Verslag Afstudeerproject

ONTWIKKELING VAN DE BASIS VOOR EEN NIEUW CONTENT MANAGEMENT SYSTEEM

webstores

Nummer: 20050116

Mens & Informatica

Stenden Hogeschool

Bedrijf: Webstores internet totaalbureau

Nummer: 20050116

2. Mei 2012 – 28. September 2012

Inhoud

Summary

To complete the the study „Mens & Informatica“ it's required to realize a project on a approriate level in a suitable IT company. This report deals with the realization of a new Content Management System for „Webstores Internet totaalbureau“. Webstores is fast growing midsize company with 25 internet experts which successfully delivers custom internet solutions for more than 200 clients on a daily basis. Most of this sites are made with their own CMS WebManager2 which was developed some years ago. But in every sector and in particular the fast growing IT world there is always the need to innovate regulary to remain competetive. That's the reason Webstores decided to develop a brand new CMS which matches up the today's requirements. The purpose of this project is the development of the basis for that new CMS.

It's important to choose a good basis for the new system, because developing a complete new CMS from the scratch without a framework isn't an option. There exist a few very good PHP frameworks which offer good components to simply the the task of developing a complex system. Reinventing the wheel is rarely the right choice.

As a first step it was necessary to priorize possible aspects of the frameworks. A survey of employees with a suitable interview technique was needed to identifiy and priorize the needed functionalities of the framework. With these prioritized requirements it was possible to evaluate the right framework for Webstores. It turns out, that Symfony2 was the best suited framework for the new CMS. Symfony2 is a modern PHP framework with high quality components and an built-in Dependency Injection solution, which makes everything extremly flexible. A good choice.

The second part of the project was the development of the softwaredesign based on a well-conceived datamodel. To simply the development, everything was splitted into four separated bundles:

- CoreBundle
  A core component, responsible for channels and layouts
- WidgetBundle
  Responsible for a convienient creation of widgets and administration of them
- PageBundle
  Responsible for creation and administration of pages. This bundle also acts as a „glue bundle“ of the CoreBundle and WidgetBundle.
- ContentBundle
  This bundle provides a simple interface for the creation and administration of new content types

As a second requirement, the code must be well documented and unittested. This report especially deals with the acquired experiences made during the development of the system. It also describes the motives and reasons of the chosen approaches.

Result

Eventually, the final product fulfills the required functionalities. Every component of the software is replaceable, flexible, well documented and unittested. It's extensible and offers a good basis for the future development of a complete new and modern CMS for Webstores.

1. Inleiding

Voor de opleiding “Mens en Informatica” moet voor het succesvol afronden een afstudeerproject zelfstandig uitgevoerd worden. Dit project dient in een bedrijf uitgevoerd worden met een voldoende grootte en goed bij de studie passen.

Omdat ik mij tijdens mijn studie gespecialiseerd heb in webontwikkeling was het voor de hand liggend mijn kennis ook tijdens mijn afstudeerproject te verdiepen. Ik moest niet lang zoeken. Meneer Blankestijn vertelde mij over Webstores in Hardenberg. Webstores is een “internet totaalbureau” en ontwikkelt maatwerk internetapplicaties.

Webstores en ik waren benieuwd en ik werd uitgenodigd. Het gesprek was veelbelovend en ik had een goed indruk van Webstores: vele jonge medewerkers, een goede sfeer en veel kennis in mijn vakgebied.

Ik mocht de basis voor hun nieuw CMS schrijven (WM3). Dit was natuurlijk een uitdagend project, omdat hun internetapplicaties in de toekomst ook via dit systeem zouden draaien.

1.1 Webstores

Webstores is een Internet totaalbureau en biedt maatwerk oplossingen. Om dit te bereiken, zijn er meer dan twintig medewerkers bezig. Er zijn in principe drie (productie) afdelingen. Design, Frontend en Backend met vele specialisten. Voor webapplicaties maakt Webstores meestal gebruik van de taal PHP en hun eigen Content Management System WebManager 2. Een groot aantal webapplicaties draait via dit systeem. De medewerkers werken vaak zelfstandig aan één specifiek project. De sfeer is aangenaam, rustig, professioneel en iedereen is gemotiveerd.

“Sinds 2000 brengt Webstores ideeën tot leven. We doen dit met een hecht en enthousiast team van 22 gespecialiseerde professionals. Hiermee hebben we al het talent voor interactieve media in eigen huis. Webstores is dan ook een internet totaalbureau.

Samen werken we verrassende concepten uit tot frisse en toegankelijke internetoplossingen. Een evenwichtige mix van functionaliteit, design, technologie en communicatie.

Webstores werkt aan websites, webshops en (maatwerk) applicaties zoals order- en voorraadsystemen. Hierbij verzorgen we het complete traject. Van ontwerp en techniek tot marketing en beheer." – www.webstores.nl

1.2 Inleiding afstudeeropdracht

Uit meerdere mogelijke afstudeerprojecten heeft Webstores ervoor gekozen, een basis voor het nieuwe Content Management System (CMS) te laten ontwikkelen door mij.

Situatie: WebManager2

WebManager2 is een CMS geschreven via PHP, het framework CakePHP en de JavaScript library ExtJS. Het systeem biedt de mogelijkheid, om een webpagina te beheren – nieuwe pagina’s aan te

maken en (beperkt) widgets voor een pagina aan te maken en te plaatsen. Ook werden er verschillende modules geschreven zoals Webshop, FAQ en Agenda.

Sinds jaren gebruikt Webstores nu dit CMS. Het praktische en commerciële gebruik hiervan heeft natuurlijk behoorlijk veel evaring opgeleverd. Het werd duidelijk waar de voordelen en – vooral - nadelen van het system liggen en wat beter kon. Webstores had al langer een concreet idee van de eisen van het nieuwe systeem.

Doel: WebManager3

Met WM3 wil Webstores een compleet CMS systeem opbouwen, welk aan moderne eisen kan voldoen. De basis moet ten eerste heel flexibel zijn. Alles in WM3 moet uitbreidbaar en aanpasbaar zijn. Hoe flexibeler het systeem is, des te minder nieuwe code moet geschreven worden. Dit bespaart tijd en tenslotte geld. WM3 moet daarom van gevestigde technieken en patterns gebruik maken.

Veel code en tijd kan bespaard worden, als men al een framework (=toolbox of raamwerk) ter beschikking heeft, die veel voorkomende taken overbodig maakt. De keuze van een passend framework voor WM3 is een heel belangrijke beslissing. Hiervoor moet een onderzoek uitgevoerd worden.

Tenslotte moet de basis door andere mensen gebruikt worden en daarom schoon (=coding standards) en goed gedocumenteerd zijn.

Channels

Voor WM3 wil Webstores graag het concept van kanalen gebruiken. Een kanaal representeert een applicatie binnen een applicatie. Dit betekent, dat een klant één WM3 instantie heeft waarmee hij verschillende webapplicaties kan beheren. Een kanaal kan bijvoorbeeld een webshop zijn of een blog. Ieder kanaal heeft dan een eigen URL en verschillende eigenschappen zoals een taal. Dit zou voor een gebruiker tot een beter overzicht leiden. Per kanaal moet een gebruiker layouts kunnen kiezen en pagina's en content beheren.

Layouts

Kanalen in WM3 moeten over layouts kunnen beschikken. Een layout kan bijvoorbeeld een HTML template zijn. Deze layouts moeten kunnen onderverdeeld worden in verschillende bereiken (slots). In deze slots wordt uiteindelijk de content geplaatst. Meest prominente voorbeelden qua inhoud in WM3 zouden met name Widgets zijn. Slots moeten configureerbaar en hiërarchisch zijn (slots bevatten subslots enz.).

Widgets

Functionaliteiten voor een webapplicatie moeten via het widget concept omgezet worden. Widgets in het kader van WM3 zijn kleine subapplicaties die in slots geplaatst kunnen worden. Dit kan bijvoorbeeld een weergave voor Twitter zijn of een registratie formulier voor een gebruiker. Widgets moeten configureerbaar zijn: Een Twitter widget zou bijvoorbeeld flexibele eigenschappen zoals een accountnaam en een wachtwoord hebben. Maakt een gebruiker een Twitter widget aan, dan moet hij instaat zijn dit widget te configureren (flexibele widgets)

Ook moeten widgets op de hoogte zijn van de slot waar zij geplaatst worden. Een navigatie widget zou in een header bijvoorbeeld automatisch via tabs getoond worden, maar in een sidebar als een lijst.

Pages

Pages in WM3 moeten toonbare pagina's zijn die bij een kanaal horen. Ze hebben een layout en een slots met widgets. Daardoor is een Page het eindresultaat van alle bovenstaande concepten.

Supermodule

Met de supermodule moet er eenvoudig vanuit het CMS een nieuw content type gedefinieerd kunnen worden. Op basis van dit content type kan uiteindelijk nieuwe content geschreven worden die gebruikt kan worden binnen diverse kanalen.

Er is bijvoorbeeld behoefte aan het publiceren van persberichten. Via de supermodule wordt dan een nieuw content type met de naam 'Persbericht' aangemaakt en voorzien van de velden 'titel', 'intro', 'inhoud' en 'auteur'.

Onderzoek: Passend framework

Een professioneel CMS "from scratch" te bouwen zonder hulpmiddelen van een stabiel framework is niet zinvol. Er zijn vele PHP frameworks op internet te vinden, die allemaal van verschillende kwaliteit zijn en deels beter of slechter geschikt zijn voor dit type project. Daarom moet vooraf een onderzoek uitgevoerd worden. Een slechte keuze zou op lange termijn de doorontwikkeling van het CMS bedreigen. Heeft het framework een grote user base? Is het framework stabiel genoeg? Wordt het framework op lange termijn doorontwikkeld? Dit zijn belangrijke aspecten en moeten via een onderzoek beantwoord kunnen worden.

Documentatie en coding standards

Uiteindelijk moet het systeem goed gedocumenteerd zijn. Dit houdt in dat de code goed te lezen is en een bruikbare informatie voor een ontwikkelaar biedt. Ook moeten er externe documenten komen, die niet alleen de code beschrijven, maar ook de werkwijze en concepten van het systeem uitleggen. Het CMS zal in toekomst uitgebreid en in de praktijk gebruikt worden, waardoor een goede documentatie een belangrijk eis is.

De code moet aan een bepaald coding standard (PSR-1) voldoen.

REST API

Het CMS moet onafhankelijk zijn van de gebruikersinterface. Daardoor kunnen verschillende interfaces ontwikkeld worden. De eerste interface zal wel via een browser functioneren. Toch moet er een mogelijkheid komen het systeem op een compleet andere manier te beheren. De oplossing hiervoor is een gestandaardiseerd interface voor requests. Binnen een kader van het afstudeerproject moet een REST API gedefinieerd en ontwikkeld worden.

Unit tests

De code moet door unit tests gevalideerd worden.

Grenzen

Onderdeel van het afstudeerproject is uitdrukkelijk niet een afgerond CMS systeem maar de basis hiervan. Ook moet er geen gebruikersinterface voor het systeem gemaakt worden.

2 Onderzoek

2.1 Introductie

Voor het CMS moet een geschikt framework gekozen worden. Op internet zijn tegenwoordig enorm veel frameworks te vinden. Het onderzoek gaat voornamelijk om het vinden van een geschikt framework voor webstores.nl. Er zijn goede frameworks en slechte frameworks. Geschikte frameworks en ongeschikte frameworks - of frameworks die goed zijn, maar niet bij de doelstelling passen (CMS bouwen). Er zijn vele aspecten waarmee rekening gehouden moet worden. Aan de ene kant is Webstores als bedrijf, die bepaalde eisen aan het framework stelt. Aan de andere kant de verschillende frameworks met hun features.

De eerste stap van het onderzoek houdt dus in:

- De eisen te bepalen
- De eisen in categorieën onder te brengen
- De eisen te prioriteren

Vervolgens zal ik via verschillende bronnen mogelijke frameworks opzoeken en grof inschatten of ze geschikte kandidaten zijn. Is dit het geval dan komen zij in een lijst met alternatieven. In dit hoofdstuk zal ik ieder framework kort presenteren en hun doelstelling/filosofie beschrijven.

2.2 Interviews

Het is belangrijk om vóór het onderzoek grenzen te bepalen. In het kader van het afstudeerproject is vanzelfsprekend niet genoeg tijd om tientallen van frameworks en hun interne werkwijze exact te bestuderen. Vier frameworks zijn tegenwoordig heel populair en zijn geschikte kandidaten voor het nieuwe CMS van Webstores:

De eerste zinvolle stap is het definiëren van mogelijke eisen. Anders zou de zoektocht naar het geschikte framework heel blind verlopen. Hier was mijn eigen ervaring met PHP en de frameworks zinvol. Ook de eisen aan het project zelf zijn een goede bron om eisen te classificeren.

