David Pope

Final Report

Employer: DST

Contact Person/Info: Tony Tubbs

This summer of my internship has been the most unique and busy since I’ve been interning for DST. I began interning for DST as a Missouri Innovation Campus (MIC) intern at the beginning of the summer of 2016. This summer was the beginning of my third year with DST so I have had quite a bit of experience in working in a corporate environment and working on real world projects prior to this summer. Having that experience did not stop me from learning a lot, being placed into new situations, and working with new technologies. All of that combined with the company being bought, laying off 6% of the employees, and reorganizing the structure of the company, made this the most unique summer at DST since I began interning there. For the purposes of this report, I will only focus on this summer of my internship rather than anything that happened prior unless it is necessary or relevant.

**Summer Projects**

Before starting the main project I have been working on this summer, I worked on my own on a conference center kiosk application. DST has two conference centers in their downtown Kansas City offices. Each of these conference centers has a touchscreen kiosk in a central location where people can easily find it. Before working on the application, the kiosks were going unused. The people who work in and manage the conference centers came to my team to see if they could create an application that showed each conference centers. The application would need to show a layout of the conference center as well as other relevant information to the conference centers including the room names, what meeting was currently going on in a room, and the timeframe each meeting was supposed to last. I was eventually tasked with working on this project, mainly on my own, back in December of 2017. I worked on this project up until about the second week of the summer portion of my internship. After I finished working on the project at the beginning of the summer, the project was to go to a group of people who would put the project out into production. Unfortunately, the people who would normally be tasked with doing this were let go, causing the project to go on the backburner for the foreseeable future. The hope is to get that project moving along so that it will soon be used in the conference centers here in Kansas City. If that does eventually happen, it will be the first application that I have worked on at DST to go into production and used in the company. Although the project is not being used, working on the application gave me a lot of experience and an insight in what seeing a project go from start to finish looks like.

This summer began with my team switching its focus to two specific projects. “The Hive,” our first project, is an application that lets people ask software development and programming related questions to anyone else who has an account on the app. The hope for this project is for it to be used in the future for other DST employees and interns if they are ever seeking help, guidance, or advice. Ideally the project will be similar to an internal version of Stack Overflow where the questions would be geared more towards DST technologies and standards. The other project that we started focusing our efforts on, and the one that I have been tasked with working on, is the Pricing Dashboard project. The Pricing Dashboard was something we had worked on in the past. Back in the fall of 2017, the Pricing Team at DST came to my then boss to see if the Research and Development team I was on at the time could create an application for them that would make the task of pricing mutual funds faster and more efficient.

The way the pricers on the Pricing Team currently do their job is inefficient to say the least. As of now, the pricers on the team receive a list of all the new mutual fund prices at a certain time each day at the end of the work day. Once the prices are received, the pricers must manually enter in each new price within the next couple hours so that a list of all the mutual funds and their prices can be sent out to DST’s clients. Right now, the pricers access this mutual fund information using Microsoft Access. This is incredibly inefficient because Microsoft Access currently only allows for two gigabytes of data at a time, while the amount of data the pricers need to access is much larger than two gigabytes. All this extra data causes Microsoft Access to run very slowly and break down occasionally. This forces the pricers to rush to make sure every mutual fund price is entered by the time they need to go out to clients, making it more and more likely for someone to mistype or make a mistake regarding entering in the new prices. This makes using Access not only inefficient, but unsustainable for the future as DST ostensibly adds more clients in the future. The Pricing Team clearly knew this would be a problem and came to us to help them create a better application than Microsoft Access to help better serve the pricers. We began working on the project soon after during the fall of last year. A couple of different mock ups as well as some design ideas were created so that we would have a place to start. After learning a couple new technologies, JavaScript frameworks, and experimenting with different ways to implement graphs into the project, the application was put on the backburner. After a few weeks of working on the project, we learned that the project was not confirmed to be financially supported yet. Instead of continuing the Pricing Dashboard, we decided to switch to projects that were of a higher priority. We continued to work on other projects until earlier this year when we found out that the project would be supported.

