Software Architectural Patterns: A Performance Analysis

A Technical Research Paper
In STS 4600
Presented to
The Faculty of the
School of Engineering and Applied Science
University of Virginia
In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science in Computer Science

By
Vineeth Gaddam

April 30, 2020

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: ______________Vineeth Gaddam_________________________________

Approved: _______________________________________ Date ________________________
Nada Basit, Department of Computer Science

Software Architectural Patterns: A Performance Analysis

Nada Basit
Research Supervisor
UVa Department of C.S

Sai Konuri
UVa Class of 2020
B.S. Computer Science

Aman Garg
UVa Class of 2020
B.S. Computer Science

Vineeth Gaddam
UVa Class of 2020
B.S. Computer Science

***Abstract*—A rise in demand for speed and reliability in websites has focused performance as a pivotal factor in the development of applications across the IT industry. This document is a compare and contrast analysis of how four different architectural patterns perform with respect to pre-defined metrics. The architectural patterns under study are the following: Model-View-controller (MVC), Layered, Microservice, and Command Query Responsibility Segregation (CQRS). The findings suggest that the MVC pattern performs the most efficiently for the sample application used. However, they also suggest that when considering performance, there are other factors such as ease of development that are equally as important as performance.**

***Index Terms*—architecture, performance, metrics**

I. INTRODUCTION

The digital world has undergone a revolution and demand for internet services of all aspects in daily life have pushed the boundaries of the software industry. People's attention spans have dwindled and it is more important than ever for software businesses to adapt to such demands from their users [4]. The need for real time data requires enhancing speed, computational resources, and distribution. Some methods that the industry has developed in order to sufficiently meet these demands include: caching, vertical and horizontal scaling, content distribution networks (CDN's), compression, sharding, etc [4]. These are all effective methods of improving the performance of an application, but they fail to consider whether the underlying software architecture could possibly have an effect on the performance. This is what this documents aims to explore and find any suggestions that architecture could be a factor affecting the performance of thousands of web applications running around the world.

The team selected four different architectures: MVC (Model-View-Controller), Layered, Microservice, and CQRS (Command Query Responsibility Separation). These are all common patterns used by developers, which convinced the team to explore them. The application that was chosen to be implemented in each of these architectures separately was a simple textbook catalog service. This was chosen due to the availability of public data at the institution. Each architecture used a replica of the same database in order to ensure that the database did not have an impact on the performance. The goal of this compare and contrast analysis of these architectures is to get more insight on whether an architecture could potentially impact the performance of an application.

II. ARCHITECTURAL PATTERNS

A software architectural pattern is the layout of a software application with respect to both the codebase and the higher level tools. It helps to define the basic characteristics and behavior of an application [1]. Although it is common practice in the industry to use architecture patterns, selecting one that meets the different needs required by their application is a difficult task. These requirements include scalability, reliability, development, and testability [1]. For this research, the team focuses on the performance needs of an application. Performance requirements may not seem important at first glance, but they can negatively impact user experience of a product which, in turn, can hurt customer trust and brand loyalty.

A. Model View Controller

In the Model View Controller (MVC) architecture, there primarily exists three main layers; model, view, and controller. The MVC architecture is mostly used to develop web-based programs. The three layers are further described below: [2]

- Model - The logic used to interact with the data. The model helps in creating, retrieving, and modifying data in the database [2].
- View - Allows the user to interact with the application [2].
- Controller - Works with the model layer to determine which view should be displayed based on the users requests [2].

The application that was made for the analysis was developed by utilizing Ruby on Rails, which is a framework that contains structures for web pages and web services. [3] The chosen database was SQLITE due to its simplicity (a replica of the same database is used in the other architectures). While working on the application, it was assumed that the application was based on requirements created by the team and not a client. The MVC framework could be further explored by using different frameworks such as Spring MVC, Django, and ASP.NET, which could potentially have different degrees of performance and reliability.

B. Layered

In the layered architecture pattern, different components are organized into horizontal layers [1]. A typical layered architecture has 4 layers which are described as follows:

- Presentation - Handles user interface and browser communication logic [1].
- Business - Responsible for handling specific business rules associated with the request [1].
- Persistence - Responsible for communicating directly with the database.
- Database - All the data pertaining to the application is stored in this layer.

Smaller applications may only utilize 3 layers and combine the business and persistence layers into a one business layer if persistence logic is inherently a part of the business logic [1].