- Flexibiliteit: Flexibiliteit van het framework
- Snelheid: Hoe snel is de uitvoering van scripts?
- Kwaliteit: Hoe hoog is de kwaliteit van de code (unit tests, coding standards)
- Lage kosten: Onmiddellijke kosten van het framework
- Goede documentatie: Hoe omvangrijk is de documentatie?
- User Base: Heeft het framework een grote user base op internet?
- Third Party Components: Zijn er goede componenten beschikbaar waarvan gebruikt gemaakt kan worden?
- Kennis in het bedrijf: Hebben de medewerkers kennis van het framework?
- Leercurve (weinig moeite): Is het framework makkelijk te begrijpen?
- Geschiktheid voor een CMS
- Long Time Support: Blijft het framework in ontwikkeling?
- IDE Ondersteuning: Hoe goed is de integratie in ontwikkelingsomgevingen?
- Server Ondersteuning: Heeft het framework specifieke server configuraties nodig?

- Omvang Core: Hoe omvangrijk is de core. Hoeveel functionaliteit is al beschikbaar?
- Weinig configuratie: Is een applicatie makkelijk in te richten?
- Ontwikkeling (efficiëntie): Hoe snel kan met de hulpmiddelen van het framework snel een applicatie gebouwd worden?

Een subjectieve classificatie is vanzelfsprekend nog geen geschikte basis voor een onderzoek. Het is daarom belangrijk om de prioriteiten door de ervaren medewerkers te laten beoordelen. Een wegingsmatrix, die van ervaren en betrokken personen ingevuld wordt, is in dit geval een beproefde en exacte interview techniek. Aan de hand van het resultaat van deze interviews kunnen eisen exact geprioriteerde worden, waardoor een goede basis voor het opzoeken van een goed framework ontstaat.

De medewerkers moeten in deze matrix de prioriteit van elk eis met elkaar vergelijken. Aan het eind komt een percentage voor ieder eis tot stand (100% - heel belangrijk; 0% - helemaal niet belangrijk)

Voor ieder medewerker moet bovendien een weging bepaald worden (voor inspraak/medezeggenschap). Op basis van deze weging kan heel exact bepaald worden, waar de eisen voor Webstores liggen. De ingevulde tabellen zijn als bijlage beschikbaar.

Samenvattend kwam volgend resultaat tot stand:

	Christiaan	Holger	Jan	Johannes	Mark		
Weging	20	15	20	10	35	100,00	
Goede documentatie	73,3	64,3	100,0	78,6	92,9	**84,67**	*81,8*
Kwaliteit	86,7	100,0	33,3	100,0	100,0	**84,00**	*84,0*
Flexibiliteit	100,0	85,7	80,0	100,0	71,4	**83,86**	*87,4*
Ontwikkeling (efficiëntie)	66,7	57,1	73,3	57,1	92,9	**74,79**	*69,4*
Server Ondersteuning	80,0	78,6	73,3	92,9	35,7	**64,24**	*72,1*
Long Time Support	53,3	71,4	20,0	57,1	71,4	**56,10**	*54,7*
Snelheid	93,3	78,6	53,3	92,9	7,1	**52,90**	*65,0*
Kennis in het bedrijf	33,3	42,9	86,7	42,9	50,0	**52,21**	*51,1*
Leercurve (weinig moeite)	26,7	21,4	80,0	35,7	50,0	**45,62**	*42,8*
Geschiktheid voor een CMS	46,7	92,9	33,3	64,3	21,4	**43,86**	*51,7*
IDE Ondersteuning	60,0	7,1	46,7	35,7	35,7	**38,48**	*37,0*
User Base	13,3	35,7	20,0	14,3	71,4	**38,45**	*31,0*
Weinig configuratie	20,0	14,3	46,7	21,4	57,1	**37,62**	*31,9*
Third Party Components	6,7	35,7	13,3	7,1	64,3	**32,57**	*25,4*
Lage kosten	40,0	35,7	0,0	42,9	14,3	**22,64**	*26,6*
Omvang Core	0,0	35,7	0,0	14,3	0,0	**6,79**	*10,0*

Het percentage aan de rechte kant (blauw achtergrond) is heel betekenisvol. Dit percentage representeert de uiteindelijke prioriteit van elk van de eisen. Het blijkt bijvoorbeeld, dat de kosten of

de omvang van de core een heel kleine rol spelen. Veel belangrijker is aan de andere kant bijvoorbeeld de efficiëntie van de ontwikkeling van eigen code. Ook een goede documentatie van het framework is heel belangrijk voor Webstores.

Op basis van deze informatie kan doelgericht het passende framework gezocht worden.

2.3 Beoordeling

2.3.1 Flexibiliteit

Een goed framework moet ontwikkelaars de mogelijkheid bieden om wijzigingen in de applicatie snel en efficiënt uit te voeren. Met name is Dependency Injection heel belangrijk. Dit is een belangrijk feature/pattern om objectstructuren vast te lijmen en vaste afhankelijkheden in een (web) applicatie klein te houden. Hierdoor wordt het systeem heel flexibel: De afhankelijkheden van objecten worden door een Dependency Injection component beheerd.

CakePHP

Vergelijkbaar met Symfony is CakePHP niet flexibel. Wel is de database bijvoorbeeld niet strikt en ook de routering is flexibel. Maar helaas wordt met name niet gebruik gemaakt van Dependency Injection.
Hier zijn wel oplossingen op internet te vinden om dit in CakePHP te integreren. Toch is het beter, als het concept al vanuit de basis van een framework ondersteund wordt.
(Cijfer 5/10)

Symfony2

Symfony biedt als feature geïntegreerd Dependency Injection, waarvan zelfs core componenten gebruik maken. Hier regelt Symfony2 voornamelijk binnenkomende requests en de routering. Dit zijn dus componenten, die gemakkelijk vervangbaar zijn. Daardoor wordt het framework heel flexibel.
(Cijfer 9/10)

Zend

In principe is Zend een verzameling van onafhankelijke componenten met een hoge kwaliteit. Daardoor is het systeem natuurlijk potentieel heel flexibel. Ook gebruikt de distributie net zoals Symfony2 een Dependency Injection component.
(Cijfer 8/10)

YII

Yii biedt een relatief hoog niveau aan flexibiliteit – als je het op die YII manier doet, kun je flexibele systemen schrijven. Het concept van DI zit niet in de core van YII.
(Cijfer 5/10)

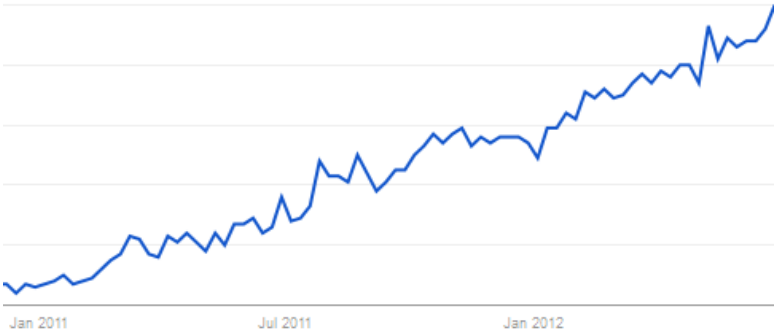
2.3.2 Documentatie

CakePHP

CakePHP heeft een uitgebreide documentatie. Door een grote user base (het project is al ouder) zijn er vele tutorials en boeken op internet te vinden.

(Cijfer 8/10)

Symfony2



De documentatie van Symfony**2** is nog niet heel goed. Symfony2 is nog redelijk nieuw en boeken zijn bijvoorbeeld helemaal niet verkrijgbaar. Toch is er op hun website al wel een redelijk goede documentatie en zijn er leuke tutorials op internet te vinden. Doordat het project heel actief is en groeit (zie diagram- *Google Trends)*, zal in toekomst ook de documentatie beter worden.

(Cijfer 5/10)

Zend

Het framework is een heel bekend PHP framework vooral ook omdat de ontwikkelaars van PHP dit framework ondersteunen. Daardoor is ook redelijk veel documentatie op internet te vinden.

(Cijfer 8/10)

YII

In vergelijking met andere frameworks is YII redelijk jong. Het eerste release was eind 2008. Er zijn verschillende boeken verkrijgbaar en hoewel dit framework vergelijkbaar met de andere frameworks niet een kleinere user base heeft, is op internet goede documentatie beschikbaar.

(Cijfer 6/10)

2.3.3 Kwaliteit

Kwaliteit is principieel niet gemakkelijk via een cijfer te meten. Om de kwaliteit van de frameworks toch in te schatten moet aan de hand van geschikte indicatoren de kwaliteit gemeten worden. Andere belangrijke indicatoren zouden te gelijker tijd ook eisen zijn (zoals documentatie of snelheid) en worden in een apart onderdeel van dit onderzoek behandeld

- Tested Code
  Unit Tests en de resulterende Code Coverage zijn een geschikte indicator
- Doorontwikkeling
  Door onder andere de leeftijd, de omvang, een roadmap en de grootte van de community kan een doorontwikkeling zeker gesteld worden
- Succesvolle voorbeelden in de praktijk
  Zijn er succesvolle en grote webapplicaties op internet te vinden? Een webpagina zou niet succesvol zijn, als die basis niet van een geschikt kwaliteit was.

Symfony

Bekende pagina's:

- http://delicious.com/
- http://www.dailymotion.com
- http://bookmarks.yahoo.com

Dat populaire pagina's zoals DailyMotion Symfony2 als basis gebruiken, verteld veel over het kwaliteitsniveau.
Alle core functionaliteiten worden principieel getest via unit tests en zijn daardoor van hoge kwaliteit. Symfony2 gebruikt vaste coding standards, waardoor de code heel schoon blijft.
(Cijfer 9/10)

YII

Bekende pagina's:

- http://yii.poweredsites.org/top

Er zijn niet heel bekende pagina's (YII is nog niet oud en daarom nog niet populair) maar wel professionele pagina's.

YII beschikt over omvangrijke unit tests. Coding standards zijn in YII redelijk goed, waardoor de code schoon blijft. Toch heeft Symfony2 bijvoorbeeld striktere en betere standards.
(Cijfer 6/10)

CakePHP

Bekende pagina's:

- http://book.cakephp.org/1.2/view/510/Sites-in-the-wild

Er zijn behoorlijk vele pagina's via CakePHP op Internet te vinden. Toch moet rekening gehouden worden, dat CakePHP in vergelijking met andere frameworks redelijk "oud" is

CakePHP beschikt over omvangrijke unit tests en er wordt met bepaalde coding standards gewerkt.
(Cijfer 7/10)

Zend

Bekende pagina's: Zend wordt wereldwijd op duizenden pagina's ingezet

Tested Code: Zend beschikt over omvangrijke unit tests met een hoog code coverage van minstens 80%. Door goede coding standards is de kwaliteit heel hoog. *(Cijfer 9/10)*

2.3.4 User base

Aan de hand van de activiteiten op Internet kan de user base goed ingeschat worden. Via populaire services (zoals Google, Github en StackOverflow) kan de content dichtheid bepaald worden (**Fehler! Verweisquelle konnte nicht gefunden werden.**). Dit zal de basis voor de cijfers zijn.

- **CakePHP:** Het framework heeft een gemiddelde user base in vergelijking met de andere frameworks
  (Cijfer 6/10)
- **Symfony:** Er zijn vele projecten op basis van Symfony op GitHub te vinden.
  (Cijfer 7/10)

- **Zend:** Zend Framework is volwassen en heeft daarom uiteraard een grote user base
  (Cijfer 8/10)
- **Yii:** Te nieuw en daarom nog niet zo populair als de medestrijders
  (Cijfer 4/10)

2.3.5 Third party components

Uiteraard zijn er niet voor alle problemen componenten op internet verkrijgbaar - maatwerk internet oplossingen betekent Core Business voor Webstores.nl. Desondanks zijn Third-Party-components een eis. Het wiel hoeft niet opnieuw uitgevonden worden.

Als steekproef dienen drie typische componenten: Een webshop, een forum en een blog component. Hoe hoger de kwaliteit van beschikbare componenten zijn, hoe beter.

