
Coupling Server-Side Templates and Client-Side Models

Anders Ingemann, 20052979

Digital version

Master's Thesis, Computer Science

December 2012

Advisor: Michael Schwartzbach

Abstract

►in English...◄

Resumé

►in Danish...◄

Acknowledgements



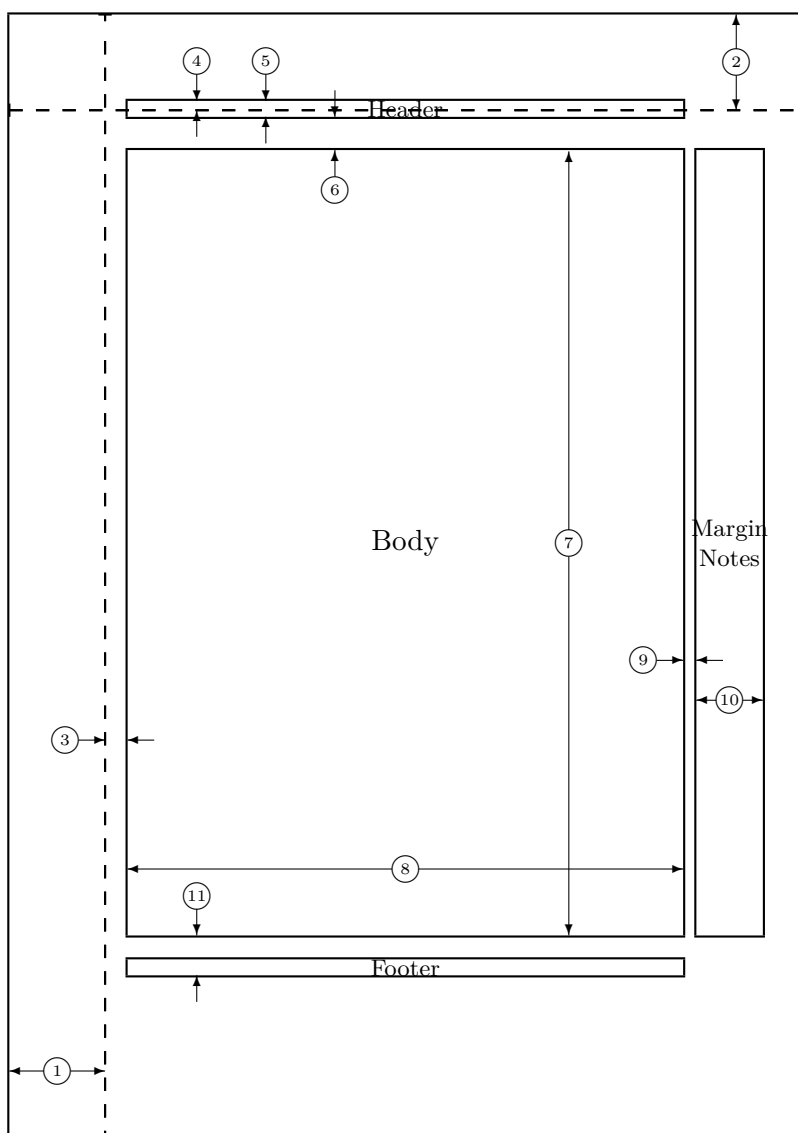
*Anders Ingemann,
Aarhus, December 27, 2012.*

Contents

Abstract	ii
Resumé	iii
Acknowledgments	1
Contents	2
1 Introduction	6
1.1 Development work flow	6
2 Developing web applications	7
2.1 What are web applications?	7
2.1.1 Synchronous and asynchronous communication	7
2.1.2 Javascript alternatives	8
2.1.3 Server-side web applications	8
2.1.4 Client-side web applications	10
2.1.5 Combining the strengths	10
2.2 The development process	11
2.2.1 Usual design patterns	11
3 Tools of the trade	13
3.1 Client frameworks	13
3.1.1 Backbone.js	13
3.1.2 XPath	13
3.2 Templating languages	13
3.2.1 Mustache	14
3.2.2 Limitations	19
4 Requirements	20
5 First iteration - The initial prototype	22
5.1 Architecture	22
5.1.1 Libraries	23

5.1.2	Templating	23
5.1.3	Models and ViewModels	25
5.1.4	Collections	25
5.2	Coupling client-side and server-side models	25
5.2.1	Parsing the DOM	26
5.3	Results - Plans for next iteration	27
6	Revised Requirements	29
6.1	Additional requirements	29
6.2	Simplifying the project	30
6.2.1	Server-side	30
6.2.2	Client-side	30
6.2.3	The consequences of our simplification	31
6.2.4	chaplin	32
6.2.5	Data types	33
7	Final Architecture	34
7.1	Compiling template information	34
7.2	Communicating with the client	35
7.3	Alternatives	36
7.3.1	Integrating with the template engine	36
7.3.2	Decorating templates	36
8	Second iteration - Implementation	37
8.1	Parsing mustache templates	37
8.1.1	Mustache EBNF	37
8.1.2	Mustache-XML EBNF	37
8.1.3	Alternative parsing strategies	40
8.2	Mustache-XML DOM Paths	41
8.2.1	Resolver	41
8.2.2	Lists of numbers as paths	41
8.3	Variable boundaries	42
8.3.1	Other boundaries	43
8.4	Recognizing iterations	43
8.4.1	Content list	43
8.4.2	Lambda sections	43
8.4.3	If-else constructs	44
8.5	Outputting information	44
8.6	Client library	44
8.6.1	Technology choices	44
8.6.2	Input	44
8.6.3	Representing mustache tags	45
8.6.4	Parsing sections	46
8.6.5	Joined text nodes	46

8.6.6	Verifying nodes	47
8.6.7	Returning values	48
8.7	“Comb”	48
9	Demo application	49
9.1	Goal	49
9.2	The application	49
9.2.1	Rendering a template	49
9.2.2	Displaying template information	49
9.2.3	Parsing the form	51
9.2.4	Loopbacking Comb	51
10	Evaluation	52
10.1	Limitations	52
10.1.1	Removal of standalone lines	53
10.2	Challenges	53
10.3	Advantages	53
10.3.1	Comparison	53
11	Future Work	54
12	Related Work	55
13	Conclusion	56
A	Technical Appendix	57
A.1	Choosing the tools	57
	Bibliography	58



- | | | | |
|----|-----------------------|----|----------------------------------|
| 1 | one inch + \hoffset | 2 | one inch + \voffset |
| 3 | \oddsidemargin = 17pt | 4 | \topmargin = -7pt |
| 5 | \headheight = 12pt | 6 | \headsep = 25pt |
| 7 | \textheight = 591pt | 8 | \textwidth = 418pt |
| 9 | \marginparsep = 10pt | 10 | \marginparwidth = 50pt |
| 11 | \footskip = 30pt | | \marginparpush = 5pt (not shown) |
| | \hoffset = 0pt | | \voffset = 0pt |
| | \paperwidth = 597pt | | \paperheight = 845pt |