The application developed by the team consisted of all 4 layers and each layer was implemented using the following technologies:

- Presentation Layer - ReactJS
- Business Layer - ExpressJS
- Persistence Layer - Python
- Database Layer - SQLite

While developing the application, the team had to make assumptions about what technologies to use in the development process. Since there were no external or client requirements, the technologies used to develop the app were ones the team was previously familiar with. It is possible that there were better alternatives to the technologies chosen, for example if a different framework had been used for the presentation layer instead of ReactJS, there could have been performance benefits but the team assumed that those differences were negligible.

C. Microservice

The microservice architecture is one that emphasizes many small, decoupled units as opposed to large components that handle a variety of functions [1]. The granulated structure provides for a more streamlined production and reduces the need for application-wide changes when one component of the application is changed [1]. Multiple service components will interact with the user interface layer to allow the application to function as a whole [1].

The sample application created by the research team was monolithic in nature and did not need many services. Thus, the team opted to build the application with docker rather than develop a true microservices application. Docker provides a method of virtualization that simplifies the complexity of dependencies by packaging components and their dependencies into containers. [7]. It is a lightweight virtual machine that virtualizes at the operating system level rather than at the hardware level and is popular in microservice architecture because containers allow the developer to spin up many small services [7]. In reality, this application served to test the enhancements or limitations of docker rather than a truly microservice architecture. Other than the use of docker containers, this application was built to be the same as the application developed using the layered architecture.

D. Command-Query Responsibility Segregation

The Command-Query Responsibility Segregation (CQRS) architecture is a complex architecture that serves a niche programming audience. The primary function of CQRS is the notion of having operations that are able to read data be segregated from the operations that update data. This segregation of data is significant in certain situations. For example, when there is a scenario where there are a lot of writes occurring, the segregation will allow the operations to go into only one part of the database which is the write and not worry about the reads. Having the operations be split will allow the requests to the database to be more streamlined since they are being handled by the write or the read. A single operation cannot be both a read from the database and write to the database. [3] CQRS also utilizes models to manage data, controllers to process requests, and views for the interface. CQRS utilizes a database for the read and the write. The textbook application was once again used with the CQRS implementation in mind. The team created a read and write database. [3]

The application was made by using React js, Python Flask, and SQLite. When a request was made if the command was a write operation it communicated with the write file and then was put on a queue called the event bus. The event bus then gradually also updated the read database. Read operations went directly to the read database. It was assumed that the user would be performing more read tasks than write tasks, which may have had an impact on the effectiveness of the architecture.

III. TESTING

A. Performance Testing with Jmeter

Jmeter is an open source project used for performance testing web applications [5]. It provides features such as thread groups, iterations, HTTP samplers, and Chrome drivers. The team set up three scenarios:

- A user searches for their university, their department, a specific course, and its textbooks.
- The same as the above scenario but the user makes an order for a specific textbook.
- A user makes an order and checks the past orders.

This combination was used in order to use a balance of read heavy, mixed, and write heavy processes. Due to the limitations of the hardware that the team had, the maximum number of threads hitting the application was set to 25. In a more perfect scenario, multiple machines with a large amount of threads would be used to mirror real usage. JMeter was configured to report on the response time (total scenario time) and the size of the data received. The connection time was near 0 ms for all the architecture, so this was left out of the report.

The key components to interpreting the data collected from JMeter are the following:

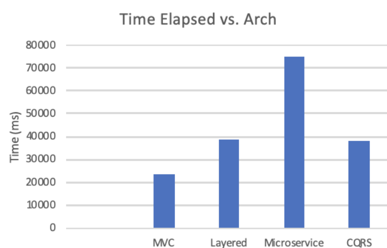
- Response Time - A measurement of how long the architecture takes to return a response. (measured in milliseconds)
- Data Size - The number of bytes that are in the response provided by the architecture.
- CPU Usage - The percentage representing how much of the machine's CPU is utilized by the application
- Memory Usage - The percentage representing how much of the machine's memory is utilized by the application

B. *Testing Environment*

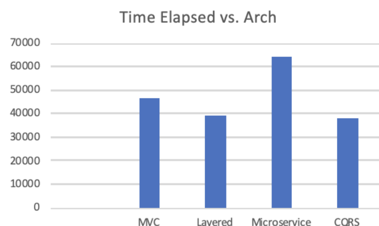
- CPU: 2.3 GHz 8-Core Intel Core i9
- Memory: 16 GB 2400 MHz DDR4
- Number of Threads used for Scenarios: 25
- Number of Threads used to measure CPU usage and memory usage: 100

