April 2016

Endicia Webtools Optimization and Migration

Justin Daniel Canas
Worcester Polytechnic Institute

Nicholas Matthew Muesch
Worcester Polytechnic Institute

Xiaoman Xu
Worcester Polytechnic Institute

Endicia Webtools Migration and Optimization

A Major Qualifying Project Report

Submitted to the Faculty of

Worcester Polytechnic Institute

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science

By

Justin Canas

Nicholas Muesch

Xiaoman Xu

Sponsored by

Endicia

Approved By

Professor David Finkel, Advisor

Acknowledgements

We would like to thank the following individuals for their support and assistance throughout this project:

Patrick Farry, Manager

Harry Whitehouse, CTO

Sanjaya Nepak, Mentor

United States Postal Service WebTools Team

All of Endicia

Professor David Finkel, WPI

Table of Contents

Abstract

This project worked with Endicia to bring an external United States Postal Service system into Endicia’s environment. The system is used to calculate data needed to print shipping labels for packages. We built a full-stack server-to-database application that interfaces with Endicia’s services. We received requirements to prioritize performance and to handle hundreds of requests per second while never incurring downtime. Through this project, we made this service’s response times 30x faster. The program was then handed off for deployment.

Chapter 1 Introduction

Sending and receiving postage has been a major part of the United States for centuries. According to statistics released by the United States Postal Service (USPS) for 2014, approximately 5,935 mail pieces were processed each second, satisfying 500 million online customers for 2014. [23][24] With the amount of people using technology in this day and age, speed and scale are always factors to consider when developing software solutions. With such a large demand for shipping mail, many companies have made it their mission to make the overall process a more efficient one. This necessity for speed is why the eCommerce company Endicia created this project for our team.

Endicia works closely with the USPS to allow businesses to ship mail in a fast and affordable manner. Endicia's customers are numerous, and the company satisfies around 300 requests per second to print shipping labels. The goal of this project was to increase the performance and reliability of this shipping process. In the previous Endicia system, every request to print a shipping label would have to go to an external USPS server and wait for the server to calculate and output the delivery date. The physical location of the USPS server was far enough to cause response times to be unacceptably slow, and the system itself had some inefficiencies such as calculating unnecessary data for legacy requests. In addition to the slow performance, internal Endicia systems relied upon this slow response from an external USPS server, which resulted in unreliability for Endicia's customers. To increase performance and reliability, Endicia asked us to create a localized version of the USPS system.

Chapter 2 outlines the history of Endicia and the core features and services they provide to their clients. Chapter 3 illustrates the requirements of our project as well as its importance to Endicia. Chapter 4 details the tools used in our application as well as the architecture of the application. Chapter 5 discusses some of the specific implementation details made in the codebase of the application. In Chapter 6, we discuss the optimizations made to both the program as well as the environment on which it is deployed to improve performance. Chapter 7 outlines the performance results of the application. Chapter 8 details the conclusion and future works from this project.

Appendix A displays the documentation written for users of our application. Appendix B illustrates how to maintain our application’s codebase after we leave Endicia.

Chapter 2 Background

This chapter focuses on the history of Endicia, as well as some of their core services and programs.

Section 2.1 PSI Associates

In 1982, a company known as PSI Associates was founded as a technology consulting company. One of their first customers was the United States Postal Service (USPS). Over the course of a few years, PSI analyzed the energy consumption of the USPS and assisted in creating hardware to make USPS services more efficient. Their first big contract with USPS was to create the Postal Numeric Encoding Technique (POSTNET) System. This would allow them to print a barcode on an envelope to sort mail in an efficient manner. After years of working with USPS, PSI Associates was tasked with solving the problem of address cleansing. To accomplish this PSI created their first software solution, Envelope Manager, in 1989. Due to this, PSI Associates rebranded to Endicia Internet Postage and continued to create innovative software solutions. [9]

Section 2.2 Endicia

Endicia and PSI Associates shared the same core goals, making shipping and mailing an easier and more efficient process for businesses. All of their programs and services are geared towards allowing businesses to create and print postage for all of their mailing needs. Some examples of these products are Dial-A-Zip and the Envelope Manager. Two of Endicia's biggest services are DAZzle and the Endicia Label Server (ELS), which allows businesses to design, purchase, and print shipping labels [15]. These services are discusses in more detail in Sections 2.4 and 2.5 respectively. To date, Endicia has made over $14 billion in postage printed [9] making them one of the largest eCommerce shipping companies. On November 18th 2015, Endicia's primary competition, Stamps.com, finalized the $215 million acquisition of Endicia [21]. Despite this acquisition, Endicia is still focused on helping make the shipping process easier for many businesses.

Section 2.3 Dial-A-Zip

One of PSI's first software creations was Dial-A-Zip. Dial-A-Zip was created by PSI Associates in the 1990s to solve the problem of obtaining Zip +4 addresses. Prior to the PSI solution this had to be done manually, making it a lengthy process. This software was "the first remote address verification system" [10]. In less than one second, PSI's Dial-A-Zip software can correct and subsequently validate any zip code against the Postal Service's list of addresses [15]. All of this is necessary to do prior to printing any shipping label. Dial-A-Zip is still a core feature of Endicia's services.

Section 2.4 DAZzle

DAZzle is a software product created by Endicia to allow customers to design a shipping label, print that label, track shipments and integrate with online marketplaces. There are "Quick Print", "Design", "Address Book" and "Postage Log" tabs in the DAZzle software. "Quick Print" allows users to enter origin and destination addresses, select the desired mail service, and add insurance value to their packages. After a user chooses a mail class, a preview of the shipping label will be created. If the user would like to create a customized label or envelope, he or she can go to the "Design" tab. "This section is geared toward more advanced label creation and allows users to adjust fonts, alignment, postage options, and more." [11] Once the design phase is complete, users are ready to print the label. [7]

Section 2.5 Endicia Label Server

Endicia Label Server (ELS) API is a web service that provides functionalities such as printing postage labels, calculating postage rates, buying postages and so on. Most of these features are also available in DAZzle mentioned in section 1.4. However, unlike DAZzle, which is a software product, the ELS API allows Endicia's customers to develop their own applications to offer their users USPS shipping without forcing end-users to install new software or change workflow. Since the API is operating system independent, customers do not have to worry about the compatibility problem in development. [12] This API is where we fit in. This project is intended to increase the performance and decrease overhead of the ELS API, by creating the local equivalent service that USPS currently provides for Endicia.

Chapter 3 Requirements

Endicia currently interfaces to a United States Postal Service (USPS) Web Service to receive service commitments for parcel shipments. A service commitment is information about a package or item being shipped such as its estimated delivery time and type of package. The aforementioned external interface with USPS is relatively slow and unreliable. This unreliability is because whenever a query about a package comes in, Endicia as a client has to forward the request to USPS Web service and wait for the response. Such communication takes around 200-300 milliseconds and is extremely unreliable as it depends on the server in USPS being up at all times. Endicia eCommerce customers expect faster performance and more dependability. The objective for our project is establishing an Endicia equivalent service by bringing the USPS data into the Endicia environment and keeping it current via data files provided by the USPS. With the data files downloaded, we can develop a program that calculates the estimated delivery time within a server local to Endicia so that Endicia no longer requires forwarding a request to an external USPS server. Internal interfaces would also be built for various Endicia products to utilize the localized server, making the experience more efficient and reliable for customers. Essentially, the local server would be a faster, more reliable building block for other Endicia products which require information from USPS.

Currently, no company has a localized equivalent of the aforementioned USPS service, as Endicia is the first company to attempt to implement such a system. As such, creating this system demanded a necessity to communicate with various USPS developers and iterate through a first version of algorithm and calculation specifics.

Section 3.1 Performance Requirements

The current use of a USPS Web Service by Endicia causes unreliability and slow performance of some of Endicia’s core services. The slow performance is primarily due to the fact that the Endicia server, the one sending a request, is located geographically far from the server that hosts the USPS Web Service. The physical distance causes the variability of network latency to be introduced into each request's performance. Creating a localized server that can be used by Endicia's core services eliminates that network latency as these servers will be hosted on the same network as the services which our program provides information to.

Using the current architecture of sending a request to the external USPS server, each request takes upwards of around 200 milliseconds. After creating a localized version of this server, we have a goal of between 10-50 milliseconds per request. This allows Endicia's clients to both receive responses at a faster rate as well as send more requests in the same amount of time. Another business requirement of this localized server is to be able to accommodate between 50-200 requests per second, as set forth by Endicia.

Section 3.2 Business Requirements

Endicia has a service level agreement (SLA) with their clients that for all of their systems, Endicia can only experience 5 seconds of downtime per year. This SLA becomes more difficult to accommodate when Endicia has to rely on third party servers. Hosting their own version of the USPS service allows Endicia to more easily fix and control any errors that may arise and ultimately reduce downtime of their own services.

The localized service we created also requires the use of data files provided by the USPS. These files are updated once a week. To provide Endicia's clients with the most up to date data, we had to create a means of updating the data files our program uses on a weekly basis. Since Endicia has the SLA of no more than 5 seconds of downtime a year, careful planning and designing was done in advance to ensure we can update the programs files without shutting down the program. Details on how this is implemented can be found in Section 5.3.1.

Chapter 4 Approach/Methodology

This Chapter focuses on the overall methodology for our project. Section 4.1 discusses the tools that we used in developing the core application. Section 4.2 outlines the tools we used in deploying the core application into a production environment. Section 4.3 describes the tools we used in debugging and testing the core application. Sections 4.4 and 4.5 discuss the USPS database and the accompanying files. Section 4.6 walks through an example of one of the USPS algorithms that our program uses to calculate package information. Section 4.7 discusses the architecture of the core application. Finally, the remainder of the chapter goes over the deployment phase of the application.

Section 4.1 Core Application Tools

This section outlines the tools that we used to create the core application. These tools are necessary for the deployment and development lifecycles.

Section 4.1.1 JBoss Drools

JBoss Drools is a business rules management system (BRMS) created by developer Red Hat [3] Using a Java like syntax, a business can utilize JBoss Drools to specify a set of rules for validating and manipulating Java Objects. This is useful for comparing a set of data against some known business facts and then manipulate the data accordingly.

```
rule "validate holiday"
dialect "mvel"
dialect "java"
when
        $h1 : Holiday( month == "july" ) //$h1 is a holiday object, and a fact
then
        System.out.println($h1.name + ":" + $h1.month);
End
```

Figure 1: Example of a Drools Rule

Figure 1 displays an example of a rule validating a holiday object. In this code fragment, the fact h1 (in this case the fact is a holiday object) is verified to have its month attribute equal to “july”. If the holiday input object has the month of July, the “if” block will evaluate to true and the “when” block will execute, ultimately printing a statement. If the holiday input object has any other month, then the “when” block will not execute and no more code will be executed.

In the program we developed for this project, JBoss Drools is used to update a user’s shipping label information based on specific rules specified by the United States Postal Service. This includes information such as an estimated delivery date based on a specified drop off time or the drop off date.

Section 4.1.2 Embedded Jetty

Jetty was created in 1995 to provide a server or servlet container to deliver content over the Web and has been constantly improving. Since 2005, from version 7, Jetty has been hosted by Eclipse Foundation. [14] It is used in both development and production environments to create various web based applications. This allows Java developers to piggy back off already well-developed server architecture for their applications.

Embedded Jetty takes this one step further and allows the user to embed this server into their application. This application can be a web service, tool, or framework [4]. The main slogan of embedded Jetty is “Don't deploy your application in Jetty, deploy Jetty in your application!” [4] Such flexibility allows us to utilize of an HTTP module inside the program and running the module instead of putting our application in a different server environment.

Section 4.1.3 Web Service Frameworks

“Web services are client and server applications that communicate over the World Wide Web’s (WWW) HyperText Transfer Protocol (HTTP).” As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions, thanks to the use of XML. Web services can be combined in a loosely coupled way to achieve complex operations. Programs providing simple services can interact with each other to deliver sophisticated added-value services.” [2]

We chose Apache CXF as the Web Service Framework for this application over other choices such as Apache Axis due to its greater compliance with industry standards and its ease of development. This framework was created by the Apache Foundation with its most recent stable release, 3.1.5, in February 2016. We chose Apache CXF for the following reasons: The first was because it is one of the most commonly used Web Service frameworks in production. Secondly, it adheres to the JAX-RS standard, a Java API standard for creating Representational State Transfer based (RESTful) Web Applications. More importantly, Apache CXF can be used in conjunction with Embedded Jetty to receive an HTTP request and parse through the headers and inputs.

A RESTful Web Service is a standard architecture style for an application to be built upon. The essential parts of a RESTful Web Service are as follows. First, a RESTful service must use the client-server model of a requesting client and a responding server. Next, the server must be stateless, meaning that the server does not remember anything about a user between requests. A RESTful Web Service must also specify cacheability. This cacheability allows a user or server to cache the results of a previous request as to not force the server to repeat an unnecessary calculation. This is also necessary to specify so that a user does not cache a result that was calculated with what is now an outdated information source. A RESTful Web Service should also be layered so that a user does not know if it is communicating with a server directly or an intermediate source. Layering allows for future features of the service to be built without modifying the server itself. Finally, a RESTful Web Service must have a Uniform Interface. A Uniform Interface is also known as a Uniform Resource Locator (URL), which a user inputs in their browser to reach the server. A URL must contain a way for a user to specify the resource they want from the server, a way to manipulate a resource on the server, and use self descriptive messages for a user to find a resource. The implication of using a self descriptive message is that the URL should not be an obscure phrase, but contain words and phrases relating to the resource the user wishes to receive or modify. [26]

Our choice to use a Web Service Framework allows the application to communicate with other core Endicia services without the need to be programmed in the same language or know anything about the program’s implementation and codebase. This Web Service Framework also allows the local Endicia services to request information from the application and request a specific return type. From there, Apache CXF will take the information calculated by the application and serialize it into the format the user requested.

Section 4.1.4 NGINX

NGINX is an HTTP server and reverse proxy that was created by the NGINX team in 2004. NGINX is still in active development with its most recent stable release in January 2016. A reverse proxy is a server that essentially intercepts the client’s request, receives a response from the actual server manipulating the data, and sending that response back to the client as though it were the server.

NGINX is known for its high performance and load balancing ability. This tool is used by many largely populated websites including: Facebook, Dropbox, Zynga, and Wordpress.com to name a few. [16]

Using NGINX allows the core application to handle many requests simultaneously at a level above the server itself. This ensures that the server never needs to know about how many requests are incoming.

Section 4.2 Deployment Tools

This section outlines the tools we used to productionize the application from a development environment. These tools were essential to the deployment life cycle of our application.

Section 4.2.1 Apache Maven

Apache Maven is a software project management and comprehension tool. It was created to provide a standard way to build a project, an easy way to publish that project and a means to share JARs [18] across several projects. These JARs consist of other Java libraries and code on which a project may rely. Succinctly, Maven is a tool which can be used for building and managing any Java-based project.

Maven builds and manages projects based on its Project Object Model (POM). By specifying plugins and JARs that a project requires in the project's pom.xml file, Maven build can save developers immense amounts of time when trying to navigate many projects. The builds are model based: Maven can build any number of projects into predefined output types such as a JAR, WAR, or another distribution based on metadata about the project, without the need to do any scripting in most cases. This allows developers to generate executable files in an automated way for various environments. [1]

Maven also has the ability to execute pre written tests for a project during its build process. This allows for a means of continuous integration for a project and means that developers can create new features for their project, and when building an executable, check to ensure that all tests run and pass prior to putting the new features into production. Such testing process allows an autonomous way for developers to create and test their projects prior to placement into a production environment.

Section 4.2.2 Docker

Docker is an application created by Docker Inc. to house server applications in a containerized environment. This tools is similar to the concept of a virtual machine in that they both can house a set of applications on a level above the hardware. The key difference is that, with Docker, a single file is used to contain information about the configuration of a Docker environment prior to starting it, with the use of a Dockerfile. With a virtual machine, the environment would have to be started and then manually configured.

Docker also allows the creation of many “containers” to be run on a single machine. This allows the creation of multiple server instances on one machine, which ultimately increases the number of requests the server can handle at any given time. Our application will be housed in these containers on a production environment. This process is further outlined in Section 8.1.

Section 4.3 Helper Tools

This section contains the tools used for both performance testing and visualization of the codebase. These tools were useful to the development life cycle but not used in final application.

Section 4.3.1 Object Aid

Object Aid is a plugin for the Eclipse development environment. This tool creates and displays Unified Modeling Language (UML) class diagrams for the codebase of a project. [17] UML allows us to visualize the structure of the project and how the various packages and classes work together. The visualization can help identify dependencies as well. For this application, we used Object Aid as a method to debug and verify class structures against their intended structure, and eliminate unintended dependencies.

Section 4.3.2 JMeter & RestEasy Client

RestEasy is an add-on for Firefox and Chrome browsers. This add-on is a tool created to test RESTful web service applications. RestEasy allows us to input a URL, change HTTP headers and methods, and send a request to the server located at the specified URL. RestEasy then allows us to see information about the response such as the response headers, the message body, and the status of the response. These conveniences can help us both debug and test our RESTful web service.

In this specific context, we used RestEasy to test the differing serialization types as performed by Apache CXF. We also used RestEasy to ensure that we can establish a connection to our application.

JMeter is a tool developed by Apache for the purpose of simulating requests to a server. We used JMeter extensively for sending a request to our application and recording the response times. JMeter was also used for load testing our program, using multiple threads to send requests simultaneously from multiple computers.

