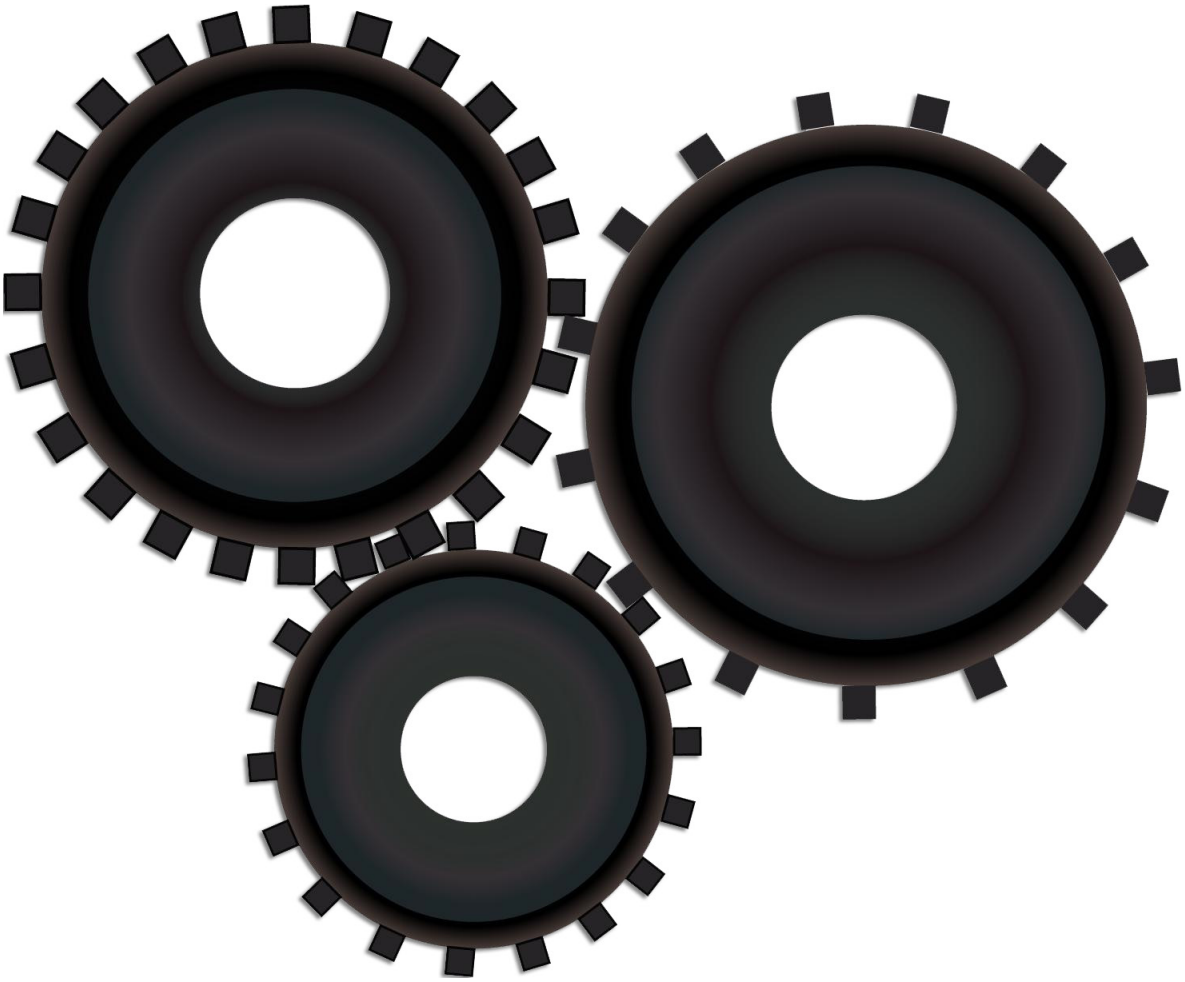
REST API Skeleton

De theorie en ontwikkeling



ibuildings
WEB & MOBILE APP DEVELOPMENT

Tim van Beek | 1568270 | Hogeschool Utrecht

REST API Skeleton

Afstudeerscriptie

Tim van Beek

1568270

Vlissingen, 1 juni 2012

Ibuildings

Edwin de Vries

Sandy Pleyte

Hogeschool Utrecht

Ronald van Essen

Digitale Communicatie

Voorwoord

De populariteit van REST is de laatste jaren zeer gegroeid. Het wordt veel toegepast voor de communicatie tussen servers en clients. Veel mobiele applicaties en websites maken dan ook gebruik van dit principe. Dit met als gevolg dat bedrijven als Ibuildings de toepassing hiervan willen benutten.

Voor u ligt de scriptie “REST API Skeleton: De theorie en ontwikkeling”. In dit document staan de resultaten van het traject dat is doorlopen om een REST API te ontwikkelen. Deze scriptie is geschreven in het kader van mijn afstuderen voor de opleiding Digitale Communicatie aan de Hogeschool Utrecht. Met het afronden hiervan komt er een einde aan de drie jaar die ik met deze opleiding bezig ben geweest.

De opdracht is verricht gedurende de periode van februari 2012 tot en met juni 2012 voor het bedrijf Ibuildings. De begeleiding vanuit Ibuildings werd verzorgd door Edwin de Vries (projectmanager) en Sandy Pleyte (lead developer). Ik wil hen hartelijk danken voor de tijd en energie die zij voor mij vrij hebben gemaakt. Zonder hun begeleiding had ik niet tot dit resultaat kunnen komen.

Daarnaast gaat mijn dank uit naar alle medewerkers van Ibuildings die mij hebben ondersteund bij de uitvoering van deze opdracht. Ross Tuck (lead developer bij Ibuildings Utrecht) met wie ik de gehele periode intensief heb samengewerkt, dank ik hierbij in het bijzonder.

Vlissingen, juni 2012

Tim van Beek

Samenvatting

Achtergrond
In deze scriptie wordt beschreven op wat voor manier een REST API kan worden ontwikkeld met het Zend Framework 1.1x. Deze scriptie is geschreven aan de hand van een praktijkopdracht die is uitgevoerd voor het bedrijf Ibuildings.

Ibuildings is een software ontwikkelbedrijf dat gespecialiseerd is in de ontwikkeling van web applicaties. Aangezien er tegenwoordig steeds meer gebruik wordt gemaakt van verschillende websites en mobiele platformen wil Ibuildings een client-server structuur gaan inzetten.

Om de communicatie vanaf de server richting clients mogelijk te maken gaat Ibuildings gebruik maken van REST. In dit document worden de resultaten van het onderzoek naar REST, HTTP en Zend beschreven. Er wordt hierbij antwoord gegeven op de vraag: *Hoe moet de REST API voor Ibuildings worden ontwikkeld en geïmplementeerd met het Zend Framework?* De uitkomst hiervan is toegepast tijdens de ontwikkeling van de REST API. Het doel van deze opdracht is een stabiele REST API te ontwikkelen die voldoet aan de eisen van Ibuildings.

Methode
Voor het uitvoeren van deze opdracht is gebruik gemaakt van literatuuronderzoek. Alle relevante literatuur is verzameld door middel van deskresearch. Daarnaast is er voor de ontwikkeling van de verschillende componenten gebruik gemaakt van Test Driven development. Dit is een onderdeel uit de Agile ontwikkelmethode Extreme Programming.

Het ging hierbij om een iteratief proces waarbij de ontwerp- en bouw fase in elkaar overliepen. Aan de hand van de resultaten uit het onderzoek werd een ontwerp opgesteld waar de Unit Tests weer op werden gebaseerd. Met gebruik van deze Unit Tests is vervolgens de functionaliteit in code ontwikkeld.

Resultaten
Veel ontwikkelaars interpreteren het idee achter REST op een verkeerde manier. REST wordt namelijk veel als architectuur gezien die gebruik maakt van URI's. Maar REST is in werkelijkheid een set van richtlijnen die voorschrijven hoe er met gebruik van HTTP via netwerken moet worden gecommuniceerd. Het gaat hierbij om de toepassing van HTTP en URI's binnen een applicatie.

Hoe goed de applicatie aan REST voldoet kan worden gemeten aan de hand van het Richardson Maturity Model (RMM). De requirements van Ibuildings zijn gebaseerd op het RMM level 2.

De componenten waaruit de REST API moet bestaan zijn een: route, request decoder, header parser, responsetype switcher, precondition helper en een serializer.

Het Zend framework 1.1x bevat volledige ondersteuning voor HTTP en URL's. Daarnaast wordt er gebruik gemaakt van een dispatch proces dat precies de uitvoer volgorde van componenten regelt.

Conclusie
Er kan worden gesteld dat de REST API moet voldoen aan de standaarden van HTTP. De onderdelen die hiervan moeten worden gebruikt zijn HTTP methodes, HTTP headers en URI's. Dit is gebaseerd op het RMM level 2 en vertegenwoordigt tevens de requirements waaraan de API voor Ibuildings moet voldoen.

Binnen de REST API is gebruik gemaakt van de specifieke fases die het dispatch proces van Zend Framework biedt. Het is namelijk zeer van belang dat de gestelde componenten op de juiste volgorde worden uitgevoerd.

Uit mijn onderzoek is naar voren gekomen dat bovengenoemde componenten essentieel zijn om de REST API volgens de richtlijnen te laten functioneren. Wanneer dit wordt toegepast dan kan de API in vrijwel iedere situatie worden ingezet.

Summary

Background
This thesis describes how to develop a REST API with the Zend Framework 1.1x. This document is written on basis of a practical assignment carried out for the Ibuildings company.

Ibuildings is a software development company specialized in web applications. These days it is common to use many different websites and mobile platforms. That's why Ibuildings wants to make use of an advanced client-server structure.

To enable the communication from the server towards clients Ibuildings is preparing to use REST. In this document the results from the research on REST, HTTP and Zend are described. The following question will be answered: *In which way does the REST API for Ibuildings needs to be developed and implemented with the Zend framework?* The results from this research are applied during the development of the REST API. The goal of this assignment is to develop a stable REST API that complies with the requirements of Ibuildings.

Method
During the research on these subjects literature study is done. All relevant literature is collected during desk research. Besides that Test Driven development is used to develop the various components. Test Driven development is a part of the Agile software development method Extreme Programming.

The complete development was an iterative process whereby the design and construction phases melted together. On basis of the results from the different researches a design was made. Based on this design many different Unit Tests where set up. With use of these Unit Tests the functionality in the code was developed.

Results
Many developers use a wrong interpretation of the idea behind REST. A lot of people see REST as an architecture that uses URI's, but REST is not an architecture. It is a set of guidelines that prescribe how to communicate with HTTP over networks. The focus here is on the use of HTTP and URI's inside an application.

The amount of "RESTfulness" of an application can be measured with the Richardson Maturity Model (RMM). The requirements of Ibuildings are based on the RMM level 2.

The components of which the REST API needs to consist are: route, request decoder, header parser, response type switcher, precondition helper and a serializer.

The Zend framework 1.1x contains full support for HTTP and URL's. Besides that Zend uses a dispatch process that manages the order of execution of different components.

Conclusion
It can be said that the REST API needs to meet the standards of HTTP. The parts it needs to use are HTTP methods, HTTP headers and URI's. This is based on the RMM level 2 en represents the requirements specified by Ibuildings.

Within the REST API the different phases from the Zend dispatch process are used. This enables the possibility to manage the order of execution from the different components.

My research has shown that the mentioned components are essential parts of the REST API to let it function according to the REST guidelines. When this is applied the API can be used in almost all circumstances.

Inhoud

1 Inleiding

Het bedrijf Ibuildings is in 1999 opgericht met als doel zich te specialiseren in het ontwikkelen van websites en web applicaties. Bij de start is ervoor gekozen om dit alleen te doen op basis van PHP-technologie. Vanwege deze duidelijke afbakening beschikt Ibuildings inmiddels over een grote hoeveelheid kennis. Met behulp van deze grote hoeveelheid kennis levert Ibuildings een brede variëteit aan producten en diensten. Ze verkopen en ontwikkelen software, verzorgen PHP trainingen en bieden technische consultancy. Ibuildings beschikt over ruim 50 Zend gecertificeerde software engineers en is officieel partner van Zend Technologies, het bedrijf achter PHP.

De afgelopen tien jaar heeft Ibuildings zich bezig gehouden met het ontwikkelen en uitbreiden van het PHP ATK Framework. Het ATK Application Framework is een geavanceerde ontwikkelomgeving waarmee ontwikkelaars snel en gemakkelijk applicaties kunnen bouwen. Er kan hierdoor meer worden gefocust op het implementeren van bedrijfsprocessen in plaats van het typen van PHP code. Met behulp van dit framework is het mogelijk om met enkele regels code een complete applicatie te schrijven waarbij de focus ligt op aanpasbaarheid en uitbreidbaarheid.

Een van de problemen waar zij nu tegen aanlopen is dat dit framework behoorlijk verouderd is. Onderdelen van ATK zijn de afgelopen jaren al een aantal keer vernieuwd maar omdat ATK al verschillende PHP versies meegaat zit er veel inefficiëntie in de globale structuur. Ook is er gebrek aan uniformiteit en documentatie. Het aanpassen van dit framework kost simpelweg teveel tijd om rendabel te kunnen zijn.

1.1 Onderwerp en aanleiding

Omdat het huidige ATK framework flink verouderd is, is er besloten om een skeleton* te ontwikkelen dat gebaseerd is op Sencha Ext JS 4. Daarnaast is het van belang dat er een skeleton wordt opgezet waarmee snel een REST API kan worden gebouwd. Deze REST API zal worden ontwikkeld met het PHP Zend Framework. Het is de bedoeling dat dit een soort verzameling wordt van componenten die medewerkers “out of the box” kunnen gebruiken om snel een applicatie in elkaar te zetten. De basis zal dus niet meer volledig in PHP worden ontwikkeld, Sencha Ext JS4 is namelijk gebaseerd op Javascript.

Deze nieuwe structuur wil Ibuildings in gaan zetten voor de ontwikkeling van grotere applicaties. Een project waar deze werkwijze op wordt toegepast is de vernieuwing van de verzekeringssite.nl.

Het idee achter deze nieuwe structuur is dat er aan de serverkant een uniforme REST interface bestaat waartegen requests uitgevoerd kunnen worden. Denk hierbij aan het ophalen van de tekst die op een website wordt ingeladen of een gebruiker die moet worden gevalideerd. Door deze benadering is het makkelijk om meerdere clients aan de server te koppelen. Voorbeelden hiervan zijn mobiele applicaties of een website.

De afstudeeropdracht wordt in opdracht van Edwin de Vries en Sandy Pleyte van Ibuildings uitgevoerd. Deze zal bestaan uit een aantal samenhangende onderdelen die samen een REST API vormen. Het eindproduct is gebaseerd op de richtlijnen die voor REST gelden.

1.2 Probleembeschrijving

Het is van belang om aan de serverkant van deze applicaties, een stabiele en uniforme REST service te hebben. Om hier voor te zorgen moet er een vooraf ingerichte basis bestaan waarmee snel een standaard REST service volgens de eisen van Ibuildings kan worden ingericht. Het zal hier moeten gaan om een skeleton waarin de verschillende onderdelen van de REST API zijn toegevoegd. Deze structuur moet de uniformiteit binnen applicaties stimuleren en ervoor zorgen dat applicaties stabiel functioneren.

* Een framework dat veelgebruikte componenten bevat die out of the box gebruikt kunnen worden

1.3 Doelstelling

1.3.1 Doelstelling Ibuildings

In de situatie die Ibuildings wenst, is het mogelijk om aan de hand van componenten die standaard functionaliteiten bevatten snel een Rich Internet Application te bouwen. Deze applicatie moet uit twee delen bestaan. Een client-side en een server-side. Het moet hierbij niet uitmaken met welke taal de client-side is opgebouwd.

De server-side zal worden opgezet met het PHP Zend 1.1x Framework. Het zal hier gaan om een REST API skeleton die gemakkelijk kan worden aangepast en waarmee snel een RESTful web service kan worden opgezet.

De doelstelling wordt als volgt geformuleerd:

- *Het ontwikkelen van verschillende componenten voor de REST API skeleton met het Zend Framework 1.1x waarmee client-componenten kunnen communiceren via HTTP.*

1.3.2 Doelstelling Afstuderen

De doelstelling van het onderzoek is het opleveren van een eindscriptie waarin een antwoord wordt geformuleerd op de gestelde onderzoeksvragen. De eindscriptie zal zich beperken tot een onderzoek naar het opbouwen van RESTful applicaties en de toepassing hiervan. Uitgangspunt hierbij is de bestaande situatie van Ibuildings. Het resultaat van de afstudeerperiode zal op basis van deze scriptie worden beoordeeld.

1.4 Onderzoeksvragen

De opbouw van deze afstudeerscriptie is gebaseerd op de verschillende onderzoeksvragen die onderstaand zijn opgesteld.

1.4.1 Hoofdvraag

De hoofdvraag is opgesteld aan de hand van de probleembeschrijving en doelstelling.

- *Hoe moet de REST API voor Ibuildings worden ontwikkeld en geïmplementeerd met het Zend Framework?*

1.4.2 Deelvragen

De onderstaande deelvragen zijn sub-vragen van de opgestelde hoofdvraag. Deze deelvragen bestaan op zichzelf ook weer uit sub-vragen.

1. *Wat is REST?*
    1. *Waar moet een applicatie aan voldoen om RESTful te zijn?*
    2. *Welke levels bestaan hierbinnen?*
    3. *Van welke protocollen is REST afhankelijk?*
2. *Hoe werkt een RESTful applicatie?*
    1. *Wat is de technische werking van een RESTful applicatie?*
    2. *Op welke architecturen is een RESTful applicatie gebaseerd?*
    3. *Wat zijn de alternatieven voor REST?*
3. *Uit welke onderdelen bestaat een RESTful applicatie?*

4. *Wat zijn de requirements van Ibuildings aan de componenten van de REST API?*

5. *Hoe werkt het Zend Framework en wat zijn hierbinnen de mogelijkheden voor REST?*

6. *Hoe kunnen de gewenste componenten worden geïmplementeerd binnen Zend?*

1.5 Opbouw document

De indeling van dit document is opgezet in chronologische volgorde. Dit houdt in dat er wordt gerefereerd aan voorgaande hoofdstukken.

- **Hoofdstuk 1** beschrijft de achtergronden, doelstellingen en onderzoeksvragen
- **Hoofdstuk 2** beschrijft de gebruikte methodieken. Het gaat hier zowel om de onderzoeksmethodiek als ontwikkelmethoden.
- **Hoofdstuk 3** beschrijft de voornaamste terminologie.
- **Hoofdstuk 4** gaat in op de betekenis van REST en de ideeën hierachter.
- **Hoofdstuk 5** beschrijft de onderdelen van HTTP die van toepassing zijn op REST.
- **Hoofdstuk 6** gaat in op de verschillende onderdelen van een REST API.
- **Hoofdstuk 7** beschrijft de specifieke eisen van Ibuildings. Deze eisen zijn zeer concreet en voldoen aan de HTTP standaard.
- **Hoofdstuk 8** gaat in op de werking van het Zend Framework. De technische werking wordt hier globaal besproken.
- **Hoofdstuk 9** beschrijft de resultaten van alle voorgaande hoofdstukken in de praktijk. Hier worden de ontwerp afwegingen van de verschillende componenten beschreven.
- **Hoofdstuk 10** gaat in op toegepaste de werkwijze en bevat evaluatie van de stage.
- **Hoofdstuk 11** geeft met een conclusie antwoord op de onderzoeksvragen.

1.6 Informatie voor de lezer

In dit document wordt aan de hand van literatuuronderzoek beschreven wat REST inhoudt en hoe dit in de situatie van Ibuildings kan worden toegepast. Er wordt hierbij van uitgegaan dat de lezer van deze scriptie beschikt over kennis van object georiënteerd programmeren. In hoofdstuk 3 wordt de specifiek gebruikte terminologie beschreven.

Bronverwijzingen in dit document zullen worden gedaan volgens het Vancouver Systeem[16]. Dit houdt in dat aan het einde van het document een pagina is ingevoegd met genummerde bronnen. In de tekst van het document wordt verwezen naar deze bronnen door middel van het bronnummer tussen blokhaken.

Het is van belang om tijdens het lezen van dit document appendix A door te nemen. Hieraan wordt gedurende de hele scriptie gerefereerd. Binnen dit document worden de begrippen REST API en REST API Skeleton door elkaar gebruikt. Er wordt hiermee verwezen naar hetzelfde product.

2 Methodieken

In dit hoofdstuk zijn de methodieken die tijdens dit project worden gebruikt weergegeven. Er wordt hierbij onderscheid gemaakt tussen onderzoek en ontwikkeling.

2.1 Onderzoek

Alle informatie voor het onderzoek naar REST en de ontwikkeling van de API zal worden verzameld aan de hand van onderzoek binnen de literatuur en verschillende briefings. Deze briefings zullen plaatsvinden met de REST-specialist binnen Ibuildings. De specialist geeft hierbij aan waar het component aan moet voldoen. De literatuur zal worden verzameld door middel van deskresearch.

In eerste instantie zal er een onderzoek naar de REST-richtlijnen in het algemeen worden uitgevoerd. Hieruit moet naar voren komen wat de bestaande modellen en richtlijnen voor RESTful applicaties zijn. Hierna zal de werking van het Zend Framework worden onderzocht, hierbij ligt de focus op de algemene werking die van toepassing kan zijn in het geval van een RESTful-architectuur.

Wanneer duidelijk is wat de richtlijnen van REST precies zijn en hoe het Zend framework kan worden toegepast dan zal er worden gekeken naar de gewenste componenten. Er zal per component worden onderzocht hoe deze volgens de algemene REST-richtlijnen het beste in combinatie met het Zend framework ontwikkeld kan worden. Van alle componenten zullen de resultaten en keuzes worden beschreven.

De resultaten van de briefings zullen worden verwerkt in een overzichtelijk functioneel ontwerp waarin de requirements zijn gesorteerd volgens de methode MoSCoW. MoSCoW[7] is een methode waarmee eisen aan het systeem gestructureerd worden aan de hand van hun prioriteit.

Daarnaast zal er verschillende documentatie over REST-richtlijnen die binnen Ibuildings bestaat worden gebruikt om algemene eisen aan de structuur naar boven te halen.

Deze bevindingen zullen onderzocht worden tijdens de ontwikkeling. Er wordt hierbij gekeken of de gevonden oplossing wel de juiste uitwerking heeft. Zie paragraaf 2.2 voor meer informatie over dit proces.

2.2 Ontwikkeling

Vanwege het vele onderzoek dat verbonden zit aan ieder component is gebleken dat het efficiënter en beter werkt om de ontwerp en bouw - fase in elkaar te laten overlopen. Hierbij wordt van te voren een concept ontwerp opgesteld dat iteratief uitgewerkt wordt.

Er is daarom gekozen om dit project aan de hand van een AGILE ontwikkelmethode uit te voeren. AGILE methodes worden ook wel lightweight methods genoemd. Hierbij ligt de focus meer op de requirements en implementaties hiervan binnen de code dan op lange ontwerp periodes en vergaderingen.

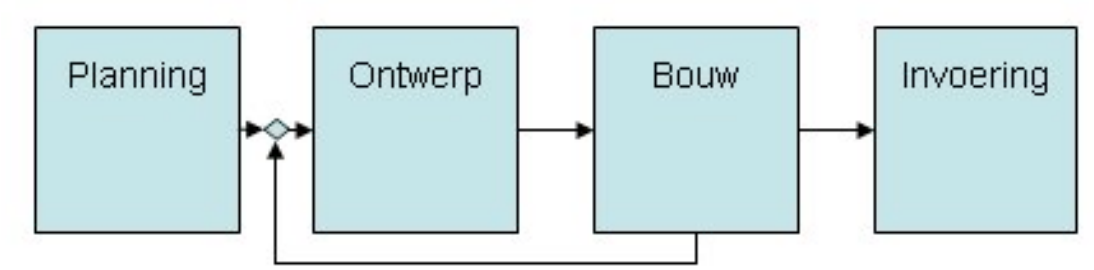
Als AGILE methode gaat gebruik worden gemaakt van delen uit de methodiek Extreme Programming[6]. Het gaat hierbij voornamelijk om korte iteraties waarin ontwerp, tests en programmacodes worden opgesteld. Er zal hierbij gebruik worden gemaakt van een vorm van Test Driven development. De focus ligt hierbij op het testen, er wordt van tevoren bepaald wat een component moet kunnen door middel van functionele en technische ontwerpdocumentatie.

