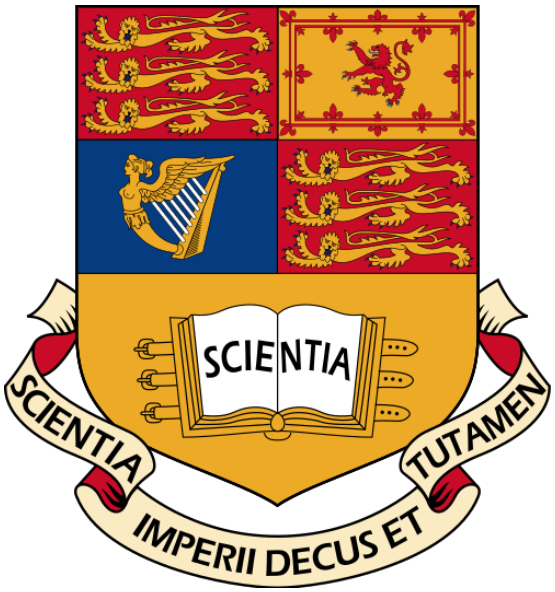

Morgan Stanley

Four-Month Industrial Placement at Morgan Stanley

Vaibhav Krishnakumar
MEng Mathematics and Computer Science

Tutor: Thomas Heinis

Manager: Ganesan Thangamuthu
Email: ganesan.thangamuthu@morganstanley.com
Special thanks to Ivaylo Varbanov

Summer 2017

Introduction

My placement in the summer between Years Three and Four was undertaken with Morgan Stanley, an international organisation with headquarters in New York. I worked in their London office for four months as an Application Developer in the Technology division. The placement is part of the Technology Summer Analyst Program, a scheme lasting between ten weeks and a year, that is designed for students to understand how technology fits in a bank and the specific work undertaken at Morgan Stanley [1]. I was one of forty-five interns in the Technology division; there were four others on the program from the Imperial DoC courses.

Morgan Stanley

Morgan Stanley (MS) is a global financial services provider, with clients ranging from governments and large institutions to high net worth individuals. It provides investment banking, wealth management and securities sales and trading services [2]. It acts as a link between parties interested in generating capital and investors looking for returns on their earnings.

The firm was founded in 1935 as a result of a split in services offered by J.P. Morgan & Co. It has gone through several mergers, acquisitions and rebrandings in its 80 year history; most notably, Mitsubishi Financial Group, Japan's largest bank, invested $9bn during the financial crisis in 2008 giving it a 22% stake in the firm. Other notable owners of MS shares include China Investment Corporation and BlackRock [3].

MS has been a market leader in its association with the Technology industry. One of the services offered by MS is that of underwriter in initial public offerings (IPOs), and the firm has acted as such for a number of the largest technology companies in recent years [4] - these include Google (2004), Facebook (2012) and most recently Snap (March 2017). Furthermore, MS acted as advisor to WhatsApp when it was bought by Facebook for $19bn.

Technology at MS

Technology underpins all services offered by MS. It makes up 20% of the workforce, employing nearly 10,000 people globally. In general, the division is not client facing - almost all users of applications developed by the division are internal to the firm, resulting in frequent communication between app users and developers. While the division is not a revenue-generating arm of the bank, MS recognises the importance of technology to provide both the firm and its clients a cutting-edge.

My particular team during the placement was the Corporate Funding Technology (CFT) team which is split across offices in India, London and New York. We support the Corporate Treasury (CORPTSY) team in the front-office - their core functions include funding the firm and managing the amount of liquidity held at any given time. The team is further split into subteams that are responsible for the different functions.

Treasury Technology

My sub-team in TreasuryTech largely works on Liquidity Risk monitoring and reporting. At a high level, this involves ensuring that the firm has sufficient cash and unencumbered assets to finance its day-to-day operations while being protected from the risk this poses. My sub-team was made up of five people - while initially working on different projects, towards the end of my placement the team was preparing to work on a new regulatory report that comes into force in March 2018.

Liquidity Risk

Liquidity can be defined as the ease of converting an asset to cash. By definition, cash is the most liquid form of capital. Examples of highly liquid assets include stocks and shares, bonds and mutual funds. On the contrary, assets such as property or real estate are harder to convert to cash and hence considered illiquid.