**Pricing Dashboard Structure**

On the first version of the Pricing Dashboard project we worked on last year, we solely worked on the front-end of the project using the AngularJS JavaScript framework. When we began working on the project the second time a couple months ago, our team created a structure of how the application would be ran. The basic structure of the application includes an AngularJS based front-end, a Java SpringBoot API, and a MySQL database that stores the data relevant to the application. The front-end of the application will be used to show the information the pricers need to see to do their job, including mutual fund names, prices, rates, and system names. The main page of the application begins with a navigation bar at the top of the screen that contains a drop-down menu. The drop-down menu contains a list of systems. Each system contains a different set of mutual funds. A mutual fund can belong to multiple systems, but each system is unique in what it contains. From this drop-down menu, you can select as many or as few systems as you need. The menu also contains a search functionality, so you can easily find a specific system, as well as a select all functionality if you want to see every system at once. Once a system is selected, a gauge chart appears on the screen showing the current information about the system. The chart contains the prices and rates that have yet to be priced contained within that system. Each of these system charts can be clicked on to navigate to a system details page that delves deeper into information related to that system. On this system details page you can see each mutual fund found in that system and how many funds have yet to be priced in that system. There are also buttons that sit next to each chart that navigate to screens that show information about prices and rates for each system. Once off of the home screen, the navigation bar at the top has a button that lets you navigate back to the home page of the application. This will be the primary format for the front end of the application for the time being. It is likely that as new requirements for the project come in that the system details portion of the app will change, but as of creating this report there is nothing new on regarding that.

The Application Programming Interface, or API, for the Pricing Dashboard project serves as a midpoint between the front-end of the application and the database. The most essential portion of the API is a collection of Java classes that instantiate a private variable for each column in the database. The defined variables are later used as a way to access and query data from the database. Along with these Java classes, each table in the database has its own unique Java interface. These Java interfaces serve as a Java Persistence API, or JPA, repository for each table. JPA is an API that is used to access and manage data that is used between Java objects and relational databases. The JpaRepository that is used in the Java interfaces creates a sort of virtual table within the API portion of the application. The next portion of the API is what connects the pieces altogether: the controller. The controller is a Java class that serves as a place where RESTful calls are defined and created. In the controller, the original Java classes where the variables are stored and the Java JPA repositories come together to get exactly what we will need for the application. Within the controller, functions are defined that are used to query the database and return the exact information that we need. Each function acts as a GET request and returns the information needed for the application. Each of those functions are defined in the table’s repository to be used wherever they need to be in the controller. Connected to this controller is a service that instantiates the objects that store the data that comes from the database.

The third and final part of the application is the MySQL database that stores the data and tables for the application. All of the data is stored on a local database called “Pricing” that anyone on the team can access using the MySQL workbench or any other application that allows access to the database. Within the Pricing table there are three separate tables with different types of data in each. The first table contains global data that manages and stores data that applies to all mutual funds and fund sponsors. The second table contains specific mutual fund data that may not apply to each and every mutual fund. The third table contains pricing data for each mutual fund. All these tables together have different relationships to each other that ensure they are connected. Currently these tables are storing mock data that we were sent by the pricing team so that we could work with and design our application around real data. Towards the end of the project, the application will be working with real time data within the database. Using this mock data ensures that any changes we must make in the future in regard to the code will be minimal to none. This project is particularly exciting to me because it will directly help other people in the company as soon as it is done and put into production. Once this project is completed, it will be the first project I have worked on that has gone into production and the first project that has been for a different team in the company.

**Technologies Used and How They Are Implemented**

The projects I have been able to work on this summer have introduced to many new technologies and languages, as well as reinforced my knowledge on other technologies and languages that I have had previous experience with in school or in past work. Throughout my internship, I have been able to learn a wide array of technologies and programming frameworks that are commonly used in the real world. Several of these technologies I had never heard of until beginning my internship. Deepening my knowledge in those technologies will be extremely useful in the future as it will make me a more valuable employee and asset to my future employers.

