In-depth Survey of MVVM Web Application Frameworks

Konstantinos Zarifis
University of California, San Diego
zarifis@cs.ucsd.edu

ABSTRACT

Frameworks that adopt the Model-View-Controller (MVC) design pattern have been extensively used in the web community for the development of fully-fledged web applications. Such frameworks enable efficient incremental updates on the application's state and visual layer, but they usually enforce the extended use of imperative logic in order to accomplish this effect. As an application is extended with additional functionality, the development process soon becomes extremely arduous and error-prone. This has lead to the emergence of Model-View-ViewModel (MVVM) and Web Component libraries that achieve higher developer productivity by keeping the required source code minimal and well-organized. Such frameworks can also provide additional mechanisms that automatically maintain the application state and the respective visual layer in sync, thus alleviating the application developer from this task. On the downside such mechanisms can negatively impact the performance of a given application and cause noticeable irregularities to the user experience.

This research exam surveys MVVM and Web Component libraries that constitute the state-of-the-art in the web community. It also provides accurate definitions of the modules that compose an MVVM and a Component library and contains detailed description of the internal workings of each individual framework. Furthermore, this survey focuses on the mechanisms that are employed by MVVM and Component libraries to propagate changes from the application state to the respective part of the visual layer and describes the advantages and disadvantages of each individual mechanism. Lastly, we introduce FORWARD and its respective mechanisms that accomplish change propagation in an asymptotically more efficient manner than the respective mechanisms employed by competing frameworks.

1. INTRODUCTION

Web application frameworks and libraries have proven their importance in building web applications since the Web 1.0 era. They have managed to absolve application developers from the distraction of implementing mundane boilerplate code, thus allowing them to work at a higher level of abstraction. Opinionated frameworks have accommodated application developers structure their code in a way that favors readability, consistency and maintainability by adopting well known design patterns, while at the same time assisting in avoiding common bad practices that ultimately lead to error-prone code.

Frameworks that adopt the Model-View-Controller (MVC) design pattern were, until very recently, the state-of-the-art for implementing fully fledged client-side web applications. When using such frameworks, the application developer is solely responsible for manually specifying the logic that instantiates the state and the respective visual layer of the application. Furthermore, as changes occur to the back-end services and as the user interacts with the application's view, certain mutations typically need to be propagated to the state and the visual layer of the application. This change propagation is explicitly handled by the application developer who employs imperative logic for every event that might cause such changes. The extended use of imperative code makes application development very laborious and error-prone, and results in applications that are very difficult to debug and maintain.

These issues have led to the recent emergence of frameworks that adopt the Model-View-ViewModel (MVVM) design pattern, which significantly improves development productivity and code reliability. MVVM frameworks typically allow the use of declarative code that specifies the view of the application, given an object that represent the application state. As the developer mutates the application state, the framework is able to automatically infer and apply the respective changes to the view of the application. This notably decreases the amount of code that has to be written by the application developer since the added logic is only responsible for applying updates to the state of the application and not to the visual layer.

Another set of web libraries that assist in limiting the use of imperative code when implementing applications, are Web Component libraries. While these libraries do not necessarily follow the MVVM design pattern, they promote the concepts of compartmentalization and Separation of Concerns [1] in web development by supporting the implementation of self-contained reusable Web Components. Web Components can be declaratively utilized as building blocks of bigger more complex applications, thus enriching their functionality while minimizing the amount of code required. Web Components can be used both for managing the state and the visual layer of an application in a declarative manner, but since their modular design can be quite restricting, the application developer is often forced to implement Custom Web Components which in most cases requires the use of imperative logic.

Both MVVM and Component libraries are equipped with various mechanisms that enable automatic change propagation from the application state to the view. These mechanisms are a significant part of a library since they are respon-

sible for simplifying the development process of a modern application. On the downside, under certain circumstances they can negatively affect the memory utilization and the run-time performance of an application. Since these mechanisms are a fairly important part of the aforementioned libraries we will examine them extensively by specifying their internal operations and describing how particular scenarios can dramatically impact their performance.

Additionally, this survey contains a creative part that describes the internal architecture and mechanisms of FORWARD, an MVVM framework that is currently under development by the UCSD Database Group. This framework employs advanced techniques that promote efficient propagation of changes throughout the life-cycle of an application. The employed techniques and the general architecture of FORWARD is heavily influenced by extended research that has been conducted in the area of Incremental View Maintenance (IVM) [2, 3], thus making it particularly suitable for applications that receive frequent updates on their logical and visual layer.

Paper Outline. The paper is structured as follows: In Section 2 we introduce the background information that is required for understanding this survey. This section describes the historical evolution of web frameworks, introduces the terminology that will be used throughout the paper and defines critical, for our analysis, concepts. In Section 3 we provide information about existing MVVM frameworks. Specifically, we describe their programming model, their internal architecture and general advantages and disadvantages of each framework. In Section 4 we describe the special attributes that define a Component library and lastly we provide an extensive description of the special characteristics of each individual Component library. Section 5 provides an in-depth analysis of the architecture and internal workings of FORWARD and finally, Section 6 concludes the paper.

2. BACKGROUND

Web applications have come a long way since the beginning of the WEB 1.0 era ([4, 5]), from simple read-only static pages they have evolved into fully-fledged apps that are capable of completely extinguishing the need for equivalent desktop applications. As the requirement for more exotic features in modern apps increases, so does the need for application frameworks that simplify the process of implementing such complex systems. In this section, we describe the historical evolution of web applications by reviewing the architectural design changes that occurred over time and we define the basic concepts that are used for the classification of the frameworks we investigate.

2.1 Web 1.0

Starting with Web 1.0 applications, we will define the building blocks they consist of and analyze the advantages and disadvantages of their architecture. While there are various definitions throughout the Internet about what constitutes a web 1.0 application [4, 6, 7] these definitions are often too abstract and inconsistent, thus leading to severe misconceptions. As far as this survey is concerned, Web 1.0 applications typically consist of a client, a server and optionally, a remote database used for persistency (as shown in Figure 1). The server is responsible for computing a description of the view that is later transferred to the client in order to be displayed. The client of a Web 1.0 application lacks any business logic (thin client) and it is only responsible for displaying the view that is transmitted from the server.

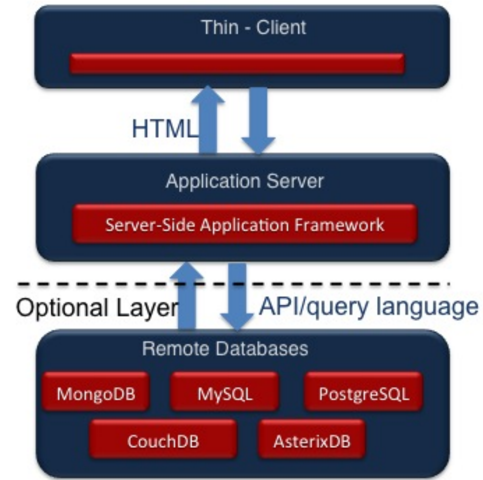

Figure 1: Web 1.0 Architecture

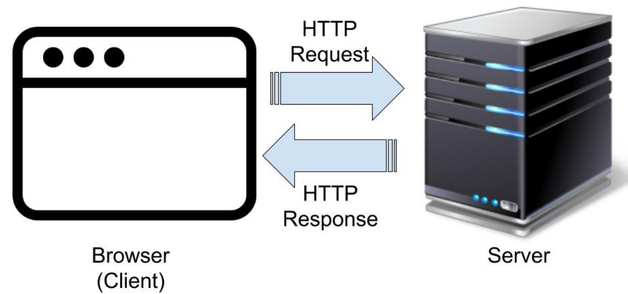

Figure 2: Client-Server Interaction

The life-cycle of a typical WEB 1.0 application requires frequent interactions between the client and the server (as shown in Figure 2). It begins when the user utilizes a browser to navigate to the address of a remote server. When this event occurs, an HTTP request is transmitted from the browser to a remote server. Upon retrieval, the server replies with an HTTP response, which contains a static HTML string that describes the initial view of the application. When the response is received by the client, the HTML text is parsed, evaluated and rendered, thus generating the view of the application. The evaluation stage includes the instantiation of an internal data structure called *DOM Tree* [8]. This data structure is mostly [9] isomorphic to the HTML text transmitted by the server and it is instantiated and utilized by the browser in order to render the view of the application.