Chapter 1

Introduction



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

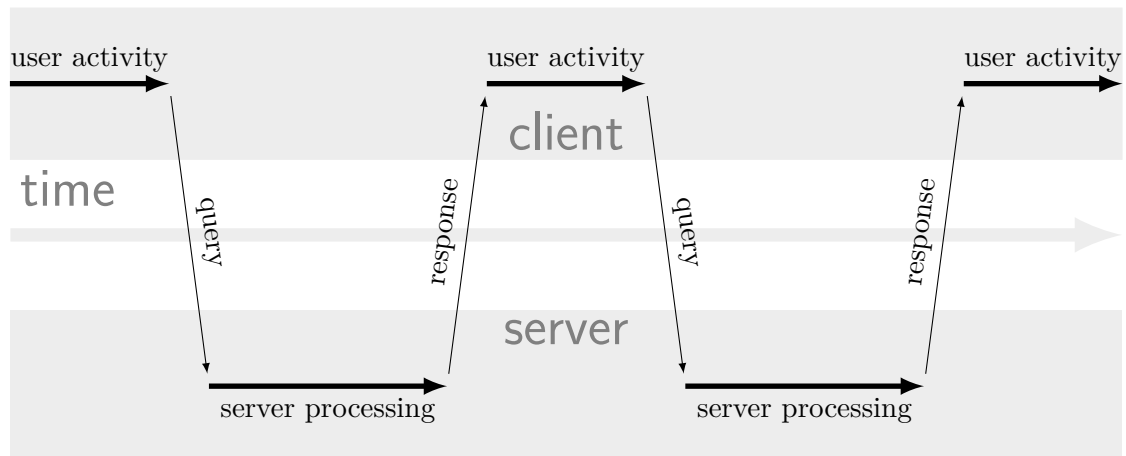


Figure 2.1: Diagram of synchronous communication between client and server

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 Javascript alternatives

►Needs a better title, last paragraph is a non sequitur◄ 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

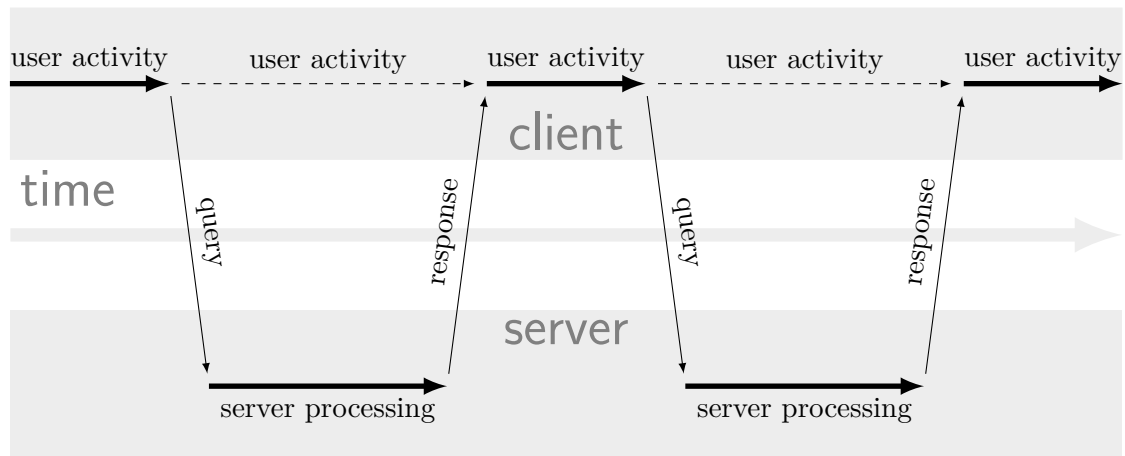


Figure 2.2: Diagram of synchronous communication between client and server

on the server can be advantageous in a number of situations:

- *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>◄). 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 advanced control flow statements. Instead there are only tags. Tags in this context can be understood as an advanced form of placeholders. Some tags are replaced with a string, some are replaced with HTML tags, yet others repeat their content multiple times.

Tags are easily recognizable by their delimiters which always start and end with two curly braces (e.g. `{{identifier}}`).

To render a mustache template, the template in conjunction with an identifier mapping (i.e. the dataset) is passed to the engine, which returns the rendered template.

Variables

Variables in mustache are string placeholders. We can recognize them by their two curly braces before and after an identifier (`{{identifier}}`). The template engine replaces these tags with the corresponding value in the dataset. The identifier in the tag points at a key in the dataset with the same name. Normal variables are HTML escaped. The engine outputs an empty string if the key is not specified in the dataset.

An identifier may also point at properties on objects in the dataset. Like many other languages the `.` character is used for that purpose.

Unescaped variables

To output strings as unescaped HTML, triple curly braces are used: `{{{identifier}}}`. Alternatively an ampersand may be used as well: `{{&identifier}}`.

Sections

Sections in mustache can be considered equivalent to foreach loops in other languages. We write them with an open and close tag: `{{#identifier}}CONTENT{{/identifier}}`. Depending on the dataset value the identifier points at, a section may also behave like an if-block, which is rendered if the value is true. The values that are considered true and false depend on the language specific implementation of the engine.

if the data is truthy (e.g. `!!data == true`), use a single-element list containing the data, otherwise use an empty list. [1, sections.yml]

If the value is a list, the contents of the section are rendered once for each entry in the list. Once a section is entered, the key-value pairs of the corresponding entry are pushed onto the stack of identifiers. Consider the example in figure 3.1. Here the placeholder "nickname" is used in two different contexts. The first usage occurs while the identifier stack is only one level high. The second usage occurs inside a section. Here the "nickname" identifier refers to the "nickname" key of the corresponding dataset entry in the list of messages. If we were to insert a `{{messagecount}}` tag inside the section, the output would be "2" for every iteration.

Figure 3.1: An example of a mustache section

```
1 | Template |
2 <p>
3   Hello {{nickname}},<br/>
4   you have {{messagecount}} new messages:
5 </p>
6 <ul>
7   {{#messages}}
8   <li>{{subject}} from {{nickname}}</li>
9   {{/messages}}
10 </ul>
11 | Dataset |
12 nickname: "andsens"
13 messagecount: 2
14 messages:
15 0:
16   subject: "How did the presentation go?"
17   nickname: "carl"
18 1:
19   subject: "Welcome to Messageservice Inc."
20   nickname: "Messageservice Inc."
21 | Result |
22 <p>
23   Hello andsens,<br/>
24   you have 2 new messages:
25 </p>
26 <ul>
27   <li>How did the presentation go? from carl</li>
28   <li>Welcome to Messageservice Inc. from Messageservice Inc.</li>
29 </ul>
```

Figure 3.2: Mustache template using the ‘.’ variable

```
1 | Template |
2 <ul>
3   {{#messages}}
4   <li>Message {{.}}: {{subject}} from {{nickname}}</li>
5   {{/messages}}
6 </ul>
7 | Result |
8 <ul>
9   <li>Message 0: How did the presentation go? from carl</li>
10  <li>Message 1: Welcome to Messageservice Inc. from Messageservice Inc
11  .</li>
12 </ul>
```

Figure 3.3: Inverted sections in mustache templates

```
1 {{#messages}}
2 <li>{{subject}} from {{nickname}}</li>
3 {{/messages}}
4 {{^messages}}
5 <li>You have no messages</li>
6 {{/messages}}
```

Mustache¹ also allows the key of the current iteration to be referenced with a “.” (see figure 3.2).

Inverted sections

In case a value in the dataset is an empty list or false, we use an inverted section to output fallback content. We simply replace the hash-mark in the opening tag of a section with a caret (see figure 3.3).

We can exploit this behavior to create equivalents of if-else blocks. Using the above template we can allow users to leave the subject line empty and display a message accordingly.

The `{{subject}}` variable in figure 3.4 could also have been replaced with a `{{.}}`. As mentioned earlier, variables are replaced with empty strings if they do not correspond to an entry in the dataset. This fact allows us to shorten the template even further as can be seen in figure 3.5. We do not need to check if a variable is not the empty string before outputting it. The effect of outputting it regardless of its content is the same.

