Ontwerp en implementatie van web-API's

Practicum Internettechnologie

Academiejaar 2014–2015

1 Web-API's volgens de REST architecturale stijl

In dit practicum zal je een web-API ontwerpen die kan gebruikt worden als *webtoepassing* in een browser en tegelijkertijd ook als *interface* voor andere toepassingen. Voor we hieraan beginnen, moeten we ons eerst afvragen wat een web-API precies is en hoe die verschilt van andere soorten API's. We bespreken hier in het bijzonder web-API's volgens de **REpresentational State Transfer (REST)** architecturale stijl.

1.1 Web-API's bestaan uit resources

Een Application Programming Interface of API biedt – zoals de naam zegt – elementen aan waarmee een toepassing kan ontwikkeld worden. In de Java-context ben je dit al verschillende keren tegengekomen. Bijvoorbeeld, de Swing-bibliotheek laat toe om grafische toepassingen te ontwikkelen en biedt daarvoor een interface aan die bestaat uit klassen zoals `JButton` en `JLabel`. Het woord "klasse" toont aan dat een API sterk verweven is met de onderliggende omgeving. Als we bijvoorbeeld kijken naar de Linux kernel API, zullen we zien dat deze is opgedeeld in methodes, aangezien de methodegebaseerde programeertaal C daar domineert.

Ook het web heeft dus een eigen type API's. De essentiële bouwstenen van web-API's zijn **resources**. Een resource is een constante, *conceptuele* relatie waarvan de waarde mogelijk varieert met de tijd. Hieronder vind je voorbeelden van dergelijke resources:

- de voorpagina van de krant *De Standaard*
- het weer van vandaag in Gent
- het weer op 28 februari in Gent
- *iTunes* versie 11.1.3
- de laatste versie van *iTunes*

Merk op dat al deze resources *verschillend* zijn. Zelfs al zou vandaag 28 februari zijn, dan nog is "het weer van vandaag" een andere resource dan "het weer op 28 februari". Dat komt omdat een resource gedefinieerd is door haar conceptuele relatie, niet door de waarde van die relatie op een bepaald moment.

Resources op het web worden **uniek geïdentificeerd** door HTTP-URL's. Dit wil zeggen dat één URL ten hoogste één resource identificeert, maar meerdere URL's kunnen dezelfde resource identificeren. Hieronder vind je enkele voorbeelden van URL's die "het weer van vandaag in Gent" zouden kunnen identificeren:

- *http://example.org/weather/ghent/today*
- *http://gent.be/weer/vandaag*
- *https://weather.com/forecasts?city=Ghent&date=today*

Essentieel is het inzicht dat web-API's *niet* bestaan uit methodes of klassen; het is dan ook *niet* de bedoeling om methodes of klassen na te bootsen over het web. In de praktijk gebeurt dit echter nog vaak: je kan deze API's herkennen aan URL's die methodenamen bevatten zoals `showProduct` of `addComment`. Dergelijke API's noemen we "HTTP API's" omdat ze wel HTTP als protocol gebruiken, maar niet de informatie-architectuur van het web. De API die we in dit practicum zullen bouwen, is een web-API volgens de REST architecturale stijl.

1.2 Resources worden voorgesteld als representaties

De conceptuele aard van resources zorgt meteen voor een probleem: hoe kunnen we iets abstracts als "het weer van vandaag in Gent" doorsturen over een internetverbinding? Uiteraard is dit onmogelijk; in plaats daarvan stellen we het weer voor als een HTML- of JSON-document, die we **representaties** van de resource noemen. Figuur 1 toont een mogelijke HTML-representatie van het weer van vandaag; figuren 2 en 3 tonen andere representaties van dezelfde resource. Telkens wordt het MIME-type vermeld.

```html
<div id="main">
  <h1>The weather in Ghent</h1>
  <h2>Today</h2>
  <dl>
    <dt>temperature</dt>
    <dd>20&deg;C</dd>
    <dt>wind speed</dt>
    <dd>5km/h</dd>
  </dl>
  <p><a href="/weather/ghent/tomorrow">Tomorrow's weather</a></p>
</div>
```