CakePHP

Webshop: http://cakeforge.org/projects/bakesale/
Een webshop van redelijk slechte kwaliteit
Forum: http://milesj.me/blog/read/changelog-forum-3.1
Het project lijkt heel actief te zijn
Blog: http://code.google.com/p/lilblogs/
Een omvangrijk blog systeem

Uiteindelijk lijkt er op internet geen grote en keurige bron voor goede componenten te zijn.
(Cijfer 4)

Symfony2

Webshop: https://github.com/vespolina, https://github.com/Sylius
Vespolina lijkt een heel goede webshop oplossing te zijn met onafhankelijke onderdelen. Ook Sylius lijkt heel goed in elkaar te zitten.
Forum: Een simpel maar goed forum
Blog: https://github.com/sonata-project/SonataNewsBundle
Een omvangrijk Blogging systeem voor Symfony2

De componenten voor Symfony2 maken uiteindelijk een heel goede indruk. Opvallend is dat de meeste projecten op Github te vinden zijn.
(Cijfer 8)

Yii

Webshop: http://www.yiiframework.com/extension/yiishop/
De webhop is op een redelijk laag niveau
Forum: http://www.yiiframework.com/extension/yii-forum/
Het forum lijkt iets beter te zijn, maar is niet omvangrijk
Blog https://github.com/WebDevPT/Yii-Blog-Enhanced
Een blog component is afgeleid van een demo project voor een blog. Het lijkt desondanks van een voldoende niveau te zijn.

De community van YII lijkt nog niet groot genoeg te zijn waardoor nog niet genoeg componenten beschikbaar zijn
(Cijfer 3)

Zend
Het is moeilijk, om componenten voor Zend te vinden, omdat hier vaak alleen delen van het framework gebruikt worden en niet het hele framework. Dit betekent, dat dit soort componenten ook eigen concretere structuren hebben waardoor een integratie in een CMS zoals WM3 moeilijk wordt.
(Cijfer 3)

2.3.6 Kennis in het bedrijf
Het inwerken in een framework kost tijd en daarom geld. Er zijn in totaal tien medewerkers bezig met PHP en het backend server al getraind is dit een groot voordeel. Bij webstores.nl hebben vier medewerkers kennis van Symfony2. 10 mensen hebben kennis van CakePHP (de basis voor WM2).

- **CakePHP:**
  (Cijfer 10/10)
- **Symfony:**
  (Cijfer 4/10)
- **Zend:**
  (Cijfer 0/10)
- **Yii:**
  (Cijfer 0/10)

2.3.7 Leercurve (weinig moeite)
Flexibiliteit en dynamiek heeft niet alleen voordelen. Flexibiliteit gaat vaak gepaard met hogere complexiteit en abstractie. Hogere complexiteit leidt vaak tot een steilere leercurve wat weer leidt tot langere training en meer kosten voor het bedrijf.

Een exacte leercurve te verkrijgen is heel moeilijk, omdat dit een subjectieve inschatting is en geen bronnen op internet zijn die dit kunnen valideren. Algemeen lijken Symfony2 en Zend iets moeilijker te zijn aan het begin (telkens *Cijfer 4*). Yii en CakePHP zijn redelijk snel te leren (telkens *Cijfer 6*). De cijfers zijn dus heel voorzichtig bepaald, wat betekent, dat zij niet erg afwijkend zijn van elkaar.

2.3.8 Long Time Support
Het beste framework is geen goede keuze als het systeem niet op lange termijn ondersteunt wordt. Er moet voor een voortdurende ontwikkeling gezorgd worden. Misschien worden bugs in toekomst niet meer gefikst van de community. Dit is een risico en zou invloed hebben op die kwaliteit van WM3.

CakePHP
De toekomst van CakePHP is niet goed in te schatten. Dit komt ook doordat er sommige (head)ontwikkelaars een fork ontwikkelen (Lithium)
(Cijfer 4/10)

Symfony
Symfony heeft een roadmap waarbij versie 2.2 LTS heeft. Bovendien groeit de community op dit moment heel sterk.
(Cijfer 8/10)

Yii
Doordat Yii relatief klein en jong is, is de support in de toekomst ook niet zeker
(Cijfer 3/10)

Zend
Zend Framework is nauw verbonden met de ontwikkeling van PHP in het algemeen. De actuele versie (2) is een versie met LTS.
(Cijfer 9/10)

2.3.9 IDE Ondersteuning
Moderne editors zijn tegenwoordig krachtige hulpmiddelen geworden, waarmee veel tijd wordt bespaard. Vaak worden framework-specifieke taken via plug-ins overgenomen. De medewerkers bij Webstores werken onder andere met Netbeans, Aptana, Notepad++, Vim, PHPStorm en Eclipse. In de bijlage is een vergelijking toegevoegd met uiteindelijke cijfers.

2.3.10 Server ondersteuning
PHP en MySQL worden wereldwijd grootschalig op de servers ondersteund. Een framework heeft maximale ondersteuning, als het met de standaard distributie van PHP (>= v5.3) werkt.

Alle frameworks draaien op een standaard webserver met PHP 5.3. *(Cijfer 10)*

2.3.11 Omvang Core
Hoeveel functionaliteit zit al in het framework? Werden al standaardmatig bibliotheken voor herhalende problemen (bijvoorbeeld E-mail) geïntegreerd of moeten ze eerst moeizaam handmatig geïntegreerd worden (resultaten zie bijlage)?

2.3.12 Weinig configuratie
Is er een verplichte configuratie voor het framework nodig en hoe omvangrijk en complex is deze?

CakePHP
De configuratie van CakePHP is te vergelijken met Yii, waarbij die configuratie van YII intuïtiever is. Bovendien moesten de schrijfrechten van de mappen aangepast worden, omdat anders geen toegang mogelijk was.
(Cijfer 7)

Symfony2
De installatie van Symfony2 is eenvoudig. Het pakket wordt gedownload en in de root map van de webserver geplaatst. Een configuratie is niet nodig.
(Cijfer 8)

YII
De installatie is vanzelfsprekend. Yii moet gedownload worden en in root map van de webserver geplaatst worden. Via een console commando kan een project aangemaakt worden waardoor feitelijk een map met nodige bestanden ingericht wordt.
(Cijfer 8)

Zend Framework
De configuratie in Zend is iets moeilijker, omdat een composer gebruikt wordt, die na de initiële installatie uitgevoerd moet worden. Toch heeft Zend ook een console, waarmee projecten aangemaakt kunnen worden.
(Cijfer 6)

2.3.13 Ontwikkeling (efficiëntie)

Het is positief, als het framework flexibel is. Slecht zou zijn als daarom bijvoorbeeld via een framework veel meer code nodig is om een soortgelijke functionaliteit te ontwikkelen. Ook een console is tegenwoordig belangrijk (zoals in Ruby On Rails). Via een console kunnen herhalende taken automatisch en interactief overgenomen worden. (Aanmaken van projecten, inrichten van de database, aanmaken van models enz.). Toch is het moeilijk, om de efficiëntie heel exact in te schatten, omdat het ene framework deels problemen beter oplost dan het andere – en omgekeerd. Bovendien hebben de aspecten, die invloed hebben op de efficiëntie de configuratie, de leercurve en andere gekozen factoren in dit onderzoek invloed op de efficiëntie. De volgende cijfers zijn daarom deels beïnvloed door andere factoren van dit onderzoek. Vele taken kunnen via console van het framework automatisch uitgevoerd worden, waardoor de efficiëntie van de ontwikkeling stijgt.

CakePHP

CakePHP biedt een goede console, waarmee net zoals in Symfony2 vele taken overgenomen worden, waarbij de Symfony2's console een groot stuk overzichtelijker lijkt en meerdere opties ter beschikking stelt.
(Cijfer 7)

Symfony2

Symfony2 heeft een omvangrijke methode om databases te beheren en models en controllers aan te maken. De console lijkt heel intuïtief een gemakkelijk te bedienen.
(Cijfer 8)

Yii

Yii biedt een minimale console om projecten aan te maken en te beheren.
(Cijfer 6)

Zend Framework

Zend is redelijk moeilijk om ermee te beginnen. Door de iets hogere leercurve en de configuratie, die niet vanzelfsprekend is, kan het lastig worden voor toekomstige gebruiker, om in het begin al snel te ontwikkelen. De andere frameworks lijken allemaal overzichtelijker in elkaar te zitten. Toch biedt Zend een hoog aantal aan basis functionaliteiten, waardoor de efficiëntie weer stijgt.
(Cijfer 7)

2.3.14 Snelheid

De snelheid van de frameworks bleek uiteindelijk heel moeilijk te zijn. Er zijn – indien aanwezig – alleen grove tests beschikbaar. Met name worden hier "Hello World" tests gemaakt, die helemaal niet realistische resultaten kunnen leveren. Bovendien verschillen de resultaten enorm waardoor geen zinnige conclusies getrokken kunnen worden. Ook is de snelheid van een applicatie afhankelijk van de eigen code. Op vele plekken kan caching gebruikt worden en alle frameworks zijn flexibel genoeg om dit soort dingen te integreren. (telkens *cijfer 5*)

2.4 Resultaten en conclusie

Symfony2 is de beste keuze voor Webstores. Het framework is door professionele mensen en voor professionele projecten gemaakt. In het geheel is het framework heel indrukwekkend. Ook op internet wordt vooral positief geschreven over het framework. In de wereld van PHP, waar vele projecten een heel slechte kwaliteit qua code heeft, is Symfony2 als weldoordacht framework een echte rariteit. Negatieve punten van het framework zijn met name de steile leercurve. Door het professioneel niveau moeten vele aspecten geleerd worden voordat er productief ontwikkeld kan worden. Dit was ook het grote voordeel van CakePHP, omdat de meeste medewerkers bij Webstores redelijk goede kennis van de werkwijze hebben. Toch is het grote nadeel van CakePHP de lagere flexibiliteit in vergelijking met Symfony2 of Zend Framework.

CakePHP	Symfony2	YII	Zend Framework
6,33	6,88	4,99	6,65

3 Introductie Symfony2

Het onderzoek was klaar en een framework was gekozen: Symfony2. Symfony2 is een redelijk nieuw framework die van moderne concepten in PHP gebruik maakt en uitstekende features ter beschikking stelt. Om het framework te leren kennen, is het een goede strategie, om het Symfony2 boek te bestuderen (beschikbaar op symfony.com) en via Trial & Error projecten de concepten te begrijpen.

Afhankelijk van het kennisniveau en de ervaring met PHP en frameworks kan het begrijpen van een framework lastig worden. Met name kan het moeilijk worden, de concepten en werkwijzen van het framework te begrijpen. Heeft een ontwikkelaar nooit met code van andere mensen gewerkt, dan kost het accepteren van andere benaderingen bovendien nog extra moeite. Waarom doet Symfony2 bepaalde dingen op de ene manier, hoewel het ook korter kan?

Het wiel moet niet opnieuw uitgevonden worden. Dit geldt vooral als de tijd voor project beperkt is. Symfony2 is een framework met een steile leercurve. Er zijn zeker eenvoudigere frameworks op internet te vinden.

Achteraf was Symfony2 een perfecte keuze. Als de concepten eens duidelijk zijn, is het mogelijk om krachtige applicaties met het systeem te ontwikkelen. Een van de hoofdvoordelen van Symfony2 zijn de flexibiliteit, stabiliteit en overzichtelijke code (coding standards). De standaarddistributie van Symfony2 komt met enkele krachtige componenten, waarvan WM3 deels intensief gebruikt maakt.

Voor WM3 is het belangrijk de concepten volledig te begrijpen om van de voordelen gebruik te maken. Daarom bevat dit verslag een eigen hoofdstuk, welk de belangrijke aspecten van dit framework beschrijft.

3.1 Routing

Symfony2 heeft een eigen routing component. Hierdoor zou de scheiding van verschillende acties in WM3 gemakkelijk zijn. Routing in Symfony2 werkt redelijk simpel. In een configuratie bestand wordt de mapping bepaald. Symfony2 voert op basis van de mapping dan een de goede actie uit.

```
# app/config/routing.yml
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

In het voorbeeld is te herkennen, dat ten eerste een pattern gedefinieerd wordt: (blog/{slug}). Ten tweede wordt een controller gedefinieerd, die uitgevoerd wordt als een URL met dit formaat opgeroepen wordt. Terloops: Configuratie bestanden in Symfony2 kunnen verschillende formaten hebben. In het voorbeeld wordt YAML gebruikt. YAML is heel overzichtelijk en voor dit soort configuraties een goede keuze. Alternatief kunnen de configuraties via XML, PHP of annotations (als PHP commentaren die geïnterpreteerd kunnen worden) bepaald worden.

WM3 maakt gebruik van YAML configuraties (voor routing) en van XML configuraties (DI, Doctrine2)

3.2 Form

Een van de meest uitgebreide complexe onderdelen in Symfony2 is het form component. Het form component in Symfony2 zorgt voor de validatie en afhandeling van request acties. Ook is het form in staat om HTML formulieren te renderen. In de praktijk wordt voor ieder model (bv. Channel, Page, Layout) in de applicatie een bijhorend form type aangemaakt.

