This paper is also available in English

Het lijken twee uitersten. 'Technisch ontwerp' is iets voor de oude rotten in het IT-vak, die nog denken in termen van watervalmethodes zoals SDM. 'AngularJS' is een speeltje van Web-nerds, die snel een leuke app in elkaar zetten. Denk je nou echt dat die gaan wachten op ontwerpdocumentatie? Dit artikel legt uit dat technisch ontwerp voor professionele webapplicaties belangrijk is en biedt concrete voorbeelden van hoe zo'n ontwerp er uit zou kunnen zien indien het AngularJS-framework wordt gebruikt. Ik concentreer me hierbij op de vorm en ga niet in op design patterns of ontwerpbeslissingen.

Technisch ontwerp in UML voor AngularJS-applicaties

en andere webapplicaties

Enkele begrippen

Om misverstanden te voorkomen geef ik eerst een definitie. In dit artikel bedoel ik met een *web-applicatie* een applicatie die door een webbrowser wordt uitgevoerd. Het kan dus bijna niet anders of zo'n applicatie is grotendeels geschreven in HTML, CSS en JavaScript. Zo'n applicatie is meestal slechts een onderdeel van een oplossing, waarbij de rest op één of meer servers draait. Met deze definitie wijk ik bewust af van definities zoals Wikipedia die geeft, waarin ook de back-end software tot de webapplicatie wordt gerekend, omdat ik dat nodig heb voor dit artikel. Bovendien is de afbakening duidelijker. Veel webapplicaties maken namelijk gebruik van meerdere APIs, ook APIs van externe organisaties (bijvoorbeeld Google) en met de Wikipedia-definitie is het onduidelijk welke software er nu precies tot de webapplicatie behoort.

Een AngularJS-applicatie is een webapplicatie die gebaseerd is op het AngularJS framework, een door Google ontwikkeld open-source framework.

UML staat voor Unified Modeling Language, een standaardnotatie voor het modelleren van software, die beheerd wordt door de Object Management Group. UML definieert allerlei begrippen en de grafische weergave daarvan, bijvoorbeeld een class wordt weergegeven als een rechthoek.

Om dit artikel te begrijpen, moet je enige basiskennis hebben van AngularJS en van UML.

Technisch ontwerp in een agile omgeving

Het zou vanzelfsprekend moeten zijn, maar voor de zekerheid stellen we de vraag toch maar: Waarom zou je een technisch ontwerp maken voor een webapplicatie? De meeste webapplicaties worden tegenwoordig ontwikkeld door teams, die werken volgens een agile methode, zoals Scrum of Kanban. Deze methodes doen geen uitspraak over het al dan niet produceren van ontwerpdocumentatie, waardoor de indruk kan ontstaan dat zulke documentatie niet nodig is. Bovendien zegt het Agile Manifesto: “We value working software over comprehensive documentation”. Dat hetzelfde manifest daarbij wel zegt, dat ook documentatie waardevol is, wordt soms vergeten. Goede ontwikkelaars kunnen applicaties van hoge kwaliteit bouwen zonder technisch ontwerp, maar toch pleit ik ervoor, om professionele applicaties altijd tot op een zeker niveau te voorzien van een technisch ontwerp, overigens met behoud van iteratief werken en prototyping. Waarom? Omdat het tijd en geld bespaart en de gebruikers een beter werkende applicatie krijgen! Hierbij geldt wel de voorwaarde, dat de ontwikkelaars er achter staan. De genoemde besparingen vloeien voort uit het feit dat een ontwerp inzicht geeft in de structuur van de software en daardoor de kans op een slechte structuur (spaghetticode, inconsistenties, code duplicatie) vermindert. Zowel het inzicht in de structuur als de hoge kwaliteit van de structuur verminderen vervolgens...

a. de kans op fouten en dus de tijd die het kost om fouten op te lossen;
b. de tijd die het kost om uit te zoeken hoe een bepaalde wijziging of uitbreiding het beste ingepast kan worden;
c. de tijd die het kost om onvolkomenheden in de structuur later recht te trekken.

Ik zei al: applicaties moeten *tot op een zeker niveau* voorzien zijn van een technisch ontwerp, d.w.z. niet te gedetailleerd. Het moet voldoende zijn om vervolgens, wanneer je meer details wilt weten, deze gemakkelijk te kunnen vinden in de broncode. Als de broncode nog geschreven moet worden, moet het ontwerp voldoende zijn voor de ontwikkelaar om de details verder naar eigen inzicht in te vullen. Overigens is het niet altijd zo, dat het ontwerp eerst af moet zijn voordat er geprogrammeerd mag worden. Je kunt ervoor kiezen om eerst een prototype te maken, dit te evalueren, een ontwerp te maken van de gewenste situatie en vervolgens het prototype te verbeteren conform het ontwerp.