Figuur 1: Gedeelte van een HTML-representatie van "het weer van vandaag in Gent" (`text/html`)

```json
{
  "weather": {
    "title": "Today",
    "temp": 20,
    "wind": 5
  },
  "related": [{
    "title": "Tomorrow",
    "href": "/weather/ghent/tomorrow"
  }]
}
```

Figuur 2: JSON-representatie van "het weer van vandaag in Gent" (`application/json`)

```
Vandaag is het licht bewolkt, 20 graden,
met kans op regen in de namiddag.
```

Figuur 3: Tekstuele representatie van "het weer van vandaag in Gent" (`text/plain`)

De reden dat één resource meerdere representaties kan hebben, is dat verschillende cliënten hierdoor van dezelfde resource gebruik kunnen maken. Als mensen in hun browser de URL *http://example.org/weather/ghent/today* intikken, krijgen ze een gerenderde versie van de HTML-representatie, waarmee ze dan kunnen interageren. Als een webtoepassing in JavaScript *dezelfde* URL opvraagt, krijgt deze een JSON-versie terug voor verder gebruik. De tekstuele representatie kan bijvoorbeeld handig zijn voor gebruik in de opdrachtprompt, die geen grafische capaciteiten heeft. Opnieuw is *dezelfde* URL geschikt om deze representatie op te vragen. Het mechanisme dat dit mogelijk maakt heet **content negotiation**: iedere cliënt geeft aan welke representaties het beste geschikt zijn, en de server probeert deze voorkeur te respecteren door te antwoorden met de best passende representatie die ondersteund wordt. Dit is een belangrijk verschil tussen web-API's en andere API's, waarbij het uitgewisselde formaat typisch onveranderlijk blijft.

Het is cruciaal om te begrijpen dat een web-API *niet* verschilt van de "website" die je bezoekt in een browser. Met andere woorden: de website *is* de API. Hoewel we in de praktijk vaak gescheiden toegang zien voor mensen en geautomatiseerde cliënten, bestaat hiervoor geen enkele technische noodzaak. In dit practicum zullen we daarom een web-API ontwerpen met één enkele interface die werkt voor zowel mensen (via HTML) als geautomatiseerde cliënten (via JSON).

1.3 Resources worden bewerkt met zelf-beschrijvende boodschappen

Web-API's volgens de REST architecturale stijl gebruiken het HTTP-protocol op een manier waardoor iedere boodschap op zichzelf kan geïnterpreteerd worden. Langs de ene kant betekent dit dat we HTTP **stateless** gebruiken: iedere boodschap is onafhankelijk van de vorige. Neem bijvoorbeeld het klassieke scenario waarin we zoekresultaten opvragen voor het sleutelwoord "technologie". We sturen dan een verzoek naar de server: "geef me de eerste pagina zoekresultaten voor 'technologie'". Als we ook de tweede pagina willen bekijken, dan sturen we *niet* "geef me pagina 2", want deze boodschap is betekenisloos zonder de vorige. In plaats daarvan sturen we "geef me de tweede pagina zoekresultaten voor 'technologie'". Hierdoor hoeft de server niet te onthouden wat onze vorige zoekopdracht was of kan zelfs een andere server ons tweede verzoek behandelen.

Langs de andere kant betekenen zelf-beschrijvende boodschappen dat we enkel **vooraf gedefinieerde methodes** gebruiken. In Java ben je vrij om zelf methodenamen en hun semantiek te kiezen, bijvoorbeeld `calculateHeight` of `createField`. Deze namen hebben echter enkel betekenis voor mensen; voor machines zijn dit gewoon tekenreeksen die variëren per toepassing. Het HTTP-protocol definieert een beperkte set methodes zoals `GET`, `PUT`, `POST` en `DELETE`, waarvan de betekenis vastligt. Elk van deze methoden neemt een URL als argument, en sommigen ook een representatie van een resource. Hun betekenis is als volgt.