IV. RESULTS

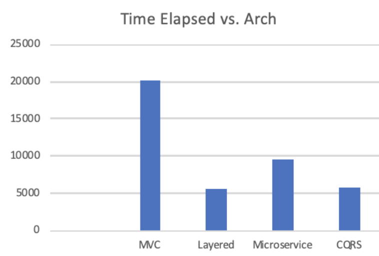
A. *Response Time*



Scenario 1



Scenario 2



Scenario 3

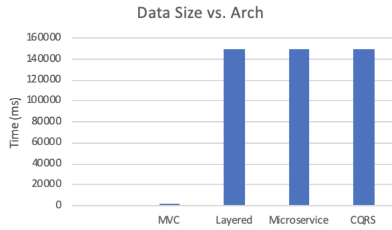
Based on the data gathered for response time, MVC performs the best for the read heavy scenario (Scenario 1), CQRS performs the best for the mixed scenario (Scenario 2), and Layered/CQRS perform equally well for the write heavy scenario (Scenario 3). Microservice performs the worst for all the scenarios except Scenario 3.

The microservice data demonstrates one of the key limitations of the microservice architecture. The textbook catalog application is not complex, and does not require various services as one might see on a heavier website like Amazon. Services such as search indexing, advertising, and recommendations all would benefit from application separation due to the loose coupling to the underlying database schema. However, given the simplicity of the textbook catalog, this was unnecessary. By introducing Docker there is an extra layer of networking required for the components to communicate with each other, thus increasing the average response time immensely. Docker, though lightweight, serves the purpose of virtual machines. Since other architectures that did not utilize docker ran natively, they did not require as many resources from the computer. The microservice architecture, however, leveraged docker and needed many more resources from the computer and, as a result, slowed down the performance of the application in Scenario 1. Since Docker utilizes so many computer resources, it does not seem that it is likely to speed up an application.

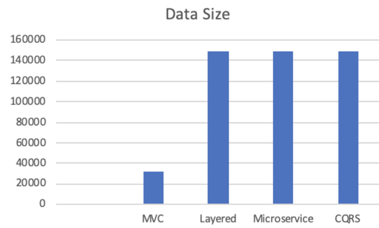
MVC performed considerably well compared to the other architectures in the read heavy scenario. This potentially could be influenced by the high coupling of the architecture. Although MVC emphasizes separation of concerns between the model, view, and controller, the application as a whole is modulated tightly. The controller directly sends function calls to the model and template calls to the view, limiting the need for expensive network calls. In a client server model, as opposed to MVC, the controller is not as tightly coupled with the other components. Communication requires the use of HTTP requests, data parsing, and frequent cycles to the database. This also lends an explanation for the poor response time of the layered/CQRS architecture when compared with MVC. In the layered pattern, communication between the business logic, the persistence layer, and the presentation layer is done through HTTP calls to each other. Since each one is a web server, the only method of communication is the HTTP protocol. While MVC uses one process for the entire application, the layered pattern required three separate servers each passing JSON data to each other. MVC has an internal communication method that doesn't require cross server requests.

One interesting finding is that MVC performed relatively poorly in Scenario 2 and Scenario 3, both of which have writes to the database. MVC, specifically the Ruby on Rails framework, generates SQL queries through a class called the ActiveRecord [8]. Query optimization is more difficult due to this layer above each model. The database access layer in the layered architecture used a Python module called Peewee which is more lightweight as opposed to ActiveRecord.

