Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

12-6-2017 10:30 AM

Rendering real-time dashboards using a GraphQL-based UI Architecture

Naresh Eeda
The University of Western Ontario

Supervisor
Madhavji, Nazim H.
The University of Western Ontario

The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science
© Naresh Eeda 2017

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Software Engineering Commons

Recommended Citation
Eeda, Naresh, "Rendering real-time dashboards using a GraphQL-based UI Architecture" (2017). *Electronic Thesis and Dissertation Repository*. 5136.
https://ir.lib.uwo.ca/etd/5136

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

Abstract

With the increase in the complexity of the systems being built and demand in the quality of service by the customers, developing and providing highly efficient real-time systems is one of the biggest challenges today for software enterprises. Bluemix™ – IBM's cloud offering implemented on Cloud Foundry, an open source "Platform as a Service" (PaaS), is an example of such a system. Currently, there are approx. 26 infrastructural services running in the background from where the data is fetched and is rendered on different dashboards of the system. However, the system suffers from performance issues.

This thesis focuses on the front-end and business-logic technologies with an aim to improve the performance of web applications. This thesis proposes and implements a GraphQL-based UI architecture which can be integrated into any of the new/existing web applications to boost the performance by fetching only the client-specific data faster i.e. the client specifies the server about the exact data to be fetched thereby avoiding over-fetching or under-fetching of the data.

We also integrate and implement the proposed architecture into the existing Bluemix system and explore the key performance improvements of the real-time dashboards. This architecture implements caching and asynchronous loading for serving the required data to display the dashboards in the Bluemix console fast and incrementally loading the data. This thesis paves the way for further work in the performance improvement of the web applications. The test results of this architecture's implementation on the Bluemix Usage Dashboard show that the Real data renders 245% faster and the Switching Account 153% faster than the existing system.

Keywords

GraphQL, Dashboards, Performance, Bluemix, Real-time, Web Sockets, Data Streaming, Caching, React, Redux.

Acknowledgments

First, I would like to express my profound gratitude to my advisor Dr. Nazim H Madhavji, for the continuous support of my master's program and research, for his patience, motivation, and immense knowledge. It was his supervision and guidance that helped me to decide my research path and walk through it. His planned regular meetings and presentations taught me to read papers and analyze them.

I am grateful to IBM (International Business Machines Corporation) for giving me the opportunity to do part of my research in their premises through an internship program. My sincere thanks to Jon Bennett, Senior Software Engineer – Bluemix UI, for being an excellent supervising mentor at IBM who not only guided me throughout the internship but also in my Master's research.

I thank my fellow lab mates, Nikita Sokolov, Darlan Arruda and Ibtehal Noorwali, for the stimulating discussions during our group meetings and presentations, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last few years. It was our lab environment that made the whole research thing smooth and thought provoking.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis and my life in general.

Table of Contents

Abstract i

Acknowledgments ii

Table of Contents iii

List of Figures vi

Chapter 1 1

1. Introduction 1

  1.1 Performance 1

  1.2 UI Development 2

  1.3 Research Question 2

  1.4 Solution Approach 4

  1.5 Key Research Results 4

  1.6 Novelty 5

  1.7 Structure of the thesis 5

Chapter 2 6

2. Background and Literature Review 6

  2.1 Related Work 6

  2.2 Background — IBM Bluemix 7

  2.3 Account Usage Dashboard 8

  2.4 Client-side Technologies and Communication Protocols 10

  2.5 Analysis of client-side technologies 10

    2.5.1 React 10

    2.5.2 Regular HTML and jQuery 11

    2.5.3 Vue 11

    2.5.4 Redux 12

2.5.5 Redis 13
2.6 Analysis of Communication Protocols 13
2.6.1 SOAP 14
2.6.2 Remote Procedure Call 15
2.6.3 Apache Thrift 15
2.6.4 gRPC 15
2.6.5 REST 16
2.6.6 GraphQL 16
2.7 Discussion 19
Chapter 3 21
3. Research Methodology 21
Chapter 4 23
4. Proposed Architecture 23
4.1 Technologies 24
4.2 New Usage Dashboard 30
Chapter 5 35
5. Validation and Results 35
5.1 Research Results 35
5.2 Validation 36
Chapter 6 39
6. Implementation 39
6.1 GraphQL Components 39
6.1.1 GraphQL Server 39
6.1.2 Connector 44
6.1.3 Model 46
6.1.4 Schema 46

6.1.5 Query Resolvers 47

6.1.6 Data Resolvers 49

Chapter 7 53

7. Conclusion and Future Work 53

References 54

Curriculum Vitae 60

List of Figures

Figure 1: Bluemix's Architecture 3

Figure 2: Account Usage Dashboard 10

Figure 3: MVC Pattern.. 12

Figure 4: Front-End Technologies (Pelletier, 2015) 20

Figure 5 GraphQL-based UI Architecture 23

Figure 6 Sample Redux state management code 25

Figure 7: React Performance 26

Figure 8: Data Fetching with REST.. 27

Figure 9: Data Fetching with GraphQL.. 28

Figure 10 GraphQL Subscription Server-Side Pub-Sub Configuration 29

Figure 11 Binding the Subscription Server with the GraphQL engine................................ 29

Figure 12 Binding GraphQL queries with React components................................ 30

Figure 13 Bluemix Architecture 31

Figure 14 Redux Workflow 32

Figure 15 Bluemix UI state change user actions 33

Figure 16 Redux and React Components.. 34

Figure 17 Loading times of Current Usage Dashboard vs New Usage Dashboard................ 37

Figure 18 Comparison of the previous and the present loading times................................ 37

Figure 19 GraphiQL Interface — An in-browser IDE for GraphQL................................ 43

Figure 20 Message bus data source format ... 45

Figure 21 GraphQL connector with a REST endpoint ... 46

Figure 22 GraphQL Schema ... 47

Figure 23 GraphQL Resolver ... 49

Chapter 1

1. Introduction

In this section, we introduce the two most common problems associated with enterprise applications development and the research needed to tackle these problems.

With increase in the complexity of the systems being built and the demand in the quality of service by the customers, developing and providing highly available and efficient systems is one of the biggest challenges today for software enterprises. IBM's Bluemix™ is an example of such a system, IBM's cloud offering implemented on Cloud Foundry (Bernstein, D., 2014) - an open source "Platform as a Service" (PaaS). IBM Canada is centrally involved in the evolution and improvement of Bluemix, for example, with respect to availability, efficiency, extensibility, and customizability of the software services running on Bluemix. Currently, there are approximately 26 infrastructural services running in the background.

1.1 Performance

Website characteristics/metrics indicate if a website is successful or not. These metrics are measured based on different parameters such as the website access speed, availability, security, and usability. Of these, performance metric plays a crucial role in attracting new customers and retaining existing users. It strongly influences the number of transactions being performed on the website. It is therefore necessary to identify and focus on the factors that influence the performance of a website (Monideepa, Tarafdar et al., 2008).

Some of the common performance related issues arise due to poorly written code, inefficient database design and SQL queries, poor load distribution, network strength, and dependency of web application on Virtual Machines. (360Logica Web Performance, 2017)

The impact of web performance on business success has been demonstrated many times in the real world. A couple of them are listed below (Wei, C et al., 2012):

- Amazon: 100 ms delay caused a 1% drop in revenue.
- Google: 400 ms delay caused a 0.59% decrease in search requests per user
- Yahoo!: 400 ms delay caused a 5-9% decrease in traffic.
- Bing: 2 seconds delay caused a 4.3% drop in revenue per user.
- Mozilla made their download page 2.2 seconds faster and was rewarded with an increase of 15.4% in downloads.
- Google Maps reduced the file volume by 30% and observed a 30% increase in map requests.
- Netflix enabled gzip on the server; simply by this single action pages became 13-25% faster and saved 50% of traffic volume!
- Shopzilla succeeded in reducing the loading time from 7 down to 2 seconds, whereby the conversion rate increased by 7-12%, they observed a 25% increase in page requests, they were able to retire 50% of their servers, thus saving energy costs.
- AOL observed the number of page views on several websites. While the fastest users requested 7-8 pages, the slowest only viewed 3-4.

1.2 UI Development

Developing User Interfaces (UI) for enterprise applications is complicated because of the large business data that the system should fetch, process, manipulate and render. With the recent advancements in the web development technologies, business logic is being coupled with the UI. The demand for a faster data manipulation has become a matter to concern to increase the User Experience (UX). Thus, integrating and coupling the data into the UI is always a focus in UI development.

Therefore, there is a need for a new technique for improving the performance of existing applications in retrieving the data faster from the backend data sources.

1.3 Research Question

Information from section 1.1 and section 1.2 leads to a very important research question:

"How do we design and develop high performance web applications?"

Thus, this thesis focuses on improving the performance of the existing applications by implementing a novel GraphQL-based UI architecture that enables efficient retrieval of data from backend data sources and creation of high-performance real-time dashboards. To evaluate the performance improvement made by this architecture, we implement the architecture by integrating it into the existing Bluemix architecture and explore the key performance improvements.

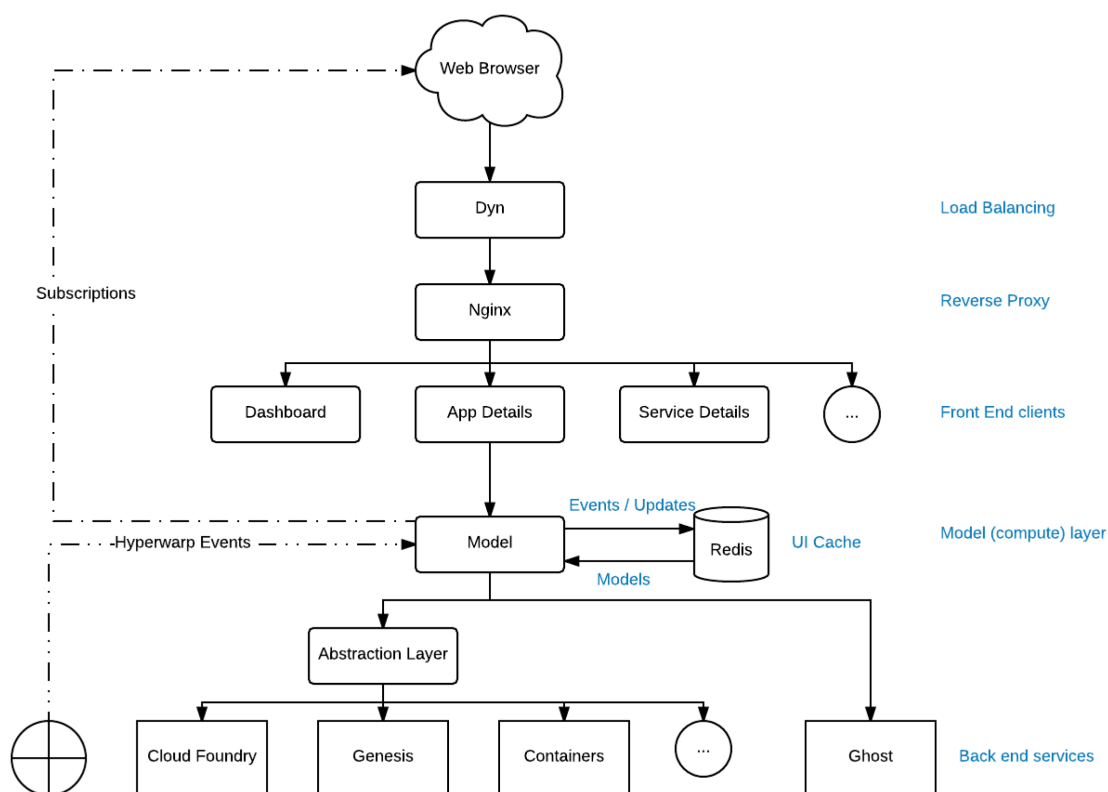


Figure 1: Bluemix's Architecture

Figure 1 represents the existing GhoST (Bluemix's message bus) architecture. Once a request is triggered from the browser, it is handled by the load balancer – Dyn. Dyn reroutes the request to different servers ensuring that all the servers are equally loaded with the number of incoming requests. The request is then sent to the reverse proxy – nginx. Nginx retrieves resources on behalf of the client from multiple servers. There are

26 services running on top of CloudFoundry in the backend. Examples of these services are: Genesis, Containers, and Billing. All these services push data onto a message bus called as the GhoST. Thus, GhoST acts an abstraction layer on top of all the data sources. Front end services like Dashboard, App Details, Service Details etc. consume data from this message bus and return the data to the front end. Each front-end service might consume data from one or more backend data sources that push data onto the message bus.