`GET` */books/23*	ontvang een representatie van de resource met URL */books/23*
`DELETE` */books/23*	verwijder de resource met URL */books/23*
`PUT` */books/23*	plaats de gegeven resource op de URL */books/23*
`POST` */books/23*	voer een actie uit met de gegeven resource op de resource met URL */books/23*

Zoals je merkt, is de `POST`-methode het vaagst gedefinieerd, en dit laat toe om alle acties te voorzien die niet onder de andere methodes vallen. Het is jouw taak als web-API-ontwerper om te definiëren wat de `POST`-methode betekent voor iedere resource. In het geval van */books/23* zou dit bijvoorbeeld het toevoegen van een bericht kunnen zijn, of het plaatsen van een item in een virtueel winkelmandje.

Het is van het grootste belang dat je steeds de HTTP-methodes correct gebruikt, en dat heeft alles te maken met correct resource-ontwerp. Denk niet "`viewBook`" maar denk "`GET /books/23`". Dat geldt ook voor resources die je meestuurt als argument met `POST` of `PUT`. Denk niet "`POST /users/stefan` met body `addFriend`" maar denk "`POST /users/stefan/friendships` met body `/users/sara`". Dit toont het verschil tussen methode-georiënteerd *(een vriend toevoegen)* en resource-geörienteerd ontwerp *(een vriendschap-resource aanmaken)*.

1.4 De toepassing wordt bestuurd via hypermedia

De laatste unieke eigenschap van de REST architecturale stijl is het gebruik van hypermediabesturingselementen om de staat van de toepassing te veranderen. HTML is de HyperText Markup Language, en bijzonder aan hypermedia is dat hyperlinks en formulieren gebruikt worden om van één resource naar een andere te navigeren. Dankzij dit mechanisme is het inderdaad mogelijk om *stateless* boodschappen te versturen: als de server de resource "zoekresultaten voor 'technologie'" genereert, zal deze in de HTML-representatie al de links plaatsen naar de tweede en derde pagina. Op die manier hoef jij als gebruiker niet telkens het volledige verzoek op te stellen, aangezien je gewoon deze links kan activeren. Dat betekent ook dat je als cliënt niet op de hoogte hoeft te zijn van hoe de URL- of resource-structuur van de server eruit ziet. De server hoeft ook niet te onthouden op welke pagina je gebleven bent. Op geen enkel moment hoef je de URL in de adresbalk manueel aan te passen – tenzij je uiteraard naar een andere website wil springen.

Vermits eenzelfde resource verschillende representaties kan aannemen voor verschillende cliënten, is het noodzakelijk om voor elk representatieformaat na te gaan wat de hypermediamogelijkheden zijn. Momenteel is het vaak zo dat enkel in de representaties voor mensen (HTML) hypermediabesturingselementen voorzien worden. Dat betekent dat de URL's in automatische toepassingen hardgecodeerd moeten zijn. Meer flexibele toepassingen worden echter mogelijk als we ook hyperlinks aanbieden aan deze automatische toepassingen: op die manier kunnen deze gewoon links volgen zoals wij dit doen, in plaats van ieder verzoek opnieuw te construeren. Een voorbeeld vind je in figuur 2, waar links opgenomen werden in de JSON-representatie. Een toepassing die deze representatie ontvangt, kan de link naar het weer van morgen volgen, zonder dat kennis nodig is over de URL-structuur van de server. Dit is opnieuw typisch aan REST web-API's.

2 Het Ruby on Rails-raamwerk voor webontwikkeling

Voor het ontwikkelen van een webtoepassing zullen we gebruikmaken van het Ruby on Rails-raamwerk voor de programmeertaal Ruby. Dit laat toe om op een efficiënte manier webtoepassingen aan te maken en uit te breiden, en maakt het makkelijker om de REST architecturale stijl te volgen.

