Coupling Server-Side Templates and Client-Side Models

Anders Ingemann, 20052979

Master's Thesis, Computer Science

June 2012

Advisor: Michael Schwartzbach

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English. . . ◄

Resumé

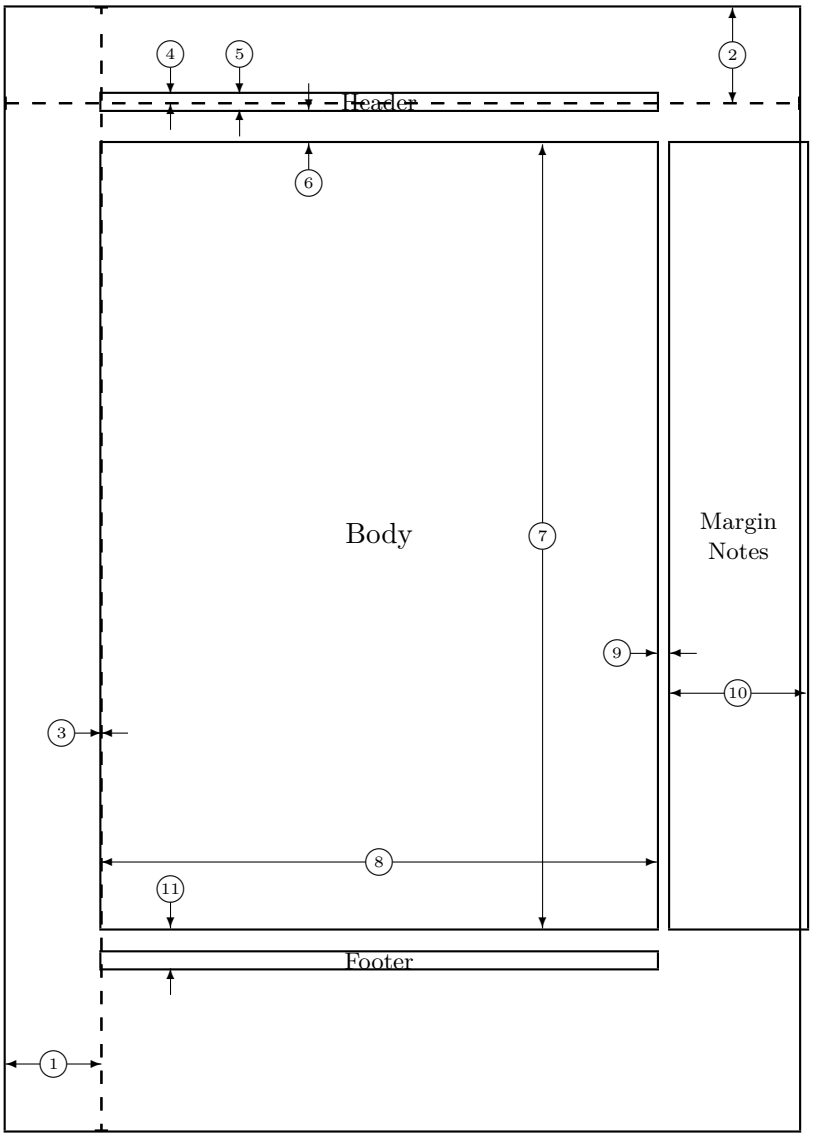
►in Danish. . . ◄

Acknowledgements

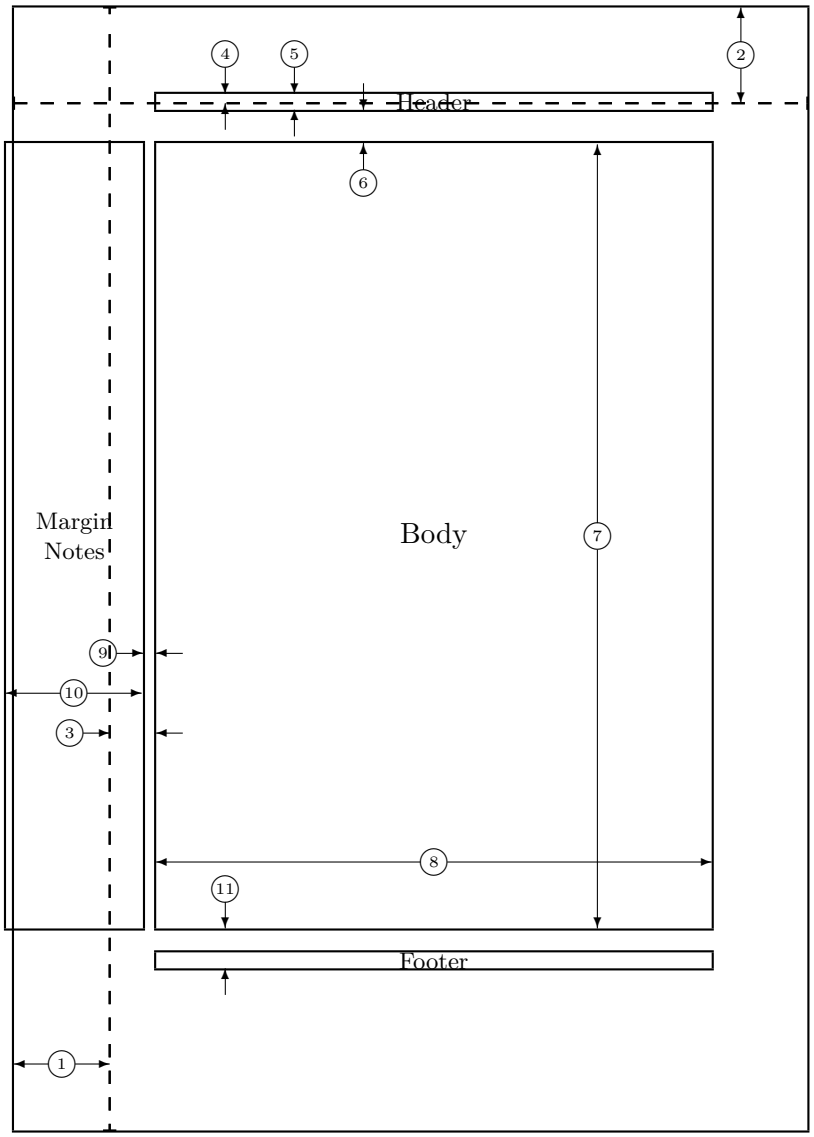
►. . . ◄

Anders Ingemann,
Aarhus, June 13, 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]◄

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?

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. This problem is solved by letting the client part of the web