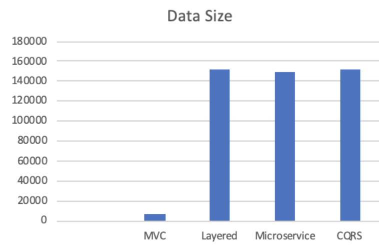
B. Performance vs Data



Scenario 1



Scenario 2



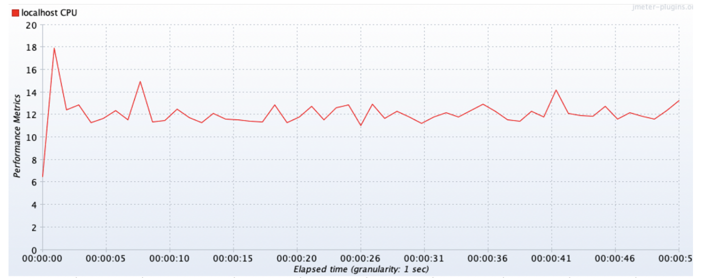
Scenario 3

In all three scenarios, the MVC architecture consistently preformed the best by far. The gap in performance is due to the fact that the team developed the applications for the other architectures using ReactJS which inherently sends back large amounts of data to the browser because it has many components to manage and has to package all of the JavaScript files whenever it sends information back. A different, lighter framework like PHP or AJAX could have been leveraged, but the performance benefits from such a framework would still not bring the architectures anywhere near the performance of the MVC architecture. Had the application been developed strictly HTML and CSS, it would have been much faster, but most real life applications use a framework like ReactJS. The team wanted to model a practical web application instead of building an impractical product that had the sole purpose of performing well against the team's metrics.

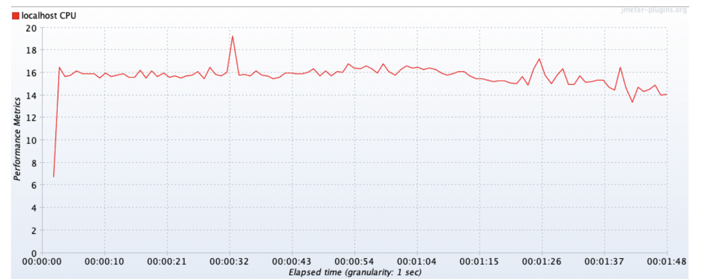
MVC, on the other hand, uses templates and creates the HTML before sending data back to the browser. As a result, since it does not need JavaScript to keep track of things like application state, opposite to ReactJS, it is able to send back an HTML page which is more lightweight and requires much less data than its ReactJS counterpart in this case. In this test, the performance benefits of an MVC application shine through, but that does not provide conclusive evidence that MVC as an architecture is superior to the other three. It simply shows that the applications built in this case suffered in performance because they incorporated a framework that MVC did not.

C. Architecture vs CPU Usage

The performance of the architectures are summarized in the chart below[1]:



MVC



Layered

Looking at the two graphs, it is evident that the MVC architecture had a smaller CPU usage throughout the length of the tests than the layered architecture. The layered architecture is generally hovering around 16% of the CPU while the MVC architecture initially is at 18%, it then normalizes and stays around 12% for the duration of the time. The layered architecture depends on a decoupled system of primarily independent servers, which quickly increases the burden on the CPU. [1] In the application that was created by the team, the layered architecture had four layers and was built by using the ReactJS framework, which relies on HTTP requests and communication with server components. These additional HTTP requests add to the increased CPU usage.

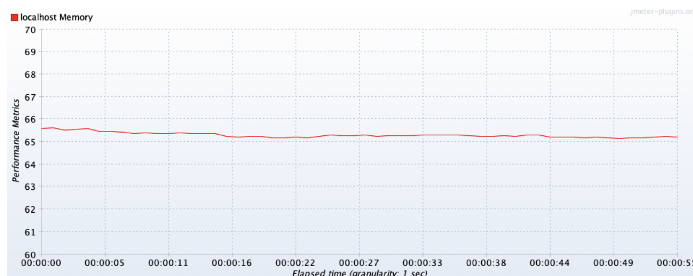
For the MVC framework on the other hand there are some attributes that helped the architecture have a lower CPU usage. MVC allows for the models to work directly with the database. MVC also utilizes templates and does not need the JavaScript as much as ReactJS does. The MVC architecture for the project was built using the Ruby on Rails, which is a more

[1] Measurements for the Layered architecture are representative of the Microservices and CQRS architectures

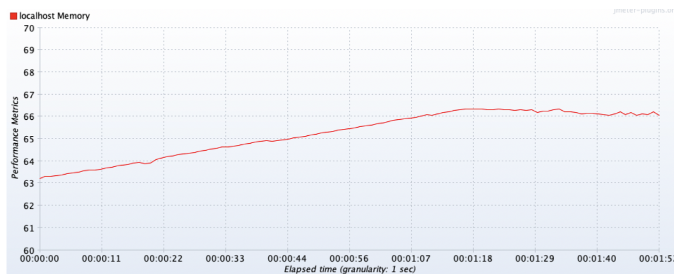
coupled framework than the layered architecture. There is less communication across multiple servers and the request handling system based on user actions flows directly from one part of the framework to the other since coupling is high. [9] Overall, one cannot say definitively that MVC is the superior architecture for every scenario. Another factor that heavily impacts the architecture's CPU usage is the constant accessing and utilization of the database. In any given instance the database handling is the bottleneck for the architectures as there are many requests that have to be handled.