Section 4.3.3 JProfiler & JConsole

JProfiler [6] and JConsole [20] are Java application monitoring tools written by ej-Technologies Apache, and Oracle respectively. JProfiler was used to observe which methods were utilizing the most CPU time. This tool allowed us to determine where the bottlenecks in our application existed and focus optimizations in that area. JConsole was used to view the memory performance of our server during both times of high and no requests. JConsole was a key indicator of the garbage collector issue outlined in Chapter 6.

Section 4.3.4 Java Management Extensions

Java Management Extensions are used in this project to monitor the memory usage of the application. This tool allows more accurate assessment of the trade off between time performance and memory usage.

The way it works is a few options are selected when the program is run. Then, a separate program called JConsole connects with the Java application being developed via the network, and provides real-time visualization of the program’s memory usage and garbage collection.

Section 4.4 USPS Database

The USPS database consists of comma separated (csv) plain text files. These files are called Automated Transit Files or ATF. There is a PDF provided by USPS that details the implementation of processing logic for these ATF. According to the USPS document, there will be eight csv data files and four Jboss Drools rule files to manage. Updates of these ATF files are received weekly, before 11:59pm CT on Fridays. Only the files that have changes are uploaded by the USPS. The USPS provides three methods for obtaining the ATF files; the one we plan on utilizing is their web service. Through the web service, we specify which files we wish to download. For this, due to the fact that only updated files are uploaded, any files present will need to be copied and used as the next version of that specific ATF data file. The strategy we chose to handle these text files is to load the entirety of the files into memory, eliminating costly file I/O, as the files amount to little more than 200MB. As such, a relational database is not necessary. As a result of this, the weekly updating required us pre-plan the design of the database to prevent down time. This is detailed in Section 5.3.1.

Section 4.5 ATF File Usage Example

This section gives a brief example of how a calculation based on ATF files may go. The following list enumerates the data items necessary to compute transit time, and therefore delivery date, for a package. The usage example is not necessarily important in its own right, but such algorithms appear frequently within the USPS ATF specifications. [22]

1) Origin Facility
    a) Origin Facility from ATF_ADDRESS_CLOSE
2) Destination Type
    a) Street, PO Box, Hold for Pickup (HFPU)
3) Destination ZIP Code or Facility
    a) 5 or 9-Digit ZIP Code for Street Delivery
    b) Destination Facility from ATF_ADDRESS_CLOSE for PO Box Delivery and Hold for Pickup
4) Mail Class
    a) Choose Mail Class Code from USPS ATF document [22]
    b) Ship Date (the date a mail piece is dropped off)
5) Drop Off Time (the time a mail piece is dropped off)
    a) 0000 – 2359

Once these pieces of data are collected or set in their appropriate files, the algorithm displayed in Figure 2 is run to determine transit time.

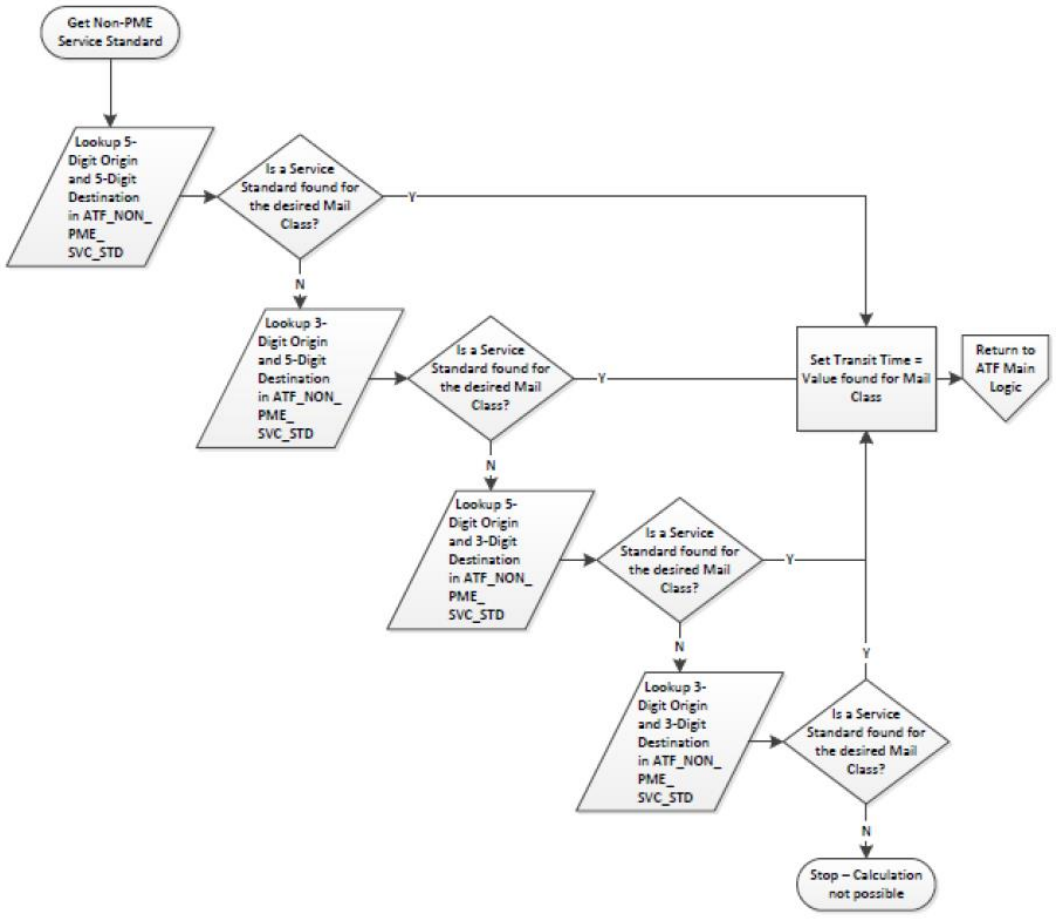
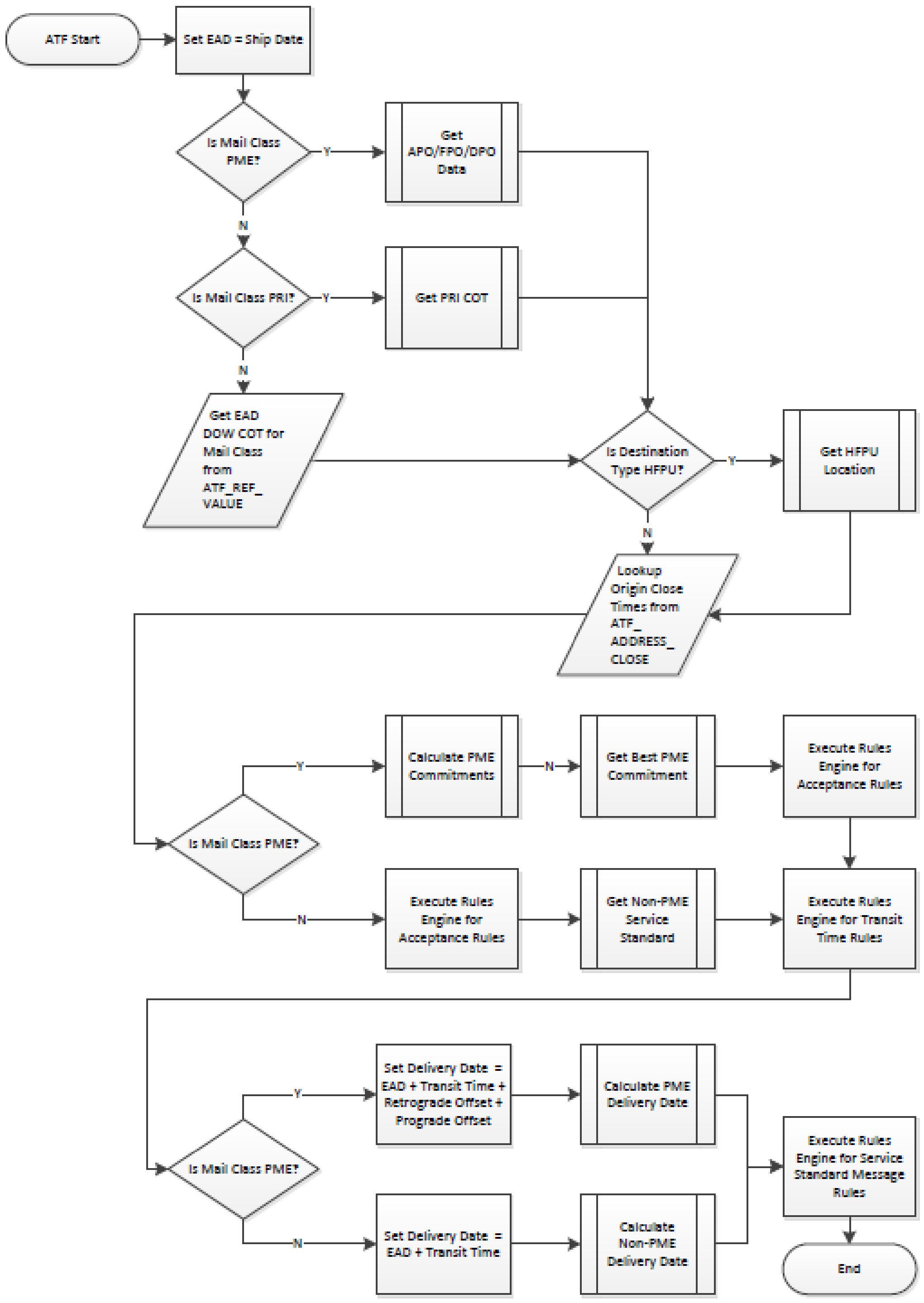

Get Non-PME Service Standard
Lookup 5-Digit Origin and 5-Digit Destination in ATF_NON_PME_SVC_STD
Is a Service Standard found for the desired Mail Class?
Y
N
Lookup 3-Digit Origin and 5-Digit Destination in ATF_NON_PME_SVC_STD
Is a Service Standard found for the desired Mail Class?
Y
N
Set Transit Time = Value found for Mail Class
Return to ATF Main Logic
Lookup 5-Digit Origin and 3-Digit Destination in ATF_NON_PME_SVC_STD
Is a Service Standard found for the desired Mail Class?
Y
N
Lookup 3-Digit Origin and 3-Digit Destination in ATF_NON_PME_SVC_STD
Is a Service Standard found for the desired Mail Class?
Y
N
Stop – Calculation not possible

Figure 2: Transit time Algorithm for a Priority Mail Package (Provided by USPS)

Section 4.6 Algorithm Example

The algorithms provided in the USPS ATF Documentation are mainly for calculating estimated delivery time. Figure 3 displays the main process for all Mail Classes containing the order of sub processes to gather data necessary to calculate the Delivery Date. It takes Origin ZIP Code, Destination ZIP Code, Ship Date, Drop Off Time, Mail Class, Destination type as input and outputs Delivery Data, Service Standard Message, and Guarantee.



ATF Start
Set EAD = Ship Date
Is Mail Class PME?
Y
N
Get APO/FPO/DPO Data
Is Mail Class PRI?
Get PRI COT
Get EAD DOW COT for Mail Class from ATF_REF_VALUE
Is Destination Type HFPU?
Get HFPU Location
Lookup Origin Close Times from ATF_ADDRESS_CLOSE
Is Mail Class PME?
Calculate PME Commitments
Get Best PME Commitment
Execute Rules Engine for Acceptance Rules
Execute Rules Engine for Acceptance Rules
Get Non-PME Service Standard
Execute Rules Engine for Transit Time Rules
Is Mail Class PME?
Set Delivery Date = EAD + Transit Time + Retrograde Offset + Prograde Offset
Calculate PME Delivery Date
Set Delivery Date = EAD + Transit Time
Calculate Non-PME Delivery Date
Execute Rules Engine for Service Standard Message Rules
End

Figure 3: ATF Main Logic Flow Chart (Provided by USPS)

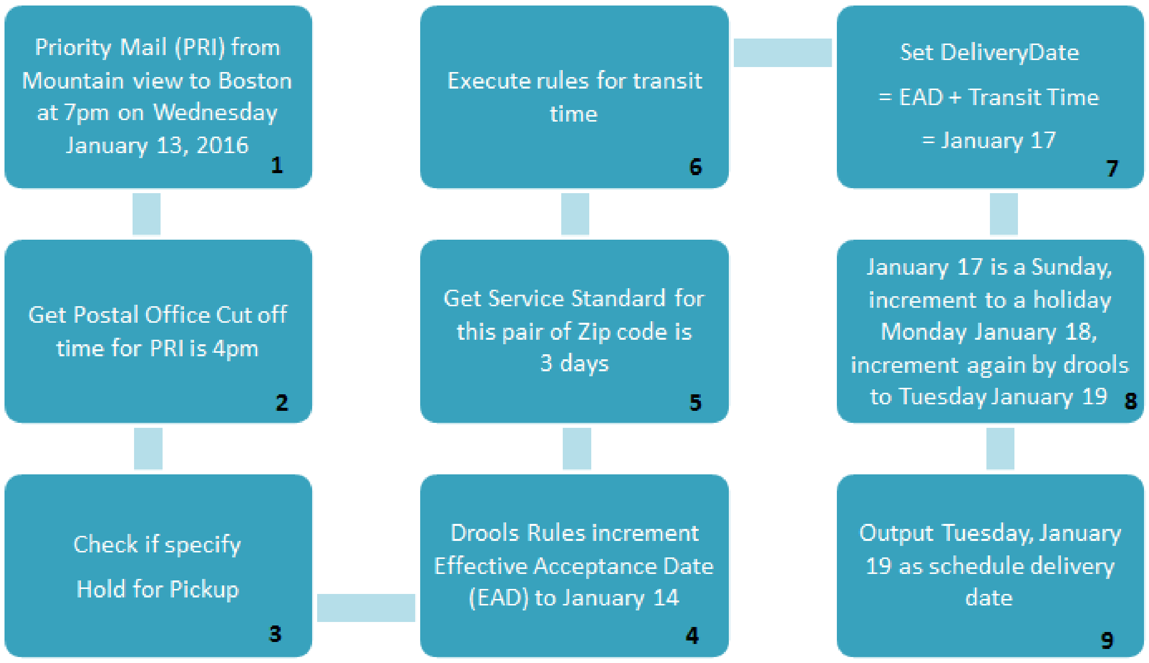

Priority Mail (PRI) from Mountain view to Boston at 7pm on Wednesday January 13, 2016
1
Get Postal Office Cut off time for PRI is 4pm
2
Check if specify Hold for Pickup
3
Drools Rules increment Effective Acceptance Date (EAD) to January 14
4
Get Service Standard for this pair of Zip code is 3 days
5
Execute rules for transit time
6
Set DeliveryDate = EAD + Transit Time = January 17
7
January 17 is a Sunday, increment to a holiday Monday January 18, increment again by drools to Tuesday January 19
8
Output Tuesday, January 19 as schedule delivery date
9

Figure 4: Calculation Example

Figure 4 displays an example calculation of the delivery date for a specific package.

- Box 1 displays the details about the package, the input of the program. In this example, the scenario is as follows: a Priority Mail (PRI) came in the postal office in Mountain View, CA 94041 at 7pm on Wednesday January 13, 2016, and would be sent to Boston, MA 02112.
- Box 2: First the cut off time of the office with ZIP code 94041 for mail class PRI was retrieved from file ATF_COT, which was 4pm.
- Box 3: Then the algorithm checked whether the sender had requested “Hold For Pick Up” (HFPU) once the mail has arrived at destination. If so, a HFPU location would be retrieved from the ATF_ ADDRESS_CLOSE data file as part of the output if the destination was eligible for such an option.
- Box 4: Then, JBoss Drools executed the ACCEPTANCE_RULES file for the time accepting mail. Though the mail came in on January 13, it missed the local cut off time, 4pm, for its mail class. Therefore, the Drools Rules file incremented the Effective Acceptance Date (EAD) to January 14.

- Box 5: The algorithm then tried matching the ZIP code pair 94041 and 02112 in file ATF_NON_PME_SVC_STD and found that the corresponding Service Standard was 3 days.
- Box 6: After setting the transit time as 3 days, the Drools Rules TRANSIT_RULES file executed to check whether a delay should be added, which did not happen in this case.
- Box 7: The initial Delivery Date was then set as the EAD plus Transit Time, which was January 14 plus 3 days, which calculates to January 17.
- Box 8: Unfortunately January 17 was a Sunday, and there was no delivery on Sunday for this mail class in Boston, so the algorithm incremented the delivery date to Monday, January 18. January 18 happens to be a US public holiday, meaning that there was still no delivery. Detecting this, the Drools DELIVERY_RULES file incremented the delivery date again to Tuesday, January 19, and it was a usual business day when mail delivery can happen.
- Box 9: The algorithm would finally output Tuesday, January 19 as scheduled delivery date, along with other important information such as cut off time of the origin postal office, HFPU location, and so on.

Section 4.7 Architecture Design

Our design of the program was an evolving subject. The project involved us taking the current Endicia/USPS architecture, shown by Figure 5, and porting it to a localized equivalent service. Figure 5 displays an Endicia customer that requested information to print a label for their package. This request comes into the Endicia servers to be serviced. The Endicia server now needs information currently located on and calculated by the USPS servers. Endicia then communicates with this third party server, waits for a response, and sends the final response back to the client. This can be a slow and unreliable method for a few reasons, as indicated by the aforementioned figure. Firstly, Endicia has no control over the health of the USPS servers and if the USPS servers are down, Endicia cannot service the client’s request. This is also a slow method because Endicia has to send information over a network layer to communicate with USPS servers.