1.4 Solution Approach

Our research emphasizes on improving the performance of the web applications by (i) making the data retrieval faster from the backend data sources using multiple parallel asynchronous queries for a single request from the client and (ii) storing and caching the state of the User Interface (UI) elements. Initially, a wide range of modern front-end technologies and communication protocols, listed in section 2.3, have been studied and analyzed to integrate in the proposed architecture to boost the performance of web applications. A prototype was then implemented based on the shortlisted technology stack as a proof of concept for the architecture proposed. We finally implemented and integrated the proposed architecture into the existing IBM Bluemix console to compare the differences in performance before and after updating the architecture.

1.5 Key Research Results

There has been a significant boost in the performance of the Bluemix console after replacing the existing workflow with the proposed architecture. Our recorded loading times indicate that the usage information section loads, on an average, 245% faster and the account switching is done 153% faster than the existing system. The updated system has been tested with and without page caching, and hosting different number of applications in Bluemix under different user accounts as mentioned in Figure 18. All the tests show that this architecture contributed to a significant performance boost in the Bluemix system.

1.6 Novelty

The novelty of this architecture lies in splitting a single HTTP request from the client into multiple asynchronous calls and incremental loading of the UI elements thereby reducing the number of calls between the client and server systems, securely caching the state of the UI elements based on the user logged-in session thereby rendering the UI instantaneously instead of hitting the backend data sources multiple times, and rendering the skeleton UI elements even before the actual data is available.

1.7 Structure of the thesis

Chapter 1 of this document gives an outline of the thesis's motivation, goals and objectives, and our contributions. Chapter 2 presents a literature review and an analysis of the existing technologies for data retrieval, consumption and presentation. Chapter 3 gives the details of the research methodology followed while conducting this research. A detailed explanation of the proposed architecture is presented in Chapter 4. Validation of the proposed architecture on the existing IBM Bluemix system and the results of the research are described in Chapter 5. Implementation details of the architecture are described in Chapter 6. Chapter 7 summarizes the thesis research work and specifies the directions of the future work.

Chapter 2

2. Background and Literature Review

In this section, we present the background about the existing Bluemix system and the underlying architecture. We present an overview of the existing literature on boosting enterprise applications performance and focus on a specific component of Bluemix ─ Usage Dashboard, as our research platform.

In the last 10 years, there has been a great deal of modification and introduction of new technologies. Thereafter, web has been the area of intense focus and research of today because of its unimaginable future. The impact of open-source and the versatility of the web are accessible and outspread radically into every sector regardless of the hardware or software platforms used by engineers, developers and designers.

2.1 Related Work

There has been a significant amount of research done on improving the performance of web applications. The research is spread across multiple components such as efficiently storing/caching the data in Relational/NOSQL databases (Ghosh, R et al., 2006), improving the data transfer time (Myers et al., 2007, and improving the user experience by updating only a part of the web page without reloading the entire page (Paulson, 2005).

MapJAX — a data-centric framework accelerating the call back time of Ajax (set of web development technologies to make web applications more interactive and dynamic) applications and data transfer times was proposed by Myers to replace asynchronous Remote Procedure Calls. Vingraleka et al. (1999) developed a fast and reliable HTTP service — Web++ for improving the response times by dynamically replicating web resources over multiple servers.

There has been decent research on reducing file size as it contributes to boosting the performance of web applications. Tools such as YUI Compressor, Page Speed and JSMin have been developed to remove spaces and comments found in JavaScript files. King

(2008) proposed ten techniques for improving the display speed of the web pages. These include optimizing and resizing images, JavaScript files and minimizing the number of backend interactions.

Web caching is another area of focus for improving the performance. Ramaswamy et al. (2007) demonstrated that embedding cache can significantly boost the performance of the edge cache networks. In the proxy-based caching, the edge server (refers to client-side proxies, server-side reverse proxies, or caches within a content distribution network (CDN)) helps in increasing the scalability and reduces the client response latency. Luo et al. (2008) presented a form-based proxy-caching framework for database-backed web servers.

2.2 Background ─ IBM Bluemix

As a test platform, to validate the architecture and technology stack proposed in this thesis, we implement the architecture over the existing IBM's cloud platform – Bluemix. The current system suffers from performance issues. For example, the loading time of the user's application dashboard increases significantly with increase in the number of applications being hosted: e.g., 2.10 sec. for 10 applications to approximately 45 sec. for 64 applications. There are other such dashboards in the Bluemix console, e.g.: Resource Usage Dashboard and User Profile Dashboard. There is thus pressure on the organization to improve performance figures.

Besides this, there are monitoring concerns (e.g., False positive — misleading the user to believe the system is working properly, delayed — some systems need updates in real-time without a delay, and possibly missing alerts — could result in missing important information related to systems health) while the applications are running on Bluemix, shown on the system's dashboard. This system is complicated as there are multiple instances of each of the 26 services deployed globally e.g.: Sydney, London, Rome, and Dallas (and numerous other dedicated installations not shown). Examples of macro-services are: CloudFoundry – an opensource cloud platform; MCCP – a multi cloud controller proxy that provides multi-region support for Bluemix; and BSS – a business

support system for managing and administering processes that support SaaS services. (Abe, N et al., 2006)

Underlying the monitoring aspects are metrics that play a foundational role in defining data to be captured (Basili, V.R. et al., 2007) (e.g., CPU utilization, Mean Response Time, etc.) for analysis and alerting agents upon violation of conditions. Thus, on the Bluemix dashboard, there is an application called “Grafana”, which is a gateway for capturing specific metrics. Examples are: inbound and outbound call times and error rates; return code (e.g., 200 for normal, 400 for an error, 500 for a serious error); application crashed flag; up-time; CPU and memory utilisation; spikes in CPU or in memory; etc. One can click on any one of the services and visualize the underlying measures described above. Grafana thus helps in checking the health of the system. Beyond this, there are logs where staff dive into (as charts may not point to all the malfunctions) to manually check if something has gone wrong. This is a typical pattern during monitoring by the Bluemix staff.

While many services are monitored, the staff are concerned that: “All of this is art than science at the moment. When things go bad, the ‘peaks’ on the graphs shoot off and the support teams manually validate whether this is real or not. As there are multiple instances of each of these services, things get complicated. At the moment, we do not do very much with this data, we do not do alerts properly, do not analyze data properly, … and should be able to do much more on this monitoring space”*. There is thus a strong desire to investigate Bluemix with a view to bringing order to the lack of rigor in applications monitoring.

*This information is interpreted from discussion with the IBM Bluemix UI Console team, IBM Canada Lab, Toronto, Canada.

2.3 Account Usage Dashboard

Usage dashboard enables the account owner or a billing manager for an organization to view the real-time changes for the runtimes, services, support used per month and

containers in their organizations. They can either select a particular region or all regions to view the service consumption and runtime GigaByte-hours.

Navigating to the Account —> *account_name* —> Usage Dashboard will open the Usage Dashboard page. Billing managers can see the details for only the organizations in which they are billing managers.

At the end of each billing cycle, the account owner is charged based on the total usage of all resources incurred across all the organizations in their account. An account owner can filter the usage summary by region and organization. They can also click a month to see the usage for that month. One can select **All Organizations** from the **Organization** list to see the usage for all organizations in the account. Figure 2 represents the existing Account Usage Dashboard UI.

Implementation: The current system is developed using DOJO ─ an open source modular JavaScript library. DOJO is used in developing cross-platform JavaScript applications and websites quickly. The key features of DOJO help in providing APIs that will work across all the browsers by abstracting their differences. However, the existing Usage Dashboard has certain drawbacks as listed below:

- The existing UI waits until all the data is received and loaded into the components. If a user has multiple organizations, the UI will wait until all the data associated with those organizations is fetched.
- Page renders all at once after the data is received. Earlier DOJO versions had a reputation for being bulky and slow to load. (Zhang, X et al., 2011)

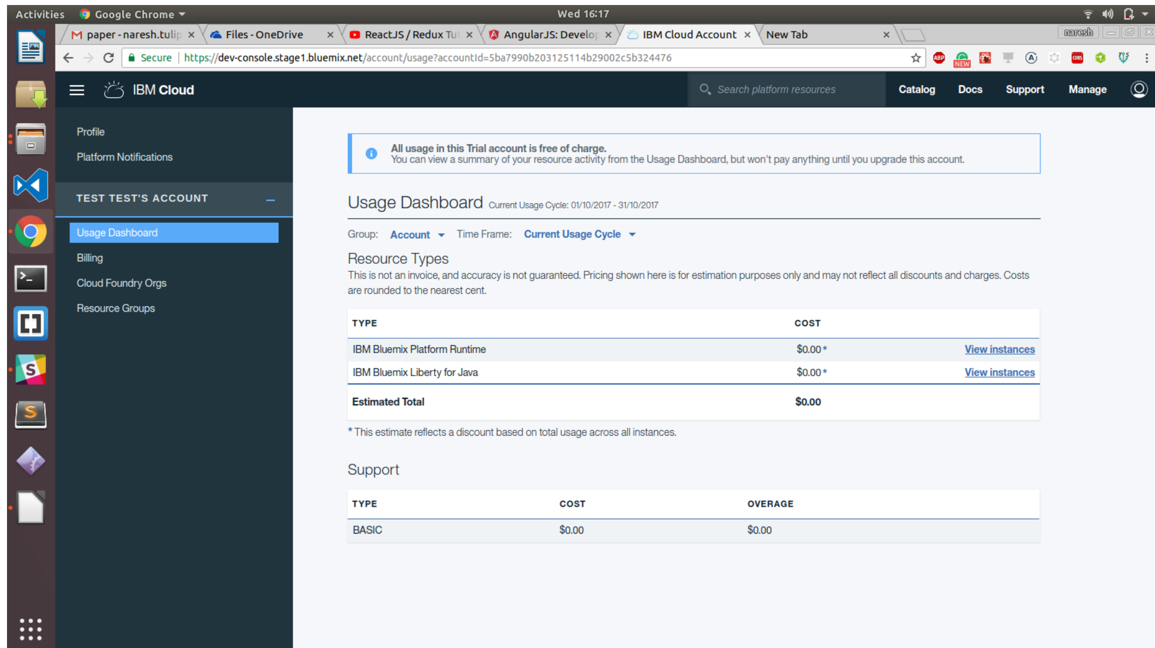


Figure 2: Account Usage Dashboard

Therefore, there is a need for a new technique for improving the performance of existing applications.

2.4 Client-side Technologies and Communication Protocols

One of the most common problems across many systems and applications is efficiently fetching the data from the backend data sources and rendering it on the User Interface. Although there has been intense research and progress (Chaudhuri, S et al., 2007; Van, Aken et al., 2017) on optimizing data storage and retrieval techniques for the past 15 years, very less emphasis was put on JavaScript frameworks. However, the emphasis is slightly moving towards optimizing the data processing and presentation layer recently. (Few, S et al., 2008).

2.5 Analysis of client-side technologies

2.5.1 React

Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces on computers. It divides a given application into three interconnected parts –

Model (central component managing the data, logic and rules of the application), View (representation of the information) and Controller (component accepting the inputs and converting them to commands for the model or view). This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development (Ping, Y et. al., 2009).

React (Gackenheimer, 2015) performs specific, well defined operations which represents view component in MVC architecture for both web and mobile applications. It permits developers to provide a detailed graphical representation of a view which is directly proportional to the model representing state. When there is any data manipulation in the model, React has a capability to update its relevant Presentational / UI elements.

A Document Object Model (DOM) is a cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document (Document Object Model, Wikipedia). React does this very effectively. When there is any change in the data /model, using JavaScript a virtual DOM is built. This newly generated DOM replaces with the current actual DOM in the browser which makes React to be robust. There are many alternatives to React, two of which are listed below.

2.5.2 Regular HTML and jQuery

A traditional approach of developing websites is by using JavaScript, jQuery, HTML and CSS (Dalmasso, I et al., 2012). However, this approach is rarely in use because of its limitations ─ the imperative nature of altering the DOM and the complexity to keep track of the current state of the website. Furthermore, this approach does not offer the latest DOM manipulation technique ─ componentization that helps in code reusability and modularity.

