ResearchDocument

Mickey en Matthijs

# Versiebeheer:

|  |  |  |
| --- | --- | --- |
| Versie nr. | Datum: | Aanpassingen |
| 1 | 04/11/2020 | Opzet document |
| 2 | 18/11/2020 | Toevoeging frontend tests |
| 3 | 19/11/2020 | Architectuurschets toegevoegd |
| 4 | 20/11/2020 | Interview |
| 5 | 2/12/2020 | Toevoeging 2e onderzoeksvraag |
| 6 | 11/12/2020 | Inleiding security |
| 7 | 17/12/2020 | Toevoeging Owasp zap |

Inhoud

[Versiebeheer: 2](#_Toc59096630)

[ResearchMethodes 6](#_Toc59096631)

[Onderzoeksvragen: 7](#_Toc59096632)

[Hypotheses 7](#_Toc59096633)

[Wat zijn onze grootste valkuilen qua security binnen ons project? 8](#_Toc59096634)

[Inleiding 8](#_Toc59096635)

[Security 8](#_Toc59096636)

[Owasp 8](#_Toc59096637)

[1. Injection 8](#_Toc59096638)

[2. Broken Authentication 9](#_Toc59096639)

[3. Sensetive data exposure 10](#_Toc59096640)

[4. Xml external entities 11](#_Toc59096641)

[5. Broken acces control 12](#_Toc59096642)

[6. Security Misconfiguration 13](#_Toc59096643)

[7. Cross-site Scripting xss 13](#_Toc59096644)

[8. Insecure deserialization 14](#_Toc59096645)

[9. Using components with known vulnerabilities 15](#_Toc59096646)

[10. Insufficient logging and monitoring 16](#_Toc59096647)

[Wat: 16](#_Toc59096648)

[Hoe: 16](#_Toc59096649)

[Security test 17](#_Toc59096650)

[Sanitization and security 17](#_Toc59096651)

[XSS Sanitization loading in a htmlsnippet 17](#_Toc59096652)

[XSS Sanitization Url loading 19](#_Toc59096653)

[Best Good and bad practices 21](#_Toc59096654)

[Code review 21](#_Toc59096655)

[Owasp Zap 21](#_Toc59096656)

[Voordelen Zap 22](#_Toc59096657)

[Gebruik van Zap 22](#_Toc59096658)

[Passive mode 22](#_Toc59096659)

[Spider 23](#_Toc59096660)

[Active Mode 23](#_Toc59096661)

[Owasp resultaat 25](#_Toc59096662)

[Conclusie 26](#_Toc59096663)

[Nabespreking 26](#_Toc59096664)

[Hoe test je een microservice architectuur effectief? 27](#_Toc59096665)

[Inleiding 27](#_Toc59096666)

[Testen van een applicatie 28](#_Toc59096667)

[TestStructuur 29](#_Toc59096668)

[29](#_Toc59096669)

[1. UI tests 30](#_Toc59096670)

[2. End to end tests 30](#_Toc59096671)

[3. Intergration tests 30](#_Toc59096672)

[4. Unit tests 30](#_Toc59096673)

[5. Smoke tests 30](#_Toc59096674)

[6. Performance testing 30](#_Toc59096675)

[Interview 31](#_Toc59096676)

[Vragen 31](#_Toc59096677)

[1. Heb je ooit gewerkt met het testen van een microservices architectuur? 31](#_Toc59096678)

[2. Welke soort tests vind je het belangrijkst bij het testen? 31](#_Toc59096679)

[3. Bij welke tests begin je meestal? 31](#_Toc59096680)

[4. kom je vaak fouten tegen na het testen? 31](#_Toc59096681)

[5. Hoe breidt je je tests uit na mate je applicatie groter wordt? 31](#_Toc59096682)

[6. Maak je een plan voordat je gaat testen? 31](#_Toc59096683)

[7. Probeer je altijd alles te testen of zijn er componenten die niet belangrijk genoeg zijn. 31](#_Toc59096684)

[8. Wanneer trek je de conclusie dat je applicatie genoeg getest is? 31](#_Toc59096685)

[9. Hoe belangrijk is de benaming als je je tests schrijft? 32](#_Toc59096686)

[10. Waar testen jullie voornamelijk jullie applicatie mee? 32](#_Toc59096687)

[11. Wat vind je van jullie manier van testen? 32](#_Toc59096688)

[12. verlopen de tests altijd volgens plan? 32](#_Toc59096689)

[13. eigen toevoeging. 32](#_Toc59096690)

[Conclusie 32](#_Toc59096691)

[Document Analysis 33](#_Toc59096692)

[Angular testing 33](#_Toc59096693)

[Hoe test je angular? 34](#_Toc59096694)

[Hoe schrijf je testen binnen spec.ts file? 35](#_Toc59096695)

[TestBed 35](#_Toc59096696)

[Imports 36](#_Toc59096697)

[Declarations 36](#_Toc59096698)

[Provides 36](#_Toc59096699)

[IT 36](#_Toc59096700)

[Html en Ts testen 36](#_Toc59096701)

[Wat zijn End-to-end tests? 38](#_Toc59096702)

[Protractor 38](#_Toc59096703)

[Wat is Protractor? 38](#_Toc59096704)

[Waarom zou ik Protractor gebruiken? 38](#_Toc59096705)

[Hoe moet je Protractor installeren? 39](#_Toc59096706)

[41](#_Toc59096707)

[Hoe schrijf je tests met Protractor? 42](#_Toc59096708)

[Intergration tests (Activemq) 44](#_Toc59096709)

[Hoe test je activeMq? 44](#_Toc59096710)

[Opzet 44](#_Toc59096711)

[Conclusie 46](#_Toc59096712)

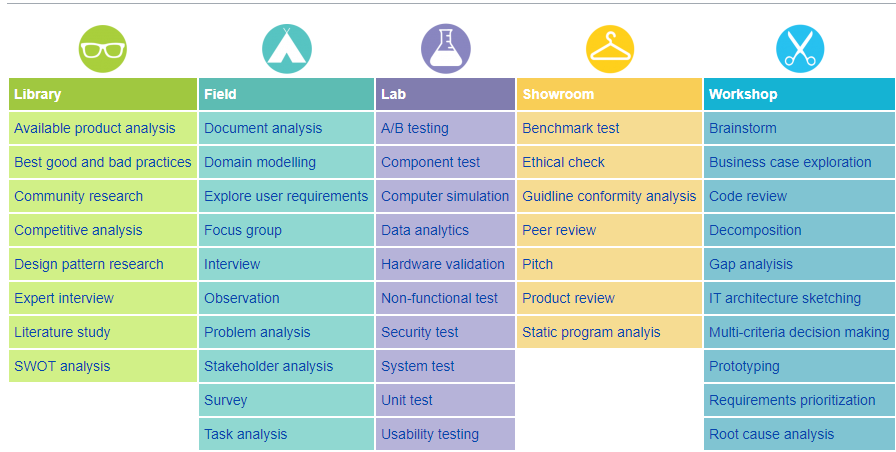
[NaBespreking 46](#_Toc59096713)

# ResearchMethodes

Binnen ons project is het verstandig om bepaald onderzoek te doen naar verschillende aspecten die misschien gevoelig kunnen zijn. Hierbij kun je denken aan een stuk security en het afweren van aanvallers of aan het schalen van je project als er meerdere gebruikers komen.

Om dit onderzoek van start te laten gaan hebben we eerst een onderzoeksvraag nodig over het onderwerp dat we willen behandelen. Nadat we de onderzoeksvraag hebben gesteld hebben we nog een hypothese nodig om ons uiteindelijke antwoord mee te vergelijken en dus te kijken of we gelijk hadden en of we meer te weten zijn gekomen.

Na deze stappen kunnen we gebruik maken van de verschillende researchmethodes om zo meer te weten te komen over ons project en om dichter bij het antwoord op de vraag te komen.



(bron 1)

Hierboven staan de methodes beschreven. Elke soort heeft zijn eigen voor en nadelen maar om de vraag juist te kunnen beantwoorden is het handig als we een aantal uit verschillende categorieën pakken zodat we meer vlakken raken op het gebied van onze research.

Nadat al het onderzoek voltooid is kun je de onderzoeksvraag beantwoorden en dus een conclusie trekken. Aan de hand van deze conclusie ben je hopelijk beter geworden rondom het onderwerp en weet je meer hierover.

# Onderzoeksvragen:

1. **Wat zijn onze grootste valkuilen qua security binnen ons project?**
2. **Hoe Test je onze microservicesstructuur effectief?**

## Hypotheses

1. Wij denken dat de grootste security valkuilen liggen bij het niet hebben van de authorization tokens dus blijven gebruikers altijd ingelogd. Het andere grote probleem is het hebben van Sensetive data exposure, wij maken veel gebruik van api’s en omdat we dit niet hebben kan iedereen bij deze api’s.
2. Wij denken dat een effectieve manier van testen is vooral de UI tests in Angular gebruiken en de api’s los te testen. Dit zijn in onze ogen de belangrijkste onderdelen om onze volledige architectuur te kunnen testen.

# Wat zijn onze grootste valkuilen qua security binnen ons project?

## Inleiding

Onze applicatie draait met een microservices architectuur, dit zorgt ervoor dat er al snel valkuilen zitten binnen de applicatie zelf. Wij maken namelijk een game waar currency een belangrijk doel van het spel is. Om deze reden is het al snel gevoelig voor cheaters en hackers die snel extra currency op hun account willen zetten. Hierdoor moeten wij onze applicatie volledig beschermen tegen dit soort acties.

## Security

Er zijn ontzettend veel manieren om je applicatie te beveiligen en dit maakt het zo moeilijk voor ons omdat wij de juiste vormen van de applicatie moeten gaan vinden. Uiteraard willen we de focus leggen op het beveiligen van de frontend omdat hier de gebruikers naartoe zullen komen. Maar ook het beveiligen van de backend en voornamelijk de currency en de wachtwoorden is van groot belang. De currency wordt gekocht met echt geld of langzaam verdient en lokt dus al snel hackers naar zich toe. Om wat meer research te gaan doen naar deze security methodes maken wij gebruik van de OWASP lijst waarin verschillende security risks staan beschreven.

## Owasp

### Injection

#### Wat:

Injectie fouten zoals SQL injectie vinden plaats wanneer er geen data maar een commando als data wordt verstuurd, zo kunnen aanvallers een hoop data achterhalen.

#### Hoe:

##### Input Validation:

Kijken en verifiëren of de input van de gebruiker wel mag volgens het programma, hierbij moet je bijvoorbeeld denken aan whitelisten van data.

##### Parameterized queries:

Hiermee kun je het verschil zien tussen input van een gebruiker en een script of commando, hierdoor kun je dus voorkomen dat er überhaupt een script geactiveerd kan worden op de applicatie.

##### Stored procedures

Met stored procedures groepeert het programma in een bepaald plan van executie, je hoeft dan niet iedere keer een bepaalde query aan te maken maar haalt dan een voor gegenereerde op.

