RESTful client for Open Data on the Human Semantic Web

Productopdracht om snellere en schaalbare ontwikkeling mogelijk te maken door een transitie naar zwak gekoppelde architectuur

Thom van Kalkeren

Hogeschool Utrecht

Scriptie

Argu B.V.
https://argu.co
Utrecht, 19 december 2016

Opleiding HBO-ICT
Examinator: Jan Zuurbier
Studentnummer 1570479
Versie 1.3

© 2016 Thom van Kalkeren
This work is licensed under a Creative Commons Attribution
ShareAlike 4.0 International License.

Voorwoord

De maatschappij is continu in beweging, beïnvloed door een scala van redenen: nieuwe mensen, nieuwe technologie, vergane gebruiken en gedeelde belevenissen. Zoals het verplaatsen van een steen in een rivier laat elke gebeurtenis een indruk achter op de maatschappij. Het verandert haar permanent; ook na het terugleggen zal de stroom nooit meer precies hetzelfde zijn. Hoewel dit vanuit het individu bekeken vaak willekeurig — of zelfs chaotisch — lijkt te zijn, is in de stroom van deze miljarden maatschappelijke gebeurtenissen een rode draad te vinden: ze zijn verbonden door keuzes; *besluiten*.

Het juiste besluit vinden is echter niet altijd even makkelijk. Het aantal mogelijkheden en invloeden van zelfs de meest triviale dagelijkse dingen — zoals het kiezen van de juiste tandpasta[1] — kan grote gevolgen hebben voor de gebeurtenissen die daarna plaatsvinden. Op het moment dat een besluit duizenden dan wel miljoenen mensen treft, wordt het overwegen van alle opties met hun voor- en nadelen zeer belangrijk. Een foute keuze kan veel mensen buitenspel zetten[2].

Dankzij de recente opkomst van het alom aanwezige internet is het nog nooit zo makkelijk geweest om immense groepen mensen bij elkaar te brengen. Dit stelt diverse organisaties, zoals overheden, in staat om bij het maken van keuzes de voor- en nadelen overzichtelijk in kaart brengen in samenspraak met de burger. Dit is wat Argu faciliteert voor diverse organisaties in Nederland.

Zoals de maatschappij verandert, verandert ook Argu. Ontwikkelingen in de afgelopen jaren hebben het noodzakelijk gemaakt om de werking van de technologie aan te passen om beter aan te kunnen sluiten bij de wensen van klanten en de interne visie. Dit voorstel beschrijft daarom de ontwikkeling van de behoeftes en een passende oplossing om de architectuur aan te passen, zodat Argu mee kan blijven groeien met de veranderende maatschappij.

De invulling en uitvoering van deze opdracht zal een van de vele gebeurtenissen zijn die zijn invloed uitoefent op de maatschappij en een permanente verandering achter zal laten.

---

[1] Cambridge, E. (2016, 14 June 2016). Man dramatically rushed to hospital after brushing his teeth with mint toothpaste. [Article]. Retrieved 17 June 2016, from https://www.thesun.co.uk/news/1279215/man-rushed-to-hospital-after-brushing-teeth-with-toothpaste/

[2] Woudegraaf. (2016, 25 Aug 2015). Ombudsman zeer kritisch over PGB: 'De overheid leert niet van fouten'. [Weblog]. Retrieved 17 June 2016, from http://www.woudegraaf.nl/nl/ombudsman-zeer-kritisch-over-pgb-de-overheid-leert-niet-van-fouten/

Schrijfstijl, afkortingen, en begrippen

In de tekst komen namen uit programmatuur voor, deze zijn omringd met het accent grave (`).

Hier volgt een lijst van veelgebruikte afkortingen en begrippen. Dit zijn afkortingen of Engels jargon waarbij ervoor gekozen is deze in originele vorm in de tekst te gebruiken.

Afkorting / Begrip	**Volledig / Nederlands**	**Uiteenzetting**
AOD	Argu Open Data	Het project binnen Argu om open data toegankelijker te maken voor het grote publiek.
BE	Back-end	Het gedeelte van een service die de business logica implementeert, waaronder data opslag, autorisatie, starten van afhankelijke processen, etc.
CRUD	Create, Read, Update, Delete	De methodes die binnen op een object op het web uitgevoerd kunnen worden; Aanmaken[c], Lezen[r], Bewerken[u], Verwijderen[d].
Deploy	Uitrol	Het process van het inwerkingstellen van een (nieuwe versie van een) applicatie(onderdeel). Vaak betekent dit het starten van de nieuwe applicatie en de verzoeken van gebruikers naar de nieuwe versie sturen.
FE	Front-end	Het gedeelte van een service waar de gebruiker mee interacteert, ontwikkeling is gestoeld op gebruikservaring.
IRI	International Resource Identifier	Het internationale broertje van de URI (Universal Resource Identifier), het specificeert hoe een resource geïdentificeerd kan worden. Denk aan hyperlinks en het ISBN.
Key	Sleutel	De noemer waaronder een waarde opvraagbaar is.
Monolith	Monoliet	Een systeem waarbij de delen sterk verweven en praktisch onlosmaakbaar zijn.
RMM-n		Duidt een laag in het Richardson Maturity Model aan, met n als nummer van de laag
Stack	-	De ‘stapel’ van technologieën welke een (web- of mobiele) applicatie vormen. Dit bestaat meestal uit een database, webserver, operating system, programmeertaal, applicatieframework, en deploymentprocess. Een bekend voorbeeld is de LAMP stack (linux, apache, mysql, php).

State	Staat	De opgeslagen informatie waar de applicatie toegang tot heeft. Het resultaat van invoer op de uitvoer in een applicatie is afhankelijk van de bijbehorende algoritmes en de huidige staat. Een voorbeeld van staat is welke gebruiker ingelogd is.
Uptime	-	Percentage van de tijd waarin de service bereikbaar moet zijn voor gebruikers.
Het web	-	De subset van het internet bedoeld voor de (menselijke) consumptie van informatie (HTTP/HTML).

Managementsamenvatting

Door uitbreiding van het team voldeed de integratie van back-end met de front-end van de Argu webservice niet meer aan de eisen voor snelle ontwikkeling. Een tweede product was reeds in ontwikkeling waarbij deze componenten gescheiden waren. Bij dit product was een data-warehousing systeem opgezet om data uit meerdere bronnen te consolideren zodat deze in één webapplicatie gebruikt zouden kunnen worden. Echter werd al snel duidelijk dat het scheiden van de webservice om deze later samenvoegen met het tweede product problemen zou opleveren. De front-end zou veel code en logica van de back-end moeten dupliceren. Bovendien zou consolidatie geen rendabele aanpak zijn door het grote aantal benodigde bronnen. Om dit probleem op te lossen is onderzoek gedaan naar een toekomstbestendige backwards- compatible architectuur voor het overbrengen en weergeven van diverse data uit meerdere bronnen.

Bij het oplossen van het probleem is er gekeken naar welke soorten data door het systeem verwerkt moesten gaan worden. Vervolgens zijn de mogelijkheden tot abstractie binnen deze data afgegaan om zo een geconsolideerde interface voor de overdracht van data te kunnen definiëren. Uiteindelijk is gekeken naar een interface voor de betekenis van data zodat de data van verschillende bronnen met één front-end geïnterpreteerd kan worden.

De meerderheid van door de overheid beschikbaar gestelde open data is in JSON format. Het toepassen van de principes van REST op de front-end applicatie koppelt de applicatie los van de datastructuur, waardoor deze vrij is om over meerdere bronnen te navigeren. Tevens kunnen modules worden geïmplementeerd om de applicatie breder inzetbaar te maken. Tot slot bieden de principes van het Semantic Web de sleutel om de front-end data te laten interpreteren. Op deze manier kunnen semantisch aangestuurde applicaties gebouwd worden die niet gebonden zijn aan een dataleverancier.

Om dit concept te bewijzen is een proof-of-concept framework ontwikkeld die ontwikkelaars in staat stelt om semantisch gedreven applicaties te bouwen. Dit biedt de mogelijkheid om op een eenvoudige manier semantische client applicaties te schrijven voor menselijk gebruik, wat de ontwikkeling van het semantische web kan bespoedigen.

Inhoudsopgave

1. Inleiding

Argu is lange tijd ontwikkeld door een klein team die breed inzetbaar was. Door uitbreiding binnen Argu kwamen specialistische ontwikkelrollen binnen het team; losse personen voor de ontwikkeling van front- en back-end. Dit leidde tot frictie doordat de componenten technisch niet van elkaar gescheiden waren. Dit zette Argu aan tot het laten scheiden van de front- en back-end. Echter was het onduidelijk hoe de scheiding bereikt moest worden met het oog een nieuw intern project 'Argu Open Data', waar deze componenten ook benodigd waren. Het gescheiden onderhouden van de twee projecten zou onnodig veel druk op het team leggen.

Daarom is een onderzoek verricht voor het vinden van een methode om de front- en back-end te scheiden om de ontwikkeling te versnellen. Hierbij moest AOD in het achterhoofd gehouden worden.

Tijdens het onderzoek kwam het Semantic Web/Resource Description Framework naar voren, dit is een set van ideeën en technologieën om een web van *machine readable* data te maken. In RDF is datanormalisatie tot het uiterste genomen, alle data is omvat in individuele verklaringen over resources[3]. Diverse websites geven deze informatie al weer in simpele tabellen, maar het omzetten van deze data naar informatie voor menselijke consumptie ontbreekt nog. Het omzetten van data naar deze vorm is eenvoudig, waardoor het semantische web de mogelijkheid leek te bieden om de twee front-ends te unificeren onder deze standaard.