D. Architecture vs Memory Usage

The performance of the architectures are summarized in the chart below[1]:



MVC



Layered

In the graphs of the memory usage of each architecture, it can be seen that both architecture patterns level out around 66%. The memory usage for the MVC architecture levels out at approximately 65.5% while the memory usage for the Layered architecture levels out at approximately 66%. One potential reason for both architectures using nearly the same percentage of memory could be the fact that the database access is often the bottle neck of many processes. So, since both architectures are accessing the same database, which in the current case is stored in the computer's local memory, both are showing approximately the same amount of memory usage. The database, which is approximately 5GB in size, puts stress on the memory usage of the computer. When load testing was performed, the stress was amplified and a majority of the computer's memory resources were required to perform the operations. Since the test only considered a "read" scenario, it was expected that all of architectures would have a similar percent of memory usage. If the test was changed to a "write" scenario and the databases were stored on separate machines, it would have been expected that the CQRS architecture saw better performance in terms of memory usage than the other architectures. In that case, the machine would experience a lighter load from the memory access and could still perform reads while the writes were happening. Both databases for the CQRS architecture, however, were stored on the same machine. For this reason the team was not able to test the scenario. Another obstacle that arose from testing the architectures on one machine was the limitations in testing ability. Due to relatively small amount of resources available on the single machine, the team was unable to perform load testing of more than 100 threads. Therefore, it was assumed that the results seen in the memory usage graphs were comparable to what would have been seen in a production environment, but that cannot be confirmed without testing the applications in an actual production environment.

V. DISCUSSION

One of the key ideas extracted from the gathered data is the concept of coupling. Particular to reads, MVC (which has highly coupled modules) responded noticeably faster to client requests. In the layered architecture, the response time was longer due to network latency from multiple HTTP requests across servers. As a result, the first consideration when choosing an architectural pattern should be the network latency. More services and servers that deliver content also means more layers to pass through to get to the client. One business request could require multiple sub requests [1]. If one were to use layered architecture, increasing response time is also possible through load balancing and replicating each of the servers at every layer. Although this would require more computational resources, the lower load on each individual server decreases, so the client will receive a response faster.

Another key idea gathered is the choice of technology and its usage. For the microservice, layered, and CQRS, the response time could have been different if they utilized different frameworks. In this research, the team's implementation involved ReactJS, ExpressJS, Flask, and SQLITE. Each of these frameworks have benefits but cause additional overhead for simpler applications like a textbook catalog. For example, ReactJS comes with state management which can speed up many business processes by allowing the client itself to bear the load of computation in dynamic websites [10]. Flask and ExpressJS have multi-threading and asynchronous processing features [11]. In MVC, the N+1 query problem could be solved by eager loading ActiveRecord associations [13]. MVC also provides profilers which report on latencies in different areas of the code in order to allow the developer identify bottlenecks. Databases, which are typically the bottlenecks in a CRUD heavy application, also provide features such as indexing, sharding, and scaling [12]. Choosing a database that meets the needs of a website could improve response time as well. Therefore, the second consideration when improving response time is choosing the right technology or framework within an architectural pattern.

Performance is not necessarily a measure of how fast the website responds to the user. It also includes processing load and the managing the health of the website. This requires that a website is not only responding fast, but is also not overloading the processor that could eventually lead to a crash. The results suggest that MVC requires less CPU and memory resources, however, not by much. The layered architecture, while offering the flexibility to choose separate technologies in each layer, actually uses similar resources. A machine (several machines) with high processing power would allow a layered architecture to perform similar to the MVC. Therefore, the third consideration for improving the performance of a website is analyzing how an increase in computational resources could help. This includes both the vertical and horizontal scaling of the application servers and the database.

The final consideration for improving performance is how easily the architecture can be expanded to a distributed system. Distributed systems allow websites to manage the different processes and services of a website through message communication between the different components [1]. Using a messaging system allows for asynchronous processing, separation of jobs, useful redundancy, and scalability. With the rise in data processing, running multiple programs and jobs at different machines helps solve the load problem. The website can potentially respond faster and maintain its availability. In consideration with the architectures discussed in this document, MVC proves to be the most inflexible in allowing for expansion to distributed computing. The tight coupling of the MVC makes it difficult to spread the website's services across multiple machines outside of simple replication. CQRS is an example of a simple distributed system. By separating the writes and reads, the load is partitioned across different machines. Communication is maintained between the read store and the write store through an event messaging bus. CQRS is not a method of increasing the computational resources, rather it is an effective method of separating work into different components.

