Software Testing Plan

04/08/2018

Version 1.0

Team: Nimbus Technology

Sponsor: IBM

Faculty Mentor: Austin Sanders

Team Members:

Itreau Bigsby

Matthew Cocchi

Richard “Riley” Deen

Benjamin George

Table of Contents

1) Introduction 2

2) Unit Testing 3

3) Integration Testing 19

4) Usability Testing 22

5) Conclusion 24

1 Introduction

In the industry of cloud services, one of those most commonly used services is data storage, provided by vendors such as Amazon Web Services (AWS) or Microsoft. This cloud storage tends to come with little in the way of storage *management*, which is instead offered by other vendors like IBM, whose Spectrum Protect Server Development department has contracted this project with us.

One of the key services that Spectrum Protect offers is the ability to reduce a client's data storage consumption without losing any actual data, making these services quite lucrative for clients. Spectrum Protect have recently come upon an idea for a new way to further reduce a client's cloud storage: mark data as "expired" based on whether or not any other data in the cloud storage is reliant on it, and then remove all expired data.

This process is referred to as reclamation, and we are building a product which will automate the process of reclamation for Spectrum Protect. This will include a backend service that handles the actual process of reclamation, a frontend UI to display reclamation statistics (such as storage and monetary savings), a database to store metadata files about the reclaimed files, and the ability to communicate with Amazon Web Services' S3 cloud data storage service.

Having finished the implementation of this product, we are moving into testing it to ensure that it will satisfy Spectrum Protect's goals and properly handle any input with no error. This software testing will occur in three steps:

- Unit Testing: Ensuring that each "unit" of code behaves correctly with any given input. The scope of a "unit" of code can vary from project to project, but in our case a unit will refer to a single function. All of our functions will be tested to verify that they return correct values for valid inputs, and do not fail or cause the program to crash when given invalid inputs.

- Integration Testing: Ensuring that the different modules and components of the project are communicating with each other correctly. Our product is composed of a series of components and modules which provide the resources and services requested by end-users. All of these components and modules will need to be tested to verify that they pass data between themselves correctly, and verify that the entire system as a whole is working correctly.

- Usability Testing: Testing our frontend UI with an audience to ensure that the design of our frontend is easy to navigate for end-users within our intended target audience. This will involve testing the arrangement and design of frontend display elements, as well as testing sample users' ability to find specific resources that are not immediately available in the display.

This project is mostly focused on the backend computation processes, and less focused on the frontend. To add to this, the frontend is intended for an audience for IBM administrative employees who will have a fair level of technical experience, meaning our frontend design does not have to be particularly intuitive or non-technical. For these reasons our testing is primarily focused on unit and integration testing.

First we must make sure that all of our code functions correctly and reliably, especially seeing as, if our product should do anything wrong and incorrectly erase IBM clients' data, IBM will be held liable and their reputation will be damaged. Second, we will ensure that all of the components that compose our product can communicate correctly, since our software must be able to interact with S3 cloud storage as well as a database. Finally, we want to do some minimal usability testing to at least ensure that our product will be usable by IBM employees with little to no trouble. With this testing regimen laid out, we now cover our plan for unit testing.

2 Unit Testing

Having outlined the strategy and rationale behind our testing, we now explore in more depth the question of unit testing our software. As with any unit testing, our tests will be based on:

- Equivalence partitions: Broad categories of input for a tested unit, for which the output produced in response to inputs from the same partition is also the same.
- Boundary values: "Extreme" inputs which stress the ability of our software to handle any input--properly formed or not.
- Example input: The input given to a unit that is being tested.
- Expected output: The output that should be produced by a tested unit in response to a given example input.

In considering unit testing, we decided to focus exclusively on the backend of our software, since that is where all of the operations and inputs will actually occur. The frontend takes no inputs beyond what is fed to it by the backend (as a response to requests for data pertaining to containers and reclamation), and consists of small Javascript components that perform no logic or computation other than "display this data here." As such, our unit testing is focused on the backend, where the real logic of our application occurs, and where variable inputs could cause issues if not handled properly.

With this in mind, we break the backend down into the following groups of modules:

- Analysis: Modules responsible for analyzing and tracking statistics on containers for which requests to reclaim have been received.
- Loggers: Modules which log activities pertaining to reclamation requests.
- Talkers: Modules which handle connecting the backend to the other components of our system (AWS, MongoDB, and the frontend). These modules are covered more in integration testing.

