# Restful webservice

### Voorbeeld

https://edhub.novi.nl/study/courses/319/content/12843

### Te gebruiken bij:

-

### Eigenschappen:

-

### Info

#### **REST**

Omdat jouw backend door iedere willekeurige frontend aangesproken moet kunnen worden, is het handig als de communicatie volgens een vaste set regels verloopt.

<https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>

verschillende protocollen die hiervoor kunnen worden gevolgd: één daarvan is **REST(RE**presentational **S**tate **T**ransfer)**.**

REST is dus geen Framework of Library het bestaat niet uit een heleboel voorgedefinieerde code die je kunt aanspreken.

 Het is een set van principes en beperkingen die definiëren hoe webstandaarden (zoals HTTP en URLs) moeten worden gebruikt om schaalbare en onderhoudbare webdiensten te creëren.

Wanneer iedereen (frontenders, backenders en data engineers) zich aan deze principes en beperkingen houdt, kunnen we al onze applicaties (frontend, backend en database) vrijwel naadloos op elkaar aansluiten.  
om uiteindelijk een gedistribueerde webapplicatie te maken. Of web-API,

##### **Wat is een gedistribueerde webapplicatie?**

In de context van REST, verwijst "gedistribueerd" naar een systeem waarin verschillende delen van de applicatie of service zich op verschillende machines of locaties bevinden en met elkaar communiceren via het internet.

In een gedistribueerd systeem draaien bijvoorbeeld de database, de backend en de frontend allemaal op een ander systeem of een andere Cloud-omgeving. REST is erop gebouwd dat deze applicaties, hoe verschillend ze ook zijn, toch in staat zijn om met elkaar te communiceren.

##### API's

Een API is een **A**pplication **P**rogramming **I**nterface

Als we een API in de praktijk willen uitbeelden, zouden we naar een restaurant gaan. Stel je voor dat je uit eten bent: je zit aan tafel en bent klaar om te bestellen. Je hebt een manier nodig om jouw bestelling door te geven aan de keuken, maar ook om jouw gerecht bij jou aan tafel terug te ontvangen. Dit kan de kok niet zelf doen, want die is aan het koken. Evengoed kun je dit niet zelf doen, want jij bent de klant (en anders was je wel thuis gaan eten).

Je hebt iets nodig om de klant die het eten bestelt (de frontend die de data nodig heeft) en de kok die het bereidt (de backend en de database waar de gegevens staan), met elkaar te verbinden. Hier komt de ober - of in ons geval de API

Als het klaar is brengt hij de response, in dit geval het eten, terug naar jou. Een API is dus eigenlijk simpelweg de tussenpersoon.

Door middel van een vastgestelde set regels (het REST protocol) begrijpen de applicatie en de database wat ze van elkaar willen.

1. We beginnen onze bestelling altijd met “Ik zou graag wat eten willen bestellen” en vullen dit aan met onze bestelling: “Graag de pasta bolognese, alsjeblieft.” In een API werkt dit net zo. We specificeren wat voor verzoek we doen (data opvragen, verwijderen of juist versturen)
2. en waar dit verzoek naartoe moet, door middel van een specifiek webadres: het **endpoint.**
3. Dat verzoek komt daarom bij onze database controller uit, omdat alleen *die* controller naar dat endpoint luistert.

Web-API's worden gebruikt in een breed scala aan toepassingen, zoals social media platforms, financiële diensten, e-commerce websites.

Het werken met een API is namelijk heel flexibel. Ontwikkelaars gebruiken web-API's om nieuwe applicaties te bouwen, bestaande applicaties uit te breiden en applicaties te integreren met andere systemen

Wil je een voorbeeld zien van een web-API? Kijk dan eens naar de [Poké API](https://pokeapi.co/docs/v2) die te gebruiken is om informatie over Pokemons op te vragen.

<https://pokeapi.co/docs/v2>

*Side note: In de tekst hierboven spreken we over web-API, dan zul je je vast afvragen “is er dan ook een niet-web API?” Die is er zeker! Je JDK zit er vol mee zelfs. Wanneer je bijvoorbeeld een ArrayList maakt in Java, dan maakt je gebruik van de Collections API die je kunt vinden in de package Java.Utils.Collections.)*

##### De regels van REST

Je mag jezelf pas een RESTful web-API noemen wanneer je je houdt aan de volgende regels:

* Client en Server staat los van elkaar. Dit betekent ook dat de code aan de client-kant (de frontend) zomaar omgeruild kan worden met een compleet andere applicatie, zonder dat de server (de backend) daar iets van hoeft te weten.
* Een REST applicatie is **stateless**. Dit houdt in dat de server niks hoeft te weten over de staat van de client.   
  In REST wordt dit geïmplementeerd door gebruik te maken van resources in plaats van commandos.   
  Resources zijn de zelfstandige naamwoorden van het web. In de zin: “Geef mij een banaan”, zou banaan een resource zijn.    
  Een resource kan elk object, document of “ding” beschrijven dat je wilt opslaan of wilt verzenden over het web.   
  Door deze beperking levert REST een betere prestatie en is het betrouwbaarder en schaalbaarder.
* De data is **cacheable**. Cacheable betekend dat de data op verschillende plekken opgeslagen kan worden om later hergebruikt te worden.
* Het is een gelaagd systeem. Deze hiërarchie is natuurlijk terug te vinden in de verdeling van frontend/backend/database, die we eerder al hebben genoemd, maar binnen de backend zien we ook een hiërarchische verdeling in de presentation layer (controller), business layer (service) en de persistence layer (repository).
* Een REST applicatie heeft een uniforme interface (API). Dit kun je bereiken door rekening te houden met de volgende 5 principes:
  + Gebruik HTTP-methoden zoals GET, POST, PUT, DELETE en PATCH. Deze methoden hebben elk hun specifieke functie en geven een stukje context over het verzoek dat wordt gemaakt. Door deze methoden correct te gebruiken, kunnen API-gebruikers gemakkelijker voorspellen hoe ze met de API moeten interacteren en kunnen API-ontwikkelaars gemakkelijker de API implementeren en onderhouden.
  + Gebruik een uniforme URI-structuur: Zorg ervoor dat de URI's van je API eenvoudig, consistent en gemakkelijk te begrijpen zijn voor API-gebruikers. Gebruik bijvoorbeeld duidelijke namen voor resources en gebruik placeholders voor parameters.
  + Gebruik een uniforme datastructuur: Het is belangrijk om een uniforme datastructuur te gebruiken om ervoor te zorgen dat de API-gebruikers gemakkelijk de data kunnen begrijpen en verwerken. Je kunt hiervoor XML of JSON gebruiken, maar moderne applicaties gebruiken hoofdzakelijk JSON.
  + Hanteer consistentie in foutafhandeling: Zorg ervoor dat de API-foutcodes en foutmeldingen uniform zijn en goed gedocumenteerd zijn, zodat API-gebruikers gemakkelijk kunnen begrijpen wat er mis is gegaan en hoe ze het probleem kunnen oplossen. Zorg dat er betekenisvolle error-messages worden verstuurd.
  + Documenteer de API duidelijk: Het is belangrijk om de API goed te documenteren, zodat API-gebruikers een duidelijk beeld hebben van de functionaliteit van de API, de parameters en de mogelijke antwoorden.
* (Optioneel) Een REST applicatie geeft ook de mogelijkheid om code op te sturen die uitgevoerd kan worden of gedownload kan worden. Deze functionaliteit is niet verplicht voor REST en wordt verder ook niet behandeld in deze cursus.

##### Communicatie in REST

Communicatie tussen de frontend en de backend van een REST applicatie verloop via HTTP requests.

HTTP (**H**yper**t**ext **T**ransfer **P**rotocol) is een algemeen protocol voor communicatie op internet en is dus niet specifiek voor REST bedoeld.

een aantal handige tools waar we in REST dankbaar gebruik van kunnen maken,

* **HTTP-methodes**: Eerder al hebben we resources genoemd als de zelfstandige naamwoorden van een REST applicatie.   
  HTTP-methodes kun je zien als de werkwoorden van een REST applicatie.   
  REST maakt gebruik van de HTTP-methoden om aan te geven welke bewerking op een resource moet worden uitgevoerd.   
  Er zijn verschillende methoden beschikbaar in HTTP, zoals GET, POST, PUT, DELETE en PATCH.   
  Deze methodes staan ook wel bekend als CRUD (Copy, Read, Update, Delete), hoewel dat eigenlijk slaat op de bewerkingen die je kunt doen op een database.

Uiteindelijk is dat ook wel ons doel met een HTTP-request, om iets te doen in de database, maar het is wel belangrijk om te realiseren dat een HTTP-methode geen database-methode is.