2.5.3 Vue

This is another recent front-end library and a great competitor to React (Passalia, A 2017). It is a progressive, incrementally-adoptable JavaScript framework for building the UI. It uses the traditional web development approach and enables the use of templates. Vue's compact size, intuitive API, component-based approach, and blazing fast core, make Vue.js a great solution to craft our next front-end application. However, Vue is relatively new in the market and the least matured of the other JavaScript frameworks and is being developed by only one person with help from the community (Koetsier, J 2016).

We chose for React because of the large scale of our web application: the component nature of React allows a very clear structure for large projects, is more easily testable and promotes code reuse. It also is more popular than Vue and has a very healthy ecosystem. Another reason for our choice is that our client (IBM) showed interest in expanding the Bluemix to also include a mobile application in the future. React, through React-Native, would easily allow much of the web-application code to be reused for an eventual

2.5.4 Redux

A problem that arises when using React is how to share data among different components. Traditionally, this would give rise to a MVC pattern. However, in practice it is found that the MVC pattern is not able to handle the high level of complexity introduced by larger web applications. Figure 3 illustrates this pattern:

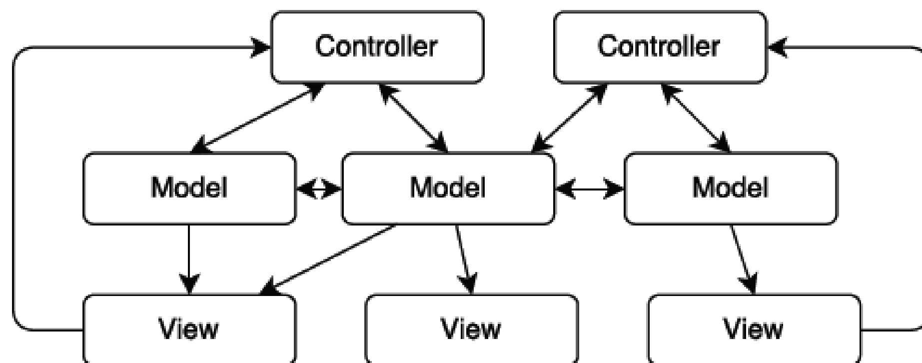


Figure 3: MVC Pattern

The problem is that in traditional web application MVC, each component has its own associated model. Furthermore, many views might depend on a certain model and models might have interdependencies. As indicated in Figure 3, as the web application scales complexity soon becomes very high as the interdependencies between controllers, models, and views become very hard to track. The Flux architecture (Gackenheimer, C 2015) is a solution to this proposed by Facebook in 2014, and it attempts to solve these issues by putting more constraints on the flow of information through a web application. Redux (Farhi, O 2017) is the most popular implementation of this architecture, and we decided to use this in our web applications in order to manage the complexity of shared data. An overview of the redux pattern can be seen in Figure 4.

The core feature of Flux architecture is unidirectional data flow. A view never directly changes state, it has to dispatch an action which indirectly triggers the change of state through the reducer. The reducer is a simple pure function which takes the current state and an action and returns the new state. The Redux state acts as a single source of truth for shared data. Use of Redux greatly improves the structure and predictability of data flow in our web applications (Ubbink, et al., 2017).

2.5.5 Redis

The cache provider (Redis (Xu, M et al., 2014) by default) is used to reduce the workload of the server when handling requests.

In particular, the memoisation is used to cache data that is commonly queried or is particularly computationally heavy to retrieve. An example is the previous months resource usage invoice of a Bluemix customer. Using the cache, we store the usage bill of the customer for the previous months in the currently logged in session as it will never change.

2.6 Analysis of Communication Protocols

There are several research papers on web services and the communication between different services (Slee, M et al., 2007; Cerami, E et al., 2002). In

a performance review of Representational State Transfer Protocol and Simple Object Access Protocol based web services by Tihomirovs, J et al., 2016, REST is shown to have better performance.

Maeda, K, 2012 researches the performance of serialization and concludes that binary serialization can be much more effective than text serialization formats such as XML or JSON. Popic, S et al., 2012; Sumaray, A et al., 2016 presents results of binary and text base serialization on mobile platforms and confirms that a binary serialization is faster on mobile platforms as well.

The microservice architecture has recently become a popular evolution of the SOA design pattern. The microservice architecture has been studied and reviewed in papers from different angles. Villamizar, M et al., 2015 presents a case study where the performance of different microservice architectures compared to monolithic applications with the same functionality.

Messina presents a review of design pattern closely related to the microservice architecture and gives a good overview of the design problems one must face when opting for a microservice architecture (Messina, et al., 2016).

The communication technologies presented in this section are chosen because they are widely used in the industry and they all have some degree of cross platform support.

2.6.1 SOAP

One of the cross-platform protocol for communication over Internet is Simple Object Access Protocol (SOAP). To encapsulate messages and to define how to transmit and receive them XML is used by SOAP. Though SOAP is not tied to any application layer for data transmission HTTP has become the most commonly used (Grabis, J et al., 2016). The SOAP architecture can be divided into three parts: Service providers, service registry and service consumer. The SOAP service which maps data from storage into SOAP messages will be created by service provider and it also provides a description of the available service to the service registry. The descriptions of services and how to

communicate with them will be exposed by service registry and then the service consumer can access the registry to find out which services are available and communicate with that service using SOAP messages (Belqasmi, F et al., 2012).

2.6.2 Remote Procedure Call

A procedure call from one process to another over a network is called remote procedure call (RPC). There are frameworks built specifically to allow for cross platform RPCs though most modern programming languages has the support for RPC built in. To make cross platform calls possible data resources, procedures, their input parameters and expected outputs in a language independent Interface Definition Language (IDL) will be described by these frameworks. The server and client code which implements the procedures in different programming languages, on different machines can then be generated by procedures in the IDL. Remote calls are made by a client who calls the locally implemented procedure, the message is then packed together with the parameters and sent to the server which has implemented the same procedure. The procedure will be executed by server and sends a response with the result (Bershad et al., 1990).

2.6.3 Apache Thrift

One of the framework for cross language application communication is Thrift. In addition to above, Thrift also functions as a communication protocol for remote procedure calls between web services. Thrift is not completely platform independent as it is a framework. It has support for 14 different languages, including Java, C#, C++, JavaScript, Python and PHP. The JSON and binary are serialization alternatives which are present across all languages in the Thrift framework. Thrift has support for both HTTP and raw TCP as transportation layers. (Apache Thrift Documentation, 2017)

2.6.4 gRPC

Just like Thrift, gRPC is also a framework for cross platform remote procedure calls. Protocol Buffers are used both as a serialization format for messages and also as an interface definition language that makes the cross platforms calls feasible in gRPC. The

gRPC framework is available in 10 different languages, including Java, C#, C++, JavaScript and Python. gRPC has support for the newly released HTTP 2 protocol, but can also use TCP as its transportation layer. (Wang, X et al., 1993)

2.6.5 REST

Representational state transfer (REST) or RESTful web services is an architectural style for communication between computer systems on the web. Unlike SOAP, REST web services are not tied to any specific transportation protocol or serialization method. (Christensen, J et al., 2009) The transportation of data is done over HTTP and the data is serialized to either JSON, HTML or XML which has become the most widely used alternatives.

An individual uniform resource identifier (URI) is used to access the data resources in REST. These resources can be identified with a unique URL when REST is implemented with HTTP as a means for transportation. HTTP verbs such as GET, POST, PUT and DELETE are used to manage the Resources. All the operations pre-defined by the HTTP methods — GET, PUT, POST & DELETE are available in the REST architecture for using and managing the resources. A GET request that uniquely identifies a specific resource is sent to the REST architecture to fetch data. A POST message is sent to the REST architecture to add/update information on the data resource. As REST is lightweight and can be adopted in any language, it has become one of the most popular alternatives to communication between web services.

2.6.6 GraphQL

When Facebook built its mobile applications, they needed a data-fetching API powerful enough to describe all of Facebook, yet simple enough to be easy to learn and use by their product developers. They then developed GraphQL in 2012 to fulfill this need. Currently, it powers hundreds of billions of API calls a day (Byron, L 2015).

GraphQL is a query language for APIs, and a server-side runtime for executing queries by utilizing a framework we characterize for our information. It is developed by

Facebook in 2012 before being publicly released in 2015. It is a generic query language which implies that it is not bound or tied to a particular storage engine or a specific database. Instead, it is backed by our existing code.

Why GraphQL?

Back in 2012, Facebook began an effort to rebuild their native mobile applications. Their android and iOS applications were thin wrappers around views of their mobile website. However, as the Facebook's mobile applications became more complex, they suffered poor performance and frequently crashed. They evaluated all their options for delivering News Feed data to their mobile applications, including RESTful server resources and FQL tables (Facebook's SQL-like API) but there were large differences in the data they needed to render in the mobile applications and the server queries that were required. They needed a new way to think of data not in terms of resource URLs, secondary keys, or join tables; but in terms of a graph of objects and the models that ultimately were used in their applications like JSON. This inspired them to develop a new project that ultimately became GraphQL.

Sample GraphQL Query

```
{
    items{
        service_name
        cname
        creation_date
        crn{
            doc_type
            service_instance{
                domains_available
                ssh
            }
        }
    }
}
```

Sample GraphQL Query

Key Features of GraphQL

- No over fetching or under fetching of data: XML RPC, SOAP, REST and gRPC always either over fetch or under fetch the data needed by the client but GraphQL only fetches the data specified by the client. It fetches only the data needed, nothing more nothing less.
- One endpoint: Unlike REST, where the client needs to consume resources from different endpoints, GraphQL has only one endpoint - /graphql. This eliminates the need of remembering different endpoints to consume different data sources.

Example:

For a REST web service, there could be multiple endpoints as below for different GET and POST operations:

- GET /books/:id
  GET /authors/:id
  GET /books/:id/comments
  POST /books/:id/comments

But, in GraphQL, we only need one endpoint, "/graphql" for all types of requests

- Multiple resources in one request: GraphQL supports merging queries for different data sources into one query thereby allowing the client to fetch data from multiple resources in just one request thereby eliminating the number of HTTP calls.
- GraphiQL: GraphQL provides powerful developer tools like GraphiQL- an in-browser integrated development environment for faster application development and debugging.
- Evolve APIs without versioning: Since GraphQL has only one endpoint, this eliminates the necessity of informing the clients about any changes in the API versions as the endpoint will still remain the same.
  Example:

In REST web service, whenever a new version evolves, the client has to update his code to consume resources from the modified URI.

GET /books/v1/:id

GET /books/v2/:id

But in GraphQL, it will always be

/graphql

- Self-Documenting: GraphQL automatically generates documentation for the APIs based on the comments provided during development. This establishes a consistent documentation pattern across all projects.

2.7 Discussion

Developing web applications with the use of HTML (Hyper Text Markup Language), CSS (Cascading Style Sheet) and core JavaScript (Scripting Language) in the presentation layer is rarely in use and is considered traditional. The demand for dynamic and more interactive web grew exponentially with the advancements in the web. As a result, the data-access layer of web was a focus point in web application development. Now, the design and development trend has been changing every year, and emphasis has been given to performance, security, cost and efficiency to achieve a better user experience. There has been a rise of different JavaScript frameworks and libraries to maintain these aspects. However, a rapid change in the development platform has brought a paradox in selecting the right technologies for developers.

On the other hand, there are technologies that support methods such as: (i) polling, actively sampling the status of an external device by a client program as a synchronous activity (Langendoen, K et al., 1997); (ii) interrupts, a signal to the processor emitted by hardware or software indicating an event that needs immediate attention (Langendoen, K et al., 1997); and (iii) subscriptions for real-time monitoring (Cherukuri, R 2010). But none of the existing technologies, except GraphQL (Apollo Developers Blog, 2016), has features which enable accurate and efficient streaming of data useful for visualizing real-time characteristics of a system. Examples features of GraphQL are: Live (notify the server about the desire to receive live updates about a certain field) (Apollo Developers

Blog, 2016); and deferred querying (specifying that some part of the query can arrive later) (Apollo Developers Blog, 2016).

Figure 4: Front-End Technologies (Pelletier, 2015)

Figure 4 represents a subset of the list of front-end technologies, Package managers, JavaScript frameworks, Build Tools, Testing tools, Version control tools, UI frameworks, and a variety of other tools currently available and used in the web application development. This figure is to illustrate the rise in the number of frameworks and technologies associated with the front-end development in the last couple of years.