Naar aanleiding hiervan is het onderzoek uitgebreid naar het vinden van een methode voor het scheiden van de front- en back-end waarbij de front-end op basis van Semantic Web principes voor zowel Argu als AOD data moest kunnen fungeren.

In dit document zal eerst de context en de kwestie van de opdracht uiteen worden gezet. Uit de kwestie zijn daarna de doelstellingen voor het onderzoek geformuleerd. Vervolgens is het bijbehorende theoretisch kader beschreven om de onderzoeksvragen mee te formuleren.
Daarna wordt per onderzoeksvraag de uitvoering en de conclusie uiteengezet waarbij met de hoofdvraag geëindigd wordt. Daarna is een beknopt overzicht van het bijbehorende prototype beschreven. Tot slot zijn verdere aanbevelingen voor Argu en de verdere mogelijkheden voor het Human Semantic Web gegeven.

[3] Bijv. "http://argu.co/m/8 http://schema.org/name 'Titel' ." drukt uit dat de 'naam' van 'motie 8' de waarde 'Titel' heeft

2. Organisatie

De opdracht wordt uitgevoerd bij Argu B.V, een internetbedrijf gevestigd in Utrecht[4]. Met zeven medewerkers[5] is Argu gespecialiseerd in (burger)inspraak voor (semi)overheid. Argu heeft een informele cultuur. Discussie op de werkvloer wordt aangemoedigd en beslaat diverse onderwerpen, meestal gerelateerd aan werk of maatschappelijke kwesties, zoals hoe de keuzes binnen het bedrijf de maatschappij beïnvloeden.

Het hoofdproduct 'Argu' is een discussie- en inspraakplatform die afgenomen wordt door diverse gemeentes en woningbouwcorporaties in Nederland. Door het plaatsen van vraagstukken en ideeën kunnen medewerkers en leden problemen identificeren en oplossen.

Als developer binnen het team zullen de verantwoordelijkheden het ontwerp, de implementatie en het opzetten van de testomgeving omvatten. Bij de implementatie van de software betreft het de technische kant. Andere gebieden zoals opmaak en user experience worden door specialisten in het bedrijf gedaan.

3. Kwestie

De Argu dienst is een webservice waarbij klanten en hun leden via een website kunnen discussiëren over diverse onderwerpen. Hierbij kunnen de gebruikers via een web-interface (de 'front-end') data bekijken en invoeren in het systeem. De data wordt opgevraagd dan wel verwerkt door de achterliggende service (de 'back-end'). De dienst is geschreven met het applicatie-framework 'Ruby on Rails'. Hierbij zijn de back- en front-end van de applicatie dusdanig sterk geïntegreerd dat deze in principe één applicatie vormen.

Binnen een startup kunnen requirements snel veranderen en snel inspelen op deze veranderende requirements is essentieel (Nobel, 2013). In het huidige proces bestaan meerdere beperkingen die de ontwikkeling vertragen. Wanneer functionaliteit moet worden aangepast moeten wijzigingen in zowel de FE als de BE worden gedaan omdat deze in hetzelfde package zitten. Deze dependency betekent ook dat het als geheel getest moet worden omdat de impact van fouten niet geïsoleerd kan worden.

Daarnaast zijn de FE en BE beide applicaties op zich en moeten ontwikkelaars getraind zijn in de beheersing van twee verschillende technologie stacks om effectief te kunnen ontwikkelen. Dat heeft tot gevolg dat ontwikkelaars bovenop de verschillende technologieën ook ingewerkt moeten worden op de twee specifieke applicaties met de bijbehorende implementatie details. De ontwikkelaars zijn genoodzaakt het gehele mentale model te begrijpen, iets wat onnodig

[4] Koningslaan 62, 3583 GP Utrecht
[5] Zie bijlage 1 voor persoonsgegevens

veel van hen vraagt. Dit heeft Argu ook meegemaakt in het afgelopen jaar waarbij veel nieuw personeel aangenomen is. Het kostte veel extra tijd om ontwikkelaars in te werken voor meerdere frameworks en applicaties waar ze niet hoofdzakelijk aan werken.

Wanneer de behoefte ontstaat om nieuwe libraries of technologieën te gebruiken moeten deze tevens vaak compatibel zijn met de twee bestaande stacks, wat opties voor de versoepeling van ontwikkeling beperkt.

Daarnaast is het project ‘Argu Open Data’ in ontwikkeling. Het project bestaat o.a. uit een aantrekkelijke interface om burgers inzicht te geven in de overheid, een stemwijzer gebaseerd op echte stemmen, een zoekmachine, en diverse statistieken uit datamining. Het project is nu nog los ontwikkeld, maar het project moet uiteindelijk geïntegreerd of samengevoegd worden.

De data voor de eerste versie van AOD komt uit twee verschillende API’s en beslaat 8 datasets uit drie systemen. Deze bronnen beslaan voor een groot deel gelijksoortige data, parlementsdata, en zijn daarom vrij uniform. Desalniettemin moet de data uit elke bron getransformeerd worden alvorens deze in de AOD applicatie gevoed kan worden. De aanwezigheid van een overhead is voor twee API’s nog goed te doen, maar maakt het systeem inherent onschaalbaar. Wanneer de maatschappelijke veranderingen en nieuwe achterliggende informatiesystemen worden meegenomen in het toekomstbeeld wordt dit probleem nog erger.

Argu is een maatschappelijk betrokken organisatie. Het is dus van belang om gedurende het project ethische afwegingen in het achterhoofd te houden. Omdat Argu een belangrijke rol wil spelen in politieke processen, is transparantie van groot belang. Het moet uitgesloten zijn dat achter de schermen met data gesjoemeld wordt. Ook wil Argu bijdragen aan de verdere ontwikkeling van het web. Daarom is het belangrijk kennis te delen, via open-source projecten of andere kanalen.

4. Doelstellingen

Het ontwikkelen van Argu gaat gepaard met sterke vertraging omdat de BE en FE van Argu dusdanig sterk verbonden zijn dat deze niet los ontwikkeld kunnen worden. Daarnaast moet voor het wijzigen van functionaliteit die alleen de BE of de FE beslaat het geheel weer getest worden. Hieruit volgt de eerste doelstelling:

> *"De werking van de FE en BE isoleren waardoor deze los ontwikkel- en testbaar worden"*

Doelstelling 1

Bij het bouwen van Argu Open Data wordt de meeste tijd geïnvesteerd in 'data warehousing', het verzamelen en transformeren van de data uit externe bronnen zodat deze in de front-end gevoed kunnen worden. Data komt echter van steeds meer verschillende organisaties ("Actieplan open Overheid 2016-2017," 2016). Deze transformaties zijn tijdsintensief, leveren relatief weinig op en brengen het risico met zich mee dat data niet juist wordt opgeslagen. Dit laatste is in strijd met de ethische afweging van zoveel mogelijk transparantie. Om deze redenen kan de tweede doelstelling gedefinieerd worden:

> *"Transformatie van databronnen naar in de FE verwerkbare data minimaliseren"*

Doelstelling 2

Bij de transformatie komen zeer veel soorten en varianten van data naar de oppervlakte. Varianten zoals een 'volledige naam' en een combinatie van 'voornaam' en 'achternaam' zijn eenvoudig te transformeren, maar er komen veelal compleet nieuwe datatypen voor zoals 'hamerstukken' die niet lijken op de 'Motie' die in het platform centraal staat. Omdat het product door journalisten en wetenschappers gebruikt moet worden is het echter wel van belang dat deze data vindbaar is. Hieruit is een derde doelstelling te definiëren:

> *"De FE moet breed inzetbaar zijn voor (voorlopig) onbekende datatypen"*

Doelstelling 3

Het herschrijven van werkende software is niet rendabel (Spolsky, 2000). De Argu back-end is geschreven met Rails. Hierdoor kan een vierde eis gedefinieerd worden:

> *"De interface moet met het Rails framework uitgeserveerd kunnen worden."*

Doelstelling 4

5. Methode

Alvorens een ontwerp te maken om de beoogde doelstellingen mee te bereiken, wordt eerst een onderzoek gedaan naar bijkomende problemen die moeten worden opgelost.

Het onderzoek wordt uitgevoerd door een hoofdvraag te formuleren die aansluit aan de doelstellingen. Deze wordt vervolgens opgesplitst in losse deelvragen die individueel beantwoord zullen worden op basis van diverse onderzoeksmethoden. Tot slot zal het antwoord op de hoofdvraag geformuleerd worden.

Uit het onderzoek zal het ontwerp voor het uiteindelijke architectuur voortkomen die zal dienen als input voor de implementatie.

Vervolgens zal het ontwerp worden geïmplementeerd in een werkend prototype die de basis van de beoogde functionaliteit omvat. Wanneer het prototype stabiel genoeg is wordt deze naar productie gebracht, waarna iteratieve verbetering toegepast zal worden totdat de stabiliteit en functionaliteit hoog genoeg is om de huidige functionaliteit te vervangen.

De uiteindelijk opgeleverde software zal met de voor het platform geldende tools gedocumenteerd en getest worden. Ook zal een overzicht van de (functionele) werking van de software in de scriptie opgenomen worden.

