07-02-10 - MVC-Model: Model, View & Controller

Een goede programmeur c.q. webdeveloper bouwt zijn (web)applicatie gestructureerd op. Iedere ontwikkelaar heeft zo zijn eigen mening over wat een goede structuur is. Een hele bekende is het MVC-Model: Model, View & Controller. Dit geeft eigenlijk een hele globale scheiding aan binnen de architectuur van de software. Met dit artikel wil ik uitleggen hoe het MVC-Model werkt. Ik ga je dus niet overhalen om het MVC-Model te gebruiken, maar hoop dat je zelf in gaat zien dat dit een goede methode is.

MVC of MVC?

Iedere programmeur heeft zo zijn eigen opvattingen over wat een goede structuur is. Maar ook binnen het MVC-Model zijn er veel verschillende opvattingen. Bij het MVC-Model scheiden we de code over drie lagen, namelijk Model, View en Controller. Maar wat stop je in welke laag en vooral: waarom?

Drie lagen

Model

Het Model is de kern van de applicatie. Dit wordt ook wel de business-layer genoemd, omdat in deze laag het ?doel' van de applicatie wordt vervuld.

Een applicatie heeft altijd één of meerdere doelen. Deze doelen kunnen heel eenvoudig zijn, zoals het weergeven van de tijd. Maar het kan ook een ingewikkeld proces zijn, zoals de aansturing van een machine. In mijn werkgebied is het doel meestal het weergeven en/of wegschrijven van data naar een database. Een website kun je bijvoorbeeld beschouwen als twee losse applicaties: een voor de bezoekers en een voor de webmaster(s). Deze applicaties kunnen dan weer onderverdeeld zijn in losse applicaties. Bij een weblog is het doel van de applicatie om artikelen met reacties weer te geven en daarnaast nieuwe reacties weg te schrijven naar een database. Het Model heeft in dat geval de verantwoordelijkheid om artikelen en reacties op te halen uit een database en nieuwe reacties toe te voegen aan de database. Het model doet dus helemaal niets met de weergave. In een OO-gebaseerde omgeving krijgen we twee classen in het Model: Article en Comment.

View

De View verzorgt de presentatie van de applicatie. Dit kan een GUI zijn voor een

programma of een template van een website. Let er op dat de View zelf helemaal niets doet. De View zorgt er voor dat de gebruiker iets heeft om naar te kijken en eventueel nog iets in kan vullen, maar verder helemaal niets. Geen validatie en al helemaal geen afhandeling!

Controller

De Controller is de schakel tussen het Model en de View. In Voorbeeld 1 komt dit eigenlijk nog het beste naar voren in de vorm van een Servlet. Een Servlet is een "applicatie" die de response van de GUI kan afhandelen. Wanneer er in de GUI een formulier wordt ingevuld en verzonden, wordt de data opgehaald door de Controller/Servlet. Deze weet vervolgens dat hij de data in een object moet stoppen en hoe hierop te reageren. Als we verder bouwen op het weblog-voorbeeld, zou dit de Controller zijn die het toevoegen van een comment afhandelt:

12345678910111213141516	<?php$message = "";$error = "";$comment = new Comment ( );$comment->addData ($_POST);if ($comment->validate ( )) if ($comment->save ( )) $message = "Thank you for your comment!"; else $error = "Something went wrong while saving your comment!";}else $error = "Please fill in all required fields!";?>

In commentForm.php wordt vervolgens $message of $error weergegeven.

Voorbeelden

Ik zal nu aan de hand van enkele theoretische voorbeelden uitleggen waarom het handig is om volgens het MVC-Model te werken. Ik heb geprobeerd de voorbeelden zo gevarieerd mogelijk te houden, zodat het duidelijk wordt dat je het MVC-Model in vrijwel iedere situatie kunt toepassen.
Ik heb de voorbeelden niet in code uitgewerkt, omdat me dit een overbodige toevoeging leek. Ik omschrijf alleen wat er gebeurt in een bestand. Mocht je toch willen weten hoe het werkt, zoek dan eens verder op MaxiBlog of Google.

Voorbeeld 1: Ajax/PHP Applicatie

In het vorige hoofdstuk heb ik al een voorbeeld gegeven van een PHP-applicatie volgens het MVC-Model. Daarom ga ik nu gelijk door met een voorbeeld dat ook gebruik maakt van Ajax.

Casus

Ontwikkel een contactformulier dat een bericht stuurt naar x@x.x. Er moet gebruik worden gemaakt van Ajax, aangezien de rest van de website ook in 2.0-stijl is ontwikkeld. Verder is het belangrijk dat er gebruik wordt gemaakt van PHPMailer, zodat de e-mail niet bij de ongewenste post komt.