¹excluding mustache.js, which may have some implementation problems, see <https://github.com/janl/mustache.js/issues/185>

Figure 3.4: If-else constructs with sections in mustache templates

```
1 <li>
2   {{#subject}}{{subject}}{{/subject}}
3   {{^subject}}(No subject){{/subject}}
4   from {{nickname}}
5 </li>
```

Figure 3.5: Variables output empty strings in mustache, if there is no values in the dataset for them

```
1 <li>
2   {{subject}}{{^subject}}(No subject){{/subject}}
3   from {{nickname}}
4 </li>
```

Lambdas

Mustache tags identified by the hash mark may also represent another form of placeholder namely the lambda sections. These sections are not iterations, but calls to functions that have been bound to keys in the dataset. The functions are called with the contents of section as their parameter, the output of a lambda section is the return value of the function call.

Comments

A comment in mustache is identified by an exclamation mark: `{{! This is a comment }}`
The comment tag creates no output.

Partials

Partials allow templates to be split up into smaller parts. Their tags are replaced with the contents of other templates. They behave as if the referenced template was inlined directly at the location of the partial tag. The contents of the partial templates have to be passed to the rendering engine together with the main template and the dataset.

The inlining happens when a template is rendered. This enables mustache to render recursive data-structures (see an example in figure 3.6).

Set delimiter

Mustache curly braces may clash with delimiters used in other languages, such as LaTeX. Using the "set delimiter" tag, the delimiter can be changed to something else (see figure 3.7).

Figure 3.6: Mustache templates can be recursive by utilizing partial tags

```
1 | Filesystem template |
2 {{! this template is mapped as a partial named "filesystem" }}
3 <ul>
4   {{#directories}}
5   <li>{{name}}: {{>filesystem}}</li>
6   {{/directories}}
7   {{#files}}
8   <li>{{name}}</li>
9   {{/files}}
10 </ul>
```

Figure 3.7: Usage of set delimiter tags in mustache templates

```
1 {{ default_tags }}
2 {{=<? ?>=}}
3 <? php_tags ?>
4 {{ This will be normal text enclosed by curly braces }}
5 <?={{ }}=?>
6 {{ default_tags }}
```

Removal of standalone lines

Mustache removes lines which beyond a section beginning or end contains only whitespace (i.e. standalone lines). This is done to match the expectation a user may have when writing templates.

The behavior is however not defined clearly:

Section and End Section tags SHOULD be treated as standalone when appropriate. [1, sections.yml]

Mustache specification

Documents specifying the exact behavior of mustache can be viewed at <https://github.com/mustache/spec>. The specification is written in YAML to allow automated tests of rendering engines.

3.2.2 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 still not 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

First iteration - The 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 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.

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

```
1 <li id="movie-<?php echo $id ?>">
2   <details>
3     <summary class="title"><?php echo $title; ?></summary>
4     <header>
5       <section class="operations">
6         <span class="edit command">Edit</span>
7         <a>Delete</a>
8       </section>
9     </header>
10    <div class="main">
11      <h1><?php echo $title; ?></h1>
12      (<span class="year"><?php echo $year; ?></span>)
13      <div class="poster">
14        <?php echo $poster; ?>
15      </div>
16      <details class="plot">
17        <summary class="synopsis">
18          Synopsis: <span><?php echo $synopsis; ?></span>
19        </summary>
20        <p><?php echo $plot; ?></p>
21      </details>
22      <table class="cast">
23        <thead>
24          <tr>
25            <td>Actor</td>
26            <td>Character</td>
27          </tr>
28        </thead>
29        <tbody>
30          <?php foreach($cast as $role) {
31            $actor = $role->actor;
32            extract($role->to_array());
33            require 'view/types/movie/role.tpl';
34          } ?>
35        </tbody>
36      </table>
37      <span id="add_actor_button" class="command">add</span>
38    </div>
39  </details>
40 </li>
```

- *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 XPath's 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 then 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

Revised Requirements

In this chapter we will analyze which parts of our exploratory prototype we can utilize in our next iteration. To that end we will examine every tool used and arrive at a subset of these tools, which we will complement with a fresh set of parts. Our goal is to have a plan laid out for the next iteration at the end of this chapter.

The prototype featured a large amount of moving parts that were constructed for the occasion. Among others this includes our templates, that were not mustache templates but simple repeatable patterns in PHP. The recursive tree parser, building a set of models and views for us to use on the client side is another example of an ad-hoc constructed tool.

These parts bring with them their own set of problems and bugs. Since they are custom developed in a limited time frame they will have coding errors other seasoned related tools do not have. Seeing how the last iteration resulted in a prototype, we decided to develop these tools regardless, because evaluating alternatives that would fit the purpose precisely would have taken up more time. This is however not a sound strategy going forward, assuming we have the goal to develop a reliable tool to realize our concept of binding server-side templates to client-side models. Succinctly put: There is no reason to reinvent the wheel. Most of these parts have nothing to do directly with the concept of this thesis. They are rather tools that help achieve the goals of it. For a proper tool, that we can consider usable, to emerge from our process, we will need to reduce the amount of said custom parts. To that end we will first have to identify the superfluous parts of the prototype, that can be replaced by existing well maintained tools. Once we have achieved this, we can begin concentrating on the core of our concept and define it with greater precision.

6.1 Additional requirements

In the prototype we omitted some parts of the concept architecture by "faking" them. This holds true especially, when we look at how the information gained by parsing the server templates is transmitted to the client. The information itself is hard coded into the client, meaning both the part for parsing templates and the matter of client

communication is missing.

The traversal of the DOM happened in a very limited way, where the parser was finely tuned to parse only our specific case but nothing more. We will have to analyze which, if any, of these gaps can be filled by generic tools and which gaps belong to the core of our concept.

Some gaps may not even need to be filled. This will surely be the case, when we consider that the desired end result is a tool and not a framework. Many frameworks and tools in the prototype were used to illustrate the concept via a demo application. There are a lot more parts needed to create a working web-application than there are to create a tools for said application.

6.2 Simplifying the project

Bear in mind that despite the following simplifications we may still use some of the tools. We intend our tool to perform in an ecosystem of other software, which can integrate loosely with our tool instead of requiring deep integration with it.

6.2.1 Server-side

We begin our simplification on the server. Here we communicated with a database to persist our movies, actors etc. in the MySQL database. The database and the object relational mapper (ORM) php-activerecord are not at all necessary for our tool to work. They are interchangeable with any other type of software, that can persist data on the server. Our concept should work with even ephemeral data.

Our server-side language of choice - PHP -, also belongs to this category. The server could have been written in any other server-side language. As a consequence, the template engine (mustache), will of course need to be able to interface with that language. For the prototype we omitted these templates and wrote them directly in PHP instead.

Our plan however is to write the server-side templates in mustache in the next iteration. Locating placeholders in these templates and outputting their location is the solution we proposed in the beginning of this thesis to the problem we identified with server side templates. By extension parsing server side templates pertains to the core of our concept. Since parsing arbitrary template syntaxes, would go out of the scope of this thesis we must conclude that mustache belongs to the category of tools that cannot be removed.

6.2.2 Client-side

The client-side tools we have used in our prototype interact with the data we "retrieved" (remember: we did not actually retrieve any template information) from the server. This makes the setup of the client more intricate. We will have to look carefully at each tool and determine by the nature of its interaction with that data, whether it is a crucial part of our concept.

Regardless of which tools we remove, we must remember that the information about our server-side templates must be used somehow. This would suggest that the client can have more than one structure and set of interconnected parts, which leverage the additional information.