### Broken Authentication

#### Wat:

Applicaties met authenticatie worden vaak verkeerd geimplementeerd waardoor aanvallers mogelijk session keys, wachtwoorden en gebruikers kunnen achterhalen.

#### Hoe:

##### Multi factor authentication

Dit is waarbij de applicatie een andere applicatie vraagt om de gegevens nog een keer te controleren, hierdoor voorkom je makkelijk brute force attacks tegen gaan

##### Hashing

Hier hashen wij het wachtwoord waardoor deze niet makkelijk te decrypten is.

##### Login timeout

Als de gebruiker x aantal keer fout heeft ingelogd moet er een timeout komen zodat niemand een brute force attack zou kunnen gebruiken

##### Strong password check

Het controleren of een wachtwoord sterk genoeg is voorkomt het gebruik van rainbow lists tegen de applicatie.

##### Session keys

Genereert willekeurige session id’s per gebruiker om te detecteren wanneer er ergens anders is ingelogd.

### Sensetive data exposure

#### Wat:

Veel api’s verbergen belangrijke data niet zoals financiele gegevens, aanvallers veranderen deze data of stelen het om dingen zoals fraude te kunnen plegen.

#### Hoe:

##### Verschillende wachtwoorden

Het gebruik van verschillende wachtwoorden per account kan ervoor zorgen dat als er een wachtwoord gelekt wordt dat een aanvaller niet toegang heeft tot alles.

##### Https

Gebruik maken van een https verbinding zorgt ervoor dat we veilig data versturen en dat een aanvaller dit niet zomaar kan vangen.

##### GateWay

Een gateway geeft een extra laag security bovenop onze applicatie die ervoor zorgt dat niet iedereen zomaar alle apis kan aanroepen.

### Xml external entities

#### Wat:

Vooral oude XML processors evalueren alles in xml documenten waardoor aanvallers deze kunnen toesturen om zo aanvallen binnen in het systeem te krijgen.

#### Hoe:

##### Uitschakelen Parsing libraries

Het uitschakelen van de functies die in de parsing xml libraries zitten zorgt ervoor dat xml niet meer blootgesteld wordt aan potentiele aanvallers.

##### Json

Het gebruik van minder complexe gegevensindelingen kan er al voor zorgen dat xml niet blootgesteld wordt.

##### White listing

Het gebruik van een whitelist invoervalidatie betekend dat alleen gevalideerde gebruikers(of admins) xml bestanden aan kunnen passen en niet zomaar willekeurige gebruikers.

### Broken acces control

#### Wat:

Restricties van wat een gebruiker wel en niet mag doen worden vaak slecht gecontroleerd, hierdoor kan iemand die niet geautoriseerd is makkelijk op plekken komen waar hij niet hoort te zijn en of kunnen komen.

#### Hoe:

##### Toegang

Om broken access te kunnen voorkomen is het weigeren van toegang door het zetten van ‘private’ erg gemakkelijk

##### Gebruik controleLijsten

Het gebruik van controle en rollenlijsten zorgt ervoor dat niet elke gebruiker dezelfde rechten heeft, een admin kan bijvoorbeeld op een pagina waar een gebruiker niet op kan

##### Verbergen

Het verbergen van functies en variabelen net zoals sommige modellen en controllers kan er al voor zorgen dat een aanvaller niet zomaar in een applicatie kan kijken.

### Security Misconfiguration

#### Wat:

Een verkeerde configuratie van de beveiliging is het meest voorkomende probleem. Dit is meestal het gevolg van onveilige standaardconfiguraties, verkeerd geconfigureerde HTTP-headers en uitgebreide foutmeldingen met gevoelige informatie.

#### Hoe:

##### DebugginG uitschakelen

Het uitschakelen van debugging zorgt ervoor dat een aanvaller niet binnen de code kan kijken.

##### Standaard accounts en wachtwoorden uitschakelen

Het uitschakelen van testwachtwoorden zorgt ervoor dat niet iedereen kan inloggen als een admin.

##### Server initialiseren

De server initialiseren om alleen ingelogde mensen toe te laten werkt tegen aanvallers die op een server proberen te komen

### Cross-site Scripting xss

#### Wat:

Met XSS kunnen aanvallers scripts uitvoeren in de browser van het slachtoffer die gebruikerssessies kunnen over nemen, websites kunnen beschadigen of de gebruiker kunnen omleiden naar kwaadwillende sites.

#### Hoe:

##### Hoe vaak komt XSS vulnerabilities voor?

XSS-kwetsbaarheden komen veel voor, en XSS is waarschijnlijk de meest voorkomende kwetsbaarheid op het gebied van webbeveiliging.

##### Hoe vaak komt XSS attacks voor?

Het is moeilijk om betrouwbare gegevens te krijgen over echte XSS-aanvallen, maar er wordt waarschijnlijk minder vaak misbruik van gemaakt dan andere kwetsbaarheden.

##### Wat is het verschil tussen XSS and CSRF?

XSS houdt in dat een website kwaadaardig JavaScript retourneert, terwijl CSRF een slachtoffergebruiker ertoe aanzet om acties uit te voeren.

##### Wat is het verschil tussen XSS and SQL injection?

XSS is een kwetsbaarheid aan de clientzijde die zich richt op andere toepassingsgebruikers, terwijl SQL-injectie een kwetsbaarheid aan de serverzijde.

##### Hoe voorkom je XSS in PHP?

Filter uw invoer met een white list met toegestane tekens en gebruik typetips of typecasting. Ontsnap aan uw uitvoer met htmlentitiesen ENT\_QUOTES voor HTML-frameworks, of JavaScript Unicode-escapes voor JavaScript-frameworks.

##### Hoe voorkom je XSS in Java?

Filter uw invoer met een witte lijst met toegestane tekens en gebruik een bibliotheek zoals Google Guava om uw uitvoer HTML-coderen voor HTML-contexten, of gebruik JavaScript Unicode-escapes voor JavaScript-contexten.

### Insecure deserialization

#### Wat:

Onveilige deserialisatie leidt vaak tot uitvoering van externe code. Dit kan worden gebruikt om aanvallen uit te voeren, waaronder injection -aanvallen en aanvallen met privilege escalation aanvallen.

#### Hoe:

###### HOE VOORKOM JE iNSECURE DESERIALIZATION:

1. Het implementeren van integriteitscontroles zoals digitale handtekeningen op alle geserialiseerde objecten om het knoeien met gegevens te voorkomen.
2. Het isoleren en uitvoeren van code die indien mogelijk deserialiseert in omgevingen met weinig bevoegdheden.
3. Het registreren van uitzonderingen en fouten bij deserialisatie, zoals wanneer het inkomende type niet het verwachte type is, of de deserialisatie genereert uitzonderingen.
4. Inkomende en uitgaande netwerkverbindingen van containers of servers die deserialiseren, beperken of bewaken.
5. Monitoring van deserialisatie, waarschuwen als een gebruiker constant deserialiseert.

### Using components with known vulnerabilities

#### Wat:

Componenten, zoals libraries, frameworks en andere software modules, worden uitgevoerd met dezelfde rechten als de applicatie.  Als een kwetsbare component wordt misbruikt, kan een dergelijke aanval leiden tot ernstig gegevensverlies of serverovername.

#### Hoe:

##### Voorkomen insuddicient logging

###### HANDMATIGE UPDATES

Een optie is om geen componenten te gebruiken die u niet hebt geschreven. De meeste libraries en frameworks creëren geen kwetsbaarheidspatches voor oude versies. In plaats daarvan lost u het probleem eenvoudig op in de volgende versie. Upgraden naar deze nieuwe versies is dus een manier om dit te voorkomen.

###### GEBRUIK HDIV

Detectietools voor kwetsbare software van HDIV bevorderen een simpelere aanpak, waarbij softwareafhankelijkheid wordt geanalyseerd, zowel tijdens de bouwtijd als tijdens runtime, om gemakkelijk kwetsbare stukjes software te detecteren die moeten worden vervangen door nieuwere versies. De tools zijn ontworpen om de hele levenscyclus van applicaties te dekken:

### Insufficient logging and monitoring

### Wat:

Onvoldoende logging en monitoring, in combinatie met ontbrekende of oneffectief integration met incident response, stelt aanvallers in staat om systemen verder aan te vallen.

### Hoe:

###### Voorkomen van insuddicient logging and monitoring

1. Zorg ervoor dat alle aanmeldingsfouten, fouten bij toegangscontrole en validatiefouten aan de serverzijde kunnen worden gelogd met voldoende gebruikerscontext om verdachte accounts te identificeren.
2. Zorg ervoor dat logboeken worden gegenereerd in een indeling die gemakkelijk kan worden gebruikt door gecentraliseerde oplossingen voor logboekbeheer.
3. Zorg ervoor dat transacties met een hoge waarde een audit trail hebben met integriteitscontroles om manipulatie of verwijdering te voorkomen.
4. Zorg voor effectieve monitoring en alarmering zodat verdachte activiteiten tijdig worden gedetecteerd en beantwoord.

# Security test

Het eerste en waarschijnlijk ook het belangrijkste onderdeel van het testen van de security zelf, hierbij kunnen we hopelijk gemakkelijk te zien krijgen wat er wel en niet getest is en tegelijkertijd een beeld te schetsen van security problemen in onze applicatie.

## Sanitization and security

### XSS Sanitization loading in a htmlsnippet

cross-site scripting ook wel XSS genoemd zijn injectie aanvallen, waarbij kwaad willige scripts worden geïnjecteerd in het slachtoffer zijn website. Gelukkig heeft angular build in Sanitization and security, Die voorkomt dat scripts binnen de HTML geinjecteert kunnen worden.

Dit is een demo project waarbij ik een XSS Attack simuleer.

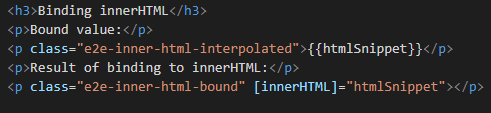
Dit is de ExampleXSSAttackComponent.ts file. Hier in laad ik html snippet in met een Script tag.



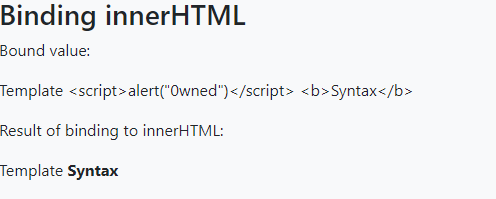
In dit voorbeeld als het niet word gestopt word er alleen maar een Alart melding in de console getoond. Maar in een echte situatie zou het een kwaad willig script kunnen zijn.