Model

In het model hoort uiteraard PHPMailer. Dit is de class die de e-mail gaat versturen. Verder kunnen we nog meer Model-classes ontwikkelen, zoals een FormGenerator, maar dit is niet strict noodzakelijk en zal ik daarom ook niet doen.

View

De view bestaat in dit geval enkel uit een index.html. Hierin komt het formulier te staan en een link naar de javascript-file(s). De javascript-file(s) horen niet bij de View, omdat dit niets meer te maken heeft met de weergave aan de gebruiker, maar de afhandeling.

Het is overigens slim om een fallback in te bouwen voor het formulier. Mocht de browser van de bezoeker geen javascript ondersteunen, dan kan het bericht alsnog verzonden worden.

Controller

In de controller komen twee bestanden, namelijk een servlet.php en een javascript-file. In de javascript-file wordt een request gedaan naar servlet.php. De servlet zal vervolgens de gegevens doorzetten naar PHPMailer en opdracht geven aan PHPMailer om de e-mail te verzenden.

Voorbeeld 2: Java Applicatie

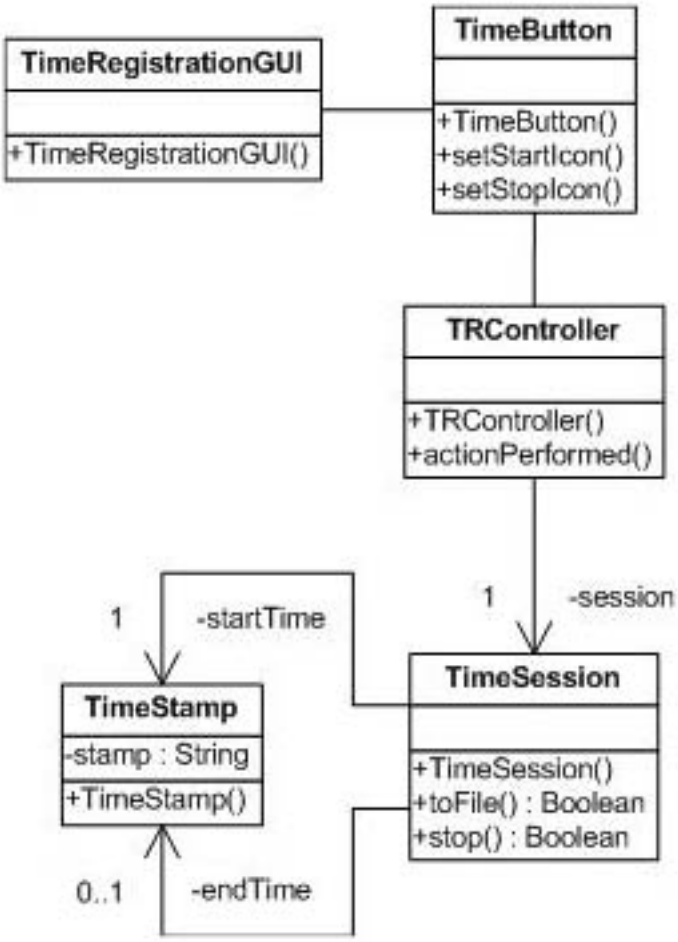
Casus

Er moet een kleine applicatie ontwikkeld worden, waarmee gebruikers kunnen registreren hoe lang ze gewerkt hebben. Het is de bedoeling dat een gebruiker aan de zijkant van het scherm een knopje krijgt waarmee hij de urenregistratie aan en uit kan zetten. Wanneer de registratie wordt uitgezet, wordt het resultaat weggeschreven naar een text-file op de computer. Aan het einde van de maand kan de gebruiker deze file zelf doorsturen naar zijn manager.

Er hoeft geen rekening gehouden te worden met stroomuitval. Er hoeft ook geen bewerk-optie in te zitten, zodat een medewerker zijn uren kan veranderen wanneer hij bijvoorbeeld vergeten is uit te loggen.

Ontwerp

Ik heb voor deze situatie een klein klassendiagram gemaakt.



Model

In het model hoort in dit geval TimeSession en TimeStamp. Wanneer er een nieuwe TimeSession wordt aangemaakt, zal deze automatisch een TimeStampobject maken. Dit object zal dan de huidige systeem-tijd pakken en in een variabele opslaan.
Wanneer de Controller de stop-functie aanroept, zal deze een nieuwe timestamp genereren en de functie toFile aanroepen, zodat de tijden weggeschreven worden naar een bestand. Wanneer de stop-methode nogmaals wordt aangeroepen, zal er false terug gestuurd worden.
In de TimeStamp zal ook een toString gebouwd moeten worden, zodat deze naar het bestand weggeschreven kan worden. We hoeven in deze situatie niet bang te zijn dat objecten tegelijk dezelfde file willen beschrijven. Er kunnen immers nooit

twee timesessions tegelijk worden gesloten door één medewerker. Anders zouden we hier een apart object voor moeten maken die de file beschrijft. Voor de snelheid van het programma is dit overigens ook beter. Aangezien we nu in dezelfde ?thread' zitten als de GUI, zal de GUI onbruikbaar zijn tot het schrijven voltooid is. Maar dat laten we even liggen voor versie 2!