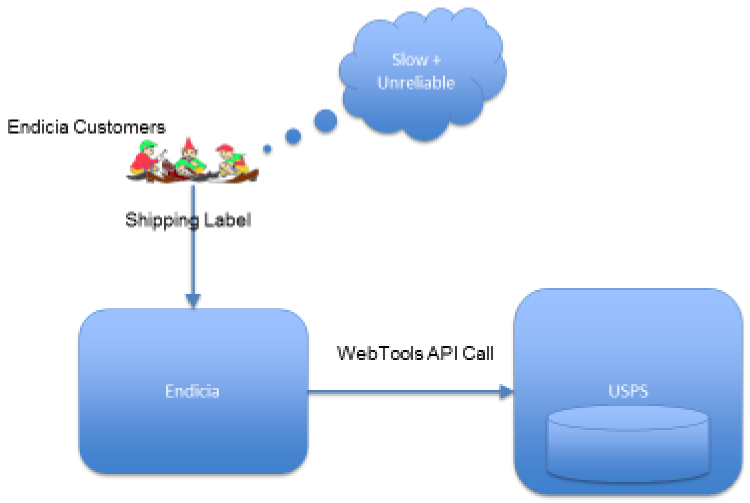


Figure 5 Old Endicia Webtools Architecture

To remedy this, the Endicia server will handle the entire request and not send any information out to USPS. In our design of the equivalent service, we initially thought that a simpler architecture with few layers to the stack, such as Figure 6, would be sufficient. This image depicts the initial response, as created by the end client or Endicia server, communicating with NGINX to reach our server. Then the Java program handles the calculations and manipulation of data, using the ATF Data files, and sends the response back up the stack. The main application would be hosted in a Docker container, and the ATF data files would be located on a machine visible to the local network.

Throughout the design phase of the development lifecycle we iterated over our original design, and a more complex architecture design was forged. This finalized architecture is displayed in Figure 7.

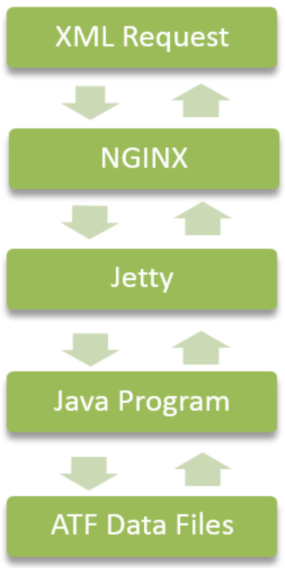


Figure 6: Preliminary Endicia Webtools Architecture

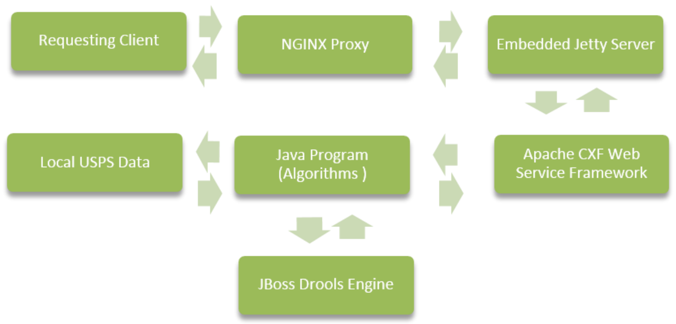


Figure 7: Current Design

Figure 7 displays a few of the key changes made in the architecture. In this design, a Web Service Framework, Apache CXF, was included. This is an important tool that sanitizes and deserializes user input as well as serializes the output for sending back to the client. Due to this addition to the program, the “XML Request” block of Figure 6 was replaced with the “Requesting Client” block of Figure 7. With the inclusion of Apache CXF, the requesting client can use any HTTP method (methods being either POST or GET). This also allows different input types, such as fields within the URL or an XML POST body. Another key addition to Figure 7 is the “JBoss Drools” block. This was included due to its separation from the main java program. JBoss Drools can be seen as a modular component to the main Java code that interacts with the data flowing through the various algorithms.

Section 4.8 Updating and Quality Testing

As previously mentioned in Chapter 3, the database will update at a minimum of once per week. A major goal of the methodology we used to accommodate the weekly update was that it should be completely automated. We used a script separate from the core application to schedule the appropriate files for download from the USPS servers weekly. The script can also handle download upon demand. New files will be downloaded automatically when available and placed into a test environment which is also separate from the production environment. Endicia will used this test environment to assure that the program is compatible with the new data, ensuring any changes within the format of the data does not cause Endicia down time. The JAR generated by Maven will automatically run the provided test suite. If the tests fail, the Endicia staff will be notified that an error has occurred and human intervention will be necessary; this is the only case in which the process is not automated, as fixing an error of such magnitude will most likely require changes to the code base.

Once the new data is assured to work with the code by the aforementioned quality testing, the data files on the production server will be deleted and replaced by the newly updated files. The replacing can be done while the program is running because file I/O is performed only during server initialization. It is at this point that our program will be notified by a secure connection that it can begin constructing the new database. Specifics on how the database update is implemented are described in Section 5.3.1.

Section 4.9 Testing Methodology

Testing is an important part of the development life cycle. As opposed to repeated manual tests, we implemented regression testing for the majority of the program. As such, if any aspect of the data changes, or for some reason the code base needs to be changed, any disruption of service will be identified by future Endicia maintainers before the change enters a production environment and causes loss of money for the company. JUnit is used primarily for regression testing. Maven also runs these JUnit tests each time it builds the project.

Regression testing is useful for detecting obvious errors that may cause system downtime, but it is hard to make tests for a program which may return different results as the data evolves. Diff tests, comparing the output, between the USPS equivalent service and the Endicia application were useful for our team to assure accurate results. The USPS document also walks through a few examples, and we referenced these in combination with a debugger to ensure the program follows the correct logical flow.

It was a requirement for us to ensure the application met a performance baseline. We did this by using the top layer's timing of web requests, NGINX being that top layer. NGINX is capable of logging how long requests to the application take. This, in combination with scripts able to send large volumes of requests are used to identify if the program handles requests in a timely fashion.

Finally, an application called JConsole is provided with JMX, the Java Management eXtension mentioned in Section 4.3.4. JConsole allows monitoring of the memory used by the JVM, or Java Virtual Machine. The JVM controls all memory used by the application. JConsole produces data that describes how the JVM deals with memory management over long periods of being bombarded with requests. This allows any memory problems that may occur to be highlighted before they cause any issues in a production environment.

Section 4.10 Deployment

The deployment lifecycle of the project has several stages. First, Maven is used to build an executable of our project. In this case, this will be a standard .jar file, which is an executable that can be run in the standard JVM on any machine.

Jenkins is used to test the output of Maven in continuous integration style. This means that Jenkins will automate the build process, and let any developers and relevant business personnel know if any new changes break the build process. A break of the build would occur if any one of the regression tests were to fail when Maven builds the JAR.

After Jenkins has assured that the new build will not crash the production servers, the JAR executable can be used inside of a Docker container. As mentioned previously, the Docker container will provide an application-level virtual machine as opposed to an operating system grade virtual machine. The Docker will allow easy deployment on a server, essentially reducing configuration to the single JAR file.

The Docker container will then be deployed to a server, and the application can be treated as standalone. Finally, Endicia’s other systems can interface with the server housing the Docker container. This concludes the discussion of deployment as we understand the process, but Endicia may very well change this process as they move forward. Refer to Section 8.1, Future Works, for a bit more context.

Chapter 5 Implementation Details

This Chapter discusses the specifics of some key implementations. Section 5.1 discusses the API Endicia used, as provided by USPS, prior to our server implementation. Section 5.2 introduces details about our API as well as the details surrounding a request and response. Section 5.3 outlines some of the decisions we made about the server's codebase when creating the server. Finally, Section 5.4 talks about how our server handled edge cases and errors.

Section 5.1 USPS Interface

At the onset of the project, the USPS interface was a bit of a mystery. This USPS interface is the "webtools" API. A few sets of sample input/output were provided, but the primary mode of use and behavior of the interface were unknown. This section details the information discovered about the ways the service being recreated responds to input.

Section 5.1.1 Using the current webtools API

We asked around Endicia and found the server to point our program to so we could access the webtools API, the API that our program is being developed to replace locally, which is currently used by Endicia services. The current webtools API, according to USPS teams that we talked with, is messy with legacy compatibility requirements. Our team has been the first to recreate this webtools API, as mentioned in Chapter 3, and the USPS team recommended that we make an effort to trim down the verbose output of the legacy system.

Our team was given Visual Studio projects to test the output of this API with Diff tests in mind. A Diff test displays the differences between two pieces of text. It was a good baseline to ensure we had the relevant information calculated and to verify certain results, but Diff tests were not possible as it was clear that the webtools API sought compatibility with far more requirements than our project was given. These discrepancies, along with some ambiguity in the USPS-provided documentation slowed development significantly. The following section, Section 5.2, goes into detail on how the API we have developed should be accessed. There will be no further discussion of the USPS webtools API, but interested readers can find more information at the API's documentation webpage. [24]

Section 5.2 New API Details

This section discusses the inputs and outputs to our application as well as the URLs needed to access our API. More information concerning the procedure for using the application can be found in the Users' Documentation found in Appendix A.

Section 5.2.1 Input

This section details the type of requests one can make to the API our program provides. The program requires specific inputs, which are explained later in this section. Some examples of requests are also provided.

The program we developed offers two modes of input, listed below. Both forms of input require the same parameters.

1. POST request with relevant parameter in an XML document
2. GET request with relevant parameters as query parameters in the URL.

The base URL for requests is `http://[IP address]/webtools/v1/` Our program currently offers one service through the API, located at the path /mail off the base URL, so at `http://[IP address]/webtools/v1/mail/`. This exposes the modular architecture made to allow easy extensions for additional output. For example, all one would have to do is to create a new class and specify a /newresponse path off of the base path.

Table 1 details the input required for the program. The parameter names are required and case sensitive for both GET requests and POST requests. We made this decision because RFC 3986 [13] states that the query parameters of a URL should be case sensitive. Some parameters are optional, as indicated by column 2. In addition, the parameters must be in a specific format for the program to accept, as validation must be done to ensure valid user input. The description field details anything else a user should know about the API with regard to that specific parameter.

Parameter Name	**Required**	**Parameter Type**	**Restrictions**	**Description**
Originzip	Yes	String	5 or 9 digits	ZIP code where the package originates
Destzip	Yes	String	5 or 9 digits	ZIP code where the package is destined for
Dropofftime	Yes	Integer	0000-2359	The time at which the package was dropped off
Mailclass	Yes	String	Must be one of the following: PER, PME, FCM, STD. PKG, PRI.	The mail class with which the package is being shipped. PME and PRI both result in outputs different from the other classes.
Desttype	Yes	Integer	1-3	This is the destination type. 1 corresponds to Street Address 2 corresponds to a PO Box 3 corresponds to Hold for Pick Up (HFPU).

				If 3 is specified, an HFPU location will also be returned.
Shipdate	No	Date	Must be in format dd-MMM-yyyy, ex 01-Jan-2016	The date which the package is dropped off. This defaults to the current date if not specified.
Deliveryoption	No	Integer	0-7	This specifies if the package should not be delivered on certain days. The default value is 0 if none is specified. 0: Omit no days 1: No Saturday delivery 2: No Sunday delivery 3: No weekend delivery 4: No holiday delivery 5: No Saturday or holiday delivery 6: No Sunday or holiday delivery 7: No delivery on Saturday, Sunday, or Holidays.

Table 1: A table of the parameters the mail API requires as input

Figure 8 is an example of a request made using a GET request, otherwise known as a request via a URL. Note that the optional parameter “shipdate” is specified but “delivery option” is omitted. The order of the parameters does not matter.

```
http://{IPAddress}/webtools/v1/mail/?originzip=32669&destzip=816
54&dropofftime=1000&mailclass=PME&desttype=3&shipdate=01-Jan-
2016
```

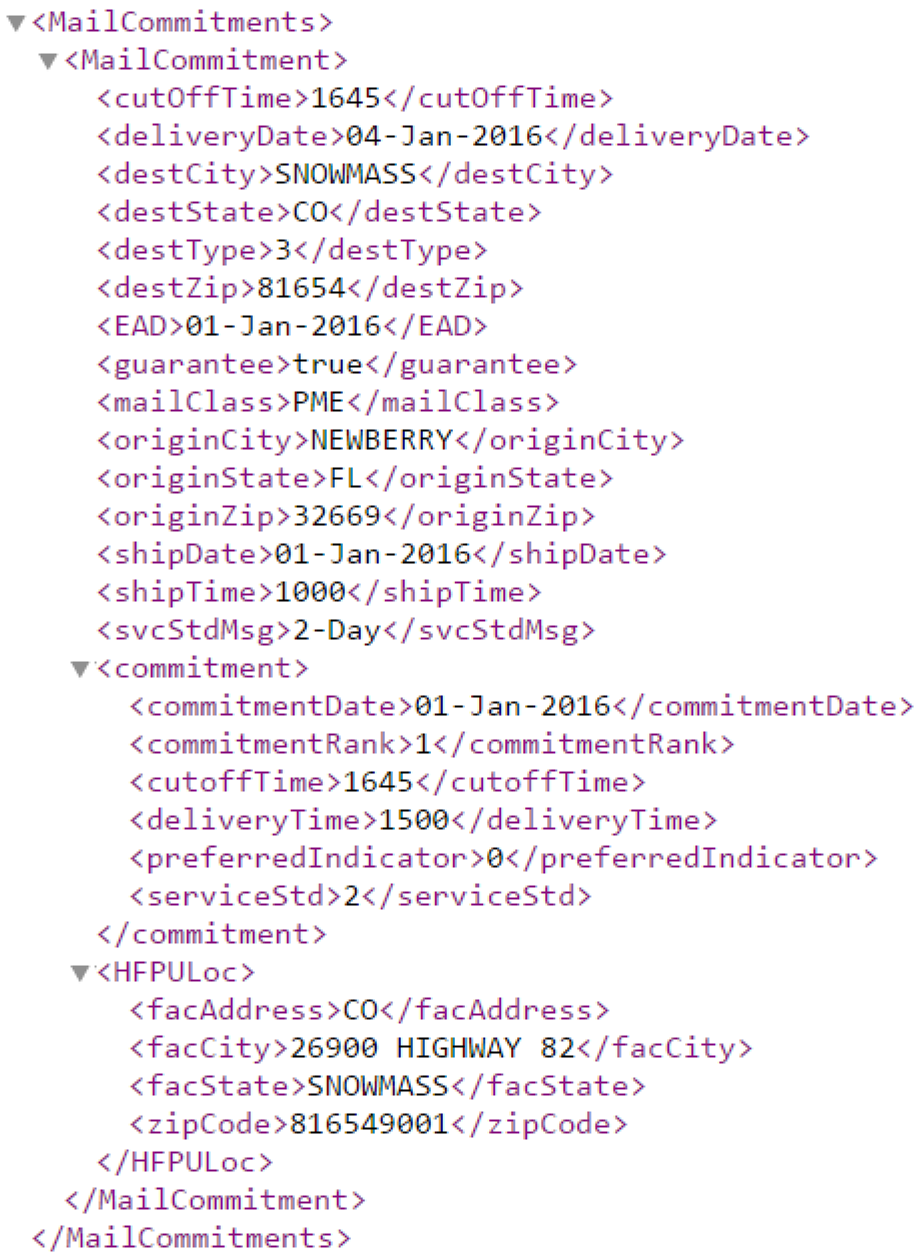
Figure 8: A sample request using the default option for return type

We made sending a POST request very similar to the GET request described in Figure 8. For a POST request, a user needs to specify their HTTP “Content-type” header to the “application/xml” value. This tells the server that the incoming request contains an XML based body. All of the required and optional fields are the same between a GET and POST method request. Figure 9 is an example of a POST request xml body with all of the fields specified except for the “deliveryoption” field. The URL of which to send the request is also illustrated in this figure. The POST requests can also specify the return type of xml or json in the same way as described for the GET methods.

```
http://{IPAddress}/webtools/v1/mail.xml
```

Figure 9: Sample POST request

Section 5.2.2 Output



```
<MailCommitments>
  <MailCommitment>
    <cutOffTime>1645</cutOffTime>
    <deliveryDate>04-Jan-2016</deliveryDate>
    <destCity>SNOWMASS</destCity>
    <destState>CO</destState>
    <destType>3</destType>
    <destZip>81654</destZip>
    <EAD>01-Jan-2016</EAD>
    <guarantee>true</guarantee>
    <mailClass>PME</mailClass>
    <originCity>NEWBERRY</originCity>
    <originState>FL</originState>
    <originZip>32669</originZip>
    <shipDate>01-Jan-2016</shipDate>
    <shipTime>1000</shipTime>
    <svcStdMsg>2-Day</svcStdMsg>
    <commitment>
      <commitmentDate>01-Jan-2016</commitmentDate>
      <commitmentRank>1</commitmentRank>
      <cutoffTime>1645</cutoffTime>
      <deliveryTime>1500</deliveryTime>
      <preferredIndicator>0</preferredIndicator>
      <serviceStd>2</serviceStd>
    </commitment>
    <HFPULoc>
      <facAddress>CO</facAddress>
      <facCity>26900 HIGHWAY 82</facCity>
      <facState>SNOWMASS</facState>
      <zipCode>816549001</zipCode>
    </HFPULoc>
  </MailCommitment>
</MailCommitments>
```

Figure 10: The output of the sample request in Figure 8

Figure 10 shows the response to Figure 8. The details of the structure of an output are explained in Table 2, broken down by each element in the order that they appear. The conditional inclusion field specifies whether or not the field is present in the output conditionally; “No” if the fields is always present, “Yes“, followed by the input upon which it is conditional otherwise.