Chapter 3

3. Research Methodology

We conducted this research in three increasingly sophisticated stages:

In the preliminary stage, we performed a literature review of the existing works on rendering real-time dashboards, various front-end technologies, communication protocols and web services available in the market. We then shortlisted a few technologies, based on the literature results, to develop initial prototypes. We first developed a Proof-Of-Concept (POC) to check the integration of these technologies into the existing products. For the POC, we initially implemented an abstract REST endpoint on the Bluemix development server's message bus to capture the data being generated. We then designed a GraphQL Schema based on the data format being generated by the REST endpoint as a response to the webservice consumption calls. We then implemented a GraphiQL layer to test the correctness of the results being fetched by the GraphiQL queries. The result of this phase gave us an insight into the technology stack and algorithms to be used for the actual system implementation.

In the intermediate stage, we evolved the preliminary prototype into a transitional prototype. We updated the preliminary POC's GraphQL layer to consume the real-time data being pushed on to the message bus. We then implemented a React layer in the front-end and GraphQL subscriptions using the in-built Publish-Subscribe system. We rendered the raw data on the UI, being received by the GraphQL layer, using the created react components. We automated the process of triggering GraphQL mutations to publish events on the subscription channels to validate the working on WebSockets and the data being pushed onto the UI. We then integrated Redux into the front-end layer to store the states of the react components and a Redis cache to store the frequently queries parameters.

In the advanced stage, we integrated the Bluemix IAM Token authentication and created a separate domain hosting the updated GraphQL-based Usage Dashboard. We analyzed

thee the performance based on the mirrored data and conditions from the production environment. We updated the dashboard rendering technique with the caching capability based on the logs from production. We explicitly induced multiple applications to the dashboard given by the Bluemix staff as opposed to our toy examples. This helped us validate the performance of our rendering. We also explored for new metrics that might help capture the additional parameters for faster rendering of the UI. We are continuously monitoring and validating the architecture based on the inputs from the stakeholders. In future, for the metrics work, we intend to create and use an instrument and the GQM (Basili, V.R. et al., 1994) method to gather data from the stakeholders and customers, identifying Goals, related Questions, and Metrics. We shall then implement and integrate the metrics with the existing metrics capturing gateway — “Grafana”.

This, three-stage incremental paradigm has built-in risk management because the later research results are built upon earlier, foundational results through iterative prototyping.

Chapter 4

4. Proposed Architecture

In this section, we present a model to boost an application's performance and the motivation of selecting a specific set of technologies to implement it.

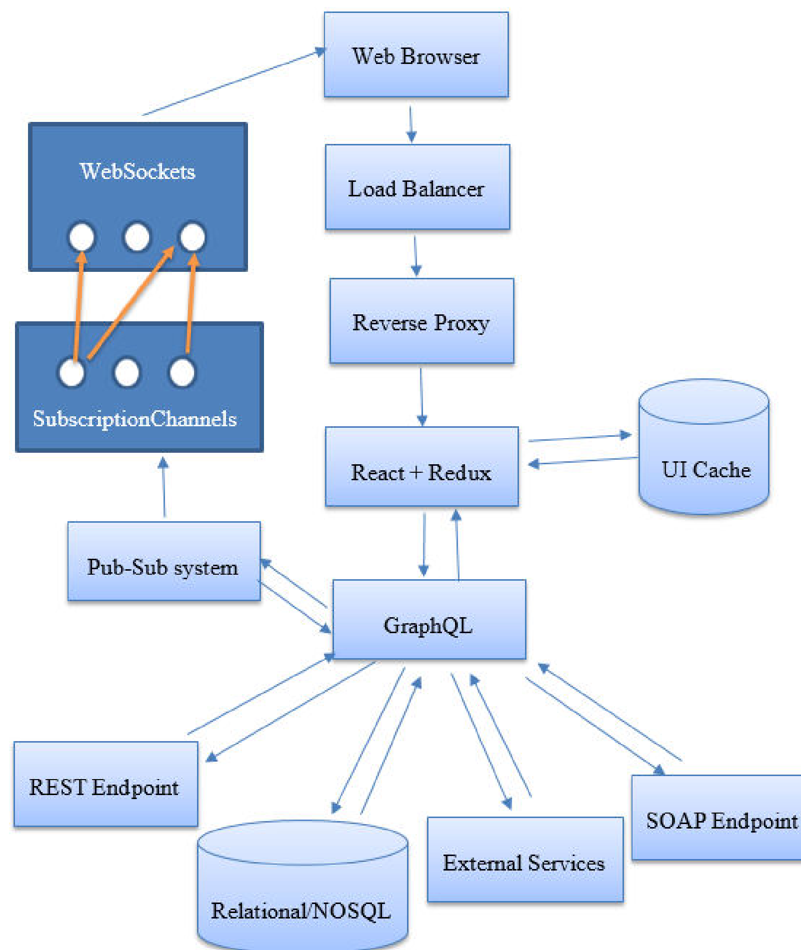


Figure 5 GraphQL-based UI Architecture

Figure 5 represents our proposed GraphQL-based UI Architecture. This architecture starts with creating a GraphQL-layer on top of all the existing backend data sources of a system. These data sources could be a relational database, NOSQL database, REST

endpoint, SOAP endpoint, message bus or a combination of any or all of these data sources. We then integrate a store management technology in the front end to store the states of the application elements, a DOM rendering mechanism for rendering the UI components, a UI caching technology to cache the web pages, and a socket-based subscription channel between the client and server systems. We therefore implement this architecture using the technologies listed in section 4.1 because of their efficiency compared to other technologies in the section 2. Integrating GraphQL into the architecture brings in all the advantages listed in section – key features of GraphQL. Also, we will now be able to split a single request triggered by the client into multiple requests after hitting the GraphQL layer as proposed in the architecture. We design and store the states of all the components that are to be rendered on the UI using Redux and bind its values with the smart React components. As opposed to the existing method of rendering all the data at once, we incrementally load the data onto the UI without having to wait for the other queries to complete their execution. We bind components of the UI to specific queries and render them as soon as a response is received. This allows us to asynchronously load the data onto the UI. Also, having the initial component skeleton stored in the redux helps us in rendering the dummy components i.e. the UI elements without the actual data to be rendered as soon as the request i received. This improves the user experience by displaying the web page almost instantaneously.

4.1 Technologies

React + Redux: Based on our analysis of different technologies listed in Section 2.2 and 2.3, React + Redux and GraphQL proved to be the most efficient technology stack for implementing the new architecture. React is not just the UI, it is a framework. Redux can be integrated into React thereby enabling the state management. (SourceToad, 2015) Since React creates its own virtual DOM, it gives enormous flexibility and amazing gain in the performance. Figure 7 represents the performance of React compared to the other JavaScript libraries currently available. (Auth Blog, 2016)

```
import { applyMiddleware, createStore } from "redux";
import axios from "axios";
import logger from "redux-logger";
import thunk from "redux-thunk";
import promise from "redux-promise-middleware";

const initialState = {
  fetching: false,
  fetched: false,
  users: [],
  error: null,
};

const reducer = (state=initialState, action) => {
  switch (action.type) {
    case "FETCH_USERS_PENDING": {
      return {...state, fetching: true}
      break;
    }
    case "FETCH_USERS_REJECTED": {
      return {...state, fetching: false, error: action.payload}
      break;
    }
    case "FETCH_USERS_FULFILLED": {
      return {
        ...state,
        fetching: false,
        fetched: true,
        users: action.payload,
      }
      break;
    }
  }
  return state
}

const middleware = applyMiddleware(promise(), thunk, logger())
const store = createStore(reducer, middleware)

store.dispatch({
  type: "FETCH_USERS",
  payload: axios.get("http://com.naresh/apis/demo/users")
})
```

Figure 6 Sample Redux state management code

Figure 6 represents a sample code for the functioning of the Redux store management. The initial state of the components is stored in an object– initialState with default values. The reducer function listens to the actions triggered by the smart components on the UI and performs the associated actions and updates the state of the values in the store. Store.dispatch function helps dispatch an event/action to the associated reducers.

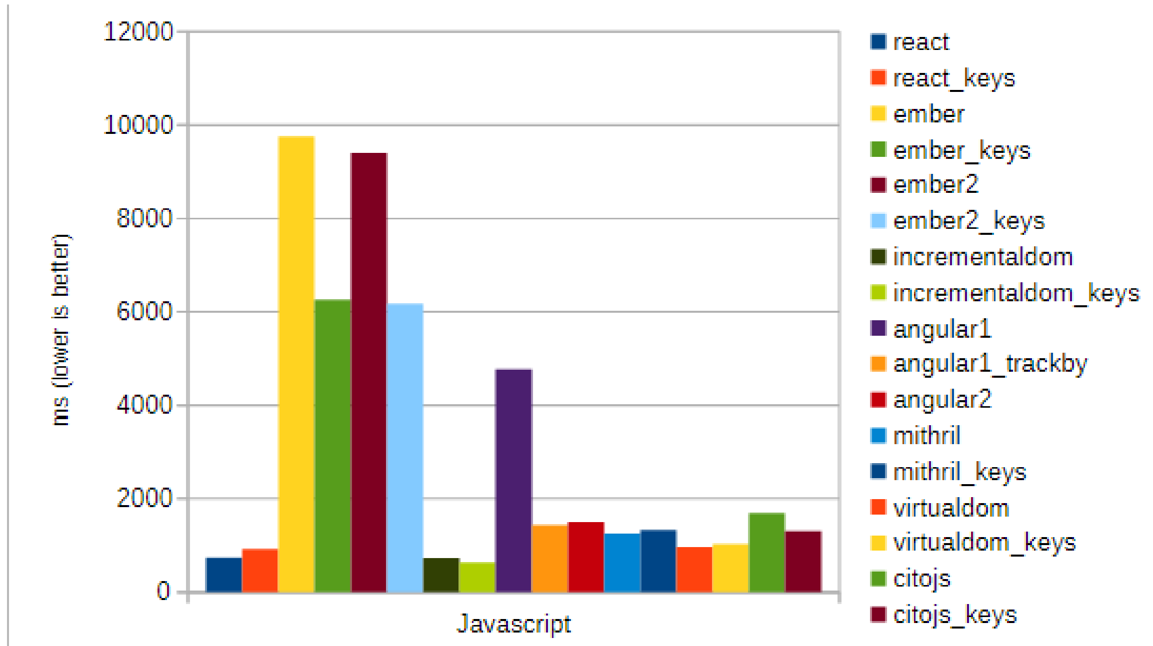


Figure 7: React Performance

Figure 7 illustrates the significant performance provided by the use of React technology compared to the other front-end technologies.

GraphQL: GraphQL, on the other hand, can help in efficiently fetching the data. Figure 7 and Figure 8 represent a scenario on how data fetching is dealt using REST APIs and GraphQL. With REST APIs, we typically gather the data using multiple calls to different endpoints. In Figure 7, the first call fetches the initial user data, the second call fetches the posts for a user and third call retrieves a list of followers per user. However, there is no explicit control to the user on restricting the parameters to be fetched. On the other hand, in Figure 8, GraphQL allows the user to send a single query to the GraphQL server specifying the exact parameters to be fetched. The server then responds with a single JSON object containing the entire information queried.