View

De view is niet zo moeilijk denk ik: TimeRegistrationGUI en TimeButton. De TimeRegistrationGUI is het venster/frame van de applicatie. De positionering van deze GUI kan gedaan worden door TimeRegistrationGUI, maar dit zouden we ook over kunnen laten aan een controller. Dat ligt er een beetje aan hoe ?dom' je de view wilt houden. Persoonlijk vind ik dit een taak die wel degelijk bij het GUI-object hoort, aangezien het alleen betrekking heeft op zichzelf en te maken heeft met de weergave van het object. Zo vind ik ook dat de TimeButton de afbeeldingen mag inladen die op de button weergegeven wordt. Er zijn ongetwijfeld mensen die het hier absoluut niet mee eens zijn, maar dat is gewoon een kwestie van smaak.

Controller

De overgebleven class (TRController) is dan uiteraard de Controller, zoals de naam al doet vermoeden. TRController implements ActionListener en de TimeButton is hier aan gelinkt. Op het moment dat iemand op de knop drukt, zal de actionPerformed van TRController de huidige TimeSession proberen te sluiten. Is de TimeSession gelijk aan null, of geeft de stop-methode een false terug, dan maakt de controller een nieuwe TimeSession aan. De controller zal afhankelijk van de situatie het icon van de button laten veranderen door setStartIcon of setStopIcon aan te roepen.

Voorbeeld 3: J2EE Applicatie

Casus

Ontwikkel een webapplicatie waarmee mensen de actuele waarde van hun auto kunnen berekenen. De bezoeker selecteert een merk, type en bouwjaar, waarna de prijs getoond zal worden op het scherm.

Model

Bij J2EE (zonder JSF of ander framework), bestaat het model uit Beans met gegevens en eventueel validators. In dit geval hebben we aan een CarBean en

ClientCarBean genoeg. In de CarBean slaan we de database-auto's op. Een CarBean bestaat dus uit:

```
private String brand;private String type;private HashMap prices;
```

Deze gegevens kunnen zo bijgehouden worden, of in een HashMap (wat de voorkeur heeft, omdat je de Bean dan een HashMap kunt laten extenden, waardoor deze automatisch een get-functie heeft).

View

Bij J2EE is de view altijd een JSP-file. Hierin zet je **alleen** html-code en Expression Language. Met EL roep je de gegevens uit de bean op. Voor dit project kan ik de invoervelden maken en een veld met het resultaat. Wanneer er nog niets is ingevuld, blijft dit veld gewoon leeg.

Controller

Servlets dienen als Controller bij J2EE. Deze zagen we ook al bij het Ajax-voorbeeld. In dit geval zorgt de controller er voor dat de JSP-pagina wordt ingeladen en dat de beans aan de pagina wordt gekoppeld. Bij J2EE zijn de Servlets "gedeelde" objecten. Ze blijven in het geheugen van de server staan en

worden dus gebruikt voor alle bezoekers. Het is dus niet de bedoeling dat je gegevens op gaat slaan in deze Servlet. Dan gaan de gegevens van alle bezoekers door elkaar heen! Je moet daarom gebruik maken van de Beans. En deze Beans moeten gemaakt worden in de doGet of doPost van de servlet. Niet in de constructor!

Conclusie

Ik hoop dat ik aan de hand van deze voorbeelden duidelijk heb kunnen maken waarom het MVC-Model zo verschrikkelijk handig is. Misschien gebruik je het eigenlijk al een beetje en geeft dit meer houvast bij het ontwerpen van nieuwe applicaties.

Als laatste moet ik nog een keer zeggen dat er veel verschillende opvattingen zijn over het MVC-model. In het tweede voorbeeld liet ik dit al een beetje doorschemeren door twijfel te zaaien over wie de positie van de GUI moest bepalen. In (X)HTML zou je hier geen seconde over twijfelen, omdat dit duidelijk een CSS-taak is. Bij Java is dat een ander verhaal. En dat maakt het allemaal zo verdraaid ingewikkeld en geeft reden voor discussie. Ik nodig alle critici dan ook van harte uit om hun mening te geven op mijn voorbeelden. Vergeet niet, er zijn velen wegen die naar Rome leiden en iedereen heeft zijn voorkeur: de snelste, kortste, mooiste, etc...