- Handler: The most front-facing module of the backend, through which requests to reclaim a container or for frontend data are received.
- Reclamation: Modules which handle the actual process of reclaiming a container.
- Core: The module that is responsible for communicating with the other backend modules; acts as a driver for all backend functionality.

Some of these modules, particularly those that communicate with AWS or MongoDB, consist primarily of functions that are simply thin wrappers for functions provided by outside libraries (e.g. Amazon's AWS Go SDK). In such cases we can only control whether or not we are supplying those functions with correct data; whether they return correct data is out of the scope of our project, and thus we won't be concerned with such issues.

Fortunately, Go provides a testing package (aptly named "testing") that simplifies the creating and running of unit tests. However the designers of Go have intentionally left a common unit testing construct out of the testing package: Assert statements, which allow for quick checking of conditions. This has a silver lining effect of forcing us to create our tests in more detail, such that we explicitly check the passing and failing conditions ourselves. All of our unit tests will be conducted in this manner, with a separate test file for each module. Additionally, it is standard practice when writing code in Go to have all functions return, in addition to their intended returns, an error value indicating whether or not anything went wrong during the course of the function. Since these error returns can ripple up a chain of function calls, this ensures that an error that occurs a low level of execution can be caught there and handled at a higher level, allowing us to handle all errors gracefully.

For each category of module, we consider each module in that category and put all of the testing criteria for that module into a table that shows what inputs we expect the module to take and corresponding outputs we expect it to produce. These tables are each followed by a section explaining the rationale of our testing criteria, such as equivalence partitions and boundary values.

Analysis

This is a pair of modules, cbAnalyze and cbStats, which perform the task of analyzing a container's metadata to decide whether that container meets the threshold of expired data and is worth reclaiming. In addition, these modules gather useful statistics which are displayed in the frontend such as fragmentation percentage (different from typical hard drive fragmentation; this simply refers to the percentage of a container's contents which is expired) and total data reclaimed.

The functions that compose these modules will be tested as shown in the tables that follow:

cbAnalyze:

Function Name	Equivalence Partitions	Boundary Values	Example Input	Expected Output
setMinimumBytes	< 0, ≥ 0 and ≤ 2^30, > 2^30	-1, 1000, (2^30) + 1	-1	Exception
			1000	N/A
			(2^30) + 1	Exception
getMinimumBytes	N/A	N/A	N/A	Int, the minimum number of bytes to trigger reclamation
performAnalysis	Container with at least minimumBytes expired bytes, Container with less than minimumBytes expired bytes, Malformed container	Container with exactly minimumBytes expired bytes, Container with 0 expired bytes, Malformed container	Container with exactly minimumBytes expired bytes	True, indicating that the container meets the threshold and should be reclaimed
			Container with 0 expired bytes	False, indicating that the container does not meet the threshold and should not be reclaimed
			Malformed container	Exception

The rationale for these testing criteria is as follows:

- setMinimumBytes: The minimum bytes in question here is the minimum number of bytes in a container that must be expired to trigger reclamation of the container. Since a container can't have a negative file size, values below zero are invalid. Likewise, the maximum file size of a container is one gigabyte, or 2^{30} bytes, thus any value greater than 2^{30} is invalid. Any value greater than or equal to zero and less than or equal to 2^{30} is acceptable.
- getMinimumBytes: This function takes no input, and simply returns the minimum bytes value discussed above.
- performAnalysis: This function takes as input a struct which holds metadata on a single container. This metadata describes, among other things, the amount of data in the container that is expired. Either that amount is less than the minimum bytes required to trigger reclamation, in which case the function returns false, or that amount is greater than or equal to the minimum bytes required to trigger reclamation, in which case the function returns true. However if the container struct is malformed, perhaps missing certain attributes, then analyzing the container for potential reclamation is pointless, in which case the function throws an exception.

cbStats:

Function Name	Equivalence Partitions	Boundary Values	Example Input	Expected Output
setTotalBytes	< 0, ≥ 0	-1, 0	-1	Exception
			0	N/A
getTotalBytes	N/A	N/A	N/A	Int, the number of bytes of data thus far analyzed
setReclaimedBytes	< 0, ≥ 0	-1, 0	-1	Exception
			0	N/A
getReclaimedBytes	N/A	N/A	N/A	Int, the number of bytes of data thus far reclaimed
setTotalChunks	< 0, ≥ 0	-1, 0	-1	Exception
			0	N/A
getTotalChunks	N/A	N/A	N/A	Int, the number of chunks thus far analyzed
getFragPercent	N/A	N/A	N/A	Float32, the proportion of a container's contents which are expired
getBytesInContainer	N/A	N/A	N/A	Int, the number of bytes currently in a container
getAmountSaved	N/A	N/A	N/A	Float32, amount of money saved through reclamation, in USD