Het werkt het beste om het ontwerp door iemand anders te laten maken dan degene die de code schrijft, waarbij ze wel nauw samenwerken en elkaars werk beïnvloeden. Dit geeft een veel betere kwaliteit dan wanneer ontwikkelaars zelfstandig bepaalde functionaliteit implementeren. Afhankelijk van de individuele vaardigheden en affiniteiten kun je ervoor kiezen om dezelfde personen afwisselend te laten ontwerpen en te laten programmeren.

Veel meer informatie over ontwerpen in een agile omgeving is te vinden op de website www.agilemodeling.com van Scott Ambler.

De context van een webapplicatie

Het maken van een webapplicatie is meestal geen geïsoleerd project, maar onderdeel van een traject, waarin ook een back-end ontwikkeld wordt en misschien wel een hele website. Vanuit deze context komt er allerlei input voor het bouwen van de webapplicatie (gedeeltelijk tijdens het bouwen), bijvoorbeeld:

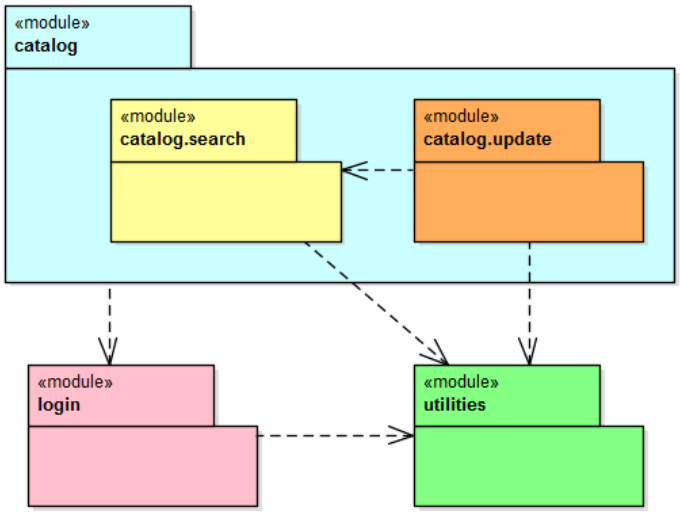
- Modellen van bedrijfsprocessen en –objecten.
- Functionele eisen, meestal in de vorm van user stories of use cases.
- Niet-functionele eisen, variërend van eisen vanuit de business tot een referentie-architectuur en programmeerrichtlijnen.
- User interface design, meestal een prototype of een serie afbeeldingen.
- Specificaties van de back-end API(s) en uiteindelijk de API(s) zelf.
- Documentatie van de JavaScript libraries die gebruikt zullen worden of mogen worden.

Het is niet mijn bedoeling om in dit artikel een compleet stappenplan te geven voor hoe je vanuit deze input komt tot een technisch ontwerp. De bovenstaande lijst wil slechts de context aangeven, zodat tevens duidelijk is wat er niet tot het technisch ontwerp zelf behoort.

Architectuur

Als er binnen de organisatie vaker AngularJS-applicaties worden gebouwd, is er misschien een referentiearchitectuur, die een bepaalde vorm van applicatiearchitectuur voorschrijft, bijvoorbeeld over welke soorten componenten er onderscheiden worden en hoe die zich tot elkaar verhouden. Dan kun je in het technisch ontwerp daarnaar verwijzen en hoef je minder uit te leggen.

Hoe het ook zij, van elke specifieke applicatie zal de architectuur beschreven moeten worden. Net als de rest van het ontwerp is de architectuur een combinatie van tekst en schema's. Bij het maken van schema's volg ik vrijwel altijd de UML-standaard, zodat ik niet hoef uit te leggen wat alle symbolen betekenen. Op het hoogste niveau is een AngularJS-applicatie ingedeeld in modules, dus het ligt voor de hand om het ontwerp te beginnen met een schema waarin de modules staan afgebeeld. Je hoeft de hele architectuur niet in één keer te ontwerpen: naarmate het project vordert, voeg je nieuwe modules toe. Het geëigende UML-symbool voor een AngularJS-module is de package. Voor alle duidelijkheid zou je elke package het stereotype «module» kunnen geven, zoals in Figuur 1.