**Kiosk**

The Kiosk application introduced me to quite a few new technologies. The main portion of the application that I worked on was coded in Java which I had quite a bit of experience in going into the project. The front-end and user interface of the application used HTML and CSS, as well as JavaScript that connected the HTML front-end with the Java back-end. The main technology that the Kiosk application introduced to me was the Spring Framework. The Spring application framework provides an easy to implement way to structure and run a Java based application. The specific portion of the Spring Framework I used was Spring Boot. Spring Boot’s main functionality is that it makes it easy to create and run Spring applications and have them start running almost immediately. Spring Boot allows you to use different Java class annotations so that when the application is run, it is easy to follow and see the runtime logic of the application. For instance, above the class containing main inside it, a “@SpringBootApplication” is placed that tells the application where to begin running. Inside the main, there is also a “SpringApplication.run()[” command that begins running the other portions of the application. Once that command begins to run, the application looks for a class with the annotation “@RestController](mailto:)” that tells the application where to continue running at. Other annotations that are used within the Kiosk application include “@Service, @PostMapping, and @RequestBody.” To build the application, the Spring Framework requires either Maven or Gradle. For the Kiosk application, we decided to use Gradle. I had never used Gradle prior to working on the Kiosk app. Gradle is able to work with Spring to create an easy to both and build and run a Java based application. The most essential part of this is a build.gradle file that is used to define several things including how the .jar file is supposed to be built, what dependencies and external APIs are to be included, and any other plugins that are needed to provide all of the functions needed to run the application. In the build.gradle for the Kiosk app, I included the repository link for the version of Spring Boot I needed as well as the link for the Exchange Web Services API that I needed to access meeting and room information for the conference centers. The Exchange Web Services API or EWS, is used to provide an interface for interacting with Microsoft applications that use EWS such as Office 365 or Microsoft Outlook. For the purposes of the Kiosk application, I used the EWS API to connect to Outlook. Each room in the conference centers has an email address attached to it that allows easy access to information about the rooms and meetings.

To access all of the meetings in every room of the conference center, I first added all of the rooms’ email addresses to a properties file. Inside the main Java service used for the application, the values are called from the properties file and stored inside of a private list. I also got to learn how encryption works within a Java application. To access the meeting and conference center information, I needed to log in to an administrative email account that had access to the lower level room-specific email accounts. The main issue was that we did not want to have a username and password to an administrative email account hardcoded into the application. To solve this, I first put both the username and password into an external file. Then I needed to encrypt the password. I found a tool called Jasypt that integrates with Spring based applications that makes it easy to encrypt and decrypt within the app. I added the Jasypt dependency to the Gradle build file and made a function that passes in the encrypted password into the Kiosk application and decrypts into a new variable in the app. Once the account credentials were pulled into the application, the Exchange API finally came into use. I created an “ExchangeService” that used both the account credentials needed to login, and a URI that pointed to the location of Exchange on DST’s server. Once that service had a set URL, it was able to establish a connection to the admin account containing all of the conference center info I would need. I created another function that takes in several parameters such as the date, email account, Exchange Service, and room name, and pulls all of the meetings currently going on in each room. Each meeting is stored inside of an ArrayList and then afterwards put into an object that is accessed by the front-end of the application. The function was called within a “for each” loop that ran each time it found a new Exchange account under the administrator account.

The front-end part of the application only included a couple of different screens. The home screen of the application opens to give you an option of which conference center you want to see the meetings for. Once a conference center is selected, a map of the conference center created with Adobe Animate is loaded on the screen. If the room has a meeting currently going on in it, the room will be animated into a 3D shape of the room in a color that sets it apart from the rest of the map. Next to each room there is a line that goes from each room to a div next to the map that displays all the information about the room and meeting. It contains the room name, the meeting name, and the beginning and end time of the meeting. The top of the screen contains a couple of different things you can do to change what happens with the application. Currently, the app is set up to refresh on its own every 15 minutes to check if there are new meetings or if meetings have ended. One of the things on the conference center page is an automatic refresh button just as a last measure in case the application gets stuck. The refresh button calls a “window.location.reload()” to make everything load back in again. The other thing you can change is the rate at which the application refreshes automatically. A user can type in any number of minutes that they want and the app will begin refreshing at the new time interval. The large majority of the front-end is comprised of JavaScript that provides all of the functionality and animation for the application. The HTML is primarily only used as a way to structure everything rather than give any functionality to the application. CSS also plays a large part in the animation and styling in the app. I put this project onto DST’s internal GitLab where anyone on the team would have access to it, as well as have the ability to make changes if they needed to. The JavaScript based front-end, and the Spring Framework, Java based back-end, created an easy to run, easy to follow conference center application that will hopefully be fully implemented in the future. Although, I have not been able to see the application used at DST yet, it is still the first application that I have worked on all the way through to completion, as well as the one that I have learned the most from.

**Pricing Dashboard**

The Pricing Dashboard application that I have been working on the past two months has been the most in depth project I have worked on thus far. It has incorporated the most technologies and components to run the project. Each major part of the application contains several different technologies that work together to create a fully fleshed out app. This has also been the project that has introduced to the most kinds of software development technologies, even in the relatively short span I have been working on it. The first major part of the application is the front-end, comprised mainly with AngularJS. Angular as it is usually called, is a web application platform mainly based around a language called TypeScript and the NodeJS framework. Typescript is a language very similar to JavaScript in its syntax and functionalities with a few additional capabilities in it. NodeJS is a JavaScript runtime environment that executes JavaScript on the server side rather than the client side. Angular projects, including the Pricing Dashboard project, are typically structured with Angular components. Components are comprised of four different files. The first two are a basic HTML and CSS file, typical of most web based applications. The other two files are TypeScript files. One file is a spec file that usually goes unedited, and the other is where the functionalities of the component are established in. Angular also comes with a global index.html and styles.css files that apply to the entire project rather than a specific component. Angular apps also contain a file called app.module.ts where components are declared for use in the application. It is also where routing is established for the app, one of the most important things used in Angular applications.