6. Theoretisch kader

De drie meest gebruikte modellen voor een interface via netwerkarchitectuur zijn RPC, SOAP en REST.

RPC is een sterke koppeling waarbij een subroutine op een andere computer wordt uitgevoerd en het resultaat gereturned wordt, de programmeur heeft hierbij veelal niet door dat deze procedure via een netwerk verloopt.

Bij SOAP worden alle methoden in een service op een vaste manier beschreven waardoor deze benaderd kunnen worden door andere services. Routing gebeurt vaak via een ESB.

REpresentational State Transfer (REST) is een architecturaal ontwerp voor het bouwen van schaalbare netwerkgebaseerde applicaties (Fielding, 2001).
Omdat zowel Rails als het web gebaseerd zijn op REST technologie, is het onderzoek op REST ingekaderd.

6.1. REST

Fielding (2001) deed een onderzoek naar de werking van het web en heeft een aantal constraints gedefinieerd die de gewenste eigenschappen van het web waarborgen, de set van deze constraints heet REST. Omdat er nogal wat misverstanden zijn rondom de exacte definitie van REST (Fielding, 2008) zal elke constraint volgens de beschrijving van Fielding hieronder uiteen worden gezet.

Client-server. De architectuur begint met de opbouw van client en server. De (gebruikers)-interface is gescheiden van zaken omtrent dataverwerking. Dit zorgt ervoor dat de interface (de browser) los ontwikkeld kan worden van de databron (webservice) en maakt beide onderdelen simpeler.
Stateless. De volgende constraint legt op dat de communicatie tussen de client en server stateless moet zijn. Dit houdt in dat de server een binnenkomend request moet kunnen verwerken zonder kennis van vorige requests. Dit vergroot de schaalbaarheid van het systeem aanzienlijk omdat informatie tussen requests niet bijgehouden hoeft te worden, en in modern perspectief, dat de server applicatie probleemloos horizontaal geschaald kan worden (Vaquero, Rodero-Merino, & Buyya, 2011).
Cache. De cache constraint houdt in dat een response impliciet of expliciet gelabeld kan worden als cachebaar. Dit geeft de client en tussengelegen processors de mogelijkheid om voor soortgelijke requests de opgeslagen response terug te geven in plaats van het request opnieuw over het netwerk te versturen. Dit is een aanzienlijke versnelling, maar kan bij slechte beheersbaarheid zorgen voor verouderde data. Omdat de API stateless is, kan op basis van het

request in isolatie bepaald worden of de gecachte response nog voldoet of dat de cache geïnvalideerd moet worden.

Uniform Interface. De centrale eigenschap van REST is de uniforme interface tussen componenten. Het doel hiervan is om een simpelere interface te maken die uniform is over verschillende soorten componenten waardoor deze onafhankelijk kunnen evolueren. Hierbinnen vallen vier constraints om de uniforme interface te bereiken; 'identification of resources', 'manipulation of resources through representations', 'self-descriptive messages', en 'hypermedia as the engine of application state'.

Identification of resources. Een van de belangrijkste eigenschappen van REST, zoals ook vergaand gebruikt in Rails, is het abstraheren van informatie in *Resources*. Kortweg komt dit neer op dat benoembare zaken (personen, documenten, handelingen, het weer in Amsterdam) geïdentificeerd kunnen worden als losse resources. Specifieker houdt het in dat ieder concept waarnaar verwezen kan worden binnen de definitie van een resource moet vallen. Een resource is dus een functie van een concept op een bijbehorende set entiteiten, niet één entiteit op een bepaald moment. De waarde van de functie kan dus veranderen over tijd (zoals "het weer van vandaag" waar de waarde van 'vandaag' elke dag verandert).

Het mechanisme voor identificatie is niet vastgesteld en is al een aantal keer veranderd. Tegenwoordig wordt dit vervuld met de 'International Resource Identifier', de geïnternationaliseerde versie van de bekende URI.

[...] **Representations**. De resources in een systeem worden benaderd door middel van acties op een representatie van het resource. De representatie is een sequentie van bytes met bijbehorende metadata (**self-descriptive**). In het geval van HTTP omvatten de headers de metadata en bevat de body de weergave van de staat van het achterliggende resource. De specifieke invulling van de achterliggende bytes wordt bepaald door het *media-type*. Veelal is dit HTML, maar ieder formaat kan hiervoor gebruikt worden. Het opvragen en aanmaken van resources zijn voorbeelden van acties.

Hypermedia As The Engine Of Application State, of 'HATEOAS', houdt in dat de hypermedia documenten de transities van staat moeten drijven. De staat van de client applicatie wordt dus getransitioneerd door middel van beschikbare transities in de huidige set van representaties. Het klikken op een link of het posten van een form in een HTML document is een voorbeeld van zo'n transitie. De client applicatie heeft daarbij dus geen kennis over de semantische inhoud van de desbetreffende transitie en wordt in geheel aangeleverd door de hypermedia.

Layered System. In het systeem mogen componenten alleen kennis nemen van hun eigen laag. Hierdoor kunnen deze eenvoudig gecomposeerd worden in een groter geheel. Elke laag kan de berichten voetstoots of getransformeerd doorgeven. Door de combinatie met statelessness en de uniforme interface is het bijvoorbeeld mogelijk dat load balancers een header toevoegen om meerdere requests van dezelfde client naar dezelfde server te routen.

Code on Demand. De laatste constraint ligt bij de client en is optioneel. Het toevoegen maakt de client echter wel een stuk krachtiger. De code on demand constraint vereist dat de client extra code kan downloaden van de server om extra functionaliteit te verkrijgen. Hierdoor kan de client simpeler geïmplementeerd worden maar is het mogelijk om over meerdere organisationele domeinen toch de complete functionaliteit voor elk domein te bereiken. In het moderne web is de toevoeging van javascript aan HTML pagina's de de facto manier om extra functionaliteit toe te voegen aan de client (browser).

Door te testen of software voldoet aan deze constraints kan bepaald worden waar modificaties nodig zijn om het ontwerp beter bij de principes van het web aan te laten sluiten.

6.2. Richard Maturity Model

Omdat de meeste API's op het web niet aan alle contraints van REST voldoen, is een model gedefinieerd waarmee de mate van REST compleetheid kan worden gemeten; het Richardson Maturity Model (Richardson, 2008). Het model bestaat uit drie lagen waarbij elke laag op de vorige voortbouwt.

De onderste laag, laag 0 (RMM-0), wordt gekarakteriseerd door het gebruik van één HTTP methode op één URL. Hier vallen onder andere de meeste RPC en SOAP systemen onder, waarbij de inhoud van de API call zich volledig in het applicatieprotocol bevindt.

In de volgende laag, laag 1 (RMM-1), beschrijft het systeem IRI's om resources te identificeren. Dit geeft de client de mogelijkheid om elk resource los te benaderen onder een URL.

In laag 2 (RMM-2) worden meerdere 'HTTP methodes' gebruikt. Hiermee worden de constraints van de bewerkingen op resources gedefinieerd door het onderliggende applicatie protocol HTTP(s) (Fielding & Reschke, 2014) waardoor de client aannames kan doen over een te maken request.

De laatste laag, laag 3 (RMM-3), introduceert hypermedia, wat inhoudt dat de client van de server instructies krijgt over hoe de IRI te construeren. Het kan zijn dat de IRI kant en klaar gegeven wordt, zoals in HTML anchor tags, of dat in het media type beschreven is hoe een IRI geconstrueerd kan worden, zoals bij HTML forms. Wanneer een API geen hypermedia controls biedt moet de client de interactie met diverse resources in het systeem hard-coden.

Wanneer een API aan laag 3 conformeert betekent dit echter niet dat de API ook aan alle contraints van REST voldoet. RMM-3 is namelijk een subset van REST.

6.3. Semantic Web & Linked Data

Op het web worden hyperlinks gebruikt om gebruikers van een webapplicatie door de applicatie te laten navigeren. Wanneer de eerste pagina in de browser geladen wordt, staat deze vol met links naar andere pagina's en forms om interactie met de dienst uit te oefenen. De client waar de gebruiker mee werkt, de browser, heeft hierbij geen kennis van de inhoud van de website[6].

Mensen die het web gebruiken, kunnen hiermee uit de voeten omdat de links een logische naam hebben waaruit blijkt wat de betekenis ervan is. Wanneer een applicatie de website zou gebruiken is het echter niet direct duidelijk wat een link zoals 'volgende' betekent.

Het *Semantic Web* lost dit op door de 'identification of resources' toe te passen op de velden in de data (Antoniou, Groth, Van Harmelen, & Hoekstra, 2012, p. 4), waardoor ook computers de betekenis achter de data kunnen interpreteren. De belangrijkste toevoeging van Semantic Web is dan ook *ontologie*[7], de tak van filosofie die zich bezighoudt met wat dingen betekenen.

Waar de filosofie zich bezighoudt met het abstracte concept, gebruiken we in computer-wetenschappen de gediscretiseerde versie; "*een ontologie*". In deze context hebben Borst & Nico (2012) een goede definitie gegeven: "*An ontology is a formal specification of a shared conceptualization*". De data in het SW/RDF worden door middel van URIs gekoppeld aan een betekenis. Er bestaan meta-ontologieën, maar in dit document zal naar de concrete versie die gebruikt wordt door applicaties gerefereerd worden.