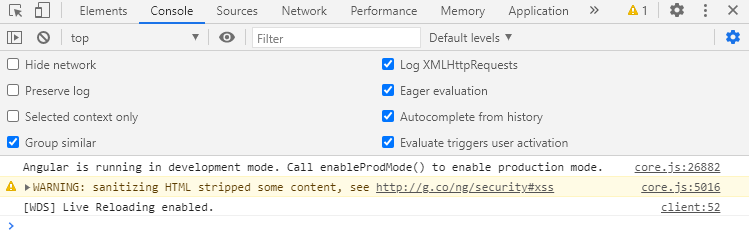
Dit laad ik vervolgens in de Html via een innerHTML.



Maar zo als het resultaat laat zien word dit door angular gezien en word dit niet uit gevoerd.



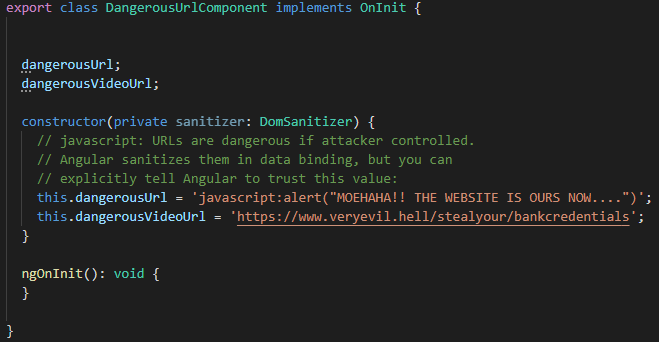
De console laat zelf zien dat de sanitizing de script tag verwijderd heeft.



### XSS Sanitization Url loading

Angular Biedt ook bescherming tegen het inladen en erheen routen van mogelijke gevaarlijke URL’s. in sommige applicaties moeten uitvoerbare code bevatten bijvoorbeeld een <iframe>van een bepaalde URL weergeven.

Dit is een demo project waarbij ik een aanvaller simuleer die de user naar een andere URL wil laten routen. Dit project bevat 2 URLS waarvan een URL die schadelijke code bevat die word uitgevoerd door het klikken van een knop. En een die een video toont van een onbekende URL.



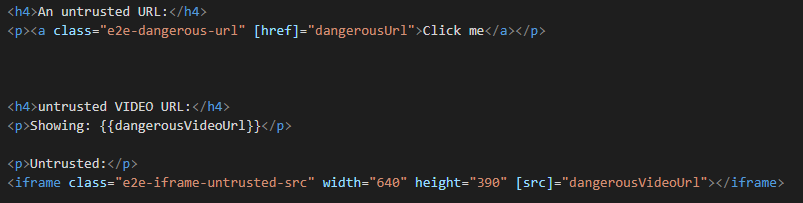
Deze URL voert een kwaadwillige script uit.



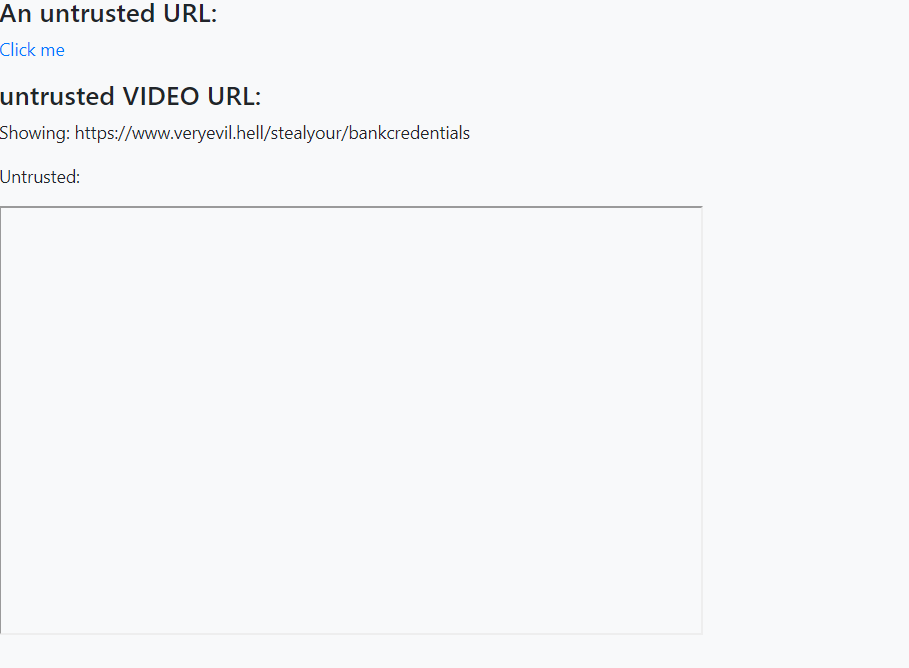
Deze URL kan de user mogelijk naar een kwaadwillige website leiden.



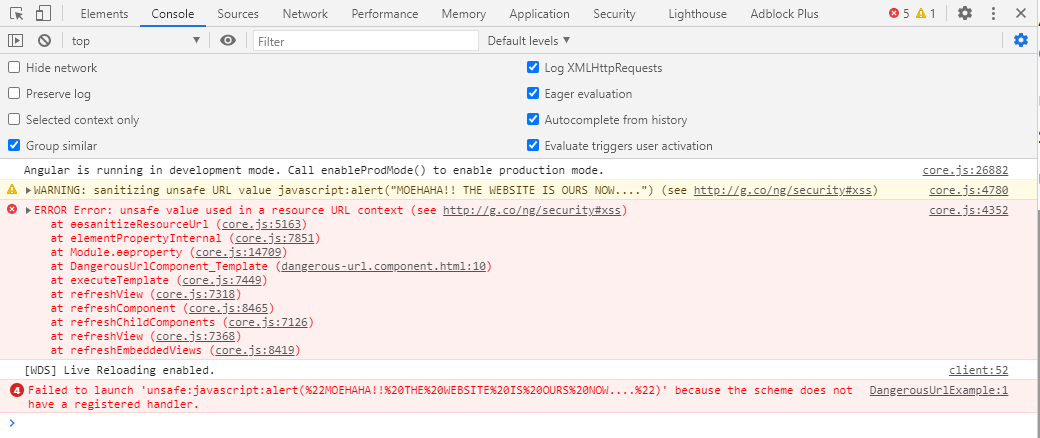
Dit is de HTML waar de URL’s worden ingeladen via een href en een src.



Angular toned de website.



En de kwaad willige URL’s worden geblokkeerd.



## conclusie

Angular heeft een build in beveiliging tegen XXS attacks, wat bij het implementeren van een webapplicatie erg handig is. Na dat ik meerdere test had uit gevoerd om het systeem te omzeilen blokkeerde Angular me nog steeds. In het kort ik ben blij dat we als front-end framework Angular gekozen hebben.

# Best Good and bad practices

Om zelf een beter beeld te krijgen met wat goed en slecht is qua security willen we een good and bad practices maken. Hierbij willen we graag laten zien wat er goed en slecht is voor het implementeren van beveiliging om zo ook zelf na te gaan denken wat er fout is bij security.

Voor de good en bad practices willen we graag gaan kijken binnen ons eigen project, we willen graag dieper duiken in de componenten die we slecht en goed vinden qua beveiliging zodat we een beter beeld krijgen bij de qualiteit van ons project.

## Good

### Gateway

Het gebruiken van een gateway tussen de front-end en de back-end heeft veel voordelen.

Het bied bescherming tegen overmatig gebruik en misbruik. Het bied ook als een hub voor authenticatie en beveilig checks. Ook Analyseren-en monitoring tools kunnen makkelijker worden toegevoegd. Het is vooral een erg goede optie als je met een microservice architectuur werkt. Wanneer microservices vervallen of worden vervangen zullen de users de services nog steeds op dezelfde plek kunnen vinden.

Voor ons project gebruiken wij de C# library Ocelot, de reden is dat Ocelot veel mogelijkheden biedt op het uitbreiding van onze app. Het toevoegen van authenticatie en het verbinden met bijv. een discovery service (Consul)is erg makkelijk.

### activemq

Eigenlijk is het hebben van een messagebroker al een good practice in zichzelf, een messagebroker zorgt er namelijk voor dat de berichten die verstuurd zijn altijd aan komen, zelfs als een api of service halverwegen stopt. Dit zou dramatisch kunnen zijn als de broker er niet tussen zat omdat er dan berichten verloren zouden kunnen gaan.

#### Security

ActiveMq zorgt direct niet zelf voor security, maar als je deze goed op hebt gezet kan het gemakkelijk ook voor security zorgen. We zorgen ervoor dat alle data via Dto’s gaat dus als iemand er iets in zou sturen wordt dit eruit gefilterd door active mq zelf, hier wordt dus niks mee gedaan. Ook om requests te versturen heb je een wachtwoord nodig van activemq zelf, dit kun je niet zomaar aanroepen omdat we alleen de requests manager buiten de firewall hebben staan en niet de User interface. Dit bij elkaar zorgt voor een veilige omgeving voor api’s om requests naar elkaar te versturen.

### mongodb

De integratie van mongodb heeft ons ook al een stap in de juiste richting gezet qua security. Door deze repository compleet onzichtbaar te maken voor de gebruiker kan er niemand ook misbruik maken hiervan.

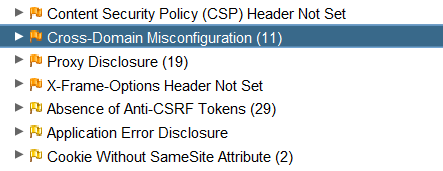
Mongodb zelf werkt met een url die je in moet voegen om de connectie te maken, mensen die deze url niet weten zullen ook nooit weten waar de data vandaan komt en hebben heir dus weinig aan.

In tegenstelling tot sql databases die gevoelig zijn voor sql injection is Mongodb dit niet of in ieder geval een stuk minder. Ook omdat we de requests via Http doen en controleren met Dto’s zal er ook nooit zomaar een commando door onze api’s heen komen en zorgen voor onnodige problemen.

## Bad

### Cors

Een van de ‘bad practices’ die we nog in ons project hebben zitten is onze cors policy. Aan het begin van ons project hebben we veel moeite gehad met deze ingebouwde beveiliging en hebben deze dus uitgeschakeld. Hierdoor is wel het probleem veroorzaakt dat de applicatie en de api’s minder goed zijn beveiligd. In Owasp zap kun je ook met een active scan zien dat hierbij problemen kunnen komen kijken.



Hier kun je al zien dat op onze frontend alleen al 11 problemen zijn ontstaan binnen het cors probleem. Het is dus duidelijk verstandig om deze functie beter erin te zetten omdat we nu iedereen toelaten, als we dit beveiligen met een eigen sleutel per api of door alleen bepaalde ip’s toe te laten in de applicatie zal de applicatie hier beter tegen zijn beveiligd.

### Consul

Opzich is consul een prima discovery service en er zit ook een hele hoop in. Je hebt health checks en kan altijd de url zien van api die het nodig heeft. Hier komt automatisch ook het probleem kijken. Omdat de discovery service dusdanig gebouwd is dat de requests via dezelfde port gaan als de User interface kun je (als je de Url weet) gemakkelijk alle url’s van de microservices achterhalen.