A typical view may contain links, checkboxes, buttons and other user interface (UI) elements. As the user interacts

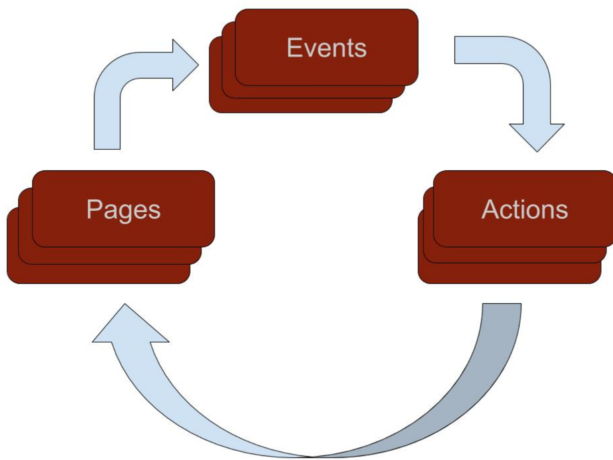

Figure 3: Action-Page Cycle

with such UI elements he triggers events that result to new requests to the back-end server, which responds with the HTML string of the next page. This circular process of actions causing the generation of new pages is called *action-page cycle* (shown in Figure 3), and it is a very simple conceptual model that has been used by most web technologies since the very first years.

It's worth mentioning that other than page reloads, an action can also cause side-effects. We define *side-effects* as operations that cause mutations on data but do not necessarily trigger changes to the view of the application. Examples of such events include storing information to a remote database, charging credit cards, or performing any other operations that mutate the state of the application.

Disadvantages of Web 1.0 applications. While 1.0 applications have a fairly straightforward programming model (action-page cycle), they also have several disadvantages, mainly due to their monolithic architecture. Such applications require both a client-side and a server-side part in order to function properly, as shown in Figure 1. The client-side application is mostly a stateless, thin client, that depends on the server for the re-computation of the view and the execution of side-effects. This results in a heavily interrupted user experience, since every time a user triggers an action, a new HTTP round-trip has to be completed before the new view appears on the screen. While the round-trip is still in progress, the client remains idle and the user witnesses a blank screen until the response is retrieved and displayed by the client.

Furthermore, even if the consecutive views that appear as the user interacts with an application have little to no difference from one another, a new page still has to be recomputed in its entirety, transmitted over the wire and rendered on the client. This leads to a significant increase in both the computational footprint and bandwidth utilization of the application, especially since HTML tends to be heavily nested and verbose. Furthermore, Web 1.0 applications are unable to support visualizations, which significantly limits the features that can be provided by an application.

2.2 Infusing the client with logic(Web 1.5)

In an attempt to resolve the issues that are associated with Web 1.0 applications, developers started infusing the client with logic by injecting JavaScript code and external JavaScript libraries (such as jQuery [10]) on the client [11]. One of the biggest advantages of this approach is that it completely eliminates the "blocking" user experience, Web 1.0 applications could not avoid. Particularly, by utilizing client-side logic, applications can perform asynchronous calls to the back-end and retrieve essential for the application datasets, while the user is still able to interact with the UI. Additionally, after retrieving these datasets, the application developer can manually update only the DOM elements that need to be modified, thus assisting in decreasing the computational penalty that was caused by the full reevaluation and rerendering of the entire view at the end of the action-page cycle.

Incorporating logic into the client also leads to limited interaction with the server, since certain actions can now be performed directly by the client. For cases in which the interaction with the back-end cannot be avoided, this architecture enables the use of formats that are far less verbose than XML/HTML for transmitting information "over the wire" (such as JSON [12]), thus significantly decreasing the bandwidth utilization that is required by applications. Lastly, another advantage of using JavaScript on the client is the advanced user experience that can be achieved by leveraging visualization libraries. Application developers can now use 3rd-party libraries to enrich the visual layer of their applications with reactive components such as maps [13, 14], [15, 16], charts [17, 18] and graphs [19, 20], thus improving the user experience of their applications and enabling data exploration.

Disadvantages of Web 1.5 applications. While infusing the client with logic certainly enriched web applications with features that were not possible before, this logic was introduced in a mostly ad hoc way, which triggered various negative side effects. Perhaps the biggest downside of this, is the inevitable inconsistent and tangled code (also known as "spaghetti code" [21]) that most client-side applications consist of. The main reason behind this is the fact that developers simply inject JavaScript code within HTML documents without following a particular design pattern. This approach leads to severely disorganized applications, that become more convoluted as new features are being added over time. An inconsistent and tangled piece of software is very difficult to maintain and debug, especially in the case of loosely-typed languages such as JavaScript.

This problem cannot be resolved even if the application developer decides to adopt some design pattern, in order to organize his/her code. Particularly, if the developer chooses to split his/her code into self-contained JavaScript files and import them from the respective HTML files, the fact that all the instantiated variables and functions are loaded into the global name-space often leads to naming conflicts. This proves that a design pattern has to be natively supported by a framework, in order for such issues to be avoided. For this reason, *opinionated* frameworks were introduced to assist in application development, by enforcing design patterns and general best practices that comply with the concept of *Separation of Concerns* [1].

2.3 Design Patterns on the Client (Web 2.0)

In this section we will describe the two main design patterns that are adopted by most opinionated frameworks cur-

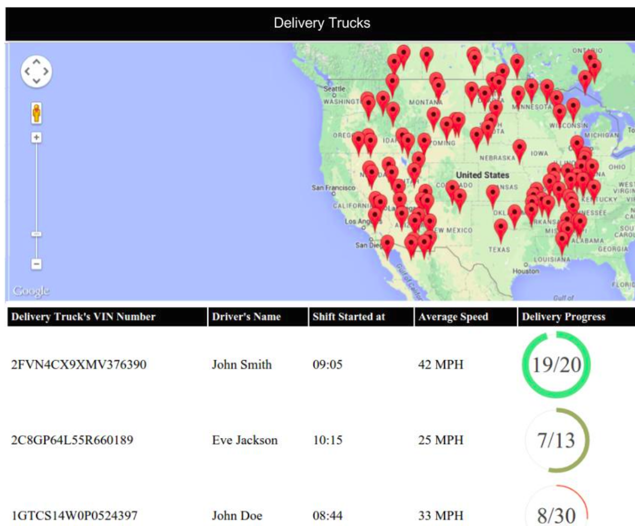

Figure 4: Product delivery map

rently. To aid in the presentation of these design patterns we will introduce a sample application utilized by a delivery company. The application contains a dashboard that enables easy tracking of delivery trucks that are currently delivering products. The view of the dashboard (shown in Figure 4) contains a feed of the location of the company's delivery trucks on a Google Maps [22] component, followed by an HTML table that contains information about each individual delivery truck. Particularly, each row contains the VIN number of the delivery truck, the name of the driver, the time when the current driver's shift started, the average speed of the delivery truck and lastly a Progress Bar component [23] showing the fraction of items that have been delivered. Both the individual cells of the HTML table and the markers shown on the map are updated in real-time

2.3.1 Model-View-Controller (MVC) Design Pattern

One of the most widely used paradigms in modern applications is the Model-View-Controller design pattern. According to this pattern, an application consists of three individual components:

- A **View** is the visual layer of the application. It contains all the UI elements the user sees and interacts with, such as charts, diagrams, check-boxes or plain text.
- A **Model** is an abstract representation of the data utilized by the application.
- A **Controller** is the part of the architecture that manages the other two components. Most of the times the Controller consists of imperative code written by the application developer, and contains both the business logic of the application and the logic responsible for updating the visual layer.

We will now utilize the running example introduced earlier in this section to showcase the various components of the MVC design pattern. The Model in this case is simply a list of delivery trucks. Each individual delivery truck is represented by a JavaScript object but since these objects

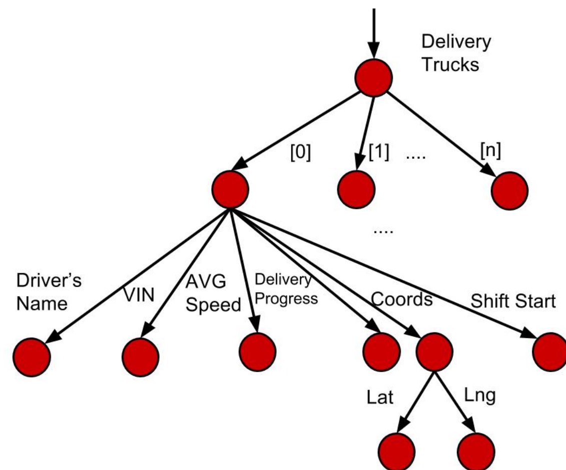

Figure 5: Model used by Running Example

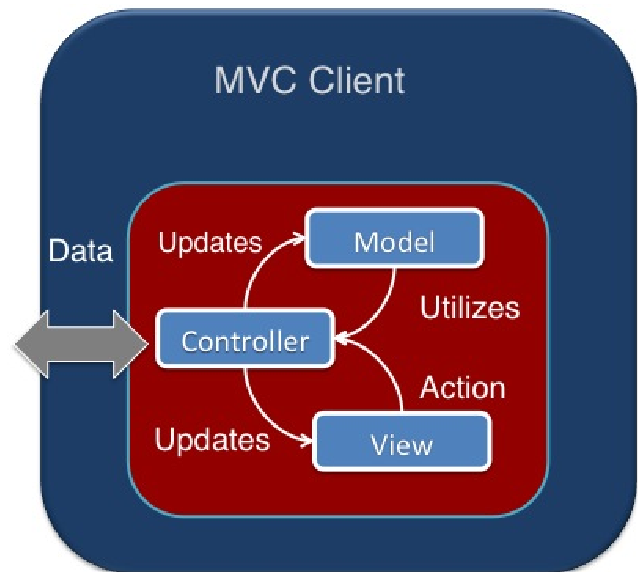

Figure 6: Anatomy of an MVC Client-Side Application

are usually nested we will utilize a tree structure in order to represent them. In Figure 5, we display a tree representation of the Model; as we see each delivery truck contains the current coordinates of a truck, along with all the other information contained in the HTML table (as shown in Figure 4), such as the VIN number, name of the driver and so on. The View in this application is the entire dashboard shown in Figure 4, it consists of the HTML table and the two components (Google Maps and Progress Bar). The Controller is the piece of code that interacts with the back-end server to fetch information about updates on the model and it also manually propagates the corresponding changes to the view.

