UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Facilitating the development of Analytical Dashboards on the Web

Permalink
https://escholarship.org/uc/item/4174z90w

Author
Zarifis, Konstantinos

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Facilitating the development of Analytical Dashboards on the Web

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Konstantinos Zarifis

Committee in charge:

Professor Yannis Papakonstantinou, Chair
Professor William G. Griswold, Co-Chair
Professor Alin Deutsch
Professor Ranjit Jhala
Professor Kevin Patrick

2019

Copyright

Konstantinos Zarifis, 2019

All rights reserved.

The Dissertation of Konstantinos Zarifis is approved and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California San Diego

2019

DEDICATION

This dissertation is dedicated to my family and friends you have been by my side all these years. I would not have reached this milestone in my life without their endless love and support. This is for you!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am hugely grateful to my advisor Yannis Papakonstantinou for his support and guidance throughout my years as a graduate student. Yannis is one of the most intelligent people I have ever met. He has the unique ability to comprehend complex concepts almost instantaneously. His talent to focus on the big picture and introduce abstractions that filter out the noise of low-level intricacies, has helped me resolve a plethora of technical issues numerous times. His critical thinking and talent to always ask the right questions never ceases to amaze and inspire me. Additionally, I am honored to have worked with my co-advisor, Bill Griswold, who supported me while conducting the final user-study and writing the last chapters of this dissertation. His advising style is one of a kind, he managed to make research a lot less stressful and a lot more structured and fun. Despite his busy schedule, he always had time for me every time I needed his advice. His insightful comments helped me improve on my writing skills and his emotional support and genuine interest gave me the strength I needed to push through to the finish line.

Special thanks to my committee members Ranjit Jhala, Alin Deutsch and Kevin Patrick for their perceptive comments and meaningful discussions. Knowing that I could always turn to such accomplished people and scientists for advice was truly invaluable. Their personal touch was always appreciated. I could not have asked for better committee members.

I am also thankful for the mentors and managers I have worked with during my internships: Raja Hayek from Annai Systems (currently Triality), Mohan Sankaran from Informatica (currently Microsoft) and Ryne Chaloux from AppFolio. They helped me demystify industry and inspired me to work on equally interesting problems in a non-academic setting. They helped me obtain industry experience and reminded me how much I enjoy working with others.

Being part of the CSE department and the DB lab has been an honor. I would like to thank all the members of the lab, for the interesting discussions we have shared over the years, and for always being there for me when I needed their advice. Particularly I would like to thank the Professors Arun Kumar and Victor Vianu, for always trying to help me improve and succeed academically. I would also like to thank all the amazing PhD students I had the pleasure of having

in my life all these years: Chunbin Lin, Jianguo Wang, Jules Testard, Rana Alotaibi, Supun Nakandala and Ainur Smagulova. My PhD would not be the same without all of you. I would especially like to thank Kian Win Ong and Yannis Katsis for being amazing mentors. Working with them and learning from them has been a unique experience. I will always remember our conversations, the drawings of FORWARD's architecture on the white board and the extended analysis of our newly designed internal data structures. I am glad they believed in me and pushed me to reach my full potential.

These past years would not be the same without the amazing people I met in San Diego that am lucky to call my friends. They made San Diego feel like home and became my family. I am grateful for Konstantinos, Andreas, Panos, Vicky and Nikos who were always there for me. Thank you for always having my best interest in mind and for always making my day. I will always remember "The Garage" and the endless conversations until late at night with Andreas and Dimos, hanging out with you guys after a long day truly felt like an oasis.

I am grateful to Tal who has been there for me devotedly through thick and thin all these years. She is one of the strongest and most independent women I know, while at the same time the most sensitive, affectionate and supportive person in my life. Her personality and sense of humor always brightens my day.

Last but not least, this journey would not have been possible without the support of my parents: Vasilis and Dimitra, and my sister, Vicky, who have been by my side since day one. Their strength and perseverance through hardships has taught me to be strong and never give up on my goals despite any hurdles I may face. Their endless emotional support has given me the courage and strength to face challenges head-on. Their unconditional love has always warmed my heart. They would sacrifice everything for me and I would too.

Work adapted in this Dissertation

Chapters 2 and 3 contain material adapted from the technical report: "**In-depth Survey of MVVM Web Application Frameworks**", by Konstantinos (Costas) Zarifis. The dissertation

author was the primary investigator and author of this paper.

Chapter 9 contain material adapted from the publication "**ViDeTTe Interactive Notebooks**" appearing in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, Article No. 2, (HILDA) SIGMOD*. Houston, TX, USA - June 10, 2018. Konstantinos (Costas) Zarifis; Yannis Papakonstantinou. The dissertation author was the primary investigator and author of this paper.

Chapters 4-8 and 10 contain material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission to *ACM Transactions on Internet Technology* or *ACM Transactions on Database Systems*. The dissertation author was the primary investigator and author of this paper.

VITA

2007–2013	Honors Engineering Degree in Electronic and Computer Engineering (ECE) at Technical University of Crete. 5 year University Program (Equivalent to B.Eng and M.Sc.)
2013–2016	Master of Science Degree in Computer Science and Engineering at the University of California, San Diego.
2013–2019	Graduate Research Assistant, Department of Computer Science and Engineering University of California, San Diego
2019	Doctor of Philosophy, Department of Computer Science and Engineering University of California, San Diego

PUBLICATIONS

Konstantinos (Costas) Zarifis, Yannis Papakonstantinou. "**ViDeTTe Interactive Notebooks**". *Human-In-the-Loop Data Analytics (HILDA), SIGMOD 2018.*

Yannis Katsis, Natasha Balac, Derek Chapman, Madhur Kapoor, Jessica Block, William G. Griswold, Jeannie Huang, Nikos Koulouris, Massimiliano Menarini, Viswanath Nandigam, Mandy Ngo, Kian Win Ong, Yannis Papakonstantinou, Besa Smith, Konstantinos (Costas) Zarifis, Steven Woolf, Kevin Patrick. "**Big Data Discovery for Public Health**", *Chase 2017, Second IEEE/ACM Conference on Connected Health, Applications, Systems and Engineering Technologies.*

Konstantinos (Costas) Zarifis, Yannis Katsis, Yannis Papakonstantinou. "**Application View Maintenance: Optimizing Change Propagation in Mobile Applications**", *Technical Report 2017.*

Konstantinos (Costas) Zarifis. "**In-depth Survey of MVVM Web Application Frameworks**", *Technical Report 2016.*

ABSTRACT OF THE DISSERTATION

Facilitating the development of Analytical Dashboards on the Web

by

Konstantinos Zarifis

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Yannis Papakonstantinou, Chair
Professor William G. Griswold, Co-Chair

Developing custom, reactive dashboards that deliver continuously updated visualizations, has been a traditionally laborious task, due to the advanced technical expertise they require in web development. MVVM frameworks have attempted to simplify this process, by offering a template language that enables the declarative specification of dashboards, and by automatically maintaining the displayed visualizations when the visualized data change. These frameworks, however, still exhibit significant drawbacks. When mutations occur in underlying data sources, developers have to observe and manually propagate (using imperative code) said mutations to the framework, which still requires advanced skill-set and experience in web development. Addi-

tionally, even though these frameworks automatically mutate the dependent visualizations (thus absolving developers from manually performing this task), once mutations have been propagated to the framework, their incremental rendering algorithms incur significant performance penalties, especially when visualizing big data.

To address these issues, we present FORWARD, a framework that simplifies the development of custom live dashboards. FORWARD offers a declarative template language that simplifies the integration of (a) database and web service systems (such as Postgres and GraphQL), (b) visualization libraries (such as Google Maps and HighCharts), and (c) data processing functions (performing aggregations, ML computations, and more), thus enabling a truly declarative specification of dashboards. This significantly lowers the technical expertise needed for dashboard development, thus allowing programmers with limited experience in web development (such as data analysts) to produce custom, information-dense, highly-reactive dashboards. FORWARD templates describe dashboards as semi-structured views. As a result, when mutations occur in the base-data of such views, FORWARD employs novel incremental view maintenance techniques, that automatically propagate changes from data sources all the way to the visual layer more efficiently than existing frameworks.

In this Thesis, we illustrate FORWARD's template language and incremental rendering algorithms, and show their superior algorithmic complexity compared to the state of the art. Experimental results validate the complexity results and show that FORWARD's incremental rendering can be orders of magnitude more efficient than existing approaches. Line-of-code and development time experiments show that the performance gains are accompanied by productivity gains for developers that use FORWARD to build information-dense dashboards. Lastly, we present ViDeTTe, a system that integrates FORWARD with Jupyter notebooks, and enables data analysts to build Jupyter Notebooks with reactive visualizations. Notebook readers can then directly interact with these visualizations to further explore the underlying data, a functionality that is currently not supported in Jupyter Notebooks.

Chapter 1

Introduction

In the recent years we have been witnessing the proliferation of live web dashboards that depict rapidly changing data through complex JavaScript visualization libraries, such as Google Maps [55], Highcharts [61] and others. Examples include among others live traffic maps [77], real-time sensor and IoT monitoring dashboards [92, 24, 64, 94, 65], live stock market graphs [93, 23, 28, 86] and more.

Dashboards are information reporting tools, that present data in a graphical interface. They employ visualizations (such as charts and maps) and textual content to summarize complex information and present it in an easily digestible way. Typically, dashboards are implemented as web applications and are displayed through web browsers. The dashboards we will be focusing on in this dissertation are:

- **Information-Dense**: They display Megabytes or Gigabytes of data
- **Schema-Flexible**: They support a variety of different data schemas, from structured (relational) data, to semi-structured and unstructured datasets (such as JSON, key-value pair and more)
- **Source-Flexible**: They display datasets that reside in a variety of sources, such as database systems, third party web services, local and remote files and more
- **Visualization-rich**: A dashboard can employ a variety of visualizations such as charts,

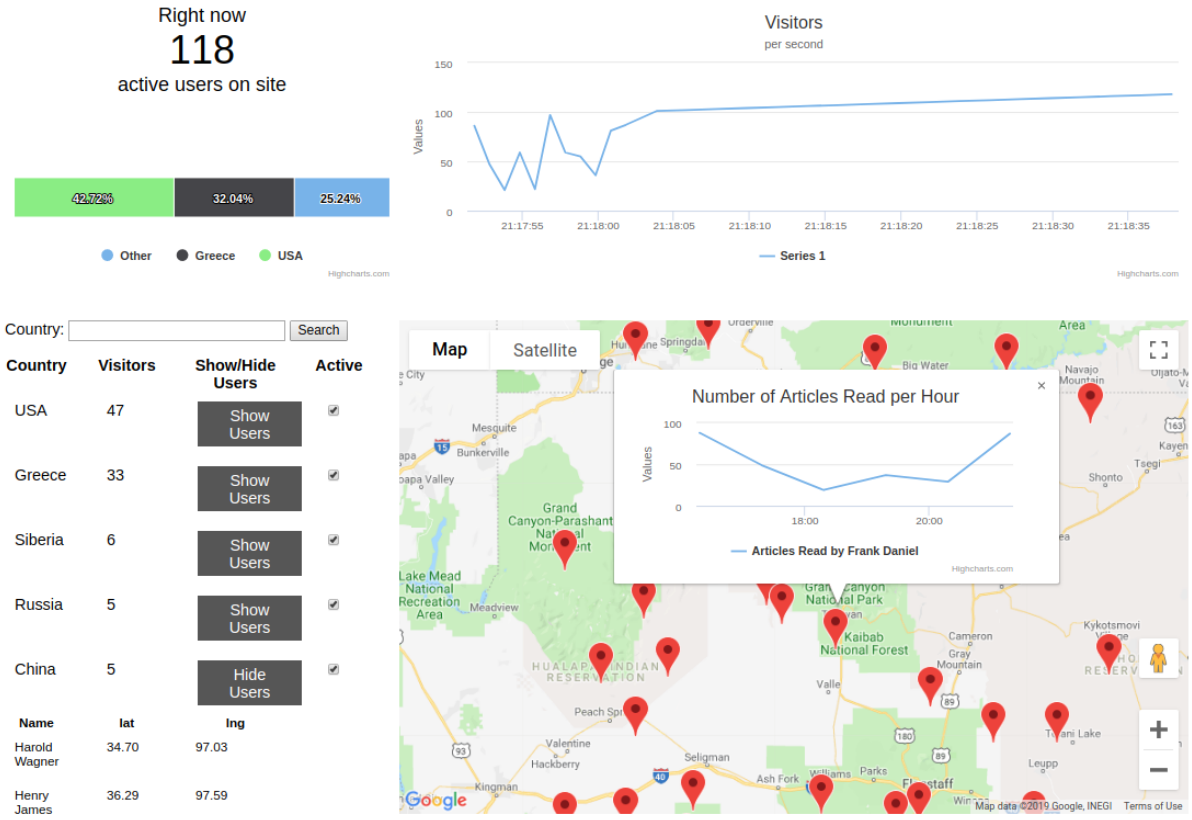


Figure 1.1. Running Example

maps and textual content.

- **Reactive**: They react to user input. As a result, users can utilize dashboards to explore the underlying data by simply interacting with the user interface, which lowers the required technical expertise (dashboard users do not need to write any code to analyze the underlying data)

- **Customizable Visual Layer**: A dashboard can present data in any possible format and layout. Form-and-Report softwares that are available by database vendors cannot be used for the development of fully customizable dashboards due to the limited layouts, visualizations and data sources they support.

- **Continuously updated**: They are capable of updating the visual layer in real-time whenever the underlying data change

1.1 Running Example

Consider the dashboard shown in Figure 1.1 that enables website administrators to monitor user activity on a news portal website. At the top left corner, the screen displays the number of active users that are currently visiting the website. Below, there is a graph showing the top two countries from which users originate and a table showing the full list of countries with active users, along with the names of each individual user. On the right side, there is a chart showing historic data about the total number of the users that have visited the website up to this second and a map graphically depicting each currently active user as a marker. When the dashboard administrator clicks on a particular marker, a popup window appears containing a chart that provides information about the website articles that the respective user has read in the past few hours. Since this application shows real-time data, the view is updated continuously. As new users are visiting the website, old users leave the website or as users move around the respective visualizations are updated accordingly in real-time.

1.2 Categories of dashboard developers

It is worth noting that there exists a very diverse audience that is interested in building such dashboards. We classify this audience based on their technical background into database developers, data analysts and advanced application developers. We next describe the characteristics and skill-set of each audience:

- **Database developers** are familiar with declarative markup and sql-like languages used to obtain data from various database systems and simply want to visualize the obtained dataset. This audience lacks the technical expertise that would enable them to generate any form of visualizations or dashboards.

- **Data analysts** can employ high level programming languages such as Python or R to obtain data, apply certain data processing functions (such as ML algorithms) and potentially

generate individual visualizations (for instance within a Jupyter Notebook or R Studio), but cannot generate any kind of custom user-facing dashboard.

- **Advanced application developers** have the required technical expertise to obtain data from any data source they wish, process them accordingly and build any kind of dashboard. As we will describe later in this chapter, however, this is a relatively difficult, labor intensive and time consuming process. The user-study presented in Chapter 8, verifies that building such custom dashboards, using existing frameworks, is an arduous and labor intensive process.

1.3 What makes manual dashboard development challenging

The dashboard described in Section 1.1 illustrates a combination of live data and complex visual components. This combination makes the manual development of such reactive dashboards using imperative logic tedious and error-prone, as the application developer needs to not only specify the initial creation of the visual layer but also *incrementally* update the employed visualizations in response to data changes. This process requires, among others, writing imperative code to propagate changes from data sources, identifying individual visualizations that are affected by changes and lastly, incrementally updating each affected visualization, by invoking renderers provided by the underlying visualization libraries.

Example 1.3.1 *For instance, if a new user visits the news portal, an existing visitor leaves, or the location of a visitor gets updated (because the user physically moved), the developer of the presented dashboard needs to provide code that observes the respective mutations and incrementally updates only the affected visualizations. In order to incrementally update the affected visualizations, dashboard developers need to use renderer methods provided by the underlying visualization libraries (HighCharts, Google Maps, HTML DOM etc.,). Furthermore,*

in order to allow the website administrator (i.e., the user of the dashboard) to search for the visitors that originate from a particular country (using the text box that appears above the table with the countries), the dashboard developer needs to write imperative code that filters the visualized visitors based on search criteria and updates the visualizations that appear on the dashboard.

Note that the amount of effort and code needed is proportional to (a) the complexity of the dashboard's visual layer (dashboards with more visualizations require more effort), (b) the number of individual mutations supported (each mutation requires explicit logic that propagates changes to the visual layer) and (c) the number of widgets that collect user input which affects the visual layer or the state of the dashboard (each input widget such as text-fields, check-boxes and others, requires explicit logic that handles the user's input). In order to examine the level of difficulty of dashboard development, we asked an advanced web developer to implement the dashboard shown in Figure 1.1. The implementation of this dashboard took roughly 48 hours, the majority of which (roughly 30 hours) was spent writing code that propagates changes from underlying sources to the visual layer (roughly 18 out of 30 hours were spent incrementally updating just the visual layer). Note that the reported time does not include any kind of testing, which is an essential step of every software system. Adding tests would double or even triple the development time. It is also worth noting that building such dashboards requires familiarity with many web technologies (web services, web sockets, DOM manipulation and more), therefore developers that do not have expertise in this area would simply not be able to build such dashboards, as we will show in the user study described in Chapter 8.

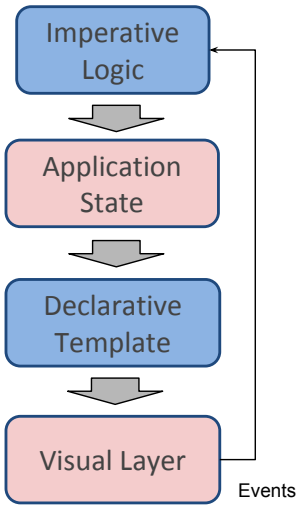


Figure 1.2. Developing Dashboards using existing MVVM (Model-View-ViewModel) frameworks. The blue boxes represent the code that application developers need to provide, in order to produce the pink boxes.

1.4 MVVM Frameworks have attempted to simplify Dashboard Development

Various Model-View-ViewModel (MVVM) web application frameworks such as React (developed and maintained by Facebook) and Angular (maintained by Google) have attempted to simplify the process of web development by providing a declarative template language that is expressive enough to describe the creation of a visual layer using data objects representing the application state (this process is depicted in Figure 1.2). For instance, MVVM frameworks provide a declarative template language that can be used to generate the visual layer, shown in Figure 1.1, given an array of *visitors* (shown in Figure 1.3). Many of these frameworks are also capable of automating the propagation of changes from the application state to the visual layer. For instance, if the location (lat, lng) of a particular visitor changes, these frameworks will automatically relocate the respective marker on the map. Because of these features, MVVM frameworks have been widely successful in the web community, as they have managed to simplify an integral part of dashboard development: the generation and maintenance of the dashboard's visual layer.

```
Visitors = [
  {
    "lat":32.7523,
    "lng":-117.13,
    "name":"Yannis Papakonstantinou",
    "visitedPagesPerMonth": [
      { "month":"Jan","numberOfVisits":133 },
      { "month":"Feb","numberOfVisits":88 },
      { "month":"Mar","numberOfVisits":128 },
      { "month":"Apr","numberOfVisits":95 },
      { "month":"May","numberOfVisits":158 },
      { "month":"Jun","numberOfVisits":154 },
      { "month":"Jul","numberOfVisits":176 },
      { "month":"Aug","numberOfVisits":91 }]
  },
  {
    "lat":32.7883,
    "lng":-117.19,
    "name":"Bill Griswold",
    "visitedPagesPerMonth": [
      { "month":"Jan","numberOfVisits":33 },
      { "month":"Feb","numberOfVisits":43 },
      { "month":"Mar","numberOfVisits":15 },
      ...
  } ...
]
```

Figure 1.3. Array of Visitors

1.5 Limitations of MVVM Frameworks

Unfortunately, despite their remarkable success, the productivity gains MVVM frameworks offer to dashboard developers are still insufficient as they do not successfully facilitate every task needed for the development of such dashboards. In this section, we discuss these limitations.

1.5.1 Generating the application state requires imperative code

A major issue of current MVVM frameworks is that the provided template language is not **expressive** enough to facilitate the generation of the application state declaratively. Specifically, while these frameworks provide a template language that is capable of specifying the visual layer given an object representing the application state (for instance, the *visitors* array shown in Figure 1.3), this application state is, in most dashboards, derived from data residing in a plethora of data sources (such as Database Systems, Web Services and more). None of these frameworks, however, provides template constructs that are **expressive** enough to integrate with

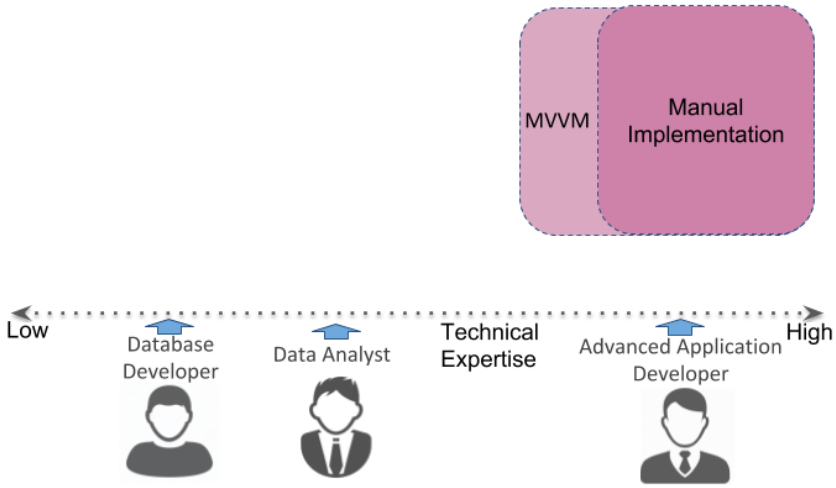


Figure 1.4. Required technical expertise for dashboard development using manual implementation (imperative logic) VS currently available MVVM frameworks (not including FORWARD)

remote sources and obtain and process the respective data accordingly in order to generate the application state. This limited expressiveness also impacts the **generality** of the frameworks, as dashboard developers cannot easily obtain and visualize data that reside in diverse data sources. Furthermore, while MVVM frameworks successfully automate the process of incremental rendering of the visual layer given mutations on the application state, they do not propagate changes from underlying data sources to the application state. As a result, developers need to write imperative code to (a) obtain data from remote database systems and web services and process them accordingly in order to instantiate the application state, and (b) observe the underlying data source for every individual mutation that might occur and manually mutate the application state accordingly.

The required technical expertise for dashboard development is high even with MVVM frameworks

As a result, implementing even simple live dashboards still takes considerable effort and advanced technical expertise in web development. Indicatively, the development of the dashboard shown in Figure 1.1, using Angular, took 38 hours and 927 lines-of-code to be developed by an experienced web developer. Note that the development time was reduced from

48 hours to 38, because Angular is able to generate the visual layer declaratively and automate the process of incremental rendering. Instantiating the application state, however, still requires the design and implementation of a three-tier architecture (data layer, middleware and client-side), and the manual implementation of the majority of this architecture through imperative logic, which requires advanced technical expertise. As a result, even with the use of such frameworks, only developers that have advanced expertise in web development are capable of building such dashboards, albeit in a considerable amount of time, while users that lack this expertise, such as data analysts and database developers, are simply not capable of building custom dashboards at all (Figure 1.4 graphically depicts this).

1.5.2 The automatic maintenance of the visual layer incurs performance penalties

Another major issue introduced by MVVM frameworks, is the **inefficiency of the propagation algorithm** that maintains the application's visual layer given mutations to the application state. Specifically, the complexity of the algorithms that perform this task is proportional to the entire size of the dashboard's visual layer. This is because the propagation algorithms (a) fully generate the abstract representation of the visual layer (also called ViewModel) by evaluating the template given the new application state and (b) compare the pre-state of the ViewModel with its post-state to identify the parts that were mutated. Both operations, however, are very expensive computationally. As a result, such frameworks simply cannot be used for the development of live dashboards that contain information-dense visualizations, as the size of the visual layer (and respectively, the ViewModel) impacts the runtime of the change propagation algorithms. More importantly, the increased complexity of the propagation algorithm can also cause significant deterioration of the user experience, as the generated dashboard responds slowly to user inputs, while executing change propagation algorithms. This is because the browser (due to the single-threaded nature of JavaScript) cannot react to user events while executing these algorithms, and since they run for extended periods of time, it makes the dashboard seem unresponsive.

1.5.3 Some MVVM Frameworks cannot generate visualizations

Another major issue is that many MVVM frameworks (such as React, Mithril and more) do not facilitate the creation and maintenance of visualizations. For instance, React is only capable of facilitating the creation and maintenance of textual content (using HTML) by default[1]. This impacts **generality**, as developers cannot build any dashboard they wish. Instead, they can only build applications that contain textual content using the framework. Developers that wish to build visualization-rich dashboards are required to work outside of the framework by providing imperative code that generates or updates each employed visualizations, which defeats the purpose of using such a framework in the first place. Fortunately, a few of these frameworks (such as Angular) do in fact support the generation and incremental maintenance (rendering) of visualizations.

1.5.4 Extending MVVM frameworks is labor intensive

The MVVM frameworks that support the generation and maintenance of a visual layer that contains visualizations, do so by offering *template constructs* that automatically generate visualizations given a dataset that describes the state of said visualization (ViewModel). This ViewModel is typically generated automatically through the provided template language. The template constructs are also capable of automatically propagating changes from the ViewModel to the respective visualization, thus absolving dashboard developers from the intricacies of underlying visualization libraries.

Enhancing these frameworks by wrapping new visualization libraries into reusable constructs, that can be used in a template is a labor-intensive and error-prone process as well. Indicatively, in an experiment we conducted (presented in chapter 7), the implementation of a very basic (and rather inefficient) Google maps template construct in Angular took roughly 17106 lines

[1] It is possible for more advanced developers to create React constructs that generate visualizations, but the framework does not provide a mechanism that can be used to incrementally render such visualizations. As a result most existing react components, simply rerender the entire visualization, when a mutation are observed, which negatively impacts user experience, due to flickering.

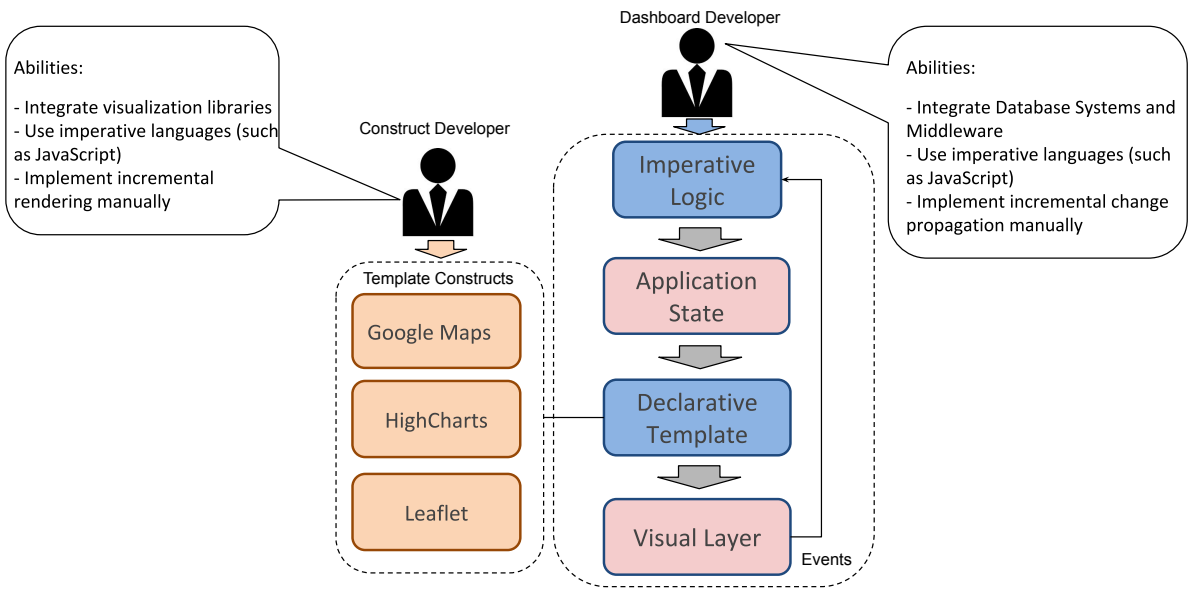


Figure 1.5. Developing Dashboards and Template Constructs in MVVM Frameworks. Note that while MVVM frameworks simplify the generation of the visual layer by providing declarative templates that employ template constructs, dashboard developers still need to perform tasks that only advanced web application developers have the ability to complete.

of code. Such constructs need to be able to automatically generate and maintain visualizations. The APIs (Application programming interfaces) these frameworks provide, however, do not offer the appropriate abstractions that would facilitate easy and modular development of such constructs. Specifically, the provided **API does not provide an event-like mechanism for each mutation that might take place in the ViewModel**. As a result, construct developers that want to introduce new such template constructs need to provide imperative logic that (a) manually navigates the entire ViewModel comparing its pre-state with the post-state, in order to infer which part of the ViewModel changed and (b) provide the action that reflects the mutation to the visual layer, using the API of the underlying visualization library. Note that template construct developers and dashboard developers are not necessarily the same person (as shown in Figure 1.5). Construct developers are responsible for creating reusable template constructs that can be used by Dashboard developers to generate visualizations.

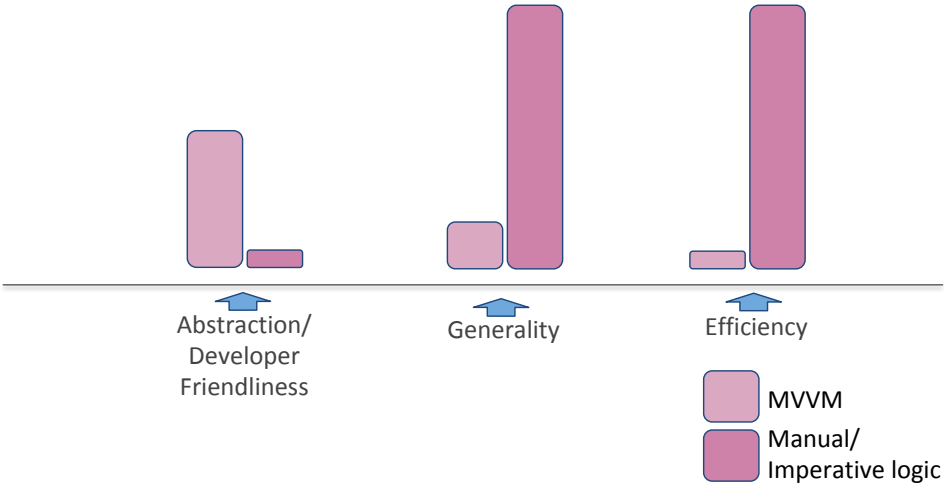


Figure 1.6. Dashboard development using MVVM frameworks VS Manual (imperative) Code

1.6 Trade-offs of Dashboard Development with and without MVVM frameworks

To sum up, dashboard development has the following trade-offs given the way it is implemented (as shown in Figure 1.6):

- **Manual dashboard development**, using imperative logic, is fairly **general** since developers can build any kind of dashboard and perform any kind of data access, data processing and data visualization they wish. The employed programming language (typically JavaScript) is **expressive** and **general** enough to facilitate any such task. Additionally, **propagating changes is efficient**, since the developer knows exactly how each mutation in underlying data sources affects the visual layer and can provide logic that applies this exact set of changes to the visual layer. On the other hand, building dashboards manually requires a lot of imperative code, is very labor intensive and requires advanced technical expertise, therefore it is **not developer-friendly** or **concise**.

- **Dashboard development using MVVM frameworks**, however, is more **developer-friendly** as it offers a higher level of abstraction. Specifically, such frameworks (a) offer a declarative template language that is expressive enough to facilitate the generation

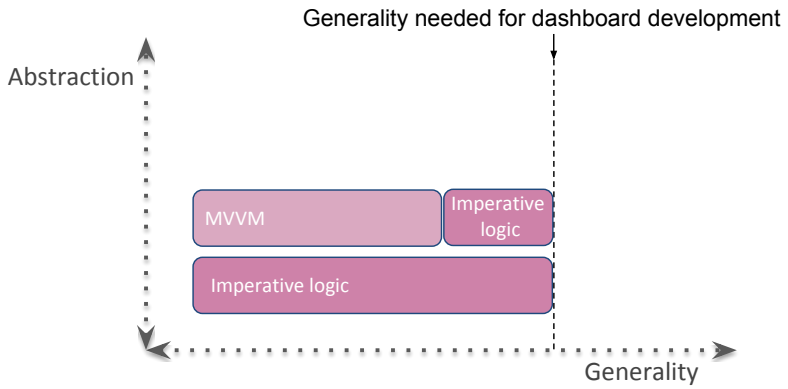


Figure 1.7. Abstraction and Generality in imperative logic, existing MVVM frameworks and FORWARD

of the visual layer, and (b) automatically maintain the visual layer given mutations to the application state, thus absolving developers from having to perform these tasks manually. These frameworks, however, still have the following limitations:

- **Limited conciseness and generality**. The provided template language is not expressive enough to describe data access and data processing. As a result, dashboard developers are required to manually perform these tasks, using imperative code, in order to create the application state (as shown in Figures 1.2 and 1.7), which also limits **conciseness**. More interestingly, propagating changes from underlying sources to the application state also requires complex imperative logic for every mutation that might occur during the life-cycle of an application. Performing these tasks is very labor intensive and requires deep understanding of web technologies (as shown in Figure 1.5). Additionally, dashboard developers cannot generate any visualization they wish, they are limited to generating either (a) just textual content (since many MVVM frameworks do not support visualizations) or (b) visualizations supported by template constructs that are available (for the few frameworks that do support the generation of visualizations). However, extending the framework with new reusable visualization constructs is very labor intensive and as a result there are not a lot of

such constructs available.

- **Inefficiency of supported change propagation Algorithm**. Dashboard developers cannot employ existing MVVM frameworks, to build live, information-dense dashboards, since the employed change propagation algorithm is very inefficient. Using such frameworks for the development of live information-dense dashboards would not only result in delays during the propagation of changes, but also in a significantly deteriorated user experience (due to the single-threaded nature of JavaScript). As a result, often developers have to manually propagate changes, using imperative logic, instead of allowing the framework to automatically handle this task, in an effort to improve the performance and user experience, which deems such frameworks unnecessary. More information about the reason why the change propagation algorithm of these frameworks is inefficient can be found in Section 2.3.

1.7 Hypothesis for Resolving Limitations of MVVM Frameworks

We claim that it is possible to resolve the aforementioned problems by offering a framework that:

- *Provides a declarative template language, that can describe not only the generation of the visual layer but also data access, from a variety of sources, and data processing*. Specifically, the template language should include reusable template constructs, thus enabling the declarative specification of such operations. When using these constructs dashboard developers do not have to add imperative logic to obtain, process or visualize data, as the respective logic is essentially encapsulated in these constructs. The resulting template language is more **expressive** than languages used by existing frameworks, which also improves the **developer-friendliness** and **conciseness** of the framework. As a result, developers are able to build fully-fledged dashboards using purely declarative code, thus

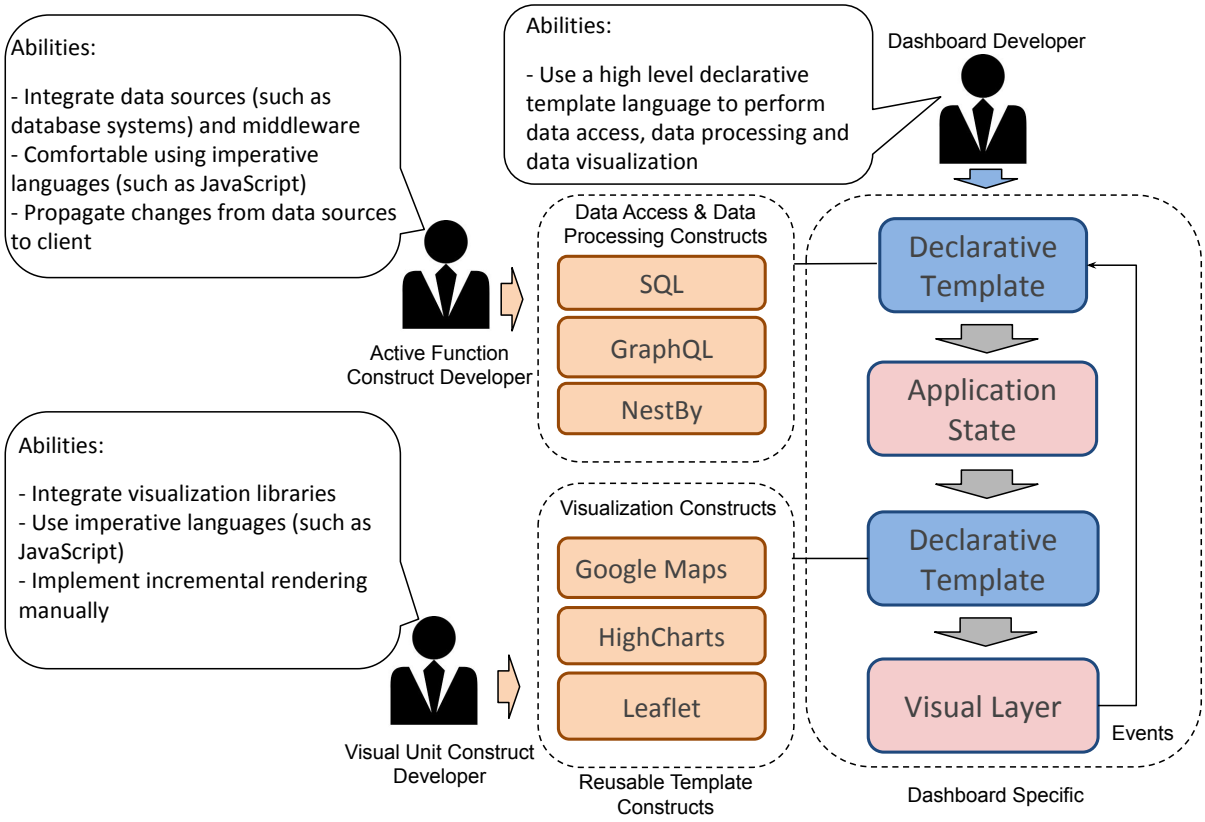


Figure 1.8. Developing Dashboards and Template Constructs in FORWARD. Notice that the technical expertise needed by dashboard developers is much lower compared to existing MVVM frameworks. The only requirement is to be familiar with the provided declarative template language, which can be used to perform data access, data processing and data visualizations. The imperative logic that is typically required for such tasks has been moved to template constructs (such as active functions and visual units), that can be declaratively embedded in the template. Responsible for the development of such reusable template constructs are visual unit and active function developers, that have a higher expertise in JavaScript and more integration capabilities than the dashboard developer. As result, in FORWARD we accomplish a better division of labor between the potentially novice dashboard developers and the more advanced construct developers.

absolving them from the intricacies of database systems and visualization libraries (Figure 1.8, shows the assumptions about the abilities of the dashboard developer and the reuse of such template constructs). This improvement also increases the **generality** of the framework, since it would enable novice developers to build any dashboard they wish, even if they do not have the expertise of advanced web developers.

