PROGRAMMING WEB APPLICATIONS DECLARATIVELY

A QUALITATIVE STUDY

A Thesis

by

PAWAN KUMAR SINGH

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Jaakko Järvi
Co-Chair of Committee,	Jeff Huang
Committee Member,	Alex Sprintson
Head of Department,	Dilma Da Silva

August 2016

Major Subject: Computer Science

Copyright 2016 Pawan Kumar Singh

ABSTRACT

In the declarative programming approach of property models, a dataflow constraint system manages the behavior of a user interface. The dataflow constraint system captures the user-interface logic as a set of variables and dependencies between those variables. This thesis builds on the prior work that realizes the property models approach as a concrete library for web development called HotDrink. This thesis evaluates the effectiveness of the declarative programming approach of property models, describes the experience of implementing a medium-size web application following the approach, and compares property models with existing web frameworks. A particular focus is on how programming with property models helps programmers to avoid defects related to asynchronous execution of responses to user events.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Jaakko Järvi for the continuous support, patience and immense knowledge. His guidance helped me throughout, from theoretical concepts to research approach. The weekly meetings and discussions have trained me a lot and helped me in becoming a better researcher. I am also thankful for his constructive feedback I received while writing this thesis.

Besides my advisor, I would like to thank the rest of my advisory committee: Dr. Alex Sprintson, and Dr. Jeff Huang, for their support and enthusiasm.

I would like to express my heartfelt thanks to my parents for their love and support. I would like to thank Taujee for his continuous support and motivation. To my siblings and friends: Pankaj Bhaiya, Menka Didi, Sweta Didi, Pushkar, Aarti, Anurag, Anurag, Anindya, Anirrudh, Prannay and others, thank you for supporting and helping me in difficult times.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

The support for developing asynchronous web applications has improved significantly over the past decade. User interaction on the Web has become richer and web applications have become more user friendly. One factor behind this development is that programmers are taking advantage of asynchronous functionality with Ajax [22]. In asynchronous web applications, it is easier to deliver changes to an application's presentation layer without the need of the browser to make server requests repeatedly. Today, numerous web frameworks (mostly JavaScript based) exist for rich user interface development, with easily programmable event handlers.

While asynchronous handling of user events can improve the user experience, it complicates programming. Even though web applications are to a large extent built using predefined software components, when the components are put together their composition becomes a complex software artifact in itself. These compositions are not reusable, and when code in individual components executes asynchronously, the interactions are difficult to program correctly. This difficulty is due to complex dependencies and a large amount of non-reusable code present in different event handlers of rich user interfaces [18]. In these scenarios, it is difficult to ensure that all data dependencies are fulfilled when new user events are triggered while past events are still being processed [9]. This eventually gives rise to asynchronous programming defects.

A possible solution to these challenges is the declarative UI programming approach called *property models* [18]. This approach aims for increasing reusability of user interface code. A property model defines the data dependencies between the different components of a user interface explicitly. When the value of a variable in

a particular component changes by a user event, a property model identifies which other components must be changed and how. Concretely, a property model responds to user events by resolving data dependencies and scheduling asynchronous computations for them [9, 18, 17, 11]. The *HotDrink* library is a concrete realization of property models in JavaScript and TypeScript [9].

Currently property models are at an experimental stage and not yet a properly validated idea. The approach seems promising but more experience with realistic application programming is needed. This thesis is one step in providing such experience. This thesis reports on a qualitative evaluation of the effectiveness and limitations of the declarative programming approach of property models for web application development. The evaluation was carried out with the help of the HotDrink library by building a medium-size realistic web application using HotDrink, together with other JavaScript libraries, and then by analyzing the effort and experiences of developing software with this approach.

In this thesis, we discerned some significant outcomes about the practice of developing user-interfaces with property models. The most important inference is that property models can be used effectively to develop UI logic of modern web applications built with JavaScript templating libraries. Another important result is that property models can be effectively use to build complex UI widgets that are not prone to asynchronous programming defects.

2. REVIEW OF TRADITIONAL WEB PROGRAMMING

2.1 Event Driven Programming

In early days of computing, programs used to run in batch mode. The programs read input files in batch operations, computed operations on the data and finally emitted the results as output. The arrival of time-sharing and text-based terminal systems made interactivity possible: programs asked for user input and provided their responses based on this input. Later, graphical displays and pointing devices transformed user interaction to a richer experience. This change was the result of the event-driven programming model. In the event-driven programming model, events in the form of mouse clicks, keystrokes, etc. decide the flow of a program.

The majority of all web applications are designed and implemented using the event-driven programming model. When an event is triggered, the browser invokes an associated event handling function to respond to the triggered event. Events can occur in any order, there is no predetermined order for them. In the usual case, the user's actions determine, or at least affect, the sequence of events. The system can also trigger events.

An event can be triggered because of three reasons:

1. User interaction with a GUI.

2. Web application's periodic processes or response to user interaction.

3. Client operating system notification.

User interaction is the most common way due to which events are triggered in a web application session [8]. The GUI of a typical web application is composed of a set of forms or HTML controls. Each of these forms or controls recognizes a set of

events specific to its functionality. For example, a HTML element can recognize an event that triggers whenever a user clicks that particular HTML element, or an event that triggers when the value of the HTML element changes. Similarly, an event can be triggered when a HTML page finishes loading or when a user presses a keyboard key.

The web application's objects can trigger events too. These events are primarily periodic events that are executed to update the GUI with the latest information from the server.

Lastly, the client operating system can trigger its own events. These events are associated with permission access for hardware control or errors in executing the GUI code block.

2.2 Event Handling in JavaScript

Event handling in JavaScript provides programmers the core technique of controlling the GUI content, generate responses to user interaction, and maintain the application flow. Over the past decade, event handling code has become central to web programming [21].

Modern browsers support dynamic page rendering. With this feature, the browser switches intermittently between processing client-code, rendering the presentation-layer, and waiting for new events to trigger. This perpetual loop allows a web page to be partially rendered even before the browser has completed fetching all the resources associated with the page. Dynamic page rendering leverages JavaScript to execute event driven actions and control web application behavior.

There are two ways in which event handlers can be implemented in JavaScript. First approach is as HTML attributes. In this approach, JavaScript code is specified as a value of a HTML attribute. In Listing 2.1, the `onclick` attribute of the

HTML `input` tag has been assigned some JavaScript code. This code creates an alert popup. Multiline JavaScript code can be added by separating statements with semicolons.

Listing 2.1: Use of HTML attributes

```
<input type="button" value="Submit" onclick="alert('Hello');"/>
```

The second approach involves assigning JavaScript code to a DOM Object's property. Each HTML document is a tree of HTML elements. This HTML tree has a corresponding representation as a JavaScript object, the DOM tree. The attributes of HTML elements are properties of objects in the DOM tree. In the above example, `onclick` is an attribute of the `input` element, so by first finding the correct object from the DOM tree and then binding a JavaScript function to the `onclick` property of the object we can register an event handler for the `onclick` event of the `input` element, as shown by Listings 2.2 and 2.3.

Listing 2.2: Specifying element names in HTML.

```
<form name="frmMain">
    <input type="button" name="btnSubmit" value="Submit"/>
</form>
```

Listing 2.3: Using HTML elements to access properties of DOM object

```
document.frmMain.btnSubmit.onclick = function(){ alert('Hello');};
```

2.3 Asynchronous Programming

In early web applications, the browser made a request for a new presentation of the current page for every user interaction that required data from the server. The

execution of JavaScript code on the client was blocked until the request had been fulfilled by the server. This form of interaction is known as synchronous communication. The programming paradigm that supports developing applications with such communication is called synchronous programming.

The major limitation of synchronous web applications is the large time interval between each GUI update, or alternatively frequent updates that each freeze the application for a moment. Even if a synchronous web application has been programmed in such a way that it frequently refreshes itself with new information from the server, individual refresh operations cause delays. For applications that rely on data that is continuously updating, for example stock trading applications, synchronous communication does not provide an ideal user experience.

With the advent of Web 2.0 [25], asynchronous communication came into existence. In this form of communication, multiple requests between the client and server can be submitted simultaneously, and the client continues to execute its JavaScript code while the requests are being handled. This capability enabled web programmers to develop web applications that can deliver continuously updated information to the user. The programming paradigm of developing applications performing communication of such nature is referred to as asynchronous programming.

By using the request/response mechanism differently, one can achieve different communication and update strategies. Three common models for asynchronous communication are the *polling*, *long-polling*, and *push* models [27].

The polling model is the simplest communication mechanism. The client keeps the GUI up-to-date by regularly sending a request to the server to see if there are any new data. There is an inherent trade-off in the basic polling mode: good user experience requires frequent polling, but frequent polling consumes both the client's and server's resources. In the long-polling model the browser requests the server for

new information and the server only responds after there has been an update. Immediately after the receipt of the server response, the browser sends another request, so that there is constantly a request open. This immediate response by the browser and waiting until there is a change by the server differentiate long-polling from the simple polling model [27].

Like the other two models, the push model also initiates communication by the client sending a request to the server. The response received from the server is, however, kept open. This allows the client to keep an active connection with the server. The server then sends a sub-message over the open connection every time there is an update. This model maintains an open communication link between the client and server at all times [27].

2.4 Ajax and jQuery

Ajax (Asynchronous JavaScript and XML) is the practical realization of asynchronous communication in JavaScript. It enables the browser to update a web page without the need to reload it. The Ajax protocol is complex and constructing data for Ajax requests is tedious. The popular jQuery [32] library hides some of this complexity and is commonly used for implementing client-server communication in JavaScript programs.

Listing 2.4 shows an example of invoking an Ajax call using jQuery. In this example, we are calling the Flickr web service [35] using jQuery's `ajax` function. The `ajax` function requires two important parameters: the web service URL and the result's data-type. In Listing 2.4, these parameters are represented by the variables `url` and `datatype`, respectively. The variable `url` has been assigned the URL of the Flickr web-service. The variable `dataType` has been assigned the `json` format. Based on the success of the asynchronous call, the `ajax` function calls one of the two

functions, namely `success` and `error`. The last line of Listing 2.4 add a text-box to the HTML Document.

Listing 2.4: Ajax call using jQuery.

```
$.ajax({
url: "http://www.flickr.com/services/feeds/photos_public.gne?
   tags=cars",
dataType: json,
success: function (data) {
     $("#results").val(data); },
error: function(error) {
    console.log("Error: " + error); }
});

$("#body").appendTo( "<input type="text" id="from" name="from"
   />" );
```

The strength of asynchronous programming comes from the fact that asynchronous computations do not interfere rendering of the web page. For example, the task of adding a DOM element to the presentation layer on the last line of Listing 2.4 does not depend on the completion of the `ajax` function. Independent asynchronous computations provide a performance improvement for the web application. On the other hand, asynchronous computations managed by event handlers seem to often exhibit defects when complex tasks involving user inputs are performed. In Chapter 5, we discuss in detail some prevalent defects in commercial websites. In Chapter 6, we discuss the avoidance of asynchronous programming defects using the approach of property models.

3. BACKGROUND

The proposed work is based on prior work done in the field of declarative GUI programming at Parasol laboratory of Texas A&M University. This prior work builds on hierarchical multi-way dataflow constraint systems [11] and introduces property models as an abstraction for the GUI state [11, 17]. To place the proposed work in a proper context, we discuss constraint systems, property models, and the HotDrink library.

3.1 Constraint Systems

A *constraint* system represents a system of variables and constraints among those variables. A constraint establishes a relationship between the variables. When the constraint is true, it is considered to be satisfied. A *constraint system* ensures that all constraints are satisfied. Whenever a constraint becomes unsatisfied, the constraint system re-enforces it by updating variables associated with the constraint. This update task (or computation) partitions the variables of the constraint as input and output variables. This partitioning represents a direction of computation; not all directions are feasible in a constraint [11].

The feasible computations for each constraint are expressed by a set of methods known as *constraint satisfaction methods* (CSM). The code associated with a CSM is considered a black box, and is provided by the programmer [18]. A multi-way dataflow constraint has one or more methods that can enforce it. A multi-way dataflow constraint system ensures that a set of multi-way dataflow constraints are satisfied [15]. This is done by executing one method from each constraint. The methods are executed in such an order that once a variable has been used by a method as input, no other method updates the value of that variable [17]. Any

execution sequence that satisfies this property is called a *valid execution order* [9].