Er is gekozen voor deze pragmatische manier van werken vanwege zijn flexibiliteit en omdat deze zeer snel tot resultaat leidt. Er is namelijk een grote kans op wijzigingen in requirements

tijdens dit project. Er bestaat op dit moment nog maar een globaal idee over hoe het product moet worden vormgegeven. Het moet dus zo eenvoudig mogelijk zijn om deze wijzigingen weer snel door te kunnen voeren.

Aan de hand van dit ontwerp wordt er een aantal geautomatiseerde Unit Tests bepaald alvorens de code geschreven wordt. Zie hoofdstuk “Terminologie” voor uitleg Unit Testing. Tijdens het programmeren worden er steeds meer tests bijgemaakt. Ook worden deze regelmatig uitgevoerd. Zelfs al is er maar een enkele regel code veranderd. Het voordeel hiervan is dat mogelijke fouten door veranderingen elders in de code direct gedetecteerd worden.

Omdat tijdens de ontwikkeling vaak nieuwe inzichten naar voren komen kan ook tijdens de ontwikkeling het ontwerp nog worden bijgeschaafd. Onderstaand is het proces te zien waarin de iteratie ontwerp en bouw één geheel zijn.



Agile Ontwikkeling

Wanneer een component is ontwikkeld dan zal deze uitgebreid gedocumenteerd worden. Hierin wordt de technische werking benadrukt maar ook hoe het component kan worden gebruikt.

Fases
De ontwikkeling van de API kan worden onderverdeeld in een aantal fases. Dit zijn Alpha 1, Alpha 2 en Alpha 3. Binnen deze fases is de MoSCoW priorisering te herkennen.

Alpha 1 omvat alle “Must haves” van de API. Het gaat hierbij om alle functionaliteit die nodig is om de API te laten functioneren.

Alpha 2 bevat alle “Should haves”. De focus ligt hierbij op de functionaliteiten die zeer gewenst zijn maar waar het product niet van afhankelijk is om te functioneren. Daarnaast worden er tijdens deze fase de inmiddels gedetecteerde “bugs” uit Alpha 1 opgepakt.

Alpha 3 bevat alle “Could haves”. Deze fase kan worden gezien als optioneel. Hierin worden alle functionaliteiten die mooi zijn om te hebben maar niet perse nodig weergegeven. Alpha 3 is in dit geval een soort uitloopfase die alleen in gang wordt gezet wanneer er tijd over is.

3 Terminologie

URI/URL/URN
URI staat voor Uniform Resource Identifier. Een URI identificeert een resource. Er bestaan twee soorten URI's, een URL en een URN. URL staat voor Uniform Resource Locator en is een verwijzing naar een resource d.m.v. tekst. URN staat voor Uniform Resource Name en is een verwijzing naar een resource door middel van een reeks van cijfers.

Resource
Een resource representeert iets van waarde binnen een systeem[14]. Denk hierbij aan een artikel, persoonsgegevens of een order bij een webshop.

API
Een interface waarmee kan worden gecommuniceerd met een applicatie(onderdeel). Een API kan worden gezien als het doorgeefluik naar de applicatie of resource.

HTTP
Hypertext Transfer Protocol[5]. Het protocol voor communicatie tussen een webclient en een webserver. Binnen HTTP staat vastgelegd welke requests op wat voor manier worden afgehandeld. HTTP is een stateless protocol.

HTTP methods
Een HTTP methode[5] is het type actie dat via HTTP op een resource wordt uitgevoerd. Er zijn veel verschillende methoden. De methoden waar in dit document naar zal worden gerefereerd zijn GET, POST, PUT, DELETE, HEAD en OPTIONS. HTTP methods worden ook wel HTTP verbs genoemd.

Stateless protocol
Een protocol dat ieder request behandeld als een individueel request[18]. Voorgaande acties hebben geen invloed op de afhandeling hiervan.

Web service
Een web service[11] is een web applicatie die bedoeld is om gebruikt te worden door andere applicaties. Dit gebeurt via een netwerk. Tijdens dit proces kan een server en een client worden onderscheiden. Het maakt niet uit met wat voor technologie deze applicaties geïmplementeerd zijn.

Scaling
Een actie om de capaciteit[15] van een software omgeving te vergroten.

MIME
Multipurpose Internet Mail Extensions[4]. Dit is de internet standaard voor e-mail. MIME legt de structuur van een bericht verstuurd over het internet vast. Dit kan voor zowel e-mail als een HTTP request zijn. Deze structuur wordt beschreven in headers van het bericht.

Meta Informatie
Beschrijvende informatie over een hoofdzaak. MIME bevat meta informatie in de headers van het bericht.

JSON
Javascript Object Notation. Wordt gebruikt voor het uitwisselen van data tussen verschillende applicaties. JSON kan worden gezien als het alternatief voor XML.

XML
Extensible Markup Language. Een taal waarmee op een gestructureerde manier data kan worden

weergegeven.

HTTP Message
Een bericht[5] dat wordt verstuurd via het HTTP protocol. Dit bericht bestaat uit verschillende vastgestelde onderdelen.

Klasse
Een klasse is een blauwdruk voor een object. Deze klasse kan data en verschillende methodes bevatten.

Object
Een instantie van een klasse. Dit is de klasse in werking.

Regular Expression
Een manier om patronen te beschrijven. Een Regular Expression is dus een patroon waarmee de computer stukken tekst of getallen kan herkennen.

Stack
Letterlijk een stapel (denk aan een stapel papier). Een stack heeft de eigenschap dat er alleen maar iets bovenop de stapel kan worden gelegd en ook alleen maar iets van bovenaf de stapel kan worden weggenomen.

Unit test
Een methode waarmee individuele stukken code geautomatiseerd kunnen worden getest. Er wordt hierbij gekeken of een bepaalde invoer ook de verwachte uitvoer genereert.

Bugs
Ongewenste gedraging van software. Denk hierbij aan fouten binnen een programma.

CRUD
Staat voor de acties Create, Read, Update en Delete[15]. Zie hoofdstuk 4 paragraaf 2.2 voor meer informatie hierover.

SOA
Staat voor Service Oriented Architecture. Zie hoofdstuk 4 paragraaf 2 voor meer informatie hierover.

SOAP
SOAP staat voor Simple Object Acces Protocol[12]. Dit is een Service Oriented Architecture. Zie hoofdstuk 4 paragraaf 3.1 voor meer informatie hierover.

WSDL
Staat voor Web Service Definition Language[12]. Dit is een contract waarin staat wat de web service toestaat. Hierin wordt gedefinieerd welke acties kunnen worden uitgevoerd. Voor meer informatie zie hoofdstuk 4 paragraaf 3.1.

Hypermedia
Hypermedia[14] zijn URI verwijzingen naar resources die in de HTTP response worden meegestuurd. Dit is eigenlijk precies hoe een “klassieke” webpagina wordt weergegeven. Hierin staan namelijk verschillende hyperlinks naar andere pagina’s. Deze hyperlinks zijn de hypermedia van de pagina.

4 REST

In dit hoofdstuk zal worden ingegaan op de betekenis van REST. Er zullen hier onderdelen van de deelvraag “Wat is REST?” worden behandeld. Daarnaast wordt er ingegaan op de achterliggende werking, architecturen en mogelijke alternatieven. Het hoofdstuk wordt afgesloten met een conclusie waarin alle bevindingen worden aangehaald.

4.1 Wat is REST?

REST staat voor Representational State Transfer[1]. Dit zijn ontwerpcriteria die kunnen worden gebruikt om te communiceren tussen verschillende systemen via het HTTP protocol. REST is in het jaar 2000 geïntroduceerd door Roy Fielding. Hij deed deze introductie in een proefschrift[13] aan de Universiteit van Californië.

De laatste jaren zijn applicaties gebaseerd op REST in flink tempo opgekomen als een zeer populair[17] principe. Het is gebleken dat REST een stuk gemakkelijker is toe te passen dan andere technieken zoals SOAP. Grote bedrijven als Dropbox, Google en Amazone maken al een lange tijd intensief gebruik van REST. Een applicatie die voldoet aan de REST-richtlijnen wordt ook wel RESTful genoemd. Het bepalen hiervan kan worden gedaan aan de hand van het Richardson Maturity Model, zie appendix A.

De focus van REST ligt op de resources van een systeem. Het idee hierbij is dat iedere resource wordt geïdentificeerd door middel van een URL. Op ieder van deze resources kan een set van acties worden uitgevoerd. Bij een resource moet dus gedacht worden aan een URI soortgelijk aan /company/user/25.

De implementatie van REST bestaat uit clients en servers. De client initieert een request aan de server. De server antwoordt hierop met een response message en HTTP statuscode. Deze response kan bestaan uit alle gangbare formaten maar gebruikelijk is JSON of XML.

De kracht van REST zit hem, naast het gebruik van resources, vooral in het applicatie en platform onafhankelijke karakter. De enige benodigde onderdelen zijn het Hypertext Transfer Protocol (HTTP) en de Uniform Resource Locator (URL). Alle communicatie gaat via dit protocol, REST is hier volledig op gebaseerd. Alle acties en logica zijn afkomstig vanuit de mogelijkheden die HTTP biedt. REST services en REST clients kunnen dan ook met vrijwel iedere taal worden ontwikkeld. REST is eigenlijk meer een oude filosofie dan een nieuwe technologie. Hiermee wordt bedoeld dat er teruggegrepen wordt op de oude en simpele principes van HTTP.

Volgens Richardson[14] kan het volledige World Wide Web wezenlijk worden gezien als een architectuur die is opgezet volgens REST. Alle protocollen en handelingen zijn hier namelijk gebaseerd op HTTP. Op de website van Martin Fowler [2] staat hier de volgende quote over:

“the web is an existence proof of a massively scalable distributed system that works really well, and we can take ideas from that to build integrated systems more easily.”

Het boek REST in Practice[15] geeft een aantal karakteristieken van het web die aangeven wat de voordelen van REST in de praktijk zullen zijn. Er wordt hierbij een vergelijk getrokken met een traditioneel enterprise software systeem. De conclusie die hieruit getrokken wordt is dat de voordelen die een complete REST implementatie geven precies de karakteristieken zijn die een enterprise systeem kwalitatief goed maakt. De punten die hierbij worden genoemd zijn:

- Scalable
  Het web is zeer loose coupled. Dit houdt in dat er weinig afhankelijkheden tussen verschillende systemen zijn. Wanneer een applicatie RESTful wordt opgezet is het zeer

gemakkelijk om dit systeem uit te breiden of breder in te zetten. Denk hierbij aan desktop/mobiele applicaties. Wanneer in eerste instantie alleen een desktop applicatie nodig was dan is het in het geval van REST mogelijk om met weinig moeite bijv. een mobiele variant te ontwikkelen. De business logica blijft hetzelfde op de server. Er hoeft alleen maar ingehaakt te worden op de REST service.

Daarnaast is het van belang dat HTTP een stateless protocol is. Deze eigenschap zorgt ervoor dat er geen afhankelijkheden aan de hardware bestaan. Hierdoor kan het systeem gemakkelijk uitgebreid worden. Dit wordt ook wel horizontal scaling genoemd.

- Fault-tolerance
  Onder het voorgaande punt, scalable, staat beschreven dat vanwege de stateless eigenschap van HTTP het zeer gemakkelijk is om hardware uit te breiden omdat hier geen afhankelijkheden mee bestaan. Hetzelfde geldt hierdoor natuurlijk ook voor hardware falen. Het is vanwege deze onafhankelijkheid geen probleem om snel een defecte server te vervangen of uit te wijken naar een ander systeem.

- Recoverable
  HTTP bevat verschillende methoden die op een veilige stabiele manier herhaald kunnen worden. Het gaat hier om bijvoorbeeld de methode GET, PUT en DELETE. Meer hierover in hoofdstuk Hypertext Transfer Protocol. In het geval van falen wordt het probleem direct duidelijk gemaakt door middel van een statuscode en foutmelding.

- Secure
  HTTPS is een zeer volwassen technologie die op een veilige manier point to point communicatie kan uitvoeren. Daarnaast zijn er vele manieren om HTTP verbindingen te beveiligen.

- Loosely coupled
  Zoals al eerder besproken is het gebruik van HTTP web technologie zeer loosely coupled. Het toevoegen van een nieuwe website heeft geen effect op bestaande websites. Daarnaast ondersteunt alles in het WWW het HTTP protocol; het is hierdoor dus erg uniform in te zetten.

Dit zijn dan ook direct eigenschappen van een RESTful applicatie. Vanwege het aanhouden van het stabiele en uniforme karakter van HTTP worden al deze punten gestimuleerd. Het hangt af van de mate van “RESTfulness” waarmee de applicatie wordt geïmplementeerd in hoeverre deze eigenschappen ook naar voren komen.

4.2 De werking van REST

In deze paragraaf wordt verder ingegaan op de naar de achterliggende theoretische werking van RESTful applicaties. Er zal hierbij in worden gegaan op de architectuur maar ook op mogelijke alternatieven en de verschillen hiertussen.

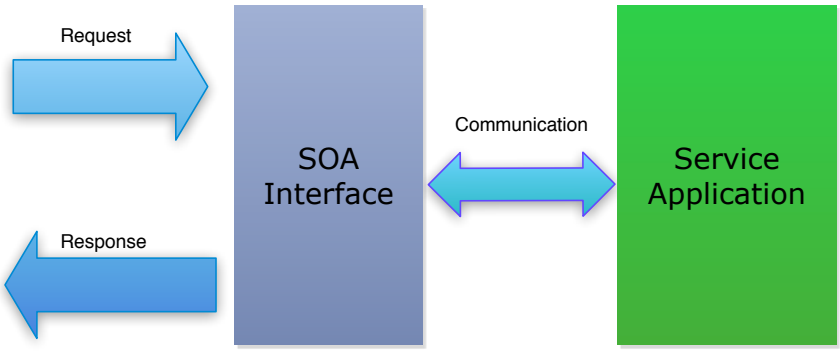
4.2.1 Architecturen

Het is goed om te beseffen dat REST geen architectuur is maar een set van ontwerpcriteria. Zoals Richardson in zijn boek “RESTful webservices”[14] aangeeft: “Er zijn architecturen die voldoen aan de criteria van REST maar er is niet iets als een REST architectuur”. De twee belangrijke architecturen die met communicatie over netwerken in verband kunnen worden gebracht zijn SOA en ROA.

4.2.1.1 SOA en ROA

Service Oriented Architecture

SOA gaat uit van de service. Bij een SOA implementatie wordt er altijd tegen een vaste interface gecommuniceerd. Acties worden hierbij altijd uitgevoerd door de service. Een service kan eigenlijk worden gezien als een vaste applicatie die alle bewerkingen uitvoert. Onderstaand is dit proces versimpeld weergegeven.



Service Oriented Architecture

Service Oriented Architectures kunnen worden gezien als Level 0 REST volgens het Richardson Maturity Model, zie appendix A. Er is namelijk maar een enkel toegangspunt tot de service en er wordt niet met resources gewerkt.

Resource Oriented Architecture

ROA gaat uit van de resource. In dit geval wordt een resource gezien als belangrijke data binnen het systeem. Bij een Resource Oriented Architecture[14] worden bewerkingen direct op de resource uitgevoerd. In het geval van ROA is er kortweg voor iedere resource een aparte interface waartegen kan worden gecommuniceerd. Zie de volgende paragraaf voor uitgebreide uitleg.

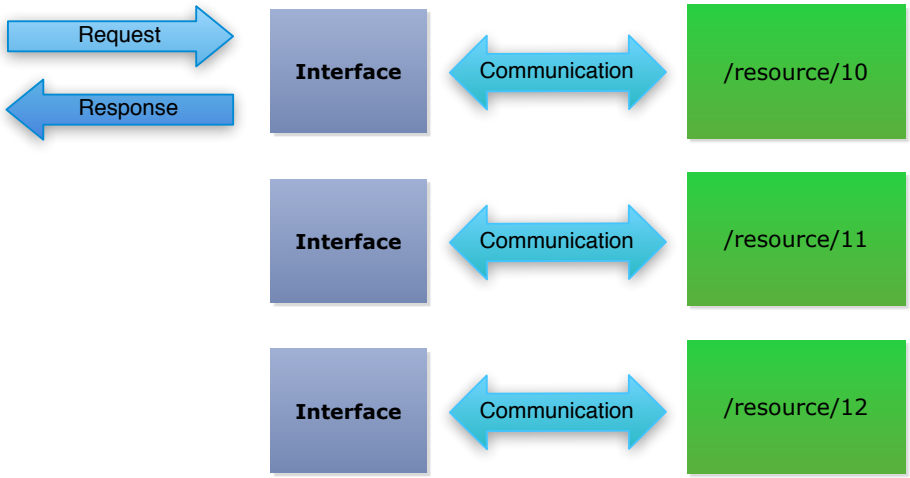
Volgens Richardson is de Resource Oriented Architecture (ROA) de architectuur waar REST criteria het beste in terug te vinden zijn. Richardson heeft in 2007 met zijn boek “RESTful web Services” met deze architectuur eigenlijk een standaard voor de implementatie van REST gespecificeerd. Het model dat Richardson hiervoor heeft ontwikkeld, zie appendix A, is breed geaccepteerd en wordt tegenwoordig als de standaard voor RESTful applicaties gezien.

4.2.1.2 Resource Oriented Architecture

Het idee achter een “truly RESTful applicatie” is dat iedere resource wordt geïdentificeerd door zijn eigen unieke URL, zie hoofdstuk appendix A. Het idee hierachter is dat op een logische en natuurlijke wijze alle bewerkingen kunnen worden uitgevoerd op een willekeurige resource. Het is dus mogelijk om een read, update of delete request op bijvoorbeeld /article/34 uit te voeren.

De benadering hierbij is logica. In de “echte wereld” worden alle acties ook uitgevoerd op de resource die van toepassing is. Wanneer iemand bijvoorbeeld een verhaal op papier schrijft dan zal deze alle bewerkingen ook direct op dat stuk papier uitvoeren. Het beschreven papier kan hierbij als resource worden gezien en de bewerkingen als CRUD, zie volgende paragraaf.

Deze Resource Oriented Architectuur is ook net wat de toepassing van REST zo populair[17] maakt. Het voelt namelijk erg natuurlijk om naar de URL /article/34 toe te gaan wanneer er een actie moet worden uitgevoerd op artikel 34. De resources definiëren hierdoor als het ware een hiërarchische structuur die direct de opbouw van de data weergeeft. Onderstaande afbeelding geeft de Resource Oriented Architectuur schematisch weer. Hierin is te zien dat er voor iedere resource een bestaat. Deze interface bevat de acties die op de resource kunnen worden uitgevoerd.



Resource Oriented Architecture

Het gaat hierbij nog steeds om een digitaal informatiesysteem en het is daarom niet mogelijk om ook fysiek de resource direct te benaderen. Om toch te kunnen communiceren met de resource is het van belang om een logische laag[15] aan te brengen die de resource representeert. Deze laag moet “snappen” dat wanneer er een actie op bijv. /resource/10 wordt uitgevoerd er wordt verwezen naar de fysieke resource[18]. Deze laag voor de fysieke resource bevat de acties die kunnen worden uitgevoerd op de resource (CRUD). Het idee van een software representatie van de fysieke resource wordt ook wel “Pass by Value semantiek”[18] genoemd.

Er zijn een aantal punten die gelden voor communicatie met resources in een RESTful applicatie:

- De communicatie vindt plaats met de representatie van resources
- De representatie kan worden teruggegeven in ieder ondersteund formaat (XML,JSON etc.)
- Iedere resource maakt gebruik van de uniforme HTTP interface
- Iedere resource heeft een unieke identifier (URL)
- Iedere resource heeft een of meer representaties

Hierbij is er dus duidelijk onderscheid tussen de logische resources en de fysieke resources. De logische resources kunnen worden gezien als het doorgeefluik (facade) tussen de client en de fysieke resources.

Vanaf level 1 van het Richardson Maturity Model, zie appendix A, kan worden gesproken van een simpele Resource Oriented Architecture. Dit omdat iedere resource apart kan worden geadresseerd door middel van een URI.

Het is hierbij niet zo dat een resource maar een enkele URI kan bezitten. Er bestaat hier namelijk een one to many relatie. Aan iedere resource kunnen meerdere URL’s zijn verbonden. Ook dit kan weer worden terug gekoppeld naar de echte wereld. Er kunnen namelijk meerdere wegen bestaan om ergens te komen.

4.2.2 CRUD

De volledige werking van een RESTful API is gebaseerd op de beschikbare HTTP methodes. Deze methodes definiëren in feite de basis acties die ieder informatiesysteem bevat. Het gaat hier om de methodes PUT, POST, GET en DELETE. Deze methodes worden uitgebreid behandeld in het hoofdstuk “Hypertext Transfer Protocol”.

In principe definiëren deze methodes de basis voor een CRUD systeem[1]. CRUD staat voor Create, Read, Update en Delete. Hiermee worden eigenlijk de basisfunctionaliteiten[15] van een informatiesysteem gedefinieerd.

CRUD implementatie binnen de REST API komt voor vanaf Richardson Maturity Model Level 2, zie appendix a. In dit geval wordt er namelijk gebruik gemaakt van de standaard HTTP methodes om bewerkingen op resources uit te voeren.

4.3 Alternatieven voor REST

Veel ontwikkelaars zullen bij communicatie tussen verschillende applicaties[11] direct denken aan Remote Procedure Calls, RPC. Voorbeelden hiervan zijn RMI en CORBA.

Het belangrijkste verschil met REST is, dat er bij REST gebruik wordt gemaakt van het HTTP protocol. Dit is lichter en breder ondersteund dan de hiervoor genoemde alternatieven. Deze HTTP basis neemt veel werk uit handen en zorgt ervoor dat een deel van de API niet ontwikkeld hoeft te worden. The interface van de API is namelijk al bepaald door HTTP. De HTTP methodes zijn de methodes die ook in de API zitten. Zie het hoofdstuk Hypertext Transfer Protocol voor meer uitleg hierover.

Het gebruik van RPC is nog steeds zeer afhankelijk van de implementatie, de gebruikte taal of framework moet dit namelijk ondersteunen. Het voordeel van REST is dat dit hierbij geen enkel probleem is. REST is puur afhankelijk van HTTP en is erg gemakkelijk te consumeren. De resultaten die worden teruggegeven zijn zelfs door een mens goed te begrijpen. Deze eenvoud is dan ook direct het punt waardoor REST succesvol is geworden.

Er zijn op dit moment twee hoofdrichtingen[17] binnen de ontwikkeling van web services. De nog relatief jonge RESTful applicaties en de traditionele SOAP architectuur.

4.3.1 SOAP

SOAP maakt gebruik van het RPC principe maar is dat niet. SOAP is in de jaren ‘90 door Microsoft ontwikkeld als middleware technologie. De afkorting staat voor Simple Object Acces Protocol.

Het idee achter SOAP[12] is uitwisselbaarheid zodat andere standaarden kunnen inhaken en worden geïntegreerd binnen SOAP. De communicatie bestaat uit volledig tekst based berichten. Een SOAP bericht bestaat uit XML en bevat altijd een header en body. In dit bericht kunnen verschillende headers worden meegegeven die zorgen voor autorisatie en authenticatie. SOAP berichten kunnen worden verstuurd over verschillende protocollen. Dit kan zowel HTTP, SMTP of FTP zijn.

Voor het ontwikkelen van SOAP services zijn verschillende development kits beschikbaar. Het opzetten van een dergelijke implementatie vergt dan ook veel specifieke aanpassingen. SOAP is namelijk een standaard op zich en maakt hierdoor gebruik van zijn eigen regels en eisen. Dit in tegenstelling tot REST die gebruik maakt van verschillende technologieën en hun standaarden.

