Introduction

This document provides details (not always complete) about how the code works, the various APIs and some details about how to use the different parts of the support tools. Hopefully, the information in this document is helpful, but if there are any areas that would benefit from more detail or explanation, please let me know.

Overview

The Object Server implementation that is in this git repository (https://github.com/notterness/ObjectServer.git) is an example used to try out a number of different design ideas and technologies.

The Object Server actually consists of five different services (meaning potentially independent applications that can reside at different IP addresses and different TCP Ports). The services are:

- **Object Server** - This is the interface that deals with the saving and retrieval of objects. This is what handles the following methods:
  - ObjectPut - Write an object to storage
  - ObjectGet - Retrieve an object
  - ObjectDelete - Delete an object
  - ObjectList - List the current saved objects
  - CreateBucket - Buckets provide a method to group objects
  - DeleteBucket - This is used to remove a Bucket from a Namespace. A Bucket must be empty (i.e. not contain any Objects) for it to be deleted.
  - CreateNamespace - Partially implemented
  - DeleteNamespace - Not implemented
- **Storage Server** - This is what saves the pieces of the objects (chunks) onto the disk. For this implementation, each chunk is represented by a file, but in a real Storage Server, the chunks would simply be written to a block device. There are multiple instances of Storage Servers as that is how chunk redundancy is obtained. The Storage Servers support the following methods:
  - ChunkPut - Writes a chunk to a file
  - ChunkGet - Reads a chunk from a file
  - DeleteChunk - Deletes the file that represents a chunk.
- **Chunk Manager** - This is what manages the chunks and the various services. The services are managed through database entries (this provides a generic lookup method to access the various services and could be replaced by a version of DHCP or something to allow for easy lookup based on a server name). The chunks are associated with a Storage Server and this service provides a centralized way for Object Servers to allocate them. This supports the following methods:
  - CreateServer - This is what adds servers that manage a service to the database. AllocateChunks - This allocates a set of chunks from the various Storage Servers based upon the object's storage tier.
  - DeleteServer - Remove a service from availability (not supported in the initial submission).

- DeleteChunk - Remove a chunk from use. This will in turn call DeleteChunk on the appropriate Storage Server to clear out the backing storage.
- ListServer - List all the servers for the different services.
- ListChunks - List the chunks on the Storage Servers and their state. This allows searching by numbers different methods to limit the amount of data returned.
- **Account Manager** - This service is used to manage customer Tenancies and User account within the customer Tenancy. It provides the security validation for the operations a user can perform. Users within a Tenancy are provided with a set of roles they are allowed to perform and this is used to limit what methods they are allowed to execute in different services. The Account Manager supports the following methods:
  - CreateTenancy - This creates the top level grouping for a customer to organize resources.
  - DeleteTenancy - Not implemented.
  - CreateUser - This creates a user within a Tenancy. The user is what provides access to the resources within a Tenancy.
  - DeleteUser - Not implemented.
  - GetAccessToken - The access token is provided with the different requests to the Object Server (so far, but needs to be extended to the different services) to provide a quick lookup to determine if a user has access to the Tenancy and the different resources contained within it.
- **Client Test** - This provides numerous different test cases to validate the behavior of the various different services. It also provides the entry point for running the various services within the IntelliJ framework.
- **Health Check** - This is a simple module that allows a health-check command to be sent to a service running within the Kubernetes POD. It is a useful debug tool to determine if a service is working and responding at the expected IP address and Port.

The different services can all run within three different frameworks: IntelliJ, as Docker Images and within a Kubernetes Pod. The setup of the Kubernetes Pod is within the repository under the "kubernetes" module. There are the ".conf" files to setup the different fluent-bit loggers for each service along with the "socat" image to allow access to the Kubernetes API. This is also where the Pod setup yaml files reside.

The Kubernetes Pod also brings in Socat (to allow access to the Kubernetes API) and Fluent-Bit (to push logs to an external logger, in this case Elasticsearch). Once the logs are pushed to Elasticsearch, then Kibana can be used to view and search them.

The implementation uses MySQL as a backing database to store persistent data in. There are three different databases created. There is to ObjectStorageDb that is used by the Object Server to maintain information critical to its operation. There is the ServiceServerDb (not the best name) used to manage the data for the Chunk Manager. In addition to the normal table create, update and modify operations, there are examples of stored procedures in the ServerChunkMgr

class. There is the AccessControlDb which maintains information about the customer tenancies and the users assigned to the tenancies.

The KubernetesInfo class provides some examples of how to access details of the Kubernetes Pod through the Java API. This is what is used to populate the lookup information about the different services when they are running within a Kubernetes POD (essentially providing a cheap version of DNS that is run once at the startup of the Pod). Currently, there is an issue with resources from the Kubernetes API not cleaning up correctly in the initial submission.

Definitions

Tenancy - This acts as the highest construct and provides an organization of all resources owned by a client (a client can also have multiple tenancies, but they are distinct and resources cannot be shared across tenancies).

Namespace - Each region within a tenancy will have a unique namespace where all the Buckets within a region are placed.

Bucket - A client can create as many Buckets as they desire within a Namespace. The Buckets provide a method to group Objects. The Bucket is where the redundancy level for the Objects within the Bucket is specified.

Object - This is what contains the clients data. There may be multiple versions of the same Object with each version containing different data.

Storage Tier - This is what defines the level of protection an Object has against Storage Server failures. A higher level of redundancy means that there can be more Storage Server or disk failures before the data can no longer be accessed. The Storage Tier also defines the level of accessibility for an Object. For example, if the object is accessed at a high frequency it needs to be placed on higher performing storage versus Objects that are accessed infrequently, which can be placed on slow access media (i.e. tape).

Chunk - A standard size block of data that objects are broken up into to distribute the data across multiple Storage Servers. A chunk is the unit of redundancy for an Object. For example, if the Standard storage tier is specified for a Bucket, then there are three copies of each Chunk placed on different Storage Servers.

Unique ID (UID) - The UID is a 36 character (with a format that looks like "aab5d5fd-70c1-11e5-a4fb-b026b977eb28") String that represents a 128 bit value. The dashes are not part of the actual 128 bit value and are present in the String to improve readability.

Tenancy UID - The unique identifier of the Tenancy that the Namespace must reside within.

Storage Tier - The storage tier defines the reliability and access characteristics of an Object. Currently only the Standard Tier is supported. The possible storage tiers are:
- Standard - 3 copies of the data within the same data center
- Intelligent-Tiering - Moves data between fast and slow disk depending on access patterns. Always uses 3 copies of the data.

- Standard-IA (Infrequent Access) - 3 copies on slow disk.
- OneZone (Another form of Infrequent Access with less redundancy) - 2 copies of the data on slow disk.
- Archive (slower access than Standard-IA)
- DeepArchive (slowest access of all, data may be kept on offline storage)

Access Management

Access Management is how the access to resources and data is controlled by the system. Customers own a Tenancy, which is the top level container for all of a customer's data and the resources used to manage that data. Within a Tenancy, a customer will have users with different levels of access. To manage access, an Access Token is associated with every user. The Access Token is provided with commands to the Object Server (though this needs to be expanded to support other services). This Access Token is used to look up the Tenancy the request is for and the permissions for the user. Assuming the Tenancy is valid and the user has sufficient permissions than the request is permitted.