- Beginning with the periphery, we can easily see how a framework to ease the development of CSS is not part of our tool.
- The Javascript language is required on the basis that we need some form of client side programming. We have discussed its alternatives earlier in this thesis and are going to keep this choice.
- underscore.js helps us to iterate through arrays and manage other operations more easily than in pure Javascript. We can solve the same problems without it (although it requires more effort). This makes underscore.js a non-crucial part of our tool.
- jQuery is used to retrieve the values from the DOM. We may later change how we retrieve those values, but some method of retrieval will be needed. At this point jQuery seems to fit the bill and we must consider jQuery a crucial part of our tool.
- We use backbone.js to hold the values we retrieve from the DOM. The framework enables us to interact with these values. They can however also be modeled with simple JavaScript objects. Because of that backbone.js can not be considered a crucial part of our tool.

This concludes the evaluation of tools we used in our prototype. There is still the matter of categorizing our own custom DOM traverser, which takes a root model as input and recursively reads the nested values in the DOM. During this traversal it also creates Models attached to corresponding ViewModels. Using this strategy we obtain a consistent and predictable control flow of the initialization process. The disadvantages include rigidity in the way the placeholder information from the server is used and a need to specify what type of view each placeholder represents. We desire a decentralization of responsibility and a more liberal use of the template placeholder data. This means that we must replace the parser with something else that is not a substitute, but a rethinking of its role in the process of utilizing the placeholder data.

6.2.3 The consequences of our simplification

Let us summarize what these changes mean for our next iteration.

We have kept quite a few client libraries and those we deemed non-essential were only discounted to reduce complexity. The only client side code we decided to remove, was the one we wrote ourselves. Similarly we removed from the server-side our ad-hoc implementation of a template engine. We deemed the ORM and the database backing inconsequential to our tool, although many fully fledged web applications of course can not function without it.

These reductions leave us with a clear picture of which tools we carry over into the next iteration. On the server-side we will only need the mustache template engine and our parser, which we will create in the next iteration. On the client-side we have a collection of libraries that when combined gives us part of the architecture that will allow us to utilize the information generated by the server. There are however still gaps in this architecture. We require a framework, that allows us to organize our models and views properly. It should also support the addition of our discarded parsing feature, by supplying a structure to bind it to. "Chaplin" will in this case be our framework of choice.

►Figure out if we want to fight the fight and call the view viewmodel◄

6.2.4 chaplin

Chaplin is a new client side framework, which was created in February 2012. The motivation behind it was to create a framework that allows developers to follow a set of conventions more easily. Backbone.js has both views and models (and routes, for controllers), but does not force any specific way of structuring code. In this respect Backbone.js can be seen more as a tool than a framework. Chaplin extends the models and views from Backbone.js and adds more features. It introduces concepts such as "subviews" - views that aggregate other views. This allows the developer among other things to better mirror the structure of the DOM.

The framework also allows the developer to use any template engine he desires. The engine simply needs to return an object, which jQuery can append to the wrapping DOM element of the view.

A very useful feature of Chaplin is the automatic memory management. When creating single page web-applications, the developer has to dispose each view manually. This challenge is best illustrated with regard to eventhandlers. Eventhandlers are functions, that are called when an event on a DOM node or an other object is trigger. Often this function manipulates and accesses properties stored on a view. To allow for this access, the function stores a pointer to the view via a closure. Since the function is stored with the DOM node or object on which it is listening for events, any view the developer wants to dispose needs to stop listening on those events as well. Chaplin unbinds these eventhandlers for the developer when the view is disposed, allowing the browser to free up memory.

The framework has however a major drawback: A view does not add subview elements to its DOM tree by means of the template function. Instead the developer must use jQuery to append these elements to an element in the DOM. Supposing that these subview elements are not attached directly beneath the root element of the view, the developer will need to traverse parts of the DOM tree with a jQuery selector to find the correct position to attach a subview element. This breaks the fundamental principle of dividing the view(-model) from the template data. A view now has to hold information about the layout of the DOM, it is hard-coded into the view.

Our tool can solve this problem by supplying the view with those selectors. The only requirement would be that the developer wrote placeholders into the templates that

identified subviews of the corresponding view. The view can use the selectors generated for those placeholders to pinpoint the exact location where a subview element should be attached.

A quicker solution to this problem would be to insert these placeholders and modify the template engine to understand the concept of subviews. Unfortunately there is a major drawback to that strategy. Every subview will be inserted into the DOM when the parent view is rendered, even though the developer may wish to delay that insertion. Also, the attachment of subviews that are yet to be created requires a rerendering of the entire DOM sub-tree.

We intend to modify Chaplin to take advantage of these placeholder selectors automatically. The developer should not be required to interact with the selectors.

6.2.5 Data types

In the first iteration we not only retrieved the values located in the DOM but also specified what types these values were. In fact, retrieving data from the DOM and mapping data fields to types are two very different processes. We will not try to map data to types or Models in the next iteration, it would exceed the scope of our tool and expose the user to a layer of abstraction that may not be desired.

In section 5.3 of chapter 5 we recognized the need for knowing the types of data. This is not in conflict with this decision, we are still interested in knowing when data (or rather a mustache tag) behaves recursively, so that we may take appropriate steps to parse a rendered template accordingly. This holds true specifically for mustache partials, which can refer to the template they are specified in.

Removing the data typing feature may still reduce the immediate usefulness of our tool. The removal increases the applicability of it however. A mapping layer may still be put on top of our tool to abstract raw data into backbone models. Such a mapping layer could also be a form validation tool or a custom data model to abstract a very specific use case. This way we construct a tool instead of a framework.

►What about our initial goal, reading the information from the DOM◄
►What about not rerendering ANYTHING? modelbind changes and replace the corresponding nodes◄

Chapter 7

Final Architecture

Our goal is to create a tool that allows the user to pass a rendered template (DOM) to it and receive a data structure that equates the original dataset passed to the template engine. Parsing can be a process that requires a lot of processing power. We do not want our tool to slow down the web application every time a new template is rendered and the values are retrieved on the client. To minimize the effort required to parse a rendered template we try to compile as much information about a template as possible before it is rendered. Using this information we should be able to parse a rendered template more quickly. This strategy implies a pre-parsing step that outputs data which aids the client library in the parsing process. Since this is an operation that only needs to be run before a web application is deployed, we are not constrained by the environment the actual web server runs in. The running time of our tool +will only a minor concern.

7.1 Compiling template information

We need to be able to parse a mustache template and acquire the necessary information to enable the client library to parse a rendered template. We analyze the capabilities of the mustache template language and arrive at the following pieces of information, which we can deduce before the template is rendered.

- The location of variables in the document
- Whether a variable is escaped or unescaped (`{{identifier}}` vs. `{{{identifier}}}`)
- The location of sections in the document
- Whether a section is inverted (`{{#identifier}}` vs. `{{^identifier}}`)
- The contents of a section
- The location of partials in the document
- The location of comments in the document

Because of the nature of mustache templates we can however not retrieve the following data:

- The contents of variables (including their type, e.g. integer, string)
- The number of iterations a section will run
- Whether a section is a loop or an if block (except in the case of an inverted section)
- Whether a section is a lambda section or an actual section.
- The behavior of a lambda section.
- The template a partial points at
- Whether an identifier refers to a key in the current stack level or a to a key in one of the lower stack levels.

