Extracting Data from Rendered HTML Templates

Anders Ingemann, 20052979
anders@ingemann.de

PC edition

Master's Thesis, Computer Science
January 2013
Advisor: Jan Midtgaard

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Extracting information from HTML and binding event handlers to the Document Object Model (DOM) in web applications using CSS selectors is a tedious exercise. There have been many attempts to both rectify the root cause and alleviate the symptoms with jQuery being the most prominent among them. We present a tool which solves precisely this problem and leverages the fact that web applications use HTML templates, which can be analyzed to automate the otherwise manual navigation through the DOM. We also show how a clear-cut, minimal scope can increase the applicability of a tool and allow it to thrive in an ecosystem of other software.

Our tool is based on the mustache template engine which is implemented in numerous programming languages. On the client it recreates the original dataset that was used to render a template. This reconstructed dataset allows the developer to read values from — and interact with — the DOM using the already familiar identifiers that were used as placeholders in the templates. Because these identifiers do not change when templates are restructured, this method is also more robust than using CSS selectors, which we demonstrate with two web applications.

Resumé

Det er et langsommeligt arbejde at indhente informationer fra HTML og knytte *event handlers* til *Document Object Model* (DOM) i web applikationer via *CSS selectors*. Der har været mange forsøg på at rette op på den grundlæggende årsag til dette, men også nogle forsøg der afhjælper symptomerne i stedet. jQuery er det mest velkendte blandt disse forsøg. Vi præsenterer et værktøj der løser netop dette problem. Det udnytter web applikationers brug af HTML skabeloner (*templates*), som kan analyseres for at automatisere navigationen gennem DOM'en — en førhen manuel opgave. Desuden viser vi hvordan en klar afgrænsning og minimal omfang kan øge værktøjers anvendelighed og hjælpe dem med at trives i et økosystem af anden software.

Vores værktøj er baseret på *mustache template engine*; et template sprog, der er implementeret i mange forskellige programmeringssprog. Værktøjet genskaber på klienten det oprindelige datasæt der blev brugt til at rendere et template. Dette rekonstruerede datasæt gør det muligt for en udvikler at læse værdier fra — og interagere med — DOM'en ved at referere til de i forvejen kendte nøgler der blev brugt som variable i skabelonerne. Denne metode er også mere robust end at bruge CSS selectors, fordi navnene på nøglerne ikke ændrer sig når templates omstruktureres. Vi demonstrerer dette med to web applikationer.

Acknowledgements

I want to thank my family, ex-girlfriend and friends for the understanding and support they offered me throughout the writing of this thesis.

The original idea for this concept originated in some intriguing conversations with Casper Bach Poulsen, without whom the proverbial light bulb would not have lit up above my head; Thank you.

A lot of the novel practices used in this thesis can be attributed to the thriving work environment at my employer, Secoya. Thank you to all my colleagues at work who challenge the status quo and always push to try out new and shiny things.

I thank my friends from university Martin Christensen, Steffen Lund Andersen, Ubbe Welling, Jens Christian Junge, Peter Hvidgaard and Casper Bach Poulsen for the great times we had during our studies and the even greater times between them.

Finally I extend my sincere gratitude to Jan Midtgaard, my advisor, who provided stellar feedback and went far beyond the call of duty to help me see this through to the end. Without him, this thesis may never have seen the light of day.

Anders Ingemann,
Aarhus, Tuesday 15th January, 2013.

Contents

Chapter 1

Introduction

The field of web application development is young; the first prominent web application (Viaweb) was conceived as late as in 1995. Over the years the field has received a potent influx of novel ideas and solutions to deal with the challenges of client-server architecture. Grounded in the dynamic nature of web applications, developers use templates to abstract data from the presentation layer. Most template engines work by replacing placeholders in templates with corresponding values from a dataset. Once a template has been rendered the information about the location of these placeholders is discarded and the HTML is sent to the client. On the client, the information required to bootstrap an application structure with data may be embedded in the HTML. Currently the developer accesses such information by leveraging the structure of the template to create selectors pinpointing its location. Actionable parts of the HTML are designated in the same way.

We propose a tool that utilizes this otherwise discarded information to parse the HTML on the client-side and extract the original dataset, while preserving the locations whence any value was extracted. The dataset allows the developer to interact more naturally with the HTML and makes the client-side application less fragile in regard to template changes.

By basing our tool on an existing template language, externalizing its analysis and not introducing extraneous requirements we distinguish it from existing solutions, that require the use of their own template language or only solve the problem for one specific web server programming language.

This thesis is divided into three parts. First we delve into the structure and development of web applications (chapter 2). Based on what we have uncovered we then go on to specify requirements (chapter 3) for our exploratory prototype (chapter 4). Once we have analyzed the results and revised our requirements (chapter 5) we proceed to the second part of the thesis in which we devise an architecture (chapter 6) for the subsequent implementation (chapter 7) of our tool. We also build two demonstration applications (chapter 8) that each employ our tool in different ways. In the third part we evaluate our tool (chapter 9) and compare it to related work (chapter 10), on these notes we then conclude the thesis (chapter 11).

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 The structure of 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

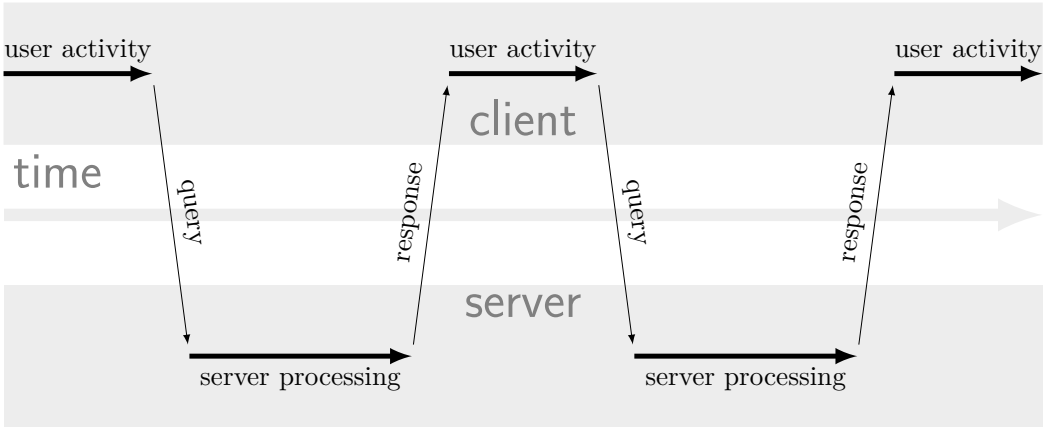


Figure 2.1: Diagram of synchronous communication between client and server

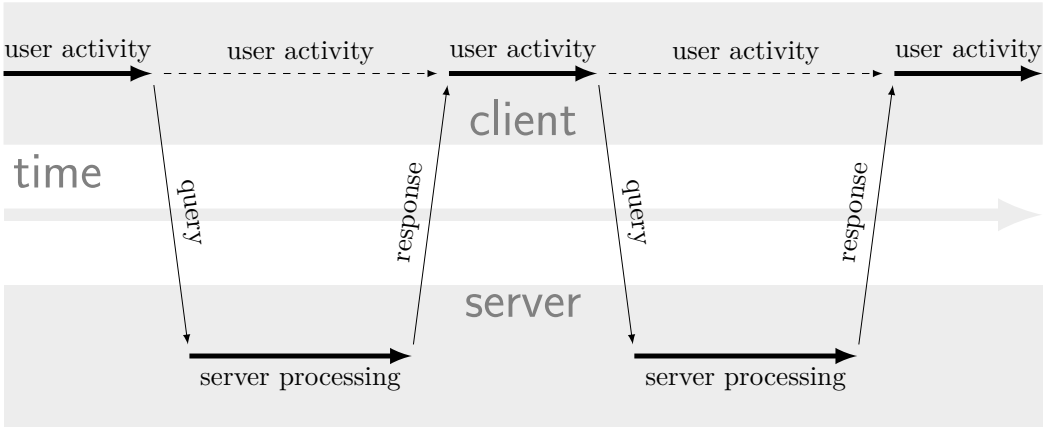


Figure 2.2: Diagram of asynchronous communication between client and server

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

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 [1].

2.1.3 Server-side web applications

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

- *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" 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[2]). 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" are fitting examples of such an approach. These web applications rely on external JavaScript APIs (Google Maps, Twitter, Weather services) to combine readily available data 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 directly to neither the user nor the developer.

[1] which it almost always is

[2] `http://www.positioniseverything.net/explorer/guillotine.html`

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.

- *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[3].

- *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[4].

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.

A simple example is the deletion of an item from a list: Once a user has given the command to delete an item, the client sends a request to the server to perform the deletion. This can be a file, folder or an address from an address book. The client does in such a case not need to wait for the server to respond to remove this item from the

[3] An example of such an application is *Strongcoin*, an online wallet for BitCoins. The code is not compressed or obfuscated in any way, allowing the user to audit the code.

[4] Google's GMail email service for example can be accessed in Google Chrome without an Internet connection in a limited fashion.

view, it can do so immediately. This increases the felt performance of an application for the user. If the deletion did not succeed (e.g. the user has insufficient rights, the address does not exist any longer, the folder is not empty), the client can upon receival of the error reintroduce the item into the list and notify the user of the error. A drawback of such a strategy is an interruption of the work flow, where the user may already be working on another task. It must therefore be applied only when the interrupted work flow is a price worth paying (e.g. interface responsiveness is paramount, errors rarely happen).

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 be completed. 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 are many others, which are relevant in web application development. Nonetheless, the most prevalent design pattern is the Model-View-Controller pattern, which we will examine in the following section.

Model-View-Controller

The Model-View-Controller design pattern has proven itself to be a sound 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 any 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[5] and Silverlight but also HTML5[6].

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 from the View 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 code refactoring harder.

The advantage of this version 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.

[5] `http://msdn.microsoft.com/en-us/magazine/dd419663.aspx`
[6] `http://knockoutjs.com/documentation/observables.html`

Chapter 3

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. Nevertheless, this does not mean that the information is useless.

Consider a minimal template used for editing profile information as in figure 3.1 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 users 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 across the application, 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

```
<form action="/save_user">
  <label>
    Nickname: <input type="text" value="{{nickname}}" name="nickname" />
  </label>
  <label>
    Real name: <input type="text" value="{{realname}}" name="realname" />
  </label>
  <label>
    Birthday: <input type="text" value="{{birthday}}" name="birthday" />
  </label>
  <input type="submit" />
</form>
```

Figure 3.1: A template for displaying user information

of an HTML template are filled these positions are known, but as soon as the result is reduced to simple a HTML 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 however is 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. Based on findings we can formulate a goal for our project:

> Our goal is to retrieve information about the binding of model fields to placeholders in templates and use this information to bind client-side models to the DOM.

Chapter 4

Exploratory 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 final implementation. The 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.

4.1 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.

4.1.1 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.

- *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 template engine which will allow us to generate DOM elements and insert them into the page.

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

- *XPath*
  XPath is a language that allows us to define a path through the DOM from a on 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 not implemented in any browsers yet and likely will not ever be (the first candidate recommendation was released in November 2005[1]).

4.1.2 Templates

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 decrease the probability of encountering compatibility problems with any template language we wish to employ later on. 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 4.1 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.

[1] `http://www.w3.org/TR/2005/CR-xpath20-20051103/`

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

Figure 4.1: The file `movie.view.tpl`. A template in the initial prototype.

- *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 associative arrays so their fields can be initialized as variables with the `extract` method (the array keys become variable names).

- *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 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 (4).

4.1.3 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[2] that can be extended.

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.

4.1.4 Collections

In addition to 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 any given Collection the models all have the same type.

[2] By "class" we of course mean a JavaScript prototype.

4.2 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 the 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 useful when binding event listeners to the DOM or manipulating DOM nodes in other ways.

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

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

4.2.1 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, given that 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 [3]. 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[4].

- *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 in one string. This is undesirable, since we currently only place a server-side field value into the "value" part of a DOM node attribute.

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

4.3 Results

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.

[3] this is hard-coded for the time being

[4] The coupling of ViewModels and XPaths via the `view()` function implies that a way to map templates to ViewModels will be required

- *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 other models, 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 final implementation. 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. We will require a process, which automates the coupling of the ViewModels with server-side templates.

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 5

Revised Requirements

In this chapter we will analyze which parts of our exploratory prototype we can utilize in our implementation. 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 architecture of our tool 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 simple repeatable patterns in PHP. The recursive tree parser that builds 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 similar but seasoned tools do not have. We decided to develop these tools for our prototype regardless, because evaluating alternatives that would fit the purpose precisely would have taken up more time. This is not a sound strategy going forward. To implement our goal and later on maintain it, we require a nucleus of code, which only incorporates the parts that are necessary and unique to our solution. 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.

5.1 Revised Goal

In chapter 3 we concluded that our goal is to retrieve information about the binding of model fields to placeholders in templates and use this information to bind client-side models to the DOM.

In reality this goal consists of two parts: First we need to extract the dataset that was passed together with the template into the template engine from the rendered tem-

plate[1]. In the prototype we did this by utilizing hard-coded XPaths. The motivation behind hard-coding this part was the assumption that we would generate the those paths dynamically in our real implementation. Once the dataset is rebuilt the second part consists of mapping its values to models on the client-side. In the prototype we realized the mapping process by hard-coding the types of backbone fields into the ModelView. However, our final implementation will only feature the first of these processes. To state it clearly:

> Our goal is to extract the dataset passed to the mustache template engine from the rendered template.