Element Name	**Element Type**	**Description**	**Conditional Inclusion**
cutOffTime	Integer	The cut off time at which facility accepts the mail.	No
deliveryDate	Date	The date the package will arrive at the destination.	No
destCity	String	The city the package is being sent to based on the destZip	No
destState	String	The state the package is being sent to based on the destZip	No
destType	Integer	The destination type originally input by the user	No
destZip	String	The destination ZIP code originally input by the user	No
EAD	Date	Effective acceptance date	No
guarantee	Boolean	Whether a guarantee is present	No
mailClass	String	The mail class originally input by the user	No

originCity	String	The city the package is sent from based on the origin ZIP	No
originState	String	The state the package is sent from based on the origin ZIP	No
originZip	String	The origin ZIP code originally input by the user	No
shipDate	Date	The date the package was shipped on, originally input by the user	No
shipTime	Integer	The time the user reported dropping the package off at, originally input as dropofftime	No
svcStdMsg	String	The service standard message generated from USPS data based on the input mail class	No
commitment	Commitment	A commitment containing info on the Priority Mail Express (PME) shipping details	If PME was the input mailclass
PRILocation	List of Location	There can be any number of these nodes, and they each contain a Location where drop off is possible	If PRI was the input mailclass
HFPULoc	Location	The location where the package will be held for pick up	Present if the input desttype is 3, aka HFPU

Table 2: A table of the fields returned in an output.

As Figure 10 shows, the output of a request is by default xml, but a user can specify either json or xml explicitly by modifying the URL. Figure 11 shows how to construct a URL that requests json output.

```
http://{IPAddress}/webtools/v1/mail.json/
```

Figure 11: A sample request using the json option for return type

Allowing the user to request a specific format of the server's response is a key part of a RESTful Web Service. We chose to allow the user to specify the return type by adding the type extension to the end of a URI Path. We chose this method to adhere to an industry standard; sites such as Twitter allow their users choose a response's content type in this manner. It is also standard to include a version number in the URL Path to let the user know which version of the API they are accessing. This removes any confusion between versions and provides a more user friendly experience. [8]

Section 5.3 Code Structure

While creating classes to do calculations, we designed our code structure to be as robust as possible to adapt to potential future changes in data files such as the number of records. For example, when looking for which dates are holidays, we decided to iterate over the file to discover how many holidays exist, as opposed to pulling out a pre-set range of row numbers. This section details other code-level decisions made during implementation.

Section 5.3.1 Endicia SLA Requirement - Database Design

Endicia has a service level agreement (SLA) with their clients that guarantees a maximum of five seconds of downtime for an entire year. As stated in Chapter 3, this was a major requirement that influenced the design of our project. Due to this limitation, a database was implemented with a structure that could update weekly without downtime. As a result of this SLA, it was a necessity that the database can be loaded into memory in the background while requests can still be serviced.

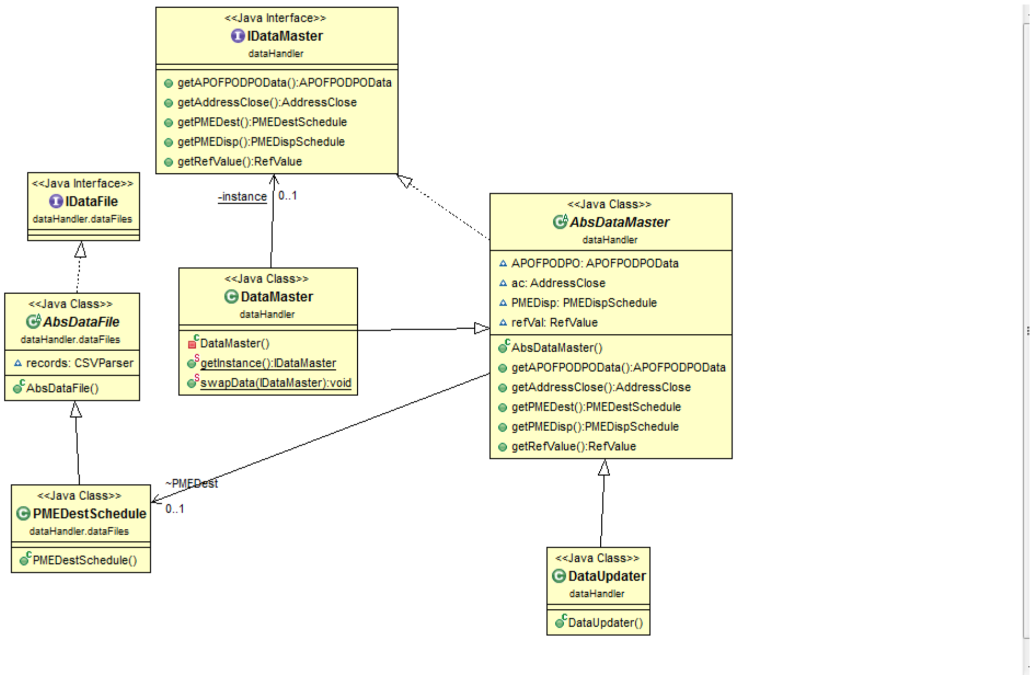


Figure 12: UML Class Diagram denoting the architecture of the database

As Figure 12 denotes, the database was implemented using a singleton pattern. This pattern allows the database to be updated once a week by a daemon thread. Updating is done by having the daemon thread request files from a local Endicia server and download any files that have been updated. Then the daemon creates an instance of the “DataUpdater” class using the new files. This class has the same constructor as the current in-memory database class: “DataMaster”, as they share the same Interface and abstract class. Once the new files have been loaded into memory, the “DataUpdater” class calls the “swapData” method that changes the pointers of the current singleton database, “DataMaster”, to have the contents of the new files. Any request made at this point will use the get instance method defined in “DataMaster” and utilize the newly updated files. This pattern allows for zero downtime while servicing user requests with the most up to date data.

The singleton pattern is also ideal for making static data retrievable across the code base, while enforcing the rule that only one database exists at a time. This prevents the database from being instantiated at any time other than when the server first loads. The reason for this removal of File I/O is that the load the server has to handle is somewhere between fifty to two hundred requests per second. File IO is one of the most costly operation in terms of time. Hence, all of the file IO is done at time the server loads, or while the database is being updated in the background, and never during a request.

Section 5.3.2 Algorithm Flowchart Implementation

The ATF main logic flow chart, as provided by the United States Postal Service, asks “What is the mail class of the input?” three times. Instead of doing that, we split the flow chart into three branches, PME Subroutine, PRI Subroutine, and Non-PME Non PRI Subroutine. The logic of PRI Subroutine and other Non-PME subroutine does not have much difference so they merge together quickly in our logical flow. The split could help us to organize our code so that subroutines do not mix together.

Figure 13 displays the original flow chart. As mentioned, there are many branches due to the program repeatedly asking which mail class was inputted by the user. In our implementation, shown by Figure 14 we choose to use an object oriented approach where the created object implicitly knowns which mail type it is processing due to the methods it overrides from the abstract superclass; this is known as the Strategy Pattern. This allowed us to cut down on duplicated code, and simplify the structure of the main algorithm slow significantly.

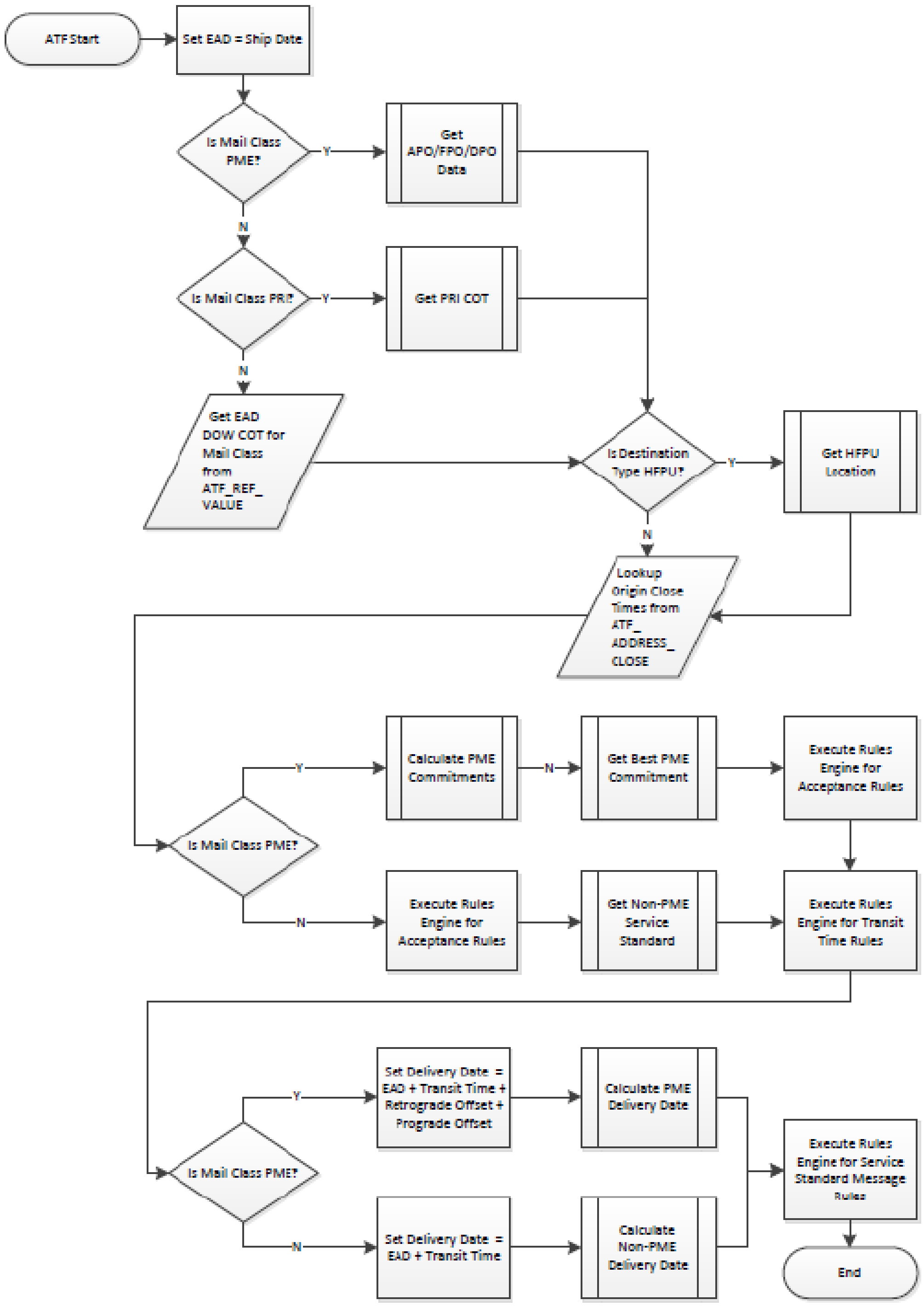

ATF Start
Set EAD = Ship Date
Is Mail Class PME?
Y
Get APO/FPO/DPO Data
N
Is Mail Class PRI?
Y
Get PRI COT
N
Get EAD DOW COT for Mail Class from ATF_REF_ VALUE
Is Destination Type HFPU?
Y
Get HFPU Location
N
Lookup Origin Close Times from ATF_ ADDRESS_ CLOSE
Is Mail Class PME?
Y
Calculate PME Commitments
N
Get Best PME Commitment
Execute Rules Engine for Acceptance Rules
N
Execute Rules Engine for Acceptance Rules
Get Non-PME Service Standard
Execute Rules Engine for Transit Time Rules
Is Mail Class PME?
Y
Set Delivery Date = EAD + Transit Time + Retrograde Offset + Prograde Offset
Calculate PME Delivery Date
N
Set Delivery Date = EAD + Transit Time
Calculate Non-PME Delivery Date
Execute Rules Engine for Service Standard Message Rules
End

Figure 13: ATF Main Logic (Provided by USPS)

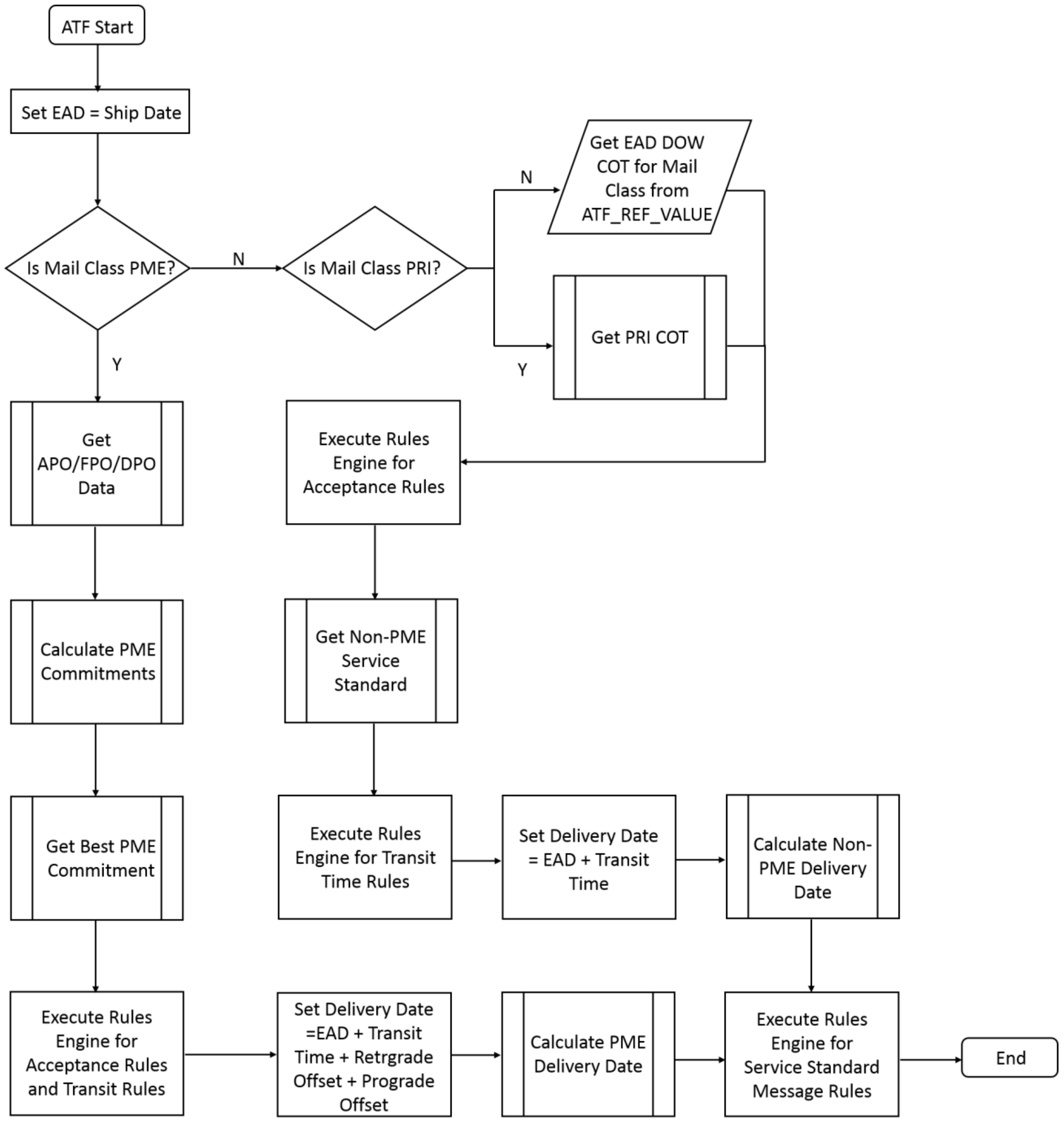

ATF Start
Set EAD = Ship Date
Is Mail Class PME?
N
Is Mail Class PRI?
N
Get EAD DOW COT for Mail Class from ATF_REF_VALUE
Y
Get PRI COT
Y
Get APO/FPO/DPO Data
Execute Rules Engine for Acceptance Rules
Calculate PME Commitments
Get Non-PME Service Standard
Get Best PME Commitment
Execute Rules Engine for Transit Time Rules
Set Delivery Date = EAD + Transit Time
Calculate Non-PME Delivery Date
Execute Rules Engine for Acceptance Rules and Transit Rules
Set Delivery Date =EAD + Transit Time + Retrgrade Offset + Prograde Offset
Calculate PME Delivery Date
Execute Rules Engine for Service Standard Message Rules
End

Figure 14: Flow Chart Split

Section 5.4 Error Handling

Robust error handling was a requirement of the application. It is of the utmost importance that the application have no downtime, as mentioned earlier, so runtime faults were inexcusable. As such, the program needed to gracefully handle all forms of user input and any internal calculation errors. It was also desirable to differentiate between user errors and errors caused by faults in internal logic. With such a differentiation the user could receive appropriate feedback if their query was malformed or if they had used invalid values for a parameter, and any internal logic errors would not give excessive information to an attacker about internal code structure. Finally, this differentiation allows a logging infrastructure to log important code bugs while giving less importance to errors caused by users providing inappropriate input.