The user database table stores the following fields in a secure manner:
- User Name - MySQL AES_ENCRYPT() using a passphrase the customer provides when they create their Tenancy.
- User Password - MySQ AES_ENCRYPT() using the Tenancy passphrase.
- Access Token - This consists of the userUID, tenancyUID and accessRights run through an Md5 digest. This creates a 32 character value that is used to the Access Token. A more secure token could be created with the SHA2() hash. The reason for not using this initially is the size of the token. Using the SHA2() hash length of 224 results in a String of 56 characters versus 32 for the MD5() digest.

Event Driven Object Server Design

This presents a design that is reactive in that operations take place in response to events that they have registered with occurring. The operations are responsible for setting up their own dependencies and for obtaining required information from the overall request state as needed to determine the work they will perform. Each operation is designed to handle a small piece of work that is easy to implement and test. To create more complex operations, an operation can create multiple sub-operations and setup itself to be evented when the sub-operations complete.

They key idea is that the intelligence and how the operations interact is setup through dependencies and what the operations produce. This is a distributed intelligence model and does not have a single overall state machine that can be examined. This is slightly more complex to understand, but by distributing the intelligence, it makes it easier to validate individual or collections of operations. The design allows for easy expansion to handle different operations within the overall operation state.

The dependencies between the states are defined in code and setup at object initialization time. In other words, the dependencies are static for a particular HTTP Request type and will not

change. By having the dependencies static, it allows the interactions between operations to be easily tested.

Goals of the Design
The design intends to accomplish the following:
- Have a design that is easy to understand and maintain (a subjective goal).
- Implement a high performance Object Server to allow high overall throughput for each client connection.
- Provide a design where each piece can be individually tested and does not require the entire code base.
- Provide a design where the use of threads is explicit and it is easy to understand the threads responsibility.
- Insure that the design can be tested independently of the data transport mechanism (the design will use the NIO socket layer to communicate with clients and Storage Servers).

Terminology
Event – For this document, an event is something that is generated when some piece of work that is interesting completes. When an event takes place, that allows a different piece of work to take place. Events and their handlers can be considered the glue that helps to define the work ordering needed to complete an HTTP request. Items that generate events include:
- Data being placed into a ByteBuffer from the SocketChannel read operation
- Data being written out of a ByteBuffer through a SocketChannel write operation
- All the data for a client object being read into buffers (meaning the entire object has been received)
- The HTTP Request from the client has been parsed and the HTTP method determined
- The HTTP Request has been sent to the Storage Server(s)
- The HTTP Response has been received from the Storage Server(s)
- All the data for a chunk has been written to the set of Storage Servers

API Definitions

POST Tenancy - Account Manager Service
This API is used to create the customer Tenancy. This method is only valid when sent to the Account Manager Service. This is the top level construct which all object information is stored under.

URI Fields
- *Tenancy Name* (required) "/t" - This is used to indicate that this POST method is being used to create a customer Tenancy. There is no information that follows the "/t".

Headers
- *Host* (required) - Who is sending the request.

- *opc-client-request-id* (optional) - A unique identifier for this request provided by the client to allow then to track their requests.
- *x-content-sha256* (required) - The computed Sha256 Digest for the content related to the Tenancy.
- *Content-Length* (required) - The size in bytes of the Tenancy definition content information.

Content
- *tenancyName* (required) - The name of the Tenancy to be created. It must be unique when combined with the customer field.
- *customer* (required) - The owner of the Tenancy.
- *passphrase* (required) - The customer unique string that is used when encrypting and decrypting information within the Tenancy.

Example
POST uri: /t version: null
ClientTest httpPort: 0
content-type: application/json
connection: keep-alive
accept: */*
user-agent: ClientRequest/0.0.1
accept-language: en-us
accept-encoding: gzip, deflate
client-request-id: PostTenancySimple-7-22-2020.01
content-sha256: Ap3Bw7mQJrMPP2lKnwgMOfFRe+n4y7KDtkE9b3RcQ68=
content-length value: 108
{
    "tenancyName": "Tenancy-Test-Post",
    "customer": "testCustomer-1",
    "passphrase": "test-passphrase"
}

Response
The response to the POST Tenancy request returns the following information in the response headers. The etag returned is one associated with the tenancy account that was created.

opc-client-request-id: PostTenancySimple-7-22-2020.01
opc-request-id: 2
etag: 500fad02-db88-11ea-bcdf-25f6620803f3

POST User - Account Manager Service

This API is used to create the user accounts within a customer's Tenancy. This method is only valid when sent to the Account Manager Service. The users are what are used to allow access to the Object Store.

URI Fields

- *User Name* (required) "/u" - This is used to indicate that this POST method is being used to create a user within a customer's Tenancy. There is no information that follows the "/u".

Headers

- *Host* (required) - Who is sending the request.
- *opc-client-request-id* (optional) - A unique identifier for this request provided by the client to allow then to track their requests.
- *x-content-sha256* (required) - The computed Sha256 Digest for the content related to the Tenancy.
- *Content-Length* (required) - The size in bytes of the user definition content information.

Content

- *tenancyName* (required) - The name of the Tenancy to be created. It must be unique when combined with the customer field.
- *customer* (required) - The owner of the Tenancy.
- *userName* (required) - The name the user will use to login in to the system and to send requests to the Object Server.
- *password* (required) - The password for the user.
- *permissions* (required) - What permissions are associated with the user. The defines what operations the user is allowed to perform.

Example

POST uri: /u version: null
ClientTest httpPort: 0
content-type: application/json
connection: keep-alive
accept: */*
user-agent: ClientRequest/0.0.1
accept-language: en-us
accept-encoding: gzip, deflate
client-request-id: PostUserSimple-7-24-2020.01
content-sha256: eTWLpRTW3t+TZpsO8h5/HnPMhOhkRrgyJNshGSaAQUo=
content-length value: 160
{
    "tenancyName": "Tenancy-Test-Post",
    "customer": "testCustomer-1",
    "userName": "TestUser1@test.com",

```
    "password": "testpassword",
    "permissions": "32",
}
```

Response

The response to the POST User request returns the following information in the response headers. The etag returned is one associated with the user account that was created.

opc-client-request-id: PostUserSimple-7-24-2020.01
opc-request-id: 3
etag: 51752aaa-db88-11ea-bcdf-25f6620803f3

GET AccessToken - Account Manager Service

This API is used to obtain the access token for a particular user within a customer's Tenancy. This method is only valid when sent to the Account Manager Service. The access toke is what is required to perform any requests to the Object Store. The access token is an encrypted string that is used to determine the Tenancy the request is for and the access permissions for the user to determine if they are allowed to perform the request.

URI Fields
- *User Name* (required) "/t" - This is used to indicate that this GET method is being used to obtain the user's access token for the user within a customer's Tenancy. The name of the Tenancy must follow the "/t/".