This drastic change in course is grounded in the desire to create a tool that is applicable in as many types of web applications as possible. The mustache template engine is the only precondition for using the first part of our mapping process. The second part however can have preconditions other than backbone that were not explored in our prototype: To generate a mapping for client-side models, our tool will require to know the origin of the values in the retrieved dataset[2]. These values usually originate in models on the server-side[3], where the developer may have chosen from any number of server-side languages and model frameworks. Unless we require the developer to specify the types and relations between models in a format our tool can understand, we will have to choose a language and model framework to automate this process. This choice reduces the applicability of our tool greatly. Leaving only the option of requiring the developer to specify the relations. For big web applications this requirement may slow the development process significantly. This is why we choose to pursue only the first part of the mapping process. With that choice the implementation will no longer concentrate on mapping values sent by the server via rendered templates to client-side models, but on extracting data from rendered templates.

By concentrating on one part of the process we are also able to create a tool in the proper sense. As a guide for the properties of such a tool we can apply some of the rules set by Eric Steven Raymond in his book "The Art of Unix Programming". The success of Unix can partially be attributed to its many small tools that work in concert to create more complex systems. We believe this design philosophy can be applied to web development tools as well.

- *Rule of Modularity: Write simple parts connected by clean interfaces.*
- *Rule of Clarity: Clarity is better than cleverness.*
- *Rule of Composition: Design programs to be connected to other*

[1] Read: From the HTML in the browser.

[2] The dataset itself will contain only string values retrieved from the rendered template. Its structure can similarly only resemble the structure of the template.

[3] They may also come from client-side models, where the mapping would make less sense, because the retrieved values originates from these models. Section 9.5.2 details why such a mapping would be advantageous regardless of that fact.

programs.

- *Rule of Separation: Separate policy from mechanism; separate interfaces from engines.*
- *Rule of Simplicity: Design for simplicity; add complexity only where you must.*
- *Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.*
- *Rule of Transparency: Design for visibility to make inspection and debugging easier.*
- *Rule of Robustness: Robustness is the child of transparency and simplicity.*
- *Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.*
- *Rule of Least Surprise: In interface design, always do the least surprising thing.*
- *Rule of Silence: When a program has nothing surprising to say, it should say nothing.*
- *Rule of Repair: When you must fail, fail noisily and as soon as possible.*
- *Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.*
- *Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.*
- *Rule of Optimization: Prototype before polishing. Get it working before you optimize it.*
- *Rule of Diversity: Distrust all claims for "one true way".*
- *Rule of Extensibility: Design for the future, because it will be here sooner than you think.*

[8, Chapter 1]

By focusing on data extraction we can follow the Rule of Composition more easily: A layer to map the values we extract to client-side models can still be implemented on top of it, thereby enabling developers to integrate our tool into other client libraries than backbone.

The binding of models is a feature which would add to the complexity of our tool. The tool still has a purpose and relevance without this feature[4]. We simplify our approach and thereby follow the Rule of Simplicity.

[4]This also means that we will no longer focus on the task from section 4.3 to couple server-side templates to ViewModels on the client-side.

By using these rules for guidance we hope to be able to develop a tool that will be able to fit into an ecosystem of existing software much the same way Unix tools exist in a similar ecosystem of software. To do this we focus on our previously stated goal and remove all superfluous features that do not contribute to that goal. To see a list of which rules we follow throughout the rest of this thesis refer to appendix C.

5.2 Simplifying the project

Bear in mind that despite the following simplifications we may still use some of the libraries. As stated in the previous section, we intend our tool to perform in an ecosystem of other software, minimizing dependencies can help developers integrate it into their existing software without conflicts. As an example of such an ecosystem consider Bower[5] by Twitter. Bower is a package manager for client-side applications. It contains over 700 components ranging from asynchronous module loaders[6] to graphing libraries such as "d3".

5.2.1 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 necessary at all 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 even with 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, will of course need to be able to interface with that language. For the prototype we omitted such an engine and wrote the templates directly in PHP instead.

Our plan is to write the server-side templates in mustache (B). Mustache is a template language that is very predictable in its output, given that it cannot perform any computations on the dataset or modify its input. This will be an advantage when we want to extract values from rendered templates. Mustache also follows some of the rules set by Terence Parr in his paper titled "Enforcing Strict Model-View Separation in Template Engines" [7], which details how separating the business-logic of a web application from the presentation layer is both desirable and possible. One such rule states:

> *2.* ***the view cannot perform computations upon dependent data values*** *because the computations may change in the future and they should be neatly encapsulated in the model in any case. [...] the view cannot make assumptions about the meaning of data.* [7, Chapter 7]

[5] `http://twitter.github.com/bower/`
[6] See section 7.6.1

As noted above mustache does not support computations on data, which makes it compatible with this rule[7].

Another rule concerns the comparison of data:

> *3.* ***the view cannot compare dependent data values****, but can test the properties of data such as presence/absence or length of a multi-valued data value.* [7, Chapter 7]

Sections in mustache[8] support this exact behavior, save the length comparison.

Locating placeholders in 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 are essential.

5.2.2 Client-side

The client-side tools we have used in our prototype interact with the data we extracted[9] from the rendered templates. 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 suggests 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 *less.js* — 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 in section 2.1.2, depending on the challenges we face in the implementation, we may choose a language which compiles to JavaScript instead.
- 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[10]. This makes underscore.js a non-crucial part of our tool.

[7] Although lambda sections (B.5) allow computations on data, they cannot be considered *embedded* computations, since their functions are part of the dataset. The authors sentiment ("they should be neatly encapsulated in the model") is therefore honored.

[8] See appendix B.3

[9] Remember: In the prototype we did not actually retrieve any information from the templates automatically

[10] although it may require more effort.

- 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.

In essence we will not retain any client-side libraries.

Chapter 6

Architecture

Our goal is to create a tool that allows the developer 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 retrieve the dataset from 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 run once for every template before a web application is deployed, we are not constrained by the environment the actual web server runs in.

6.1 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. Analyzing the capabilities of the mustache template language we arrive at the following pieces of information, that can be deduced before the template is rendered.

- The location of variables[1] in the document
- Whether a variable is escaped or unescaped[2] (`{{identifier}}` vs. `{{{identifier}}}`)
- The location of sections[3] in the document
- Whether a section is inverted[4] (`{{#identifier}}` vs. `{{^identifier}}`)
- The contents of a section

[1] See appendix B.1
[2] See appendix B.2
[3] See appendix B.3
[4] See appendix B.4

- The location of partials[5] in the document
- The location of comments[6] 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[7] (except in the case of an inverted section)
- Whether a section is a lambda section[8] 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 context level[9] or to a key in one of the lower context levels[10]

The rendered template is — once it has been rendered by the client — in the Document Object Model format. 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. Using this property to extract the dataset has a 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 “ <img >”. In Firefox the last line returns “ <img xmlns="http://www.w3.org/1999/xhtml" / >”. This means that the rendered templates cannot 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[11]. We

[5] See appendix B.7
[6] See appendix B.6
[7] See appendix B.4
[8] See appendix B.5
[9] See appendix B.3
[10] Excluding the lowest context level. That is: Identifiers with no parent section
[11] case sensitivity can be avoided with a simple `element.tagName.toLowerCase()` when comparing tag names client-side

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.

6.2 Communicating with the client

The information gathered by the pre-parser is saved in files next to the original templates. The developer 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 developer 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[12] to load templates and their information in parallel while developing and subsequently inline both template and template information into one big JavaScript file when deploying.

Figure 6.1 illustrates our resulting architecture. We use the pre-parser tool to analyze the templates and save the results in new files. The pre-parser can analyze the templates on any given machine before both the templates and the results are deployed to the server. Together they are part of the static content (among CSS, images etc.) of a web application. The data is fed into the mustache template engine on a page request to generate HTML that is served to the client. Once on the client, the HTML is parsed and we can access it via the Document Object Model. The parser in our client library uses the DOM in concert with the template information retrieved from the server to parse the rendered template and to output the data that was originally fed into the template engine.

6.3 Alternatives

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

6.3.1 Integrating with the template engine

In chapter 3 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

[12] See section 7.6.1

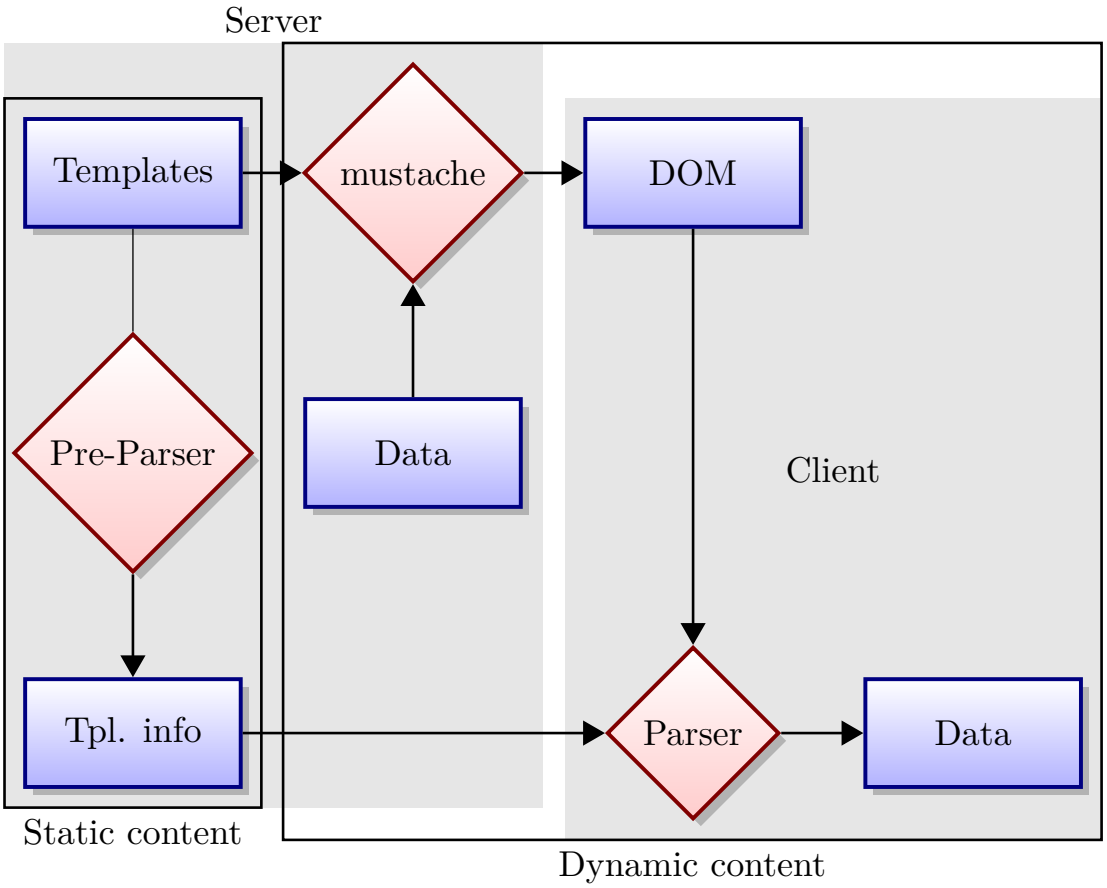


Figure 6.1: Architectural diagram of our tool

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[13]. This fact makes the goal of such an approach very hard to achieve.

6.3.2 Decorating templates

The developer 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 thereby obsoleting the pre-parser.

However, apart from simply shifting the workload of locating template variables from the ViewModel maintainer to the template maintainer[14], the task of retrofitting existing web applications becomes much greater.

6.3.3 Rule of Parsimony

The Rule of Parsimony states

> *Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.* [8, Chapter 1]

By examining the output of the `innerHTML` property in different browsers and hypothesizing[15] how our goal could be solved by decorating templates, we have shown "that nothing else will do". The approach to parse rendered templates after analyzing them may be a more complex operation than other solutions, but it is the only one "that will do" to achieve our goal. We have also shown that integrating directly with the template engine would in fact result in a bigger program. We are therefore following The Rule of Parsimony.

[13] `https://github.com/defunkt/mustache/wiki/Other-Mustache-implementations/9ff07950f0983b58248e6a18d17cce5c47743344`

[14] The scenario in chapter 3 illustrates this point

[15] though not really"demonstrating" in terms of implementation

Chapter 7

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.

7.1 Parsing mustache templates

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 quite effortlessly convert an EBNF grammar into Haskell code by using the combinators and parsers the library supplies us with.

7.1.1 Mustache EBNF

The EBNF for mustache is fairly simple and can be seen in figure 7.1. The behavior of the `set_delimiter` tag is ignored in this EBNF.

7.1.2 Mustache-XML EBNF

We want our parser to not only be able to understand mustache, but also HTML intermingled with it. As such we extend our mustache grammar to incorporate HTML as well.

There are many flavors of HTML we may choose from and allow in our combined mustache-HTML grammar. To simplify our approach we will only allow well structured XML tags, as this should ostensibly cover most of HTML. HTML 5 allows for self-closing tags on void elements (such as “img”) and boolean attributes (such as “checked” on a checkbox), which with our grammar is not something we can support. We will therefore refer to HTML tags in our templates as XML tags[1].

[1]This does however not imply that we intend Comb to be able to perform data extraction on the full spectrum of XML documents.

⟨*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*⟩*

Figure 7.1: Mustache EBNF

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 7.2 represents the structure our parser understands.