Een voorbeeld hiervan is dat SOAP gebruik maakt van een enkele facade. Deze facade kan worden gezien als een voorgevel van de web service. Dit is de enkele toegangspoort waardoor de service benaderd kan worden. De communicatie zal dus enkel via dit punt verlopen. De

service wordt via een enkele URI benaderd. Voor dit punt is een WSDL document geplaatst. WSDL[12] staat voor Web Services Description Language. Deze toepassing is ontwikkeld door IBM, Ariba en Microsoft.

WSDL is een XML document waarin staat gedefinieerd welke acties de service kan uitvoeren en wat voor type data er na deze actie wordt geretourneerd. Dit is eigenlijk een soort contract waarin staat wat de web service toelaat en wat niet.

Omdat er gebruik wordt gemaakt van een enkele URI kan er worden gesteld dat SOAP niet gebaseerd is op de Resource Oriented Architectuur maar op de Service Oriented Architecture. Alle acties en data worden namelijk via een enkel punt uitgevoerd en niet zoals bij een RESTful applicatie waarbij iedere resource een unieke URI bezit.

Op basis hiervan kan worden gezegd dat SOAP thuishoort[15] op Level 0 van het Richardson Maturity Model, zie appendix A. Met SOAP vergelijkbare methoden zijn XML-RPC en POX.

4.3.1.1 SOAP t.o.v. RESTful applicaties

Het is een feit dat SOAP, dus de Service Oriented Architecture, goed ingezet kan worden voor verschillende problemen.

Een voordeel van SOAP ten op zichte van RESTful applicaties is dat er door middel van het WSDL document standaard type checking zit ingebouwd op de data die wordt gecommuniceerd. Daarnaast is SOAP een standaard op zichzelf en wordt dit in de basis door vele frameworks en talen ondersteund. Zoals al eerder genoemd is SOAP een Service Oriented Architecture. Dit houdt in dat er een enkel toegangspunt naar de service bestaat. In sommige gevallen kan dit handig zijn.

Wat duidelijk naar voren komt bij SOAP is dat dit protocol niet volledig gebaseerd is op de web standaarden. Een RESTful applicatie Level 2 volgt alle standaarden die voor het Web gedefinieerd zijn. SOAP doet dit niet, een voordeel hiervan is dat SOAP niet afhankelijk is van het transport protocol HTTP maar ook gemakkelijk over andere protocollen kan worden uitgevoerd.

4.4 Conclusie

Op het internet zijn vele definities en opvattingen over REST te vinden. Veel hiervan zijn een misinterpretatie van wat Roy Fielding in 2000 in zijn paper [13] beschreef. REST wordt vaak als architectuur gezien maar dit is een misverstand. Kortgezegd is REST een verzameling van ontwerprichtlijnen die volledig gebaseerd zijn op HTTP. REST is eigenlijk hoe het World Wide Web werkt.

Er zijn verschillende architecturen waar REST zowel gedeeltelijk als volledig op kan worden toegepast. De architectuur waarin alle aspecten van REST verwerkt zitten, is de Resource Oriented Architecture. Binnen deze architectuur wordt er uitgegaan van de mogelijkheid om iedere resource apart te benaderen en hier ook bewerkingen op uit te kunnen voeren.

De mate waarin een applicatie is ontwikkeld volgens de richtlijnen van REST kan worden bepaald door middel van het Richardson Maturity Model, zie appendix A. Dit model meet aan de hand van het gebruik van URI's, HTTP en Hypermedia of een applicatie "RESTful" is.

Er bestaan vele alternatieven voor RESTful applicaties, SOAP is hier een voorbeeld van. SOAP is gepositioneerd op level 0 van het Richardson Maturity Model, dit houdt in dat er niet wordt voldaan aan een enkele richtlijn van de REST guidelines. SOAP is een voorbeeld van Service Oriented Architecture. Hierbij is er een enkel toegangspunt van de web service waar via gecommuniceerd kan worden.

Het is belangrijk om te beseffen dat REST niet per definitie beter is dan SOAP. Het verschilt

per situatie welke de beste oplossing is. Een belangrijk voordeel van SOAP is dat het niet gebonden zit aan HTTP. Ook is SOAP een protocol dat in vele talen en frameworks standaard verwerkt zit en standaard type checking ondersteunt.
REST daarentegen heeft als voordeel dat het volledig gebaseerd is op HTTP. Hierdoor is er om een RESTful applicatie te laten functioneren niet veel meer nodig dan alleen HTTP. REST kan hierdoor in principe met iedere taal of framework gebruikt worden.

Het grote voordeel van een RESTful applicatie is dus dat deze zonder veel technologie kan worden geïmplementeerd. Een nadeel daarvan is dat REST op zichzelf geen protocol is maar puur een set van richtlijnen. Het hangt dus van het inzicht van de programmeur af hoe RESTful de applicatie is. SOAP daarentegen is een protocol en forceert dat er aan zijn standaarden hiervan wordt voldaan.

Er kan worden vastgesteld dat de basis van REST, HTTP, zeer betrouwbaar is en rekening houdt met schaalbaarheid en uitbreidbaarheid. Een bewijs hiervan is het grootste geautomatiseerde systeem ter wereld, het World Wide Web.

5 Hypertext transfer protocol

Een erg belangrijk onderdeel waar REST gebruik van maakt is het Hypertext transfer protocol[5] beter bekend als HTTP. HTTP kan worden gezien als de basis van REST. REST maakt namelijk volledig gebruik van de bestaande functionaliteiten binnen HTTP. De mate waarin HTTP wordt toegepast bepaalt hoe RESTful een applicatie is, zie het Richardson Maturity Model appendix A.

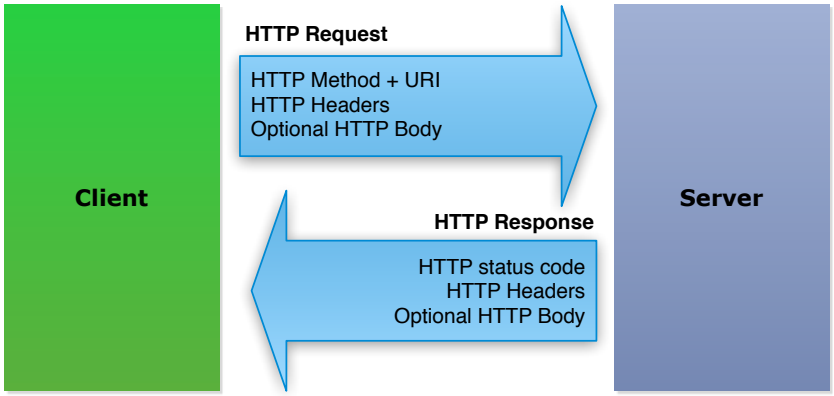
In een RESTful systeem is HTTP het application protocol, dit houdt in dat alle acties volgens deze standaard moeten worden uitgevoerd. Wanneer er een HTTP request naar de server wordt gestuurd dan bevat dit request alle informatie om een actie op deze server uit te voeren. Denk hierbij aan de HTTP methode, de verschillende headers en de request body. Het op een correcte manier gebruik maken van HTTP headers en methoden draagt bij aan de “RESTfulness” van de applicatie, zie appendix A.

Een belangrijke eigenschap van HTTP is dat dit een stateless protocol is. Dit houdt in dat ieder request wordt behandeld als een op zichzelf staand request. Dit heeft als uitwerking dat het ene request het andere niet kan beïnvloeden. Er wordt namelijk geen informatie over voorgaande aanvragen opgeslagen.

HTTP wordt al sinds 1990 gebruikt en is sinds die tijd doorontwikkeld. De eerste versie staat bekend als HTTP 0.9. Dit protocol was puur bedoeld om simpele text data over het internet te versturen. Vanwege de opkomst van het internet ontstond er de behoefte om meer dan alleen tekst te versturen. HTTP 1.0 was een verbetering op 0.9 en creëerde de mogelijkheid om MIME type berichten te versturen. Dit houdt in dat er meta informatie over het bericht in de vorm van headers wordt meegestuurd.

Vanwege het steeds groter worden van het internet was er behoefte aan strakkere richtlijnen en meer mogelijkheden. HTTP 1.1 was hier het antwoord op. Versie 1.1 is strikter in zijn richtlijnen om betrouwbaarheid te borgen. Dit is terug te zien in de eisen rondom headers en de verschillende request methoden. Deze “eisen” aan HTTP 1.1 zijn vastgelegd in RFC 2616[5].

De basis onderdelen waaruit het protocol bestaat zijn een connectie, request en response. De request en response zijn de onderdelen waar we ons vooral op zullen focussen in verband met REST. De inhoud van dit hoofdstuk is gebaseerd op RFC 2616[5]. Alleen de onderdelen en methoden van HTTP die relevant zijn voor REST worden besproken in dit hoofdstuk.



HTTP communicatie overzicht

5.1 HTTP methods

HTTP methods zijn de standaard acties die gedefinieerd zijn om bewerkingen op resources uit te voeren. Onderstaand zijn de, voor REST, relevante methodes beschreven. HTTP methodes worden ook wel “verbs” genoemd.

GET
Moet alleen gebruikt worden voor veilige requests. Het gaat er hierbij om dat er geen bijeffecten plaatsvinden waarvoor de client verantwoordelijk is. Ook moet het request idempotent zijn, dit houdt in dat onafhankelijk van de hoeveelheid keer dat het request is uitgevoerd, de uitvoer gelijk blijft. GET moet dus niet gebruikt worden voor het aanmaken, updaten of verwijderen van een informatie op de server. Een bekend voorbeeld van een GET request is het opvragen van een webpagina. Een dergelijk request geeft iedere keer een gelijke HTML output terug zonder aanpassingen te doen op het systeem.

POST
Kan worden gebruikt voor het creëren of het updaten van een resource. Volgens HTTP maakt POST een verandering binnen het systeem en zal het request nooit herhalen in geval van een error (in tegenstelling tot GET). Dit houdt in dat POST niet idempotent of safe is. Algemeen is aangenomen[14] dat het beter is om POST alleen voor het creëren van resources te gebruiken en PUT voor het updaten hiervan.

Wanneer een POST request is geslaagd dan wordt er in de response altijd een Location header[5] toegevoegd. Deze Location header bevat de locatie van de nieuw gecreëerde resource.

PUT
Creëert of update een resource aan de hand van de URI. PUT voor creëren dient alleen gebruikt te worden voor URI's die zijn gecreëerd door een client. Bij automatisch gegenereerde URI's moet altijd POST worden gebruikt. Voor het updaten van een resource moet altijd PUT worden gebruikt. PUT voert, wanneer een resource bestaat, een volledige update uit van een bestaande resource. Ongeacht hoeveel keer het request wordt uitgevoerd, PUT blijf dezelfde resource updaten. Dit maakt dat PUT idempotent is.

DELETE
Geeft door aan de server dat er een 'logische' delete actie op een resource moet worden uitgevoerd. Dit betekent niet direct dat de resource ook verwijderd zal worden. Het is net welke actie er op de server aan de resource gekoppeld zit. Er kan bijv. worden gespecificeerd dat bij een delete de resource moet worden verplaatst naar het archief. DELETE voert net als PUT hetzelfde uit, ongeacht hoe vaak het request wordt aangeroepen op een resource. De eerste keer wordt er een verandering doorgevoerd, daarna niet meer want de resource is weg. Dit maakt ook DELETE idempotent.

OPTIONS
De OPTIONS methode geeft informatie over de beschikbare communicatie opties. Wanneer er een OPTIONS request wordt uitgevoerd op een specifieke resource dan zal deze de beschikbare HTTP methodes retourneren. Dit zal gebeuren met een 200 HTTP response code. Een message body is optioneel en standaard zal de response dan ook geen body bevatten, alle informatie is gedefinieerd in de header.

De OPTIONS methode kan niet gecached worden en is, net als GET, safe en idempotent. Dit houdt in dat ongeacht hoeveel keer de methode wordt uitgevoerd de representatie iedere keer gelijk moet zijn zonder dat er een verandering op de server plaats vindt.

Wanneer een asterisk "*" als URI wordt verstuurd dan wordt er informatie opgevraagd over de server in plaats van een specifieke resource.

Wanneer er informatie via de body wordt teruggegeven dan mag content-negotiation (zie par 5.2) worden gebruikt voor het bepalen van de juiste output. Het kan dan eigenlijk worden gezien als een "normaal" request.

HEAD
De HEAD methode maakt in principe hetzelfde request als GET maar geeft geen body terug.

HEAD geeft puur de headers van het request terug. Dit is handig wanneer er meta informatie nodig is over een resource zonder dat er een volledige response hoeft worden verstuurd.

Het is van belang dat de headers bij het HEAD request identiek zijn aan die van het GET request. De response van het HEAD request mag worden gecached. HEAD kan worden gebruikt om voordat een resource wordt opgevraagd, via de headers te controleren of er iets is veranderd. Ook HEAD is net als GET een safe en idempotent method.

5.2 HTTP message

In deze paragraaf worden verschillende onderdelen van het HTTP message besproken. Er wordt hierbij ingegaan op de onderdelen die van belang zijn bij het compleet en stabiel opzetten van een RESTful applicatie. Een HTTP message is van toepassing op zowel de request als de response.

HTTP requests zijn MIME type berichten. MIME staat voor Multipurpose Internet Mail Extension. Hiermee wordt de inhoud van een bericht beschreven. Dit wordt gedaan door middel van headers in het bericht. Deze headers beschrijven het request. Het kan worden gezien als meta informatie.

Accept header
Het kan voorkomen dat de output van een request in verschillende formaten kan worden terug gegeven. Denk hierbij aan XML, HTML of JSON. Het is natuurlijk mogelijk om het data type in de URI te verwerken. Het probleem hiermee is dat er op deze manier wordt afgeweken van de HTTP standaard.

HTTP heeft hier een goede oplossing voor, dit wordt content negotiation genoemd. Het HTTP request bevat hier een speciale header voor. Deze wordt accept header genoemd. Hierin kan worden aangegeven wat voor data formaat de client terug verwacht. De accept header is een string die met prioriteit aangeeft welke data types gewenst zijn. Zie Quality values voor meer informatie over priorisering.

In de accept header is het mogelijk om aan ieder gegeven datatype, naast de qvalue, parameters mee te geven. Deze parameters kunnen zelf worden gekozen om datatypes nog verder te specificeren. Alleen in de accept header is het mogelijk om op deze manier meerdere parameters mee te sturen. De andere accept-* headers ondersteunen alleen qvalue.

Accept-language header
Net als voor content negotiation is er een mogelijkheid om de taal van een request te specificeren. Dit gebeurt op een soortgelijke manier. De taal wordt gespecificeerd in de accept-language header. Ook deze header bestaat uit een string waarin verschillende talen met prioriteit staan aangegeven.

Accept-encoding header
De inhoud van een bericht kan worden gecodeerd. Denk hierbij aan het comprimeren of versleutelen van data. Ook de encoding van het bericht kan worden gespecificeerd in een header deze header wordt accept-encoding genoemd. De opbouw van de inhoud is gelijk aan die van de accept-language header.

Quality values
Alle accept-* headers binnen een HTTP request maken gebruik van quality values ook wel qvalue genoemd. Deze qvalue is een parameter van een entiteit en geeft aan hoe hoog de prioriteit van de entiteit is. Dit principe wordt het duidelijkst d.m.v. een voorbeeld. De accept header `application/json; q=0.7, text/html; q=0.8, text/xml` bevat prioritering d.m.v. qvalues. Parameters worden aan een entiteit meegegeven door middel van een puntkomma “;”. Alle entiteiten hebben standaard de maximale qvalue 1. Wanneer er een lagere qvalue wordt meegegeven dan betekent het dat deze entiteit een lagere prioriteit heeft. Het hiervoor beschreven voorbeeld heeft als volgorde: tekst/xml, tekst/html en daarna

application/json. Als er geen q parameters zijn meegeven dan telt de volgorde als prioriteit.

Message Body
De body van een message is een optioneel onderdeel. Het is daarom van belang dat de server die het bericht verwerkt controleert of deze gevuld is. Zo niet dan kan deze worden genegeerd. Indien dit wel het geval is, dan moet worden bepaald wat het type van de inhoud van het bericht is. Dit wordt gedaan aan de hand van de content-type header. Dit geldt voor zowel een HTTP request en response. In principe kan met iedere HTTP request, met uitzondering van HEAD requests, een message body worden meegestuurd. Het is wel een feit dat dit naast PUT en POST niet algemeen geaccepteerd is.

De response daarentegen kan natuurlijk vrijwel altijd een message body, met uitzondering van HEAD response, bevatten. Maar ook hier is het van belang om te kijken naar wat algemeen geaccepteerd is. OPTIONS bijvoorbeeld mag een response body bevatten maar veel middleware verwacht dit niet en verwijderd deze body.

Content-type
De content-type header beschrijft wat voor type data de body van een HTTP bericht bevat. Dit is handig omdat de server dan weet dat bijv. de JSON body van een request JSON is. Aan de hand van de content-type header kan op de server worden bepaald hoe er met de inhoud van het bericht moet worden omgegaan.

Content-length
Deze header geeft de lengte van het bericht aan in de vorm van het aantal karakters. Wanneer een response een zin met 12 tekens retourneert dan zal de Content-length header de waarde 12 bevatten.

Etag
Ook wel entity tag genoemd, is een van de mechanismen dat binnen HTTP gebruik kan worden voor web cache validatie. Een Etag kan worden gezien als een unieke vingerafdruk van een resource. Wanneer de resource verandert dan zal de Etag ook veranderen. Er kan hierdoor worden gekeken of een resource in een cache nog up to date is. Dit gebeurt door de controle If-None-Match/If-Match. Ook iedere versie van een resource moet een unieke Etag hebben. Denk hierbij aan resultaten in verschillende talen. Om collisions tussen verschillende Etags te voorkomen is het verstandig om voor de unieke resource hash een collision-resistant hashmethode te gebruiken. MD5 is hier goed voor geschikt.

Custom header
In sommige gevallen is het nodig om extra header informatie mee te sturen met een HTTP message. De regel hierbij is dat de header met een X moet beginnen. Een voorbeeld hiervan is X-HTTP-OVERRIDE.

Het is niet aan te raden om hier veelvuldig gebruik van te maken. Het kan namelijk voorkomen dat apparatuur waar het request door gecached wordt de custom header van het request afhaald omdat deze niet aan de standaard HTTP regels voldoet.

Status codes en errors
Om een applicatie volgens de HTTP standaard te laten functioneren is het van belang dat ieder request een response met correcte statuscode terug geeft. Deze statuscodes zijn gestandaardiseerd volgens het HTTP en hebben ieder een aparte betekenis. Zie appendix D voor een overzicht van alle statuscodes. Het correct gebruiken van deze codes draagt bij aan de uniformiteit en zorgt ervoor dat de service goed ondersteund wordt.

Een statuscode geeft de staat van het systeem terug. Dit is correct maar niet erg specifiek. Wanneer er bijvoorbeeld verschillende velden van een formulier niet goed zijn ingevuld is het van belang om te weten om welke het gaat en waarom. Het is dus van belang om in het geval van een fout naast de juiste statuscode ook een error met uitleg terug te geven. Dit moet

gebeuren in het gespecificeerde formaat.

Precondition headers
Etags worden gebruikt in combinatie met precondition headers. Een Etag is, zoals eerder aangegeven, een unieke hash representatie van een resource. De Etag is dus voor iedere resource uniek.

Precondition headers[8] worden in een HTTP request toegepast door middel van een If-* header. Dit kan met zowel een If-Match als een If-None-Match header worden gedaan. De If-* header zit in het request en bevat de Etag die in de laatste response van de resource is terug gegeven.

In het geval van een If-None-Match header wordt het request pas uitgevoerd wanneer de Etag in het request niet matched met de Etag van de resource. Wanneer deze wel matched dan wordt er door de server geantwoord met een 304 not modified status. Deze status geeft aan dat er geen veranderingen hebben plaatsgevonden sinds het laatste request.

Het voordeel hiervan is dat wanneer er geen veranderingen hebben plaatsgevonden binnen een resource, de resource vanuit de cache kan worden terug gegeven. Dit ontlast het systeem beduidend.

De if-match header daarentegen, duidt aan dat het request alleen mag worden uitgevoerd wanneer de Etag van de resource matched met die van de if-match header. De If-match header wordt voornamelijk toegepast bij het manipuleren van een resource. Denk hierbij aan een PUT request om een update op een resource uit te voeren. Wanneer er geen match plaatsvindt betekent het dat de resource in de tussentijd veranderd is. In dit geval mag de update dus niet worden doorgevoerd en moet er worden aangegeven dat de precondition gefaald is. Om dit aan te geven wordt er een 412 “Precondition Failed” status code verstuurd.

Voorbeeld
In dit hoofdstuk zijn de verschillende headers en karakteristieken van HTTP berichten besproken. Om een beter beeld te krijgen zijn onderstaand de headers van een GET request en zijn response te zien.

```
GET /album/45 HTTP/1.1
Host:               localhost
User-Agent:         MyBrowser 1.0
Accept:             text/html,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding:    gzip, deflate
Accept-Language:    nl,en-us;q=0.7,en;q=0.3
```

```
HTTP/1.1 200 OK
Date:               Fri, 11 May 2012 13:39:18 GMT
Server:             Apache/2.2.21 (Unix) PHP/5.3.6
Content-Language:   nl
Content-Length:     106
Content-Type:       application/json;charset=utf-8
Etag:               "c06aac804897194d19be00f9b28124de"
```

5.3 Conclusie

In dit hoofdstuk wordt beschreven hoe HTTP hoort te functioneren. Deze informatie is gebaseerd op RFC 2616. Een belangrijke conclusie die hier kan worden getrokken is dat HTTP volledig is gebaseerd op berichten die tekst bevatten. Deze berichten bevatten verschillende headers die de context van het request of response beschrijven. De methode die in een request bericht staat gedefinieerd, geeft aan wat voor actie er op de geadresseerde resource moet worden uitgevoerd.

6 RESTful Applicatie

In de voorgaande hoofdstukken wordt uitgebreid ingegaan op het ontstaan en de theoretische werking van RESTful applicaties. In dit hoofdstuk wordt dieper ingegaan op het praktisch functioneren van een RESTful applicatie.

De hoofdprocessen binnen de REST API worden hier beschreven. Voor ieder van deze processen is een apart onderdeel van de API verantwoordelijk. De structuur die hier wordt beschreven is toe te passen op vrijwel iedere taal of framework. Maar omdat het uiteindelijke product dat voor Ibuildings wordt ontwikkeld, met het Zend framework wordt opgezet, zal als uitgangspunt de globale werking van Zend worden gebruikt.

Omdat REST bestaat uit globale richtlijnen, is er nergens precies vastgelegd uit welke onderdelen een RESTful applicatie moet bestaan. De onderdelen die hier worden beschreven zijn bepaald aan de hand van de werking van HTTP en de ideeën die binnen Ibuildings over REST bestaan. Dit is gebaseerd op het Richardson Maturity Model Level 2, zie appendix A. De namen die aan de componenten zijn gegeven zijn gebaseerd op hun functionaliteit.

Houd er rekening mee dat deze beschrijving alleen de hoofdprocessen in schematische volgorde beschrijft. In werkelijkheid zijn er meerdere onderdelen bij dit proces betrokken en is de volgorde niet zo zwart wit als hier wordt weergegeven. Sommige onderdelen worden namelijk deels voor en deels na de actie op de resource uitgevoerd.

In dit hoofdstuk zal veelvuldig worden verwezen naar eerder in dit document behandelde principes omtrent HTTP en REST.

6.1 De onderdelen van een RESTful applicatie

URI templates
Het is van belang dat voor iedere beschikbare resource URI routes[15] zijn gedefinieerd. Dit zijn templates die de opbouw van de URI beschrijven en aangeven naar welke module en controller, zie hoofdstuk “Zend Framework”, er binnen de applicatie wordt verwezen. Een voorbeeld van een URI template is:

Name = album
URL = /album/{id}
Module = music
Controller = album
Type = RESTroute

Ook wordt hier het type route bepaald. Dit type verwijst naar een beschikbaar route component. Zie volgende onderdeel voor meer uitleg hierover.