De invulling van het Semantic Web is het Resource Description Framework. Het precieze verschil is niet relevant, daarom zullen de termen inwisselbaar gebruikt worden.

Een ontologie bestaat grofweg uit twee concepten: klassen en eigenschappen die programmatisch beschreven zijn. Elke klasse en eigenschap heeft een eigen IRI. Door de termen in bijvoorbeeld een JSON document te linken aan een ontologie kunnen programma's alle data die gelinkt is aan de geïmplementeerde concepten verwerken.

Zo komt bijvoorbeeld de eigenschap `name` niet meer als losse string voor, maar verwijst het naar de IRI `http://xmlns.com/foaf/0.1/givenName`, iemands voornaam. Ook kunnen klassen en hun relaties beschreven worden. Zo kan met `http://www.w3.org/2000/01/rdf-schema#subClassOf` de relatie van `http://schema.org/Person` en `http://schema.org/CreativeWork` met de superklasse `http://schema.org/Thing` beschreven worden.

Door deze concepten vast te leggen kunnen applicaties de betekenis van de data aannemen.

[6] Dit volgt logischerwijs uit de constraints van REST.
[7] Van het Grieks; ὄντος (ontos) "zijn, dat wat is" en -λογία (logia) "redenatie, de studie van, wetenschap"

7. Onderzoeksvragen

REST is een client-server model waarbij de front-end verantwoordelijk is voor weergave en de back-end verantwoordelijk is voor dataverwerking. AOD heeft een gescheiden front- en back-end naar dit model. Het is de bedoeling dat de Argu dienst wordt omgezet naar datzelfde model. De applicatie van AOD probeert dit model momenteel aan te houden door diverse databronnen in de back-end te transformeren naar een universele vorm die direct door de front-end ingeladen kan worden.

Zoals uit de kwestie naar voren komt brengt dit model problemen met zich mee. Logica moet gedupliceerd worden om de functionaliteit van de back-end te repliceren[8]. Omdat dit voor elk uniek type per databron moet gebeuren, zal dat snel in omvang toenemen. Dit is in feite een vorm van RPC omdat de client aannames doet over de vorm en structuur van de resources op de server. Het scheiden van de BE en FE voor Argu zal deze problemen ook ervaren. Daarnaast is een (intensieve) transformatie per databron vereist om het model te laten werken.

Uit de doelstellingen kan de vraag naar de onderliggende architectuur gedefinieerd worden;

> *"Wat is een toekomstbestendige backwards-compatible architectuur voor het overbrengen en weergeven van diverse data uit meerdere bronnen?"*

Hoofdvraag

Voordat geconcludeerd kan worden dat de architectuur geschikt is voor meerdere bronnen moet gekeken worden naar de inhoud van de (relevante) bronnen. Er bestaan immers verschillende bronnen met verschillende data en diverse manieren van serialisatie:

> *"Welke typen data en resources bestaan er in huidige bronnen?"*

Deelvraag 1

[8] Meer dan 100 acties over meer dan 20 modellen.

Het huidige systeem is gebaseerd op de REST architectuur. Echter voldoet het nog niet aan alle vereisten. Met het oog op de scheiding van de client en server, een van de fundamentele constraints van REST, is het interessant om te kijken naar de toegevoegde waarde van een volledige pure REST implementatie:

> *"Wat kan REST betekenen in de unificering van de interface?"*

Deelvraag 2

De REST architectuur is ontworpen als interface voor één applicatie per domein, dit wordt momenteel bereikt door het gebruik van data-warehousing. Om aan doelstelling twee te voldoen moet een manier gevonden worden om verschillende data uit meerdere bronnen eenvoudig te kunnen verwerken. In de terminologie van REST:

> *"Hoe kunnen verschillende data uit meerdere bronnen uniform beschreven worden?"*

Deelvraag 3

8. Onderzoek

8.1. Data en resources

Om antwoord te kunnen geven op de eerste deelvraag (*Welke typen data en resources bestaan er in huidige bronnen?*) is een analyse uitgevoerd op twee gebieden, het media-type en de resource typen. Ze geven inzicht in hoe data opgeslagen is en welke data beschikbaar is, waardoor een goed totaalbeeld van het landschap in open data ontstaat.

8.1.1. Media-type

Databronnen kunnen hoofdzakelijk in drie categorieën opgedeeld worden: ongestructureerd, semi-gestructureerd en gestructureerd (Sint et al, 2009). De databronnen die momenteel worden verwerkt binnen Argu is de data uit de Argu dienst zelf en de in Argu Open Data gebruikte bronnen. De data uit Argu is volledig (relationeel) gestructureerd en kan hierdoor omgezet worden in elk media-type.

Door te kijken naar de beschikbare publieke datasets kunnen we een beeld vormen van de beschikbare data op grote schaal. Er moet rekening gehouden worden met het feit dat dit datasets zijn, het percentage van datasets staat nog los van het percentage in beschikbare data omdat dat per set varieert.

Wanneer we kijken naar het dataportaal van de Nederlandse overheid zijn in Nederland 9725 datasets beschikbaar (“Monitor overzicht”, 2014). De drie meest voorkomende serialisatieformaten zijn JSON[9] (4447 sets), diverse geo-formaten (3361+ sets), en CSV (802 sets) (“Datasets”, n.d). Van de 9725 datasets beslaat de meerderheid (45%+) dus JSON geformatteerde data (“Datasets”, n.d). De Nederlandse open data is dus grotendeels semi-gestructureerd.

[9] Onder ‘JSON’ verstaan we het serialisatie- formaat, de structuur van de data is niet gestandaardiseerd en verschilt per bron.

8.1.2. Resource typen

Door het media-type te kennen weten we nog niet wat de overgedragen data betekent en dus ook niet hoe we het moeten weergeven. De data die zowel Argu als AOD weergeven is echter zeer gelijksoortig, het lijkt beide zeer veel op parlementaire data en gerelateerde objecten.

De data van Argu is op te splitsen in twee soorten, content objecten en overige objecten.

Onder ‘content objecten’ worden de resources verstaan waarin de discussie plaatsvindt; ‘Organisatie’, ‘Forum’, ‘Vraagstuk’, ‘Idee’, ‘Argument’, ‘Comment’. Deze reeks van typen zijn opgeslagen in een boom waarin elke type meerdere resources van het type eronder kan hebben (organisaties hebben meerdere fora, fora hebben meerdere vraagstukken, etc.).
De typen hebben veel eigenschappen gemeen, zo worden al deze objecten gekenmerkt met een titel[10] en een tekst, meerdere kinderen en een gemeenschappelijke parent (de organisatie). Vanaf het Vraagstuk niveau komen daar meer functionaliteiten bij; o.a. stemmen, besluiten, en blog posts.

De overige objecten in Argu zijn bedoeld om de content objecten te faciliteren zoals ‘User’, ‘Vote’, ‘Activity’, ‘Decision’, ‘Photo’, ‘Place’ en ‘BlogPost’ hebben geen sterke onderlinge verbanden.

Opvallend is de gelijksoortigheid van de modellen tussen AOD en Argu. Zo hebben o.a. de AOD ‘Person’, ‘Organisation’, ‘Motion’, ‘Vote’, en ‘Area’ allemaal een gelijksoortig type in Argu. Ook andere objecten vertonen veel van dezelfde overeenkomsten zoals ‘Kamerbrief’, ‘Motie’, ‘schriftelijke vragen’, etc.

8.1.3. Conclusie

De resources in zowel Argu als het Nederlandse open data ecosysteem bestaan voornamelijk uit semi-gestructureerde data opgeslagen in JSON formaat en beschrijft een divers spectrum aan resources die in types op te delen zijn. De grote overeenkomstigheid in de data biedt de mogelijkheid om generaliserende principes toe te passen.

[10] Alleen comments hebben geen titel.

8.2. REST

Om de BE van de FE te scheiden is een interface nodig om data tussen de twee uit te wisselen. De huidige norm is het maken van een 'RESTful' interface (Vitvar & Musser, 2010). Deze manier van gegevens overdragen zou moeten zorgen voor een schaalbaar mechanisme. Om de tweede deelvraag te kunnen beantwoorden (*Wat kan REST betekenen in de unificering van de interface?*), kunnen we kijken naar de theoretische gevolgen van het voldoen aan alle constraints. Hiervoor moet eerst gekeken worden naar de invulling van de ontbrekende eisen.

Wanneer de API voldoet aan de constraints van REST (Fielding, 2000) kan deze beschouwd worden als 'RESTful'. Aan de meeste constraints wordt al voldaan omdat de API op de HTTP specificatie is gebouwd. De HTTP specificatie is namelijk stateless, uniform, layered, en cachebaar (Fielding & Reschke, 2014; Nottingham et al, 2014). De belangrijkste die nog overblijft is 'Hypermedia As The Engine Of Application State'.

8.2.1. HATEOAS

8.2.1.1. Inleiding

Rails wordt verstaan als een 'RESTful' framework, zo zijn de objecten opgesplitst in los identificeerbare resources met een eigen URL die op een vaste manier geconstrueerd wordt ("Rails Routing from the Outside In", n.d.). Echter zijn de responses van Rails (in JSON) niet gelinkt met een URL. In de responses wordt een vaste manier gebruikt om naar een ander resource te verwijzen:

Template: "type_name_plural_id: <id van het resource>"
Ingevuld met resource "5" van het type "Photo": "photos_id: 5"
URL: "https://example.com/photos/5"

Figuur 1, transformatie naar URL's

Dit betekent dat Rails in JSON modus een RMM-2 framework is. Het heeft de front-end applicatie nodig om de ingevulde templates om te zetten in URL's. Wanneer voor deze implementatie gekozen zou worden zijn de front- en backend dus nog steeds onlosmakelijk verbonden; elke client heeft uitgebreide kennis nodig van de implementatie van de back-end.