Figuur 1. Overzicht van AngularJS-modules in een package diagram

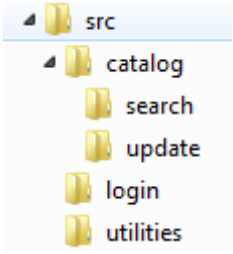
De pijlen, dependency associations, moeten overeenkomen met de afhankelijkheden zoals ze in de source code gedefinieerd zijn of zullen worden:

```
angular.module('catalog.update', ['catalog.search', 'utilities']);
```

Voor geneste packages geldt hetzelfde:

```
angular.module('catalog', ['catalog.search', 'catalog.update', 'login']);
```

De nesting moet overeenkomen met de mappenstructuur van de ontwikkelomgeving:



Figuur 2. Mappen in de ontwikkelomgeving

Je kunt in het package diagram ook de JavaScript libraries en de back-end APIs weergeven als packages, met andere stereotypes uiteraard, en met dependency associations aangeven welke modules gebruik maken van welke libraries/APIs. Als het schema te complex wordt, kun je de informatie verdelen over meerdere schema's, dan wel bepaalde informatie niet grafisch, maar alleen tekstueel vastleggen. Als er bijvoorbeeld tien modules afhankelijk zijn van 'utilities' en het schema onoverzichtelijk wordt door die tien pijlen richting 'utilities', dan kun je die pijlen ook weglaten en dit uitleggen in de begeleidende tekst.

Ik geef elke module altijd een eigen kleur. In de rest van mijn technisch ontwerp krijgen alle componenten dezelfde kleur als de module waartoe ze behoren. Hierdoor zijn de scheidslijnen tussen de modules in elk diagram in één oogopslag te zien.

Een plaatje alleen is niet voldoende. Elke module verdient een korte beschrijving, die duidelijk maakt wat het doel en de verantwoordelijkheden van die module zijn. Verwijzingen naar user stories of delen van het user interface design zijn hierbij nuttig.

De applicatiearchitectuur dient verder in te gaan op aspecten, die gelden voor de applicatie als geheel, zoals:

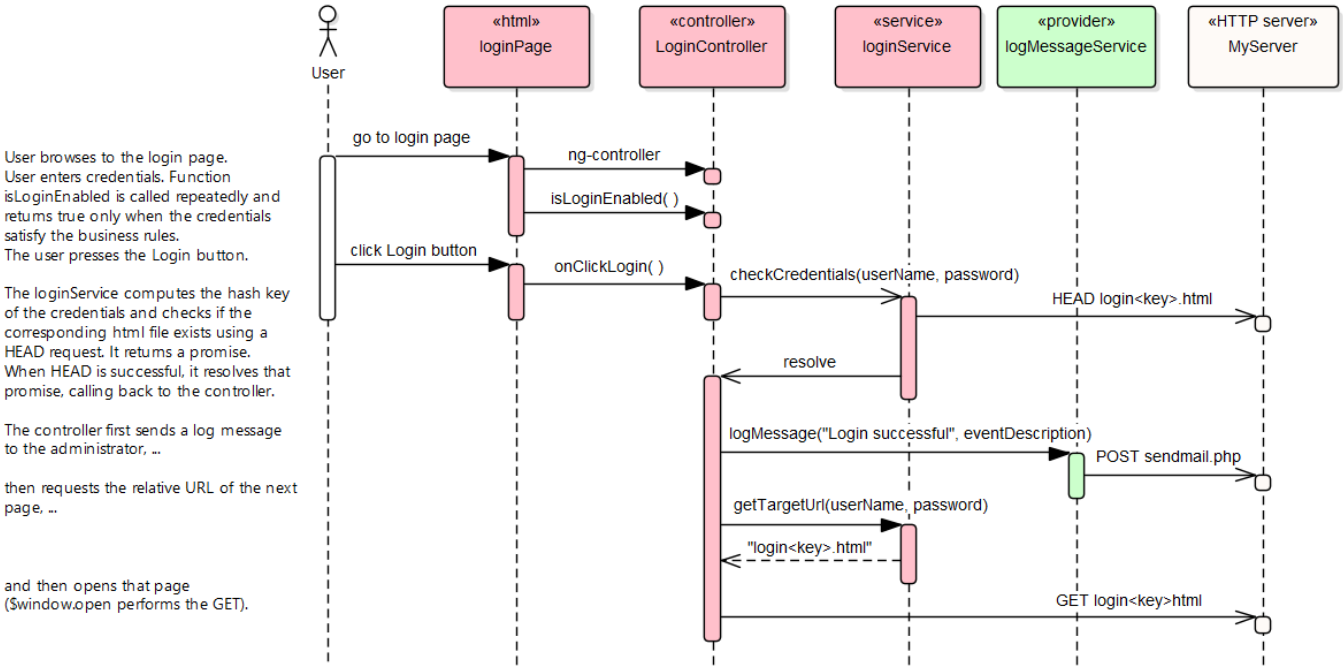
- Een beschrijving van de productieomgeving, inclusief het evt. gebruik van een content delivery network (CDN) en de cachingstrategie (hoe krijgen gebruikers de meest recente versie en wordt toch optimaal gebruik gemaakt van browser cache en CDN cache?).
- Een beschrijving van de ontwikkelomgeving, inclusief unit test voorzieningen.
- Een beknopt overzicht van het build- en deploymentproces. Welke tools (Gulp, Protractor, Jenkins, ...) en welke preprocessors worden er gebruikt (SASS, uglify, ...)?
- Het beleid omtrent inputvalidatie en foutafhandeling.
- Beveiliging. Dit is in de eerste plaats een issue voor de back-end software, maar ook bij het ontwerp van webapplicaties moet goed nagedacht worden over beveiliging.
- Het verzamelen van statistieken omtrent het gebruik en het gedrag van de applicatie ('analytics').
- De wijze van integratie met een content management systeem, indien van toepassing.
- Programmeerrichtlijnen (afzonderlijk voor HTML, CSS, JavaScript en AngularJS).