application compute responses where information on the server is not required. Queries to the server can happen both synchronously and asynchronously, meaning a function can either wait for the server response and block further execution of code until the response is available or it can define a callback function which is invoked once the response is available.

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

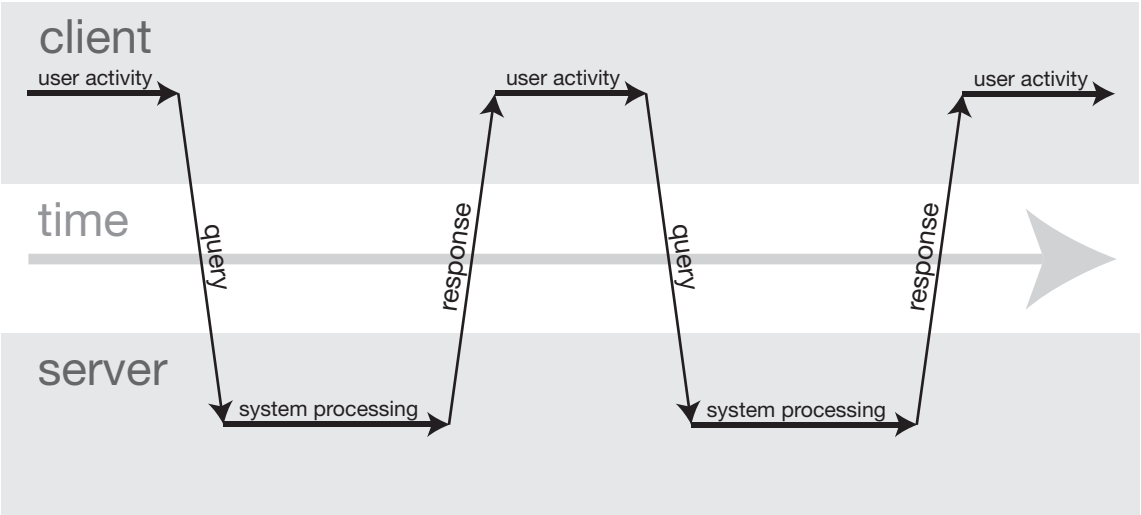


Figure 2.1: Diagram of synchronous communication between client and server

application. For simplicity however we will for the remainder of this thesis treat it as fact.