```
class PageWidgetType extends AbstractType
{
    //..
    public function buildForm(FormBuilderInterface $builder, array $options)
    {

        $builder
            ->add('position')
            ->add('slotName')
            ->add('description', array(
                'is_required' => false
            ))
            ->add('widget', 'entity', array(
                'class' => $this->widgetClass
            ))
            ->add('page', 'entity', array(
                'class' => $this->pageClass
            ))
        ;

    }
}
```

Een form type definieert voor ieder veld (bv. de velden van Channel: *naam*, *taal* en *URL* in het model bijhorende regels, het verwachte formaat en filter voor dit veld) Symfony zorgt dan voor een correcte validatie van dit model. Deze benadering is flexibel, omdat de Controllers bijvoorbeeld niet meer aangepast moeten worden als velden verwijderd, veranderd of toegevoegd worden. Als dan bovendien de form types als service gedefinieerd worden is een verandering alleen nog een kwestie een passend (service) configuratie.

3.3 Doctrine2

WM3 moet onafhankelijk zijn van een onderliggend data layer. Welk database uiteindelijk gebruikt wordt mag geen rol spelen. Via ORM wordt het paradigma van relationele database vertaalt naar het OO paradigma. Een model $channel representeert dan een rij van de Channel tabel -velden van deze rij zijn variables van het object.

In het begin wordt Doctrine2 als ORM layer voor WM3 gebruikt en nodige managers hiervoor geïmplementeerd. Doctrine2 biedt uitstekende mogelijkheden en is op de hoogte van tabelrelaties. Daardoor wordt het eenvoudig, gerelateerde objecten op te halen of waarden te veranderen. Cascade acties kunnen ook automatisch uitgevoerd worden, waardoor bv. het handmatige verwijderen van gerelateerde objecten niet meer nodig is.

3.4 Twig

Twig is de standaard template engine van Symfony2. Via een template engine kan HTML verplaatst worden. Door Twig is in WM3 geen enkel HTML code binnen een PHP bestand te vinden. WM3 maakt gebruik van Twig om de pagina en hun widgets te tonen. Gelukkig kan Twig gemakkelijk via extensies uitgebreid worden, waardoor Twig in het eindproduct in staat is om slots met widgets re renderen.

3.5 Dependency Injection (DI)

Het DI component in Symfony2 kan worden geïnterpreteerd als de basis van Symfony2. Dit component zorgt voor het correcte aanmaken en samenstellen van alle objecten. Deze samenstellingen zijn via doordachte configuratie strategieën extreem flexibel. De configuratie van de objecten gebeurt via configuratie bestanden in het XML formaat.

Wil een ontwikkelaar een bundle (bijvoorbeeld het WebstoresContentBundle) gebruiken, dan moet het bundle eerste geconfigureerd worden. In het geval van WM3 wordt YAML als formaat gebruikt. Een configuratie class zorgt ervoor, dat alle benodigde configuraties gemaakt worden.

Ten tweede heeft ieder bundle een "extension" object, die in staat is deze configuraties te interpreteren. De parameter $configs is hierbij de bovenstaande configuratie. In dit methode wordt Symfony2's container geconfigureerd en parameters geregistreerd. De container stelt alle services parameters ter beschikking.

```
webstores_content:
  models:
    ContentType:
      class: 'Devpunk\AppBundle\Entity\ContentType'
    ContentField:
      class: 'Devpunk\AppBundle\Entity\ContentField'
    ContentFieldType:
      class: 'Devpunk\AppBundle\Entity\ContentFieldType'
    Content:
      class: 'Devpunk\AppBundle\Entity\Content'
    ContentData:
      class: 'Devpunk\AppBundle\Entity\ContentData'
  FieldDefinition:
    manager: 'Webstores\Bundle\ContentBundle\Content\Definition\Manager\Fie
    class: 'Webstores\Bundle\ContentBundle\Content\Definition\Type\FieldDef
```

Hoewel alle services ook in een extension aangemaakt kunnen worden, is het toch gebruikelijk hiervoor een aparte configuratie bestand te gebruiken. Symfony2 analyseert dit bestand ***nadat*** de extensie werd uitgevoerd. Mogelijke parameters zijn dus al bekend. Daadwerkelijk worden hier classes aan een service id gekoppeld en de parameters van hun constructor gedefinieerd. Deze constructor parameters kunnen wederom services zijn waardoor een complex objectstructuur samengesteld kan worden.

```
<?php

class WebstoresContentExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {

    }

}
```

Als Symfony2 klaar is met de analyse, kan de applicatie uitgevoerd worden. De service container is hierbij zelf een service, waardoor andere services gebruik kunnen maken van het object en hierdoor in staat zijn handmatig services aan te vragen. Vaak maken de controllers in WM3 gebruik van dit object, om hierdoor diverse services op te halen (zonder "bad practice" `new` keywords te gebruiken).

```
public function deleteWidgetAction($id)
{
    $wm = $this->container->get('webstores.widgetbundle.widget.manager');
    $wm->removeWidget($wm->findById($id));
}
```