Het probleem ligt dan met name bij het feit dat de gateway dit ook gebruikt dus consul er nog voor ligt. Hierdoor krijg je dus de echte links naar de microservices en niet die van de gateway.

Hiernaast hebben we niet ver genoeg in Consul gedoken en weten we niet hoe we dit moeten beveiligen. Er staat ook op internet dat het opzich niet nodig is om een gebruikersnaam en wachtwoord te hebben voor deze service en dat dit ook veel moeite kost.

Om deze redenen is Consul niet heel veilig in zichzelf en is het niet handig om zo’n service maar alf te integreren zoals het bij ons project nu in elkaar zit.

### Authentication token

Authorization token bieden ontzettend veel voordelen binnen de applicatie op het gebied van beveiliging. Deze tokens zijn ‘decoupled’, dit houdt in dat het zowel op de user staat als in de backend waardoor er een extra laag beveiliging buiten het inloggen zit.

Een ander voordeel is dat deze tokens bescherming bieden tegen mogelijkke XSS en XSRF aanvallen die makkelijk gevaarlijke scripts injecteren door bugs in de beveiliging van de webapplicatie.

Momenteel hebben wij de authorization tokens nog niet geimplementeerd maar gebruiken wij alleen de id van de gebruiker. Hierdoor is de applicatie minder beveiligd en kunnen aanvallers direct toegang krijgen tot een stukje data van de backend. Dit is gevaarlijk en vandaar dat wij dit ook als bad-practice hebben beschreven binnen ons project.

# Code review

Als laatste willen we gaan kijken naar onze api’s en daarover een review maken. De beste manier om dit te kunnen doen is via sonarqube. Sonarqube is in het kort gezegd een static code analysis tool. Deze laat de code coverage, code smells en de security risks zien en hoewel we alleen de security nodig hebben is dit alsnog een uitstekende tool voor het maken van een code review op security basis.

## Owasp Zap

Ons eerste plan was om sonarqube te gaan gebruiken voor de code review, dit is een uitstekende keuze voor bugs en de basis controle, maar dit is niet handig voor als we willen gaan testen binnen een applicatie. Om dit wel te kunnen doen hebben we even gezocht naar een alternatief. Hier kwamen we bij de volgende video uit:

<https://www.youtube.com/watch?v=8fdh7CntpmE> . hierin werd beschreven waarom en hoe je Owasp kunt gebruiken, hier hebben we dus verder ingedoken omdat deze applicatie(net als sonarqube), ook een test rapport uitschrijft, deze kun je zelfs in je CiCd pipeline verwerken voor een geautomatiseerde test.

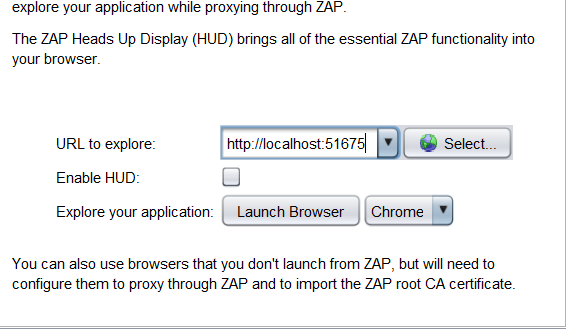
## Voordelen Zap

We hebben nog niet naar concurrentie gezocht voor een code review tool in het security opzicht maar zap kwam bij veel mensen als nr. 1 omdat deze veel ingebouwde tools heeft en het testrapport makkelijk te converteren naar een excel bestand.

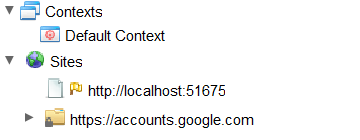
## Gebruik van Zap

### Passive mode

In passive mode ga je zelf door de applicatie heen en kijkt zap of er eventuele breach punten zijn in je applicatie, dit werkt doormiddel van een ‘spider’. Deze spider gaat elke url die jij in de gegenereerde browser invult controleren. Owasp valt de applicatie hierbij niet aan en kijkt hij alleen maar of er gevoeligheden zijn.



Op zap vullen we eerst de url in van de website die we willen testen, in dit geval is dat localhost omdat we onze applicatie lokaal aan het draaien zijn. Maar deze site kan net zo goed online staan.



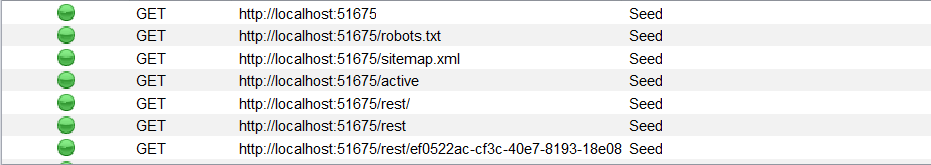
Op het moment dat we deze browser openen komt er links in de applicatie een file bij te staan van de site waar we naartoe zijn gegaan, in ons geval staat er direct een gele vlag bij. Dit houdt in dat er een minimale breach is ontstaan, zonder dat ik iets heb gedaan.

Als we meer willen weten over onze api en waar deze fout op gaat kunnen we de ‘spider’ gebruiken binnen owasp.

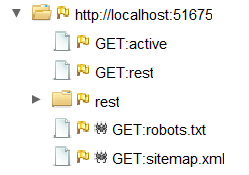
### Spider

Een spider in Owasp wordt gebruikt om nieuwe url’s te achterhalen op een website / api. Deze vind de spider voornamelijk in hyperlinks of als de gebruiker naar een andere url gaat.

Op het moment dat de spider een hyperlink vind kan deze automatisch weer gescand worden door owasp zelf en krijgen we daar weer de breaches weergegeven.



Hierboven staat een sequence weergegeven van acties die ik zelf heb uitgevoerd, ik heb eerst **/rest** gedaan om alles op te halen en vervolgens **/{id}** erbij gezet om een specifiek dataobject aan te vragen. Het groene rondje geeft aan dat de spider daar op de site heeft kunnen komen.



Als je vervolgens weer naar het explorer scherm kijkt zie je dat er 2 onderdelen bij zijn gekomen van de spider (spin). Robot.txt en sitemap.xml.

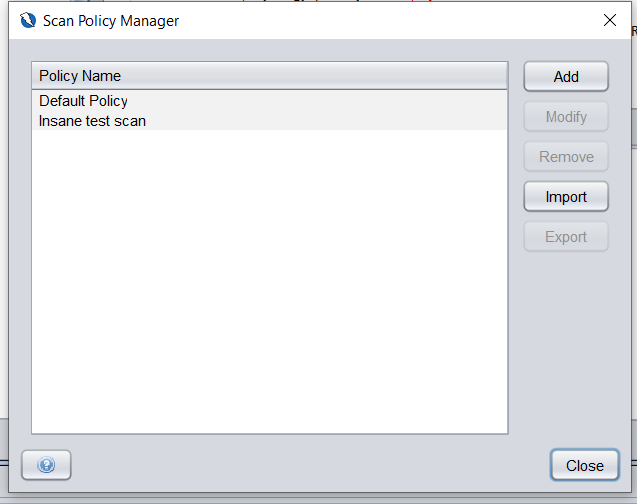
De **Robot.txt** is het configuratie gedeelte van de spider, hierin wordt aangegeven wat de spider allemaal wilt ophalen.

De **Sitemap.xml** is eigenlijk de plek waar alle mappings van de sites worden opgeslagen (de acties die ondernomen moeten worden).

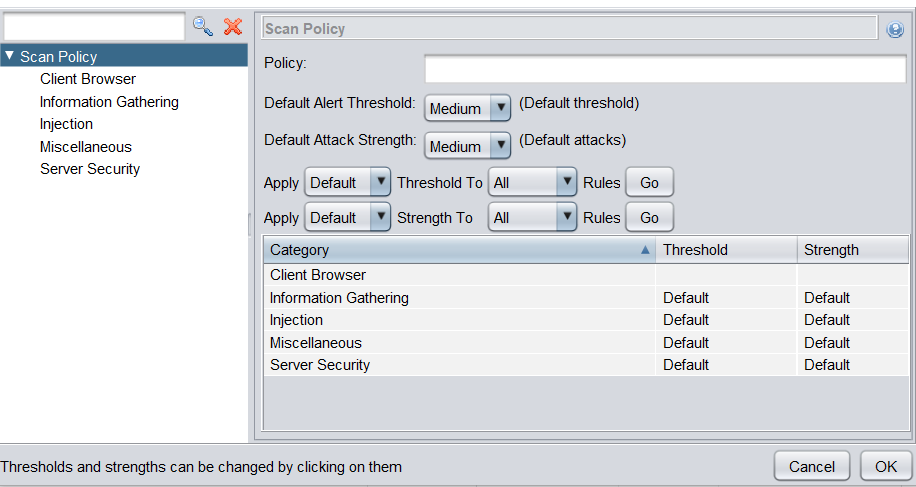
### Active Mode

Een active scan valt de webapplicatie echt aan, hierbij gaat owasp kijken hoe de applicatie misbruikt zou kunnen worden met fouten die in de database zelf staan. Hierbij kun je ook je eigen scripts toevoegen om zo een eigen scan te maken. Maar je kunt ook de voor gegenereerde gebruiken.

Als je wilt scannen moet je eerst een nieuwe policy aanmaken, dit doe je door naar Analyse in de toolbar te gaan en vervolgens op policy manager te klikken.



Hierin kun je tests aanpassen, importeren en aanmaken.



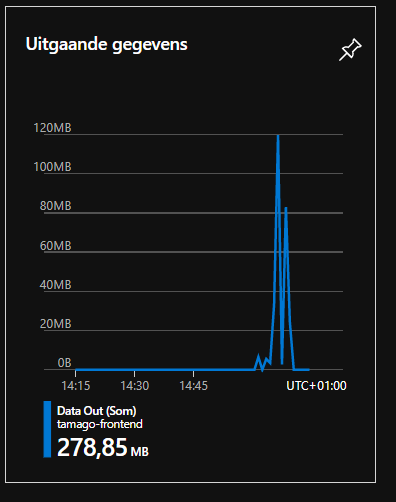
Als je er een wilt toevoegen krijg je het volgende scherm, hierin kun je aangeven welke sterkte van aanval je wilt en op wel aspect van de app je het wilt draaien.

Als je eenmaal de policy hebt gemaakt en een spider op de passive scan hebt laten werken heb je als het goed is al een lijst met de verschillende urls waar Owasp naartoe kan gaan. Als je vervolgens op active scan drukt via het attack menu kun je daar je policy selecteren en de active scan zijn gang laten gaan. (dit duurt een tijdje).

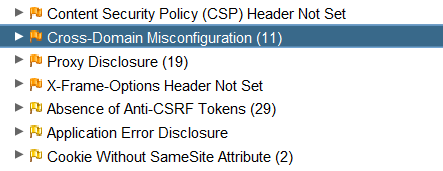
### Owasp resultaat



Een scan van onze eigen frontend resulteerde in 9223 requests die waren verstuurd en ook 72 nieuwe alerts.