Banks such as MS want to hold minimal amount of cash and liquid assets to operate. Cash is expensive to "store" in bank accounts; moreover, using the cash allows more capital to be available for investment, leading to higher returns. However, this poses an increased risk of not being able to cover costs in the short-term, especially in case of stressed market situations. To mitigate this risk, regulatory authorities require banks to hold a certain amount of liquid assets at all times. This trade-off needs constant monitoring and forms a key part of the roles of CORPTSY and TreasuryTech.

Regulatory Reporting

With increased regulations on banks since 2008, there are numerous reports that must be submitted at varying frequencies to the regulatory bodies in the UK and US. In the UK, the Prudential Regulation Authority (PRA), a sub-organisation of the Bank of England is in charge of these. As MS is a US-registered bank, it is also subject to the regulations issued by the Federal Reserve in Washington D.C.

Part of the role of my team is to calculate and produce these reports, which are verified by the Liquidity Financial Controllers (LFC) at the bank and eventually submitted to the regulators. The major UK reports to be currently submitted are the Liquidity Coverage Ratio (LCR) and the Additional Liquidity Monitoring Metrics (ALMM). They form a comprehensive set of metrics that take into account all operations of the bank and process huge amounts of data.

Project: Building a Proof and Control RulesEngine

During my placement, I was given a project of my own to develop, with help from the team. This was a green-field project, with no existing codebase and some flexibility to design and implement my own solution to the problem at hand. Before discussing this in detail, I will provide a brief overview of the current workflow for the generation of these reports. In particular, I will focus on data collection, which is the first stage of in the process.

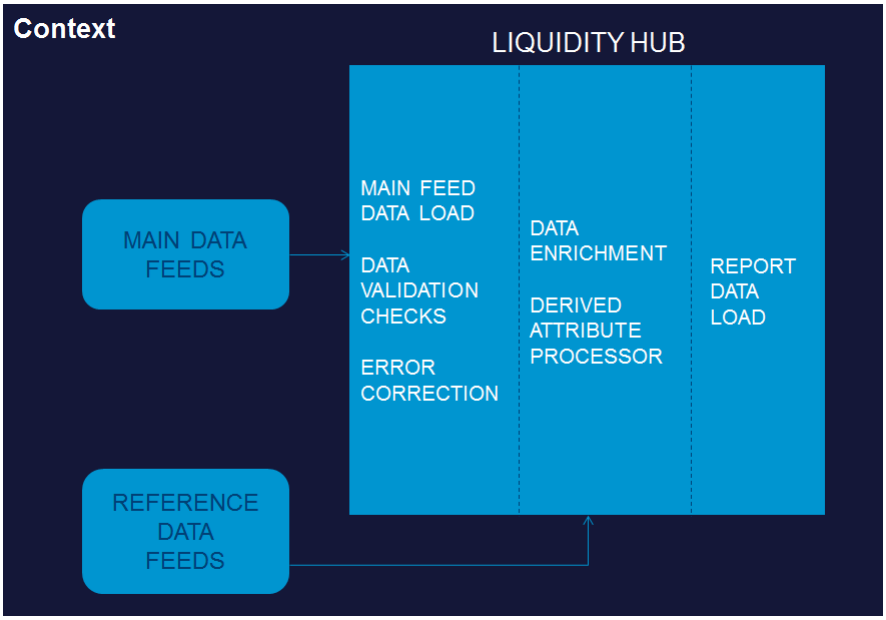

Figure 1: Feed Processing Pipeline

Context

While the process varies from report to report, a general trend is shown in Figure 1. The data collected includes information about cash flow into and out of the bank, the trades executed by different desks, information about specific products traded, exchange rates etc. As can be imagined, this is a lot of data - it is collated into a single store, called the *Liquidity Hub* ("the Hub"). The data arrives in the form of information *Feed*s. The Hub can be considered as a large database - at the time of writing, there are 170 feeds that provide data to the Hub.

Feeds can be largely classified into two types based on the data they provide - they are either used directly in the calculations of liquidity metrics (*Main Feeds*) or to enhance existing data (*Reference Data Feeds*). These feeds arrive in the Hub as large CSV files and are processed and validated before being loaded in database tables. Main Feeds undergo type checks and data validation checks (see below) before being loaded while Reference Data Feeds currently only have type checks.

Once the data is collated and verified, the data enrichment stage can begin. This involves combining Main and Reference Data Feeds to derive attributes, using a predefined logic. It is an automated process that is performed by a Drools-based Rules Engine on every load of a feed. The enriched data is then loaded into "Reporting Schema Tables" in the Hub - these are the tables that are used for all calculations performed with regards to liquidity data.