The mustache comment tag[2] 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[3]. This was mostly done to keep our first implementation of the tool simple.

Using this grammar we can construct an abstract syntax tree (figure 7.3b) from the mustache template in figure 7.3a.

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 developer 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 tree structure as XML tags[4].

[2]See appendix B.6
[3]See appendix B.8
[4]This means that sections may not interleave, something mustache does not support in any case

⟨*variable*⟩ ::= '{{{' ⟨*ident*⟩ '}}}' | '{{&' ⟨*ident*⟩ '}}' | '{{' ⟨*ident*⟩ '}}'

⟨*partial*⟩ ::= '{{>' ⟨*ident*⟩ '}}'

⟨*content_section*⟩ ::= '{{#' ⟨*ident*⟩ '}}' ⟨*template_content*⟩* '{{/' ⟨*ident*⟩ '}}'
| '{{^' ⟨*ident*⟩ '}}' ⟨*template_content*⟩* '{{/' ⟨*ident*⟩ '}}'

⟨*attribute_section*⟩ ::= '{{#' ⟨*ident*⟩ '}}' ⟨*attribute_content*⟩* '{{/' ⟨*ident*⟩ '}}'
| '{{^' ⟨*ident*⟩ '}}' ⟨*attribute_content*⟩* '{{/' ⟨*ident*⟩ '}}'

⟨*comment_section*⟩ ::= '{{#' ⟨*ident*⟩ '}}' ⟨*comment_content*⟩* '{{/' ⟨*ident*⟩ '}}'
| '{{^' ⟨*ident*⟩ '}}' ⟨*comment_content*⟩* '{{/' ⟨*ident*⟩ '}}'

⟨*content_mustache_tag*⟩ ::= ⟨*content_section*⟩ | ⟨*partial*⟩ | ⟨*comment*⟩ | ⟨*variable*⟩

⟨*attribute_mustache_tag*⟩ ::= ⟨*attribute_section*⟩ | ⟨*partial*⟩ | ⟨*comment*⟩ | ⟨*variable*⟩

⟨*comment_mustache_tag*⟩ ::= ⟨*comment_section*⟩ | ⟨*partial*⟩ | ⟨*comment*⟩ | ⟨*variable*⟩

⟨*attribute*⟩ ::= ' ' ⟨*ident*⟩ '="' ⟨*attribute_content*⟩* '"'

⟨*xml_tag*⟩ ::= '<' ⟨*ident*⟩ ⟨*attribute*⟩* '>' ⟨*template_content*⟩* '</' ⟨*ident*⟩ '>'
| '<' ⟨*ident*⟩ ⟨*attribute*⟩* '/>'
| '<!-' ⟨*comment_content*⟩* '->'

⟨*attribute_content*⟩ ::= ⟨*attribute_mustache_tag*⟩ | ⟨*char_without_doublequote*⟩

⟨*comment_content*⟩ ::= ⟨*comment_mustache_tag*⟩ | ⟨*char*⟩

⟨*template_content*⟩ ::= ⟨*content_mustache_tag*⟩ | ⟨*xml_tag*⟩ | ⟨*char*⟩

⟨*content*⟩ ::= ⟨*template_content*⟩*

Figure 7.2: Mustache-XML EBNF

(a) A mustache template

```
<div>
  Hello {{#user}}<a href="/user/{{id}}">{{name}}</a>{{/user}}
  <img src="about:blank" />
</div>
```

(b) Abstract syntax tree generated by figure 7.3a

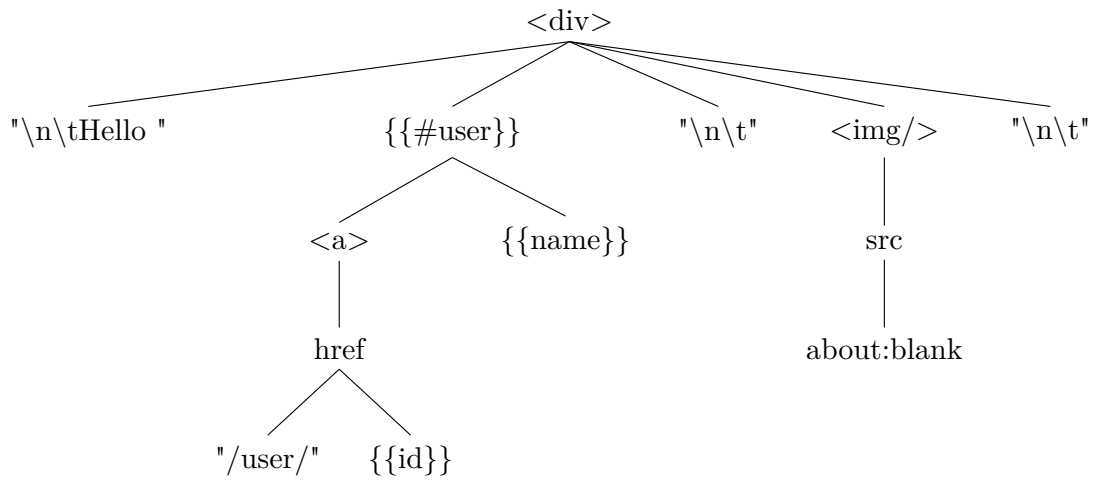


Figure 7.3: Converting a mustache template to an abstract syntax tree

- Variables and sections may not exist in the identifier part of an XML tag or XML attribute.

It is hard to predict how often a developer will encounter these structural restrictions. Regardless, they give our client library very useful guarantees about the rendered templates it parses. They also allow us 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.

Character References

The EBNF omits character references (e.g. ` `, `&aring;`). 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 (A.5), which 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 for example.

Significant white space

In the case of our template parser, the otherwise advantageous properties of lexeme token parsers are not desirable. White space in the beginning of an attribute value or after an XML tag can be very significant. We will need to be sure where a mustache variable begins and ends. If a variable is surrounded only by white space, 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 space recognized in the rendered template belongs to the value of the variable.

7.1.3 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 layer on top of the HXT XML structure.

XML (Text.XML.Light)

XML is an easy-to-use XML library, which exports its data constructors. This allows our functions to pattern match the data records the library has created. 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, variable locations and partial locations. These overlay techniques would quickly outnumber the 250 lines of code our Parsec parser spans now.

7.2 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[5]. 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.

7.2.1 Resolver

Our tool passes the abstract syntax tree generate by the parser into the 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.

7.2.2 Lists of numbers as paths

There are several ways to pinpoint a node in the DOM. CSS-selectors, XPaths 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, 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.

[5]Figure 7.6a provides an example with a diagram of joined nodes.

```
<p>
  Hello {{nickname}},<br/>
  you have {{messagecount}} new messages:
</p>
<ul>
  {{#messages}}
  <li>{{subject}} from {{nickname}}</li>
  {{/messages}}
  <li>That's all {{realname}}</li>
</ul>
```

Figure 7.4: Offsets in templates

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.

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 B.1b from appendix B.3 illustrates this difference, once we add a last list element as shown in figure 7.4. 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 `<li>` element, this is however not an approach that is easily generalized). The path as highlighted in figure 7.5 for `{{realname}}` would consist of two numbers:

- The node index of the `<li>` 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 in the DOM, our tool still counts it as a separate node. We will adjust for this way of counting child nodes in the client library[6]. Here we also arrive at index *1*.

[6]See section 7.6.5

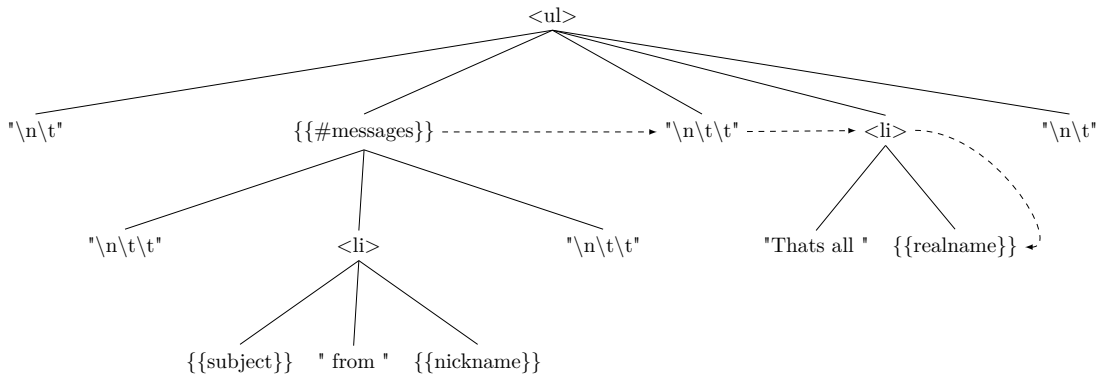


Figure 7.5: The path from `{{#messages}}` to `{{realname}}`

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 generate a path for the `{{subject}}` variable in template 7.4 like we did for the `{{realname}}` variable. 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.

7.3 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 text[7].

7.3.1 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 section 9.3.2.

[7]Why we cannot parse in a different way is detailed in section 9.3.2

7.4 Recognizing iterations

To detect whether a section is skipped because its value is an empty list, 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.

7.4.1 Content list

We accompany each section with a list of child node types. The client library shall consult this list to determine the number of child nodes (size) in each iteration. The size of an iteration is constant if a section only contains normal HTML tags as its children. Once we introduce mustache tags as children[8] of the section, this changes. The size of a subsection will increase the size of a parent section, while unescaped variables[9] may do the same. In order for us to still reliably determine said size, we also include mustache tags in this content list. The client library may then access information about these tags to asses the impact they have had on the size of an iteration.

7.4.2 Lambda sections

Lambda sections[10] and normal mustache sections cannot 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 developer is aware when a function is bound to a dataset instead of any type of value.

7.4.3 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.* [4, sections.yml]

7.5 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).

[8] 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.

[9] See appendix B.2

[10] See appendix B.5

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 index[11] 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.

7.5.1 Rule of Modularity

The Rule of Modularity states:

> *Rule of Modularity: Write simple parts connected by clean interfaces.*
> [8, Chapter 1]

The pre-parser and the client library are two distinct components with clear-cut responsibilities. By using a well documented data format like JSON and letting it be the only binding link between the pre-parser and the client library we are following the Rule of Modularity.

7.6 Client library

7.6.1 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.

require.js

require.js is a module loader for the browser. It allows a developer to specify any dependencies a piece of JavaScript code may have and rely on require.js to load these dependencies before evaluating the code. This dependency management allows developers to concentrate on writing function modules or even "classes" for the client-side of a web application. We will use require.js to split our client library into manageable files representing the logical parts of our library.

7.6.2 Input

The client library expects the developer 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

[11] This index is the same number we prepend to the paths of tags in section 7.2.2.

format. Additionally it needs to be wrapped in a container node, which is the actual node that should be handed to the library. The client library also expects the JSON data to already be interpreted and converted into a JavaScript array.

7.6.3 Representing mustache tags

Mustache tags are represented in our architecture as a one-to-one mapping with classes. Each section instance holds all of its iterations. A section, variable or partial inside a section is instantiated as many times as the section iterates.

Offsets

Through the whole parsing 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`: If 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. Such DOM navigation 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 path array excluding the first entry, which is a mustache tag reference (see section 7.2.2):

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

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

An attribute of a node is represented with a string (the attribute name) in the path array. 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, allowing us to keep all other parts of the path traversal generic. The string offset is reset to 0 once we move the pointer away from current text node. This happens when there is at least one XML tag between the mustache tag the path is based on and the node the path points at.

7.6.4 Parsing sections

The parsing of sections constitutes the heart of operations in our library. We bootstrap our parser by initializing the fake root section[12] with the template container node as its parent. The root section will only have one iteration. This bootstrapping 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 subsection.

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 preceding the first node of 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.

Once all tags in a section are instantiated and saved in an object with their their identifier as they key, we determine the size of our section using the content list as described in section 7.4.1. This size 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 last node of an iteration can be identified as a first node of the next iteration or as a node following the section, 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.

[12] See section 7.5

7.6.5 Joined text nodes

Our pre-parser tool recognizes a section with preceding text and text inside it as two nodes with the section itself being 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 accordingly. Sections 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. Figure 7.6b illustrates these relationships between elements and sections. Note how the DOM of a rendered template in figure 7.6 is different from our abstract syntax tree in figure 7.6b. Sections no longer exist and text nodes that were previously separated have now joined. Variables have similarly joined with adjacent text nodes.

7.6.6 Verifying nodes

Previous nodes, next nodes, first children and last children are matched with a single data structure. The pre-parser specifies their type and in some cases — depending on the type — a second parameter:

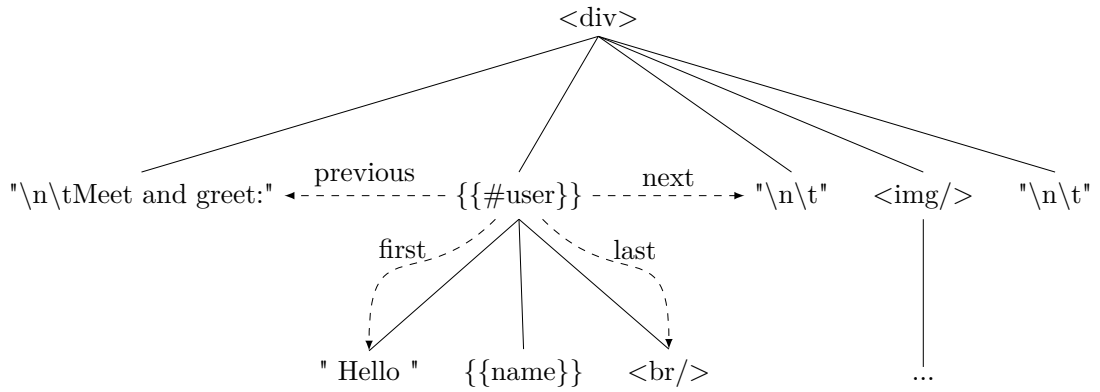
- *emptynode*: A self-closing XML tag. The second parameter specifies the tag name
- *node*: An XML tag. The second parameter specifies the tag name
- *comment*: An XML comment. It has no second parameter.
- *text*: A text node. The second parameter is the text itself.
- *null*: No node (e.g. the variable is the last child of an XML tag, so a next node does not exist). 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.