Headers
- *Host* (required) - Who is sending the request.
- *opc-client-request-id* (optional) - A unique identifier for this request provided by the client to allow then to track their requests.
- *x-content-sha256* (required) - The computed Sha256 Digest for the content related to the access token request.
- *Content-Length* (required) - The size in bytes of the user definition content information.

Content
- *customer* (required) - The owner of the Tenancy.
- *userName* (required) - The name the user will use to login in to the system and to send requests to the Object Server.
- *password* (required) - The password for the user.

Example
GET uri: /t/Tenancy-Test-Post version: null
ClientTest httpPort: 0
content-type: application/json
connection: keep-alive

accept: */*
user-agent: ClientRequest/0.0.1
accept-language: en-us
accept-encoding: gzip, deflate
client-request-id: GetAccessTokenSimple-7-31-2020.02
content-sha256: VvNGS2mvIxVFXRCt5O/1C3s068BgIdnLkc8wj4spt/4=
content-length value: 100
{
   "customer": "testCustomer-1",
   "userName": "TestUser1@test.com",
   "password": "testpassword",
}

Response

The response to the GET AccessToken request returns the following information in the response headers. The etag returned is one associated with the user the request is for.

opc-client-request-id: GetAccessTokenSimple-7-31-2020.02
opc-request-id: 4
etag: 51752aaa-db88-11ea-bcdf-25f6620803f3
access-token:3563653839653465643833623630353864463136663635373862393634343063

POST Server - Chunk Manager Service

This API is used to create a Storage Server that can be used to provide storage for objects or it will create an entry for a Service so that the Chunk Manager Service can be queried for information (IP address, Port) for other clients to access the service. This method is only valid when sent to the Chunk Manager Service.

URI Fields

• *User Name* (required) "/" - This is used to indicate that this POST method is being used to create a Storage Server entry. There is no information that follows the "/".

Headers

• *Host* (required) - Who is sending the request.
• *opc-client-request-id* (optional) - A unique identifier for this request provided by the client to allow then to track their requests.
• *x-content-sha256* (required) - The computed Sha256 Digest for the content related to the Tenancy.
• *Content-Length* (required) - The size in bytes of the user definition content information.

Content

- *service-name* (required) - The name of the Storage Server to be added. It must be unique within the Chunk Manager Service.
- *server-ip* (required) - The IP address of the Storage Server.
- *server-port* (required) - The port that will be used to access the Storage Server.
- *allocated-chunks* (required) - The number of chunks of storage that can be allocated on the Storage Server. If the allocated-chunks are set to 0, the interface is creating a Service reference.
- *storageTier* (required) - This is the type of storage that is provided by this Storage Server (i.e. it could be STANDARD_TIER which uses SSD drives or ARCHIVE_TIER which uses spinning media). For Services, this should be set to StorageTierEnum.INVALID_TIER.

Example

POST uri: / version: null
ClientTest httpPort: 0
content-type: application/json
connection: keep-alive
accept: */*
user-agent: ClientRequest/0.0.1
accept-language: en-us
accept-encoding: gzip, deflate
client-request-id: PostCreateServer-5-20-2020.01
content-sha256: T/gQnj1DuB8ouHwNqDWjFWP7lPGHH3MP5Udwo3Qoqq8=
content-length value: 154
{
    "service-name": "storage-server-4",
    "server-ip": "localhost",
    "server-port": "5013",
    "allocated-chunks": "10",
    "storageTier": "Standard",
}

Response

The response to the POST Server request returns the following information in the response headers. The etag returned is one associated with the server that was created within the Chunk Manager Service.

opc-client-request-id: PostCreateServer-5-20-2020.01
opc-request-id: 2
etag: cc1a0ae4-dbf3-11ea-bcdf-25f6620803f3

GET Chunks - Chunk Manager Service

This API is used to obtain the chunks of storage on the various Storage Servers that will be used to store an Object. An Object is made up of multiple chunks that may be replicated as determined by the storage tier for the Bucket that Object is being stored within. This API will return at least one chunk and may return more that are used to save the same piece of the Object. Each chunk returned by this API will hold the same data after the object chunk writes are completed.

URI Fields

- *Qualifier* (required) "/" - This is used to indicate that this GET method is being used to obtain an array of chunks. There is no information that follows the "/".

Headers

- *Host* (required) - Who is sending the request.
- *opc-client-request-id* (optional) - A unique identifier for this request provided by the client to allow then to track their requests.
- *x-content-sha256* (required) - The computed Sha256 Digest for the content related to the Tenancy.
- *Content-Length* (required) - The size in bytes of the user definition content information.

Content

- *object-chunk-number* (required) - This is more for tracking purposes and it represents the index of the chunk in the object (starting at zero). It can be determined by the following: (Object chunk write starting byte index) / (chunk size).
- *storageTier* (required) - This is used to determine which Storage Server(s) will be used to when allocating the chunks and the chunk redundancy (how many copies of the data are required).

Example

```
GET / HTTP/1.1
ClientTest httpPort: 0
content-type: application/json
connection: keep-alive
accept: */*
user-agent: ClientRequest/0.0.1
accept-language: en-us
accept-encoding: gzip, deflate
client-request-id: PutObjectSimple-5-12-2020.01
content-sha256: 7QgI5LXCNR7e/hPDfXUF8gMhDsoPaZL0q7Wxn4xH8Lo=
content-length value: 63
{
```

```
        storageTier: Standard
        object-chunk-number: 0
}
```

Response

The response builder for the GET Chunks request is located in AllocateChunks.java.

```
opc-client-request-id: PutObjectSimple-5-12-2020.01
opc-request-id: 12
allocated-chunks: 3
content-length: 1024


{
    "chunk-1":
    {
        "service-name": "storage-server-2"
        "chunk-lba": "131072"
        "storage-id": "3"
        "chunk-id": "22"
        "server-ip": "127.0.0.1"
        "chunk-location": "c74024c2-dbf3-11ea-bcdf-25f6620803f3"
        "server-port": "5012"
        "chunk-tag": "c74024c2-dbf3-11ea-bcdf-25f6620803f3"
    }
    "chunk-2":
    {
        "service-name": "storage-server-1"
        "chunk-lba": "262144"
        "storage-id": "2"
        "chunk-id": "13"
        "server-ip": "127.0.0.1"
        "chunk-location": "c73b11f8-dbf3-11ea-bcdf-25f6620803f3"
        "server-port": "5011"
        "chunk-tag": "c73b11f8-dbf3-11ea-bcdf-25f6620803f3"
    }
    "chunk-3":
    {
        "service-name": "storage-server-4"
        "chunk-lba": "131072"
        "storage-id": "4"
        "chunk-id": "32"
        "server-ip": "127.0.0.1"
        "chunk-location": "cc1cd56c-dbf3-11ea-bcdf-25f6620803f3"
```

```
    "server-port": "5013"
    "chunk-tag": "cc1cd56c-dbf3-11ea-bcdf-25f6620803f3"

  }
}
```

POST Bucket - Object Server

This API is used to create a Bucket for a client to group Objects within a Namespace. The Bucket is what defines the storage tier for the Objects contained within it.