The calculations are aggregated at different levels to produce the various reports, which are then sent to the Financial Controllers. The controllers check the numbers to ensure that they are valid, often replicating some of the calculations by hand. After sign-off, the data is loaded into templates provided and sent to the regulators. Depending on the report, the process above is carried out daily, weekly or monthly. Since the deadlines to submit to regulators are strict, any delays in the

process could result in major issues for the firm.

Specification

Under the current process, any corruption to Reference Data Feeds are not caught until the reporting stage. The incorrect data is used for data enrichment and calculations before the controllers have a chance to verify and inform the technology team that an error has occurred. As an example, the *Bloomberg* feed, an external Reference Data Feed, had five loading errors over a two-week period - these were caught during report generation, which is too far down the process and is highly likely to delay report submission.

The solution was to introduce validation checks for Reference Data feeds, similar in style to those currently performed on the Main feeds. In addition, the results of these checks were to be displayed on a new tab of an existing web application called *Consolidated Dashboard*. A rule-based approach was used for the validation checks which allows users (i.e. non-technical front-office departments) to define the logic for valid data entries. The rules are to be run on feed load without interfering with the current processing pipeline.

Requirements Gathering

The first stage in the project was requirements gathering - this involves speaking to LFC and evaluating the business need for the project. With the help of my manager and buddy, I set up multiple calls with the users of our application to find out more about the current process that they go through on a day-to-day basis. I also looked at the documentation for the application that performs the data validation checks on the Main data feeds - this gave me a good idea of the kinds of checks that I would need to incorporate in my rules.

One of LFC's daily tasks is to perform *Proof and Control* checks on the generated reports before submission. This involves looking at each number in the final report and tracing its source via calculations in previous steps. A sanity check is performed on these numbers, by looking at variance over previous days and valid range of values. These checks are currently performed manually and can certainly be automated in part.

In addition to this, the users requested a drill-down feature with regards to the errors. It would be less useful for users to simply be told that an error had occurred as a rule had failed. Instead, they would prefer to see a detailed view to pinpoint the reason for the error, going down to the specific account or counterparty in a trade that caused it. In the long term, this would also be useful to look at patterns in errors to prevent further misloads of feeds from happening.

Having gathered these initial requirements from the user, I looked at the current checks that are performed on the main feeds. The code that performs this is tightly coupled with the database load operations for the feed. As a result, I relied mostly on the documentation to provide clarity. There are two key types of checks performed:

- *DoD Variance* checks are responsible for comparing static data across days. These are used to pick up unexpected changes in data - for example, the rating of a product or the classification

of a certain asset. We would expect these to be fairly constant so any changes could signify an error, and must certainly be flagged for further verification.

- *Aggregate* checks are performed one level higher than individual row entries. These group parts of the feed based on common characteristics and perform range based checks. An example of this would be the rule "the market value of all traded companies must be less than $10bn".

Rules are defined on a per feed basis. A special kind of aggregate rule, which is applicable to all feeds, is the row count rule. This says that the volume of data received for a feed, i.e. number of rows, is largely the same every day - to verify this, we simply perform a row count operation on the entire feed and compare this to the previous day, flagging if the variance is higher than a predefined threshold.

At this stage, I had an idea of the kinds of check that needed to be written for the Reference Data feeds. The calls with the user also allowed me to start working on mock-ups for the UI which would help them drill-down to the root cause of the errors quickly. Given this, I started designing the codebase for my project.

Design

My project had a natural divide into a front and backend. The communication of the two parts was via a RESTful service responding to `HTTP` requests. While the project was in two parts, it was most definitely not an equal split - the backend required significantly more work and planning whereas the frontend had a good starting point given the template module repository (see below). As a result, my manager and I decided to focus on the backend to ensure it was in a working condition before starting on the frontend. If I were not able to finish the project in time, the frontend would be easier for someone else to pick up and complete as a separate task.

Backend

A high level overview of the design is shown in Figure 2. The service is fairly typical of any web application - it acts as a handler for UI requests, delegating to defined functions to produce a response per endpoint. It communicates with the database via the Database Access Object (DAO). The main requests are to read the rules defined per feed and the results and errors (if any) from a particular run of a rule. As an extension, time permitting, we would also be able to write rules to the database and trigger runs of the rule via the UI.