Custom exceptions were implemented for any errors possibly caused by the user, and the program recognizes such errors and gives the user a verbose error message as to the perceived cause. If a non-custom exception is thrown, the message will be saved to a system log for further inspection by an Endicia staff member.

Chapter 6 Optimization

This Chapter discusses the ways in which we increased the performance of our server application. Since the server must be able to satisfy between 50-200 requests per second, or 5 million-15 million requests a day, as detailed in Chapter 3, we looked at all possible forms of optimization. As stated, we made the design decision early on to remove costly file I/O from the runtime of the main algorithm. All file I/O is done at initialization, meaning that the data is entirely in speedy RAM memory. Once this optimization was made, we moved to optimizing the in-memory file access. Section 6.1 outlines the caching infrastructure added to the codebase in order to reduce the time the main algorithm takes to process data and execute. Section 6.2 displays the various changes we made to the Java Virtual Machine such that it is optimized for our application. Finally, Section 6.3 details the tuning done to the Linux Operating System via the command line so it is better configured to handle server-level loads. This chapter refers heavily to Chapter 7, which is where the results and corresponding graphs are located and explained. More information regarding the maintenance of the application can be found in Appendix B.

Section 6.1 Caching

Throughout the development lifecycle, we maintained a list of potential future optimizations that would increase the performance of our application. This is the highest level, in terms of program stack, of the optimization changes that we made. We used a Java profiler introduced in Section 4.3.3 to test which portions of our code were bottlenecks, and aligned those results with our prioritization of optimizations. A profiler attaches to the Java Virtual Machine (JVM) and displays the amount of time each method took to execute in the CPU. Through the execution time, we were able to see that our largest bottleneck was due to the in memory data access of files obtained by the United States Postal Service. As described in Section 4.2, these data files have thousands of records that must be iterated through to perform calculations. To reduce iterations through this data, we created various HashMaps to store every record. In doing this, we reduced the time complexity of each “get” operation from O(n) to O(1).

This means the algorithm originally ran in linear time with respect to n, the number of records (or lines) in each file. Caching made the equivalent of a lookup table, reducing the lookup time to a single pointer lookup operation, O(1). The effect in terms of response time is detailed in Chapter 7, Figure 22 and Figure 23.

Section 6.2 JVM Tuning

The Java Virtual Machine, JVM, is the environment in which all Java applications are run. To further increase the performance of our application, we looked into a few of the JVM settings. One of these is the garbage collector. A garbage collector is used to remove any objects from main memory that are no longer needed by a Java application. The removal of objects keeps the memory footprint of an application as low as possible.

While reducing the amount of memory required to run our program was not a primary concern, it was something that we kept note of during the development lifecycle through monitoring tools such as JConsole. JConsole allowed us to see how much memory the program needed to be allocated via the JVM command line such that the program didn’t use up all memory available.

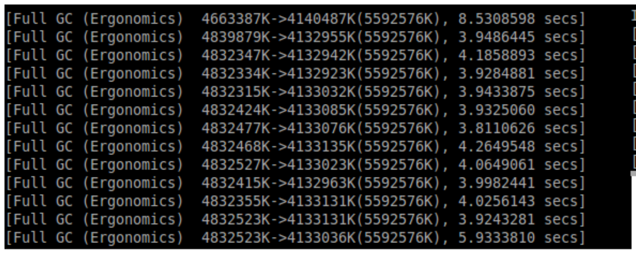
Table 3 displays the various changes we made to the JVM for the purpose of optimization. These JVM options are essential for the program to properly initialize and run without periodic performance spikes or out of memory errors.

JVM Argument	Description
`-server`	Runs the JVM in server mode. Usually a default option for the JVM.
`-Xms5g`	This is the starting memory the JVM is allocated. In this case, it is set to 5GB. The program takes around 4GB in its resting state, with a bit more during high load times.

`-Xmx10g`	This is the memory cap the JVM is given, in this case 10GB. This is necessary if the data updater is used, as the program creates another database, swaps the pointers, and then garbage collects the old database.
`-XX:+UseConcMarkSweepGC`	This selects the Concurrent Mark Sweep garbage collector for use by the JVM. This option is necessary to prevent large “stop the world” performance degradation.

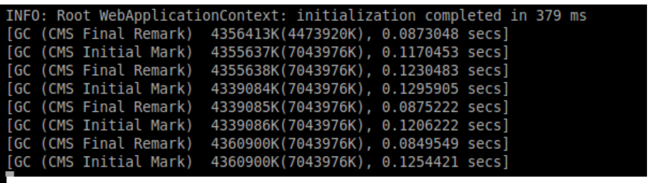
Table 3: JVM Command Line Options

During performance testing, we noticed that the response time of a single request would drastically increase whenever the JVM performed a garbage collection. We then began looking into the various garbage collectors provided by the JVM. We found that the default garbage collector, Parallel GC, uses multiple threads to remove unused objects from memory. While useful for many Java application with small memory footprints, this collector had a significant downside. Whenever it would collect garbage, it resulted in a state known as “Stop the world,” meaning all of the threads running the application would cease until collection was complete. This caused very large spikes in response time. Due to massive reduction in performance, detailed in the following chapter, we switched from the default garbage collector to the Concurrent Mark Sweep collector. This collector would run threads in parallel with the application to remove any unused objects [2]. Using this collector increased the response time of a request by a few milliseconds, however it removed the “stop the world’ issue, ultimately reducing the variance of response times drastically. This option to the JVM can be seen in Table 3 The difference between the performance of the two garbage collectors is showcased by Figure 15 which shows that the Parallel GC takes upwards of 5 seconds to perform a “stop the world” garbage collection.



```
[Full GC (Ergonomics)  4663387K->4140487K(5592576K), 8.5308598 secs]
[Full GC (Ergonomics)  4839879K->4132955K(5592576K), 3.9486445 secs]
[Full GC (Ergonomics)  4832347K->4132942K(5592576K), 4.1858893 secs]
[Full GC (Ergonomics)  4832334K->4132923K(5592576K), 3.9284881 secs]
[Full GC (Ergonomics)  4832315K->4133032K(5592576K), 3.9433875 secs]
[Full GC (Ergonomics)  4832424K->4133085K(5592576K), 3.9325060 secs]
[Full GC (Ergonomics)  4832477K->4133076K(5592576K), 3.8110626 secs]
[Full GC (Ergonomics)  4832468K->4133135K(5592576K), 4.2649548 secs]
[Full GC (Ergonomics)  4832527K->4133023K(5592576K), 4.0649061 secs]
[Full GC (Ergonomics)  4832415K->4132963K(5592576K), 3.9982441 secs]
[Full GC (Ergonomics)  4832355K->4133131K(5592576K), 4.0256143 secs]
[Full GC (Ergonomics)  4832523K->4133131K(5592576K), 3.9243281 secs]
[Full GC (Ergonomics)  4832523K->4133036K(5592576K), 5.9333810 secs]
```

Parallel Garbage Collector



```
INFO: Root WebApplicationContext: initialization completed in 379 ms
[GC (CMS Final Remark)  4356413K(4473920K), 0.0873048 secs]
[GC (CMS Initial Mark)  4355637K(7043976K), 0.1170453 secs]
[GC (CMS Final Remark)  4355638K(7043976K), 0.1230483 secs]
[GC (CMS Initial Mark)  4339084K(7043976K), 0.1295905 secs]
[GC (CMS Final Remark)  4339085K(7043976K), 0.0875222 secs]
[GC (CMS Initial Mark)  4339086K(7043976K), 0.1206222 secs]
[GC (CMS Final Remark)  4360900K(7043976K), 0.0849549 secs]
[GC (CMS Initial Mark)  4360900K(7043976K), 0.1254421 secs]
```

Concurrent Mark Sweep Garbage Collector

Figure 15 Garbage Collector Time Comparison

On the other hand, Concurrent Mark Sweep garbage collection takes less time by an order of magnitude; roughly less than 0.1 seconds. The effect this had on response time performance is detailed in Chapter 7 in Figure 17 and Figure 19

Section 6.3 Linux Tuning

As a final step of optimization, we looked into the network settings provided by the Ubuntu Server Operating System. We found that the configurations of the kernels networking parameters allowed for improved response time. These options are more tunable to the exact production environment than the strict requirement of the JVM options in Table 3. The performance results from this can be viewed in Chapter 7. A list of the values we changed to obtain these results can be seen in Table 4. Note that each command's start can be identified by "`sysctl -w`".

Linux Command Line Argument	Description

`sysctl -w net.core.rmem_max=16777216`
 	
`sysctl -w net.core.wmem_max=16777216`
	These commands change the network read (rmem) and write (wmem) buffers capacity.

`sysctl -w net.ipv4.tcp_rmem="4096 87380 16777216"`	These commands change the tcp layer's read and write buffer size. Look to further

`sysctl -w net.ipv4.tcp_wmem="4096 16384 16777216"`	tuning of these parameters and the ones above if bufferbloat becomes a problem.
`sysctl -w net.core.somaxconn=4096`	This changes the size of the connection listening queue at the TCP level. This is a very tunable option.
`sysctl -w net.core.netdev_max_backlog=16384`	This is the size of the incoming packet queue for java-layer processing.

`sysctl -w net.ipv4.tcp_max_syn_backlog=8192`
`sysctl -w net.ipv4.tcp_syncookies=1`
	The first command raises the queue size allowed for processing syn packets, otherwise known as the TCP message to open a connection. The second option protects against a DDoS attack known as syn flooding.

`sysctl -w net.ipv4.ip_local_port_range="1024 65535"`
`sysctl -w net.ipv4.tcp_tw_recycle=1`
	The first command changes the range of usable ports to almost the entirety of the available port range for typical hardware. The second command allows the kernel to reuse sockets more efficiently.

`sysctl -w net.ipv4.tcp_congestion_control=cubic`	This command changes the network congestion algorithm the kernel uses. In the case of this application, cubic is selected which is an algorithm designed for high throughput networks. This parameter has a relatively high effect on performance.

Table 4: Linux Kernel Options

Chapter 7 Results

Throughout our optimizations, we kept record of the response time, as well as several other statistics about the performance of our application. This chapter serves as a walkthrough of the optimizations detailed in Chapter 6 in the form of graphs and statistics. The chapter is structured as a story of the results as we obtained them on the path to reach our final optimized state. Accordingly, each figure represents an optimization, and figures are arranged in the chronological order in which the corresponding optimization was made.

Before we dive into the results, we will provide an explanation of the layout of the figures in this section. All graphs detail the response time of a large number of requests made. The y-axis is in milliseconds, and the x-axis consists of timestamps. At the bottom of the graphs, in blue, statistics can be found. All graphs will have their axis as such, and all graphs will include statistics at the bottom. The statistics consist of the sample size, throughput of requests, the deviation of response time, the average response time, and the median response time. The median response time and throughput will be used instead of average in most cases to adjust for outliers. All graphs were made using the tool JMeter, which can send requests from any number of threads, each thread simulating a user making requests. Unless otherwise specified, one thread (simulating one user) was used to make the graph. In addition, the requests were made inside an infinite loop that was terminated when an adequate result and sample size was obtained. It serves to note that all requests were made with the same query parameters to limit the number of variables. The input string was selected to maximize the amount of code the request hit; the mail class PME and the Hold for Pick Up destination type both result in additional output, so these were chosen. The query string is as follows

`“/?originzip=32669&destzip=81654&dropofftime=1000&mailclass=PME&desttype=3”.`

When making a web application, it becomes necessary to separate the performance overhead of network latency and the tools used in the server to receive and response to messages from the performance of the application itself. Figure 16 displays the null operation test. This test was performed by specifying a path that did not perform any logic, simply returning a hard coded string to the requesting user. This is the only request that does not use the aforementioned constant query parameters. The response time of such a request is shown to have a median of 1 ms. This means every future result, except those that use multiple threads, includes an extra 1 ms of overhead for the request to be sent over the network and for the base server tools to process. The massive response time spikes shown are an anomaly that we did not understand at first, but they were understood when the optimization made at the time of Figure 19 was done.

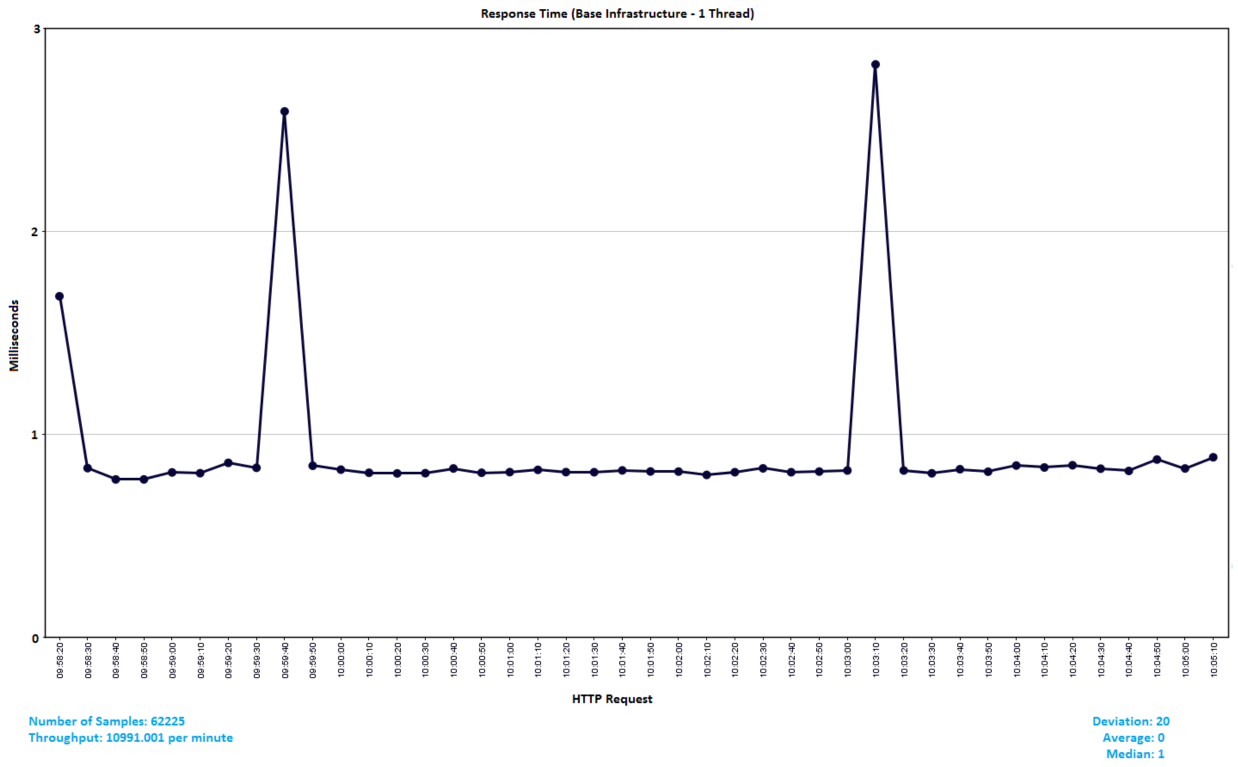


Figure 16: Base Infrastructure Response Time

The next performance test displayed in Figure 17 was to observe the upper bound of a single thread making the largest possible request, before any optimizations were made. We observed the median response time to be 112 ms, with a throughput of 523 requests/min = ~8 requests/second. This was far below our requirement benchmark, so we had a lot of optimizations left, and we still had to figure out what the up to four-fold performance spikes were caused by.