Alle of veel van deze onderwerpen zijn applicatieoverstijgend en kunnen beter vastgelegd worden in een referentiearchitectuur, ten behoeve van de consistentie tussen de diverse webapplicaties binnen de organisatie.

Scenario's

Nu we de modulestructuur in kaart hebben gebracht, is het verleidelijk om top-down door te gaan en de interne structuur van elke module te tekenen. Daarover straks meer, maar eerst kies ik een scenario-gebaseerde benadering. De volgorde van de acties die in een AngularJS-applicatie worden uitgevoerd, is vaak niet zo gemakkelijk af te leiden uit de code. Dit komt omdat er veel met events en callbacks gewerkt wordt. Om dit soort scenario's inzichtelijk te maken, kent UML de zogenoemde sequence diagrams. Figuur 3 is hiervan een voorbeeld. De applicatiecomponenten die in het scenario geraakt worden, zijn als gekleurde rechthoeken afgebeeld, met daarin de naam van de component en met als stereotype het componenttype. De kleur van een component geeft aan tot welke module hij hoort. Zo zie je in één oogopslag de verdeling in modules in het schema terugkomen.

Het voorbeeld betreft een kleine loginapplicatie. De gebruiker moet zijn naam en wachtwoord invullen, op de Login button klikken en dan opent de applicatie een bepaalde HTML-pagina met de informatie of de applicatie waar die gebruiker voor geautoriseerd is. De bestandsnaam van de



Figuur 3. Sequence diagram

HTML pagina is login<key>.html, waarbij <key> een hashgetal is, dat uit de naam en het wachtwoord wordt afgeleid. Bij een succesvolle loginpoging stuurt de applicatie ook nog een mailtje naar een bepaald emailadres, dat dient als een soort logboek van wie er allemaal heeft ingelogd. Het scenario dat weergegeven is in Figuur 3 is dat van een succesvolle login.

Behalve dat dit soort modellen een goed inzicht geven in de werking van de applicatie, zijn ze ook gemakkelijk te relateren aan de user stories of de use cases, waardoor er een goede binding is tussen functioneel en technisch ontwerp.

