Card Services API Optimization

A Report Prepared for:

The University of Waterloo,

Department of Mechanical and Mechatronics Engineering

Prepared By:

Hank Wu

April 18, 2021

April 18, 2021

Professor Andrew Kennings

Director, Mechatronics Engineering

Department of Mechanical and Mechatronics Engineering

University of Waterloo

200 University Ave W.

Waterloo, Ontario, N2L 3G1

Dear Professor Kennings,

This report, titled “Card Services API Optimization”, was prepared as my 2B work term report. The purpose of this report is to provide you with the concept that I have selected, and to outline the progress I have made towards developing this project.

The concept I chose to write my report on is the optimization of card services application programming interface, otherwise known as API, for the Royal Bank of Canada, abbreviated as RBC, Technology and Operations department. Optimization in this context is defined as the improvement of efficiency in terms of speed and data usage, along with the improvement of effectiveness in terms of design and user experience. Although there are existing APIs for all card services, some are not tailored to the use case and business logic. These unoptimized APIs are the ones in focus for the scope of this project.

This report was written entirely by me, and has not received any previous academic credit at this or any other institution. I would like to acknowledge the help of Mr. Ernie Shu, Senior Engineering Director, who helped me understand the framework at RBC, gave me very helpful advice, and proof-read my final report. My role in the project was to work on designing, building, and testing the new and improved API. The project intermittently spanned 3 months and two weeks. Mr. Shu can be contacted at ernie.shu@rbc.com. I would also like to thank Ms. Esther Itimi-elo, who helped me with onboarding and was my mentor for the work term.

Table of Contents

1.0 Introduction

The goal of the project outlined in this report is to analyze and explain the process required to build a reliable and effect application programming interface (API) that is able to serve up to 9 million Royal Bank of Canada (RBC) customers. The card service API at RBC is built up together from a collection of specific APIs and it is called each time a user accesses their cards on the RBC web application or mobile application. The features it provides include card lock, card balance, card name, and card description.

To gain a better understanding of the structure of the card service API and its respective downstream calls, it can be thought of as learning a new language from a dictionary. To begin with only fundamental words would be known, thus when searching for a new word, its definition must be further traced until it reaches a known fundamental word.

That is analogous to the case with card services, as the architecture of RBC's API services places a large emphasis on modularization, meaning that ideally each piece of unique information will have its own call signature. The advantage of this design is beneficial in its maneuverability, allowing developers to choose exactly which pieces of data need to be pulled.

1.1 Card Services API Initial State

By January 2021, RBC had fully completed version 1 of its card services API. However, there were still many areas of improvement and features to be added. Along a new User Interface (UI) overhaul incoming, migrating the RBC website from Hub 2.0 to Hub 3.0, the decision was made to build a new version 2 API from the ground up.

Within RBC, there are two main users of the Card Services API. The first is the Banking With Control team, known as BWC, who offer a lost or fraudulent card protection service, in which customers are able to report, lock, and view activity on cards they deem to be stolen or misused. The second is the frontend of the RBC website and mobile application, colloquially known together as RBC Hub, which is undergoing a migration with significant changes and improvements.

These two use cases serve as the gateway between our RBC server's backend services and the customer's client side frontend services. BWC and Hub 2.0 handle over 9 million unique users across the platforms. The plan was to roll out the service to 20% of users and gradually scale up to gauge both the effectiveness of the service and any possible bugs or failure causes.

1.2 Facilitating and Enabling Software

The card services API is made possible through the assistance of countless downstream microservices, along with security cloud pipelines and health monitoring applications. These three internal software services helps enable the card services API to deploy into production with all its functionality intact while preventing security issues and catching bugs and anomalies that may occur on deployment. It is crucial to consider compatibility and support for the software that the API will depend on, as failure to accommodate constraints set forth by dependencies can cause a downgrade in efficiency or even breaking points.

Each downstream microservice handles a function call that retrieves information from the database or another microservice and returns the result in a predetermined format. Microservices can be nested to call lower level microservices in order to retrieve multiple data points from one call. For example, when the card service API needs to lock a card, it requires the encrypted card id, authentication token, and account name. In this case, the authentication token interacts directly with the cloud database to provide a JWT token that can be used to manipulate the card. On the contrary, card id and account name can be received through a singular call by calling a nested microservice and encrypting the card id on the api backend.