```
<!--contentfieldtype-->
<service id="webstores.contentbundle.contentfieldtype.form" factory-method="create" f
    <argument type="service" id="webstores.contentbundle.contentfieldtype.formtype" /
</service>
<service id="webstores.contentbundle.contentfieldtype.formhandler" class="%webstores.
    <argument type="service" id="request" />
    <argument type="service" id="webstores.contentbundle.contentfieldtype.manager" />
    <argument type="service" id="webstores.contentbundle.contentfieldtype.form" />
</service>
<service id="webstores.contentbundle.contentfieldtype.formtype" class="%webstores.con
    <argument>%webstores.contentbundle.contentfieldtype.entity_class%</argument>
</service>
```

4 Realisatie

4.1 Projectdefinitie

Aan het begin van het afstudeerproject moest het project goed gedefinieerd worden. Regelmatige terugkoppelingen met de begeleider zijn daarom belangrijk om een goed beeld van de eisen van de doelen van hat CMS te krijgen. Het is zinloos, met de ontwikkeling van het systeem te beginnen, als nog niet alles duidelijk gedefinieerd is. Webstores had al een relatief concreet idee van het project, eigen interpretaties en begrippen zoals channels, widgets en layouts. Deze interpretaties gedetailleerd te leren kennen kostte tijd. Om een goed idee van alle eisen te krijgen en puzzelstukjes te sorteren kostte ongeveer 3 weken tijd.

4.2 Planning

Om alles overzichtelijk te houden is WM3 in vier delen onderverdeeld. Alle delen moeten zo onafhankelijk mogelijk van elkaar kunnen werken. Omdat een concrete implementatie nog niet bestond was dit meer een grove scheiding. Later bleek, dat deze scheiding inderdaad zinvol was.

4.3 Het datamodel

De basis van het CMS was een passend datamodel. Het datamodel beschrijft in principe het hele systeem en alle relaties. De eerste logische stap na de projectdefinitie was daarom het maken van een geschikt datamodel. Door de gespreken met de begeleider was de doelstelling duidelijk geworden en het ontwerpen van een eerste versie van het datamodel ging daarom heel snel. De versie was op zich in orde. Toch was het nog niet helemaal correct omdat er al aspecten in het datamodel stonden, die nog niet van belang waren. Daardoor werd het datamodel te complex voor de basis. De volgende versie was een stukje eenvoudiger.

4.4 Strategie & werkwijze

Het was vooraf al duidelijk, dat de CoreBundle (met name de channels) het makkelijkste onderdeel van het hele systeem zou worden. De channel implementatie is op zich een eenvoudige CRUD implementatie. Dit betekent, dat er Channels opgeslagen, verwijdert, veranderd kunnen worden. Voor een Symfony2 beginner is het toch niet zo triviaal dan het lijkt. Symfony2 heeft uitdagende features, maar de leercurve is redelijk stijl en door de hoge eisen aan flexibiliteit van WM3 moet het framework eerst goed begrepen worden. Tegelijkertijd is het een goede strategie om bijhorende Symfony libraries her te gebruiken omdat vele libraries een hoge kwaliteit hebben.

Er is altijd het gevaar, dat er vaak voorkomende dingen opnieuw worden ontwikkeld terwijl er al goede oplossingen bestaan. Vooraf de documentatie en Symfony's code intensief en herhalend te bestuderen is daarom heel zinvol. Dit kost veel tijd. Toch is het gevaar, code te ontwikkelen terwijl het framework voor het ene probleem misschien al een oplossing ter beschikking stelt heel hoog. Uiteindelijk is de tijd daarom goed geïnvesteerd.

De documentatie van Symfony2 is tegenwoordig rudimentair. Dit betekent: Voor vaak voorkomende scenario's en algemene taken levert de documentatie genoeg informatie voor een ontwikkelaar. Aan de andere kant is het nodig, voor specifiekere dingen de broncode actief de te bestuderen.

De beginfase van de ontwikkeling was moeilijk. De eisen van WM3 waren bekend en de hulpmiddelen van Symfony2 ook duidelijk. Een concreet software design te ontwerpen kostte desondanks vooral aan het begin veel moeite. Een slechte beslissing in deze fase is moeilijker te herkennen (door ontbrekende cases) en kan in de toekomst nadelig zijn. Hier is de verzamelde ervaring van eerdere projecten heel behulpzaam. Slechte keuzes in het design maken, waardoor dan later tijdintensief code herschreven moet worden levert heel veel inzichten van goed en slecht design op. Het tweede probleem in deze fase was de toepassing van Symfony's concepten. Het begrijpen van de Symfony2 concepten is één probleem. Deze nieuwe tools creatief en efficiënt te gebruiken is een compleet andere verhaal. Op internet zijn bijvoorbeeld vele "Symfony2" projecten te vinden. Ze werken en zijn functioneel. Maar door de code te bestuderen blijkt snel, dat van de krachtige hulpmiddelen van het framework niet gebruik gemaakt wordt.

De structuur en syntax van het framework (de methoden, de classes, de configuraties enz.) te begrijpen is daarom niet voldoende. De voordelen van goede benaderingen te realiseren kost veel meer tijd. Dit zijn vaardigheden, die eerst moeten groeien. Achteraf lijkt een softwaredesign misschien duidelijk en triviaal. De code is gemakkelijk te lezen. Niet heel triviaal zijn de keuzes, gedachten en beslissingen, die tot de uiteindelijk code leiden.

Gelukkig bieden web application frameworks zoals Symfony2 naast hun bijhorende componenten ook een aanbevolen basisstructuur. Ondanks de flexibele natuur door Dependency Injection wordt men (gelukkig) toch gedwongen om een bepaald e structuur aan te houden. Dit is in feite geen nadeel maar behulpzaam. Code van andere ontwikkelaars te bestuderen en te begrijpen wordt door een logische structuur eenvoudiger.

4.5 Algemeen ontwerp

Geen new keywords

Objecten die direct in een class aangemaakt worden verhogen automatisch de afhankelijkheid van het systeem. Ze staan vast in een class gedefinieerd en zijn niet vervangbaar. In WM3 wordt helemaal geen gebruik van het new keyword gemaakt. Bovendien is een class nooit afhankelijk van een specifieke implementatie. **Class A** is dus nooit afhankelijk van een **Class B** – maar ten hoogste afhankelijk van een **Interface B.** Het daadwerkelijke aanmaken en "lijmen" van de objecten wordt door de Dependency Injection afgehandeld.

Door deze strikte regeling blijven alle classes in WM3 vervangbaar.

PSR-2 (Coding standard)

Bij Webstores werken minimaal een tiental mensen met elkaar in de backend afdeling. Broncode zal bestudeerd en begrepen moeten worden. Volgt iedereen zijn eigen coding standard, dan wordt het lastiger om code van een collega te begrijpen.

De complete broncode werd dus volgens de coding standard PSR-2 geformatteerd. Deze standaard bepaalt regels voor het opmaken van broncode (CamelCase variables, lege rijen). Feitelijk mag de

coding style van de mensen niet te onderscheiden zijn. Ook op implementatie niveau van WM3 worden bepaalde best practices aangehouden.[1]

REST in WM3

Binnen het kader van het project moet een REST API voor WM3 geschreven worden. Principieel is REST een gestandaardiseerde routing om bepaalde acties uit te voeren. Uiteindelijk worden toch weer een specifiek functie in een controller uitgevoerd. Binnen een controller wordt dan het request afgehandeld. Nieuwe records worden hier opgeslagen, getoond, verwijderd of veranderd.

Principieel mag in een RESTful applicatie per request alleen één resource benaderd worden.

MVC in WM3

Voor bijna alle resources in WM3 (channels, records, layouts) worden in WM3 dus aparte controllers gedefinieerd. Ook is het belangrijk om te weten, dat iedere resource zo veel mogelijk dezelfde strikte OO structuur gebruikt, waardoor de leesbaarheid van de code verbeterd wordt.

WM3 gebruikt een variant van het MVC pattern. Deze scheiding was een fundamenteel eis van WM3. Toch is het nog niet flexibel genoeg. Het traditionele concept van MVC is redelijk oud en houdt weinig rekening met web gerelateerde problemen zoals formulieren en requests. Het volgende scenario beschrijft een mogelijk probleem (of tenminste grenzen qua flexibiliteit):

Situatie: Er is een Channel controller, die ervoor zorgt, dat een channel via REST API beheerd kan worden (CRUD acties). Actueel ziet de channel tabel als volgt uit:

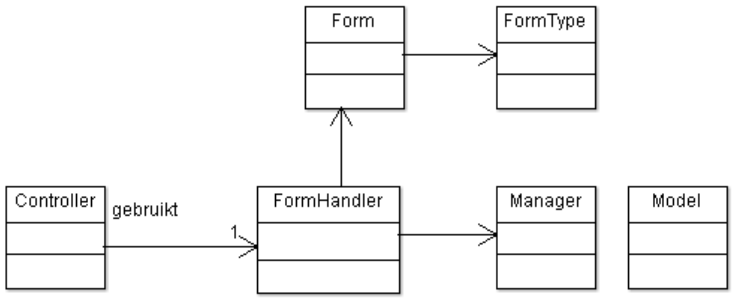
wm3_johannes.channel: 2 rows total (approximately)

id	type	uri	language
1	1	http://www.example.com	nl_NL
2	2	http://www.example.de	de_DE

De tabel Channel heeft dus een **id**, **type**, **uri** en een **language** veld. Het probleem is, dat in toekomst de tabel kan (en zal) uitgebreid worden. Dit zou als gevolg hebben, dat niet alleen de models gewijzigd moeten worden, maar ook de controllers, die met name verantwoordelijk zijn voor het afhandelen van form requests. Ook is er een grote kans, dat het template met de formulieren uitgebreid moet worden.

[1] Een engelse beschrijving is onder https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md te vinden

Dit was geen optie voor WM3. Inputgegevens kunnen in toekomst veranderen. Gelukkig was op internet een oplossing voor dit soort probleem te vinden. Met name was de FOSUserBundle een goede inspiratie. Hier wordt de verwerking van formulieren naar aparte objecten (form handlers) verplaatst. Dit betekent: De controllers zijn op die manier onafhankelijk van toekomstige wijzigingen. Een form handler kan ervoor zorgen, dat een passend formulier voor een model (in dit voorbeeld voor Channel) gebruikt wordt en zorgt voor de correcte afhandeling van requests. Het bijhorende object – de Channel form type – definieert mogelijke velden van het formulier en is de tegenpartij van het Channel model.

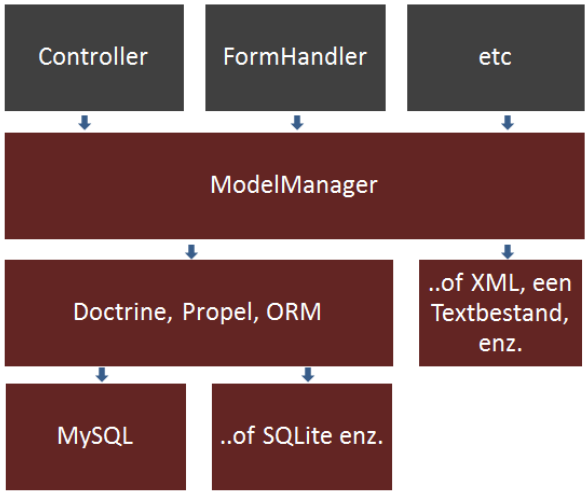


Grove illustratie van Controller/Form/Model relatie

Form Type: Form Types definiëren de velden, die nodig zijn voor een bepaalde entiteit (zoals Channel)

Form Handler: Form Handlers verwerken het formulier. Een form handler heeft kennis van een **manager** en kan een entiteit *laten* opslaan. Bovendien gebruikt iedere form handler het request object van Symfony2 (injected via Dependency Injection). Daardoor worden bijvoorbeeld ook unit tests gemakkelijker, omdat bij een POST request vervangen kan worden door een Fake object.

Manager (Startpunt model): Is een formulier geldig, dan kan een form handler een entiteit opslaan via een manager. Principieel kunnen instanties aan de controller kant alleen contact met de model kant via een **manager** opnemen. De concrete interne implementatie van de database speelt dan geen rol meer.



Communicatie met de database vindt uitsluitend via een model managers plaats

Requests, form types, managers, models, controllers en form handlers zijn allemaal vervangbaar. Verandert de database, moet bv. alleen de manager veranderd worden. Worden velden toegevoegd/veranderd, moet bv. alleen de form type en het model vervangen worden. Komen er controllers bij of worden veranderd, heeft

dit geen invloed meer op andere onderdelen. Deze manier wordt globaal voor alle resources aangehouden. In combinatie met de Dependency Injection blijft WM3 hierdoor extreem flexibel.

4.6 CoreBundle

De taak van het corebundle is het beheer van de channels en layouts.

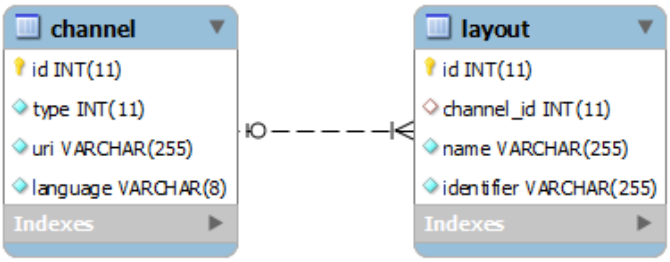
Channels

Channels in WM3 kunnen gezien worden als instanties van (delen van) webapplicaties Bijvoorbeeld wil een klant graag een webshop en een blog hebben. In WM3 worden deze dingen als semantische, logische onderdelen van een webapplicaties geïnterpreteerd – als channels. Het hoofdvoordeel is een intuïtieve structuur voor een beheerder.

Zonder channels zou er voor een klant voor iedere website een nieuwe instantie van WM3 op een server ingericht moeten worden.

Via het Channel concept is dit niet meer nodig. Een klant kan verschillende webpagina's via een WM3 instantie beheren (gebruikersvriendelijker). Ook moet hierdoor WM3 alleen één keer per klant op een server ingericht worden.

In de basisversie van WM3 kan per channel een taal, een uri en een type opgegeven worden. Ook heeft ieder channel een vast aantal van layouts over die een beheerder kan beschikken.



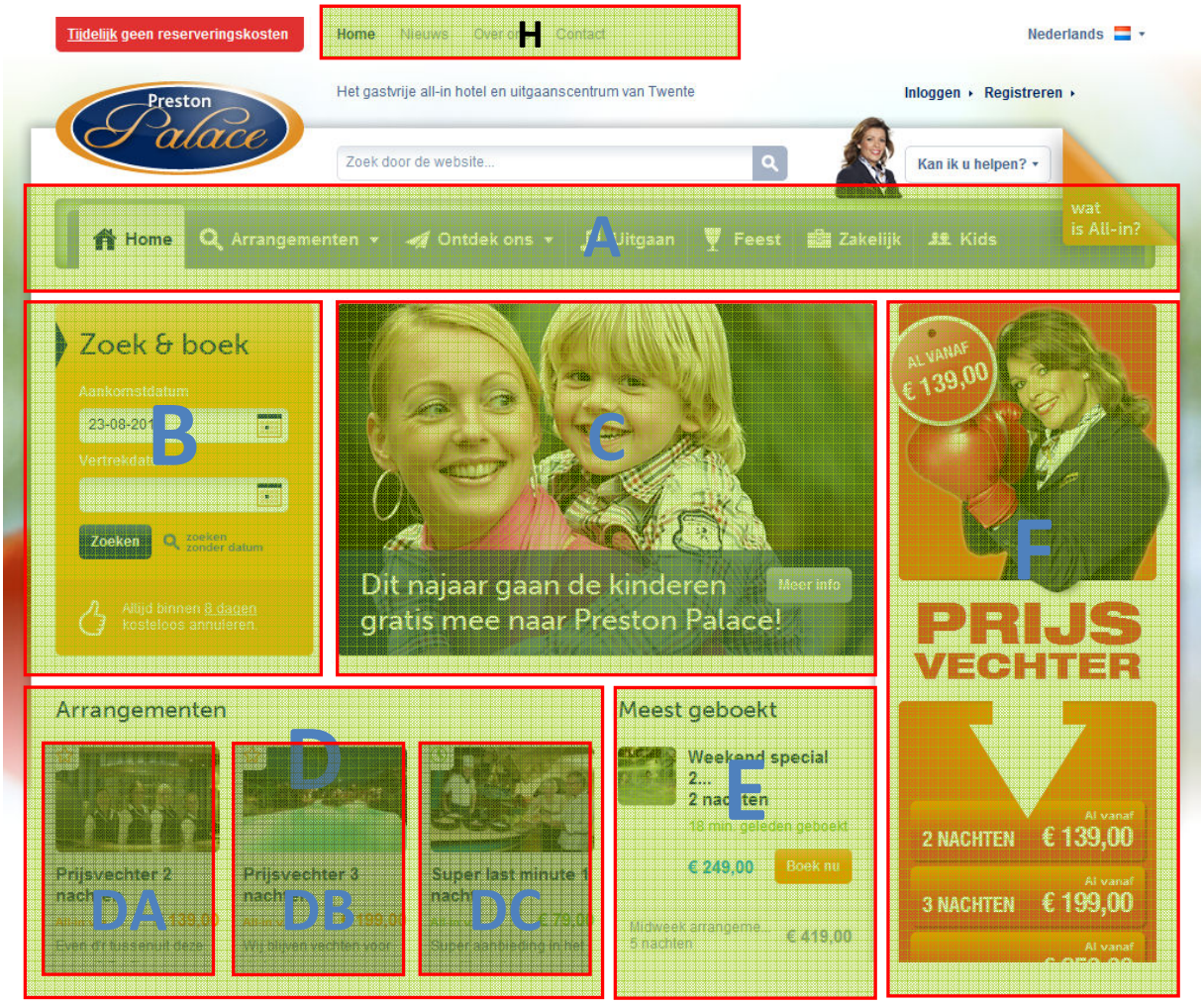
www.example.com/en/blog)

Benadering

Het datamodel van Channel is heel triviaal. Het is een simpele tabel met standaard velden. Hierdoor was de implementatie heel simpel en komt exact met de eerder beschreven structuur (hoofdstuk Algemeen ontwerp) overeen.

Layouts

Goede CM systemen bieden een sitebeheerder de mogelijkheid om de pagina's zelf samen te stellen. Een blog pagina zou bijvoorbeeld meerdere functionaliteiten op een pagina hebben: Een artikel in het hoofdgebied van een layout, een lijst met categorieën op de rechte kant en een navigatie aan de bovenkant van de pagina. Dit is een van de flexibele eigenschappen, die een CMS attractief maken. De uitdaging is een systeem te bouwen, waarmee een beheerder zonder programmeerkennis in staat is om complexe pagina's in elkaar te zetten.



Een layout is daadwerkelijk een tekstbestand met plaatshouders. Dit betekent voor HTML bestanden, dat de plaatshouder in feite gebieden (slots) in een template vertegenwoordigen. In deze slots kan willekeurige content geplaatst worden. Ook moeten instellingen voor slots gedefinieerd kunnen worden. Mogelijke configuraties kunnen in toekomst veranderen. Tenslotte kunnen slots andere slots bevatten, waardoor een boomstructuur ontstaat.

Benadering

Zoals voor het hele CMS worden in toekomst hoge eisen aan het layout systeem gesteld worden. Alles moet uitbreidbaar en configureerbaar blijven – zonder de basis aan te passen. Maar voor alle

mogelijke scenario's al een geïmplementeerde oplossing ter beschikking te hebben is bijna onmogelijk. Op dit moment was niet eens duidelijk, hoe het concept widgets geïmplementeerd moest worden. Hoe kan een systeem dus flexibel ontworpen worden zonder een concreet plan? Het antwoord was abstractie.

Is nog niet bekend hoe in toekomst iets geïmplementeerd moet worden, schrijf dan gewoon een zinvolle (en minimale) interface in plaats van een implementatie.

In verband met layouts, slots en widgets betekende dit: 1) Dat in ieder geval als bekend was, dat er slots in een layout waren. 2) Dat content in de slots moesten getoond worden. En 3) Dat widgets iets waren wat in deze slots getoond moesten worden.

Deze benadering had ook het leuke neveneffect, dat de CoreBundle hierdoor onafhankelijk was van overige componenten. De CoreBundle kon op deze manier stand alone gebruikt worden.

Het tweede probleem was de dynamische instellingsopties van slots. Een eerste naïeve oplossing: Extra velden voor instellingen in de slot tabel in de database opslaan. Als de eisen dan in toekomst veranderen, kan snel het database schema aangepast worden. Toch frequente veranderingen aan het database schema waren geen optie. De configuratie geserialiseerd op te slaan was al een betere oplossing. Toch heeft deze benadering ook een groot nadeel. Het formaat van de instellingen kan niet door de database engine efficiënt gevalideerd worden. Er moest een regelwerk komen om mogelijke configuratiestructuren te valideren.

De validatie van configuraties is een probleem, welke zich in de core van Symfony2 ook heel vaak voordoet. Symfony2 maak uitgebreid gebruik van configuratie bestanden, die ergens op het schijf opgeslagen worden. Met name worden deze configuratie bestanden ook heel vaak door andere configuraties overschreven (hierdoor blijft Symfony2 flexibel). Symfony2 lost dit soort problemen met een zogenoemd treebuilder component op. Treebuilders in Symfony2 definiëren de structuur van instellingen. In tegenstelling tot XML en XSD bestanden, die ook geschikt zijn voor dit soort validatie, biedt de TreeBuilder de uitgebreide mogelijkheid om regels voor het overschrijven van waarden te definiëren. Aan de rechterkant een illustratie van het probleem:

- Het eerste configuratiebestand definieert een configuratie voor de node **acme_hello**.
- Vervolgens een tweede bestand die de waarde probeert te overschrijven.