Solving a constraint system requires two steps. The first step is choosing the methods (one from each constraint) for execution in an order such that it constitutes a valid execution order. This selection of methods is called the *plan* [9, 16]. The second step is executing the methods in the chosen valid execution order.

If a multi-way dataflow constraint systems has multiple plans it is called *under-constrained*. For such systems, plans can be ordered based on a prioritization of variables, and the best plan can be selected as a solution. This is done in the following three steps.

In the first step, we add a *stay constraint* for each variable in the constraint system. A stay constraint consists of a single variable and one method that outputs to the variable [16, 9]. As stay constraints are added, the system can become over-constrained. In an *over-constrained* system, no execution order can enforce all constraints.

In the second step, we prioritize the stay constraints to order partial solutions of the system. The stay constraint whose variable has been updated most recently is given the highest priority. The priorities build a hierarchy among the constraints [15]. This hierarchy corresponds to the order in which variables have been updated.

In the third step, the hierarchy of constraints is utilized to select the plan for the highest priority constraints. Each plan represents a sequence of constraints it enforces in a priority order from highest to lowest. The constraint system solver selects the plan which is greatest in the lexicographical order [9].

In the context of property models, we define two editing operations for constraint systems. These editing operations define how the View modifies the constraint system. A *touch* operation promotes a constraint to the highest priority and a *set* operation assigns a new value to a variable and performs a touch operation on the

stay constraint of the variable. A sequence of one or more touch and set operations represents an *edit* of the system. The constraint system is solved whenever an edit takes place [9].

3.2 Property Models

Traditionally GUI programming has been centered around events. The programmer develops the business logic of an application using callback subroutines that are registered to relevant events [9]. The difficulties in developing rich user interfaces using this approach have long been identified [18] and alternative solutions proposed. Property models are one of the solutions.

A property model is an abstraction aimed at reducing the complexity of GUI programming [18]. Property models derive GUI behavior from an explicit representation of the data dependencies between different components of the GUI. Traditionally all GUI behavior is managed using event handling code.

A property model consists of a constraint system and a declarative language for describing variables and constraints among them [18]. In terms of functionality, a property model resembles a spreadsheet. Setting the value of one variable leads to recalculating dependent values [18]. When bound to a user interface of an application, a property model captures the logic associated with the behavior of the user interface.

Architecturally, a property model cleanly fits the role of the View-Model in the Model View View-Model [20] (MVVM) design pattern. A View-Model helps in consolidating the logic of a user interface's behavior, independent of the View of the application. A crucial role is played by data bindings in achieving this ability. The data bindings provision the mapping of constraint system's variables to the corresponding components in the View. These data bindings help to translate changes in the View to the property model and vice versa.

3.3 HotDrink

HotDrink is the implementation of property models. It is written in TypeScript, a typed superset of JavaScript [15]. TypeScript can be compiled to plain JavaScript. HotDrink as a JavaScript library can be used to develop the View-Model of a web application, while at the same time other libraries can be used to implement the View. HotDrink is thus non-intrusive.

The View-Model in HotDrink contains all the logic governing the GUI behavior and state while the View is devoid of any such logic (usually it consists of just HTML declarations). This helps in developing complex GUI behaviors without the use of event handlers. The resulting code is clear, concise, and reusable [11].

The heart of the HotDrink library lies in the constraint system. In HotDrink, programmers are required to explicitly specify each method associated with a constraint. A View-Model in HotDrink is said to be *updated* whenever the constraint system is solved. An *update* can be triggered by adding a variable or a constraint, changing a variable's value, or by an explicit call [11].

HotDrink supports incremental development of a constraint system [11]. It means that new variables and constraints can be added to the constraint system that is already bound with the View. This allows the View-Model to adapt along with a dynamic web user interface [11].

Bindings play an important part in the HotDrink library. They manage the exchange of data between the View and View-Model. Bindings listen for a change in the elements of the View and write the changed value to the corresponding variable in the constraint system. They also listen for changes in the variables of the constraint system and update the corresponding elements in the view [11].

User interaction with the View is captured by user events [11]. HotDrink responds

to user events by solving the constraint system and producing a new GUI state, which is translated back to the View by data bindings [9].

Apart from the traditional HTML widgets, the HotDrink library also supports defining bindings on custom GUI elements (non-native HTML elements built using third party JavaScript libraries). This ability helps in extending HotDrink to support Views developed by other JavaScript libraries.

Figure 3.1, presents an example of a web application developed using HotDrink. We have named this web application as Carpet Cleaning Estimator. As suggested by its name, this web application estimates the cost of carpet cleaning for an apartment. The GUI of this web application consists of a text box, two checkboxes, and multiple labels. The user needs to input the number of bedrooms in the text box. The first checkbox needs to be checked if the staircase needs to be cleaned. The second checkbox needs to be checked if scotch guard treatment is required. The estimated total value of carpet cleaning is displayed below these inputs.

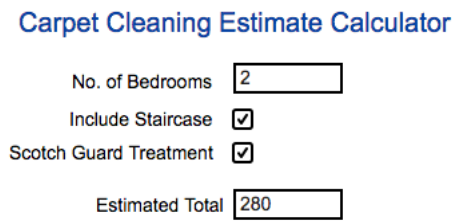

Figure 3.1: Carpet Cleaning Estimator developed using HotDrink.

Listing 3.1 describes the View of Carpet Cleaning Estimator. The View has been built using HTML tags. The HTML tags in Listing 3.1 corresponds to labels and text-boxes present in the View. For the sake of simplicity, we have presented the HTML tags without the styling component.

Listing 3.1: View for Carpet Cleaning Estimator.

```
<label>Carpet Cleaning Estimate Calculator</label>
<label for="editRooms">No. of Bedrooms</label>
<input id="editRooms" type="text"/>
<label for="editStairs">Include Staircase</label>
<input id="editStairs" type="checkbox"/>
<label for="editSGT">Scotch Guard Treatment</label>
<input id="editSGT" type="checkbox"/>
<label for="esTotal">Estimated Total</label>
<span id="esTtotal"><span/>
```

Listing 3.2: View-Model for Carpet Cleaning Estimator.

```
 var component = new hd.ComponentBuilder()
  .variable('bedrooms',0)
  .variable('staircase',false)
  .variable('sgt',false)
  .variable('estimate')
  .constraint('bedrooms, staircase, sgt, estimate' )
  .method('bedrooms, staircase, sgt -> estimate',
    function( bedrooms, staircase, sgt ) {
           if (bedrooms <1) {return 0;}
           var total = bedrooms*90;
           if (staircase) { total += 50;}
           if (staircase) { total += bedrooms*25;}
           return total; })
  .component();
var pm = new hd.PropertyModel();
pm.addComponent(component);
```

Listing 3.2 describes the View-Model of Carpet Cleaning Estimator. In the View-Model, we are creating the property model object using a *component*. The *component* defines the variables and the constraints between those variables. In other words, the *component* defines a constraint system. The Carpet Cleaning Estimator has just one constraint and that constraint has just one method. This method takes three parameters representing the number of bedrooms, inclusion of staircase cleaning, and scotch guard treatment, and computes the estimated total of carpet cleaning. These parameters are bound to the three inputs from the View. The last two lines of Listing 3.2 uses the *component* to create an instance of property model.

Listing 3.3: Bindings for Carpet Cleaning Estimator.

```
window.addEventListener( 'load', function() {
 hd.bind( {view: new hd.Edit(document.getElementById('editRooms')),
           model: component.bedrooms });
 hd.bind( {view: new hd.Edit(document.getElementById('editStairs')),
           model: component.staircase });
 hd.bind( {view: new hd.Edit(document.getElementById('editSGT')),
          model: component.sgt });
 hd.bind( {view: new hd.Text( document.getElementById('esTotal')),
           model: component.estimate});
});
```

Listing 3.3 describes the data bindings of the View. In this code, we are using the `addEventListener` method to attach a property model to elements in the GUI. In this code, we are referencing each element of the View by its element id. We then bind each element with the variable that represents it in the View-Model. In this code, we are creating data bindings for DOM elements representing the number of rooms, staircase inclusion, scotch guard treatment, and estimated total. These

DOM elements are bound with the respective variables present in the property model. Whenever the input values are edited by the user, the property model evaluates a new value for the estimated cost, and data bindings reflect it immediately on the View.

4. METHODOLOGY

Our research methodology consisted of three phases. The first phase assessed how prevalent asynchronous programming defects are in modern web applications. We studied causes and severity of these defects. The second phase involved the development of a medium-size realistic web application. The GUI logic of this web application was developed using the declarative programming approach of property models. The web application was analyzed with different use-cases. These use-cases showcase and test the behavior of its user interface. The use-cases involved manipulating large data-sets over asynchronous tasks to assess the impact of this approach on real web applications. In the final phase, we assessed and evaluated the development experience.

4.1 Identification

To understand the severity of the problems in programming asynchronous GUIs, we investigated web applications from different domains. The emphasis was on web applications from the travel industry because of their feature-rich user interfaces and massive user bases—in the United States, the minimum number of daily visits to the 30 most popular travel booking websites have been estimated to be between 300,000 and 1,000,000 [1]. We also investigated applications from education, healthcare, and the online shopping industry. Based on our investigations, we concluded that defects are so prevalent that these are not mere simple inconsistencies but indicate a bigger methodological problem. The traditional design and implementation of web application that centers around event handlers is responsible for the high occurrence of defects [9].

4.2 Prototyping

To examine the challenges and benefits of our declarative programming approach in developing web applications in practice, we chose to build a travel booking web application. We developed this application with a popular JavaScript web framework called AngularJS [12] combined with the HotDrink library. The aim of this combination was to assess if the usage of HotDrink limits or benefits the development of modern web applications. This web application was developed using the MVVM [20] design pattern.

As its name suggests, the Model-View-ViewModel pattern separates the application into three parts. The first part, called View, is responsible for delivering presentation changes to the user and capturing user actions. The View in our application was developed using HTML and CSS.

The AngularJS library comes with rapid application development features. One such feature is templating, which allows the library to control HTML on the browser's DOM level, rather than via string manipulation. Another strength of the AngularJS library is the *directives* feature. This feature allows the library to extend existing HTML widgets or create new ones by encapsulating custom behaviors on top of existing ones.

The second part, the View-Model, is an abstraction of the View component. It maintains the data of the View and provides the logic describing the View's behavior. It consists of GUI processing elements that are JavaScript objects from HotDrink and AngularJS libraries.

In our solution, we are controlling the View with AngularJS objects, which in turn are controlled by HotDrink objects. The responsibility of maintaining the data dependencies lies with HotDrink while AngularJS captures user responses and man-

ages the View. AngularJS notifies HotDrink of changes in the View due to user events. HotDrink assesses the changes and computes new dependent values. It then notifies AngularJS to update the View elements with those new values.