In Figure 6 we preview the internal structure of an MVC client-side application. As we observe the Controller is typically the synthetic link of the application; it interacts with the back-end by transmitting and receiving essential information, it uses the received data to update the Model of the application, it utilizes the Model to generate or update the

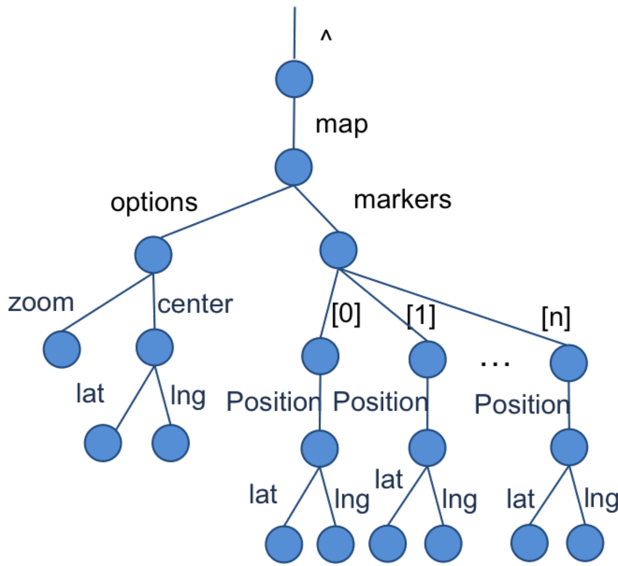

Figure 7: ViewModel of Google Maps Component

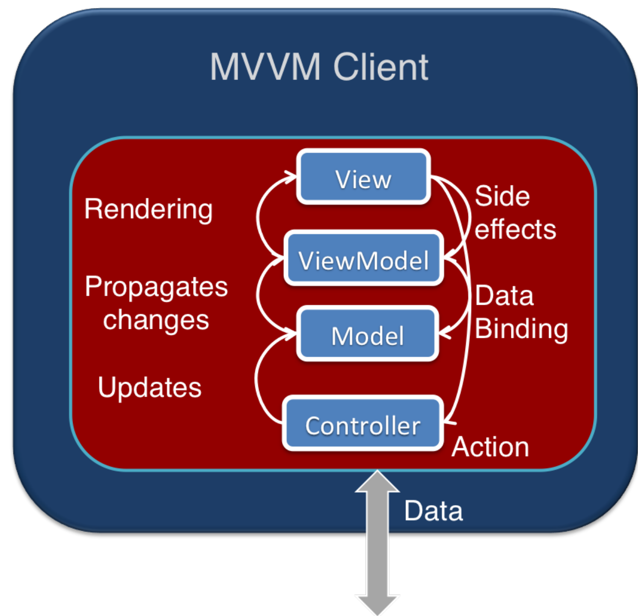

Figure 8: Anatomy of an MVVM Client-Side Application

view and it contains the callback functions (actions) that will be executed when the user triggers UI events.

Disadvantages of MVC applications

While this design pattern assists in the implementation of more organized and maintainable applications, the fact that the Controller is responsible for managing both the Model and the View definitely opposes to the concept of Separation of Concerns. Additionally, despite the fact that application developers that follow this pattern, manage to create a seamless user experience, (as it enables the developer to apply incremental updates to the view, thus making the interface very responsive), at the same time these incremental updates have to be performed manually by the application developer using imperative code. Most of the times, this results in writing multiple lines of boiler-plate code even for simple tasks.

2.3.2 *Model-View-ViewModel (MVVM) Design Pattern*

In order to resolve the aforementioned issues, a new set of frameworks were introduced with the intent to simplify the process of application development by supporting the MVVM design pattern. Since this pattern is relatively new, there are very few sources that manage to provide clear definitions of what constitutes an MVVM framework and what its main characteristics are, thus causing many misconceptions in the web community. One of the contributions of this survey is to provide these definitions and address the respective misconceptions.

Two of the main characteristics of the MVVM design pattern is that it enforces a stricter Separation of Concerns, while at the same time limiting the size of the imperative code required for a web application. The building blocks of this pattern contain: A Model, a View and a ViewModel. The first two are the same entities we covered in the description of the MVC design pattern. The ViewModel is an abstract representation of the View, which contains the JavaScript objects required by the respective component APIs that are responsible for the instantiation of the visual layer. Particularly, in the running example presented earlier, the ViewModel of the Google Maps component is the JavaScript object shown in Figure 7. Notice that the ViewModel is essentially generated by utilizing attributes that already exist on the Model (which shown in Figure 5). For parts of the page that correspond to simple HTML content, the ViewModel is typically a data structure that is introduced and maintained by the respective MVVM frameworks and it is typically isomorphic to the DOM-Tree of the page.

In most MVVM frameworks the mapping between the Model and the ViewModel, occurs declaratively by utilizing a template language that is introduced by the respective frameworks. Specifically, by utilizing the template language the application developer binds a particular part of the Model to the respective part of the ViewModel, thus creating the premise that enables frameworks to automatically perform incremental maintenance of the application's View when the Model that is bound to it gets updated. This significantly limits the lines of code the application developer needs to write, since he no longer has to write and maintain any imperative logic in order to update the View.

One of the misconceptions that exist in the web community about this design pattern, is that it does not contain a Controller. Most MVVM frameworks still utilize a Controller but the logic that it contains typically does not mutate the View of an application (in contrary to the Controller of an MVC Framework). Instead, the Controller is only responsible for specifying the actions and side-effects that execute when the user triggers events as he/she interacts with the View. Additionally, in most MVVM frameworks the Controller is also responsible for interacting with the back-end in order to receive and transmit data that synchronize the client-side application state (Model) with the server-side state. In Figure 8 we show the internal structure of an MVVM client-side framework.

```
1  /* ... Additional logic ... */
2  $http.get('http://forward.ucsd.edu/delivery_truck_service')
3  .then(function(result) {
4     $scope.delivery_trucks = result.data;
5  });
6  /* ... */
```

Figure 9: Part of the Controller used in running example

3. EXISTING MVVM FRAMEWORKS

In this section we examine existing frameworks that follow the MVVM design pattern as defined in Section 2.3.2. All these frameworks can be used for the development of client-side web applications but can also be packaged as hybrid applications with the intention to operate on mobile devices. For each web framework we will provide the complexity numbers that show how efficiently each employed mechanism reflects the changes to the View when changes are applied to the Model. This is an important factor as it shows how viable a framework is for mobile development (since the resources of most mobile devices are quite limited) or for applications that are designed to display big data visualizations.

3.1 AngularJS

Perhaps the most widely used MVVM web framework currently is AngularJS [24]. Angular is mainly supported and maintained by Google, but since it is an open source framework [25] it also has a very rich community of contributors, which ranges from individuals to big corporations. From the perspective of the application developer AngularJS requires very little boilerplate code for most simple applications. Particularly the developer's only responsibility is to specify the Model of the application and create the bindings between the Model and the View by utilizing the template language. When the state of the application is modified, Angular is able to infer how the respective view will be affected, and it automatically applies the appropriate changes to it.

Angular's template language consists of HTML tags that may contain additional Angular specific attributes which are utilized for binding data to the corresponding parts of the View. Other than HTML, the template language can also contain custom tags that instantiate reusable units, namely **Angular Directives**. These modules wrap existing JavaScript visual layer components such as Google Maps[22], HighCharts[17] and more, in a way that favors reusability and code minimalism. When importing such directives the application developer is able to specify the state of the visual layer using declarative logic. This greatly reduces the imperative code that has to be written and maintained, since the developer is no longer responsible for explicitly implementing code that reflects every potential modification of the Model to the View. The Model of the application however, is specified by using JavaScript code which, despite the fact that it may contain functional expressions, it is mostly imperative.