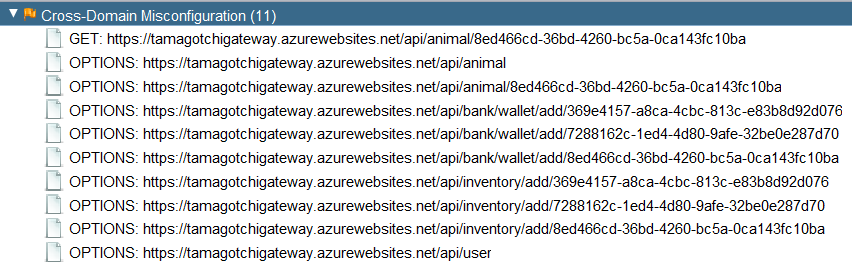
Ook zijn de uitgaande gegevens van onze azure app gigantisch omhoog gegaan door deze scan.



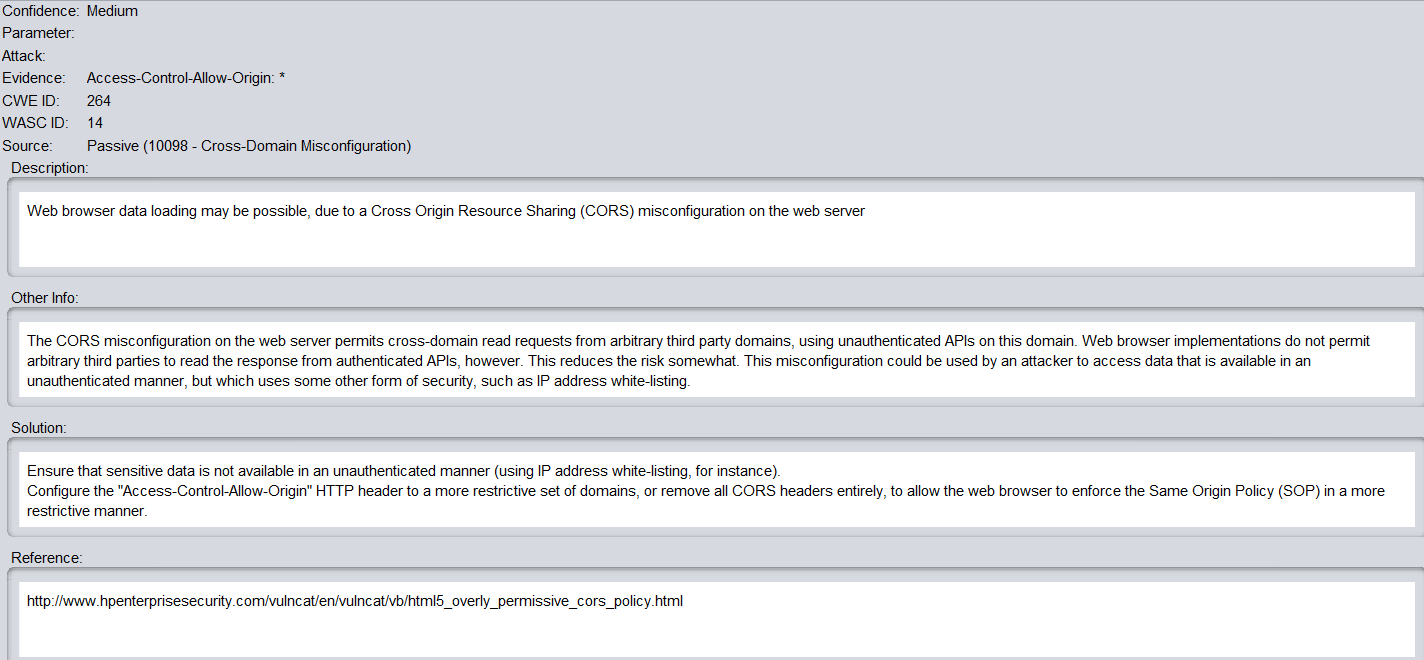
Bij onze alerts zitten veel dezelfde problemen zoals de proxy ‘Disclosure’ en de ‘Cross-domain misconfiguration’. Om deze te kunnen verhelpen kunnen we dieper in de problemen duiken.

#### Resultaat

Bij de alerts staan gelukkig ook waar ze vandaan komen, wat het inhoud en hoe je het zou kunnen verhelpen.



Bij ons cross domain alert kunnen we zien waar die allemaal in is voorgekomen, je kunt dus zien dat er veel requests hetzelfde probleem hebben.



Verder zien we zoals hierboven wat het probleem is, hoe groot het probleem is en hoe je het kan verhelpen. Dit probleem ligt met name aan CORS dus is voor ons nu niet van grote waarde omdat we dit risico al wisten, maar als je als bedrijf hier nog niet aan had gedacht kan dit van groot belang zijn.

## Conclusie

Owasp zap is een uitstekende keus als het gaat om code reviews ten opzichte van de security. Owasp zap schrijft een rapport uit waardoor wij gemakkelijk kunnen beoordelen of de applicatie goed genoeg is voor deployment of niet, voor ons heeft dit al een hele hoop geholpen met het vinden van security fouten en is gemakkelijk te gebruiken.

# Conclusie

We hebben veel onderzoek gedaan naar security en zijn achter een helehoop handige tools gevonden en deze ook kunnen implementeren om de beveiliging van een applicatie te verbeteren, maar wat is nou de beste manier van beveiligen?

Het belangrijkste van het beveiligen is dat je de kleine componenten niet vergeet, in ons project zijn dat voornamelijk de authentication tokens, de Cors policy en de Consul discovery service die momenteel zonder wachtwoord te bereiken is. Via Owasp zap zijn we erachter gekomen dan Cors een groter probleem is dan we dachten en ook vaker een probleem is. Ook via owasp kwamen we achter de authentication tokens die dus ook een probleem waren.

Het is dus voornamelijk belangrijk om te blijven testen en externe tools te gebruiken om achter deze kleine fouten te komen.

# Nabespreking

Het onderzoek ging erg goed, we kwamen achter een hoop tools en ingebouwd testmateriaal wat we voorheen nog niet wisten. Dit heeft ons niet alleen met het onderzoek geholpen maar vooral met de beveiliging van het project.

We hebben het onderzoek synchroon gedaan met het implementeren van de tools binnen het project waardoor we ook gelijk een stukje uit ervaring spraken en hierdoor de keuzes ook beter konden onderbouwen.

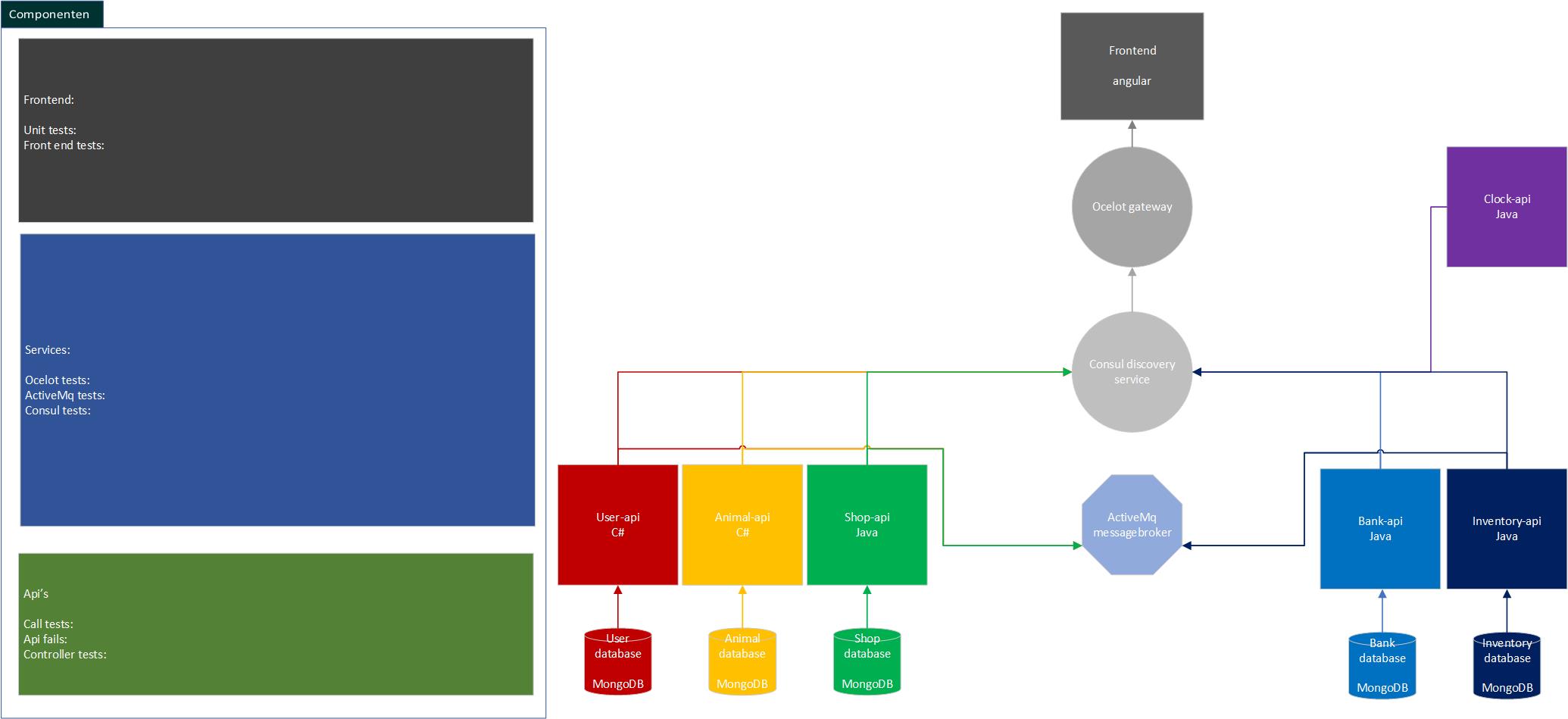
.

# Hoe test je een microservice architectuur effectief?

## Inleiding

Wij waren van plan om als onderzoek eens de tests onder de loep te nemen, er zijn zo veel verschillende manieren van testen dat we eigenlijk wel eens wilde kijken wat er allemaal gedaan moest worden wil je de volle applicatie getest krijgen. We besloten om hiervoor apart naar onze eigen onderdelen te kijken en daarvoor de juiste tests voor aan te maken.

Omdat wij een microservices architectuur hebben is het extra belangrijk om de applicatie ook in zijn geheel te kunnen testen, vandaar dat wij deze onderzoeksvraag hebben gesteld en hier in dit document het onderwerp gaan onderzoeken.