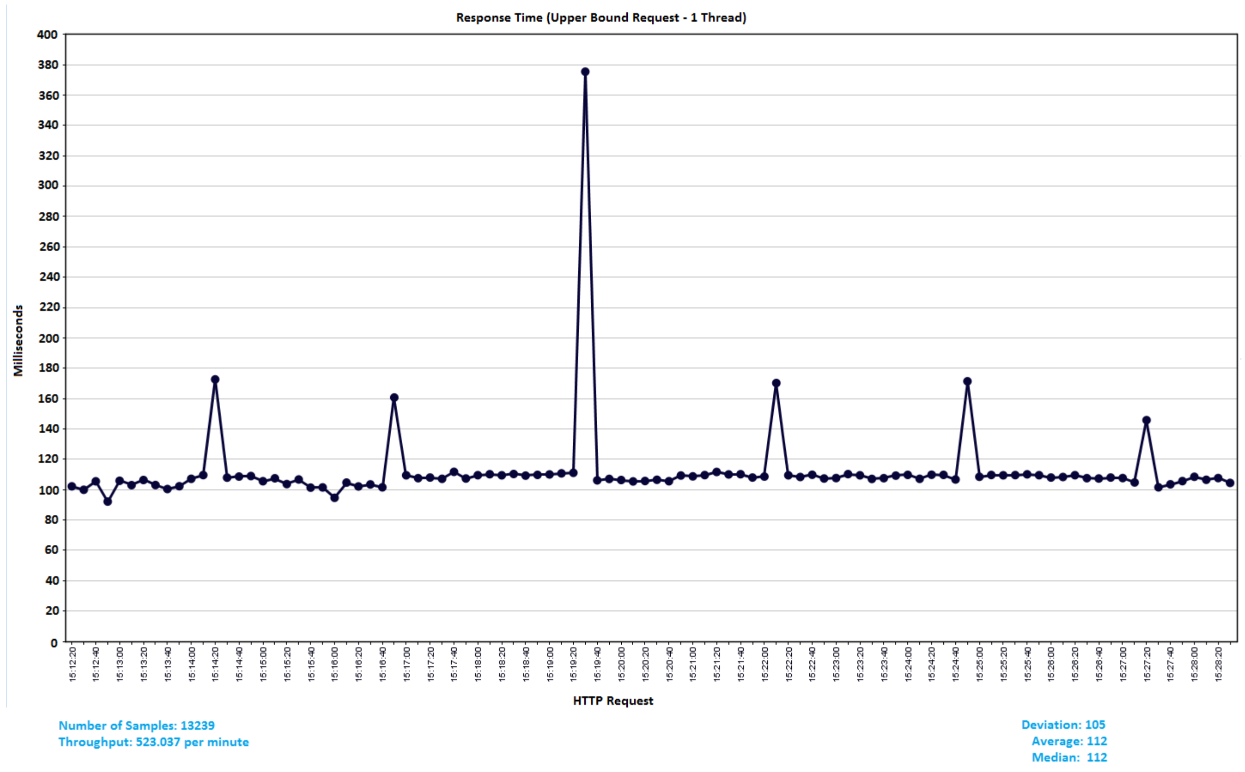


Figure 17: Upper Bound Request Response Times

After this baseline, we researching why there were massive spikes in performance. In our research, we found several tools for analyzing performance, one of them named JConsole. JConsole allowed us to monitor the memory usage of the JVM as the program was running. We found that the JVM uses memory and garbage collects in a saw-tooth graph, indicated by Figure 18; the same saw-tooth pattern shown by response time in the previous two figures. In the case of the JVM, a saw-tooth pattern indicates a large increase in the amount of memory, followed by a swift garbage collection. In Figure 18, the y-axis indicated memory usage and the x-axis is time.

We inspected the timestamps, and found that these “stop the world” garbage collection spikes corresponded to the response time spikes in our application! It was at this stage that we researched alternative garbage collectors (GC). The JVM offers alternative garbage collectors to the default, available via a command line option when the JVM is run. The default garbage collection algorithm is Parallel GC, which means the garbage collection runs periodically. This results in a massive 1+ GB garbage collection that stops the program. On the other hand, the Concurrent Mark Sweep (CMS) runs in concurrently to the application, resulting in slightly higher response times but better performance overall. Parallel GC is better for smaller application that don’t require much heap memory during runtime, while CMS is much better for programs that use large amounts of memory, such as ours.

Figure 19 shows the result of switching to this new GC; the relatively same response time as Figure 17, but without the spikes. There is a slight increase in response time however; the median rises by 12 ms. This 10.7% increase can be explained by the fact that the CMS garbage collector runs concurrently with the application, causing slightly reduced performance across the board, instead of making massive periodic spikes in response time. The noise at the beginning of Figure 19 is simply the start up time, unrelated to performance over-time.

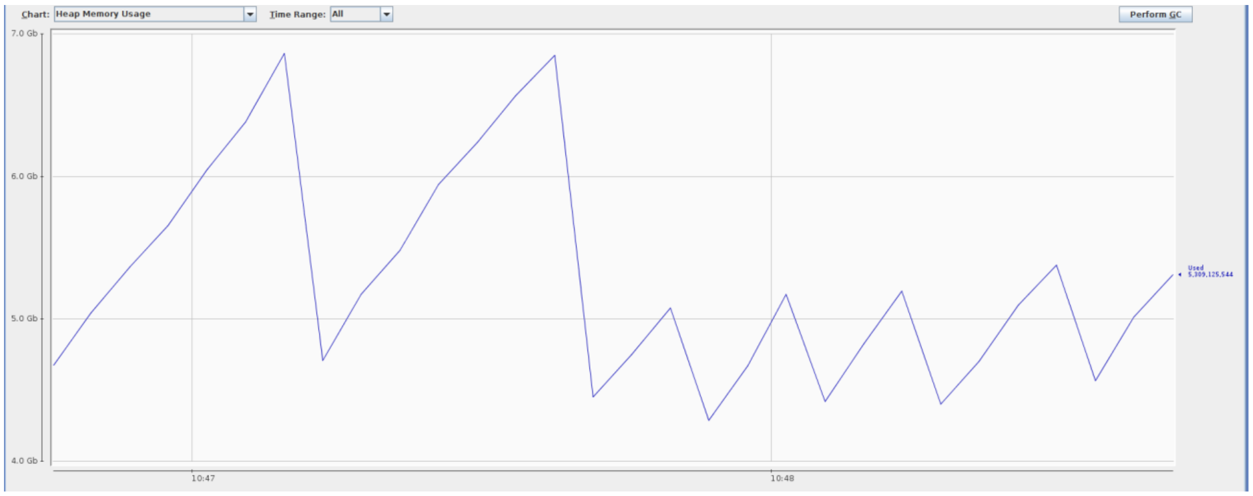


Figure 18: JVM memory usage with Parallel GC

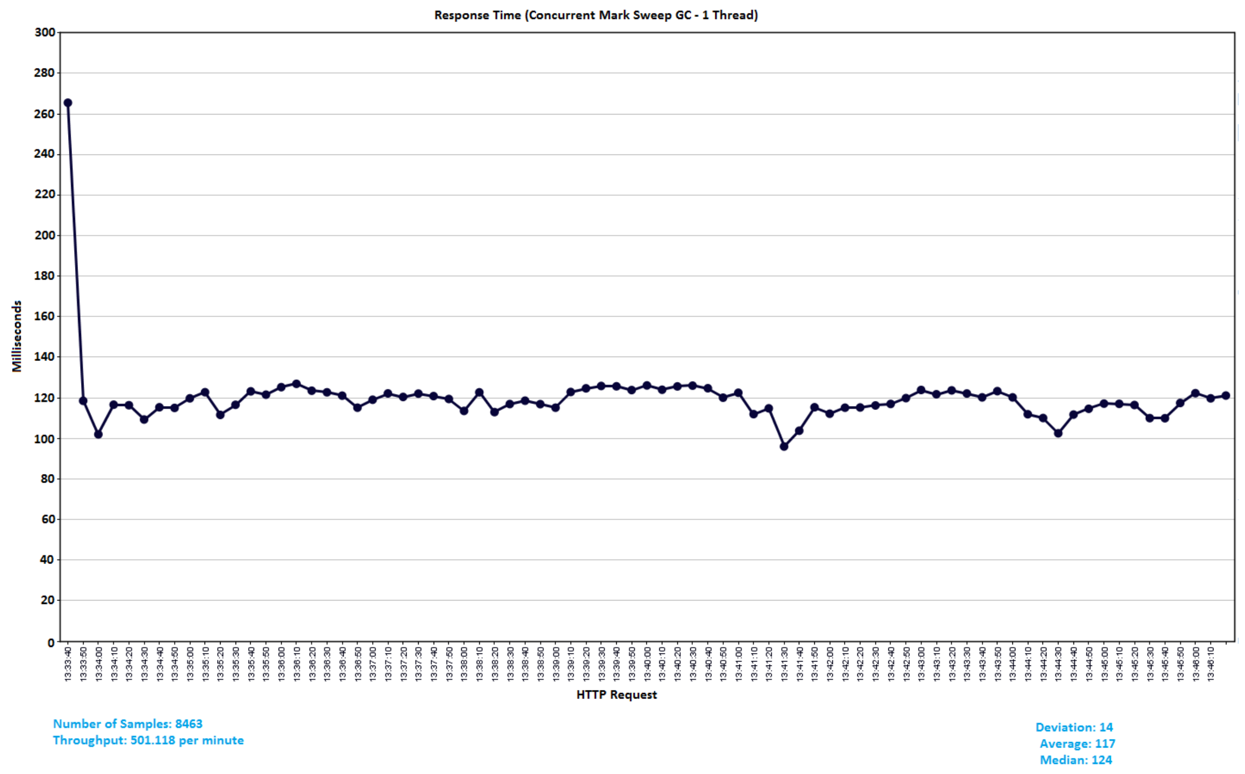


Figure 19: Concurrent Mark Sweep Response Times

Next, we decided to see how much throughput our application could produce using multiple threads to send requests before further optimizations. Figure 20 shows the response time with 20 requesting threads. The throughput increased to 2182 requests/min = ~36 requests/sec. This is about a 4x increase in throughput, which makes sense due to with the fact that the machine we were testing this on has 4 cores. Still, the response time median of 541 ms was abysmal. Further optimization was needed.

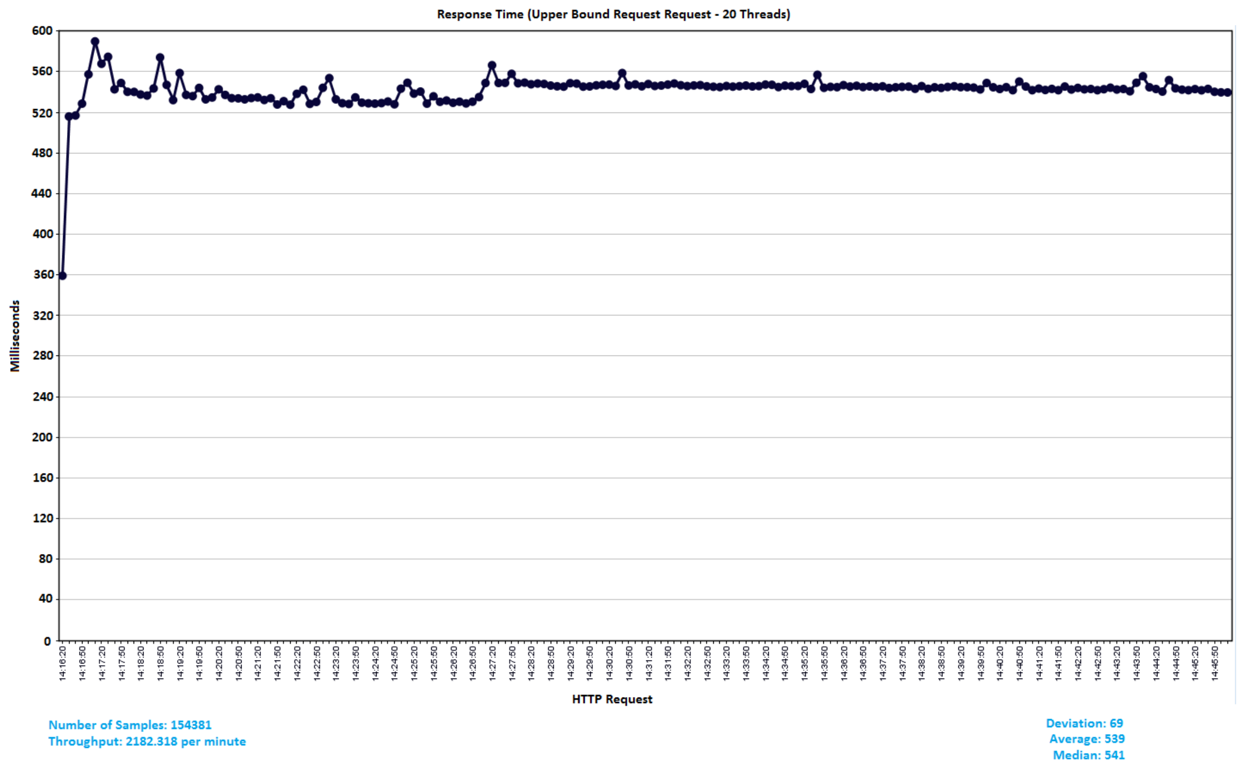


Figure 20: Upper Bound Request Response Times using 20 Threads

Once this test was done, we wondered if perhaps the performance issues stemmed from the operating system being configured poorly for high-load traffic. We researched a bit, and found the Linux command line options detailed in Table 4 in Chapter 6. In short, we increased the buffer sizes for packets at the TCP and Java layer; increased the waiting socket queue size; allowed more ports to be used by programs; and, finally, looked into TCP congestion control algorithms. With these options on we ran the program with 20 threads, as the optimizations were made to improve load handling. Figure 21 shows the results, and details that the resultant median response time was 495 ms. This was a 46 ms, or an 8.5%, improvement in response time; not quite the magnitude we were looking for.

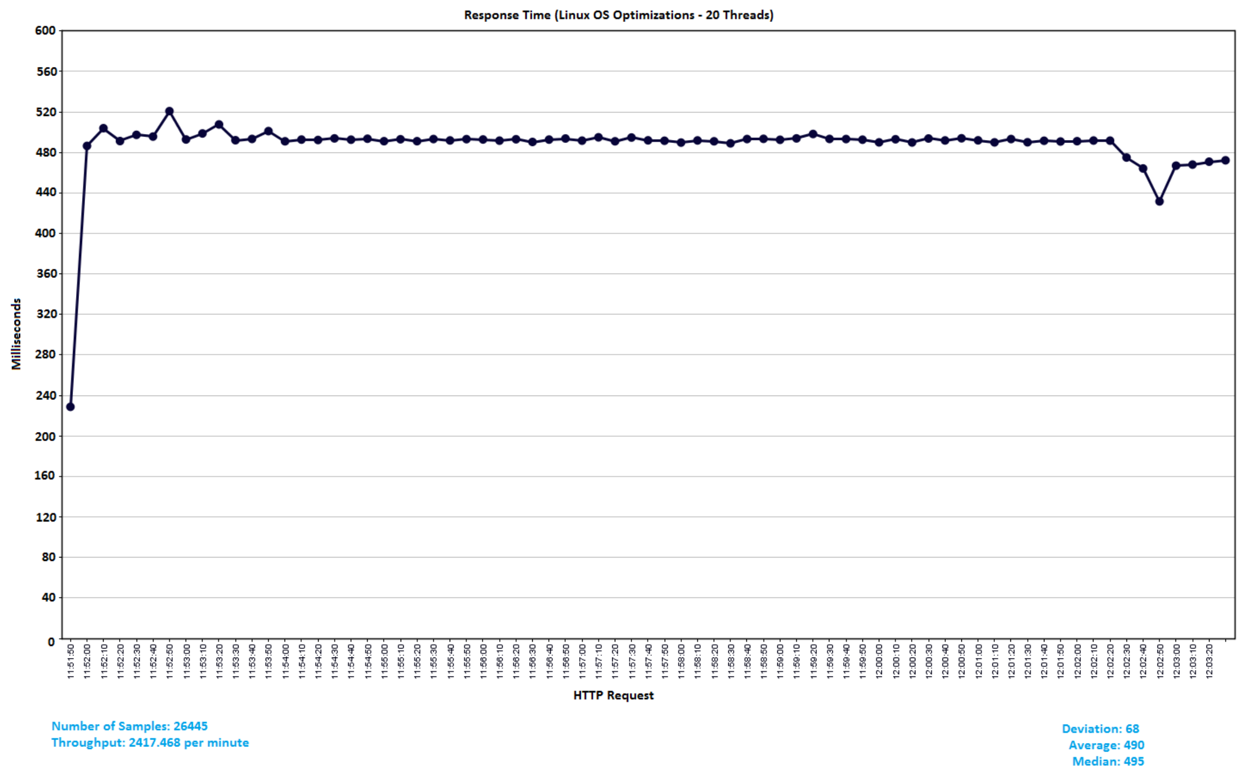


Figure 21: Linux OS Optimizations Response Times

Next we began optimizing the codebase and returned to 1 thread testing, as we had performed our load tests without much improvement. So far we had done JVM and Linux command line optimizations. All that remained was to optimize the code base. With the help of JProfiler, we identified the methods in the code that used the most CPU time. We discovered, to not much surprise, that iterating over the thousands of lines of files was taking the majority of CPU cycles. As such, we began moving from the data from List data structures to Hashmaps to reduce lookup time.

Lists and Hashmaps are fundamental data structures. A List is good for storing sequential data, and allows lookup time of O(n), where n is the number of elements. This means lists are good for smaller sets of data, or data with nothing to uniquely identify each piece. A Hashmap on the other hand is a data structure which can be conceptualized as a table. A unique key is used to lookup the values stored within the hashmap. A hashmap has lookup time of O(1), but requires more memory to store as each item in memory now has a corresponding entry in a table to allow for this speedy lookup. In our case, we did not care much about taking up space, but lookup time was essential. As such, we opted for using a Hashmap with its O(1) lookup time but increased memory usage over O(n), which would involve iterating over each row in a lengthy file to find the record we were looking for.

After about half the codebase had been optimized in this fashion, we produced the results in Figure 22. The results were extremely pronounced, with a median response time of 11 ms and a throughput of 4517 requests/min = 75 requests/sec. This was a 113 ms or 91.1% improvement in respose time! At this point, we had reached both our requirement of 10-50 ms/request and 50-200 requests/sec. This was still single threaded tests though, and we had not yet completed codebase optimization.