(a) A mustache template

```
<div>
  Meet and greet:{{#user}} Hello {{name}}<br/>{{/user}}
  <img src="about:blank" />
</div>
```

(b) The relationships of a section in an abstract syntax tree generated by figure 7.6a



(c) Input to the template in figure 7.6a

```
{ user: [ {name: "user1"}, {name: "user2"} ] }
```

(d) Resulting HTML when rendering the dataset in 7.6c using the template from figure 7.6a

```
<div>
  Meet and greet: Hello user1<br/> Hello user2<br/>
  <img src="about:blank" />
</div>
```

(e) DOM tree for 7.6d

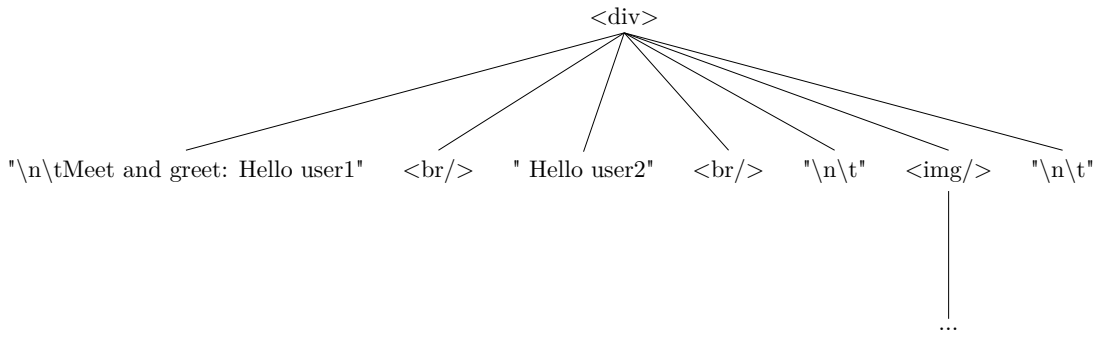


Figure 7.6: Tree transformations when rendering a mustache template

Filter

Before our pre-parsing tool outputs the template information, it runs the resolutions generated by the Resolver[13] through a set of filters in the order they are listed below.

- *unescaped_offset* Check if a any mustache tags use an unescaped variable as their path base[14].
- *empty_section* Check for empty sections. They should be removed.
- *unescaped_pos* An unescaped variable may only be the last child of an XML tag. It may not be the child of a section (see section 9.4.4).
- *partial_only_child* A partial must be the only child of an XML tag (see section 9.4.3). This guarantees the provision in section 7.6.4.
- *no_lookahead* Neighboring nodes and first and last children may not be mustache tags (see section 9.3.2).
- *ambiguous_boundaries* The first child of a section must be distinguishable from the next node of a section. This prevents the conflict discussed in section 7.6.6
- *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.

Having created all the phases necessary for our pre-parser tool to output the right information, we can now overview the architecture in figure 7.7. As you can see all the phases we have described are present. The process is linear. It has no input beyond the mustache template. The JSON output is accompanied by warnings and errors, which we write to the console; this is not shown in the diagram.

7.6.7 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. Partials in mustache behave as if the template was inlined. We therefore merge the contents of the partials template root section with the contents of the parent section[15]. Unescaped variables will return a list of their nodes as their value. Previous nodes will be included if they are text nodes and have merged with the content of the variable.

[13] See section 7.2.1

[14] Because of the *unescaped_pos* filter, this filter has become superfluous.

[15] As detailed in section 7.6.4 we handle partials like new templates, this means they only have one iteration like the root section.

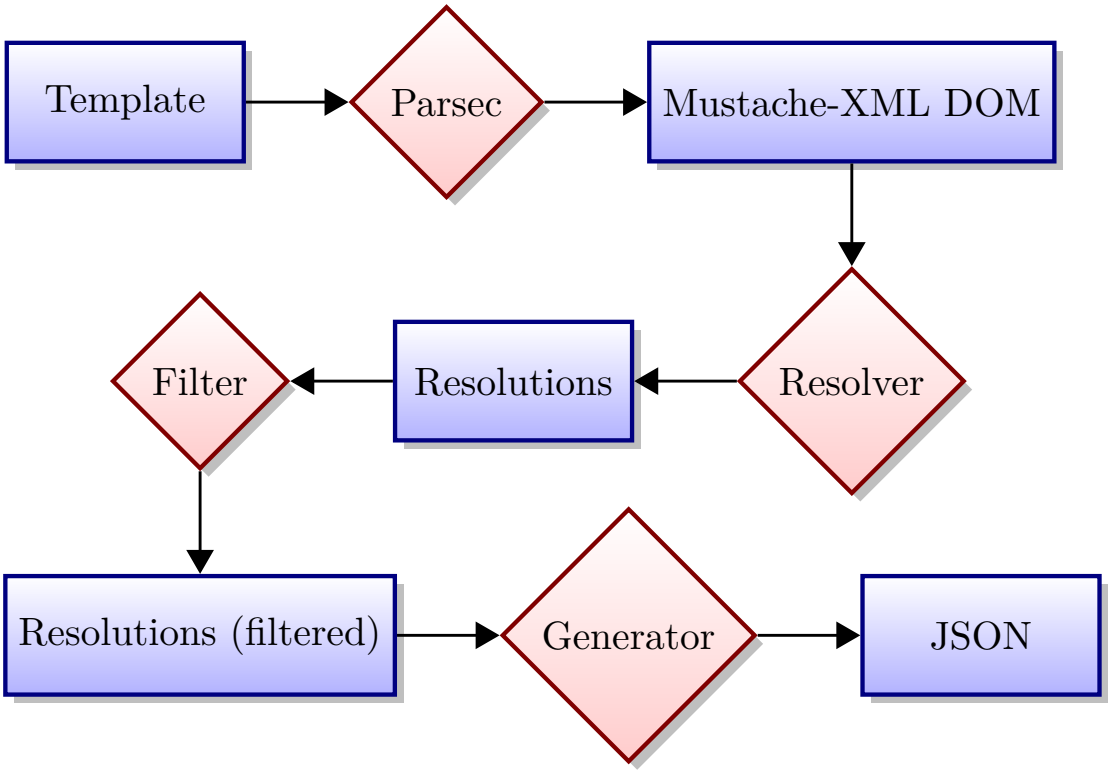


Figure 7.7: Architectural diagram of the pre-parser tool

Parent nodes

We add a second useful value to the return value of variables. Often times a developer 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.

7.7 "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".

Chapter 8

Demo applications

This chapter describes two applications utilizing Comb to make a case for its relevance in the web development area and to illustrate the internal workings of Comb. The first application shows an actual use case of Comb and highlights its strengths by comparison with jQuery. To that end the application has been developed in four different versions.

The second application demonstrates the internal workings of Comb by using Combs ability to analyze templates and extract the original dataset fed into the template engine. The structure of the dataset will then be used to build a form to allow changing its values.

8.1 Movie Database #2

In the exploratory prototype (chapter 4) we created a Movie Database to organize information about the cast, year of release as well as the plot and synopsis of a movie. We reimplemented the same application with Comb this time.

We use chaplin[1] to structure our application on the client-side, while a PHP REST framework together with php-activerecord functions as a communication layer to the MySQL database for the client. The client code is written in CoffeeScript[2].

The document root of the web application presents the user with two links to the same Movie Database implemented in two slightly different ways. The first link leads to an implementation using the CSS and HTML structure from our exploratory prototype. The second link leads to a different version of the application where we use the Twitter Bootstrap[3] library to style the interface.

In both implementations, Movies can be browsed, created and edited. Examples of both versions can be seen in figures 8.1 and 8.2

[1] See appendix A.4
[2] See section 7.6.1
[3] See appendix A.3

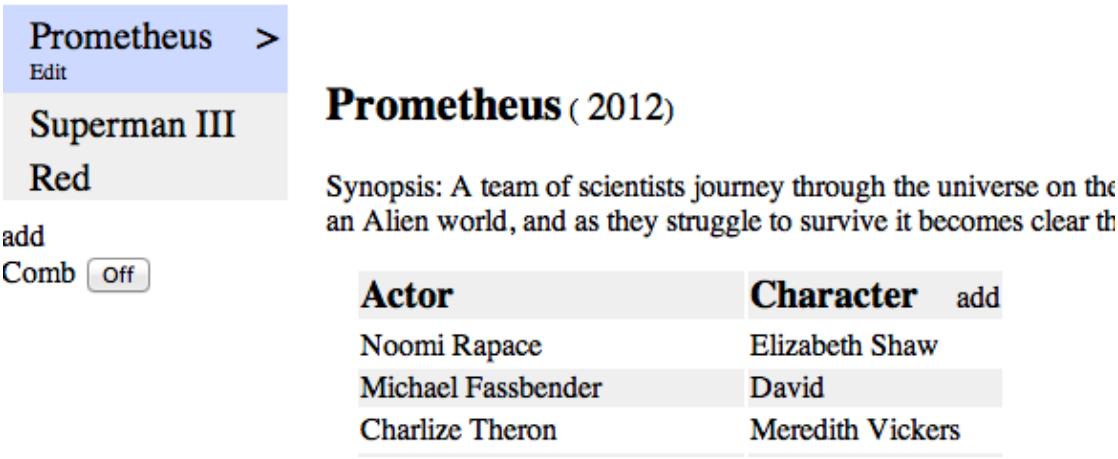


Figure 8.1: Original prototype implementation of the Movie Database application

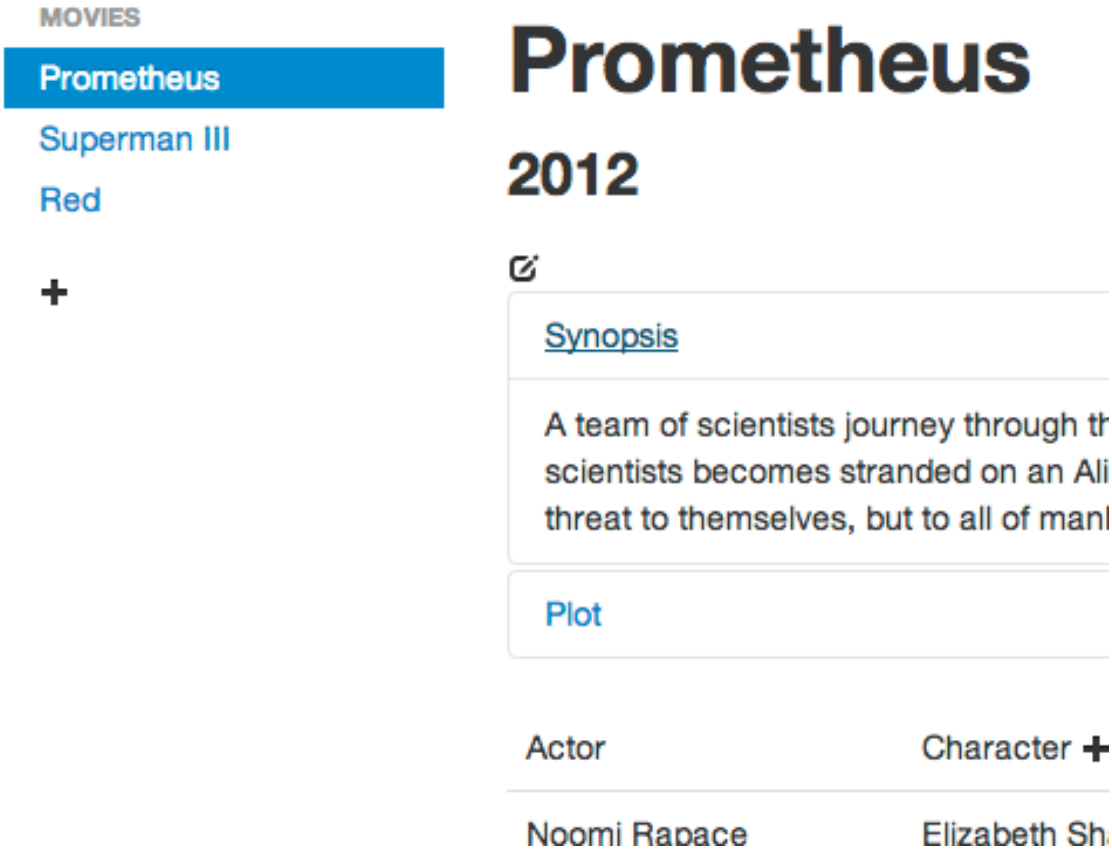


Figure 8.2: Bootstrap implementation of the Movie Database application

```
@$list = @$ @listSelector
for el in @$list.children()
  view = new MovieView {el}
  @subview "itemView:#{view.model.cid}", view
  # Silent push, we do not want chaplin to create a new view
  @collection.push view.model, silent: true
```

Figure 8.3: Initializing Movie Views

8.1.1 Architecture

The HTML page is built with server-side mustache templates. Interactivity is added once the client application has started. We use chaplin views to split sections of the web application into the following manageable parts:

- *Movies*: A ViewCollection representing the list of movies available in the database
- *Movie*: A View managing the DOM subtree of a single movie.
- *Cast*: A ViewCollection containing all roles in a movie.
- *Role*: A single Role in the Cast, this is the relationship connecting a movie and an actor.
- *Actor*: An actor playing a role

When initializing the client application, we bind DOM subtrees from the HTML created by the server to the corresponding Views. Each view is responsible for retrieving values from the DOM to populate the Model it is holding, while also initializing subviews that belong to it. In the case of the Movies CollectionView, we iterate over the list of movies and instantiate a new Movie View for each list element we encounter. Once a Movie View has been created we access the Model, which the View has populated with the values retrieved from the rendered template, and push it onto our collection of movies. The corresponding code can be seen in figure 8.3.