Onze architectuur ziet er ongeveer zo uit, we hebben een aantal microservices die met elkaar praten via een messagebroker (in dit geval ActiveMq). Ook praten die via een discoveryservice en gateway met de frontend. Met deze structuur kunnen we sowiesow goed gebruik maken van alle tools die we geimplementeerd hebben, maar moeten we dus wel meer testen.

## Testen van een applicatie

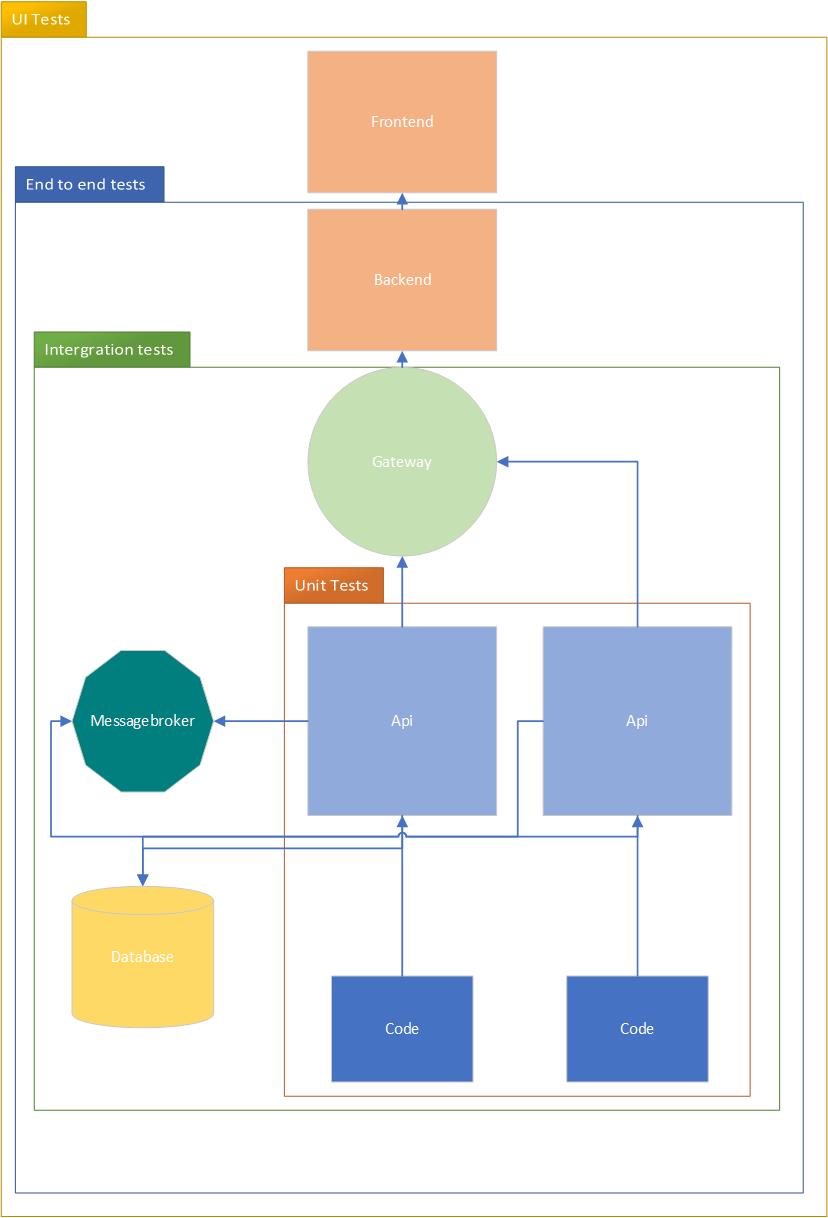
Omdat we zoveel verschillende onderdelen hebben in ons project moeten we ook veel verschillende manieren hebben om te testen. Als je alleen een stukje code hebt, heb je ook alleen maar een unit test nodig. Wij hebben daarintegen ook connecties en html elementen die we moeten testen zowel als messagebrokers en gateways, desgelijks moeten we ook weten wat er gebeurt als er een api niet zijn werk doet of crasht. Daarom is het essentieel dat we elk component testen. Wij hebben om deze reden onderzoek gedaan naar de verschillende soorten tests die er zijn en hoe je die integreert.

# TestStructuur

Voor ons eerste dot net onderdeel willen we graag een visualisatie brengen aan de verschillende soorten tests die er zijn, vanwege het hoge aantal tests wat we moeten doen is het ook belangrijk om prioriteiten te kunnen stellen zodat we efficient te werk kunnen gaan, hiervoor zijn we begonnen met het maken van een schets met daarin alle vormen van tests die we willen gebruiken en voor welke onderdelen van ons project dat relevant is.

# 

(bron2)

In het plaatje hierboven zie je de structuur van een ’microservices architectuur’, Dit werkt goed voor de applicatie die boven beschreven wordt maar wij hebben een andere architectuur dus hebben we hiervoor ons eigen plaatje gemaakt.

Om deze reden hebben we onze eigen visualisatie gemaakt, Hierin staan dezelfde kleur gecodeerde tests in alleen hebben wij ook de onze messagebroker en de gateway toegevoegd, hierdoor hebben we ook voor ons project een beter beeld.

## UI tests

De eerste laag houdt de UI tests in, deze tests zijn eigenlijk om alle html en js (in ons geval ts) componenten te testen, hier testen we wat er gebeurt als er op een knop wordt gedrukt en of de juiste functie dan wel aangeroepen wordt.

## End to end tests

Als je je UI test hebt geschreven weet je al wat elk html component gaat doen maar je weet niet wat er gebeurt als een gebruiker daadwerkelijk je applicatie gaat gebruiken, daarvoor gebruik je End-to-end tests.

Deze tests doen eigenlijk een gebruiker na om te zien of alles werkt zoals jij dit wilt. Op deze manier kun je gemakkelijk de gehele frontend testen. En ook een beetje de communicatie tussen front en backend.

## Intergration tests

De Intergration tests testen het verloop tussen alle van een applicatie. Deze tests zijn essentieel voor onze applicatie omdat we zo niet alleen alle verschillende api’s moeten gaan testen maar ook de services die we ertussen gebruiken. Een messagebroker wordt ook op deze manier getest.

## Unit tests

Als we van de front naar backend kijken zijn we nu aangebroken bij de laatste specifieke test, de Unit test. Bij deze test ga je kijken naar de verschillende functies binnen bijvoorbeeld een api controller en kijk je of deze functies wel goed werken. Bij deze test pak je alle api’s los van elkaar om zo Unit tests uit te voeren.

## Smoke tests

Nu zijn we aangebroken bij de laatste 2 test soorten, deze zijn universeel en zorgen ervoor dat de volledige applicatie goed in elkaar gaat steken. De eerste daarvan is de smoke test

De smoke test is een test die de connectie tussen het platform en de applicatie testen, een applicatie of api kan anders regeren als deze op een azure platform is gehost ipv lokaal, dit is vooral handig bij onze java api’s als we dezen willen hosten op azure.

## Performance testing

Al deze tests zijn goed in hun ding alleen hebben ze allemaal een grote fout, ze testen maar een ding tegelijk. Hierbij speelt performance testing een grote rol.

Performance testing kijkt naar de applicatie en gaat deze eigenlijk proberen te overbelasten, op deze manier komen we erachter of onze applicatie daar goed tegen kan en ook wat er gebeurt mocht een applicatie falen.

# Interview

Om ons 2e gedeelte van het Dot framework hebben we gekozen om iemand te interviewen. Kees Kotterink is erg bekend met software omdat hij al lang actief is bij verschillende bedrijven zoals atos origin en Asml. We hebben hem gekozen om te interviewen omdat we denken dat we door andermans ervaringen een stuk dichter bij de conclusie kunnen komen en hierdoor ook meer te weten komen over het testen in het algemeen.

Voor het interview hebben we 12 vragen opgesteld met daarin alle informatie die wij denken nodig te hebben, deze vragen gaan onder andere over zijn eigen ervaring en wat zijn standpunten zijn in de wereld van het testen.

## Vragen

### 1. Heb je ooit gewerkt met het testen van een microservices architectuur?

Ja, met oracle heb ik gewerkt met een soortgelijke architectuur.

### 2. Welke soort tests vind je het belangrijkst bij het testen?

End to end omdat ik weinig in ui tests geloof maar wel in de koppeling tussen front en backend.

### 3. Bij welke tests begin je meestal?

Unit test, hierbij ga je eerst nadenken wat de applicatie moet doen en krijg je het proces beter gemaakt.

### 4. kom je vaak fouten tegen na het testen?

Unit testen falen vooral als de code geupdate is maar de unit test niet, hierbij vraag ik af of de unit test wel effectief is.

### 5. Hoe breidt je je tests uit na mate je applicatie groter wordt?

Eerst de microservice unit testen en vervolgens zo stapsgewijs naar boven werken in de test heragie

### 6. Maak je een plan voordat je gaat testen?

Ja, wij maken altijd een plan voordat we beginnen met testen.

### 7. Probeer je altijd alles te testen of zijn er componenten die niet belangrijk genoeg zijn.

Nee, ik test alle elementen in een geheel maar vind de losse services zoals de gateway niet relevant genoeg om te testen.

### 8. Wanneer trek je de conclusie dat je applicatie genoeg getest is?

Vooral met code coverage tools, hiermee kun je inspecteren wat ik allemaal getest heb en door te kijken naar de requirements om zo te zien of we alle elementen gehad hebben.

### 9. Hoe belangrijk is de benaming als je je tests schrijft?

Super belangrijk, Je hebt in deze structuur heel veel verschillende tests en je moet deze op een manier geordend houden.

### 10. Waar testen jullie voornamelijk jullie applicatie mee?

De meeste frameworks die wij hebben bij Asml maken wij zelf dus de testing tools worden ook zelf gemaakt om deze reden.

### 11. Wat vind je van jullie manier van testen?

We werken veelsteveel met zelfgemaakte software en frameworks en om die reden denk ik dat we onze applicaties te veel maken voor die zelfgemaakte software.

### 12. verlopen de tests altijd volgens plan?

Eigenlijk nooit, We zijn eigenlijk altijd te optimistisch, maar hierdoor halen we wel snel de fouten eruit.

### 13. eigen toevoeging.

Ik vind dat er teveel focus op testen is en er vervolgens over andere componenten heen gekeken wordt, hierdoor zie je bijvoorbeeld bepaalde algoritmes over het hoofd.

## Conclusie

Volgens het interview met kees trekken we de conclusie dat Unit tests en UI tests een beetje buiten de boot vallen omdat deze vaak minder van belang zijn binnen een groot project, dit komt omdat Unit tests niet gemakkelijk uitbreidbaar zijn en UI tests overschaduwd worden door end to end tests. Daarom willen wij de focus leggen op End to end en Intergration tests.

# Document Analysis

Voor het laatste onderdeel willen we zelf een stuk dieper duiken in de testen die we niet kennen, wij hebben voornamelijk alleen maar met unit tests gewerkt en kunnen om die reden zelf weinig zeggen over wat er juist is voor ons project. Om dit goed onder de knie te krijgen zijn we zelf maar aan de slag gegaan.