Door de identifiers te vervangen met volledige URL's voldoen we al beter aan de 'Hypermedia as the engine of application state' constraint. We kunnen nu immers door de API navigeren[11] door middel van hyperlinks, waardoor de client-staat wordt bepaald door de server.

[11] Navigatie is een transitie in staat

In HTML wordt staat aan de server kant gewijzigd met forms. Echter biedt JSON geen soortgelijke mogelijkheid omdat het geen hypermedia formaat is. Het ontbreken van deze functionaliteit houdt in dat mensen bijvoorbeeld niet meer kunnen stemmen. Daarom moet worden gekeken naar een manier om staat ook aan de server kant te kunnen wijzigen.

8.2.1.2. Overwegingen

Er bestaan meerdere media-typen die voortbouwen op JSON en die hypermedia functionaliteit bevatten. Hieronder zullen meerdere media-typen uiteen gezet worden die hypermedia functionaliteit bevatten en JSON-compatibel zijn, gevolgd door de gedachtegang achter de uiteindelijke keuze.

HAL+JSON

De HAL+JSON specificatie is een IETF internet draft (Kelly, 2016) die een JSON structuur beschrijft om “RMM-3 API’s mee te ontsluiten”. De specificatie beschrijft standaard platte JSON documenten met twee gereserveerde keys voor respectievelijk links en bijgevoegde documenten.

Hydra

Hydra is een specificatie met ontologie voor het beschrijven van API’s (Lanthaler, 2016). Het wordt gebruikt in samenhang met JSON-LD om data uit te wisselen. De ontologie geeft definities om resources, collecties en acties van een API mee te beschrijven die gebruikt kunnen worden in de JSON-LD context.

JSON-LD

Ook JSON-LD is een officiële W3C aanbeveling (Sporny, Longley, Kellogg, Lanthaler, & Lindström, 2014). In JSON-LD staat linked data centraal.
Gebaseerd op de JSON syntax definieert JSON-LD een aantal manieren om ontologische metadata (“context”) over te brengen. Op deze manier biedt JSON-LD een manier om keys (“termen”) in JSON documenten te linken aan diverse waarden (veelal een IRI naar een ontologie) waardoor deze betekenis krijgt.
De invoeging van deze context kan door het te embedden in het document zelf onder de “@context” key of wanneer aanpassing niet mogelijk is, het toevoegen van een header aan het HTTP bericht.

JSON:API

JSON:API is een community specificatie die naast de datastructuur ook de manier en vorm van interactie met de API van tevoren vastlegt (“JSON API — latest specification (v1.0)”, 2015). Het stelt vast hoe resources in het systeem moeten worden benaderd met het uitgangspunt van CRUD operaties op resources. In de nieuwste versie van Rails is een serialisatie module toegevoegd. De module is in principe niet gebonden aan een formaat, maar de eerste en aanbevolen implementatie om te gebruiken is JSON:API.

8.2.1.3. Conclusie

Om de keuze te maken moeten we kijken naar de belangrijkste eisen van het formaat. Dat is de functionaliteit om hypermedia besturing aan de API toe te voegen voor lezen én schrijven.
Alle beschreven standaarden bieden hypermedia besturing, maar slechts twee bieden ook mogelijkheden voor het schrijven van informatie; Hydra en JSON:API.

Beide formaten voldoen dus aan de constraints van REST. Buiten REST heeft JSON:API nog een ander merkbaar voordeel, de module voor Rails bestaat al waardoor deze niet door Argu geïmplementeerd hoeft te worden. Dit is dan ook doorslaggevend voor de uiteindelijke keuze, die valt op JSON:API.

8.2.2. Conclusie

Door het gebruik van HTTP als bouwsteen van het platform, in combinatie met de eigenschappen van Rails om losse URL's en HTTP methodes te gebruiken, en het nieuw toegevoegde hypermedia formaat, staat het theoretisch ontwerp voor een RMM-3 REST API.

In deze API hoeft de front-end geen kennis meer te hebben over de links naar andere resources, want deze worden door de back-end gegeven. Ook heeft de front-end in theorie een manier om informatie op te vragen en weg te schrijven wanneer het type van het resource bekend is.

Door de verdere implementatie van REST is onze applicatie dus ontkoppeld van de databron, waardoor de gebruiker vrij is om tussen verschillende bronnen te navigeren. Dit is echter wel beperkt tot vooraf bekende resources.

8.3. Meerdere bronnen

8.3.1. Inleiding

Nu we weten hoe data tussen BE en de FE uitgewisseld kan worden in een REST interface moeten we vooruitkijken naar de integratie met AOD. Hiervoor is het benodigd om de front-end applicatie te laten werken met data uit meerdere bronnen. In deze paragraaf beantwoorden we de deelvraag "*Hoe kunnen verschillende data uit meerdere bronnen uniform beschreven worden?*".

Zoals blijkt uit '7.1.1 Media-type' gaat het om semi-gestructureerde data geserialiseerd in JSON formaat. Om de data te verwerken moet de front-end deze kunnen interpreteren.

In de meeste REST applicaties kent de FE de datastructuur van de BE. Dit houdt in dat het voor de FE bekend is welke resource-typen op welke URL's geserveerd worden en wat de velden in de JSON response betekenen. Dit leidt veelal tot duplicatie van code omdat zowel de URL's als de beschrijving van de achterliggende modellen al in de BE bestaan. Bovendien is dit niet volgens de REST principes omdat de client en server dan sterk gekoppeld zijn: "A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server)" (Fielding, 2008).

Het doel van dit project is in eerste instantie om de BE en FE van de Argu dienst te ontkoppelen. Echter blijkt de data van Argu en AOD dusdanig gelijksoortig te zijn[12] dat een interessante invalshoek is om Argu als een van de databronnen te zien. Dan kunnen de twee losse applicaties met kennis van de achterliggende API's worden getransformeerd naar één uniforme echt RESTful client.

Met REST hebben we een uniforme interface voor het benaderen van de data, maar REST is origineel hoofdzakelijk ontworpen voor menselijk gebruik (Fielding, 2000, p. 86). Het aanleveren van data in HTML formaat met toegevoegde JavaScript en CSS was voldoende om informatie per domein beschikbaar te maken voor mensen. In de zin van REST is alleen de browser de client, de FE is simpelweg 'Code on Demand' om de client extra functionaliteit te geven. Om van de FE een volwaardige REST client die meerdere databronnen kan interpreteren te maken, is een manier nodig om data uniform te beschrijven.

[12] Zie '7.1 Data en resources'

8.3.2. Semantic Web

Het framework die deze eis kan vervullen is het ‘Semantic Web’. Wanneer we kijken naar de definitie van W3C (2013) is te zien dat deze aansluit bij de eis van het uniform weergeven van data uit meerdere bronnen: *“The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries.”*

Om de benodigde informatie voor Semantic Web over te kunnen brengen moeten de responses uit de API verrijkt worden met semantische informatie. Omdat zowel de API van Argu als de meeste open data API’s data beschikbaar stellen in JSON is het logisch om uit de vele formaten voor het semantic web de JSON extensie ‘JSON-LD’ te kiezen. Deze kan geïntegreerd worden in bestaande API’s door middel van het toevoegen van een header in de HTTP response of door twee extra velden in de JSON body (Sporny, Longley, Kellogg, Lanthaler, & Lindström, 2014).

Zoals uit het resultaat van *7.2.1. HATEOAS* blijkt, specificeert JSON-LD geen besturing voor het schrijven van data. Dit is voor open data geen probleem, dat is immers alleen uitlezen. Voor de Argu data wordt dit afgehandeld door JSON:API. De JSON-LD extensie kan zonder problemen worden toegevoegd (json-api, n.d.) om semantische metadata over te brengen.

8.3.3. Conclusie

Hoewel de overdracht van data geünificeerd was, gold dit nog niet voor de overdracht van de betekenis van data. Hierdoor was er geen mogelijkheid voor de FE om de data te interpreteren zonder voorkennis over de BE. Om dit probleem op te lossen is gezocht naar een manier om de data uniform te beschrijven.

Deze beschrijvingen zijn gevonden in de concepten en uitwerkingen van het Semantic Web/Resource Description Framework. Door het gebruik van ontologieën kunnen dataleveranciers betekenis overbrengen naar externe applicaties, bedrijven en gemeenschappen. Ook is een backwards-compatible manier aanwezig om JSON data te voorzien van zulke semantische beschrijvingen, wat de deur opent naar het verwerken van open-datasets zonder transformatie.

8.4. Conclusie onderzoek

Het doel van het onderzoek was een architectuur ontwerpen waarmee van een monolithische applicatie een losse front- en back-end gemaakt kan worden, die (toekomstige) open data met zo min mogelijk transformatie kan weergeven.

Hiervoor is eerst gekeken naar welke soorten data verwerkt moeten worden. Dit blijkt voor het grootste deel uit semi-gestructureerde JSON data te bestaan. Ook is er een coherente overeenkomst in de achterliggende concepten van de data. Veel data bestaat uit variaties van dezelfde concepten, zo is het grootste verschil tussen een `Hamerstuk` en een `Motion` de naam en mogelijke relaties. De grote overeenkomsten bieden mogelijkheid tot generalisering.