Once a View has populated its Model, it binds event handlers to any buttons in the interface it should respond to when clicked.

8.1.2 Retrieving DOM values

The original version of the Movie Database from our exploratory prototype comes in two flavors: jQuery and Comb. These flavors refer to the strategy we use in a View to retrieve values from the DOM and populate our Models with. We can switch between the strategies by clicking on the On/Off button below the list of Movies.

```
@model.set 'id', (@$('>details').attr 'id').substring 6
@model.set 'title', @$('summary.title').text()
@model.set 'year', @$('span.year').text()
@model.set 'synopsis', @$('summary.synopsis span').text()
@model.set 'plot', @$('details.plot p').text()
```

Figure 8.4: Using jQuery, we populate a Model by using selectors matching the template of the exploratory prototype

```
@model.set 'id', @data.id.value
@model.set 'title', @data.title.value
@model.set 'year', @data.year.value
@model.set 'synopsis', @data.synopsis.value
@model.set 'plot', @data.plot.value
```

Figure 8.5: To populate a Model with Comb, we access the values in the dataset, by the names the matching Model properties has on the server.

In the jQuery strategy we use CSS selectors and jQuery accessors[4] to pinpoint values in the DOM. Once a value is retrieved, we may also need to strip some of the parts that do not belong to the actual value present in the database. In figure 8.4 we retrieve all the values of a Movie and add them to the Movie Model.

Figure 8.5 shows the same procedure, only here we use Comb to retrieve those values. We can clearly see that there are no references to the structure of the movie template, instead we use the exact same identifiers to set model properties as we use to access values in the dataset. This is no coincidence, the properties of our client-side Movie Model and those of the server-side Movie Model are the same. Therefore we must also use them when creating our templates, which leaves us with a dataset, that mirrors its server-side counterpart.

8.1.3 Changing templates

The independence from the template structure in our Comb strategy is exemplified by our second implementation using the Bootstrap library. This version also exists in both a jQuery and Comb flavor.

The structure of the HTML is quite different from our original implementation, because Bootstrap requires the developer to set up his layout in a grid system[5]. To expand the synopsis and plot of a movie we use the "accordion" widget[6]. This is a departure from our original layout where the native HTML5 elements `details` and

[4] e.g. `.attr('name')` or `.text()`

[5] http://twitter.github.com/bootstrap/scaffolding.html#gridSystem

[6] http://twitter.github.com/bootstrap/javascript.html#collapse

```
@model.set 'id', (@$('>div').attr 'id').substring 6
@model.set 'title', @$('h1').text()
@model.set 'year', @$('h3').text()
@model.set 'synopsis', @$("#plot-#{@model.id} div").text()
@model.set 'plot', @$("#synopsis-#{@model.id} div").text()
```

Figure 8.6: The jQuery selectors used to extract values from the rendered template have all changed when compared with the previous selectors in figure 8.4

```
@model.set 'id', @data.id.value
@model.set 'title', @data.title.value
@model.set 'year', @data.year.value
@model.set 'synopsis', @data.synopsis.value
@model.set 'plot', @data.plot.value
```

Figure 8.7: The data extraction code utilizing Comb needed no modifications

`summary` were used.

In figure 8.6 we can see how the selectors when using the jQuery strategy for retrieving DOM values have changed. Contrasting this with the Comb strategy shown in figure 8.7 leaves us with a clear picture of the robustness Comb has with respect to retrieving values from the DOM. The code in figure 8.5 and 8.7 is exactly the same.

8.2 Template Editor

The Template Editor is an application with which we can explore how Comb functions. It also serves as a tool for testing Comb itself. We present the user with an interface where templates can be loaded from the navigation bar at the top of the page. The templates all originate from our Movie Database application. Figure 8.8 shows the navigation bar with the template selection menu open. Upon selecting a template, the application loads the template and the Comb file using require.js (7.6.1). The template is then rendered[7] with no initial dataset and the results are displayed in the right view port. Beneath it the template source code is also shown verbatim.

In the left view port a form with various buttons and input fields appears once the template has been rendered. In figure 8.9 we can see how this form can be used to control all aspects of the dataset that is fed into the template rendering engine.

As such, the application itself is rather uninteresting. The more interesting part is the underlying architecture.

[7] We use the client-side mustache.js template engine to render templates directly in the browser.

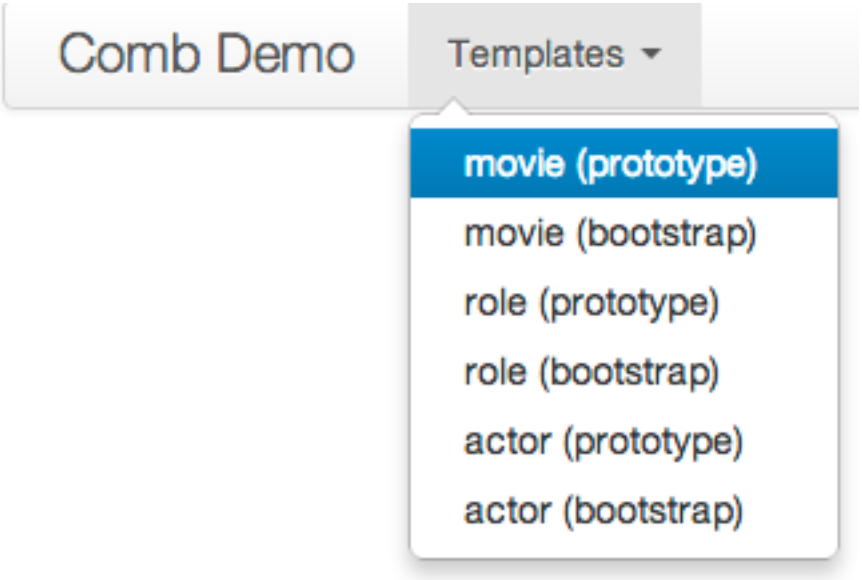


Figure 8.8: The template selection menu from the editor application

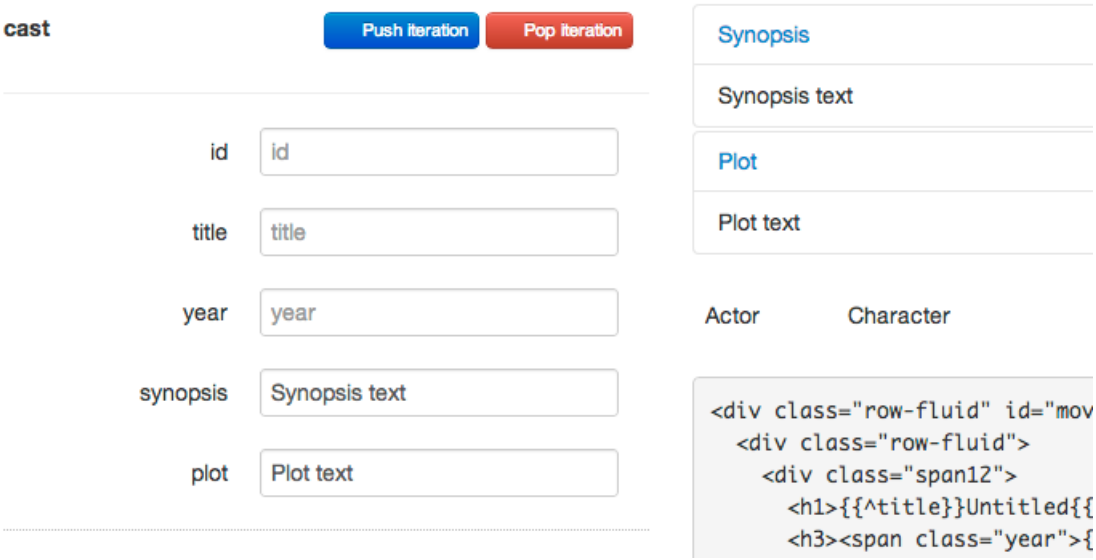


Figure 8.9: Editing the dataset for a template with a form

```
{{#section}}<fieldset class="section">
  <h5>{{name}}</h5>
  <div class="buttons">
    <button class="btn btn-mini btn-primary" data-target="{{name}}"><
        icon class="icon-plus"/> Push iteration</button>
    <button class="btn btn-mini btn-danger" data-target="{{name}}"><
        icon class="icon-minus"/> Pop iteration</button>
  </div>
  {{#iterations}}<div class="iterations">{{>mustache}}</div>{{/
      iterations}}
  <hr/>
</fieldset>{{/section}}
{{#escaped}}<div class="control-group">
  <label class="control-label">{{name}}</label>
  <div class="controls">
    <input type="text" placeholder="{{name}}" value="{{value}}" />
  </div>
</div>{{/escaped}}
{{#unescaped}}<div class="control-group">
  <label class="control-label">{{name}}</label>
  <div class="controls">
    <textarea rows="3">{{nodes}}</textarea>
  </div>
</div>{{/unescaped}}
```

Figure 8.10: The file `mustache.mustache`. A mustache template intended for viewing values retrieved by comb.

8.2.1 Generating the form

To generate the form in the left view port, we transform the dataset retrieved from the rendered template in the right view port into a structure that fits the form template (figure 8.10), which we have written in mustache as well.

This template uses sections and partials to recursively iterate over the transformed data structure. Sections from the rendered template are displayed as a headline and two adjacent buttons. Variables become input fields or text areas depending on whether they are escaped. Partials are unfortunately not supported.

8.2.2 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 and text areas for the variables in much the same way.

8.2.3 Loopbacking Comb

The dataset entries from our form template correspond to 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 7.4.3) and variable values as empty strings (see appendix B.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 entries in an array, we re-render the loaded template[8].

[8] Section 9.5.1 explains this necessity in more detail.

Chapter 9

Evaluation

In this chapter we will assess the usefulness of Comb in light of the demo applications from chapter 8. Following this, we will analyze whether we achieved the goals we set ourselves. We will also examine potential improvements of Comb and discuss its shortcomings.

9.1 Assessment of Comb

9.1.1 Movie Database #2

The MovieDatabase #2 has highlighted various advantages of using Comb over normal jQuery selectors to extract values from rendered templates. The transition between two very different templates is seamless when using Comb, where none of the identifiers we used to retrieve values from the dataset required any modification. This stands in stark contrast with the jQuery version of our demo, where in fact none of the selectors stayed the same.

In our small example this need for adjustment may seem inconsequential. However, when projected onto larger projects it is easy to imagine how this process may become a daunting task: First the developer will need to locate all Views the change affects. Following that, he will in his mind have to construct the DOM tree of the template in question to check whether a given selector requires adjustment. If an adjustment is required, he will also need to make sure that it captures only the elements that he intends to capture. The template may even need to be changed because matching only a specific element is impossible or too complex. During all of this the developer has to switch back and forth between the template and the view. With Comb, this process is not necessary, the developer (or template designer) simply runs our pre-parser tool to refresh the comb file and continues with his next task.

Consider also how the id of a movie was fetched in the jQuery version (figure 9.1). The id is embedded in an element attribute value with a prefix. To arrive only at the id, we had to strip away the "movie-" part of the attribute value by discarding the first 6 characters of the "id" attribute. In an era where first-person shooters can be

```
@model.set 'id', (@$('>details').attr 'id').substring 6
```

Figure 9.1: Retrieving the id of a movie with jQuery

```
@model.set 'id', @data.id.value
```

Figure 9.2: Retrieving the id of a movie with Comb

played directly in the browser and processors are manufactured on the nanometer scale, counting characters should not be part of the daily routine of a web developer. Looking at the Comb alternative in figure 9.2, we can see that Comb already extracted the id from the attribute and discarded the "movie-" prefix. We did not even have to look up where the value was placed in the template. This kind of dataset extraction is also visible when we examine the list of movies in the bootstrap version of our application in figure 9.3. In our application we did not need to extract data from this specific part of the template, since the subsequent HTML for the actual tab-panes contained the same and additional data. If we were to determine the number of movies using this list with jQuery, we would count the number of list items it contains. Subsequently we would need to subtract 2 from this number to account for the additional list headers. Alternatively we could add a class name to the list item in the `{{#movies}}` loop, making it distinguishable from the header list items. Figure 9.4 illustrates these possible selectors.