2.1 De programmeertaal Ruby

We kiezen voor Ruby in plaats van Java, omdat Ruby een **dynamische programmeertaal** is. Voor dit concept bestaan verschillende definities, maar in het geval van Ruby betekent dit *geïnterpreteerd* en *dynamisch getypeerd*. Java daarentegen wordt gecompileerd naar bytecode en heeft een typesysteem dat geverifieerd wordt tijdens de compilatie. Dynamische talen zijn vaak handiger voor webontwikkeling om verschillende redenen:

- Je kan sneller aanpassingen maken aangezien geen compilatie nodig is. Bij een webapplicatie ben je vaak met verschillende soorten bronmaterialen bezig (programmacode, HTML, CSS, ...) en ook met verschillende programma's (ontwikkelomgeving, server, één of meerdere browsers). Een dynamische taal laat je toe om een kleine aanpassing te maken zonder hercompilatie of herstart van programma's.
- Dynamische talen bieden doorgaans syntactische ondersteuning voor flexibelere datastructuren zoals arrays en maps. Deze zijn zeer geschikt voor fijne databewerkingen, die vaak nodig zijn in webcode. Daarnaast kan je ook snel lokale wijzigingen aanbrengen in deze structuren, zonder dat je dit op voorhand moet ontwerpen, wat meestal wel het geval is bij gecompileerde datastructuren.
- De rijkere syntax van dynamische talen elimineert vaak de noodzaak voor externe configuratiebestanden, aangezien de programmeertaal zelf toelaat om configuratieparameters efficiënt te beschrijven.

Deze opgave bevat geen overzicht van de programmeertaal Ruby. In plaats daarvan raden we sterk aan om de interactieve tutorial te volgen op *http://tryruby.org/*. Deze bestaat uit 7 niveaus en introduceert de belangrijke concepten in Ruby: datatypes, arrays, hashes, methodes, objecten en klassen. Een alternatief of aanvulling vind je in de interactieve les *Ruby Primer* op *https://rubymonk.com/*, die iets uitdagender is. Spendeer hier echter niet te veel tijd aan: Ruby is een vrij eenvoudige taal die je snel genoeg zal oppikken.

2.2 Ruby on Rails

Ruby en Ruby on Rails worden vaak in één adem genoemd, maar het zijn twee verschillende dingen. Ruby is de programmeertaal, en Ruby on Rails is een **raamwerk om webtoepassingen te ontwikkelen** in Ruby. Alle Rails-toepassingen volgen het Model-View-Controller-patroon (MVC), zoals je dit ook kent in de context van grafische gebruikerstoepassingen. In Rails ziet dit er meestal uit als volgt:

- Een **model** bestaat uit een combinatie van een datatype-definitie (een lijst met velden en types) en een klasse die het gedrag in programmacode bevat. In REST-termen zou je kunnen stellen dat een model overeenkomt met een resource-type; een instantiatie van een model is een resource.
  Voorbeeld – Een blogpost kan gedefinieerd worden door de velden `title` (`string`), `body` (`text`), en `published` (`boolean`), en een klasse die vereist dat `body` tenminste 300 woorden moet bevatten vooraleer de blogpost gepubliceerd mag worden.
- Een **view** zet één of meerdere modellen om in een serializeerbaar formaat, zoals HTML of JSON. Views worden in Rails gegenereerd aan de hand van sjablonen en/of programmacode. Dit komt overeen met een representatie in de REST-terminologie.
  Voorbeeld – een HTML-representatie van een lijst van blogposts wordt gegenereerd door alle titels op te vragen en deze vervolgens in een HTML-sjabloon te plaatsen.
- Een **controller** voert bewerkingen uit op modellen als antwoord op een HTTP-verzoek. Controllers zorgen ervoor dat de HTTP-methodes het gewenste effect hebben voor een bepaald resource-type.
  Voorbeeld – Als antwoord op een verzoek "`POST /articles`" maakt een controller een nieuwe blogpost aan en plaatst een statusbericht op de pagina van de auteur.

Tijdens de ontwikkeling van een Rails-project wissel je doorgaans in hoog tempo af tussen programmeren aan modellen en controllers, en het ontwerpen van views. Hierdoor zal je snel de voordelen merken van een dynamische programmeertaal.

3 Opgaven

Voor dit practicum dien je opgaven 1, 2, en 3 hieronder elk *afzonderlijk* in als volgt:

- Maak een **zip-bestand** aan waarin de hoofdmap van je project zit. Deze bevat submappen zoals `app` en `db`.
- De **bestandsnaam** is `itech-p1-g<groep>-o<opgave>.zip`. Bijvoorbeeld: `itech-p1-g05-o0.zip`.
- Zorg ervoor dat de **databank** in elk zip-bestand gevuld is met **voldoende voorbeelden**.

