Mohammad Darabi and Patrick Vuscan

Prof. Ala Shaabana

CSC301

Assignment 1

04 October 2020

Recommendation of a Shopping App

The purpose of this report is to define an effective technology stack for the development of a simple shopping and checkout application. This idea must span both a web application, as well as a mobile application.

From the dawn of web and application development, there has always been one common fear for a startup or beginner: what do I use? Flowing from the specifications and the utility of the application, we came up with a good technology stack that prioritizes lean development, structure, and future proofing. Written in *TypeScript* across the board, and using *ReactJS* with *Material UI* for the front end, with *GraphQL* and a ***POSTGRESQL*** database in the backend, we have come up with a strong combination of technologies which can bring together this project efficiently, while also being mindful of what the future may have in store.

As per the *Stackoverflow 2019 Survey* (Stack Overflow), it is clear that *JavaScript* is the most widely used and known language for web and application development as of now. Holding 1st place, at a 67.8% market share, it is clear that JavaScript will continue to be widely used for years. Furthermore, this language continues to iterate on itself and improve, labeling it as one of most future-proof languages, for the foreseeable future.

However, we chose TypeScript instead of JavaScript. TypeScript is superset of JavaScript, which, as the name might imply, enables the use of strong types. Holding 10th place, at a 21.2% share of the market, it is not quite as renowned as JavaScript, but more than 46.6% of developers could easily switch to it from JavaScript! This implies that there is a majority of readily trained developers who could undertake the continuation and further development of this project, under this framework, if needed. As a result of TypeScript's typing structure, it also provides many features over JavaScript that make developing and debugging easier, resulting in very agile development. Altogether, it should be clear that TypeScript is a wonderful language to base the development of this application on. However, there is still one downside to having a strict type system, which is that developers must write longer programs. So if for example we are to measure the efficiency of a programmer by the number of lines of code he must write to produce a specified feature, with all other things considered equal, he would be able to produce the specified feature much faster if he were to use a non-statically typed language like JavaScript, solely due to him having to write fewer lines of code. One of the reasons why Java for example is not a good language for writing quick and dirty programs to reach a proof of concept, is due to all the boiler plate code required to simply write a "hello world" program. Relative to language such as Python or Javascript, the number of lines of code required to tell the computer to print "hello world" is significantly more. Thus, being well aware of both arguments for, and against having a typed system, we decided to go for Typescript since we believe that as the size of an organization grows, it is more beneficial to have such a type system.

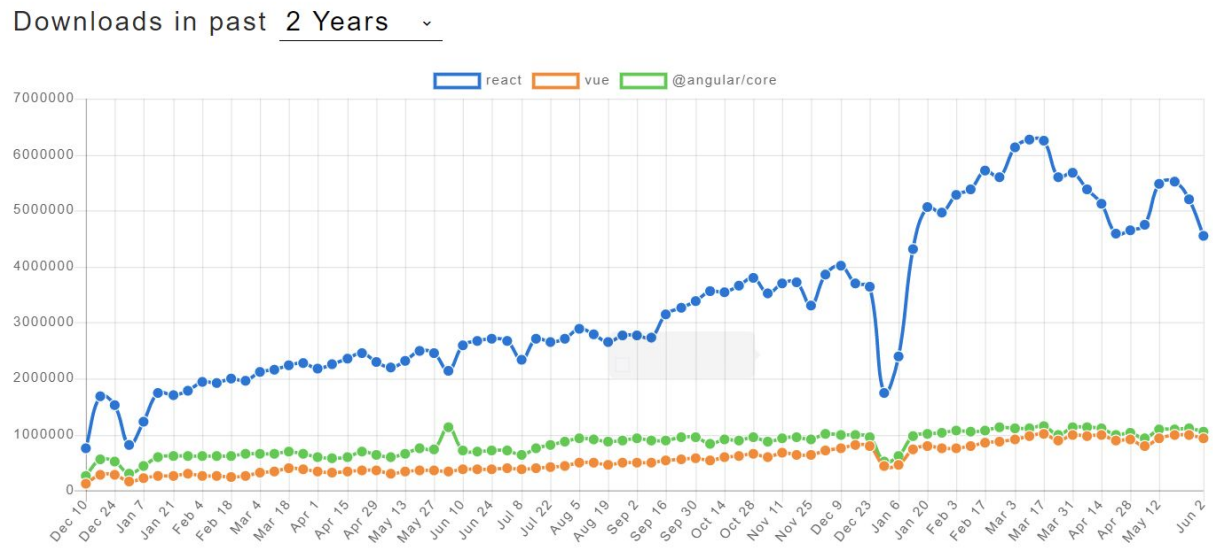
Through the next sections of the report, we will further discuss the individual reasoning of choice for the tools and frameworks we use in the frontend, the backend, and the Continuous Integration/Continuous Deployment of our application (CI/CD).