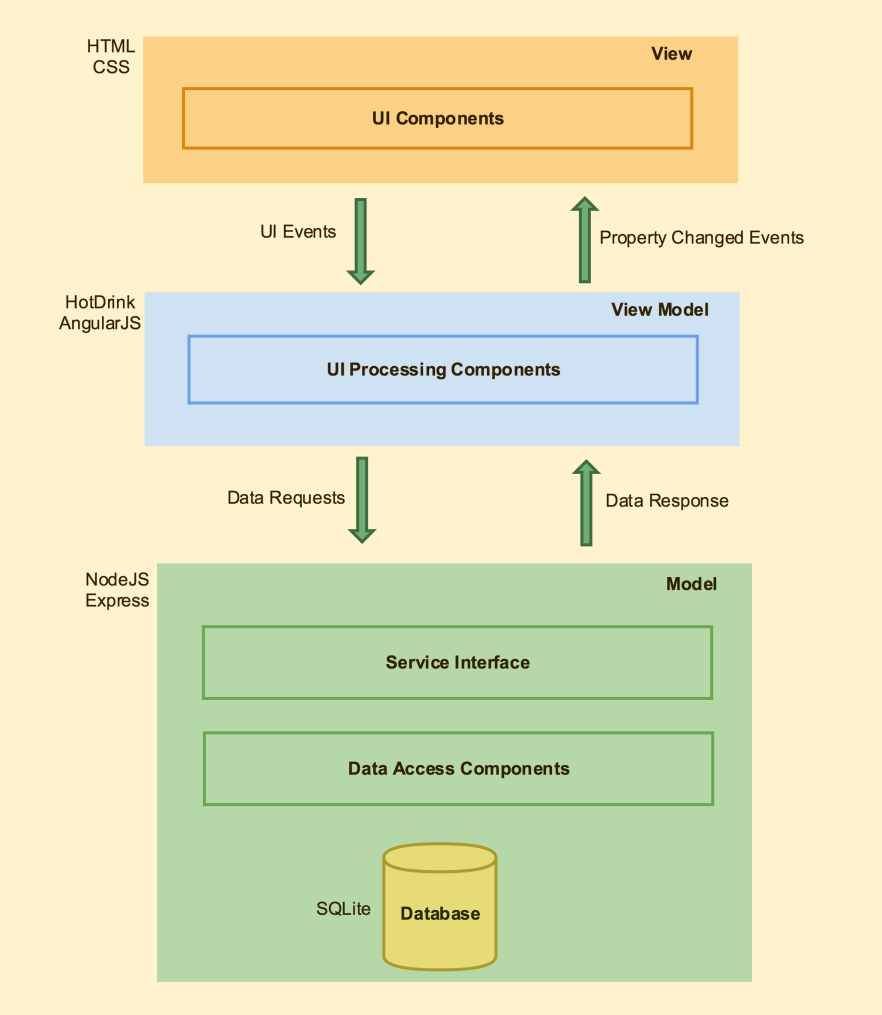

Figure 4.1: The MVVM architectural pattern in our prototype web application.

The third and final part of the MVVM pattern is the Model. It is responsible for the data and logic independent of the GUI. In our solution, the Model was developed using a server-side runtime environment called NodeJS [6]. This runtime environment

uses a light-weight database called SQLite [14].

Figure 4.1 shows the diagrammatic representation of the MVVM architectural pattern for our web application. In this diagram, we can clearly identify the software stack for each component of the MVVM pattern.

4.3 Evaluation

In the evaluation of constructing the travel booking web application, we focused on assessing the effort and experience from the developer's perspective. We focused on three aspects in particular. First, we compared the implementations of popular web components (widgets) such as auto-complete text-boxes, sliders, and multi-tabs, developed with and without HotDrink. Second, we observed challenges of using HotDrink with existing JavaScript frameworks popular for rapid application development. Finally, we compared HotDrink with other modern solutions that are being used to tackle asynchronous programming defects.

5. DEFECTS IN WEB APPLICATIONS

In order to better understand the complexities and defects caused by combining asynchronicity and event-handling, and how they manifest in modern web programming, we analyzed web applications from multiple industries. These industries include online travel booking, education, and health-care. As a result of this analysis, we present examples that show how asynchronous execution of programming logic in web applications produce incorrect results [9].

5.1 Travel Booking

Online Travel Booking came into existence during mid-1990s and is ubiquitous nowadays. Online Travel Booking applications vary from airlines' own flight reservation systems to travel aggregators and vacation package planners. These applications make it possible for travellers to plan and book trips at any time. In the spring of 2015, around 59.8 million online travel bookings were made in United States alone [31]. In terms of sales, Expedia Inc. was the largest travel agent in the world, followed by Priceline Group and Orbitz.

In our sampling, we collected examples from 35 most popular travel booking web applications. The measure of popularity was in terms of the daily unique visitor count [1]. For each of these web applications, we evaluated their workflow for booking a reservation for a simple itinerary. Our focus was on the HTML widgets which we used in each step of the booking process. We will now discuss these widgets in detail.

5.1.1 Auto-complete Text-Box

An auto-complete text-box is one of the most common GUI elements present in travel booking applications. See Figure 5.1 for an example. This GUI element assists

the user with suggestions to select the right string for input. The text entered by the user is utilized for input as well as for generating the suggestions that matches the input. The suggestions are generated using an asynchronous web-service call in which the user-entered string is itself used as the input parameter. The suggestions are displayed below the text box as a menu from which the user can select a matching suggestion. If an item is selected from the suggestion menu, the content of the menu decides the final input. If no matching suggestion is generated, the input string itself becomes the final input.

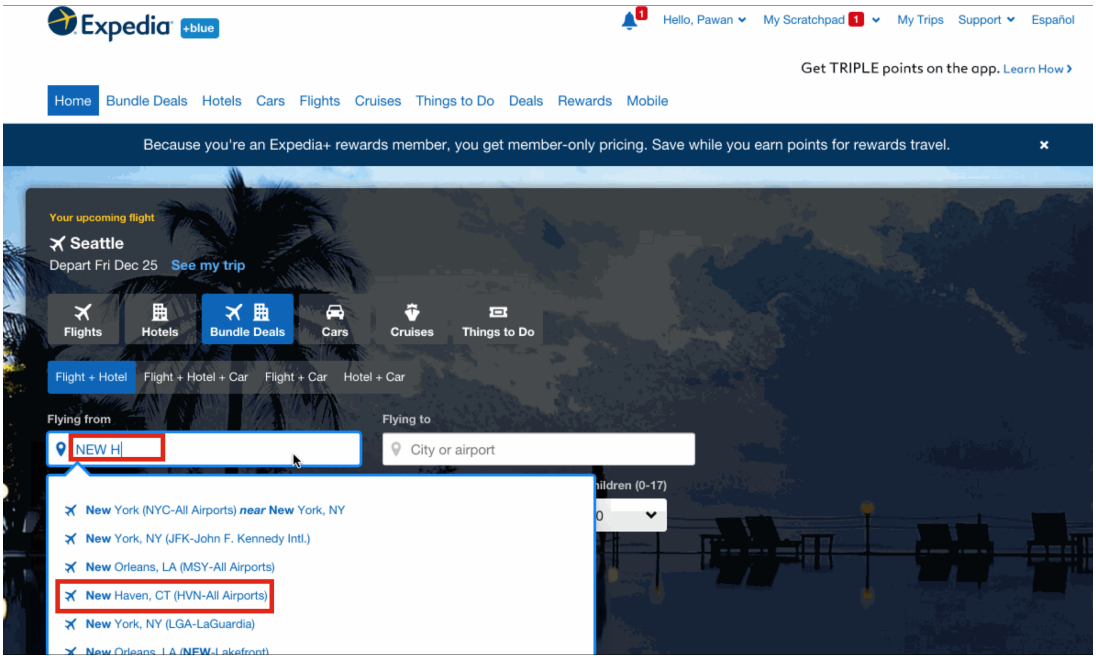

Figure 5.1: Input suggestions of Auto-complete [7].

A common defect in auto-complete text-boxes appears, for example, in the following scenario. The user first types the input string “New ”, and the text-box gives a suggestion list of cities whose name starts with “New ”. In this list “New York”

usually comes on the top (the list is often not in alphabetical order). If the user modifies the keyword to "New H", the list refreshes itself, and "New Haven" comes as the top suggestion. If, however, instead of waiting for the list to refresh, the user hits the `enter` key moderately quickly after entering the "H" character, the top result from the previous suggestion, i.e., New York, gets selected as the resulting value of the auto-complete text-box. As another manifestation of this defect, it is entirely possible that the user types in P-A-R-I-S and hits `enter`, yet gets a search for flights from Paragould, Arkansas.

The event handling mechanism of an auto-complete text-box revolves around four pieces of information: the *input-string*, *suggestion-list*, *current-selection*, and *final-value*. The first event handler captures keystrokes from the user as *input-string*. It then requests a list of suggestions from the server based on *input-string*. We term this list as *suggestion-list*. A second event handler gets triggered by arrow keys and it selects an item from the *suggestion-list*. We term the currently selected item from the *suggestion-list* as *current-selection*. A third event handler gets triggered by the `enter` key and sets the *current-selection* as the *final-value*. The *final-value* serves as the output of auto-complete text-box. All these event handlers are *keypress* event handlers that are triggered when the user presses keyboard keys.

The reason that the user experiences inconsistencies with auto-complete text-boxes is that newer event handlers are processed before older ones have completed. Concretely, the keypress event-handler yields instead of blocking after it has sent the server a request for new content for the *suggestion-list*, and by this allows the handler for the `enter` keypress event to proceed with stale information [9]. In Section 6.3.2 we discuss an implementation of an auto-complete text-box using constraints, and explain in details what the dependencies between different pieces of data are.

Out of 35 commercial travel websites we examined, 33 exhibited the above-

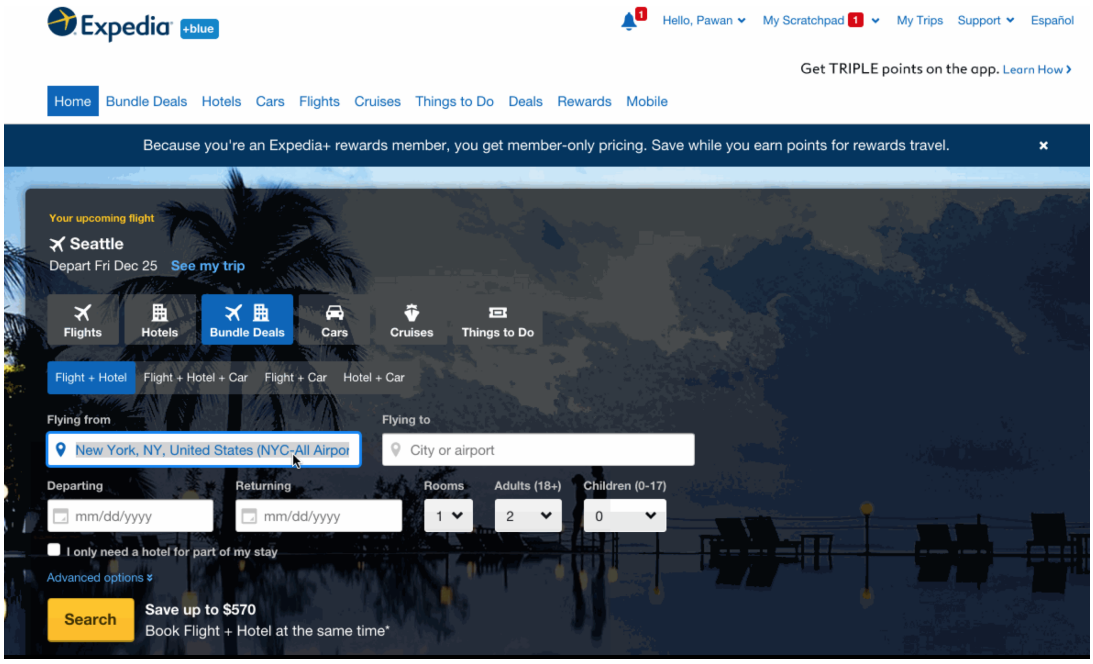

Figure 5.2: New York gets selected when `enter` key is hit immediately after typing 'New H'. This is an incorrect selection as we can observe in Figure 5.1 that shows that the correct matching entry is New Haven not New York [7].

described inconsistency. Figures 5.1 and 5.2 show an example of the inconsistency manifesting in the popular travel website `expedia.com`.

The two websites that do not have this problem, `kayak.com` and `makemytrip.com`, avoid it by avoiding asynchronicity altogether: they keep the auto-complete text-box from accepting input until the suggestion list has been fully updated, which is not particularly user-friendly.

5.1.2 Multi-Tab Search Results

By *multi-tab search results* we mean the presentation of search results from different queries in separate tabs. This functionality is similar to opening multiple tabs in any modern web browser. Tabbed content helps in grouping related information in the same view. This feature makes it possible to switch to another tab, that

has previously processed information, while information in the current tab is being processed.