## Angular testing

Als eerste hebben we gekeken naar de UI en End to end tests, Dit is essentieel om ons angular framework te kunnen testen en hiermee krijgen we ook een beter beeld op de tools die we ervoor nodig hebben.

1. Jasmine

Karma is een tool die een webserver broncode uitvoert tegen. De resultaten van elke test worden onderzocht en weergegeven, zodat deze kan zien welke browsers en tests zijn geslaagd of mislukt.



1. Karma.

Jasmine is een zeer populaire JavaScript framework voor het testen van JavaScript-applicaties. Het biedt hulpprogramma's die kunnen worden gebruikt om geautomatiseerde tests uit te voeren voor zowel synchrone als asynchrone code.



## Hoe test je angular?

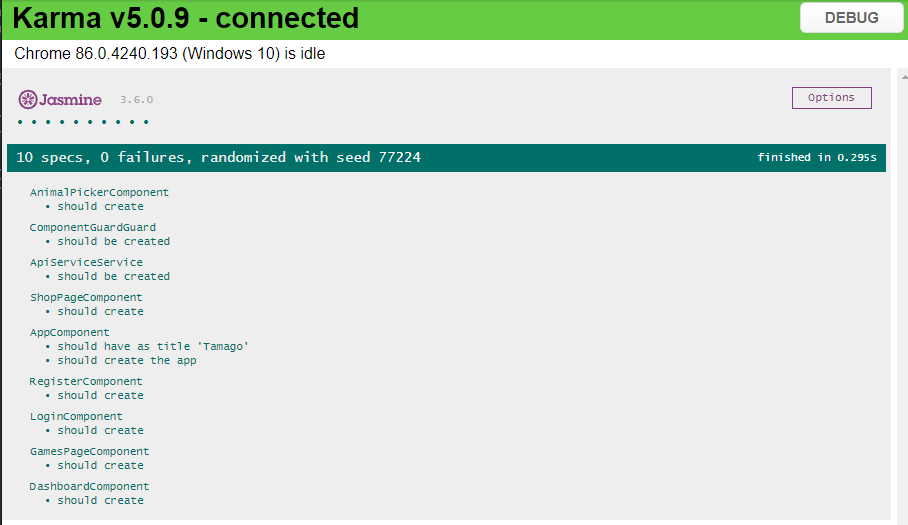
Binnen het “spec.ts” file kunnen onderdelen van de component worden getest.



Via de commando “ng test” worden allen “spec.ts” uit gevoerd.



Karma het Javascript test framework die samen met Jasmine werkt binnen Angular geeft een informatie venster weer waarbij je makkelijk kunt zien welke testen wel en niet zijn geslaagd.



### Hoe schrijf je testen binnen spec.ts file?

Bij het genereren van een component word er automatisch een test file aangemaakt. Binnen die test file staat de basis al opgesteld. En als je het commando “ng test” zou runnen in terminal, zou de test ook slagen. Maar als je bij je component andere classes in de constructor laad en niet import en opzet in de test file kan dit leiden tot een error.



### TestBed

Binnen het test zijn 3 onderdelen: imports , Declarations en providers. Deze onderdelen maken samen de test file op.

### Imports

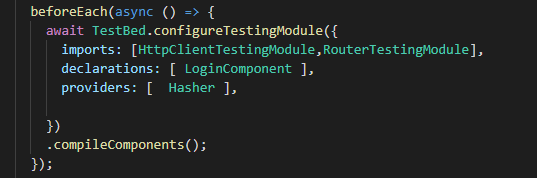
In de imports plaats je de mock modules die zijn gemaakt door angular zelf. (Bijv. Router , HttpClient)

### Declarations

Binnen de Declarations plaats je de module die je wilt testen.

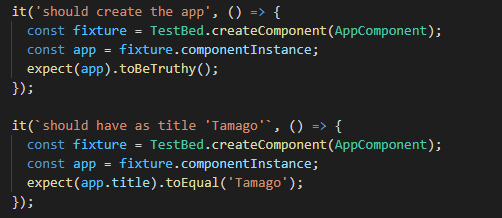
### Provides

De provides is voor classes in te laden die worden gebruikt door de module. Dit geld meestal voor eigen gemaakte classes.



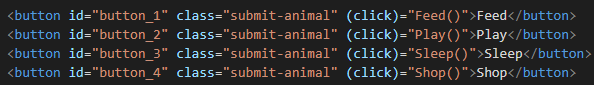
### IT

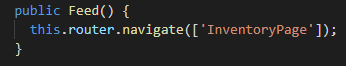
De IT functie is de plek waar de tests daadwerkelijk worden uitgevoerd. In dit voorbeeld word de getest of het component aangemaakt kan worden. Het is ook mogelijk om functies en variabele in de Ts file te lezen(bijv: bij app.title toEqual tamga).



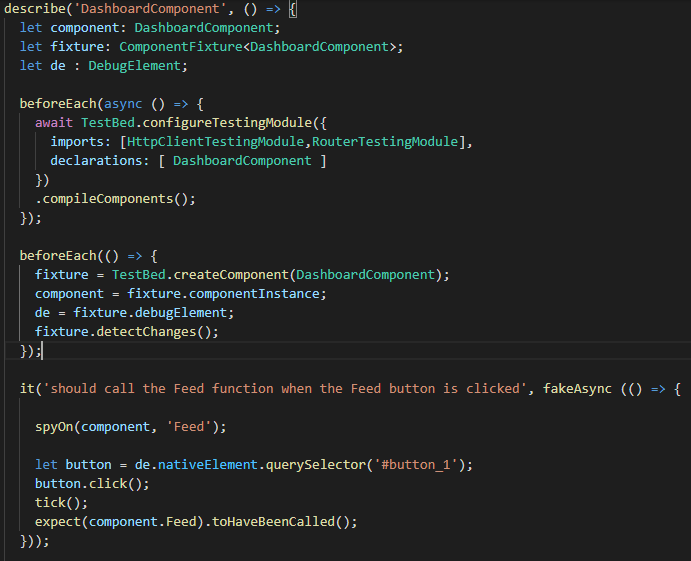
### Html en Ts testen

Sommige onderdelen van de html code roepen functies aan binnen de Ts file. Bijv. deze button met een id van “button\_1”, deze button roept de functie Feed aan in de Ts file.





Om dit te kunnen testen moet er de functie “Async”worden aangeroepen. In Karma word dit gedaan door middel van “fakeAsync” die dit simuleert. Via “de” (DebugElement) kunnen we het button element selecteren en clicken. De “tick” functie wacht een aantal miliseconde zo dat de functie aangeroepen kan worden, en vervolgens testen we het resultaar met de “expect(component.feed)” en de “toHaveBeenCalled” functie.



# Wat zijn End-to-end tests?

End-to-end testen is een test techniek die wordt gebruikt om de stroom van een applicatie te testen zodat die zich van begin tot eind gedraagt ​​zoals verwacht. Het doel van het uitvoeren van end-to-end-tests is om systeemafhankelijkheden te aantonen en ervoor te zorgen dat verschillende systeemcomponenten en systemen goed behouden blijven.

# Protractor

## Wat is Protractor?

Protractor is een end-to-end test framework voor AngularJS-applicaties. Het framework combineert krachtige tools en technologieën zoals NodeJS, Selenium, webDriver, Jasmine, Cucumber en Mocha.

## Waarom zou ik Protractor gebruiken?

Waarom Protractor AngularJS?

1. Het is een wrapper rond Selenium WebDriverJS en Selenium Server.
2. Locator strategieën en -functies specifiek voor Angular-apps.
3. Kan het voordeel van het seleniumraster benutten om meerdere browsers tegelijk uit te voeren.
4. Kan tests uitvoeren op zowel echte als headless-browsers.
5. Kan Jasmine of Mocha of Cucumber gebruiken om test mee te schrijven.

Voordelen van Protractor

1. Eenvoudige installatie en configuratie.
2. Gemakkelijk leesbaar Jasmin framework.
3. Ondersteuning van Data-Driven tests.
4. Bevat allen voordelen van Selenium WebDriver.
5. Automatische synchronisatie.
6. Ondersteunt parallelle tests via meerdere browsers.
7. Uitstekende snelheid.

## Hoe moet je Protractor installeren?

Het installeren van Protractor binnen je Angular project is gelukkig erg simpel, er zijn 3 commando’s nodig die je moet installeren via de terminal namelijk:

1. **npm install -g protractor**

Hiermee worden command line tools geïnstalleerd, Protractor en webdriver-manager. Probeer Protractor -versie uit te voeren om te controleren of het werkt.



1. **webdriver-manager update**

de webdriver-manager is een hulpmiddel om gemakkelijk een instantie van een Selenium Server te laten draaien.

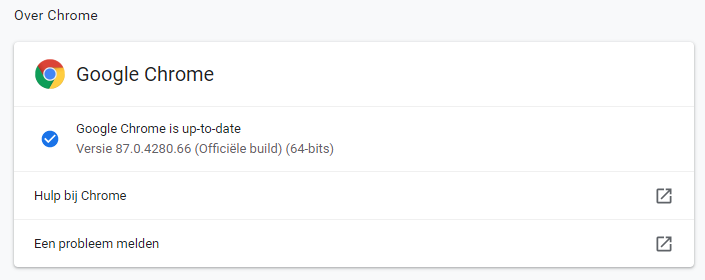


1. **webdriver-manager start**

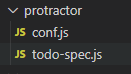
test vervolgens de webdriver- manager met deze commando.



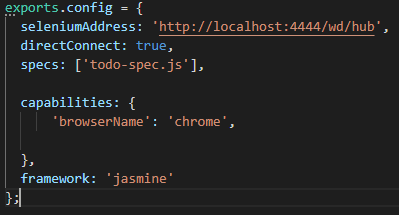
Als er een error ontstaat tussen het instaleren kan het zijn dat je versie van je webbrowser niet up-to-date is. In ons project is dit bijv. Chrome.



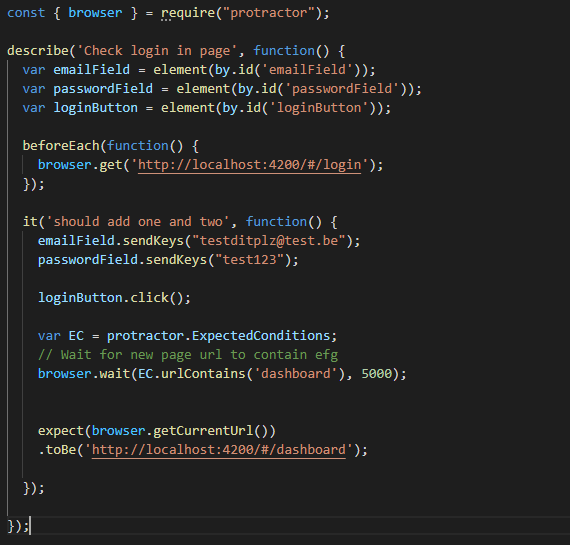
Als alles eenmaal geïnstalleerd is moeten er 2 JS files worden aangemaakt namelijk een Conf.js en een specs.js. Voor wat meer overzicht en netheid heb ik dit in een mapje geplaats genaamd protractor.