Frontend

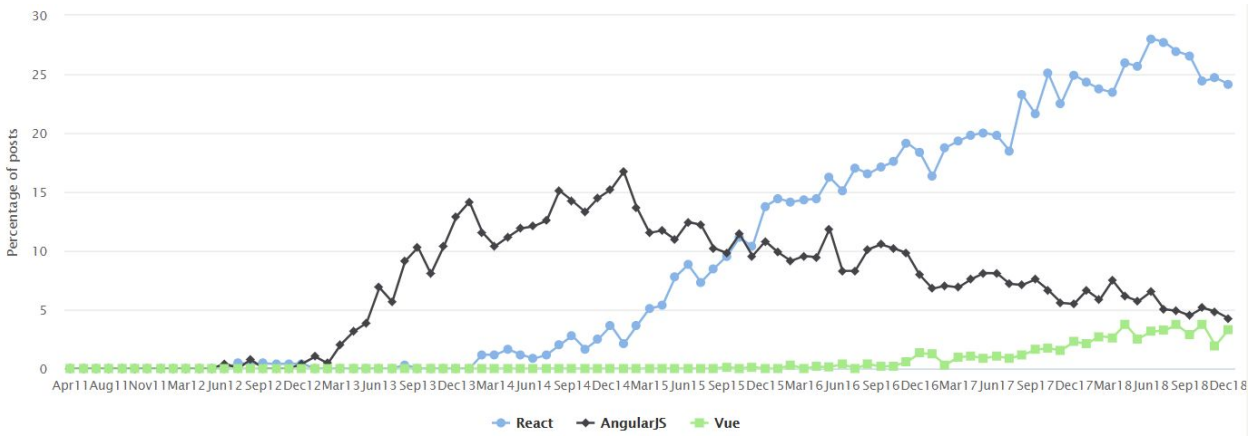
As a result of the specifications we have agreed on, we will not be creating a separate native mobile application. We believe that having a separate mobile application divides attention and resources resulting in a shift of focus away from developing a core product, to just supporting said product natively. Furthermore, for something as simple as a shopping application, we do not find it reasonable to believe that a user will both willingly, and *happily*, download a full-fledged native application just for the purpose of occasional shopping on our platform. The real estate of mobile applications has very much saturated over the years making it very difficult to compete with the top 10 current applications on the market. More than 90% of attention is taken by the top 10 leaving very little for other developers to profit from. After all, even Amazon, which is the biggest online retail shop, is having trouble winning in such an intense market.

For the frontend development of both our web and mobile applications, we will create a responsive web application built with ReactJS, in conjunction with Material UI. This stack took preference over most others we have reviewed, as a result of its wide use throughout the industry, and its general support. As per *2019 Stats on Top JS Frameworks: React, Angular & Vue* (Hlebowitsh), ReactJS is currently the second most popular framework in development, preceded only by *jQuery*, which is slowly being phased out by many companies in favour of more modern frameworks such as React. The reason we chose React over *Angular* and *Vue*, is because of its wide support, as it is developed by Facebook, and the continuous development of

its core features and stability. This is supported by *2019 Stats on Top JS Frameworks: React, Angular & Vue* (Hlebowitsh), where one can see that over the past two years, the number of downloads of ReactJS greatly overshadows those of Vue and Angular:



The number of jobs available in the field with React supersedes that of both:



This further abholds to the principle of using a future-proof framework, one with which future developers will likely be able to continue to advance the application. And lastly, it's wide support and the size of the community around it can be made clear by its *34.2k* stars on GitHub.