More specifically, in order for a variable v to be used within Angular's template language the application developer has to utilize a Controller to instantiate the contents of the aforementioned variable and then attach it to the *scope* object; the *scope* object is a crucial part of AngularJS internal architecture as we will describe later in this section.

```
1  <html>
2     <!-- ... imports and other irrelevant
3             parts of the template ... -->
4     <div ng-app="truck_delivery" ng-controller="delivery_ctrl">
5       <!-- ... other HTML tags ... -->
6       <div id="map_container">
7        <ui-gmap-google-map
8        center="map.center" zoom="map.zoom" bounds="map.bounds">
9          <ui-gmap-marker ng-repeat="truck in delivery_trucks"
10         idKey="truck.truck_key"
11         coords="truck.coords">
12         </ui-gmap-marker>
13        </ui-gmap-google-map>
14      </div>
15    </div>
16    <div>
17      <table>
18        <tr> <!-- ... column labels ... --> </tr>
19        <tr ng-repeat="truck in delivery_trucks">
20             <td> {{truck.VIN}} /td>
21             <td> {{truck.driver}} /td>
22             <td> {{truck.shift_start_time}} /td>
23             <td> {{truck.avg_speed}} /td>
24             <td>
25               <progressbar type="Circle"
26               strokeWidth="10" trailWidth= "1"
27               easing= "easeInOut" fromColor="#FC5B3F"
28               fromWidth="1" toColor="#6FD57F" toWidth="10"
29               numerator = "truck.delivered_items"
30               denominator="truck.total_items"
31               ></progressbar>
32             </td>
33        </tr>
34      </table>
35    </div>
36     <!-- ... rest of template ... -->
37 </html>
```

Figure 10: Angular Template `Delivery-Trucks`

The variables that the application developer attaches to the scope contain Plain Old JavaScript Objects (POJO), which simplifies the Model definition since the application developer is not required to extend any framework specific classes for this purpose, which is the case for other frameworks, as we will describe in the next sections. Lastly, Angular provides two more toolsets, namely Services and Factories, that mostly assist in data transfer to and from local or remote databases or web-services. After the required datasets have been collected by the client, they can be used to instantiate or update the Model of the application.

In Figures 9 and 10, we show a small snippet of the controller and the template language that is used to generate the running example shown in Figure 4. The snippet in Figure 9 shows how an application developer can utilize the *http* service to perform an asynchronous HTTP GET request to a remote server, retrieve the result, assign it to the variable *delivery_trucks* and then attach this variable to the scope object. In Figure 10 we show a snippet of the template that generates the majority of the view of the running example. In line 4 of the template, we specify the name of the application and the controller by utilizing the Angular specific attributes “ng-app” and “ng-controller”. In lines 7-13 we use a Google Maps custom directive to generate the map shown in Figure 4; in lines 25-31 we use a ProgressBar directive that manages the bars that appear in the last column of the HTML table. Notice that in these two cases other than the custom tags used to instantiate the Google-Maps and ProgressBar directives, we also use special attribute names to create bindings between some parts of the Model (for

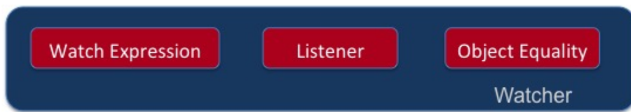

Figure 11: AngularJS Watcher - Building Blocks

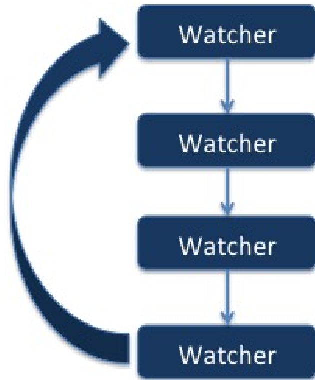

Figure 12: AngularJS - Digest Cycle

instance, the *coords* attribute of the $delivery - trucks$ variable) and the View (for instance the coordinates of the respective marker). In lines 9 and 19, "ng-repeat" iterates over the entire *delivery_trucks* array and at each iteration it initializes an alias , namely *truck*, for each delivery truck, which can be used to declare bindings between each individual truck and the respective replicated instance of the directive. The directive (and every child directive) is replicated n times, with n being the total number of elements the *delivery_trucks* contains.

AngularJS Watchers - Dirty Checking.

Every time the application developer binds a single variable or expression to the template, Angular attaches a *Watcher* (Figure 11) to the *scope* object. Watchers are essentially trigger definitions that execute when a mutation occurs on the part of the Model they watch. A Watcher contains the expression that is being watched, namely: *WatchExpression*, the *Listener* and the *ObjectEquality* variable. The WatchExpression can be a function call, an arithmetic operation or a reference to some part of the Model. When the current result of the WatchExpression changes, Angular triggers the Listener, which is a callback function that is responsible for updating the respective part of the view. In order for the framework to decide if the result of the WatchExpression changed, it needs to compare the current state of the returned object with the previous state, this process is called dirty-checking. The kind of comparison that will take place is defined by ObjectEquality; in general two types of comparisons are allowed: deep and shallow; in a shallow comparison, Angular will trigger the Listener function when the WatchExpression returns a completely new object (The reference to the watched object changes), while in a deep comparison Angular will iterate over all the children of the watched object and it will trigger the Listener function if one of these nodes is different from the respective node of the previous state of the object. In order to perform dirty-checking, Angular needs to store both the pre-state and the

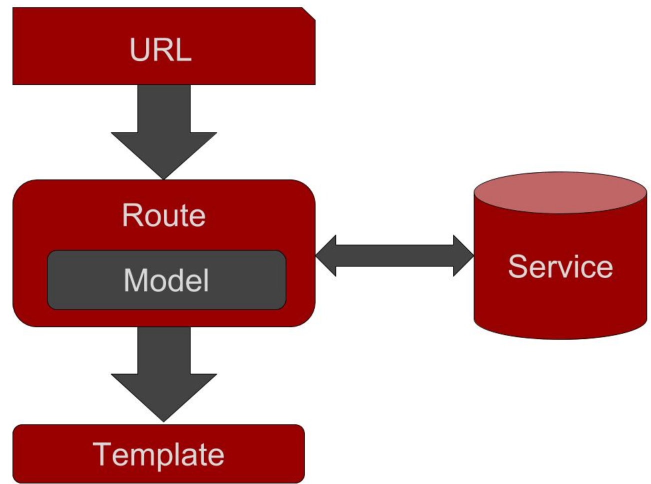

Figure 13: Ember's Programming model

```
1 Router.map(function() {
2   this.route('index',  path: '/' );
3   this.route('post', { path: '/post/:post_id' });
4 });
```

Figure 14: EmberJS - Route Example

current state of the watched object, therefore the bigger the watched object is, the higher the memory and the processing footprint will be.

The Digest Cycle.

A crucial part of the internal architecture of Angular is the Digest Cycle (shown in Figure 12). This algorithm in conjunction with dirty-checking performed for each watcher is responsible for propagating to the View changes that occur on the Model. The digest cycle is initiated when an event triggers an action that belongs to Angular's *scope*. During this process, Angular iterates over all the watchers that exist in the scope and performs a comparison of the old state of the watched expression with the current one. If a difference is found, then the corresponding Listener function is triggered. If some Listener function modifies the Model, then the digest cycle will get initiated once again. This iteration will continue until either the Model stabilizes or the digest cycle executes ten times, after which Angular throws an exception and the Angular application is killed.

Complexity of Digest Cycle

The complexity of the digest cycle is proportional to the total numbers of watchers that have been declared. Additionally, if deep watchers are used, then the complexity becomes even worse since for each deep watcher the entire watched object will have to be traversed so that nested values will be inspected for changes. If w is the number of watchers that have been declared and d is the size of the watched object, with d being equal to 1 in the case of shallow watches, then the algorithm that performs updates to the view, given changes on the Model, is $O(wd)$.

3.2 EmberJS

```
Person = Ember.Object.extend({
  firstName: null,
  lastName: null,

  fullName: Ember.computed('firstName', 'lastName', {
    get(key) {
      return `$this.get('firstName') ${this.get('lastName')}`;
    },
  })
});

var Yannis = Person.create();
Yannis.set('firstName', 'Yannis');
Yannis.set('lastName', 'Papakonstantinou);
Yannis.get('fullName'); // Yannis Papakonstantinou
```

Figure 15: EmberJS - Model Definition

Another framework that follows in the footsteps of Angular is EmberJS [26]. Similarly to every other MVVM framework, Ember tries to achieve code minimalism by freeing the application developer from unnecessary boilerplate code and by utilizing declarative code when appropriate. Ember's ecosystem comprises Routes, Models, Templates and Services. As shown in Figure 13, Ember's life-cycle starts when the user navigates to a URL that is bound to a particular Route. When the Route receives the event, it constructs the Model and calls the appropriate renderer that will generate the visual layer of the application. The Route can also utilize the appropriate reusable Services to receive essential data for the construction of the Model (application state).

EmberJS uses regular expressions to bind a Route to a particular URL. These expressions can also treat parts of the URL as variables, which enables message passing to and from other pages. In Figure 14, we show a snippet that illustrates how a Route gets bound to a particular URL. More specifically, in line 2, we bind the root of the application to the *index* route, while in line 3 we bind all the URLs that match the pattern: "*/post/∗*" to the *post* route. This expression automatically creates a variable *post_id* that is accessible from the respective route. The value of this variable is equal to the URL step found immediately after the "*/post/*" step, for instance if the user navigates to the URL "*/post/*5" the variable *post_id* will contain the value 5.