The DAO acts as the interface for all communications with the database. It is responsible for constructing and executing `SQL` queries, and parsing the results produced. It is used by the service - as described above - and by the `RulesEngine`. The DAO can be extended with relative ease as the `SQL` queries are abstracted away from the main codebase. For example, adding a new rule type would involve modifying the DAO to account for the new data stored - this is supported in the current design.

The third component in the backend is a MS-specific implementation of a `CPSListener`. CPS is a proprietary pub-sub system for messaging between components. The current pipeline, described in

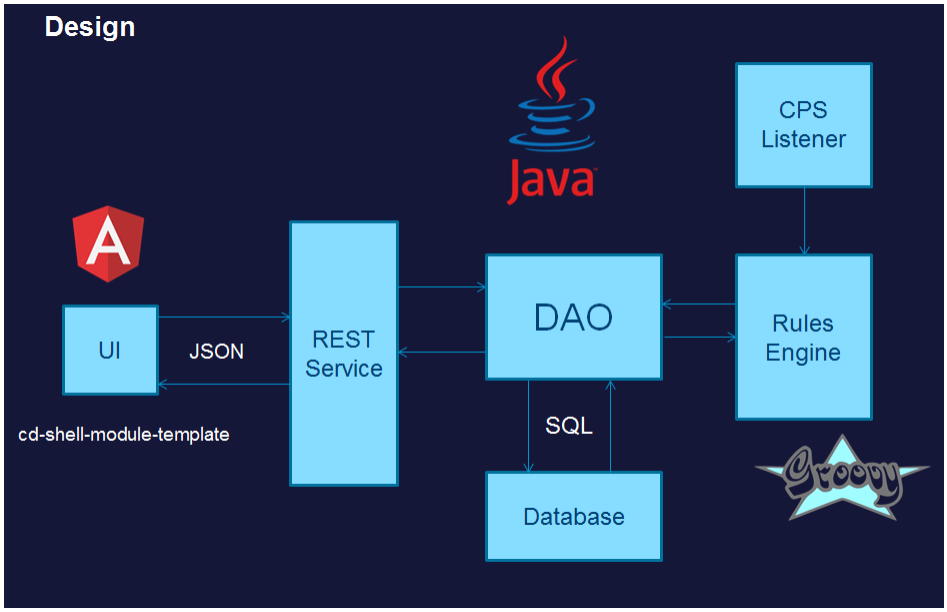

Figure 2: High-level Project Design

the *Context* section above, has multiple components which process the feed at different times. By listening for the messages between these components, we can choose to trigger the `RulesEngine` at a suitable time such that the rules are run automatically once the feed has successfully loaded. Thus, the listener acts as another service which handles "requests" from existing components.

The final part of the backend, where the bulk of the work happens, is the `RulesEngine`. It reads the rule from the database and creates an internal representation of the rule which is compiled and executed. The engine then writes the results and errors from the run to the database. Once again, the engine is designed to be both extensible and replaceable. Adding a new rule is very straightforward - we simply have to provide a template, and the relevant information in the database to construct the rule. Furthermore, the engine can be replaced entirely with a different implementation; as long as it satisfies the provided interface, the rest of the backend would work unchanged.

Frontend

The frontend involves adding an extra tab (module) to an existing dashboard application. A template module is provided which we used as the foundation for the UI. The initial design involved the base templates, suitably modified to show the rules and results screens. However, based on feedback from the users and from other members of the team, we changed the layout to a custom tabular interface.

This decision was based on the viewing of errors. Users wanted to be able to view errors from multiple feeds at the same time - this could potentially highlight trends or patterns that would enable them to get to the root cause faster. As a result, we use a hierarchical structure on the UI with the Feed at the top level, moving down to Rules then Results and Errors. Each section can be expanded to give more details.

During the initial weeks, we produced mock-ups of the UI to help visualise the end goal and gather feedback. Since then, it largely remained unchanged as compared to the final product. Producing the mock-ups involved using the existing template module and adding sample data to the views. This proved useful when returning to work on the frontend later in the project.

Implementation

Having designed and planned the application as a whole, much of the rest of the placement was spent implementing it. I had a fair amount of flexibility with the languages, tools and frameworks, however it would be ideal to stick to what the team was currently using - this would allow me to benefit from the huge amount of experience they had, especially with regards to working within the MS developer environment.