2.1.1 Server side web applications

Another way to incorporate business logic ►definition◄ and interactivity into a web application, is by rendering customized HTML pages on the server.
This has both advantages and disadvantages to the client-side scripting method:

- *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

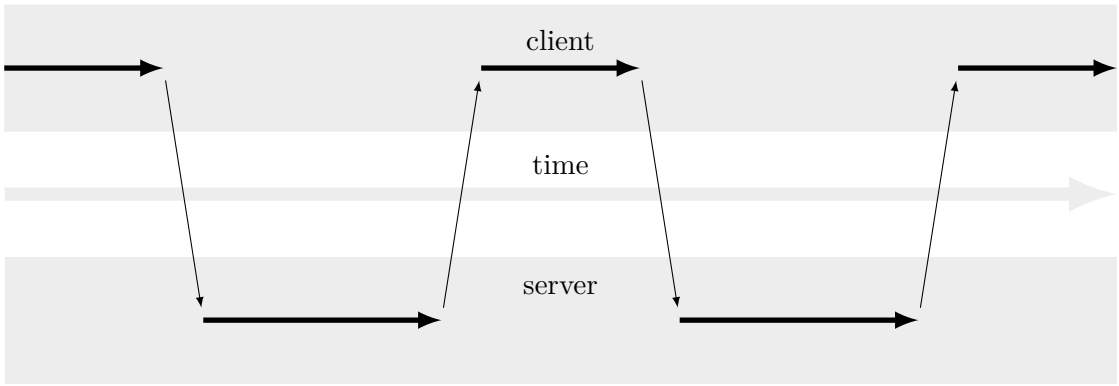


Figure 2.2: Diagram of asynchronous communication between client and server

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 on the server, because every software version and the software itself can be controlled by the developer.

2.1.2 Client side web applications

These arguments do not make the case for a server-only web application. They highlight the strengths of server side processing. A combination of server-side and client-side processing where their respective advantages are utilized and their drawbacks avoided, would in part help in creating an optimal web application.

►Explain why client side programming advantages are obvious. The above is mainly done to draw lines in the sand, between client and server side.◄

2.2 The development process

The development process of a web application is similar to any other software development process. 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

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

The information lost after a view is rendered might not be useful and the behavior of existing templating languages therefore inconsequential. This is not the case. 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. Add the requirement that this form is not to be submitted via a synchronous browser request but via AJAX. Now the developer has to reverse engineer the generated HTML with JavaScript to obtain the user information that is to be submitted. Any changes to the template will now also require a change in client side code, particularly the code, which finds the values in the form. This example contains a rather obvious loss of information. The position of the field attributes of the user object. At the time of the rendering 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.

►**Introduction to what I want to do about it**◄

4.1 Development work flow

The tool introduced in this thesis, will be developed via an iterative work flow. Two or three 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 5

Initial prototype

First we make a rudimentary proof of concept. It is an exploratory prototype, meaning none of its code or architecture is intended to be carried over into the next iteration. This prototype features an interactive application with which a movie library can be maintained. Details can be edited and actors can be added. It is not very useful in practice, but serves to make the basic idea more concrete in the following ways:

- *Explicit conceptualization*
  The prototype is meant to capture the core of the initial idea (to couple client models with server templates). Many of the less pronounced concepts of the 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 application is initialized with data from a MySQL database. The data is outputted to HTML via PHP. We do this by fetching the data from the database and printing it into HTML templates that have embedded PHP in them. As an example; what the template for a movie looks like can be seen in figure 5.

