Coupling Server-Side Templates and Client-Side Models

Anders Ingemann, 20052979

Master's Thesis, Computer Science

July 2012

Advisor: Michael Schwartzbach

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English. . . ◄

Resumé

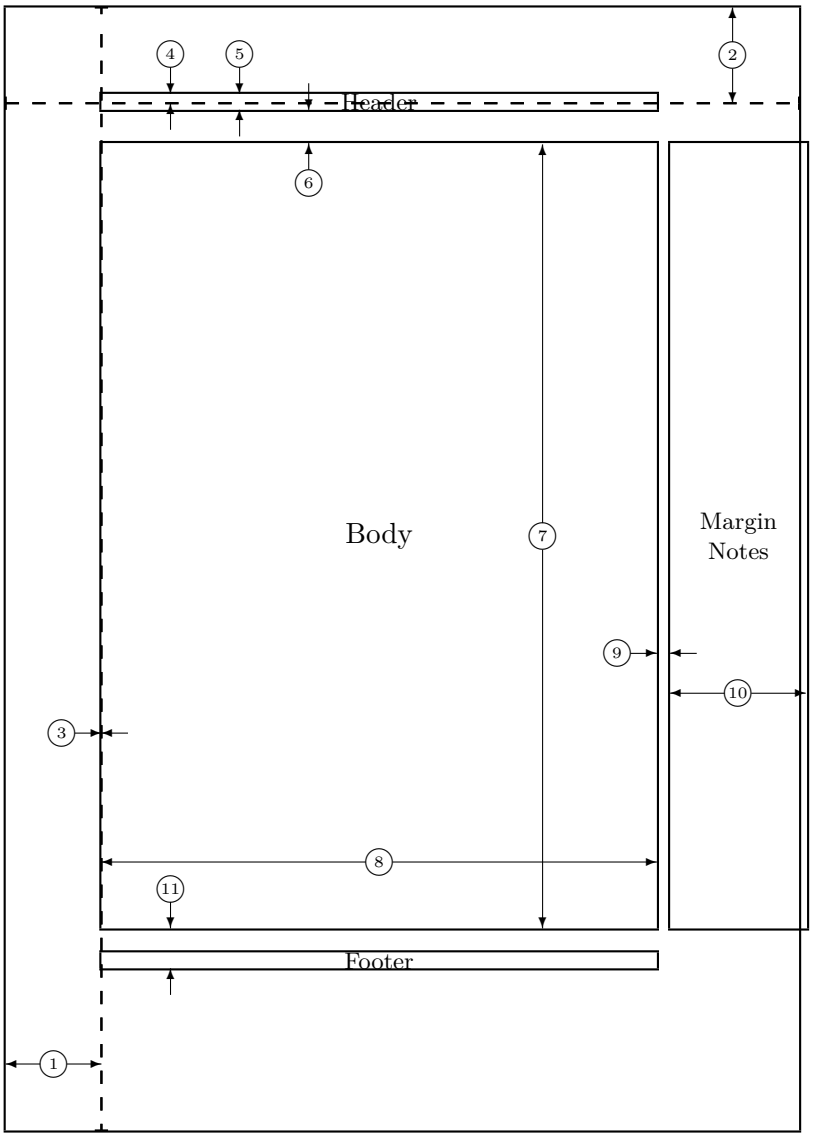
►in Danish. . . ◄

Acknowledgements

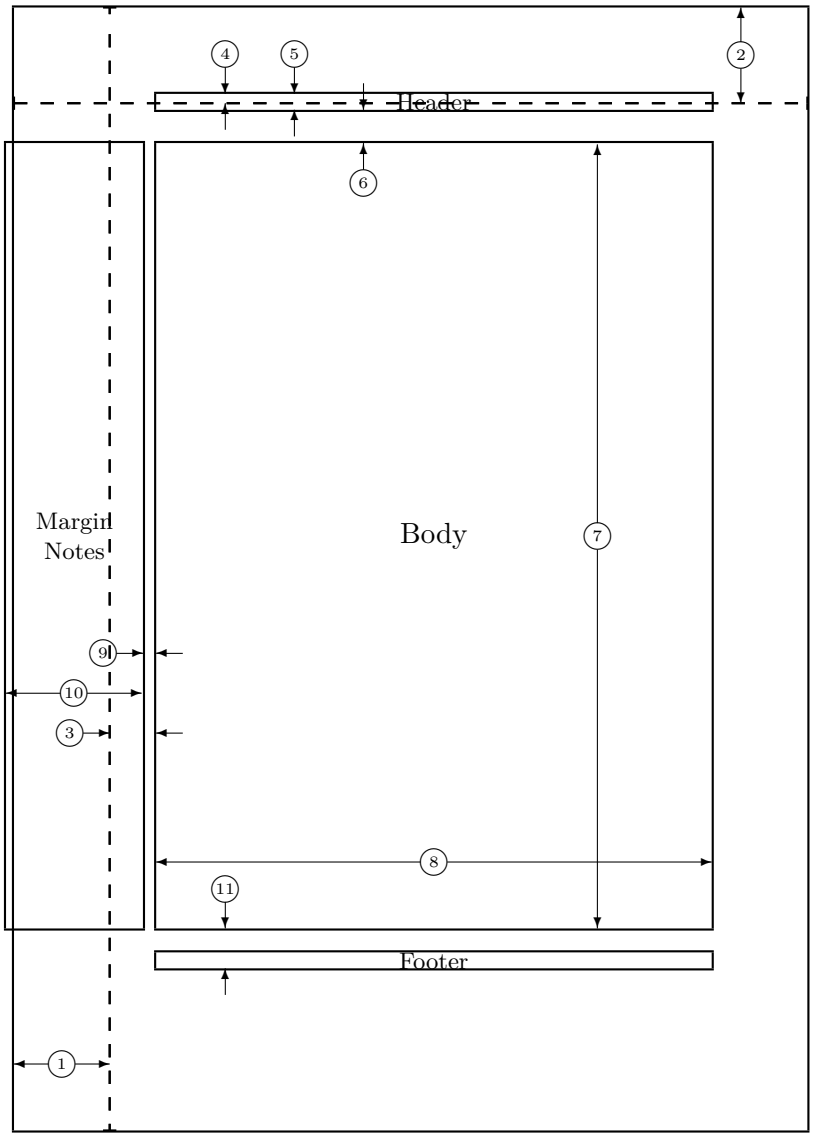
►. . . ◄

Anders Ingemann,
Aarhus, July 11, 2012.

Contents



1	one inch + \hoffset	2	one inch + \voffset
3	\oddsidemargin = 0pt	4	\topmargin = -7pt
5	\headheight = 12pt	6	\headsep = 25pt
7	\textheight = 591pt	8	\textwidth = 418pt
9	\marginparsep = 10pt	10	\marginparwidth = 103pt
11	\footskip = 30pt		\marginparpush = 5pt (not shown)
	\hoffset = 0pt		\voffset = 0pt
	\paperwidth = 597pt		\paperheight = 845pt



1	one inch + \hoffset	2	one inch + \voffset
3	\evensidemargin = 35pt	4	\topmargin = -7pt
5	\headheight = 12pt	6	\headsep = 25pt
7	\textheight = 591pt	8	\textwidth = 418pt
9	\marginparsep = 10pt	10	\marginparwidth = 103pt
11	\footskip = 30pt		\marginparpush = 5pt (not shown)
	\hoffset = 0pt		\voffset = 0pt
	\paperwidth = 597pt		\paperheight = 845pt

Chapter 1

Introduction

►. . .◄

►example of a citation to primary literature: [A1], and one to secondary literature: [B2]◄

1.1 Development work flow

The tool introduced in this thesis, will be developed via an iterative work flow. Two or three ►which is it?◄ versions of the solution to the requirements outlined above will be discussed in this thesis. Each version building on the knowledge acquired in the development process of the previous.

Chapter 2

Developing web applications

Web applications are on the rise. Not a day goes by where a new web application isn't popping up for uses that were previously reserved for a program locally installed on a computer. Even more so: Previously unimagined uses for any Internet enabled device seem to be developed at a rate that surpasses the former.

2.1 What are web applications?

2.1.1 Synchronous and asynchronous communication

Any web application can be divided into a server part and a client part. Mostly both parts play a role in providing functionality to a web application. The server holds persistent data in order for the user to be able to connect from any machine. Since the server is not in the same location as the client machine, latency in responses to user actions are a problem. Diagram 2.1 illustrates what such an interplay between client and server looks like.