A. *Summary of Analysis*

Consider the following factors when deciding on an architectural pattern:

- Does the architecture have high network latency? Could this be solved using methods such as caching, CDN's, load balancing, and prefetching?
- Does the architecture have different options for technologies that implement it? If so, one can weigh the options to suit the type of website being developed. For example, a real time application might require multithreading and a technology which offers that could help.
- Can the architecture respond well to more computational resources? Does it allow the database to scale vertically or horizontally?
- Can the architecture adapt well to distributed computing if needed?

VI. Conclusions and Improvements

The project could be improved for the next iteration in several ways. The first is to choose an application that is much broader to allow us to test several architectures thoroughly without having to make many assumptions. Although the analysis primarily focuses on MVC, Microservice, CQRS, and layered architectures there exist many other services that could be deemed beneficial for different scenarios. With the team deciding to implement a textbook ordering application, the types of architectures that were feasible for the program also differed so some patterns were not considered. Another improvement is to expand the testing environment used to test the architectures. Having a testing environment that would be able to handle a variety of tests and handle many threads could reveal new information. A proper testing infrastructure would allow the team to test thousands of concurrent users in isolation.

In a real software development setting there can be many other factors that can determine the successful implementation of a software pattern. These include the following: agility, ease of deployment, scalability, testability, and ease of development [1]. One should keep in mind the technologies a team has access to, the type of coding knowledge the team has, and the adaptability of the pattern. These are some of the many outlying factors that exist in choosing a successful architecture. The team has gained valuable knowledge not only about software architectural patterns, but also about general engineering research. In the future, the goal is to expand this project further to other architectures, other metrics, and technologies.

References

[1] M. Richards, *Software Architecture Patterns*. Sebastopol, CA: O'Reilly Media, Inc., 2015.

[2] : Abdul Majeed, Ibtisam Rauf. *MVC Architecture: A Detailed Insight to the Modern Web Applications Development* Peer Rev J Sol Photoen Sys

[3] dragon119, "CQRS pattern - Azure Architecture Center," CQRS pattern - Azure Architecture Center — Microsoft Docs. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs. [Accessed: 27-Apr-2020].

[4] J. Hall, "Council Post: Speed Matters: How Your Website's Page Speed Can Affect Your Marketing Efforts" *Forbes*, 14-May-2019. Available: https://www.forbes.com/sites/forbesagencycouncil/2019/05/14/speed-matters-how-your-websites-page-speed-can-affect-your-marketing-efforts. Accessed: 27-Apr-2020

[5] "What is JMeter? Introduction and Uses" *Guru99*. Available: https://www.guru99.com/introduction-to-jmeter.html. Accessed: 27-Apr-2020

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, pp. 76–91, Mar. 2014.

[8] Urbanek, "Fix Slow Active Record SQL Queries in Rails" Paweł U. — Ruby on Rails Web Development Consultant Full Stack Blog, 12-Mar-2018. Online. Available: https://pawelurbanek.com/slow-rails-queries. Accessed: 27-Apr-2020.

[9] A. Majeed and I. Rauf, "MVC Architecture: A Detailed Insight to the Modern Web Applications Development," Peer Review Journal of Solar & Photoenergy Systems, 26-Sep-2018. [Online]. Available: https://crimsonpublishers.com/prsp/fulltext/PRSP.000505.php. [Accessed: 26-Apr-2020].

[10] "React – A JavaScript library for building user interfaces," – A JavaScript library for building user interfaces. [Online]. Available: https://reactjs.org/. [Accessed: 27-Apr-2020].
[11] "Welcome to AIOHTTP," Welcome to AIOHTTP - aiohttp 3.6.2 documentation. [Online]. Available: https://docs.aiohttp.org/en/stable/. [Accessed: 28-Apr-2020].
[12] "Database Bottlenecks: The Hidden Cause of App Slow Downs?," *DevOps.com*, 02-Mar-2020. [Online]. Available: https://devops.com/database-bottlenecks-hidden-cause-app-slow-downs/. [Accessed: 28-Apr-2020].
[13] "Active Record Query Interface." Ruby on Rails Guides, guides.rubyonrails.org/active_record_querying.html.