With Comb it is possible to retrieve such *implicit* values quite easily: We simply access the `movies` array in the extracted dataset as shown in figure 9.5. In addition to this strategy being simpler we also conserve the meaning behind our code. Any developer looking at the code will immediately understand what the number from the `length` property represents without having to examine the associated template. This holds true especially for the first selector of figure 9.4 where the magic number "2" seems to appear out of nowhere. Adding classes to the template for the sole purpose of being able to extract data from the rendered template like we do in the second selector of figure 9.4 is not a sound strategy. We pollute the CSS selector namespace with classes that

```
<ul class="nav nav-list">
  <li class="nav-header">Movies</li>
  {{#movies}}<li><a data-toggle="tab" href="#movietab-{{id}}">{{title}}
      </a></li>{{/movies}}
  <li class="nav-header"><a href="#" id="add-movie"><i class="icon-plus
      "></i></a></li>
</ul>
```

Figure 9.3: Tab buttons for the list of movies in the bootstrap version of the Movie Database

```
@$('ul>li').length-2

// requires adding a "movie" class to the list items
@$('ul>li.movie').length

// Additional possibility
@$('ul>li[class!="nav-header"]').length
```

Figure 9.4: Three selectors to count the number of movies in 9.3

```
@data.movies.length
```

Figure 9.5: Counting the number of movies with Comb

have nothing to do with the layout and design of the web application. Some scenarios may however require us to do exactly that even when using Comb, because there is no section surrounding a group of tags. We can however extend Comb to also recognize mustache comments, they do not change a rendered template in any way, but give the developer the possibility to annotate parts of a template. In section 9.3.1 we examine this extension in more detail.

Mustache comment annotations could be used to take advantage of the `parentNode` property we return for all variables. In the Comb version of our Movie Database we still used jQuery selectors to access parts of the HTML where we wanted to listen for click events from the user. Specifically we wanted to take action, when the plus-signs for adding a movie or a new role in a movie were clicked. Instead of relying on those selectors we could have added a variable to an attribute of the button and accessed the `parentNode` property. The identifier of that variable should of course not reference any identifier we might expect in the dataset, since we only intend to use it to access its parent node and not its value. In light of this use-case a more sound approach would indeed be to use mustache comments instead.

The `parentNode` property is used in our Movie Database for another purpose: We update the properties of a movie Model while the user is editing it. To bind event listeners to the nodes containing the editable text, we wrap the node referenced from the dataset in a jQuery object and leverage its event abstraction layer to update the corresponding property on the model by listening to "keyup" events. This way any changes to text nodes by the user will effect a corresponding change in the Model attributes as well. We abstracted this process and added it to the abstract View object all our Views extend, the task to make an element editable then becomes very simple: `@editable('title')` is all it takes. Since the identifier in the dataset is identical to the property name on the Model, `editable()` requires only one parameter to listen to changes on the parent node, extract the new value from it and update the corresponding model property.

The instances in which we use jQuery are not grounded in a lack of functionality

in Comb; Our goal has never been to replace jQuery. In section 5.2 we did however mention the desire to fit Comb into an ecosystem of existing software. Being able to wrap properties of our dataset with a jQuery object to empower it with additional features is a step in exactly that direction.

However, Comb does in no way require the developer to use jQuery for such tasks. Tools such as prototype.js[1], a library that extends native JavaScript object prototypes such as `Array` with additional methods, and zepto.js[2], a light-weight version of jQuery, may be used instead. By allowing diversity like this we follow another rule set out in "The Art of Unix Programming":

> *Rule of Diversity: Distrust all claims for "one true way".*
>
> [8, Chapter 1]

The need for us to implement an `editable()` function highlights the price we pay for this diversity. We can currently not bind properties of backbone models to values in the dataset as easily as we intended to do at the conclusion of chapter 3. In chapter 5.1 we revised that goal for a reason we can state more clearly now:

> Imbuing Comb with the ability to detect changes in the DOM via jQuery and interfacing directly with the backbone API forces us to forego integration with any other type of framework or tool.

A more sensible approach to such an integration would be a separate layer which handles communication between Comb and whichever other library we wish to integrate with. Such a layer has the possibility to choose libraries that enhance Combs features in ways specific to its purpose.

The developer from the scenario in chapter 3 for example has no need for an entire client-side Model-View-Controller framework. For him a simple form layer that retrieves values and submits them as JSON to a URL would suffice.

9.1.2 Template Editor

The Template Editor shows another side of Comb where the templates do not need to be rendered on the server for our tool to be useful. In the template editor we render a recursive template on the client to generate a form. The form is made interactive by binding event listeners to the element referenced in the `parentNode` property of variables. In fact we also make client-side rendered templates interactive in the Movie Database when a new movie or role is created.

9.2 Reviewing our goal

In the requirements for the prototype (chapter 3) we set out to couple the dataset fed into the server-side template engine with the models on the client-side. In section 5.1

[1] `http://prototypejs.org/`
[2] `http://zeptojs.com/`

we revised this requirement by splitting the task into two distinct actions and changed the goal to implementing the first part only: Extracting the dataset from a rendered template.

The architecture we devised to achieve this goal aimed at a two-staged process where we first retrieve information from templates and later use that information to parse rendered templates. The demo applications from chapter 8 integrate Comb to do exactly that. They render templates with a given dataset, extract the data from the rendered templates and employ that data in the rest of the application.

By keeping the footprint of our application low we have also increased the maintainability of our tool. As stated in chapter 5 we managed to create a "nucleus of code, which only incorporates the parts that are necessary and unique to our solution". Through increased maintainability we follow another rule set out by Eric Raymond:

> *Rule of Extensibility: Design for the future, because it will be here sooner than you think.* [8, Chapter 1]

To extract data, the Comb client does not depend on any client libraries, which may need upgrades or become superseded by better projects. We achieved this not by copying functionality from other libraries directly into the code of the client library, but by restricting the feature-set of Comb to the core functionality — a "nucleus of code".

9.3 Pre-Parser tool improvements

As with any piece of software, there is always room for improvement. Comb is no different. In this section we will look at parts of the pre-parser that can be improved.

9.3.1 Parser

Extending the template grammar

Looking at the Mustache-XML EBNF[3] we can see how the grammar is simplistic, when compared with the full XML EBNF [3, section 2/#sec-documents] or to HTML [2][4]. The intention is not to build a fully capable HTML parser, but the question remains whether our grammar matches a superset which is generic enough to allow for all templates that can be converted into a DOM. Among the possible problems that may be encountered is namespacing of tag names. The `identLetter` [6][5] property of our XML token parser in our pre-parser tool lists the allowed letters in an identifier, which in our case are tag names and attribute names. Among those letters is the colon, which separates tag names from their namespace. Comb should therefore be able to recognize namespaced tag names since the DOM API returns their name with the namespace included.

[3] figure 7.2

[4] HTML 5 has no EBNF, the reason for that is detailed at `http://lists.w3.org/Archives/Public/www-tag/2009Sep/0013.html`

[5] Go to: /doc/html/Text-Parsec-Token.html#v:identLetter

Standalone mustache tag lines

As we describe in appendix B.9 mustache removes lines containing only white space and a mustache section tag. Our parser does currently not account for this detail. This causes our client library to expect a whitespace line where in fact there is none. To work around this problem the demo application runs a modified version of mustache.js where this feature is removed[6].

Mustache comments

In section 7.1.2 we deemed a mustache comment not relevant "since it does not output any content our client library can retrieve". In section 7.6.7 however we introduced parent nodes into the dataset which we return together with variable values. This addition makes comments useful even though they do not output any data. By placing comments in key locations in a template, the developer can access nodes in a rendered template by referencing those comments and accessing their parent node. Because comments do not affect the rendered template we can even adjust our EBNF to accommodate comments in all parts of a template. This includes the space between XML attributes.

Comments do however not contain identifiers but free form strings. To simplify referencing we could consider anything up to a set of separators as the identifier. The alternative would be to simply require the developer to specify the entire string.

Although this feature addition can be considered a significant improvement to the possibilities of our tool, it cannot be deemed a shortcoming. Our goal was to retrieve the original dataset from a rendered template. Comments do not receive any dataset value and do not output any value, as such they are out of scope.

9.3.2 Resolver

Inverted sections

In section 6.1 we determined that we would not be able to determine the origin of a key in the dataset precisely, when the context stack is greater than one. This holds true for all cases except the inverted section. The inverted section does not push a new context on top of our existing context stack because it is only rendered when its identifier points at an empty list[7] in the dataset. This is guaranteed by the specification[8]:

> *This section MUST NOT be rendered unless the data list is empty.*
> [4, inverted.yml]

We can be certain that all identifiers referenced in an inverted section do not belong to the identifier of the section. This fact allows us to lift the parsed values into the parent context.

[6]Specifically we removed the highlighted line seen at `https://github.com/janl/mustache.js/blob/master/mustache.js#L512`

[7]Or the data it points at is coerced into an empty list

[8]Read: "This section MUST be rendered if and only if the data list is empty."

Lookahead

The filter in our pre-parse tool currently requires mustache tags to be separated by strings or XML tags. This restriction is necessary to ensure that our client library can recognize the end of variables as well as the beginning and end of section iterations. The parser employs a limited form of lookahead when it checks whether an iteration is followed by another iteration. Variables stop parsing as soon as they encounter the first occurrence of their next sibling. This is a rather limiting restriction which can hinder developers in writing templates. We can overcome this restriction by enhancing the parsing capabilities of our client library. From the pre-parser tool we only need to remove the *no_lookahead* function located in the list of filters.

To create a simplistic form of lookahead we can catch errors thrown by mustache tag objects when they are unable to verify their siblings and instead try alternative possibilities until we succeed. This is however not only resource intensive but may result in a faulty dataset extraction, because more than one combination of possibilities can be applied to a DOM tree.

A more sound approach requires extending our pre-parser tool as well. It can combine the same aforementioned possibilities of parsing a template and replacing any unknowns (i.e. variables) with wildcards. Whether a section is entered is also not knowable when analyzing a template, but we can create a binary tree of possibilities, where each node represents a list of XML tags. Upon encountering a section we branch and continue the tree generation. This tree generation can continue until we encounter the end of a DOM tree. We can also set a limit on the depth of the tree, thereby limiting the amount of possibilities our client library has to try. Once the tree is generated the filter may analyze it and print errors if two paths are indistinguishable. The client library can for example use this tree to determine whether a section containing a variable as its first child and a variable as its next sibling has another iteration by analyzing the content that immediately follows this ambiguous (variable-) text node. Such an improvement would also obsolete our filter that checks whether the first child and next node of a section can be confused (*ambiguous_boundaries*, section 7.6.6).

Variable boundary ambiguity

Our lookahead improvement cannot help alleviate a similar problem a developer may encounter with two or more variables in one text node as shown in figure 9.6. The parser cannot in any meaningful way split a string originating from two variables when they are not separated by text. It is impossible to determine how much text belongs to the first variable and how much text belongs to the second variable[9].

Separating variables with text can alleviate this problem only if the variable values do not contain the separator itself. Consider the example in figure 9.7, here the library cannot split the string in any meaningful way either. We know that `var_two` at least contains “slashes”, because there is no slash after the second variable in the template. We can also determine that `var_one` at least contains “path”, because we

[9]This also applies to more than two variables of course.

(a) Template

```
<span>{{var_one}}{{var_two}}</span>
```

(b) Result

```
<span>This text can be split between the two variables or belong to
   only one.</span>
```

Figure 9.6: Two variables without a separator between them

(a) Template

```
<a href="http://www.example.com/{{var_one}}/{{var_two}</a>
```

(b) Result

```
<a href="http://www.example.com/path/with/multiple/slashes</a>
```

Figure 9.7: Two variables separated with a '/'

would see two subsequent slashes if `var_one` was empty. Of the remaining slashes in the "/with/multiple/" string the original separator can be any one of them.

9.3.3 Filter

Suggestions

Thanks to the Parsec library, the developer receives helpful messages when a template cannot be parsed. Our filter component also outputs detailed error messages. The errors can be improved by accompanying them with suggestions for how to resolve them. Filters have access to enough information to calculate how an error may be corrected.

9.3.4 Generator

Notification of changes

In section 9.1 we highlighted how the Movie Database #2 Comb version needed no modifications to the identifiers used to access values in the dataset. In some cases this may however be necessary, since a change may remove a variable or section entirely from the template. Identifiers may be renamed or part of the template is factored out into a partial.

We can warn the developer of such changes if we are asked to overwrite an existing comb file. Assuming the file we are about to overwrite originates from the old template, we can parse its content and compare it with the newly generated content. Armed with this information, the pre-parser tool can alert the developer to any changes between two

versions of a template that require the code, which accesses the extracted dataset, to be changed.

More importantly, the developer should also be aware when this is *not* necessary. If regenerating a comb file produces no alerts the developer can be *certain* that no client-side code needs adjustment.

9.4 Client library improvements

9.4.1 Mustache tags as tag and attribute names

In section 7.1.2 we listed the inability to specify mustache tags in XML tag names and attribute names as a restriction on the templates our tool can parse. We may however be able to allow this by changing our grammar and by modifying our resolution phase to link mustache tags in a more generic way. Our client library addresses DOM elements by child node indices instead of tag names and can therefore find these tags with no modifications. Tag names are however used to determine various aspects of mustache tag boundaries. These checks will have to be rewritten to allow for variable tag names.

9.4.2 Fall back to lambda sections

Lambda sections may currently cause unexpected errors in our client library. Comb assumes every section is a normal section and throws errors if it does not find what is expected to be found. Instead of stopping the parsing process the client library should fall back to the assumption that the current section is a lambda section and simply collect its contents. Similar to ambiguous variable boundaries however this strategy may be problematic. The parser can in many cases not be sure when a lambda section is actually finished because its content can vary wildly.