These factors combine to make it clear that ReactJS is the strongest candidate for our frontend framework.

Material UI is a UI theming and component library which will be used in conjunction with our ReactJS framework. This library provides countless precreated components and functionality, along with the ability to create an overarching theme which styles the components to look the same across the application! The use of Material UI does not only provide leaner code with fewer custom components, but also results in leaner development and less programming needed. One final strong point for Material UI, is the large community of developers upon which to fall back unto, whenever UI/UX development problems arise, as is the programmer's curse.

The use of ReactJS and Material UI will result in a future-proof and well supported application, upon which future developers will easily be able to continue development, and which can quickly put together a clean and simple application.

Web Application. To further upon what was previously mentioned, this application will use responsive web application principles and practices. Developing with desktop, tablets, and mobile in mind, means that a user will have a clean and well-designed experience regardless of which device they use to access our application.

Mobile Application. As explained at the beginning of this section, we have decided to not have a separate mobile application. However, one point should be made here: if it is ever necessitated to have a native mobile application, whether for Android, iOS, or both, our framework hosts a strong option for mobile development. React Native is a mobile development framework on top of ReactJS, which lets you repurpose large portions of your web application for the mobile

application, with few code changes! We believe that this forward thinking of the capabilities of the framework may further aid in choosing ReactJS over other frameworks. Given that this is no more than a simple checkout system, we will also not need native performance to deliver a solid mobile experience to our users. Having the mobile application rewritten in the same language also enables us to keep hiring overhead as low as possible since the same team of developers that built the web application will also be able to transition to the mobile side. Also, the backend that we have considered building is also very efficient for different user scenarios, and makes no assumption on where the application will be serviced. Hence, for us to expand to a purely mobile environment will require us to simply build a frontend app in React Native. **Continue reading to understand how we are able to deliver such a profound statement.**

Backend

How are we going to build a backend that can support both a mobile app, and a web app at the same time? In order to build API endpoints that can support the huge diversity of devices that will potentially connect to our app, we need to consider flexibility as the centerpiece for our design decisions. An API that requires the user to make multiple requests to simply be able to aggregate all the necessary data to fill out the screen is no longer a viable option. Especially given the huge diversity of users that end up using a given app. There are countries where 4G/5G internet is simply not the case - and even closer to home we find that there at many gaps in our network, where a good internet connection is not accessible. Have you ever got on a train or in the subway where you were writing the final version of your report discussing how you are planning to build your amazing product to your CEO, only to realize that your connection went

down, and the app would no longer register your fine points? Well, those hotspots can also get taken care of if we simply put forth a little extra thought about such situations.

Now let us discuss how we plan to design our API in order to address the issues stated above, starting with the tech stack that will be used to develop the backend for this product. This will include the API server itself, the database service, and the infrastructure that we are going to use to deploy the backend. That is a lot of tech so let us dive in.

API

For the API server we are going to use a *GraphQL* featured server. Given that we are writing the whole app using *Typescript* the server that we are using specifically is *graphql-yoga*. The reason why we have chosen to go with a GraphQL server instead of a *REST API* server is to address some of the concerns discussed in the previous section. By using a GraphQL API we can query only the data that we need by sending one request to the server. The way that this is achieved is the following: instead of exposing many endpoints for all the different entities that we might want to serve through our API, which could lead to the N+1 problem, we expose one endpoint only. By only exposing one end point, we can eliminate the need for users to make more than one request. If we were instead to have used a REST API, we could have also used GraphQL to query the API itself. However, instead of having the N+1 problem on the client side, we alleviate this issue again by making the requests on the server side. Hence, even in this situation, by abstracting all the endpoints away from the client side, we are in a good position to solve the issue with slow networks.

In addition to this, we are using a GraphQL server because of how it enables us to query the database itself. Instead of having to send multiple requests to get data from all the different

entities that are within our database, we can instead use one query that can link to all possible relations that each entity can have.