REST Route component
Om ervoor te zorgen dat alle resources benaderd kunnen worden met de juiste HTTP methode is het route component ingezet. Voor iedere route wordt er een instantie van dit component gedefinieerd. Een instantie van het route component wordt gedefinieerd in de URI templates. Hier wordt namelijk per gedefinieerde route aangegeven welk route component deze route moet afhandelen.

Om de routes af te handelen is een router nodig die de routes verwerkt. De meeste frameworks bevatten standaard een router. De router zelf doet niet veel meer dan alle gedefineerde routes nalopen totdat de juiste is gevonden. Wanneer dit het geval is dan zal de gevonden route worden uitgevoerd.

Het REST route component heeft als taak de opgegeven URL te vergelijken met zijn gedefinieerde URI en deze aan de hand hiervan naar de juiste module, controller en action door te sturen. De action die op deze controller moet worden uitgevoerd wordt bepaald door de gebruikte HTTP methode. Het is de taak van het route component om al deze onderdelen te verwerken en het request door te sturen naar de juiste locatie. Daarnaast is het ook mogelijk dat er parameters in de URI worden meegeven, zie URI templates, het route component moet deze parameters uit de URI extraheren.

Voorbeeld
Wanneer er een GET request op mijnservice.nl/album/25 wordt uitgevoerd dan zal de router alle gedefinieerde routes gaan nalopen. Zoals in de URI template is gedefinieerd verwijst de route “album” naar de module “music” met de controller “album”. De action die op de controller moet worden uitgevoerd is de getAction. Dit is bepaald aan de hand van de HTTP methode waarmee het request op de resource is uitgevoerd. De parameter id die is meegegeven is 25. Er wordt hierbij dus de getAction methode in de controller album uitgevoerd met de parameter id=25.

Request decoder
Alle data die naar de service wordt gestuurd moet in een bepaald formaat zijn opgemaakt om verwerkt te kunnen worden. De service moet namelijk snappen wat er met de data bedoeld wordt. Het type data waarin een HTTP bericht wordt verstuurd is beschreven in de content-type header.

De Request decoder bepaald[14] of het bericht een body bevat en of het type data hiervan overeenkomt met de content-type header. Wanneer er een body aanwezig is, dan wordt deze aan de hand van het gespecificeerde content-type gedecodeerd naar request parameters.

Voorbeeld
Dit kan handig zijn wanneer er bijvoorbeeld een nieuw item wordt toegevoegd aan de database. De body bevat hiervoor een JSON weergave van het nieuwe item. De request-decoder vormt deze JSON om naar variabelen en voegt deze toe aan het request object. Op deze manier zijn deze variabele beschikbaar tijdens het hele proces.

Responsetype switcher (content negotiation)
Het is van belang om te bepalen welke response-formats[15] de client kan verwerken. Zoals in het hoofdstuk Hypertext Transfer Protocol is beschreven worden de toegestane formaten meegezonden in de accept header van het request. Er moet aan de hand hiervan worden bepaald welke van deze formaten de service ondersteund. Als er een beschikbaar formaat aanwezig is, dan zorgt de responsetype switcher ervoor dat het resultaat in dit formaat wordt teruggegeven.

Om de responsetype switcher te laten functioneren is het van belang dat er een component wordt toegevoegd dat de headers van een HTTP bericht kan ontleden (parsen). Het idee hierbij is dat deze headers in een bruikbaar formaat worden geretourneerd.

De responsetype switcher maakt gebruik van een headerparser om de accept header van het HTTP bericht te ontleden. Deze headerparser geeft een gesorteerde lijst met data types terug. Deze lijst is gesorteerd aan de hand van de prioriteit dat ieder gewenst datatype heeft.

Deze lijst met datatypes wordt afgelopen. Er wordt steeds gekeken of het gewenste datatype volgens de applicatie beschikbaar is. De toegestane datatypes zijn van te voren door de eigenaar van de API geregistreerd.

Het datatype dat uiteindelijk geselecteerd wordt is de vorm waarin het resultaat van het request zal worden geretourneerd.

Voorbeeld
Wanneer er een GET request naar /person/45 wordt gestuurd met in de accept-header application/json en text/xml. De responsetype switcher loopt deze lijst aan de hand van prioriteit af en kijkt of een dergelijk formaat is geregistreerd. Wanneer dit het geval is dan wordt er in het systeem aangegeven dat het te retourneren resultaat in bijv. JSON moet worden teruggegeven.

Precondition helper
Een erg belangrijk onderdeel is de precondition[15] helper. Zoals al eerder in het hoofdstuk HTTP is aangegeven worden preconditions gebruikt om te controleren of de laatste staat van van de resource nog wel gelijk is zijn huidige staat.

Bij ieder GET, HEAD, PUT en OPTIONS request is het mogelijk om gebruik te maken van de precondition helper. Bij deze requests wordt er bij het aanroepen van de resource namelijk een Entity tag gegenereerd. Zoals eerder aangegeven is de Etag een unieke representatie van een resource.

De precondition helper heeft als taak om bij de toegestane requests te controleren of er if-* headers zijn gedefinieerd. Wanneer dit niet het geval is dan zal deze na de aanroep van de resource een Etag header terug sturen.

Deze Etag zal afhankelijk van het request de keer daarop als if-match of if-none-match header worden meegestuurd. Dit kan onder andere gebruikt worden voor caching.

Voorbeeld
Er wordt een GET request uitgevoerd op /person/45. In de response zit een Etag header. De client cached deze Etag en stuurt deze in het GET request mee als if-none-match header. De service weet dan dat als de resource niet veranderd is dat er dan mag worden geantwoord met de HTTP status 304 “Not Modified”. De client herkent dit en laat de respresentatie uit de cache zien.

Action op resource
Dit is de action die gekoppeld zit aan de geselecteerde HTTP methode. Wanneer er bijvoorbeeld een HTTP GET wordt uitgevoerd op resource /album/25 dan betekent dit dat er een GET action wordt uitgevoerd. Het bepalen van de action wordt gedaan in het route component.

De beschikbare acties op de resource worden ook wel CRUD acties genoemd. Zoals al eerder is besproken staat CRUD voor create, read, update and delete. Deze acties komen overeen met de POST, GET, UPDATE en DELETE.

Serializer
De serializer is de laatste stap in het proces. Deze vormt het resultaat van de action om naar het gewenste datatype. Dit datatype is eerder bepaald in de responsetype switcher. Het resultaat hiervan wordt uiteindelijk terug gestuurd in een HTTP response.

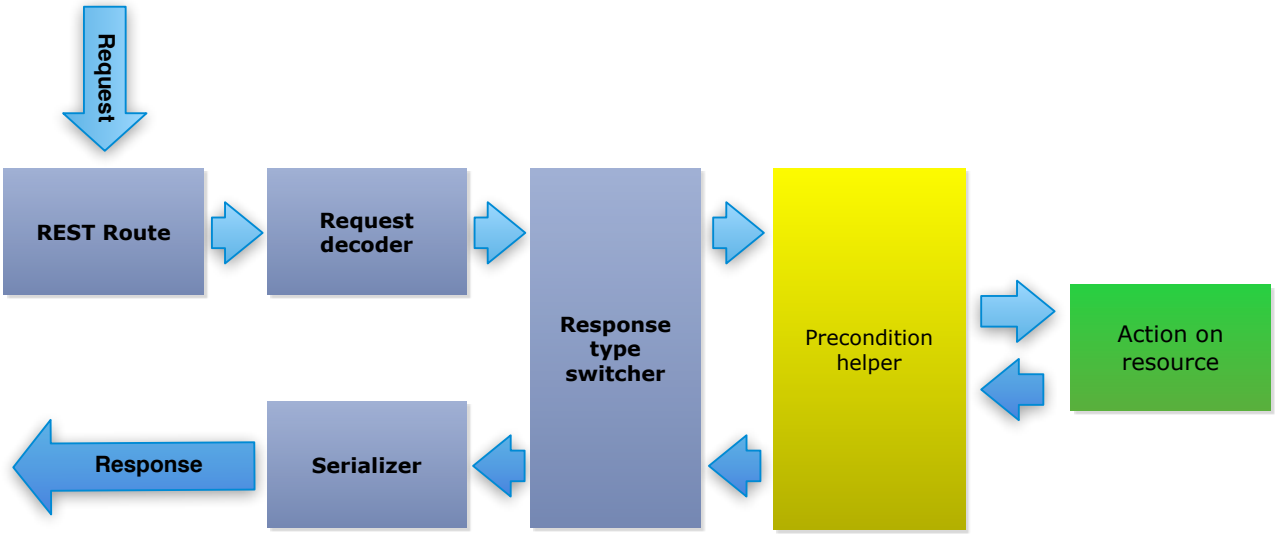
Voorbeeld
Het resultaat van het GET request op /album/25 is: “artiest = Elvis Presley” en “album = almost in love”. Het gevraagde datatype is JSON. Het resultaat wordt naar de serializer gestuurd en hier omgevormd naar JSON. Het resultaat dat in de HTTP message wordt geretourneerd is hierdoor:

```
{
      “artiest”:”Elvis Presley”,
      “album”:”Almost in love”
}
```

Headerparser
Ter ondersteuning van de voorgaand genoemde componenten is een headerparser nodig. Deze headerparser wordt gebruikt om de inhoud van de verschillende verstuurde HTTP headers om te vormen naar bruikbare informatie. Een header bestaat altijd uit een of meerdere entiteiten. Het is de taak van de headerparser om deze entiteiten te extraheren en om te zetten naar objecten die de entiteit representeren.

Voorbeeld
Wanneer de accept header `application/json, tekst/html` naar de server wordt verstuurd dan moet deze worden omgevormd naar een object. In dit geval bevat de header twee entiteiten. Het resultaat van het parse proces is een lijst met twee objecten. Ieder van deze objecten bevat informatie over het gewenste formaat en het type. In het geval van application/json is het type “application” en het formaat “json”.

De werking
Het onderstaande model geeft het proces met bovenstaande componenten grafisch weer. Hierbij zijn de paarse en gele blokken de verschillende componenten en het groene blok de uiteindelijke actie op de resource. Hieruit komt naar voren dat de componenten bestaan ter ondersteuning van de actie op de resource. Ook is hier te zien dat delen van sommige componenten zowel voor als na het uitvoeren van de action worden gebruikt. De headerparser is niet zichtbaar in dit proces. Deze wordt namelijk ter ondersteuning van de request decoder en responsetype switcher gebruikt.

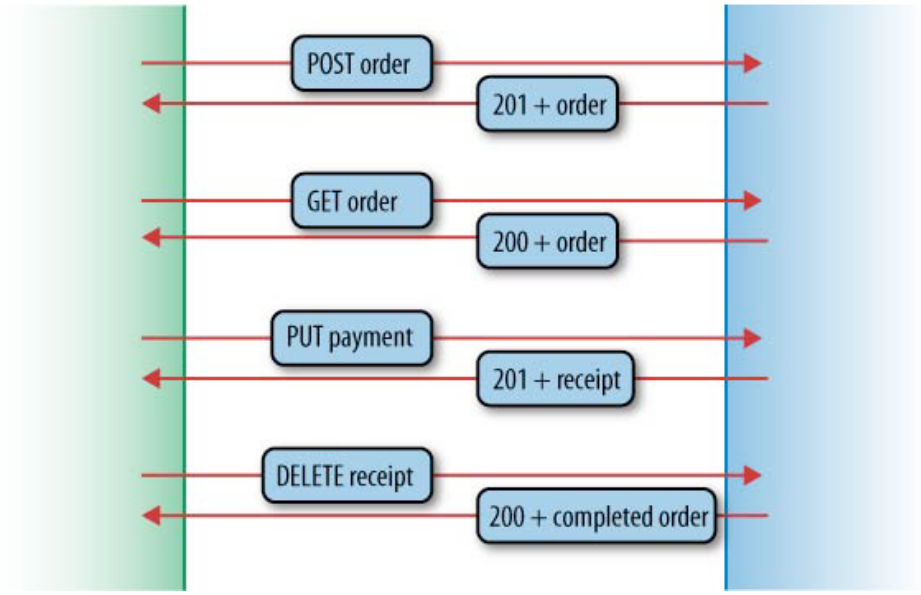


Schematische weergave werking REST API

Het is belangrijk om te beseffen dat niet ieder type request alle componenten gebruikt. Een GET request heeft bijvoorbeeld standaard geen body. De request decoder detecteert dit en zal niet worden uitgevoerd. Hetzelfde geld voor de serializer, in het geval van bijvoorbeeld een HEAD request wordt er geen body geretourneerd. Het is dan dus niet nodig om de Serializer aan te roepen.

6.2 De werking

Ter verduidelijking van het proces dat plaatsvindt met gebruik van een RESTful applicatie wordt hier een voorbeeld gegeven van een bestelling via een RESTful applicatie. Deze stappen zijn verduidelijkt in de onderstaande afbeelding. Op de volgende pagina staan de stappen van deze processen uitgeschreven. Sommige komen op veel punten overeen maar zijn voor de duidelijkheid toch volledig uitgeschreven.



Schematische weergave van aankoop via RESTful applicatie

1. Via een HTTP POST request wordt de order bij de web service geplaatst. Dit POST request heeft een, in dit geval, JSON body die de besteldetails bevat.
    - Het route component zorgt ervoor dat de POST action op de juiste resource wordt uitgevoerd.
    - De request-decoder zorgt ervoor dat de JSON body wordt omgevormd tot bruikbare variabelen.
    - De responsetype switcher bepaalt aan de hand van de accept header in welk formaat de response moet worden teruggegeven.
    - De POST action wordt op de resource uitgevoerd. In de response wordt een location header toegevoegd die de locatie van de nieuw gecreëerde resource bevat. In dit geval is de response code 201, dit houdt in dat de gecreëerde resource in de response wordt meegestuurd.
    - De serializer vormt de response om naar het gewenste data formaat.

2. Er wordt een GET request op de net geplaatste order resource uitgevoerd. Er wordt antwoord gegeven met de 200 status code en een representatie van de order.
    - De route zorgt ervoor dat de GET action naar de juiste resource wordt gestuurd.
    - Een GET request stuurt geen body mee, de Request decoder wordt dus niet aangeroepen.
    - De Responsetype switcher bepaalt aan de hand van de accept header in welk formaat de response moet worden teruggegeven.
    - De precondition helper controleert of er If-* headers zijn gezet. Dit is niet het geval.
    - De GET action wordt op de controller uitgevoerd
    - De precondition helper genereert een Etag en voegt deze aan de response toe.
    - De serializer vormt de response om naar het gewenste data formaat.

3. Er wordt een PUT request op de payment resource van de order gemaakt. Door middel van PUT wordt deze payment geüpdatet. Er wordt geantwoord met een 201 status code en een representatie van een bonnetje resource.
    - De route zorgt ervoor dat de PUT action naar de payment resource wordt gestuurd.
    - In de message body zit een volledige JSON weergave van de geüpdatete payment resource. De requestdecoder vormt deze weergave om naar bruikbare variabelen.
    - De Responsetype switcher bepaalt aan de hand van de accept header in welk formaat de response moet worden teruggegeven.
    - De precondition helper controleert of er If-* headers zijn gezet. Dit is niet het geval.
    - De PUT update wordt uitgevoerd op de resource.
    - De precondition helper genereert een Etag en voegt deze aan de response toe.

- De serializer vormt de response om naar het gewenste data formaat. Er wordt een bonnetje geretourneerd.

4. Er wordt een DELETE request op de receipt resource van de payment gemaakt. Er wordt geantwoord met een 200 statuscode + een weergave van de completed order.
    - De route zorgt ervoor dat de DELETE action naar de juiste resource wordt gestuurd.
    - Het DELETE request stuurt geen body mee, de Request decoder wordt dus niet aangeroepen.
    - De Responsetype switcher bepaalt aan de hand van de accept header in welk formaat de response moet worden teruggegeven.
    - De DELETE action wordt op de controller uitgevoerd
    - De serializer vormt de response om naar het gewenste data formaat.

6.3 Conclusie

In de basis bestaat een REST API uit een route component, request decoder, responsetype switcher, headerparser, precondition helper en een serializer. Ieder van deze componenten bezit de verantwoordelijkheid voor een onderdeel van het HTTP en de URI standaard.

Het route component zorgt ervoor het verzonden request naar een URI ook daadwerkelijk terecht komt bij de juiste resource met de juiste bewerking.

De request decoder maakt gebruik van de content-type header en zorgt ervoor dat de applicatie gebruik kan maken van de data die met het request is meegezonden.

De responsetype switcher kijkt aan de hand van de accept header wat voor formaat de client terug verwacht en zorgt ervoor dat de serializer hiervan op de hoogte is.

De precondition helper bekijkt of er wordt voldaan aan de meegestuurde precondion headers. Wanneer dit niet het geval is dan wordt er geantwoord met een 304 of 412 HTTP status code. Daarnaast zorgt dit component ervoor dat er na een request een Etag header mee wordt terug gestuurd.

De serializer vormt de resultaat data om naar het, door de responsetype switcher, bepaalde formaat.

Aan de hand van deze relatief simpele processen kan een grote variëteit aan acties worden uitgevoerd. Deze componenten representeren de uniforme interface van de applicatie aan de hand van HTTP en kunnen voor vrijwel iedere toepassing worden ingezet.

7 Ibuildings

Zoals in de inleiding van dit document al naar voren is gekomen heeft Ibuildings behoefte aan een constructie waarmee gemakkelijk een client-server architectuur kan worden opgezet. De client zal in het overgrote deel van de situaties bestaan uit een web-browser waarbinnen een Javascript applicatie draait. Deze applicatie draait aan de client kant en moet verbinding maken met een server waarop alle business data/logica zich bevindt.

De communicatie met deze server zal mogelijk worden gemaakt door gebruik te maken van een REST web service. Deze REST web service draait op de server en handelt de communicatie met de client-applicatie af. Het grote voordeel van deze constructie is, dat er zonder veel moeite een extra client bij kan worden gezet. Denk hierbij aan een mobiele applicatie naast de Javascript applicatie die ook verbinding maakt met de REST service.

Alle voorgaande hoofdstukken beschrijven de werking van RESTful applicaties in combinatie met HTTP. Ook is er ingegaan op de verschillende gradaties die binnen REST bestaan. Bij Ibuildings is ervoor gekozen om RESTful applicaties te ontwikkelen op Level 2 van het Richardson Maturity[15] Model, zie appendix A.

Er is voor Level 2 gekozen omdat hierin alle HTTP-standaarden worden nageleefd en dus stabiliteit en uitbreidbaarheid worden gestimuleerd. Het onderdeel dat ontbreekt om de applicatie volledig RESTful te maken is hypermedia. Er is op dit moment voor gekozen om hypermedia (nog) niet te ondersteunen. Dit is gedaan omdat er nog geen duidelijke richtlijnen en voorbeelden bestaan voor het implementeren hiervan

Sinds kort bestaat er binnen Ibuildings een aantal specifieke richtlijnen rondom het gebruiken van RESTful applicaties. Ook deze richtlijnen zijn gebaseerd op Level 2 van het RMM maar meer toegespitst en concreter. Het gaat hierbij om de specifieke punten die een service RESTful maken. Alles dat gebruikt wordt, is toegestaan volgens RFC2616[5]. Er wordt voornamelijk ingegaan waarvoor verschillende HTTP methodes mogen worden gebruikt en hoe de URI's van resources moeten worden vormgegeven.

Daarnaast is er een checklist opgesteld waarmee snel kan worden bepaald of de gebouwde applicatie RESTful is.

Onderstaand zijn de reguirements van Ibuildings beschreven. Al deze punten zijn ook verwerkt in de MoSCoW requirement analyse voor de componenten. Zie appendix E voor deze MoSCoW analyses.

7.1 Requirements

HTTP Methoden

Ibuildings ondersteunt alleen de methoden GET, POST, PUT, DELETE, HEAD en OPTIONS. Deze moeten functioneren zoals gedefinieerd in RFC2616. Ook kan het voorkomen dat verouderde frameworks of mobiele platformen alleen GET en POST ondersteunen. In dit geval moet er gebruik worden gemaakt van de header X-HTTP-Method-Override in combinatie met een POST request.

Er is de keus gemaakt om geen message body voor een GET of DELETE requests te ondersteunen. Dit is niet expliciet verboden maar wordt in principe niet algemeen ondersteund. Het is namelijk mogelijk dat de body van het request door middleware systemen wordt verwijderd omdat dit niet gebruikelijk is.

Resources

In het geval van een REST service kan een resource worden gezien als een service-laag die als façade dient om een of meerdere models te benaderen. De resource dient eigenlijk als interface richting het systeem. Er kan hierbij onderscheid worden gemaakt tussen drie verschillende resource types.

Entity resource

Een resource om een enkel item op te vragen. Denk hierbij aan /order/23. Deze resource verwijst naar het 23e order in de database. Op een entity resource kunnen alleen GET, PUT, DELETE, HEAD en OPTIONS methoden worden uitgevoerd.

Collection resource

Een resource om acties uit te voeren omtrent een collectie. Denk hierbij aan /order. Deze resource verwijst naar de collectie met orders. Op een collection resource kunnen alleen GET, POST, HEAD en OPTIONS methoden worden uitgevoerd.

Behavior resource

Dit zijn normale resources die custom operations uitvoeren wanneer er een post request wordt gedaan. Post is het enige request wat hiermee gebruikt mag worden. Denk hierbij aan POST /order/23/report. Hiermee wordt er een PDF rapport van order 23 gemaakt. Met GET order/23/report kan de laatst gegenereerde worden opgehaald. Maak maximaal gebruik van twee custom operations per resource. Zijn er meer nodig dan moet een subresource worden overwogen.

URI design

Aangezien de REST service wordt aangesproken via een URI is het van belang om goed na te denken over het ontwerp van deze URI's. Het uitgangspunt hierbij is dat de basis URI naar het type resource is genoemd. Hierbij moet het enkelvoudige woord van de resource worden gebruikt. Onderstaand een voorbeeld van URI design. Belangrijk hierbij is dat er gebruik wordt gemaakt van een prefix. In dit geval /company/, dit zorgt ervoor dat er onderscheid kan worden gemaakt tussen verschillende groepen. Het zou namelijk voor kunnen komen dat person in zowel /company en /client bestaat.

```
HTTP Method    URI                    Description
-----------    ---                    ----------------
GET            /company/person        list members
POST           /company/person        create member
GET            /company/person/1      retrieve member
PUT            /company/person/1      update member
DELETE         /company/person/1      delete member
```

Een belangrijk punt hierbij is dat de URI nooit moet worden ontworpen voordat de resources zijn bepaald. De URI's moeten namelijk een resultaat worden van de resource structuur.

Subresources

Kunnen goed worden ingezet, het is wel van belang dat dit goed overwogen en met mate gebeurt. Een goed voorbeeld hiervan is GET /article/42/comments. Deze vraagt alle comments op die aan article 42 zijn gekoppeld. Zoals eerder is aangegeven moet bij het implementeren van een subresource altijd goed worden nagegaan of deze het systeem niet te ingewikkeld maken. Een subresource die twee id's bevat is over het algemeen verkeerd. Het is dan namelijk zo dat deze weer doorverwijst naar een andere toplevel resource. Denk hierbij aan /article/42/comments/79. Het is in dit geval beter om deze op te splitsen in twee delen. /comments/79 en /article/42.

Output

Binnen Ibuildings is bepaald dat de standaard output van een REST service in JSON moet zijn. Het is van belang dat naast de JSON output een correcte statuscode wordt meegegeven. Deze

statuscode moet overeenkomen met de status van het request.

Authenticatie

Het is natuurlijk van belang dat alleen gebruikers met de juiste rechten contact kunnen leggen met de REST service. Hiervoor moet gebruiker authenticatie plaatsvinden. Volgens de guidelines van Ibuildings mag hier alleen gebruik worden gemaakt van bestaande authenticatie systemen. Voorbeelden hiervan zijn Basic en Digest.