Lifelines

Het is erg belangrijk om vooraf goede keuzes te maken over wat je precies laat zien in een sequence diagram. Als je teveel details erin opneemt, ben je bijna aan het programmeren. Het kost veel tijd en het is ondoenlijk om het schema up-to-date te houden. Laten we eerst bepalen welke lifelines (want zo heten die verticale banen in een sequence diagram officieel) we in het schema opnemen en welke niet. Figuur 3 toont, van links naar rechts, eerst de gebruiker als een lifeline, daarna de applicatiecomponenten die een rol spelen in het scenario en helemaal rechts de server. Wat zijn dan precies de applicatiecomponenten die ik in een sequence diagram wil zien? In AngularJS-termen gaat het om de volgende componenten:

- HTML bestand (een template of een hele pagina)
- Controller
- Provider
- Factory
- Service
- Directive

Dit lijstje kan naar eigen smaak worden uitgebreid of ingekort, als het maar voor iedereen duidelijk is welke keuzes er gemaakt zijn. Alle andere JavaScriptobjecten zijn te onbelangrijk om in het sequence diagram te tonen. Daarvoor moet men maar in de broncode kijken. Als de applicatie ook gebruikt maakt van JavaScriptcomponenten buiten AngularJS om, bijv. JQuery of Google Maps JavaScript API, dan moet per geval gekozen worden of zo'n component altijd of nooit in een sequence diagram wordt weergegeven als lifeline. In het geval van JQuery zou ik kiezen voor 'nooit' (te detaillistisch) en in het geval van Google Maps voor 'altijd'.

Figuur 3 toont één lifeline voor de server. Als er in het scenario meerdere back-end APIs worden aangesproken, kun je één lifeline per API tekenen. De witte kleur geeft aan, dat de server buiten de applicatie valt, maar als de API's in je architectuurplaat een bepaalde kleur hebben gekregen, dan kun je die kleur ook gebruiken in je sequence diagram.

Messages

Goed, we weten nu welke lifelines we in het schema opnemen en welke niet. Nu moeten we de messages tussen die lifelines tekenen. Ik adviseer om alle messages te tonen, die in het betreffende scenario tussen de getekende lifelines vóórkomen. Als er dus in de code een component is die een andere component aanroept, maar deze aanroep staat in geen enkel sequence diagram vermeld, dan moet ofwel de code aangepast worden, ofwel een sequence diagram uitgebreid worden of toegevoegd worden. De interne werking van een component wordt daarentegen niet weergegeven – om die te kennen, zul je de broncode moeten bekijken. Deze regels geven duidelijkheid voor degenen die de sequence diagrams moeten begrijpen of implementeren.

Wat is nu precies een message in termen van JavaScript? De meest voor de hand liggende message is een functieaanroep. In Figuur 3 zijn dat alle messages die haakjes hebben, bijvoorbeeld isLoginEnabled( ) en onClickLogin( ). Deze vinden we één op één terug in de broncode:

```
<button type="submit"
    ng-click="ctrl.onClickLogin()"
    ng-disabled="!ctrl.isLoginEnabled()"> Login </button>
```

Eventuele parameters staan tussen de haakjes, maar als dat het diagram onoverzichtelijk maakt, kun je ook met drie puntjes aangeven dat de parameters zijn weggelaten.

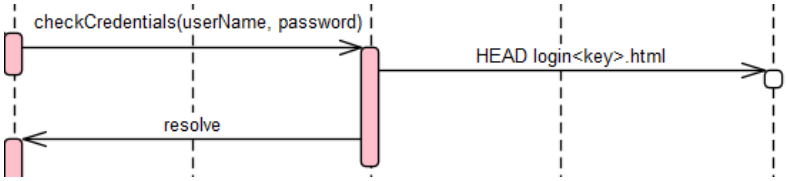
Een ander soort message betreft de acties van de gebruiker. Deze worden weergegeven als messages richting de HTML-component. De tekst bij de pijl geeft kort weer wat de gebruiker doet, bijvoorbeeld "open Login page". Omdat hiermee loginPage.html in actie komt, heb ik een pijl getekend van de gebruiker naar loginPage.html. Dit is een volledige HTML-pagina, die <script> tags bevat voor AngularJS, voor de loginapplicatie en voor de utilitiesmodule, waarin zich o.a. de sendMailService bevindt.

Vaak roept de ene component de andere aan via AngularJS, of via een ander object dat niet is afgebeeld als lifeline. In dat geval teken ik een pijl met een free-format tekst die duidelijk maakt wat er gebeurt. Zo staat er bij de tweede pijl in Figuur 3 de tekst “ng-controller”. In loginPage.html staat:

```
<body ng-controller="LoginController">
```

De message met de tekst “ng-controller” geeft dit weer in het sequence diagram.

Voor de duidelijkheid beschrijf ik het scenario ook altijd nog in woorden in de linker kantlijn.