Context

Every developer at MS is set up with a Morgan Stanley Development Environment (MSDE). This loads the basic tools and programs needed for software development. We develop in the Windows OS, with TreasuryTech working mostly in Java for server side code. UI code is usually in Angular (for web applications) or C# for Windows desktop applications. The IDE of choice for the team was IntelliJ, with Git the preferred tool for version control.

In addition, we used some MS specific tools to help set up the environment. One particular tool, known internally as *the Train*, is used to create, deploy and continuously integrate all projects. This links with Git (for source control), JIRA (for issues and feature tracking) and Jenkins (for CI) to give a comprehensive portal from where we can manage all code and configuration related to any project.

Backend

Given my team's preference for the language and my own experience with it, I decided to use Java for the server side code. I used Spring for dependency management with Gradle for build configuration. I have experience working with both (3rd Year Group Project), with the team also being familiar with the setup of these tools internally - despite this, many of the issues that I initially faced were related to configuration.

The use of Jackson for `JSON` serialisation combined with Spring MVC made the services quite straightforward to implement. Again, the fact that I had used both these tools at university helped considerably. In contrast, the DAO was very new to me since I have never worked with databases before. While the code to do this is relatively straightforward, unfamiliarity made it longer to write than I expected.

Spring provides an abstraction to the traditional Java Database Connectivity (`JDBC`), making the core implementation of the DAO significantly shorter. The `SQL` queries are extracted into templates which are accessed as resources at runtime; additionally we use `NamedParameterJDBCTemplate`s to substitute query parameters dynamically. This provides performance benefits as queries can

be pre-processed; however, the disadvantage is the need for table name injection at compile time, introducing both security risks and extra boilerplate code for string substitution. The other role of the DAO is to parse the results of the queries. This is made much cleaner through the use of Java 8's lambdas - writing `RowMapper`s to convert the `ResultSet`s to POJOs becomes easy and concise.

The bulk of the backend codebase is contained in the `RulesEngine`. By design, this is designed to be replaceable and extensible. One of the biggest decisions made during the implementation was choice of `RulesEngine`. I had multiple constraints, both performance-based and logistical - the various choices considered were Drools, Groovy, a custom-implementation using Java or open source projects such as EasyRules or RuleBook. The biggest criteria for comparison were the easy availability of the application internally within MS and a simple DSL for the rules. Furthermore, given the relative mathematical simplicity of the rules that I would be running, a lightweight engine would work best. Given these, I decided to work with Groovy for my implementation of the `RulesEngine`.

The Groovy DSL is very easy to read, both for programmers and non-technical users. The syntax is clean and it integrates well with Java as it is a JVM based language. The setup and configuration required minimal effort, allowing me to spend more time on the rule logic. However, as mentioned earlier, it is possible to replace the Groovy implementation with an alternative rules engine. As long as we satisfy the interface provided, we can support a Drools based engine (for more complex rules) or a custom engine (for more flexibility). Under the current design, the engine is also responsible to create the Rule script to execute - only the minimal configuration is stored in the database. This allows us to add additional rule types or logic for new rules with relative ease.

Frontend

The frontend implementation was much less complex in comparison. Given that I had a base to start with, there was little to no setup and configuration. The current codebase is written in Angular 1.5 with pre-configured testing frameworks Karma and Jasmine. The majority of time for the frontend was spent in wiring up the backend for data access and integration with the current dashboard application which is already in production.

At the time of writing, this is still work in progress so I cannot add more detail; however, I expect that the frontend implementation will be relatively straightforward given that much of the UI is read-only as of now. If I have more time, I hope to allow users to insert rules into the database and run rules from the UI. This would require additional Rule validation checks for security reasons, and further additions to the UI to be fully functional. Both are likely out of scope for my placement but viable future extensions.

Feedback and Workflow

Since the project is backend intensive, there wasn't much opportunity for user feedback during the implementation stage. As a result, much of the feedback was internal; my code was reviewed thoroughly by my buddy before being approved. In contrast, for the UI, I initially created mock-ups during the design phase. These were shown to both users and other members of the team. Based on their feedback, and further information on how the application would be used, I improved the

drilldown feature and the way errors are displayed on the module.

For the project as a whole, feedback is best gathered via user testing. I hope to deploy the application to the QA testing environment before the end of the placement, allowing for more extensive tests by users. This turnover from the local development environment to the user environment introduces new challenges. Any side effects of my application would affect other functionality of the team's apps and needs significant impact analysis before deployment.