URI Fields

- *namespaceName* (required) "/n/" - This is the namespace that holds the Bucket where the Object(s) will be kept.
- *bucketName* (required) "/b/" - This has not content, just the "/b/" is provided for the URI.

Headers

- *Host* (required) - Who is sending the request.
- *opc-client-request-id* (optional) - A unique identifier for this request provided by the client to allow then to track their requests.
- *x-content-sha256* (required) - The computed Sha256 Digest for the content related to the Bucket.
- *Content-Length* (required) - The size in bytes of the bucket content information.

Content

- *compartmentId* (required)
- *namespace* (required) - The Namespace where the Bucket will be created within.
- *name* (required) - The name of the Bucket to create.
- *storageTier* (optional) - This is what defines the storage tier for all the Objects contained within the Bucket. If the *storageTier* is not specified, it will use the Standard Tier.
- *objectEventsEnabled* (optional) - Set the "true" for events to be generated when changes are made within the Bucket. Set to "false" to disable events for the Bucket. Currently, only "false" is supported.
- *freeformTags* (optional) - A collection of tags which can be used to search for the Bucket.
- *definedTags* (optional) - A set of user defined tags which can be used to perform Bucket searches.

Example

POST /n/Tenancy-12345-abcde/b/ HTTP/1.1
ClientTest httpPort: 0
content-type: application/json
connection: keep-alive

accept: */*
user-agent: ClientRequest/0.0.1
accept-language: en-us
accept-encoding: gzip, deflate
client-request-id: PostBucketsSimple-6-12-2020.01
access-token: 32323039323565663262663865373839656134313632643235611663638633839
content-sha256: 74Fk0G+3Qs1ND9v2K6JnzCon8IqcqoQpzA2FPq2QDZ8=
content-length value: 308
{
    "compartmentId": "clienttest.compartment.12345.abcde",
    "namespace": "testnamespace",
    "name": "CreateBucket_Simple",
    "objectEventsEnabled": false,
    "freeformTags": {"Test_1": "Test_2"},
    "definedTags":
    {
       "MyTags":
       {
          "TestTag_1": "ABC",
          "TestTag_2": "123",
       }
    }
}

DELETE Bucket - Object Server

This API is used to delete a Bucket from a Namespace. In order to be deleted, a Bucket must not have any Objects contained within it. The following are its URI fields and the headers:

URI fields
• *Namespace Name* (required) "/n/" - This is the namespace that holds the bucket where the object will be kept in.
• *Bucket Name* (required) "/b/" - This is the bucket that is to be deleted.
• *Object Name* (must not be present) "/n/" - The lack of the "/n/" piece of the URI is what differentiates between the DELETE Bucket and DELETE Object methods.

Headers
• *Host* (required) - Who is sending the request.
• *opc-client-request-id* (not required) - A unique identifier for this request provided by the client to allow them to track their requests.
• *access-token* (required) - Used to validate the user making the request.
• *Content-Length* (required) - Must be set to 0.

Example
DELETE /n/Namespace-xyz-987/b/CreateBucket_Simple HTTP/1.1
Host: ClientTest-DeleteBucket
Content-Type: application/json
Connection: keep-alive
Accept: */*
User-Agent: Rested/2009 CFNetwork/978.0.7 Darwin/18.7.0 (x86_64)
Accept-Language: en-us
Accept-Encoding: gzip, deflate
opc-client-request-id: DeleteBucket-6.1.5
access-token: 6462363436376561663365656563646262616233363435633231626662623539
Content-Length: 0

Bucket Delete Succeeded (NO_CONTENT_204)
Status: 204
opc-client-request-id: DeleteBucketSimple-6-18-2020.01
opc-request-id: 9

Bucket Not Empty (CONFLICT_409)
To fix the error, delete all the Objects within the Bucket and retry the command.

Status: 409
opc-client-request-id: DeleteBucketSimple-6-18-2020.01
opc-request-id: 7
Status: 409
{
  "code": 409
  "message": "Bucket must be empty prior to deletion"
  "bucketName": "CreateBucket_Simple"
  "objectCount": 1
}

PUT Object - Object Server
This API is used to upload an Object to the Object Server. The following are its URI fields and headers:

URI fields
- *Namespace Name* (required) "/n/" - This is the namespace that holds the bucket where the object will be kept in.
- *Bucket Name* (required) "/b/" - This is the bucket that will hold the object.

- *Object Name* (required) "/n/" - This is the name of the object where the file data will be retained within the Object Server.

Headers

- *Host* (required) - Who is sending the request.
- *opc-client-request-id* (not required) - A unique identifier for this request provided by the client to allow them to track their requests.
- *if-none-match* (not required) - This can only be set to "*". If it is set to "*" and there is already an object that matches the namespaceName/bucketName/objectName then an error will be returned. If this is not set and there is a matching object, a new record will be created with a higher version number. It is possible for the same object to have multiple versions.
- *Content-MD5* (not required) - The computed Md5 Digest for the file being uploaded to the Object Server.
- *access-token* (required) - Used to validate the user making the request
- *Content-Length* (required) - The size in bytes of the object being uploaded.

Example

PUT /n/Namespace-xyz-987/b/CreateBucket_Simple/o/TestObject-1234-abcd HTTP/1.1
Host: ClientTest-PutObject
Content-Type: application/json
Connection: keep-alive
Accept: */*
User-Agent: Rested/2009 CFNetwork/978.0.7 Darwin/18.7.0 (x86_64)
Accept-Language: en-us
Accept-Encoding: gzip, deflate
opc-client-request-id: PutObject-6.1.5
Content-MD5: RUyS5v1cKdbFcsRdO7NgfQ==
access-token: 646236343637656166336565656364626261623336343563332316266626623539
Content-Length: 8192

Response (OK_200)

HTTP/1.1 200 OK
opc-client-request-id: PutObject-6.1.5
opc-request-id: 24
opc-content-md5: RUyS5v1cKdbFcsRdO7NgfQ==
Etag: 967d6800-afdd-11ea-9636-6cab1464ba76
last-modified: 2020-06-16 08:28:03
Content-Length: 0

Failed Md5 Comparison Response (UNPROCESSABLE_ENTITY_422)

Status: 422
opc-client-request-id: PutObjectBadMd5-6-17-2020.01

opc-request-id: 3
```
{
  "code":422
  "message": "Failed Md5 Compare"
  "Content-MD5": "vXhywKtpTIEQPRlFoO8f4g=="
  "Computed-MD5": "RUyS5v1cKdbFcsRdO7NgfQ=="
}
```

Content-Md5 Header Not Sent Response (BAD_REQUEST_400)

Status: 400
opc-client-request-id: PutObjectBadMd5-6-17-2020.01
opc-request-id: 3
```
{
  "code": 400
  "message": "Invalid or missing Content-Md5"
  "Computed-MD5": "RUyS5v1cKdbFcsRdO7NgfQ=="
}
```

Unable to Allocate Chunks on Storage Servers (INTERNAL_SERVER_ERROR_500)

Status: 500
opc-client-request-id: PutObjectValidMd5-6-17-2020.01
opc-request-id: 2
```
{
  "code":500
  "message": "Unable to allocate any chunks on Storage Servers"
}
```

Invalid Bucket (NOT_FOUND_404)

Status: 404
opc-client-request-id: PutObjectBadBucket-5-12-2020.01
opc-request-id: 12
```
{
  "code": 404
  "message": "Bucket not found"
  "bucketName": "BadBucketName"
}
```

Invalid Namespace(NOT_FOUND_404)

Status: 404
opc-client-request-id: PutObjectBadNamespace-5-12-2020.01
opc-request-id: 13
```
{
```

```
 "code": 404
 "message": "Namespace not found"
 "namespaceName": "BadNamespaceName"
}
```

GET Object - Object Server

This API is used to retrieve an Object from the Object Server. The following are its URI fields and headers:

URI fields
- *Namespace Name* (required) "/n/" - This is the namespace that holds the bucket where the object will be kept in.
- *Bucket Name* (required) "/b/" - This is the bucket that will hold the object.
- *Object Name* (required) "/n/" - This is the name of the object to be uploaded from the Object Server.