* **URI’s** (**U**niform **R**esource **I**dentifiers): URI’s zijn de grammatica van een REST applicatie.   
  URI’s worden opgebouwd uit de verschillende elementen, waaronder een HTTP-methode en de naam van een resource, om de juiste resource te kunnen bereiken.   
  URI's worden gebruikt om de resources te identificeren.   
  Elke resource in de REST-API heeft een unieke URI die deze identificeert.   
  De URI kan worden gebruikt om de resource op te halen, te updaten of te verwijderen.
* **HTTP-Statuscode**s: HTTP-statuscodes worden gebruikt om de status van de operatie aan te geven. Er zijn verschillende statuscodes beschikbaar in HTTP:

Hier zijn enkele veelvoorkomende HTTP-statuscodes en hun betekenis:

* + 200 OK: Het verzoek is succesvol afgehandeld en er wordt een juist antwoord teruggegeven.
  + 201 Created: Het verzoek is succesvol afgehandeld en er is een nieuwe bron gemaakt.
  + 400 Bad Request: Het verzoek kon niet worden begrepen vanwege een foutieve syntaxis of andere clientfouten.
  + 401 Unauthorized: De client is niet geautoriseerd om toegang te krijgen tot de gevraagde bron. Authentificatie kan vereist zijn.
  + 403 Forbidden: De client heeft geen toestemming om toegang te krijgen tot de gevraagde bron, zelfs na authenticatie.
  + 404 Not Found: De gevraagde bron kon niet worden gevonden op de server.
  + 500 Internal Server Error: Er is een onverwachte fout opgetreden aan de serverzijde, waardoor het verzoek niet kon worden voltooid.
* Deze statuscodes worden gebruikt om de API-gebruiker te informeren over het resultaat van de operatie en worden meegegeven als return waarde aan een request. Wanneer de statuscode niet expliciet gezet wordt, vult Spring het in als *200 OK*.
* **Headers**: HTTP-headers worden gebruikt om extra informatie te verstrekken over de request of response. Ze kunnen worden gebruikt om verschillende dingen te specificeren, zoals de geaccepteerde content-types, de gebruikte encoding, de lengte van de response enzovoort.

##### Een voorbeeld

Een RESTful webservice bestaat dus uit een combinatie van HTTP methodes, de HTTP request URL en de HTTP status code.

Stel dat een webservice is geïmplementeerd op een server met het adres http://mijnwebservice.nl. Via deze webservice is de collectie books beschikbaar. Door de volgende endpoints aan het adres toe te voegen en de juiste HTTP methode te gebruiken kunnen de volgende acties worden

uitgevoerd:

* **GET /books** om alle book items in de collectie op te vragen
* **GET /books/2323** om één specifiek boek (met id 2323) op te vragen.
* **POST /books** om een boek toe te voegen aan de collectie. De gegevens worden dan meegegeven in de body van het request.
* **DELETE /books/2323** om een specifiek boek uit de collectie te verwijderen
* **PUT /books/2323** om de gegevens van een specifiek boek aan te passen. De gegevens worden dan meegegeven in de body van het request.

De PATCH methode wordt ook wel gebruikt in een RESTful webservice om aan te geven dat een item moet worden bijgewerkt.

##### Een tweede voorbeeld

Stel dat we een webservice hebben voor het beheer van gebruikersgegevens. We kunnen de volgende endpoints en bijbehorende HTTP-methoden definiëren:

1. GET /users: Hiermee kunnen we alle gebruikersgegevens opvragen.
2. GET /users/{id}: Hiermee kunnen we de gegevens van een specifieke gebruiker opvragen door zijn/haar ID mee te geven in de URL.
3. POST /users: Hiermee kunnen we een nieuwe gebruiker toevoegen door de vereiste gegevens mee te sturen in de body van het request.
4. PUT /users/{id}: Hiermee kunnen we de gegevens van een specifieke gebruiker bijwerken door zijn/haar ID mee te geven in de URL en de bijgewerkte gegevens in de body van het request.
5. DELETE /users/{id}: Hiermee kunnen we een specifieke gebruiker verwijderen door zijn/haar ID mee te geven in de URL.

Bijvoorbeeld:

* GET /users: Geeft alle gebruikersgegevens terug.
* GET /users/123: Geeft de gegevens van gebruiker met ID 123 terug.
* POST /users: Voegt een nieuwe gebruiker toe met de verstrekte gegevens.
* PUT /users/123: Werkt de gegevens van gebruiker met ID 123 bij met de verstrekte gegevens.
* DELETE /users/123: Verwijdert de gebruiker met ID 123 uit de gegevensset.

#### Mapping

Afbeelding met tekst, schermopname, Lettertype, nummer

Automatisch gegenereerde beschrijving

Voorbeeld: books

Met een RESTful webservice willen we eigenlijk alle CRUD-operaties op de resource kunnen uitvoeren. Dat kan door gebruik te maken met de verschillende HTTP- methodes in combinatie met verschillende endpoints.