The rationale for these testing criteria is as follows:

- All of these functions are responsible for tracking statistics on containers analyzed and reclaimed. All of the "set" functions update a certain statistic, while the corresponding "get" functions return the values of those statistics.
- "set" functions: It is not possible to have tracked a negative amount of bytes, so values below zero are invalid for all of these functions. Any value greater than or equal to zero is valid.
- "get" functions: None of these functions take any input, but instead fetch stored statistics or calculate a new statistic and return the requested statistic.

Loggers

This set of modules handles the creation of activities, which are individual reclamation events, for the purpose of organizing data to display in the frontend.

logger:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
Logger	Malformed strings, Valid strings	“” - empty string, “HandleContainer”	“” - empty string	Exception
			“HandleContainer”	http.Handler, a new handler which has logging capabilities

The Logger function takes as input a string signifying a request for a resource or process that our system offers. The expected string for each resource or process is hardcoded, and any string that does not exactly match one of those options is considered malformed. In the case of a valid string, the function should return a new HTTP request handler than has logging capabilities. Otherwise the function will throw an exception.

activityLog:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
createActivity	Malformed containers, Valid containers, Times before handler.Start, Times after handler.Start	Container with any unset fields, Container with all fields set, Times before handler.Start, Times after handler.Start	Container with an unset field	Exception
			Time before handler.Start	Exception
			Container with all fields set + time after handler.Start	ActivityLog, a struct containing information about the reclamation activity

This function takes two inputs: a container that has been reclaimed, and a time at which that reclamation occurred. If the container is malformed (that is, has any unset attributes) or if the given time is before the HTTP request handler was created, the input is deemed invalid and an exception is thrown. Otherwise an activity is created and stored, which represents the reclamation event and will be used for data display purposes in the frontend.

Talkers

This set of modules handles communicating with components of our system outside of the backend: the frontend, AWS, and MongoDB. These modules mostly rely on and build around libraries, and as a result involve little of our own code.

awsModule:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
downloadObject	Names of S3 buckets and objects that do not exist, Names of S3 buckets and objects that do exist	Name of S3 bucket or object that does not exist, Names of S3 bucket and object that do exist	Name of S3 bucket or object that does not exist	Exception
			Names of S3 bucket and object that do exist	S3 object contents downloaded to file
uploadUsed	Files that do not exist on the host machine, Files that do exist on the host machine, Names of S3 buckets that do not exist, Names of S3 buckets that do exist	File that does not exist on host machine or S3 bucket that does not exist, File that does exist on host machine and S3 bucket that does exist	File that does not exist on host machine or S3 bucket that does not exist	Exception
			File that does exist on host machine and S3 bucket that does exist	N/A
connectToAWS	Strings of regions that do not exist, Strings of regions that do exist	String of region that does not exist, String of region that does exist	String of region that does not exist	Exception
			String of region that does exist	An AWS connection session

The rationale for these testing criteria is as follows:

- downloadObject: This function takes as input an S3 storage bucket name and the name of an object file to retrieve from that bucket. If either does not exist (or cannot be connected to due to invalid credentials or permissions) then the function will throw an exception. Otherwise the contents of the S3 object will be downloaded into a file on the host machine.

- uploadUsed: This function takes as input the name of a file on the host machine to upload to S3 and the name of an S3 storage bucket to upload the file into. If either does not exist (or the bucket cannot be connected to) then this function throws an exception. Otherwise the upload is successful and no output is produced.
- connectToAWS: The only input for this function is a string representing the region of AWS servers to connect to (e.g. “us-east-1”). If the string is invalid, the connection will fail and the function will throw an exception. Otherwise the connection is successful and the connection session is returned.