The Pricing Dashboard app starts with the dashboard component. The dashboard component contains the main starting screen for the application. To create a nicer design for the app, I decided to implement Angular Material. Material provides several UI components that make it easy to both structure and design a web based application with consistent styling throughout. At the very top of the screen, I created a toolbar using Material that contains the title, the main dropdown menu, and a hamburger menu that provides navigation for the application. At the start of the app, the toolbar is the only thing shown on screen. The dropdown menu provides further navigation for the app. The dropdown menu is made with a multiselect tag provided by Angular. When clicked on, a list of systems is displayed that can be selected to show more information related to the system on the screen. Along with the list of systems is a search functionality if you are looking for a specific system in the list. There is also a select all option to show all of the systems and their information at the same time. Whenever a system is selected, a div shows up on the screen that shows a gauge chart. These charts are created with a JavaScript library called D3. D3 is used to display data in easy to follow, easy to create ways in any JavaScript or TypeScript based applications. For the purposes of our application, we used an “Ngx-charts-gauge” to display our data. The first gauge shows the percentage of prices for funds in the system that have yet to be priced. The second gauge shows the same thing but for rates in that system. The gauge charts contain many different built-in options that let you customize the chart to your liking. Some of the options that I used in the application include sizing, including a legend, showing the axis, and how many degrees around the center you want the chart to be. There is also an option to change the color scheme. In the TypeScript file for the dashboard component, there is a function that will change the color of the gauge depending on how high or low the percentage of unpriced funds is. If there are a high percentage of funds yet to be priced or given rates, the gauge will become redder. If there are a low percentage of funds yet to be priced or given rates, the gauge will become greener. That function returns a value for each gauge determining what color it will end up being.

In the middle of the gauge there are two buttons. The first button shows the percentage of funds that have yet to be priced. This button navigates to a system-specific prices page that shows more in-depth information regarding that system, the funds, and the fund sponsors and how the prices look for each of them. The second button is similar except that it shows that same percentage but for rates. That button navigates to a system-specific rates page. Currently, we do not know what will be on the rates page as we need more information from the pricing team on what they would want there. For now, the rates page acts a placeholder for what will come in the future. The prices page also is not fleshed out yet but will be in the coming weeks as we do now know what will be needed to be shown there. If someone using the application wanted to look at information that was specific to the system but did not involve the prices and rates belonging to the system, they could also click on the div itself to navigate to the system details screen or component. The system details page shows a list of fund sponsors that belong to that system, (each fund sponsor contains multiple funds). This page contains one single “Ngx-charts-gauge” that represents the entire system. Each gauge represents one fund sponsor, and the percentage shown in the gauge is the amount of funds that have not been priced for each sponsor. Once again, the color scheme for the gauges is the same as it is in the dashboard component.

The application also contains several Angular based services. The purpose of these services it to essentially get the data from either a mock file within the Angular part of the application, or from the API’s provided endpoints. I first got these services working using mock data from a JSON file I created in the project. This mock file contained system names as well as a price and rate percentage attached to each system. Next to the mock file there is a JavaScript file that establishes an endpoint for the service to look for to grab the mock data from. The service contains a function to return the systems from the mock data. The services in the application are set up to work with both the mock data and the real data from the API. The node package manager that is used to run the front-end has two main start commands: “start” and “start-api.” The start command does not look for any API endpoints to interact with. When the service functions are run, it will look for the mock JSON file to get data from. The start-api command uses an “api.proxy.conf.json” file to look for endpoints in the API that match up to the ones listed in that config JSON file. This makes it easy to run and test the front-end application without having the API functioning or running all the time.