Vervolgens is de impact van het behalen van het RMM-3 REST niveau geanalyseerd. Eerst is een afweging gemaakt van het te gebruiken media-type, vervolgens is de impact daarvan beredeneerd. Van de backwards compatible formaten is JSON:API het meest geschikt. Het voldoet aan de eisen en er bestaat reeds een Rails module om de functionaliteit te realiseren. De front-end kon al gescheiden worden van de back-end, maar verdere implementatie van REST zal er voor zorgen dat het ook van elkaar ontkoppeld zal zijn. Dit stelt de front-end in staat om door data uit meerdere bronnen te kunnen navigeren.

Nu de BE en FE ontkoppeld zijn, komt het laatste vraagstuk aan bod. Hoewel de front-end getransformeerd is naar een volledige REST applicatie, kan deze alleen een bron uitlezen als deze exact voldoet aan de Argu back-end API. Om dit probleem te overkomen is er gezocht naar een methode om verschillende data uniform te beschrijven. Deze functionaliteit is vormgegeven in het *Resource Description Framework*, een manier om data programmatisch te beschrijven waardoor deze machine interpretable wordt. De JSON-LD maakt het mogelijk deze informatie over te brengen, ook zonder aanpassingen aan de datastructuur.

Nu we hebben gezien hoe de deelvragen opgelost zijn, is het antwoord op de hoofdvraag ook duidelijk;

> *Wat is een toekomstbestendige backwards-compatible architectuur voor het overbrengen en weergeven van diverse data uit meerdere bronnen?*
>
> *De REST architectuur met de toegevoegde constraint van semantisch beschreven hypermedia*.

De uiteindelijke architectuur creëert een stabiele basis door een scheiding tussen server en client door REST, met de toegevoegde scheiding tussen databron en betekenis door het gebruik van het Resource Description Framework. Het is geen gouden ei, de data moet immers semantisch geannoteerd worden, maar dat kan met de toevoeging van een header. De FE staat los van de BE en transformatie van data is overbodig gemaakt.

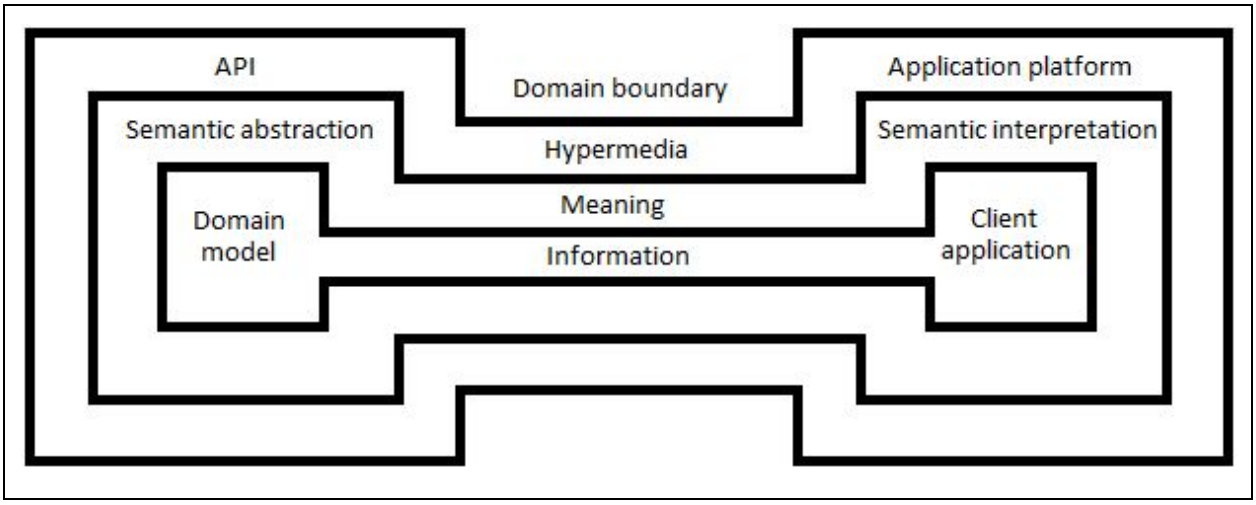
9. Architectuur

Zoals uit de onderzoeksresultaten naar voren is gekomen zal de architectuur voortbouwen op de REST architectuur die ten grondslag ligt aan het moderne web. Dit voldoet echter nog niet aan alle benodigde eisen waardoor de extra constraint van semantische annotatie toegevoegd moet worden.

De bedoeling van de semantische annotatie is om data uniform te beschrijven zodat de client applicatie is gescheiden van de data(bron). Specifiek gezegd, de client verwerkt informatie die is overgebracht met de semantische inkapseling van data. In figuur 2 is deze architectuur schematisch weergegeven.

Dit model bouwt voort op REST, met links de '*server*' en rechts de '*client*' weergegeven. De buitenste laag '*hypermedia*' valt ook binnen REST. De binnenste laag is de informatielaag, dit omvat het domein model aan de *server* kant en de applicatie aan de *client* kant. Een concreet voorbeeld van de informatielaag is de nos.nl webserver met de HTML5 applicatie die zij leveren om het te laten 'zien en voelen' als nos.nl.

De semantische constraint is omvat in de '*meaning*' laag. Deze laag zit tussen de informatie en de hypermedia laag, wat wil zeggen dat de client en de server moeten communiceren door middel van semantische uitdrukkingen. Het voorbeeld aanhoudende, de nos.nl server stuurt geen specifiek geformateerde HTML pagina meer, maar "'een lijst van 'http://schema.org/Article'", en de client verwerkt geen `h1.article__title` maar een 'http://schema.org/headline', gebruikt door een miljoen andere domeinen.



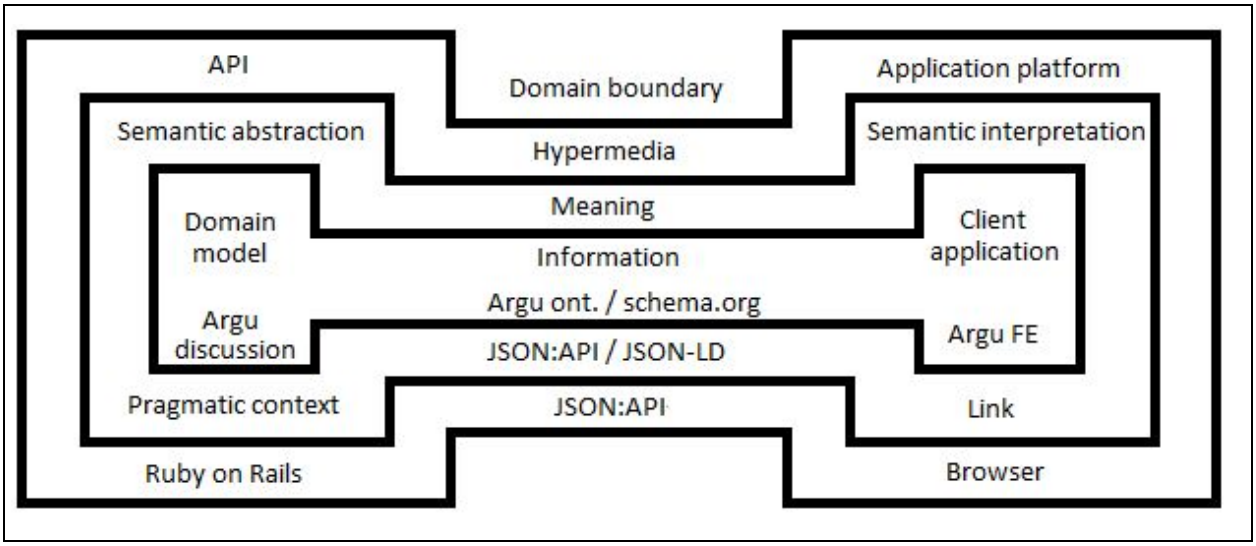
Figuur 2, semantische inkapseling van informatie binnen de REST architectuur.

Binnen het model kan, net zoals in het OSI model, elke laag worden ingevuld met invulling van de functionaliteit. In figuur 3 is het model ingevuld met de voor Argu te gebruiken technologieën.

De client en server software uit de *hypermedia* laag staan al vast, dit zijn Ruby on Rails en de browser van de gebruiker. Voor de hypermedia interconnectie is uit het onderzoek gebleken dat JSON:API het best geschikt is.

Voor de *meaning* laag zal JSON-LD als interconnectie gebruikt worden. Ook JSON:API bevat semantische informatie voor schrijfoperaties waar gebruik van gemaakt zal worden. De server kant zal met de 'pragmatic context' library ingevuld worden, waarmee JSON-LD contexten gegenereerd kunnen worden. Aan de client kant zal een custom library ('*Link*') de overdracht en interpretatie van data tussen de client applicatie en de API voorzien.

Voor de *information* laag wordt relevante functionaliteit rondom het 'Argu discussie' domein geïmplementeerd, het domein zal beschreven worden met RDF, waardoor de informatie automatisch verwerkt kan worden. Bij het definiëren van de termen zal per term een afweging worden gemaakt tussen het hergebruiken van termen in bestaande ontologieën en het uitbreiden van nieuwe termen in de 'Argu ontologie'.