##### Get request

Een GET-request mag nooit een verandering veroorzaken in de resource en wordt daarom ook wel *veilig* genoemd.   
Bij het doen van een GET-request wordt er dus data opgevraagd bij de server. De server gaat in de database opzoek naar de data en stuurt dit terug in een HTTP-response.

Het return-type van de methode die alle boeken opvraagt, verschilt natuurlijk per implementatie. In ons geval zouden we een String, een List<Book> of een ResponseEntity<Object> kunnen teruggeven.

Spring Boot vertaalt de return-value van de methode naar JSON en plaatst dit in de body van de response. We kunnen immers alleen JSON of XML over het internet verzenden.

@RestController

public class BooksController {

    @GetMapping("/book")

    public ResponseEntity<String> getBook() {

        return ResponseEntity.ok(“Novi”);

    }

}

Met behulp van de library Jackson wordt de return-value omgezet naar JSON. Naast de data die erin staat, wordt iedere response voorzien van een status code die laat zien of het request succesvol is afgehandeld. Bij een GET-request wordt er een 200 OK status code teruggegeven als de operatie is gelukt.

Afhankelijk van de reden waarom een request niet is geslaagd, kan een andere status code worden teruggegeven.

##### **POST-request**

Het POST-request wordt gebruikt om een item *toe te voegen* aan de collectie. Denk bijvoorbeeld aan het toevoegen van een boek aan de bibliotheek.  
Hiervoor wordt er dus data *verstuurd* naar de server. De server voegt het item toe, maar stuurt geen data terug,

Wel wordt er in de response-header een endpoint meegegeven waarmee het nieuwe item weer kan worden opgehaald. Deze kun je vinden onder de veldnaam “location”.

De status code die wordt teruggegeven als het toevoegen van het item is gelukt, is een 201 Created code.

@PostMapping("/book")

public ResponseEntity<Object> addBook(@RequestBody String title) {

   …(hier wordt iets opgeslagen in de database)...

    String name = “NOVI”

    URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{name}")

        .buildAndExpand(name).toUri();

    return ResponseEntity.created(location).build();

}

##### **DELETE-request**

Een DELETE-request geeft aan dat het item uit de collectie moet worden verwijderd. Bij het doen van een DELETE-request wordt er geen data meegestuurd naar de server. Via het endpoint, waarin bijvoorbeeld de ID van het item wordt gespecificeerd, wordt al duidelijk welke item er precies verwijderd moet worden.

@DeleteMapping("/books/{id}")

public ResponseEntity<Object> deleteBook(@PathVariable int id) {

    …(hier wordt een naam verwijderd uit de database)...

    return ResponseEntity.noContent();

}

##### **PUT-request**

Een PUT-request geeft aan dat het item uit de collectie moet worden aangepast of vervangen. De gegevens van het nieuwe item worden, net als bij een POST-request, *verstuurd* naar de server. De server wijzigt daarmee het item en stuurt een status-code terug. Bij success is dat standaard 204 No Content.

@PutMapping("books/{id}")

public ResponseEntity<Object> updateBook(@PathVariable int id, @RequestBody String bookTitle) {

    …(hier wordt een resource aangepast)

    return ResponseEntity.noContent();

}

##### PATCH-request

De @PatchMapping-annotatie wordt gebruikt om HTTP PATCH-verzoeken af te handelen in specifieke handlermethoden in je controllerklasse. Het stelt je in staat om gedeeltelijke updates van een bron te verwerken door alleen de velden op te geven die moeten worden gewijzigd, in plaats van de volledige representatie van de bron te verzenden.

Hier is een voorbeeld van hoe @PatchMapping kan worden gebruikt:

java

@RestController

@RequestMapping("/gebruikers")

public class GebruikerController {

@Autowired

private GebruikerService gebruikerService;

@PatchMapping("/{id}")

public ResponseEntity<Gebruiker> updateGebruiker(@PathVariable Long id, @RequestBody GebruikerPatchDto patchDto) {

Gebruiker gebruiker = gebruikerService.findById(id);

if (gebruiker == null) {

return ResponseEntity.notFound().build();

}

// Pas gedeeltelijke updates toe op het gebruikersobject

if (patchDto.getVoornaam() != null) {

gebruiker.setVoornaam(patchDto.getVoornaam());

}

if (patchDto.getAchternaam() != null) {

gebruiker.setAchternaam(patchDto.getAchternaam());