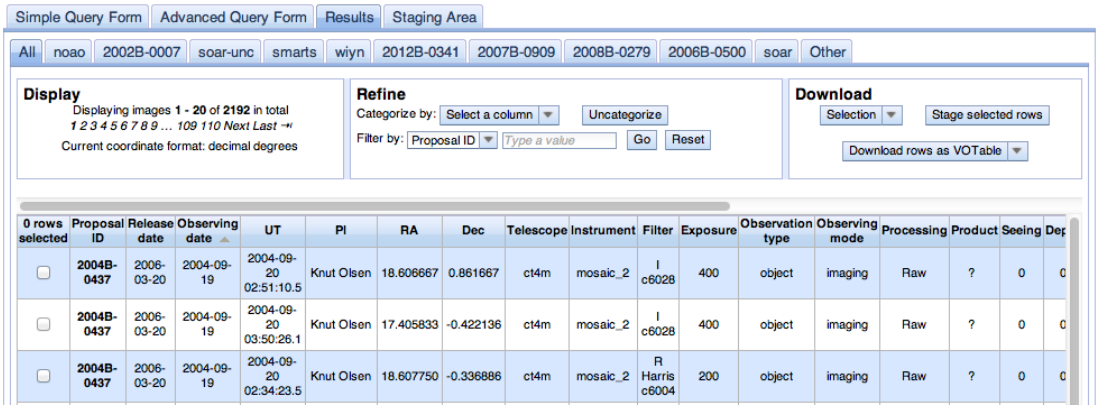

Figure 5.3: Example of Multi-tab search results in NOAO Science Archive website [3].

To give a concrete example of an application that supports *multi-tab search results*, we will take an example from `archive.noao.edu`. Figures 5.3 shows a screenshot of search functionality present on this website. In this figure, we can observe that different search results have been presented in individual tabs. Unlike commercial travel booking applications, this website uses the server-side scripting languages PHP and Pearl for building GUI. Due to extensive server-side scripting, this website is slow in responding to user interaction and not particularly user friendly.

All the commercial travel booking websites we examined provide an elaborate search functionality but do not present *multi-tab search results*. Most commercial booking websites use the same search result pane and every new search completely replaces the old results. A handful of commercial booking websites open new windows for individual search results.

We suspect the biggest reason for why commercial travel booking sites do not support *multi-tab search results* is the complexity of implementing such a design using JavaScript event handlers. We obtained some confirmation to this conjecture of ours from Expedia's developers [30]. As a reason for why multi-tab search results are not supported in Expedia's user interface, the UI developers there cited the complexity of client-side code when trying to combine multi-tab search results with *search filters*, another feature in their GUI. A search filter is a widget that helps the user in selecting a subset of information from a result set. A typical search filter widget captures user input from check-boxes, radio buttons, or sliders. It then uses this user input to restrict the result set.

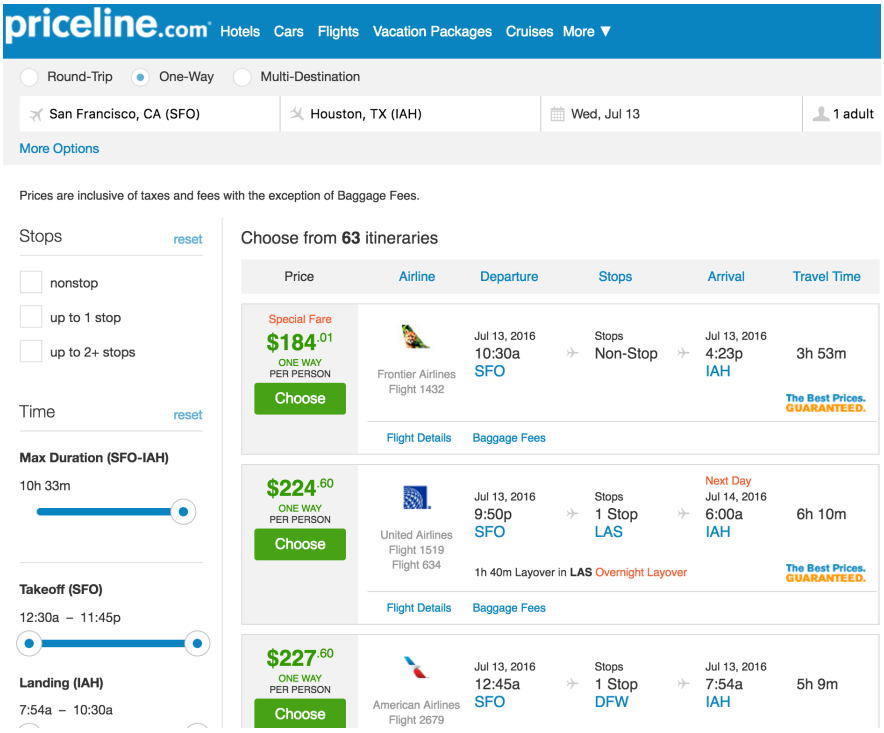

Figure 5.4: A screenshot from the Priceline website. Search filters for the number of stops and the duration of flight can be seen on the left side [28].

Figure 5.4 shows the screen-shot of search result filter from `priceline.com`. The left side of this image shows search filters for the number of stops and the duration of flight. The right side of this image displays the list of flights matching the search criteria.

In our own experiment of implementing a flight reservation client using HotDrink, we included both tabbed search results and search filters. In preparation for this, we briefly outline a search filter implementation using explicit event handlers and discuss how the overall software artifact becomes complex with multiple tabs.

Listing 5.1: HTML code snippet demonstrating search filters.

```
<input type="checkbox" ng-model="nonStops" value="false" id="non-
    stop" />
<article data-ng-repeat="flight in (flights | filter:Filter.nonStops
    )" class="result"/>
```

Listing 5.2: JavaScript code snippet demonstrating search filters.

```
app.filter('nonStops', function() {
  return function(items, search) {
    if (!search) { return items; }
    var stopType = search.stopType;
    if (!stopType || '' === stopType) {
      return items;  }
    return items.filter(function(element, index, array)
    { return element.stopType.nonStop === true;  });
  };
});
```

Listings 5.1 and 5.2 show code snippets from the GUI referred to in Figure 5.4. Listings 5.1 shows two HTML elements. The first HTML element is a checkbox for

selecting non-stop flights. The second HTML element is for the rendering of search results. Listings 5.2 represents the event handling code written using the AngularJS library to filter search results. The AngularJS code iterates the search result, selects the flight entries that are non-stop, and returns the selected set of entries.

Listings 5.1 and 5.2 together represent code just for a single search filter criterion. For each new criterion, a complete new event handler, a HTML tag, and data bindings need to be added. The new event handler code will be very similar to Listing 5.2. The new HTML tag will be similar to `input` element present in Listing 5.1. To add the new data binding, the `article` element in Listing 5.1 will be modified. A new filter will be piped after the existing one.

The complete search result page of `priceline.com` contains 22 criteria, which means 22 event handlers manipulating the HTML element containing results. The complete search-filter mechanism built with 22 event handlers becomes a complex software artifact. If multiple-tabs comes into the picture, this scenario gets even more complex. Every time a new tab is added, 22 event handlers are introduced to the GUI.

A simplified approach can be achieved if a single set of 22 event handlers can filter search results for each tab. In this approach, every time a tab is selected, the set of search filters needs to perform two significant operations. First, save filter criteria values for the previously selected tab. Second, recall the most recently modified values for the newly selected tab. Saving and recalling the most recently modified filter values is a complex task for event handlers, unless there is a persistence mechanism. Transforming filter criteria values into persistent data will also likely affect the performance of the GUI.

Out of the 35 surveyed web applications, none has the feature of *multi-tab search results*. In our travel booking web application, we have implemented it using the

declarative programming approach of property models.

5.1.3 Sliders

Search results filters often feature one or two slider controls in addition to check-boxes and radio-buttons. The sliders help in setting a range of values for the search filters. Figures 5.5 shows the screen-shot of a slider control to filter flight search results on `kayak.com`.

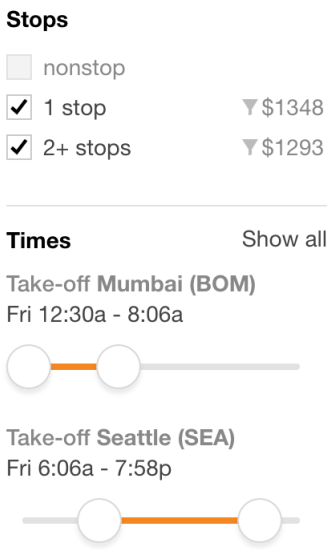

Figure 5.5: Slider controls to filter results on the Kayak website.

These slider controls can be moved with a mouse or arrow keys. When we experimented with sliders and moved them with arrow keys in a swift manner, we observed that they do not behave consistently. In such scenarios, the swift inputs from arrow keys were reflected on the slider but the result set did not reflect the correct filtered values.

The reason is as follows: upon pressing arrow keys, the associated event handler gets triggered to retrieve a result set with new criteria. When these input requests

are swift and successive in nature, few of the inputs get lost. It is an asynchronous programming defect similar to what we discussed in Section 5.1.1. The event handler sends an asynchronous request to retrieve data with new criteria. The keypress event handler, instead of blocking and waiting for new result set, goes ahead and modifies the slider range immediately.

A majority of the booking applications that we examined have disabled input from arrow keys. Out of 35 web application that we examined, 27 have sliders in their GUI. Out of these 27 web application, 11 allow arrow keys to control sliders and they all behave inconsistently with swift successive inputs.

5.2 Education

In the education industry, grading applications are becoming very popular. Majority of these systems are web-based and their implementations rely on JavaScript frameworks to be able to provide rich GUIs to their users. We surveyed six grading applications: Blackboard, Gradescope, Edmodo, Engrade, Vocareum, and zyBooks. These applications were selected because of their wide adoption in educational institutions throughout the world. For example, the Blackboard application is being used in approximately 17,000 schools across 100 countries [34]. Out of the six applications, we observed serious defects in three of them. Our analysis concentrated on the work-flow and mechanisms for grade entry and automatic grading of answers entered by students.

5.2.1 Blackboard

Blackboard is an online learning management environment and course management system. It is used for publishing course content and for managing assignments and grades. We examined the grading module of Blackboard. One of its main features is an online spreadsheet for recording student grades. Each column represents

an assignment, exam, quiz or some other graded task. Each row represents a student. To make grade entry smooth, the spreadsheet supports keyboard commands for navigation. Hitting the `enter` key after entering a grade will move the cursor to one row below the current row, in the same column.

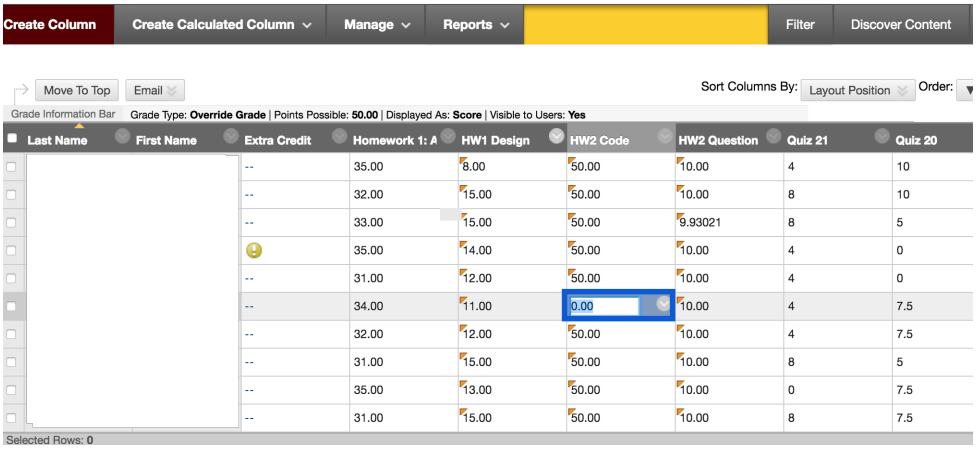

Figure 5.6: Grade recorded as 0 instead of 50 in the Blackboard application when the user types inputs swiftly [4].

In our experiment, we entered grades for students in the order they appear in the spreadsheet. Thus, we entered a numeric grade, hit `enter`, entered another numeric grade, hit `enter`, and so on. When we started doing this process quickly, we found on several occasions that keystroke entries were missed. In Figure 5.6, we can observe this behavior. It is a screen-shot of the grading spreadsheet from the Blackboard application. In our experiment, we entered the grade 50 for each student. When we started doing this process at a faster pace, the key '5' was missed and 0 was recorded as the grade.