```
acme_hello:
    foo: fooValue
    bar: barValue
```

```
acme_hello:
    foo: OVERSCHRIJVEN MAG
    bar: OVERSCHRIJVEN MAG NIET
    meerconfig: MAG NIET
```

Om problemen en chaos ter vermijden worden voor nodes zoals **acme_hello** deze treebuilders gebruikt. Hier wordt de benodigde structuur bepaald en regels voor het overschrijven van waarden gedefinieerd. Bijvoorbeeld kan het overschrijven van waarden (zoals **bar**) expliciet verboden worden.

Dit was een heel elegante oplossing en kon ook voor de slots toegepast worden. De finale versie gebruikt dus een veld **configuratie** in de tabel **slots**. De validatie van deze configuraties wordt via een treebuilder bepaald.

4.7 WidgetBundle

De taak van de WidgetBundle is het beheren van widget gerelateerde aspecten van het systeem. Deze bundle is omvangrijker dan de CoreBundle.

Widgets

Widgets in WM3 kunnen geïnterpreteerd worden als subapplicaties binnen een applicatie. Widgets kunnen eigen logica hebben of alleen als een weergave van iets dienen. Een van de belangrijkste eisen van widgets is een mogelijke configuratie. Widget moeten aanpasbaar zijn vanuit het CMS. Door configuraties moeten widgets hergebruikbaar blijven. Is een widget eenmaal klaar, dan moet het toepasbaar blijven voor verschillende scenario's. Widgets moeten dus zo generiek mogelijk zijn. Ontwikkelaars dienen zo weinig mogelijk van de code te wijzigen. Het doel is duidelijk: Ontwikkelingstijd en uiteindelijk geld besparen.

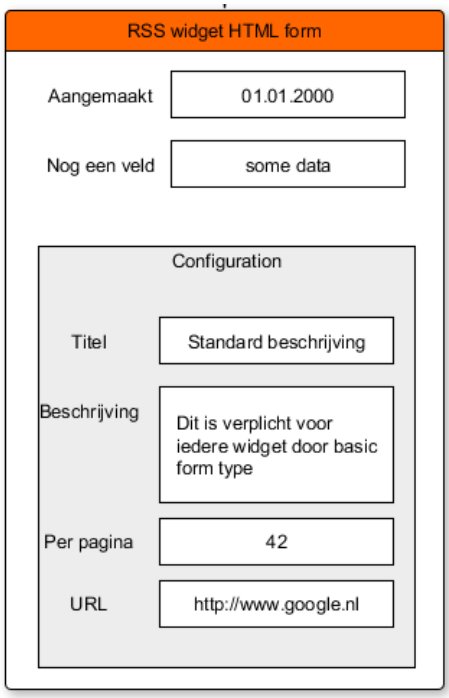
Een passend voorbeeld voor een widget zou een widget zijn die RSS inhoud op een pagina kan tonen. Een inflexibel rss widget zou statische velden in de broncode hebben voor een titel, een URL en het aantal artikelen per pagina. Dit zou een voorbeeld zijn van een inflexibele widget. Beter is het, deze widget toepasbaar te maken. Het doel is, een gebruiker in een CMS de mogelijkheid te bieden, om een widget zelf te configureren via een formulier.

Benadering

CRUD

Het eerste aspect gaat om het opslaan en beheren van widgets via de REST interface. Widgets moeten op de een of andere manier gewijzigd worden.

De WidgetBundle is een van de complexere componenten in WM3. Principieel volgt deze bundle ook weer het algemeen ontwerp. Dit betekent: Widgets worden via REST aangemaakt, aangepast en verwijderd (CRUD acties). Toch is het niet zo triviaal als het lijkt. Widgets kunnen van alles zijn en zullen allemaal verschillende instellingen hebben, die via de REST API beheert moeten kunnen worden. Tegelijkertijd zouden de widgets ook velden hebben, die voor alle widgets nodig zijn. Tenslotte zijn de instellingen van ieder widget een combinatie van specifieke eigenschappen + algemene eigenschappen.



In het algemene concept wordt een vast form/form type gebruikt om een form voor een specifiek model te genereren (bijvoorbeeld channel, layout, slot). Daarom moest het ontwerp voor deze bundle een beetje aangepast worden. In vergelijking met de corebundle zijn de velden en hierdoor de formulieren voor widgets altijd dynamisch. Dit betekent helaas ook, dat de widget form handler geen vast form type kon gebruiken.

Een geschikte oplossing waren zogenoemde form managers voor widgets. Een form manager in WM3 is een object, die in staat is om formulieren op basis van een parameter te genereren. De interface van een form manager voor widgets is heel schoon:

```
    public function getForm(WidgetType
$widgetType);
```

In plaats van vaste form types aan een form handler te koppelen was het een beter idee, een form manager aan de widget form te koppelen die instaat is om dynamisch formulieren aan te maken. Feitelijk veranderde er niet veel aan het algemene ontwerp en de interface van de form moest alleen minimaal uitgebreid worden. Dit was dus een heel elegante oplossing.

WidgetServices

Het tweede aspect van de WidgetBundle zijn de widgets zelf. Dit zijn klassen die daadwerkelijk de functionaliteit van een widget uitvoeren: Widgetservices.

Waarschijnlijk worden in toekomst intensief gebruik gemaakt van widgets, wat betekent, dat hier heel goed over een oplossing nagedacht moest worden. Dit is een belangrijk deel van het systeem, dat ook heel sterk door de backend ontwikkelaars gebruikt zal worden. Daarom moet het ook heel gemakkelijk zijn om nieuwe widgets voor het systeem te implementeren.

De interface voor een widgetservice is daarom ook heel schoon:

```
public function execute(WidgetModel $widget, Environment $env);
```

Widget ontwikkelaars moeten ten minste deze interface implementeren. Het eerste argument zou een widget model uit de database zijn. In dit model zijn ook de instellingen opgeslagen (bijvoorbeeld URL of titel bij een RSS Widget). Het tweede argument is een object, die de omgeving beschrijft waarin het widget getoond wordt. Een kwalitatief hoogwaardig widget kan theoretisch met behulp van deze informatie heel flexibel reageren.

Wil de ontwikkelaar een goede widget schrijven, die configureerbaar is, dan kan hij de optionele tweede methode implementeren:

```
public function getFormType();
```

Via deze methode wordt het passende formulier voor de configuratie geretourneerd, waarvan de form (eerder beschreven) gebruik maakt.

Problemen

In een vroeg concept probeerde ik het probleem via een treebuilder op te lossen, die ik ook eerder voor layouts en slots voor dynamische configuraties had gebruikt.

Het bleek, dat treebuilder voor dit soort probleem niet geschikt was. In vergelijking met slots moeten instellingen van widgets via een CMS gewijzigd kunnen worden. Configuraties, die via een treebuilder gedefinieerd worden, voor een eindgebruiker in een CMS configureerbaar te maken zou heel complex zijn. TreeBuilders kunnen heel goed overschrijfbare configuratiestructuren in lezen van bestanden.

Dit betekende: Een stap terug. Ik moest een andere oplossing vinden, die beter geschikt was. Met het form component van Symfony2 had ik een perfecte oplossing gevonden. Configuraties van widgets konden ook prima via formulieren vertegenwoordigd worden. Het form component biedt talloze opties en tools voor het aanmaken van formulieren.

4.8 PageBundle

De PageBundle is verantwoordelijk voor het tonen van widgets in een layout. Het kan gezien worden als koppeling van de CoreBundle (channels, layouts) en de WidgetBundle.

Benadering

CRUD

De REST implementatie voor het PageBundle volgde weer het algemene ontwerp en was gemakkelijk binnen uren om te zetten.

Implementatie van interfaces

Via de CoreBundle en de WidgetBundle waren nu alle belangrijke basisfunctionaliteiten gerealiseerd:

- Layouts met slots konden getoond worden.
- Geconfigureerde widgets konden getoond worden

Wat ontbrak, was een link van beide bundles. Het was alleen nog nodig, gebruik van de interfaces te maken en concrete implementaties te schrijven.

Met name werden nu de positieve effecten van interfaces duidelijk. In het CoreBundle hoofdstuk werd beschreven, dat een goede interface de beste oplossing is, als de toepassing nog niet bekend is. Nu was een punt bereikt, waarop deze interfaces geïmplementeerd moesten worden. Snel was de SlotRenderer geïmplementeerd welke in de CoreBundle bepaald is om inhoud in slots te renderen. De SlotRenderer instantie haalt goede widget models uit de database, maakt widgetservice instanties aan en retourneert de inhoud hiervan, die tenslotte in een slot terecht komt.

4.9 ContentBundle

De laatste bundle is de ContentBundle en moet een mogelijkheid bieden om nieuwe content types aan te maken.

Klassieke content types zijn bijvoorbeeld blogartikelen, gastenboekberichten of producten. In de webontwikkeling worden dit soort content types meestal opgeslagen in een relationele database. Hiervoor moet een tabel aangemaakt worden en ook bijhorende controllers en views geschreven worden.

Is het datamodel geschreven en de code klaar, dan kost het tijd om kleine veranderingen aan de code te maken. Draait de pagina al live op een server, dan zijn veranderingen van een database schema bij voorkeur geen optie meer.

Webstores wil graag een mogelijkheid hebben, content types dynamisch aan te maken via het CMS zonder fysiek nieuwe tabellen aan te moeten maken.

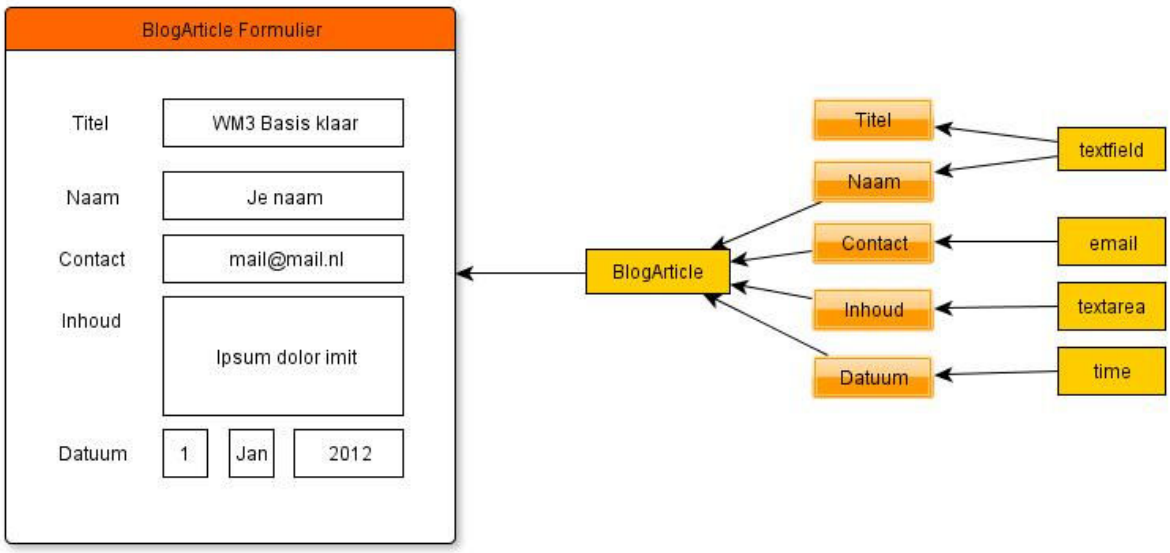
Benadering

Van buitenaf bekeken lijkt dit component misschien eenvoudig te zijn. Toch is deze bundle een van de meest complexe onderdelen van het hele systeem. Duidelijk wordt dit als men over de CRUD acties nadenkt: Het moet mogelijk zijn, via een generieke controller alle mogelijke dynamische content types te beheren zonder een enkel rij code te schrijven.

Dit heeft de volgende consequenties:

- Er moeten formulieren voor alle mogelijke dynamisch content types gegenereerd kunnen worden.
- Ten tweede moeten de content types gevalideerd kunnen worden.
- Verder kan er een willekeurige aantal aan verschillende veldtypen zijn (text velden, text area's, radiobuttons en checkboxes, om er maar een paar te noemen).
- Deze velden kunnen verschillende eigenschappen hebben, die bovendien nog configureerbaar moeten blijven (minimale lengte, benodigde waarde, een passend titel in een formulier).
- Nieuwe veldtypen moeten gemakkelijk ontwikkeld en toegevoegd kunnen worden (geboortedatum, bestanden)
- Tenslotte moet de user interface voor het opgeslagen content type transparant blijven, wat betekent dat de (CRUD) acties voor dynamisch content types niet mag verschillen in vergelijking met andere models van het systeem.

Het probleem begint al bij het bij het datamodel. Relationele databases zijn van aard al niet geschikt voor dynamische tabellen (wat content types feitelijk zijn) en de kunst was het de relationele database te gebruiken en gelijktijdig een transparant en eenvoudige interface voor een ontwikkelaar ter beschikking te stellen, dat op een vast content type lijkt. Ook moet er rekening gehouden worden met de configureerbare eigenschappen van de velden. Een simpel veldtype definiëren zoals dit in een database schema gebeurt (varchar, text, int enz.) is hier niet voldoende. Omdat van contenttypes ook formulieren gegenereerd moeten worden, moet er de mogelijkheid zijn, om concrete eigenschappen voor velden te bepalen: Bijvoorbeeld is een veld optioneel of mag een minimale lengte hebben of een nummer moet binnen een bepaald bereik liggen. Trefwoord validatie. Voor crud acties mag geen enkele code gewijzigd of toegevoegd worden.



Form Types

In het basisdesign wordt sterk gebruik gemaakt van form types om formulieren te genereren. Hoewel dit redelijk goed gescheiden is (Controllers, form handlers, models enz.) zijn veranderingen in de code nodig, als bijvoorbeeld een nieuw veld aan een model toegevoegd wordt. Verandert het model (bijvoorbeeld krijgt de tabel **channels** en nieuw veld **createdAt**) dan moet parallel ook het form type aangepast worden. Feitelijk zijn dit dus statische form types.