The client uses the web application to enter data, this we will call user activity. Once the user issues a command that requires data from the server, the web application issues a request to the server and halts any further execution until the server has responded. Once the client receives a response, it continues execution where it left off and the user can continue interacting with the web application. This is called a synchronous communication. The client acts in synchronicity with the server and does not act independently from the server. The higher the latency of the response, the longer the user has to wait to interact with the web application again. Such waiting periods are undesirable as, it results in a loss of productivity and user-friendliness.

This challenge is solved by letting the client part of the web application continue to compute responses when information on the server is not required. Queries to the server can then happen asynchronously, meaning a function can define a callback function which is invoked once the response is available. This is called asynchronous communication; the client continues execution after it has sent off a request to the server. Once the server responds, a function on the client will be invoked to handle that response. Asynchronous

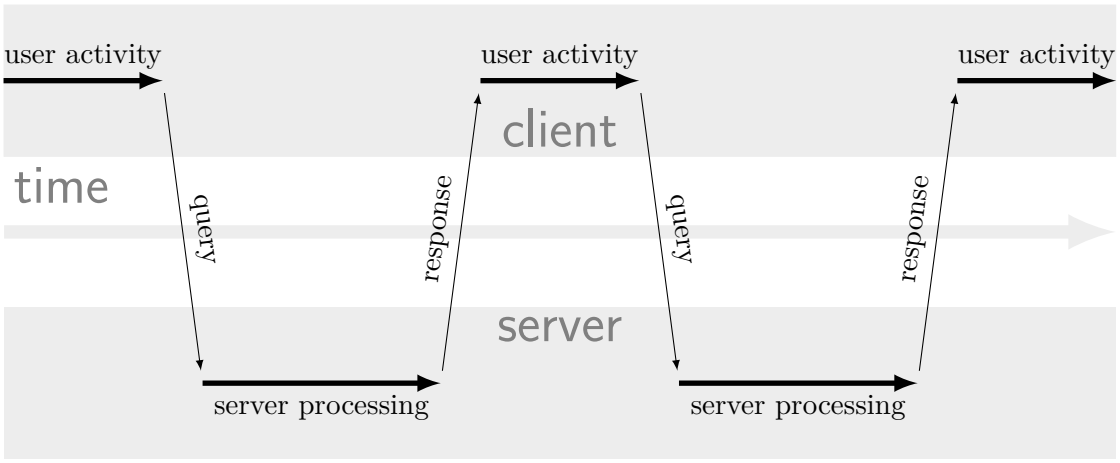


Figure 2.1: Diagram of synchronous communication between client and server

communication allows the user to continue to interact with the web application. Diagram 2.2 shows how user activity can take place, while the server computes a response to a previously issued request.

2.1.2 ►Needs a title◄

In this thesis, we will focus on web applications which use a modern web browser and with it HTML as their basis (HTML5 in particular). The non-static parts, which control the heart of the web application, are supplied by JavaScript. This not only includes interactivity, but also animation and updates from the server.
Interactivity in this context is defined as anything in the web application the user can modify directly via an input device or modify indirectly, e.g., the back button in the browser and the window size of the browser.

Alternatives to JavaScript like Dart, CoffeeScript and Google Web Toolkit do exist and are meant to ameliorate the shortcomings of JavaScript. However, they are all translated into JavaScript if cross-browser compatibility is a requirement (which it almost always is).

The claim that web applications are meant to be ubiquitous, operating system independent and run in the browser, is not a claim shared by all definitions of a web application. For simplicity however we will for the remainder of this thesis treat it as fact.

2.1.3 Server-side web applications

A web application can incorporate business logic ►**definition**◄ and interactivity by rendering customized HTML pages solely on the server. Rendering the HTML entirely on the server can be advantageous in a number of situations:

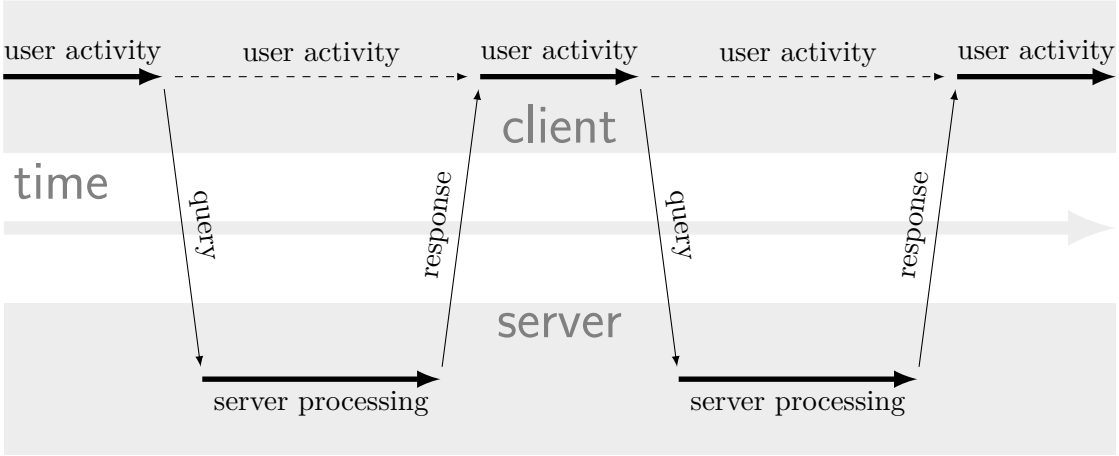


Figure 2.2: Diagram of synchronous communication between client and server

- *Heavy computations can be run in a controllable time frame regardless of the client device.*
  Especially phones and other portable devices have reduced computing capacity in order to save battery power.

- *Sensitive data can be handled without leaking it to the client.*
  Any data that the client is not supposed to see, can never leave the server. This means if any computation on the data should take place, it would have to be made insensitive, e.g., in the case of personal data for statistical purposes, the data would have to be anonymized first.

- *The client application has to be initialized with data for each page load.*
  Data that gives the application context, is – depending on the language and implementation – loaded in RAM and/or saved in a database. On the client this data would first have to be loaded either from the server or from the local storage.

- *The technology stack is more controllable.*
  The main browser technology stack, i.e., CSS, HTML and JavaScript, has suffered greatly under the "browser wars" **►reference◄** and has only gained widespread standardization in the last 5 years. There are still many inconsistencies, especially when tackling edge cases (for example the "Guillotine bug" in Microsoft Internet Explorer 6 and 7, each with their own variation **►http://www.positioniseverything.net/explorer/gu**
  This technology stack and its edge cases is greatly reduced when the application runs on the server, because every software version and the software itself can be controlled by the developer.

2.1.4 Client-side web applications

Web applications can also be developed solely using client-side code, leaving the server to only supply static content. "Mashups" ►definition◄ are fitting examples of such an approach. These web applications rely on external JavaScript APIs (Google Maps, Twitter, Weather services) to combine readily available in new ways. The client browser requests and combines this data without interacting with the server. These external APIs retrieve their data from servers of course, but those servers are not maintained by the developer and are exposed to neither the user nor the developer.

As with the server-side only approach, the client-side only approach has some exclusive advantages.

- *Low server load*
  The server needs only serve static content. For most web applications, all of that content fits into the RAM, allowing fast response times and scalability. The server could be removed entirely if the content is hosted on a Content Delivery Network (CDN ►definition◄).

- *Accountability*
  When handling data sensitive to the user, the ability to audit the code that handles the data removes the necessity to trust the provider of the web application. Provided the client-side code is not obfuscated every operation can be audited by a third party or the user himself. ►Examples: Passpack, Strongcoin◄

- *Portability*
  To run a web application, which relies on communication with a server, requires an Internet connection. A client-side only web application does, in some cases, not have that requirement. The browser can store all the code that is necessary to run the web application, provided that no data from other services is necessary, the user can open the web application without an Internet connection and still use it. ►Examples: Google Chrome apps, Offline GMail with Google Gears◄

JavaScript is of course not the only way to create interactive web applications. Technologies like Java Applets and Adobe Flash have existed for a long time and made their impression upon the world wide web. We will not use those technologies for anything in this thesis. They will not be included in any comparisons or alternatives.

2.1.5 Combining the strengths

The arguments from 2.1.4 and 2.1.3 do not make the case for either approach to construct a web application. They instead highlight the strengths of both. A combination of server-side and client-side processing where their respective advantages are utilized and their drawbacks avoided, will help in creating responsive and maintainable web applications. An example of that would be guessing server responses:
Often lag between action from the user and response from the server cannot be avoided.

Instead, asynchronous communication allows the interface to stay responsive. The client-side code can then guess what the result from the server will be and update the interface accordingly. Later it can correct any discrepancies between the guess and the actual response from the server.