- Basic authentication
  De basic acces authentication is een methode om gebruikersnaam en wachtwoord door te sturen via HTTP. Voordat deze wordt verstuurd, worden het wachtwoord en gebruikersnaam gescheiden door een ":", versleuteld door middel van het base64 algoritme. Deze versleuteling vindt niet plaats vanwege beveiliging. Base64 is namelijk zeer makkelijk te ontsleutelen. De versleuteling vindt plaats om er zeker van te zijn dat er geen karakters in het HTTP request worden meegestuurd die niet HTTP-compatible zijn.

- Digest
  Digest acces authentication is net als Basic een methode om gebruikersnaam en wachtwoord door te sturen via HTTP. Digest kan worden gezien als een evolutie van Basic. Hierbij wordt door middel van MD5 de gebruikersnaam + salt + wachtwoord gehasht. Dit wordt weer gecombineerd met de methode en URI.

Authenticatie zal in dit document niet verder worden behandeld. Dit omdat het buiten de scope van het afstudeerproject valt.

Versies

Het kan voorkomen dat er meerdere versies van een REST service bestaan. Om een versie van een service aan te geven mag geen parameter worden gebruikt. De versie moet altijd worden aangegeven in de MIME type. Het is gebruikelijk om deze versie in de Accept header mee te gegeven. Een voorbeeld hiervan is `application/vnd.ibuildings.blog-v2+json` deze Accept header geeft aan dat het hier gaat om de Ibuildings blog versie 2 met het formaat JSON.

Error handling

Error handling is een zeer belangrijk onderdeel van een RESTful applicatie. Iedere actie die wordt uitgevoerd wordt namelijk beschreven door middel van een HTTP status code en een beschrijving. In het geval van een error moeten deze statuscodes en het bericht dus ook precies aangeven wat er mis gaat binnen de service.

De error/statuscodes afhandeling moet worden geïmplementeerd volgens RFC 2616. Er moet hierbij aan de standaarden worden voldaan.

7.2 Conclusie

De requirements die Ibuildings aan de REST API skeleton stelt kunnen worden gezien als een concretisering van het Richardson Maturity Model level 2. Het gaat hier voornamelijk om het gebied van URI design. Daarnaast zijn er richtlijnen gesteld aan het gebruik van custom methods en headers. Een ander belangrijk punt is het gebruik van versienummering, er is voor gekozen om de versie van de REST web service mee te geven aan het mediatype in de accept header.

Alle specifieke functionele eisen zijn verwerkt in MoSCoW analyses bijgevoegd in appendix E.

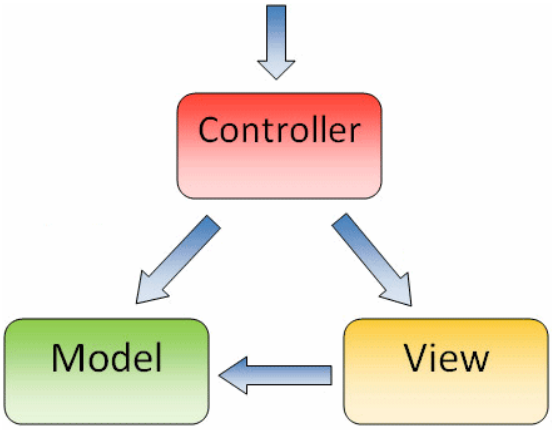
8 Zend framework

Er is voor gekozen om de gewenste REST API met het Zend Framework 1.1x te gaan ontwikkelen. De keus is hierbij gevallen op het Zend Framework 1.1x omdat dit tegenwoordig het standaard server-side framework binnen Ibuildings is. Ibuildings heeft hiervoor gekozen omdat Zend standaard een hoop veelgebruikte functionaliteiten bevat en omdat Zend gebruikmaakt van een uitgebreide variant van het MVC design-pattern. Deze MVC structuur is zeer belangrijk voor het ontwikkelen van de REST API. Het draagt namelijk bij aan een stabiele en schaalbare opzet van de applicatie. Zend framework is opgebouwd met PHP en bevat hierdoor volledige ondersteuning voor HTTP en URI's.

De Zend REST API
Binnen het Zend framework is standaard een REST API module geïntegreerd. Het probleem met deze API is dat deze erg beperkt is. Zo wordt er altijd uitgegaan van XML output en ondersteund deze alleen GET en POST. Daarnaast is de module erg slecht gedocumenteerd. Al met al is de standaard API een incompleet component waarbinnen een zeer beperkende structuur is verwerkt.

Vanwege deze structuur en onvolledigheid heeft Ibuildings de voorkeur gegeven aan het volledig opnieuw ontwikkelen van een REST API. Deze keuze is al gemaakt voor de start van het REST API project. In deze scriptie zal ook niet verder worden ingegaan op deze keuze.

Model View Controller
Een groot voordeel van het Zend Framework is de uitgebreide implementatie van het design-pattern model view controller[19]. Deze structuur zorgt ervoor dat er binnen de code gemakkelijker de design-principes: separation of concerns, loose coupling en strong cohesion kunnen worden toegepast.



Het MVC design pattern

Hiernaast is in de afbeelding de structuur van het model view controller pattern te zien. Het idee[9] hierachter is dat alle code zit onderverdeeld in controllers, models en views. De model bevat alle hoofd applicatie logica. De view bevat alle weergave logica. De communicatie tussen de view en de model vindt plaats via de controllers. Zoals in de afbeelding is te zien komt het request via de controller binnen. Iedere controller bestaat uit verschillende actions die ieder een model en een eigen specifieke view aanspreken. De controller zorgt er dus voor dat door middel van actions informatie wordt opgehaald uit de model en wordt doorgezet naar de view.

Zend heeft ervoor gekozen om het MVC pattern uit te breiden om er optimaal gebruik van te kunnen maken, zie appendix C. Een groot verschil is dat Zend voor de controllers nog een enkele front-controller heeft geplaatst. De front-controller initialiseert mogelijke extra plugins / action helpers en roept een router aan. Deze router bekijkt aan de hand van de verstuurde URL welke controller met de bijbehorende action moet worden aangeroepen.

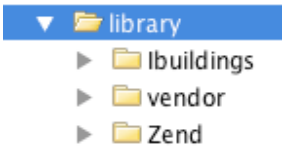
Zoals eerder genoemd ondersteunt MVC een aantal design-principes[9]. Deze principes zijn onderstaand beschreven.

Separation of concerns houdt in dat code met gelijksoortige functionaliteit gegroepeerd en gesorteerd wordt. Strong cohesion en loose coupling hangen hier sterk mee samen. Strong cohesion houdt namelijk in dat de gegroepeerde code samenhangend moet zijn en aan elkaar gerelateerde taken moet uitvoeren. Loose coupling houdt in dat componenten onderling weinig tot niet afhankelijk van elkaar zijn.

Dit houdt in dat de applicatie zo moet worden ontworpen dat er een duidelijke samenhang zit tussen de gegroepeerde code en hun functionaliteit. Wanneer er op deze manier wordt gewerkt zal er alleen een minimale hoeveelheid aan onderlinge koppeling tussen de verschillende componenten bestaan. In de praktijk houdt dit in dat een applicatie stabieler functioneert omdat de kans op fouten tijdens de ontwikkelfase kleiner is. Daarnaast is deze uiteindelijk beter beheersbaar omdat het duidelijk is waar de verantwoordelijkheden binnen de applicatie liggen.

Om de opbouw van de applicatie nog verder te structureren wordt er gebruik gemaakt van namespaces. Namespaces kunnen worden gezien als mappen waarin applicatie klassen worden verpakt. Binnen Zend is dit ook echt een fysieke mappenstructuur waarin PHP klassen gesorteerd zijn opgeslagen.

Library
Ieder Zend Framework project bevat een library. In deze library staat de applicatie code van het Zend framework onder de namespace Zend. In deze library kan door de ontwikkelaar zelf ook een namespace worden toegevoegd. In deze library wordt de zelf ontwikkelde code toegevoegd. In het hiernaast weergegeven figuur is te zien dat er onder library een namespace Ibuildings is toegevoegd. Onder Ibuildings staan alle action helpers plugins en klassen die door de programmeur ontwikkeld zijn.



Library in Zend framework

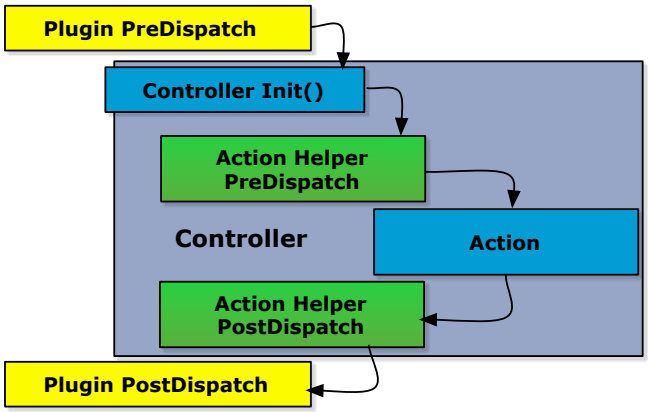
Deze ontwikkelde klassen kunnen vanuit de controller worden aangeroepen. Action helpers en plugins worden door de frontcontroller uitgevoerd. Zie paragraaf 8.1 voor meer informatie hierover.

8.1 Request afhandeling

De request afhandeling binnen het Zend Framework 1.1x is vrij complex maar toch zeer gestructureerd. Er wordt gewerkt met verschillende fases waaruit ieder request handling proces is opgebouwd. Zie Appendix C voor een schematische weergave van het volledige proces. Hierin is te zien dat een request binnenkomt bij index.php. Index.php start de initialisatie van het bootstrap proces. Het bootstrap proces voert alle gewenste instellingen door en registreert alle plugins en action helpers bij de frontcontroller[19]. De frontcontroller start het dispatch proces. Dit dispatch proces bestaat uit verschillende niveaus. Ieder van deze niveaus kan worden onderverdeeld in drie fases.

- PreDispatch
  Wordt voor het hoofdproces uitgevoerd
- Dispatch
  Voert het hoofdproces uit
- PostDispatch.
  Wordt na het hoofdproces uitgevoerd

Verbonden met deze fases zijn de plugins, controllers, action helpers en actions. Ieder van deze onderdelen hebben weer een relatie met een specifiek moment in het dispatch proces. De structuur hiervan is onderstaand weergegeven in een model. Dit model is zichtbaar op de volgende pagina. Hierin komt duidelijk naar voren wat de rangorde binnen het Zend dispatch proces is. Door middel van de pijlen wordt aangegeven in welke volgorde de uitvoering plaatsvindt.



Zend dispatch proces

Als eerste wordt de preDispatch methode van de plugin uitgevoerd. Hierna wordt direct de controller geinitialiseerd. Tijdens deze initialisatie wordt de init methode uitgevoerd. Na de init wordt de preDispatch van de action helper uitgevoerd. Direct hierop volgend de action, dit kan worden gezien als het hoofdproces. Na de action komt de action helper postDispatch met daaropvolgend de postDispatch van de plugin.

Zoals is te zien wordt er een zeer duidelijke opbouw aangehouden. Het is van belang om dit proces goed voor ogen te hebben tijdens het ontwikkelen van applicaties. Daarnaast geeft dit weer wat binnen de controller wordt uitgevoerd en wat daarbuiten. Alles wat binnen de controller bestaat is namelijk beschikbaar in de action.

Action helper
Zoals al eerder in dit hoofdstuk is aangegeven bevat een action helper mogelijkheden om processen voor en na de action uit te voeren. Ook is de helper beschikbaar in de action. Methodes uit de helper kunnen hierdoor worden gebruikt vanuit de action. Dit is erg handig aangezien de action helper al tijdens het bootstrap proces wordt geinitialiseerd. Hierdoor is de action helper door de gehele applicatie beschikbaar. Data die tijdens de bootstrap wordt toegegevoegd aan de action helper is hierdoor beschikbaar in iedere action.

Plugin
Een plugin is bijna exact gelijk aan een action helper. Ook deze heeft een pre en post dispatch methode. Het verschil zit hem in het moment van uitvoeren. De plugin, zie het Zend dispatch proces model, wordt voor en na de controller uitgevoerd. Hierdoor is de plugin niet beschikbaar tijdens de action.

Request en Response object
Tijdens het bootstrap proces wordt het ontvangen HTTP request omgevormd naar een Zend HTTP request object. Dit object is beschikbaar door de gehele applicatie en bevat alle informatie van het HTTP request.

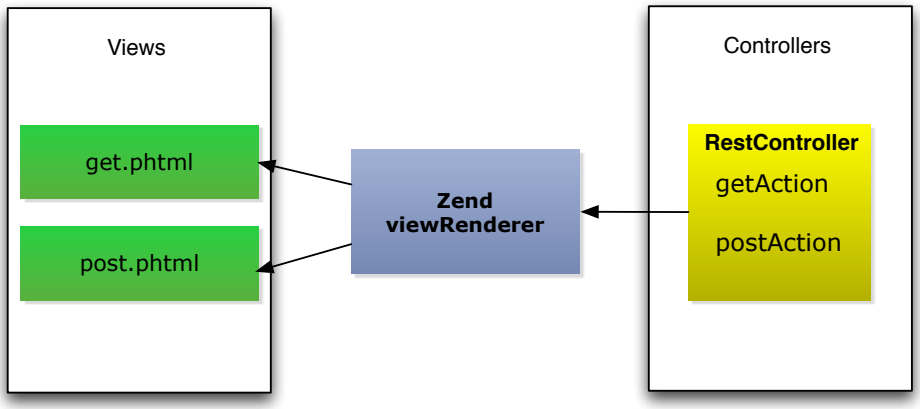
Een soortgelijk proces vindt plaats met de response. Tijdens de bootstrap wordt ook een Zend response object aangemaakt. Aan dit object wordt gedurende de uitvoer van de applicatie informatie en data toegevoegd. Aan het einde van de uitvoer wordt dit response object omgevormd tot een HTTP response die richting de client wordt gestuurd.

8.2 Het view rendering proces

Om het resultaat van een action weer te geven aan de client is de ViewRenderer helper[20] in Zend Framework geïntroduceerd. Het ViewRenderer object zorgt ervoor dat het resultaat van de action op een correcte manier in het juiste view-script wordt weergegeven. Dit view script geeft het resultaat van de action weer.

In het onderstaande model wordt op schematische wijze weergegeven hoe het framework gebruik maakt van verschillende view-scripts. De Zend ViewRenderer is tijdens dit proces

verantwoordelijk voor het selecteren van het juiste view-script behorende bij de action. De ViewRenderer is een Zend action helper die tijdens het postDispatch, dus na de action, wordt uitgevoerd. Hierin wordt ook bepaald welke extensie het view-script moet hebben. Standaard is dit .phtml.



Schematische weergave van Zend viewRenderer

8.3 Conclusie

Er is voor gekozen om voor de ontwikkeling van de REST API Skeleton gebruik te maken van het Zend framework. Binnen Ibuildings bestaat hier veel ervaring mee en wordt hier tegenwoordig als standaard framework gebruikt.

Zend werkt op een gestructureerde manier en bevat een aantal belangrijke onderdelen en processen. De router is hier een van, deze zorgt ervoor dat het request bij de juiste controller en action terecht komt. Daarnaast is er de frontcontroller waar action helpers en plugins kunnen worden geregistreerd. Ook zorgt de frontcontroller ervoor dat het dispatch wordt doorlopen. Dit dispatch proces zorgt ervoor dat er zeer precies kan worden bepaald wat op welk moment moet worden uitgevoerd. Dit kan zijn voor tijdens of na een uitgevoerde actie.

Een ander belangrijk onderdeel is het viewrendering proces. Deze zorgt ervoor dat het resultaat van de action door het juiste view script wordt weergegeven.

9 Implementaties

In dit hoofdstuk wordt ingegaan op de ontwikkelde componenten, hierbij worden afwegingen die zijn gemaakt tijdens het ontwikkelen van de REST API skeleton per component uitgelegd en verantwoord. Tijdens de ontwikkeling van de REST API is gebruik gemaakt van Unit Testing. Iedere functionaliteit binnen de componenten is aan de hand hiervan getest.

Componenten
In de volgende paragrafen worden de ontwerpafwegingen van de verschillende componenten uit de REST API Skeleton beschreven. Er zal hierbij dieper worden ingegaan op bepaalde onderdelen dan andere. Dit heeft te maken met de mate waarin dit component van belang is voor het functioneren van de API. Belangrijke onderdelen zullen vanzelfsprekend meer aandacht krijgen.

Onderdelen die niet kritiek zijn voor het functioneren van de API zullen onder het kopje “Overige componenten” worden behandeld. In de beschrijving zal veelvuldig worden gerefereerd naar termen en principes die in voorgaande hoofdstukken zijn beschreven.

De componenten zijn beschreven op volgorde van ontwikkeling. Dit is niet de chronologische volgorde van uitvoer die in hoofdstuk 6 wordt beschreven. De componenten zijn op deze volgorde ontwikkeld omdat sommige eerder nodig waren in lopende projecten dan andere. Hierdoor kregen deze een hogere prioriteit. Het serializer component bestond al binnen Ibuildings. Deze zal daarom ook niet behandeld worden in dit hoofdstuk.

Voor een specifieke opsomming van de eisen kan worden gekeken naar de MoSCoW analyses uit appendix E. Hierin staat aan welke voorwaarden de componenten moeten voldoen.

De alpha fases die in hoofdstuk 2 “Methodieken” zijn beschreven zullen niet zichtbaar zijn in dit hoofdstuk. Er wordt hier namelijk het complete eindproduct met zijn overwegingen beschreven.

9.1 Rest route

Binnen het Zend Framework zit, zoals in het hoofdstuk Zend Framework beschreven, een router geïmplementeerd. De verantwoordelijkheid van deze router is: het request dat via de URI binnen komt door te sturen naar de juiste module/controller/action. De router kan hiervoor verschillende route klassen inladen. Iedere route klasse bevat een match en een assemble methode.

De match methode bevat applicatie logica om de binnengekomen URI te vergelijken met een voor gedefinieerde URI. Aan de hand hiervan wordt bepaald welke module/controller/action moet worden aangeroepen.

De routes die standaard in het Zend Framework zitten geïmplementeerd zijn puur gericht op website implementaties. Het werken met URI’s in combinatie met REST heeft, zoals in de voorgaande hoofdstukken is beschreven, een iets andere benadering.

Aangezien de REST API, die gaat worden ontwikkeld, moet voldoen aan het Richardson Maturity Model Level 2 is het van belang dat er vanuit resources wordt gewerkt. Er moet dus op iedere resource zowel een GET, PUT en DELETE actie kunnen worden uitgevoerd. De standaard route binnen Zend gaat er vanuit dat in de voorgedefinieerde route precies staat aangegeven naar welke controller en action er moet worden verwezen.

In het geval van de REST route moet de action worden bepaald aan de hand van de gebruikte HTTP methode. Daarnaast moet het mogelijk zijn om deze HTTP action te overschrijven met een X-HTTP-Method-Override header of _method parameter in de URL. Een belangrijke eis hierbij is dat de in eerste instantie gebruikte HTTP methode POST moet zijn, om de HTTP methode in de route te kunnen overschrijven.

Aangezien er binnen sommige voorgedefinieerde routes ook staat aangegeven dat er parameters in de URI moeten worden meegegeven is het van belang dat deze parameters worden geïdentificeerd en geëxtraheerd.

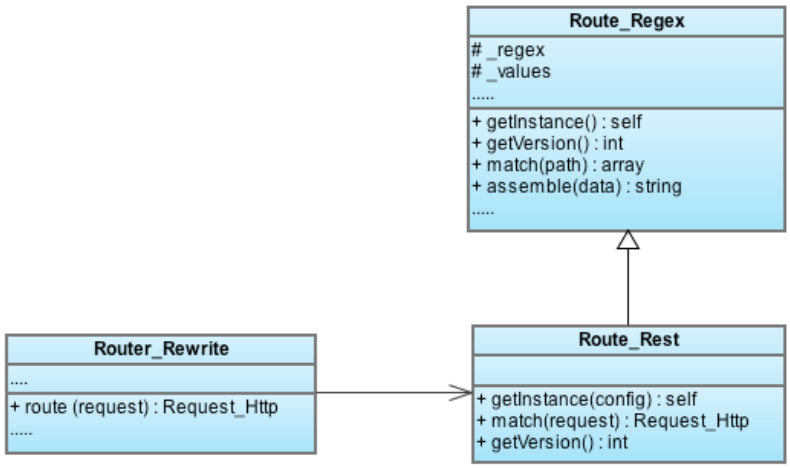
In het ontwerp van de REST route werkt het zetten van de bepaalde action (POST, GET, etc.) net iets anders dan in de standaard route. Tijdens een standaard Zend routing wordt er aan de route klasse alleen een URI in de vorm van een string meegegeven. De match functie controleert of de URI overeenkomt met de gedefinieerde route. Dit gebeurt aan de hand van een regular expression vergelijking. Wanneer dit niet het geval is wordt er “false” geretourneerd. Komen de URI’s overeen dan wordt er gecontroleerd of er parameters binnen de URI aanwezig zijn. Deze worden geëxtraheerd en geretourneerd in de vorm van een array. De router plaatst in dit geval de voor gedefinieerde controller, action en parameters binnen het HTTP request object.

De nieuwe REST route moet dus een action kunnen bepalen aan de hand van de gebruikte HTTP methode. Op basis hiervan moet de action + parameters worden vastgezet zodat de juiste action (POST, GET, etc.) wordt aangeroepen en dat de parameters beschikbaar zijn binnen deze action.

Er moet dus voor worden gezorgd dat het zetten van de action en parameters in het request object niet meer in de router plaatsvindt maar binnen de route zelf. Een probleem hierbij is dat de router gewend is om een URI string aan de route te geven en geen request object.

De oplossing van dit probleem zit hem in de combinatie van de router en het request. De router controleert namelijk, voordat deze de match-controle gaat uitvoeren, wat voor versie de route vertegenwoordigt. Standaard is dit versie 1. Wanneer de route een andere versie vertegenwoordigt geeft de router een HTTP request object aan de route in plaats van een URI string.

De route rest klasse
Zoals voorgaand is besproken is het van belang dat de Route_Rest klasse op een juiste manier gebruikmaakt van de al bestaande router. In het onderstaande diagram is deze structuur te zien. Hierbij zijn Router_Rewrite en Route_Regex native klassen van Zend. Hierin is te zien dat Router_Rewrite (de router) een instantie van Route_Rest maakt en de match methode aanroept. Voor de duidelijkheid zijn de methoden en attributen die niet van toepassing zijn binnen de klassen weggelaten, zie de puntjes.



Klassen diagram structuur rest route

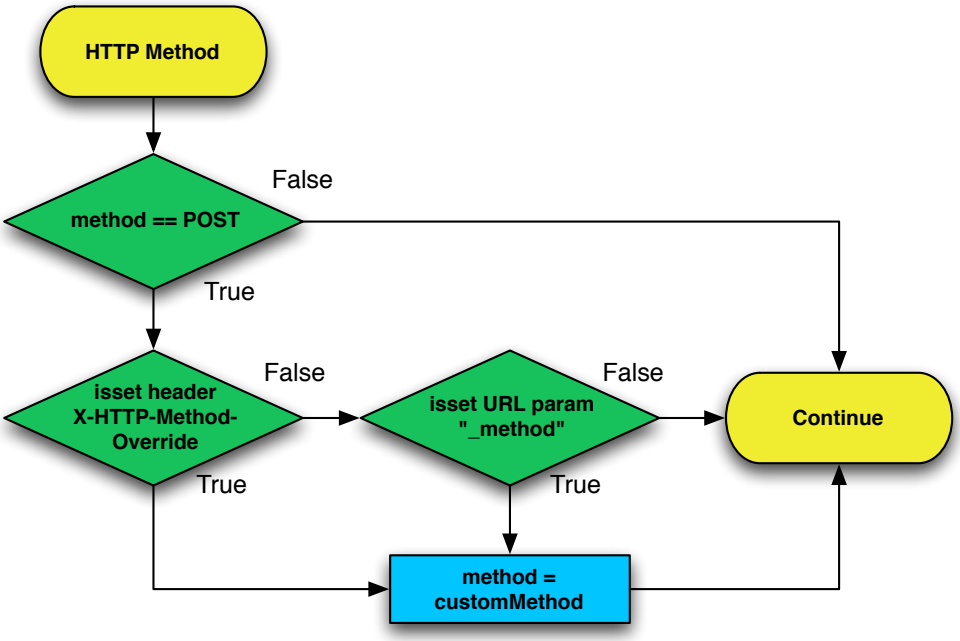
Binnen de structuur wordt gebruik gemaakt van inheritance. Dit is te zien aan de relatie tussen Route_Regex en Route_Rest. Door middel van inheritance, ook wel overerving genoemd,

worden alle methoden en attributen van Route_Regex overerft. Dit is handig want in sommige delen van de klasse komen de routes exact overeen. De onderdelen die worden overschreven zijn de methoden: getInstance, getVersion en match. De methode getInstance heeft als enige functie om een instantie van de klasse terug te geven. Deze methode wordt aangeroepen vanuit de router. De getVersion methode is eerder in dit hoofdstuk al genoemd. Deze heeft als taak versienummer 2 terug te geven. Dit nummer zorgt ervoor dat de router weet dat er moet worden gewerkt met een request object in plaats van een URI string.