- *Provides an easy-to-use interface that enables construct developers to enhance the framework by creating reusable template constructs that perform data access, data processing or visualization*. By supporting the introduction of such constructs, while minimizing the amount of effort needed for their development, we improve the **extensibility** of the framework which enables the construction of more **general** dashboards, as they can obtain data from any data source, perform any type of data processing and construct any potential visualization. This extensibility offers a better division of labor, since more skilled construct developers can build reusable constructs, which can then be used by the potentially novice dashboard developers. To make the introduction of new template constructs easy, we eliminate typical pain-points of this process. Specifically, we provide a declarative trigger language that notifies developer-provided actions when mutations are observed on the input of the template constructs. As a result, when using this language, developers do not need to provide logic that navigates the entire input comparing its pre-state with its post-state, in order to infer the parts that were mutated (which is necessary in existing frameworks). Additionally, to absolve construct developers from providing logic that handles every mutations that might occur on the input, a simulation mechanism is used to automatically simulate an unsupported change (i.e., a change that does not have a respective action to handle it), with one that is supported (i.e., one that has a respective developer-provided action that handles the mutation).

- *Provides an automatic, Incremental-View-Maintenance-inspired algorithm to propagate changes from data sources to the visual layer in a constant time*. This algorithm does

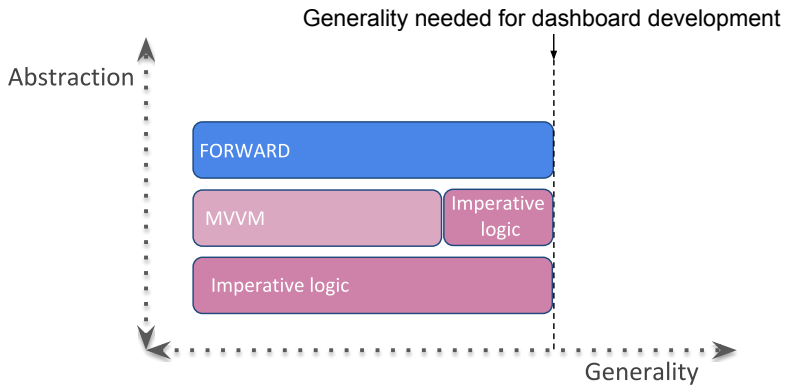


Figure 1.9. Abstraction and Generality in imperative logic, existing MVVM frameworks and FORWARD

not need to fully re-evaluate the template given a new application state and then compare the pre-state with the post-state of the ViewModel. Instead, given Diffs (change-sets) describing mutations on data sources (that could be generated using database triggers), it uses the template, to infer the parts of the ViewModel that need to be updated. To achieve this it navigates the template (in constant time) and identifies dependent template constructs and causes their incremental evaluation. This diff-propagation-based algorithm significantly increases **efficiency** as it reduces the time it takes to propagate changes to the visual layer and as a result leads to dashboards capable of propagating changes almost instantly. At the same time, since this propagation algorithm propagates changes from the sources all the way to the visual layer automatically, it absolves developers from having to perform this task using imperative code, which also improves **developer-friendliness** and **conciseness**.

1.8 The FORWARD Web Application Framework

This Thesis realizes the aforementioned research ideas into a developer-friendly, declarative web application framework called FORWARD. FORWARD simplifies the development of live web visualizations, thus bridging the gap between database developers, data analysts and

advanced web developers, who can now build custom, reactive dashboards regardless of the level of their technical expertise. Simultaneously, it improves the efficiency of more advanced developers thus enabling them to build dashboards that would normally take days or weeks, in only a few minutes or hours. FORWARD shields developers from the intricacies of manual change propagation and manual use of visualization libraries, through complex imperative logic, as it offers a higher level of abstraction, that is sufficient for dashboard development (as shown in Figures 1.8 and 1.9). When using FORWARD, the developer simply specifies what will be displayed on the application pages, during the initial load, using a declarative template language. The template specifies how JSON data coming from data sources are transformed, using template constructs, into appropriate inputs of JavaScript visualization libraries (for instance as we will describe in Chapter 4, the template shown in Figure 4.2 is responsible for the generation of the dashboard shown in Figure 1.1). To shield the developer from the diverse APIs of complex visualization libraries, FORWARD provides *visual units*, which are template constructs, that take as input a JSON value produced by the template (for example, lines 11-23 in Figure 4.1 illustrate the Google Maps visual unit and the respective value that generates the map in Figure 1.1). *Active functions* are template constructs used to obtain data from remote sources and process them, thus absolving developers from having to perform these tasks manually using imperative code. Most importantly, once a developer has specified the declarative templates, FORWARD automatically infers how to efficiently and incrementally propagate the changes from the data sources all the way to the visualizations, using a novel Incremental View Maintenance (IVM)-inspired propagation algorithm.

As a result, FORWARD not only limits the amount of effort and time needed for the development of dashboards (for instance the same dashboard presented above was implemented with FORWARD in only 129 lines-of-code and 2 hours), but also significantly lowers the required technical expertise, thus making the development of custom reactive dashboards approachable to users that would not be able to build custom dashboards (as depicted in Figure 1.10). FORWARD's IVM-based propagation algorithm offers efficient propagation of changes, since the

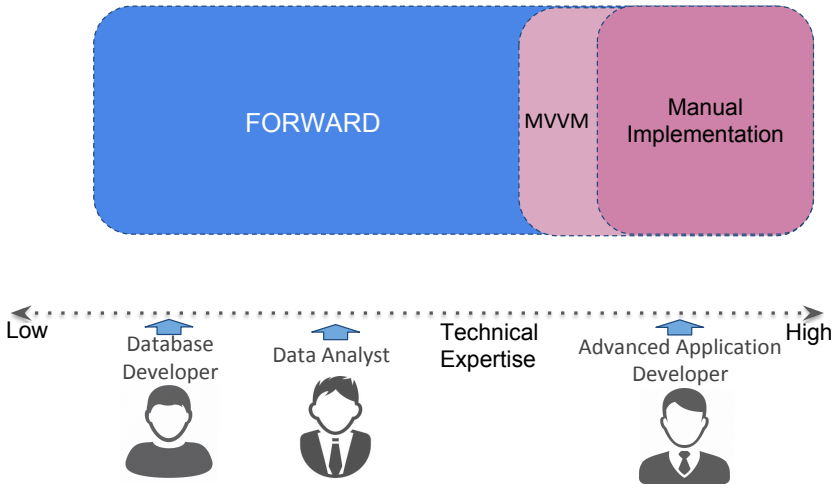


Figure 1.10. Required Technical Expertise for Dashboard Development (including FORWARD)

complexity of the algorithm does not depend on the size of the visual layer (and therefore has constant complexity). This makes it more suitable for dashboards with bigger visual layer. Lastly, in order to enhance FORWARD with new visualizations, we provide a novel trigger-like language that significantly simplifies the creation of reusable visual units, as it allows developers to easily specify logic that describes how the output of the unit is affected by the described input changes. A similar API is also used for the creation of reusable modules that perform data access and data processing, a feature that is not supported by other frameworks. Figure 1.11 qualitative depicts how FORWARD compares against existing MVVM frameworks and manual implementation in the criteria we have established.

1.9 Contributions

To summarize, this Thesis contributes the following novel features:

- *A Declarative template language that enables developers to specify all tasks needed for the development of a dashboard.* Specifically, the template language provides constructs that simplify the process of data access, processing and visualization. Developers can also declaratively specify that the generated dashboard is live without having to provide

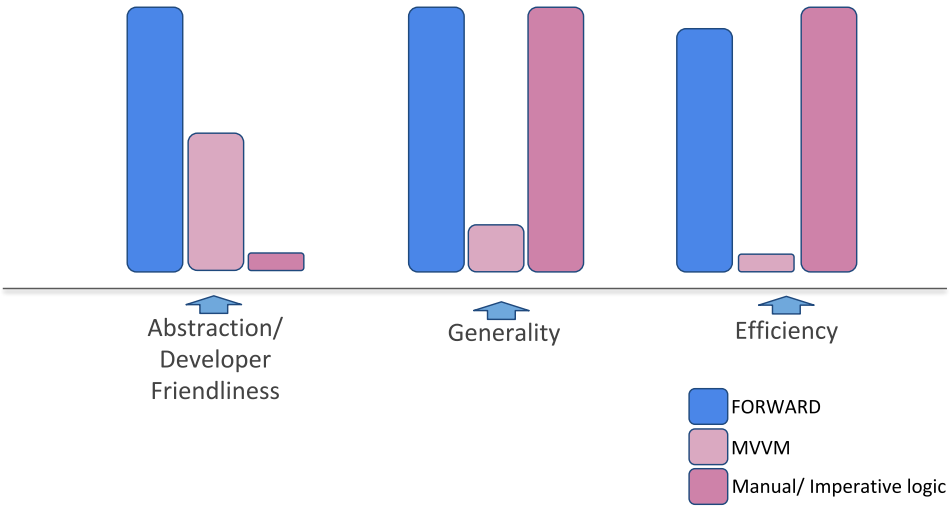


Figure 1.11. Dashboard development using FORWARD vs. other MVVM frameworks vs. Manual (imperative) Code

logic for every mutation that might occur. As a result, this language significantly lowers the technical expertise needed for the development of dashboards, so that even novice programmers that have no experience in web development can build dashboards. At the same time, this language minimizes the amount of time and effort needed by more advanced web developers to build these dashboards.

- *An extensible architecture that enables the introduction of new visual units and active functions.* We provide an easy way for more experienced JavaScript developers to wrap existing JavaScript visualization libraries into reusable visual units that can be used by beginner developers in their applications. To wrap an existing visualization library into a unit, the unit developer specifies how diffs (change-sets) targeting the input of the unit are translated into incremental renderer calls of the underlying visualization library API. To accelerate this process, we offer the following features: (a) a trigger-like language that can be used to describe changes on the input of the visual unit. This language allows the declaration of user provided functions that are responsible for propagating the changes to the visual layer, (b) the input of the visual unit is modeled as a JSON value, which is also the data model used by the APIs (render functions) of JavaScript visualization

libraries, making it very easy to translate an input diff to an incremental renderer call, and (c) the framework allows the “as-you-go" specification of incremental renderers that are leveraged by the framework. As long as the developer provides support for certain input changes, FORWARD automatically simulates the incoming diff with another diff that is supported. As a result, visualization libraries can be wrapped into units considerably faster than in existing frameworks that use declarative template languages, such as AngularJS. This same features can be used for the development of generic active functions that can perform data access to any data source or describe any possible computation.

- *An IVM-inspired change propagation algorithm.* A novel and efficient change propagation algorithm that given a template and diffs describing mutations to Model variables, calls the appropriate incremental procedures of visual units. Since the template is a declarative specification of an application page, the page can be thought of as a view-over-data, and the change propagation algorithm as an incremental view maintenance algorithm. However, in contrast to existing work in incremental view maintenance, our change propagation algorithm operates on the semi-structured JSON data model, instead of the relational data model used by most incremental view maintenance approaches. This means that in addition to relational data, it supports nested and ordered data (in the form of arrays), a problem that does not arise in traditional relational database IVM systems.

- *Diff-propagation based architecture.* A novel architecture based on diffs (change-sets) that allows changes to be propagated all the way from data sources to complex visualization libraries. The architecture contains four of the components we described above, to automatically reflect changes that occur in underlying sources to the visual layer. Specifically, (a) Data source wrappers notify the architecture about changes by providing the appropriate data source diffs when the underlying data source changes. (b) Affected data processing functions that perform data manipulation (such as schema changes, aggregations and other computations), are identified by the architecture and are incrementally evaluated

to produce diffs on the output of the function. (c) The employed change propagation algorithm translates the produced Model diffs to ViewModel diffs, and (d) visualization wrappers in the form of *visual units*, specify how diffs targetting the input of the unit can be translated to renderer calls of the underlying visualization library capable of incrementally updating the visual layer.

- *The FORWARD Web Application Framework.* We contributed an artifact, called FORWARD, that realizes the aforementioned contributions. This artifact is used in a user study demonstrating that it is feasible to rapidly create custom, reactive and live dashboards, even by novice programmers (as shown in chapter 8). Furthermore, the framework is used to evaluate that the novel, IVM-inspired change propagation algorithm, is in fact more performant than the respective algorithms employed by existing framework. Lastly, it is used to demonstrate that more advanced developers can build reusable template constructs (visual units), easier than competing frameworks (as shown in chapter 7.2).

- *The ViDeTTe Notebooks.* We contributed a second artifact, called ViDeTTe that demonstrates the adaptability and generality of FORWARD in use cases that pertain to data science. ViDeTTe is a FORWARD-powered engine that enhances Python Jupyter Notebooks (which are traditionally used in data science) with capabilities that benefit both data analysts and non-technical notebook readers. ViDeTTe enables the generation of visualizations within Jupyter notebooks that are capable of collecting the reader's input and reacting to it. As the (potentially, non-technical reader) interacts with a notebook visualization, ViDeTTe identifies subsequent parts of the notebook that depend on the user's input and causes their reevaluation. By doing this, ViDeTTe offers enhanced data exploratory capabilities to notebook readers, without requiring any coding skills, while at the same time lowering the technical expertise needed from data analysts for the development of reactive notebooks. By seamlessly integrating ViDeTTe into Jupyter notebooks, data analysts can take advantage of the reactive behavior of charts without having to learn

a new programming model or programming language. Instead, with ViDeTTe, they can use Python and the programming model of Jupyter notebooks, they are already familiar with, to add reactive data exploratory functionality to their notebooks.

1.10 Road Map

The remainder of this dissertation is organized as follows:

- In Chapter 2, we describe the state of the art of modern MVVM frameworks and how they evolved to this point. We start by providing an overview of the historic progression and evolution of web and dashboard development and the respective frameworks and programming models that have been used until today. We also describe the MVC (Model-View-Controller) and MVVM design patterns and the programming model used by most modern web development frameworks such as Angular, React and others, and lastly we describe the shortcomings of existing frameworks.

- In Chapters 3-6, we describe our solution to the aforementioned shortcomings. Specifically, Chapter 3 provides a high level overview of FORWARD the framework that realizes the proposed solutions. Chapter 4 describes the developer-facing API and programming model. In Chapter 5 we describe the API we expose to template construct developers (for the development of visual units and active functions) and Chapter 6 describes the internal architecture and algorithms that facilitate the propagation of changes all the way to the visual layer.

- In Chapters 7 and 8 we evaluate the proposed solutions. Specifically, in Chapter 7 we evaluate the performance of FORWARD's change propagation algorithm against the respective algorithms used by other modern web application frameworks. In the same chapter we also evaluate how easy it is to extend the framework with new template

constructs, compared to other frameworks. In Chapter 8, we evaluate the level of developer-friendliness of FORWARD by conducting a user-study, in which three categories of developers: (a) Database Developers, (b) Data Analysts and (c) Advanced Web Developers are asked to implement a custom dashboard using the developer-facing template language we designed.

- In Chapter 9 we describe ViDeTTe, a library that demonstrates the adaptability and generality of FORWARD by applying its features to a new use case. ViDeTTe integrates FORWARD functionality into the widely used Jupyter Notebooks, thus enabling data-analysts and other Python developers, to generate visualization-rich notebooks that react to user input. The observed user input is automatically used to reevaluate dependent cells of Jupyter notebooks, which enables non-technical notebook readers to further explore the underlying data, a feature that is very difficult to implement manually.

- In Chapter 10 we present FORWARD's related work.

- In Chapter 11 we recapitulate the main points of this dissertation.

Chapter 2

Background

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. The downside, however, is that such mechanisms can negatively impact the performance of a given application and cause noticeable irregularities to the user experience.

In this chapter we survey MVVM and Web Component libraries that constitute the state-of-the-art in the web community. This chapter 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, we survey 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 describe the advantages and disadvantages of

each individual mechanism. Lastly, we introduce the concept of Incremental View Maintenance, that inspired FORWARD's change propagation algorithm.

2.1 Background Overview

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 application state and the respective visual layer of the application. Furthermore, as changes occur to 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 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 [63] 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 responsible 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. Lastly, we briefly describe the concept of Incremental View Maintenance (IVM) [50, 53], which inspired the change propagation algorithm of FORWARD. IVM is used extensively in database systems to propagate changes from base tables to materialized views.

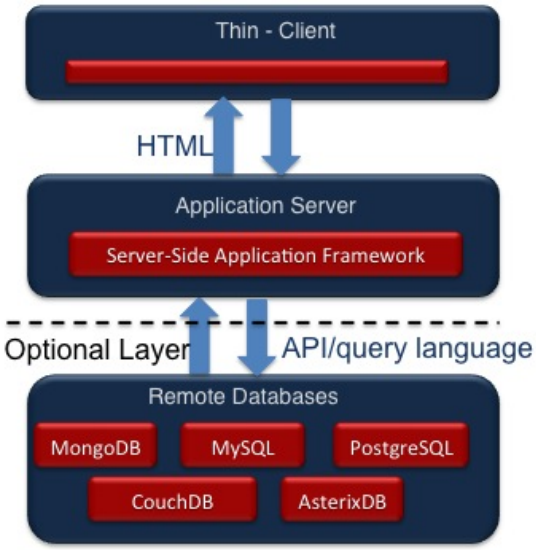


Figure 2.1. Web 1.0 Architecture

2.2 Evolution of Application Development

Web applications have come a long way since the beginning of the WEB 1.0 era ([35, 19]), 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.

2.2.1 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 [35, 17, 9] these definitions are often too abstract and inconsistent, thus leading to severe misconceptions. As far as this dissertation is concerned, Web 1.0 applications typically consist of a client, a server

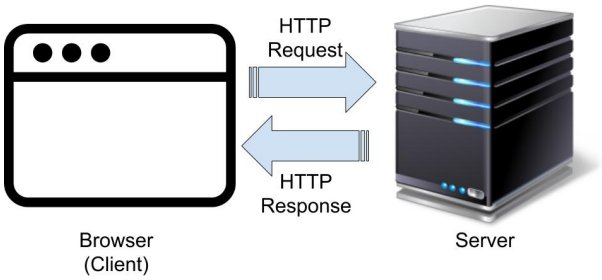


Figure 2.2. Client-Server Interaction

and optionally, a remote database used for persistency (as shown in Figure 2.1). 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.

The life-cycle of a typical WEB 1.0 application requires frequent interactions between the client and the server (as shown in Figure 2.2). 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* [97]. This data structure is mostly [7] 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 with such UI elements they trigger 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*

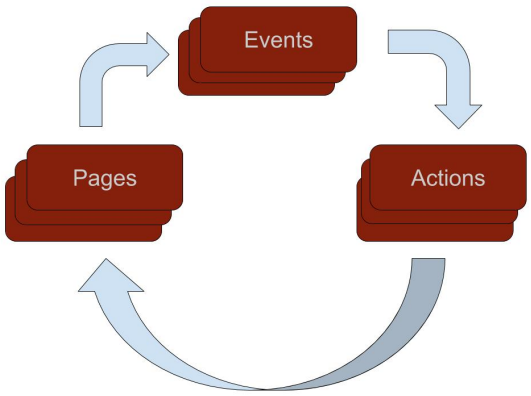


Figure 2.3. Action-Page Cycle

(shown in Figure 2.3), 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 2.1. 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.

2.2.2 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 [85]) on the client [82]. 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 [37]), 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 [91, 1], [36, 59], charts [73, 3] and graphs [43, 39], 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" [79]) 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 the 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* [63].

2.2.3 Design Patterns on the Client (Web 2.0)

In this section, we describe the two main design patterns that are adopted by most opinionated frameworks currently. Due to the nature of this chapter we introduce a new running example that can better depict the design patterns we will cover. The application contains a dashboard that enables easy tracking of delivery trucks that are currently delivering products. This sample application is utilized by a delivery company. The view of the dashboard (shown in Figure 2.4) contains a feed of the location of the company’s delivery trucks on a Google Maps [4] component, followed by an HTML table that contains information about each individual

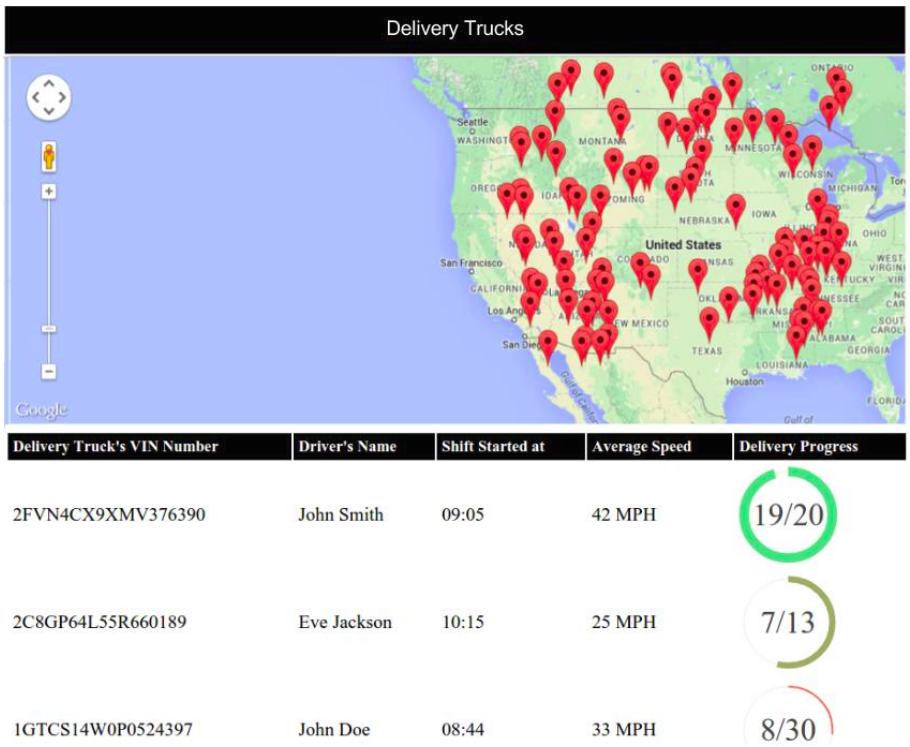


Figure 2.4. Product delivery map

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 [5] 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

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.

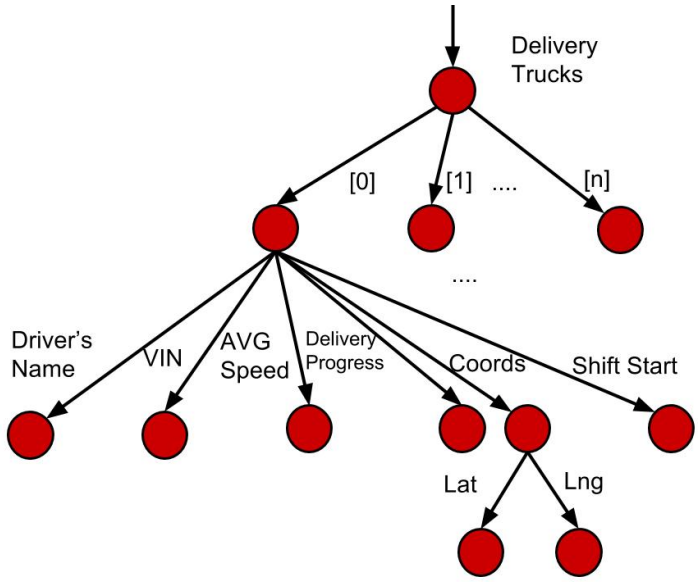


Figure 2.5. Model used by Running Example

- 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 are usually nested we will utilize a tree structure in order to represent them. In Figure 2.5, 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 2.4), such as the VIN number, name of the driver and so on. The View in this application is the entire dashboard shown in Figure 2.4, 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.

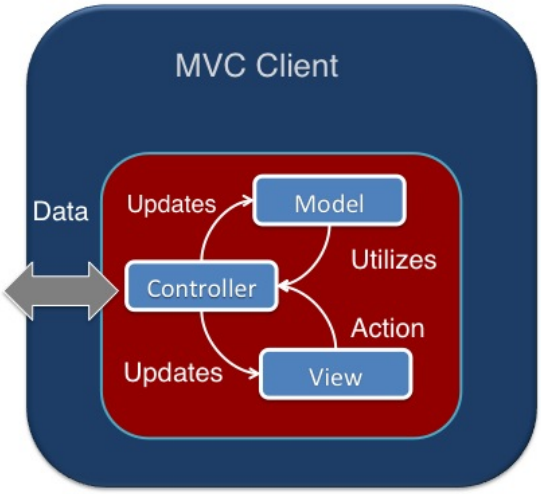


Figure 2.6. Anatomy of an MVC Client-Side Application

In Figure 2.6 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 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.

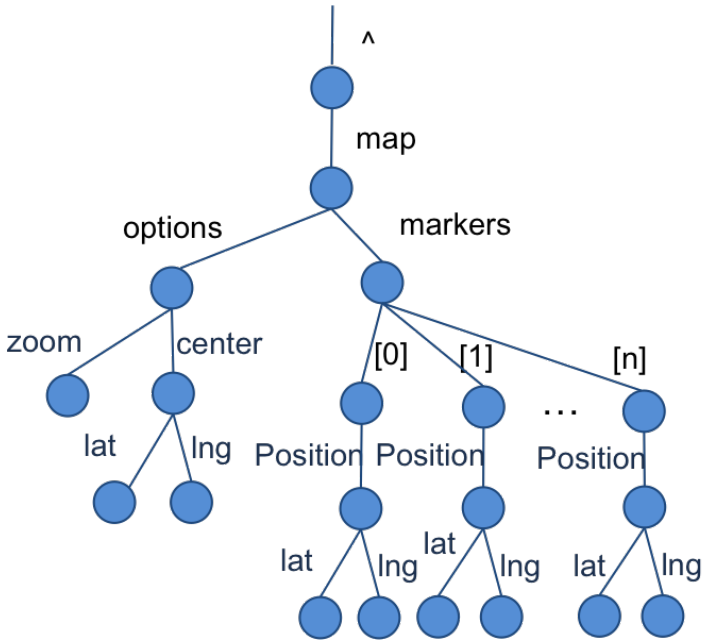


Figure 2.7. ViewModel of Google Maps Component

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 dissertation 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

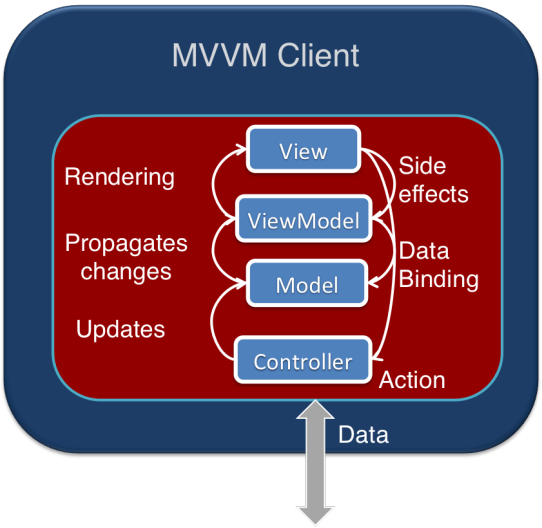


Figure 2.8. Anatomy of an MVVM Client-Side Application

Maps component is the JavaScript object shown in Figure 2.7. Notice that the ViewModel is essentially generated by utilizing attributes that already exist on the Model (which shown in Figure 2.5). 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 they no longer have 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 (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 2.8 we show the internal structure of an MVVM client-side framework.

2.3 Existing MVVM Frameworks

In this section we examine existing frameworks that follow the MVVM design pattern as defined in Section 2.2.3. 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.

2.3.1 AngularJS

Perhaps the most widely used MVVM web framework currently is AngularJS [22]. Angular is mainly supported and maintained by Google, but since it is an open source framework [2] 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.

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

Figure 2.9. Part of the Controller used in running example

```
<html>
   <!-- ... imports and other irrelevant
            parts of the template ... -->
   <div ng-app="truck_delivery" ng-controller="delivery_ctrl">
     <!-- ... other HTML tags ... -->
     <div id="map_container">
      <ui-gmap-google-map
      center="map.center" zoom="map.zoom" bounds="map.bounds">
        <ui-gmap-marker ng-repeat="truck in delivery_trucks"
        idKey="truck.truck_key"
        coords="truck.coords">
        </ui-gmap-marker>
       </ui-gmap-google-map>
     </div>
   </div>
   <div>
     <table>
       <tr> <!-- ... column labels ... --> </tr>
       <tr ng-repeat="truck in delivery_trucks">
            <td> {{truck.VIN}} /td>
            <td> {{truck.driver}} /td>
            <td> {{truck.shift_start_time}} /td>
            <td> {{truck.avg_speed}} /td>
            <td>
              <progressbar type="Circle"
              strokeWidth="10" trailWidth= "1"
              easing= "easeInOut" fromColor="#FC5B3F"
              fromWidth="1" toColor="#6FD57F" toWidth="10"
              numerator = "truck.delivered_items"
              denominator="truck.total_items"
              ></progressbar>
            </td>
       </tr>
     </table>
   </div>
    <!-- ... rest of template ... -->
</html>
```

Figure 2.10. Angular Template `Delivery-Trucks`

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[4], HighCharts[73] 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. Note that the code that appears in the controller is imperative (using JavaScript). The *scope* object is a crucial part of AngularJS internal architecture as we will describe later in this section. 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 tool-sets, 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 2.9 and 2.10, we show a small snippet of the controller and the template language that is used to generate the running example shown in Figure 2.4. The snippet in Figure 2.9 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 $delivery\_trucks$

Figure 2.11. AngularJS Watcher - Building Blocks

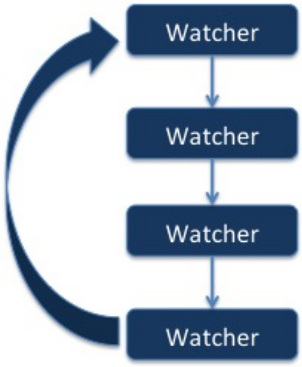


Figure 2.12. AngularJS - Digest Cycle

and then attach this variable to the scope object. In Figure 2.10 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 2.4; 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 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 2.11) 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 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 2.12). 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

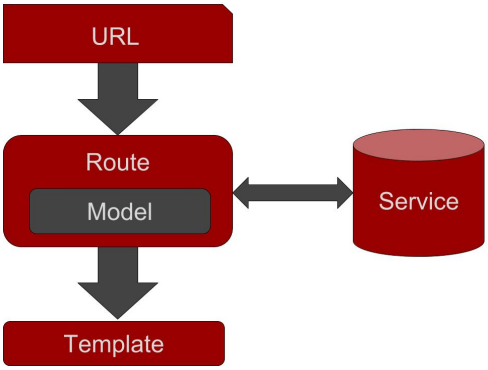


Figure 2.13. Ember's Programming model

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

Figure 2.14. EmberJS - Route Example

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 $O(wd)$.

2.3.2 EmberJS

Another framework that follows in the footsteps of Angular is EmberJS [44]. Similarly

```
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 2.15. EmberJS - Model Definition

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 2.13, 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 2.14, 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 2.15, 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 [6] 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 2.15, 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.

2.3.3 KnockoutJS

Knockout[10], 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 [76]) 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 [11] 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.

2.4 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 [16, 18]. These Components are an attempt to bring component-based software engineering [60] 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

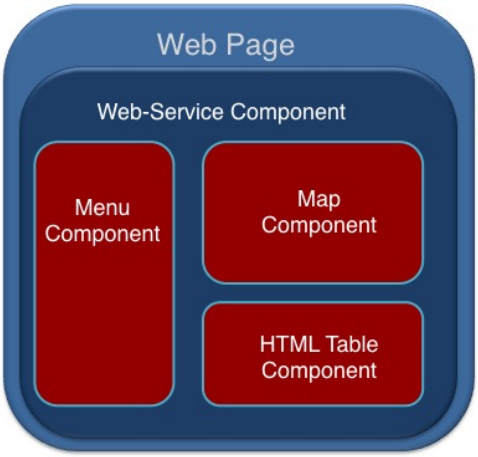


Figure 2.16. Component Based Architecture

(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**[15]. 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.

2.4.1 Polymer

Perhaps the most typical example of a Component library is PolymerJS [14], which is

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

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

```

Figure 2.17. Custom Polymer Element

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 effectively generate modern applications in a modular manner. For instance, in Figure 2.16 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 2.17, 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 2.17), 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.

2.4.2 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 2.18 and 2.19 show how a React Component can be implemented imperatively and declaratively. In lines 2-10 in Figure 2.18, 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 2.19, 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 2.18 and in lines 10-12, in Figure 2.19, 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 2.18. 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 2.19. 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

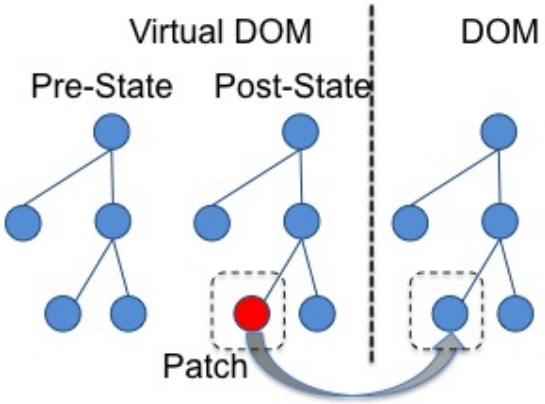


Figure 2.20. Virtual DOM Diff-ing

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 2.20).

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 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.

2.4.3 MithrilJS

The last Component library that will be included in this comparison is MithrilJS[12]. 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 life-cycle 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 [8] is used instead. Another difference is that Mithril does not support declarative templates for specifying the View of a component. However, there are 3rd-party libraries (such as MSX [13]) that allow the use of declarative logic for that purpose.

2.5 Incremental View Maintenance (IVM)

In this section we will briefly describe the problem IVM algorithms solve. FORWARD uses an IVM-inspired algorithm in order to enable efficient change propagation from the un-

derlying sources to the visual layer FORWARD. Note that existing IVM solutions could not be applied to FORWARD's change propagation algorithm, due to differences in the data-model (relational vs. semi-structural-nested) as well as more practical differences, such as the fact that the end consumer of existing IVM is relational tables, while the end consumer of the propagation algorithm is JavaScript renderers that update the visual layer. We will describe the differences in higher detail in Chapter 6.

In database systems, 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 a variety of database tables or even tables 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 2.21), 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.

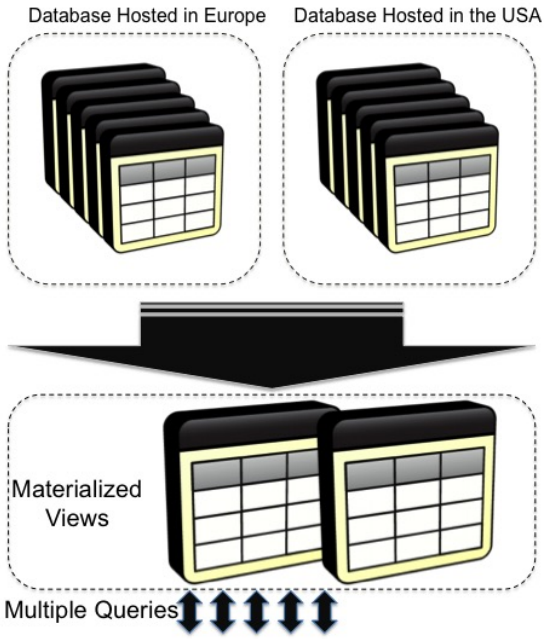


Figure 2.21. Materialized Views

FORWARD 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.

A major difference between existing IVM algorithms and the algorithm that was designed and developed for use in FORWARD, is that the former algorithms only operate on relational schema. Most web applications however employ a Model or ViewModel that is heavily nested and in many cases completely unstructured. One technique that is often used in cases when IVM must be performed on nested views is the relational decomposition of the view into a normalized form. This technique, however, can lead to increased memory footprint (because of replications). More interestingly, performing relational decomposition on heavily nested data, every time a mutation needs to be propagated to views is a very time-consuming process and therefore would not be realistic to perform for real-time applications. Instead, we designed and implemented a novel algorithm that performs change propagation on heavily nested views created by the

template language we provide.

2.6 Summary

In this section, we described the historical evolution of web and dashboard development, starting from simple monolithic applications that rely heavily on a middleware for data access and data processing all the way to modular applications that are developed using modern MVVM web application frameworks. We also provided a detailed analysis of the differences between the MVC and MVVM paradigms, that are typically used for application development, demonstrating their advantages and disadvantages. We provided an in-depth demonstration of the individual programming models of most modern web application frameworks such as Angular, React and more, and described their limitations with regard to developer friendliness, extensibility and efficiency while propagating changes to the visual layer. In the next chapter we provide an overview of FORWARD and show how it overcomes the limitations of current MVVM application frameworks.

Acknowledgements

This chapter contains material adapted from the technical report: "**In-depth Survey of MVVM Web Application Frameworks**", by Konstantinos (Costas) Zarifis. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Overview of FORWARD

FORWARD is an advanced MVVM web application framework that resolves many of the pain-points of the frameworks we described in the previous chapter. FORWARD provides features that pertain to both (a) application developers that wish to build web dashboards, and (b) template construct developers that wish to extend the framework with new template constructs, that perform access to new data sources, new data processing operations or generate visualizations using new visualization libraries. To application developers, FORWARD offers the ability to be used both as a (modular) full-stack framework and as a client-side framework in an existing application stack. When FORWARD is used as a full-stack framework, it allows developers to employ its template language to describe data access, data processing and data visualization declaratively, thus absolving them from having to write complex imperative logic for these tasks. When FORWARD is used as a client-side framework, in an existing application stack, it allows developers to use the template language to generate the application's visual layer declaratively, using the existing application state. In both cases, FORWARD offers a change propagation algorithm, inspired by incremental view maintenance techniques, that automatically and, most importantly, efficiently propagates changes to the visual layer without requiring explicit imperative logic by dashboard developers. For template construct developers, FORWARD provides an easy to use API that enables them to build reusable constructs that perform data access, data processing and data visualization. The provided API simplifies many of

```
1  <% template delivery-truck-tracking %>
2   <% import functions %>
3   <% import actions %>
4
5   <% let delivery_trucks = sql(
6       'SELECT latitude, longitude, VIN, driver,
7              shift_start_time, avg_speed,
8              delivered_items, total_items
9       FROM   delivery_trucks_table dtt,
10             product_delivery_truck_relation r,
11             products p
12      WHERE  p.name = '?'
13             AND dtt.id = r.t_id AND p.id = r.p_id', [product_name]).onChange(redisplay())
14  %>
15  <% html %>
16    <div>
17      <% unit Google-Maps %>
18      {
19        options : {
20          zoom: 10,
21          center: { lat: -25.363882, lng : 131.044922 },
22        },
23        markers : [
24          <% for truck in delivery_trucks %> {
25            position : {
26              lat : <% print truck.latitude %>,
27              lng : <% print truck.longitude %>
28            }
29          }
30        ]
31      }
32      <% end unit %>
33    </div>
34    <div>
35      <table>
36        <tr> <!-- ... column labels ... --> </tr>
37        <% for truck in delivery-trucks %>
38           <tr>
39             <td> <% print truck.VIN %> /td>
40             <td> <% print truck.driver %> /td>
41             <td> <% print truck.shift_start_time %> /td>
42             <td> <% print truck.avg_speed %> /td>
43             <td>
44                <% unit ProgressBar %> {
45                  type = 'Circle',
46                  value : {
47                   numerator :<% print truck.delivered_items %>
48                   denominator : <% print truck.total_items %>
49                  }
50                 }
51                 <% end unit %>
52             </td>
53           </tr>
54          <% end for %>
55      </table>
56    </div>
57  <% end html %>
58 <% end template %>
```

Figure 3.1. Template `Delivery-Trucks`

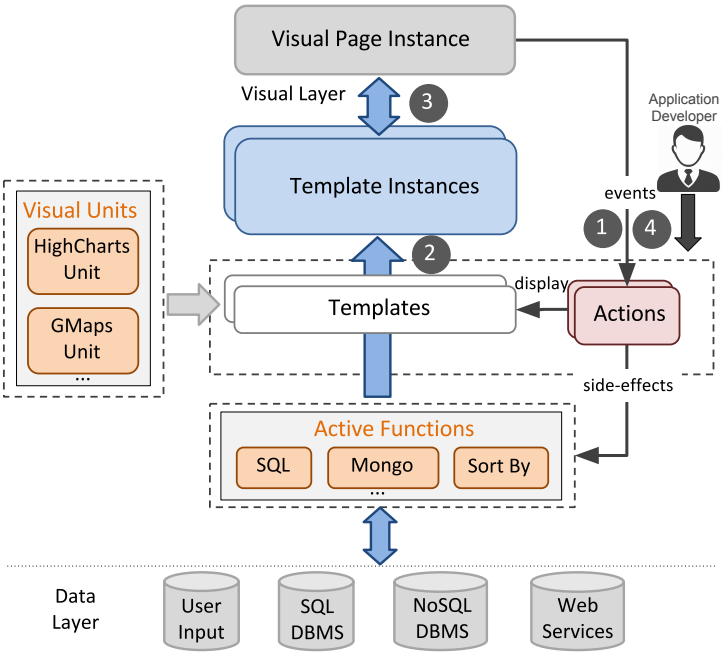


Figure 3.2. Application developer's programming model. Actions and templates are the pieces that are provided by the developer.

the tasks that in other frameworks have to be performed manually, such as identifying individual mutations in the ViewModel, caching elements that facilitate incremental rendering and more.

