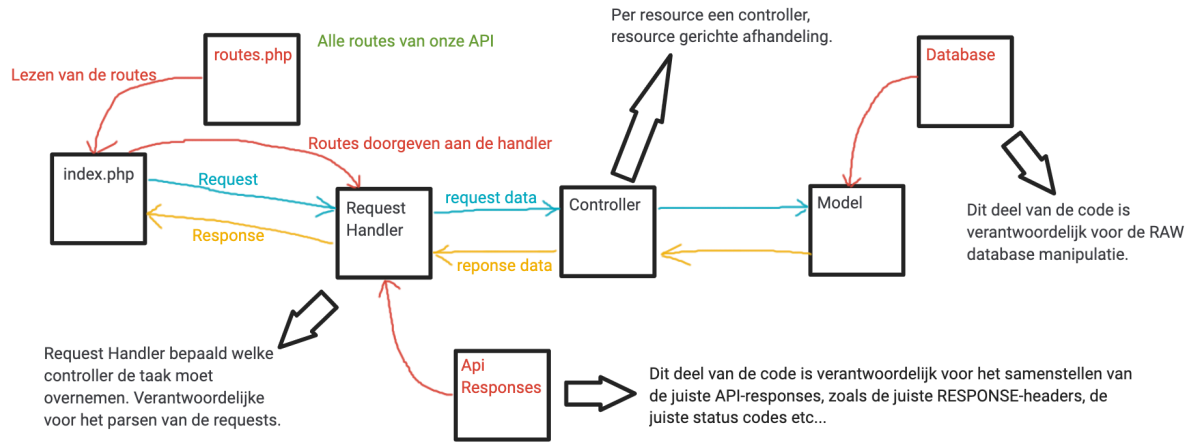
Voor iedere resource, in overeenstemming met MVC, maken we een apart Model.

RAW DATABASE BENADERING

FILE: Database.php

In een apart bestand programmeren we op het laagste niveau de communicatie met de database server. Dit doen we door gebruik te maken van PDO.

SCHEMATISCHE WEERGAVE



Figuur 5 - Schematische structuur API

MAPPENSTRUCTUUR

Onze API richten we in volgens het PSR-4 principe. Dit betekent dat we a) een rootmap voor al onze classes. Maar ook dat we iedere class op basis van z'n verantwoordelijkheid in eigen sub map plaatsen. Of wanneer ze dezelfde verantwoordelijkheid hebben, maar betrekking hebben op andere resources, groeperen we de classes in eenzelfde map. De mapnamen hebben duidelijke zelf beschrijvende namen en, volgens de PSR-4 regels, beginnen met een hoofdletter. Door een juiste mappenstructuur aan te maken kunnen we met logische namespaces in PHP werken.

app	Dit is de rootmap voor al onze classes en is de root van onze namespace.
app/Database	Dit is de map voor onze class Database
app/Http	Dit is de map waarin we sub mappen en classes plaatsen die direct met het http-protocol te maken hebben.
app/Http/Controllers	De map waarin we alle Controllers in hun eigen bestand verzamelen.
app/Models	De map voor onze Models.
app/Routes	De map waarin we onze URL's beschrijven en welke acties daarbij moeten worden uitgevoerd.

BESTANDSNAMEN

Wanneer we een PHP-bestand aanmaken waarin we een class gaan definiëren geven we steeds een naam volgens dezelfde principes (PSR-4). Dit betekent dat een bestandsnaam dezelfde naam krijgt als de class die we daarin programmeren/definiëren. In elk bestand definiëren we ook steeds maar één class.

De bestanden **routes.php** en **index.php** wijken hierbij uiteraard af, want we definiëren hierin geen class.

NAMESPACES, PSR-4

We maken gebruik van namespaces en autoload volgens de regels van PSR-4. Dit kunnen we doen met behulp van de commandline opdracht composer.

We initialiseren ons project met de opdracht:

```
$ composer init
```

Figuur 6 - Initialiseren voor autoload en PSR-4

Vervolgens moeten we aan de file **composer.json** nog het onderstaande toevoegen:

```
{
    "name": "johan/php_api",
    "type": "project",
    "authors": [
        {
            "name": "Johan Strootman",
            "email": "jj.strootman@alfa-college.nl"
        }
    ],
    "require": {
      "ext-pdo": "*",
      "ext-json": "*"
    },
    "autoload": {
        "psr-4": {
            "App\\": "app/"
        }
    }
}
```

Figuur 7 - Composer.json aanpassen

Hiermee hebben we het nog niet volledig voor elkaar. We moeten de autoload.php file nog genereren met composer. Dit doen we door de volgende opdracht te geven in de terminal:

```
$ composer install
```

Figuur 8 - Genereren autoload

Nu hebben we een ***vendor*** map in ons project waarin we de file **autoload.php** vinden.