Headers
- *Host* (required) - Who is sending the request.
- *opc-client-request-id* (not required) - A unique identifier for this request provided by the client to allow them to track their requests.
- *if-match* (not required) - This is used to contain the specific ETag (entity tag - a unique ID associated with every object that is present in the Object Store) of the object to upload.
- *versionId* (not required) - This is set to the specific version of the object the client wants returned. If it is not set, then the most recent version of the object (i.e. object with the highest version number) will be returned.
- *access-token* (required) - Used to validate the user making the request
- *Content-Length* (required) - The size in bytes of the object being uploaded.

If both *versionId* and *if-match* are set, then the *if-match* will override the *versionId* matching.

Example
GET /n/Namespace-xyz-987/b/CreateBucket_Simple/o/TestObject-1234-abcd HTTP/1.1
Host: ClientTest-GetObject
Content-Type: application/json
Connection: keep-alive
Accept: */*
User-Agent: Rested/2009 CFNetwork/978.0.7 Darwin/18.7.0 (x86_64)
Accept-Language: en-us
Accept-Encoding: gzip, deflate
opc-client-request-id: GetObject-6.2.5

Content-MD5: RUyS5v1cKdbFcsRdO7NgfQ==
access-token: 646236343637656166336565656364626261623336343563323162662623539
Content-Length: 8192

Response (OK_200)
HTTP/1.1 200 OK
Content-Type: application/json
opc-client-request-id: GetObject-6.1.5
opc-request-id: 24
opc-content-md5: RUyS5v1cKdbFcsRdO7NgfQ==
Etag: 967d6800-afdd-11ea-9636-6cab1464ba76
last-modified: 2020-06-16 08:28:03
versionId: 12
Content-Length: {size in bytes of the object being returned}
<Object Data>

Generic Failure Response
HTTP/1.1 {Error Number} FAILED
Content-Type: application/json
opc-client-request-id: GetObject-6.1.5
opc-request-id: 24
Etag: 967d6800-afdd-11ea-9636-6cab1464ba76
Content-Length: {size in bytes of the error content being returned}
<Error Content Message>

Object Not Found (PRECONDITION_FAILED_412)
HTTP/1.1 412 FAILED
Content-Type: application/json
opc-client-request-id: GetObject-6.1.5
opc-request-id: 24
Status: 412
{
  "code": "412"
  "message": "Object not found - TestObject-1234-abcd"
}

Invalid Bucket (NOT_FOUND_404)
Status: 404
opc-client-request-id: GetObjectBadBucket-5-13-2020.01
opc-request-id: 10
{
  "code": 404

```
  "message": "Bucket not found"
  "bucketName": "BadBucketName"
}
```

Invalid Namespace(NOT_FOUND_404)

Status: 404
opc-client-request-id: GetObjectBadNamespace-5-12-2020.01
opc-request-id: 11
```
{
  "code": 404
  "message": "Namespace not found"
  "namespaceName": "BadNamespaceName"
}
```

Invalid Access Token Provided (NETWORK_AUTHENTICATION_REQUIRED_511)

This is returned if the *accessToken* header is not valid. This means that it was unable to verify the user and the Tenancy for the provided *accessToken*.

Status: 511
opc-client-request-id: GetObjectInvalidToken-6-19-2020.01
opc-request-id: 12
```
{
  "code":511
  "message": "Unable to access Tenancy information, invalid accessToken"
}
```

No Access Token Provided (NETWORK_AUTHENTICATION_REQUIRED_511)

This is returned if the *accessToken* header is not provided as part of the GetObject request.

Status: 511
opc-client-request-id: GetObjectInvalidToken-6-19-2020.01
opc-request-id: 12
```
{
  "code":511
  "message": "Unable to access Tenancy information, accessToken not provided"
}
```

Code Overview

Object Overview

To help describe the operation of the system, there are the following objects:

- **RequestContext** – An overall object that is a placeholder for the state associated with a particular HTTP request.
- **HTTP Information** – The information pulled out of the HTTP headers and the URI. There are two pieces of HTTP information related to a request, the incoming request headers and content and the outgoing response headers and content.
- **BufferManager** – An object used to keep track of Buffers that has registration functions that allows "events" to be triggered when producers change the state of a buffer. The BufferManager provides the following methods:
    - register – This allows Operations to register as producers or consumers of Buffers. Consumers register as dependent upon a producer. By having the consumer registers as a dependent of a producer, that allows the Operations event handler to perform work when the producer updates a Buffer. The Operations event handler may do something as simple as mark its execute method as "ready" so that the EventThread will call it. Or it may actually perform some work and then decide what to do.
        - BufferManagerPointer – This object is used by producers and consumers to access Buffers in the BufferManager. The BufferManagerPointer is returned when the producer or consumer registers with the BufferManager.
    - unregister – This removes an Operation from the BufferManager.
    - offer – This is how a producer updates the BufferManager and will trigger the calls to the event handlers that are registered as dependent upon the producer.
    - poll – Provides a consumer access to a Buffer and updates it's BufferManager pointer. It returns null if the consumer's BufferManagerPointer is the same position as the producer's BufferManagerPointer that it depends upon
    - peek – Provides a consumer access to a Buffer without updating its BufferManagerPointer. It returns null if the consumer's BufferManagerPointer is the same position as the producer's BufferManagerPointer that it depends upon.
    - bookmark – Add a placeholder in the BufferManager that can be used to distinguish operations on different chunks. This allows a new operation that is a consumer to pick a particular location within the buffer manager to begin pulling data from.
- **Buffers** – A generic term for an object that has a ByteBuffer backing it. This is used to handle the clients request information and data coming in from the wire.
- **Operations** – An Operation implements a common interface and provides the building blocks to perform requests. The methods within the Operations perform work on the Buffers or results generated from the buffers.