3.1 FORWARD as a full stack framework

When using FORWARD as a full-stack framework, developers can specify the application state (model) by employing active functions. Active functions, are template constructs that automatically resolve impedance mismatch issues that developers typically have to deal with when developing full stack applications. For instance, converting database rows, XML data (and other data models used by database systems and web services) to JSON datasets (which are typically used in web applications), while not particularly difficult, is an extra step that developers have to deal with when building web applications, which adds to the amount of code and the architectural complexity of the application. Active functions also offer location transparency, since the application developer is able to utilize datasets that may reside in the middleware or in

a database as if they were located on the client-side. This means that the technical expertise that is needed for the development of web applications and dashboards is significantly reduced, as developers do not need to coordinate and exchange data between various application tiers. Lastly, FORWARD's IVM-inspired change propagation algorithm significantly improves the efficiency of applications when automatically propagating changes from the data sources all the way to the visual layer. As a result, developers do not need to write imperative code to propagate changes for every mutation that might occur in the underlying data or spend any time optimizing the change propagation, which again significantly limits the amount of time and code that is needed.

3.1.1 The Programming Model of FORWARD

FORWARD's programming model (shown in Figure 3.2) comprises Templates and Actions. The template is able of accessing data from a plethora of sources. Once data access has been completed the obtained data are stored into template variables (also called VDB variables). These template variables are the Model in MVVM (or application state, as it has been described in the previous sections). Notice that in stark contrast to existing MVVM frameworks the Model in FORWARD is instantiated declaratively and not through imperative code that typically is written in a Controller (although this option is also supported by FORWARD as we will describe later in this Chapter). The declarative Template is then 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 other side-effects or trigger the evaluation of a different template.

3.1.2 Declarative specification of visual layer, data access and data processing

FORWARD, like most other MVVM frameworks, provides reusable template constructs that generate visualizations. These constructs are called *Visual Units*. Visual Units can be utilized

by application developers within a template to declaratively specify a visual layer that contains visualizations (such as charts and maps) and textual content (using HTML). Similarly to Visual Units, Active Functions are reusable template constructs that can perform data access and data processing. Active Functions can be utilized by application developers within a template to declaratively obtain data from a variety of sources (Database Systems and Web Services) or process data that are assigned to template variables (Model). Due to Active Functions the Model of FORWARD applications can be instantiated declaratively.

Example 3.1.1 *In Figure 3.1 we show the FORWARD template that is used to generate the View shown in Figure 2.4 (which was used to demonstrate the features and the programming models of existing MVVM frameworks in the previous Chapter). The declarative specification of the application's Model is shown in lines 5-14. In these lines, we employ an sql active function to obtain data from a remote database, and assign them to the variable delivery_trucks. 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 in the underlying database. In lines 17-36 we use a Google Maps unit to generate the map visualization shown in Figure 2.4 and in lines 39-65 we instantiate the HTML table shown bellow the map. Lastly, in lines 48-61 we instantiate the progress visualization that is visible in each row of the HTML table.*

Note that FORWARD was able to generate the entire visual layer entirely declaratively, without requiring any code in JavaScript or some other imperative language. Additionally, due to the statement `onChange(redisplay())`, shown in line 13, the visual layer will be incrementally updated when any mutation (change) is observed in the underlying data source, again without requiring any imperative logic.

3.2 Template construct extensibility

FORWARD enables the extension of the framework with new visual units and active functions, as it provides an easy to use API that can be used by template construct developers to integrate new visualization libraries, new data sources and new data processing functions. The provided API allows the declaration of sets of *delta functions* that describe how changes to the construct's input result to changes to the output (which is the visual layer for visual units and template variables for active functions). Each delta function is declared by specifying a $diff$ *signature*. Diff signatures are trigger definitions that uniquely dictate which delta function will be invoked given particular changes on the construct's input. Chapter 5 provides more details about the extensibility aspect of FORWARD.

3.3 Using FORWARD alongside another web framework, or as a purely client-side framework

FORWARD can also be used alongside other web frameworks or as a client-side framework. This enables developers that already have an application stack in place to use the features supported by FORWARD without rewriting the entire application stack. If developers have an existing middleware, they can create active functions that directly connect to that middleware thus allowing the use of the declarative template language for data access, data processing and visualizations. Alternatively, if the application stack also contains a data access and data processing layer on the client, they can use FORWARD as a purely client-side framework, thus taking advantage of the declarative specification of the visual layer (View in MVVM) and the automatic incremental maintenance of the visual layer it offers.

When FORWARD is used as a client-side framework, application developers can instantiate the Model (template variables) within the action (using JavaScript) and then use the declarative template to construct the visual layer, similarly to how application development works in existing frameworks (in this case, note that actions essentially, act as Controllers). As

mentioned, FORWARD utilizes IVM techniques that propagate diffs throughout the application. When FORWARD is used as a purely client-side framework, in which case the application state is defined manually, through imperative logic, the application developer can either use FORWARD's diffing module that compares the pre-state with the post state of a particular variable, or manually construct diffs that target the local application state, using the API FORWARD provides. Manually generating diffs is typically more efficient than using the automatic diffing module. Once the diffs have been created they are passed on to FORWARD and eventually lead to the respective incremental 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 they have to manually generate diffs or invoke the diffing of template variables.

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 “diffing" algorithm on the entire Model. 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 “diffing" approaches that component libraries perform on instances of the Virtual DOM. Despite the similarities, this “diffing" algorithm has several advantages, since it is run on the Model instead of the ViewModel. The complexity of a “diffing" 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 is typically 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 “diffing" 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 and MVVM frameworks.

3.4 Summary

In this chapter, we provided an overview of FORWARD. We described its developer friendly Programming model and the expressiveness of the FORWARD templates as well as the extensibility it provides by supporting the addition of new template constructs that can perform data access, processing and visualization. Additionally, we mentioned that FORWARD can be used both as a client-side or full-stack framework and can also be used alongside existing frameworks. In the next chapter, we discuss the developer facing programming model and API in greater detail.

Acknowledgements

This chapter contains material adapted from the technical report: "**In-depth Survey of MVVM Web Application Frameworks**", by Konstantinos (Costas) Zarifis. The dissertation author was the primary investigator and author of this paper.

Chapter 4

FORWARD Application Development

Recall that one of the contributions of FORWARD is that it makes dashboard development easy for programmers, regardless of their technical expertise, in this Chapter we present the developer facing structure and semantics of a FORWARD application. The part of the architecture seen by template construct developers (i.e., visual unit and active function developers) and the respective API they use to extend the framework with new template constructs is the topic of Chapter 5. In order to illustrate all the developer facing features of FORWARD we will be using the running example we described in the Introduction (Section 1.1). Figure 1.1 shows the visual layer of said dashboard. Note that this dashboard is different from the one used in the previous two Chapters, this is because, the dashboard shown in Figure 1.1 is more elaborate and therefore it allows us to illustrate all the developer-facing features of FORWARD.

4.1 FORWARD Developer Oriented architecture

Figure 3.2 summarizes the conceptual, architecture of a FORWARD application as it is perceived by the developer. In order to build web applications, developers have to provide: (a) declarative *templates*, that describe how data contained in sources can be used to generate web pages and (b) *actions*, which are procedures that can cause side-effects to the application. Such side-effects may include specifying the template that will be evaluated next, sending data to remote services and more.

```
1  <% unit html %>
2   <div>
3    <table>
4     ... <!-- table headers --> ...
5     <tr> <td> USA </td> ... </tr>
6     <tr> <td> Greece </td> ... </tr>
7     ...
8    </table>
9    <div>
10  </div><div>
11   <% unit Google Maps %>
12   {
13    markers: <
14     { lat: 55, lng, 54, icon: 'Red'
15       infowindow:<% unit Highcharts %>... },
16     { lat: 34, lng, 66, icon: 'Red'
17       infowindow:<% unit Highcharts %>... },
18     { lat: 67, lng, 44, icon: 'Green',
19       infowindow:<% unit Highcharts %>... },
20     ...
21    >
22   }
23   <% end unit %>
24  </div>
25 <% end unit %>
```

Figure 4.1. Template instance

4.1.1 Template Instance

The evaluation of a template results in the creation of a *template instance*, which is an abstraction of the visual layer of an application. It is essentially the dataset that describes the visual layer (also known as ViewModel). Figure 4.1, shows a part of the template instance that generates our running example (BNF appears in Figure 4.3). A template instance, typically contains instances of visual units. A *visual unit instance* specifies the name of the employed visual unit and the value representing its state (also known as unit state). Lines 1-10, contain an HTML visual unit instance[1] that generates the table showing the countries from which visitors originate along with a nested list that contains information about each individual user. The table is displayed on the left side of the page in Figure 1.1. Lines 11-23, in Figure 4.1, show a Google Maps visual unit instance generating the map visualization of the running example, the unit state includes the markers and other visual components contained in the map. Line 14 is responsible for generating the marker that appears below the pop-up window in Figure 1.1. Some visual

[1]For simplicity, an HTML unit state is represented as an HTML string.

```
1  <% template user_activity %>
2    <% unit html %>
3     <% let visitors = sql(`
4         SELECT latitude, longitude, country
5         FROM Users u
6         WHERE  active = true` ).onChange({redisplay()}) %>
7     <% init typed_country = '' %>
8     <% let countries =
9             sort_by( apply( nest_by(visitors, 'country', 'vs'),
10                             count('vs'), 'total'),
11                     'total', 'descending') %>
12     ...
13     <% unit html.text_input %> {
14             text : <% bind typed_country %>
15             onChange : <%  redisplay() %>
16     }
17     ...
18     <table> ...
19       <!-- Displaying an HTML table with countries in
20            desc order -->
21       <% for group in countries %>
22        <% if typed_country == '' | matches(group.country, typed_country) %>
23         <tr>
24          <td> <% print group.country %></td>
25          <td> <% print group.total %>
26          <% unit html.checkbox %> {
27            checked : <% bind group.selected = true %>
28            onClick : <% action register_action(group) %>
29          }
30           </td>
31          ...
32         </tr>
33        <% end if %>
34       <% end for %>
35     </table>
36    </div>
37     ...
38   <div>
39    <% unit Google Maps %>
40    { markers: <
41        <% for country in countries %>
42         <% if country.selected %>
43          <% for visitor in country.visitors %>
44            {
45              lat: <% print visitor.location.x %>,
46              lng: <% print visitor.location.y %>,
47              icon: <% print colorize(country) %>,
48              infowindow : <% unit highcharts %> ...
49             }
50           <% end for %>
51          <% end if %>
52         <% end for %>
53      >
54     } <% end unit %>
55     </div>
56  <% end unit %>
57 <% end template %>
```

Figure 4.2. Template `user_activity`

1	*template_instance*	→	*unit_instance*
2	*unit_instance*	→	**<% unit** *unit_class* **%>** *tuple* **<% end unit %>**
3	*value*	→	*json++_value*
4		|	*unit_instance*
5	*json++_value*	→	*scalar* | *tuple* | *array* | *bag*
6	*scalar*	→	*string* | *number* | *boolean* | null
7	*tuple*	→	**{**(string **:** *value* (**,**string**:***value*)*)?**}**
8	*array*	→	**[**(*value* (**,***value*)*)?**]**
9	*bag*	→	**<**(*value* (**,***value*)*)?**>**

Figure 4.3. BNF Grammar for Template Instances that contain `JSON++` visual units

units also support the display of nested visualizations. In lines 15, 17 and 19 we show 3 nested highcharts visual unit instances attached to the “infowindow" attribute of the google maps unit state. These unit instances describe nested visualizations (charts) that will appear in infowindows (popups) when a user clicks on the respective markers, line 15 is responsible for generating the pop-up window shown in Figure 1.1.

4.1.2 JSON++ Data Model

The parts of the unit state that describe a single visualization (excluding nested unit instances) follow the **`JSON++`** data model (BNF appears in Figure 4.3), which is an extension of the (commonly used in web applications and third-party visualization libraries) JSON data model with added support for unordered collections (bags) and visual unit instances (that declaratively generate visualizations). Bags are useful for the representation of elements the order of which is irrelevant. For instance, since the order in which markers appear on a map is not significant, a bag is used to represent the `markers` collection, as shown in lines 13-21 in Figure 4.1.

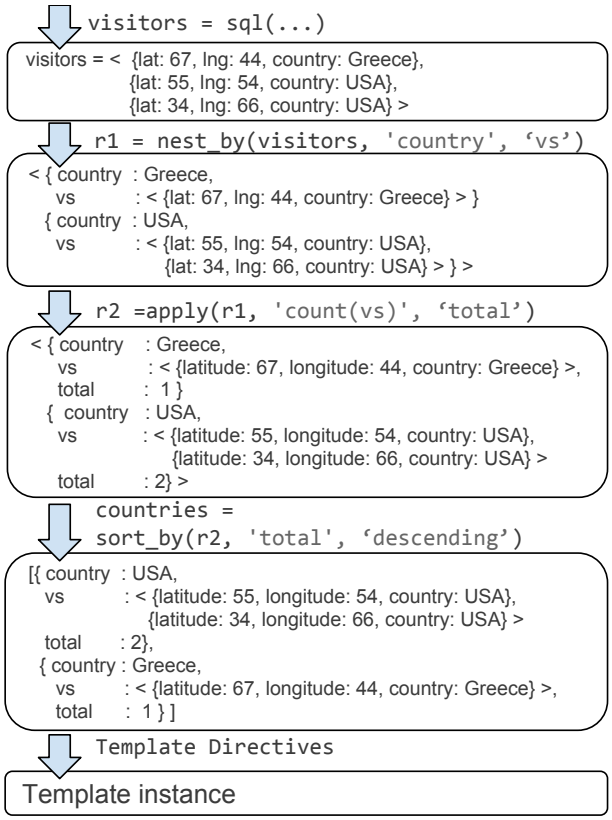


Figure 4.4. DAG of views specified in template

4.1.3 FORWARD templates

Templates are declarative specifications that describe the template instance as a view over data. Specifically, a template describes (a) accessing data from sources, via *non-deterministic functions* (b) potentially processing the obtained data (such as performing aggregations) and creating views with the result, via *deterministic active functions*[2] and (c) constructing the template instance. The *template directives* transform the results of active functions into a template instance. The template as a whole, essentially, describes a DAG of views, with the source being the data originating from data sources and the sink being the template instance. Figure 4.4 shows this DAG of views for our running example. The `sql()` active function obtains source data, the

[2]The precise definition of the distinction between non-deterministic functions and deterministic functions is given in Section 3

1	*template*	→	**<% template** *template_name* **%>**
			(*let* | *init*)*
		|	*unit*
			<% end template %>
2	*unit*	→	**<% unit** *unit_class* **%>**
			tuple
			<% end unit %>
3	*value*	→	*json++_value*
		|	*unit*
		|	*print*
		|	**[** *for** **]**
		|	**<** *for** **>**
		|	*if*
4	*json++_value*	→	*scalar* | *tuple* | *array* | *bag*
5	*scalar*	→	*string* | *number* | *boolean* | null
6	*tuple*	→	**{**(string **:** *value* | *bind* | *action* (**,** string **:** *value* | *bind* | *action*)*)?**}**
7	*array*	→	**[**(*value* (**,***value*)*)?**]**
8	*bag*	→	**<**(*value* (**,***value*)*)?**>**
9	*let*	→	**<% let** *var_name* **=** *expr* **%>**
10	*init*	→	**<% init** *var_name* **=** *js_expression* | *json++_value* **%>**
11	*print*	→	**<% print** *expr* **%>** | **<%=** *expr* **%>**
12	*for*	→	**<% for** *var_name* **in** *json_path* **%>**
			value | *for*
			<% end for %>
13	*if*	→	**<% if** *expr* **%>**
			value
			(**<% elif** *expr* **%>**
			value)*
			(**<% else %>**
			value)?
			<% end if %>
14	*bind*	→	**<% bind** *var_name* (= *js_expression* | *json++_value*)? **%>**
15	*action*	→	**<% action** *action_name* **%>**
16	*expr*	→	*js_expression*
		|	*active_function*
		|	*json_path*

Figure 4.5. BNF Grammar for Templates

```
1  register_action(group){
2    sql(`INSERT INTO sys_admin_activity
3        VALUES (user.id, NOW(), group.country,
4                group.active)`)
5    FORWARD.displayTemplate('user_activity');
6  }
```

Figure 4.6. `register_action()` action

deterministic active functions (`nest_by()`, `apply()`, etc.) process them and the template directives construct the template instance shown in Figure 4.1.

Template Directives. The supported set of template directives, defines a syntax similar to well-known template languages and provides the expressiveness of a functional programming language without recursion. As shown in the BNF (Figure 4.5) the FORWARD template language consists of seven template directives: **`<% print %>`**, **`<% for %>`**, **`<% if %>`**, **`<% let %>`**, **`<% init %>`**, **`<% bind %>`**, and **`<% event %>`**. These are used to describe computation, define variables, set up data collection and specify events. We next describe each of them in detail.

View Variables and View Expressions. A template may define variables, which are essentially views over data.

The **`<% let`** x **`=`** E **`%>`** directive defines the template variable x. During instantiation of the template, the value of x will be the result of E. Technically, x is a view over E and as we will see later, when the template is redisplayed the system avoids its full reevaluation when the data, on which E depends on, change (instead FORWARD incrementally updates x). The expression E can be an invocation of an active function or a path that uniquely identifies a `JSON++` value. The template in Figure 4.2 employs a **`let`** directive in lines 3-6 in order to create a variable (view) containing the users that are currently visiting the website by invoking the '`sql`' active function, and another **`let`** directive in lines 8-11 which invokes a sequence of active functions that process the obtained data (Figure 4.4 depicts this process) and assigns the result to the variable "`countries`".

Reporting syntax and semantics. The directives **`print`**, **`for`** and **`if`** can be used to describe the template instance as a view over template variables. The template instance's data-

model is not pure `JSON++` but also contains visual unit directives (as seen in Figure 4.1 in lines 1, 11, 15 and so on) and can contain HTML content[3] if HTML visual units are employed in the template (as seen in Figure 4.1 in lines 2-10). The provided template directives describe a language that while declarative, mimics the look and feel of well-known scripting languages (such as Python and JavaScript).

The **<% for** *x* **in** *E* **%>** *B* **<% end for %>** directive specifies that variable *x* iterates over the result of the expression *E*. In each iteration, the body *B* of the **for** loop is instantiated. Lines 41-52, in Figure 4.2, show two **for** directives that iterate over two nested collections (countries, in line 41 and individual visitors, in line 43), in order to generate the collection of `JSON++` tuples that describes the Google Maps markers.

The **<% *if*** *E* **%>** *B* **<% *end if* %>** *condition directive*[4] specifies that once the expression *E* evaluates to true, the respective body *B* will be instantiated. Line 42, in the displayed template, shows an if-statement that checks if the current country is selected. If it is, the body of the conditional expression (lines 43-50) will be instantiated thus creating the respective markers on the map.

The **<% print** *E* **%>** produces an instance of the expression *E*. Lines 45-47 illustrate the use of **print** directives to generate the values of the attributes: `lat`, `lng`, and `icon` for each marker.

Collecting data. The template's *"string" :* **<% *bind*** *var_name* **%>** directive allows the developer to specify data collection from the page (UI). Specifically, the *attr* : **<% bind** *x* **%>** directive describes that the template instance attribute *attr*, will be bound to the input variable *x*. As a result, if the *attr* unit state variable gets updated due to the user's interaction with the visual layer, then the variable *x* will also reflect this update. For instance, the template of our running example (line 27) uses a **bind** directive to create a binding between the variable `selected` (created for each group) and the value returned by the attribute `checked` (which returns the

[3]template directives can also be embedded in HTML DOM elements to dynamically generate HTML content
[4]Note that **else** and **elif** statements are also supported, as seen in the BNF shown in Figure 4.5.

values “true” and “false” depending on whether the checkbox is checked or not, as a result of user interaction). The **bind** directive can also contain an optional component “= *E*” that allows the developer to specify an expression *E* (which can be an invocation of a function or any other JavaScript expression or JSON++ value), the result of which will be assigned to the variable *x* when *x* is first instantiated (it is essentially, syntactic sugar of the **init** directive described below). For example, in our running example all the checkboxes appear checked after the initial evaluation, since the template in line 27, specifies that they are instantiated with the value “true”.

The **<% init** *x* **=** *E* **%>** directive initializes the value of *x* to be the result of *E* during the initial evaluation of the template. If during the life-cycle of an application this value changes (for instance, as a result of user interaction/input due to the **bind** directive), FORWARD will maintain the updated value between reoccurring template instantiations. The expression *E* can be an invocation of a function or any other JavaScript expression or JSON++ value. The template in Figure 4.2 employs an **init** directive in line 7 to create a variable typed_country (initially empty), which will be used later in an input field to collect user input. Note that the **let** directive defines an automatically maintained view (typically over data sources), while **init** simply initializes a variable that can be updated only by the **bind** directive.

Setting triggers that invoke actions when UI and non-UI events occur. Developers can specify the invocation of actions in response to UI events through the use of the *event_name* : **<% action** *a* **%>** trigger directive. Specifically, this directive specifies that whenever UI event *event_name* occurs, action *a* is executed. The **action** directive, shown in line 28, specifies that the action “register_action” will be invoked when the user clicks on a checkbox. Developers can also link non-UI events to actions. Specifically, template (view) variables (similarly to database tables) support trigger-events which are activated in response to changes on their values. For instance, in line 6 on the template, an ‘onChange’ trigger-event is installed to the variable ‘visitors’. Once this variable is modified in any way, the encompassed action is invoked (in this case, this action is the ‘redisplay()’).

Actions are procedures invoked in response to events. Such procedures can cause side-

effects to external systems (such as storing data to a remote database) or to the application itself (such as specifying the template that will be displayed next). Typically, side-effects to external systems are caused by actions through the invocation of active functions. For instance, the action shown in Figure 4.6, employs the '`sql`' active function to store the recent user activity (i.e., the user clicking on a checkbox) to a remote database. In visualization applications, such as the running example, actions can also dictate the template that will be evaluated next. For simple read-only, 1-page dashboards, that contain no side-effects other than redisplaying the same template, developers do not need to specify any custom actions, instead they can describe this behavior through the system-provided `redisplay()` action, as shown in line 6 in template 4.2.

Life-cycle of FORWARD Applications. FORWARD applications follow an *action-template cycle* in which actions activate templates that produce visual pages. More specifically, an action-template cycle consists of the following steps (denoted by numbers in Figure 3.2): An event activates an action (step 1) which causes side-effects to the application, such as storing data to a database (typically contained in template variables, which can be accessed by actions) or choosing the template that will be displayed next. If a template is specified, FORWARD proceeds by evaluating it (step 2), thus producing the *template instance* . Then FORWARD automatically produces a new visual page instance by evaluating the template instance (step 3). Finally, as new events occur, they can trigger subsequent actions (step 4), that lead to the repetition of the action-template cycle. Note that, if during this cycle, an already evaluated template is reinvoked, FORWARD avoids its full reevaluation, instead, it infers and incrementally applies mutations to the previous state of the template instance and visual page instance, thus efficiently bringing them up-to-date.

4.2 Acknowledgements

This chapter contains material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 5

FORWARD Extensibility & Generality

Recall that one of the goals of this dissertation is to make it easy for novice developers to build dashboards quickly and efficiently by abstracting implementation details that make dashboard development so taxing. FORWARD does this by offering a declarative template language, that aside from the directives we covered in the previous chapter, also provides template constructs that can declaratively perform operations required during dashboard development (such as data access, data processing and data visualization), thus providing higher abstractions that eliminate the need for low-level imperative logic, while simultaneously lowering the required technical expertise needed for dashboard development. Additionally, FORWARD supports the extension of the framework with new such template constructs. This chapter describes how construct developers (that are fairly experienced in JavaScript) can introduce their own reusable template constructs, which can then be imported and employed by the (potentially novice) dashboard developers to build any kind of dashboard.

Recall also that FORWARD is capable of propagating changes that occur in underlying sources to affected template constructs, thus causing their incremental evaluation, this means that these template constructs, apart from their initial (full) evaluation, must also be capable of incrementally producing changes to their output given mutations to their input. In this chapter, we start by providing an overview of the steps involved in FORWARD's partial evaluation, that construct-developers need to be aware of, and then we describe the template construct-developer-

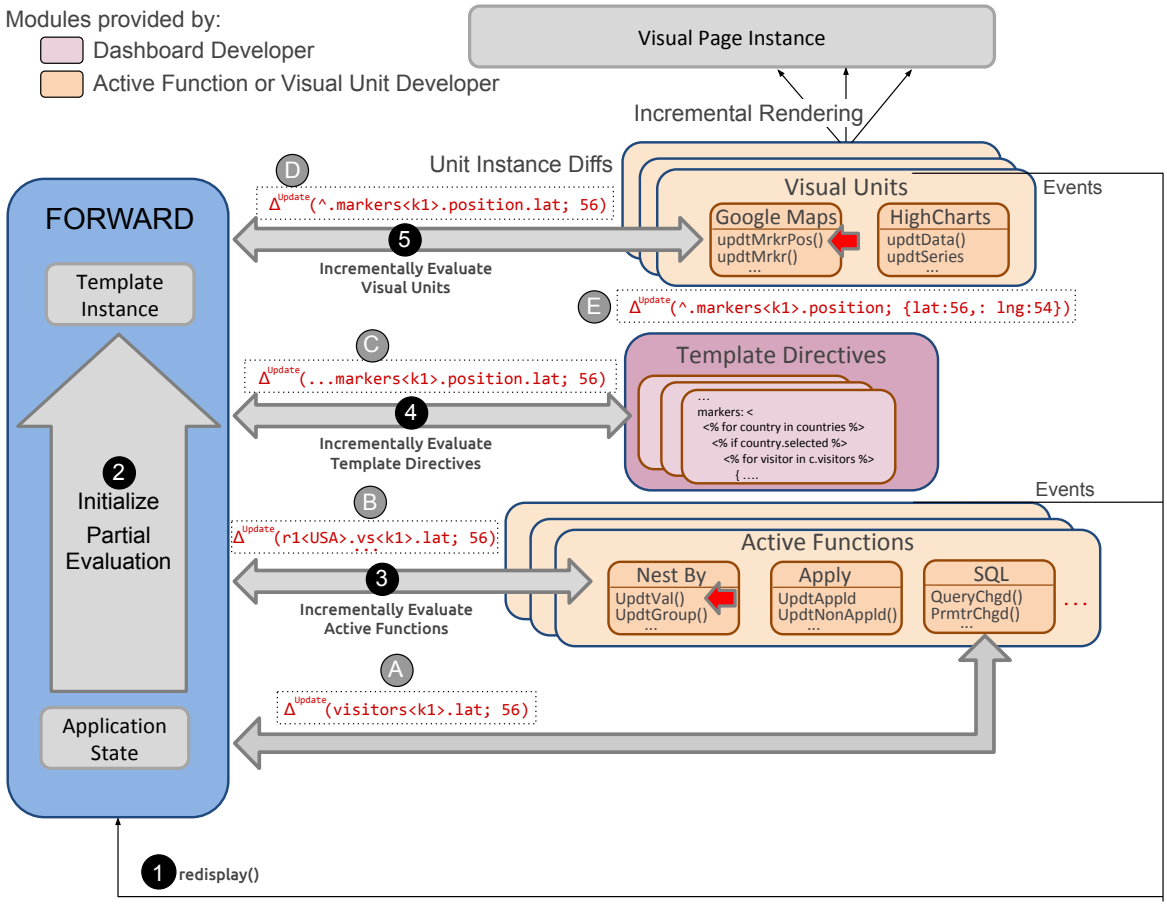


Figure 5.1. Stages of Partial Evaluation.

facing features of FORWARD, that simplify this seemingly daunting task.

5.1 FORWARD's partial evaluation life-cycle and architecture

FORWARD propagates changes from variables all the way to the visual layer through a process called *Partial evaluation*. Partial evaluation is performed in 5 steps (as seen in Figure 5.1). The first step is initiated when an event triggers FORWARD's "*redisplay*" action. This event could occur when a unit instance is mutated, as a result of user interaction, or when a mutation is observed in underlying sources (by non-deterministic active functions, as we will describe in 5.3). Mutations in FORWARD are described using *diffs* (as we will describe in 5.2).

For instance, consider a scenario in which the latitude of a visitor is modified in the underlying database. In this case, the *sql* active function generates a diff describing this mutation (diff A in Figure 5.1[1]). At this point, since an event has occurred (mutation at the underlying database), FORWARD, invokes the specified, by the application developer, action, which in this example is the *redisplay()*[2] (due to the trigger instantiation, shown in line 6, in Figure 4.2). Once the *redisplay()* action is triggered, FORWARD's partial evaluation commences (step 2). During this process, FORWARD propagates changes (i) to other affected active functions (step 3), (ii) to affected template directives, that produce template instance diffs (step 4), and (iii) to affected visual units, which reflect the changes to the visual layer (step 5).

5.1.1 Incrementally evaluating affected active functions

When the *redisplay()* action is triggered, FORWARD's partial evaluation utilizes the developer-provided template[3] to identify other active functions affected by incoming diffs (step 3). Once an affected active function is identified, FORWARD triggers its incremental evaluation. For instance, in this example, active function '*nest_by()*' is affected by diff A, therefore FORWARD will invoke its incremental evaluation. During their incremental evaluation, active functions must produce diffs describing how their output is affected by incoming diffs (the output diff of *nest_by()* is diff B). Responsible for describing this behavior are active function developers, that use the API provided by FORWARD (which as we will describe later in this section, is a trigger language) to define a conglomerate of delta functions. Each delta function expects a diff describing a particular change to the input of the active function and returns a diff describing a change to its output. Given this collection of delta functions and a diff, FORWARD, identifies and invokes the appropriate delta function. For instance, in our example, *UpdtVal()* is the appropriate delta function for handling diff A which in turn generates diff B (section 5.3 contains the

[1]The actual details of the diff notation will be described in Section 5.2.

[2]Note that the *redisplay()* action is equivalent to invoking the *display()* action with argument the name of the currently evaluated template.

[3]Recall that the template describes a DAG that can contain multiple active function invocations, as shown in Figure 4.4

specifics). Note that active functions might also produce diffs that affect other active functions, in which case FORWARD will cause their incremental evaluation as well. For instance, as seen on template 4.2 the diff produced by *nest_by* affects the *apply* active function, followed by the *sort_by* active function and so on.

5.1.2 Incrementally evaluating affected template directives

After all affected active functions have produced their respective diffs (step 3), FORWARD then invokes the partial evaluation of template directives (step 4). During this process FORWARD's change propagation algorithm (described in Section 6), takes diffs that target template variables and produces diffs that target the template instance (diff C describes the mutation of the affected marker, which is part of the template instance).

5.1.3 Incrementally evaluating affected visual units

Once template instance diffs are generated, FORWARD identifies unit instances, the state of which is affected by the incoming diffs, and invokes their partial (incremental) evaluation, which results in the incremental rendering of the application's visual layer (step 5). Responsible for specifying incremental rendering are visual unit developers that utilize FORWARD's API to provide a set of delta functions. Each delta function takes a diff, that describes a particular unit state change, as input, and incrementally propagates that change to the visual layer (Figure 5.7, shows the declaration of delta functions for the Google Maps visual unit[4]). Note that in certain cases the incoming diffs might not match the incremental capabilities (delta functions) of visual units, for instance, in our example, there is no delta function that can update the latitude of a marker (diff D in Figure 5.1). In such cases, FORWARD is responsible for generating equivalent diffs that are supported by visual units (as we will describe in section 6.5). In this example, FORWARD automatically generates diff E (shown in 5.1), that describes the update of the position of the marker, which includes both the latitude and the (existing and unchanged)

[4]More details about the declaration of delta functions and about the trigger language used for this purpose, will be provided later in this chapter

Table 5.1. Supported diff types

Notation	*Semantics*
$\Delta^{\text{insert}}_{\text{array}}(\hat{p}[k];v)$	Insert into array at path $\hat{p}$, at element with position k the value v
$\Delta^{\text{insert}}_{\text{bag}}(\hat{p}<k>;v)$	Insert into bag at path $\hat{p}$, the value v with key k
$\Delta^{\text{append}}_{\text{array}}(\hat{p};v)$	Append to array at path $\hat{p}$ the value v
$\Delta^{\text{insert}}_{\text{tuple}}(\hat{p};n,v)$	Add to the tuple at path $\hat{p}$ the attribute/value pair $n:v$
$\Delta^{\text{update}}(\hat{p};v)$	Replace element at path $\hat{p}$ with value v
$\triangle^{\text{delete}}(\hat{p})$	Delete element at path $\hat{p}$

longitude of the marker, and invokes the matching delta function: *updtMrkrPos()*.

Lastly, as the user interacts with the UI, they may cause changes to the unit instance, in such cases visual unit developers must produce diffs that describe the respective unit instance changes (similarly to source-specific active functions, when underlying data sources are mutated). If the mutated part of the unit state is bound to a template variable (using the ***bind*** directive, as described in Chapter 4), then FORWARD generates the respective diff targeting the affected template variable. If this event is associated with the *redisplay()* action (as shown in template 4.2, line 15), the partial evaluation cycle is triggered again.

5.2 Describing mutations

Diffs. In FORWARD modifications are described using diffs. A *diff* to a `JSON++` value is of the form $\Delta^{type}(\hat{p}; v)$, where *type* is the type of the modification (e.g., update, delete, append-array), $\hat{p}$ is the path to the element that is modified, and v is a `JSON++` value representing the new value (post-state) of the element (also known as the *payload* of the diff). Table 5.1 shows the supported diff types, their notation, and their semantics. Note, that delete diffs do not contain a payload, since a new element value is not needed.

Paths. Paths in diffs are expressed using a simple path language, where $\hat{p}[i]$ denotes the i'th element of the collection at path $\hat{p}$, $\hat{p}\langle k\rangle$ denotes the element with key k of the bag at path $\hat{p}$ and $\hat{p}.n$ the value of attribute n within the tuple at path $\hat{p}$.

Example 5.2.1 *The diff that describes the modification of a visitors's latitude (mentioned in Section 5.1 and shown in Figure 5.1 as diff A) is the following:* $\Delta^{\text{update}}(visitors\langle k_1\rangle.lat; 56)$. *This diff describes that the latitude of the element with key:*k_1 *will be updated to the value 56.*

FORWARD, simplifies the process of diff creation by providing an easy to use API. Specifically, the statement: *Diff('visitors*$\langle k_1\rangle$*.lat', 'update', 56)* generates the diff described above. This statement uses a FORWARD-provided function call: *Diff()*, that takes 3 arguments; the first argument corresponds to the path of the element that is being modified (provided as a string), the second to the operation that will be applied at the targeted element (also provided as a string) and the third to the `JSON++` value that describes the new value (post-state) of the targeted element. Additionally, given a diff d: $\Delta^{\text{update}}(visitors\langle k_1\rangle.lat; 56)$ the function "*d.getTarget()*" returns the target of the diff (i.e., $visitors\langle k_1\rangle.lat$), and the function "*d.getPayload()*", returns the payload (i.e., 56).

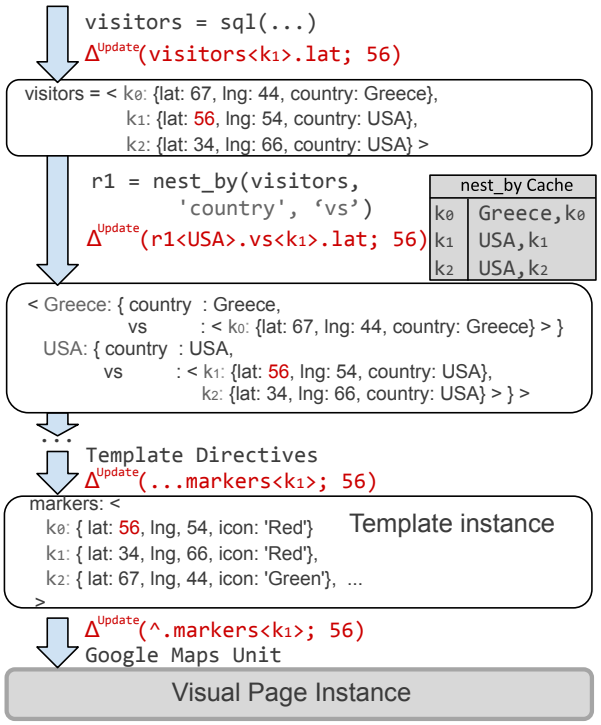


Figure 5.2. Partial evaluation of data DAG.

5.3 Implementing Active functions.

Active functions belong to one of two categories: (a) *non-deterministic* and (b) *deterministic*. If a non-deterministic active function is invoked with the exact same input multiple times, there is no guarantee that the output will be the same in each invocation, since it depends on factors that are outside of FORWARD's scope, such as data residing in remote database systems. For instance, invoking the '*sql*' active function with the exact same input (SQL query) might return different results in each invocation, depending on the data stored in the underlying database tables. *Deterministic active functions*, on the other hand, only depend on the input provided during the invocation. For instance, invoking the functions '*sort_by*' and '*nest_by*' multiple times with the same input, always returns the same output. Other than full evaluation (that performs the initial data acquisition and computation), active functions are also responsible for issuing diffs that describe changes to their output. In the following sections we describe the API used

for specifying the initial and incremental behavior of deterministic and non-deterministic active functions.

5.3.1 Construct developer facing API

The provided API allows the declaration of procedures that are invoked when an active function is first instantiated. This is performed through the function *initial(f)*, which specifies that the procedure f will be invoked during the instantiation (full evaluation) of the template and the active function. In line 2 (Figure 5.6), we show the declaration of the procedure *instantiate_sql()* as the initial function (the body of which is shown in lines 4-26) which establishes a communication with the *sql* active function back-end using websockets (lines 5-9) and returns the initial results when the back-end responds (line 16). When mutations occur at the underlying database system, the active function is responsible for notifying FORWARD by generating and returning the diffs that describe these changes (as shown in lines 17-25). Similarly, in Figure 5.5, we show the definition of the deterministic '*nest_by*' active function, which nests input data based on a provided key (Figure 5.2 demonstrates, among others, the output of the *nest_by* given its input). The initial procedure *constructNesting* is declared in line 3.