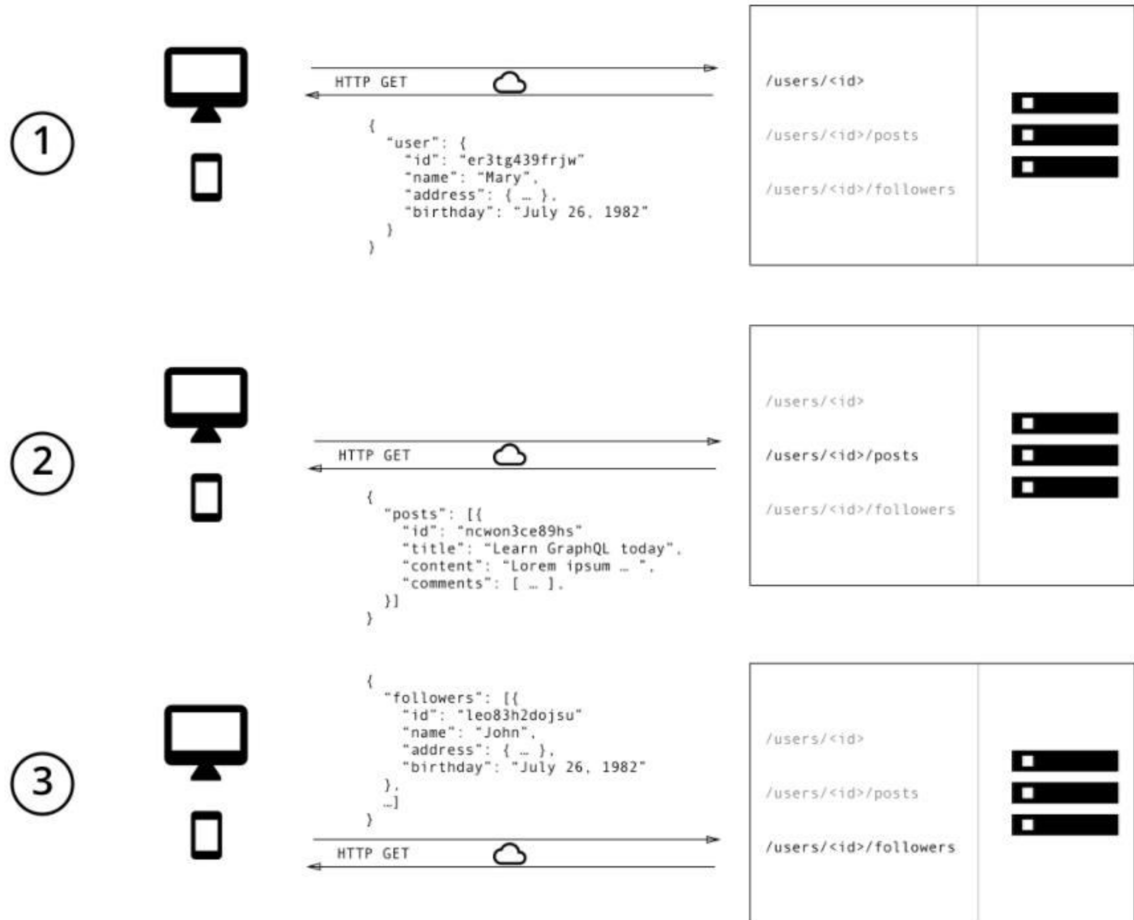


Figure 8: Data Fetching with REST

Figure 8 represents the number of calls made using REST web service to fetch the user related data that include user information, user posts and their followers. It had to make three calls to three different endpoints to get the complete user related information. However, Figure 9 illustrates the advantage of implementing GraphQL which retrieves the same information as REST but in just one HTTP call by merging all the parameters into one request.

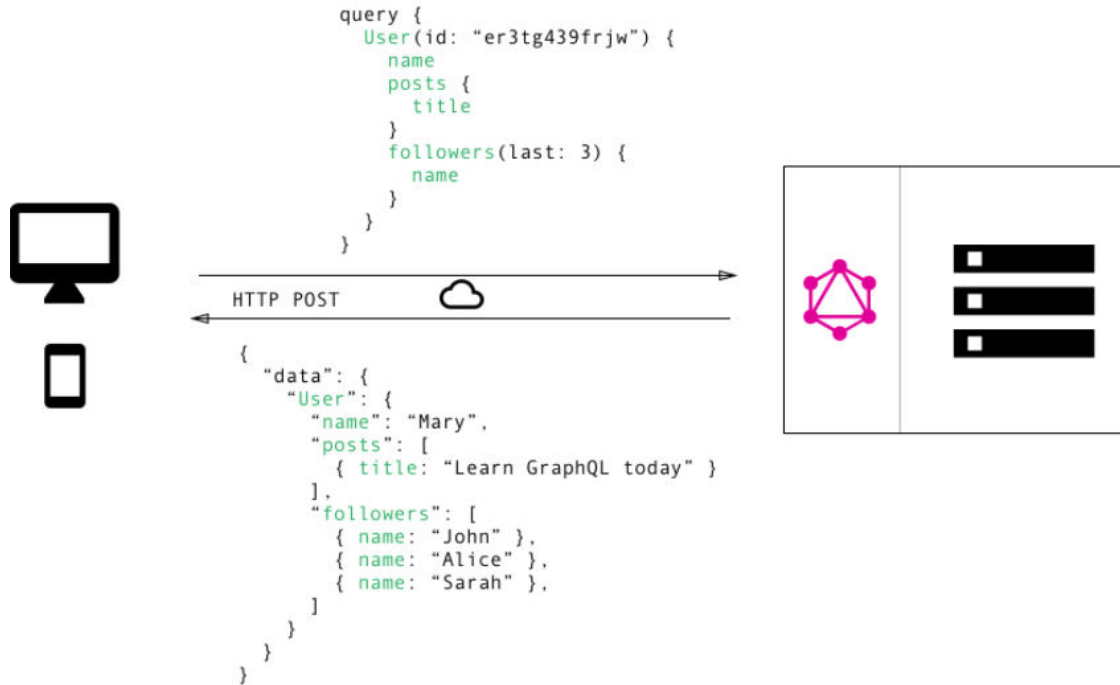


Figure 9: Data Fetching with GraphQL

Publish-Subscribe: GraphQL subscriptions are based on a simple publish-subscribe system. In the GraphQL server-side subscriptions package, when a client makes a subscription, we simply use a map from one subscription name to one or more channel names to subscribe to the right channels. The query (subscription query) will be executed every time some data/event is published to one of these channels. We think a common pattern will be to publish mutation/update results to a channel, so a subscription can send a new result to clients whenever a mutation happens.

GraphQL Subscriptions: Publish-Subscribe based subscriptions help in getting real time updates from a backend data source and send it to the clients when a specific event occurs. They are usually implemented using Web Sockets ─ a steady connection held by the server to the clients. (Apollo GraphQL, 2016) Each client opens a steady connection to the server specifying the events in which they are interested. From then, whenever

those events get triggered, the server pushes the data through the connection established earlier with the clients.

```
import { PubSub, SubscriptionManager } from 'graphql-subscriptions';
import { schema } from './schema';

const pubsub = new PubSub();
const subscriptionManager = new SubscriptionManager({
  schema,
  pubsub,

});
export { subscriptionManager, pubsub };
```

Figure 10 GraphQL Subscription Server-Side Pub-Sub Configuration

Figure 10 represents the server side pub-sub engine configuration that will be bound onto a web socket.

```
    Mutation: {
        updatedItems: (root,feed) => {
                pubsub.publish(FEED_UPDATED, {feedUpdated: feed.input});
                return feed.input;
        },
    },
     Subscription: {
        feedUpdated:{
          subscribe: () => pubsub.asyncIterator(FEED_UPDATED)
        },
      },
```

Figure 11 Binding the Subscription Server with the GraphQL engine

Figure 11 represents the mutation- insert/update operation triggering a publish message with the updated data and subscription channel listening to the published message.

```
import React, {
    Component
} from 'react';
import client from './ApolloClient'

import {
    gql,
    graphql
} from 'react-apollo';
import {
    ApolloProvider
} from 'react-apollo';


class MyComponent extends Component{
    componentWillMount() {
    this.props.data.subscribeToMore({
      document: UserAccountSubsQuery,
                updateQuery: (prev, {subscriptionData}) => {
                    const newFeedItem = subscriptionData.data.feedUpdated;
                    console.log(newFeedItem);
                    return {
                        items: [newFeedItem],
                        };
                },
    });
    }
    render(){
        const { loading, error, items } = this.props.data;
        return <p> {JSON.stringify(items)}   </p>
    }
}
```

Figure 12 Binding GraphQL queries with React components

Figure 12 represents the code binding the GraphQL subscriptions query with the react components in the front-end i.e. client page.

4.2 New Usage Dashboard

Based on the technologies shortlisted, we designed the following GraphQL-based UI architecture:

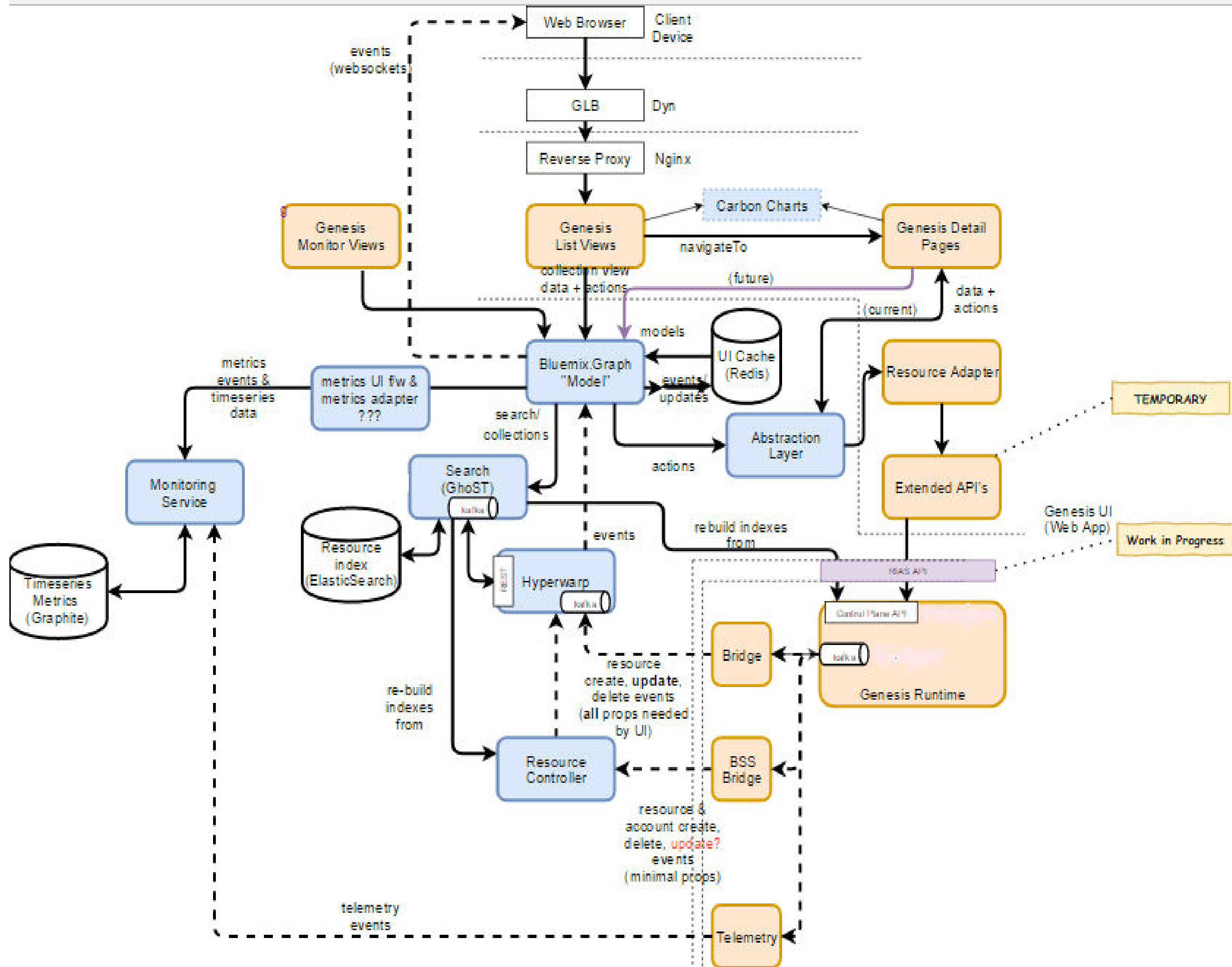


Figure 13 Bluemix Architecture

Figure 13 represents the updated Bluemix console system after integrating our proposed architecture. We created a layer – "Bluemix.Graph" model on top of the message bus thereby having only one endpoint to any of the services below. We also bound a Redis cache with the model to store the frequently accessed but no changed pages in the same session for rendering the UI components instantly.

This architecture has the following key features:

- The application is now React, Redux and GraphQL-based
- Response page does not wait for all the data to be fetched. Instead, it starts rendering as the data becomes available

  - Since we are invoking multiple asynchronous calls and binding each of the results to separate components on the UI, the data gets rendered at a component level rather than waiting for one large data set from the backend data source. Also, since the state skeleton is already stored in Redux, skeleton components are rendered immediately after the request trigger.
- There are no blocking requests
  - We overcome the problem of blocking requests in the existing applications by splitting a single HTTP request into multiple sub queries internally without having a dependency between each of those queries.