2.2 The development process

The development process of a web application is similar to most software development processes. One starts with the data to be modeled. It may be developed for the client and server part simultaneously. A protocol for communication between the two is then established. The design of a web application is usually the last component to fall into place. It may have existed in the very beginning of the development process, but is usually only finished and implemented when most other critical components are in place.

2.2.1 Usual design patterns

Design patterns help developers to organize software projects into agreed upon components, where each component has a specific function (also referred to as "concerns" or "responsibility"). Although their exact features may not be known when a developer is first introduced to a new software project, design patterns help him to quickly recognize where functionality may be located in the code.

Instead of requiring developers to think up new structures, Design patterns also help developers with grouping new code into well known components.

We will focus on one specific design pattern in this thesis. There many others, which are relevant in web application development. There is nonetheless one prevalent design pattern which we will examine in the following.

Model-View-Controller

The Model-View-Controller design pattern has proven itself to be a sane choice for developing web applications. Most frameworks today use this pattern or variations thereof. It lends itself very well to web applications because of the client server model, components of this pattern can be present on both sides allowing the structure to be homogeneous.

- The "Model" part represents the "data". All dynamic parts of an application modify, create or delete data, however ephemeral this data may be. Since much of the data can grouped, because it belongs to the same entity, it makes sense to represent those entities in the code and attach the data to them. This constitutes a Model. Besides this data, the Model can also have functions attached to it, which can act upon the data in various ways.
- A "Controller" implements the business logic that is decoupled from one specific Model. It draws on the functions tied to the models to perform its duties. Both of these components may be present on the server as well as the client.

- This is true for the "View" component as well. Its purpose however only comes to fruition on the client. This component is present on both the server and the client. Any HTML the server sends to the client is considered part of the "View" component.
  The job of the "View" component is to present the data to the user and tie calls to the controller to elements of the interface that can be acted upon by the user.

Designs of web applications change with time, features are added or removed and common processes simplified. In light of this, it is desirable to ensure that the "View" part of the Model-View-Controller pattern is easily modifiable.

Modifications of this pattern have evolved in the web application domain to cater to this specific purpose. The most notable of those would be the Model-View-ViewModel pattern. It was designed by Microsoft and targeted at interface development platforms including Windows Presentation Foundation and Silverlight but also HTML5.**►reference◄**

The "ViewModel" component allows developers to implement interface logic that lies between the Model and the View, allowing the view to be entirely free of code. This component is meant to hold data-bindings to the Model while listening to interface events at the same time. It "translates" user actions into operations on the Model. Without it, the view would have to be aware of how data is laid out internally, making refactoring of code harder.

The advantage of this version of Model-View-Controller is the improved separation of responsibility between interface developers and application developers. Neither will be required to modify or thoroughly understand the code of the other. Even if both roles are filled by the same person, separation of responsibility in an application still has its merits.

►MVVM from Microsoft, MVP by Taligent (Apple, IBM, HP)◄

Chapter 3

Tools of the trade

In order to achieve a separation of responsibility various frameworks and tools are at a developers disposal. In this thesis we will focus on two of them specifically.

3.1 Client frameworks

3.1.1 Backbone.js

underscore.js

prototype.js is a library developed by Sam Stephenson to improve upon the DOM API itself. It brought with it various improvements to native JavaScript prototype objects. underscore.js carries these improvements into the world of jQuery. It includes a small templating engine which will allow us to generate DOM elements and insert them into the page.

3.1.2 XPath

XPath is a language that allows us to define a path from a root node to another node. Although limited, the language is fairly concise and directly built into JavaScript. We will only be dealing with XPath 1.0, because version 2.0 is, with the exception of Microsoft Internet Explorer, not implemented in any browsers yet and likely will not ever be (the final specification was released 2006).

3.2 Templating languages

A templating language allows the developer to create HTML documents containing placeholders, which later can be filled by a Model and its attributes. Templating is part of the "View" component in the Model-View-Controller pattern. In the following we will have a look at one such templating language.

3.2.1 Mustache

Mustache is a so called "logic-less" template engine. This subtitle derives from the fact that there are no control flow statements (e.g. if and else statements and while loops). Instead there are only tags. Tags in in this context can be understood as an advanced form of placeholders. Some tags are replaced with a string, some are replaced with nothing, yet others are replace with series of strings or even more tags.