```
class ChannelFormType
{
      public function buildForm(FormBuilderInterface $builder)
      {
            $builder->add('language', 'text', array('required' =>
            true);

            $builder->add('uri', 'text');

            $builder->add('updatedAt', 'time'); //moet veranderd
            //worden
      }
}
```

Het meest zinvolle idee is daarom, van een statisch form type een dynamische form type te maken. Op basis van reeds aangemaakte content types moet deze dynamische form type class in staat zijn, het gewenste formulier te bouwen. Hierdoor heeft dit content type minimaal onderstaande gegevens nodig:

- Content type: welk type moet geconfigureerd worden?
- Contentvelden: Welke velden zijn gedefinieerd voor dit content type?
- Configuratie van contentvelden: Wat voor eisen gelden voor deze velden (validatie)
- ContentVeld typen: Wat voor basis types zijn gedefinieerd voor ieder contentveld?

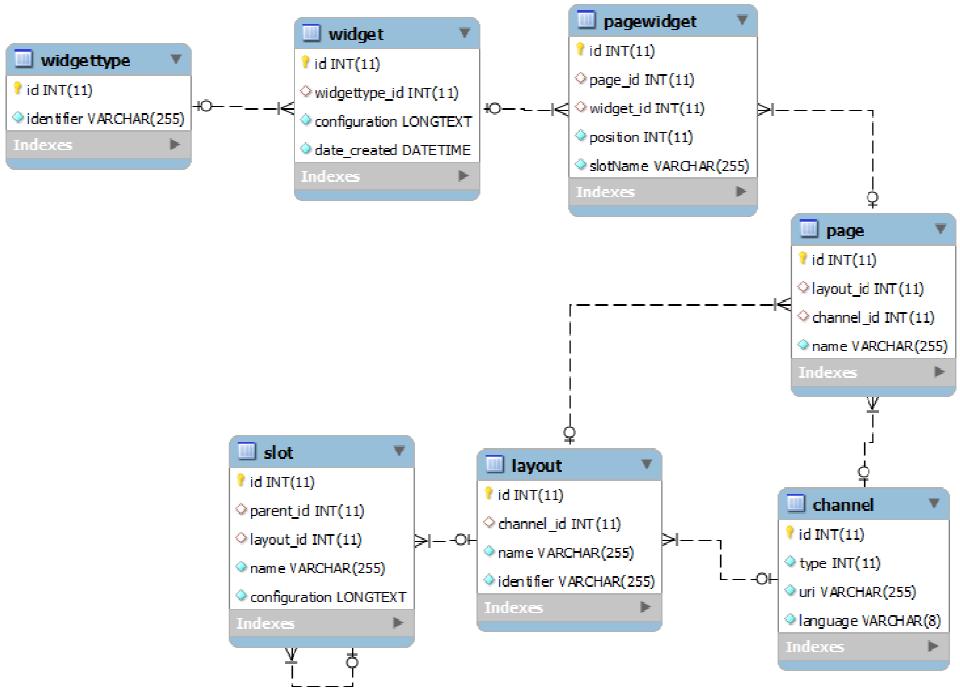
Een dynamische form type lijkt dus een goede strategie te zijn. In samenhang met het algemene ontwerp betekent dit, dat de interface slechts minimaal veranderd wordt ten opzichte van statische form types: Dit form type moet alleen weten, welk dynamisch content type gegenereerd moet worden.

5 Resultaten

Door de abstracte natuur van dit afstudeerproject kan het eindresultaat bijvoorbeeld niet gemakkelijk via screenshots en andere illustraties uitgelegd worden. Een user interface schrijven was geen eis. Het complete project omvat alleen de broncode en de documentatie, die deze code beschrijft. De meeste tijd moest eraan besteed worden het softwaredesign te ontwikkelen en te optimaliseren waardoor de som van alle gedachten tenslotte een softwaredesign + implementatie op een professioneel niveau opleverden, welk aan de hoge eisen kan voldoen.

5.1 Datamodel

Uitwerken van het datamodel voor het basissyteem (genormaliseerde database)



Datamodel CoreBundle:

Kanalen (channel) in WM3 worden in de tabel channel opgeslagen. Ieder **kanaal** heeft een eigen **URL**, een **taal** en een **type** (bijvoorbeeld Webhop, Blog, Mobile). Aan een **kanaal** zijn **layouts** gekoppeld, die voor dit **kanaal** ter beschikking staan. Ieder **layout** heeft wederom één of meer **slots**, die hiërarchisch opgeslagen worden.

Datamodel WidgetBundle

Mogelijke **widget types** worden in de tabel widgettype opgeslagen. Dit zijn widgets, die in het systeem beschikbaar zijn. Door deze tabel en een **identifier** weet het systeem, waar de class te vinden is, die het widget daadwerkelijk uitvoert. Widget instanties worden in de **widget** tabel opgeslagen. Ieder **widget** heeft een **widget type** (bijvoorbeeld is de type RssWidget). Hier wordt ook de configuratie voor de widget instantie opgeslagen. **Widgets** zijn dus instanties van een **WidgetType**.

DataModel PageBundle

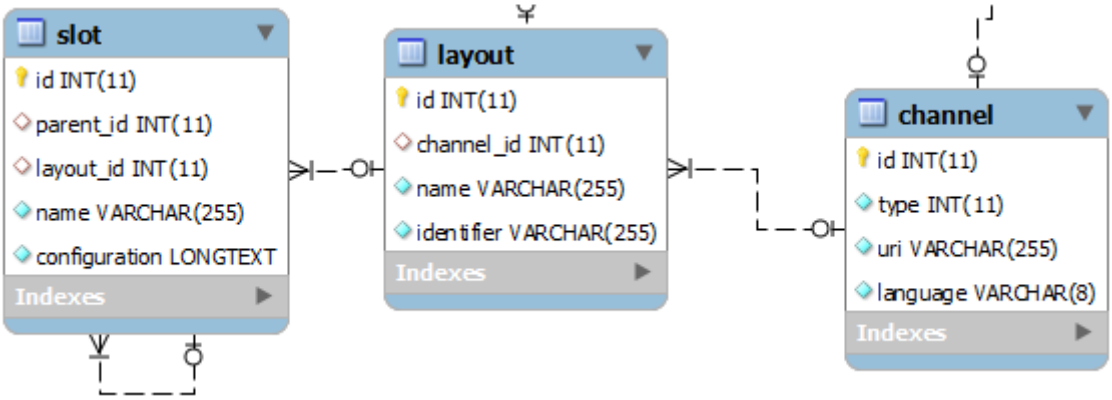
Iedere **pagina** (page) in WM3 vertegenwoordigt een enkel website pagina, die bijvoorbeeld aan een browser gestuurd kan worden. Elk **page** moet daarom ook een **layout** en een **kanaal** hebben. Door

de **pagewidget** tabel worden widgets aan een pagina gekoppeld. Meer informatie is niet nodig om een pagina aan een browser te sturen.

5.2 Flexibel layout systeem

Eis: Elk kanaal moet voorzien kunnen worden van één of meerdere layouts. Elke layout bestaat uit één of meerdere gebieden waarbinnen content geplaatst kan worden. Content wordt per gebied getoond d.m.v. widgets. Per gebied moet dan ook configureerbaar zijn welke widgets geplaatst kunnen worden. De volgorde van de widgets moet bepaald kunnen worden. De configuratie van de layouts moet in te toekomst uitbreidbaar zijn met nieuwe eigenschappen zoals bijvoorbeeld rechten. Layouts moeten overerfbaar zijn waarbij de child alle eigenschappen van de parent overneemt.

Resultaat: Op datamodel niveau heeft de entiteit Channel een vaste relatie met de layout tabel:



In de Slot entiteit is te herkennen, dat er een veld Configuration is waarin de configuraties van de slots opgeslagen worden. De overerving van deze configuratie wordt binnen het systeem via het treebuilder component overgenomen, waarin overigens van het Dependency Injection component ook sterk gebruik wordt gemaakt om overerfbare configuraties te beheren. Mogelijke configuraties kunnen in toekomst zelf bepaald worden door een eigen implementatie van een treebuilder. Als bonus bestaat ook de mogelijkheid slots in het template zelf te configureren (in-template configuraties).
Frontend ontwikkelaars zijn hierdoor instaat om eigen layouts aan te maken waarin hier al instellingen gedefinieerd kunnen worden.