In order to promote incremental computation in active functions FORWARD supports the declaration of delta functions. *Delta functions*, are procedures that can be invoked when diffs are applied to the input of an active function and are responsible for incrementally generating diffs that describe how the output of the active function is affected. The declaration of a delta function is performed using a tuple of the form *(sign; deltafunc)*, where *sign* is a diff signature and *deltafunc* the delta function that will be invoked if an incoming diff matches the signature. This declaration, essentially, describes the instantiation of triggers, but it expands the SQL concept of triggers, in two ways: (1) by having signatures that operate on nested data and (2) by enabling the declaration of active functions the input diffs of which do not have a fixed input schema but rather can operate on a family of diff schemas. Specifically, a *diff signature* of the form *sign* = $(t, \hat{p})$, supports diffs of type t (e.g., update, delete, append-array, etc.) that have a path that adheres

to the path signature $\hat{p}$.

Path Signatures

A *path signature* is expressed using the same language used in paths, with the added support of wildcards, negations and print statements that can embed content of variables in the signature. Specifically, wildcards include:

- A bag wildcard $p.\langle * \rangle$ that matches any bag element that follows the path p (e.g., the path signature: $p\langle * \rangle$ matches the path: $p\langle k_1 \rangle$)
- A tuple wildcard $p.*$ that matches any tuple attribute that follows the path p (e.g., the path signature: $p.*$ matches the path: *p.attr*)
- An array wildcard *p[∗]* that matches any array element that follows the path p, (e.g., the path signature: *p[∗]* matches the path: *p[1]*)

Path signatures, other than static steps, can also include variables. The *nest_by* active function, for instance, which nests input data based on an attribute that is given during the evaluation of the template (as shown in Figure 4.2, line 9, in this example this attribute is '*country*'), the active function developer must embed the content of a variable that contains the given attribute in the path signature. In order to do so, the developer has to employ the print statement. Specifically, when a variable v is used in path signatures it must be surrounded with **<%=**v**%>** (print) annotation. For instance, the path signature shown in line 5 (Figure 5.5), includes the "*in*" and "*group_attr*" variables the contents of which are provided as input during the instantiation of the *nest_by* active function . In such cases, FORWARD will automatically, replace the name of the variable with its content. Lastly, a path signature can contain a negation (using the "!" notation) before a path signature step. For instance, the syntax in line 5, specifies that the function '*updtVal*' will be invoked when an incoming diff of type '*update*' with target that matches the path signature '**<%=***in***%>**.$\langle * \rangle$.!**<%=***group_attr***%>**' is generated.

This path signature, after the variables have been replaced with their contents, has the format: 'visitors.$\langle * \rangle$.!country' and matches the diffs that target any bag element attribute other than the '*country*' attribute (i.e., it matches the diff: $\Delta^{\text{update}}(visitors\text{<}1\text{>}.lat; 56)$, but it does not match the diff: $\Delta^{\text{update}}(visitors\text{<}1\text{>}.country; \text{'}Greece\text{'})$).

Caching in active functions. While specifying the incremental behavior of active functions, often, construct developers need to instantiate and cache data structures that will be used by individual delta functions. This is because, often the incoming diffs do not contain sufficient information to produce output diffs, since in an attempt to minimize memory footprint FORWARD does not maintain the pre-state and post-state of all nested values. For instance, Figure 5.2 shows the initial and incremental output of the *nest_by* active function. Notice that given the incoming diff $\Delta^{\text{update}}(visitors\langle k_1 \rangle.lat; 56)$, *nest_by* produces the diff $\Delta^{\text{update}}(r_1\langle USA \rangle.vs\langle k_1 \rangle.lat; 56)$. Since the incoming diff does not contain the key of the output diff (namely *USA*), the construct developer needs to cache the hash table "nest_by cache" (shown in the same figure) in order to infer the keys of the output diff. In order to facilitate this process FORWARD automatically caches all member variables of active functions and makes them accessible from individual active functions.

Visual Units

Visual units operate in a very similar manner as active functions, that take as input other template variables, since they both react to input diffs. The difference, however, is that while active functions produce diffs describing how their output is affected by an incoming diff, visual units directly cause mutations to the visual layer of the application, by invoking the appropriate renderer of the underlying visualization library (i.e., Google Maps, Highcharts etc.). Responsible for invoking the appropriate renderers are procedures contained in visual units, named *renderer wrappers* (which are equivalent to delta functions). Additionally, as we will describe later in this section, the process of developing renderer wrappers (also known as visual unit wrapping) is "pay-as-you-go", which means that developers are not required to create a renderer wrapper

for every potential input diff in order for the unit to operate correctly (which is a fairly daunting task), instead they can provide the bare minimum renderer wrappers and the visual unit would still be functional. In certain cases, however, having a renderer wrapper that perfectly "matches" a diff could improve performance and (most importantly) the user experience, as we will describe later in this section.

Declaring a renderer wrapper is performed (in a similar manner to active functions) through a tuple of the form *(sign; rwfunc)*, where *sign* is a diff signature and *rwfunc* a renderer wrapper procedure. Figure 5.7 shows a snippet of the Google Maps visual unit. As seen in lines 3-6, the developer declares initial and delta functions in a similar manner as in active functions. If an incoming diff of the form $\Delta^{\text{update}}(map.markers\text{<}1\text{>}.position; \{lat : 56, lng : 54\})$, is applied to the visual unit (which describes the relocation of an existing marker), FORWARD will automatically invoke the renderer wrapper, shown in lines 8-17 (its declaration is shown in lines 4-5).

Note that in addition to the diff types described in subsection 5.2, renderer wrappers also accept two new types of diffs unique to visual units. These are the construct-unit diff $\Delta^{\text{construct}}_{\text{unit}}(v)$ describing the construction of a new unit instance that has unit state equal to the `JSON++` value v (the declaration of which is shown in line 3) and a destruct-diff $\Delta^{\text{destruct}}_{\text{unit}}()$ describing the destruction of an entire unit instance. Such diffs are automatically generated by FORWARD during the instantiation and destruction, respectively, of a visual unit instance. Note that the $\Delta^{\text{destruct}}_{\text{unit}}()$ diff does not include a payload (similarly to the delete diff).

Caching in Visual Units. Renderer wrapper developers often need to cache various JS objects that enable incremental rendering and associate these JS objects with respective objects of the JSON++ unit state. For instance, in order to update the location of an individual Google Maps Marker, the developer of the respective, renderer wrapper must first (a) obtain the appropriate Google Maps Marker object (line 11, in figure 5.7) and then (b) invoke the renderer function: "*setPosition*", by providing as input a JavaScript object of the form: *{lat:..., lng:...}*, that describes the new location of the marker (line 15, in figure 5.7). FORWARD provides

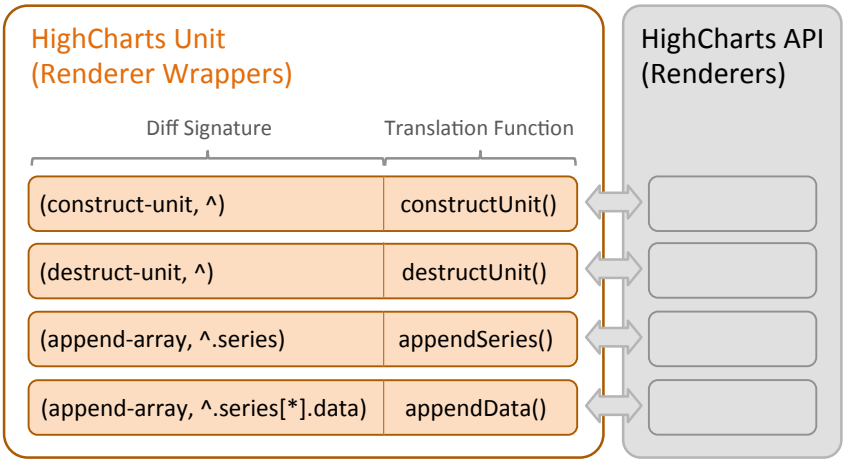


Figure 5.3. Unit Architecture.

an API that facilitates those two steps. Firstly, it provides two utility functions that automate caching UI objects (such as a Google Maps Marker object). Specifically, these functions are: (a) *setUIObject(p, ui_obj)*, that caches the UI object: *ui_obj*, which is identified by the path: *p* and (b) *getUIObject(p)*, that obtains the already cached ui object identified by the path: *p* (note that the underlying data structure used for caching, is automatically maintained by FORWARD as diffs are applied to the visual unit instance, thus absolving developers from having to perform this task manually). In line 11, the unit developer uses this API to obtain the Google Maps marker identified by the path: "*map.markers<1>*". This path was returned after the invocation of the *up()* method on the target path of the diff (given a path that uniquely identifies a node *n*, this method returns the path of the parent of the node *n*, which in our running example is the path of the marker UI object). Secondly, after the appropriate UI object has been obtained, the unit developer simply invokes the "*setPosition*" function by providing as input the payload of the diff. . Note that no data-model transformations are needed between the payload provided by FORWARD and the, underlying renderer function, this is because FORWARD automatically performs these transformations, thus absolving the construct developer from having to perform this task manually (as we will describe in higher detail later in 5.4).

As-you-go approach in visual unit and active function development

Note that developers of active functions and visual units do not need to provide renderer wrappers for every possible incoming diff. As long as there exists an *initial* function (in active functions) or a combination of construct-unit and destruct-unit renderer wrappers (in visual units), FORWARD will be able to translate any incoming diff (which may not be supported) to an equivalent supported diff through a process referred to as *simulation*. Simulation is especially important for the usability of FORWARD, as it decreases the effort required for the development of template constructs, by allowing developers of constructs to create them in an as-you-go fashion: They can start by implementing the required functions that are responsible for the full evaluation (construct-destruct, for visual units and initial function, for active functions) and wrap additional finer-grained incremental functions as time permits. These modules will always work, but as additional finer-grainer delta functions/renderer wrappers are added, FORWARD will automatically utilize them, leading to more efficient change propagation.

There are 3 types of visual unit wrappings that can be performed by a construct developer: (a) complete unit wrapping, (b) application-optimal unit wrapping and (c) application-satisfactory unit wrapping. In a *complete unit wrapping*, developers must declare a renderer wrapper for every renderer provided by the underlying visualization library. For instance, the API of the Google Maps visualization library provides 55 renderers, therefore a complete unit wrapping must declare an equal number of renderer wrappers. In *application-optimal unit wrapping*, developers must declare a renderer wrapper for every diff that could be applied in an application. For instance, since the application described in this paper supports insertions and relocations of users, which would result in insertions and relocations of markers, the application-complete unit wrapping only contains these 2 (plus the construct and destruct) renderer wrappers. Lastly, the *application-satisfactory unit wrapping* is a middle ground between a unit wrapping that only contains a destruct and construct renderer wrapper and the application-optimal. For instance consider a scenario in which there is no renderer wrapper that updates the location of a marker

Table 5.2. Unit Wrapping

Renderer Wrappers	*G-Maps*	*LeafletJS*	*HighCharts*
No-Simulation	>500	>253	14958
Complete Wrapping	55	42	58
Application Optimal	4	4	4

but there is one that updates the entire marker (along with the color, icon, etc.,). In this case, the rendering performance is not going to be significantly compromised (since the visualization will not be completely destructed and reconstructed), but at the same time it is going to be marginally more inefficient than the application-optimal case. The latter two cases are both connected to the pay-as-you-go aspect of unit wrapping. It should be noted that the same categories of wrappings exist for active function development as well.

Note that visualization libraries do not typically provide a renderer for every possible diff that could be applied to the unit state of the respective unit. For instance, the Google Maps API does not provide a renderer capable of mutating only the latitude of a marker, it only provides a renderer for updating the entire location of a marker (i.e., both the latitude and the longitude). If simulation was not provided by FORWARD as a feature, visual unit developers would have to perform simulation manually. This means that they would have to provide a renderer wrapper for every potential diff that could be targeting any part of the unit state, this would translate to hundreds or thousands delta functions. Table 5.2 shows the number of renderer wrappers that would be required for wrapping Google Maps, Leaflet JS and Highcharts, (a) if simulation was not supported, (b) for a complete unit wrapping and (c) for an application optimal unit wrapping (for our demonstrated application). Notice that the HighCharts visual unit would need roughly 15,000 delta functions, if simulation was not supported. More details about Simulation are provided in the following chapter.

Reacting to user input. FORWARD supports the implementation of visual units that can collect user input. User input can be collected in two ways: (a) through a two-way binding between a vdb variable and a unit state value and (b) through the declaration of an action, invoked

in response to an event.

Example 5.3.1 *For instance, the checkbox that appears next to each country in our running example, is created by the* `checkbox` *visual unit (shown in the template, in lines 23-26), that allows a two-way binding between the unit state value:* `checked` *and the vdb variable group.selected (shown in line 24). Additionally, in line 25 we declare that when the user clicks on a checkbox, the action redisplay() will be invoked.*

Visual unit developers can support user input collection in their units by using the API of the underlying visualization library, which allows them to declare callback functions that will be executed when each event occurs. Within a callback function they generate diffs that describe the mutation that takes place in the unit state and then employ the FORWARD API to propagate this change into FORWARD. Lastly, if an action is specified for the particular event, it is invoked at that time.

Example 5.3.2 *For example, Figure 5.4 illustrates the body of the checkbox's construct renderer wrapper. Lines 2-5 describe the creation of a checkbox element (line 15 describes the addition of said element into the DOM tree). Line 6 shows the declaration of a callback function that will be invoked when the checkbox is clicked. Lines 7-8 illustrate the creation of an update diff with path ^.checked, the payload of which is equal to the current state of the checkbox (true or false, depending on whether it's selected or not). Once the diff is created the function handleBindDiff(diff) is invoked (line 9), which translates the unit instance diff to a variable diff and then adds the newly created diff to the diff log. Additionally, if an action is specified in the unit state, it is invoked (as shown in lines 10-13, line 11, checks whether it is specified and line 12 contains the actual invocation). In this particular running example, the action onClick corresponds to the function redisplay().*

```
function construct(unit_instance, diff) {
  var unit_state = diff.getPayload();
  var checkbox = document.createElement("INPUT");
  checkbox.setAttribute("type", "checkbox");
  checkbox.checked = unit_state.checked;
  checkbox.onclick = function() {
    var diff = new Diff(new Path('^.checked'),
      'update', checkbox.checked);
    unit_instance.handleBindDiff(diff);
    var onClick = unit_state.onClick;
    if (onClick) {
      onClick.action(...onClick.args);
    }
  }
  unit_instance.container_element.appendChild(checkbox);
}
```

Figure 5.4. Checkbox Visual Unit Construct Renderer Wrapper

5.4 Data-Model Object Representation

Note that while the `JSON++` data model is an extension of the JSON data model (with added support for bags), FORWARD reduces `JSON++` objects into Plain Old JavaScript Objects (POJSO). This is done to simplify visual unit wrapping, since renderers expect valid JSON variables as input, if this reduction did not happen automatically, unit developers would need to perform this task manually, which would require more effort and lines of code. Specifically,

- `JSON++` arrays are represented as JavaScript arrays
- Scalars (i.e., strings, number, booleans) are represented by (string, number or boolean) JavaScript primitives
- Both tuples and bags are represented by JavaScript (key/value) objects. Developers can distinguish between the two by using the appropriate path signature when defining delta functions (i.e., $d\langle * \rangle$ defines a bag while $d.*$ defines a tuple)

This design decision is beneficial because it reduces the learning curve required for developers to start building active functions and visual units. POJSO's are widely used in web applications so JavaScript developers are already familiar with such objects. Additionally, all third-party visualization and data processing libraries expect POJSO's, so no transformations

```
1   nest_by(in, group_attr,
2           output_collection_attr){
3     initial(constructNesting);
4     delta("insert", "<%=in%>.<*>", insertGroup);
5     delta("update", "<%=in%>.<*>.!<%=group_attr%>", updtVal);
6     delta("update", "<%=in%>.<*>.<%=group_attr%>", updGroup);
7     ...
8     function constructNesting () {... perform nest by ... },
9     function updtVal ()  {... mutate value ... },
10    function updGroup () {... mutate key ... },
11    ...
12  }
```

Figure 5.5. Nest-by active function

between object representations are necessary when wrapping such libraries into active functions and visual units. This ensures a seamless integration and eliminates the time and lines of code that would be required for performing such transformations. For instance, in Figure 5.7 in line 15, the payload of the diff (a `JSON++`) value is directly plugged as parameter to the *setPosition* function that expects a JSON value. Lastly, by using primitives, instead of JavaScript object wrappers, to represent scalars we witness up to 99% faster processing times when performing common operations (such as declarations of new variables, string concatenations, number multiplications etc.).

5.5 Acknowledgements

This chapter contains material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission for publication. The dissertation author was the primary investigator and author of this paper.

```
sql(query, ...and other arguments...){
  initial(instantiate_sql);

  function instantiate_sql(arg){
    /* setting up connection with the backend */
    var ws = new WebSocket("http://backend_url")
    ws.onopen = function() {
        ws.send(arg); // Web Socket is connected, send data
    };
    ...
    /* This function will be triggered when the backend
       mutations returns initial data or when mutations
       take place at the underlying database */
    websocket.onmessage = function(msg) {
     if (msg.type == 'initial_evaluation') {
       return msg.data
     } else if (msg.type == 'partial_evaluation') {
      ...
      // if user latitude is modified, msg.data contains:
      // { trg: '^<1>.lat', op: 'update', p: 56 } therefore
      // the following invocation will be equal to:
      // return Diff('^<1>.lat', 'update', 56)
      return Diff(msg.data.trg, msg.data.op, msg.data.p)
    ...
     } ...
    } ...
  } ...
}
```

Figure 5.6. sql active function

```
GoogleMaps(){
  ...
  delta("construct", createMapComponent);
  delta("update", "map.markers<*>.position",
                   updtMrkrPos);
  delta("update", "map.markers<*>", updtMrkr);
  ...
  function updtMrkrPos(diff) {

    // using diff path to access existing marker
    var m = getUIObject(diff.getTarget().up());

    // using the Google Maps API and the payload of
    // the diff to update the marker.
    m.setPosition(diff.getPayload());
  }
}
```

Figure 5.7. Google Maps Unit

Chapter 6

Internal architecture

In section 5.1 we described the five steps of partial evaluation that construct developers need to be aware of. In this chapter, we will be expanding the FORWARD module and will be examining the internal architecture that facilitates the efficient propagation of changes from sources all the way to the visual layer. We start by providing a high-level description of the operations performed by each individual internal module, during the process of partial evaluation, and then we illustrate the internal workings and algorithms of each module.

6.1 Architecture of Partial Evaluation

Figure 6.1 illustrates the internal architecture that performs partial evaluation, for the same scenario we described in section 5.1, in which the latitude of a visitor is modified in the underlying database. In this case, the `sql` active function generates a diff describing this mutation (diff A in Figure 6.1). Note that this diff currently, does not target a particular template variable (recall that active functions are generic, non-application-specific template constructs, as a result, they do not produce diffs that target application-specific template variables), therefore once obtained, FORWARD invokes the diff translator to produce a diff that does target a template variable (diff B targets the variable `visitors`). FORWARD then adds the new diff to the Variable Diff Log. The *Variable Diff Log* maintains template diffs. Other modules of the FORWARD architecture can access any diff that is kept in this diff log. At this point, since an

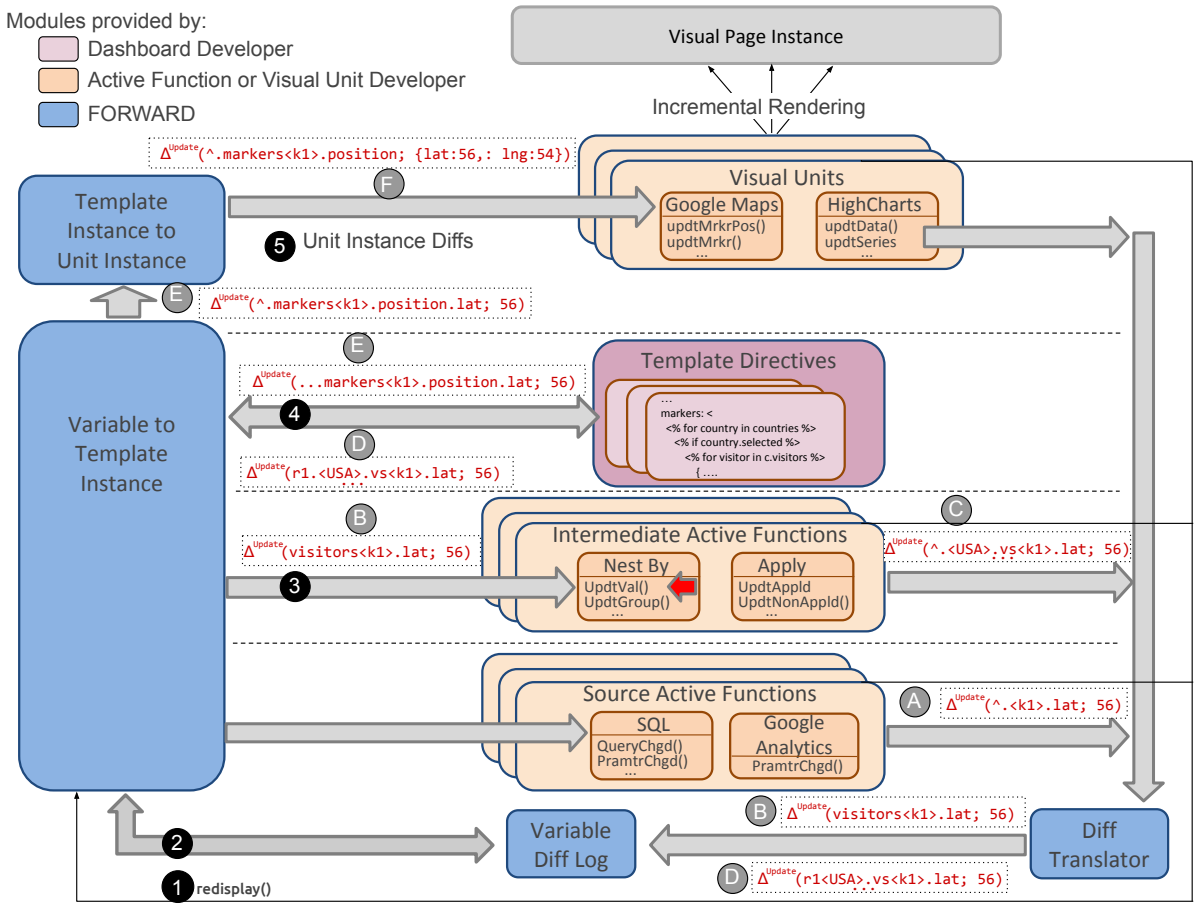


Figure 6.1. Stages of Partial Evaluation - Internal Architecture

event has occurred (mutation of the underlying database), FORWARD, invokes the specified, by the application developer, action, which in this example is the `redisplay()` (due to line 1 in Figure 4.2). The `redisplay()` action invokes the *Variable to Template Instance (V2TI) Module*, the goal of which is to propagate changes from template variables to the template instance.

Specifically, when the "`redisplay`" action is triggered (step 1), it initiates the V2TI module that requests the set of diffs that occurred in template variables (step 2) from the *Variable Diff Log*. V2TI, then utilizes the diffs and the template to identify other active functions affected by the diffs and causes their incremental evaluation (recall that the template describes a DAG that can contain multiple active function invocations, as shown in Figure 4.4). For instance, active function '*nest_by()*' is affected by diff B, therefore FORWARD will invoke its incremental

evaluation. As we described in section 5.1, the '*nest_by()*' is a deterministic active function, that produces diff C, which describes how the output of the active function is affected. During this process, V2TI is responsible for causing the incremental evaluation of any other active function that would be affected by other template variable diffs. For instance, based on the template shown in figure 4.2 the diff produced by the `nest_by` active function will affect the `apply` active function, followed by the `sort_by` active function and so on.

After all affected active functions have produced their respective diffs (step 3), which have then been funneled through the Diff Translator and back to the V2TI, the V2TI then invokes the partial evaluation of the template directives. During this process, FORWARD's change propagation algorithm (described in section 6.3), takes diffs that target template variables and produces diffs that target the template instance (step 4) (diff E describes the mutation of the affected marker, which is part of the template instance). Note that the V2TI propagates changes first to active functions and then to template directives. To accomplish this, FORWARD employs a template pre-processor that extracts active function invocations that appear within for-loop and if-then-else statement directives, into let statements that appear at the root of the template (before any for-loops or if-then-else statements). This not only simplifies the incremental evaluation of said directives, but also allows V2TI to propagate changes both to active functions and template directives in one step, by traversing the template only once (as we will describe later in this chapter).

Once V2TI generates the appropriate template instance diffs (diff E in this example), it then initializes the *Template Instance to Unit Instance (TI2UI) Propagation Module*, which identifies unit instances the state of which is affected by the incoming diffs and invokes their incremental evaluation. To do so, TI2UI, finds and invokes the appropriate delta function that causes the incremental rendering of the visual layer of the application (step 5). Note that in certain cases the incoming diffs might not match the incremental capabilities (delta functions) of visual units (for instance, in our example it is possible that no delta function might be able to update the latitude of a marker, diff E). In such cases TI2UI is responsible for generating

equivalent diffs (diff F) that are supported by visual units (as we will describe in section 6.5).

6.2 Variable to Template Instance (V2TI) Module

As explained above, FORWARD's V2TI module is responsible for transforming diffs that target template variables into diffs that target parts of the template instance. During this transformation, V2TI often needs to (i) maintain cached objects, used by active functions to perform efficient, incremental evaluation, (ii) potentially maintain the state of expressions that appear within for-loops and print and if-then-else statements and (iii) automatically perform transformations between bags and arrays. In order to facilitate these tasks, V2TI uses an auxiliary data structure, named annotated template instance (ATI).

6.2.1 Annotated template instance (ATI)

The annotated template instance is constructed as a result of the full evaluation of a template and is automatically maintained in real time as diffs are constructed by the propagation algorithm. Essentially, the ATI is a fully evaluated template AST (abstract syntax tree), that is constructed after evaluating all the expressions that appear in a static template AST. This means that it contains the results of every evaluated expression, as well as tree branches constructed after FORWARD performed the unrolling of for-loop statements. Additionally, as we will describe later, it also maintains information required for the propagation of diffs that target if-statements and facilitates automatic conversions between arrays and bags. For instance, Figure 6.6 shows a representation of the ATI for the template shown in Figure 6.5a

6.2.2 Translating template variable diffs into template instance diffs

The propagation of template variable diffs to diffs that target the template instance, occurs in two main steps:

1. *Variable to Annotated Template Instance (V2ATI) Diff Propagation*: This step is responsible for translating diffs that target template variables, into diffs that target the annotated

ALGORITHM 1: Template Diff Propagation

```
function TemplateDP(Template T, Annotated Template Instance ATI, Env U, Set of Input Diffs
  D_in, Path Signature targeting current Template t̃)
    Output : A set of ATI Diffs
    D_out ← empty set;
    for top-level directive d ∈ T do
        k ← semantic key of d ;
        if d is <% let y = e %> then
            D_in ← D_in ∪ ExpressionDP(U, D_in, y, ATI);
        else if d is <% print e %> then
            D_out ← D_out ∪ ExpressionDP(U, D_in, t̃<k><%print%>, ATI);
        else if d is <% for y in e %>B<% end for%> then
            Δs ← ExpressionDP(U, D_in, t̃<k><%for%>, ATI);
            for Δ^type(t̂; p) in Δs do
                if t̂ matches t̃<k><%for%> then
                    if type = update then
                        Let T′ be the template rooted at d, where e is replaced by p;
                        p′ ← instantiateTemplate(T′, U);
                        D_out = D_out ∪ {Δ^update(t̂; p′)};
                    else if type = delete then
                        D_out = D_out ∪ {Δ^delete(t̂)}
                    else if type = append array then
                        U′ ← U ∪ {y ← p};
                        p′ ← instantiateTemplate(B, U′); D_out = D_out ∪ {Δ^append_array(t̂; p′)};
                    else if type = insert bag then
                        (k, v) ← p; U′ ← U ∪ {y ← v};
                        p′ ← instantiateTemplate(B, U′);
                        D_out = D_out ∪ {Δ^append_bag(t̂; k, p′)};
                else if t̂ matches t̃<k><%for%>[i] and type = insert array then
                    U′ ← U ∪ {y ← p};
                    p′ ← instantiateTemplate(B, U′);
                    D_out = D_out ∪ {Δ^insert_array(t̂; p′)};
                else if t̂ matches t̃<k><%for%>[i]ŝ then
                    D′_in ← D_in ∪ {Δ^type(y ŝ; p)}; t̂′[i]ŝ ← t̂;
                    D_nested = TemplateDP (B, U, D′_in, t̂′[i]);
                    D_out ← D_out ∪ D_nested;
            end
            D_nested = TemplateDP (B, U, D_in, t̃<k><%for%>[*¬e]);
            D_out ← D_out ∪ D_nested;
        else if d is <%if e_1 then%> B_1 (<%elif e_i then%> B_i)* then
            D_out ← D_out ∪ ConditionDP(d, U, D_in, ATI, t̃<k><%if%>);
        else if d is static directive then
            D_out ← D_out ∪ TemplateDP (B, U, D_in, t̃<k>);
    end
    return D_out;
```

ALGORITHM 2: Expression Diff Propagation

```
1  function ExpressionDP(Env U, Set of Input Diffs D_in, Path Signature t̃, Annotated Template
     Instance ATI)
       Output : A collection of ATI Diffs created as a result of applying diffs D_in to expression of
                ATI node identified by path signature t̃
2      D_out ← empty Set;
3      ns ← find ATI nodes targeted by t̃ ;
4      for each node n in ns do
5          ê ← expression of node n;
6          if ê is a path expression then
7              for each Δ^type(t̂; p) ∈ D_in do
8                  if t̂ is êŝ then
9                      D_out ← D_out ∪ {Δ^type(t̃ŝ; p)}
10                 else if ê is t̂ŝ then
11                     if type = update then
12                         p′ ← navigate(p, ŝ) ;
13                         D_out ← D_out ∪ {Δ^update(t̃; p′)}
14                     if type = delete then
15                         D_out ← D_out ∪ {Δ^delete(t̃)}
16                     else
17                         p′ ← navigate(U, ê) ;
18                         D_out ← D_out ∪ {Δ^update(t̃; p′)}
19             end
20         else if ê is an active function invocation then
21             D_out ← D_out ∪ {ActiveFunctionDP(U, ê, D_in, t̃)} ;
22     end
23     return D_out;
```

ALGORITHM 3: Expression Diff Propagation

```
1  function ActiveFunctionDP(Env U, Active Function f, Set of Input Diffs D_in, Path Signature
     t̃)
       Output : A collection of ATI Diffs created as a result of applying diffs D_in to the active
                function f
2      D_out ← empty Set;
3      for each path parameter ê of f do
4          for each Δ^type(t̂; p) ∈ D_in do
5              if t̂ is êŝ then
6                  D_out ← D_out ∪ {invoke_delta_or_simulate(U, f, Δ^type(t̂; p))}
7              else if ê is t̂ŝ then
8                  D_out ← D_out ∪ {Δ^update(t̃; ê.initial())}
9          end
10     end
11     return D_out;
```

ALGORITHM 4: Diff Propagation on condition directives

```
1  function ConditionDP(Condition Directive d, UAS U, Set of Input Diffs D_in, Annotated
     Template Instance ATI, Path Signature d~)
       Output : A collection of ATI Diffs created as a result of applying diffs D_in to expressions of
                condition directive d
2      D_out ← empty Set; ns ← find ATI nodes targeted by d~ ;
3      for each node n in ns do
4          n̂ ← path that leads to ATI node n ;
5          for each condition statement instance c in n do
6              k ← semantic key of c;
7              D ← ExpressionDP(U, D_in, d~ < k >, ATI);
8              for Δ^update(t̂, p) in D do
9                  Set value p as output of c.exp in n;
10             end
11         end
12         D'_out ← empty Set; c' ← get active condition of previous invocation ;
13         for each condition statement instance c in n do
14             k ← semantic key of c;
15             if c.exp coerces to true then
16                 if c' is null then
17                     if c.B is not instantiated then
18                         c.B ← instantiateTemplate(c.TB, U);
19                         D'_out ← D'_out ∪ Δ^insert(n̂, c.B);
20                     else
21                         D'_out ← D'_out ∪ Δ^show(n̂[k], c.B);
22                     Set c as the currently active condition in ATI;
23                 else if c' is equal to c then
24                     D_nested ← TemplateDP (c.TB, U, D_in);
25                     D'_out ← D'_out ∪ prefixDiffs(n̂[k], D_nested);
26                 else if c' is not equal to c then
27                     if c.B is not instantiated then
28                         c.B ← instantiateTemplate(c.TB, U);
29                     D'_out ← D'_out ∪ Δ^update(n̂, c.B);
30                     Set c as the currently active condition in ATI ;
31                     if FORWARD does not maintain c'.B then
32                         c'.B ← null
33                 break;
34         end
35         if None of the conditions coerce to true and c' is not null then
36             D_out ← D_out ∪ Δ^delete(n̂);
37             Update n' so that no active expressions exist;
38             if FORWARD does not maintain c.B then
39                 c.B ← null
40         D_out ← D_out ∪ D'_out;
41     end
42     return D_out;
```

ALGORITHM 5: Annotated Template Instance Diff Translation

```
1  function ATI_DP(Annotated Template Instance ATI, UAS U, Queue of Input Diffs Q_in)
2      while Q_in is not empty do
3          Δ(t̂; p) ← Q_in.dequeue();
4          n ← find ATI node targeted by t̂ ;
5          if n is a condition statement then
6              apply p to output of n.exp ;
7              n′ ← find parent condition directive ;
8              Δ(t̂; p) ← peek at next output of Q_in ;
9              while t̂ targets condition statement n of n′ do
10                 Δ(t̂; p) ← Q_in.dequeue();
11                 apply p to output of n.exp ;
12             end
13             for each condition statement c in n′ do
14                 if c.exp coerces to true then
15                     c′ ← get active condition of previous invocation ;
16                     if c′ is null then
17                         if c.B is not instantiated then
18                             c.B ← instantiateTemplate(c.TB, U);
19                             // How can U include the additions made to it by for loops and
                               refresh/let statements?
20                         t̂′ ← omit annotation steps from t̂;
21                         D_out ← D_out ∪ Δ^insert(t̂′, c.B);
22                         Set c as the currently active condition in ATI;
23                     else if c′ is equal to c then
24                         // Do nothing! ;
25                     else if c′ is not equal to c then
26                         if c.B is not instantiated then
27                             c.B ← instantiateTemplate(c.TB, U);
28                         t̂′ ← omit annotation steps from t̂;
29                         D_out ← D_out ∪ Δ^update(t′, c.B);
30                         Set c.exp as the currently active expression in ATI; ;
31                         if we do not maintain c.B then
32                             c.B ← null
33                     return D_out;
34             end
35             exp′ ← active expression of previous invocation ;
36             if exp′ is not null then
37                 D_out ← D_out ∪ Δ^delete(d.path);
38                 Update ATI so that no active expressions exist;
39
40     end
41     return D_out;
```

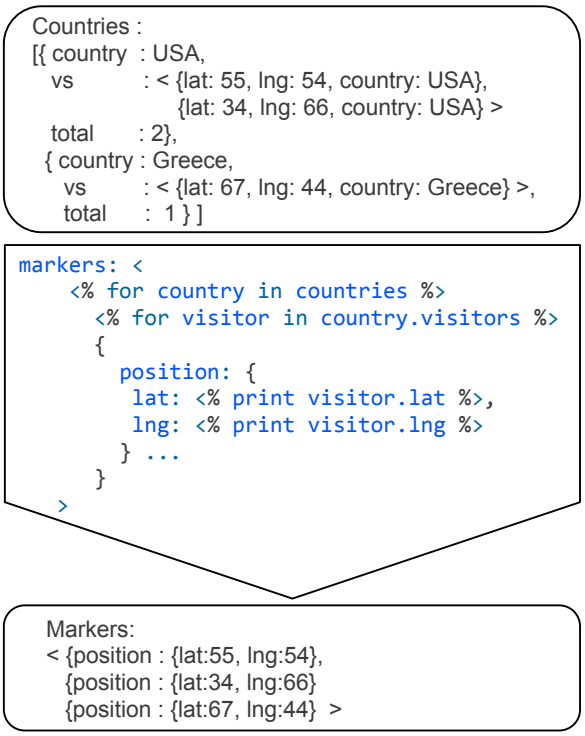


Figure 6.2. Array to bag conversion.

template instance.

2. *Annotated Template Instance to Template Instance (ATI2TI) Diff Propagation*: This step is responsible for translating diffs that target ATI nodes into diffs that target the respective parts of the template instance.

Provenance tracking. Recall that `for` directives offer the ability to formulate computations that iterate over template variable `JSON++` collections (bags or arrays) and generate template instance collections.

Example 6.2.1 *Consider the example shown in figure 6.2, where the template language performs an automatic conversion from the template variable* `Countries`*, which is an array of nested elements, with each nested element containing a bag collection (named* `vs`*), to the template instance collection* `Markers`*, which is a bag.*

Figure 6.3. Array to bag with filtering.

Note, that FORWARD is capable of automatically handling conversions from bags to arrays and the reverse. Most importantly, during partial evaluation, FORWARD is able to translate template variable diffs to template instance diffs. In order to do so, the diff propagation algorithm has to be able to find the correspondence between an element contained in a template variable and the elements of the template instance that were created from it. Note that when automatic conversions take place (from bags to arrays and the reverse) this correspondence cannot be accomplished by purely relying on variable diffs. The reason is that array elements are only identifiable using ordinals (i.e., the diff paths contain ordinals), while bag elements are identified using immutable keys (i.e., the diff paths contain keys that remain constant regardless of the mutations that might occur over time). Therefore, the ATI will be used to facilitate the conversion of variable diffs that contain immutable keys into template instance diffs containing ordinal positions and the complementary.

Additionally, note that if-statements can also filter out parts of the template instance

that depend on template variable values. This means that in some cases, the provenance of the directives has to be taken into account during the partial evaluation.

Example 6.2.2 *In figure 6.3, the if-statement only creates markers for visitors that originate from* `active` *countries (i.e., countries the user has selected, using the checkbox on the UI). In this case, the propagation algorithm must not construct any template instance diffs for variable diffs, targeting users that belong to inactive countries, since the respective variable element did not contribute to the creation of any part of the template instance (unless an incoming diff changes the value of this attribute to '*`true`*').*

In order to track the provenance of the template instance elements, FORWARD again employes the ATI module, which is used to annotate each element e of a template instance collection with its provenance. Formally provenance tracking can be describe as follows:

Let e be an element of a template instance collection and let $d_1, \ldots, d_n$ be the n nested **`for`** and **`if`** directive instantiations that led to the creation of e. Let also k_i be the provenance of the element used in the particular instantiation of the i-th **`for`** or **`if`** directive that led to the creation of element e. Then, the provenance of e is $\#(d_1, k_1, \ldots, d_n, k_n)$. Note that included in the provenance of an element is an identifier of the particular **`for`** or **`if`** directive that was used to generate it. This is important in cases where two or more **`for`** directives, or combinations of **`for`** and **`if`** directives, are used to generate a single output collection. Keeping the identifier of the **`for`** directive that led to an element, allows us to know which original collection an element came from. For instance, the tuples contained in the `markers` collection, in Figure 6.3 have provenance $\#(d_1, k_1, d_2, k_2, d_3, k_3)$, where d_1 is the identifier of the first for-directive appearing in the template, k_1 is the identifier of the country array element, d_2 is the identifier of the **`if`** directive and so on. Note that if a part of the template instance is about to be modified (as a result of an incoming template variable diff), FORWARD will internally, identify the appropriate element (using the provenance annotation described above) and will produce a diff the target of which contains ordinals (for array elements) and immutable keys (for bag elements) instead

of the provenance annotations, since that's how the underlying visual units identify individual elements.