Figuur 4. Asynchrone messages

Een dichte pijlpunt geeft een synchrone message weer en een open pijlpunt een asynchrone message (of een reply message, maar die komt zodadelijk aan bod). Wanneer de message een functieaanroep weergeeft, dan is die synchroon, behalve als de functie een promise oplevert. In Figuur 3 is o.a. de message checkCredentials asynchroon (ook weergegeven in Figuur 4). De code is als volgt:

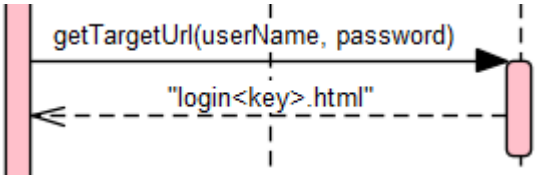
```
loginService.checkCredentials(userName, password).then(loginSuccessful, loginFailed);
```

Op het moment dat de loginService vastgesteld heeft dat de credentials goed zijn, doet hij een ‘resolve’ van de promise en zal de functie loginSuccessful worden aangeroepen. De ‘resolve’ is in Figuur 4 weergegeven als een asynchrone message terug naar de controller. Omdat de controller niet rechtstreeks, maar via het promise object wordt aangeroepen, heeft de pijl een free-format tekst, in dit geval ‘resolve’.

Om te controleren of de credentials goed zijn, doet de loginService een HTTP-call van het type HEAD. De broncode is:

```
$http.head(targetUrl).then( ...
```

Omdat $http geen lifeline is, heb ik ook hier bij de message een free-format tekst gebruikt. Ook deze functie geeft een promise terug en de message is dus asynchroon.



Figuur 5. Synchrone message met reply message

Als een functieaanroep gegevens teruggeeft, dan kun je een reply message tekenen, zoals in Figuur 5. In dit voorbeeld levert getTargetUrl een string op. Ik laat de meeste reply messages weg, om het schema overzichtelijk te houden, maar als er gegevens terugkomen die cruciaal zijn in het betreffende scenario, dan teken ik wel een reply message.

Foutscenario

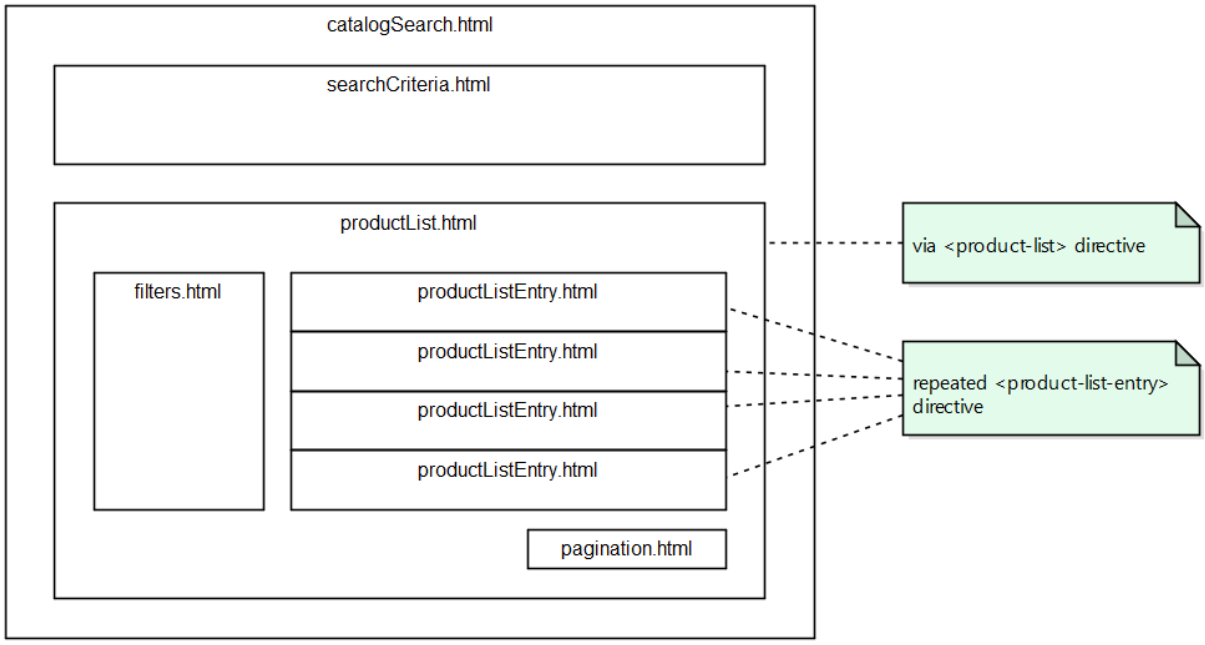
Dan hebben we nog het scenario waarin de login-poging mislukt. We kunnen hiervoor een tweede sequence diagram tekenen, we kunnen de twee scenario's ook combineren in één schema m.b.v. een combined fragment (met operator 'alt'), of we kunnen ervoor kiezen het foutscenario helemaal niet te modelleren. In dit geval vind ik het foutscenario niet zo interessant om te modelleren. De 'resolve' message in Figuur 3 wordt een 'reject' message en dan stopt het scenario. Dat kunnen we beter in één zin even erbij vermelden in plaats van het schema te compliceren met een combined fragment of een heel nieuw schema te tekenen. Als er in het foutscenario een nieuwe interactie tussen componenten zou plaatsvinden, dan zou ik dat scenario wel modelleren. Het hangt er dan van af hoeveel overlap er is met het basisscenario, of ik het foutscenario in hetzelfde schema inteken, of dat ik een nieuw sequence diagram maak. In het algemeen geldt, dat ik er niet de voorkeur aan geef om allerlei scenario's in één schema te proppen met gebruik van meerdere (evt. geneste) combined fragments, omdat dat de leesbaarheid vermindert.