9.4.3 Partials must be only children

The filter of our pre-parser tool requires partials to be contained within an XML tag as its only child. This is done to simplify the parsing process, which would otherwise become more complicated by having to account for siblings of root nodes in a template.

9.4.4 Unescaped variables as last children

Much like lambda sections the content of unescaped variables is hard to predict and can be confused with the next sibling of the variable. We added the filter *unescaped_pos* in the filter component of our pre-parser tool for that reason. With the guarantee that an unescaped variable is always the last child of an XML tag, our client library can add all nodes it encounters to the value list of an unescaped variable — beginnning at the location of that variable (and until it finds no more nodes).

9.4.5 Improving the retrieved dataset

The Rule of Least Surprise states:

> *Rule of Least Surprise: In interface design, always do the least surprising thing.* [8, Chapter 1]

We strive to recreate the original dataset passed into the template engine as closely as possible. This is in part motivated by the above rule and grounded in the fact that the developer may be more familiar with the structure of the dataset passed to the template engine than the structure of the template[10] The structure of a template is however all our tool knows. This structure therefore dictates the structure of the dataset retrieved from the rendered template. In the following section we will propose improvements that can counteract this effect and bring the retrieved dataset closer to its origins.

Properties of values

When a property of an identifier is accessed, we know this identifier points at an object rather than a list. If we in the same template encounter a section using the same identifier, we can recreate the original dataset more precisely by having the section return an object instead of a list with a single item containing that object.

9.4.6 Decorating lists

A more general approach to improving the retrieved dataset is to introduce ambiguity. When a section intended as an if block is retrieved from the rendered template the client library returns an array regardless of the amount of iterations the section has performed[11] (which in the case of if blocks should be zero or one). Since the original dataset held a single object at that position, the developer will not expect to find an array.

Our tool has no way of determining the semantic meaning of a section in a template. Fortunately we do not need to be certain what the intended usage of the section is to rebuild the original dataset structure. If a section iterated only once we can decorate the resulting array with every property of the first entry in the array. Whichever version the developer expected, he will now be able to access it that way. This approach will however not work when there are no iterations since we cannot let a key in our dataset be both an empty list and the value false or null[12]. This means the developer will not be able to check for the existence of an object with a simple if-block, he will have to check if a list is empty.

[10] Also, we want to ease integration for additional layers. The common denominator in that case will ostensibly be the structure of the data/entity relationships and not the structure of the templates.

[11] not accounting for effects of the modification proposed in section 9.4.5

[12] Double negating an array in JavaScript (`!![]`) returns true instead of false unfortunately (for our use-case)

9.5 Future Work

9.5.1 Modifying rendered templates

The `update()` function[13] returned by variables is helpful when the developer intends to update strings in the template. Similarly we demonstrated in section 8.2.3 how our demo application could "push" and "pop" iterations onto and from lists in a dataset represented as sections in the template and then re-render the template with the new values to display the changes. This re-rendering performs a lot of unnecessary work considering it is only the rendered template content of one section that needs to be appended or removed.

If our pre-parser tool were to extract the template content of a section, we could append an additional iteration by only rendering this content with the newly pushed item using *mustache.js*. If we are to remove an iteration we simply remove the nodes that were recognized as belonging to that iteration when the rendered template was parsed.

9.5.2 Two-way binding of models

After implementing the extension suggested in section 9.5.1 layers can be constructed as mediators between Comb and client model libraries. As an example of such a library we will choose backbone. The backbone library[14] features an event architecture which allows the developer to be notified when properties of objects change. Using this architecture we can create a layer that allows the developer to specify which identifier in the dataset returned by Comb a backbone model property corresponds to. Lists in the dataset may be mapped to collections.

Once such a mapping is established, our layer can propagate changes in the models to the DOM. Reversing this effect is also possible by listening for changes in the DOM (e.g. a change of an input field value) and updating the mapped model instead. Such a layer would implement the feature we originally set out to implement, but chose to forgo in the revised requirements of chapter 5. A feature like this will surely also encourage the use of Comb even if the template was rendered on the client, making the retrieval of values from the DOM a secondary priority.

9.6 Emergent Properties

In section 5.1 we outlined the rules from Eric Steven Raymonds book "The Art of Unix Programming". Although, the rules are meant for developing Unix programs, we chose to apply them to the development of Comb. Comb should not be regarded as a framework but a tool. It does not dictate what software the developer should use (beyond mustache)[15] and it does not expect a specific structural layout of a web

[13] See section 7.6.7

[14] See section 4.1.1

[15] In fact, Comb is even agnostic about HTTP. How the comb file is transfered is up to the developer.

application. This fact allows developers to integrate Comb in a larger set of software combinations, extending its applicability. In fact, Comb may be used in ways we may not have outlined in this evaluation: Node.js[16] allows execution of JavaScript on the server-side. With node.js Comb could be used in the parsing of rendered templates without involving any browser[17].

By choosing to develop Comb as a tool we can also observe a set of emergent properties that have become apparent during the implementation, usage and evaluation of it.

- Templates are verified for proper HTML syntax through the pre-parser tool.
- Data extraction implies a one-way communication, but using `parentNode` allows Comb to supply the developer with additional information, that can be used for various kinds of extensions supporting DOM interactions.
- The `update()` function allows for some limited DOM manipulation. In section 9.5.1 we described how this could be extended to sections.
- Although we focused on retrieving values from templates that were rendered on the server, we have seen how the demo applications were able to take advantage of Comb even when the templates were rendered on the client[18].

[16] See appendix A.6
[17] This could also be a way to write unit tests for Comb.
[18] See section 9.1.2

Chapter 10

Related Work

In this chapter we will discuss various tools and frameworks that bear similarities to Comb through either their goals or their domain.

10.1 Similar goals

First we will discuss two tools which have goals similar to Comb.

10.1.1 Template::Extract

Template::Extract[9] is a tool written as a perl module with a functionality very similar to that of Comb. It is written for the "Template Toolkit" template language and allows the developer to extract values from a rendered template when given the template. This extraction is accomplished by compiling a regular expression based on the template and applying it to the rendered template.

> *Extraction is done by transforming the result from Template::Parser to a highly esoteric regular expression, which utilizes the (?{...}) construct to insert matched parameters into the hash reference.* [9]

Our revised goal exactly matches Template::Extract, Comb however operates on the Document Object Model while Template::Extract operates on a single string. In chapter 6 we evaluated that strategy but discovered that the strings returned by the `innerHTML` property of DOM elements differ between browsers and do not reflect the original rendered template. By extracting values from the DOM we also gain the added advantage of retaining references to the nodes the strings were extracted from[1]. This additional piece of information allows Comb to become more than a simple value extraction library. The emergent properties discussed in section 9.6 and the potential additional layers Comb supports differentiate Comb from the sole purpose of extracting data from templates.

[1] as detailed in section 7.6.7

10.1.2 Rivet.js

On the other side of the spectrum we have Rivet.js[2], which is a tool that can bind templates written in plain HTML to Model frameworks like backbone.js[3]. Changes in the Model will be mirrored in the rendered template and vice-versa. As such this tool is in its features very similar to the original goal we formulated in chapter 3.

However, rivet.js uses its own template language to achieve this goal. The rendering takes place when the models are bound to the template. This precludes any form of server-side rendering without node.js[4]. In fact, rendering a template on the server-side and transmitting it to the client is not possible without losing the bond between the DOM and the Model. Rivet.js is however an example of how a two-way binding layer[5] for Comb may function. It is also reminiscent of the architectural alternative we considered in section 6.3.1, in which we proposed a direct integration with the mustache template engines.

10.2 Similar domain

In this section we will discuss academic projects which touch upon the same domain as Comb. Although they employ their own template language, their strength draws from the ability to use meta-information about the HTML they generate.

10.2.1 MAWL

Mawl[1] is a framework and domain-specific language developed in 1999 to aid the handling of HTML forms as well as telephone forms. It does this by supplying the developer with a framework which compiles a set of MAWL templates and "sessions" to executables the browser can communicate with via the CGI on the web server.

The primary goal behind MAWL is to empower the developer with a better organization of the data flow in form-based services. This goal is achieved by creating a form abstraction language in which the semantics of forms can be specified while the presentation layer of these forms is handled by MAWL templates (MHTML). CGI programs handling the data flow need no longer be programmed in perl, Tcl or the Korn Shell, but can be compiled from said language.

A secondary advantageous property of MAWL is its ability to verify the type signature of the form abstraction against the structure of an MHTML template. Although the paper does not go into detail explaining how these templates are analyzed, a parallel can be drawn to our Comb pre-parser tool which employs Parsec to construct an abstract syntax tree. Much like MHTML is checked for consistency, our parser also verifies mustache templates and displays any possible errors[6].

[2] `http://rivetsjs.com/`
[3] See section 4.1.1
[4] See appendix A.6
[5] As discussed in section 9.5.2
[6] See section 9.6

> *First, the sessions and the MHTML can be independently analyzed to ensure that they are internally consistent. For the sessions, this means standard type checking and semantics checking. For MHTML templates, this means verifying that a template is legal MHTML.* [1]

10.2.2 Typed Dynamic Documents

The paper "A Type System for Dynamic Web Documents" by Anders Sandholm and Michael I. Schwartzbach proposes — like MAWL — a typed template language where *gaps* in a template can be filled with HTML fragments. The type of fragments these gaps can be filled with is inferred by their placement in a template. This allows the resulting content of a rendered template to fall within a predictable range. A similar yet less pronounced quality is recognizable in mustache when we compare escaped variables and unescaped variables. Together with the large list of mustache engine implementations we chose mustache primarily for its predictability[7]. However, since typed dynamic documents go beyond the guarantees mustache can deliver and implement an even stricter subset, a solution like Comb could also be implemented for this template language.

10.2.3 WASH/CGI

WASH/CGI [10] is a CGI library written in Haskell to ease integration with web servers over the Common Gateway Interface. This is done by supplying the developer with various type classes and other interfaces that allow him to write server-side web applications in a more abstract manner than directly interfacing with the CGI protocol. Like modern web application frameworks WASH/CGI provides a handler for session data, which is persisted with the help of the file system. The library generates HTML with the help of abstract data types and functions that construct and combine HTML elements.

10.2.4 SMLserver based combinator library

"Typing XHTML Web Applications in ML"[5] is a paper by Martin Elsman and Ken Friis Larsen describing a system with which XHTML documents can be created, whose conformity is guaranteed by the type system. In contrast to Haskell where there is support for type classes, this library solely uses phantom types to ensure XHTML validity.

The second contribution in this paper is a framework for interfacing scriptlets with the data submitted by forms generated using the library. It ensures consistency between the form generation and its retrieval much like Comb aids in the retrieval of datasets from templates.

10.2.5 Frameworks vs. Tools

A framework can be defined as an underlying structure, on which software can be built. Using proven paths they guide the developer in creating applications to increase efficiency

[7] See section 5.2.1

	Template language	Typed	Server language	Architecture
Comb	External	No	*Any*	Tool
Template::Extract	External	No	perl	Tool
Rivet.js	Internal	No	*Any*	Tool
MAWL	Internal	Yes	mawl	Framework
DynDoc	Internal	Yes	Java	Framework
WASH/CGI	Internal	Yes	Haskell	Framework
SMLserver library	Internal	Yes	ML	Framework

Figure 10.1: A comparison of related works

and the probability of a satisfiable outcome. Frameworks are usually comprised of not only a structure but a set of tools that are deeply interconnected.

The frameworks in the four aforementioned papers can be describe as such, because they supply the developer with a predefined set of tools that work together to enable him to achieve a goal. The assurance that the templates are created in a specific fashion allow them to ease the development of other parts of the application[8].

Tools on the other hand do not dictate a predefined structure and only supply the developer with a specific output given an input. They need not be versatile in their capabilities to be good tools. Their reduced set of requirements for performing makes them applicable in more than one situation. Because of their simplicity a developer can use a tool in situations it may not have been intended for and still produce a satisfying output. A tool is an extension of the wielders capabilities rather than a recipe written by a master chef.

In this light we can classify Comb as a tool. It does not require the developer to write in any new specially designed template language and dictates no structure the developer has to follow. In section 9.6 we even speculated about unintended use-cases for Comb. Comb extends capabilities by supplying the developer[9] with information about templates and their rendered version. How this information is used is not the concern of the tool.

Table 10.1 compares Comb and the related works using four parameters. "Template language" indicates whether the template language in use is internal to the tool or an external language from another project (*internal DSL* versus *external DSL*). Whether the template language is typed can be seen in the neighboring column. The "Server language" criterion lists which language the server for the web application can be programmed with. The last column designates the architecture of the project as either a tool or a framework

10.2.6 All or nothing

Frameworks such as MAWL, Dynamic Web Documents, WASH/CGI or the SMLserver library use an all-or-nothing approach when it comes to typing templates and extracting

[8] In both cases this would be the HTML forms

[9] ...or additional layers as described in section 9.1.1

additional information from them. Comb does not operate in the exact same domain as those frameworks, but it stands to reason that the applicability of our tool is greater because of its framework agnosticism.

The highly advantageous guarantees these frameworks supply may fall on deaf ears when they require developers to switch to an unfamiliar framework or even to a new server-side programming language. It is not easy to extract a part of these frameworks and integrate it into another application context.

This fact — which can hardly be described as a shortcoming, since it is an integral part of these frameworks uniqueness — is what allows Comb to stand apart. The template language is not one specifically designed for our purpose, we simply latch on to an already successful project that has been ported to many languages. Through this ubiquity Comb gains a higher chance of adoption.