RBC has nearly all of its active software services executing within third party cloud servers, such as Microsoft’s Azure and Amazon’s AWS. These cloud hosting and database services offer many options for cloud security, such as config rules and security policies. These are vital whenever working with personal user data or finances, as an unflagged or open entry point could spell disaster for the entire company and its 9 million customers. As such, it’s important to always consider the enablement and even improvement of existing software such as the card services API to accommodate cloud security pipeline checks.

While it is important to extensively test APIs before launch, there will always been uncaught anomalies that occur when exposed to the open. These anomalies can be identified and recorded through the use of health monitoring third party applications like Dynatrace and Taplytics. Dynatrace is used to monitor API requests and responses and provides a comprehensive user interface to help identify time frames, patterns, and drill down towards exactly which section of the API is failing. Taplytics helps identify correlations and causations of the API, such as IP addresses it’s being called from and frequency of requests from a single user.

2.0 Performance, Load, and Speed

The main reason for the proposed version upgrade for the card services API is to increase its performance levels to match the necessary requirements for a full deployment to 9 million users. While it's not necessary to reserve the capacity for 9 million users to access the same feature at the exact same time, it is reasonable to assume that at any given time a certain amount of people will be constantly pinging the server for their card information or to lock their card. To understand exactly what times of day receive the highest traffic and to gauge a numeric value for the traffic, a launch to 20% of RBC users was set in place by January 2021.

The results from the partial launch were mostly positive, except for the card lock service, which could only handle 5 transactions per second. This resulted in the API occasionally returning an error when sent too many requests at once. Often this was caused by a curious user, who would attempt to lock and unlock the card in succession. Though the issue may be fixed by implementing a cooldown to card lock, it would have been a temporary solution as a full 100% deployment would see the same issue even with restrictions on the user.

In the context of card services API, the load upon its requests and responses are considered lightweight due to the data being transported simply in the form of JSON and strings, essentially text based information streams. However, the load upon the server is a major factor, as there is only a set amount of server processing power allocated to each microservice. This means that at any given time, a single microservice will always have a maximum number of requests it can handle concurrently before needed to delay and queue further requests.

Performance and load are two quantities that can be considered as analog as opposed to binary. This means that they are not either full or not full, but instead gradually reach their limits as they handle higher demands of requests and information. Speed is a function of performance and an inverse function of load. This means that to achieve the highest speed possible for an API, the performance must be maximized, while the load must be minimized.

2.1 Identifying Performance and Load Issues

Initially, the causation of the subpar deployment of card lock service was ambiguous. To figure out the culprit responsible for errors, timeouts, and crashes, the

API needed to be stress tested. Stress testing is when an API is rapidly and continuously tested, putting it to its limits and identifying when it breaks. To achieve this, a JavaScript library called k6 was used to code multiple tests. K6 allows developers to simulate asynchronous requests from multiple users by rapidly sending mock requests concurrently to a running API service. Therefore, a method was devised to isolate the API calls and test each individual endpoint or route to map out its patterns and determine the one slowing down the overall process. Examples of routes are methods to create, remove, update, or delete data elements such as card holder information, card id, and card lock status. The following were tested:

GET: card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v1/{cardID}

PUT: card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v1/{cardID}

GET: card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v1/{cardID}/health

GET: card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v1/{cardID}/lockstatus

PUT: card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v1/{cardID}/lockstatus

When calling the API, the cardID will be automatically provided from the backend along with an authentication token to ensure users can only access their own cards. Through rigorous testing and considering over 20 data points, two definitive conclusions were able to be drawn.

First, the slowest endpoint was by far the GET endpoint for card lock status, reaching only 5 Transactions per second, known as TPS. This was a combination of the frequency that customer checks their cards page, along with the limited bandwidth allotted by the retrieve card lock status microservice.

The second issue was that many improper calls were left hanging on the microservice for up to 40 seconds, delaying other API calls and slowing down the API all together. Improper calls could be caused by a multitude of reasons, from incorrect cardID and authentication token, to the common card lock eligibility check fail, which is a prevalent scenario among business credit card users.

2.2 Fixing performance issues with API version 2

The previously identified performance and load issues needed to be fixed in order to achieve an API with 100% test coverage that can be scaled from the current 25% of users to the expected 100%. To accomplish this, the backend of the API was built completely from the ground up, reimplementing the card services in new and efficient methods. Along with this, other internal teams within RBC needed to know about the issue in order to coordinate a fix on their service that serves as a dependency for card services API.