databaseModule:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
mongoSession	Invalid URL strings for MongoDB connection, Valid URL strings for MongoDB connection	Invalid URL string for MongoDB connection, Valid URL string for MongoDB connection	Invalid URL string for MongoDB connection	Exception
			Valid URL string for MongoDB connection	MongoDB connection session
storeActivity	Malformed activities, Properly formed activities, Invalid pointers to a MongoDB database, Valid pointers to a MongoDB database, String names of collections that do not exist in the database, String names of collections that do exist in the database	Malformed activity, Properly formed activity, Invalid pointer to a MongoDB database, Valid pointer to a MongoDB database, String name of collection that does not exist in the database, String name of collection that does exist in the database	Malformed activity, invalid pointer to a MongoDB database, or string name of collection that does not exist in the database	Exception
			Properly formed activity, valid pointer to a MongoDB database, and string name of collection that does exist in the database	N/A
getActivities	Invalid pointers to a MongoDB database, Valid pointers to a MongoDB database, String names of collections that do not exist in the database, String names of	Invalid pointer to a MongoDB database, Valid pointer to a MongoDB database, String name of collection that does not exist in the database,	Invalid pointer to a MongoDB database or string name of collection that does not exist in the database	Exception

	collections that do exist in the database	String name of collection that does exist in the database		
			Valid pointer to a MongoDB database and string name of collection that does exist in the database	List of all reclamation activities in the specified collection in the given database
getByDate	Invalid pointers to a MongoDB database, Valid pointers to a MongoDB database, String names of collections that do not exist in the database, String names of collections that do exist in the database, Pairs of datetimes in which the “from” occurs after the “to”, Pairs of datetimes in which the “from” occurs before the “to”	Invalid pointer to a MongoDB database, Valid pointer to a MongoDB database, String name of collection that does not exist in the database, String name of collection that does exist in the database, Pair of datetimes in which the “from” occurs after the “to”, Pair of datetimes in which the “from” occurs before the “to”	Invalid pointer to a MongoDB database, string name of collection that does not exist in the database, or pair of datetimes in which the “from” occurs after the “to”	Exception
			Valid pointer to a MongoDB database, string name of collection that does exist in the database, and pair of datetimes in which the “from” occurs before the “to”	List of all reclamation activities that occurred between the “from” and “to” datetimes found in the specified collection in the given database
removeAll	Invalid pointers to a MongoDB database, Valid pointers to a MongoDB database, String names of collections that do not exist in the database, String names of collections that do exist in the database	Invalid pointer to a MongoDB database, Valid pointer to a MongoDB database, String name of collection that does not exist in the database, String name of collection that does exist in the database	Invalid pointer to a MongoDB database or string name of collection that does not exist in the database	Exception
			Valid pointer to a MongoDB database and string name of collection that does exist in the database	N/A

getTotals	See above	See above	See above	Aggregate log, containing statistics on all containers reclaimed up to the time of the request
getUsage	See above	See above	See above	Unsigned 32 bit integer, representing the total amount of bytes of storage in use by reclaimed containers up to the time of the request
getReclamation	See above	See above	See above	Unsigned 32 bit integer, representing the total number of bytes of storage reclaimed up to the time of the request
getContainers	See above	See above	See above	Integer, representing the total number of containers reclaimed up to the time of the request
getFragRate	See above	See above	See above	64 bit float, representing the average fragmentation (i.e. expiration) percentage of all containers reclaimed up to the time of the request
getSavings	See above	See above	See above	64 bit float, representing the monetary savings (in USD) from all reclaimed containers up to the time of the request
getCosts	See above	See above	See above	64 bit float, representing the monetary cost (in USD) of storage

				still in use by all containers reclaimed up to the time of the request

The rationale for these testing criteria is as follows:

- All functions other than mongoSession take essentially the same parameters (a database to pull from, and a collection in that database with objects to pull) and respond to malformed input in the same manner, so for the sake of brevity several rows in the above table were omitted.
- mongoSession: This function takes as input a string, the URL of a MongoDB database to connect to. If the string is not properly formed or does not represent a database that exists, the connection will fail and the function will throw an exception. Otherwise the connection will succeed and the connection session is returned.
- storeActivity: This function takes a pointer to a database and a string name of a collection in that database, as well as a reclamation activity struct. If either is malformed or invalid, the function throws an exception. Otherwise the struct is converted to JSON and stored in the specified collection in the given database.
- getActivities: As with storeActivity, this function takes a pointer to a database and a string name of a collection in that database. If either is malformed or invalid, the function throws an exception. Otherwise all reclamation activities in the collection are returned.
- getByDate: This works exactly the same as getActivities, with the added parameters of a start date and end date between which to retrieve all activities. Seeing as the end date must occur after the start date, a pair of such inputs with an end date occurring before the start date will also cause an exception. Otherwise all activities between those dates are returned.
- removeAll: This function takes the same database and collection identifiers and, provided both are correct, deletes everything from the collection and returns no output.
- getTotals: This function calls all of the following functions using the same database and collection identifiers as the above functions. If either identifier is invalid, an exception is thrown. Otherwise the function returns the aggregate of the returned values from the following functions:
    - getUsage
    - getReclamation
    - getContainers
    - getFragRate
    - getSavings
    - getCosts