The rendered template is in the Document Object Model format, once it has been rendered by the client. We can still access the rendered template as a string after it has been inserted by accessing the `innerHTML` property on the element node it was inserted into. This has one major drawback. The browser does not return the actual string that was inserted but rather a serialized version of the DOM. This is demonstrated quite easily by executing the following lines in the Google Chrome Developer Console:

```
1 var div = document.createElement("div")
2 div.innerHTML = "<img/>"
3 div.innerHTML
```

The last line does not return the string "`<img/ >`" but "``". In Firefox the last line returns "``". This means that the rendered templates can not be parsed reliably by using the `innerHTML` property.

However, no matter which way the browser decides to represent an HTML tag, we can still rely on the ordering of tags and on the names of tags to be the same (case sensitivity can be avoided with a simple `element.tagName.toLowerCase()` when comparing tag names client side). We therefore opt to relate the information about mustache tags to the DOM instead.

This choice requires our pre-parsing tool to be able to understand HTML documents and construct a data structure resembling the DOM in which the mustache tags can be located and their location converted into a DOM path.

7.2 Communicating with the client

The information gathered by the pre-parser is saved in files next to the original templates. The user may choose whatever technology fits best to transfer this information to the client library. All the library should expect is a DOM node with the rendered template as its children and the information created by our pre-parser.



By choosing this approach we leave the user with many optimization possibilities.

- Gzip templates and template information statically to optimize server performance.
- Prepend template information to the template and subsequently cut it out before passing the template to the template engine. This way only one file needs to be handled and transferred.
- Use require-js to load templates and their information in parallel while developing and inline both template and template information into one big JavaScript file when deploying.

7.3 Alternatives

Instead of using a pre-parser, we could also take other paths to help the client library in parsing rendered templates.

7.3.1 Integrating with the template engine

In chapter 4 we touched upon the fact that it was really the template engine that discarded the information and only outputted the rendered template. We could choose to modify the template engine to output that exact information. This poses a rather big challenge: For mustache templates there exists no such thing as *the* template engine. Currently the mustache website lists mustache engines in 29 different programming languages. (<https://github.com/defunkt/mustache/wiki/Other-Mustache-implementations>  **ref** ) This fact makes the goal of such an approach very hard to achieve.

7.3.2 Decorating templates

The user could decorate mustache template tags with specific HTML tags that have no effect on the visual layout but can be retrieved by the library. This would make the client side code a very lightweight value retriever and it obsoletes the pre-parser.

Apart from simply shifting the workload of locating template variables from the viewmodel maintainer to the template maintainer¹, the task of retrofitting existing web applications becomes much greater.

¹The scenario in chapter 4 illustrates this point very well

Chapter 8

Second iteration - Implementation

In this chapter we will walk through the implementation of the pre-parsing tool and the client library. We begin by detailing the process of parsing mustache templates.

8.1 Parsing mustache templates

► **Think of a reason for haskell other than "I wanted to try it"** ◀

Our language of choice for implementing the parser is Haskell. We utilize the Parsec parser combinator library to analyze our mustache templates. With Parsec we can convert an EBNF grammar very effortlessly into Haskell code by using the combinators and parsers the library supplies us with.

8.1.1 Mustache EBNF

The EBNF for mustache is fairly simple and can be seen in figure 8.1. The behavior of `set_delimiter` tag is ignored in this EBNF.

8.1.2 Mustache-XML EBNF

We want our parser to not only be able to understand mustache grammar, but also HTML grammar intermingled with it.

There are many flavors of HTML we may choose from to allow in our mustache-HTML grammar. To simplify our approach we will only allow well structured XML tags, this should ostensibly cover most of HTML. HTML 5 allows for self-closing tags on void elements, which with our choice is not something we can support. We will therefore refer to HTML tags in our templates as XML tags.

We build an abstract syntax tree with Parsec, in order for our tool to be able to create DOM paths through this tree. The EBNF in figure 8.2 represents the structure our parser understands.

The mustache comment tag has been left out in this grammar, since it does not output any content our client library can retrieve. We did also not include the `set_delimiter` tag. This was mostly done to keep our first implementation of the tool simple.

Figure 8.1: Mustache EBNF

```

<variable>      ::= ‘{{{’ <ident> ‘}}’ | ‘{{&’ <ident> ‘}}’ | ‘{{’ <ident> ‘}}’

<section>       ::= ‘{{#’ <ident> ‘}}’ <content>* ‘{{/’ <ident> ‘}}’
                  | ‘{{^’ <ident> ‘}}’ <content>* ‘{{/’ <ident> ‘}}’

<partial>       ::= ‘{{>’ <ident> ‘}}’

<comment>       ::= ‘{{!’ <comment> ‘}}’

<set_delimiter> ::= ‘{{=’ <delim_start> ’ ’ <delim_end> ‘=}}’

<tag_or_char>   ::= <section>
                  | <partial>
                  | <comment>
                  | <set_delimiter>
                  | <variable>
                  | <char>

<content>       ::= <tag_or_char>*

```

XML Comments

An XML comment is recognized as simple text in the DOM of the browser. The EBNF still allows for mustache tag structures. This allows the user to communicate additional information to the client, without showing it in the browser.

Template constraints

Note that the EBNF restricts the types of templates our tool can parse.

- XML tags must be closed in the same template and section as they are opened
- Sections must adhere to the same structure as XML tags¹.
- Variables and section may not exist in the identifier part of an XML tag or attribute

A user may not run into these structural restrictions very often. Regardless, they give our client library very useful guarantees about the rendered templates it passes. They also allow use to create proper abstract syntax trees for any template. If XML tags were to be opened outside the scope of a template and closed in the template we are parsing, there would be no way to determine the location of mustache tags in the DOM without performing complicated cross-references with the template that opened these tags.

¹ This also means that sections may not interleave, something mustache does not support in any case

Figure 8.2: Mustache-XML EBNF

$\langle variable \rangle$	$::= \text{'\{\{\}' \langle ident \rangle \}\{'\}}' \mid \text{'\{\{\&' \langle ident \rangle \}\{'\}}' \mid \text{'\{\{' \langle ident \rangle \}\{'\}}$
$\langle partial \rangle$	$::= \text{'\{\{>' \langle ident \rangle \}\{'\}}$
$\langle content_section \rangle$	$::= \text{'\{\{\#' \langle ident \rangle \}\{'\}}' \langle template_content \rangle^* \text{'\{\{/ ' \langle ident \rangle \}\{'\}}' \mid \text{'\{\{\^ ' \langle ident \rangle \}\{'\}}' \langle template_content \rangle^* \text{'\{\{/ ' \langle ident \rangle \}\{'\}}$
$\langle attribute_section \rangle$	$::= \text{'\{\{\#' \langle ident \rangle \}\{'\}}' \langle attribute_content \rangle^* \text{'\{\{/ ' \langle ident \rangle \}\{'\}}' \mid \text{'\{\{\^ ' \langle ident \rangle \}\{'\}}' \langle attribute_content \rangle^* \text{'\{\{/ ' \langle ident \rangle \}\{'\}}$
$\langle comment_section \rangle$	$::= \text{'\{\{\#' \langle ident \rangle \}\{'\}}' \langle comment_content \rangle^* \text{'\{\{/ ' \langle ident \rangle \}\{'\}}' \mid \text{'\{\{\^ ' \langle ident \rangle \}\{'\}}' \langle comment_content \rangle^* \text{'\{\{/ ' \langle ident \rangle \}\{'\}}$
$\langle content_mustache_tag \rangle$	$::= \langle content_section \rangle \mid \langle partial \rangle \mid \langle comment \rangle \mid \langle variable \rangle$
$\langle attribute_mustache_tag \rangle$	$::= \langle attribute_section \rangle \mid \langle partial \rangle \mid \langle comment \rangle \mid \langle variable \rangle$
$\langle comment_mustache_tag \rangle$	$::= \langle comment_section \rangle \mid \langle partial \rangle \mid \langle comment \rangle \mid \langle variable \rangle$
$\langle attribute \rangle$	$::= \text{' ' \langle ident \rangle '=' \langle attribute_content \rangle^* \text{' '}}$
$\langle xml_tag \rangle$	$::= \text{'<' \langle ident \rangle \langle attribute \rangle^* '>' \langle template_content \rangle^* '</' \langle ident \rangle '>' \mid \text{'<' \langle ident \rangle \langle attribute \rangle^* '/>' \mid \text{'<!--' \langle comment_content \rangle^* '-->'}$
$\langle attribute_content \rangle$	$::= \langle attribute_mustache_tag \rangle \mid \langle char_without_doublequote \rangle$
$\langle comment_content \rangle$	$::= \langle comment_mustache_tag \rangle \mid \langle char \rangle$
$\langle template_content \rangle$	$::= \langle content_mustache_tag \rangle \mid \langle xml_tag \rangle \mid \langle char \rangle$
$\langle content \rangle$	$::= \langle template_content \rangle^*$