5.2.2 Gradescope

Gradescope is an application dedicated to grading. Instead of the simple numeric entry of grades, Gradescope provides a customizable set of grading rubrics to grade assignments. This methodology gives clarity to students in understanding their mistakes and ensures that the grader does not miss any sections. In a typical grading process of Gradescope, the grader has to examine the submitted assignment, click the appropriate rubric items, and finally click next for moving to the next ungraded assignment. Gradescope supports this same functionality with keyboard shortcuts as well.

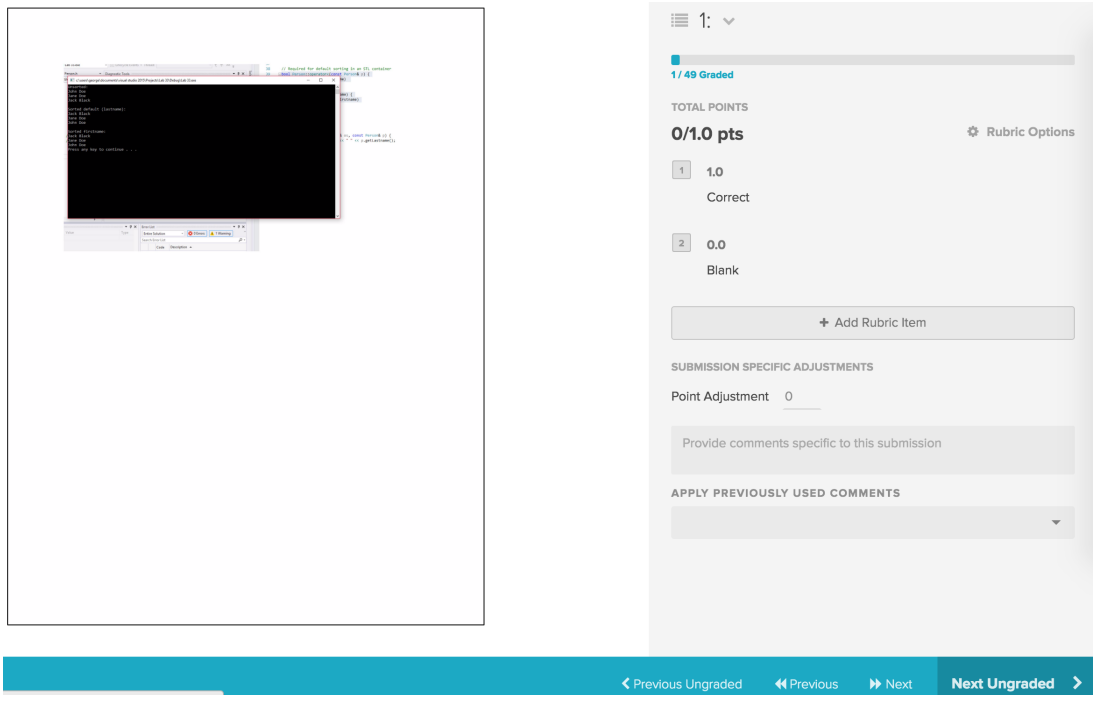

Figure 5.7: Rubric entry remained ungraded due to quick data entry in the Gradescope application [13].

We used keyboard shortcuts to examine the grading procedure. Similar to the

clicking procedure, we used one keyboard shortcut to select the appropriate rubric items, followed by another keyboard shortcut to view the next submission. When we started doing this procedure in a swift manner, we encountered inconsistent behavior. With a moderately swift input speed, the GUI can switch to the next assignment entry before selecting the rubric items. This inconsistent behavior resulted into ungraded or incompletely graded entries. Figure 5.7 shows a screenshot of Gradescope application. In this screenshot, we can observe that the rubric entries (on the right) are not recorded.

5.2.3 zyBooks

The zyBooks application is an interactive textbook. It uses rich media to enhance the text book experience. We examined coding assignments in zyBooks. There are two types of coding assignments in zyBooks. The first type does not provide any code or hints. The students have to write the complete code in the provided area. The second type are in the form of a *cloze* test. In a cloze test, certain portions of code are removed and replaced with text-boxes. The students are required to fill these text boxes with their answer code.

For the first type of assignments, we observed that the grading process was simple and consistent. For the second type of assignments, we observed issues on multiple occasions. The grading mechanism for second type of assignment is automatic and runs simultaneously as the focus on the text boxes progresses forward. If the student modifies a text box that was previously edited, the auto-grader mechanism does not detect it at that time. The detection happens when the focus restarts from first line of code and gradually reaches the target text box.

The GUI for the second type of coding assignments in zyBooks is a case of inconsistent dependencies. The grading evaluation is taking place with event handlers

sequentially. These event handlers trigger on the focus event of text boxes. Unless all the event handlers are triggered again in the sequence, incorrect evaluation persists.

These defects reiterate our claim that the problems of inconsistencies in user interfaces are systematic in nature [9]. The issues are originating due to lack of a programming paradigm which can handle the data dependencies in a consistent manner.

5.2.4 Edmodo, Engrade, and, Vocareum

The remaining three applications we examined are Edmodo, Engrade, and, Vocareum. All three of these applications use simplistic UIs and minimalistic client-side scripting. Due to more server side scripting, these applications are consistent in behavior but extremely slow.

5.3 Industrial Interaction

To better understand the software developer's perspective, we contacted and discussed with GUI developers of two organizations, Expedia India and zyBooks. We presented them our findings and asked about the implementation mechanisms they used and rationale for their design decisions.

In our discussions with Expedia India's developers, we tried to find more information about the issues with the auto-complete text-box and *multi-tab search results*. Regarding the auto-complete text-box, they mentioned that they rely on jQuery's native auto-complete because it is difficult to create a completely new widget which can support all browsers perfectly.

Regarding the reservations to implement *multi-tab search results*, they mentioned two challenges. First, writing comprehensive search filters supporting multiple tabs results in complex event handling code. Second, supporting complex GUI elements in a tabbed structure has led to the crashing of the stylesheet mechanism in their

experiments.

In the second interaction, we met the two developers from the team of zyBooks. The developers mentioned that the grading mechanism logic is solely dependent on the event handlers. They are aware of the auto-grader issues and are working on a new GUI which will have multiple active listeners to record grading, which they think will solve the problems.

Our discussions with GUI developers indicate that in general developers are aware of the quality problems that come with event-driven GUI programming, and even with defects in their software. There are, however, no credible alternatives to event-driven GUIs. Therefore, developers today try to cope with GUI defects (caused by event-handling logic's complexity) either by not implementing or restricting certain features, or by patching up complex event-handling code from one revision to another.

6. EXPERIMENT IMPLEMENTATION DETAILS

In chapter 5, we examined commercial websites that have feature-rich user interfaces. The focus of our examination was the event handling mechanism of these applications. We observed that the event handlers in these applications are complex in nature and many of them exhibit asynchronous programming defects.

We argued that a major reason for these defects is the inconsistent design of data dependencies within event handlers. We know that the declarative approach (and HotDrink) gives strong guarantees about respecting data dependencies [9]. We can be assured that the kinds of defects we found will not appear if data dependencies are designed and implemented using HotDrink. To test, however, whether HotDrink can express the kind of feature-rich GUIs that appear in real practical systems, we developed a medium-size travel booking application. The goal of this implementation was not to present a fix to each individual kind of defect we found.

An additional goal of our implementation was to test if the declarative approach of HotDrink is non-intrusive with other libraries commonly used in practical GUI development. We wanted to present an example where HotDrink governs the logic of a GUI whose visual aspects have been built using other JavaScript libraries.

We start this chapter with a discussion of the main features of our application. We then, in Section 6.2, discuss the main technologies and frameworks we used in addition to HotDrink. After that, in Section 6.3, we explain some key implementations by discussing the related programming paradigms and code examples. Finally, in Section 6.3.3, we discuss the overall flow of data requests in our web application.

6.1 Website Features

The online travel booking application we developed for our experiment is a medium-size realistic website. The focus in our application design was to create a good user experience for a website visitor. We designed the GUI of our web application to resemble the look and functionality of commercial booking websites. We will now detail some important features of our website.

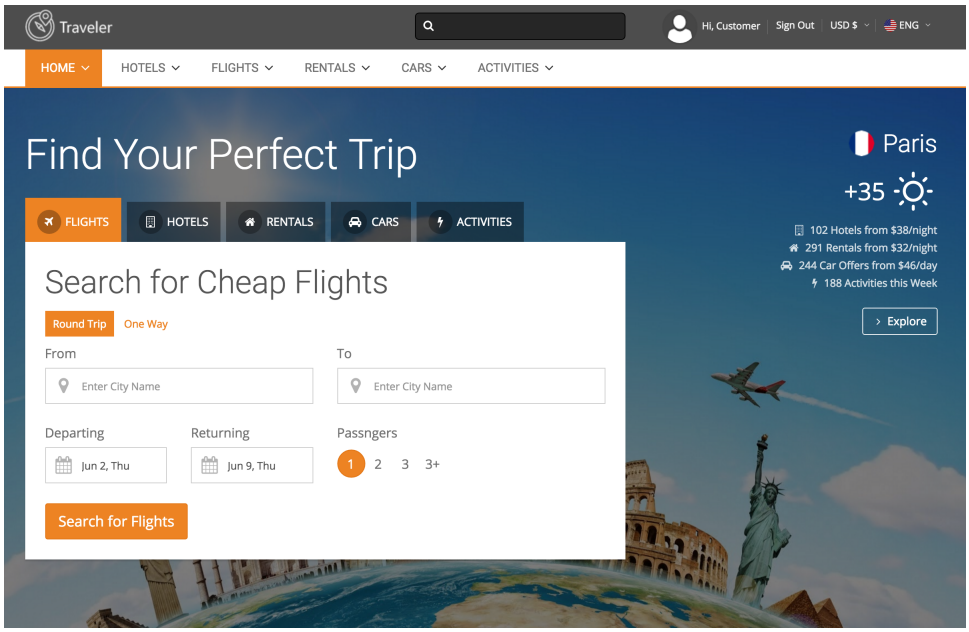

Figure 6.1: Screenshot of our travel booking website.

Our web application adapts to different screen sizes and dimensions. It is possible to use the application across a wide set of devices from desktop to mobile browsers. Figures 6.1 and 6.2 show the screenshots of our travel booking website from a desktop browser and a mobile browser, respectively.

The website features a booking platform for flights, hotels, rentals, and cars.

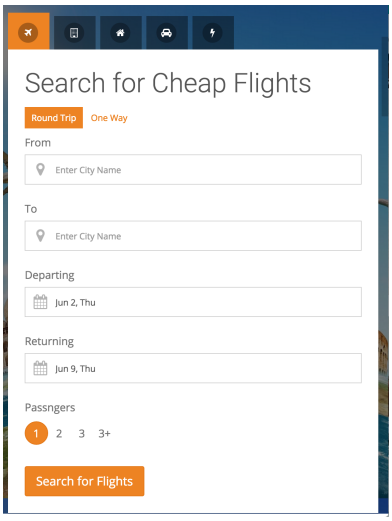

Figure 6.2: Screenshot of our travel booking website on a mobile browser.

The focus of our experiment is on online flight reservations, so we populated this functionality with a real database. We used a flight data-set of year 2012, provided by `openflights.org/data`.

With a realistic database, our web application features searching of flights between major international airports of the world. The flight search functionality of our web application has two significant features. The first feature is a consistent auto-complete text-box. This auto-complete text-box exhibits a consistent behavior irrespective of input speed variations. We will discuss the details of this implementation in Section 6.3.

The second significant search feature is *multi-tab search results*. The search dashboard presents the results of individual searches in a tabbed manner. With this feature, previous search results are not lost and the user can initiate a new search while the current search results are loading. Figure 6.3 shows the screenshot of *multi-tab search results* in our website.

The search results presented in individual tabs can be modified with search filters.