As soon as a request is sent from the client to the server, the data from the cache (Redis) is loaded initially while the live data is being fetched from the GraphQL Server. Each component on the UI has a corresponding state stored in the Redux. When the live data from the GraphQL server is received, the redux store and the redis cache are updated with the live data. Any changes in the component state of the redux store will immediately reflect the changes in the UI.

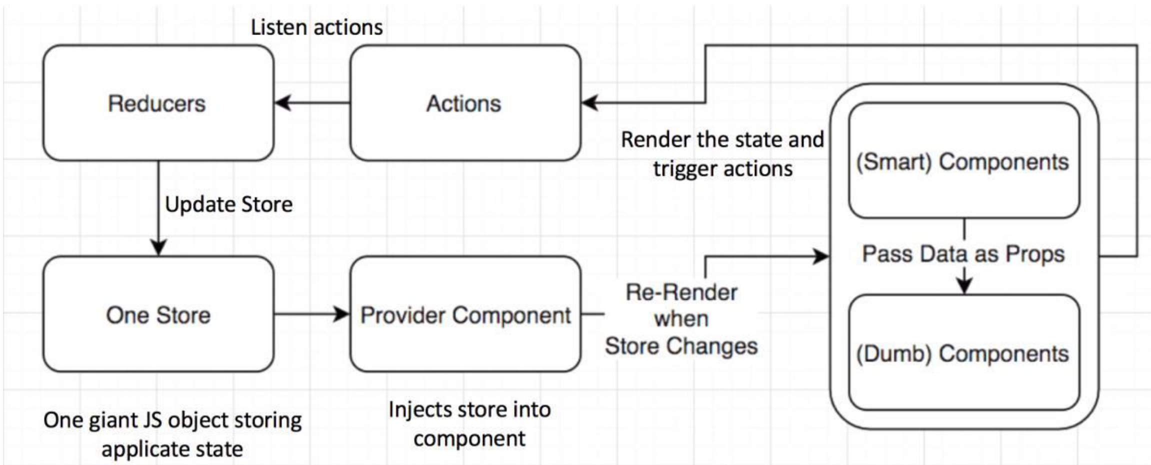


Figure 14 Redux Workflow

State Change: Figure 14 is a React + Redux ecosystem, the workflow usually is:

1. The UI is defined by the state tree and the associated action callbacks are defined through "props" attribute.
2. *Action Creator* normalizes all the user actions such as clicks, reload etc
3. The actions (resultant redux actions) are then sent to a *reducer*. The actual application logic is implemented by the reducer.
4. The state tree is then updated by the reducer and dispatched to a *store* where it is stored.
5. The UI is then updated based on the new store tree in the store.

Based on this ecosystem workflow, we designed and integrated the front end react + redux components architecture to the existing Bluemix Architecture. Figure 15 represents the various actions that a user can trigger on the applications being hosted ─ Open URL, Stop App, Restart App, Rename App, and Delete App.

Whenever a user clicks on one of these actions, an action event is triggered and dispatched to the action creators (both synchronous and asynchronous). The reducer then updates the state tree with the new states in the store. The changes in the state are reflected on the UI by applying on the delta. Figure 15 represents the list of actions that can be triggered on a running service. All these actions will trigger a state change in the redux store management.

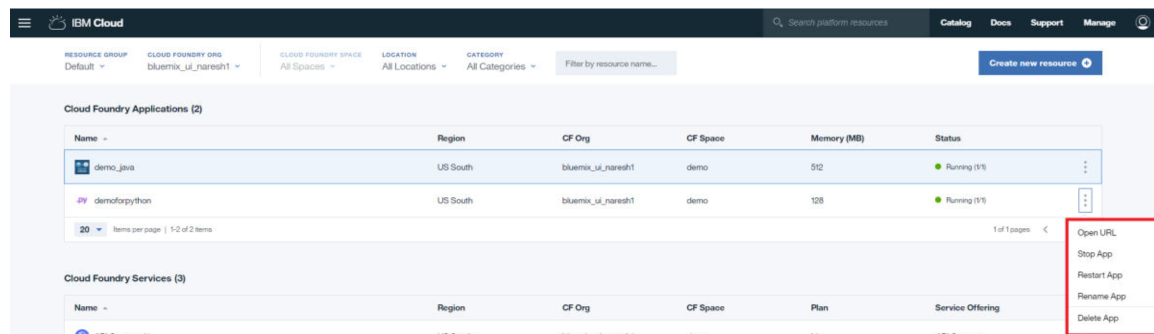


Figure 15 Bluemix UI state change user actions

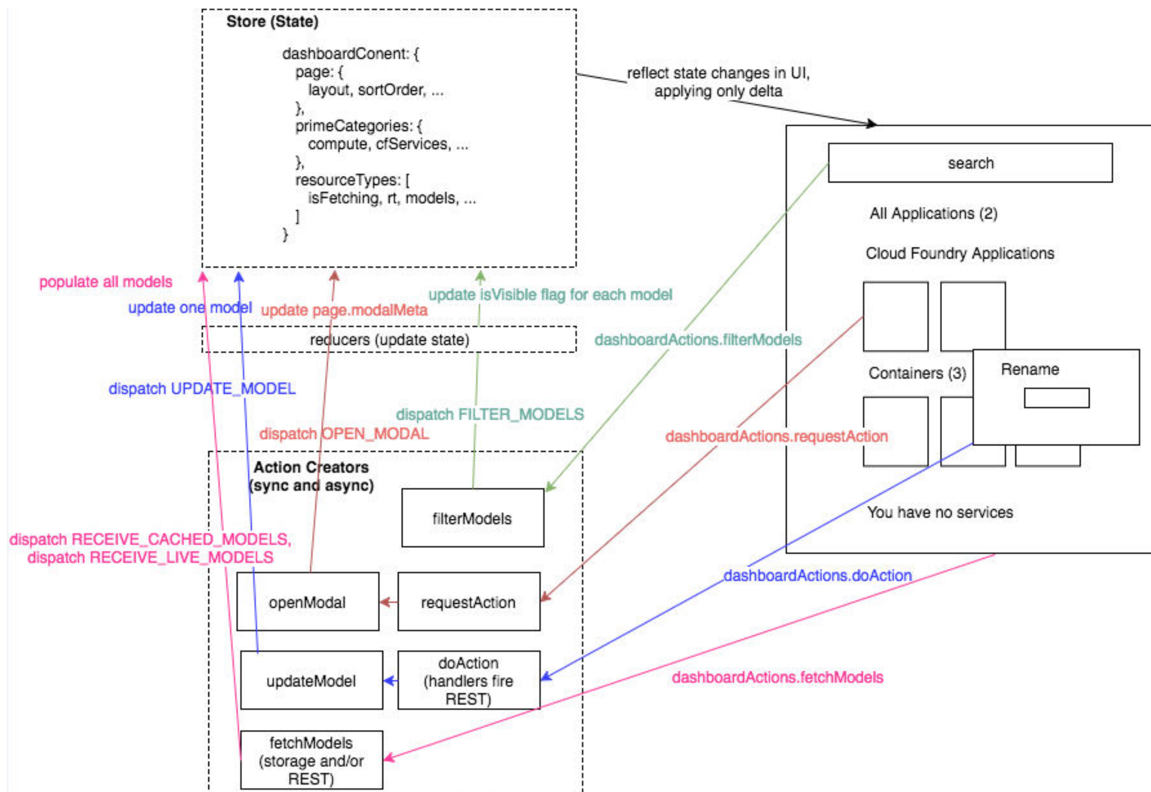


Figure 16 Redux and React Components

Figure 16 represents the Redux and React configuration in the Bluemix. The react components are categorized as Smart and Dumb components. Smart components are those components that trigger actions for change in state and re-render themselves as soon as an action is triggered. However, dumb components only render the data that is passed to them as input source. They do not trigger any actions or interact with the state management in the redux.

Chapter 5

5. Validation and Results

In this section, we present the process of validating the proposed architecture and the results of this research.

It is mandatory, for any research in the domain of Software Engineering, to select a form of validation that is appropriate for the type of research and the method used to obtain the result. Analysis for a formal model/empirical model, evaluation for a descriptive model, experience for a qualitative model and Persuasion for a technique are some examples of the different types of the software engineering research validation.

A typical software engineering research result falls into one of the following types – Procedure/Technique, Qualitative or Descriptive model, Empirical model, Analytic model, Tool, Specific solution or a Report (Mary, Shaw., 2003).

5.1 Research Results

As a result of our research, we were successful in creating a GraphQL-based UI architecture and implementing it on a real-time system to validate the model. We could achieve a significant performance boost by:

- Decreasing the number of HTTP requests by introducing a GraphQL middle tier layer.
  Instead of invoking multiple HTTP calls to the different backend data sources, we only make a single HTTP call and split the call into multiple GraphQL calls internally thereby allowing asynchronous streaming of data from multiple backend data sources which cannot be done using the traditional HTTP protocol.
- Rendering the skeleton components on the UI even before the data is available.
  We store the states of the components in a Redux store and render the skeleton components i.e. UI components without the actual data render immediately as

soon as a request is triggered. This helps in providing an optimistic UI to the user i.e. showing a response as soon as a request is received.

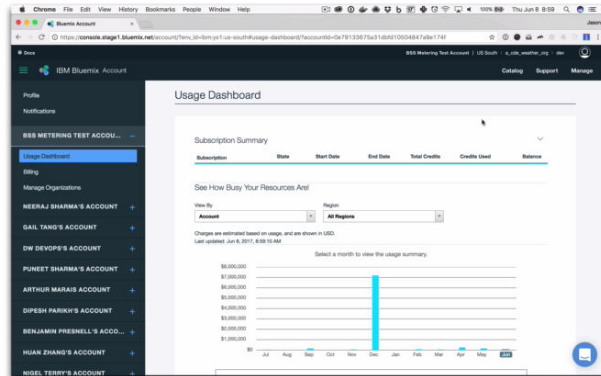
- Loading the components asynchronously before all the requests are complete.
  We build the entire UI using components and bind each of them with a different query to the backend systems. This allows us in executing multiple queries in parallel and decoupling the dependency between the components. Components render the data as soon as they receive from the query without having to wait for the result from other queries which are slower.
- Integrating Redis cache to make the subsequent page loads fast.
  We perform a MD5 of the session token and the URI requested and store the components states in a Redis cache. This allows us to render the data instantaneously from the cache instead of hitting the backend systems again and again even though the data does not change during the session.
- Implementing GraphQL data management to make view switching faster.

5.2 Validation

The proposed GraphQL-based UI architecture in Figure 5 is generic and can be integrated into any of the existing web applications. To validate our proposed architecture, we implemented and integrated the GraphQL-based UI architecture into the existing Usage Dashboard system of the Bluemix cloud. We analyzed and recorded the performance improvements in the usage information section loading time and account switching time in the Usage dashboard. We also analyzed the performance of loading the cached web pages whose values do not change in the same session.

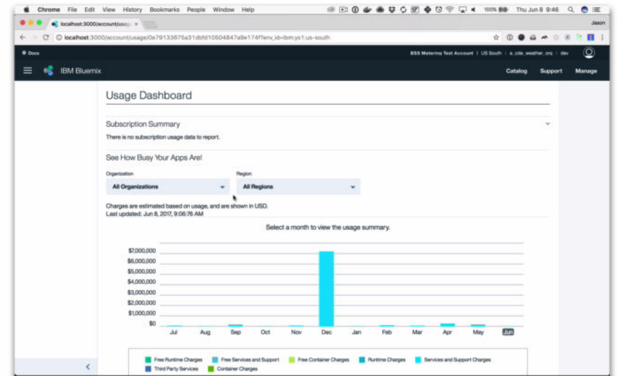
Our recorded loading times indicate that the usage information section loads 245% faster and the account switching is done 153% faster than the existing system.

Current Usage Dashboard



Explorable UI: 43.61s
Real Data: 43.61s

New Usage Dashboard



Explorable UI: 1.40s
Real Data: 17.79s

Figure 17 Loading times of Current Usage Dashboard vs New Usage Dashboard

Figure 17 represents the comparison of the load times between the existing usage dashboard and the new usage dashboard after implementing the GraphQL-based UI architecture. Real data in the existing system renders after 43.61 seconds while the new GraphQL-based dashboard renders the same data within 17.79 seconds which is 245% faster.