As an intern project, the app was designed to be developed by me in the specified timeframe. However, as the requirements became clearer, we changed the scope of the project, monitoring this on a per-sprint basis. All the development was done by me, with significant input from my manager and buddy during the design phase. We adopted a semi-agile workflow, working in two week sprints and pushing code to the release branch at the end of the sprint.

Over the course of a two-week sprint, I worked on features using JIRA to keep track of tasks. We aimed to discuss and approve one pull request (PR) a day, although in practise, the system worked better with one PR every two days. The review was thorough, down to a line by line analysis of the code written. As a result, I aimed to keep changes minimal and make incremental improvements.

Implementing features on different branches allowed me to continue working on future tasks while waiting on a current PR to be approved. After discussing and implementing the changes required for approval, I repeated the process with the new branch. Changes were reviewed on merge into a `sprint-<i>` branch, which remained active for two weeks. At the end of the sprint, a second, quicker review was conducted to merge the code from the `sprint` branch into `master`. This double safety mechanism ensured that only production ready code was deployed at all times.

The DoC Course

In general, I would say that the courses taught at DoC prepared me well for the challenges faced on placement. On the technical side, the courses cover the material in depth while the style of teaching and the culture of independence at Imperial help to pick up any other necessary skills on the job. The two most useful courses would undoubtedly be the Java programming course by Alistair Donaldson and the "Software Engineering: Design" course, taught by Robert Chatley.

The former is incredibly useful to start thinking about programming in an object oriented paradigm. While the topics covered are fairly basic, they lay the foundation for much of the advanced Java concepts that I used and needed to understand for my project. In addition, looking back at the lab exercises that worked alongside the course proved useful to brush up on the basics of Java before building up.

While the software design course did have weekly exercises, I still felt it was too theoretical. However, having done the placement now, I realise the true value it brings. The design patterns covered form part of the everyday discussions that I had on the placement. Often, solving the problem at hand was straightforward - doing it the best way, which made it readable and extensible proved the challenging part. The course helped me think about programming as more than just coding,

proving invaluable when designing my application.

As a JMC student, I had many optional modules which I did not pick. The two in particular that I felt would have been useful - had I done them - were the Databases course in Year 1 and the WebApp project in Year 2. My project was very closely related to both and I would have significantly benefited from the knowledge gained. However, I felt that the general DoC culture of independent learning came to the rescue. While I did pick up the database and web app knowledge that I needed for the project, doing the modules would have certainly made this faster.

In addition to the taught modules, the importance of group projects and teamwork that is provided at Imperial is particularly useful in the working environment. Specifically, the group project in Year 3 prepared me the best for life at MS. The concepts of starting from scratch with no existing codebase, integrating existing (legacy) software into my app, using an agile workflow with fortnightly sprints and building a full-stack web application were common to both my group project and placement project. Going through the process before made some of the tasks on my placement much easier - the challenging aspects were to do it in the MSDE and largely independently. Moreover, the biggest issues that I came across during the placement were configuration related - the group project helped prepare me for this, such that the frustration that comes with setting up a project was not entirely unfamiliar!

Finally, I think there are two aspects specific to my placement that were *not* covered as well by the DoC courses. One relates to the business context in which my app is used; the financial knowledge is difficult to cover in a core Computer Science degree, however it would have been useful to have the basics taught such that picking up job-specific knowledge would be faster. With BPES options and the Finance for Computer Engineers module in Year 4, this gap could be bridged; I look forward to them in the year ahead.

The other skill that I felt unprepared for was the scale of the application that I was building. In general, I feel that scalability is not covered in detail in the modules taught. The various projects undertaken during the degree are never analysed by "end-users" in sufficient detail to highlight inefficient code or poor running/response times. While the core remains the same, putting an application in production where it is used by multiple people is significantly more challenging than writing the same app for a university project, where it is only ever used by a handful of people. While some of the groups in Year 3 did have to overcome this for their project, my personal experience didn't cover this. Once again though, the skills required here can be picked up in the office environment and the course certainly helps us do that.

Reflection

Overall, I found the placement to be a thoroughly enjoyable experience. The chance to apply the technical knowledge gained from the degree was invaluable. Much of the team's working style and approach was similar to what I followed in my projects at university; as a result, adjusting to this was straightforward. The major issues that I faced early were to do with the specifics of the MSDE and the associated configuration. This was frustrating to overcome but made me better appreciate the jump from academic development to the industry standard.