Figuur 3, model ingevuld voor Argu

Aan de hand van dit model kan de implementatie vervuld worden om de overstap te maken naar een volledige ontkoppeling tussen front- en back-end.

10. Opgeleverde producten

10.1. Front-end

10.1.1. Algemeen

Om aan de architectuur te voldoen moet de front-end van een simpele ‘Code on Demand’ toevoeging uitgroeien naar een volledige REST client. Waar een normale FE simpelweg data van de bijbehorende back-end kan verwerken moet de nieuwe applicatie die uit meerdere bronnen kunnen verwerken. Hiervoor zijn de navigatie, de dataoverdracht, en de semantische interpretatie in het ontwerp verwerkt als onderdeel van de overkoepelende API.

Om de FE te laten uitgroeien tot een volledige client is het opgedeeld in drie lagen;

Applicatielaag. De laag waarin applicatiebouwers nieuwe applicaties ontwikkelen. Deze zijn gefocust op het vertalen van data naar informatie en het doen van bewerkingen op die informatie. De invulling hiervan is voor de ontwikkelaar, het framework bevat deze code niet.
Ontology connector. Dit is de laag gedeeltelijk blootgesteld aan applicatiebouwers. Door een puur ontologische interface bloot te stellen worden de resources losgekoppeld van de weergave. Hierdoor kan de server de compositie en vorm van de data aanpassen zonder dat de client applicatie breekt.
Media-type connector. Deze laag is verantwoordelijk voor het deserialiseren van responses uit externe API’s zoals JSON:API. Tevens kan het worden uitgebreid met bijv. Hydra en RDFa.

Deze lagen zijn uitgewerkt in een nieuw hypermedia framework genaamd *“Link”*. Dit framework stelt de programmeur in staat om hypermedia gedreven applicaties te bouwen op een declaratieve manier. In plaats van een applicatie te schrijven of aan te passen op basis van de databron, kan de programmeur met het framework beschrijven hoe bepaalde soort data weergegeven moet worden. Het ontkoppelt dus informatie (presentatie) van data.

Dit heeft als voordeel dat implementatiedetails van de interactie met de server worden afgehandeld door de onderliggende componenten. Het verandert het paradigma, waarbij de taak van de applicatieontwikkelaar wordt beperkt tot het veld van specialisme; het ontwerpen van weergave en interactie met eindgebruikers in plaats van met machines.

Zoals in figuur 4 te zien is kan de applicatieontwikkelaar zich volledig bezighouden met de presentatie van informatie. Alle implementatiedetails van de achterliggende service zijn weg geabstraheerd; Argu gebonden termen zoals `Motion` en `title` zijn vervangen door IRI's uit gestandaardiseerde ontologieën: 'http://schema.org/CreativeWork' en 'http://schema.org/name'. Zo wordt de term 'http://schema.org/name' door meer dan 1.000.000 domeinen gebruikt. In dit voorbeeld wordt de URL van een motie gerenderd met de `LinkedObjectContainer` (buiten beeld) die de render klasse `CreativeWork` deduceert uit de ontologie.

Standaard React:

```
const MotionShow = ({
  title,
  children,
  creator,
  createdAt,
  onVote,
  voteData,
}) => (
  <Card>
    <CardHeader noSpacing>
      <Heading>{title}</Heading>
      <DetailsBar>
        <DetailType type="motion" />
        {creator && <PersonContainer user={creator} />}
        <DetailDate date={createdAt} />
      </DetailsBar>
    </CardHeader>
    <CardContent noSpacing>{children}</CardContent>
    <CardActions>
      {Object.keys(options).map(option => (
        <CardButton
          key={option}
          active={voteData === option}
          action={() => onVote(option)}
          type={option}
        >{options[option]}</CardButton>
      ))}
    </CardActions>
  </Card>
);
```

Type:

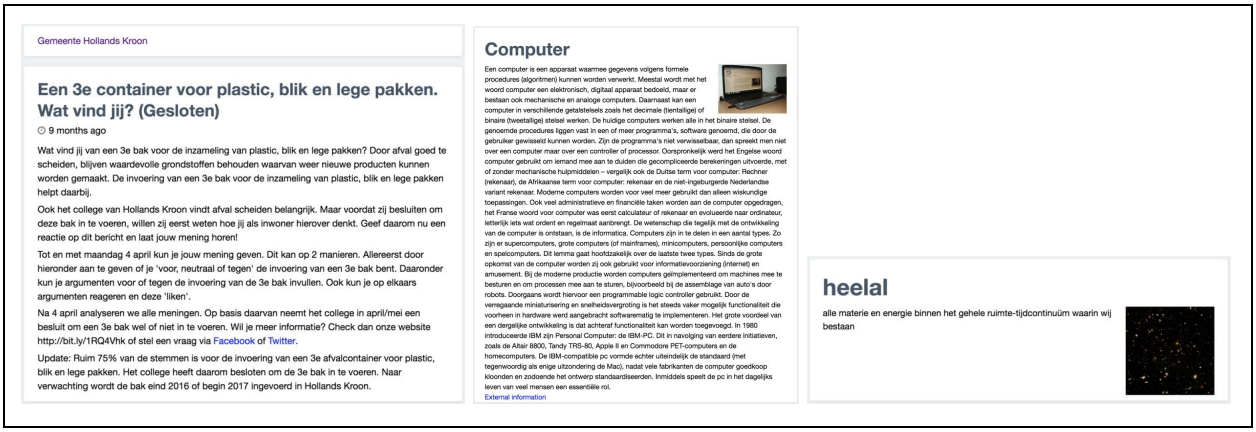
```
const CreativeWork = () => (
  <Card>
    <CardHeader noSpacing>
      <Property label="schema:name" />
      <DetailsBar>
        <LinkedDetailDate />
        <Property label="schema:creator" />
      </DetailsBar>
    </CardHeader>
    <CardContent noSpacing>
      <Property label="schema:text" />
    </CardContent>
    <CardActions>
      <Property label="schema:updateAction" />
    </CardActions>
  </Card>
);

LinkedRenderStore.registerRenderer(
  CreativeWork,
  'http://schema.org/CreativeWork'
);
```

'name' property:

```
class Name extends PropertyBase {
  render() {
    return (
      <Heading size="2">
        {this.getLinkedObjectProperty()}
      </Heading>
    );
  }
}
```

Figuur 4, implementatie van een view voor en na Link

De toegevoegde constraints aan REST hebben als gevolg dat de interface dusdanig generiek is geworden dat de applicatie voor elke voldoenende databron compatibel is, zolang een koppeling in de client aanwezig is om de informatie naar de volgende laag door te geven. Een voorbeeld hiervan is te zien in figuur 5.



Figuur 5, Argu motie via JSON:API, DBpedia 'computer' via 'text/turtle', Wikidata 'heelal' via 'application/rdf+xml' met dezelfde code.

Link is opgedeeld in twee componenten. De onderliggende laag die de media-type connector en de ontology connector bevat; "*Link-Library*", en een implementatie van de applicatielaag voor React; "*Link-React*".

10.1.2. Link-Library[13]

Dit onderdeel omvat zowel de ontology connector als de media-type connector.

Omdat de applicatie data uit meerdere bronnen moet weergeven, moet de in-memory representatie van resources losgekoppeld worden van het bijbehorende media type. De media-type connector is hier verantwoordelijk voor. Zo lang een media-type semantisch geannoteerd is kan de ontologische informatie met attributen worden doorgegeven naar de resource connector. Wanneer de ontologische beschrijving beschreven is in het media-type zelf kan dit in de connector verwerkt worden. De ontkoppeling van het media-type van de betekenis van de data zorgt ervoor dat het media-type aangepast kan worden zonder dat deze niet meer kan worden omgezet naar een ontologische beschrijving. De eerste implementatie kan onder andere JSON:API, JSON-LD, Turtle, en NTriples responses verwerken[14].

De volgende laag is het hart van het framework: de ontology-connector. Deze verwerkt en interpreteert de inkomende data, en bevat de mechanismen om queries van de bovenliggende componenten te verwerken en te beantwoorden. Het belangrijkste component is de `*LinkedRenderStore*`. Dit component draait hoofdzakelijk om twee dingen: de programmeur kan registreren welke klassen welke typen of attributen moeten renderen en vervolgens kan de programmeur opvragen welke klasse het best geschikt is voor een bepaald datum.

Met de onderliggende lagen van Link kan een ontwikkelaar dus databronnen inladen die klaar worden gemaakt voor gebruik door de applicatie.

[13] Open source gepubliceerd op http://github.com/fletcher91/link-lib
[14] Media typen anders dan JSON:API worden verwerkt door een externe RDF library.

10.1.3. Link-React[15]

Nadat de data in het systeem is ingeladen moet deze weergegeven worden. Dit is in principe niet gebonden aan een enkel framework, maar Argu had voor het React framework gekozen, waardoor de implementatie volgde.

Link-React draait voor de applicatieontwikkelaars voornamelijk om een tweetal React componenten om invulling te geven aan de weergave van de data.

Met de `*LinkedObjectContainer*` kan een ontwikkelaar aangeven dat op die locatie[16] een resource gerenderd moet worden. Het te renderen resource wordt aangegeven met een IRI. Het component is gekoppeld aan de ontology-connector, die de juiste klasse voor het renderen van de data beredeneerd (multiple inheritance en polymorphisme worden hierbij ondersteund). De implementatie die de uiteindelijke weergave rendert ligt dus in het domein van de ontwikkelaar (de *applicatielaag*). De LinkedObjectContainer creëert een context waarin een 'huidig resource' aangegeven en beschikbaar gemaakt wordt. Hoe het resource opgehaald wordt, is voor de ontwikkelaar niet meer relevant.