De methode match
Zoals eerder is beschreven ontvangt de match methode een instantie van de klasse Zend_Controller_Request_Http. Dit object bevat alle informatie over het HTTP request dat door de client is uitgevoerd. Het is aan de match functie om te bepalen naar welke controller en action het request moet worden doorgestuurd. Hierbij is het van belang dat de meegegeven parameters uit de opgegeven URI worden geëxtraheerd.

Matchen
Het eerste wat hierbinnen plaatsvindt, is het vergelijken van de URI uit het request ten opzichte van de voor gedefinieerde route. De URI van een voor gedefinieerde route kan er als volgt uitzien: `/articles/(?P<id>[0-9]+)` hierin is te zien dat een URI met /articles/ een parameter moet bevatten die alleen uit getallen mag bestaan. De referentie naar deze parameter kan worden gedaan met de naam id. Tijdens deze match worden de parameters direct bepaald en in een array geplaatst.

Request method
Na het matchen wordt er gekeken met wat voor een HTTP request method het request binnen is gekomen. Er vindt dan direct een controle plaats of de methode die gebruikt wordt gelijk is aan POST. Is dit het geval dan bestaat er de mogelijkheid dat er een custom header wordt gebruikt. Hier wordt dan ook direct een controle op uitgevoerd. In de onderstaande flowchart is het proces van deze controle te zien. Wanneer er een custom method wordt gevonden dan wordt de request methode overschreven met de custom method.



Controle op custom headers

Afhandeling
Wanneer de methode is bepaald dan wordt deze samen met de parameters op een stack geplaatst. Deze stack wordt vervolgens afgelopen en iedere waarde wordt toegevoegd aan het Zend HTTP request object. Wanneer dit allemaal is afgerond geeft de match methode het HTTP request terug aan de router.

9.2 HTTP header parser

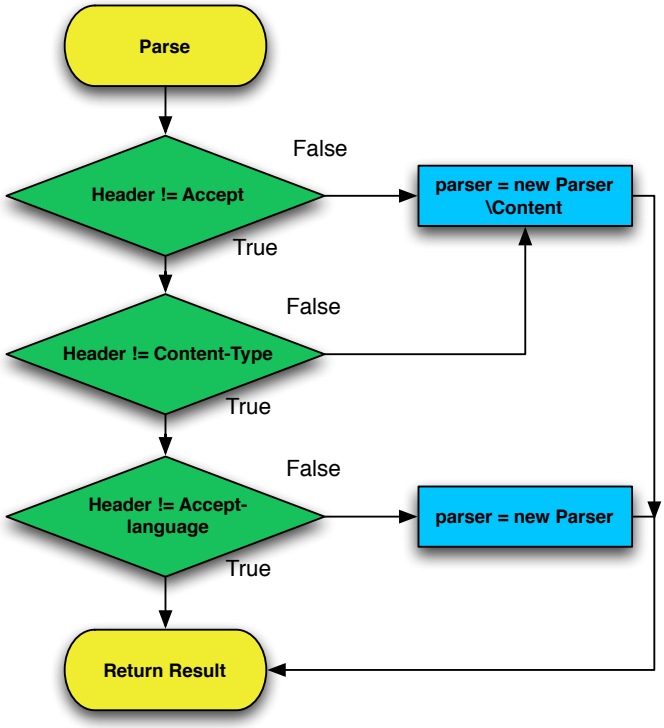
In het HTTP request worden verschillende headers meegestuurd. Ieder van deze headers bevat informatie over het request. Een beschrijving over verschillende van deze headers staat in paragraaf 5.2 van het hoofdstuk “Hypertext transfer protocol”.

Het is van belang dat deze headers uitgelezen kunnen worden en de informatie hieruit kan worden toegepast. De inhoud van de headers is opgebouwd volgens een gedefinieerde structuur, deze structuur is gedefinieerd in RFC 2616[5].

Om de headers te kunnen gebruiken moeten deze worden verwerkt door een parser die per specifieke header alle relevante informatie extraheert. Een veelgebruikte header is de accept header. Deze header bevat de datatypes, met prioriteit, die de client accepteert. Een voorbeeld van een accept header is: `application/json;q=0.8,tekst/html,application/xml`. Hieruit komt naar voren dat tekst/html de voorkeur heeft daarna application/xml en mocht dat niet beschikbaar zijn dan application/json. Het is de bedoeling dat de parser alle data uit de header haalt en iedere entiteit omzet naar een object. Als er meerdere entiteiten binnen een header staan dan wordt er een collectie met objecten geretourneerd. Ieder object moet alle informatie van het datatype, ook wel entiteit genoemd, bevatten.

De header parser is ontwikkeld met uitbreidbaarheid in het achterhoofd. In eerste instantie worden alleen de accept, content-type en accept-language header ondersteund. Zie appendix B voor het klassendiagram van het header parser component. In dit klassendiagram zijn de verschillende namespaces en klassen te zien. De opbouw komt hier duidelijk uit naar voren.

De klasse HeaderParser is de interface voor het parsen van de verschillende headers. Het design pattern Factory Method[9,10] is hierbinnen toegepast. De static methode “parse” krijgt tijdens het aanroepen twee parameters mee. Deze parameters bevatten het type van de header en de inhoud hiervan. Volgens Factory Method wordt aan de hand van het header type de juiste parser geïnstantieerd. Onderstaand is de flowchart hiervan te zien. Hierin is zichtbaar dat er afhankelijk van het type header een type parser wordt geïnitieerd.



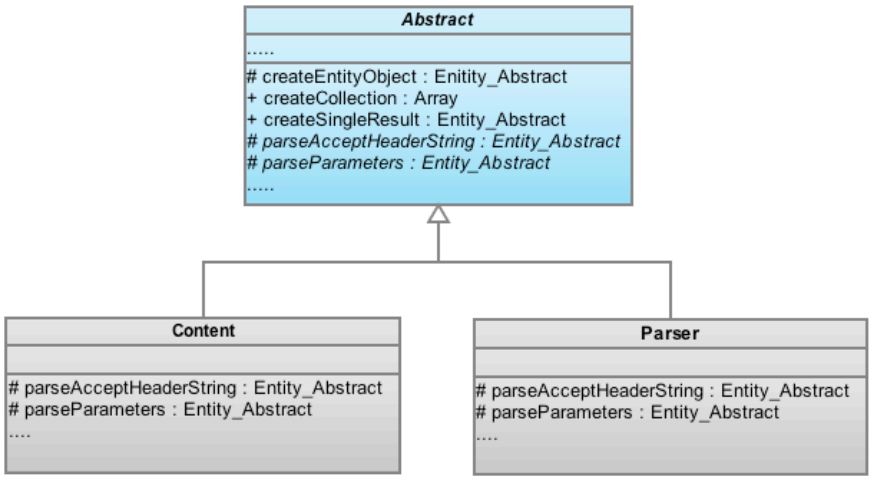
Factory Method parser creation

Als de parser is aangemaakt dan wordt de header hieraan meegegeven. Afhankelijk van het

type header wordt er gevraagd om een collectie met datatypes of een enkel datatype. In het geval van een accept header wordt een collectie opgevraagd. Het principe wat hierin wordt toegepast heet subtype polymorfisme[10]. Het is namelijk zo, zie klassendiagram of onderstaande afbeelding, dat de parser is opgebouwd volgens een parent-child inheritance structuur. Alle parsers stammen hierbij af van een abstracte parent klasse. Deze abstracte klasse fungeert als een uniforme interface voor alle parsers.

Deze structuur verzekert dat alle methoden die de abstracte klasse bevat in iedere parser geïmplementeerd zit. De gedachte hierachter is dat het niet belangrijk is van wat voor type het object is, het is immers zeker dat deze afstamt van de abstracte klasse. Er is hierdoor bekend dat deze in ieder geval de methode createCollection en createSingleResult bevat.

Zie onderstaande diagram voor een versimpelde weergave van deze structuur. Hierin is te zien dat de klassen namen die in het flowchart staan beschreven terugkomen.



Factory klassen structuur

Dit ontwerppatroon is ook gebruikt bij de verschillende entiteit klassen en is goed vergelijkbaar met de bovenstaande tekening. Een instantie van een entiteit is een representatie van een geëxtraheerd stuk data uit een HTTP header. Ook hier bestaat een abstracte basis klasse waarvan alle entiteit klassen afstammen. Zie het klassendiagram in appendix B voor een uitgebreide weergave.

De abstracte Parser klasse

Zoals eerder is aangegeven wordt er bij de parsers gebruik gemaakt van een abstracte basis klasse waarvan iedere specifieke parser klasse de functionaliteit overerft. Er is gekozen voor deze structuur om maximale flexibiliteit te behouden. Het kan namelijk voorkomen dat er andere headers moeten worden ondersteund die een volledig afwijkende structuur bevatten. Met deze opzet is het eenvoudig om met weinig code een nieuwe parser te bouwen en toe te voegen.

De methodes createCollection en createSingleResult

De werking van de daadwerkelijke parser kan worden geïnitieerd door zowel de methode createCollection als createSingleResult. Het verschil in functionaliteit wordt eigenlijk al duidelijk door de naamgeving.

CreateCollection is opgezet om een header met meerdere entiteiten te kunnen parsen. Het resultaat van deze methode is een array waarin de verschillende entiteit objecten op prioriteit zijn geplaatst. CreateCollection zorgt ervoor dat de headerstring wordt opgedeeld in strings met ieder een individuele entiteit.

CreateSingleResult is opgezet voor de headers die maar een enkele entiteit kunnen bevatten. Het zou in dit geval onnodig zijn om een enkel resultaat te retourneren in een array. CreateSingleResult parsed daarom de header naar een enkel entiteit object.

Het daadwerkelijke parsen vindt plaats in de createEntityObject methode. Hierin wordt de entiteit van de header gescheiden in mediatype en parameters. Voor zowel het mediatype als de parameters wordt een specifieke parse methode aangeroepen. Deze specifieke methodes zijn in iedere parser klasse opnieuw geïmplementeerd. Dit omdat de opbouw van de headers steeds verschilt.

De abstracte entiteit klasse

Ook de entiteit klassen zijn al eerder genoemd. De opbouw is vergelijkbaar met die van de parser. Ook hier wordt gebruik gemaakt van een abstracte basis klasse waar specifiekere klassen de functionaliteit uit overerven. Dit maakt het mogelijk om wanneer het nodig is een entiteit klasse te gebruiken die veel gedetailleerde informatie bevat of juist niet. Voorbeelden hiervan zijn de geïmplementeerde entity Mediatype en Entity klassen. Zie appendix B voor het klassendiagram.

De methode getDetailRank

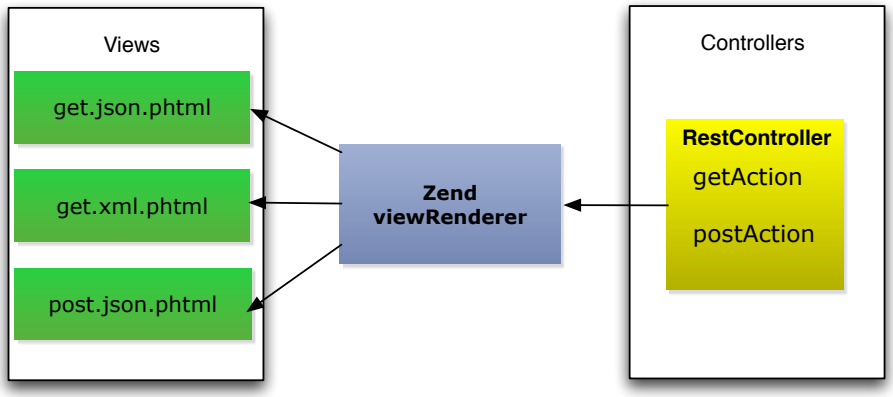
Een belangrijke methode die ook weer in iedere entiteit moet worden geïmplementeerd is getDetailRank. GetDetailRank controleert hoe gedetailleerd de enkele entiteit is beschreven. Er wordt hiermee eigenlijk gekeken hoeveel informatie er bekend is binnen het entiteit object. Deze methode wordt aangeroepen wanneer moet worden gekeken welke entiteit de hoogste prioriteit heeft. Het is namelijk zo dat hoe meer gespecificeerd een entiteit is, des te meer prioriteit deze krijgt om gebruikt te worden, RFC2616 [5].

9.3 Responsetype switcher

Zoals al eerder is aangegeven in dit document is de responsetype switcher verantwoordelijk voor het bepalen van het gewenste data formaat. Dit data formaat staat voor de vorm waarin de response aan de client moet worden geretourneerd. Dit kan bijvoorbeeld JSON, XML of HTML zijn. Het gewenste formaat wordt bepaald aan de hand van de HTTP accept header.

Om ervoor te zorgen dat er geen afhankelijkheden ontstaan tussen de action op de resource en de verschillende datatypes is ervoor gekozen om het encoderen van de response-data in het view-script af te handelen.

Zoals in de onderstaande figuur is te zien bestaat er voor ieder geregistreerd formaat een apart view-script. De action geeft het resultaat terug in een uniform formaat. In ieder view-script wordt een specifieke serializer aangeroepen om dit uniforme formaat om te vormen naar het gewenste formaat. Het view-script get.json.phtml zorgt er bijvoorbeeld voor dat het resultaat van de action omgevormd wordt naar JSON.



Schematische weergave van relatie tussen actions en verschillende datatypes

Zoals in het hoofdstuk Zend Framework wordt beschreven bevat de Zend ViewRender helper de verwijzing naar het viewScript dat hoort bij de action. Het is dus de taak van de Responsetype switcher om te bepalen welk datatype gewenst is en daarna de verwijzing naar het bijpassende view script in de ViewRenderer helper door te voeren.

Datatype registratie

De responsetype switcher moet in eerste instantie weten welke datatypes toegestaan zijn en hoe er met deze toegestane datatypes moet worden omgegaan. Het is dus van belang dat de programmeur of beheerder dit op een gemakkelijke wijze kan aangeven. Deze registratie moet plaatsvinden voordat de responsetype switcher wordt uitgevoerd.

Het bootstrap proces is het moment waarop de registratie van types moet plaatsvinden. Zoals in het hoofdstuk “Zend Framework” al naar voren kwam, is het bootstrap proces het moment van initialisatie. Tijdens dit proces wordt alles gereed gemaakt zodat de applicatie naar behoren kan worden uitgevoerd.

De toegestane datatypes worden geregistreerd met behulp van de registerType methode. Deze methode heeft drie parameters die mee kunnen worden gegeven. De eerste parameter is het datatype ook wel mediatype genoemd bijv. `application/json`. De tweede is een instantie van een handler-klasse die bij het mediatype hoort bijv. new Jsonhandler. De derde parameter geeft aan of het mediatype het standaard mediatype moet worden door middel van true of false. Het default datatype is het type waar in geval van onduidelijkheid op kan worden teruggevallen.

Om de registratie te vereenvoudigen is er voor gekozen om binnen de REST API de

mediatypes en de bijbehorende configuratie aan te geven in het application.ini configuratie bestand.

Dit configuratiebestand wordt uitgelezen door de REST resource. Dit is een object dat is gecreëerd om alle registratie van helpers en plugins op de juiste volgorde plaats te laten vinden. De REST resource voegt alle mediatypes uit het configuratiebestand toe aan de responsetype switcher alvorens deze wordt geregistreerd. Een korte beschrijving van deze REST resource is te vinden onder het kopje “Andere componenten”.

Daarnaast moet het mogelijk zijn om ook tijdens de uitvoer van de actie datatypes en handlers te registreren of te forceren. Op deze manier kan de programmeur afwijken van de door de client opgegeven datatypes.

Er is daarom voor gekozen om de responsetype switcher als een action helper binnen het Zend framework op te zetten. Een action helper heeft namelijk twee methodes een pre-en post – dispatch. De preDispatch wordt voor iedere action uitgevoerd en de postDispatch na iedere action. Daarnaast is de helper op ieder gewenst moment binnen de action beschikbaar. Het is hierdoor dus mogelijk om tijdens de action het gewenste datatype aan te passen.

Om dit mogelijk te maken is ervoor gekozen om voor iedere action, tijdens preDispatch, het content-negotiation proces te laten plaatsvinden. Het gewenste datatype is dus al bekend voordat de action is uitgevoerd. Het uiteindelijke vastzetten van dit datatype vindt plaats na de uitvoer van de action, tijdens postDispatch. De reden hiervoor is dat deze structuur het mogelijk maakt om tijdens de action het gewenste datatype aan te passen.

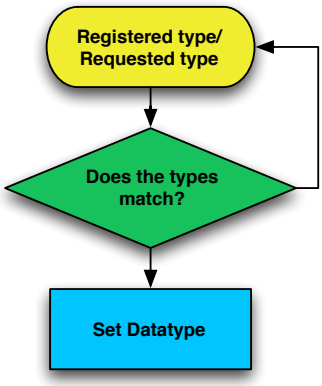
Onderstaand is een overzicht te zien van de acties die plaatsvinden in de responsetype switcher rondom een action.

Het responsetype switch proces

Het volledige klassendiagram van de Responsetype switcher is weergegeven in Appendix F.

Content negotiation

Tijdens het content negotiation proces, dat tijdens preDispatch wordt uitgevoerd, wordt aan de hand van de accept-header in het HTTP request het gewenste data formaat bepaald. De acceptheader wordt eerst door middel van de HTTP header parser, zie implementatie 9.2.2, verwerkt. Het resultaat van dit proces is een op prioriteit gesorteerde array met datatypes.



Deze gesorteerde array wordt van boven naar beneden afgelopen. Tijdens dit proces wordt er steeds een geregistreerd type vergeleken met het verzochte type uit de array. Wanneer een verzocht type overeenkomt met een geregistreerd type, dan wordt automatisch dit datatype vastgezet. Dit proces is versimpeld weergegeven in het bijstaande diagram.

In het geval van een lege accept-header of een wildcard[†] wordt het default geregistreerde

[†] Een wildcard geeft aan dat ieder beschikbaar datatype mag worden verstuurd. Een wildcard wordt aangegeven door */*

datatype gezet.

Zoals al eerder aangegeven bevat ieder geregistreerd datatype een specifiek gedefinieerde handler. Wanneer het type wordt vastgezet dan wordt dit uitgevoerd door de handler.

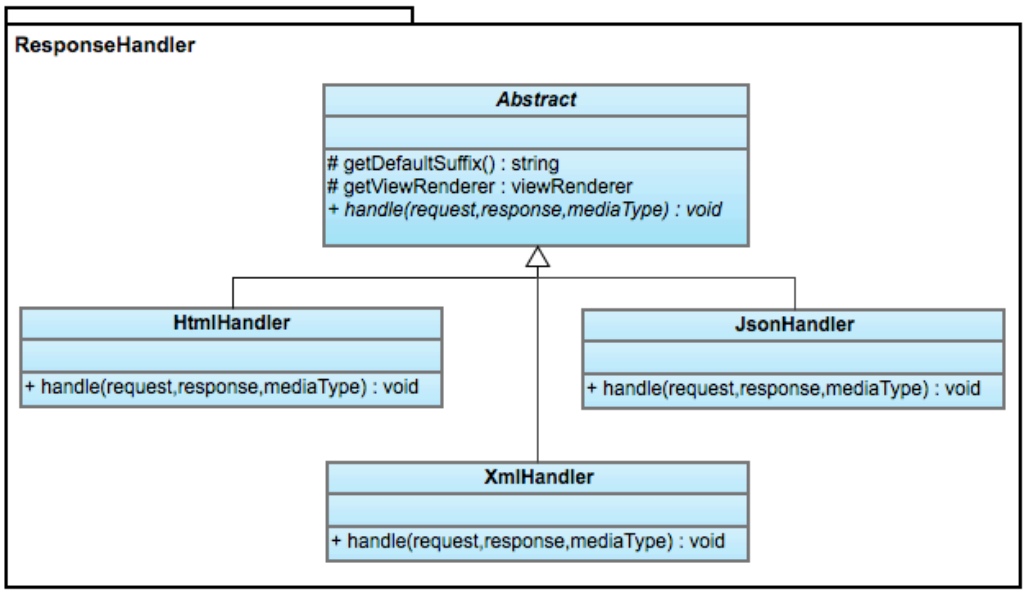
De Handler

De actie die moet worden uitgevoerd per datatype wordt gedefinieerd in een “handler”. Deze handler zorgt ervoor dat in het “viewRenderer object” wordt aangegeven naar welk datatype template moet worden verwezen, bijv. get.json.phtml. Daarnaast zorgt deze er ook voor dat de Content-type header in de response het juiste datatype bevat.

Voor ieder geregistreerd datatype moet er een handler bestaan. Voor de opzet hiervan is gebruik gemaakt van subtype polymorfisme[10]. Zoals al eerder in dit document is benoemd maakt dit principe gebruik van een parent-child inheritance structuur. Alle handlers stammen hierbij af van een abstracte parent klasse. Deze abstracte klasse fungeert als een uniforme interface voor alle handlers.

Het is dus bekend welke methoden de child klasse minstens bevat, omdat de abstracte parent forceert dat de children al zijn methoden implementeren. Dus als bekend is wie de parent van een child is dan is automatisch duidelijk over welke methoden deze beschikt.

In dit geval is het bekend dat een handler altijd afstamt van ResponseHandler\abstract en dat deze dus over de methode handle() beschikt. Zie het onderstaande voor verduidelijking van deze structuur.



ResponseHandler inheritance structuur

Tijdens het post-dispatch proces van de responsetype switcher wordt altijd de handle methode van de handler aangeroepen. Vanwege subtype polymorfisme is het altijd zeker dat deze binnen de handler bestaat. Het maakt dus niet uit welke handler van welk type wordt uitgevoerd. Deze handle methode zet de verwijzing naar het juiste view-script in het viewRenderer object en de content-type header in de HTTP response.

9.4 Request decoder

De request decoder is een krachtig maar vrij simpel onderdeel. Zoals de naam al zegt is het component verantwoordelijk voor het decoderen van de HTTP request body naar PHP variabelen.

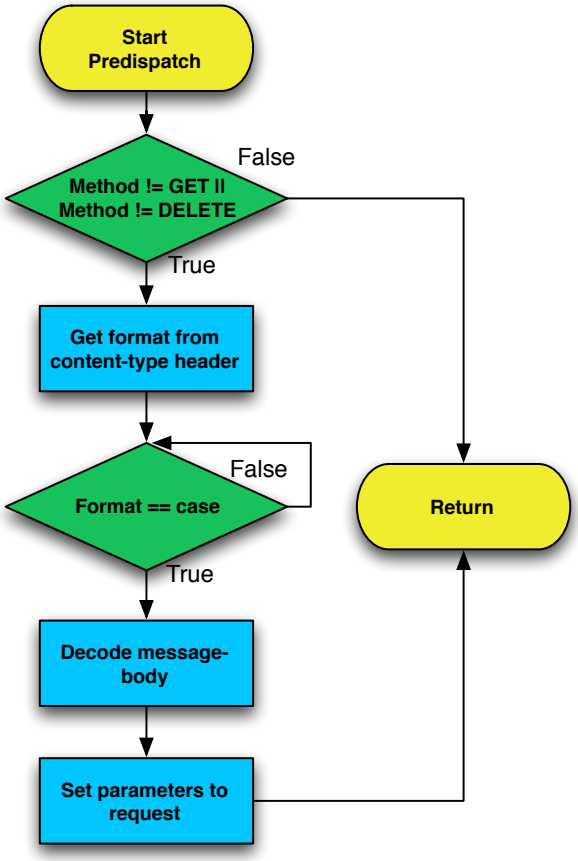
Deze variabelen worden toegevoegd aan het request object en zijn hierna binnen de gehele applicatie beschikbaar.