As we mentioned earlier, Routes are responsible for constructing the Model of the application. Unlike Angular's Model which comprises Plain Old JavaScript Objects, Ember requires the extension of its internal Model objects. This negatively impacts the user-friendliness of the framework, since it forces application developers to familiarize themselves with the internal data structures used by Ember. More specifically, developers are required to use Ember's API in order to set and retrieve values to/from the Model, which steepens the learning curve of the framework.

Ember's Model class can contain *observable* and *computed properties*. An observable is typically a part of the Model that when updated, will trigger further updates either on other parts of the Model or the ViewModel of an application. Ember enables the declaration of callback functions that are triggered when the associated observable object is updated. When declaring such callback functions, a developer can use imperative logic within its body in order to specify the side-effects that will take place when a mutation is observed. If one part of the Model needs to be updated every time some another part is mutated, it can be defined as a computed property. By doing this, Ember "hardwires" the two parts of the Model and allows automatic updates of computed properties when the observable base properties they depend on are updated.

In Figure 15, we illustrate how the Model is specified in an Ember application. As we notice in lines 2 and 3 we specify the attributes included in the Person object. In lines 5-9 we specify a computed property, namely $fullName$ that depends on the $firstName$ and $lastName$ properties. By explicitly specifying the base-variables of a computed property, the application developer dictates that this property will get reevaluated when the respective base-variables are modified. When the developer utilizes a setter to modify an observed property (as shown in lines 13 and 14) Ember is able to propagate the respective changes to the computed property.

Lastly, EmberJS does not have its own custom template language, it instead utilizes a third-party template language called HandlebarsJS [27] to generate the templates. Additionally, Ember enables component wrapping by providing expendable Components (that are very similar to Angular's Directives). When importing such components the application developer can declaratively pass the Model that will be utilized by the component for the generation (or incremental updating) of the component specific view. In order to assist in data transfer between local and remote databases and web services Ember provides the Service class, which is comparable to Angular's Factories and Services.

Incremental Updates using Accessors

As mentioned earlier, Ember requires the use of its internal object class to represent the Model of the application. Particularly, the application developer uses getters and setters when he wishes to retrieve or update the value of some Model variable (as shown in Figure 15, in lines 12-15). If some part of the Model is used in a template, Ember implicitly declares this part of the Model as an observable. When a mutation on that part of the Model is observed, the respective part of the View will be reevaluated and rerendered automatically.

Observing mutations in Ember is a fairly simple task, since a setter has to be explicitly invoked by the developer. As a result, the algorithm that propagates changes in Ember does not need to identify which part of the Model was mutated. Instead, when a setter is called, Ember can simply propagate this change to the respective getters, that are associated with the mutated variable, and cause them to reevaluate the variables that depend on it. This significantly enables Ember to keep the Model and View in sync at all times, without wasting resources by iterating over all the observed variables (which is the case with Angular). More specifically, given an observable variable v and a set of variables s that depend on v, the complexity of the algorithm that propagates changes to each variable in s given d updates on a is $O(d|s|)$, with $|s|$ being equal to the number of variables in s.

Despite the relatively efficient algorithm that achieves change propagation, the fact that Ember requires the extension of its own Model classes and explicit calls to setters and getters, significantly steepens the learning curve of the framework. Furthermore, application developers need to be aware of which parts of the Model will get updated during the life-cycle of the application and explicitly trigger these

updates. Lastly, using getters and setters can be very dysfunctional in cases when the objects used in an application are heavily nested. For those reasons, Ember appears to be less "developer-friendly" than Angular.

3.3 KnockoutJS

Knockout[28], is another MVVM framework that was introduced in October 2010 which makes it the oldest framework in this comparison. Despite its age, Knockout has its fair share of modern features, such as declarative templates, data-bindings and automatic updates on the view given changes on the respective bound Model, which makes it fulfill all the requirements that classify it as an MVVM framework. Additionally, Knockout is very lightweight (54kb when minified, which reduces to 20kb when using HTTP compression [29]) in comparison to all the other frameworks we have described so far, which is explained by the fact that it is missing some features found in other frameworks. Knockout's ecosystem mainly consists of Models and Templates, which means that it lacks all the extra utility components that the rest of the frameworks contain, such as Routes, Services, Factories and so on. For this reason many online sources consider Knockout to be a lightweight MVVM library [30] instead of a framework.

Knockout's life-cycle begins when the user loads the HTML page that contains the JavaScript files that instantiate the Model. After the Model has been instantiated Knockout parses the template that is contained in the HTML page and dynamically generates the View. Knockout's internals have several similarities to Ember. Particularly, both frameworks require the application developer to extend internal object classes, in order to define the Model and both of them share the concepts of observable properties and computed values. Dependency tracking also works in the same way in these two frameworks, if a computed value depends on some observed variable, a subscriber is declared; when the observed variable changes all subscribers are triggered and the corresponding computed value gets updated. The same mechanism is also used to update parts of the View that depend on observed variables.

Another feature that Knockout supports is Components. Similarly to Angular and Ember, the developer can choose to implement reusable Components and introduce them as custom tags into Knockout's template language. The main difference between Knockout's and Ember's Components is the fact that the former cannot contain imperative logic. This makes Knockout Components suitable for generating reusable HTML widgets that can be introduced in multiple parts of an application, but unfitting for wrapping existing visual libraries such as Google Maps in a reusable Component. The reason is that, visual libraries require the use of imperative logic that utilizes the respective library API in order to instantiate or update a View while an HTML widget can be generated by utilizing Knockout's declarative template language. That being said, if the developer wishes to include a Component he/she can either write imperative code outside of Knockout's scope or use Knockout's Custom Bindings module. Custom Bindings are special Component-like modules that allow the usage of imperative code, thus enabling the use of external library APIs. The way Custom Bindings interact with the rest of the application however does not favor reusability, therefore the application developer has to implement a separate Custom Binding every time he wishes to use a visualization library.

4. WEB COMPONENT FRAMEWORKS

Since the Separation of Concerns is a crucial aspect of web frameworks we also surveyed libraries that are used for the implementation of Custom Web Components [31, 32]. These Components are an attempt to bring component-based software engineering [33] in the Web, by providing crucial characteristics such as encapsulation, reusability and extensibility to the web developer. Web Components have several similarities to the respective Component/Directive modules of MVVM frameworks, however the biggest differences between the two lie in the way these components are structured internally and the way they interact with the part of the application that crosses the framework's scope.

In general, Web Components are self contained modules with a pre-defined functionality, which makes them less parameterizable than the respective Components/Directives of MVVM frameworks. A Web Component can be used in different parts of a single page (or in different pages) by simply being imported and injected to the page. The application developer, typically, it not responsible for performing any additional operations in order to utilize a 3rd-party Component (such as generating the Component state, or map the state of the Component to the respective Component attributes), which makes Web Components a good "plug-n-play" solution for web applications. Generally, MVVM frameworks are equipped with features that make them better candidates for building larger applications, while Web Components are intended for developing widgets, which can be used as small self-contained parts in a larger application.

A very crucial feature, that most Web Components support, which favors encapsulation and reusability, is the **Shadow DOM**[34]. This feature assists in encapsulating the DOM tree that belongs to a Component, thus making it independent from the parent DOM nodes. One of the biggest issues, application developers have to deal with, when specifying the view of a particular page, is its styling. In order to describe the styling of a particular part of the page, they usually have to write style-sheet rules within a CSS file. When this CSS file is loaded in a particular page, however, other parts of the page may be affected by the newly introduced rules. Shadow DOM limits the scope of these rules and prohibits them from being applied to the DOM elements that belong to a Component; thus making the latter completely independent both from the page that it belongs to and from other Components.

4.1 Polymer

Perhaps the most typical example of a Component library is PolymerJS [35], which is developed and maintained by Google. In Polymer, developers can create their own reusable Components in order to integrate them into their applications or publish them on the Internet, so that other developers can utilize them. There is also a big list of reusable Components offered by the official website that ranges from Components responsible for introducing generic UI elements to the view (such as layout components that generate forms, tables and so on) to Components that only introduce logic (for instance Components that are responsible for performing requests to back-end services, adding push notification and bluetooth capabilities and so on). Such components when combined appropriately they can effec-

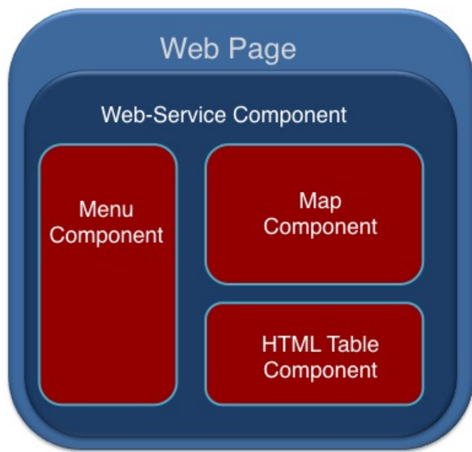

Figure 16: Component Based Architecture

```
CustomElement = Polymer({
  is: 'custom-element',

  created: function() {
    this.textContent = 'My Custom Element!';
  }
});

```

Figure 17: Custom Polymer Element

tively generate modern applications in a modular manner. For instance, in Figure 16 we show how different Components can be composed together in order to generate a single page application. This application comprises 4 different components; the "Web-Service Component" is responsible for accessing a remote web-service in order to retrieve the data that will be used in this application. After the data have been retrieved they are passed on to the three child components: "Menu Component", "Map Component" and "HTML Table Component" in order to get visualized.