Handler

This group of modules consists of some thin wrappers around library functions that allow us to create a set of handler functions all tied to one HTTP request router. We have four routes set up, as follows:

- /container: The route to send a request to reclaim a container. A request for this should contain in its body a JSON file that lists metadata about the requested container.
- /overview: The route to send a request for aggregate reclamation data from the backend. Used only by the frontend when rendering graphs that show aggregate data.
- /activity/{from}/{to}: The route to send a request for reclamation activity data over a range of dates, starting at “from” and ending at “to”. Used only by the frontend when rendering graphs that show a range of activity data.
- /container/{name}: The route to send a request for a JSON metadata file for the container with the given name. This route is mainly for administrative purposes for IBM.

handler:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
handleContainer	Invalid HTTP response writers, Valid HTTP response writers, Malformed HTTP requests, Properly formed HTTP requests	Invalid HTTP response writer, Valid HTTP response writer, Malformed HTTP request, Properly formed HTTP request	Invalid HTTP response writer or malformed HTTP request	Exception
			Valid HTTP response writer and properly formed HTTP request	HTTP Response OK
retrieveOverview	See above	See above	See above	HTTP Response OK, with aggregate data
retrieveActivity	See above	See above	See above	HTTP Response OK, with range of activity data
retrieveActivity	See above	See above	See above	HTTP Response OK, with JSON container metadata file in its body

In general, all of these functions follow the same format: take as input an HTTP response writer (a source to write HTTP responses to) and an HTTP request. If either is invalid or malformed, the function will throw an exception. Otherwise, the function outputs an HTTP OK response to the given response writer, along with any requested resource.

httpListener:

Function Name	Equivalence Partitions	Boundary Values	Example Input	Expected Output
startListener	N/A	N/A	N/A	N/A

This function takes no input and produces no output. Instead, it starts and opens the router, allowing HTTP requests to come in. This function is simple to test: if after executing it, the system is not open for requests then the function has failed. Otherwise it has succeeded.

router:

Function Name	Equivalence Partitions	Boundary Values	Example Input	Expected Output
newRouter	N/A	N/A	N/A	A router with the aforementioned routes attached to it, as well as logging capabilities

Similar to the httpListener function, this function takes no input. However it does produce output, in the form of an HTTP request router. To verify that this function works correctly, we will make sure that all routes are open and working correctly via their respective handler functions.

Reclamation

This set of modules is responsible for acting as a driver for all functionality necessary to reclaim a container. This includes creating a container struct for a JSON container metadata file, handling file and bucket names (obtained from the container metadata) when dealing with AWS, and maintaining stored container metadata files after reclamation.

container:

Function Name	Equivalence Partitions	Boundary Values	Example Input	Expected Output
newContainer	Malformed HTTP requests, Properly formed HTTP requests	Malformed HTTP request, Properly formed HTTP request	Malformed HTTP request	Exception
			Properly formed HTTP request	A container struct which holds all important

				attributes from the JSON file given in the HTTP request
reformatLayout	Containers with any attributes not set, Containers with all attributes properly set	Container with any attributes not set, Container with all attributes properly set	Container with any attributes not set	Exception
			Container with all attributes properly set	N/A
expiredBytes	Containers with any attributes not set, Containers with all attributes properly set	Container with any attributes not set, Container with all attributes properly set	Container with any attributes not set	Exception
			Container with all attributes properly set	Unsigned 32 bit integer, representing the number of bytes in the container that are expired

The rationale for these testing criteria is as follows:

- newContainer: This function takes the JSON from the body of the HTTP request and unmarshals it into a container struct. If the request is malformed (e.g. has no JSON in the body) then an exception is thrown. Otherwise the container struct is made and returned.
- reformatLayout: This function takes in a container struct and reads through its contents to find any expired chunks, which are removed from the container. If the container is malformed in any way, this function will instead throw an exception. Otherwise no output is generated.
- expiredBytes: Similar to reformatLayout, this function takes in a container and reads through its contents. If a malformed container is given, an exception is thrown. Otherwise the function returns the amount of bytes in the container that are expired.