Ontwerp per module, per component

Naast de scenario's is er meestal ook een stukje ontwerp nodig op basis van top-down decompositie: een ontwerp per module en daarbinnen voor sommige componenten ook een apart ontwerp. In de AngularJS-applicaties waar ik aan gewerkt heb, zag ik vooral behoefte aan:

1. Een overzicht van de structuur van complexe HTML-pagina's.
2. Een ontwerp van generieke services.

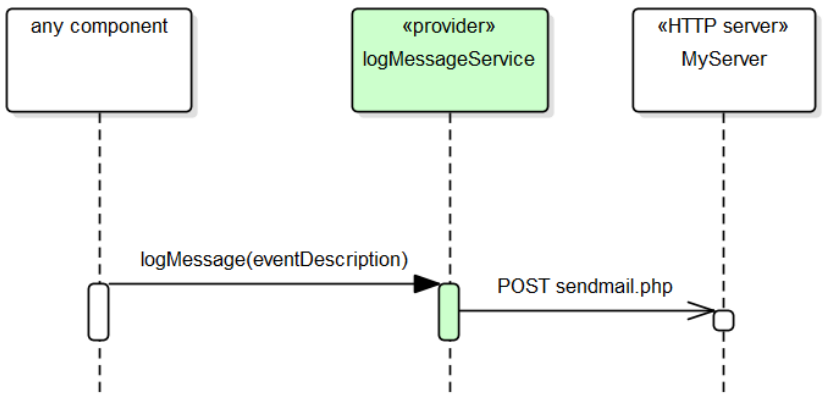
Figuur 6 geeft een voorbeeld van de eerste categorie. Het is geen UML-diagram, maar het is een gestyleerde weergave van de layout van HTML-templates binnen een HTML-pagina. Het is goed om zo'n plaatje te tekenen en met elkaar te bespreken vóórdat de implementatie begint. Voor degene die in een latere fase een wijziging moet doorvoeren, geeft het snel inzicht in de structuur.



Figuur 6. HTML-structuur

De andere categorie is het ontwerp van generieke services. Hoewel de charme van sequence diagrams is, dat je de message flow door meerdere applicatielagen heen kunt weergeven, kan dit ook leiden tot veel duplicatie. In Figuur 3 wordt de logMessage functie van de logMessageService één keer aangeroepen en deze verzendt één HTTP-bericht. Stel nu, dat logMessage veelvuldig in allerlei scenario's vóórkomt, moeten we dan telkens ook dat HTTP-bericht weergeven? Als er dan iets verandert aan logMessage, moet we het op al die plaatsen aanpassen. Stel dat logMessage niet één, maar meerdere andere componenten gebruikt, dan wordt helemaal duidelijk dat die herhalingen ongewenst zijn. Het ontwerp van zulke generieke services kunnen we dan ook beter centraliseren, zoals in Figuur 7. De meest linker lifeline is hier een anonieme component, die de generieke service aanroept.

Verder kan er naar behoefte gebruik gemaakt worden van andere ontwerpvormen. Ik maak meestal wel een paar class diagrams om complexe datastructuren inzichtelijk te maken en af en toe een state machine diagram. Onderschat ook niet het belang van gewone tekst, die kort en helder uitlegt welke ontwerpkeuzes er zijn gemaakt.