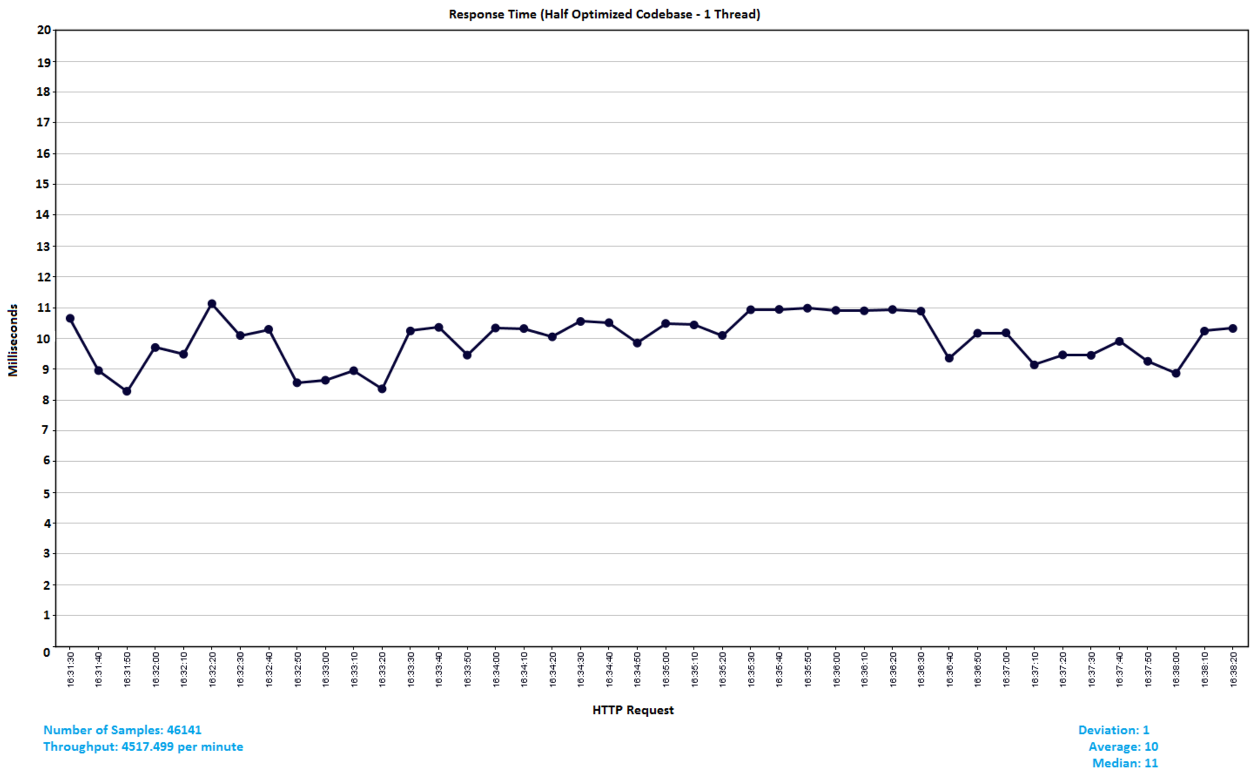


Figure 22: Half Optimized Codebase Response Times

Figure 23 details the results once the codebase was fully optimized. After the Figure 22, we had a few lists that had not yet been converted to Hashmaps. The median response time was now 4ms, a further 36% increase from the previous response time of 11ms! The throughput was now 9584 requests/min = 160 requests/sec. At this point we had surpassed our single-request lower bound requirement of 10-50 ms/requests. All that remained was to try it with multiple threads to see our new throughput with multiple users.

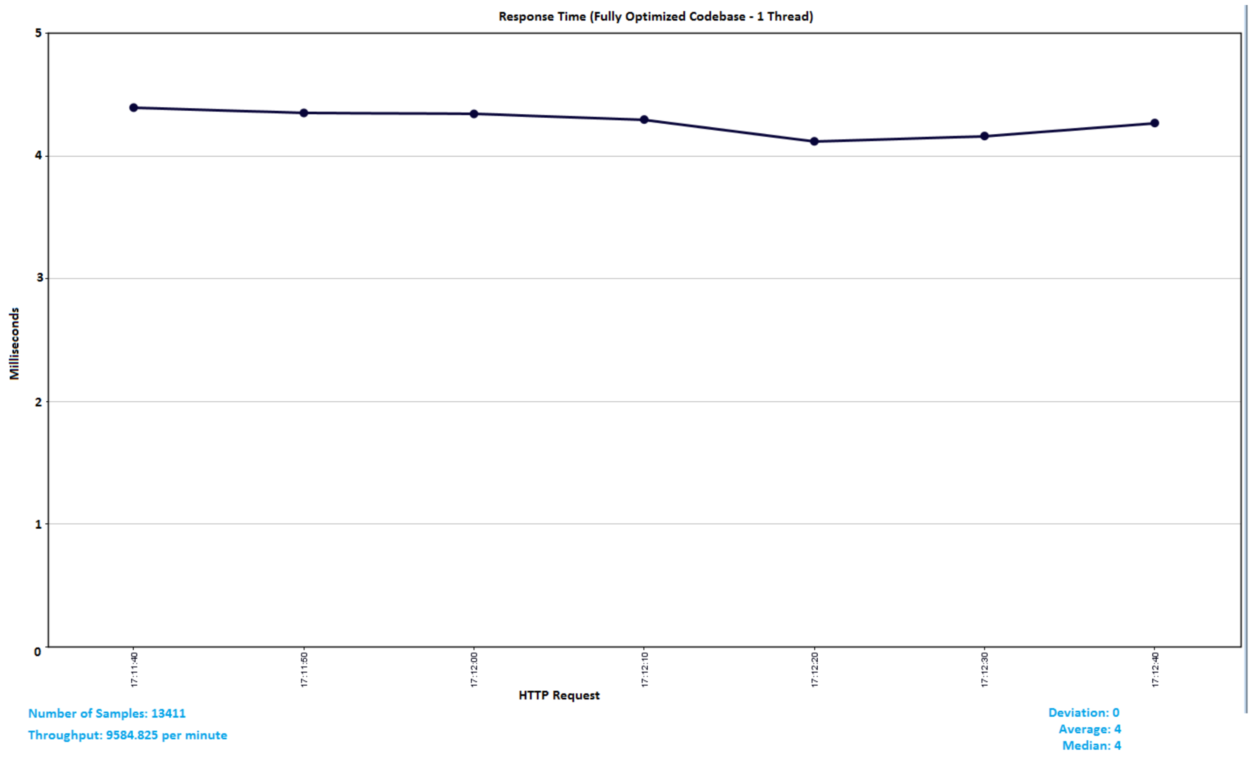


Figure 23: Fully Optimized Codebase Response Times

Figure 24 details the results of 75 threads (in other words, 75 simulated users) bombarding the test server with requests. We noticed the results were fairly erratic, but this noise is to be expected with 75 making requests in for loop forever. This bombardment resulted in a slight increase of response time, now 6 ms, which is a 50% increase from one thread to seventy five threads. However, it also resulted in a throughput of 25770 requests/min = 429 requests/sec. We had now far exceeded our requirement of 50-200 requests/sec. At this point, having exceeded both requirement of 10-50 ms/request and 50-200 requests/sec, we felt we had optimized the program enough. With 96,998 requests as our sample size, and a deviaition of 5, the mean holds fairly steady.

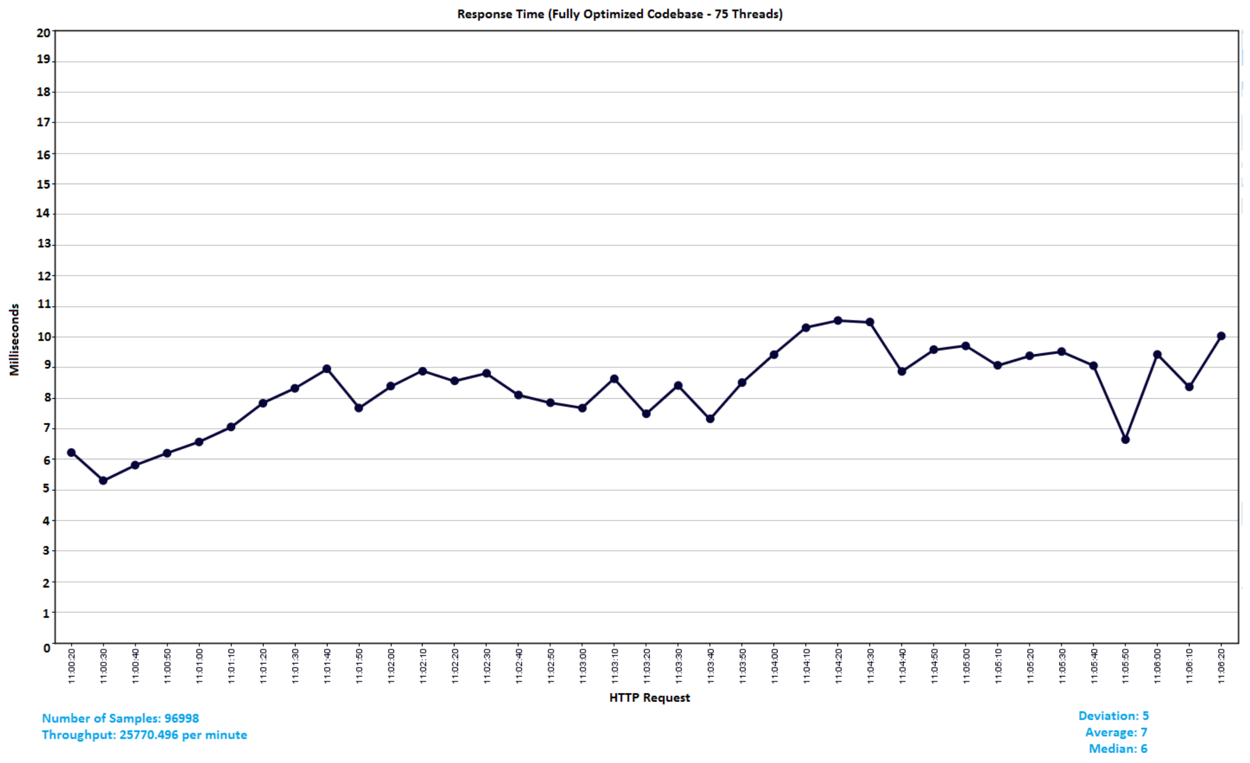


Figure 24: Fully Optimized Codebase Response Times using 75 Threads

Chapter 8 Conclusion

The results speak to the amount accomplished in this project. We started with over one hundred pages of USPS documentation and 12 data files, and were able to keep current with the ever-changing technical requirements while keeping in focus the agreed upon performance requirements. An example of this changing requirement was the inputs and outputs of our application as well as whether our application would be going into a production environment. As displayed in Chapter 7, we were able to handle over 50-200 requests per second with each request taking less than the specified goal of 10-50 milliseconds. We made a system capable of performing the equivalent service of the USPS Webtools API, while providing Endicia with a reliable and speedy service to query for various information about a package ready to be shipped.

To create this USPS Webtools equivalent system, we researched a variety of tools and created a full stack server-to-database application. The application utilizes an embedded Java server and a Web Service Framework to receive and parse through incoming requests. From here, the server uses various data files provided by USPS to execute our program’s algorithms which calculate various information about a package being shipped. Finally, the server again uses the Web Service Framework to return the response to the user in the requested response format, json or xml.

There were definitely difficulties along the way, and many things we needed to learn. The numerous tools detailed in Chapter 4 had to be researched and learned on site. Working with a government contracted company also brought an interesting dynamic, where the requirements are not always something we could ascertain from someone in the company, but members of the government. In the end, we were able to deliver a rich application API to provide what the company wants and also establish a framework for future work. As with most things in software, the project is far from achieving the elusive status of “done”. Much future work remains for deployment. We have left a maintainers’ document and a users’ document in the hands of Endicia to help them with this endeavor and to provide clear documentation for a project of this scope. Overall, it was a successful project.

Section 8.1 Future Works

While we met the performance goals outlined in Chapter 3, there is more to accomplish outside of the scope of this project. The primary future work is to put the server into a production environment. This includes allowing updated USPS data files to be downloaded and placed into the production environment as well as creating an environment in which to house the application.

While the infrastructure exists within the codebase to swap data files without any downtime, a system of downloading the updated files from the USPS database and placing them within the production environment needs to be constructed. This can be done by creating an application in any language to send GET requests to the proper USPS server to download the data files once a week. Once all of the files are downloaded, they can be sent to the Quality Assurance (QA) team to run the unit tests from our program against the new files. After this, a service needs to be created to download the new data files from an Endicia machine to the production server. Once this process is complete, a daemon can be created for our program to check if the filesystem has been modified, and if so swap the database to utilize the new data.

Creating a suitable production environment involves a deployment process through the Endicia deployment stack. This process begins with the QA team ensuring there are no bugs throughout the code. Next, a team of Endicia developers will create some Docker containers to host our application on a local server. After this, a load balancing mechanic, most likely using NGINX, will be put in place to dynamically load balance between Docker containers, where any container can be created or destroyed at any time. Finally, the Endicia Label Server will be modified to now send requests to the newly deployed server as opposed to the current USPS Webtools API.

References

[1] Apache. (n.d.). Maven – Maven Features. Retrieved February 29, 2016, from https://maven.apache.org/maven-features.html

[2] Chapter 11 Introduction to Web Services. The Java EE 6 Tutorial. Oracle, n.d. Web. 12 Dec. 2015. https://docs.oracle.com/cd/E19798-01/821-1841/gijti/index.html.

[3] Drools - Drools - Business Rules Management System (Java™, Open Source). (n.d.). Retrieved February 29, 2016, from http://www.drools.org/

[4] Eclipse. (n.d.). About the Jetty Project. Retrieved February 29, 2016, from http://www.eclipse.org/jetty/about.php

[5] Eclipse. (n.d.). Jetty. Retrieved February 29, 2016, from http://www.eclipse.org/jetty/

[6] Ej-technologies. (n.d.). Retrieved February 29, 2016, from https://www.ej-technologies.com/products/jprofiler/overview.html

[7] Endicia (n.d.). About DAZzle. Retrieved February 29, 2016, from https://www.endicia.com/EndiciaProfessionalHelp/Content/install_gs/download_install_dazzle.htm

[8] Endicia. (2015). API Strategy [PPT].

[9] Endicia. (n.d.). About Us. Retrieved February 29, 2016, from http://www.endicia.com/about-us

[10] Endicia. (n.d.). Endicia Company History. Retrieved February 29, 2016, from http://www.endicia.com/about-us/company-history

[11] Endicia. (2014). DaZzle Instructions for Quick Label Tab Tips: Test Before you Print - Online Shipping Blog | Endicia. Retrieved February 29, 2016, from http://online-shipping-blog.endicia.com/dazzle-test-print-tutorial/

[12] Endicia (n.d.). Endicia Label Server API. Retrieved February 29, 2016, from http://www.endicia.com/developer-resources/endicia-label-server-apis

[13] Internet Society, & Berners-Lee, et al. (2005, January). Uniform Resource Identifier (URI): Generic Syntax. Retrieved February 29, 2016, from https://www.ietf.org/rfc/rfc3986.txt

[14] Jetty - Servlet Engine and Http Server. Eclipse, n.d. Web. 12 Dec. 2015. http://www.eclipse.org/jetty/

[15] Kumar, A., & Apache. (n.d.). Apache CXF vs. Apache AXIS vs. Spring WS - DZone Integration. Retrieved February 29, 2016, from https://dzone.com/articles/apache-cxf-vs-apache-axis-vs

[16] NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. (n.d.). Retrieved February 29, 2016, from https://www.nginx.com/

[17] ObjectAid UML Explorer for Eclipse. (n.d.). Retrieved February 29, 2016, from http://www.objectaid.com/home

[18] Oracle and Sun MicroSystems. (n.d.). Java Archive (JAR) Files. Retrieved February 29, 2016, from http://docs.oracle.com/javase/7/docs/technotes/guides/jar/

[19] Oracle and Sun MicroSystems. (n.d.). Java Garbage Collection Basics. Retrieved February 29, 2016, from http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[20] Oracle and Sun MicroSystems. (n.d.). Using JConsole - Java SE Monitoring and Management Guide. Retrieved February 29, 2016, from http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html

[21] Stamps.com. (2015). Stamps.com Inc. - Stamps.com Announces the Completion of the Endicia Acquisition From Newell Rubbermaid. Retrieved February 29, 2016, from http://investor.stamps.com/releasedetail.cfm?ReleaseID=943463

[22] United States Postal Service. (2015). Automated Transit Files Implementation Guide [PDF].

[23] United States Postal Service. (n.d.). One Day in the Life of USPS... By the Numbers - Postal Facts. Retrieved February 29, 2016, from https://about.usps.com/who-we-are/postal-facts/one-day-by-the-numbers.htm

[24] United States Postal Service. (2015). Postal Facts 2015 [PDF]. Corporate Communications.

[25] United States Postal Service. (n.d.). Service Delivery Calculator Documentation. Retrieved February 29, 2016, from https://www.usps.com/business/web-tools-apis/service-delivery-calculator-api.htm

[26] What is REST (representational state transfer)? - Definition from WhatIs.com. (n.d.). Retrieved February 29, 2016, from http://searchsoa.techtarget.com/definition/REST

Appendix A (Users' Documentation)

Users' Documentation, Version 1

Inputs

The Webtools program offers two modes of input, listed below. Both forms of input require the same parameter names.

POST request with relevant parameter in an XML document

GET request with relevant parameters as query parameters in the URL.