Application Module	Initial Loading Time	Present Loading Time
Rendering Usage Dashboard (4 applications, 103 services)	43.61s	17.79s
Rendering Usage Dashboard (4 applications, 66 services)	29.37s	12.64s
Rendering Usage Dashboard (2 applications, 83 services)	37.81s	15.89s
Bluemix Metering Account Switch	8.22s	1.53s
All Accounts Switch	16.46s	5.48s
VPN Account Switch	7.39s	1.82s
All Accounts Switch (Cached)	16.46s	0.73s
Metering Account Switch (Cached)	8.21s	0.13s

Figure 18 Comparison of the previous and the present loading times

Figure 18 represents the tests performed on the Usage Dashboard using different user accounts, with and without caching pages in a session, and switching accounts. These results clearly indicate that there is a significant increase in the performance of the application.

Chapter 6

6. Implementation

In this section, we present the details of implementing the architecture and configuring the GraphQL server.

The core component of improving the performance is introducing a GraphQL layer in between the front-end and the back-end systems.

6.1 GraphQL Components

A GraphQL server is made up of 5 distinct components. Of these, few are common to any data accessing applications while the others are specific and unique to GraphQL. We will start with the common components and later dwell into the GraphQL-specific concepts and its implementation.

6.1.1 GraphQL Server

GraphQL Server is a flexible, community driven, production-ready HTTP GraphQL server plugin for Node.js. It works with any GraphQL schema built with GraphQL.js, Facebook's reference JavaScript execution library, and you can use GraphQL Server with all popular JavaScript HTTP servers, including Express, Connect, Hapi, Koa, Restify, and Lambda.

GraphQL server supports all the common semantics for sending GraphQL requests over HTTP. This enables us to query the GraphQL server from popular clients like Relay and Apollo. It also supports certain extensions to this protocol that enable the client to send multiple GraphQL operations in a single request.

Installation

We use npm package manager to install the required packages that best match our server framework.

Examples:

To install Express: **npm install graphql graphql-server-express**

To install Restify for REST APIs: **npm install graphql graphql-server-restify**

To install KOA server: **npm install graphql graphql-server-koa**

To install Hapi server: **npm install graphql graphql-server-hapi**

Adding a GraphQL Endpoint

We configure a GraphQLOptions object and register it with the GraphQL server. GraphQLOptions object has the properties listed below:

- graphiql: If set to true, an interactive graphical in-browser IDE for GraphQL is loaded.
- schema: This is a mandatory property. It is an instance of the GraphQLSchema present in the GraphQL.js and maps the data definitions with our query parameters.
- context: Information that is useful during a query execution. Checking the current logged in user is a canonical example of this property. Request object is passed as the context the graphql() function present in the GraphQL.js/src/execute.js if no explicit context value is set.
- rootValue: This property is used in resolving root fields defined in our schema. It represents the top of our metaphorical graph of data. This value is passed to the graphql() present in the GraphQL.js/src/execute.js.
- pretty: If set to true, the JSON response will be pretty-printed.
- formatError: This property helps in formatting any errors that might arise while fulfilling a GraphQL operation. The default spec-compliant formatError function will be used if not explicit formatting function is passed.

- extensions: An optional function for adding additional metadata to the GraphQL response as a key-value object

Sample code to configure GraphQL server

```
graphQLServer.use ('/graphql', _bodyParser2.default.json(), graphqlHTTP({
schema: executableSchema,
graphiql: true,
context: {},
          extensions ({ document, variables, operationName, result }) {
            return { runTime: Date.now() - startTime };
          }
}));
```

Sample code to connect with Express

```
import bodyParser from 'body-parser';
import { graphqlExpress } from 'graphql-server-express';
import express from 'express';
var app = express();
app.use('/graphql', bodyParser.json(), graphqlExpress({ schema: myGraphQLSchema }));
app.listen(8080);
```

Sending Requests

GraphQL Server accepts both the GET and POST requests.

For POST Requests: All POST requests are sent like a JSON body to the GraphQL server. Each request either contains a query or an operationName or both (in case of a named query) and may contain variables.

Sample POST request:

```
{
  "query": "query aTest($arg1: String!) { test(who: $arg1) }",
  "operationName": "aTest",
  "variables": { "arg1": "me" }
}
```

A batch of queries can be requested by simply sending a JSON-encoded array of queries.

Sample code for batch of queries in one request:

```
[
  { "query": "{ testString }" },
  { "query": "query q2{ test(who: \"you\" ) }" }
]
```

For GET Requests: A GET request must pass query and optionally variables and operationName in the URL.

Sample GET Request

```
GET
/graphql?query=query%20aTest(%24arg1%3A%20String!)%20%7B%20test(who%3A%20%24ar
g1)%20%7D&operationName=aTest&variables=me
```

GraphiQL

GraphiQL is a graphical interacive in-browser IDE for GraphQL.

```
Installing GraphiQL
npm install --save graphiql
```

GraphiQL with Express (Web framework for Node.js):

```
import { graphiqlExpress } from 'graphql-server-express';
app.use('/graphiql', graphiqlExpress({
  endpointURL: '/graphql',
}));
```

GraphiQL with Connect (Web framework for Node.js):

```
import { graphiqlConnect } from 'graphql-server-express';
app.use('/graphiql', graphiqlConnect({
  endpointURL: '/graphql',
}));
```

GraphiQL with Hapi (Web framework for Node.js):

```
import { graphiqlHapi } from 'graphql-server-hapi';
```

```
server.register({
  register: graphiqlHapi,
  options: {
    path: '/graphiql',
    graphiqlOptions: {
      endpointURL: '/graphql',
    },
  },
});
```

GraphiQL with KOA (Web framework for Node.js):

```
import { graphiqlKoa } from 'graphql-server-koa';
router.get('/graphiql', graphiqlKoa({ endpointURL: '/graphql' }));
```

Figure 19 GraphiQL Interface — An in-browser IDE for GraphQL

Below listed are the components required for configuring and exposing data through a GraphQL µ-service.

1. Connector
2. Model

3. Schema
4. Query Resolver
5. Data Resolvers

6.1.2 Connector

This component is responsible for the configuration and creation of methods responsible for connecting with the data sources. The data sources can be anything i.e. relational databases, NOSQL databases, REST APIs, SOAP APIs or a blend of all these data sources.

Sample Code:

We implement a GraphQLConnector class that adds the core functionality for all GraphQL connectors.

- Class Properties
  - apiBaseUri
  - cacheExpiry
  - cacheExpiry
  - headers
- Class Methods
  - addBluemixToken(req)
  - addIMSToken(imsUserId, imsToken)
  - get(endpoint)
  - post(endpoint [, body [, options]])
  - put(endpoint [, body [, options]])
- Internal Dependencies
  - Request: an instance of request-promise used to make all HTTP requests.
  - Redis: It is an instance of @console/console-platform-key-value-store. This is the library we use for storing and retrieving cached responses

- Loader: It is an instance of DataLoader. This is the library we use for fetching data and avoiding sending a bunch of duplicate requests to a data source during a single request to GraphQL.


```
{
    "token": "eyJjYWxsVGFibGUiOlt7Im9mZnNldCI6MCwic2l6ZSI6MTAsInByZXZpb
    "more_data": true,
    "items": [{
        "service_name": "bluemix",
        "cname": "bluemix",
        "creation_date": "2017-07-07T21:01:12Z",
        "type": "cf-application",
        "organization_guid": "d8638991-fc6b-4483-bcf5-3dbbcf97c8a0",
        "account_id": "7ba557bc2e9a6e23bae8df542a6ca3ce",
        "ctype": "public",
        "provider": "crn:v1:bluemix:public:cloud-foundry:::::",
        "scope": "s/85c9376e-c596-4791-b6d2-223f3661b605",
        "modification_date": "2017-07-07T21:01:12Z",
        "name": "yellowpages",
        "resource_id": "34a2f61f-58a0-4bac-ba85-ba796f9236b2",
        "doc": {
            "memory": 256,
            "production": false,
            "instances": 1,
            "health_check_timeout": null,
            "running_instances": 1,
            "detected_buildpack_guid": "91e3963d-710d-4b8e-b53c-a529db1
            "buildpack": null,
            "detected_buildpack": "SDK for Node.js(TM) (ibm-node.js-4.8
            "ports": null,
            "available_domains": [{
                "name": "mybluemix.net",
                "router_group_type": null,
                "guid": "f4b90d7e-2cd3-4d30-b200-f28bbaf6be20",
                "router_group_guid": null
            }],
            "routes": [{
                "path": "",
```


Figure 20 Message bus data source format

```
import rp from 'request-promise';

import { subscriptionManager, pubsub } from './subscriptions';
//Fetching data from GhoST API with bearer token
var options={
    method: 'GET',
    uri: 'http://localhost:3971/graph/search?q=thing&resource_type=virtualserver&q.limits=5',
    body: {

    },
    headers: {
        'test' : true,
        'Accept': 'text/plain',
        mocha_test: 'mocha token'
    },
    json:true

}
```

Figure 21 GraphQL connector with a REST endpoint

6.1.3 Model

This component is responsible for the configuration and creation of methods responsible for accessing and manipulating the data. CRUD operations can be performed over the data using this component.

6.1.4 Schema

This component is responsible for the configuration and creation of the data definitions that GraphQL will make available.

```
# Comments start with a hash and are auto-converted to documentation.

type MySchema {

  # Each field supports its own documentation, where you might add notes

  # about what kind of data it returns, or — if it maps to a different

  # property in the source data — the field it maps to.

  id: Int!

  # for formatting and adding links to descriptions.
```

```
  name: String

}
```

```
import { GraphQLSchema } from 'graphql';

const typeDefinitions = `
#Fetching user account information from the GhoST API
type Query{
    #Data received from GhoST API for a specific user account
    items: [Item]
}

type Item{

    #Name of the service
    service_name: String

    #CNAME of the service
    cname: String

    #Service created date
    creation_date: String

    #Domains accessible by the service
    available_domains: [Domain]

    #GUID of the organization
    organization_guid: String

    #AccountID of the user
    account_id: String

    #Customer Type
    ctype: String
```

Figure 22 GraphQL Schema

6.1.5 Query Resolvers

This component is responsible for the querying function that enables to expose the data to GraphQL clients.

The query resolver resolves how GraphQL is able to load the correct data for a given query. It has access to the data models for all defined schema types, as well as any variables passed in the query.

Working:

When a query is sent to GraphQL, we need to tell it how to find the data required to build the response.

The GraphQL server grabs the Query property of the resolver's object and fires the function that matches the query name.

For example, if this is our query:

```
query {
  endpoints {
    # desired fields listed here
  }
}
```

GraphQL will expect a query resolver like this:

```
const resolvers = {
  Query: {
    endpoints: (root, variables, context) => { /* ... */ },
  },
};
```

This resolver would be expected to return a Promise that resolves with data for all fields the query can potentially return.

```
//Resolver
const resolver = {
    Query: {
        items(obj, args, context, info) {
          return  cfPromise.getData().then((res) => {
                return res.items;
          });
        }
    },
    Item:{
        doc(obj, args, context, info) {
            return obj.doc;
        }
    },
    Docs:{
        available_domains(obj, args, context, info) {
            return obj.available_domains;
        },
        metadata(obj, args, context, info) {
            return obj.metadata;
        },
        routes(obj, args, context, info) {
            return obj.routes;
        },
        services(obj, args, context, info) {
            return obj.services;
        },
        docker_credentials_json(last_operationbj, args, context, info) {
            return obj.docker_credentials_json;
        }

    },
    Routes:{
```

Figure 23 GraphQL Resolver

6.1.6 Data Resolvers

This component is responsible for mapping the source data with the GraphQL schema definition. They are the functions that map the data loaded from the backend data sources (Examples are REST APIs, SOAP APIs, Relational Databases and NOSQL databases) to the GraphQL query. These functions inform the GraphQL on which property of the data object should be mapped to the given field.

Working: The GraphQL server accepts an object that contains the resolver functions for fields defined in the schema, as defined in section 5.1.4. Each resolver function receives

the data object returned by the query resolver, and is able to execute arbitrary code to change the data as needed.