}

// ... verwerk andere velden

Gebruiker bijgewerkteGebruiker = gebruikerService.update(gebruiker);

return ResponseEntity.ok(bijgewerkteGebruiker);

}

}

##### Request parameters en responses

We hebben drie verschillende methoden om input mee te geven aan een request:

###### 1. PathVariable Met behulp van een PathVariable. Het komt vaak genoeg voor dat we slechts één boek of één blogpost willen ophalen in plaats van alle items uit de collectie. Zo'n item heeft altijd een id: een sleutelveld die uniek is voor het item binnen de collectie van items in de resource. Vaak is dit een nummer, maar het kan ook een String zijn. In de request-mapping wordt zo'n id aangegeven als {id} om aan te geven dat deze vervangen kan worden door de letterlijke waarde van de ID. Een GET-request dat gebruik maakt van een pathVariable zal er dan zo uit zien:

@GetMapping("/books/{id}")

public ResponseEntity<Object> getBook(@PathVariable long id) {

    return ResponseEntity.ok(...);

}

###### **2. RequestParameter**

wanneer we niet precies weten naar welke ID's we op zoek zijn, kan het zijn dat we een zoekcriterium mee willen geven.We willen niet alle items uit de collectie, maar alleen die items die voldoen aan een opgegeven criterium.

een **query string**: een string die je aan het eind van het endpoint mag toevoegen door er een ? aan toe te voegen.

Denk aan /books?title=harry of /books?writer=rowling. Je mag ze zelfs aan elkaar plakken door het & teken te gebruiken: /books?title=harry&writer=rowling.

In de Controller-methode kunnen deze extra velden worden afgevangen met de @RequestParam annotatie. De Controller-methode krijgt argumenten mee die qua naamgeving overeen komen met de velden in de querystring:

@GetMapping("/books")

public ResponseEntity<Object> getAllBooks(@RequestParam String title) {

    return ...;

}

###### **3. RequestBody**

dit wordt gebruikt voor POST-requests en PUT-requests om daadwerkelijk data mee te sturen. Deze annotatie stelt je dus in staat om complete objecten mee te sturen in een HTTP request. Deze meegestuurde objecten kun je vervolgens gebruiken om op te slaan, of om bestaande data aan te passen.

##### **Responses voor bad requests**

Tot nu toe zijn we ervan uit gegaan dat de RESTful request is geslaagd en dat er een *2xx* status code wordt teruggegeven. Echter, het kan ook gebeuren dat een request niet slaagt.

Dit verzorgen we in de backend doormiddel van **exceptions**. Het is dan wenselijk eigen exceptions klassen te definiëren. Als zich een onwenselijke situatie voordoet, kan een exception worden gegeneerd.

Spring Boot heeft een annotatie @ControllerAdvice waarmee een speciale Exception-Controller kan

worden aangegeven. Een exception eindigt dan in een controller-methode die als response de juiste status code teruggeeft.

public class RecordNotFoundException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public RecordNotFoundException() {

        super();

    }

    public RecordNotFoundException(String message) {

        super(message);

    }

}

@ControllerAdvice

public class ExceptionController {

    @ExceptionHandler(value = RecordNotFoundException.class)

    public ResponseEntity<Object> exception(RecordNotFoundException exception) {

        return new ResponseEntity<>(exception.getMessage(), HttpStatus.NOT\_FOUND);

    }

}

if (...) { // determine if id exists

    throw new RecordNotFoundException('ID cannot be found');

}

bijvoorbeeld een niet bestaande ID meegegeven, of je bent als gebruiker niet geautoriseerd om de operatie uit te voeren. In die gevallen wordt een *4xx* status code teruggegeven.

##### REST(REpresentational State Transfer) API

###### **RESTful-endpoints** of RESTful webservices

Vaak praten we over **RESTful-endpoints** of RESTful webservices wanneer we het over API’s hebben, een set principes waaraan voldaan moet worden.

voor een API zichzelf een REST-API mag noemen. Dit omvat:

* **Client/server-architectuur:** de clients van de API gebruiken HTTP-requests om data op de vragen of data naar de server te sturen.
* **Stateless**: de server houdt geen verbinding in stand en slaat geen informatie op tussen requests van dezelfde client. Ieder request wordt behandeld als een apart request.
* **Cacheable**: de server maakt caching van vaak gevraagde data mogelijk om serverbelasting te verminderen.
* **Uniforme interface**: er is één gedefinieerde manier waarop een client met de server communiceert, onafhankelijk van het apparaat of de applicatie.
* **Resource-gebaseerd:** de API moet voor elke bron een specifiek endpoint hebben.