Character References

The EBNF omits character references (e.g. ` `, `å`). When those character references are accessed via the DOM in the browser they are returned in their interpreted form. This forces our tool to also be able to understand character references. To that end we simply scan any text we have recognized between tags for ampersands, all characters from that point on until a semicolon is found are passed to the `lookupEntity` function available in the TagSoup library ([►Ref to Text.HTML.TagSoup.Entity◄](#)). The `lookupEntity` function converts XML character references to UTF-8 characters.

Lexeme token parsers

Parsec can create token parsers given a configuration with definitions of allowed operator letters, reserved operator names, legal identifier letters and many other pieces of information that are useful for parsing tokens in a language. The token parsers returned by Parsec are lexeme token parsers. These token parsers consume any whitespace that follow most tokens. They also throw errors when tokens are followed by operator letters.

Significant white spaces

In the case of our template parser, the otherwise advantageous properties of lexeme token parsers are not desirable. White spaces in the beginning of an attribute value or after an XML tag can be very significant. We will need to be sure when a mustache variable begins and ends. If a variable is surrounded only by white spaces, our tool will convey data to the client which details that there is in fact no white space. Subsequently the client will assume the white spaces recognized in the rendered template belong to the value of the variable.

8.1.3 Alternative parsing strategies

Instead of Parsec we could have chosen an existing parsing technology for XML and simply extended it.

HXT (`Text.XML.HXT`)

Haskell XML Tools (HXT) is a very advanced XML parser utilizing, amongst other Haskell concepts, arrows. It is intended for querying structures the tool creates by parsing the XML. Using it to discover the structure of documents is not its main purpose. With this tool, we would have to create a new structure on top of the existing HXT XML structure.

XML (`Text.XML.Light`)

XML is an easy-to-use XML library, which exports its data constructors. This allows us to pattern match and use deconstruction assignments in our functions. Mustache tags have to be recognized by inspecting all strings in the structure we receive. After

recognizing the tags, we have to overlay the existing XML structure with section beginnings and ends and variable and partial locations. These overlay techniques would very quickly dwarf the 250 lines of code our Parsec parser spans now.

8.2 Mustache-XML DOM Paths

The abstract syntax tree our parser generates bears some resemblance to the Document Object Model available in the browser. There is however the addition of mustache section nodes and mustache variable/partial leafs. Once the template is rendered, a mustache section will not be visible and the contents of that section will be joined with the siblings of said section. Similarly, mustache variables output text which will be joined with neighboring text nodes. When constructing DOM paths this fact has to be taken into account.

8.2.1 Resolver

Our tool passes the abstract syntax tree generated by the parser into a resolver. The resolver links mustache tags together and analyzes dependencies between them by creating “Resolutions”. These resolutions have fields to point at parent sections and neighboring nodes. They are used to access relevant parts of our custom DOM more quickly.

8.2.2 Lists of numbers as paths

There are several ways to pinpoint a node in the DOM. CSS-selectors, XPath and DOM API-call chains among them. The first two methods are easily readable and writable for humans, a feature we are not interested in. Our tool is only intended to output information our client library can read. We will instead use the third option: DOM API-call chains. However cumbersome and counter-intuitive a method like this may seem in other scenarios, it is in fact the optimal tool for our purposes: We are never interested in retrieving more than a single DOM node; knowing where a section or a variable begins is our only goal for paths.

All children of a node are ordered and can be addressed by numbers. This allows us to drill down through the DOM to a specific node by iterating through a list of numbers, descending one node generation with each iteration.

Children and offsets

Paths for our mustache tags can be divided into two types which we will call children and offsets.

Offsets are mustache tags whose location is affected by the string length of a previous variable value or by the amount of iterations of a previous section. To determine their location we will have to know the value of these previous tags first. The tags may of course also only be offsets, therefore this chain continues until we meet a parent section or the beginning of the template.

Figure 8.3: Offsets in templates

```
1 <p>
2   Hello {{nickname}},<br/>
3   you have {{messagecount}} new messages:
4 </p>
5 <ul>
6   {{#messages}}
7   <li>{{subject}} from {{nickname}}</li>
8   {{/messages}}
9   <li>That's all {{realname}}</li>
10 </ul>
```

Children are tags with locations in the template that are not affected by the value of a previous tag. When parsing a rendered template in the client library, we will want to parse all children first and continue with offsets that depend on those children.

Figure 3.1 from chapter 3 illustrates the difference very well, once we add a last list element as shown in figure 8.3. The `{{realname}}` variable can only be retrieved once we know how many times the messages section has iterated (in this case we could also look for the last `` element, this is however not an approach that is easily generalized). The path for `{{realname}}` would consist of two numbers:

- The node index of the `` element measured from the closing tag of the `{{#messages}}` section. Also counting the text node between those tags, we arrive at `1`
- The node index of the `{{realname}}` variable. Although this variable will be merged with the previous text, our tool still counts it as a separate node. We will adjust for this way of counting child nodes in the client library. Here we also arrive at index `1`

The only detail missing from our new path is the reference to the node we are offsetting from. By assigning a number to each mustache tag in the template we can refer to it by that number and prepend it to our path.

Child paths are generated in much the same way. We proceed in the exact same way as with the `{{realname}}` variable, if we were to generate a path for the `{{subject}}` variable in template 8.3. The only difference lies in the classification of the path as a child instead of an offset. The advantage we gain by this classification is our ability to distinguish whether a variable is located inside or outside the section.

8.3 Variable boundaries

Variables embedded in text nodes will be merged with the neighboring text nodes once a template is rendered. To extract the original text, the client library will have to know

the exact length of the text before and after it. If two variables are located in the same text node, this extraction strategy is no longer possible. Instead we simply remember the text surrounding the variable. Using this prefix text our client library can not only find the beginning of a variable, but also verify the preceding text. The succeeding text will be used as a delimiter, it marks the end of our variable. With this strategy we can parse an arbitrary number of variables in one text node, provided a variable does not contain the text of the succeeding text².

8.3.1 Other boundaries

The previous and next nodes of a variable may also be HTML or mustache tags. In the case of an HTML tag, we will simply relay the name of the tag to the client library. If the variable is the first or last node, the client library will receive a special “null node” as the previous or next node respectively. We will tackle the case of neighboring nodes being mustache tags in chapter 10.

8.4 Recognizing iterations

To detect whether a section is skipped because its value is an empty list or false, we let the client library know what the first child of the section is. This way we can detect if a section in a rendered template begins with the first child node and has content or begins with its neighboring next node and is empty.