In die context kan met het `*Property*` component een eigenschap van het resource gerenderd worden. Dit is soortgelijk aan de LinkedObjectContainer in dat het gebruikt wordt om geregistreerde klassen mee te renderen waar zich de daadwerkelijke implementatie van de weergave bevindt. Wanneer de programmeur het `*PropertyBase*` component als superklasse voor de renderer gebruikt, komt de waarde van de eigenschap met een simpele methode beschikbaar.

Veel typen en eigenschappen worden voor andere types grofweg hetzelfde weergegeven (zoals een `titel`), deze kunnen voor een type hoog in de klasse hiërarchie geschreven worden (Zoals `Thing`), waarna ze voor alle subtypen beschikbaar komen. De weergave kan vervolgens waar nodig overschreven worden (zoals de kleur voor een 'argument voor' groen maken).

[15] Open source gepubliceerd op http://github.com/fletcher91/link-redux
[16] In React worden componenten in een XML-achtige manier genest, wat de uiteindelijke positie in de grafische weergave bepaalt.

10.2. Back-end

10.2.1. Overzicht

De back-end is geüpgraded naar de nieuwste versie van Rails, waardoor de nieuwe serializers beschikbaar kwamen. Deze zijn vervolgens ingevuld voor de belangrijkste modellen.

Vervolgens is voor de integratie van de semantische context een library gebruikt waar per model beschreven kan worden welke attributen overeenkomen met welke ontologische definitie. De implementatie van deze library was niet gericht op gebruik in tandem met de serializers. Dit is overkomen door het toevoegen van een gezamenlijke superklasse die een virtueel attribuut met de semantische beschrijving bevat.

Voor de beveiliging van de API is een OAuth2 provider opgezet. De FE kan requests naar de BE authenticeren door het meegeven van een token in de ‘Authorization’ request header. Dit houdt de API stateless. Tevens kan de header binnen de infrastructuur van Argu doorgestuurd worden zodat mogelijke microservices authenticatie kunnen delegeren waardoor ze een eenduidige implementatie behouden.

11. Verdere aanbevelingen

Het platform werkt momenteel en is al toekomstbestendig. Desalniettemin zijn er meerdere aanbevelingen waarmee de architectuur, het platform, en de software verbeterd kunnen worden. Hieronder volgt een aantal aanbevelingen die kunnen helpen om de software efficiënter te maken en de adoptie bij meerdere organisaties te verbeteren.

11.1. Gebruik van HTTP OPTIONS

Momenteel is het media-type verantwoordelijk voor het beschrijven van de mogelijkheden met de API. Dit betekent ook dat de mogelijkheden moeten worden verstuurd in de standaard response, dit principe leidt tot een verslechterde mogelijkheid tot caching omdat user dependent acties verschillen per gebruiker. De response voor een algemeen object (zoals een motie) kan dus slecht gecached worden omdat acties zoals ‘bewerken’ mogelijk aanwezig kunnen zijn.
In het zicht van HTTP/2 multiplexing (Belshe et al, 2015) wordt het aanzienlijk goedkoper om meerdere requests te sturen. Dit betekent dat de client door het gebruik van OPTIONS kan bepalen welke interface elementen met betrekking tot uit te voeren acties gerenderd moeten worden zonder dat de achterliggende resources niet (in totaliteit) gecached kunnen worden.

11.2. Code On Demand

De laatste constraint van REST is Code on Demand en is als enige optioneel en het huidige applicatieplatform ondersteunt dit nog niet. Om de architectuur op lange termijn schaalbaar te houden terwijl duizenden organisaties op maat gemaakte ontologieën uitrollen, is het wenselijk dat die organisaties ook hun eigen views voor de desbetreffende ontologieën publiceren. Dit betekent dat de schrijver van de applicatie niet meer elke variatie van elk resource type handmatig hoeft te ondersteunen.
Deze functionaliteit kan ook benut worden om de basisapplicatie klein te houden en alleen de programmatuur te downloaden wanneer data van dat type weergegeven wordt. Een voorbeeld use case zijn widgets, de voetafdruk is dan zeer klein en cachebaar en op basis van het getoonde resource wordt alleen gedownload wat benodigd is. Ook kunnen derde organisaties het widget platform uitbreiden om de overdracht van informatie naar kennis voor hun eigen datatypes te verbeteren.

11.3. Type inference

In de huidige implementatie is een referentie naar de ontologische RDF specificatie benodigd om een resource correct te renderen. In de toekomst kan dit systeem uitgebreid worden om op basis van een heuristiek een inschatting te maken van het meest waarschijnlijke type. Zodoende wordt data zonder ontologische beschrijving dan wel met technisch mankement alsnog parsebaar.

11.4. Web components

Het huidige platform is geschreven met React. Dit is voordelig omdat er een React prototype bestond en in het team al beperkte kennis over de werking en complexiteit van React/Redux applicaties aanwezig was. Naarmate browser support voor de Web Components API toereikender wordt kan het platform ook in Web Components geschreven worden. Met de juiste standaardisering van de huidige twee elementen en register calls kan het framework opgenomen worden in mainstream browsers.

12. Bibliografie

Actieplan open Overheid 2016-2017. (2016, February). Retrieved November 10, 2016, from Open Overheid, http://www.open-overheid.nl/actieplan-open-overheid-2016-2017/

Antoniou, G., Groth, P. E., Van Harmelen, F., & Hoekstra, R. (2012). A semantic web primer (3rd ed.). Cambridge, MA: The MIT Press.

Belshe et al. (2015, May). Hypertext transfer protocol version 2 (HTTP/2). Retrieved November 14, 2016, from https://tools.ietf.org/html/rfc7540

Borst, & Nico, W. (2012, April 16). Construction of engineering Ontologies for knowledge sharing and reuse. Retrieved December 1, 2016, from http://doc.utwente.nl/17864/

Datasets. Retrieved November 27, 2016, from Dataportaal van de Nederlandse overheid, https://data.overheid.nl/data/dataset?_res_format_limit=0

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Doctoral dissertation, University of California, Irvine).
JSON API — latest specification (v1.0). (2015, May 29). Retrieved November 11, 2016, from http://jsonapi.org/format/

Fielding, R., & Reschke, J. (2014, June). Hypertext transfer protocol (HTTP/1.1): Semantics and content. Retrieved November 12, 2016, from https://tools.ietf.org/html/rfc7231#section-4.2

Fielding, R., & Reschke, J. (2014, June). Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. Retrieved November 28, 2016, from https://tools.ietf.org/html/rfc7230

Fielding, R. T. (2008, October 20). REST APIs must be hypertext-driven. Retrieved November 29, 2016, from Untangled musings of Roy T. Fielding, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

json-api. Allow JSON LD to be used with JSON API by ethanresnick · pull request #995 · json-api/json-api. Retrieved December 2, 2016, from https://github.com/json-api/json-api/pull/995

Kelly, M. (2016, May 12). JSON Hypertext application language. Retrieved November 11, 2016, from https://tools.ietf.org/html/draft-kelly-json-hal-08

Lanthaler, M. (2016, June 05). Hydra: Hypermedia-Driven web APIs. Retrieved November 11, 2016, from http://www.hydra-cg.com/spec/latest/core/

Rails Routing from the Outside In (n.d.). Retrieved November 10, 2016, from http://edgeguides.rubyonrails.org/routing.html

Richardson, L. (2008, November 20). Justice Will Take Us Millions Of Intricate Moves. Retrieved November 10, 2016, from https://www.crummy.com/writing/speaking/2008-QCon/act3.html

Sint, R., Schaffert, S., Stroka, S., & Ferstl, R. (2009, June). Combining unstructured, fully structured and semi-structured information in semantic wikis. In Fourth Workshop on Semantic Spolsky, J. (2000, April 6). Things you should never do, part I. Retrieved November 12, 2016, from http://www.joelonsoftware.com/articles/fog0000000069.html

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., & Lindström, N. (2014, January 16). JSON-LD 1.0. Retrieved November 29, 2016, from https://www.w3.org/TR/json-ld/

Monitor overzicht. (2014). Retrieved November 27, 2016, from Dataportaal van de Nederlandse overheid, https://data.overheid.nl/monitor

Nottingham, M., Fielding, R., & Reschke, J. (2014, June). Hypertext transfer protocol (HTTP/1.1): Caching. Retrieved November 28, 2016, from https://tools.ietf.org/html/rfc7234

Vitvar, T., & Musser, J. (2010, December 6). Programmableweb.Com: Statistics, Trends and Best Practices. Retrieved November 28, 2016, from http://www.slideshare.net/mashups/web-ap-is

Vaquero, L. M., Rodero-Merino, L., & Buyya, R. (2011). Dynamically scaling applications in the cloud. ACM SIGCOMM Computer Communication Review, 41(1), 45. doi:10.1145/1925861.1925869

W3C. (2013, June 19). W3C semantic web activity. Retrieved November 29, 2016, from https://www.w3.org/2001/sw/

Wikis–The Semantic Wiki Web 6 th European Semantic Web Conference Hersonissos, Crete, Greece, June 2009 (p. 73).