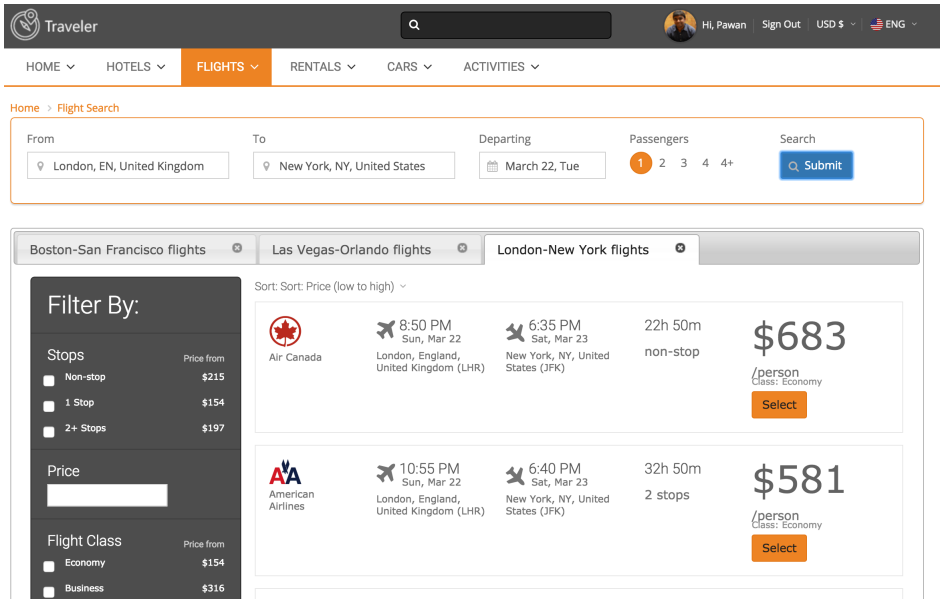

Figure 6.3: Screenshot of our travel booking website showcasing multi-tab search results.

Each result tab has its own search filter. Figure 6.3 shows search filters for one of the tabs. The search filters include check-boxes and sliders for restricting search data.

Our travel booking application features user profile management. The website user can create their profiles and track their past bookings. This feature has been modeled in the same manner as portrayed in commercial travel websites. Figure 6.4 shows the screenshot of user profile in our application.

We also worked on the typography and style-sheet of the website. We used easily readable panels and frames to improve the user-friendliness and aesthetics of the GUI.

6.2 Solution Stack within MVVM

We have designed our application using the MVVM architectural pattern. To understand the solution stack of our website, we discuss each segment of the MVVM pattern and describe the libraries and frameworks used in each segment. We will

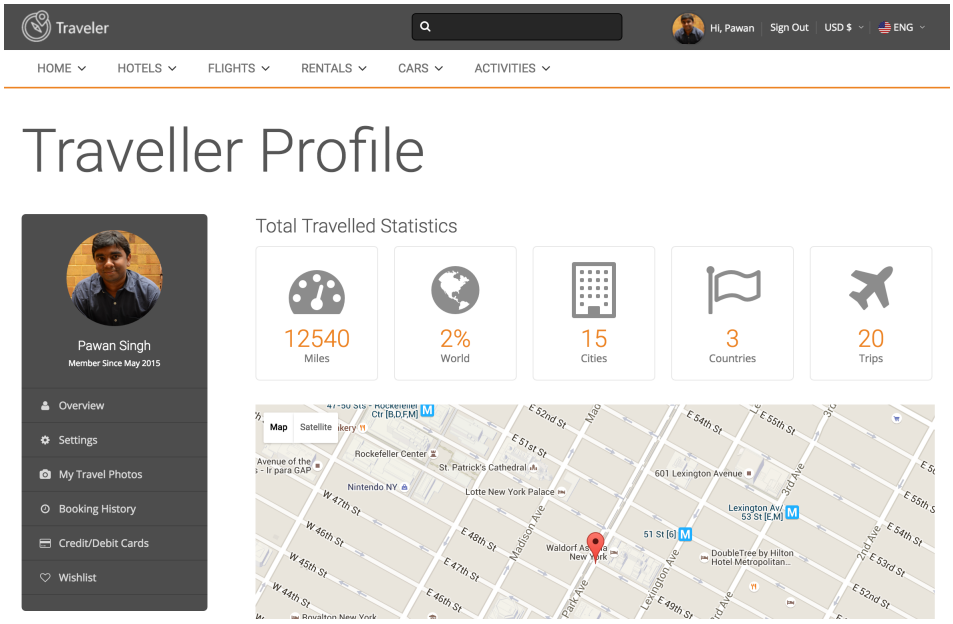

Figure 6.4: Screenshot of our travel booking website showcasing user profile.

start the discussion with the View of our web application, followed by the Model and the View-Model, respectively.

6.2.1 View

The View is the only part of the web application that a user actually interacts with. The View represents the state of a View-Model. The View does not handle the state of the application [11]. It represents the state visually and keeps itself in sync with the View-Model. The View is composed of UI elements and the data bindings associated with those elements. It requires events to capture user interaction and exhibit responsive behavior.

The UI elements in our application are developed using HTML5 and jQuery UI. The presentation theme and the style sheets have been developed using a library called Bootstrap. We will now discuss the significance of these three languages and libraries individually.

HTML5 is the fifth revision of the Hyper Text Markup Language. We selected HTML5 because of its multiple advantages for a modern web application. The most prominent advantage is its mobile-readiness, which means that it is fully supported by browsers of hand-held devices. Another advantage is the legacy and cross browser support. HTML5 also supports interactive and animated features in the GUI.

In our web application, we used multiple animated features to present HTML elements. These animations were developed using the canvas element of HTML5. The animations include a glowing effect on selected controls, disappearance of notification messages, and a fading effect of pop-ups. These effects add aesthetic value to the user experience. Apart from building UI elements, we used HTML5 extensively for declaring data bindings. The data bindings in our implementation are two-way bindings, which means that there is a synchronization between the HTML element and the corresponding DOM object.

jQuery UI [33] is a library of popular GUI components built on the top of the jQuery core library. It is composed of a set of HTML widgets, themes, and animation effects that are difficult to develop using HTML alone. The following are the specific widgets we have used from jQuery UI:

- Datepicker – We used this widget to select the journey dates.
- Progressbar – We used this widget to display the progress of producing results of a request.
- SelectMenu – We used this widget to provide a theme-able menu instead of a native HTML menu.
- Slider – We used this widget for filtering search results.
- Tabs – We used this widget to show multiple tabs on a single page.

- Tooltip – We used this widget to provide a useful theme-able tool-tip instead of native tool-tips.

The Bootstrap library [26] helps the user in designing and building web applications which can be properly rendered in all devices irrespective of the device's screen size and browser type. Bootstrap is composed of HTML element and CSS classes. The developer needs to build a single version of a web application and include Bootstrap library for CSS classes. Our web application used Bootstrap library for all its web pages.

6.2.2 Model

The model represents the actual data or information that the web application displays. This information is in form of entities, a data-source, and business objects. The Model is designed in a way that it is unaware of the details of the View or the View-Model.

In our application, we have chosen NodeJS and SQLite to serve the model. SQLite serves the role of a database while NodeJS serves as a runtime environment that hosts the web application and provides data from the database as a service to the View-Model.

NodeJS [6] is a highly scalable non-blocking I/O platform which is programmable in JavaScript. In our application, we used NodeJS to develop a comprehensive set of web-services. The web-services varies from simplistic ones, like providing airport names, to complex ones that register a complete itinerary booking. We are using a light-weight module of NodeJS called *Express* to run the server, host web-services on it and support dynamically rendering HTML pages.

SQLite [14] is an embeddable SQL database engine. It does not require a server process to provide database access. It organizes the database into a compressed set

of files and access is accomplished through standard read/write file operations.

We populated the SQLite database with the data-set obtained from `openflights.org/data.html`. In our setup, the SQLite database receives SQL query requests from NodeJS processes. The database engine of SQLite performs the query validation and executes the associated transactional operation. The result of the operation is sent back to the NodeJS process.

6.2.3 View-Model

View-Model is an intermediary layer between the View and the Model. It contains the logic to handle the View. The View-Model receives requests from the View. The View-Model then interacts with the Model by invoking web-services. The View-Model retrieves the required data and makes it available to the View for the display. It also performs formatting of data to make it compatible with the View.

In our application, a careful combination of AngularJS and HotDrink libraries was used to build the View-Model. The strength of AngularJS lies in the effective DOM manipulation of HTML elements. Additionally, it supports *partial views*. Partial views help in developing a reusable template for similar views of the web application. The strength of the HotDrink library lies in its capability to capture the essence of user interface behavior by a declarative specification of data dependencies [10]. The HotDrink library guarantees that the View-Model produces consistent UI behavior.

In order for the View-Model to participate seamlessly with the View, we need two-way data bindings. Both HotDrink and AngularJS libraries support two-way data binding. AngularJS features HTML templating. This feature helps in generating sophisticated HTML using simple AngularJS code. Within this AngularJS code, HotDrink data bindings are not recognized. On the other hand, when we use simple HTML pages to support HotDrink data bindings, AngularJS is not utilized.

The restriction of using either AngularJS or HotDrink for data bindings came as a challenge for our implementation.

In our application, we proceeded with two-way data bindings only for AngularJS and used HotDrink exclusively for handling the View-Model. We used such a combination because we wanted to utilize the templating capabilities of AngularJS and the constraint handling capabilities of HotDrink at the same time. To make the combination work, we used the *Observer-Observable* paradigm to arrange the interaction between AngularJS and HotDrink. Whenever a change in the View happens, AngularJS captures the information and passes it to HotDrink. Whenever a change in the View-Model takes place, HotDrink passes this information to AngularJS. The AngularJS reflects this change to the View. Section 6.3.1 discusses the *Observer-Observable* paradigm in detail.

6.3 Key Implementations

Until now in this chapter, we have discussed the significant features and the technological stack of our travel booking website. In this section, we discuss some of the key implementations that we developed for our travel booking website.

6.3.1 Observer-Observable

We opted for an *Observer-Observable* pattern within the View-Model of our web application. Observer is an object which is subscribed to the state-change of another object. Whenever the target object exhibits a change in its state, the Observer is notified of the change. Observable is an object whose state is of interest to another object. In a nutshell, the Observer object is subscribed to the changes in the Observable object.

We developed our UI elements with the help of AngularJS. The UI elements also have a two-way binding relationship with AngularJS objects. HotDrink registers

the AngularJS object as Observable and registers its own variable as Observer. For each variable in HotDrink there exists an object property or a variable in AngularJS to which the HotDrink variable is subscribed to. This mechanism migrates the data dependency logic from AngularJS to HotDrink. The role of AngularJS is thus restricted to generating UI elements, capturing user events and notifying HotDrink about the change in state. We leave no computations to be performed by AngularJS. HotDrink, on the other hand, examines the constraints, calculates the dependent values and notifies AngularJS about changes. AngularJS immediately updates the View.

Listing 6.1: Observer-Observable.

```
function Adapter(ngObject) {
    this.ngObject = ngObject;
    var that = this;
    ngObject.onChange = function(){
        that.sendNext( that.ngObject.ngVar ); } }

Adapter.prototype = new hd.BasicObservable();
Adapter.prototype.onNext = function(val){
    this.ngObject.hdVar = val;
    ngObject.onChange = function onChange(){
        hdVar.onNext(new Value);
    }
    hdVar.addObserver( onNext: function(val){
        ngObject.ngVar = val;  } ); }
```

The code snippet in Listing 6.1 shows an example of *Observer-Observable* pattern between objects of the AngularJS and HotDrink libraries. In this example, `hdVar` represents a HotDrink variable. The AngularJS object is represented by `ngObject`

and `ngVar` represents a property of this object. The value of this property is being observed by HotDrink variable `hdVar`.

We have used the *Observer-Observable* pattern to implement four features in our web application. These features include an auto-complete text-box, multi-tab results, result filters and the frequent flyer details retrieval. Modern travel booking websites implement these features either in a restricted or inconsistent manner. The HotDrink library simplifies the data dependency resolution for each of these features and provides an elegant methodology for implementing such complex features. In the next subsection, we discuss the implementation of the auto-complete text-box controlled by HotDrink through the *Observer-Observable* pattern.