6.3 Variable to Annotated Template Instance (V2ATI) Diff Propagation

To propagate diffs from the template variables to the annotated template instance, FORWARD has to use the provenance of the input diffs to create the correct provenance (along with the rest of the path and payload) of the output diffs. This is done as follows: Given a set D_{in} of diffs that target template variables, the diff propagation algorithm scans the template from top to bottom, performing work for every directive (apart from **`event`** directives which do not affect the generated template instance). In each invocation, it visits only top-level directives (i.e., directives that do not appear nested inside other for directives), as nested directives are acted upon during recursive calls.

Change Propagation Algorithm. The algorithm 42 describes this process. As seen in the algorithm, we distinguish four main cases based on the type of directive visited:

`<% let` *x* **`=`** *E* **`%>`** *directive.* In this case, the algorithm proceeds by calling the `ExpressionDP` subprocedure (shown in Algorithm 2), which given the environment with all template variables, the set containing all input diffs, a path expression and the annotated template instance, computes the set of diffs targeting the output of the expression. Specifically, if the expression E is a path expression, the `ExpressionDP` algorithm computes on the fly the diffs that target the specific contents of variable x (or variable x itself) that need to be mutated (lines 6-19). If on the other hand, E is an active function, the `ActiveFunctionDP` algorithm is invoked, thus performing the partial evaluation of E. During this process, the propagation algorithm tries to identify any incoming diffs targeting a path parameter of the active function (lines 5-13). If such a diff is found, then the algorithm calls the `invoke_delta_or_simulate` function, which either invokes the delta function that matches the incoming diff (if such a delta

function exists), or proceeds with the simulation of the incoming diff with another diff that is supported by a delta function. Note that if no delta functions exist, FORWARD will simply reevaluate the initial function and construct a diff that targets variable x.

Example 6.3.1 *For instance, consider the invocation of the* `templateDP` *algorithm on the template shown in Figure 4.2 with the diff* $\Delta^{\text{update}}(visitors\langle k_1\rangle.lat;56)$ *provided as its input. The* `TemplateDP` *algorithm will begin the top-down traversal of the template until it reaches the* **`let`** *statement shown in lines 3-6. Since the* `sql` *active function is not dependent on the incoming diff (in fact this active function generated this diff in the first place), when the* `ActiveFunctionDP` *algorithm is invoked, it will attempt to iterate over all path parameters of the active function, but since there are none, the active function will be ignored. The next* **`let`** *statement contains the invocation of the* `nest_by` *delta function (for simplicity reasons, we assume that the* **`let`** *statement in lines 8-11, which contains multiple active function invocations has been replaced by a sequence of* **`let`** *statements each containing a single active function invocation, as depicted in Figure 4.4). When this directive is visited, the algorithm* `ExpressionDP` *identifies that the* `nest_by` *expression is an active function and therefore invokes the* `ActiveFunctionDP` *algorithm (lines 20-21 in Figure 2). The* `ActiveFunctionDP` *algorithm attempts to find if an incoming diffs targets a path parameter or any descendants of the node identified by a path parameter. Note that only parameters of type* `path` *are considered (the* `nest_by` *function takes three parameters, only the first of which is a path. The function appears in the template as:* `nest_by(visitors, 'country', 'vs')`*, therefore the last two arguments are of type string). Since the incoming diff* $\Delta^{\text{update}}(visitors\langle k_1\rangle.lat;56)$ *mutates a descendant of the visitors variable, the active function is affected by the incoming diff, and the function* `invoke_delta_or_simulate` *will be invoked, in order to perform the partial evaluation of the* `nest_by` *function. Specifically, the* `invoke_delta_or_simulate` *function, identifies that the incoming diff, matches the delta function* `updtVal` *(the declaration of which is shown in lines 5, in Figure 5.5. Note that the path signature of this delta function for the particular invocation*

is visitors.$\langle * \rangle$.!country, as we described in Section 5.3.1). The respective delta function will generate the diff $\Delta^{\text{update}}(r1\langle USA\rangle.vs\langle k_1\rangle.lat; 56)$*, which is added to the input set of diffs by the* `TemplateDP` *algorithm (in line 6).*

`<% print` E `%>` *(or* **`<% bind`** $x = E$ **`%>`***) directive.* The treatment is similar to that of the let directive. The algorithm first computes the diffs to the expression E. Then it adjusts the paths of the produced diffs by prefixing them with the location of the current directive in the ATI (this path is denoted by $\widetilde{t}$).

`<% for` x **`in`** E **`%>`** B **`<% end for %>`** *directive.* In the case of for directives, the algorithm first computes the diffs to the output of the expression E that is being iterated over. Then the algorithm follows a case-analysis based on the type of change. (a) If the diff changes the entire value of E (lines 13-16), the algorithm reevaluates the for directive on the new value of E (by calling the `instantiateTemplate` subprocedure on the sub-template rooted at the for-directive. (b) If the diff deletes an element from the collection returned from E (lines 17-18), then that diff is simply added to the set of output diffs, (c) If the diff appends (lines 19-22) or inserts a new element into the array (lines 28-31) or bag (lines 24-27) returned by E, then the algorithm assigns to the loop variable y the new element (payload of newly created diff), adds it to the environment and instantiates the body B of the for-directive. (d) If the diff targets an element within the collection returned by E, then the algorithm replaces the prefix of the diff that leads to the for-directive (where the expression E resides) with the loop variable y, adds it to the set D'_{in} and then recursively invokes `TemplateDP` while using as a template the body B of the for-directive, in order to construct a diff that targets the nested element of the template instance residing within the body of the for-directive (lines 32-36). (e) Lastly, the `TemplateDP` algorithm, recursively invokes itself by providing as a template the body of the for-directive, and as a path the path signature that points to all the ATI subtrees that have been created by the current for-directive (lines 38-39), in order to deal with cases where there are diffs targetting variables that do not depend on the for-loop variable y.

Example 6.3.2 *(a) Consider the invocation of the* `TemplateDP` *algorithm when processing the for-loop appearing in line 41 in the template shown in Figure 4.2 when the only incoming diff is* $\Delta^{\text{update}}(countries; p)$*, where the payload p contains the new value of the countries collection. In this scenario, the* `TemplateDP` *algorithm will recognize that this belongs to the case (a) (described above, processed in lines 13-16 of the* `TemplateDP` *algorithm), since the diff targets the entire expression E of the for-directive (i.e., countries). This will cause the instantiation of the entire template rooted at the for-directive (lines 42-51 in Figure 4.2) and an update diff with payload the instantiated template that targets the for-directive instance (in the ATI) will be generated and added to the set of output diffs. Specifically, this diff will be* $\Delta^{\text{update}}(\hat{s}.markers\langle 0\rangle; a)$*, with the path* $\hat{s}.markers\langle 0\rangle$ *targeting the ATI node that corresponds to the for-directive instantiation and a being the instantiation of the body of the for-directive.*

(b) Consider the invocation of the `TemplateDP` *algorithm when processing the for-loop appearing in line 41 in the template shown in Figure 4.2 when the only incoming diff is* $\Delta^{\text{delete}}(countries[1])$*. In this scenario, the* `TemplateDP` *algorithm will recognize that this belongs to the case b (described above, processed in lines 17-18), since the diff describes the deletion of an element in the collection returned by the expression E of the for-directive (i.e., countries). In this case, the diff created by the* `ExpressionDP` *algorithm targets the appropriate part of the ATI (i.e.,* $\Delta^{\text{delete}}(\hat{s}.markers\langle 0\rangle for[1])$*, with the path* $\hat{s}.markers\langle 0\rangle for[1]$ *targeting the child ATI node of the for-directive instantiation with ordinal* 1*), and is simply added to the set of output diffs.*

(c) Consider the invocation of the `TemplateDP` *algorithm when processing the for-loop appearing in line 41 in the template shown in Figure 4.2 when the only incoming diff is* $\Delta^{\text{insert}}_{\text{array}}(countries[1]; p)$*, where the payload p is the value that will be added to the collection returned by the expression E of the for-directive (i.e., countries). The* `TemplateDP` *algorithm will recognize that this belongs to the case c (described above, processed in lines 28-31), since the diff targets an element that appears within the collection countries. In this case, the algorithm will add to the environment the variable y that has as content the payload of the diff*

returned by the `ExpressionDP` *algorithm and will invoke the instantiation of the template B (i.e., the body of the for-directive). Lastly, a new diff of type insertarray will be created with path equal to the target of the diff returned by the* `ExpressionDP` *algorithm (that points to the for-directive instantiation in the ATI) and payload equal to the instantiated payload (i.e.,* $\Delta^{\text{insert}}_{\text{array}}(\hat{s}.markers\langle 0\rangle for[1]; a)$*, with the path targeting the child ATI node of the for-directive instantiation with ordinal* 1 *and the payload a maintaining the instantiated if-directive).*

(d) Consider the invocation of the `TemplateDP` *algorithm when processing the for-loop appearing in line 41 in the template shown in Figure 4.2 where the only incoming diff is* $\Delta^{\text{update}}(countries[1].vs\langle 0\rangle.lat; 56)$*. The* `TemplateDP` *algorithm will recognize that this belongs to the case d (described above, processed in lines 32-36), since the diff targets an element that appears within the collection countries, therefore the diff* $\Delta^{\text{update}}(country.vs\langle 0\rangle.lat; 56)$ *will be generated and added to the set of input diffs (line 35 and the* `TemplateDP` *function will be invoked recursively using as a template the body B of the for-directive and path the path that leads to the particular for-directive instantiation in the ATI (the ordinal of which is obtained from the path step* [1] *in this example). If the* `TemplateDP` *invocation returns a set of diffs, it will be added to the set of output diffs (line 36).*

`<% if` E `then %>` B **`(<% elif` E_i `then %>` B_i`)`*** **`(<% else %>` B_n`)`**? **`<% end if %>`** *directive (condition directive).* Once a condition directive is visited, the TemplateDP algorithm, invokes the ConditionDP algorithm (shown in algorithm 4). For simplicity, the algorithm assumes that `else` statements are converted into `elif` statements by using as an expression the negation of the conjunction of all the expressions appearing in the preceding `if` and `elif` statements. We will be calling the `if` and `elif` statements contained in a condition directive, **condition statements**, and the respective expressions **condition expressions**. Before we describe the steps followed by this algorithm, we should mention the following points that only impact condition directives:

- The order in which condition statements appear in a condition directive affects the outcome

```
...

   <% if attr1 %>  // Δ^update(attr1; true)
      B1
    <% elif attr2 %> // Δ^update(attr2; true)
      B2
    <% end if %>
        // both attr1 and attr2 evaluate to true but since
        // the condition for attr1 appears first only B1
        // will get instantiated
  ...
```

(a) ConditionDP first example

```
...
// both attr1 and attr2 evaluate to true but since
// the condition for attr1 appears first only that condition
// is active. When the incoming diff is applied,
// the second condition needs to become active instead.
<% if attr1 %>  // true (initially active)
                // Δ^update(attr2; false) - false
  B1
<% elif attr2 %> // true (active after applying incoming diff)
  B2
<% end if %>
  ...
```

(b) ConditionDP second example

```
...
<% if attr1 %>  // initially true - Diff turns it into false
  B1
<% elif attr2 %> // initially false - Diff turns it into true
  B2
<% end if %>
  ...
```

(c) ConditionDP third example

of the algorithm. Specifically, if two consecutive condition statements have expressions that both evaluate to true at any given point, only the body of the first statement should be instantiated and displayed; we call this: **active condition** and the respective expression: **active expression**. For that reason, if two diffs target the expressions of two consecutive condition statements, causing them both to evaluate to true, only the diff that mutates the first statement should contribute to the generation of an output diff, 6.4a depicts this case.

- The output of all the condition expressions must be preserved during the time the respective template is displayed, as they can affect the output of `ConditionDP`. Consider a scenario (depicted in Figure 6.4b) in which there is only one incoming diff, that targets the expression of the currently active condition, thus causing it to evaluate to false. In this scenario the algorithm should not simply generate a delete diff that destructs the body of the active condition, as it is possible for a subsequent condition expression to still coerce to true. In such case, the ConditionDP algorithm should instead generate an update diff with payload equal to the instantiated body of the first condition statement that coerces to true.

- Incoming diffs should not be handled independently. Consider two consecutive condition statements, the first with an expression that initially evaluates to true (therefore it's initially active) and the other one evaluating to false. Also consider two incoming diffs: the first one targeting the second expression thus causing it to evaluate to true, and the second diff targeting the first expression thus causing it to evaluate to false. In this scenario, the expected behavior is the generation of an update diff with payload equal to the instantiated body of the second if statement. If, however, we apply the first diff independently from the second, no output diff will be generated, since both statements currently evaluate to true (so the first statement will remain active). When we apply the second diff, we will issue a delete diff that destructs the body of the first statement, thus causing an erroneous state.

Condition Directive Diff Propagation. The ConditionDP algorithm (shown in algorithm 4) begins by applying all the diffs to the annotated template instance (lines 5-12). Specifically,

for the expression of each condition statement, the ExpressionDP algorithm is invoked (line 8); if there is an incoming diff that targets the current expression, a set containing that diff will be created, otherwise that set will be empty. Due to the nature of condition expressions, the generated diff can only be of type Update and the payload can either be true or false. Once a diff is generated (9-11), the algorithm applies it to the annotated template instance.

Once this step has been completed, the algorithm retrieves the expression that was active during the previous invocation for the currently explored condition directive. If no expression was active, then none of the condition statement bodies was evaluated (and shown in the view). After this step, the algorithm identifies the first expression that coerces to true in this cycle (lines 15-17). If no condition statement was active during the previous invocation, and an expression coercing to true is found, then an insert diff will be generated, that targets the current condition directive, and has payload equal to the instantiated body of the current condition statement (lines 19-22). This diff, once constructed, is added to the set D'_{out} and the annotated template instance is updated to reflect the currently active expression. If the previously active expression and the currently coercing to true expression are the same (lines 26-28), then none of the diffs modified the active expression. In this case, the algorithm will simply recursively invoke the TemplateDP algorithm by passing as template, the body of the currently active condition statement, in order to identify diffs that may target it. If the two expressions are different (lines 29-35), then an update diff will be generated targeting the current condition directive with payload equal to the instantiated body of the current condition statement. This diff is added to D'_{out} and the respective expression is set as active. Lastly, if all condition expressions coerce to false (lines 37-42) and there was some active condition statement during the last invocation, then a delete diff is generated, targeting the current condition directive and the template instance is updated to reflect the fact that no active expressions exist in this cycle. Note, that this algorithm, ensures that, up to one diff will be returned for each condition directive (instead of one per condition statement), thus minimizing the number of diffs that target the ATI. By doing this, the propagation algorithm "pushes down" the process of calculating effective diffs (similar to how in modern database

optimization algorithms, we tend to push down selections and projections to limit the number of tuples that need to be calculated in subsequent steps).

6.4 From annotated template instance diffs to template instance diffs

As we mentioned in the previous subsection, the output of the algorithm 42 is a set of diffs that target the Annotated Template Instance (ATI). Since the ATI maintains constructs (annotations) that correspond to for-loop and if-statement instantiations, the respective diff paths and payloads contain tokens that represent such instantiations. The diff signature, however, that is provided by unit developers when a renderer is declared, does not contain any annotated tokens, so in order for FORWARD to locate and invoke the matching renderers for each diff, it must first prune such tokens from the path of diffs. Furthermore, since each renderer expects the payload of input diffs to be a JSON object, nodes that correspond to internal annotations must also be pruned from diff payloads as well. Algorithm 6 depicts this process.

While pruning nodes that correspond to for-loops and if-directives from the payload is trivial, since such nodes are simply replaced by their static descendants (which represent JSON values), this process is more involved for paths that contain ordinals. The reason is that due to the high expressivity of FORWARD's template language, ordinals that appear after annotation tokens in ATI diff paths cannot be directly mapped to the corresponding ordinals of the TI diff paths. Below are some cases in which FORWARD needs to recalculate the ordinals that appear in a path:

1. In cases when a template contains multiple nested for-loops, thus flattening nested arrays (each nested array could also have variable length, thus making this process even more challenging).

2. When condition directives filter from the template instance collection elements that are present in template variables (and the ATI), thus creating a mismatch between the ordinal

positions of the unfiltered elements in the template variable collection and the ordinal position of the respective template instance collection.

In order to illustrate the issue, we will be using the sample snippet that appears in Figure 6.5a. This snippet shows two nested for-loops, iterating over each visitor of each element in the 'groups' variable, followed by a condition directive that filters-out inactive visitors and another condition directive that specifies the icon of the marker depending on whether this visitor is located in Greece or anywhere else in the world. Figure 6.6 shows the Annotated Template Instance subtree that corresponds to that snippet, assuming that the template variable groups is the one appearing in Figure 6.5b. Figure 6.7 shows the respective Template Instance subtree.

Now assume that the location of the last visitor of the last group, changes to the value: x : 35 (the diff describing this mutation is the following $\Delta^{\text{update}}(\textit{countries[1].visitors}[1].\textit{location.x}; 35)$), in this case, the output of the TemplateDP algorithm will be a set that contains the diff: Δ^{update}(...`markers`$\langle$`0`$\rangle$`<for>[1]<0> <for>[1]`$\langle$`0`$\rangle$`cd`$\langle$`0`$\rangle$`cs`$\langle$`0`$\rangle$`cd` $\langle$`1`$\rangle$`cs.lat`, `35`). The tokens `<for>`, `cd` and `cs`, correspond to instantiations of for-loops, condition directives and condition statements respectively.

Before FORWARD attempts to identify the renderers that are capable of propagating those changes to the view, it must first convert those diffs into diffs that describe the respective mutations on the template instance, by invoking the algorithm shown in 6. The output of this algorithm is a set that contain the diffs: Δ^{update}(...`markers[2].lat`, `35`). In order to compute the correct ordinal that must appear in the path of the output diff, the algorithm will identify groups of tokens that correspond to annotated nodes (in this example there is only one such group in the diff: $\langle$`0`$\rangle$`<for>[1]<0> <for>[1]`$\langle$`0`$\rangle$`cd`$\langle$`0`$\rangle$`cs`$\langle$`0`$\rangle$`cd`$\langle$`1`$\rangle$`cs`), then for each of them it will find the ATI node that is targeted by the subpath appearing before the group of tokens (`...markers`) and it will perform a DFS traversal in which the nodes are visited in the order they appear (from left to right). During this traversal, only annotated nodes are visited, if the currently visited node is of type `condition directive`, only the active child (if any) will be

```
1  ...
2    markers: [
3          <% for group in groups %>
4            <% for visitor in group.visitors %>
5              <% if visitor.active then %>
6                <% if visitor.country == 'Greece' then %>
7                 {
8                   lat: <% print visitor.location.x %>,
9                   lng: <% print visitor.location.y %>,
10                  icon: 'url1'
11                }
12                <% else %>
13                {
14                  lat: <% print visitor.location.x %>,
15                  lng: <% print visitor.location.y %>,
16                  icon: 'url2'
17                }
18                <% end if %>
19               <% end if %>
20           <% end for %>
21         <% end for %>
22   ]
23   ...
```

(a) Template snippet used to add color-coded markers on a map

```
1  groups = [{
2    { visitors: [
3       { ..., active: false,...},
4       { ...,
5         active: true,
6         country: 'Greece',
7         ...
8       }
9    ]},
10   { visitors: [
11     { ...,
12       active: true,
13       country: 'USA',
14       ...
15     },
16     { ...,
17       active: true,
18       country: 'Finland',
19       ...
20     },
21   ]}
22 ]
```

(b) Model

Figure 6.5. Template and template instance for the running example

visited. When a node that corresponds to a JSON value is identified, this algorithm increases a counter by one. This traversal finishes when the ATI subtree that is bound by the group of annotated tokens is fully traversed and the ordinal position that replaces that group is the current value of the counter.

ALGORITHM 6: Transform ATI Diffs into TI Diffs

1 **function** `ATI_DP`(*Annotated Template Instance ATI, Set of ATI Diffs* S_{in})
2 $S_{out} \leftarrow$ Empty Set ;
3 **for** ***each*** $\Delta^{type}(\hat{t}; p) \in S_{in}$ **do**
4 $\hat{t} \leftarrow$ Prune from path $\hat{t}$ dynamic nodes that correspond to for-loop and if-directives ;
5 $p' \leftarrow$ Prune from payload p dynamic nodes that correspond to for-loop and if-directives ;
6 $S_{out} \leftarrow S_{out} \cup \{\Delta^{type}(\hat{t}'; p')\}$
7 **end**
8 **return** D_{out};

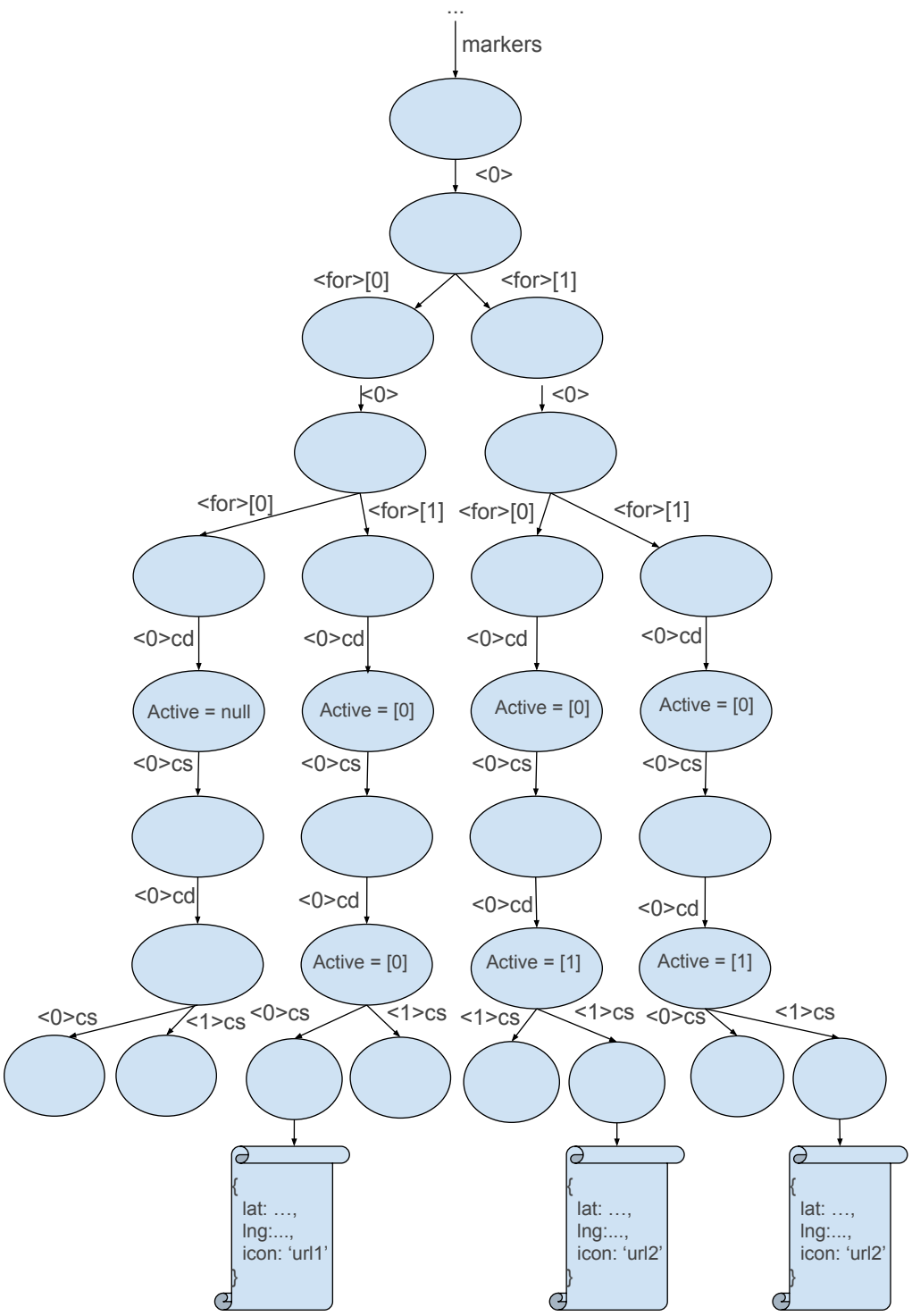


Figure 6.6. Annotated Template Instance subtree

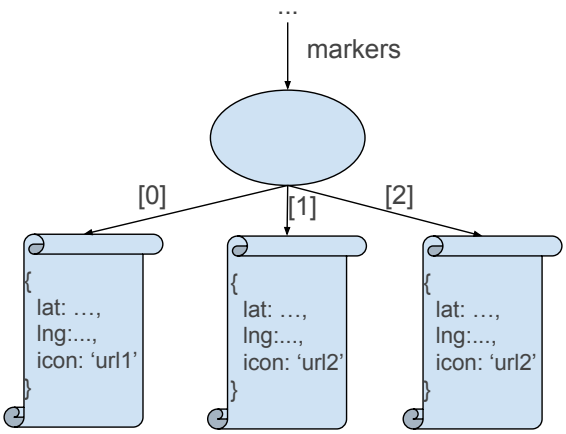


Figure 6.7. Template Instance subtree

6.5 Template Instance to Unit Instance (TI2UI) Propagation

In the previous section, we described how changes to the template variables, are transformed to corresponding changes to the template instance. In this section, we describe how these template instance diffs are translated to unit instance diffs, supported by the corresponding units. This process is not straightforward for two reasons: First, template instance diffs in general do not refer to a single unit instance, as they may span unit boundaries. Thus, they have to be translated into unit instance diffs, which refer to a single unit at a time. Second, as we explained earlier, a unit may not support all diffs. To guarantee that the application works even when a non-supported unit instance diff appears, the framework has to find an equivalent diff that is supported. These two issues are solved by the fragmentation and simulation modules, described next.

Fragmentation. Given a template instance diff, the fragmentation module translates it into one or more unit instance diffs that refer to a single instance at a time. A template instance diff of the form $\Delta^{type}(\hat{p}; v)$ is split into multiple unit instance diffs in two cases: (a) when its path $\hat{p}$ crosses unit boundaries and (b) when its payload v crosses unit boundaries (in which case FORWARD will create construct-unit and destruct-unit diffs to construct or destruct entire unit

instances).

Simulate update in array element with delete and insert
$\Delta^{\text{update}}(\hat{p}[k]; v) \rightarrow \triangle^{\text{delete}}(\hat{p}[k]) \cup \Delta^{\text{insert}}_{\text{array}}(\hat{p}[k]; v)$

Simulate update in bag element with delete and insert
$\Delta^{\text{update}}(\hat{p}\langle k\rangle; v) \rightarrow \triangle^{\text{delete}}(\hat{p}\langle k\rangle) \cup \Delta^{\text{insert}}_{\text{bag}}(\hat{p}\langle k\rangle; v)$

Simulate update to the root with construct and destruct
$\Delta^{\text{update}}(\hat{}; v) \rightarrow \Delta^{\text{destruct}}_{\text{unit}}() \cup \Delta^{\text{construct}}_{\text{unit}}(v)$

Simulate append array with insert array
$\Delta^{\text{append}}_{\text{array}}(\hat{p}; v) \rightarrow \Delta^{\text{insert}}_{\text{array}}(\hat{p}[\hat{p}.length]; v)$

(a) Same-level rules

Simulate diff with update of the parent
$\Delta^{type}(\hat{p}; v) \rightarrow \Delta^{\text{update}}(\hat{p}.up(); parent(apply(\hat{p}, \delta_{in})))$

(b) Parent-level rule

Note: δ_{in} denotes the input diff, $\hat{p}.up()$ returns the path of the parent node identified by $\hat{p}$ and $parent(v)$ returns the parent `JSON++` value of v

Figure 6.8. Simulation rules

Simulation. As explained above, a unit may not provide renderer wrappers for certain diffs. This happens typically for two reasons: Either because the unit developer, due to time restrictions, did not specify all incremental renderers, or because the API of the underlying JavaScript library, which the unit is wrapping, does not support certain changes. When such issues arise, FORWARD simulates non-supported unit instance diffs, with equivalent diffs for

which there are renderer wrappers.

Simulation rules. To simulate a diff, FORWARD rewrites it using *simulation rules*, which resemble rewrite rules traditionally used in query optimization. Due to the nested nature of the `JSON++` values, a diff targeting an element at a certain nesting level of the unit instance may be simulated by diffs either at the same level or at an ancestor. This is accomplished by the same-level and parent-level simulation rules, respectively.

According to a *same-level simulation rule*, a diff $\Delta^{type}(\hat{p};v)$ targeting an element at path $\hat{p}$ is simulated by one or more diffs at the same level. For instance, an update diff updating an element of an array may be simulated by a combination of two diffs: (a) a delete diff, deleting the element and (b) an insert diff, reinserting the element with the updated value. For instance, this could happen in our running example if the Google Maps unit did not support the update of existing markers but instead supported the deletion and insertion of markers. Another same-level simulation rule specifies that an update to the root of the unit instance can be simulated with a combination of destruct-unit and construct-unit diffs. Figure 6.8a shows all same-level simulation rules. The most interesting cases of simulation though happen when a diff cannot be simulated with diffs at the same level. In that case, FORWARD attempts to simulate a diff with a higher-level diff through the parent-level simulation rule, as explained next.

According to the *parent-level simulation rule*, a diff $\Delta^{type}(\hat{p};v)$ targeting an element e at path $\hat{p}$ is simulated using an update diff targeting e's parent. For instance, assume that the Google Maps unit does not support updates on the latitude attribute of individual markers (i.e., diffs of the kind: $\Delta^{\text{update}}(\cdot markers\langle 1\rangle.position.lat;56)$), but supports updates on the entire position attribute (that updates both latitude and longitude). In that case, if such a diff is constructed (denoted as δ_{in}), FORWARD would simulate it with an update diff that targets the entire position of the respective marker (i.e., $\Delta^{\text{update}}(\cdot markers\langle 1\rangle.position;v')$ with $v' = \{lat{:}56, lng{:}54\}$). Note that in order to compute the value v' of the new diff, FORWARD has to apply the original diff δ_{in} to the latitude of the appropriate marker (which we denote by $apply(\cdot markers\langle 1\rangle.position.lat, \delta_{in})$). To be able to achieve this simulation, FORWARD keeps track of the previous state of the unit

instances by maintaining at all times a materialized version of the template instance. Figure 6.8b depicts the parent-level simulation rule.

Simulation algorithm. FORWARD simulates a diff by combining the same-level and parent-level simulation rules as follows: Given a unit instance diff δ_{in} for the unit of type t, FORWARD checks whether unit t supports it. If it does not, FORWARD iteratively visits all same-level rules, checking whether their inferred diffs are supported by the unit. If they are not, then it applies the parent-level rule, replacing the incoming diff δ_{in} with the right-hand side of the rule and repeats the same process. The process ends whenever a unit supports the diffs generated during this procedure. Note that this algorithm is guaranteed to terminate, as any diff will in the worst case be translated to a combination of destruct-unit and construct-unit diffs, which are supported by any unit[1].

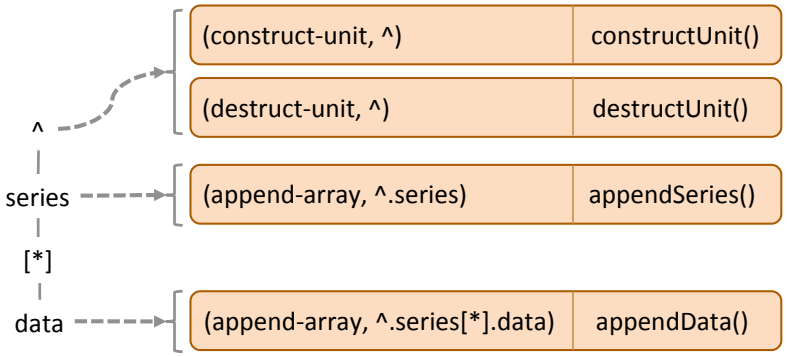


Figure 6.9. Diff signature trie: Efficient lookup of renderer wrappers

Accelerating simulation. Checking whether a simulation rule applies, involves iteratively checking whether there are renderer wrappers that support the diffs on the right-hand side of the rule. In a naive implementation, each right-hand side diff would be checked against all renderer wrappers. This could become very costly, as some units have a significant number of renderer wrappers. To accelerate this process, FORWARD leverages the fact that the

[1]This would happen through consecutive invocations of the parent-level rule, converting any diff to an update of the root and a final invocation of the same-level rule, specifying that an update to the root can be simulated by a destruct-unit diff and a construct-unit diff.

simulation algorithm operates at any point on a particular nesting level of the unit instance specification. Thus, instead of checking whether a diff is supported by any renderer wrapper, it simply checks against renderer wrappers that appear at the same level. To achieve this level-based lookup of renderer wrappers, FORWARD statically compiles a diff signature trie, whose nodes correspond to levels of the unit instance and whose edges follow the nesting structure of the unit instance. Each node of the trie has a pointer to all renderer wrappers targeting the particular level. Figure 6.9 shows the diff signature trie for the renderer wrappers of Figure 5.3.

6.6 Acknowledgements

This chapter contains material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Evaluation of Performance and Extensibility

Two major contributions of this dissertation, that are realized in FORWARD are (a) the change propagation algorithm and the performance benefits it offers when propagating changes to the visual layer and (b) the extensibility benefits that our architecture and the trigger-like language we introduced, offer when extending the framework with new template constructs. In this chapter we evaluate both these contributions (a) by comparing the performance benefits of our change propagation algorithm, against the algorithms employed by competing frameworks, and (b) by conducting a user study that evaluates how easy it is to introduce new template constructs to frameworks that support this feature.

7.1 Performance of change propagation algorithms

Recall that, as part of this dissertation, we designed a novel change propagation algorithm that is inspired by incremental view maintenance techniques used in database systems. A common denominator of IVM algorithms is the efficient propagation of changes to (database, or in this case, application) views. In this chapter, we evaluate the efficiency of FORWARD's change propagation algorithm against the algorithms employed by existing web application frameworks. An analysis of the conceptual model and the mechanics of the propagation algorithms employed by existing web application frameworks has been presented in section 2.3.

7.1.1 Experimental setup

Based on the way change propagation algorithms are designed and implemented, their complexity and, therefore, runtime performance is affected by either (a) the total size of the visualized data (ViewModel), or (b) the total number of the propagated changes. Additionally, as we described, in 1.5.3, many MVVM frameworks, do not interface with visualization libraries and therefore can only automatically propagate changes to HTML content. In order to conduct performance experiments, that allowed us to control the aforementioned parameters, we implemented 3 different dashboards:

1. *Simple line chart.* A dashboard that contains a single line chart was used to enable the maximum amount of control over (a) the total number of propagated changes and (b) the total size of the visualized data points (ViewModel). By only including a single chart, there are no other parts of the visual layer (and ViewModel) that could impact the runtime of the change propagation algorithms, which allows us to isolate the parameters that actually affect the runtime of each algorithm.

2. *Fully-fledged monitoring dashboard.* A fully fledged web dashboard, that contains a relatively dense visual layer, that comprises both HTML content and reactive visualizations, was implemented in order to obtain performance data in a more realistic use case.

3. *Dashboard containing only textual content.* A Dashboard that only contains textual content was used to evaluate the performance of FORWARD's change propagation algorithm against MVVM frameworks that support change propagations on HTML content

Eliminating bias

To ensure that the architecture and the capabilities of the machine used for the experiments do not affect the runtime of the propagation algorithms, we ran the experiments on the first two dashboards on a PC and the last dashboard on a mobile device. The PC has an Intel Core

i7-3517U 1.90GHz CPU and 7.66 GB RAM running Ubuntu 14.04.5 LTS (trusty), the browser used for the experiments is Google Chrome (version: 53.0.2785.116, 64-bit) and the performance measurements are collected using Chrome's built-in profiling tools. The mobile device (LG G4-H811) is running Android 6.0 (Marshmallow), with 1.8 GHz 64-bit Hexa-Core CPU (Chipset: Qualcomm Snapdragon 808) and 3 GiB of RAM. The dashboard runs inside a native wrapper that internally uses Android's WebView which is based on Google Chrome (version: 53.0.2785.124). Google Chrome's Remote Debugging tool is used to connect the mobile device to a desktop computer in order to collect the measurements presented in this section. Specifically, Google Chrome's profiling tools are used to retrieve measurements for each tested operation.

In order to offer a higher level of granularity while measuring the time it takes to propagate a change all the way to the visual layer we break the process down into two subprocesses:

- **The propagation time**, which is the runtime of the propagation algorithm from the moment it is invoked until right before the appropriate renderer is called and
- **The rendering time**, which is the time it takes for the browser to render the changes to the visual layer

Naturally, if the rendering time is greater than the propagation time, it does not matter how efficiently the propagation algorithm propagates changes to the ViewModel, since any performance benefits would be overshadowed by rendering.

7.1.2 Dashboard 1 - Simple line chart

The first dashboard contains a simple line chart that visualizes historic data about the temperature of a UCSD conference room. Figure 7.1 shows a snapshot of the visual layer of said dashboard. This dashboard gives us the flexibility to control (a) the total size of the ViewModel by initializing the chart with a number of data points (x) before applying smaller changes, and (b) the total size of applied changes by adding a new data series, of increasing size (y), that represents historic data about a different conference room.