When using such 3rd-party Components, the application developer is able to create a Web application by strictly writing declarative code, since he/she essentially manages both the application state and the View by utilizing custom tags that represent the respective Components. Despite the fact that Components are essentially building blocks of bigger applications, most of the times they lack more sophisticated features that are essential in most real-world applications; in such cases imperative code cannot be avoided. The reason behind this, is that if some feature is not offered "out-of-the-box" by some Component, the application developer will have to implement a custom Component himself and the only way to accomplish this is by writing imperative code.

More specifically, in order to implement a custom Component, the developer has to extend a Polymer class and override callback functions and attributes that are contained in this class. By doing so, the developer is able to introduce logic that defines the functionality of the Component. In figure 17, we show a simple Polymer class that overrides the essential functions and attributes required for a component to be defined. In line 2, we specify the name of the Component and in line 4 we override the *created* callback function that will be executed when the custom tag: <custom-element></custom-element> is inserted in a page; when this custom tag is parsed and evaluated the text: "My Custom Element!" will be added to the View. By overriding callback functions (such as: *created*, shown in Figure 17), the component developer is able to define more advanced logic that is executed at different stages of the life-cycle of a component.

Polymer allows data exchange between a host component (parent) and a guest (child) in the form of data-binding. In order for a Component to allow this behavior, the Component developer has to explicitly enable this feature by overriding the appropriate functions. Other than data binding, Polymer allows the use of computed attributes and observables (which is also supported by EmberJS and KnockoutJS as we mentioned in the respective sections). Similarly to Ember and Knockout, change propagation in Polymer is initiated when a setter is explicitly called within a Component. When this occurs, Polymer fires an event which is then propagated to the descendants and ancestors of the current Component. If some Component is "listening" for changes on the mutated part of the Model it will be notified and it will be responsible for updating the respective part of the View. The complexity of this algorithm is $O(h)$, with h being the number of consecutive Components that will be notified when a mutation is triggered.

4.2 ReactJS

Another Component library that is very widely used in modern applications is ReactJS. This open source library is developed and maintained by Facebook, but just like every other successful open source framework, it also has a big community of contributors. Similarly to Polymer, React enables the development of reusable Components that can be used as building blocks of bigger applications. The developer of React Components extends a React class and overrides the respective callback functions in order to specify the state and the View of the Component.

More specifically, in order to specify the View, the developer has to add the respective logic within the *render* function of the React class. If the Component developer wishes to use a visualization library (such as Google Maps or HighCharts), the *render* function will contain imperative code that utilizes the respective API provided by the visualization library. Otherwise, if he/she wishes to define an HTML View, he/she can either use imperative logic or React's template language, namely JSX to do so declaratively. Specifically, Figures 18 and 19 show how a React Component can be implemented imperatively and declaratively. In lines 2-10 in Figure 18, we override React's *render* function and create a *ul* DOM element with class name "customClass". This element contains a nested element *li* with class name "customList" and value: "My Custom Element!". In Figure 19, and specifically in lines 4-6 we generate the same component declaratively using the JSX template. JSX templates can contain both standard HTML and custom tags (which declaratively instantiate other custom React components); while they also allow the use of expressions that are evaluated during runtime such as arithmetic expressions, function calls and binds to parts of a Component's Model. Lastly in lines 12-15, in Figure 18 and in lines 10-12, in Figure 19, we attach the newly created React Component to the DOM tree.

```
var Component = React.createClass({displayName: 'Component',
  render: function() {
    var child = React.createElement('li',
      className: "customList", "My Custom Element!"
    );
    var rootElement = React.createElement('ul',
      className: "customClass", child
    );
    return rootElement;
  }
});
ReactDOM.render(
 React.createElement(Component, null),
   document.getElementById('content')
);
```

Figure 18: Custom ReactJS Component specified imperatively

```
var Component = React.createClass({
  render: function() {
    return (
      <ul className="customClass">
        <li className="customList"> My Custom Element! </li>
      </ul>
    );
  }
});
ReactDOM.render(<CommentBox />,
  document.getElementById('content')
);
```

Figure 19: Custom ReactJS Component specified in JSX

Virtual DOM

In general, DOM operations are particularly expensive; more specifically, the complexity of DOM operations is proportional to the size of the DOM subtrees that will be re-rendered. Most modern application frameworks do not always apply the minimum DOM manipulations necessary in order to update the View, which hinders performance. ReactJS employs mechanisms that are able to minimize the DOM operations required to update the View, thus achieving significant performance increase over competing frameworks. More specifically, when the Component developer specifies a View using JSX or imperative code, React internally instantiates an isomorphic representation of the DOM Tree; this structure is called Virtual DOM. When the underlying Model of a component is mutated, the application developer is required to explicitly trigger the action-page cycle of a component by invoking the *setState*() function. During this cycle, React generates a new instance of the Virtual DOM (post-state) and then proceeds by executing a "diff-ing" algorithm that attempts to identify parts of the two instances (pre-state and post-state) that have changed; these parts are called patches. When this procedure is completed, React performs the minimum possible renderer calls that apply these patches to the DOM Tree, thus efficiently updating the View of the application (as shown in Figure 20).

While this approach undoubtedly limits the rendering cost of a View, it also has some caveats that, depending on the use case, could result in performance penalties. The complexity of identifying changes, in React, is proportional to the entire ViewModel (Virtual DOM) of a Component, since

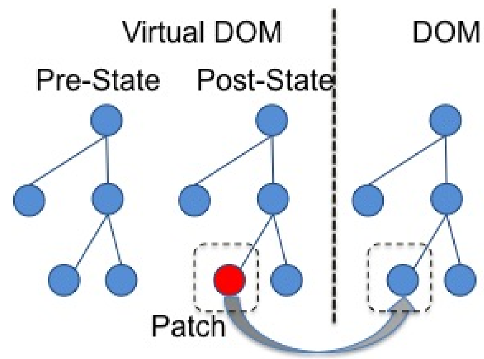

Figure 20: Virtual DOM Diff-ing

the entire ViewModel has to be reconstructed and compared with its old state (which has to be kept in memory) every-time the action-page cycle is triggered. In some real-life scenarios, the component's ViewModel can be exceptionally big, while the number of elements that are subject to changes is very small; in such cases frameworks that declare observers/watchers would actually be more efficient since the cost of re-evaluating the entire ViewModel is definitely bigger than re-rendering small parts of the View that haven't changed. Additionally, since the old state of the Virtual DOM has to be cached in memory, a Component with an exceptionally big ViewModel could cause the application to crash if it runs out of memory. Another caveat, that is mostly related to the way this approach is implemented in React, is that it only works if the application developer uses, directly (with imperative code) or indirectly (by utilizing JSX templates) the Virtual DOM. The Virtual DOM however can only be used to represent parts of the View that will be translated to HTML elements, therefore this approach does not work if the foresaid Component is used to wrap a 3rd-party visualization library. In such cases, the application developer has to introduce his own internal mechanisms to achieve more efficient rerendering, which typically leads to complex imperative logic.

4.3 MithrilJS

The last Component library that will be included in this comparison is MithrilJS[36]. This is a particularly small framework (7.8kB when zipped) that has no dependencies on other libraries. Mithril has a lot of similarities with React; particularly both these libraries utilize a "diff-ing" algorithm that uses the Virtual DOM tree to accomplish efficient rendering and they both use somewhat similar conventions when implementing a Component. Specifically, they both require from the developer to override a particular set of functions that are executed during various phases of the lifecycle of a Component.

One minor difference between the two is that Mithril does not provide any base classes that need to be extended, in order to specify a given Component. The advantage of this approach is that child classes do not inherit all the utility methods and properties of the parent, which in JavaScript, depending on the way inheritance is implemented, could potentially lead to increased memory footprint, since child class instances may carry clones of all the functions that are defined in the parent class. This however will not be the case if Prototypical Inheritance [37] is used instead. Another differ-

ence is that Mithril does not support declarative templates for specifying the View of a component. However, there are 3rd-party libraries (such as MSX [38]) that allow the use of declarative logic for that purpose.

5. FORWARD

FORWARD is a Web Application framework designed to enable rapid development of data-driven, information dense applications. It employs techniques that enable efficient propagation of changes from the server all the way to the View, thus making it the ideal framework both for applications that require dense visualizations that are updated frequently and for larger commercial applications that consider performance to be a high priority. FORWARD is a fully-fledged MVVM framework that supports declarative templates, data binding, unit wrapping and a Unified Application State (UAS) that achieves encapsulation over multiple data sources without introducing any additional overhead.