In de Conf.js file staat de data die Protractor nodig heeft om de spec.js files te kunnen testen. Als je meerdere spec.js files wilt testen kun je die hier toevoegen binnen de array specs.



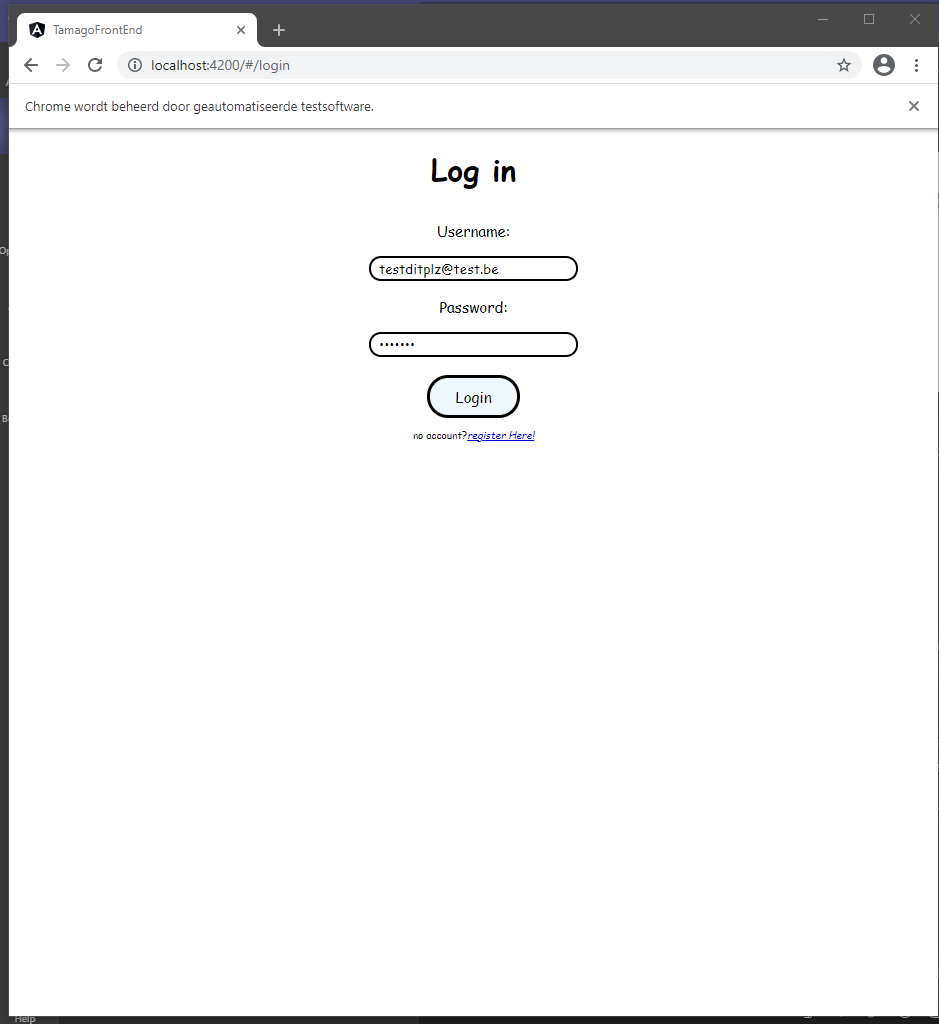
De spec.js files bevat Allen end-to-end tests. In dit voorbeeld test ik de inlog pagina.



De tests kun je vervolgens uitvoeren door het commando **protractor conf.js** uit te voeren. Binnen ons project heb ik de files binnen een mapje genaamd protractor gezet dus het commando word dan **protractor protractor/conf.js**.



Protractor set een Selenium Server op en voer daar de geschreven tests uit.



Nadat de tests zijn afgerond word het resultaat de gedisplayd in de terminal.

## 

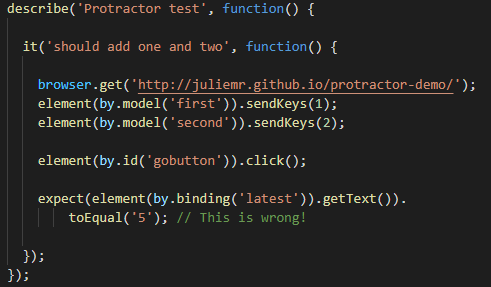
## Hoe schrijf je tests met Protractor?

In dit voorbeeld word een calculator getest. Om interactie met de verschilende elementen te kunnen hebben moet Protractor ze eerst kunnen vinden. Dit kan door middel deze verschilden functies:

* by.model('first')om het element te vinden met ng-model="first". Als u de bron van de Rekenmachine-pagina bekijkt, ziet u dat dit het geval is <input type="text" ng-model="first">.
* by.id('gobutton')om het element met de opgegeven id te vinden.

Dit vindt <button id="gobutton">.

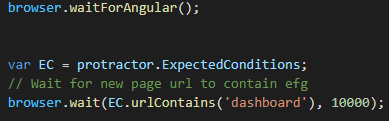
* by.binding('latest')om het element te vinden dat aan de variabele is gebonden latest. Dit vindt de span met{{latest}}



Elementen die acties uitvoeren zo als buttons kunnen de events laten triggeren door bijv. de Click() functie te gebruiken. Dit is erg handig om user interacties te simuleren.



Protactor bied ook opties voor operaties die wat langer duren. De Browser.wait functie geeft je de opties om een aantal milliseconde te wachten voor de het resultaat van de test gecheckt word.



Op het einde van de end-to-end test kan het resultaat worden gecheckt. Dit word gedaan via de expect functie, hier onder word getest of het resultaat over een komt met de string. Als dit waar is, is de test geslaagd.



Voor deze test geld het omgekeerde. De test is pas geslaagd wanneer de waar niet overeenkomt met de string.



## Intergration tests (Activemq)

Een ander onderdeel van onze test analyse was het testen van de messagebroker. Hiermee moeten onze api’s kunnen praten onderling en mocht een request falen is dit de manier om de message op te vangen, daarom is het belangrijk dat dit onderdeel getest wordt.

## Hoe test je activeMq?

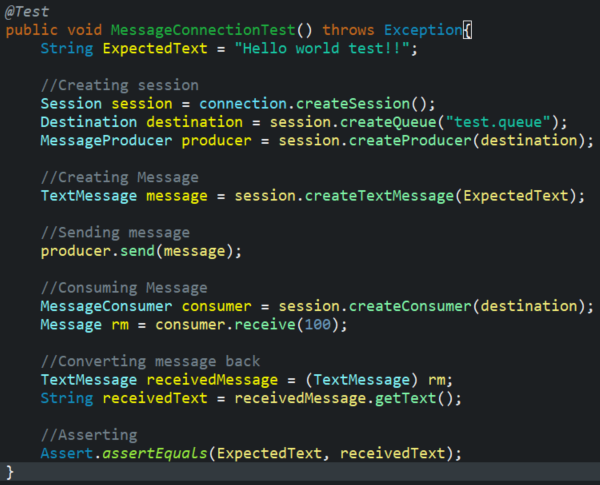
Om ActiveMq te kunnen testen maak je eigenlijk gebruik van intergration tests, Om activemq op deze manier te testen is soortgelijk aan Unit testen omdat het alleen een spring boot applicatie gebruikt, verder zijn er wel een aantal verschillen.

### Opzet

Het eerste onderdeel van de tests is de opzet, hierbij creëren we een connectie zodat we die erna kunnen testen. Dit is het belangrijkste gedeelte en ook al een test in zichzelf omdat hier de connectie tussen de api en de messagebroker getest wordt.



Nadat je de connectie hebt opgezet en afbreekt wanneer een test klaar is kun je de test zelf maken, hierbij is het belangrijk om het converteren niet te vergeten. Andes faalt de test sowieso.



Hierboven hebben wij een van de connectie tests gemaakt, Eerst maken we eerst een String waarmee we dalijk het eindresultaat gaan vergelijken

Vervolgens creeren we een session een destination en een producer, de sessie zorgt ervoor dat we dalijk alles kunnen versturen, de bestemming bepaalt de plek waar het naartoe gestuurd moet worden en de producer kan er uiteindelijk een bericht naar verzenden.

Daarna gebruiken we de sessie om een bericht te schrijven met de tekst die we als eerste hebben bepaalt en versturen we die naar de messagequeue via de producer.

Als we dit hebben gedaan staan de berichten als het goed is al in de queue, om deze te kunnen lezen creeren we een consumer. Deze leest dalijk alle berichten uit de gekozen bestemming.

Als we deze gemaakt hebben luistert deze net zolang totdat hij een bericht heeft ontvangen.

Als het bericht is ontvangen zetten we deze terug om naar een string en vergelijken we die met elkaar.

# Conclusie

Dus? Wat is het handigste om te testen met een microservices Architectuur?

Het is vooral de combinatie tussen alle verschillende soorten tests en dit is dus ook waar we mee bezig gaan zijn. Als we teveel focussen kost het teveel tijd om voor ons alles te gaan testen. Om deze reden is het belangrijk dat we elke soort test op een aantal manieren uit gaan voeren zodat alles in elkaar kan klikken.

Wij gaan met deze informatie vooral focussen op de algemene testen. We willen voornamelijk veel verschillende en niet perse super veel code coverage aan het begin. Hierdoor hebben we sneller een loop door het project getest en kunnen we dus vanuit de gebruiker kijken wat er eventueel fout zou kunnen gaan.

De testmethodes die voor ons project het meest relevant zijn zijn voornamelijk; De ingebouwde ui tests van angular(Jasmin & Karma), de End to end tests om de connectie tussen front en backend goed te testen(protractor). Dit is ook een angular package, De Ingebouwde Unit tests voor de api’s van visual studio(c#), Als laatste de Integration tests om de connectie tussen onze externe services en de api’s te testen(JUnit).

# NaBespreking

De aanpak van een researchdocument was voor ons uiteraard nieuw en om die reden zitten er ongetwijfeld verkeerde keuzes in of kleine foutjes. Terugkijkend hadden we waarschijnlijk een andere soort aanpak moeten nemen op het project. Nu hebben we voornamelijk op dit document gefocust maar we hadden voor de volgende keer meer op de code en het testen zelf kunnen focussen, hierdoor zou het document wel langzamer gaan maar kunnen we vervolgens wel meer uit ervaring spreken in plaats van te focussen op het internet en te kijken hoe anderen het doen. Alle code snippets zijn wel uit ons eigen project dus er is niks gekopieerd wat relevant is.