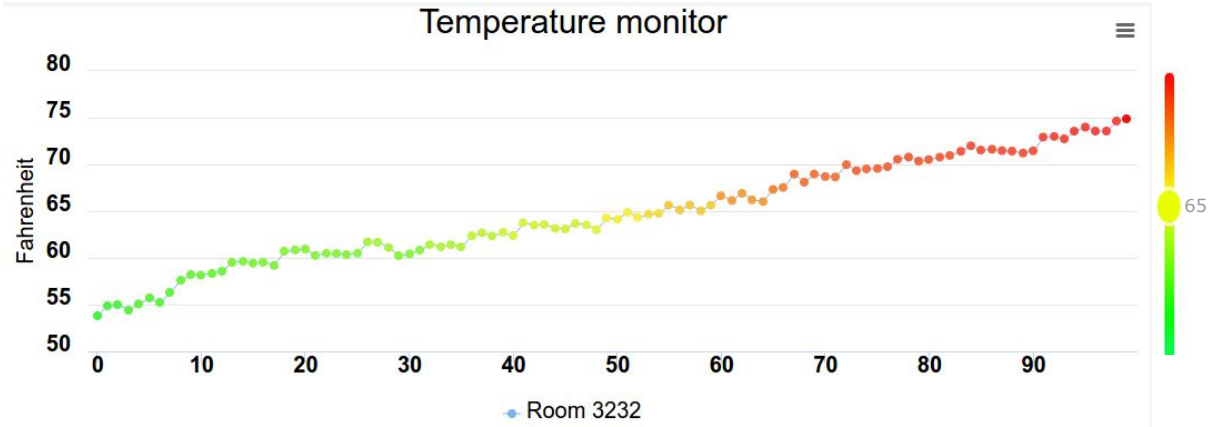
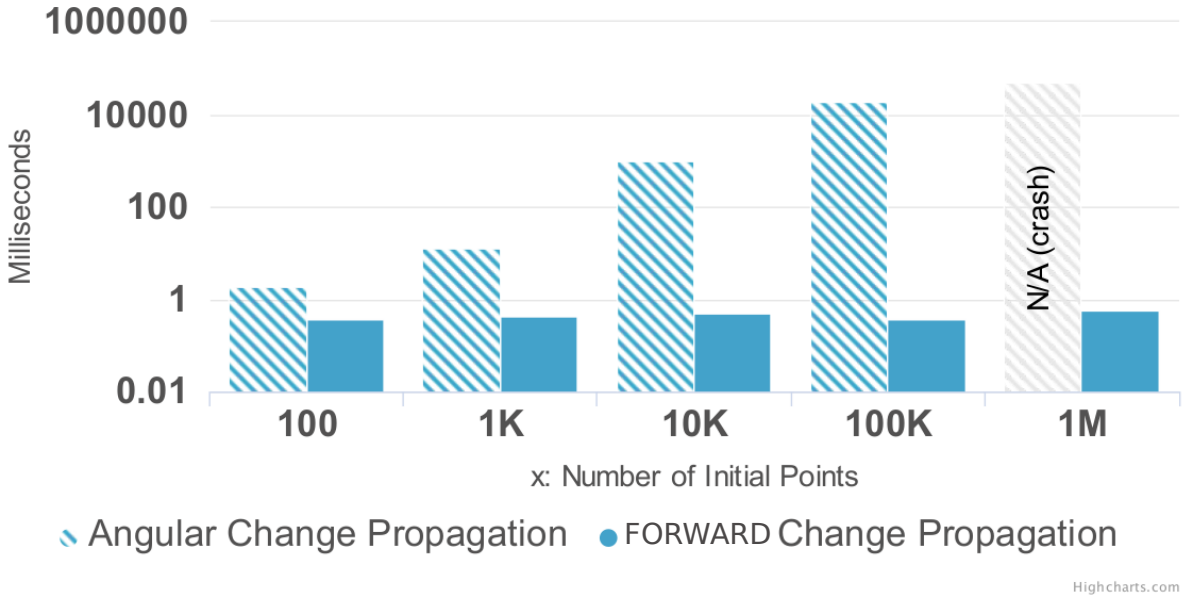


Figure 7.1. Dashboard 1 - Simple line chart



	AngularJS		FORWARD		Speedup
Initial Points (X)	Change Prop.	Rendering	Change Prop.	Rendering	Total
0.1K	1.9	46.2	0.4	45.1	1.1×
1K	12.7	24.8	0.4	24.7	1.5×
10K	1003.1	86.9	0.5	80.5	13.5×
100K	18626.9	188.0	0.4	145.5	129.0×
1000K	–	–	0.6	539.1	–

Figure 7.2. Performance comparison between FORWARD and AngularJS: Time required for propagating a new data point into an existing data series that comprises X points

Dashboard implementation details

For the implementation of this dashboard we used Highcharts as the underlying visualization library. Since, FORWARD and Angular are the only frameworks that automate the propagation of changes to visualizations, we only included these two frameworks in this experimental evaluation. Additional frameworks were used in the experiments presented later in this chapter. Highcharts was appropriately wrapped for use by the two frameworks: In the case of AngularJS, we used the highcharts-ng [62] directive (recall that AngularJS directives are equivalent to FORWARD's visual units), which is the most popular Highcharts directive publicly available (with 1,600 stars and 450 forks on GitHub), while in the case of FORWARD, we used an in-house developed visual unit that wraps the identical set of renderers as the selected Highcharts directive.

Performance evaluation

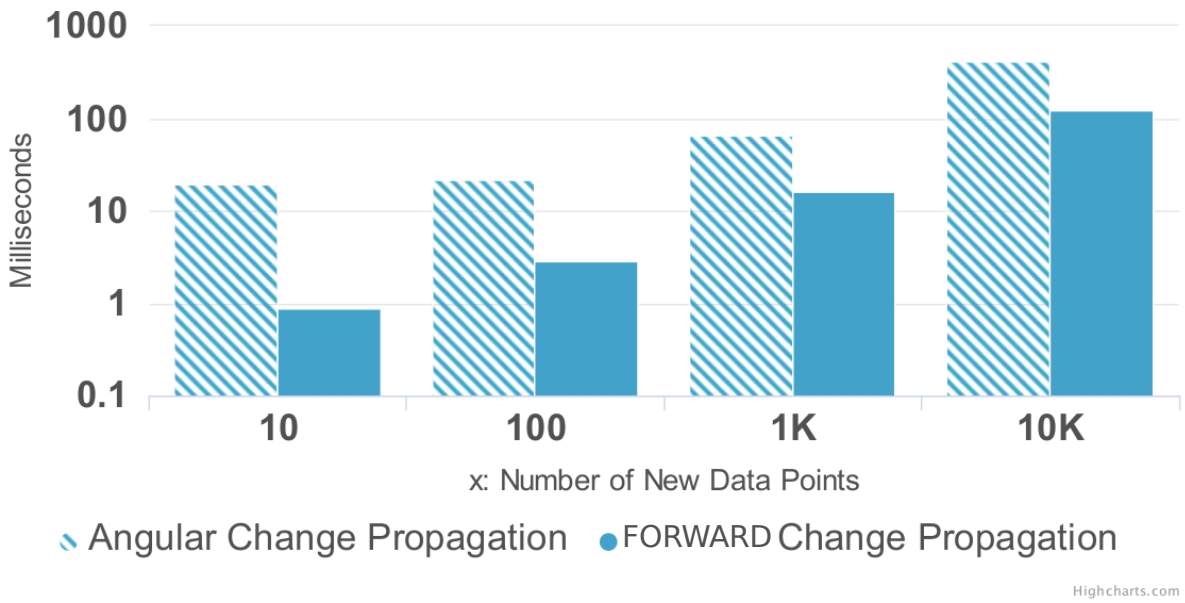
Comparison 1: Maintaining the number of changes fixed while increasing the size of the ViewModel. For this comparison, we first kept the number of changes fixed while increasing the total size of the ViewModel. Specifically, the frameworks were used to first create a dashboard, populate it with a certain number of data points (denoted as x) and measured the time needed to insert a new data point to the chart. We then iteratively, reconstructed the dashboard using a higher number of initial data points (x) and again, we measured the time needed to insert a new data point to the chart.

Change propagation runtime of Angular vs. FORWARD. In the chart shown in Figures 7.2, the left and right bars depict the time it takes for AngularJS's and FORWARD's change propagation algorithms to propagate a change for each x (a smaller bar indicates a better performance). Note that the y axis in both charts use a logarithmic scale. As observed from Figure 7.2 propagating changes in FORWARD is consistently more efficient than in Angular. Moreover, FORWARD's IVM-inspired change propagation algorithm is not influenced by the size of the ViewModel (total number of measurements that appear on the page), as it stays

consistent at about 0.5ms. This, however, does not hold true for AngularJS. While AngularJS's propagation algorithm (Digest Cycle) takes initially a few milliseconds to propagate the mutation (for $x = 100$ and $x = 1K$) it soon increases almost exponentially. From $x = 10K$ points and after, Angular's propagation algorithm is simply not viable for live applications. Lastly, we should mention that we were unable to run any experiments for AngularJS when $x = 1M$ points, since the application simply becomes unresponsive and crashes.

As we described in section 2.3, the propagation algorithms used by frameworks such as Angular and React, iteratively reevaluate the ViewModel in full, and then compare its pre-state with its post-state. This is why we notice that the change propagation of Angular is deteriorated as the total size of the ViewModel increases. FORWARD's propagation algorithm however, as we described in section 6.3, simply traverses the template for each incoming diff and infers if any expressions would be affected, thus causing its incremental evaluation. Therefore, the complexity of this algorithm is proportional to the size of the template, which is constant.

Change propagation runtimes vs. respective incremental rendering times. Comparing the runtime of a change propagation algorithm against the time needed for incremental rendering a change is also crucial. Recall that the incremental rendering time, is the time it takes for the browser to reflect the change to the visual layer once the respective renderer is invoked. If the dominating factor while propagating changes end-to-end was the rendering time, then there would be no need for more efficient change propagation algorithms. Figure 7.2, contains incremental rendering times. Notice that rendering remains relatively the same across frameworks, which is expected. Minor differences on rendering performance across frameworks could be due to the fact that FORWARD's propagation algorithm is able to invoke renderers that mutate smaller parts of the visual layer (which is one of the contributions of FORWARD), thus making incremental rendering slightly more efficient. Note that Angular's change propagation algorithm initially takes less time than incremental rendering for up to 1K data points. For $x = 10k$ points and higher, Angular's change propagation algorithm incurs a significant overhead (the break-even point was at $x = 1.2K$). FORWARD's change propagation algorithm, however,



	AngularJS		FORWARD		Speedup
Data points of new series	Change Prop.	Rendering	Change Prop.	Rendering	Total
10	19.72	43.8	0.88	35.38	1.8×
100	21.64	57.4	2.88	52.58	1.4×
1K	65.44	81.9	16.4	78.36	1.6×
10K	419.04	295.92	122.74	271.24	1.8×

Figure 7.3. Performance comparison between FORWARD and AngularJS: Time required for propagating an entire new series

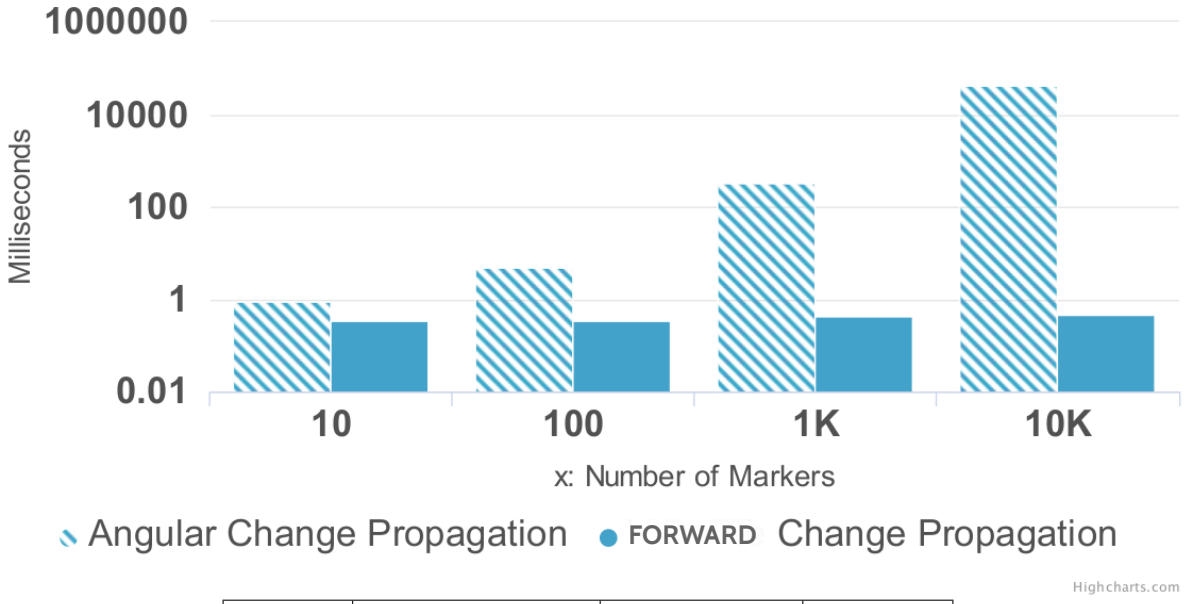
incurs minimal overhead of about 0.5 ms, that is consistent regardless of the size of the visual layer.

Comparison 2: Holding the size of the ViewModel constant while increasing the number of applied changes. To explore the effects that the number of mutations has on the performance of each framework, we created the same chart but now kept the initial size of the ViewModel fixed, while increasing the size of the propagated mutations. To accomplish this we constructed our chart with a single series with 1,000 data points and then iteratively added an entire new series (visualizing historic data about a different conference room) with an increasing number of data points (10, 100, 1,000 and 10,000).

Change propagation runtimes of Angular vs. FORWARD. As shown in Figure 7.3

both Change Propagation algorithms are affected by the size of the mutation applied. This seems somewhat counter-intuitive for Angular at first, as we would expect that the runtime of the change propagation algorithm would remain the same while the total number of changes increases. This expectation stems from the fact that the ViewModel will be reevaluated and compared with its pre-state in full, regardless of the number of individual changes that were applied to the ViewModel. As a matter of fact we do notice, that the rate of runtime increase, in Angular's change propagation algorithm, when the data points of the new series increase from 10 to 100, is not as big as when they increase from 1K to 10K. This is because the ViewModel already contains a series with 1K data points which impacts the change propagation algorithm at first, but as the data points of the new series exceed the 1K data points, they impact Angular's change propagation runtime more. On the flip side we notice that FORWARD's change propagation runtime increases proportionally to the size of mutation. We also notice that FORWARD's change propagation is again consistently more efficient than AngularJS's (by 1.4-1.8x) despite the fact that they are both increasing.

Change propagation runtimes vs. respective incremental rendering times. Notice that rendering remains relatively the same across frameworks, which is expected. Minor differences on rendering performance across frameworks could be due to the fact that FORWARD's propagation algorithm is able to invoke renderers that mutate smaller parts of the visual layer (which is one of the contributions of FORWARD), thus making incremental rendering slightly more efficient. Note that AngularJS's Change Propagation algorithm, is initially more efficient than the respective Rendering step, but after about 3K data points it starts surpassing it, thus becoming the main bottleneck, while FORWARD's Change Propagation step is consistently more efficient than Rendering. Therefore once again we notice that FORWARD introduces a smaller overhead.



	AngularJS		FORWARD		Speedup
Markers	Change Prop.	Rendering	Change Prop.	Rendering	Total
10	0.88	0.5	0.34	0.5	1.6×
100	4.8	0.44	0.34	0.46	6.6×
1K	314.482	0.54	0.44	0.46	350.0×
10K	41257.26	0.72	0.46	0.56	40449.0×

Figure 7.4. Performance comparison between FORWARD and AngularJS: Time required for mutating the position of a delivery truck

7.1.3 Dashboard 2 - Fully fledged monitoring dashboard

For this experiment, we implemented the truck monitoring dashboard that was described in Section 2.2.3. The visual layer of said dashboard is shown in Figure 2.4. Recall that this uses a map to display the real-time location of delivery trucks (represented as markers) throughout the United States. This dashboard represents a more realistic view of the kinds of dashboards developers would be implementing, as it contains a richer visual layer that contains textual content and a plethora of visualizations. For this experiment we, once again, used FORWARD and Angular as they are the only frameworks that automate the propagation of changes to visualizations.

Dashboard implementation details

In order to generate the map and attach markers to it, we used the LeafletJS [74] library. To use LeafletJS in FORWARD we used an in-house developed unit while for AngularJS we used an existing publicly available directive [21] (that has 1,426 stars and 666 forks on GitHub). In this experiment, we wanted to measure the time needed to update the position of a delivery truck to the view, while the total number of delivery trucks that are initially shown increases. Therefore, once the dashboard had been constructed with a set number of markers (y), we applied a single mutation to the application state and measured the time needed to propagate the change to the visual layer. We subsequently increased the total size of the application state and visual layer (by using a bigger array of delivery trucks) and we again applied the same mutation to the application state. Figure 7.4 shows the time required by the frameworks to reflect the changes in the two dashboards by each framework. Once again, for each framework, the cost of this task is broken down into two parts: The first part corresponds to the runtime of the change propagation algorithm, and the second part corresponds to the cost of incremental rendering by the browser.

Performance evaluation

Maintaining the number of changes fixed while increasing the size of the ViewModel. For this comparison, we kept the number of changes fixed while increasing the total size of the ViewModel. Specifically, the frameworks were used to first create a dashboard, that visualizes a certain number of delivery trucks (denoted as x) and measured the time needed to update the position of an existing trucker to the visual layer. We then iteratively, reconstructed the dashboard using a higher number of initially visualized delivery trucks and we measured the time needed to insert a new data point to the chart.

Change propagation runtime of Angular vs. FORWARD. In the chart shown in Figure 7.4, the left and right bars depict the time it takes for AngularJS's and FORWARD's change propagation algorithms to propagate a change for each x (a smaller bar indicates a better performance). Note that the y axis in both charts use a logarithmic scale. As observed from this Figure propagating changes in FORWARD is consistently more efficient than Angular. Moreover, FORWARD's IVM-inspired change propagation algorithm is not influenced by the size of the ViewModel (total number of measurements that appear on the page), as it stays consistent at about 0.34 - 0.46 ms. This, however, does not hold true for AngularJS. The runtime of AngularJS's propagation algorithm (Digest Cycle), starts at 0.88 ms at which point only 10 delivery trucks are displayed and it keeps increasing until it reaches more than 41 seconds for 10K delivery trucks (almost exponential increase). From $x = 1K$ points and after, Angular's propagation algorithm is simply not viable for live applications.

Change propagation runtimes vs. respective incremental rendering times. Notice that rendering is almost identical across frameworks. Furthermore, notice that the rendering time of the LeafletJS library is not influenced by the total size of visualized items (ViewModel). Rendering takes between 0.44 and 0.72 ms to complete. AngularJS's propagation time, however, starts at 0.88 ms at which point only 10 markers are displayed and it keeps increasing until it reaches more than 41 seconds for 10K delivery trucks. Therefore the change propagation runtime

incurs significant performance penalties, as it the main bottleneck during end-to-end change propagations. FORWARD's propagation change propagation runtime however is consistently lower than the respective rendering time. The minimal overhead it offers makes it the better framework for live visualizaitons.

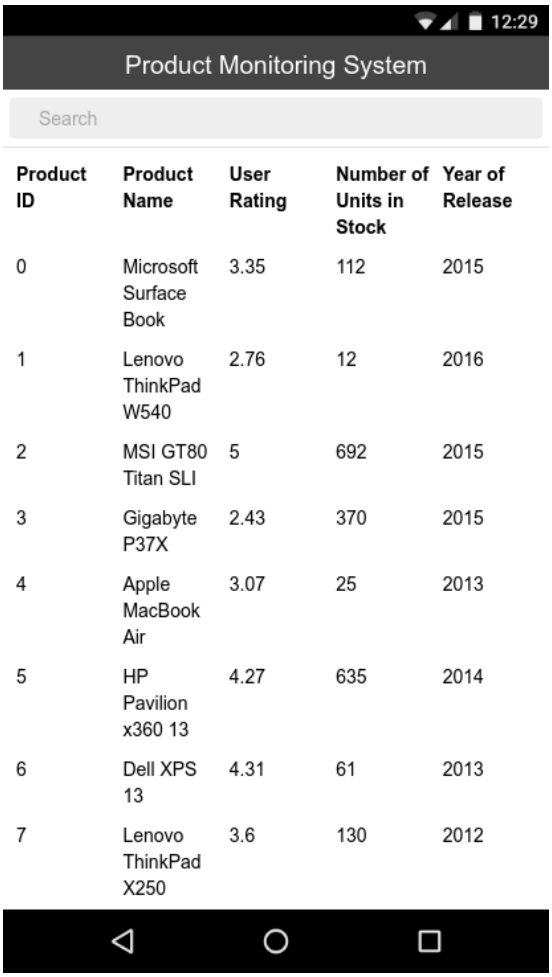
7.1.4 Dashboard 3 - Containing only textual content

Recall that, as we discussed in Section 1.5.3, many MVVM frameworks, do not interface with visualization libraries and therefore can only automatically propagate changes to HTML content. In order to compare the performance of the change propagation algorithms of such frameworks, we implemented a dashboard, that contains only HTML content, using FORWARD, Angular, React and Angular2 and then evaluated their performance during change propagation.

Figure 7.5 shows the dashboard that was used for this experiment. The dashboard is monitors live information about the product availability at a company's warehouse. Technically, this screen is rendered as an HTML table, with each row corresponding to an individual product and each column to a particular attribute of a product, such as product ID, name, user rating, stock availability and year of release. The screen also contains a search box that simplifies the process of obtaining information about individual products. As the user types the name of a particular product the application automatically scrolls up or down in order to show the row that contains information about the selected product while at the same time it automatically highlights the product name (similarly to how the search functionality works on a desktop browser). It is important to note that, this system is live, therefore the screen is updated whenever the underlying data, such as the user rating or the stock availability of a product, change.

Dashboard implementation details

To see how these frameworks perform with regard to change propagation, we used them to implement the aforementioned monitoring system. Since all competing frameworks operate strictly on the client, we used FORWARD active functions to query a database, instantiate the



Product ID	Product Name	User Rating	Number of Units in Stock	Year of Release
0	Microsoft Surface Book	3.35	112	2015
1	Lenovo ThinkPad W540	2.76	12	2016
2	MSI GT80 Titan SLI	5	692	2015
3	Gigabyte P37X	2.43	370	2015
4	Apple MacBook Air	3.07	25	2013
5	HP Pavilion x360 13	4.27	635	2014
6	Dell XPS 13	4.31	61	2013
7	Lenovo ThinkPad X250	3.6	130	2012

Figure 7.5. Running Example: Product Monitoring System

model and propagate it to the client. We then used the template language and API provided by each framework to generate the HTML table appearing in Figure 7.5. After this step was completed we began modifying the product attributes in the underlying database tables which in turn triggered our IVM algorithm. Upon this event, the FORWARD active function pushed a model diff to the client for each mutation that was applied to the database table. After the model diff was transmitted to the client, we manually applied the respective change to the model of the application and triggered the change propagation algorithm for each framework while measuring the time needed for the full propagation. Since all the steps prior to the invocation of the change propagation algorithm were common across frameworks, the respective time needed for their

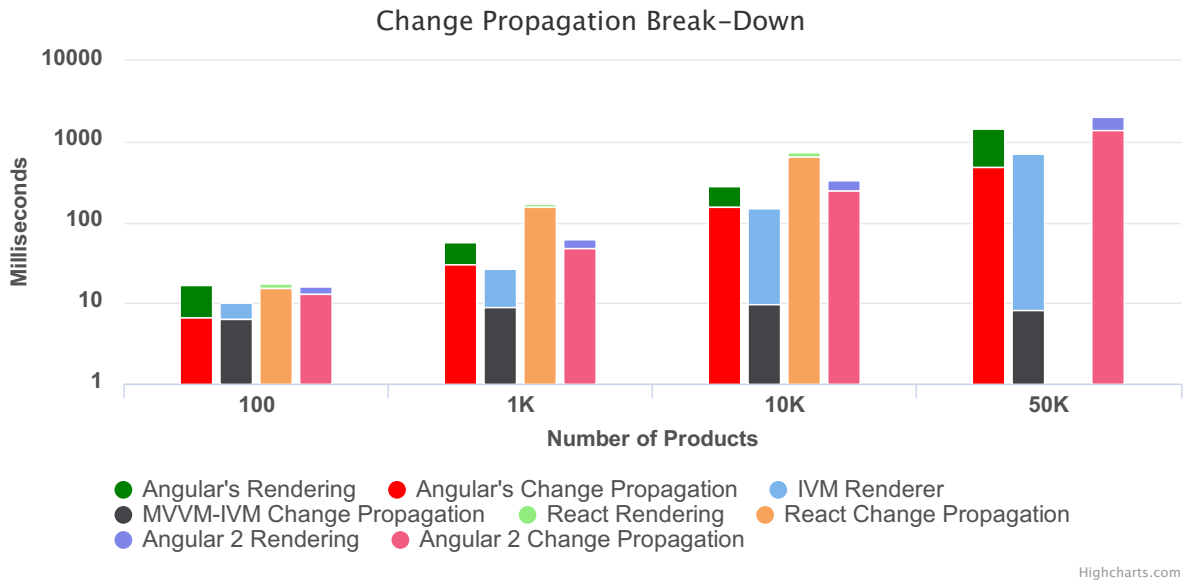


Figure 7.6. Performance comparison breakdown for increasing number of displayed products (note that the scale on y axis is logarithmic, so top bar might appear smaller than what it really is. For absolute numbers refer to table 7.1).

completion was also the same. What is not the same, however, is the time required for each change propagation algorithm to propagate the Model mutation and call the appropriate renderer that mutates the visual layer.

Maintaining the number of changes fixed while increasing the size of the View-Model. In order to explore how the change propagation algorithm is affected by the size of the visual layer (and the ViewModel), we kept the size of changes fixed and gradually increased the size of visual layer. Specifically, we first constructed a screen with a number of products x and measured the time required for the change propagation algorithm and the renderer to propagate a mutation on the User Rating for each product. We then began increasing the total number of products appearing on the screen while still modifying the User Rating of each product.

Change propagation runtime of Angular against competing frameworks. Figure 7.6 graphically depicts the time required by each framework to apply a mutation to the view. The X axis shows the number of products that appear in the view and the Y axis shows the time in milliseconds that was needed for the completion of each task. Each full bar corresponds

to the total time needed by each framework to fully propagate the changes to the visual layer, which is broken down to (a) the time required by the change propagation algorithm and (b) the time required by the incremental renderer. Note that the scale on y axis is logarithmic, so top bar appears smaller than the lower bar. In Table 7.1 we provide the absolute numbers of each framework.

As is easily observed from the table is the 7.1, the propagation algorithm of FORWARD is consistently more efficient than the respective propagation algorithms offered by the other frameworks. Additionally, note that the runtime remains roughly the same as the total size of the visual layer increases. Additionally, as we notice in the same table, there are no measurements for ReactJS when the total number of products appearing in screen is equal to 50K products. The reason is that the ReactJS framework was unable to generate the view and the application became unresponsive for an extend amount of time, which forbids us from collecting the respective measurement.

Change propagation runtimes vs. respective incremental rendering times. As we observe both from Figure 7.6 and the table 7.1 rendering performs fairly similar for all frameworks, the slight edge that Angular2 and ReactJS appear to have in rendering performance is appointed to the fact that they invoke slightly more precise renderers that update smaller parts of the view. Notice all propagation algorithms, other than FORWARD's, perform much worse than the respective rendering step for bigger ViewModels. This is interesting, because incremental rendering on HTML is a notoriously expensive computation. Still FORWARD's propagation algorithm remains consistently low even if the runtime needed for incremental rendering is increasing.

Discussion. It is worth mentioning that 30 fps is the minimum frame-rate that needs to be supported by an application in order to ensure a seamless user experience. What that means is that, each stage involved in mutating the view should be completed in about 33.3 ms, anything more than that results in a deteriorated user experience. Despite the fact that propagation algorithms appear to be independent of the rendering process, that is not the case.

Table 7.1. Performance comparison breakdown (into runtime of change propagation algorithm and runtime of rendering) for each framework

	MVVM IVM		AngularJS		Angular 2		ReactJS	
Products	Change Propagation	Rendering	Change Propagation	Rendering	Change Propagation	Rendering	Change Propagation	Rendering
100	6.49	3.588	6.61	9.79	13.044	3.118	15.306	2.492
1K	9.052	17.746	29.736	28.05	46.972	15.088	154.542	14.238
10K	9.792	137.462	152.568	124.124	247.956	85.156	646.682	79.078
50K	8.214	696.334	493.576	961.644	1390	607.826	-	-

In fact any operation that takes more than 33.3 ms to complete can negatively affect the user experience due to the single-threaded nature of JavaScript. For that reason the minimal overhead that is added by FORWARD's propagation algorithm is desired because not only it ensures the immediate propagation of mutations to the view without delays but at the same time it does not interfere with the user experience of the application.

7.2 Extensibility

Recall that one of the contributions of this dissertation is that we designed (and implemented as part of FORWARD) an easy-to-use trigger-like language that significantly simplifies the development of new template constructs. This positively impacts the extensibility of the framework which in turn positively affects the generality of the framework, as dashboard developers can import reusable template constructs that facilitate the declarative implementation of any potential dashboard. In this section, we demonstrate that implementing template constructs that generate visualizations is easier in FORWARD compared to Angular, which is the only existing web framework that supports wrapping visualization libraries in reusable template constructs. Recall that FORWARD is the only framework that supports the development of template constructs that declaratively describe **data access** and **data processing**, so we could not compare ease of development of such constructs against other frameworks.

7.2.1 Evaluation setup

In order to evaluate ease of extensibility, we performed the following steps:

Table 7.2. HighCharts Component wrapping comparison: In-house built FORWARD visual unit vs. Most popular open source AngularJS directive

	FORWARD	AngularJS
Lines of Code for Directive/Visual Unit	126	474
Testing	185 (for 100% coverage)	138 (for 54% coverage)

- **In-house visual unit vs. publicly available Angular Directive**. We implemented a HighCharts visual unit and we compared it to the most popular Angular directive available on GitHub. Specifically, we performed a "lines-of-code" comparison between the two modules. Additionally, we identified and reported the pain-points that were responsible for the additional lines of code for the Angular Directive.

- **Developer-built visual units vs. publicly available Angular Directives**. We recruited two undergraduate UCSD-CSE students and asked them to implement two visual units. We then compared the two units against the respective Angular directives. The two units wrap (a) the Google Maps visualization library and (b) the Leaflet.js visualization library. Both these libraries can be used to construct reactive maps and have several similarities regarding the APIs they expose. Once the visual units were implemented we performed a "lines-of-code" comparison between them and the most popular respective Angular directives we found on GitHub.

- **Developer-built visual unit vs. Developer-built Angular Directive**. Lastly, we asked the two students to implement the respective Angular directives while limiting the lines of code needed. We then performed "lines-of-code" and "development-time" experiments between the visual units and the respective Angular directives they implemented. As we will describe in Section 7.2.5 only one of the two developers was able to finish this task, and in an effort to limit the lines of code they used third-party libraries that automated several tasks that typically have to be implemented by the creators of Angular directives.

Table 7.3. Component wrapping effort comparison: FORWARD & AngularJS

	FORWARD	AngularJS
Total number of lines of code	126	474
Number of functions	12	11
Avg number of lines per function	3.3	39.7
Maximum indentation	2	11

7.2.2 In-house visual unit vs. publicly available Angular Directive

In order to examine how the two frameworks perform with regard to unit wrapping, we developed a FORWARD visual unit that wraps the exact same Highcharts renderers that are supported by the most popular AngularJS directive we found on GitHub and then compared the number of lines of code of each wrapper. Furthermore, we implemented unit tests to ensure that our wrapper is operating properly, and then performed a "lines-of-code" comparison against the tests provided by the AngularJS Directive developer. Table 7.2 summarizes the experiments. For the same number of renderer functions the FORWARD unit needed 126 lines of code while the respective AngularJS directive needed 474 lines of code. Table 7.3 provides additional insight regarding the code in these two cases. As we observe, the average lines of code per function is significantly smaller in the FORWARD visual unit compared to the Angular Directive. Additionally, the logic within each function is simpler and easier to follow as it is shown by the maximum indentation (smaller indentation means less nested functions for-loops and if-then-else statements, higher nesting generally means that code is more complicated and harder to follow). The Figure also contains the lines of code of the respective unit tests. Specifically, the unit tests provided by the AngularJS directive are 138 lines long and only cover 54% of the lines the highcharts directive contains. This means that almost half of the directive (specifically 218 lines) has not been adequately tested and could, therefore, contain bugs. On the other hand, FORWARD's unit tests are 185 lines long and cover the entire 100% of the unit.

Table 7.4. Leaflet Component wrapping comparison: Developer-Built FORWARD visual unit vs. most popular open source AngularJS directive

	FORWARD	AngularJS
Lines of Code for Directive/Visual Unit	590	5734
Testing	812 (for 100% coverage)	962 (for 51% coverage)

Table 7.5. Google Maps Component wrapping comparison: Developer-Built FORWARD visual unit vs. most popular open source AngularJS directive

	FORWARD	AngularJS
Lines of Code for Directive/Visual Unit	461	14176
Testing	791 (for 100% coverage)	2930 (for 63% coverage)

7.2.3 Developer-built visual units vs. publicly available Angular Directives

In order to eliminate bias while evaluating ease of extensibility, we hired two undergraduate students and asked them to wrap a visualization library into a FORWARD visual unit. The two students developed a Google Maps and a Leaflet visual unit. Once the two units were implemented we found the most popular Google Maps and Leaflet Angular Directives on GitHub and we compared them in terms of "lines-of-code". The comparison also contains the lines of code needed for unit testing which ensure the proper operation of the template construct. Table 7.4 summarizes the experiments for the Leaflet template construct and Table 7.5 summarizes the experiments for the Google Maps template construct. As we observe, the lines of code needed for the Leaflet template construct are 590 for FORWARD and 5734 for Angular. The difference is an order of magnitude. Additionally the lines of code needed for unit testing are 812 for FORWARD for 100% coverage and 962 for Angular that only provide 51% coverage. The difference is even larger for the Google Maps template constructs as the FORWARD construct only requires 461 lines of code while the Angular construct required 14176 lines. Unit testing is only 791 for 100% coverage for FORWARD and 2930 for 63% coverage for Angular.

7.2.4 What makes Angular component wrapping challenging

There are two main reasons why there is a difference in the lines of code between the two template constructs:

- **Caching UI Objects**. During the development of template constructs developers need to construct objects that contain renderer functions that once invoked incrementally update the visual layer. We call such objects *UI Objects*. In order to apply incremental changes to the visual layer these objects need to be cached by construct developers. In FORWARD developers can use the *setUiObject* and *getUiObject* functions (described in Section 5.3) to automatically cache and obtain a UI Object given its path, in Angular however this caching has to be performed manually by the construct developer as the framework does not provide any utilize functions that simplify this task. As a result, appropriate data structures have to be introduced and maintained in Angular which increases the number of lines of code needed for the implementation of a template construct.

- **Identifying mutations manually**. Another reason for the difference in the required lines of code is that AngularJS does not provide a functionality equivalent to FORWARD's (a) Simulation and (b) Subsumption check. Recall that Simulation in FORWARD ensures that if a given incoming diff does not match an existing delta function that is capable of reflecting that change to the visual layer, FORWARD automatically generates an equivalent diff or diffs that have a supporting delta function. Additionally, Subsumption check automatically ensures that given two diffs A and B, with A being more generic (mutating a bigger part of the visual layer) than B, then B is either ignored (if A already contains the respective change described by B) or is merged into A. Since Angular does not provide Simulation and Subsumption check, directive developers have to perform these tasks manually. More specifically, during initialization, directive developers have to manually navigate the ViewModel object in order to declare watchers that observe individual parts of it for changes. When those parts of the ViewModel are mutated, a

callback function that is provided by the directive developers is triggered. This function is responsible for identifying the appropriate renderer that is capable of causing the minimum possible mutations to the view and for generating the objects that will be used as parameters to the renderer functions.

Angular Watchers. Recall that there are three main types of watchers currently provided by AngularJS, a) shallow watchers, are triggered when the observed part of the ViewModel object is mutated b) deep watchers, are triggered when the observed ViewModel object or any descendant is mutated and c) collection watchers, are triggered when the observed array changes in size. Since a mutation on one part of the ViewModel might trigger multiple watchers, a directive developer has to be cautious not to provide callback functions that, when triggered simultaneously, will invoke renderers that might bring the component into an erroneous state. This is the primary reason why directive developers often declare a deep watcher near the root of the ViewModel and then proceed manually comparing the pre-state to the post-state of individual descendant objects in order to identify what exactly changed and so that they call the appropriate renderer.

Example 7.2.1 *Figure 7.7 shows a deep watcher used by the HighCharts directive to observe all descendants of the path 'config.series' for mutations. When a mutations occurs, the function that appears in lines 2-6 will execute, and the function processSeries (line 4) will be called with the post-state and the pre-state of the observed part of the ViewModel that is identified by the path 'config.series' as input. The processSeries function (which is also provided by the directive developer) performs all the aforementioned operations that attempt to identify which specific part of the ViewModel was modified so that ultimately the respective renderer will be called. Indicatively, the processSeries function is 99 lines long (155 if we inline all the functions that are called from it) and is deeply nested, due to the relatively high number of if-then-else and for-loop statements that are needed to identify all possible mutations. This nesting however makes this function very difficult to understand, debug and extend.*

```
scope.$watch('config.series',
  function (newSeries, oldSeries) {
    var needsRedraw = processSeries(newSeries, oldSeries);
    if(needsRedraw) {
      chart.redraw();
    }
  }, true);
```

Figure 7.7. Series Wrapper in AngularJS

Table 7.6. Component wrapping lines-of-code comparison: Developer-built FORWARD visual unit vs. Developer-built AngularJS Directive

	FORWARD	AngularJS
Lines of Code for Directive/Visual Unit	461	431
Testing	791	904
Total Lines	1252	1335

Example 7.2.2 *In figure 7.8 we show the respective part of the FORWARD Highcharts unit that propagates all possible mutations that might arise to the path 'config.series' or any descendant. In the first 6 lines the unit developer declaratively specifies the renderer wrapper function that will be invoked when an incoming diff is matched. Specifically, in the first two paramaters of the addRenderer function the unit developer specifies the opcode and the path signature an incoming diff must match in order for the renderer wrapper function (which is the third parameter) to be invoked. This declarative way of specifying rendering wrappers favors modularity and results in self-contained independent wrappers while at the same time minimizes boiler plate code.*

This modularity of FORWARD units also favors unit testing. This is why FORWARD's unit tests are typically smaller than the respective unit tests of the angular directive.

Table 7.7. Component wrapping development-time comparison: Developer-built FORWARD visual unit vs. Developer-built AngularJS Directive

	FORWARD	AngularJS
Hours spend on Directive/Visual Unit implementation	10:15	26:20
Hours spend on testing	7:20	8:00
Total hours	17:35	34:20

```
1  this.addRenderer(Op.DELETE, '^.series[*]', deleteSeries);
2  this.addRenderer(Op.APPEND, '^.series[*].data', appendToData);
3  this.addRenderer(Op.UPDATE, '^.series[*].data', updateData);
4  this.addRenderer(Op.UPDATE, '^.series[*]', updateSeries);
5  this.addRenderer(Op.APPEND, '^.series', appendToSeries);
6  this.addRenderer(Op.UPDATE, '^.series[*].visible',
7        updateSeriesVisibility);
8
9
10 function deleteSeries(unit_instance, diff){
11     var series = unit_instance.getUiObject(diff.target);
12     series.remove();
13 }
14 function appendToSeries(unit_instance, diff) {
15     var chart = unit_instance.getUiObject('^');
16     chart.addSeries(diff.getPayload(), true);
17 }
18
19 function appendToData(unit_instance, diff) {
20     var one_up_path = diff.getTarget().up();
21     var series = unit_instance.getUiObject(one_up_path);
22     series.addPoint(diff.getPayload(), true, false, false);
23 }
24
25 function updateData(unit_instance, diff) {
26     var one_up_path = diff.getTarget().up();
27     var series = unit_instance.getUiObject(one_up_path);
28     series.setData(diff.getPayload(), true, false, false);
29 }
30
31 function updateSeries(unit_instance, diff) {
32     var series = unit_instance.getUiObject(diff.getTarget());
33     series.update(diff.getPayload());
34 }
35
36 function updateSeriesVisibility(unit_instance, diff) {
37     var one_up_path = diff.getTarget().up();
38     var series_object = unit_instance.getUiObject(one_up_path);
39     series_object.setVisible(diff.getPayload());
40 }
```

Figure 7.8. Series Wrapper in FORWARD

7.2.5 Developer-built visual unit vs. Developer-built Angular Directive

In an effort to eliminate the issues we described in the previous section, we requested from the two students to implement the respective Angular directives. Specifically, the developers were asked to minimize the lines of code needed for the development of Angular directives by (a) defining a flat (non-nested) ViewModel, in order to avoid having to manually perform subsumption checks and simulation and (b) by employing the same data structure that is used by FORWARD to maintain UI Objects. We then performed "lines-of-code" and "development-time" experiments between the visual units, implemented earlier and the respective Angular directives they implemented in this task. Due to the steep learning curve of this task, one of the students reported that they were not able to complete this task. Tables 7.6 and 7.7 summarize the lines of code and implementation time of the second student. Notice that the development of the Google Maps visual unit (implemented by the second user) required a very similar number of lines of code with the respective visual unit and the respective test. This is because by defining a flat ViewModel there were no complex cases of simulation and subsumption that had to be implemented and tested. Additionally, by utilizing a data structure that maintains the UI objects, the student did not have to perform this task manually and as a result this process did not have to be tested either.