A simple data resolver:

```
const dataResolvers = {
  // Each schema type has its own set of data resolvers.
  SchemaName: {
    // This is the data resolver for the `fieldName` field.
    fieldName: data => data.propertyThatFieldNameReturns,
  },
};
```

A more complex data resolver:

It is possible to run arbitrary code in a resolver, so we can do anything in a resolver that we would do anywhere else in our app. This could include manipulating the data, tracking analytics, or setting up performance measurements. Here's how a more complex data resolver might look:

```
const dataResolvers = {

    // Each schema type has its own set of data resolvers.
    SchemaName: {

        // This is the data resolver for the `fieldName` field.
        fieldName: function(data) {

            // Tell GraphQL which data should be used as the field's value.
            const value = data.propertyThatFieldNameReturns;

            // You can make changes to the data, if required.
            const changedValue = `The value is "${value}".`;

            // We can also run analytics here, or other tools.
```

```
                AnalyticsTool.trackFieldUsage('fieldName');

                return  changedValue;
            },
        },
};
```

It is not necessary to define a data resolver for every field. GraphQL has a default resolver in place for any fields that don't define one.

In many cases, data resolvers are not necessary. For example, assume the data source returns an object like this:

```
{
  "firstName": "Naresh",
  "lastName": "Eeda",
  "email": "naresh.eeda@ibm.com"
}
```

And the schema is defined like this:

```
type ExampleSchema {
  firstName: String!
  lastName: String!
  email: String!
}
```

By default, GraphQL will attempt to match fields to data properties with the same name, so the `firstName` field in the query will look for data at `data.firstName`, and `email` will attempt to use `data.email` and so on.

It is good practice to map our GraphQL schema to the data response 1-to-1 whenever possible, as it reduces the amount of code required and decreases the amount of logic in GraphQL — which means one less place to check if things aren't working as expected.

Custom Resolver:

However, if there's a need to change the name of a field or create a new field from the data respose — for example, creating a fullName field from data.firstName and data.lastName — then we need to define data resolvers.

These are defined in an object that matches the schema structure, and each field receives the data (which is retrieved by the query resolver) as its argument. Here's how we'd create the fullName field resolver:

```
const dataResolvers = {
  Person: {
    fullName: data => `${data.firstName} ${data.lastName}`,
  },
};
```

Chapter 7

7. Conclusion and Future Work

The motivation behind conducting this research is to improve the performance of the existing applications by implementing a novel GraphQL-based UI architecture that enables efficient retrieval of data from backend data sources and creation of high-performance real-time dashboards. This research is not a contribution to a specific module of a specific product – Usage dashboard module of the Bluemix, or a specific field – Software Engineering for boosting web applications performance. Instead, it is a contribution to almost all the fields where there is an interaction with the data sources – Web/Console applications, or usage of data sets – Data Science, end user experience – Non-functional requirements.

We were successful in creating a GraphQL-based UI architecture and implementing it on a real-time system to validate the model. We could achieve this performance boost by (section 5.1):

- Decreasing the number of HTTP requests by introducing a GraphQL middle tier layer.
- Rendering the skeleton components on the UI even before the data is available.
- Loading the components asynchronously before all the requests are complete.
- Integrating Redis cache to make the subsequent page loads fast.
- Implementing GraphQL data management to make view switching faster.

The novelty in this research, multiple asynchronous calls and UI caching, helped boost the performance of the systems significantly. However, the upcoming features of GraphQL ─ Live (notify the server about the desire to receive live updates about a certain field); Stream (to defer loading individual items of an array); and Defer (specifying that some part of the query can arrive later) seem even more promising for faster fetching and rendering of the real-time data. We plan to integrate these new features, once publicly available, into the existing GraphQL-based architecture and subscriptions to make it even more efficient.

References

Abe, N., Zadrozny, B., and Langford, J. (2006). Outlier detection by active learning. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and Data mining* (pp. 504-509). ACM.

Apache thrift documentation (2017). Retrieved August 12, 2017 from https://thrift.apache.org/

Auth0/blog. Retrieved March 21, 2017, from https://github.com/auth0/blog/blob/master/_posts/2016-01-11-updated-and-improved-more-benchmarks-virtual-dom-vs-angular-12-vs-mithril-js-vs-the-rest.markdown

Basili, V., Heidrich, J., Lindvall, M., Munch, J., Regardie, M., & Trendowicz, A. (2007). GQM^+ Strategies--Aligning Business Strategies with Software Measurement. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* (pp. 488-490). IEEE.

Belqasmi, F., Singh, J., Melhem, S. Y. B., & Glitho, R. H. (2012). Soap-based vs. restful web services: A case study for multimedia conferencing. *IEEE internet computing*, *16*(4), 54-63.

Bernstein, D. (2014). Cloud Foundry aims to become the OpenStack of PaaS. *IEEE Cloud Computing*, *1*(2), 57-60.

Bershad, B. N., Anderson, T. E., Lazowska, E. D., & Levy, H. M. (1990). Lightweight remote procedure call. *ACM Transactions on Computer Systems (TOCS)*, *8*(1), 37-55.

Buna, S. (2016). Learning GraphQL and Relay. Packt Publishing Ltd. https://dev-blog.apollodata.com/new- features-in- graphql-batch- defer-stream- live-and- subscribe-7585d0c28b07

Byron, L. (n.d.). GraphQL: A data query language. Retrieved December 12, 2017, from https://code.facebook.com/posts/1691455094417024

Chaudhuri, S., & Narasayya, V. (2007). Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases* (pp. 3-14). VLDB Endowment.

Cherukuri, R. (2010). *U.S. Patent Application No. 12/789,487*.

Christensen, J. H. (2009). Using RESTful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* (pp. 627-634). ACM

Cerami, E. (2002). *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL*. " O'Reilly Media, Inc.".

Dalmasso, I., Datta, S. K., Bonnet, C., & Nikaein, N. (2013). Survey, comparison and evaluation of cross platform mobile application development tools. In Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International (pp. 323-328). IEEE.

Ping, Y., Kontogiannis, K., & Lau, T. C. (2003, September). Transforming legacy Web applications to the MVC architecture. In *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on (pp. 133-142)*. IEEE.

Farhi, O. (2017). Adding State Management with ngrx/store. In *Reactive Programming with Angular and ngrx* (pp. 31-49). Apress.

Few, S., & Edge, P. (2008). What ordinary people need most from information visualization today. *Perceptual Edge: Visual Business Intelligence Newsletter*, 1-7.

Gackenheimer, C. (2015). Introducing flux: An application architecture for react. In *Introduction to React* (pp. 87-106). Apress.

Gackenheimer, C. (2015). What Is React?. In *Introduction to React* (pp. 1-20). Apress.

Johnny. (2017). The Benefits of Using React. Retrieved October 31, 2017, from https://www.sourcetoad.com/app-development/the-benefits-of-using-react/

Ghosh, P., Rau-Chaplin, A. (2006). *Performance of Dynamic Web Page Generation for Database driven Web Sites*. Proceedings of the International Conference on Next Generation Web Services Practices. IEEE Computer Society: Washington, DC, USA, pp. 56-63.

Kerzner, H. (2017). *Project management metrics, KPIs, and dashboards: a guide to measuring and monitoring project performance*. John Wiley & Sons.

Kobylinski, K., Bennett, J., Seto, N., Lo, G., & Tucci, F. (2014). Enterprise application development in the cloud with IBM Bluemix. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering* (pp. 276-279). IBM Corp.

Koetsier, J. (2016). Evaluation of JavaScript frame-works for the development of a web-based user interface for Vampires. *Universiteit Van Amsterdam*.

Langendoen, K., Bhoedjang, R., & Bal, H. (1997). Models for asynchronous message handling. *IEEE concurrency*, *5*(2), 28-38.

Luo, Q., Naughton, J.F., Xue, W. (2008). Form-based Proxy Caching for Database-backed Web Sites: Keywords and Functions. *The Vldb Journal – VLDB 17(3)*, pp. 489-513.

Maeda, K. (2012). Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on* (pp. 177-182). IEEE.

Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M., & Urso, A. (2016). The Database-is-the-Service Pattern for Microservice Architectures. In *International Conference on Information Technology in Bio-and Medical Informatics* (pp. 223-233). Springer International Publishing.

Monideepa Tarafdar & Jie Zhang (2008). Determinants of Reach and Loyalty — A Study of Website Performance and Implications for Website Design, *Journal of Computer Information Systems*, 48:2, 16-24.

Myers, D.S, Carlisle, J.N., Cowling, J.A, Liskov, B.H. (2007). MapJAX: Data Structure Abstractions for Asynchronous Web Applications. *Proceedings of the USENIX Annual Technical Conference*. USENIX Association: Berkeley, CA, USA.

Passaglia, A. (2017). Vue.js 2 cookbook: build modern, interactive web applications with Vue.js.

Paulson, L.D. (2005). Building Rich Web Applications with Ajax. *Computer 38(10)*, pp. 14-17

Pelletier, J. (2015). The Front-End Spectrum – Jeff Pelletier – Medium. Retrieved November 01, 2017, from https://medium.com/@withinsight1/the-front-end-spectrum-c0f30998c9f0

Popić, S., Pezer, D., Mrazovac, B., & Teslić, N. (2016). Performance evaluation of using Protocol Buffers in the Internet of Things communication. In *Smart Systems and Technologies (SST), International Conference on* (pp. 261-265). IEEE.

Ramaswamy, L., Liu, L., Arun, I. (2007). Scalable Delivery of Dynamic Content Using a Cooperative Edge Cache Grid. *IEEE Transactions on Knowledge and Data Engineering 19(5)*, pp. 614-630.

Realtime Updates with GraphQL Subscriptions. (n.d.). Retrieved October 31, 2017, from https://www.howtographql.com/react-apollo/8-subscriptions

Shaw, M. (2003). Writing good software engineering research papers. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 726-736). IEEE.

Slee, M., Agarwal, A., & Kwiatkowski, M. (2007). Thrift: Scalable cross-language services implementation. *Facebook White Paper*, *5*(8).

Stubailo, S. (2016). New features in GraphQL: Batch, defer, stream, live, and subscribe. Retrieved October 16, 2017, from https://dev-blog.apollodata.com/new-features-in-graphql-batch-defer-stream-live-and-subscribe-7585d0c28b07

Sumaray, A., & Makki, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th international conference on ubiquitous information management and communication* (p. 48). ACM.

Tihomirovs, J., & Grabis, J. (2016). Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics. *Information Technology and Management Science*, *19*(1), 92-97.

Top 10 Issues Affecting Web App Performance. (2017, June 06). Retrieved October 12, 2017, from http://www.360logica.com/blog/top-10-issues-affecting-web-app-performance

Ubbink, J., Wouters, R., Zannone, N., Prins, M. W. J., & Brekelmans, W. W. C. C. (2017). *Architectural Design Document*.

Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017). Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (pp. 1009-1024). ACM.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th* (pp. 583-590). IEEE.

Vingralek, R., Breitbart, Y., Sayal, M., Scheuermann, P. (1999). Web++: A System for Fast and Reliable Web Service. *Proceedings of the USENIX Annual Technical Conference*. USENIX Association: Berkeley, CA, USA, pp. 171-184.

Wang, X., Zhao, H., & Zhu, J. (1993). GRPC: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, *27*(3), 75-86.

Wei, C., Buffone, R., & Stata, R. (2012). System and method for website performance optimization and internet traffic processing. *U.S. Patent No. 8,112,471*. Washington, DC: U.S. Patent and Trademark Office.

Xu, M., Xu, X., Xu, J., Ren, Y., Zhang, H., & Zheng, N. (2014). A forensic analysis method for redis database based on rdb and aof file. *Journal of Computers*, *9*(11), 2538-2544.

Zhang, X., Cao, Y., & Mu, X. (2011). The Dynamic Retrieval Tree Menu Based on Dojo. In *Information Technology, Computer Engineering and Management Sciences (ICM), 2011 International Conference on* (Vol. 1, pp. 202-205). IEEE.

Curriculum Vitae

Name:	Naresh Eeda
Post-secondary Education and Degrees:	Jawaharlal Nehru Technological University Hyderabad, Telangana, India 2007-2011 B.Tech.
	The University of Western Ontario London, Ontario, Canada 2016-2017 M.Sc.
Honors and Awards:	Western Graduate Research Scholarship 2016-2017
Related Work Experience	Teaching Assistant The University of Western Ontario 2016-2017
	Software Developer Intern IBM Software Lab 2017

Funds:

- **Approved and funded by IBM Center for Advanced Studies**
- **Applied for matching funds from NSERC**