reclamationModule:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
reclaim	Containers with any attributes not set, Containers with all attributes set, Malformed AWS connection sessions, Properly formed AWS connection sessions, String names of	Container with any attribute not set, Container with all attributes set, Malformed AWS connection session, Properly formed AWS connection session, String name of bucket	Container with any attribute not set, malformed AWS connection session, or string name of bucket that does not exist in S3	Exception

	buckets that do not exist in S3, String names of buckets that do exist	that does not exist, String name of bucket that does exist		
			Container with all attributes set, properly formed AWS connection session, and string name of bucket that does exist in S3	N/A
getUsedBytes	See above	See above	See above	N/A
putUsedBytes	See above	See above	See above	N/A
reformatData	Containers with any attributes not set, Containers with all attributes set, Invalid output filename strings, Valid output filename strings, String filenames of input files that do not exist on the host machine, String filenames of input files that do exist on the host machine	Container with any attribute not set, Container with all attributes set, Invalid output filename string, Valid output filename string, String filename of input file that does not exist on the host machine, String filename of input file that does exist on the host machine	Container with any attribute not set, invalid output filename string, or string filename of input file that does not exist on the host machine	Exception
			Container with all attributes set, valid output filename string, and string filename of input file that does exist on the host machine	N/A

The rationale for these testing criteria is as follows:

- reclaim: This function, like many others already covered, takes in identifiers for connecting to AWS and downloading or uploading files. As such, anything wrong with the connection session or S3 bucket name will cause an exception to be thrown. Additionally, this function takes in a container struct for a container that is to be reclaimed. If that struct is malformed, and exception will be thrown. If all parameters are valid, no output is generated and execution continues.
- getUsedBytes: This function takes the same parameters and operates in the same way as the reclaim function, with the addition of calling the function that downloads a file from S3--in this case, the container file referenced in the container struct.

- putUsedBytes: This function operates in the same way as the reclaim function, with the addition of calling the functions that reformat a container file by removing its expired chunks, and then uploading the reformatted file to S3.
- reformatData: This function takes in the container struct used by the previous functions, as well as the name of a file to read in (the container file that was previously downloaded) and the name of a file to output to. If the container is malformed, an exception is thrown. If either filename is malformed or the file to read does not exist, an exception is thrown. Otherwise no output is generated and the reformatted file is uploaded to S3.

Core

This module is simply a driver for all of the functionality that the backend provides. It performs little computation of its own, instead handling the function calls the necessary for any requested service or resource.

core:

Function Name	**Equivalence Partitions**	**Boundary Values**	**Example Input**	**Expected Output**
processContainer	Containers with any attributes not set, Containers with all attributes set	Container with any attribute not set, Container with all attributes set	Container with any attribute not set	Exception
			Container with all attributes set	N/A
getAggregateData	N/A	N/A	N/A	Log of aggregate container and reclamation data
getActivityData	Pairs of datetimes in which the “from” occurs after the “to”, Pairs of datetimes in which the “from” occurs before the “to”	Pair of datetimes in which the “from” occurs after the “to”, Pair of datetimes in which the “from” occurs before the “to”	Pair of datetimes in which the “from” occurs after the “to”	Exception
			Pair of datetimes in which the “from” occurs before the “to”	List of data pertaining to activities that occurred between the specified dates

The rationale for these testing criteria is as follows:

- processContainer: This function takes as input a container struct, then simply sets up AWS and MongoDB connections and calls the reclaim function. Since no real computation is performed here, the only error would be in the case of a malformed container, in which case an exception is thrown. Otherwise no output is generated.

- getAggregateData: This function takes no input and returns the aggregate data on containers and reclamations as stored in the database. Considering this, the only way to test this function is to verify that the functions it calls work, and then make sure the aggregate data is returned from this function properly.
- getActivityData: This function will operate similarly to getAggregateData, with the exception of taking in a pair of dates. As with other date-range oriented functions before, those dates are checked and an exception is thrown if the “from” occurs after the “to”. Otherwise this function returns the list of all activities that occurred within the specified range of dates.

Despite excluding the frontend, out unit testing is extensive since this project is very backend-heavy. Adding to the amount of tests we will need to run to verify our work is the fact that many functions take more than one parameter (particularly those interacting with AWS or MongoDB), wherein each additional parameter at least doubles the amount of tests that need to be done in order to catch every possible combination of values from different equivalence partitions. This results in a considerable number of unit tests to design and run, but we have laid out an effective blueprint for exactly how to design those tests to verify that all of our backend code functions correctly under any given input.