8.4.1 Content list

We accompany each section with a list of child node types. The client library shall consult this list to determine the length of each iteration. The length of an iteration is constant if a section only contains normal HTML tags as its children. Once we introduce mustache tags as children³ of the section, this changes. The length of a subsection will increase the length of a parent section, while unescaped variables may do the same. In order for us to still reliably determine said length, we also include mustache tags in this content list. The client library may then access information about these tags to assess the impact they have had on the length of an iteration. **►Eval: this can be done a lot easier◄**

8.4.2 Lambda sections

Lambda sections and normal mustache sections can not be distinguished. We also have no way of determining the input to a function by looking at its output. For that reason we will simply treat them as sections when parsing templates and assume the user is aware when functions are bound to a dataset instead of lists.

²Why we cannot parse in a different way is detailed in chapter 10

³ Note the important distinction of “children” and “descendants”. A section may contain an XML node with mustache tags inside it. Those do however not affect the `childNodes` list of the parent tag.

8.4.3 If-else constructs

Sections which are intended as if-else constructs are similarly impossible to distinguish from normal sections, which iterate over a list. We tackle this issue by regarding all sections as iterative sections. We return a list of entries, regardless of the original dataset structure that was fed in to the template rendering engine.

This behavior is also in accordance with the mustache spec:

if the data is truthy (e.g. `!!data == true`), use a single-element list containing the data, otherwise use an empty list. [1, sections.yml]

8.5 Outputting information

We transmit the information our tool retrieves from templates to the client library using the JSON data format. Our tool generates the output by using the JSON library available in the “Hackage” Haskell library database (Text.JSON).

JSON files may contain an array or an object as the top-level structure. We choose to use an array, in which each entry describes a mustache tag. Referencing between those tags functions by way of the index⁴ in said array. We also prepend a “root” section to the array. It is referred to by mustache tags that are not located inside a section and are not offset in their location by other mustache tags.

►Difference between parser combinators and parser generators◄ ►What about cabal, cmdargs?◄

8.6 Client library

8.6.1 Technology choices

CoffeeScript

We use CoffeeScript to program our client library. The language allows us to write expressions very tersely, where plain JavaScript would have required verbose instructions. This helps us to overview more code at once. CoffeeScript constructs like `unless exp`, which translates to `if(!exp)`, also help highlight the control flow in a semantically better way.

8.6.2 Input

The client library expects the user to hand it both the rendered template and the template information our pre-parser tool outputs. How this data is retrieved is not the concern of the library. The rendered template is expected to already be in the DOM format. Additionally it needs to be wrapped in a container node, which is the actual node that is to be handed to the library. The client library also expects the JSON data to already have been interpreted and converted to JavaScript objects.

⁴ This index is the same number we prepend to the paths of tags in section 8.2.2

8.6.3 Representing mustache tags

Our architecture for representing mustache tags is a one-to-one representation of the possibilities in mustache. Each section object holds all of its iterations. A section, variable or partial in a section is instantiated as many times as the section iterates.

Offsets

Through the whole process, we will maintain two pointers that identify our progress in the rendered template:

- `nodeOffset`: Given a parent, this variable indicates the index in the `childNodes` list we are currently pointing at (node offset).
- `strOffset`: Assuming the current node is a text node, this variable points at the current string position (string offset).

For all mustache tags, we will always keep a reference to the `parent` XML node it is located in. This allows us to increase and decrease the `nodeOffset` to access neighboring nodes. This would also be possible by using the `previousSibling` and `nextSibling` properties, mathematical operations like “the 5th neighbor of the current node” will however become quite intricate.

The node offset and string offset is maintained separately for each mustache tag object. An object is instantiated with those offsets, giving it a position to follow its path from and find its node.

Following DOM paths

Given the position (`nodeOffset`, `strOffset` and `parent`) of a mustache tag, we locate other mustache tags that have their location specified relative to it by executing the following steps for each entry in their respective path array excluding the first entry, which is a mustache tag reference (see section 8.2.2):

- Add the current entry to the `nodeOffset`
- Break, if this is the last entry
- Set the parent node to point at the `childNodes` index `nodeOffset` of the current parent node

We break in step 2 to let the parent point at the parent of the mustache tag instead of the tag it has been replaced with.

If an entry in the iteration is a string instead of an integer, this indicates an attribute name. In that case we set the parent node to be the attribute node the string identifies. Attribute nodes support the `childNodes` property in the same way an element node does.

The string offset is reset to 0 once we leave the string context of the current text node. This happens when there is an XML tag between the mustache tag the path is based on and the node the path points at.

8.6.4 Parsing sections

The parsing of sections constitutes the heart of operations in our library. We bootstrap our parser by initializing the fake root section (see 8.5) with the template container node as its parent. The root section will of course only have one iteration. This method allows us to create a recursive parsing process which we will initiate once per iteration for each section we encounter inside a section.

Inside an iteration we will also instantiate variables and partials that are children of the current section. Any mustache tag located inside a subsection will be handled by that section.

We begin by instantiating all mustache tags that are not offsets. These tags have their location described relative to our section, of which we know the location. This means that we can determine their location without any other dependencies. Note that we in each section iteration adjust the node offset to point at the node before the first node in the current iteration. This ensures that the path following process works correctly.

Next we instantiate all tags that are offset by a preceding mustache tag. Since our tool outputs information about tags in the order they appear in, in the template, we simply iterate through the mustache tags, knowing we will not encounter a tag with an offset pointing at a yet uninstantiated node.

►We actually don't need to instantiate all children first, because of tpl ordering◄

Once all tags in a section are instantiated and saved in an object mapped by their identifier, we determine the length of our section with the content list as described in section 8.4.1. This length is used to adjust the section node offset. Using it, mustache tags basing their path on this section can locate their node.

Depending on whether the node following the current node can be identified as a first child of the section or as a neighboring node, we either begin another iteration or terminate the parsing process for this section.

Parsing partials

Partials can be handled by simulating the beginning of an entirely new template. Provided the partial is enclosed in an XML tag, we can restart the parsing process using the matching template information for that partial.

8.6.5 Joined text nodes

Our pre-parser tool recognizes a section with preceding text and text inside it as two nodes with the section containing a third node. In a rendered template these three nodes will merge into one single text node. Variables will also merge with any preceding text node. Succeeding text has a similar effect. This effect can wreak havoc on our node offsets if we do not adjust for it. Sections will therefore detect whether their previous node is joined with their first child and reduce the node offset by one. This is also done for:

- Previous nodes and next nodes
- Last child nodes and next nodes
- Last child nodes and first child nodes

In each of these cases we adjust the node offset accordingly. We adjust for previous and next nodes of variables in the same way.

8.6.6 Verifying nodes

Previous nodes, next nodes, first children and last children are matched with a single data structure. They are specified with their type and in some cases, depending on the type, a second parameter:

- *emptynode*: A self-closing XML tag. The second parameter specifies the tag name
- *node*: An XML tag. The second parameter specifies the tag name
- *comment*: An XML comment. It has no second parameter.
- *text*: A text node. The second parameter is the text itself.
- *null*: No node (e.g. the variable is the last child of an XML tag). The type has no second parameter.

Conflicts may arise if e.g. the next node of a section and its first child are indistinguishable. These cases can be detected by our pre-parsing tool with a filtering mechanism.

Filter

Before our pre-parsing tool outputs the template information, it runs the resolutions generated by the Resolver⁵ through a set of filters.