6.3.2 Auto-complete Implementation

In traditional auto-complete widgets, inconsistent behavior appears because asynchronous executions violate the data dependencies within the widget. With the use of HotDrink, these data dependencies are explicitly defined as constraints in a property model. The property model ensures that all constraints are satisfied. If not, it solves the unresolved constraints and makes the system valid again.

With this approach, an auto-complete widget can be defined with the use of four variables. The variables are *input-string*, *current-selection*, *suggestion-list*, and *final-value* [9]. We described these four pieces of information in Section 5.1.1 with respect to event handlers. This time we define the auto-complete widget with constraints between these variables.

In place of using the regular jQuery auto-complete widget, we are using a custom-built auto-complete text-box. This custom auto-complete text-box has been built using AngularJS. This auto-complete text-box looks the same as the traditional widget, but it stores information in the form of the above mentioned four variables.

In this implementation, we have a same set of variables in HotDrink as well. The data dependencies are captured by HotDrink. The AngularJS variables just serve as the View with no logic. These variable are bound to the HotDrink variables.

Whenever the user hits a keystroke, AngularJS captures this information, notifies HotDrink, and HotDrink updates the *input-string* variable with the user input. As soon as this variable gets updated, the constraint system starts computing dependent values. The *suggestion-list* gets computed first. A method that performs an asynchronous web-service call is executed to retrieve the suggestions matching *input-string*. The variable *suggestion-list* gets updated with these suggestions when the web-service call is fulfilled. When the *suggestion-list* gets updated, HotDrink notifies AngularJS. AngularJS updates the GUI to show suggestions for the first time.

When suggestions appear in the GUI, the user can select any suggestion using the arrow keys. When this happens, AngularJS captures the index value of the selected item and sends it to HotDrink. HotDrink then sets the value of *current-selection* with this information.

The last variable is *final-value*. The computation of this variable requires *input-string*, *suggestion-list*, and *current-selection*. The method that solves this constraint uses the index value from *current-selection* to choose the appropriate item from *suggestion-list* and updates *final-value* with this item. When *final-value* is changed, AngularJS is notified of the update and it displays the output in the auto-complete text-box.

Even when the inputs are entered with a swift speed, this mechanism ensures that data dependencies are resolved in a systematic manner. The auto-complete text-box designed with the system of data-constraints exhibits consistent UI behavior.

Figure 6.5 shows the diagrammatic representation of data dependencies in an auto-complete text-box. In a similar manner, we formalized other complex widgets

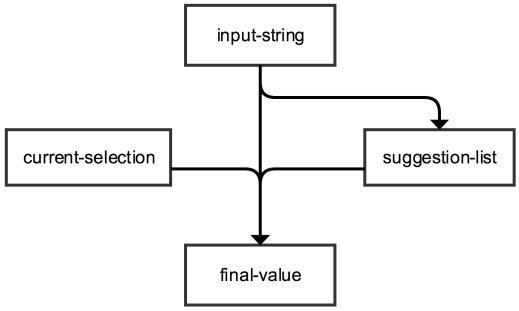

Figure 6.5: Representation of an auto-complete text-box in terms of HotDrink constraints.

in our application into structures with explicit data dependencies. HotDrink played the role of a constraint solver while AngularJS worked as a data binder in each of them.

6.3.3 Flow of Data Requests

In this section, we discuss the overall structure of data transfers in our web application. Figure 6.6 shows a diagrammatic flow of requests in our web application. The flow of requests starts when the website is loaded for the first time. At the very first time, HTML, CSS, and the associated JavaScript libraries are loaded in the client browser. The Bootstrap library ports the rendered web pages to fit the display of the device. AngularJS library uses *partial views* (or a master view) that are re-used to show different web pages with minimum loading of resources. AngularJS follows XHR or XMLHttpRequest, which ensures that the application supports HTTPS and follow modern browsers' same origin policy [23].

For simpler tasks like resource fetching, AngularJS sends request to the server directly. For complex tasks like data constraint resolutions, AngularJS acts as a mediator and passes requests to HotDrink. HotDrink examines the nature of user

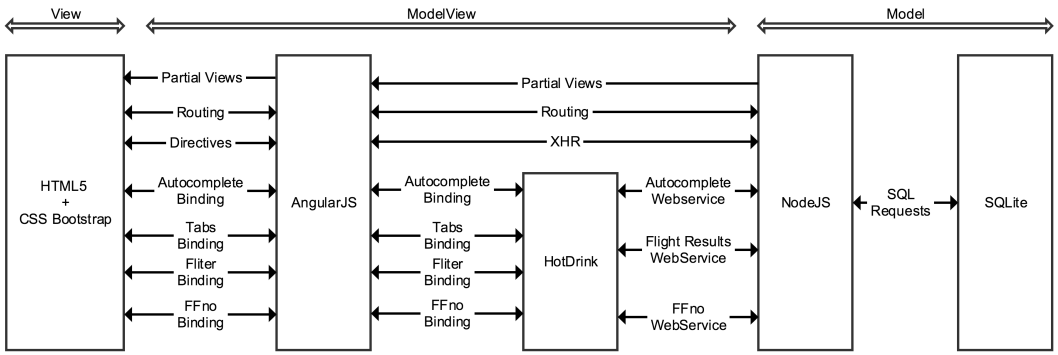

Figure 6.6: Diagrammatic representation of the request flow in our web application.

interaction, assesses the impact on data constraints, and requests new data from the server if necessary.

NodeJS fulfils the server requests. A requests is either a resource request or data request. The resource requests involve images or HTML/CSS content. For fulfilling data requests it generates SQL queries to the SQLite database. The results from the database are formatted and sent back as a response to the HotDrink Library.

7. RESULTS

In this chapter, we describe our hypotheses and a qualitative analysis of our implementation experiment.

Hypothesis 1: The programming approach of property models supports the implementation of web applications' GUIs with at least moderate size and complexity.

In our experiment, we developed a web application for online travel booking. This web application is moderate in size and features a relatively complex GUI. The GUI of this web application is similar to a commercial travel booking website. We used JavaScript templating libraries to design the GUI.

Majority of the GUI logic in our web application was implemented using constraints of property models. We did not encounter any scenarios that we could not express as constraints.

The GUI functionalities expressed by constraints in our implementation vary in their nature. Some of these functionalities are simple, such as calculating discounts, total fare, etc. There are some functionalities that involve asynchronous querying and data manipulation, like search-filters. Some of the functionalities express constraints for advanced widgets like auto-complete text-box, sliders, etc.

In addition to expressing different kinds of GUI features, from trivial to substantial, using constraints, we tested the application built with constraints using a large data-set. The database of our web application contains a real flight schedule data-set of year 2012. It contains 14136 routes between 209 airports on 51 airlines. We never faced any challenges in presenting this relatively large data-set on a GUI using constraints.

With the successful implementations of GUI functionalities and large data-set handling, we can say that the programming approach of property models and HotDrink library are well-suited for developing the GUI of web applications with at least moderate size and complexity.

Hypothesis 2: Property Models are not intrusive in nature.

In our experiment, we observed that JavaScript templating libraries rely solely on event handling. These libraries can build professional looking GUIs but they do not have a mechanism for managing dependencies for the programmer, which is what property model provide. With this fact in mind, we explored the possibility of combining a modern event handling based library and property models in the same application. To make this combination work, we tested different techniques for exploring communication between the objects of AngularJS (a JavaScript templating library) and the HotDrink library.

We found that the *Observer-Observable* pattern is effective in combining these two libraries. This design pattern enables the objects of these two libraries to send notifications to one another whenever there is a change in state. This pattern helped the two libraries to complement each other.

We used the *Observer-Observable* pattern to integrate AngularJS with HotDrink in our web application. The combination proved to be successful and the templating capabilities of AngularJS were utilized without any hindrance. We also conducted an experiment to test the integration of a Data Visualization library called D3 [5] with HotDrink. We again used the *Observer-Observable* pattern. The combination ran successfully and visualizations were rendering without any issue.

With these successful experiments, we can state that the concept of property models is not intrusive in nature. It allows other libraries to function seamlessly

when combined with property models using the *Observer-Observable* pattern.

Hypothesis 3: Property models support the implementation of responsive GUIs better than event-driven programming does.

In our experiment, we used property models to program GUI widgets that handle asynchronous executions. We observed that property models can handle the asynchronous execution of data dependencies consistently. This is possible because a property model maintains the sequence in which requests have been made and allows asynchronous execution to take place as soon as, but no sooner than, the associated input is available. This mechanism ensures that all variables contain the most recent value available and no computation is ever based on stale values [9].

A significant example demonstrating the consistent handling of asynchronous events is the auto-complete widget that we developed using HotDrink. In Section 5.1.1, we discussed briefly the asynchronous execution problems associated with the traditional auto-complete text-box. Later, in Section 6.3.2, we discussed the detailed implementation of an auto-complete text-box using HotDrink. Our auto-complete prototype serves as a reference for the implementation of any UI widget that involves asynchronous computations.

In an another example, we used property models to resolve an ordering bug in retrieving frequent flyer details. This bug occurs, e.g., in `expedia.com`, a commercial travel booking website. Due to this bug, frequent flyer details can only be fetched if the inputs are entered in the strict sequence of provided text-boxes. We programmed the same functionality in our travel booking application with the help of constraints. With no additional effort by the programmer, frequent flyer details are always retrieved irrespective of the sequence of the inputs.

In our web application implementation, there are many examples (mentioned

in Section 6.3) that show how property models resolve common web application defects. With such concrete examples, we can state that property models support the implementation of a more responsive and consistent GUI compared to event driven programming.

Hypothesis 4: Programming approach of property models reduces coding effort.

In our experiment, we tried to get a feel if the programming approach of property models can reduce coding effort compared to event-driven programming. Our experiment was concentrated on variety of tasks but we were not able to come up with a concrete evidence that differentiates coding efforts. Therefore, we cannot say anything definite based on our experiment. The two programming approaches are fundamentally different.

Event driven programming is easily programmable with event handlers. Property models require programmers to express the logic in terms of constraints. The code written using event handlers results in a complex software artifact, whereas the code written using property models results in a simpler software artifact. Our intuition is that event handlers are easier to program initially but more difficult to debug than property models. With event-driven programming it seems thus to be more difficult to produce a full correct implementation of a GUI.

8. COMPARISON OF IMPLEMENTATION EFFORT

In this chapter, we first, in Section 8.1, describe the methodology of developing a modern web application with constraint based web frameworks other than HotDrink. Then, in Section 8.2, we compare it with the development effort following our methodology of using HotDrink as a specialized ViewModel.

8.1 Knockout and ConstraintJS

The most popular web framework based on data-flow constraint is Knockout [29]. The important features of Knockout are declarative binding, observables, and dependency tracking. Declarative bindings are the same as data bindings between DOM elements and ViewModel objects. When a ViewModel variable is declared as observable, Knockout ensures that any modification to this variable is reflected everywhere and the associated datadependencies are updated. The implicit data dependency detection and computation in Knockout is called dependency tracking.

Typically, Knockout is used to develop the ViewModel and the View is developed with a JavaScript templating library. The most common complementary JavaScript templating library to Knockout is Underscore [2]. There are more popular templating libraries than Underscore, e.g., AngularJS and React, but using Underscore is most tightly integrated with Knockout.

Underscore is used to develop HTML templates and to reuse templates for different Views of a web application. Underscore supports Knockout's declarative bindings within its template code. This makes the combination of Underscore and Knockout very powerful in developing UI logic.

Once the View has been designed, ViewModel is developed to define the behavior of the View. With declarative binding, Knockout maintains two-way bindings. The

concept of observables helps in extending these bindings to data dependencies. Data dependencies are not declared explicitly like in HotDrink. Instead, variables can be declared observables and logic written to compute values for them.

Listing 8.1: Observable in Knockout.

```
var myViewModel = {
    length: ko.observable(0),
    breadth: ko.observable(0),
    area: ko.computable(function () {
                var value = this.length()*this.breadth();
                return value;
            }, this)
    };
```

In Listing 7.1, we can observe that to make a variable Observable, we declare it with function `ko.observable()`. To define a relationship, we declare a variable with function `ko.computable` and define the logic in it between the required variables. In this example, `length` and `breadth` are observables. Whenever, `length` or `breadth` is modified `area` will be computed again.