Chapter 11

Conclusion

Comb is a tool to extract the original dataset fed into the Mustache template engine from the rendered template. Over the course of this thesis we have made a case for why Comb fits into the existing ecosystem of tools for web application development. We began with a prototype that incorporated not only the dataset extraction but also the coupling of said dataset to a client-side Model. This coupling led to the realization that the applicability of Comb can be greatly increased by solely focusing on the data extraction and leaving the utilization of that data up to the developer who uses our tool.

Comb was split into two parts: the pre-parser tool and the parser running on the client-side. This split freed us to implement the pre-parser in Haskell and keep the client library free of dependencies.

We contrasted Comb with existing solutions and found that most of these require the developer to switch to new template languages explicitly made for the problem they are trying to solve. We identified this strategy as a framework approach, which is a sound approach to the problem, but brings with it a reduced applicability. The framework approach is a general tendency we have discovered in academic papers regarding web applications and template languages. With Comb we have shown an alternate path to a solution. We believe the results of this thesis warrant a reevaluation of the current method of approach to the field of template languages for web applications. This belief is grounded in the lack of dependency requirements to the developer for using Comb and amalgamated with our choice of mustache as the template language, which because of its plentiful implementations in different languages can almost be described as "language agnostic".

Our contribution, Comb, is a tool that enables developers to build web applications with a more robust access to data embedded in rendered templates. Comb does not restrict the developer to any kind of architectural pattern and is built to be extended by other libraries and tools.

Appendix A

Technical Appendix

A.1 Downloads

The prototype application from chapter 4 and the Comb pre-parser and client library can be downloaded together with the demo applications from chapter 8 as a single gzipped tarball from the URL: `https://s3-eu-west-1.amazonaws.com/ingemann/thesis-package.tar.gz`

The SHA-1 sum of the archive is: `ae84102cbc8d7b509636f9dfe8b931fd83c26a6d`

A.2 Digital version of this thesis

This thesis exists as PDFs in three different versions: Print (adjusted grayscale for better contrast), PC (colors and single sided layout) and ebook (adjusted grayscale with dimensions optimized for the Amazon Kindle).

All versions are included in the archive from appendix A.1

A.3 Bootstrap

Bootstrap is a combination of JavaScript and CSS which supplies a developer with pre-styled components to build a web application. Colors and shapes have been tuned to increase user-friendliness. The website `http://twitter.github.com/bootstrap/` itself uses the package. The blue gradient buttons are easily recognizable and can be found in many websites across the web.

A.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 event handlers. Event handlers are functions, that are called when an event on a DOM node or an other object is triggered. 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 event handlers for the developer when the view is disposed, allowing the browser to free up memory.

A.5 TagSoup

TagSoup is a library for parsing HTML and XML. Its documentation is available at `http://hackage.haskell.org/package/tagsoup-0.12.8`

A.6 Node.js

NodeJS is built on the JavaScript runtime for the Chrome browser and allows execution of JavaScript outside the browser. It complements JavaScript with additional APIs to access resources like the file system. It is available at `http://nodejs.org/`

Appendix B

Mustache

Mustache[1] is a template engine for "logic-less" templates. According to the author this subtitle derives from the fact that there are no control-flow statements:

> *We call it "logic-less" because there are no if statements, else clauses, or for loops. Instead there are only tags. Some tags are replaced with a value, some nothing, and others a series of values. This document explains the different types of Mustache tags.* [11]

This statement is debatable, in section B.3 we will see how sections can act as if statements and loops, recursion is also possible by using partials as describe in section B.7. The language is however very restrictive and does for example not support embedded calculations, removing some of the "logic".

In mustache, 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.

B.1 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 to access those properties (e.g. `{{object.field}}`).

[1] `http://mustache.github.com/`

B.2 Unescaped variables

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

B.3 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 be 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 not of a list type, it is coerced into a list as follows: if the data is truthy (e.g. '!!data == true'), use a single-element list containing the data, otherwise use an empty list.* [4, sections.yml]

If the value is a list, the content of the section is 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[2]. Consider the example in figures B.1.

Here the placeholder "nickname" is used in two different contexts. The first usage occurs while the context 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.

Mustache[3] also allows the current item in a list iteration to be referenced with a special "dot" operator, as can be seen in figure B.2.

B.4 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 B.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, figure B.4 illustrates this method.

We can also access the message subject by using `{{.}}`, since values that are not lists are converted into lists by mustache[4]. Figure B.5 illustrates this shortcut.

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

[2] This is called the "context stack"

[3] excluding mustache.js, which may have some implementation problems, see `https://github.com/janl/mustache.js/issues/185`

[4] The subject will be coerced into a list containing the subject string as its first and only entry.

(a) Dataset

```
{ nickname: "andsens"
  messagecount: 2
  messages: [
    {subject: "How did the presentation go?",
     nickname: "carl"},
    {subject: "Welcome to Messageservice Inc.",
     nickname: "Messageservice Inc."}
  ]
}
```

(b) Template

```
<p>
  Hello {{nickname}},<br/>
  you have {{messagecount}} new messages:
</p>
<ul>
  {{\#messages}}
  <li>{{subject}} from {{nickname}}</li>
  {{/messages}}
</ul>
```

(c) Result

```
<p>
  Hello andsens,<br/>
  you have 2 new messages:
</p>
<ul>
  <li>How did the presentation go? from carl</li>
  <li>Welcome to Messageservice Inc. from Messageservice Inc.</li>
</ul>
```

Figure B.1: An example of a mustache section

(a) Dataset

```
["How did the presentation go? from carl",
"Welcome to Messageservice Inc. from Messageservice Inc."]
```

(b) Template

```
<ul>
  {{#messages}}
  <li>{{.}}</li>
  {{/messages}}
</ul>
```

(c) Result

```
<ul>
  <li>How did the presentation go? from carl</li>
  <li>Welcome to Messageservice Inc. from Messageservice Inc.</li>
</ul>
```

Figure B.2: Mustache template using the "." variable

```
{{#messages}}
<li>{{subject}} from {{nickname}}</li>
{{/messages}}
{{^messages}}
<li>You have no messages</li>
{{/messages}}
```

Figure B.3: Inverted sections in mustache templates

```
<li>
  {{#subject}}{{subject}}{{/subject}}
  {{^subject}}(No subject){{/subject}}
  from {{nickname}}
</li>
```

Figure B.4: If-else constructs with sections in mustache templates

```
<li>
  {{#subject}}{{.}}{{/subject}}
  {{^subject}}(No subject){{/subject}}
  from {{nickname}}
</li>
```

Figure B.5: Using the dot variable on strings

```
<li>
  {{subject}}{{^subject}}(No subject){{/subject}}
  from {{nickname}}
</li>
```

Figure B.6: Variables output empty strings in mustache, if there is no values in the dataset for them

further as can be seen in figure B.6. 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.

B.5 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. In figure B.7 we have bound a function to convert a unix epoch time stamp into a relative date[5] to the key "reldate".

B.6 Comments

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

B.7 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.

[5] ... or an approximation of that

(a) Data

```
{ reldate: function(text) {
   return 'some time ago';
 },
 messages: [
   { subject: "How did the presentation go?",
     nickname: "carl",
     sent: 1355328898 },
   { subject: "A disturbance in the force, I felt...",
     nickname: "Yoda",
     sent: 513136200}
 ]
}
```

(b) Result

```
<ul>
 {{#messages}}
 <li>{{subject}} from {{nickname}} sent {{#reldate}}{{sent}}{{/reldate
    }}</li>
 {{/messages}}
</ul>
```

(c) Template

```
<ul>
 <li>How did the presentation go? from carl sent some time ago</li>
 <li>A disturbance in the force, I felt... from Yoda sent some time
    ago</li>
</ul>
```

Figure B.7: An example of lambda sections in mustache templates

(a) Data

```
{ directories: [{
   name: "bin",
   direcories: []
   files: ["echo", "ls"]
  },
  {
   name: "usr",
   direcories: [{
    name:"local"
    files: ["notes.txt", "donotopen.jpg"]
   }],
   files: ["randomfile.bin"]
  }]
}
```

(b) Template, the partial named “filesystem” points at this template

```
<ul>
  {{#directories}}
  <li>{{name}}/ {{>filesystem}}</li>
  {{/directories}}
  {{#files}}
  <li>{{.}}</li>
  {{/files}}
</ul>
```

Figure B.8: Mustache templates can be recursive by utilizing partial tags

The inlining happens when a template is rendered. This enables mustache to render recursive data-structures. For example the template in figure B.8 will render a directory structure using nested lists.

B.8 Set delimiter

Mustache curly braces may clash with delmiters used in other languages, such as LaTeX. Using the "set delimiter" tag, the delimiter can be changed to something else. The equals sign is used to signify a change in delimiters. In figure B.9, we change the delimiters to PHP tags, write some text using the now inactive mustache tags and enable the normal mustache tags again.

(c) Resulting filesystem list

```
<ul>
  <li>bin/:
    <ul>
      <li>echo</li>
      <li>ls</li>
    </ul>
  </li>
  <li>usr/:
    <ul>
      <li>local/:
        <ul>
          <li>notes.txt</li>
          <li>donotopen.jpg</li>
        </ul>
      </li>
      <li>randomfile.bin</li>
    </ul>
  </li>
</ul>
```

Figure B.8: Mustache templates can be recursive by utilizing partial tags

```
Subject: {{subject}}
{{=<? ?>=}}
Nickname: <?nickname?>
Curly braces (and text): {{ This will be normal text enclosed by curly
    braces }}
<?={{ }}=?>
No output: {{undefined_variable}}
PHP tags: PHP begins with "<?" and ends with "?>"
```

Figure B.9: Usage of set delimiter tags in mustache templates

B.9 Removal of standalone lines

Mustache removes lines which beyond a section beginning or end contains only white space (i.e. standalone lines). This is done to match the expectation a developer may have when writing templates. The behavior is however not defined clearly:

> *Section and End Section tags SHOULD be treated as standalone when appropriate.* [4, sections.yml]

B.10 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.

Appendix C

Rules followed

This is an overview of the rules set by "The Art of Unix Programming" that we follow:

> - *Rule of Modularity: Section 7.5.1*
> - *Rule of Composition: Section 5.1*
> - *Rule of Simplicity: Section 5.1*
> - *Rule of Parsimony: Section 6.3.3*
> - *Rule of Least Surprise: Section 9.4.5*
> - *Rule of Diversity: Section 9.1.1*
> - *Rule of Extensibility: Section 9.2*
>
> [8, Chapter 1]

List of Figures

Bibliography

[1] David L. Atkins et al. “Mawl: A Domain-Specific Language for Form-Based Services”. In: *IEEE Transactions on Software Engineering* 25.3 (May/June 1999). Special Section: Domain-Specific Languages (DSL), pp. 334–346. ISSN: 0098-5589. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00798323 (cit. on pp. 71, 72).

[2] Robin Berjon et al. *A vocabulary and associated APIs for HTML and XHTML*. 2012. URL: http://www.w3.org/TR/2012/WD-html5-20121025/ (cit. on p. 62).

[3] Tim Bray et al. *Extensible Markup Language (XML) 1.1 (Second Edition)*. 2006. URL: http://www.w3.org/TR/2006/REC-xml11-20060816/ (cit. on p. 62).

[4] Pieter van de Bruggen. *The Mustache spec*. 2010. URL: https://github.com/mustache/spec/tree/72233f3ffda9e33915fd3022d0a9ebbcce265acd (cit. on pp. 41, 63, 79, 86).

[5] Martin Elsman and Ken Friis Larsen. “Typing XHTML Web Applications in ML”. In: *Practical Aspects of Declarative Languages (6th PADL'04)*. Ed. by Bharat Jayaraman. Vol. 3057. Lecture Notes in Computer Science (LNCS). Dallas, TX, USA: Springer-Verlag (New York), June 2004, pp. 224–238 (cit. on p. 72).

[6] Daan Leijen and Paolo Martini. *The parsec package*. 2012. URL: http://hackage.haskell.org/packages/archive/parsec/3.1.3 (cit. on p. 62).

[7] Terence John Parr. “Enforcing strict model-view separation in template engines”. In: *Proceedings of the 13th international conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, 2004, pp. 224–233. ISBN: 1-58113-844-X. DOI: 10.1145/988672.988703. URL: http://doi.acm.org/10.1145/988672.988703 (cit. on pp. 25, 26).

[8] Eric Steven Raymond. *The Art of UNIX Programming*. pub-AW:adr: Addison-Wesley, 2004, pp. xxxii + 525. ISBN: 0-13-124085-4, 0-13-142901-9. URL: http://www.faqs.org/docs/artu/ (cit. on pp. 24, 32, 42, 61, 62, 67, 87).

[9] Audrey Tang. *Template::Extract. Use TT2 syntax to extract data from documents*. Oct. 16, 2007. URL: http://search.cpan.org/~audreyt/Template-Extract-0.41/lib/Template/Extract.pm (cit. on p. 70).

[10] Peter Thiemann. “WASH/CGI: Server-Side Web Scripting with Sessions and Typed, Compositional Forms”. In: *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*. PADL ’02. London, UK, UK: Springer-Verlag, 2002, pp. 192–208. ISBN: 3-540-43092-X. URL: http://dl.acm.org/citation.cfm?id=645772.667946 (cit. on p. 72).

[11] Chris Wanstrath. *Mustache(5) Mustache Manual*. 2009. URL: http://mustache.github.com/mustache.5.html (cit. on p. 78).