- o **Operation Methods** – The operation is defined as an interface, which may not actually be the best choice, but it does allow for some commonality in the implementation. Operations provide the following methods, though not all methods are required (meaning certain methods may not perform any work).
    - initialize – This sets up all preconditions for the Operation to run.
    - event – This is the method that executes when some other pre-condition is met that allows the Operation to run.
    - execute – This is the method that performs the actual work for an Operation. This work could be run on either the EventThread or a WorkerThread (in the event of a long running operation that is either CPU intensive or requires off-box resources). The execute method is placed in the "ready" state by the event handler. The amount of work done within the execute method depends upon the operation. Some are quite simple and others (such as WriteChunkToClient) implement a full state machine within the execute method.
    - complete – This is the last step when the execute method determines that there is no more work for this Operation to perform. This is used to tear down dependencies on buffer managers and to call completion functions.
    - There are a number of other methods related to queuing the Operation onto the work queue for the RequestContext. These can be changed to something more efficient than actually adding them to a queue (with its associated locking requirements), but it makes the code easier to debug for the short term. The methods related to queuing are:
        - isOnWorkQueue
        - isOnTimedWaitQueue
        - markAddedToQueue
        - markRemovedFromQueue
        - hasWaitTimeElapsed – For use with the timed wait queue

- o Example Operations include:
    - Parse HTTP Request – This processes Buffers and produces HTTP headers and URI information. It completes when all the headers have been read in. It produces information from the headers and the URI that is stored in an HTTP Information object that is associated with the RequestContext. It will also determine the operation type (i.e. PUT, GET, etc.).
    - MD5 Digest – This will take in Buffers and compute the Md5 digest from them. When it processes all the buffers (for a PUT operation this is when the entire content length amount of data has been processed) it will send an event to the Compare MD5 operation.
    - Authenticate – This will use the HTTP Information to authenticate the request.

- Encrypt – This encrypt the Buffers read in and encrypt them and place the result in a new Buffer.
- Write to Storage Server – This will take Buffers generated from the Encrypt operation and write them out the Storage Server in chunk sizes.

Thread Overview

There are the following threads in the initial design:

- **Accept Thread** – This is a thread that loops on a ServerSocketChannel and a Selector waiting for the socket to be acceptable. When the socket is acceptable, it will pull out the client SocketChannel and pass that to the EventThread.
- **Event Threads** – This is a collection of threads that handles the NIO processing for one or more SocketChannel(s) and the work that is required to process the data read into the buffers. There is one loop that handles the Selector with a set of SocketChannel(s) registered. There is a second loop that executes all of the "ready" Operations for a particular RequestContext. The second loop works through all of the RequestContext assigned to the EventThread. The Operations are set to "ready" to execute based upon "events" that take place as a result of data arriving or operations on the data completing. A decision needs to be made to assign RequestContext to an EventThread or allow it to float between different EventThreads. This will have implications on the locking model for various objects.
- **Worker Threads** – This is a collection of threads used to perform CPU intensive work (MD5 digest and encryption are examples of this) or work that accesses off-box resources (Authentication or Chunk picker are examples what these threads could be used for).

High Level Operation Flow

HTTP Parsing

When there is an accept() processed for a new connection on the ServerSocketChannel, the initial steps are as follows:

1. Call registerClientSocket() for the NioEventPollBalancer. This will pick the thread the operation will be handled on. Once it picks a thread to execute on, it then calls registerClientSocket() for the NioEventPollThread.
2. registerClientSocket() for the NioEventPollThread allocates the RequestContext and an IoInterface.
3. Allocate a RequestContext and call it's initializeServer() method. The initializeServer() method will do the following:
   a. Starting with multiple clean BufferManager(s). There are three BufferManager(s) that the RequestContext deals with:
      i. clientReadBufferManager – Used for reading data from the server side SocketChannel that the client has written.
      ii. clientWriteBufferManager – Used to write data to the client. This includes the HTTP Response.

       iii. <u>storageServerWriteBufferManager</u> – Used to accumulate data (generally after it has been encrypted) to write to the Storage Server(s).
  b. With each BufferManager, there is a fairly standard pattern that is used. For the HTTP Parser the following is setup.
       i. Register the Buffer Allocate producer – For the clientReadBufferManager, this is what adds ByteBuffer(s) to the BufferManager (though the ByteBuffer(s) could all be allocated at startup time) and then is responsible for doling out buffers for the read operation.
       ii. Register the Read consumer that is dependent upon the Allocate producer. This lets the NIO read code know that there are buffers available to read data into.
       iii. Register the Read Complete producer – This is how the BufferManager knows when there is valid data in the Buffer.
       iv. Register the Parse Http Request consumer that is dependent upon the Read Complete producer. When the Read Complete producer updates its pointer, that generates an event for the Parse Http Request that tells it that it has data and can run. The Parse Http Request operation will continue to pull data from available buffers until the entire HTTP Request has been parsed.
  c. Allocate at least one buffer and add it to the BufferManager. This is the Buffer Allocate producer.
  d. When there is a Buffer added, an "event" will be sent to the Read consumer saying it has a Buffer available to perform work on. For the Read consumer the "event", will be a method that sets the OP_READ flag in the Selector for the SocketChannel that the request's data will be transmitted across.

4. At this point, the RequestContext has been started and the request is waiting for data to be read into the Buffer.
5. Once data has been read into the Buffer (SSL will first read data into a temporary buffer and then unwrap it into the Buffer owned by the BufferManager), this will trigger the "data ready" event. The only listener for that event at this point is the Parse Http Request. This will add the Parse Http Request execute() method to the EventThread (this is setting the Operation to "ready").
6. The EventThread will then process any NIO tasks (i.e. read more data into Buffers). Once the NIO tasks are completed, the EventThread will then execute the "ready" Operations.
7. In this case, there is a single "ready" Operation, Parse Http Request. The Parse Http Request will feed the Buffer(s) through the HTTP parser and add the results to the CasperHttpInfo object. If the entire request has not been parsed, it will add one or more Buffers to the BufferManager, which will kick off the read path again. At the point there is no more data to feed through the HTTP parser and it is waiting for additional data, the Parse Http Request is in a "not ready" state. If the entire request has been parsed, it will then determine the request type and setup the request handler. In the design here, it will only deal with the PUT request.
8. Remove the Parse Http Request and Read registration from the BufferManager.

The following is how a simple dot notation to describe the HTTP parsing sequence is for a single request handler. This does not account for the fact that multiple requests will be handled at the same time and will require time to be allocated to each:

```
while (!httpParseDone) {
    Allocate.Read.ParseHttpRequest.DetermineRequestType
}
```

PUT Handler

Once the parsing is completed and the HTTP method determination is completed, the PUT method will operate something like the following:

1. Perform the initialize() method on the V2 PUT Handler. This will perform the following:
   a. The BufferManager will remain the same as for the Parse Http Buffer handler as there may be partial or complete leftover buffers from the HTTP parsing. This just means that the HTTP Request did not use an entire buffer and there may be data for the client object in the buffer.
   b. The Authentication Operation is set to the "ready" state (that assumes that this is the first operation to run following the parsing the HTTP request, but in reality, this may not be true).
   c. At this point, there are no consumer Operations registered with the BufferManager, so there is no work for the NIO tasks. The Allocation and Read Complete Producer Operations are still registered with the BufferManager.
2. The EventThread executes the Authentication Operation. As an aside, this could be pushed off to a different worker thread, but for the design it is not a critical item.
3. When the Authentication Operation completes (for the purposes of this, assume it is successful), this allows the next steps in the PutObject operation to run. The following things will take place:
   a. Potentially add more Buffers to the BufferManager allowing more reads to take place. It is a design decision that has been left till later to determine if the ring buffer should be fully populated at initialization time or as an on-demand operation.
   b. Register the Read Operation as dependent upon the Allocation Operation.
   c. Register the MD5 and Encryption Operations as dependent upon the Read Complete. Note: Currently, only the Encrypt operation is implemented and that just copies data from one set of buffers to another set of buffers (from the Client Read Buffer Manager to the Storage Server Write Buffer Manager).
   d. Register the Encrypt Buffer operation as a Producer in the "Storage Server write" BufferManager.
   e. Register the Write to Storage Server Operations as dependent upon the Encrypt Buffer producer.
   f. Note – There are all sorts Operations that are not being considered in this to keep the overall description simpler and easier to understand.