Limitations

►**fix entire paragraph**◄ These templating languages are very different in their design. All aim to improve one or more aspects of the templating task. Of those Mustache seems to be specifically tailored for web applications with interactive JavaScript parts. They all have a common trait which in some cases can be an advantage but given any specific implementation of a web application is a drawback: They are completely oblivious of their surroundings. They draw the line at the "View" part in order to encourage a separation from the other parts. This comes at the cost of lost information when sending a rendered view to the client.

Chapter 4

Requirements

In any web application we want to present data to the user. This data is embedded in HTML, which in turn is generated by the server. To this end we use templates that have placeholders for data. Different placeholders are meant for different fields from server-side models. We use server-side models to handle said data. The binding of model fields to placeholders represent information in itself. It is that information, which reveals where data in a HTML page originates from.

However, once a template is rendered, template engines discard that information. This loss of information is inconsequential to the way web applications are built with current frameworks. That does not however mean that it is useless.

Let us consider a minimal template used for displaying profile information: ►. . . ◄ As you can see, the fields of the user object are printed into the HTML at the appropriate places, leaving us with a normal page which can be displayed in the browser.

The following scenario illustrates how this simple way of handling templates, requires additional work when information about where data is put in the template is not readily available:

After the profile form has been styled with CSS, the developer decides that the submission of the form should not issue a page reload. The tool of choice for that is AJAX.

Once the form is working the way it should, everything is brought into production. Metrics however suggest that changing the layout of the form would increase usability. The designer moves form fields around to make it easier for user to update their profile. All the while, the developer has to accommodate the design changes by modifying the CSS selectors he uses to bind the form elements and the client-side version of the user model together. CSS identities are used sparingly to avoid naming conflicts, so every correction to a HTML template bears with it a correction in the selection of DOM nodes.

This example highlights a rather obvious loss of information, namely the position of the form elements and their connection to model attributes.
When the placeholders of an HTML template are filled these positions are known, but as

soon as the result is reduced to simple a string that is sent to the client, this information is lost.

The argument to uphold the status quo in this case is increased usage of CSS identities. The problem is however that this approach does not scale very well. A complex naming scheme would be required to avoid naming collisions. Every possibly modifiable DOM node would be tagged with a CSS identity, requiring more work in both the template writing and DOM binding of client-side models. **►Introduction to what I want to do about it◄**

Chapter 5

Initial prototype

First we make a rudimentary prototype. It is an exploratory prototype, meaning none of its code is intended to be carried over into the next iteration. The initial prototype is an interactive application for maintaining a movie library. Movie details can be edited and actors can be added to that library. It is not very useful in practice, but serves to make the basic idea more concrete in the following ways:

- *Materialize peripheral concepts*
  The prototype is meant to capture the core concept of the initial idea (to couple client models with server templates). Many of the less pronounced concepts of that idea will need to be made concrete in order for the core concept to work.

- *Highlight logical errors*
  Edge cases of an idea can be crucial to its successful implementation. Problems involving those cases may have been erroneously dismissed as trivial. Some of those problems may not have solutions or workarounds, which means the work on the entire project has been in vain. By making a working prototype, those errors will be discovered early on.

- *Discover additional requirements*
  The implementation of a movie library allows for practical challenges to arise, which might not have been discovered if we implemented such an application at the conclusion of the project. The advantage in the first iteration is our ability to course correct in subsequent iterations, if such errors should occur.

5.1 Architecture

The server-side back end is based on PHP and MySQL. The client-side uses HTML5, JavaScript (+XPath) and CSS as its core technologies.

5.1.1 Libraries

In order to speed up the prototyping process a plethora of libraries have been used. Excluding basic core technologies like JavaScript, MySQL and PHP, the application stack consists of the following:

- *less*
  A superset of CSS providing variables, calculations and nested selectors. It is a JavaScript library which compiles included less files into CSS.

- *jQuery*
  The de facto standard when creating web applications. Among other things it simplifies the interaction with the DOM.

- *backbone.js*
  Backbone.js is a JavaScript Model/View framework. It provides the developer with View, Model and Collection prototypes. The View prototype can be considered analogous to the aforementioned ViewModel, while the Model and Collection part make up the Model component and collections thereof, respectively.

- *php-activerecord*
  PHP ActiveRecord is the server-side library utilized to communicate with the database.