Figuur 7. Ontwerp van een generieke functie

Tools

Wat is nu een goede manier om ontwerpdocumentatie, waarin veel gebruik gemaakt wordt van UML, vast te leggen en te beheren? Laat ik van de vele mogelijkheden er drie noemen:

1. Een wiki met UML-ondersteuning.
2. Een UML-ontwerptool met een documentgenerator.
3. Een tekstverwerker met een koppeling naar een UML-ontwerptool.

Voor een grondige bespreking van dit onderwerp is in dit artikel geen ruimte, maar laat ik enkele argumenten geven waarom deze mogelijkheden gekozen worden.

Een wiki met UML-ondersteuning

Een voorbeeld hiervan is Confluence, in combinatie met de Gliffy plugin. Een groot voordeel van een wiki is, dat het gemakkelijk is om er met meerdere mensen tegelijk aan te werken. Degenen die niet zelf ontwerpen, maar het ontwerp wel lezen, kunnen er opmerkingen aan toevoegen. Wijzigingen zijn direct zichtbaar voor iedereen. Ook is de historie van wijzigingen te raadplegen. Verder zorgen de hypertext-eigenschappen ervoor, dat je allerlei kruisverwijzingen tussen de diverse onderdelen van het ontwerp kunt aanbrengen en dat de lezer goed door het ontwerp heen kan navigeren. Voor Confluence geldt verder, dat het goed geïntegreerd is met andere Atlassian tools, zoals Jira (issue management) en Bitbucket (source code management).

Vergeet niet, als je een nieuw project start, dat je het ontwerp niet alleen maakt voor de ontwikkelaars die het ontwerp implementeren, maar ook voor degenen die later willen weten hoe de opgeleverde software in elkaar zit. Vaak wordt er na oplevering doorontwikkeld aan een nieuwe versie en dan is het bij een wiki lastig om aan te geven welke versies van de ontwerppagina's overeenkomen met een bepaalde versie van de applicatie.

Een UML-ontwerptool met een documentgenerator

Een voorbeeld is Enterprise Architect van SparxSystems. Een dergelijke tool geeft een goede ondersteuning voor het ontwerpen in UML. Het maken en wijzigen van schema's gaat gemakkelijk en snel. Alle UML-elementen zijn afzonderlijke entiteiten, die je in meerdere schema's kunt gebruiken. Het wijzigen van de naam of een andere eigenschap van zo'n element kun je centraal doorvoeren en dan wordt het zichtbaar in alle schema's waarop dat element vóórkomt. Ook gewone tekst kun je in zo'n tool kwijt. Met de documentgenerator kun je een Word-document genereren, maar ook een HTML website. Dergelijke output laat vaak wel te wensen over.

Elke goede UML-ontwerptool kent wel versiebeheer, maar het is omslachtig om een vorige versie te bekijken. Zelf maak ik bij elke oplevering van de applicatie een complete kopie van het ontwerp,

zodat ik naderhand gemakkelijk het ontwerp van een vorige versie van de applicatie kan raadplegen.

Een tekstverwerker met een koppeling naar een UML-ontwerptool

Een voorbeeld is Microsoft Word in combinatie met eaDocX en Enterprise Architect. Je kunt voor de begeleidende tekst alle mogelijkheden van Word gebruiken en de UML-schema's uit Enterprise Architect op de juiste plek in het document invoegen. De tool eaDocX zorgt ervoor, dat wijzigingen in Enterprise Architect ook worden doorgevoerd in Word. Op deze manier heb je zowel een goede ondersteuning voor het ontwerpen in UML, als een goede tekstverwerker tot je beschikking. Je kunt zo ontwerpdocumenten van topkwaliteit maken. Een ander voordeel is, dat je documenten een bepaald versienummer kunt geven, gekoppeld aan de versie van de software. Zo weet je precies welke versie van het ontwerp bij welke versie van de software hoort.

Epiloog

Ik hoop dat dit artikel je wat houvast geeft bij het maken van een technisch ontwerp voor je AngularJS-applicatie. Als je de complete broncode van de login-applicatie wilt bekijken, om dit te vergelijken met het sequence diagram in Figuur 3, ga dan naar www.admiraalit.nl/admiraal/angular/loginapp. Succes!

Hans Admiraal, (freelance IT architect)
admiraal@aol.nl
www.admiraalit.nl