3 Integration Testing

For integration testing, we will be focusing on testing whether the components that compose our software, and the modules that compose the backend of our software, are passing data correctly to one another. This is paramount for our product as often the output of one module is used as the input of another. Therefore, all of our modules must produce the correct data in order for our product to correctly provide a requested process or resource. Through rigorous integration testing, we will ensure that our product's components and modules work together efficiently and accurately.

Figure 1: High Level Architecture

Our product is follows a layered architecture in which each layer has components responsible for different aspects of our project. These layers, as seen above in Figure 1, are the database, service, and presentation layers. The database layer consists of Amazon Web Services (specifically, the S3 data storage service) and a MongoDB database. The service layer is where the majority of our software exists, written in Go. Finally, the presentation layer is the frontend of our software, made with ReactJS and D3JS. In order to make sure our product is fully integrated, we will test that each component is working as intended with one another.

In order to test these components we will create variously formatted containers and make reclamation requests for them. A container reclamation event will involve the use of each component in the system depicted in Figure 1, allowing us to test the behavior and output of each component to ensure that everything is running correctly. Due to the nature of the containers having quite a few attributes, this will also allow us to effectively test for edge cases. These containers will be sent through our three main types of backend modules: cost-benefit analysis modules, the reclamation modules, and the talking modules.

Service Layer

Figure 2: Backend Modules in Service Layer

We will start out with testing the backend modules in the service layer, since this comprises the majority of the functionality of our project and is key in making sure that everything is integrated correctly. To do this we will have to test the modules that make up the layer, as seen in Figure 2 above. Making sure that these modules are integrated correctly ensures that our product has a strong foundation that can support the other two layers.

Cost/Benefit Modules:

First, we will test the integration of our cost-benefit analysis modules since these modules are the first that interact with a requested container. The cost-benefit analysis modules consist of an analysis module and a statistics tracking module. Note that the analysis module talks to the statistics tracking module, and acts as a communication point with the core module when concerning cost-benefit analysis functionality.

These modules receive a container layout from an HTTP request and return both a boolean that indicates whether the container is worth reclaiming and a custom struct that contains statistics about the container. In order to test that this is working properly, we will pass through containers that are eligible for reclamation and containers that do not meet the threshold for reclamation. Doing this will allow us to test that the modules are behaving correctly and passing back the data we expect to either stop reclamation and send a relevant response or to continue with the reclamation process.

Reclamation Modules:

Next, we will test our reclamation modules. Should a container meet the threshold of expired data for reclamation, the container will then be sent as input to the reclamation modules. This means that only containers that are eligible for reclamation are being worked on in these modules, and our tests should reflect that.

Not also that the reclamation process is strongly tied to the presentation layer seen in Figure 1 in two ways: 1) during a container's reclamation, we must request the actual container from S3, and 2) after a container is reclaimed it must be sent back to S3 and its updated metadata JSON file must be stored in the database. Considering this, in order to effectively cover the reclamation modules we must also cover the talking modules, which handle communicating with the components in the presentation layer.

Talking Modules:

The talking modules are considerably important to our product since they communicate with our database, frontend, and AWS. In order to test these modules, we will pass through both correctly and incorrectly formatted containers. The correctly formatted containers will allow us to test that all of the modules we've built are working correctly with valid input. For this purpose we define correct behavior as follows:

- The database should have an updated JSON metadata file for the container.
- The frontend should accurately show statistical data about the container.
- We should be able to verify that the container has been updated in S3 (through the web dashboard system provided by AWS).

By passing through incorrectly formatted containers, we can test that our error handling is working correctly and that we are sending back the appropriate error messages if necessary. After testing of the talking modules is complete, we will have verified that our product works as intended and all of our modules are connected properly.

Database Layer

After making sure that the service layer is fully integrated, we need to test that the backend can correctly communicate with S3 data storage and the database. To do this, we will be using testing containers. To make sure that AWS is integrated with the rest of the product we will be using the backend to push and pull containers from the cloud. This will ensure that our backend has the credentials and capability to move containers whenever necessary. To test the integration of our database, we will write container statistics to it and make sure we are able to retrieve those same statistics. After testing the integration this layer and making sure that everything is working as intended, we will move on to testing the presentation layer.