5.1.2 Templating

The application features rudimentary HTML templates, which are not backed by any engine. Instead PHP is embedded directly into the HTML files. Although PHP allows for more complex templates, we keep them simple in order to place the same constraints on the templates as we would have when using Mustache. We convert data from the database into HTML by fetching it from the database and by forwarding that data to the embedded PHP. As an example, figure 5.1.2 illustrates what the template for a movie looks like. None of the variables are scoped. Every variable can be referred to once it has been initialized. The PHP is embedded between `<?php` and `?>` tags.

- *Template inclusion*
  Starting from the root sub-templates are included via a simple PHP `require` command.

- *Simple variables*
  Simple variables are inserted via a PHP `echo` command.

- *Objects*
  Objects are converted into arrays so their fields can be initialized as variables with the `extract` method.

```
<li id="movie-<?php echo $id ?>">
  <details>
    <summary class="title"><?php echo $title; ?></summary>
    <header>
      <section class="operations">
        <span class="edit command">Edit</span>
        <a>Delete</a>
      </section>
    </header>
    <div class="main">
      <h1><?php echo $title; ?></h1>
      (<span class="year"><?php echo $year; ?></span>)
      <div class="poster">
        <?php echo $poster; ?>
      </div>
      <details class="plot">
        <summary class="synopsis">
            Synopsis: <span><?php echo $synopsis; ?></span>
          </summary>
        <p><?php echo $plot; ?></p>
      </details>
      <table class="cast">
        <thead>
          <tr>
            <td>Actor</td>
            <td>Character</td>
          </tr>
        </thead>
        <tbody>
          <?php foreach($cast as $role) {
            $actor = $role->actor;
            extract($role->to_array());
            require 'view/types/movie/role.tpl';
          } ?>
        </tbody>
      </table>
      <span id="add_actor_button" class="command">add</span>
    </div>
  </details>
</li>
```

Figure 5.1: The file movie.view.tpl. A template in the initial prototype.

- *Collections*
  Collections (i.e. PHP arrays) are simply iterated through, in this prototype only objects are present in these arrays, the block inside the `foreach` loop therefore simply contains the aforementioned method for placing object fields into the global scope.

- *Client-side templates*
  There is a small portion of client-side templates, which are used whenever new data is added. They do not have any impact on the concept explored in this prototype, instead they are an attempt to explore edge cases as outlined in the motivations for an exploratory prototype in the introduction of this chapter (5).

5.1.3 Models and ViewModels

Every Model on the server-side is linked to the database. One instance of a model represents one entity in the database. Each of these Models is also represented on the client-side using backbone.js, which provides us with a "Backbone.Model" base class that can be extended. **►Note for clarity that with "class" in JavaScript we actually mean object prototype◄**

Using the "Backbone.View", we can create ViewModels that bind the Model and the DOM together. For example, this can be used to listen to changes in form elements, which the ViewModel translates into changes of the corresponding fields in the Model. The Model can in turn synchronize those changes to the server.

Although not implemented in this prototype, the Model can also receive changes from the server (via server push or client pull methods) and notify the ViewModel about those changes. The ViewModel can then update the DOM (the user interface) with those changes.

5.1.4 Collections

In addition the above mentioned base classes backbone.js provides a third class. It is called a Collection and contains any number of other Model instances. In a given Collection the models all have the same type.

5.2 Coupling client-side and server-side models

Models and ViewModels are powerful abstractions. We can extend them to make use of the information that specifies which server-side field attribute belongs with what content on the HTML page.

Since the client-side model mirrors the server-side model a direct mapping of the information retrieved from the templates should be possible. The classification of this information is of importance: We will need to know whether a field contains a collection, a string, or an aggregated model, in order to parse the DOM properly.

We have two possible abstractions this information can be attached to and used by: The Model and the ViewModel. The information specifies where a Model field is located in the DOM, which would make the Model the optimal candidate. However, the information has a localized context since there can exist more than one server-side template per server-side model. This is at odds with the fact that there is only one client-side Model per server-side model. On the other hand there can be more than one ViewModel per Model. The ViewModels may even be coupled one to one with the templates. This property renders the ViewModel better suited to tackle this problem.

Storing the information on the ViewModel and letting it utilize it is advantageous because that information is valuable when binding event listeners to the DOM or manipulating DOM nodes otherwise.

In this prototype we will not focus on retrieving the information from the templates. Instead we assume this extraction has already taken place and simply hard-code XPaths into the ViewModels. Each XPath points to a position in the DOM where a server-side model field has been inserted. The XPath is labeled with the name of that field.