Another JavaScript library for constraint based programming is ConstraintJS [24]. It merges the concept of one-way constraints with Finite State Machines. With this combination, the UI behavior is expressed in term of states. ConstraintJS supports multiple Finite State Machines for defining the behavior of an individual UI element.

A simple example can be the check box. Depending on whether a check box is in focus or not, or checked or not, there can be multiple states, and in each state the data dependencies in a GUI can be different. UI behavior can be programmed in ConstraintJS by defining different states of the UI element and declaring which constraints are valid in each state.

Like Knockout, ConstraintJS also supports two-way bindings. Also, ConstraintJS has been primarily developed for the ViewModel and it depends on other JavaScript templating libraries for developing the View. The recommended templating library for ConstraintJS is Handlerbar [19] as it supports the data bindings of ConstraintJS inside template code.

To program the ViewModel using ConstraintJS, we need to declare variables within the function `cjs`. ConstraintJS analyses all the variables and functions defined under the `cjs` function and builds the data dependencies it finds. Whenever there is a change in a variable, all the dependent values (which are defined in the `cjs` function) are computed again. Listing 8.2, shows an example in ConstraintJS. In this example, variable `length` and `breadth` have been declared with `cjs` function. Whenever `length` or `breadth` is modified, the variable `area` is automatically updated.

Listing 8.2: Constraints in ConstraintJS.

```
var length = cjs(1); // length <- 1
var breadth = cjs(2); // breadth <- 1
var area = cjs(function(){
                return length*breadth;}); // area <- length*breadth

area.get(); // returns 2
length.set(10); // length<-10
area.get(); // returns 20
```

8.2 HotDrink

The methodology of developing web applications using HotDrink, which we presented in the previous chapter, has significant advantages over Knockout and Con-

straintJS. One significant feature is the dependency mechanism of asynchronous computations. Both Knockout and ConstraintJS express the data dependencies as functions, converting the asynchronous nature of computations to synchronous [9].

Another advantage of HotDrink over Knockout and ConstraintJS is the ease of expressing multi-way constraints. Knockout and ConstraintJS support one-way constraints. Both can simulate multi-way constraints through complicated mechanisms but they do not support them as a native feature.

Third advantage of HotDrink is the explicit declaration of data dependencies in form of a reusable data structure [9]. This data structure can serve as the foundation for different algorithms. For example, one can identify variables directly edited by the user and those computed by the system. This information can be used to simplify GUI programming, e.g., for automatically enabling and disabling widgets [9]. Also, the explicit declaration perhaps helps the programmer to better maintain a complete picture of all the system's constraints as compared to Knockout and ConstraintJS, where constraints are detected by the system.

In our implementation, we used the *Observer-Observable* paradigm for HotDrink to interact with the templating library AngularJS. Our implementation does not have any preference for a particular templating library for developing the View. We can use any templating library with HotDrink using the *Observer-Observable* paradigm.

To test the compatibility of HotDrink with more libraries, we developed complex data visualizations using the D3 library [5] with HotDrink. Figure 8.1 shows the screenshot of one such data visualization. In this visualization, the individual bubbles represent countries. The x-axis represents the per-capita income and the y-axis represents the life expectancy. The slider in the bottom can be used to find the position of countries between 1800 to 2008. The relative positions of the bubbles are computed in HotDrink and D3 is informed to reflect them in the View.

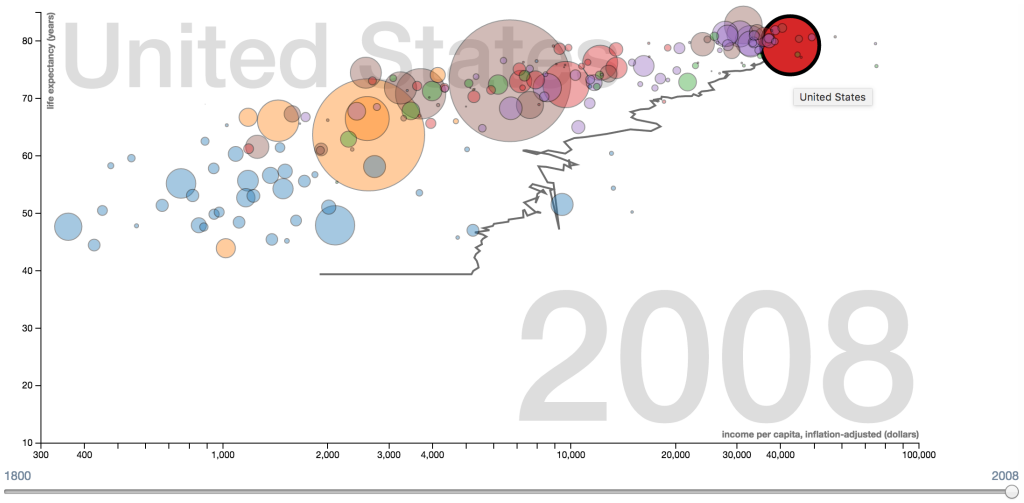

Figure 8.1: Data Visualization using D3 and HotDrink.

By clicking an individual bubble, we can see the path it traced. The path is a collective representation of all the positional data points that a bubble element can attain in this data-set. To easily obtain this data is possible due to another capability of HotDrink, namely that it retains the older values of its variables, if desired. The given screen-shot shows the path traced by the bubble representing the United States.

Our implementation shows that HotDrink is an excellent choice to serve as a GUI ViewModel. The implementation supports our claim that UIs with consistent behavior can be successfully developed with declarative data-flow constraints. It also shows that HotDrink can be used effectively with modern JavaScript libraries using the *Observer-Observable* paradigm.

9. FUTURE WORK

In this section, we outline some of the future work that we plan to undertake in this domain. There are research areas closely related to this thesis that we would like to develop further.

We would like to investigate the possibility of using property models not only on the client but also on the server. The question then becomes how to synchronize the corresponding property models on the client and server sides. The existence of JavaScript in server-side scripting provides an opportunity to use HotDrink in exploring such mechanisms.

We would like to study caching mechanisms of web applications with property models. Data intensive web application constantly make AJAX requests and cache data on the client system. The approach of property models can formalize this unstructured process into a set of dependencies. In such a system, property models can evaluate the urgency of fetching updates from the server and reduce unnecessary AJAX requests.

We would also like to explore HTML templating capabilities for HotDrink. With such capabilities, HotDrink could possibly be used to build web applications with fewer supporting libraries. Using fewer JavaScript libraries lessens the load of programmers, and can also reduce the load time and improve performance of web applications.

10. CONCLUSION

In this thesis, we present an in-depth analysis of using the declarative programming approach of property models for web application development. We discuss our implementation of a web application built with HotDrink, the concrete realization of the property model approach. We explored the use of this library with popular JavaScript web frameworks. We also compared this methodology with other declarative programming approaches. Our conclusions are as follows:

- The declarative programming approach of property models fits the role of View-Model and is well-suited for developing the GUI logic of web applications.
- Many asynchronous programming defects in web applications are avoided when using the declarative programming approach of property models. A property model ensures that data dependencies between the pieces of data in a GUI are not violated by asynchronous executions of responses to user events.
- Complex web widgets like multi-tabs and search-filters can be programmed effectively with property models.
- The *Observer-Observable* pattern can be used effectively to utilize the constraint system of property models on any View that has been built with JavaScript.

REFERENCES

[1] Alexa. Alexa — Top Sites by Category: Recreation/Travel. https://alexa.com/topsites/category/Recreation/Travel, 2016.

[2] Ashkenas, J. Underscore.js. http://underscorejs.org, 2016.

[3] Association of Universities for Research in Astronomy. NOAO Science Archive. http://archive.noao.edu/tutorials/query, 2016.

[4] Blackboard. BlackBoard Learn. https://tamu.blackboard.com, 2016.

[5] Bostock, M. D3.js — Data-Driven Documents. https://d3js.org, 2015.

[6] Dahl, R. Dockerizing a Node.js web app | Node.js. https://nodejs.org/en/docs/guides/nodejs-docker-webapp/, 2016.

[7] Expedia Inc. Expedia Travel. http://expedia.com, 2016.

[8] Flanagan, D. *JavaScript: The Definitive Guide*, chapter 17.1, pages 447–455. O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 2011.

[9] Foust, G., Järvi, J., and Parent, S. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 121–130, New York, NY, USA, 2015. ACM.

[10] Foust, G., Järvi, J., and Freeman, J. Hotdrink: JavaScript MVVM library with support for multi-way dependencies and generic, rich UI behaviors. https://github.com/HotDrink/hotdrink, 2015.

[11] Freeman, J., Järvi, J., and Foust, G. Hotdrink: A library for web user interfaces. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 80–83, New York, NY, USA, 2012. ACM.

[12] Google Inc. AngularJS – Superheroic JavaScript MVW Framework. https://angularjs.org, 2015.

[13] Gradescope. Gradescope. https://gradescope.com, 2016.

[14] Hipp, D.R. About SQLite. https://www.sqlite.org/about.html, 2015.

[15] Järvi, J., Foust, G., and Haveraaen, M. Specializing planners for hierarchical multi-way dataflow constraint systems. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 1–10, New York, NY, USA, 2014. ACM.

[16] Järvi, J., Haveraaen, M., Freeman, J., and Marcus, M. Expressing multi-way data-flow constraint systems as a commutative monoid makes many of their properties obvious. In *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*, WGP '12, pages 25–32, New York, NY, USA, 2012. ACM.

[17] Järvi, J., Marcus, M., Parent, S., Freeman, J., and Smith, J. Algorithms for user interfaces. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 147–156, New York, NY, USA, 2009. ACM.

[18] Järvi, J., Marcus, M., Parent, S., Freeman, J., and Smith, J.N. Property models: From incidental algorithms to reusable components. In *Proceedings of the*

7th International Conference on Generative Programming and Component Engineering, GPCE '08, pages 89–98, New York, NY, USA, 2008. ACM.

[19] Katz, Y. Handlebars.js: Minimal Templating on Steroids. http://handlebarsjs.com/, 2016.

[20] Microsoft Corporation. The MVVM Pattern. https://msdn.microsoft.com/en-us/library/hh848246.aspx, 2014.

[21] Mozilla Developer Network. Overview of Events and Handlers. https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Overview_of_Events_and_Handlers, 2005.

[22] Mozilla Developer Network. Ajax | MDN. https://developer.mozilla.org/en-US/docs/AJAX, 2016.

[23] Mozilla Developer Network. Same–origin policy — Web security. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2016.

[24] Oney, S., Myers, B., and Brandt, J. ConstraintJS: Programming interactive behaviors for the web by integrating constraints and states. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 229–238, New York, NY, USA, 2012. ACM.

[25] O'Reilly, T. What Is Web 2.0—O'Reilly Media. http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html, 2005.

[26] Otto, M. and Thornton, J. Bootstrap. The world's most popular mobile-first and responsive front-end framework. http://getbootstrap.com, 2016.

[27] Paterson, I., Smith, D., Saint-Andre, P., Moffitt, J., Stout, L., and Tilanus, W. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). https://xmpp.org/extensions/xep-0124.html, 2014.

[28] priceline.com LLC. Priceline.com. http://priceline.com, 2016.

[29] Sanderson, S. Knockout : Introduction. http://knockoutjs.com/documentation/introduction.html, 2016.

[30] Sarda, S. private communication, Expedia India, April 2016.

[31] Statista. Online activities: internet users who managed travel reservations online within the last month (USA), 2015. http://www.statista.com/statistics/228736, 2015.

[32] The jQuery Foundation. jQuery. http://jquery.com, 2016.

[33] The jQuery Foundation. jQuery UI. http://jqueryui.com, 2016.

[34] Warren, A.M. *Project-Based Learning Across the Disciplines: Plan, Manage, and Assess Through +1 Pedagogy*, chapter 4, pages 123–124. Corwin, 2455 Teller Road, Thousand Oaks, CA 91320, USA, 2016.

[35] Yahoo. Flickr Services. https://www.flickr.com/services/api/, 2013.