Presentation Layer

Since our product is very focused on the backend process of reclamation, frontend is of smaller concern to us and thus is the last component we will test. This component communicates with the backend using HTTP requests to get data about containers and reclamations with which to fill in rendered charts. In order to test the integration of the frontend with the backend we will be using several HTTP requests to retrieve data from the other layers.

As discussed in the Unit Testing section, there are two types of requests that the frontend can make to the backend: aggregate reclamation data, and date-ranged reclamation data. In the former case, once the frontend has been started up we will visit the appropriate web page for aggregate data and verify that all of the aggregate data, as seen in the database, is being rendered in a consumer-facing UI.

In the latter case, we will visit the web page for date-ranged data and specify a valid range of dates, after which we will verify that all reclamation activities occurring within the specified range, as seen in the database, are being rendered in a similar consumer-facing UI.

By performing this testing of the presentation layer, we ensure that not only is the presentation layer correctly communicating with the service layer, but also that the service layer is correctly communicating with the database layer, since these requests from the frontend are for resources held in the database. This will show full integration of our product, proving that all of the components and modules that we have designed are working properly as one system.

4 Usability Testing

Database storage administrator's will utilize the frontend to visualize patterns and trends in application activity from the backend. This requires that our frontend is vetted and conforms to UI/UX best practices. Our frontend design has been meticulously planned through iterative mockup design with our client. However, usability testing will strengthen our design by providing insight from a larger pool of users.

Population

The population we are meant to satisfy is database storage administrators. This means we can assume that our audience has some level of technical expertise. The population will know the inner workings of database storage reclamation/administration, but may not have any programming expertise. Due to this, our frontend displays any relevant reclamation data while omitting any unnecessary coding/implementation details.

Methods

Our testing plan will comprise two separate parts:

- **Categorical Acceptance:** User will be given flashcards with two different categories written on them. They will be asked to match them according to what they find most logical.
- **Live Usability:** We will record users interacting with our application given a testing script. The user will be asked to talk out loud about how they think as they are interacting with the application.

Plan

We plan on selecting a sample of 10-15 individuals, all 18 years of age or older, who have some level of technical experience. Little to no technical experience will make it difficult for the users to navigate the items we will ask them to identify, and would be inaccurate to the intended audience of our frontend. We will perform this test on users individually and start them with the categorical acceptance test. The categorical acceptance test will help us determine how the user will think about our application prior to ever seeing it. We will use it to determine the most logical hierarchical layout of our elements as well as an appropriate color pallette. For this test, we will ask the user to match items from the left column with items in the right column:

Category

Overview	Top
Activity Over Time	Middle
Activity Details	Bottom

Color (color cards can be matched to multiple items in right column)

Blue	Costs
Green	Savings
Red	Storage Bin
Black	Percentage
Purple	Resource Usage
White	Resource Reclamation

After completing this test, users will be provided a testing script so they can complete a live usability test. The script will include the following steps:

1. Attempt to find the total fragmentation percentage on the webpage. How easy was it to find on a scale to 1-10? What would you change and what did you like?
2. Repeat step 1 but look for the following 2 items instead: total storage cost, total savings.

3. Try to create a line graph that represents the total storage reclamation activity over the month of March. How easy was it to find on a scale to 1-10? What would you change and what did you like?
4. Repeat step 3 except try to find total savings over the month of March.
5. Attempt to find reclamation records for the day of 3/27/18. How many records are for that day?

While completing this script, we will record the user so that we can see how they interact with the application. This will help us determine how easily our items were to find, as well as show us where users thought these items should be. This will give us a clear indication as to whether or not our application will require any reorganization or additional visual cues.

Results

Once our focus group has completed their individual testing sessions, we will compile the data and see if we can identify any important patterns. We will do so by creating a table representing commonality among all users. For example, if 70% of users misidentified item 2 in our testing script, we will reorganize our layout to resolve this. Similarly, if 70% of users chose Green as the color that best represented the Savings category, we will adjust our color palette to match. These tests will help lay out the groundwork for any reorganization (both design and layout-wise) that will improve our application’s overall usability.

5 Conclusion

Like any software, our product will require a regimen of software testing to verify that it not only functions correctly under any given inputs and is highly error-resistant, but also that it satisfies our project sponsor’s goals and is usable by members of our target audience. We have laid out a plan for exactly how our software will be tested which includes rigorous and lengthy unit testing, moderate integration testing, and light usability testing.

By creating this test plan and adhering to it we can give a strong assurance to IBM that this product will appropriately handle any inputs, provide any resources or services requested of it, and do so in a way that is easy to understand for end-users in our target audience.