Er is voor gekozen om de request decoder als plugin tijdens de preDispatch fase, voordat de controller wordt uitgevoerd, uit te voeren. Dit is gedaan omdat meegestuurde data dan in de gehele controller beschikbaar is.

Het Proces

De request decoder wordt zoals aangegeven uitgevoerd in de preDispatch methode van de plugin. De decoder wordt dus aangeroepen voor de uitvoer van iedere controller.

Het proces is globaal weergegeven in de hiernaast zichtbare flow chart. In deze flow chart zijn de belangrijkste processen en controles weergegeven.



De eerste controle die plaatsvindt bekijkt of er mogelijk sprake is van een GET of DELETE request. Dit wordt volgens de requirements van Ibuildings niet ondersteunt. Het proces mag dus stoppen wanneer dit het geval is.

Hierna wordt het formaat van de content-type header bepaald. Deze header wordt ge-parsed door middel van de HTTP header parser. Het resultaat hiervan is een object dat alle headerinformatie bevat. Uit dit object kan dus het formaat worden bepaald.

Door middel van een Switch statement wordt bekeken of het formaat wordt ondersteund door de web service. Wanneer dit niet het geval is dan zal er een error worden weergegeven.

Wanneer het formaat wordt ondersteund dan wordt de message body gedecodeerd. De formaten die standaard worden ondersteund zijn JSON, XML en HTML. Voor het decoderen wordt gebruik gemaakt van standaard Zend framework en PHP functionaliteit zoals de Zend_Json::decoder.

Het resultaat hiervan is een array met alle variabelen die in de message body zijn meegestuurd. Na het decoderen worden deze variabelen toegevoegd aan het request object. Dit zorgt ervoor dat de data beschikbaar is door de gehele applicatie.

9.5 Precondition helper

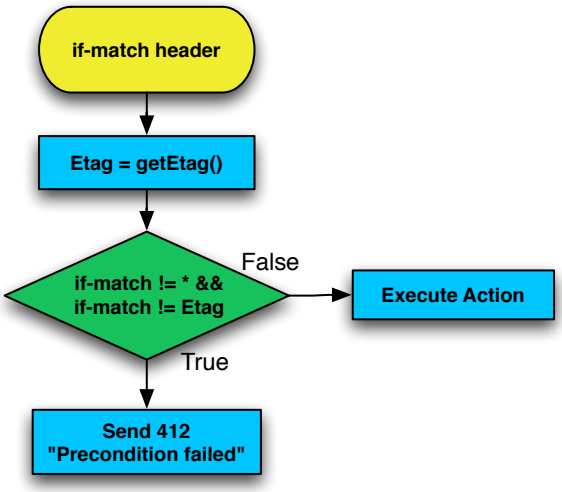
Zoals in hoofdstuk 6 is beschreven wordt de precondition helper gebruikt om te controleren of de laatste staat van de resource nog gelijk is aan zijn huidige staat. Dit kan gebruikt worden bij resource updates en caching. Er is voor gekozen om deze te implementeren in de vorm van een action helper omdat deze toegang moet hebben tot de aangeroepen controller en verschillende andere helpers.

De precondition helper heeft verschillende taken, die verdeeld over pre- en post dispatch, worden uitgevoerd. De belangrijkste zijn het controleren van de preconditions, het genereren van een Etag en het toevoegen van deze Etag in een response header.

De methode checkPreconditions

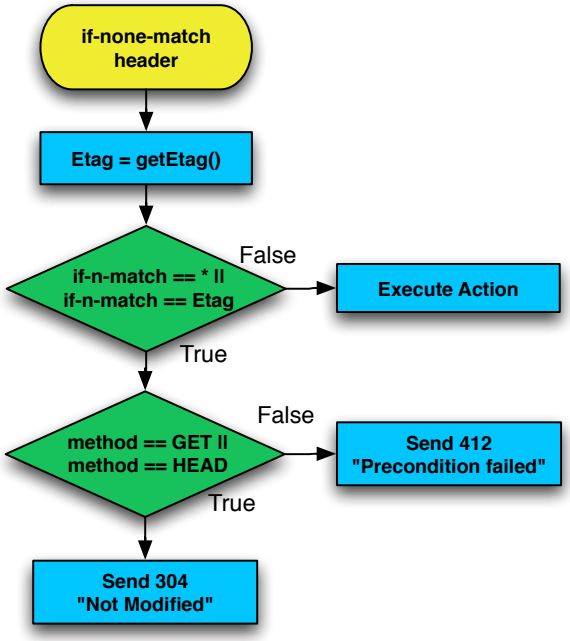
De methode checkPreconditions wordt aangeroepen vanuit de preDispatch methode wanneer er een if-none-match of if-match header aan het HTTP request is meegegeven. CheckPreconditions controleert of er wordt voldaan aan deze preconditions. Het eerste dat tijdens dit controleproces wordt uitgevoerd is het genereren van een Etag van de resource. Hiervoor wordt de methode getEtag aangeroepen. De Etag die deze methode teruggeeft is een unieke representatie van de resource in zijn huidige staat. De Etag wordt daarnaast ook opgeslagen in een cache attribuut. Dit scheelt performance omdat er in de postDispatch niet opnieuw een Etag hoeft worden te genereerd.

De Etag wordt hierna vergeleken met de gezette precondition header. Wanneer het om een if-match header gaat dan wordt er gecontroleerd of de Etag en de if-match header overeenkomen. Wanneer dit het geval is dan wordt de gewenste actie aangeroepen. Is dit niet het geval dan wordt er direct een HTTP response 412 “Precondition failed” teruggegeven. Rechts hiervan is de flow-chart van dit proces te zien



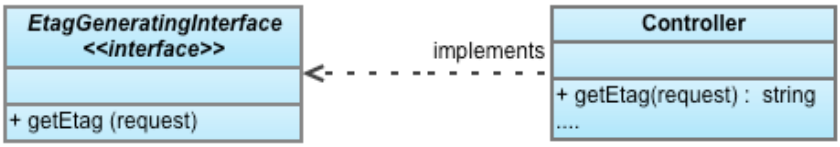
If-match precondition check

Wanneer het om een if-none-match header gaat dan wordt er juist gekeken of de Etag verschilt van de if-none-match header. Als de Etag anders is dan wordt de gewenste actie uitgevoerd. Zijn deze gelijk dan wordt in het geval van een GET of een HEAD request geantwoord met een HTTP response 304 “Not modified”. Wanneer het om een PUT of OPTIONS request gaat dan wordt er geantwoord met een HTTP response 412 “Precondition failed”. De flow-chart van dit proces is links hiervan te zien.



Er is bij HTTP preconditions wel een uitzondering op de regel. Wanneer er een asterisk “*” als if-* header wordt meegestuurd, dan komen de Etag en de if-* header altijd overeen. In het geval van een if-match betekent dit dat de precondition slaagt. In het geval van een if-none-match zal deze falen.

De methode getEtag
Het genereren van de Etag wordt gedaan door middel van de methode getEtag. Deze methode roept weer de getEtag methode uit de controller aan. De controller is namelijk de representatie van de resource en geeft een unieke string terug. In het onderstaande figuur is de constructie rondom de controller weergegeven. Om er zeker van te zijn dat iedere controller een getEtag methode bevat is er voor gekozen om dit af te dwingen door middel van een interface implementatie op iedere controller. Wanneer de controller niet volgens de interface geïmplementeerd is, dan wordt de precondition helper niet uitgevoerd.



Interface implementatie EtagGeneratingInterface

Deze string wordt gecombineerd met het gevraagde datatype uit de accept-header en de gevraagde taal uit de accept-language header. Deze combinatie wordt ge-hasht door middel van het md5 algoritme. Het resultaat is een 32 karakter tellende string die voor iedere representatie uniek is.

De methode sendEtagResponseHeader
Deze methode wordt in de postDispatch van de action uitgevoerd. De hoofdtaak van deze methode is ervoor zorgen dat er een Etag aan de HTTP response wordt toegevoegd. Er wordt hierbij gekeken of de request method “safe” is en of er een Etag is gecached tijdens preDispatch.

Een safe method voert geen verandering door op de server. Bij de methodes die gebruik maken van een Etag zijn GET, HEAD en OPTIONS “safe”. Wanneer het om een “safe” method gaat en er is een Etag gecached dan kan er zeker worden gesteld dat er tijdens de action geen veranderingen op de resource zijn doorgevoerd. In dit geval kan de cached Etag terug worden gegeven. Dit scheelt performance aangezien er niet opnieuw een Etag hoeft te worden gegenereerd.

In het geval van een PUT request of een request zonder if-* header wordt opnieuw een Etag opgehaald.

9.6 Overige componenten

Er is voor gekozen om alleen de componenten te beschrijven die van essentieel belang zijn voor de REST API. De componenten die ook ontwikkeld zijn, maar niet in dit document zullen worden uitgewerkt zijn:

Language negotiator
Goed vergelijkbaar met de responsetype switcher. Hierbij wordt aan de hand van de accept-language header bepaald in welke taal de response moet worden teruggegeven.

REST bootstrap resource
Binnen de REST API wordt gebruik gemaakt van vele plugins en helpers. Het is van belang dat deze componenten met de juiste prioriteit worden geregistreerd tijdens het bootstrap proces. Sommige componenten kunnen namelijk niet functioneren als deze volgorde niet klopt.

Om er zeker van te zijn dat de registratie van deze componenten goed verloopt, is ervoor gekozen om dit af te laten handelen door de REST resource. Deze REST resource is een klasse die tijdens het bootstrap proces wordt uitgevoerd en de registratie helpers en plugins bij de frontcontroller afhandelt.

Naast deze registratie taak leest de REST resource ook het application.ini configuratie bestand uit. In dit bestand staan namelijk de ondersteunde mediatypes en talen gedefinieerd. De REST resource leest dit uit en registreert de mediatypes en bijbehorende handlers bij de responsetype switcher voordat deze bij de frontcontroller wordt aangeboden. Hetzelfde geldt voor de ondersteunde talen bij de language negotiator.

Status code error controller
In het geval van fouten is het van belang om duidelijk aan te kunnen geven richting de client wat er precies misgaat. Binnen HTTP wordt dit gedaan door middel van statuscodes met een standaard bericht. Om gebruik te kunnen maken van deze functionaliteit is er een error controller ontwikkeld waaraan een HTTP code + een gewenst bericht kan worden meegegeven.

Wanneer er dus iets fout gaat op de server dan zal de client worden geïnformeerd wat er fout is gegaan en waar het probleem zit. Denk hierbij bijvoorbeeld aan een foutieve accept header.

Http204Listener
Het komt vaak voor dat na het uitvoeren van een DELETE request dat de programmeur vergeet de juiste HTTP status code in de response te zetten. Na een succesvol DELETE request wordt vaak geen response body geretourneerd. In dit geval moet de response een 204 statuscode bevatten, zie appendix D.

De Http204Listener controleert na ieder request of het om de methode DELETE gaat en of er een body wordt geretourneerd. Wanneer dit niet het geval is zorgt de Http204Listenere ervoor dat er alsnog een 204 statuscode aan de response wordt toegevoegd.

CRUD baseclass en demo applicatie
Om ervoor te zorgen dat de ontwikkelaar zo eenvoudig mogelijk CRUD kan implementeren op zijn resources, zijn er CRUD baseclasses gemaakt. Er bestaat een baseclass voor een collectie en voor een enkele entiteit. In deze baseclasses zit een standaard functionaliteit om bewerkingen uit te voeren op resources.

Om deze CRUD functionaliteit goed weer te geven in combinatie met de componenten is er voor gekozen om hier een demo applicatie voor te ontwikkelen. Deze applicatie geeft weer hoe een muziekcollectie kan worden beheerd met inzet van de REST API.

10 Evaluatie

In dit hoofdstuk kijk ik terug op de afstudeerstage die ik bij Ibuildings heb gevolgd. Een punt waar ik erg blij mee ben is dat de gestelde praktijkopdracht heb kunnen afronden. Er zat namelijk behoorlijk veel programmeerwerk en onderzoek aan verbonden en hierdoor was het niet zeker of ik de opdracht binnen de gestelde periode zou kunnen voltooien.

De uiteindelijke werkwijze die is toegepast voor de ontwikkeling van de producten, wijkt iets af van de vooraf bepaalde methodiek. Het idee was om voor het starten met de ontwikkeling een omvattend functioneel en technisch ontwerp op te stellen. In de praktijk bleek echter dat het van tevoren uitwerken van dergelijke documentatie in dit project niet goed werkte. Dit omdat de specifieke werking van de componenten aan het begin nog niet precies duidelijk was.

Er is daarom voor gekozen om bij de start van de ontwikkeling van ieder hoofdcomponent een beknopte MoSCoW requirements lijst op te stellen, zie appendix E, waarin alle belangrijke functionaliteit zit verwerkt. Daarnaast werd er steeds een concept klassendiagram of flow-chart opgesteld om een beeld te krijgen van de gewenste technische werking. Binnen de MoSCoW analyses zijn de fases Alpha 1, 2 en 3 terug te vinden als Must Have, Should Have en Could Have.

Vrijwel direct na deze fase werd gestart met het ontwikkelen van het component. Deze ontwikkeling vond plaats met gebruik van Unit Tests. Ieder ontwikkeld stuk functionaliteit werd met gebruik van deze tests geverifieerd. Tijdens de ontwikkeling kwam het regelmatig voor dat requirements werden aangepast of dat er tot nieuwe inzichten werd gekomen. Deze veranderingen werden dan direct in het ontwerp bijgewerkt en doorvertaald naar de applicatiecode en Unit Tests. Dit was een zeer resultaatgerichte manier van werken omdat aanpassingen direct werden doorgevoerd.

Na de ontwikkeling was het mijn taak om de componenten zowel technisch als functioneel te documenteren. Het was van belang om de technische werking nauwgezet te beschrijven zodat ontwikkelaars in de toekomst ook precies begrijpen waarom bepaalde keuzes zijn gemaakt.

De functionele beschrijving is gedaan in de vorm van tutorial-stijl documentatie. Hierin wordt aan de lezer precies uitgelegd hoe het component kan worden geïmplementeerd binnen een project. Deze documentatie is op dit moment nog niet in de praktijk gebruikt, maar wel als kwalitatief “goed” ontvangen bij de toekomstige gebruikers.

Als navolging hierop heb ik een demo-applicatie met de verschillende componenten opgezet. Hierin komt duidelijk naar voren hoe de componenten samen functioneren. Het idee achter deze demo is dat de ontwikkelaar deze gemakkelijk uit de "kast" kan pakken en direct kan omvormen tot de gewenste applicatie. Bij onduidelijkheden kan er in de demo applicatie code worden gekeken hoe bepaalde zaken daar zijn opgelost.

Uiteindelijk zijn alle requirements uit de MoSCoW analyses binnen de REST API skeleton verwerkt. In de praktijk kan de Alpha 3 fase gezien worden als demo en “bugfix” periode. Dit aangezien er bijna geen MoSCoW eisen in deze fase verwerkt zaten. Deze tijd is hierdoor gebruikt om de componenten toe te passen in een demo. Tijdens deze ontwikkeling kwam er een aantal bugs naar voren die direct konden worden aangepakt.

De componenten die zijn ontwikkeld tijdens mijn stageperiode zijn ook direct ingezet binnen bestaande projecten. Een voorbeeld hiervan is de ontwikkeling van verzekeringssite.nl. Hierbij wordt aan de server kant gebruik gemaakt van een REST API. Een groot deel van deze API bestaat uit onderdelen die tijdens mijn stageperiode zijn ontwikkeld.

De skeleton die is ontwikkeld wordt de standaard opzet voor REST API's binnen Ibuildings. Het product zal worden ingezet wanneer er een web service voor een opdrachtgever moet worden

ontwikkeld. In de loop van de tijd zal deze steeds verder worden uitgebreid met extra functionaliteiten en features.

Al met al kan ik stellen dat mijn afstudeerstage bij Ibuildings zeer leerzaam is geweest. Ik heb hier de kans gekregen om met interessante materie als REST en Zend binnen een professionele organisatie te werken. Ik kan met zekerheid stellen dat mijn kennis niveau de afgelopen maanden behoorlijk is gegroeid.

11 Conclusie

REST is een set van ontwerp richtlijnen voor het ontwikkelen van applicaties die gebruik maken van communicatie over een netwerk. Deze richtlijnen zijn opgesteld door Roy Fielding, hij deed dit in het jaar 2000 in de vorm van het proefschrift “Architectural Styles and the Design of Network-based Software Architectures”. De richtlijnen die hij hier in stelde zijn volledig gebaseerd op het Hypertext Transfer Protocol (HTTP).

Het idee achter REST is dat de werking van de applicatie gelijk is aan de werking van HTTP. Op deze manier wordt er gebruik gemaakt van het stabiele en uniforme karakter dat dit protocol bezit. HTTP is een van de meest gebruikte standaarden in de wereld en is zeer breed ondersteund. Vanwege dit vele gebruik heeft HTTP zich bewezen als zeer betrouwbaar. Wanneer applicaties hierop worden gebaseerd, dan stimuleert dit een hoge stabiliteit en brede ondersteuning. Er hoeft hierbij alleen maar gebruik worden gemaakt van de URI standaard en HTTP.

Vanwege de vele misinterpretaties van de richtlijnen die Roy Fielding heeft opgesteld introduceerde Leonard Richardson in 2007 het Richardson Maturity Model (RMM). Met dit model kan worden bepaald hoe “RESTful” een applicatie is. Deze “RESTfulness” wordt gemeten aan de hand van het gebruik van URI’s, HTTP en Hypermedia binnen de applicatie. Het model bevat vier levels waarbij level 0 niks meer met REST te maken heeft en level 3 de perfecte RESTful applicatie omschrijft.

Binnen een RESTful applicatie wordt gebruik gemaakt van verschillende resources. Ieder van deze resources wordt geïdentificeerd door middel van een URI. Een resource kan hierbij worden gezien als data met waarde, denk hierbij aan de gegevens van een persoon. Bewerkingen kunnen direct op deze resource worden uitgevoerd. Deze bewerkingen vinden plaats met gebruik van de hiervoor beschikbare HTTP methoden.

Het principe waarbij iedere resource apart benaderd kan worden komt overeen met de richtlijnen van een Resource Oriented Architecture ook wel ROA genoemd. Er kan worden gesteld dat ROA een RESTful architectuur is.

Een mogelijk alternatief voor een RESTful applicatie is de Service Oriented Architecture SOAP. SOAP is service georiënteerd en maakt in tegenstelling tot ROA gebruik van een enkel toegangspunt tot de applicatie. Resources worden hierbij niet direct benaderd.

De onderdelen die een REST API zeker moet bevatten zijn: een route component, request decoder, header parser, responsetype switcher, precondition helper en een serializer. Al deze componenten samen zorgen ervoor dat de REST API voor vrijwel iedere functionaliteit kan worden geïmplementeerd.

Ibuildings heeft bepaald dat de gewenste REST API moet voldoen aan de eisen die gelden voor een Level 2 REST applicatie volgens het RMM. Er is voor gekozen om Hypermedia (nog) niet te ondersteunen omdat er nog geen concrete richtlijnen bestaan voor het implementeren hiervan.

Het Zend Framework 1.1x bevat standaard een mogelijkheid om een web service op te zetten. Uit onderzoek van Ibuildings is gebleken dat deze mogelijkheid niet RESTful is. Daarentegen is het zeer goed mogelijk om een API met Zend te ontwikkelen. Het framework is namelijk gebaseerd op PHP en ondersteunt hierdoor de werking van HTTP en URL’s. Aangezien RMM level 2 gebaseerd is op deze twee onderdelen kunnen alle eisen van Ibuildings met Zend worden geïmplementeerd.

Binnen het Zend Framework wordt gewerkt met modules, controllers, actions en library componenten. Deze componenten worden door middel van het dispatch proces voor, tijdens of na de gestarte action uitgevoerd. Door gebruik te maken van deze eigenschappen van Zend is het

mogelijk om binnen de REST API precies te bepalen wanneer en hoe een request moet worden uitgevoerd.

Concluderend kan er worden gezegd dat de REST API moet voldoen aan de standaarden die gesteld zijn in HTTP. De onderdelen die hierbij strikt moeten worden nageleefd zijn het gebruik van HTTP methodes, HTTP headers en URI's. Dit is gebaseerd op het RMM level 2 en vertegenwoordigt tevens de requirements waaraan de API voor Ibuildings moet voldoen.

De ontwikkeling vond plaats door de resultaten van het onderzoek om te zetten naar een ontwerp. De functionaliteiten uit dit ontwerp werden vertaald naar Unit Tests. Met gebruik van deze Unit Tests werd vervolgens de functionaliteit ontwikkeld. Bij deze ontwikkeling is gebruikgemaakt van verschillende design patterns om stabiliteit te borgen. Dit ontwikkelproces vond iteratief plaats. Bij mogelijke veranderingen of verbeteringen werd dit direct in het ontwerp aangepast en doorgevoerd binnen de code.

Bij deze ontwikkeling is veelvuldig gebruik gemaakt van de specifieke fases die het dispatch proces van Zend Framework biedt. Het is namelijk zeer van belang dat de gestelde componenten op de juiste volgorde worden uitgevoerd.

Velen hebben REST op een incorrecte manier geïnterpreteerd. REST wordt namelijk veel als architectuur gezien maar is puur een set van richtlijnen. Uit mijn onderzoek is naar voren gekomen dat de voorgenoemde componenten essentieel zijn om de REST API volgens de REST richtlijnen te laten functioneren. Dit omdat deze de standaarden van HTTP implementeren.