7.3 Acknowledgements

This chapter contains material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 8

User Study: Evaluating the ease of dashboard development

As we have mentioned throughout this thesis, one of the most significant contributions of FORWARD is that it enables developers to easily build fully customizable reactive dashboards, regardless of their technical expertise. In this chapter, we describe the user study we conducted to evaluate this claim. Both qualitative and quantitative data were collected during this study about the time and processes needed to build the requested dashboard with and without FORWARD. The subjects' opinions and positive and negative observations about the framework were recorded, as well as information regarding the strategies they followed when performing individual tasks. This chapter starts by describing the user study design, which contains information about the target audience, the dashboard that subjects were asked to implement, the integrated development environment (IDE) they utilized and other procedural items. We then describe the recruiting process and lastly we describe and decipher the results of the study.

8.1 User study design

The goal of this study is to evaluate whether FORWARD actually enables its target audience to develop custom, reactive dashboards in an easy manner, without requiring advanced technical expertise or excessive time investment on their end. The subjects that participated in this study were carefully selected to match FORWARD's target audience (described in section

1.2). Special consideration was given to ensure that the subjects would not be influenced by the investigators, and that the study closely emulates real-life conditions in which developers typically implement dashboards.

8.1.1 Target Audience

As described in Section 1.2, FORWARD's main target audience is: (a) database developers, (b) data analysts, and (c) web developers. Based on the level of experience and technical expertise each audience has in dashboard development, we classified them respectively as (a) Novice Dashboard Developers, (b) Intermediate Dashboard Developers or (c) Advanced Dashboard developers:

- *Novice Dashboard Developers*: Must have a basic understanding of SQL and HTML but must not have any experience in building custom dashboards or visualizations.
- *Intermediate Dashboard Developers*: Must have a basic understanding of SQL and HTML and have experience with dashboard and visualization development, but must not be able to build fully customizable dashboards. These developers must be familiar with one or more charting/dashboard tools such as with Tableau, Vega Jupyter Notebooks, Plotly, or others.
- *Advanced Dashboard Developers*: Must have a good understanding of SQL and HTML, and must be able to build any custom dashboard. They must be comfortable using web tools and frameworks that allow them to build dashboards and web applications. These are typically professionals that work as full-stack web developers in tech companies and have experience using languages/frameworks used in web development (ReactJS/AngularJS, Ruby, Java, JavaScript or others).

Note that the classification we performed reflects only the level of expertise developers have specifically in dashboard development and not the total years of experience they have in

programming in general. The assumptions we made are (a) that novice developers do not know how to build dashboards, but perhaps know how to build pages with static textual content (no visualizations), (b) intermediate developers can generate individual charts or basic dashboards (using "form-and-report" tools such as Tableau, Oracle Wizard and more), but cannot build fully customizable dashboards such as the dashboard we requested, (c) advanced dashboard developers can build any custom, fully-fledged dashboard that visualizes live data and reacts to user input, without using FORWARD albeit in a considerable amount of time.

In an effort to avoid affecting the decisions and actions of the subjects, we decided to conduct the study in groups of 2 (using pair programming), thus eliminating the negative impact of approaches such as the Think-Aloud Protocol (which is also used in studies with human subjects)[42], in which researchers can unintentionally influence what participants say or do by asking them about their work [80]. To accurately emulate the process developers go through while building a dashboard, we identified two roles: (a) the dashboard developer and (b) the Internet proxy. *Dashboard developer* is the programmer who implements the dashboard, this role was played by the subjects. The *Internet proxy* emulates search engines (such as Google), documentation pages and online Q&A communities (such as StackOverflow and other forums) for FORWARD. This role was played by the investigators. The latter was essential, since under normal circumstances, developers are typically allowed to search on-line for posts, articles and documentation pages when trying to troubleshoot an issue. FORWARD, however, is still in alpha release phase, so there are no posts about it on-line (on StackOverflow and other forums), as a result, developers could not refer to such sources if they had questions. As a workaround to this issue, the Internet proxy was introduced, and developers could ask questions they would normally search online. Since the online presence is limited, we implemented an online tutorial that subjects completed (as we will describe later in this section) and a documentation page they can refer to as well. Lastly, in order to eliminate inconsistencies, the study was conducted in the same quiet lab. All subjects used the same IDE on the same Linux machine, connected to a 27 inch screen, for all parts of the study.

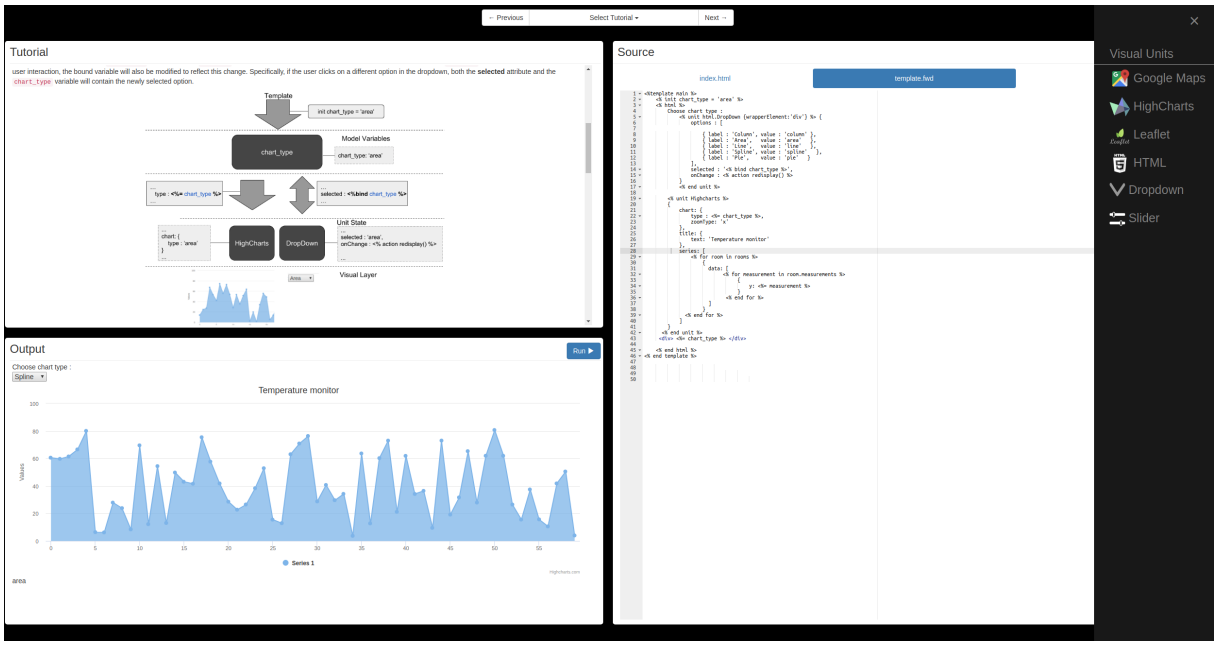


Figure 8.1. IDE used for the user study. Developers can (a) read instructions/documentation on the top left pane, (b) write code on the right pane (c) view the respective output on the bottom left pane. Using the toolbar that appears on the rightmost part of the page, developers can drag-and-drop a visual unit with a sample visual unit state in their editor to add a unit instance.

8.1.2 Integrated development environment (IDE)

Because in this modern time most developers expect and depend on an IDE to write and debug their code, we implemented an on-line IDE (shown in Figure 8.1), for this user study. This IDE was used both to facilitate the on-line tutorial (as we will describe later in this section) and the development of the requested dashboard by the subjects. It integrates FORWARD and all the employed active functions and visual units that were necessary for the study. The IDE (shown in Figure 8.1) allows developers to (a) read instructions/documentation at the top left pane, (b) write code at the right pane and (c) view the respective output at the bottom left pane. It also offers basic syntax highlighting, but does not offer auto-complete functionality, so developers were required to type in the template manually. Using the toolbar that appears on the rightmost part of the page, developers can also drag-and-drop a visual unit, with a sample visual unit state to the code editor, to add a unit instance. Developers were also allowed to copy-paste snippets they had already written themselves when completing the tutorial and edit them appropriately to complete

the task at hand. We recognize that this IDE is just a prototype and does not offer functionality that is typically supported by most IDEs, such as advanced debugging functionality (e.g., setting breakpoints), auto-completion and other features. This lack of features can only affect the results of this study negatively, but we were willing to accept this risk since the IDE offers an intuitive way of implementing dashboards using FORWARD, by presenting both FORWARD code and the respective output dashboard in a single window. Additionally, since all subjects used the same bare-bones IDE for the development of the dashboard, we eliminated inconsistencies that could introduce noise on development time between groups.

8.1.3 Tasks included in user study

The study contains three tasks, completed by the subjects in the following order: (1) complete an on-line tutorial (2) implement the requested dashboard using FORWARD and (3) complete the exit interview. Below we describe each part in detail:

- **The Online Tutorial**[1] illustrates the developer-facing programming model of FORWARD (which was described in Chapter 4) through a series of small running examples of increasing difficulty. Specifically, the tutorial starts by illustrating how to build simple pages that contain static textual content and gradually introduces features that enable the development of a more elaborate page that visualizes live data originating from remote database systems and reacts to user input. The tutorial emphasizes the overall programming model, the template language, and the template constructs (active functions and visual units) that facilitate the development of dashboards. After each tutorial, subjects have to complete a small exercise designed to surface any learning breakdowns they might have faced in the current tutorial. If a learning breakdown occurs, the subjects ask questions to the Internet proxy who answers them in detail. The online tutorial takes roughly 40-55 minutes to complete depending on the technical expertise of each group and the number of questions asked during this process.

[1]The online tutorial is available in the following URL: http://ec2.zarifis.info/ForwardTutorial/presentation_pages

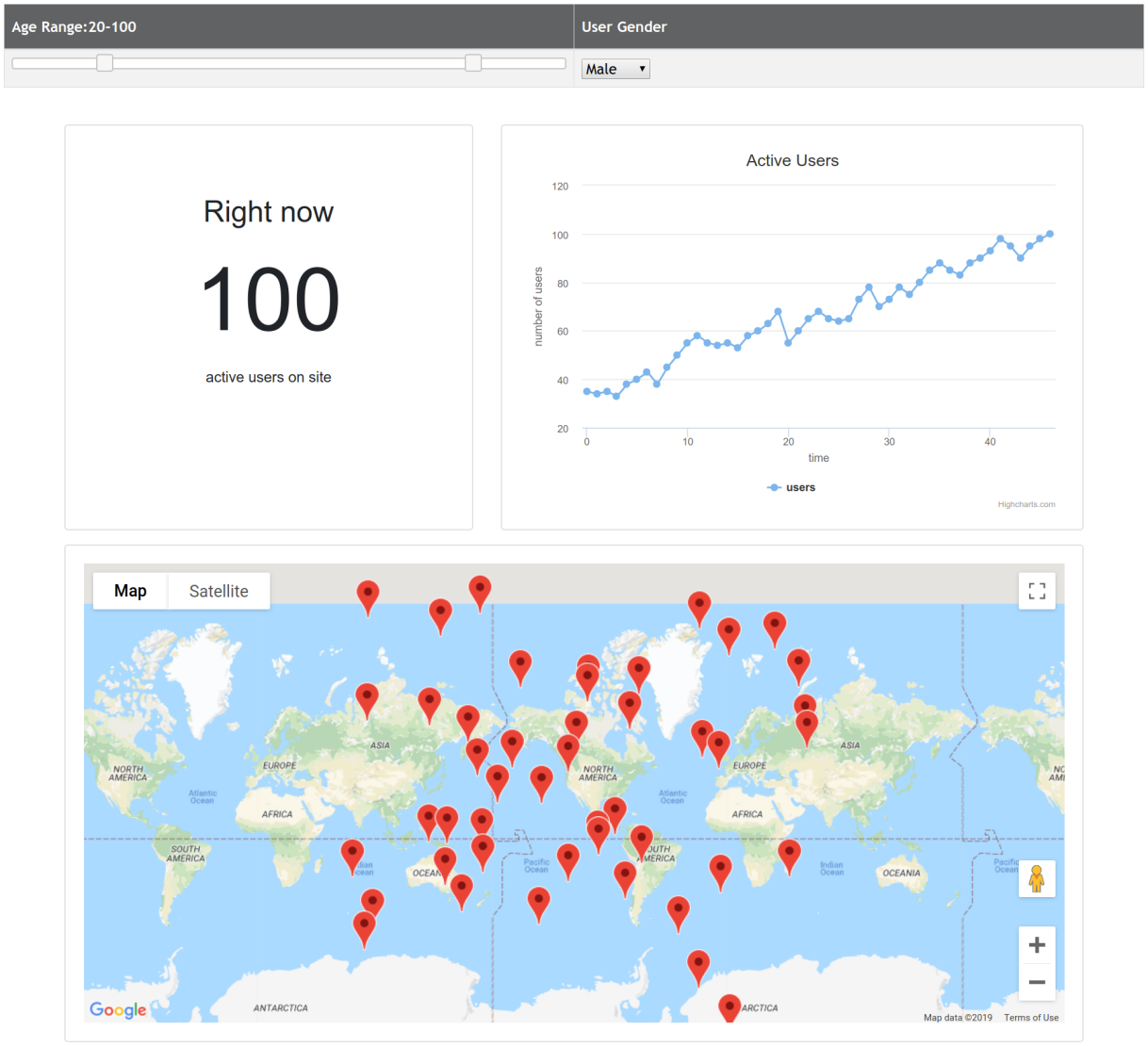


Figure 8.2. Snapshot of dashboard implemented by subjects. The dashboard gets updated real-time when the underlying data change. The visual layer is also updated as the user interacts with the slider and the drop-down that appears at the top of the page.

id	view_count
1	35
2	34
3	35
...	...

uid	lat	lng	name	gender	age
1	-28.2747	78.4102	Isabelle Costa	Female	20
2	80.034	1.5368	Andrew Gabellini	Male	23
3	-18.7925	-65.1007	Charlie Sharp	Male	25
...	...	...	...	...	...

Figure 8.3. Schema and sample data of database tables that were provided to subjects.

- **The Evaluation Dashboard**. Once the tutorial is completed, the subjects are asked to implement the dashboard shown in Figure 8.2. The dashboard is used to monitor user activity on a news portal website. At the top left corner, the screen displays the number of active users, that are currently visiting the website. On the right side, there is a chart showing historic data about the total number of the users that have visited the website up to this point in time and below there is a map graphically depicting each individual currently active user as a marker. When the user clicks on a particular marker, a popup window appears containing a chart that provides information about website articles the respective user read in the past few hours. At the top of the dashboard there is a slider that filters the displayed users based on their age and a drop down that filters users based on their gender (subjects had to implement this functionality as well). Lastly, the dashboard shows real-time data, so the view has to be continuously updated as new users are visiting the website, old users leave the website, or as users move around.

  The subjects were given an SQL database system that contains a table listing the users that are currently visiting the website and a table that maintains historic data about the users that have visited the website. Figure 8.3 contains the schema and sample data of

the two database tables. Additionally, subjects were given a basic template that contains placeholders in which they would need to add the appropriate visualizations (using visual units). Subjects were responsible for using active functions to obtain the appropriate data, process them accordingly, using other active functions and FORWARD's template language and feed them into the appropriate visual units. Lastly, they were responsible for using the appropriate FORWARD templates that makes the dashboard live and reactive.

- **Exit Interview**. Once the user study was completed, the subjects were asked to complete a questionnaire that includes high-level questions about the usability of the framework and potential issues and fixes they would like to be applied to the framework. Most importantly, one of the questions asks the subjects if they would be able to build the same dashboard without the use of FORWARD. If they would, they are asked to design the architecture of the system and estimate the time it would take them to implement each individual module. As will be seen in the results section, we could not perform a formal control, in which subjects are tasked to implement the same dashboard using a different framework, because no subject would commit to it, due to the extended amount of time this process would take (as we will see later in this chapter). Recall that we did hire an undergraduate student to implement a very similar dashboard (the running example of this dissertation, described in Section 1.1), and that took 48 hours without using a framework (Vanilla JavaScript) and 38 hours with Angular.

8.2 Recruiting subjects

For this user study we recruited 18 anonymous participants with 3 to 12 years of programming experience in academia or industry. We recruited subjects by (i) sending recruitment emails to mailing lists at UC San Diego, (ii) asking Professors and TAs to forward recruitment emails to students that qualify to participate in the study based on their background and the material taught in their class, (iii) reaching out to the engineers of a local tech company and (iv) by attaching

Table 8.1. Demographics and Labels of participants. The Group ID denotes the group Category (N:Novice, I:Intermediate, A:Advanced) + Unique identifier (1,2,3). The Subject ID denotes the Group ID (N1, N2 ... A3) + unique identifier of group participant (a, b).

Category	Group ID	Subject ID	Gender	Programming Experience
Novice Dashboard Developers	Group N1	N1a	Male	11 Years
		N1b	Male	7 Years
	Group N2	N2a	Female	6 Years
		N2b	Male	4 Years
	Group N3	N3a	Female	6 Years
		N3b	Male	3 Years
Intermediate Dashboard Developers	Group I1	I1a	Male	10 Years
		I1b	Female	12 Years
	Group I2	I2a	Male	10 Years
		I2b	Male	10 Years
	Group I3	I3a	Male	10 Years
		I3b	Female	8 Years
Advanced Dashboard Developers	Group A1	A1a	Male	11 Years
		A1b	Male	11 Years
	Group A2	A2a	Female	7 Years
		A2b	Male	5 Years
	Group A3	A3a	Male	12 Years
		A3b	Male	12 Years

Table 8.2. User study results. Implementing a dashboard with and without FORWARD

	Group N1 Novices	Group N2 Novices	Group N3 Novices	Group I1 Intermediate	Group I2 Intermediate	Group I3 Intermediate	Group A1 Advanced	Group A2 Advanced	Group A3 Advanced
With FORWARD	0:23:41	0:40:53	0:30:56	0:23:09	0:19:11	0:24:08	0:25:35	0:22:41	0:11:07
Average with FORWARD	31:49			22:08			19:47		
Without FORWARD	N/A	N/A	N/A	N/A	N/A[2]	N/A[3]	41-48 hours	56-112 hours[4]	46-106 hours[4]

recruitment posters on various poster boards around UCSD. Through this process, we recruited undergraduate and graduate students at UCSD, professional students that are attending UCSD's Data Science and Engineering Master's program, and professional developers working at a local tech company. All participants filled out a questionnaire before participating in the user study, which was used to group the subjects into 9 two-member groups of comparable experience and background. Information regarding the demographics of the participants and the groups they were assigned to is shown in Table 8.1

8.3 Results and Analysis

In this section we report and examine the qualitative and quantitative data we collected during the study and the information we obtained from the subjects through the questionnaires and the exit interviews. The primary focus of the study was to explore how effective FORWARD is in facilitating the development of custom dashboards. However, since the subjects interacted with the IDE we developed in-house, we will also be reporting their reactions towards the IDE and any recommendations for improvements they have mentioned.

[2]In this group, one subject reported that they would be able to implement the same dashboard using Angular in 36-46 hours, but does not know how to notify the application when mutations occur in the underlying database, so instead they would do polling and diff to find the changes. The other participant of that group would not be able to implement any part of the dashboard without FORWARD.

[3]In this group, one subject mentioned that they would be able to implement the basic dashboard in 16 hours but does not know how to support user interaction and live (push-event) functionality, the other subject would not be able to implement any part of the dashboard without FORWARD.

[4]The reason for the wide range, is because developers were only familiar with ReactJS and they were not sure if there exist react components that can incrementally render mutations that occur in the ViewModel, so the lower end of the range is an estimation for the case in which such a component exists and the higher end of the range is for the case in which they had to implement it themselves

8.3.1 Quantitative results about dashboard development with and without FORWARD

As seen in Table 8.2 the subjects completed the dashboard in 11 - 41 minutes using FORWARD. The same table also contains the average completion time for (a) novice developers, which is 31 minutes and 49 seconds, (b) intermediate developers (22 minutes and 8 seconds), and (c) advanced developers (19 minutes and 47 seconds). As seen in this table, subjects also reported an estimate of the amount of time they would need to implement the same dashboard, without FORWARD, using other frameworks and tools they are familiar with. Novice and intermediate developers, reported that they would not be able to implement the exact same functionality supported by the dashboard without using FORWARD(marked as N/A in the table), while advanced developers estimated that it would take roughly 41-112 hours, depending on the framework they would be using and depending on how thorough their unit and functional tests would be. We provide a more detailed analysis of these results in the following subsection.

8.3.2 Analysis of quantitative results

As seen from the results, the average completion time for novices (using FORWARD) was slightly higher than the intermediate developers, which in turn was higher than the advanced developers. This was expected, since more advanced developers are typically more familiar with frameworks that employ a declarative language to generate visualizations, so they were able to draw analogies between such frameworks and FORWARD. As was also expected, novice and intermediate developers reported that they would not be able to implement this dashboard without FORWARD, with two caveats:

- When asked to estimate the amount of time necessary for the development of the dashboard, without FORWARD, subject I2a (an intermediate developer), mentioned that they would be able to build a very simple version of the dashboard in about 16 hours. This dashboard would only contain the initial visual layer (full evaluation), and it would not support

continuously updated (live) visualizations, whenever the underlying data were modified. Additionally, the dashboard would not be reactive, so if the user interacted with the contained input widgets, no mutations would occur to the visual layer. The subject mentioned that they lacked the technical expertise that would be neccessary to implement this functionality. This 16-hour estimate does not contain any unit or functional tests either.

- Subject I2a, reported: "*I would probably be able to implement a simplified version of this dashboard in about 36-48 hours*". When investigated further, the subject mentioned that they did have some prior experience in web development, but they had not implemented any web applications in the past 3-4 years. One issue that was brought up by this subject, is that they did not know how to notify the client-side of the application when mutations occur in the underlying data, so instead they would simply request the data repeatedly (thus performing polling), and they would compare the current state of the dataset with the previous one, on the client, in an effort to identify changes. While this approach would in fact work, it would result in very expensive computations which would take place repeatedly (due to the comparisons between the pre-state with the post-state of the dataset). This would impact the user experience of the dashboard. This approach is essentially followed by existing web application frameworks and as we described in Chapter 2 it does not work well for information-dense, live dashboards. Lastly, another issue with this approach is that it requires the continuous transmission of somewhat big datasets between the client-side and the server-side (due to polling), which might not be very effective in cases when the bandwidth is limited.

Since none of these approaches actually compare in terms of functionality and efficiency with the dashboard that was implemented by FORWARD, we did not include them in the results table, but we did report them in this section for transparency purposes. Advanced developers also provided estimates of the amount of time necessary to build this dashboard without FORWARD. Specifically, Group A1 reported it would take 41-48 hours (more than a work-week), Group A2

reported 56-112 hours (1.5-3 work-weeks) and group A3, reported 46-106 hours (1-2.5 work weeks). Group A1 assumed use of Angular, while Groups A2 and A3 assumed use of React.

Recall that (as we described in Chapter 1) we requested from an advanced developer to implement a similar dashboard using Angular and that took 38 hours, so the time estimates provided by group A1 are not far off, especially considering that the advanced developer did not provide any functional or unit tests, while group A1 budgeted for a minimum set of tests that guarantee the sound operation of the dashboard. The reason for the wider range of the estimates provided by groups A2 and A3, is that both these groups assumed the use of React, and the subjects were not sure, if there exist react components that would be able to generate and incrementally maintain the formulated visualizations (map, charts etc.) that the dashboard contains. As a result, the lower end of the spectrum assumed the existence of such components, while the higher end of the spectrum assumes that these components would have to be developed by the subjects.

8.3.3 Observing developer processes & strategies

When trying to implement the evaluation dashboards all groups had to dynamically generate the unit states of all employed visual units using FORWARD's template language. Almost all novice and intermediate developers referred to previous examples of snippets illustrated in the tutorial to resolve this task. Specifically, most novice and intermediate groups, used statements from tutorial examples (such as, for-loops and print statements), which they then copied and adjusted accordingly to complete the task at hand. More specifically, since several visual units they used during this exercise, were different from the visual units they employed during the tutorial, they had to change the body of for-loops so that the correct unit state is generated dynamically. Additionally, the variables and the schema of their content used in their dashboard were different as well, so adjustments were necessary.

Given that the tutorial only lasted 40-55 minutes, during which subjects primarily focused on understand the mental model and the individual features of FORWARD (such as capturing

user input and triggering automatic updates to the visual layer when the underlying data changed), we thought it was normal that they did not retain the exact syntax of FORWARD's template language. Most novice developers that pick up a new programming language, quite often use appropriation (from online sources) and editing until they gain experience. Interestingly, most advanced developers opted for typing in their code, instead of using appropriation, even though at times (especially, when a bug was introduced), they still referred to previous examples to ensure that their code was bug-free. We believe this is because more advanced developers were able to draw analogies from the languages used by other frameworks they were familiar with (one subject reported: "*Oh this looks pretty similar to angular, even though there are a few differences on the template language*"-*A1b*), typing in their code felt more natural to them. Additionally, more advanced developers generally felt more confident in their abilities to type in code and at the same time they had a higher attention to detail, which was apparent from the fact that they introduced only minor typos and syntactic errors, which they were able to identify almost instantaneously.

We also noticed that there were no strict patterns in terms of what operation happens first and what second (in terms of data access, data processing and data visualization). Many subjects decided to first populate a generic visualization (using the drag-and-drop functionality of the IDE, that populates a visual unit with a sample unit state), and then perform the data access and the wiring of the two (using the template language), while others first performed data access and then generated the visualization. This shows that even though we did provide a high level description of the tasks that had to take place, these instructions were very high level and, as it turns out, were not followed to the letter, which proves that subjects actually understood the mental model and managed to connect the dots without following instructions blindly.

As we have described in this Thesis, in FORWARD (and most existing MVVM frameworks), developers first instantiate the Model and then use the template language to produce the ViewModel. One interesting observation was that a few groups consisting of novice and beginner developers did not comprehend why we perform this separation between the Model

and the ViewModel. Specifically, the subjects of group N2 (which comprises novice developers), tried to use parts of the unit state of a unit *a* to generate the unit state of the second unit *b*, instead of utilizing the variable that constructed the unit state of *a* in the first place. This was the primary reason why this group had the highest completion time amongst all groups. Once they realized that there exists this separation of concerns between Model (template variables) and ViewModel (derived data), they managed to resolve the bug and finish the dashboard. The subject reported: "*I think I did not take the time to understand what a unit state is conceptually. It was a lot to take in, in a limited amount of time, but now I understand things.*"- *N2a*. Interestingly, there is no reason why most MVVM frameworks (including FORWARD), enforce this separation, by not allowing the use of ViewModel variables to generate other parts of the ViewModel. Technically, the Model in FORWARD consists of template variables representing data views over base data (residing in remote data sources) and other template variables, and the ViewModel is just the final view over these template variables, but there is no inherent reason why using one part of the ViewModel to generate another should not be supported. Advanced developers, did not struggle with this concept at all, most likely because it is ingrained in them, due to their experience with existing MVVM frameworks. This is an interesting finding, that we will keep looking into, and perhaps will lead to a redesign in FORWARD's template language in future releases.

8.3.4 Reaction of subjects to FORWARD

Overall, the subjects liked using FORWARD. They acknowledged that it "*significantly simplifies the development of custom dashboards*" - *N3b* (also subjects: *A2b, A1a* and *N1a* made similar statements) and 100% of the subjects mentioned that they could see themselves using it and would recommend this framework to their peers. They thought that "*it is very intuitive to use, as it abstracts a lot of annoyances of web and dashboard development*"-*I2a* they, mentioned that the "*syntax is declarative but remains very similar to python, so it is easy to use!*"-*I1b*. They also noted that it offers "*easy integration with data sources, seamless data fetching and propagation of changes*"-*A1b* and that "*data handling is a big time saver*"-*A1b*.

Additionally, they commented on the ease of use and efficiency of the change propagation algorithm. Specifically, they mentioned that "*it's amazing how easy it was to make a dashboard live using this tool*"-*I2b* and that (due to the underlying incremental view maintenance algorithm) "*redisplaying the template to reflect changes was extremely smooth!*"-*N1a*. Additionally, during the pair programming session, one of the conversations that took place between the subjects was the following:

"*So how do we make it live?*"-*N2a*

"*Just put the onChange(redisplay()) next to the active function.*"-*N2b*

"*Oh nice! It's literally just that! Nice!*"-*N1a*

A few subjects also provided suggestions for future improvements. One such suggestion was to *increase the number of supported active functions and visual units*"-*N2b*, which is an item we are currently working on. Additionally, one subject noted that "*templates can easily get quite big which can make navigating them and editing them a chore*"-*A1b*. While looking into this issue, we decided that supporting the creation of sub-templates that can be imported into a main template is a good and simple solution to this problem. This solution would also introduce better separation of concerns, since some sub-templates could be used to only describe variable definitions (data access and data processing) and others could only describe the generation of the visual layer (generate visualizations and textual content).

Lastly, two subjects (*I1b* and *A3a*) asked us if we have taken any steps towards making sure that the resulting application is secure. We are still looking into this, however we believe that since all the information about data source connectivity exists in a config.json file, encrypting that file would make the resulting dashboard significantly more secure. Existing frameworks are typically less secure than that since developers simply initialize variables with the URLs, usernames and passwords that can be easily viewed by opening the source code in the browser or simply by printing variables in the browser console.

8.3.5 Reaction of subjects to the provided IDE

Subjects also provided feedback regarding the IDE they were using to implement the requested dashboard. They mentioned that the IDE "*should provide more advanced debugging features (such as breakpoints etc) and better syntax highlighting*"-*A1a* (also subject *N1a* made a similar statement). Additionally, error reporting could be improved (as the IDE, currently just prints the error message on the browser console). On the other hand, all subjects seemed to like the drag-and-drop functionality, with which they were able to drag sample visual unit instantiations into the editor. They thought that this was an "*intuitive way of using visual units without having to read verbose documentation pages*" - *N2a*. Many beginner and intermediate developers mentioned that they would like this feature to be extended to active functions for which some sample queries would be provided, so that they would not have to memorize the syntax of source specific languages (such as SQL, GraphQL etc).

Perhaps a spin-off of this IDE that contains a wizard-like menu allowing users to specify data access, data processing, and data visualization by filling out forms could be useful for this target audience. This software would essentially lower the required technical expertise even more, so that even users that have no experience in programming would be able to use its. This wizard could generate FORWARD code under the hood, which could also be shared amongst developers. This could make the collaboration of technical and non-technical users that want to build dashboards easier and as an extra feature we could provide a service that attempts to substitute the developer provided templates with other templates from the FORWARD developer community[45], that accomplish more performant incremental computation.

8.3.6 Conclusion

Recall that the main contribution of this chapter is to verify whether FORWARD actually facilitates the development of custom dashboards by a diverse audience. As can be seen from Table 8.2, all subjects were able to develop the dashboard, in a very reasonable time (11 to

41 minutes), regardless of their background and technical expertise. This is crucial because, FORWARD, not only reduced the time advanced developers need to build this dashboard from about one to three weeks to less than an hour, but it also enabled a new demographic of users to (a) implement a dashboard and (b) do it in a comparable time-frame to advanced web developers. This illustrates that FORWARD successfully bridges the gap between database developers, data analysts and web developers and allows all demographics to successfully implement fully customizable, reactive, live dashboards regardless of their technical expertise.

8.4 Acknowledgements

This chapter contains material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 9

ViDeTTe Interactive Notebooks

In this Chapter we demonstrate FORWARD's adaptability and generality by presenting ViDeTTe, a FORWARD-powered engine that enhances Python Jupyter Notebooks with capabilities that enable data analysts to build Notebooks that contain reactive visualizations. ViDeTTe enables data analysts to declaratively collect user-input captured by such visualizations, and use it to cause mutations to subsequent Notebook coding blocks. This feature enables the creation of parameterizable notebooks that can be used by non-technical users (now interactors), to explore the underlying data. With ViDeTTe we attempt to go one step closer to data analysts, and enable them to generate notebooks with functionality that is typically found in fully fledged web dashboards, while eliminating the technical expertise that is needed for the development of such dashboards.

Interactive notebooks allow the use of popular languages, such as python, for composing data analytics projects. The interface they provide, enables data scientists to import data, analyze them and compose the results into easily readable, report-like web pages, that can contain re-runnable code, visualizations and textual description of the entire process, all in one place. Scientists can then share such pages with other users in order to present their findings, collaborate and further explore the underlying data.

However, as we show in this work, interactive notebooks lack in interactivity for the reader of the resulting notebook. Users can rerun or extend the code included in a notebook

but cannot directly interact with the generated visualizations in order to trigger additional computation and further explore the underlying data. This means that only code-literate readers can further interact with and extend such notebooks, while the rest can only passively read the provided report. This comes in stark contrast to OLAP data cube interfaces, which utilize user interaction to trigger additional data exploratory capabilities. Adding OLAP-like reactive functionality in notebooks further increases the required technical expertise as event-driven logic has to be added by the data analyst.

To address these issues, we propose ViDeTTe, an engine that enhances notebooks with capabilities that benefit both data scientists and non-technical notebook readers. ViDeTTe uses a declarative language that simplifies data retrieval and data visualization for analysts. The generated visualizations are capable of collecting the reader's input and reacting to it. As the user interacts with the visualizations, ViDeTTe identifies subsequent parts of the notebook that depend on the user's input, causes reevaluation of the affected computations and propagates changes to the visualization units. By doing this, ViDeTTe offers enhanced data exploratory capabilities to readers, without requiring any coding skills, while at the same time lowering the technical expertise needed for the development of reactive notebooks.

9.1 Introduction

Jupyter notebooks [68] allow the use of popular, highly expressive, imperative languages, such as Python, for describing tasks such as data retrieval, processing and visualization, all in one platform. Due to the popularity of the languages they support, there is a massive collection of third-party utility libraries that can be used to facilitate such tasks. Furthermore, the web environment of interactive notebooks enables collaboration between data analysts, since it allows them to develop and run code that processes data and generates visualizations directly on the browser. Lastly, after completing an analysis, data analysts can compose their findings into an interactive, report-like page, that contains re-runnable code, visualizations and textual description

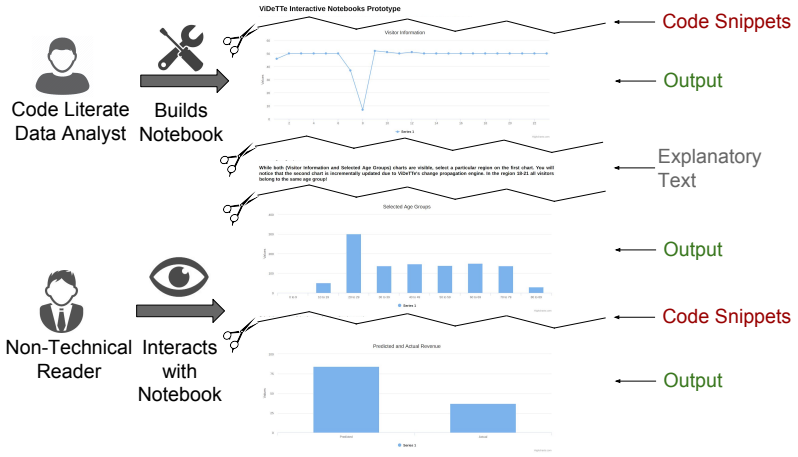


Figure 9.1. Notebook interface

of the analysis, which can also be shared with non-technical users. Figure 9.1, shows the interface of Jupyter notebooks. It comprises a sequence of blocks that are created "on-demand" by the data analyst. Each block can contain explanatory text or re-runnable coding snippets, that once evaluated, it outputs the result (which could be a visualization or plain text) into the subsequent block.

Jupyter notebooks, however, have limitations that impact the exploratory capabilities that can be performed by the (potentially non-technical) readers of the published notebooks. Particularly, while notebooks can contain visualizations that showcase important aspects of a data analysis, the readers cannot interact with them in the same way they could if these visualizations were part of a typical OLAP dashboard application. Specifically, even if the analyst that composes the notebook uses third-party web-based visualization libraries [29, 81, 66, 20] that support user interaction, the default side-effect of this interaction is to only cause local changes to the visualization the reader interacts with and, therefore, generally do not trigger additional data retrieval, re-computation of (any parts of) the notebook's analytics or mutations to other visualizations that are included in the notebook (which could be generated by subsequent coding blocks). Adding more elaborate side-effects that can perform such tasks requires complex event-driven logic that is typically used by web developers (in MVC applications) which entails a

more advanced skill-set that data analysts might lack. For this reason, only code-literate readers, capable of extending the notebook with more coding snippets, are able to further explore the underlying data (or examine additional hypotheses about that data), while the rest are limited to passively reading the generated notebook.

We are investigating two methods, that can be used for the creation of reactive notebooks. Both these methods use a view-over-data approach, without requiring additional event-driven logic from the analyst. The first method, presented in this paper, is using FORWARD [48, 51] and its DSL, with two differences: (a) Unlike FORWARD where the presentation layout is determined by markup (and CSS) [46], ViDeTTe employs FORWARD DSL scripts that are sequences of pairs consisting of computation (essentially, computing a view) and a visual unit that visualizes the view, (b) the second difference is the extension of FORWARD towards interfacing with Python code - in particular, with Python functions. The second approach is based on using existing Jupyter (Python) coding blocks and allowing them to also utilize the set of data collected by visual units and widgets during interaction. In order to avoid full-scale re-evaluation of the notebook, the developer may add data dependency hints between the coding blocks and the collected data. In this paper we will describe the first approach as the DSL saves the data analyst from having to include hints.

Contributions. In order to offer the capability to describe reactive behavior without requiring explicit event-driven logic in notebooks, we extend Jupyter with the ViDeTTe notebook engine. The main contributions of this extension is (a) a conceptually simple extension of the notebook from the usual, single evaluate-and-display into a continuous evaluate-and-display loop, driven by the interaction, while the analyst does not have to write any special event-driven code on how the notebook will be updated and (b) a declarative template language that describes the computation, data transformations, visualization and user input collection. These reactive capabilities, enable non-technical readers to further explore the underlying data used in an analysis.

Table 9.1. Schema description of database tables.

Page Views					
id	vid	url	time	date	revenue

Visitors					
vid	name	lastname	username	age	gender

9.2 Running Example

In order to illustrate the issues with existing notebooks and describe the extensions in FORWARD we will use the following example of a potential analysis:

Example 9.2.1 *Consider a data analyst, working for a news portal. The analyst wishes to convince the lead editor of the portal that by publishing more articles and advertisements that pertain to particular audiences during specific time-windows of the day, they can maximize the portal's revenue. In order to achieve this, the analyst intends to obtain and plot the number of readers that visit the website during the day; then for various time-windows she would like to obtain information about the readers' demographics (for instance, the visitors' age groups) and plot the result. Lastly, she would like to compute the portal's actual and predicted revenue (using a linear regression model) and plot the results, side-by-side, thus illustrating whether there is room for improving revenue.*

Note that, in this example, we can identify two types of users, the code-literate data scientist and the non-technical editor of the portal.