4. Now the various operations can be triggered and work to process the content for the PutObject operation can begin.

The dot notation for the above operations would look something like the following (not considering the breakup of the chunk to the various Storage Servers). This does not account for the fact that multiple method handlers (GET, POST, PUT, DELETE, etc.) can be running at the same time:

```
while (!allStorageServerWritesDone) {
    Authenticate.Allocate.Read.MD5.Encrypt.StartNewRequest.
        DetermineChunk.InitialWriteToDatabase.ChunkMd5Compute.
        WriteToStorageServer.VerifyChunkMd5.FinalWriteToDatabase.SendFinalStatus
}
```

The design is such that the write of the chunk to the Storage Server is actually multiple steps and there can be multiple of those operations running in parallel and at different speeds. The steps to write to a storage server include:

- Determine the chunk information for which storage servers to write the chunk worth of data. This is a request to the Chunk Manager Service that manages all chunk allocations.
- Writing the chunk begin information to the *objectStorageDb.storageChunk* database table.
- Setting up a connection to the Storage Server (the Object Server acts as an initiator). There will be more than one Storage Server that will receive the chunk data.
- Sending an initial HTTP Header to the Storage Server.
- Sending the chunks worth of client object data to the storage server.
- Receiving the HTTP Response from the Storage Server and verifying that the data was received properly.
- Writing the chunk completed information to the *objectStorageDb.storageChunk* database, table to indicate the data is safely written.

Several other things to consider, an Operation can generate an "event" the will cause another Operation to execute. One example of this is the MD5 Digest that will trigger the MD5 Compare when all of the content data has been run through the digest algorithm.

NIO Socket Handling

The NIO socket handling layer is designed to be independent of the handling of the HTTP Request logic. The design is such that the NIO socket layer can be replaced by a different layer that obtains data from a file or just a filled in buffer. This allows the HTTP Request logic and the various operations to be easily tested without worrying about the socket handling.

The NIO Socket layer is just one implementation of the interfaces provided by the IoInterface class. The IoInterface provides the following methods:

- startClient – This is when the IoInterface is being used as a target and the information used to communicate back to the initiator is being registered.
- registerClientErrorHandler – This is how a user of the IoInterface is informed that an error has occurred that needs to be handled. For the NioSocket implementation, this could be an error that the SocketChannel was disconnected.
- startInitiator – This is when an initiator wants to open a connection to a remote device. For the NioSocket implementation this is used to communicate with the Storage Server(s).
- registerReadBufferManager – The IoInterface is designed to use the BufferManager and a BufferManagerPointer to know where ByteBuffer(s) are that are waiting to have data placed into them.
- registerWriteBufferManager – The IoInterface uses the BufferManager and a BufferManagerPointer to know where to obtain ByteBuffer(s) that have data that is to be written out.
- unregisterReadBufferManager – This removes the read BufferManager and BufferManagerPointer references from IoInterface.
- unregisterWriteBufferManager – This removes the write BufferManager and BufferManagerPointer references from the IoInterface.
- readBufferAvailable – This tells the IoInterface that a read ByteBuffer is waiting to have data placed in it. For the NioSocket implementation, this will set the OP_READ flag for the NIO Selector.
- performRead – This is what actually places the data into the ByteBuffer. For the NioSocket, it is where the SocketChannel.read() actually takes place when the Selector says there is data waiting. For the NioSocket implementation this is called from the NioSelectHandler class.
- writeBufferReady – This tells the IoInterface that a write ByteBuffer is waiting to have data written out of it. For the NioSocket implementation, this will set the OP_WRITE flag for the NIO Selector.
- performWrite – This is what actually performs the write from the ByteBuffer. For the NioSocket, it is where the SocketChannel.write() actually takes place when the Selector says there are available socket write buffers to hold data. For the NioSocket implementation this is called from the NioSelectHandler class.
- sendErrorEvent – This is what calls the Operation that was registered with the registerClientErrorHandler() method when an error occurs. For the NioSocket implementation this is called from the NioSelectHandler class.
- closeConnection – This is called when the connection is to be closed out and unregistered from the Select handler. For the NioSocket implementation it will also close out the SocketChannel.
- connectComplete – This is used when setting up an initiator connection and is called when the connection has been completed to the target and is ready for use.

Operation Details

The RequestContext contains the overall state for the HTTP Request. It owns the following BufferManager(s) to perform work:

- clientReadBufferManager – Used for data coming in from the client. This includes the HTTP Request and the client object data.
- clientWriteBufferManager – For the initial implementation that only handle the PutObject, this is used to write the HTTP Response back to the client.
- storageServerWriteBufferManager – This holds the client object data (in an encrypted form) that is to be written to the Storage Server(s).

The following are a list of operations that take place to complete a PutObject request. All of the error handling is not spelled out and is mostly missing. The idea is to show the dependencies between the various operations and how they could be chained together.

Initial Request Parsing Setup
- Requires
    - Nothing
- Produces
    - Initializes the HTTP Parsing
- Events Generated
    - Generates an event to **Allocate Buffer** (HTTP Parsing) to setup a buffer to read in the HTTP Request

Buffer Read Metering (for HTTP Parsing) – This can simply allow buffers to be used as part of metering if all the buffers are allocated up front.
- Requires
    - How many buffers to allocate
- Produces
    - Invalid ByteBuffer(s) in the BufferManager. This means they do not have valid data.
    - *bufferMeteringPointer* registered with **clientReadBufferManager**
- Events Generated
    - When a buffer is added, generate a **Read into Buffer** (HTTP Parsing) event

Read into Buffer (for HTTP parsing)
- Requires
    - Empty Buffers
- Produces
    - Buffers with HTTP request data
    - *readBufferPointer* registered with the **clientReadBufferManager** with a dependency upon the *bufferMeteringPointer*
- Events Generated
    - When a buffer read completes, generate a **Parse HTTP Buffer** event

Parse Http Buffer
- Requires
    - ByteBuffer
- Produces
    - URI
    - HTTP Headers
    - Boolean - Has the entire request been parsed
    - *httpBufferPointer* registered with the **clientReadBufferManager** with a dependency upon the *readBufferPointer*.
- Events Generated
    - If all the HTTP Request has not been parsed, generate a **Buffer Read Metering** event to add another ByteBuffer to allow the Read into Buffer operation to run again an bring in more of the HTTP Request.
    - If all the HTTP Request has been parsed, generate a **Determine Request Type** event.