5.1 Incremental View Maintenance (IVM)

FORWARD leverages the extended research that has been conducted in the area of Incremental View Maintenance from the database community ([39, 40, 41] and more) to power real-life full-stack and client-side reactive applications. In the database world, materialized views are utilized to speed up query evaluation and execution by caching the result of commonly requested queries. A typical query may require access to different database tables or even tables that are hosted in databases that reside in completely different physical locations, which significantly limits the performance of query execution. On top of that, a query may require the execution of aggregate functions that demand the traversal of entire tables, which can be very inefficient especially in cases where such queries are performed frequently. For these reasons, the database community introduced the concept of materialized views. A materialized view is essentially a database table that caches the result of a query (view definition), so that it is easily accessible when the same query is run again in the future (as shown in Figure 21), thus avoiding the full recomputation of the result.

One caveat with this approach is that such materialized views can soon become outdated as new datasets are added to the base tables. A valid solution to this problem is to frequently recompute the result of the view definition, so that it remains up-to-date at all times. This approach however, essentially recreates the problem that materialized views are attempting to resolve in the first place, which is the prevention of the full reevaluation and reexecution of a query. Instead, the database community introduced IVM techniques that can be used to incrementally update the materialized view as updates are applied to the base tables it depends on. A typical IVM algorithm takes as input various types of diff definitions and utilizes a set of IVM rules that dictate how to efficiently update a materialized view. Most IVM implementations require at least the following diff definitions in order to describe the different kinds of updates that can occur in a base table.

- $\triangle^{\text{insert}}$, describes the insertion of a set of records to a base table
- $\triangle^{\text{delete}}$, describes the deletion of a set of records from a base table

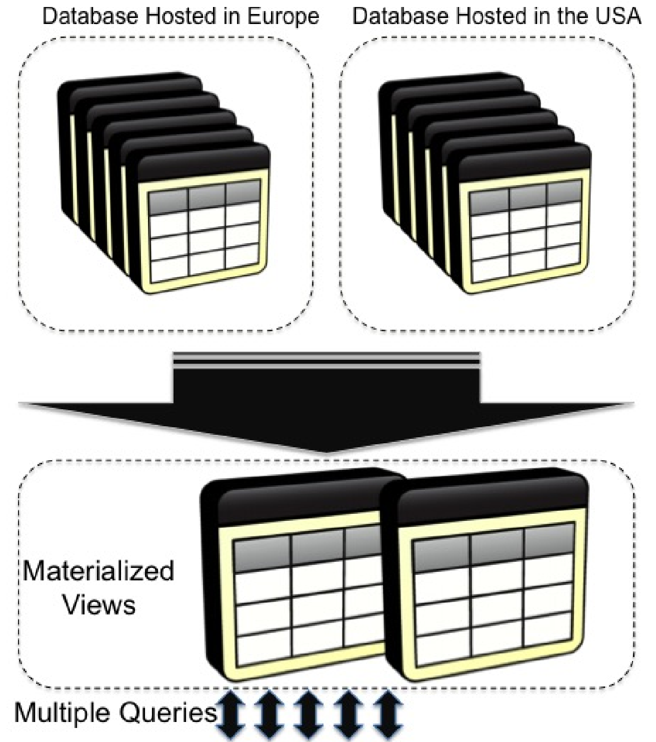

Figure 21: Materialized Views

- $\triangle^{\text{update}}$, describes the update of a set of records that belong to a base table

FORWARD is a novel MVVM application framework that applies this technique to the Web by injecting diff propagation techniques into the different modules a typical MVVM application consists of. By doing so, FORWARD essentially treats the Model, the ViewModel and the View of an application as materialized views, thus avoiding the full reevaluation of their state when changes are applied to the datasets they depend on. This leads to more efficient applications without compromising the ease of use.

5.2 FORWARD's Programming Model

FORWARD's programming model (shown in Figure 22) comprises a Template, a Virtual Database (VDB), and Actions. The VDB object essentially describes the Model that is utilized by the application. Declarative Templates are used to bind parts of the Model to the ViewModel (also called Template Instance), thus generating the View (Visual Page Instance). Lastly, when events are triggered, they invoke the execution of the actions they are associated with, which can further mutate the application state, cause side-effects or trigger the evaluation of a different template.

FORWARD, like most other MVVM frameworks, also supports the creation of reusable visual components/directives that are used to wrap 3rd-party libraries. These Visual Units can be utilized by application developers within a template to declaratively specify a visual layer that contains visualization components (such as charts and maps). A Visual Unit contains a set of *renderers* that are able to apply changes to the View given changes to the Model. Since, in FORWARD diffs are first class citizens that are

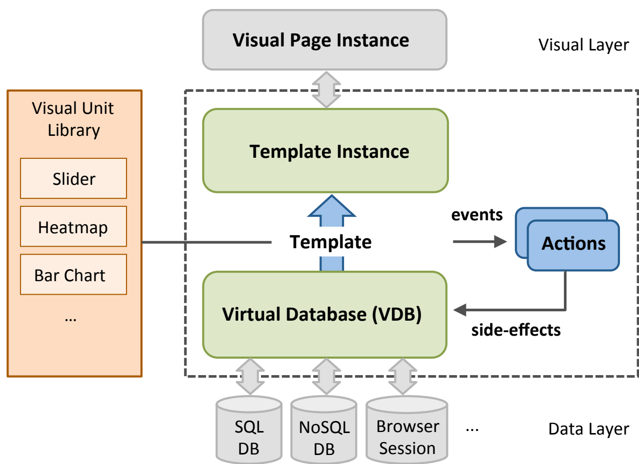

Figure 22: Programming Model for the Novice Database-oriented Developer

used throughout the framework, a renderer utilizes the information contained within a diff to reflect the appropriate changes to the respective part of the View. Each renderer is declared by specifying a *diff signature*; signatures define the rules used to identify the renderer that is capable of reflecting the changes that are described within a given diff.

As we observe, FORWARD's programming model seems to be on par with the respective programming models of most MVVM frameworks we have described. This is because FORWARD, despite the advanced IVM mechanisms that it employs, it still follows the same principles as other frameworks, which makes its conceptual model easy to understand and utilize. Despite those similarities FORWARD has some unique features that greatly simplify the development of robust applications. Particularly, FORWARD is the only framework that enables the use of declarative logic for specifying the Model of an application. The application developer is able to inject, a query that accesses remote databases or web services, directly into the template and assign its result to a VDB variable. Such variables can later be bound to the template in order to generate the View. The query language that is used for defining VDB variables, namely SQL++, is an extension of the SQL language that is designed to support queries on relational and semi-structured data.

In Figure 23 we show the FORWARD template that is used to generate the View shown in Figure 4. The declarative specification of the application's Model is shown in lines 5-14. This sample assumes that FORWARD is utilized as a full stack framework, and the base tables: *delivery_trucks_table*, *product_delivery_truck_relation* and *products* have been defined on the server-side part of FORWARD. In lines 17-35 we use a Google Maps unit to generate the map component shown in Figure 4 and in lines 39-65 we instantiate the HTML table shown bellow the map. Lastly, in lines 48-61 we instantiate the progress that is visible in each row of the HTML table.

5.3 Interaction with Remote Services and Database Systems

FORWARD has the ability to be utilized both as a full-stack and a client-side framework. In the first case, FORWARD'S Unified Application State (UAS) [2] is able to in-

```
<% template delivery-trucks (product_name) %>
 <% import functions %>
 <% import actions %>

 <% refresh delivery_trucks =
     SELECT latitude, longitude, VIN, driver,
            shift_start_time, avg_speed,
            delivered_items, total_items
     FROM   delivery_trucks_table dtt,
            product_delivery_truck_relation r,
            products p
     WHERE  p.name = <% print product_name %>
            AND dtt.id = r.t_id AND p.id = r.p_id
 %>
 <% html %>
   <div>
     <% unit Google-Maps %>
     {
       options : {
         zoom: 10,
         center: {
           lat: -25.363882,
           lng : 131.044922
         },
       },
       markers : [
         <% for truck in delivery_trucks %>
         {
           position : {
             lat : <% print truck.latitude %>,
             lng : <% print truck.longitude %>
           }
         }
       ]
     }
     <% end unit %>
   </div>
   <div>
     <table>
       <tr> <!-- ... column labels ... --> </tr>
       <% for truck in delivery-trucks %>
          <tr>
            <td> <% print truck.VIN %> /td>
            <td> <% print truck.driver %> /td>
            <td> <% print truck.shift_start_time %> /td>
            <td> <% print truck.avg_speed %> /td>
            <td>
               <% unit ProgressBar %>
               {
                 type = 'Circle',
                 strokeWidth: 10,
                 trailWidth: 1,
                 easing: 'easeInOut',
                 from: { color: '#FC5B3F', width: 1 },
                 to: { color: '#6FD57F', width: 10 },
                 value : {
                  numerator :<% print truck.delivered_items %>
                  denominator : <% print truck.total_items %>
                 }
                }
                <% end unit %>
            </td>
          </tr>
         <% end for %>
     </table>
   </div>
 <% end html %>
<% end template %>
```

Figure 23: Template `Delivery-Trucks`

tegrate data from multiple sources in an efficient manner by utilizing a distributed query processor. Additionally, FORWARD's UAS deals with impedance mismatch issues that developers typically have to deal with when developing full stack applications. At the same time FORWARD promotes location transparency, since the application developer is able to utilize datasets that may reside on the back-end as if they were located on the client-side.