Database

For the database we are choosing to go with *Postgres*. More specifically a Postgres database that is hosted by *Heroku*. The reason why we have decided to use Postgres instead of alternative SQL databases is because of our personal experience with the Database Management System (DBMS). Furthermore, it has secured a ranking of 4th against other SQL databases for many years as per *Open source threatens to eat the database market* (Asay):

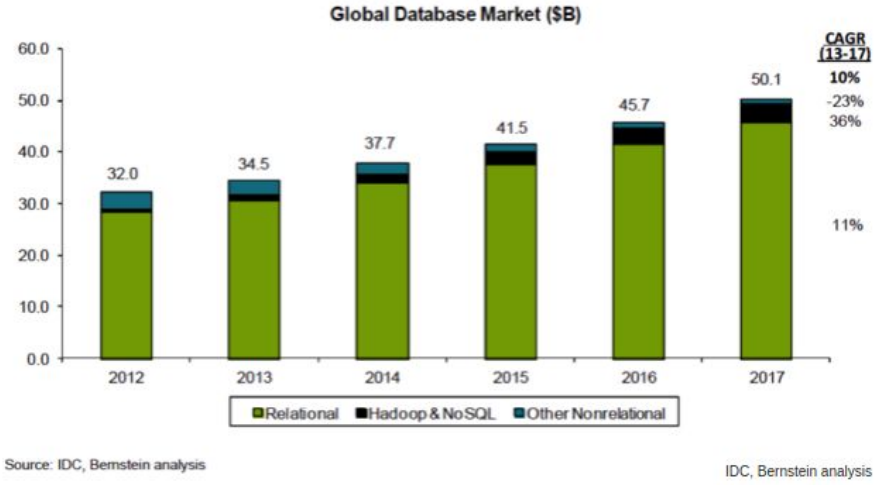
358 systems in ranking, September 2020

Rank			DBMS	Database Model	Score		
Sep 2020	Aug 2020	Sep 2019			Sep 2020	Aug 2020	Sep 2019
1.	1.	1.	Oracle	Relational, Multi-model	1369.36	+14.21	+22.71
2.	2.	2.	MySQL	Relational, Multi-model	1264.25	+2.67	-14.83
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1062.76	-13.12	-22.30
4.	4.	4.	PostgreSQL	Relational, Multi-model	542.29	+5.52	+60.04
5.	5.	5.	MongoDB	Document, Multi-model	446.48	+2.92	+36.42
6.	6.	6.	IBM Db2	Relational, Multi-model	161.24	-1.21	-10.32
7.	7.	↑ 8.	Redis	Key-value, Multi-model	151.86	-1.02	+9.95
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	150.50	-1.82	+1.23
9.	9.	↑ 11.	SQLite	Relational	126.68	-0.14	+3.31
10.	↑ 11.	10.	Cassandra	Wide column	119.18	-0.66	-4.22

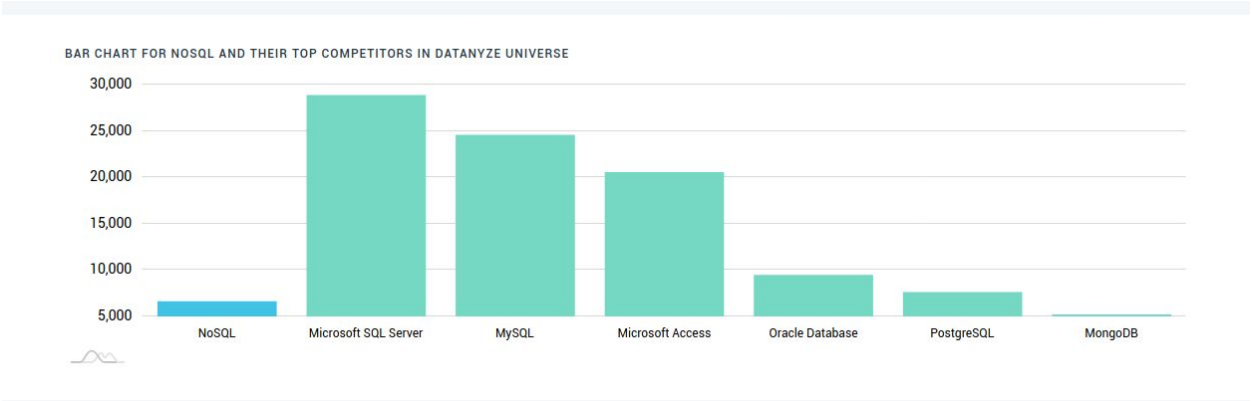
Note that the latest rankings can be seen at https://db-engines.com/en/ranking