The ViewModel is a means to an end: It does not enable any meaningful interaction with the web application by simply binding to DOM nodes. It does however function as a bridge between that DOM and the Model. The Model in turn can communicate with the server, which can process the user interaction and return a meaningful response.

5.2.1 Parsing the DOM

The Models that are to hold the data we want to handle need to be created and populated with the data from the DOM when the page has loaded. To that end we use the ViewModels to parse the HTML and create both them and the Models they are attached to. A recursive approach will simplify the parsing in this matter, since the DOM is a tree structure.

We bootstrap the parsing function by giving it a "Root" model and a "RootView" ViewModel. Both are prototype objects, that will later be instantiated. The RootView has an XPath attached to it, which points to the list of movies in the DOM. The parsing function returns an instantiated ViewModel with a Model attached to it (e.g. the bootstrapping yields a "RootView" object containing a "Root" object).

During the process every XPath in the ViewModel is examined. Since any XPath is labeled with the name of the corresponding field on the Model, we can query the Model for the type of that field (this is hard-coded for the time being). The function `getAttrType()` returns that type.

We can follow three courses of action depending on the type returned by the Model.

- *The field is a simple type*
  For strings, integers and the like we populate the field of the Model with that value, and proceed to the next XPath.

- *The field aggregates another Model*
  The function queries the Model for the type of Model its field aggregates (`getComplexType()`).

We the recurse by calling the parsing function again. This time it is called with the aggregated Model class and the ViewModel class, which is returned by the `view()` function we attached to the XPath. **►This highlights that we need some way of mapping templates to viewmodels◄**

- *The field is a Collection*
  We query the Model for the type of Model the collection contains. The XPath can return more than one DOM node. For each of these nodes, we recurse. The return value of the function is pushed on to the Collection.

Once all XPaths have been examined, the function instantiates the Model class that was passed to it in the beginning. It is populated with the field values collected while examining the XPaths. The ViewModel, also a function argument, is then instantiated with the Model instance as one of the arguments. This ViewModel is the return value of the function.

One drawback to the method we use to obtain field values is the requirement for a post-processing function, that takes an XPath result and returns the correct value of a field. This is necessary because XPath is not an exact query language.

- DOM node attributes will be returned with both their attribute name and their value as one string. This is undesirable, since we almost always only place a server-side field value into the value part of a DOM node attribute.

- Substrings can not be retrieved with XPath, it can only return entire text nodes.

5.3 Results - Plans for next iteration

In the introduction of this chapter we listed some motivations for making this prototype.

- *Materialize peripheral concepts*
  We have created a movie database that supports a simple interface for maintenance and browsing interaction. Through this process we have discovered the recursive nature of retrieving model field values from the DOM. Less pronounced concepts like the classification of model field types have been made more concrete.

- *Highlight logical errors*
  We have not uncovered any major logical errors that would require us to rethink the idea for coupling client-side models to server-side templates.

- *Discover additional requirements*
  We not only mapped fields of models that aggregated another single model, but also collections of models. We solved this challenge in the prototype, by simply iterating through the nodes and adding them to a backbone collection. This method will need to be refined in the following iteration. We also discovered another major requirement, which we elaborate upon in more detail in the following paragraph.

The function `getComplexType()` illustrates, that we will need some form of mapping between ViewModels and templates. It will be tedious and error prone for the developer to create those mappings by hand. A process, which automates the coupling of the ViewModels with server-side templates will be needed in the next iteration.

While developing this application various libraries have been examined for their viability. backbone.js has in this case proven itself to be a very good fit. Its View prototype is made to bind with the interface while being Model aware. Such a component is what is needed to put the information about the placement of data in the DOM to good use.

Chapter 6

Implementation

6.1 Templating language syntax

6.1.1 Integrating the parser

6.2 Template-aware clients

6.2.1 Client-side architecture

6.2.2 Transmitting meta-data

Chapter 7

Usage

7.1 Application example

Chapter 8

Evaluation

8.1 Performance

8.2 Limitations

8.3 Advantages

8.3.1 Comparison

Chapter 9

Future Work

Chapter 10

Related Work

Chapter 11

Conclusion

►...◄

Primary Bibliography

[A1] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium, SAS '10*, volume 6337 of *LNCS*. Springer-Verlag, September 2010.

Secondary Bibliography

[B2] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, March 2010.