9.3 Data Analysis using Jupyter

We will now describe how a data scientist would perform this analysis with a traditional interactive notebook and illustrate how the limited interactive capabilities affect the level of data exploration that can be performed. We assume that the news portal maintains data about its reader-base in a Postgres database. Table 9.1 shows a potential schema, that could be used

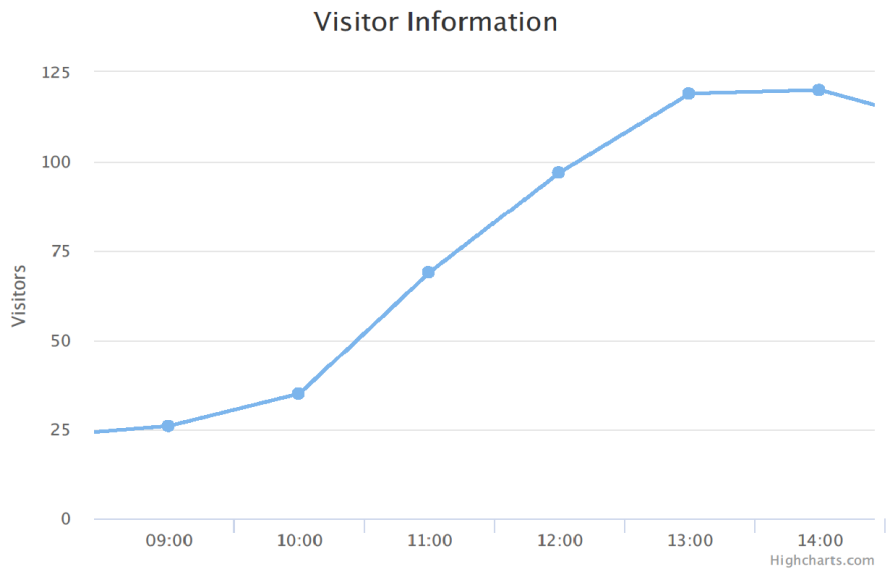


Figure 9.2. Line chart showing visitors per hour

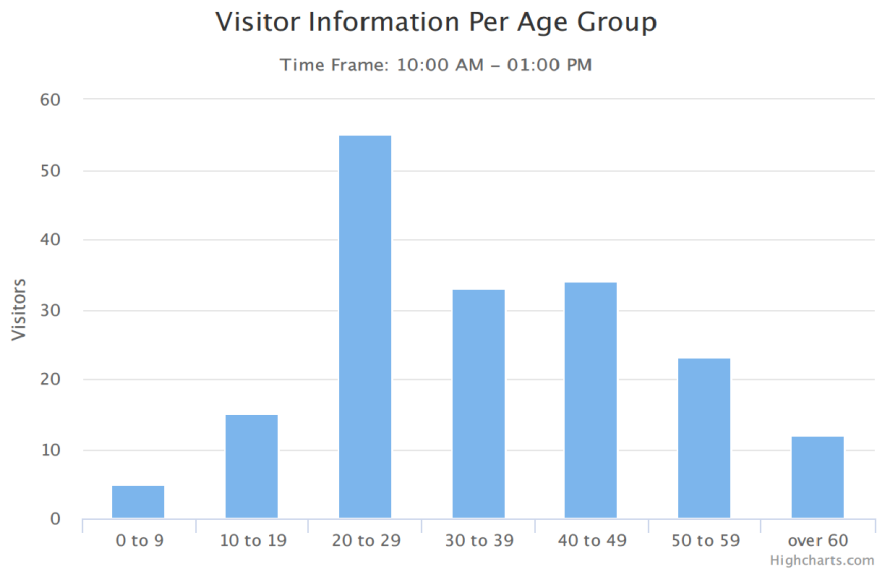


Figure 9.3. Bar chart showing age groups of visitors

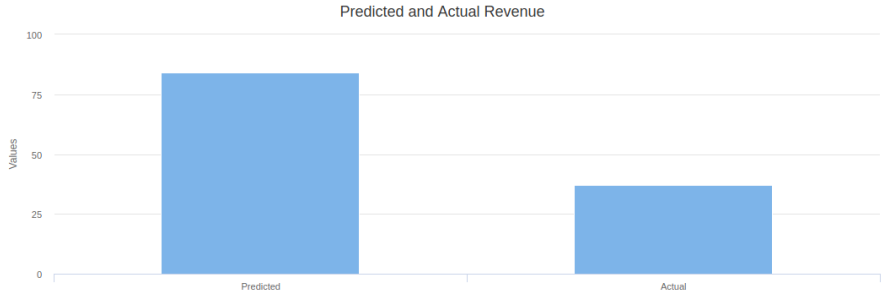


Figure 9.4. Bar chart showing predicted and actual revenue

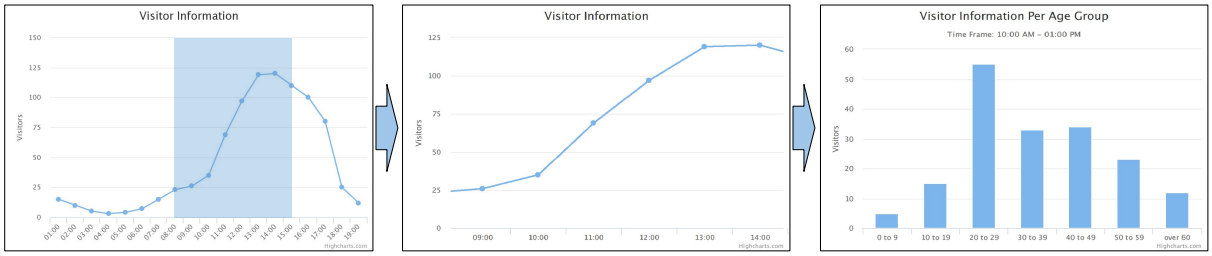


Figure 9.5. Demonstration of reactive charts. The reader's selection automatically updates both charts.

for storing visitor information. The database contains two tables, namely "Page Views" and "Visitors"; table "Visitors" contains information about each reader, such as the reader id, name, lastname, username, age and gender. The table "Page Views" maintains a tuple for each visit, and it consists of a visit id, the visitor id (foreign key referencing the visitor), the url of the visited page, the hour and date in which the user visited the website and the revenue collected during this visit. The reader information could have been retrieved, with their permission, from various social media services (such as Facebook, Google etc).

Data analysis steps. In order to construct this notebook, the data analyst must (a) retrieve website access information from the database, by joining the two tables on the visitor id (b) generate a plot that shows the number of users that visit the website during the day, (c) issue a query that counts the number of visitors per age group for a meaningful set of hours (time frame), (d) create a bar chart that shows the number of visitors per age group, (e) use an already trained predictive model in order to predict the expected revenue for the selected time window and (f) plot a bar chart, showing the actual and the predicted revenue generated in selected time window. Note that selecting a meaningful time frame (step c) in this analysis, is of utmost significance, because it will affect all the remaining steps of this analysis, and ultimately the insight, the portal editor will gain, about potential ways to increase the revenue.

Limited interactivity impacts exploratory capabilities. Unfortunately, there is no straightforward way for selecting a meaningful time frame. The analyst will have to go through a process of trial and error, by issuing an arbitrary number of such queries and plotting the results,

```
<% let readings = sql(
  SELECT count(time) as visits, time
  FROM (SELECT * FROM page_views pv
        join visitors v on pv.v_id = v.vid) AS joined_table
  GROUP BY time ORDER BY time ASC %>;)
```

Figure 9.6. Data retrieval required for first chart (shown in Figure 9.2)

```
<% unit highcharts %> {
   title: 'Visitor information' , type: 'line',
   xAxis : {
     labels : ['08:00','09:00'...], min : '08:00', max : '22:00'
   }
   series: [{ data: [ {y:15}, {y:10}...] }]
 } <% end unit %>
```

Figure 9.7. Evaluated unit state that generates the first chart (shown in Figure 9.2)

```
 readings = [ {visits: 15, time: '08:00'},...]
```

Figure 9.8. Query result needed for first chart

```
   <% init min_time = '01:00' %>
   <% init max_time = '24:00" %>
   <% unit highcharts %> {
     title: 'Visitor information',
     type: 'line',
     xAxis : {
       labels : [
         <% for reading in readings %>
           <% print reading.time %>
         <% end for %>],
       min : <% bind min_time %>,
       max : <% bind max_time %>
     }
     series: [{
       data: [ <% for reading in readings %>
           {
             y  : <% print reading.count %>
           }
           <% end for %> ]
     }] } <% end unit %>
```

Figure 9.9. Template generating unit state of first chart (shown in Figure 9.2)

until she finds a set that produces valuable insight. Furthermore, even if the analyst goes through this process for the data corresponding to a particular day, the time frame selection she made, might not produce any valuable insight for the dataset corresponding to a different day. Most importantly the non-technical readers cannot use the notebook to test similar hypotheses, they are limited to passively reading the ones that were hardcoded by the analyst. This aspect of introducing the human in the loop is improved dramatically with FORWARD notebooks.

9.4 Reactive Notebooks

Ideal reactive behavior and functionality. Co-dependent reactive charts are tremendously useful in this scenario. If the analyst was able to introduce a chart that allows the reader to select a particular range of hours, by directly interacting with it, and use that input to retrieve and plot only the user demographics for this particular range, she could enable the readers (or in this case interactors) to further explore the underlying data without requiring any code literacy from them. Figure 9.5 graphically depicts this process (note that the charts that appear side-by-side in Figure 9.5, actually appear after the respective coding blocks that generated them in the notebook). The first figure shows the reader of the notebook, selecting a particular time frame; this causes the line chart to zoom in, thus showing only the selected time frame. After the reader performs the selection on the first chart, the subsequent bar chart is also updated thus only showing the age groups of the users that visited the portal during the selected hours. This feature adds useful exploratory capabilities to the notebook, which is of great value to readers. It enables them to further analyze the underlying dataset used by the notebook by simply interacting with the provided visualizations. (For a live demonstration that illustrates how this analysis can be performed using FORWARD, visit the page: https://czarifis.github.io/vidette-prototype/)

This kind of reactive behavior cannot be easily expressed in traditional interactive notebooks. It is important to note that implementing a reactive behavior that can collect user input and use it to access additional data or mutate existing visualizations is extremely arduous

currently in interactive notebooks. It is identical to the process developers go through when implementing full blown web applications. It requires more time and effort as well as technical expertise (with third-party web components [29, 81] or input widgets [67]) that data analysts, might lack. More specifically, describing reactive behavior entails the implementation of actions (functions) that have to be invoked when particular mouse events take place on the pixels of the browser that correspond to the visualization. These actions describe the side-effects that will by applied when such events occur. The developer must install observers that listen for such events and then provide the imperative application logic that asynchronously accesses back-end databases to retrieve new data based on the user's selection, process them appropriately and cause mutations to the visualizations that depend on them.

9.5 ViDeTTe Notebooks

ViDeTTe notebooks support reactive behavior using a different approach. Instead of forcing analysts to provide logic that explicitly describes the side-effects (which typically include data access, processing and visualization steps) for each event that can take place, it allows them to treat the notebook as a view-over-data, following the tradition of many data-driven visualization and application frameworks. The advantage of this approach is that, when mutations occur at the underlying data (for instance, as a result of user interaction), the ViDeTTe engine can automatically and incrementally maintain the notebook, using incremental maintenance techniques, an early version of which was described in [49].

In order to achieve this, ViDeTTe provides constructs, called **visual and interaction units** (in short called just visual units), that create the visualizations given data and, importantly, they collect user input from said visualizations. As readers interact with visualizations, the respective visual units cause mutations to the underlying collected data. Given such mutations, ViDeTTe identifies what depends on the mutated data, causes the reevaluation of certain functions and automatically updates the state of the notebook (including the dependent computations and

visualizations that appear on it). The **declarative template language** has some built-in source access wrappers (eg, SQL access) and data analysis functions, which may be able to respond to input mutations incrementally. More generally, it can be used for the invocation of Python functions. When changes happen to the input of interfaced Python functions, the Python function is re-evaluated.

Source wrappers

Source wrappers allow the use of source specific languages for querying database systems (for instance SQL can be used to query MySQL and PostgreSQL databases, N1QL for Couchbase etc.). When using a ViDeTTe notebook, data analysts can directly issue queries from the notebook interface. This query is directed to the appropriate source wrapper that evaluates it, propagates it to the respective database system and assigns the result to a Python variable. This variable can later be used in subsequent parts of the notebook (as will be described later in this section). Figure 9.6 contains the first query of the analysis that retrieves the dataset required to plot the chart shown in Figure 9.2. Specifically, the query joins the two tables: *visitors* and *page_views* on the id of the visitor, then groups the result on the *time* attribute and runs a *count* aggregate to count the visitors per unit of time. Lastly, it sorts the resulting dataset by time in ascending order and assigns the result to the *readings* variable. Once the respective active function evaluates the query, the result is a Python array of dictionaries (used to maintain the tuples), as shown in Figure 9.8.

Visual Units

Visual units are constructs capable of generating fully reactive visualizations, that take advantage of the advanced interactive capabilities of modern browsers. In the eyes of an analyst, a visual unit is simply a black box that takes as input a value that describes the state of the visualization. This value is called *unit instance*. The visual unit internally uses the unit instance to invoke the appropriate renderer calls that will generate the expected visualization, thus absolving

the data analyst from having to perform these tasks manually. A particular instantiation of the unit can be described as **<% unit** U **%>** v **<% end unit %>**, where U the type of the visual unit and v the Python value corresponding to the unit instance. Figure 9.7 shows the unit instance of type `highcharts`, that once added to a coding block generates the visualization shown in Figure 9.2. The unit instance describes all the information that will be displayed in the visualization (such as the type and title of the chart, the labels on the x axis and so on). Each visual unit, comes with a unit instance schema that describes the format of the unit instance. As the reader interacts with visualizations, they can trigger mutations to the unit instance. For example, if the reader selects a particular area of the chart generated by the unit instance shown in Figure 9.7, the min and max attributes (in line 4) will be updated accordingly.

Templates

While source wrappers produce and visual units consume data in Python variables, data scientists typically perform additional computation between the former and the latter. The template language stitches these parts together by effectively making the functions output into variables that are essentially views over Python data. The template language supports a set of *template directives*, all of which operate on Python variables and simplify the task of creating nested data. Due to lack of space we do not include a formal definition of the grammar, instead we simply describe each of them in detail and provide a concrete example that illustrates their use.

Defining variables. A template may define variables that are added to the notebook's environment so that they can be used in subsequent statements. The **<% let** x **=** E **%>** directive defines variable (view) x and assigns to it the result of the expression E. E can denote four types of expressions: (a) path navigation on nested Python variables, (b) invocation of a Python function, (c) another subexpression containing directives that report data (defined in the next subsection) which generate nested values or (d) the invocation of a source wrapper. Note that the **let** directive, essentially, describes a view x over the expression E. This means that when

```
<% let visits = sql(
   SELECT * FROM page_views pv join visitors v
   on pv.v_id = v.vid where time BETWEEN
   <% print min_time %> and <% print max_time %>;)
<% let age_groups = sql(
   SELECT agegroup, count(*) AS total
   FROM (SELECT CASE
    WHEN age BETWEEN 0 AND 9 THEN '0 to 9'
    WHEN age BETWEEN 10 and 19 THEN '10 to 19'
     ...
    FROM  visits )
   GROUP BY agegroup
   ORDER BY agegroup ASC %>;)
```

(a) Data retrieval required for second and third chart

```
age_groups = [
   {age_group: '0 to 9', total: 12},
   {age_group: '10 to 19', total: 67},  ...]
```

(b) Query result needed for second chart (shown in Figure 9.3)

```
def predict_revenue(input):
  y_pred = LinearRegression.predict(input)
  return sum(y_pred)

def sum_revenue(input_list):
  return sum(item['revenue'] for item in input_list)
```

(c) Python functions predicting and summing actual revenue (used by third chart, shown in Figure 9.4)

```
<% unit highcharts %>
{
  title: 'Visitor information',
  type: 'bar',
  xAxis : {
    labels : [
      <% for v in age_groups %>
        <% print v.age_group %>
      <% end for %>]
  }
  series: [{
    data: [ <% for v in age_groups %>
        {
          y  : <% print v.total %>
        }
      <% end for %> ]
  }]
}
<% end unit %>
```

(d) Template generating unit state of second chart (shown in Figure 9.3)

```
<% unit highcharts %>
{title: 'Predicted and Actual revenue',
  ...
  data: [
    {
      y :
      <% print predict_revenue(visits) %>
    }
    {
      y : <% print sum_revenue(visits) %>
    }
  ]...}
```

(e) Template generating unit state of third chart (illustrating the invocation of python functions)

Figure 9.10. Declaration of "reactive" parameterized view, data visualization specifications and Python function declaration and invocation

mutations occur to variables that E depends on, ViDeTTe will automatically make x reflect the new state. In the simplest case, this happens by re-evaluation. Yet, some out-of-the-box ViDeTTe functions can exhibit incremental behavior. For instance, the template shown in Figure 9.6 employs a **`let`** directive to create a view-variable `readings` containing the visitor information that will be displayed in the chart (retrieved from a relational DBMS through an sql source wrapper). In some cases, however, the analyst may wish to simply initialize a variable during the first evaluation of the template. For instance, the lines 1-2 of the template shown in Figure 9.9, initialize, the variables *min_time* and *max_time*, which will be used in subsequent computation (as we will describe later in this section).

Reporting data. Value assignments and iterations over collections are specified by using the **`print`** and **`for`** directives. The **`<%print`** *E* **`%>`** directive evaluates the expression E and returns the result, while the **`<%for`** *x* **`in`** *E* **`%>`** *B* **`<%end for%>`** directive specifies that variable x iterates over the result of E and, in each iteration, it instantiates the body B of the **`for`** loop. Specifically, in Figure 9.9, in lines 8-10 and 15-19 a **`for`** directive is used to iterate over the readings retrieved from the database and for each reading, it generates a new value, with the use of the **`print`** directive. Particularly, in line 9 the template generates a string (the time label), which is added to the labels array (which contains the labels that will appear on the x axis of the chart). In lines 15-19 it generates a Python dictionary of the form {y: ...} and adds it to the data array. The data array in highcharts contains the points that will appear in the chart.

Invoking Python Functions. A template can also describe the invocation of Python functions, thus allowing any arbitrary computation. For instance, the snippet shown in Figure 9.10e (which generates the bar-chart illustrating the predicted and actual revenue, shown in Figure 9.4) in lines 5 and 6 invokes two functions: the *predict_revenue*() and *sum_revenue*() using the **`print`** directive, while providing as input an array of dictionaries that is initialized in Figure 9.10a; function *predict_revenue*() (shown in 9.10c), inputs the array into a trained linear regression model and returns the revenue prediction, while function *sum_revenue*() (in the same Figure), simply computes the actual revenue.

Collecting data. The template's **`bind`** directive allows the analyst to specify user input collection. Specifically, the **`<% bind x %>`** directive describes a two-way binding between the part of the unit instance appearing on the left side of the directive and the variable *x*. Once the reader interacts with the generated visualization and causes mutations to the underlying unit instance, these mutations will be propagated to the respective bound variables. For instance, in Figure 9.9 in lines 11-12 we create a two-way binding between the min and max boundaries of the chart and the min and max values (namely min_time and max_time) (defined in lines 1-2 of Figure 9.9). As the reader interacts with the generated chart she can select a particular time frame by dragging and dropping the mouse over a region. This action updates the min and max attributes of the unit instance which in turn updates the values *min_time* and *max_time*.

Reflecting changes to subsequent views

Once a set of variables is mutated by a visual unit as a result of reader interaction, ViDeTTe is responsible for reflecting these changes to all other parts of the notebook that depend on those variables. For instance, in Figure 9.10a we show a parameterized query that obtains all users that visited the website in the time-frame specified by *min_time* and *max_time*, categorizes them into age groups and lastly counts the users that correspond to each age group. The result of this query is assigned to the view-variable *age_groups* (Figure 9.10b shows the contents of this variable). The variable *age_groups* is then used in the template shown in Figure 9.10d (lines 7-9 and 12-16) to produce the bar chart appearing in Figure 9.3. In this scenario, if the reader's interaction with the chart shown in Figure 9.2, caused the mutation of variables *min_time* and *max_time* (note that the values *min_time* and *max_time* where collected by the visual unit shown in Figure 9.9, lines 9 and 10), then ViDeTTe must trigger a chain reaction that causes the reevaluation of both the query and the following visualization in order to reflect the changes.

9.6 Related Work

The FORWARD project. The specific template language used in VideTTe follows the syntax of templates in FORWARD [49, 52]. An engineering (but non-fundamental difference) is that ViDeTTe's data model is the same as Python's, while FORWARD's is JSON. Furthermore, FORWARD is capable of much more complex and dense visualizations, besides the ones shown in this paper. Specifically, it can nest visual units (thus generating nested visualizations), it provides if-then-else directives that can selectively bring visualizations and individual page-partials in-and-out of the page, it can use HTML, markup and CSS to produce arbitrary layout arrangements, and more. A question which we will investigate is whether such power makes sense to be used in the Jupyter setting, where the visualizations have smaller real estate and the analysts are comfortable with sequences of coding blocks and visualizations rather than more complex layout arrangements.

Prior work related to ViDeTTe notebooks can be classified into the following categories:

Grammar-based visualizations. The concept of using a formal grammar to specify visualizations can be traced back to Wilkinson's "The Grammar of Graphics" [98]. Since then multiple tools, such as Polaris [90] (later commercialized as Tableau), ggplot2 [96], Vega [87, 95] and others [54, 30, 39] have adopted this approach, thus enabling the description of visualizations (as well as the user interaction with such visualizations [99, 88]) in a terse and declarative manner. While the expressiveness of the employed grammars varies, their main focus is to effectively describe the specifics of the visualizations that will appear (for instance the dimensions of the chart, the colors, etc.), and not to describe data access or data processing and/or conversions. For that reason, such tools (due to their limited expressivity) do not provide constructs capable of retrieving data from diverse sources or performing arbitrary data transformations. Instead, the user has to provide imperative logic that performs these tasks manually and ultimately convert the dataset she wants to visualize into the format that is expected by the respective tool in order to generate the visualization.

Reactive Libraries. Some of the aforementioned tools (for instance ggplot2 [96], Vega [87, 95, 20, 66] and ggvis [54]), do allow readers to interact with visualizations, however, they don't allow, data analysts to declaratively specify the wide range of side effects that ViDeTTe allows. Specifically, while these tools can create visualizations (and widgets) that depend on one another, they cannot declaratively describe more elaborate side-effects that include retrieving additional data, or issuing arbitrary computations (for instance, linear regression as was shown in this paper). They take as input the entire dataset that can be included in the produced visualizations throughout their life-cycle and they provide a declarative language with limited data processing (essentially describing a subset of the operations that can be expressed in SQL) and no data access capabilities. Additionally, these tools operate as one-way doors with respect to the rest of the notebook, which means that they cannot be used to generate co-dependent visualizations that appear across multiple notebook coding blocks. For those reasons they offer more limited data exploratory capabilities compared to ViDeTTe.

Data exploration. Tools that use the human factor as an integral part of data exploration have gained popularity in the past years. Such tools provide a simple to use way that describes both computation and visualizations, thus providing the required toolkit to effectively explore data and steer computation. Some of these tools enable this process by providing a declarative language capable of performing such tasks (such as Zenvisage [89] and Devise [78]), while others (such as Vizdom [38], DataHub [72] and more[41, 25, 100, 75, 69]) rely on the user interaction with a front-end, in order to compose the appropriate queries. As a result, the former tools assume a type of user with deep understanding in databases and query languages, while the latter offer a more user-friendly way of performing an analysis but at the same time allow a predetermined set of possible analyses. ViDeTTe notebooks bridge the gap between these two types of users; to a data analyst it provides constructs that allow rapid creation of notebooks for an arbitrary number of analyses, while to a non-technical user it provides the capability to interact with the produced visualization in order to further explore the underlying data.

9.7 Conclusion

We presented ViDeTTe, an interactive notebook engine, that enhances interactive notebooks with capabilities that pertain to both data scientist that create notebooks and non-technical notebook readers. Specifically, ViDeTTe streamlines arduous tasks that usually have to be performed by data analysts, that compose such notebooks, while at the same time offering enhanced data exploratory capabilities to the readers of the published notebooks

Acknowledgements

This chapter contain material adapted from the publication "**ViDeTTe Interactive Notebooks**" appearing in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, Article No. 2, (HILDA) SIGMOD*. Houston, TX, USA - June 10, 2018. Konstantinos (Costas) Zarifis; Yannis Papakonstantinou. The dissertation author was the primary investigator and author of this paper.

Chapter 10

Related Work

In this Chapter we describe work that is more peripherally related to FORWARD and the concept of change propagation and generation of dashboards and/or individual visualizations. We described the most closely related work, namely MVVM frameworks, in Chapters 1 and 2 of this dissertation and at the end of the previous chapter we presented the related work for ViDeTTe. The work presented here has been conducted by researchers mainly in the fields of programming languages and algorithms. Specifically, prior work can be classified into the following categories:

Incremental view maintenance approaches. The concept of change propagation has been extensively studied in the database community in the form of incremental view maintenance (IVM). This has led to a substantial number of works both older [27, 26, 31, 83, 57] and more recent [71, 70] (see [56, 34] for surveys of IVM works). While inspired from IVM techniques, our change propagation algorithm extends existing IVM techniques in several important directions: it extends both the supported data model (adding support for semi-structured data and ordered collections) and the expressivity of the query language (adding support for user defined functions). Moreover, in contrast to IVM approaches which do not have any restrictions on the type of output diffs that they create, FORWARD's propagation algorithm has to respect the capabilities of the units (expressed in the form of supported unit instance diffs).

Dynamic algorithms. Change propagation has also been studied in the algorithms community in the form of dynamic algorithms. A dynamic algorithm efficiently updates the result of a computation, such as a clustering algorithm [40], a computational geometry algorithm [33] or an algorithm that generates shortest path trees [47]. Although dynamic algorithms have been successfully used in many domains, they are tailored to a particular computation and cannot be easily extended to other classes of computations, such as the generation and maintenance of the visual layer of a dashboard.

Incremental computation. Change propagation has also been studied in the programming language community under the title of incremental computation. In this community, researchers have developed techniques that achieve automatic incrementalization of programs. The main focus of these techniques [84] is the automatic translation of conventional programs into respective programs that can respond to dynamically changed data. Since these techniques provide tools or compiler techniques that automatically perform this translation, they manage to minimize the effort required for the implementation of such functions. Recent work on self adjusting computations also includes the use of specially designed high level languages (or the extension of existing languages with annotations [58, 32]) used to express incremental computations that when combined with specifically developed compilers, can generate executables capable of efficiently handling mutations of input data.

A common denominator of the majority of these techniques is the fact that the underlying languages utilize a strong type-system that enables the automatic distinction (or in many cases the explicit manual distinction), between mutable and immutable input data. Leaving aside the fact that applications developers cannot necessarily predict which input data will be modified during the lifetime of an application, the fact that JavaScript is an untyped language can also be an obstacle, in using such techniques. It is unclear if these techniques could be used without forcefully introducing a type-system that application developers would have to adapt to. While such a type-system might potentially assist in using some of these techniques to power incrementally maintained web applications, it would also steepen the learning curve of the respective

framework, since developers would have to familiarize themselves with the new type-system. Lastly, it is unclear how such techniques can be used in a distributed architecture like the one described in Section 6.

10.1 Acknowledgements

This chapter contains material from the technical report "**FORWARD; Facilitating the development of Analytical Dashboards on the Web**" by Konstantinos (Costas) Zarifis; Kian Win Ong; Yannis Katsis; Yannis Papakonstantinou; William G. Griswold. It is currently being prepared for submission for publication. The dissertation author was the primary investigator and author of this paper.

Chapter 11

Conclusion

We began this dissertation by illustrating that designing and developing continuously updated, reactive dashboards that contain information dense visualizations, such as maps and charts, is an extremely arduous and error-prone process, that requires advanced technical expertise and deep understanding of web technologies. Modern MVVM frameworks attempted to simplify this process, however, as we showed in Chapter 2, these frameworks still fall short. Specifically, such frameworks exhibit four important drawbacks: First, they still require significant effort to obtain and process data that reside in database systems or web services. Second, when mutations occur to underlying data, such mutations have to be observed and propagated to the framework manually (using imperative code), since MVVM frameworks do not automate this process. Third, while they do mutate the visualizations automatically, their incremental rendering algorithms incur significant performance penalties. Forth, although some of them are extensible, thus allowing the introduction of new template constructs that can be employed in templates, developing such constructs is still cumbersome.

In this dissertation we have presented FORWARD, a framework that realizes several research ideas that resolve these issues and make dashboard development accessible to developers with lower technical expertise, while maintaining attributes that are sought after in most frameworks, such as generality, extensibility, expressiveness and conciseness.

More specifically, FORWARD offers:

- *A Declarative template language that is expressive enough to enable developers to specify every task typically needed for dashboard development*. Specifically, the template language provides active functions that streamline the process of data access and processing, and visual units that declaratively generate visualizations. Developers can also declaratively specify that the generated dashboard is live, without having to provide complex imperative logic for every mutation that might occur in the underlying data or the model of the application. The user study, presented in Chapter 8, demonstrated that FORWARD's template language significantly lowers the technical expertise needed for the development of dashboards, so that even novice programmers that have no experience in web development can build fully-fledged and customized dashboards. At the same time, we showed that this language minimizes the amount of time and effort needed by more advanced web developers to build such dashboards.

- *An extensible architecture and developer-facing API that enables more experienced developers to introduce new visual units and active functions.* As we described in chapter 5, FORWARD provides an intuitive way for more experienced JavaScript developers to construct new active functions and wrap existing JavaScript visualization libraries into new reusable visual units. These visual units and active functions can then be utilized by dashboard developers, to declaratively specify the internal workings of the dashboard. The provided API contains a trigger language that can be used to specify how delta functions that describe how diffs (change-sets) targeting the input of visual units are translated into incremental renderer calls of the underlying visualization library API or diffs on the output and how diffs targeting the input of active functions are translated into diffs targeting the output of said active functions. Furthermore, to simplify the invocation of renderer functions, FORWARD, automatically converts the input data from its internal data model to JSON (POJSO), since this data model is used by all incremental renderers, thus absolving developers from having to perform this conversion themselves. Furthermore, an

automatically maintained internal caching data structure is provided for cases in which the developers of such units need to cache UI objects, thus simplifying the development of visual units even more. As a result, visualization libraries can be wrapped into visual units considerably easier than existing frameworks, such as Angular, as we have shown in the study presented in section 7.2. Additionally, as we showed in the same section, implementing unit and functional tests for such units is considerably easier than existing frameworks as well. Lastly, FORWARD provides a simulation algorithm, that enables the "as-you-go" specification of incremental renderers, thus exonerating developers from having to provide delta functions for every mutation that might occur.

- *An internal architecture that contains an IVM-inspired change propagation algorithm, that propagates mutations from the underlying data to the visual layer more efficiently than competing frameworks.* As we have described in chapter 6, FORWARD provides a novel diff-propagation based architecture capable of propagating changes from data sources all the way to the visual layer, in an efficient manner. This algorithm treats the template as a view description and the individual variables and unit states as a view-over-data. However, in contrast to existing work in incremental view maintenance, our propagation algorithm operates on the semi-structured JSON data model, instead of the relational data model used by most incremental view maintenance approaches, thus supporting a richer data model that contains nested datasets. Experimental evaluation (described in section 7.1) showed that the algorithm propagates changes to the visual layer is constant time, while the respective propagation algorithms of existing frameworks had an algorithmic time complexity proportional to the size of the ViewModel, which deems such frameworks unusable for larger, information-dense, live dashboards.

Lastly, we contributed a second artifact, called ViDeTTe that demonstrates the adaptability and generality of FORWARD in use cases that pertain to data science. ViDeTTe enhances Python Jupyter Notebooks with functionality borrowed from FORWARD, thus enabling the generation

of visualizations within Jupyter notebooks that are capable of collecting the reader's input and reacting to it. This allows the potentially, non-technical reader of such notebooks to interact with the existing visualizations and further explore the underlying data, even though the data analyst that generated the notebook did not explicitly define this behavior through imperative logic. Instead, ViDeTTe automatically identifies subsequent coding blocks of the notebook that depend on the user's input and causes their reevaluation. By seamlessly integrating ViDeTTe into Jupyter notebooks, data analysts can take advantage of the reactive behavior of charts, without having to learn a new programming model or programming language. Instead, with ViDeTTe they can use Python and the programming model of Jupyter notebooks, they are already familiar with, to add reactive data exploratory functionality to their notebooks.

Bibliography

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

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

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

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

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

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

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

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

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

[10] Knockoutjs. http://knockoutjs.com/index.html.

[11] 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.

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

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

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

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

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

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

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

[19] S. Aghaei, M. A. Nematbakhsh, and H. K. 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.

[20] 2018. https://altair-viz.github.io/.

[21] Angular leaflet directive. http://tombatossals.github.io/angular-leaflet-directive/.

[22] AngularJS. http://angularjs.org/.

[23] Barchart. https://www.barchart.com/login?ref=dashboard.

[24] Bella dati. http://www.belladati.com/marketplace/package/extension/pages/operational-dashboard-for-sensor-monitoring/.

[25] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang. Collaborative data analytics with datahub. *Proceedings of the VLDB Endowment*, 8(12):1916–1919, 2015.

[26] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Syst.*, 14(3):369–400, 1989.

[27] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM SIGMOD*, pages 61–71, 1986.

[28] Bloombert. https://www.bloomberg.com/markets/stocks.

[29] Bokeh Development Team. *Bokeh: Python library for interactive visualization*, 2014.

[30] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE transactions on visualization and computer graphics*, 15(6):1121–1128, 2009.

[31] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[32] Y. Chen, J. Dunfield, and U. A. Acar. Type-directed automatic incrementalization. *SIGPLAN Not.*, 47(6):299–310, June 2012.

[33] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[34] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[35] G. Cormode and B. Krishnamurthy. Key differences between web 1.0 and web 2.0. *First Monday*, 13(6), 2008.

[36] P. Crickard III. *Leaflet. js Essentials*. Packt Publishing Ltd, 2014.

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

[38] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: interactive analytics through pen and touch. *Proceedings of the VLDB Endowment*, 8(12):2024–2027, 2015.

[39] 2018. https://d3js.org/.

[40] A. Datta, H.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. In *Workshop on Algorithms and Data Structures*, pages 265–276. Springer, 1993.

[41] M. Derthick, J. Kolojejchick, and S. F. Roth. An interactive visual query environment for exploring data. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 189–198. ACM, 1997.

[42] J. S. Dumas, J. S. Dumas, and J. Redish. *A practical guide to usability testing*. Intellect books, 1999.

[43] R. Durão, W. Tie, K. Henneke, K. M. Balch, M. Hill, and R. Rayl. Visualizing the data visualization network: The dvmap project. *European Scientific Journal*, 2014.

[44] Ember. http://emberjs.com/.

[45] E. Fast and M. S. Bernstein. Meta: Enabling programming languages to learn from the crowd. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 259–270, New York, NY, USA, 2016. ACM.

[46] 2018. forward.ucsd.edu/visualizations.html.

[47] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[48] Y. Fu, K. Kowalczykowski, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Ajax-based report pages as incrementally rendered views. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 567–578. ACM, 2010.

[49] Y. Fu, K. Kowalczykowski, K. W. Ong, K. K. Zhao, and Y. Papakonstantinou. Ajax-based report pages as incrementally rendered views. In *SIGMOD Conference*, 2010.

[50] Y. Fu, K. W. Ong, and Y. Papakonstantinou. Declarative ajax web applications through sql++ on a unified application state. *arXiv preprint arXiv:1308.0656*, 2013.

[51] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos. The sql-based all-declarative forward web application development framework. In *CIDR*, pages 69–78, 2011.

[52] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos. The sql-based all-declarative forward web application development framework. In *CIDR*, pages 69–78, 2011.

[53] Y. Fu, K. W. Ong, Y. Papakonstantinou, and E. 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.

[54] 2018. http://ggvis.rstudio.com/.

[55] Google maps api, 2009. http://code.google.com/apis/maps/.

[56] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[57] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD*, pages 157–166, 1993.

[58] M. A. Hammer, U. A. Acar, and Y. Chen. Ceal: A c-based language for self-adjusting computation. *SIGPLAN Not.*, 44(6):25–37, June 2009.

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

[60] G. T. Heineman and W. T. Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, page 5, 2001.

[61] highcharts. http://www.highcharts.com/.

[62] highcharts-ng. https://github.com/pablojim/highcharts-ng/.

[63] W. L. HÃijrsch and C. V. Lopes. Separation of concerns. Technical report, 1995.

[64] Ibm iot monitoring solution. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=an&subtype=ca&appname=gpateam&supplier=897&letternum=ENUS218-186.

[65] Initial state. https://www.initialstate.com/.

[66] 2018. https://github.com/vega/ipyvega.

[67] 2018. https://ipywidgets.readthedocs.io/en/latest/.

[68] Jupyter, 2018. http://jupyter.org/.

[69] N. Kamat, E. Wu, and A. Nandi. Trendquery: A system for interactive exploration of trends. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 12:1–12:4, New York, NY, USA, 2016. ACM.

[70] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing ids to accelerate incremental view maintenance. In *ACM SIGMOD Conference*, pages 1985–2000, 2015.

[71] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[72] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, Aug. 2016.

[73] J. Kuan. *Learning Highcharts*. Packt Publishing Ltd, 2012.

[74] Leafletjs. http://leafletjs.com/.

[75] E. Liarou and S. Idreos. dbtouch in action database kernels for touch-based data exploration. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1262–1265, 2014.

[76] Z. Liu, Y. Saifullah, M. Greis, and S. Sreemanthula. Http compression techniques. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 4, pages 2495–2500. IEEE, 2005.

[77] Live traffic maps. https://www.google.com/maps/dir///@32.7503173,-117.1291518,15z/data=!4m5!4m4!1m1!4e2!1m0!3e0!5m1!1e1.

[78] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise: integrated querying and visual exploration of large datasets. In *ACM SIGMOD Record*, volume 26, pages 301–312. ACM, 1997.

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

[80] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive science*, 10(2):151–177, 1986.

[81] 2018. https://plot.ly.

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

[83] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.

[84] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[85] J. Resig et al. Jquery, 2006.

[86] Robinhood. https://robinhood.com/.

[87] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2017, 2015.

[88] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2016.

[89] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. G. Parameswaran. zenvisage: Effortless visual data exploration. *CoRR*, abs/1604.03583, 2016.

[90] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[91] G. Svennerberg. *Beginning Google Maps API 3*. Apress, 2010.

[92] Swift sensors. https://www.swiftsensors.com/products/cloud-dashboard/.

[93] The wall street journal. https://markets.wsj.com/.

[94] Things board. https://thingsboard.io/.

[95] 2018. https://vega.github.io/vega/.

[96] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009.

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

[98] L. Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[99] Y. Wu, J. M. Hellerstein, and E. Wu. A devil-ish approach to inconsistency in interactive visualizations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 15:1–15:6, New York, NY, USA, 2016. ACM.

[100] K. Zoumpatianos, S. Idreos, and T. Palpanas. Rinse: interactive data series exploration with ads+. *Proceedings of the VLDB Endowment*, 8(12):1912–1915, 2015.