Addressing the first issue, k6 performance tests were essential as the programmed tests were not only able to identify the source of the error, but also provide context as to the cause. This was possible due to the unique variation testing method offered by k6, in which different schemes would be mocked in rapid request format. For example, one method saw mock virtual users ramp up and increase to the total amount gradually while sending requests, whereas another completed the ramp up completely before sending requests. The issue arose when users sent requests before ramping up all the necessary aspects of the request object, such as the cardID and authentication token. This caused a major issue in performance, as servers must handle card requests sequentially, so an incorrect order would take exponentially longer to resolve. For sake of example cardA and cardB are two unique credit cards that are eligible for card lock. An issue can arise when cardA sends its cardID, but cardB sends its cardID token before cardA can send its authentication token, thereby interfering with the parameter order of both requests.

Though the utilized microservice accounted for such bugs and mapped out the correct cardID and authentication to the correct card, generating a data structure and running an algorithm took an immense amount of server processing power relative to the request itself. To combat this issue, new code was written to promise a result to the microservice, and have it await for a response as the API can guarantee one when called with a correct set of parameters.

Both promise and await were leading edge function capabilities of Java. Therefore, a team meeting needed to be held to allow all team members to understand its capabilities and effectiveness to solve our issues. Though seemingly redundant, the major part of working enterprise software are the people who are responsible for coding it, and it is essential that all team members are in alignment and agreement in terms of knowledge, understanding, and approach for a successful launch.

2.3 Fixing load issues with API version 2

Addressing the load issues required a two fold approach, increasing the response time limit for the API, while decreasing the response time limit for the downstream microservices. Response time limit is the maximum time a program will wait for its expected input before it returns an error or null. In the case of card services API, the response time limit for card lock status was 40 seconds while the response time limit for the API was only 3 seconds.

This issue was found by combining the data provided by Dynatrace and Taplytics to visualize each service call and identify the inconsistencies within them. As the internal services within RBC were all centralize on cloud server platforms, a response and request will almost always be passed along in similar time. Therefore, the difference in the 40 seconds and 3 seconds for response time limit between API and microservice was a major issue. This not only gradually slowed down the processing power of the microservice, but also left many requests hanging for 40 seconds, unable to retrieve a response.

The team responsible for the card lock status microservice was contacted and the entirety of the work on version 2 of the card lock status API was shared with them to provide them context of the solution. An agreement was reached, in which card service API would increase its response time limit to 5 seconds and the microservice team would lower theirs to 5 seconds too. This decision was made because analyzing the history of successful request times displayed that none required longer than 3.8 seconds of time.

Along with the updated response time limits, circuit breakers were built by both teams in order to ensure the set limits are strictly enforced. Circuit breakers are able to cut off the program from its own side to prevent any hanging requests to affect its own performance. This new function code was also able to be applied across many separate and unrelated APIs maintained by the same team, catching more bugs and lowering the error rate.

3.0 API User Experience and Coverage

Throughout the building and optimization process of the version 2 card services API, maintaining support for test coverage, health monitoring, and user experience were essential to the improvement of the API from previous iterations.

The card services API is built using the Java programming language on the Spring framework, which serves as its architecture foundation that allows it to build, manipulate, and send requests and receive responses. Java allows the option for developers to create their own unit and integration tests to validate the functionality of the program. Though the card services API was only being improved, its core code has changed drastically, leading there to be a necessity to code a new library of tests to run. Through running these tests, many error and breakpoints were located, some negligible, but others crucial to the success of the API. Through building out the tests, 100% test coverage was achieved and functionality of the card services API was guaranteed.

User experience is vital to the effectiveness of an API. As previously mentioned the card services API is called by BWC and Hub 3.0. Both the API and these two teams are within the Omni department, which attempts to consolidate its architecture and software standings, so that any developer may move across any team and be familiar with its design approach. Due to this, an idea was proposed to clarify the API request route for developers to gain an easier understanding. The type of the card would be specified along with the id in query parameters as opposed to path parameters. This allowed both developers and analytics software such as Dynatrace and Taplytics to gain further insight into the composition and execution of the API.

Routes such as:

https://card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v1/{cardID}/lockstatus

were refactored into:

https://card-lock-presentation-service-uat.apps.cf2.devfg.rbc.com/v2/cardType={DEBIT or CREDIT}&id={cardID}

This allowed users from across Omni and all of RBC to utilize the card services API without needing to understand the inner workings or study documentation. Furthermore, previously difficult data visualizations that required combining services were now able to be called simply from one service.