Hou ook steeds in het achterhoofd dat scores doorgaans omgekeerd evenredig zijn met de benodigde verbetertijd. Besteed dus voldoende aandacht aan duidelijke code en commentaar, en respecteer de bovenstaande afspraken. Je oplossingen zullen getest worden op Athena; zorg er dus zeker voor dat alles daar werkt.

3.0 Kennismaking met Ruby on Rails

Je kan aan de slag met Ruby on Rails via Aptana Studio op Athena. Dit is een gratis softwarepakket, dus je kan het ook op je eigen computer installeren (zie *http://www.aptana.com/products/studio3* en *http://railsinstaller.org/*).

Voor deze opgave bouw je een blogtoepassing aan de hand van de handleiding "Building a Rails blog application", die beschikbaar is op Minerva. Deze gidst je stap voor stap door Ruby on Rails aan de hand van een voorbeeld. Het is de bedoeling om alle stappen te volgen tot je een volledig werkende toepassing hebt.

Alle instructies uit de handleiding kan je naar keuze uitvoeren met de grafische interface van Aptana Studio, of via de opdrachtprompt die dit pakket aanbiedt. Hoewel de code van dit deel niet gebruikt wordt in de volgende delen, is het belangrijk om te leren hoe je een Rails-toepassing op correcte wijze bouwt. Deze opgave hoef je niet in te dienen.

3.1 Een webtoepassing voor evenementen implementeren

In dit tweede deel bouwen we een sociale toepassing waarin gebruikers evenementen kunnen aanmaken. Ieder **evenement** bestaat uit een *titel*, een *beschrijving* en een *begin-* en *eindtijdstip*. Iedere **persoon** heeft een *e-mailadres*, een *naam* en een *geboortedatum*. Op de pagina van ieder evenement kunnen **berichten** geplaatst worden met een *tekst*, *persoon* en *datum*. Het moet mogelijk zijn om evenementen, personen en berichten toe te voegen, te bewerken en te verwijderen. Op de pagina van een evenement kunnen we de lijst van aanwezige personen zien, en kunnen we personen aan die lijst toevoegen of ervan verwijderen. De hoofdpagina geeft toegang tot de lijst van evenementen en de lijst van personen.

Het is *niet* de bedoeling om gebruikersaccounts te implementeren. We veronderstellen centrale toegang om evenementen en gebruikers te bewerken, en berichten te plaatsen in naam van andere gebruikers.

3.2 De webtoepassing uitbreiden met JSON-representaties

In de vorige opgave heb je HTML-representaties gemaakt die kunnen bediend worden door mensen via browsers. Voor deze opgave voeg je JSON-representaties toe die door machines kunnen geconsumeerd worden.

Let erop dat alle data beschikbaar is in deze representaties, en zorg er ook voor dat de links beschikbaar zijn, zoals we besproken hebben in sectie 1.4. Figuur 2 toont een voorbeeld, maar je mag de structuur zelf kiezen. Je kan JSON-representaties testen met de toepassing *curl* of via *http://onlinecurl.com/*. Gebruik dezelfde URL als voor HTML, maar stel de optie `--header` in op "`Accept: application/json`".

3.3 Een zoekfunctie implementeren met JSON-representaties

Tot slot zullen we de JSON-representaties die je maakte gebruiken om een zoekfunctie aan de cliëntzijde te bouwen. Je kan deze zoekfunctie integreren in de hoofdpagina of een aparte zoekpagina maken.

Op deze pagina implementeer je in JavaScript een script dat de JSON-representatie van de evenementen ophaalt en weergeeft aan de gebruiker. De gebruiker kan woorden van de titel ingeven om zo de beschikbare evenementen te filteren. Je mag het jQuery-raamwerk (*http://jquery.com/*, inbegrepen in Rails) gebruiken om de JavaScript-code te vereenvoudigen.

Maak in je implementatie gebruik van de links die aanwezig zijn in je JSON-representatie. Een gebruiker moet kunnen klikken op de resultaten om terecht te komen op de pagina van het evenement.