The base url for requests is `http://[IP address]/webtools/v1/`

Parameters

Service: Webtools ELS Output

This service uses the path "mail" on top of the base URL, i.e. http://[IP address]/webtools/v1/mail

This is the base service, originally provided by the webtools application. Both POST and GET requests require the same parameters, so the following does not distinguish between the two. There will be an example of both at the end.

Parameter Name	**Required**	**Parameter Type**	**Restrictions**	**Description**
originzip	Yes	String	5 or 9 digits	ZIP code where the package originates
destzip	Yes	String	5 or 9 digits	ZIP code where the package is destined for
dropofftime	Yes	Integer	0000-2359	The time at which the package was dropped off

mailclass	Yes	String	Must be one of the following: PER, PME, FCM, STD. PKG, PRI.	The mail class with which the package is being shipped. PME and PRI both result in outputs different from the other classes.
desttype	Yes	Integer	1-3	This is the destination type. 1 corresponds to Street Address 2 corresponds to a PO Box 3 corresponds to Hold for Pick Up (HFPU). If 3 is specified, an HFPU location will also be returned.
shipdate	No	Date	Must be in format dd-MMM-yyyy, ex 01-Jan-2016	The date which the package is dropped off. This defaults to the current date if not specified.
deliveryoption	No	Integer	0-7	This specifies if the package should not be delivered on certain days. The default value is 0 if none is specified. 0: Omit no days 1: No Saturday delivery 2: No Sunday delivery 3: No weekend delivery 4: No holiday delivery 5: No Saturday or holiday delivery

				6: No Sunday or holiday delivery 7: No delivery on Saturday, Sunday, or Holidays.

GET Request example

An example URL with a GET request. In this, the shipdate is specified but the delivery option is omitted. Take note of the following:

The parameters are case sensitive.

You may use quotes to enclose the parameter values or omit them as needed; the program will strip any whitespace and quotes that it can.

The order of parameters specified does not matter.

```
http://{IPAddress}/webtools/v1/mail/?originzip=32669&destzip=81654&dro
pofftime=1000&mailclass=”PME”&desttype=3&shipdate=01-Jan-2016
```

POST Request example

An example POST request body and the corresponding URL used to perform this request can be found below. Take note of the following:

- The HTTP “Content-type” header must be set to “application/xml”
- The parameters are case sensitive
- There must not be whitespace inside the XML value between XML tags
- The order of parameters does not matter
- The surrounding “XMLInput” tags are required

```
http://{IPAddress}/webtools/v1/mail/
```

```
– <XMLInput>
     <originZip>12345</originZip>
     <destZip>12345</destZip>
     <shipDate>22-Feb-2016</shipDate>
     <shipTime>1000</shipTime>
     <mailClass>PME</mailClass>
     <destType>2</destType>
  </XMLInput>
```

Outputs

The program can output multiple types of output. The user can specify either .json or .xml to explicitly retrieve the type they are looking for. The current default if none is specified is XML. The user specifies which return type is desired in the following fashion:

URL	Return Type	Notes
http://{IPAddress}/webtools/v1/mail/	XML	Default
http://{IPAddress}/webtools/v1/mail.xml/	XML	
http://{IPAddress}/webtools/v1/mail.json/	JSON	

The output for the ELS service contains the following fields, in the following order inside the body of an XML tree with the root element named MailCommitments or a JSON object.

Note that some fields are specific to what was requested at input time. The conditional inclusion field is No if the value is always present in the output, otherwise an input condition will be specified

a “Date” type implies the format dd-MMM-yyyy, ex 01-Jan-2016\

The commitment and location types are specified below this chart

Element Name	Element Type	Description	Conditional Inclusion

cutOffTime	Integer	The cut off time at which facility closes	No
deliveryDate	Date	The date the package will arrive at the destination.	No
destCity	String	The city the package is being sent to based on the destZip	No
destState	String	The state the package is being sent to based on the destZip	No
destType	Integer	The destination type originally input by the user	No
destZip	String	The destination ZIP code originally input by the user	No
EAD	Date	Effective arrival date	No
guarantee	Boolean	Whether a guarantee is present	No
mailClass	String	The mail class originally input by the user	No
originCity	String	The city the package is sent from based on the origin ZIP	No
originState	String	The state the package is sent from based on the origin ZIP	No
originZip	String	The origin ZIP code originally input by the user	No
shipDate	Date	The date the package was shipped on, originally input by the user	No

shipTime	Integer	The time the user reported dropping the package off at, originally input as dropofftime	No
svcStdMsg	String	The service standard message generated from USPS data based on the input mail class	No
commitment	Commitment	A commitment containing info on the Priority Mail Express (PME) shipping details	If PME was the input mailclass
PRILocation	List of Location	There can be any number of these nodes, and they each contain a Location where drop off is possible	If PRI was the input mailclass
HFPULoc	Location	The location where the package will be held for pick up	Present if the input desttype is 3, aka HFPU

Commitment

The structure of a Commitment is as follows:

Element Name	Element Type	Description
commitmentDate	Date	The date the package will be delivered
commitmentRank	Integer	The rank the commitment has
cutoffTime	Integer	The time at which the post office closes
deliveryTime	Integer	The time at which the package is delivered by

preferredIndicator	Boolean	1 if the package has prefered status, 0 otherwise
serviceStd	Integer	The service standard type

Location

The structure of a Location is as follows:

Element Name	Element Type	Description
cutoffTime	String	Only output for HFPU requests. A package can be dropped off at this location no later than this time
facAddress	String	Street address of the facility
facCity	String	City where the facility is located
facState	String	State where the facility is located
zipCode	String	Zip Code of the current facility

Appendix B (Maintainers' Documentation)

Maintainers' Document

Overview of package structure

This section provides a brief overview of the factors behind the separation of packages. All packages are located within src/main/java, which is standard convention for Maven projects. Within this folder, the packages are as follows:

- **ApacheMain** contains the program stack down to where the parsed request is passed off to the algorithm which processes it. Classes needed to run Apache CXF and handle basic server functionality are found here. This package also contains the Main program, which begins a Jetty server with a CXF servlet.
    - **ApacheMain.outputwrappers** contains the classes necessary to customize the format of an output. It essentially wraps the output of a processing algorithm in the format Apache CXF requires to serialize to json and xml
- **atfImplementation** contains the algorithms required for calculating information based on ATF files. This contains classes common to both specific ATF implementations
    - **atfimplementation.PMEcommitment** contains the algorithms necessary for processing a request when the mail class is PME
    - **atfimplementation.nonPMEcommitment** contains the algorithms necessary for processing a request when the mail class is not PME
- **dataHandler** contains the infrastructure that maintains the database of ATF files used throughout the program. The functionality to update the database is also located here.
    - **dataHandler.dataFiles** contains the datafile classes, each of which is an object representation of an ATF file
- **droolsRules** is where the classes for interaction with the drools engine is located. Here is where the USPS .drl files are currently located.
- **utility** is the package where various utility classes are located.

Stack Explanation

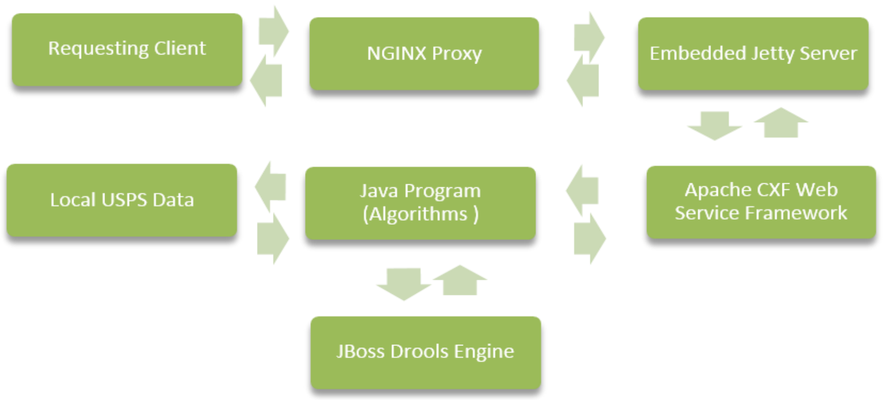


Figure 1

Figure 1 displays the stack of the program. A requesting client makes a request via a client socket, machine, or browser. If NGINX is active it will intercept requests at port 80 and forward them to Jetty which is running on port 4651. Once Jetty has the request, it passed it along to Apache CXF. Apache CXF does any validation on the parameters via a user-implemented function in the QueryParser class. Apache CXF then passes the parameters along to the underlying Java Program. The Java program communicates with the JBoss Drools engine in conjunction with the USPS data that it has loaded into memory, and then returns a response up the stack, which is passed to Apache CXF, and then finally Jetty which sends the request back over the wire. If NGINX received the initial request, it will intercept Jetty's response, do any logging or processing specified in the config file, and then pass the request back to the requesting client. If Jetty was accessed directly by specifying port 4651, NGINX will not intercept the response and it will be sent back to the client directly.

Adding New Input/Output

As mentioned previously, the `ApacheMain.outputwrappers` is where the various output classes are located. To construct a different kind of output, one must specify the path which will be used to access it and the parameters used as input in `ApacheMain.MainRestService`, and then specify the class with appropriate class variables along with getters and setters in `ApacheMain.outputwrappers.`

Figure 1 is an example of a new path and output type being created as a new method in `ApacheMain.MainRestService`:

```
@Path("/mail.example")
@Produces( {"application/xml"} )
@GET
public Collection<StringOutputWrapper> getBaselineRequest() {
    Collection<StringOutputWrapper> output = new ArrayList<StringOutputWrapper>();
    output.add(new StringOutputWrapper("This outputs a string to the browser"));
        return output;
}
```

Figure 2: Example of a new path and output

The `@Path` specifies which path off of `/webtools/v1/` the new path will be located. The `@Produces` specifies which content type will be produced, in this case XML. The `@GET` specifies the HTTP method of input, in this case GET. The return type must be a collection. In this case, the output wrapper being used is `StringOutputWrapper`. This class is simply has one variable, a String, along with getters and setters. The output is simple in this case, but many complex structures can be created and returned from different paths using this infrastructure.

Changing where the Files are located

The class, `FilenameConstants`, that contains the paths to the ATF data files is located in the directory `src/main/java/com/endicia/localServer/dataHandler/dataFiles.` Currently, the class `FilenameConstants` points to the root directory, one level above `src/`, for the text files. This allows the files to easily be changed and the database updated without restarting the JAR. How this is done is detailed in the next section.

Updating Data Without Downtime

The following section refers to classes within the dataHandler package.

The Database is currently accessed throughout the code using a singleton with the following static invocation: `DataMaster.getInstance()`. This database is initialized at the start of the program, preventing file I/O during requests.

To update the database, there is a `swapdata` method in the `DataMaster` class. One creates a `new DataUpdater()`, and calls the `swapdata` method with the `DataUpdater` object as the argument. The following code snippet shows how one might do this.

```
//example usage of the database
DataMaster.getInstance().getRefValue().isUSPSHoliday(dateString)
//Updating of the database
DataMaster.swapData(new DataUpdater());
```

How the data update is triggered is up for debate. Currently, one can look inside `ApacheMain.MainRestService` and find a GET path that allows one to trigger a database update by accessing a specific URL. Security features are necessary for this.

To update the files that the database currently uses, one can replace all of the files located one directory above `src` with an updated version of these files and call the specific path mentioned above. The swap method looks for the required files in the aforementioned directory and automatically updated all of the files. It is important to note that the USPS .drl files are currently packaged within the jar and are not located in this folder. This means that to update the .drl files, one must recompile and redeploy the jar.

Updating Tests

The unit tests are located in `src/test/java/com/endicia/localServer/unittest` directory. Building the project with maven in the directory above src will execute any tests in the .../`localserver` package and any subpackages.

JVM and Linux command line Optimizations

Below one can see the command line arguments used when running the program in the JVM, with notes as to the purpose.

JVM Options

These JVM options are essential for the program to initialize and run smoothly without periodic performance spikes. These options are unlikely to change based on the deployment environment.

JVM Argument	**Description**
`-server`	Runs the JVM in server mode. Usually a default option for the JVM.
`-Xms5g`	This is the starting memory the JVM is allocated. In this case, it is set to 5GB. The program takes around 4GB in its resting state, with a bit more during high load times.
`-Xmx10g`	This is the memory cap the JVM is given. This is necessary if the data updater is used, as the program creates another database, swaps the pointers, and then garbage collects the old database.
`-XX:+UseConcMarkSweepGC`	This selects the Concurrent Mark Sweep garbage collector for use by the JVM. This option is necessary to prevent large "stop the world" performance degradation.

Linux command line options

These are the options used by the testing environment to handle a decent volume of requests, with a noticeable performance increase. Most of these parameters are tunable based on the production environment and usage statistics.

Linux Command Line Argument	**Description**

`sysctl -w net.core.rmem_max=16777216`
`sysctl -w net.core.wmem_max=16777216`
	These commands change the network read (rmem) and write (wmem) buffers capacity.

`sysctl -w net.ipv4.tcp_rmem="4096 87380 16777216"`
`sysctl -w net.ipv4.tcp_wmem="4096 16384 16777216"`
	These commands change the tcp layer's read and write buffer size. Look to further tuning of these parameters and the ones above if buffer bloat becomes a problem.

`sysctl -w net.core.somaxconn=4096`	This changes the size of the connection listening queue at the TCP level. This is a very tunable option.

`sysctl -w net.core.netdev_max_backlog=16384`	This is the size of the incoming packet queue for java-layer processing.

`sysctl -w net.ipv4.tcp_max_syn_backlog=8192`
`sysctl -w net.ipv4.tcp_syncookies=1`
	The first command raises the queue size allowed for processing syn packets, otherwise known as the TCP message to open a connection. The second option protects against a DDoS attack known as syn flooding.

`sysctl -w net.ipv4.ip_local_port_range="1024 65535"`
`sysctl -w net.ipv4.tcp_tw_recycle=1`
	The first command changes the range of usable ports to almost the entirety of the available port range for typical hardware. The second command allows the kernel to reuse sockets more efficiently.

`sysctl -w net.ipv4.tcp_congestion_control=cubic`	This command changes the network congestion algorithm the kernel uses. In the case of this application, cubic is selected which is an algorithm designed for high throughput networks. This parameter has a relatively high effect on performance.

Logging

Logs are saved in the directory where the program is being run from. A file is created when the server starts, with the naming convention `[CurrentDate]log.txt`. There are two levels of logging used throughout the program, INFO and SEVERE. The INFO level is used whenever the user enters information that does not pass the input validation stage, or results in a calculation not possible exception; in other words, INFO is used for errors in user input. This can be used for future data mining on common user errors to perhaps make a more friendly interface.

The INFO level is differentiated from the SEVERE level as the SEVERE level indicates a fault in internal logic. If something is logged as SEVERE, an internal server error has been returned to the user. An error logged as severe requires immediate attention.

It is possible for a user to attack the server by spamming requests and filling up the hard drive with error logs. As such, regular monitoring of the logs, for server uptime and bugs alike, is necessary. One can easily disable the logging for errors caused by user input if this data is not wanted by removing and INFO level logs.

How to Deploy and Use with Current setup

Compiling the source code to a single jar requires only the Apache Maven dependency. Once this is installed one can run “mvn clean package” inside the directory where the “pom.xml” file is located. Maven will run all tests to ensure the current code passes all regression testing. After all of the JUnit tests have run and passed, a deployable jar named “USPSDataMigration-1.0-SNAPSHOT-jar-with-dependencies” will be output in the “target” subdirectory.
This jar, along with the USPS data files can then be placed in a server environment inside of a folder named “USPSDataMigration”. The only dependency required for the deployment server is a JRE version 1.8 or higher. Once inside this folder, one can run the
“java -server -Xms5g -Xmx10g USPSDataMigration-1.0-SNAPSHOT-jar-with-dependencies.jar” command to start the server. The server currently runs by default at the path `/webtools/v1/mail` and can be accessed by connecting to `port 4651`.

Currently, one can update the in memory database by accessing a specific URL path. This path includes a hardcoded 128 bit sequence for an example of a to-be-implemented security method. This path is:

```
/webtools/v1/mail.updater4d2d2808b23b618dd944e71054a44022c145a5ba7a5b67c6adaf
4544970b90b2ad87b65d0ccc5cd55745a277e3a5d76d3ee09d04deaf648bfb410dc3e7af7f23.
empty
```

Best Commitment Selection

The current webtools API simply returns all commitments found for each request. USPS provided us with an algorithm to select the best commitment from a list of commitments. As such, only the best commitment is returned in the output. One can see the USPS ATF Implementation Guide for further details, in Appendix B. Another option is to examine the `BestPMECommitment` subroutine located in `atfimplementation.PMECommitment`, which contains identical logic to the flowchart in the ATF implementation guide, just in Java code form.

Open Issues, Bugs, and Debugging

See Appendix A for issues that can be solved via USPS.

In addition to the issues that require assistance from USPS, we found an error with calculating the commitment date. It seems that the commitment date is incorrect at some point. It is highly advised that the issues with USPS documentation be resolved before looking into this incorrect commitment date issue, as Interfacility will be changed and therefore this issue may be fixed as a result.

To debug the code, the Eclipse IDE along with the Maven to Eclipse plugin (m2e) is recommended. Simply import a project as a maven project, and run the main located in `ApacheMain` in debug mode. This will allow the user to select breakpoints, and one can simply make requests via any browser at the aforementioned path and port number to trigger the program to run through these breakpoints.