```
<html>
	<head>
		<title>WM3 Layout</title>
	</head>
	<body>
		<div class="header">
			{% render_slot('header', 'headerarea', {'debug': true})  %}
		</div>
		<div class="mainarea">
			{% render_slot('mainarea', 'big', {'extra_config': 'foo'})  %}
			<div>
				{% render_slot('mainarea', 'small', {'extra_config': 'foo'})  %}

			</div>
		</div>
	</body>
</html>
```

De render_slot secties in de code representeren de plaatshouders, waarin bijvoorbeeld widgets getoond kunnen worden. Een frontend ontwikkelaar heeft de mogelijkheid, invloed op de werkwijze

van een widgets uit te oefenen via in-template configuraties (bijvoorbeeld: 'big': Een hypothetisch RSS feed widget wordt omvangrijker getoond; 'small': Alleen de titels van een RSS feed worden getoond.

Bovendien worden instellingen in de backend opgeslagen, die bijvoorbeeld ook rechten kunnen bevatten.

5.3 Ondersteuning van widgets

Eis: Ondersteuning voor widgets: Widgets zijn configureerbare elementen t.b.v. de weergave van content. Een voorbeeld is een widget voor het tonen van een lijstje met de meeste recente twitterberichten waarin de eindgebruiker configureert hoeveel berichten er getoond worden en wat het bijbehorende twitteraccount is. Widgets moeten zich bewust zijn van het gebied waarbinnen ze geplaatst worden zodat de weergave van de content geschikt is voor het gebied.

Resultaat: In de WidgetBundle is de widget functionaliteit geïmplementeerd. Deze bundle werkt onafhankelijk van het Layout/Slot/Channel systeem, wat betekent, dat widgets helemaal niets weten van slots en layouts. De WidgetBundle biedt een eenvoudig interface voor het aanmaken van nieuwe widgets:

```
class RSSWidgetService
{
        public function execute(
                WidgetModelInterface $widget,
                WidgetEnvironmentInterface $environment,
                Response $response
        ) {
                $configuration = $widget->getConfiguration(); //configuratie uit de database

                /*
                 * Dit is de configuratie die bij een widget instantie hoort en kan later via
                 * het user interface geconfigureerd worden
                 */
                $per_page = $configuration['per_page']; //aantal rss artikelen per page
                $URL = $configuration['URL'] // rss URL

                /*
                 * $environment bevat feitelijk de configuratie, die een frontend
                 * in een template voor een slot gedefinieerd heeft (zie boven)!
                 */
                $outputStyle = $environment->getOutputStyle(); //bijvoorbeeld 'big' of 'small'

                $additionalEnvironmentConfig = $environment->getConfiguration();

                //voorbeeld van frontend configuratie
                $debugMode = $additionalEnvironmentConfig['debug'] === true; //toon debug
informatie

                // ophalen van rss data
                // …
                // ...

                $response->setContent($rssDataStr);
                return $response;
        }
}
```

Dit betekent, dat het al voldoende is, als de exacte methode van een ontwikkelaar geïmplementeerd wordt. Getoond worden de widgets via de PageBundle, welke gebruik maakt van layouts, slots en widgets. In de code is ook te zien, dat een widget door het WidgetEnvironmentInterface object op de hoogte van het bereik is waarin het getoond wordt. In het $widget object zijn specifieke instellingen voor de momentele widget opgeslagen. Een widget van hoge kwaliteit zou configureerbaar zijn.

Hiervoor kan een optionele tweede methode geïmplementeerd worden, die een FormType retourneert:

```
class RssWidgetService
{
	public function getFormType()
	{
		return $this->rssFormType;
	}
}

class RssFormType
{
	public function buildForm(FormBuilderInterface $builder, array $options)
	{
		/*
		 * Hier wordt het formulier geconfigureerd, waarmee een gebruiker
		 * het rss widget kan instellen
		 */
		$builder->add('URL', 'text', ['constraints' => [new NotBlank()]]);
		$builder->add('per_page', 'number', ['constraints' => [new
NotBlank()]]);
	}
}
```

Uiteindelijk wordt het widget systeem door deze formulieren heel krachtig. Een ontwikkelaar kan door de implementatie van een form type zelf bepalen, welke instellingen configureerbaar moeten zijn door CMS gebruikers. In dit geval zijn dit het aantal pagina's en een passende RSS URL. Ten tweede heeft ook de layout invloed op de weergave van een widget door de render_slot methode. De bedoeling is, dat frontend ontwikkelaars widgets kunnen manipuleren zonder kennis van PHP te hebben. Ook heeft de ene layout andere eisen dan de andere layout. Misschien is een layout geoptimaliseerd voor mobieltjes of tablets. Dan stelt de mobiele layout waarschijnlijk andere instellingen voor dit slot ter beschikking dan de layout voor desktop browsers.

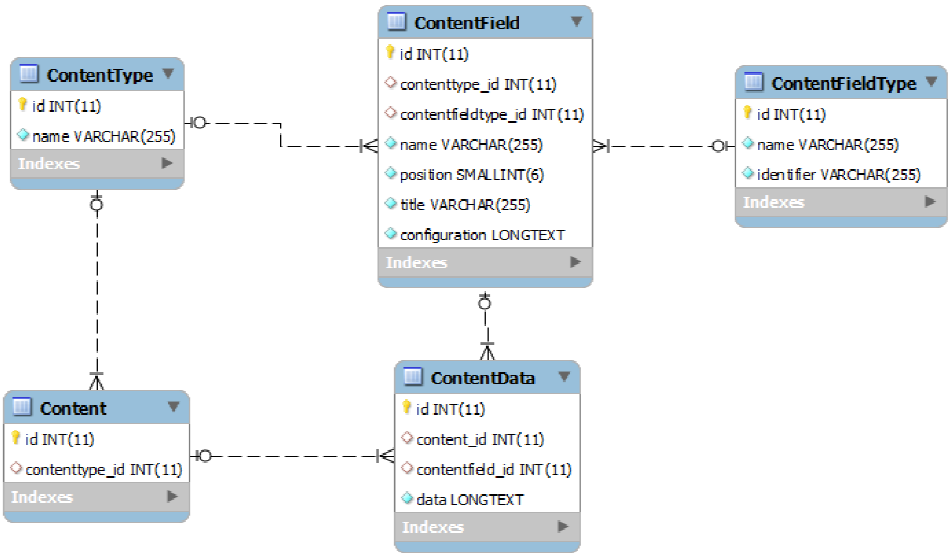
- render_slot in een desktop layout:
  `render_slot('mainarea', 'big', ['layouttype': 'desktop']);`

- render_slot in een layout voor mobieltjes:
  `render_slot('mainarea', 'big', ['layouttype' => 'mobile']);`

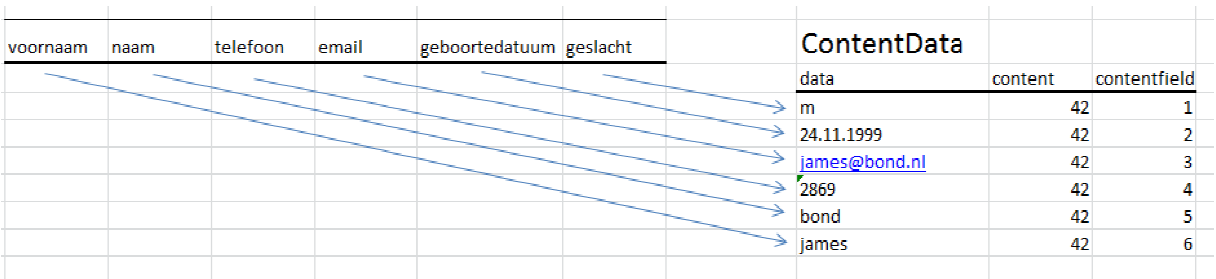
5.4 Supermodule

Eis: Implementeren van de "supermodule": Met de supermodule moet er eenvoudig vanuit het CMS een nieuw content type gedefinieerd kunnen worden. Op basis van dit content type kan uiteindelijk nieuwe content geschreven worden die gebruikt kan worden binnen diverse kanalen. Er is bijvoorbeeld behoefte aan het publiceren van persberichten. Via de supermodule wordt dan een nieuw content type met de naam 'Persbericht' aangemaakt en voorzien van de velden 'titel', 'intro', 'inhoud' en 'auteur'.

Resultaat: De supermodule is volledig geïmplementeerd. Nieuwe content types kunnen aangemaakt en gewijzigd worden. Technisch gezien worden de content types met hun inhoud via vijf tabellen mogelijk gemaakt:



- De tabel **ContentType** slaat nieuwe content types op
  Dit zou overeenkomen met een nieuwe tabel in de database. (bijvoorbeeld tabel Persbericht)
- De tabel **ContentFieldType** bevat een lijst met alle mogelijke veldtypen die in het CMS ter beschikking staan. Simpele veldtypen zouden Number, Text of URL zijn. Ieder ContentFieldType in deze tabel heeft altijd een bijhorend form type class, die het formulier voor dit veld definieert. Hierdoor kunnen gemakkelijk formulieren voor content types gegenereerd worden. Nieuwe veldtypen kunnen eenvoudig door een ontwikkelaar geïmplementeerd worden. Het ContentFieldType is te vergelijken met de **fieldtypes** in een relationele database (text, varchar, number enz.)
- De tabel **ContentField** is een koppeltabel en bevat alle veldtypen van een ContentType entiteit. Alle ContentType entiteiten (persbericht, blog artikel, gastenboek) kunnen gebruik maken van de veldtypen. Hiervoor is deze koppeltabel nodig, die de veldtypen voor een ContentType definieert. Dit is te vergelijken met een **kolom** in een relationele database (persbericht: name, date, inhoud enz.)
- De tabel **Content** slaat nieuwe content instanties van een ContentType entiteit op. Dit is te vergelijken met een nieuwe **rij** in een relationele database. (bijvoorbeeld persbericht met de id 42)
- De tabel **ContentData** is een koppeltabel en bevat de data van specifiek ContentField in de rij. Deze tabel is te vergelijken met de data in een **cel** van een relationele database, die feitelijk de data opslaat (bijvoorbeeld: name = „Webstores.nl met nieuwe homepagina“, date = „12.12.2012“ ...)



Ontwikkelaars kunnen op een transparante manier met aangemaakte content types werken (CRUD). Dit betekent, dat het concept voor een ontwikkelaar niets perse bekend moet zijn. Het lijkt een

normale tabel in een database te zijn waarbij het in feite een samenwerking van meerdere tabellen is. De volgende code zal dit illustreren:

```
//..
$persbericht = $recordManager->findById(42);
echo 'Nieuw persbericht met de titel' . $persbericht['name'] . ' aangemaakt op: ' .
$persbericht['date'];

echo 'Inhoud: ' . $persbericht['inhoud'];


//bericht wijzigen
$persbericht['name'] = 'Een nog leukere homepage is nu online!';
$recordManager->save($persbericht);
```

Met een rij van een ContentType (= een record in WM3) kan op eenvoudige manier via de Array notatie van PHP gewerkt worden.

Omdat evenals alle andere bundles hier ook met het FormType concept (zoals bijvoorbeeld in het widgetbundle) van Symfony2 gewerkt wordt, kunnen formulieren voor een ContentType op soortgelijke manier gegenereerd worden. De complete class structuur van het algemene ontwerp kan op deze manier voor dit bundle ook toegepast worden.

5.5 Documentatie

Eis: Documentatie van de ontwikkelde API.

Resultaat: De complete code werd voorzien met commentaren in het Nederlands, waarbij omvangrijkere classes deels een uitgebreid commentaar bovenaan hebben staan. De documentatie volgt de PHPDoc standaard, waardoor eenvoudig gebruikersvriendelijke documentatie gegenereerd kan worden. Omdat alleen de classes te beschrijven nog geen voldoende inzicht garandeert, is bovendien nog documentatie in het intranet van webstores.nl beschikbaar als Wiki. Hier worden belangrijke concepten met behulp van diagrammen uitgelegd. Ook zijn hier referenties te vinden zoals de REST API en service definities van WM3 classes, die via Symfony2's dependency injection ter beschikking gesteld worden. In de bijlage zijn een aantal artikelen uit het intranet te vinden.

6 Conclusie

Het project is in het geheel goed verlopen. Het eindproduct is een stuk software, dat uitbreidbaar, flexibel en gedocumenteerd is. Het stelt een goede basis voor het toekomstige CMS ter beschikking, waarin alle onderdelen vervangbaar zijn en nieuwe componenten eenvoudig geïntegreerd kunnen worden. Het product te ontwikkelen was een geavanceerde uitdaging. Aan de ene kant werd hoge flexibiliteit vereist, aan de andere kant moest de code ook gemakkelijk te begrijpen zijn van andere ontwikkelaars. Bereikt werd dit met name ook door een strikte scheiding van afhankelijkheden. Ieder class heeft één taak. Hierdoor blijft ieder class klein en overzichtelijk – en goed te bestuderen.

Uitdagend was in feite niet de implementatie van de broncode, maar de ontwikkeling van het softwaredesign. Het was deels moeilijk, om consequenties en scenario's van een gekozen benadering te herkennen.

Het eindproduct is uitdrukkelijk alleen de basis voor het toekomstige CMS van Webstores (WM3). Belangrijke componenten zoals een geschikt Security component, een routing component of zelfs een user interface ontbreken nog. Waarschijnlijk zullen de toekomstige componenten ook een soortgelijke class structuur hebben zoals de core bundles.

Een niet te onderschatten uitdaging blijft het training van de medewerkers. Om het systeem te begrijpen en uit te breiden is kennis van Symfony2 een dwingend eis. Er zijn ongetwijfeld frameworks, die gemakkelijker te leren zijn dan Symfony2.