As mentioned, FORWARD utilizes IVM techniques that propagate diffs throughout the application. When FORWARD is utilized as a full-stack framework, it automatically propagates diffs from the server-side to the client-side, as soon as the respective back-end sources that feed the application trigger a mutation. When FORWARD is used as purely a client-side framework, however, it has no control over the remote services and the respective sources they use, therefore it is not aware of whether they support IVM techniques or not. Despite that, FORWARD is still able to operate in a diff oriented fashion by employing various approaches.

The first approach we describe can be used when the underlying remote web services support IVM techniques that generate diffs. In such cases the client can simply request the respective diffs and utilize them to update the application state. This approach is implemented by either employing polling or interrupts. In the first case, the client periodically transmits HTTP requests to the server in order to get the latest diffs; while in the second case WebSockets [42] are used to propagate diffs from the server-side to the client in real-time. If the utilized back-end services do not support IVM techniques, however, this approach cannot be used to generate diffs on the application state. In this case, the application developer can implement his own delta functions thus manually generating diffs that target the local application state. These diffs are then passed on to FORWARD and eventually lead to the respective updates on the View. While this approach is fairly efficient and it works without IVM compatible remote services, it pushes some of the load to the application developer, since he/she has to manually generate the diffs.

Lastly, if the application developer does not wish to implement any additional logic in order to generate diffs, FORWARD can identify changes by employing a "diff-ing" algorithm on the client. With this approach FORWARD simply reevaluates the Model in full when an action occurs and attempts to identify changes between the current state of the Model and the previous one. This approach appears to be similar to the respective "diff-ing" approaches that component libraries perform on instances of the Virtual DOM. Despite the similarities, this "diff-ing" algorithm has several advantages, since it is run on the Model instead of the ViewModel. The complexity of a "diff-ing" algorithm is typically proportional to the size of the structure that will be explored for changes. In most cases the ViewModel of an application is larger and more heavily nested than the Model. Additionally, a single part of the Model can be used in multiple parts of the View, therefore if this part of the Model changes it will eventually trigger changes to multiple parts of the ViewModel. By essentially pushing the "diff-ing" algorithm down to the Model level, we are able to identify a change and infer the respective changes that will take place on the ViewModel more efficiently. For those reasons, even if this approach is essentially the worst-case scenario for FORWARD it still performs better than the respective approaches that are utilized by Component Libraries.

5.4 Internal Client-Side Architecture

Since the aforementioned methods generate diffs that do not explicitly state which parts of the View have to be updated, FORWARD cannot use them to directly infer which renderer calls have to be called to update the appropriate parts of the View. Instead FORWARD has to translate those diffs into diffs that target the ViewModel. This process is carried through by the template IVM algorithm, which utilizes the incoming diffs, the bindings that are included in a template and a set of IVM rules in order to generate the respective ViewModel diffs. FORWARD then uses the ViewModel diffs to identify which Visual Units are responsible for updating the respective parts of the visual layer and it eventually calls the respective renderers that can reflect those changes to the View.

6. CONCLUSION

In this paper we provide an in depth description of MVVM and Component Frameworks that are currently considered the "state-of-the-art" in the web community. Furthermore, we provide a detailed description of the internal mechanisms that these frameworks employ to propagate changes from the application state to the view and explained the tradeoffs that those mechanisms impose. Lastly, we illustrated the internal architecture of FORWARD and described how it achieves change propagation, in an efficient manner, using IVM techniques during the life-cycle of an application.

7. REFERENCES

[1] Walter L. HÃijrsch and Cristina Videira Lopes. Separation of concerns. Technical report, 1995.

[2] Yupeng Fu, Kian Win Ong, and Yannis Papakonstantinou. Declarative ajax web applications through sql++ on a unified application state. *arXiv preprint arXiv:1308.0656*, 2013.

[3] Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Erick Zamora. Forward: data-centric uls using declarative templates that efficiently wrap third-party javascript components. *Proceedings of the VLDB Endowment*, 7(13):1649–1652, 2014.

[4] Graham Cormode and Balachander Krishnamurthy. Key differences between web 1.0 and web 2.0. *First Monday*, 13(6), 2008.

[5] Sareh Aghaei, Mohammad Ali Nematbakhsh, and Hadi Khosravi Farsani. Evolution of the world wide web: From web 1.0 to web 4.0. *International Journal of Web & Semantic Technology*, 3(1):1, 2012.

[6] What is web1.0? `https://www.techopedia.com/definition/27960/web-10`. Accessed: 2016-04-05.

[7] Key differences between web 1.0 and web 2.0. `http://firstmonday.org/article/view/2125/1972`. Accessed: 2016-04-05.

[8] Wikipedia. Document object model — wikipedia, the free encyclopedia, 2016. [Online; accessed 5-April-2016].

[9] The html syntax. `https://www.w3.org/TR/2011/WD-html5-20110525/syntax.html`. Accessed: 2016-04-05.

[10] John Resig et al. Jquery, 2006.

[11] Thomas Powell. *Ajax: The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[12] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[13] Gabriel Svennerberg. *Beginning Google Maps API 3*. Apress, 2010.

[14] Amcharts. `https://www.amcharts.com/javascript-maps/`. Accessed: 2016-04-05.

[15] Paul Crickard III. *Leaflet. js Essentials*. Packt Publishing Ltd, 2014.

[16] Erik Hazzard. *Openlayers 2.10 beginner's guide*. Packt Publishing Ltd, 2011.

[17] Joseph Kuan. *Learning Highcharts*. Packt Publishing Ltd, 2012.

[18] Google charts. `https://developers.google.com/chart/`. Accessed: 2016-04-05.

[19] Rosário Durão, Wei Tie, Kristina Henneke, Karen M Balch, Maxwell Hill, and Rachel Rayl. Visualizing the data visualization network: The dvmap project. *European Scientific Journal*, 2014.

[20] Wikipedia. D3.js — wikipedia, the free encyclopedia, 2016. [Online; accessed 6-April-2016].

[21] Tommi Mikkonen and Antero Taivalsaari. Web applications: Spaghetti code for the 21st century. Technical report, Mountain View, CA, USA, 2007.

[22] Google charts. `https://developers.google.com/maps/web/`. Accessed: 2016-04-05.

[23] Google charts. `http://kimmobrunfeldt.github.io/progressbar.js/`. Accessed: 2016-04-05.

[24] Angularjs. `https://angularjs.org/`.

[25] Angularjs open source framework. `https://github.com/angular/angular.js`.

[26] Ember, a framework for creating ambitious web applications. `http://emberjs.com/`.

[27] Handlebarsjs, minimal templating on steroids. `http://handlebarsjs.com/`.

[28] Knockoutjs. `http://knockoutjs.com/index.html`.

[29] Zhigang Liu, Yousuf Saifullah, Marc Greis, and Srinivas Sreemanthula. Http compression techniques. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 4, pages 2495–2500. IEEE, 2005.

[30] Knockout.js - why it's not a framework. `https://pwkad.wordpress.com/2013/10/07/knockout-js-why-its-not-a-framework/`. Accessed: 2016-04-12.

[31] Web components. `https://developer.mozilla.org/en-US/docs/Web/Web_Components`. Accessed: 2016-04-12.

[32] Wikipedia - web components. `https://en.wikipedia.org/wiki/Web_Components`. Accessed: 2016-04-12.

[33] George T Heineman and William T Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, page 5, 2001.

[34] Shadow dom. `https://developer.mozilla.org/en-US/docs/Web/Web_Components/Shadow_DOM`. Accessed: 2016-04-05.

[35] Polymer. `https://www.polymer-project.org/1.0/`. Accessed: 2016-04-12.

[36] Mithriljs. `https://lhorie.github.io/mithril/index.html`.

[37] Inheritance and the prototype chain. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`. Accessed: 2016-05-05.

[38] Msx, jsx for mithril. `https://github.com/insin/msx`.

[39] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. Utilizing ids to accelerate incremental view maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1985–2000. ACM, 2015.

[40] Andreas Behrend and Thomas Jörg. Optimized incremental etl jobs for maintaining data warehouses. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, IDEAS '10, pages 216–224, New York, NY, USA, 2010. ACM.

[41] Latha S Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *ACM SIGMOD Record*, volume 25, pages 469–480. ACM, 1996.

[42] Websocket. `https://en.wikipedia.org/wiki/WebSocket`.