Most of the technologies and languages used for the API and the database, I discussed earlier in the structure portion of this report. This is simply because the API and database are used to provide structure for the application. The database is created with SQL, which is used to design and manage all the data needed for the application. The database has different columns that are used often throughout the app including system names, fund prices, and fund names. The API is used to communicate to the database for the front-end of the app. The API is based in Java and built with Maven. Maven is a build automation tool that makes it easy to create and build a Java application with all the resources and dependencies needed. Within the API, I also included some Java test classes that use JUnit. JUnit is used to test as you develop. For the purposes of the API, I used JUnit to test different queries to the database. If I was able to get a query to work properly in the JUnit class, I would then implement it into a service in the API and create an endpoint of it into a controller. With all these components connected to each other, and all these technologies and languages working together within their respective components, an efficiently run and nice-looking application is created in the end. The Pricing Dashboard works has come together well already and should be even greater as it is worked on more and more into the future.

**Problems I Faced**

This summer of my internship provided some unique situations and problems that I had to work through. A combination of starting a large new project and the company I work for being purchased by a larger company offered some new challenges to myself that I had to traverse and work through. The main challenge that not only I faced, but many other interns and employees also faced was the reorganization of the company at the end of June. DST was purchased by SS&C, a company based out of Connecticut that does similar work to DST, towards the beginning of this year. Up until June, nothing had changed within the company regarding the operations or organization. Earlier in June, it was announced that there would a reorganization coming to the company by the end of June. The last Tuesday in June, I and all the other interns currently working met at a meeting where we were told DST would be laying off about 6% of the company. We were also told that many of us would have different managers to report to. My current manager ended up taking a different position within the company and I was given a new manager to report to. Luckily for me and the other interns in my group, our new manager was someone we already knew from working with her in the past. In the next few days following the reorganization, I met with her a couple times to get acclimated with how the new group would work and what projects we would be working on. Fortunately, all the projects my team and I were working on were to continue under the management and organization. For a brief period we were unsure if we were going to be able to continue the projects we were working on so learning that we would be continuing those projects was quite a relief. Although the company’s reorganization did not have a huge effect on me or any of the projects I was working on, it was still interesting to see what a reduction on a large scale looks like as I had never experienced something like that before.

Another big problem that I faced during the summer was constructing a database and importing the data into it for the Pricing Dashboard application. Once the application is complete, it will ideally connect to the mainframe to get real data for the application. Since the current version of the project we’re creating is just a prototype, we wanted to get some sample data so that we would have something realistic to use to populate the application. My team collaborated with some people on the Data Platform team to get some sample data that we could use for the application. They sent us some sample data that was connected to each other between the three tables needed for the app. The sample data was sent to us as .csv files that did not have any of the properties of an SQL database. To get all this data into a database that we could use, access, and query, I needed to construct a database. Other than working with SQL in homework assignments at school, I had never used SQL for a large-scale project or in a work environment. I had to not only refresh my knowledge of SQL, but I also had to learn some new parts of SQL that I had not used before. I looked at each table and each column and defined what their name would be and what data type they would be. I also had to define which columns would be ids for the table. Once I had all the tables created and structured properly, I had to import the data from the .csv files into their respective tables. To access the database and tables, I used the MySQL workbench tool and established a connection to the database. Once the database is accessed, workbench has a “Table Import Wizard” that makes it easy to import data from something like a .csv file. This process took a lot of trial and error to make sure the data types that were defined for each column matched up to the data in the .csv files. Once all the data was imported into our database, it was easy to continue developing the application with real data like what we will be using at the end of the app’s creation.

After the database was up and running, the next roadblock I had to get across was creating the correct queries in the API for grabbing the data. All the data that we needed spanned across all three of the tables in the database. This made grabbing the data that we needed for the application a little bit more difficult than just selecting a column from a table. For the queries that we needed for the app, there was much more detail and instructions needed. For instance, the main information we wanted to get was a list of the system names, all the fund sponsors and funds under those systems, and the price attached to each fund. This requires a query with multiple inner joins and where statements to get exactly the information that we need. To test these queries, I used JUnit to create test classes that would run on their own to try and find the right query we needed. For the API, we used the Java Persistence API to query the database. JPA allows you to create a method name that functions like a query. When you give that method the proper name as it relates to the query you want to use, it will return the data as if it was just a regular SQL query. Most of the testing that was done involved changing that method name to get it exactly right. Once I had the query set up to run the way I wanted it to, I implemented it into a service in the API. To test it further after this, I used a tool called Postman. For each of the queries I used, I gave it an endpoint in the controller file in the API. Postman allows you to enter that URL using that endpoint to see what will be returned when it is called. It helped me to make sure every column and piece of data I needed was being returned. Getting those methods/queries working properly was essential to populating the front-end of the application and will be useful for the rest of the time we develop the application.

**Role for the Summer**