Postgres is also far more renown than most other SQL databases, while also being open sourced by Berkeley University. The main selling point is due to it being open sourced, as well as it having a solid history dating back to the first SQL paper that was put forth by IBM in the 1970s - it being a direct descendant of Ingres. As for the reason behind using a SQL database instead of a No-SQL database such as *Mongo*, is mainly because we think that we will be storing relational type entities more than object type entities. Most of the database will consist of an Item type -

and further down the line we have envisioned that we will most likely have relationships between a User type, Order type, and the Item type for a full out online store. Another reason why we are going with SQL is due to the supreme dollar market share it has (Asay):



and the incredible difference in number of websites that use SQL type databases vs No-SQL, as per Datanyze's Market Share statistics of NoSQL (Datanyze):



Furthermore, given the nature of GraphQL's schema first approach we feel that a SQL database would interact much more naturally with GraphQL's schema given that SQL databases are also schema based.

Infrastructure

As for the infrastructure for the deployment of our frontend application as well as for our API we are considering two options. The first options will be to use *Netlify* to deploy the frontend app. Netlify is a very nice hosting platform for JAMSTACK type applications - and it provided us with Continuous Deployment for free. As for the API, we plan to use *Google Cloud Platform (GCP)*. The way that we intend to use GCP, is to first containerize our API using *Docker* and then use the Cloud Run API from Google to deploy the container itself. This will be a simple solution which will facilitate managing requirements, as well as providing scalability for future expansion. As for where we will provision the database, as stated in the previous section, it will be through Heroku, given that it has a sandbox version of the service which would work out perfectly for our first MVP of this product as well as driving down costs. As the database will be written in Postgres there will be no issues moving to GCP platform later down the line if we require further scaling of the app.

Continuous Integration and Continuous Deployment (CI/CD)

Finally, as for the Continuous Integration and Continuous Deployment of our application to the user, we are planning on using GitHub Actions for CI/CD in the backend. Given that Netlify provides us with a free CD, we also plan to use Github Actions for frontend CI. During the CI, we are going to run Unit Tests written using Jest for both the backend and the frontend. We do not intend to run any Integration Tests given that the UI itself is quite simple, so classic manual testing should do the trick. However, if we are to make the UI more feature-packed, we have already considered the Cypress testing framework to be a good option for automated UI testing.

Conclusion

In somma, we can bring our choice of tech stack down to a few core values:

Lean development

The tech stack chosen, aids in writing less code, and quicker development. Namely, Material UI aids in writing fewer components for the frontend and stylizing, TypeScript which brings about faster debugging and development for both frontend and backend, and GraphQL which makes querying easier!

Structure

Using TypeScript helps us define a solid structure for our code, the types of our components and their properties, and a secure backend by ensuring types in all portions of the backend are correctly typed.

Future Proofing

We believe our choice of tech stack has a strong chance of still being relevant and easy to iterate on, years down the line, even in an environment that is constantly subject to fast paced change such as web development. ReactJS, TypeScript, and SQL, are nearly certain to be relevant for many years to come.

It is all these core values that have led us to believe in our chosen tech stack, and we believe that it is the perfect tech stack to use in the development of a simple checkout app as this one.

Works Cited

Asay, Matt. “Open source threatens to eat the database market.” *Infoworld.com*, Infoworld.com, 29 April 2015, https://www.infoworld.com/article/2916057/open-source-threatens-to-eat-the-database-market.html. Accessed 04 Oct 2020.

Datanyze. “NoSQL.” *Datanyze.com*, Datanyze.com, 2020, https://www.datanyze.com/market-share/databases--272/nosql-market-share. Accessed 4 Oct 2020.

Hlebowitsh, Nadia. “2019 Stats on Top JS Frameworks: React, Angular & Vue.” *Tecla.io*, 2020, https://www.tecla.io/blog/2019-stats-on-top-js-frameworks-react-angular-and-vue/. Accessed 4 Oct 2020.

Stack Overflow. “Developer Survey Results 2019.” *Stackoverflow.com*, 2020, https://insights.stackoverflow.com/survey/2019. Accessed 4 Oct 2020.