12 Bronnen

1. Rodriguez, A. (2008) IBM RESTful Web services: The basics. Vekregen op 20 februari, 2012, via http://www.ibm.com/developerworks/webservices/library/ws-restful/

2. Fowler, M. (2010). RMM: Steps towards glory of REST. Verkregen op 24 februari, 2012, via http://martinfowler.com/articles/richardsonMaturityModel.html

3. Richardson, L. (2009) The Maturity Heuristic. Verkregen op 8 maart, 2012, via http://www.crummy.com/writing/speaking/2008-QCon/act3.html

4. Steves, B. (2008) REST Patterns: MIME Type. Verkregen op 8 maart, 2012, via http://restpatterns.org/Glossary/MIME_Type

5. Fielding, R. (1999) RFC 2616: Hypertext Transfer Protocol HTTP/1.1. Network Working Group. Verkregen op 24 februari, 2012 via http://www.ietf.org/rfc/rfc2616.txt

6. Extreme Programming. The Rules of Extreme Programming. Verkregen op 10 maart, 2012, via http://www.extremeprogramming.org/rules.html

7. Coley Consulting. MoSCoW Prioritiation. Verkregen op 21 maart, 2012, via http://www.coleyconsulting.co.uk/moscow.htm

8. Nadalin, A. Don't rape HTTP. Verkregen op 21 maart, 2012, via http://odino.org/don-t-rape-http-if-none-match-the-412-http-status-code/

9. Larman, G. (2002) Applying UML and Patterns. Prentice Hall

10. Gamma, E. (1994) Design Patterns: Elements of Reusable OO Software. Addison Wesley

11. Bakker, P. RESTful Webservices. Verkregen op 4 maart 2012, via http://www.infosupport.com/RESTful_Webservices_Paul_Bakker

12. Ogbuji, U. SOAP & WSDL IBM Developer Works. Verkregen op 4 maart, 2012, via http://www.ibm.com/developerworks/library/ws-soap/?dwzone=ws

13. Thomas Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Verkregen op 29 maart, 2012, via http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

14. Richardson, L. (2007). RESTful Webservices. O'Reilly Media

15. Webber, J. (2010) REST in practice: Hypermedia and System architecture. O'Reilly Media

16. University of Groningen. Verwijssysteem Vancouver. Verkregen op 30 april, 2012, via http://www.rug.nl/noordster/schriftelijkeVaardigheden/voorStudenten/bronLiteratuurGebruik/object1244409

17. Singh, T. REST vs. SOAP – The Right WebService. Verkregen op 17 mei, 2012, via http://geeknizer.com/rest-vs-soap-using-http-choosing-the-right-webservice-protocol/

18. Webber, J. Robinson,I. QCON REST in Practice tutorial. Verkregen op 20 maart, 2012, via http://www.slideshare.net/guilhermecaelum/rest-in-practice

19. Bos, van den M. (2010) Werken met het Zend Framework. Van Duuren Informatica

20. Zend Framework. Zend Framework Documentation. Verkregen op 24 mei, 2012, via http://framework.zend.com/manual/

13 Bijlagen

13.1 Appendix A: Richardson Maturity Model

Richardson Maturity Model

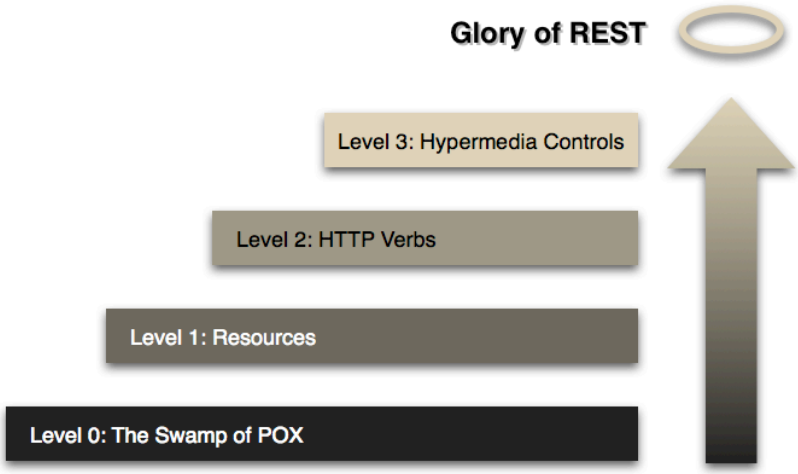
Het begrip RESTful applicatie is een erg breed en incompleet begrip. Er zijn namelijk veel applicaties die zichzelf RESTful noemen, maar op wat voor manier kan worden bepaald of deze echt RESTful zijn en wat de kwaliteit hiervan is? Dit zijn de vragen waarmee Leonard Richardson zich erg lang heeft beziggehouden. Na het bestuderen van honderden web services heeft hij in 2008 een model opgesteld waarmee kan worden bepaald op wat voor niveau REST binnen een applicatie is geïmplementeerd. De uitgangspunten hierbij zijn de verschillende onderdelen die de service bevat. Richardson heeft deze onderverdeeld in drie punten. Dit zijn URI, HTTP en Hypermedia.

- URI
  Iedere resource wordt geïdentificeerd door een unieke URI

- HTTP
  Er wordt gebruik gemaakt van de standaard HTTP methodes.

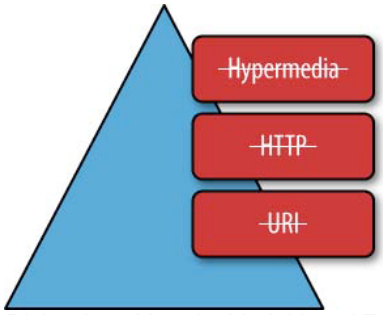
- Hypermedia[15]
  Dit houdt in dat de response URI's bevat om nieuwe acties uit te voeren. Denk aan een webwinkel waarbij een product een URI link bevat om te kopen. Hierbij is de response de pagina met het product en de hypermedia is de link om het product te kopen.

Het Richardson Maturity Model bestaat uit vier verschillend levels, nul t/m 3. Onderstaand zijn deze gevisualiseerd. Op de volgende pagina worden deze verder beschreven.



Levels binnen REST

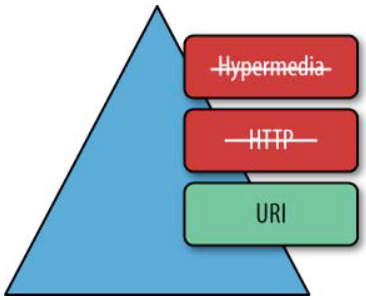
Richardson Maturity Model Level 0
Alles wat nodig is voor de actie wordt meegezonden in het request. Er wordt niet specifiek gebruik gemaakt van HTTP features. Hierdoor is REST op level 0 niet HTTP afhankelijk. Er kan ook makkelijk gebruik worden gemaakt van TCP of een ander protocol. Alles wordt met een POST request uitgevoerd. Services op level zero hebben een enkel toegangspunt en kunnen worden gezien als Service Oriented Architectures.



Richardson Maturity Model Level Zero

Richardson Maturity Model Level 1
Er worden URI's gebruikt om unieke resources te identificeren. In plaats van de request altijd naar een gelijk endpoint te sturen, wordt er voor ieder request een specifiek endpoint gebruikt. Deze service maakt gebruik van een belangrijk aspect van het web. Iedere resource moet uniek te identificeren zijn. GET of POST kunnen hierbij worden gebruikt. Er is op level 1 nog geen specifiek gebruik van POST of GET gespecificeerd. Op level 1 hoeft het HTTP protocol niet 100% nageleefd te worden.

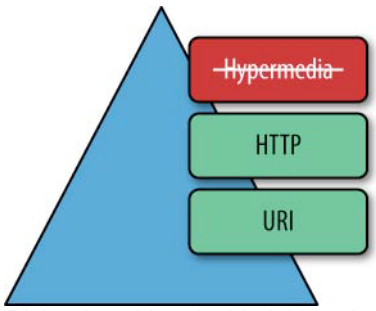
Volgens Richardson is het gebruik van unieke URI's het grote verschil met level zero. Er zijn namelijk vele zeer populaire services die RESTful zijn op level 1. Hun populariteit is volgens hem te danken aan deze unieke URI's. Dit zijn gebruikers namelijk gewend vanuit het normale web gebruik.



Richardson Maturity Model Level 1

Richardson Maturity Model Level 2
Het web bestaat uit miljoenen systemen die allemaal het HTTP protocol begrijpen. De benadering die op dit niveau wordt gebruikt is: Door je te houden aan de HTTP standaard die overal wordt gebruikt wordt de applicatie generieker. Het wordt hierdoor makkelijker om deze te schalen of breder in te zetten.

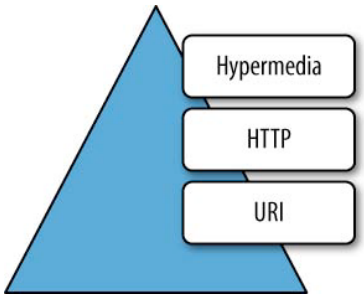
Het verschil met level 1 zit hem in de toepassing van HTTP methodes. Het is namelijk mogelijk om op iedere resource alle, ondersteunde, methodes uit te voeren. Er kan dus op bijvoorbeeld URI article/34 zowel get als put worden uitgevoerd. Iedere methode voert hierbij een andere actie, conform de richtlijnen, uit.



Richardson Maturity Model Level 2

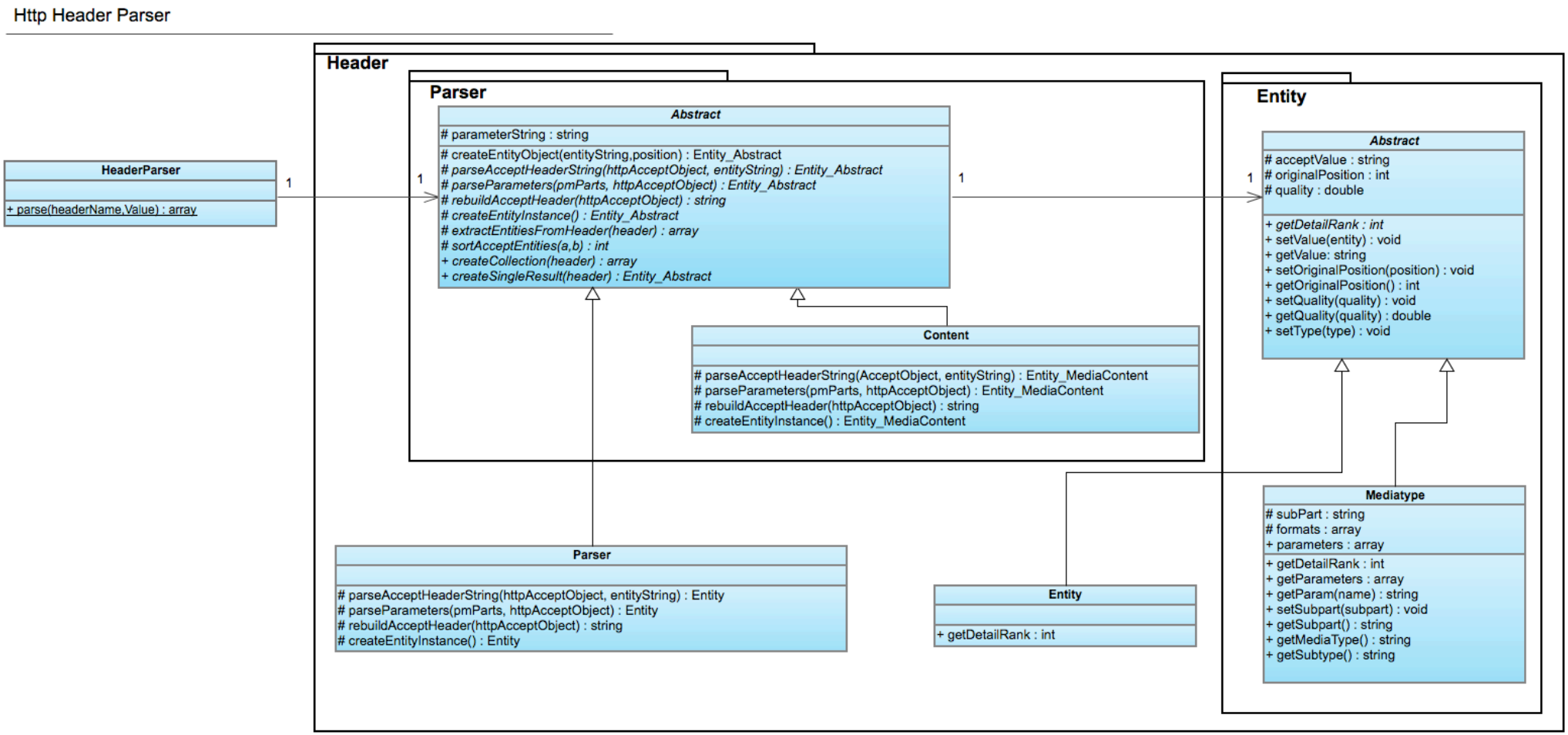
Richardson Maturity Model Level 3
Level 3 is goed te vergelijken met Level 2. Het verschil zit hem in de response die de service terug geeft. In deze response zit een nieuw element toegevoegd. Dit element bevat een URI die verwijst naar de volgende actie die op de resource kan worden uitgevoerd. Dit wordt hypermedia controls genoemd. Wanneer bijvoorbeeld een lijst met beschikbare afspraken wordt opgevraagd dan zal de response bij iedere vrije datum een link bevatten die verwijst naar het boeken van de afspraak.

Een voordeel van deze manier van werken is dat de server zijn URI's kan aanpassen zonder dat de client hier last van heeft. De te gebruiken URI wordt immers meegestuurd in de response.

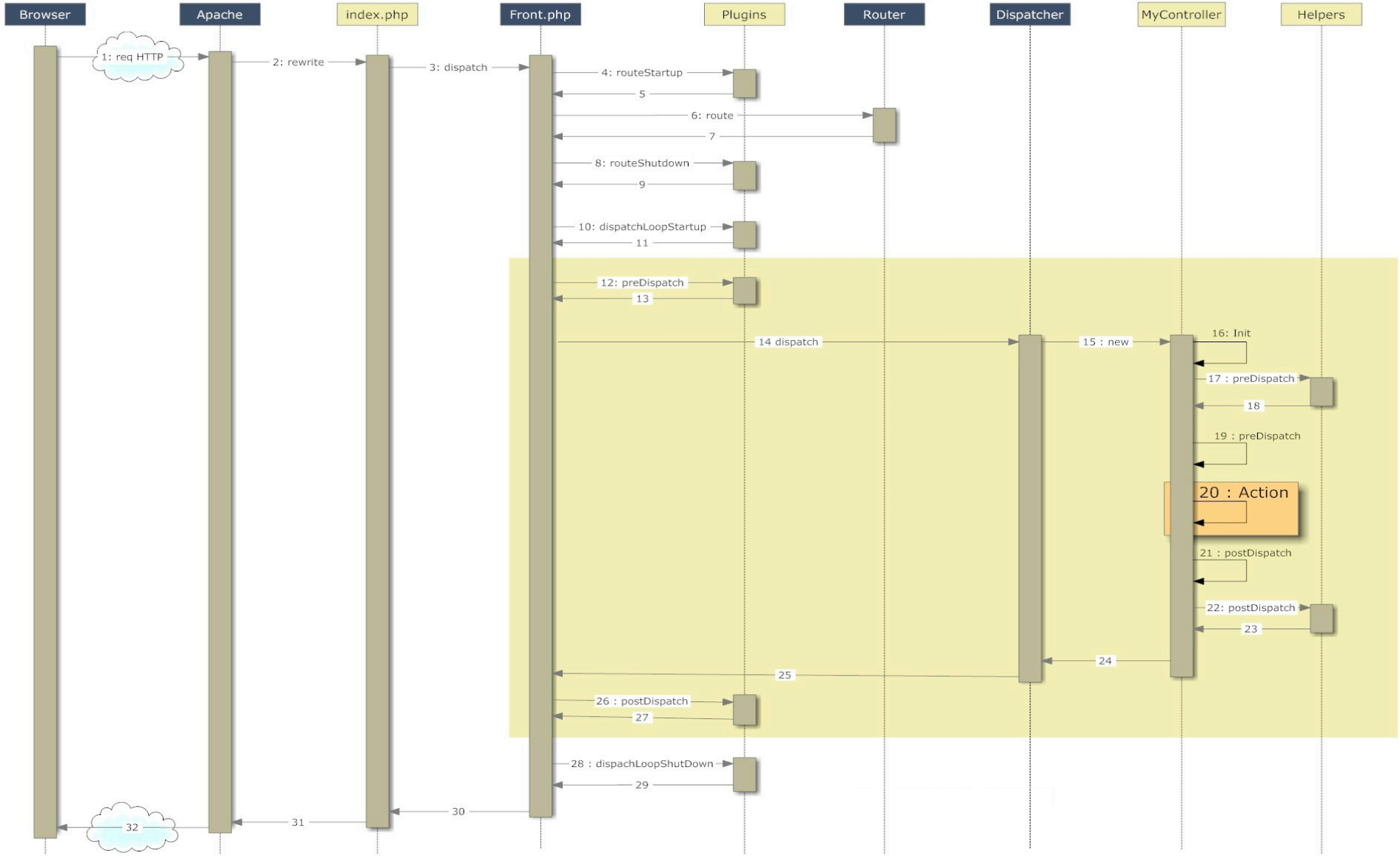


Richardson Maturity Model Level 3

13.2 Appendix B: Class Diagram HeaderParser



13.3 Appendix C: Zend Request Dispatch Diagram



13.4 Appendix D: HTTP status codes

Value	Description	Reference
100	Continue	[RFC2616]
101	Switching Protocols	[RFC2616]
102	Processing	[RFC2518]
103-199	Unassigned	
200	OK	[RFC2616]
201	Created	[RFC2616]
202	Accepted	[RFC2616]
203	Non-Authoritative Information	[RFC2616]
204	No Content	[RFC2616]
205	Reset Content	[RFC2616]
206	Partial Content	[RFC2616]
207	Multi-Status	[RFC4918]
208	Already Reported	[RFC5842]
209-225	Unassigned	
226	IM Used	[RFC3229]
227-299	Unassigned	
300	Multiple Choices	[RFC2616]
301	Moved Permanently	[RFC2616]
302	Found	[RFC2616]
303	See Other	[RFC2616]
304	Not Modified	[RFC2616]
305	Use Proxy	[RFC2616]
306	Reserved	[RFC2616]
307	Temporary Redirect	[RFC2616]
308	Permanent Redirect	[RFC-reschke-http-status-308-07]
309-399	Unassigned	
400	Bad Request	[RFC2616]
401	Unauthorized	[RFC2616]
402	Payment Required	[RFC2616]
403	Forbidden	[RFC2616]
404	Not Found	[RFC2616]
405	Method Not Allowed	[RFC2616]
406	Not Acceptable	[RFC2616]
407	Proxy Authentication Required	[RFC2616]
408	Request Timeout	[RFC2616]
409	Conflict	[RFC2616]
410	Gone	[RFC2616]
411	Length Required	[RFC2616]
412	Precondition Failed	[RFC2616]
413	Request Entity Too Large	[RFC2616]
414	Request-URI Too Long	[RFC2616]
415	Unsupported Media Type	[RFC2616]
416	Requested Range Not Satisfiable	[RFC2616]
417	Expectation Failed	[RFC2616]
422	Unprocessable Entity	[RFC4918]
423	Locked	[RFC4918]
424	Failed Dependency	[RFC4918]
425	Reserved for WebDAV advanced collections expired proposal	[RFC2817]
426	Upgrade Required	[RFC2817]
427	Unassigned	
428	Precondition Required	[RFC6585]

429	Too Many Requests	[RFC6585]
430	Unassigned	
431	Request Header Fields Too Large	[RFC6585]
432-499	Unassigned	
500	Internal Server Error	[RFC2616]
501	Not Implemented	[RFC2616]
502	Bad Gateway	[RFC2616]
503	Service Unavailable	[RFC2616]
504	Gateway Timeout	[RFC2616]
505	HTTP Version Not Supported	[RFC2616]
506	Variant Also Negotiates (Experimental)	[RFC2295]
507	Insufficient Storage	[RFC4918]
508	Loop Detected	[RFC5842]
509	Unassigned	
510	Not Extended	[RFC2774]
511	Network Authentication Required	[RFC6585]
512-599	Unassigned	

13.5 Appendix E: MoSCoW requirement analyse

Route Component

Functional requirements	**Must Have**	**Should Have**	**Could Have**	**Won't Have**
1) Er moet een route kunnen worden gedefinieerd waarbij module en controller van tevoren zijn bepaald.	X			
2) De action moet worden bepaald aan de hand van de HTTP method.	X			
3) Er moet bij een POST request door middel van X-HTTP-Method-Override een custom action kunnen worden gedefinieerd.	X			
4) Er moet bij een POST request door middel van ?_method= achter de URL een custom action kunnen worden gedefinieerd.				
5) Parameters moeten door middel van regular expressions in de route kunnen worden gedefinieerd	X			
5.1) Deze parameters moeten kunnen worden geïdentificeerd met een tag.		X		
5.2) Deze parameters moeten kunnen worden geïdentificeerd door middel van parameter mapping.	X			
5.3) De gevonden parameters moeten als request-parameters aan het request worden toegevoegd	X			
6) De parameters moeten in de route kunnen worden gedefinieerd met accolades			X	

HTTP header parser component

Functional requirements	**Must Have**	**Should Have**	**Could Have**	**Won't Have**
1) Er moeten extra parsers voor verschillende HTTP headers kunnen worden toegevoegd.	X			
2) Iedere entiteit binnen de header moet worden omgezet naar een object	X			
2.1) Ieder object moet de complete entiteit bevatten	X			
2.2) Ieder object moet mogelijke parameters bevatten	X			
2.3) Ieder object moet een mogelijk subtype bevatten	X			
3) Iedere parser moet aan de hand van het type header een collectie met objecten of een enkel object terug kunnen geven	X			
4) De accept header moet standaard worden ondersteund.	X			
5) De content-type header moet standaard worden ondersteund.	X			
6) De accept-language header moet standaard worden ondersteund		X		
7) De geretourneerde collectie moet gesorteerd zijn op prioriteit	X			
8) Parameters in de headers moeten worden ondersteund	X			

Responsetype component (content-negotiation)

Requirement	Must Have	Should Have	Could Have	Won't Have
1) De datatypes worden bepaald in de accept-header en moeten op prioriteit worden gezet door het headerparser component.	X			
2) Beschikbare types moeten kunnen worden geregistreerd tijdens het bootstrap proces van de applicatie.	X			
3) Beschikbare types moeten kunnen worden geregistreerd via het config ini bestand.		X		
4) Er moet een “default” datatype kunnen worden ingesteld	X			
4.1) Dit default datatype moet worden gebruikt in het geval van een wildcard		X		
4.2) Dit default datatype moet worden gebruikt wanneer de accept-header leeg is	X			
5) Het gekozen datatype moet tijdens de uitvoer kunnen worden aangepast.		X		
6) Datatypes moeten tijdens de uitvoer kunnen worden geregistreerd.		X		
7) Het “default” datatype moet kunnen worden veranderd tijdens de uitvoer.		X		
8) Het moet gemakkelijk zijn om nieuwe datatypes te ondersteunen	X			
9) Het component moet kunnen worden disabled en enabled			X	
10) In de HTTP response moet de content-type header zijn gezet met het juiste datatype	X			
11) In het geval van fouten moet er worden gereageerd met de juiste HTTP response code	X			
12) Er moet voor worden gezorgd dat er per type het juiste view-script wordt geladen.	X			

Request decoder component

Requirement	**Must Have**	**Should Have**	**Could Have**	**Won't Have**
1) Moet bepalen aan de hand van de content-type header om wat voor data het gaat	X			
2) Dit proces moet ver voor de gevraagde action worden uitgevoerd	X			
3) Er moeten gemakkelijk nieuwe formaten kunnen worden toegevoegd	X			
3.1) Het formaat HTML moet standaard worden ondersteund		X		
3.2 Het formaat JSON moet standaard worden ondersteund	X			
3.3 Het formaat XML moet standaard worden ondersteund		X		
4) De inhoud moet als parameters in het request worden toegevoegd	X			
5) De Request decoder moet niet worden uitgevoerd in het geval van een GET of DELETE request	X			

Preconditions component

Requirement	**Must Have**	**Should Have**	**Could Have**	**Won't Have**
1) Het component moet de “if-match” header ondersteunen	X			
2) Het component moet de “if-none-match” header ondersteunen	X			
3) De Etag moet worden gegenereerd aan de hand van de resource	X			
4) De controle op de if-* headers moet plaatsvinden voordat de action wordt uitgevoerd	X			
5) Wanneer er geen header is meegestuurd dan moet er na het uitvoeren van de action een Etag worden gegenereerd	X			
5.1) Deze Etag moet in de HTTP response in de e-tag header worden terug gestuurd.	X			
5.2) De Etag moet worden berekend aan de hand van de request gegevens + de resource		X		
5.3) De Etag is een MD5 encoded string	X			
6) Er moet onderscheidt worden gemaakt tussen safe en unsafe HTTP methods		X		

HTTP Exception controller

Requirement	**Must Have**	**Should Have**	**Could Have**	**Won't Have**
1) Voor iedere status-code moet een apparte exception worden gecreëerd	X			
1.1) Er moet een standaard bericht zijn gedefinieerd volgens de RFC2616 richtlijnen.	X			
1.2) Er moet een zelf gedefinieerd bericht kunnen worden meegegeven aan de exception. Deze overschrijft het standaard bericht.		X		
1.3) Iedere exception moet een HTTP status-code bevatten	X			
2) Er moet een basis exception zijn waaraan een status-code en message kan worden meegegeven		X		
2.1) De HTTP code wordt standaard op 500 gezet		X		

13.6 Appendix F: Class Diagram Responsetype Switcher