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

```
<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.

5.1 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 converts included less files into CSS.

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

- *underscore.js*
  Before jQuery was the standard prototype.js was a library developed by Sam Stephenson to improve upon the DOM API itself. It also brought with it various improvements to native JavaScript prototype objects. Underscore carries these improvements into the world of jQuery. underscore.js includes a small templating engine which will allow us to generate DOM elements and insert them into the page, without having to resort to page reloading.

- *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.

5.2 The application

The application features an rudimentary templates, which are note backed by any engine. Instead PHP is used directly in the HTML files.

5.2.1 Templates

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.

- *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 placed in the global scope with the `extract` method. For every class, that is used in the templating process, there exists at least on template. For two or more templates that use the same class, their names are disambiguated by appending a the name of the context each template is associated with (e.g. `movie.edit.tpl` and `movie.view.tpl`)

- *Collections*
  Collections (i.e. PHP arrays) are simply iterated through, in this prototype only objects are present in these arrays, the `foreach` block therefore contains the aforementioned method to place objects 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 functionality explored in this prototype, but are simply included and used to give the web application a more complete feel.

5.2.2 Parsing the HTML

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 which can be extended. The client side models have two static methods attached to them. `getAttrType()` takes an attribute name and returns the type of that attribute or hints at what should be done with the XPath result in order to obtain the attribute value. `getComplexType()` returns a reference to the object prototype that should be instantiated for a given attribute. The return values of `getAttrType()` are:

- *string*
  A simple string value.

- *node*
  A DOM node. This return value still represents a string, but in order to be able to post process an XPath result as a DOM node, the result type needs to be specified beforehand. **►how fucking stupid is that?! Worst. API. ever.◄**

- *model*
  A backbone.js Model. `getComplexType` is used here to instantiate the concrete Model.

- *collection*
  Collections are represented by extending "Backbone.Collection". Calling `getComplexType()` with the attribute name returns the prototype for the Collection tied to that attribute.

All models are located in `proof_of_concept/model`.

backbone.js also comes with a "Backbone.View" class. We extend this class to bind the models to elements in the DOM. This is done by listening to events fired by these elements and modifying the models correspondingly.

We attach a static field named "finders" to these ViewModels. This field contains objects defining where various attributes of the Model this ViewModel belongs to can be found (using the node this ViewModel is attached to).

To specify where the elements can be found we use XPath. If the result of a XPath query is too broad, a "process" attribute can be defined, containing a function, which processes the result and returns the desired attribute value.

An entry in the "finder" field can also contain a `view()` function which simply returns a prototype of the ViewModel that corresponds to the node the XPath points at. We use a function in this case, because not doing so would result in a circular dependency when loading, that can not be resolved.

ViewModels are attached to one node only, any element they govern is a child of that node.

parser.js

The `getViewModel()` function in the parser.js file is the heart of the prototype. Given a Model, a root node and a corresponding ViewModel, the parser iterates through the "finders" of the ViewModel. It returns an instance of that ViewModel with the root node attached to it and its "model" field pointing to an instance of the corresponding Model. For each key `getAttrType()` is called to determine how the result of the XPath query defined in the finder should be handled.

- *collection*
  The type of the Models in the Collection is retrieved with `getComplexType()` and the XPath query is executed.
  If the finder contains a "process" attribute defining a function, the return value of that function replaces the original XPath result. This replacement function is executed for all of the other procedures as well.
  A loop iterates through the result and recursively calls `getViewModel()` with the current value of the iteration, the return value of `view()` and the "model" attribute attached to the prototype returned by `getComplexType()`.

- *model*
  `getComplexType()` is called on the current model – like before – and a single node is retrieved from the XPath result. Again `getViewModel()` is called recursively with

the Model returned by `getComplexType()`, the ViewModel returned by `view()` and the single node from the XPath result.

- *node*
  The XPath result is executed and the "process" function is called with it as an argument. At this point the "process" function is crucial. An attribute of a model should contain a pointer to a DOM node, since it would only make sense in the context of the current page load and would only be understood by the actual ViewModel it belongs to.

- *string*
  The XPath is executed and the string is retrieved from the result.

All of these procedures convert a DOM node into a value usable in the context of a Model attribute. Once all of the attributes have been found a Model is instantiated and its attributes set. After that a ViewModel is instantiated with its "model" field pointing to the new Model instance and the root node pointing to the node passed to `getViewModel()`.

5.2.3 Using the results

►explaiiiin◄

5.3 Results

5.4 Plan for next iteration ►. . . ◄

While developing this application various libraries have been examined for their viability in the next iterations. 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. Only small modifications should be necessary to allow the View component to populate the Model component with data from the DOM.

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.