- *unescape_offset* Check if any mustache tags use an unescaped variable as their path base (see chapter 10).
- *empty_section* Check for empty sections. They should be removed.
- *unescape_pos* An unescaped variable may only be the last child of an XML tag. It may not be a child of a section (see chapter 10).
- *partial_only_child* A partial must be the only child of an XML tag (see chapter 10).
- *no_lookahead* Neighboring nodes and first and last children may not be mustache tags (see chapter 10).

⁵See 8.2.1

- *ambiguous_boundaries* The first child of a section must be distinguishable from the next node of a section.
- *path_with_errors* Paths may not be based on tags that have produced any errors.

We output a message if any of the filters fail and exclude that tag from the output.

8.6.7 Returning values

We save all instantiated sections, variables and partials for every section iteration. Once we have parsed a rendered template, we retrieve our root section and generate a JavaScript array containing one anonymous object per iteration. The object maps variable names to their parsed values and section names to their iterations. As noted in section 8.6.4, partials are considered sections. Since partials in mustache behave as if the template was inlined, we merge the contents of the partial section with the contents of the parent section⁶ Unescaped variables will return a list of their nodes as their value. Previous and next nodes will be included if they are text nodes.

Parent nodes

We add a second useful value to the return value of variables. Often times a user may want to listen to changes concerning the nodes in which mustache tags are placed. Variables may be values in attributes on key XML tags (e.g. value attributes in form input fields) in which case event listeners can be added to those attributes and useful actions performed when they are triggered.

Update()

Variables also return an `update(text)` function. Passing it a string will update their substring in the text node and preserve the surrounding text.

8.7 “Comb”

The tool we have created will be named “Comb”. It refers to the combing of a mustache, much like we comb through templates and rendered templates to retrieve mustache tags and variable values.

The template information generated by our pre-parser tool is called a “comb file”, its file extension is “.mustache-comb”.

⁶ As detailed in 8.6.4 we handle partials like new templates, this means they only have one iteration like the root section.

Chapter 9

Demo application

This chapter describes the workings of an application utilizing Comb. It demonstrates the results of parsing a rendered template.

9.1 Goal

The goal of our application is to showcase the return values of a rendered template that has been parsed. Additionally we also display how Comb can be utilized to implement an application.

9.2 The application

To load a template we click the “Template” drop-down in the top menu. Here we can select any of the templates we utilized in the first iteration. We have converted them into mustache templates.

9.2.1 Rendering a template

Once a template is selected, we load it and the corresponding comb file via *require-js* (►**explain**◄). The template is then rendered with *mustache-js* (►**explain**◄) and an empty dataset in the right part of the view port.

9.2.2 Displaying template information

The rendered template is parsed with Comb and its values are retrieved. At this point we hold a JavaScript object with many keys but few values, because the rendered template contained no section iterations or variable values Comb could retrieve. We transform this object into a data structure which is compatible with a template intended for viewing these values (figure 9.1). The `{{>mustache}}` tag is invoked for every section iterations we find in the dataset of the loaded template, it points back at our form template, allowing nested sections to be rendered.

Figure 9.1: The file `mustache.mustache`. A mustache template intended for viewing values retrieved by `comb`.

```
1 {{#section}}<fieldset class="section">
2   <h5>{{name}}</h5>
3   <div class="buttons">
4     <button class="btn btn-mini btn-primary" data-target="{{name}}"><
5       icon class="icon-plus"/> Push iteration</button>
6     <button class="btn btn-mini btn-danger" data-target="{{name}}"><
7       icon class="icon-minus"/> Pop iteration</button>
8   </div>
9   {{#iterations}}<div class="iterations">{{>mustache}}</div>{{/
10     iterations}}
11   <hr/>
12 </fieldset>{{/section}}
13 {{#escaped}}<div class="control-group">
14   <label class="control-label">{{name}}</label>
15   <div class="controls">
16     <input type="text" placeholder="{{name}}" value="{{value}}" />
17   </div>
18 </div>{{/escaped}}
19 {{#unescaped}}<div class="control-group">
20   <label class="control-label">{{name}}</label>
21   <div class="controls">
22     <textarea rows="3">{{nodes}}</textarea>
23   </div>
24 </div>{{/unescaped}}
```

The rendered template in the left view port presents us with a form containing buttons and input fields fitting the nature of the mustache tag. An escaped variable is a simple input field, while an unescaped variable is a text area. Sections have buttons that allow you to push and pop an iteration (the applications does not support partials).

9.2.3 Parsing the form

So far we only have a form which displays the dataset values of a template that was rendered with an empty dataset. Since this form was rendered with mustache, we can now make it interactive by parsing it with Comb. The buttons in the “section” section have `data-target` attributes so that we may bind click event listeners to them, by using the parent nodes of the “name” variables. We can listen for changes on the input fields for the escaped variables and text areas for the unescaped variables in much the same way.

9.2.4 Loopbacking Comb

The fields in our form have corresponding entries in the dataset passed to the template we loaded in the beginning. Note that although we passed an empty dataset to mustache, Comb will return section values as empty lists (see section 8.4.3) and variable values as empty strings (see section 3.2.1). By binding event listeners to our fields we can update the original dataset correspondingly. Changes in input fields and text areas trigger a call to the `update(text)` function on the original dataset and update the text in the right view port.

Pushing the “push” button on a section appends a new entry to the array of the original dataset, while the “pop” button removes the last entry from said array. When we modify the amount of iterations in an array, we re-render the loaded template¹.

¹ Chapter 10 explains this necessity in more detail.

Chapter 10

Evaluation

10.1 Limitations

- comment nodes useful, should not have been discarded
- We don't get the entire section structure of a template unless all sections have at least one iteration
- Cannot push or pop iterations without rerendering
- Lambdas
- Choice of parsing method
- Client
- Parsing problems on the client
- Filter on compiler
- Bug in mustache.js (not a bug, it's the spec) <https://github.com/mustache/spec/blob/master/specs/sc>
<https://github.com/janl/mustache.js/blob/master/mustache.js#L512>
- Removal of whitespace lines
- Full XML EBNF a bit out of scope: <http://www.jelks.nu/XML/xmlebnf.html>
- & Does not throw proper error when not closed
- Lexeme token parser versus normal token parser
- Comment not working
- variable text boundary recognition not perfect
- inverted sections don't iterate, no need to filter

- and no need to return array
- If-else section return lists with single entry
- Mustache identifiers can access more than the keys in the current context, they may also drill down and access properties of objects in the current context.

10.1.1 Removal of standalone lines

As noted in section 3.2.1, mustache removes lines containing only a section beginning/end and whitespaces. Our pre-parser tool does not take this into account. A modification of *mustache-js* was necessary to...

10.2 Challenges

- Unescaped variables can wreak havoc
- Ambiguities can occur between two boundaries
- TextNodes that are separate in the parser are merged on the client

10.3 Advantages

10.3.1 Comparison

Chapter 11

Future Work

Chapter 12

Related Work

Chapter 13

Conclusion



Appendix A

Technical Appendix

A.1 Choosing the tools

Existing frameworks supply much of the functionality we require.

- requirejs to get the information about placeholder locations to the client.
- as in the prototype: backbone to represent client models, collections and views
- chaplin to supply us with a framework. Chaplin is very lightweight and utilizes backbone. It comes with controllers and extended views and adds collectionviews
- handlebars is a port of mustache templates to the client side
- coffeescript to avoid all the javascript nonsense
- HaXML to parse html templates
- Parsec to combine HaXML and a custom parser for mustache templates

Bibliography

- [1] Pieter van de Bruggen. The mustache spec. 2010. 14, 19, 44