The biggest takeaways from the experience are the soft skills I gained; this was my first placement and gave me an opportunity to see the office environment that I would likely be working with in the near future. In particular, I think the experience at the bank varies considerably based on the team that you work in; I was incredibly lucky to join a team which was largely relaxed and supportive. There is, of course, a focus on results but the process of writing clean, readable and elegant code is much more important and appreciated.

The relaxed attitude of the team translated into flexible office hours, with the focus on getting the tasks done as opposed to spending a specific amount of time on desk. I stuck to a fairly regular routine of 9-6 but did have the occasional early start for a training workshop, or a late finish towards the end of a sprint. In particular, the users of the application are largely based in India; this involved setting up meetings with them during the morning, before they finished for the day.

Looking back, I think there are two (related) things I would change if I were to do the placement again. I spent roughly 5-6 weeks on the design of my application, followed by 7-8 weeks on its implementation. I think this split was not ideal; I should have allowed more time for the latter parts of the project, given that there were various configuration and setup issues along the way that I had not accounted for. As an extension to that, within the implementation stage, I had a 3:1 split for backend to frontend. This meant that the UI, despite needing less work, was rushed and basic. Given more time, I would like to polish the frontend codebase and ensure that it is deployment ready.

Ethical and Environmental Issues

As a firm, there is a big emphasis on ethical and professional conduct. The onboarding process for the placement involved countless training videos and exercises to ensure that we were made familiar with the code of conduct of the firm, including the correct reporting procedures for escalation. MS's reputation is key for the firm to do well and maintain its relation with clients; there is no compromise on this front.

The policies in place at the firm cover a wide range of situations and scenarios. Data integrity is key, especially with regards to protecting client data. The developers rarely work with or have access to production data, with a strict permissions process for almost all access. While this is quite frustrating, especially at the start of the placement, it is there as a protection mechanism for all parties involved. Thus, all employees go through and understand the conduct documents.

While the placement doesn't directly involve working with a specific audience, employees are allowed to volunteer with a local school once a week. In addition, June at MS is the *Global Volunteering Month*, where employees from all offices take time from work to help a local cause. As interns, we took part in both these activities; volunteering plays a big part in the firm's philosophy and *Giving Back* is one of the four core values of MS. This also involved working with children from local schools; we had extra training sessions to ensure we were prepared for this, specifically with regards to our legal obligation in the situation.

MS also has a focus on sustainability, both as a firm itself and of the surrounding environment [5]. This translates into an eco-friendly office building, with particular focus on reducing energy consumption outside of office hours. The key idea is that small changes can make a big difference given the size of the firm; using power saving monitors which turn off when not in use and motion sensors for the office lights are examples of this.

Conclusion

In retrospect, the placement at MS provided the perfect opportunity to apply the skills gained from the DoC course. I would highly recommend the Summer Analyst program [1], both as an industrial placement for DoC or as a summer internship otherwise. In addition to the educational experience, it provides a great social summer, meeting and interacting with other interns from a range of universities.

I would like to thank my team at MS for all their help and advice over the placement; without that, the project would certainly not have been possible. In addition, I would also like to thank HR, both for the initial interview process and the numerous social and career events organised during the placement. Last, but certainly not least, a huge thank you to Anandha and the DoC placement team for giving us this opportunity in Year 3 and helping us prepare for life after university.

References

[1] Morgan Stanley - Summer Analyst Program, https://www.morganstanley.com/people-opportunities/students-graduates/programs/technology/summer-analyst-emea/, *Accessed October 2017.*

[2] Morgan Stanley - What We Do, https://www.morganstanley.com/what-we-do, *Accessed October 2017.*

[3] Nasdaq Institutional Ownership Summary, http://www.nasdaq.com/symbol/ms/institutional-holdings, *Accessed October 2017.*

[4] Hibbard, J: "Morgan Stanley: No stars... and lots of top tech IPOs." in the Business Week, *Accessed via Wikipedia, October 2017.*

[5] Morgan Stanley - Sustainability, https://www.morganstanley.com/ideas/building-a-sustainable-future, *Accessed October 2017.*

Confidentiality

This report has been approved for external access by Morgan Stanley. However, I am unable to reference the resources that I used internally while producing it. In addition, I cannot show screenshots of my final application as that would involve revealing some form of client data.