Determine Request Type
- Requires
    - HTTP Headers
    - URI
- Produces
    - What type of request handler is required
- Event Generated
    - Generate an event to the **Setup V2 PUT Handler** – Simplified for this document to only handling the V2 PUT request.

SetupObjectPut Handler – This is what sets up the encryption of the client object data and the Md5 digest operations. It completes and events the Send Final Status operation when all of the client object data has been written to the Storage Server(s) and the Md5 Digest has completed.
- Requires
    - HTTP Headers
    - URI
- Produces
    - Integer – Content Length
    - Sets up for the handling of the PUT Object request
        - Currently this is only the Encrypt Buffer operation
        - Will need to be expanded to computing the Md5 digest for the client object data.
- Event Generated
    - When the Encrypt Buffer and Compute Md5 Digest operations have completed, this will event the Send Final Status operation and will clean itself up.

Authenticate Request

- Requires
  - Access Token - This is the unique identifier that is provided with a request to allow a user to access resources within a Tenancy.
  - URI - To provide the request method that needs to be checked against the users permissions.
- Produces
  - int - The unique ID of the Tenancy for the validated Access Token
- Events Generated
  - If the authentication was successful, generate an event to **Determine Request** to start the methods operations.
  - If the authentication failed, generate an event to **Send Completion Status to Client** indicating the client request failed authentication

Buffer Read Metering (for client PUT object content read) – This may be given the entire number of buffers to read in the client object and this deals with the allocation limits and potential out-of-buffer issues. This is the same Buffer Read Metering used for the HTTP Request handling.

- Requires
  - How many buffers to allocate
- Produces
  - Invalid Buffers in the BufferManager
  - *bufferMeteringPointer* registered with **clientReadBufferManager**

Events Generated
  - When a buffer is added, generate a **Read into Buffer** (client object) event

Read into Buffer (for client PUT object content read) – This is the same Read into Buffer used for the HTTP Request reads

- Requires
  - Empty Buffers
- Produces
  - Buffers with client object data
  - *readBufferPointer* registered with the **clientReadBufferManager** with a dependency upon the *bufferMeteringPointer*

Events Generated
  - When a buffer read completes, generate an **Md5 Digest** event
  - When a buffer read completes, generate an **Encrypt** event

Md5Digest – This can be setup to run on a different thread during registration and then the BufferManager could be made to be thread safe and then access to the buffer can take place from wherever this is running. Also, need to make sure that one connection cannot consume or

queue up lots of Md5 Digest buffers to the CPU threads and introduce latency for other connections.

- Requires
  - Content Length
  - Buffer with data
- Produces
  - Md5 Digest (will be a partial digest until all of the bytes in the client object have been processed)
  - boolean – Has the entire digest been produced
  - boolean – Was the computed Md5 Digest identical to the one passed in through the Content-MD5 header.
  - *md5DigestPointer* registered with the **clientReadBufferManage**r with a dependency upon the *readBufferPointer*
- Events Generated
  - If the entire client object digest has been produced, generate a **Setup V2 Put Handler** event() to indicate it has completed.

Encrypt Buffer – This operation handles encrypting the client object data and placing the encrypted data into the BufferManager used to write the data to the Storage Server(s).
- Requires
  - Buffer with data
- Produces
  - New buffer with encrypted data
  - *encryptInputPointer* registered with the **clientReadBufferManager** and a dependency upon the *readBufferPointer*.
  - *storageServerAddPointer* registered with the **storageServerWriteBufferManager**
  - *storageServerWritePointer* registered with the **storageServerWriteBufferManager** with a dependency upon the *storageServerAddPointer*.
- Events Generated
  - If this is the first encrypted buffer for a chunk, generate a **Setup Chunk Write** event. This event will be generated when an encrypted buffer is generated that will start a new chunk.
  - Generate an event to **Compute Shaw-256** to indicate there is a Buffer to process
  - Generate an event to **Write to Storage Server** to indicate there is a Buffer to write

Object Chunk Allocator
- Requires
  - HTTP Headers
  - URI Information
  - Chunk Number
- Produces

- o List of Storage Servers and the chunks where object data will be saved.
- Events Generated
  - o Generates an event to **Write Chunk Information to Database** that the meta-data write can proceed.

Setup Chunk Write – This sets up the write of a chunk worth of data to a Storage Server. There can be multiple chunks being written to multiple Storage Servers at the same time. So, once the information about where the Storage Servers are located (IP address and Port number) there will be a Setup Chunk Write operation run for each Storage Server. This is passed in the location information for the Storage Server the data will be written to.
- Requires
  - o **storageServerBufferManager** – This is used to handle writing the HTTP Request and the Md5 information to the Storage Server. It is a BufferManager that is unique to the Storage Server chunk write instance.
  - o **storageServerResponseBufferManager** – This is used to handle receiving the HTTP Response from the Storage Server. It is a BufferManager that is unique to the Storage Server chunk write instance.
- Produces
  - o *addBufferPointer* registered with **storageServerBufferManager**
  - o *respBufferPointer* registered with **storageServerResponseBufferManager**
  - o This starts the connection to the Storage Server through a call to startInitiator(). The Connect Complete operation will be evented when the connection is completed.
- Events Generated

Connect Complete – This is run when the connection to the remote Storage Server is completed.
- Requires
- Produces
- Events Generated
  - o Will event() the **Build Header to Storage Server** operation when this operations execute() method runs.

Handle Storage Server Error – This is run when the connection to the remote Storage Server is broken or some sort of error occurs on the connection such that the Storage Server cannot be communicated with.
- Requires
- Produces
- Events Generated

Write Chunk Information to Database
- Requires
  - o HTTP Headers

- o URI Information
- o Chunk Number
- Events Generated
  - o Generates an event to **Write to Storage Server** that the writes can proceed for a particular chunk
  - o Generate an event to **Write to Storage Server**

Compute Md5
- Requires
  - o Encrypted Buffer
- Produces
  - o Md5 value
  - o Boolean - Has entire chunks Md5 been produced
- Events Generated
  - o None

Build Header to Storage Server - This operation is used to build the HTTP Request to send to the Storage Server prior to sending the actual chunk data.
- Requires
  - o An empty ByteBuffer to build the HTTP Request in
- Produces
  - o A built HTTP Request to a particular Storage Server
  - o *writePointer* registered with the **storageServerBufferManager**.
- Events Generated
  - o When the *writePointer* is updated, this will cause the Write Header to Storage Server operation to be evented.

Write Header to Storage Server – This operation is responsible for sending the HTTP Request that was built by the Build Header to Storage Server operation. The linked dependency between the *writeInfoPointer* and the *writeDonePointer* is so this operation can be evented when the data has been placed on the wire by the SocketChannel.write().
- Requires
  - o Buffer filled in with the HTTP Request information
- Produces
  - o *writeInfoPointer* registered with the **storageServerBufferManager** with a dependency on *writePointer*.
  - o *writeDonePointer* registered with the **storageServerBufferManager** with a dependency on *writeInfoPointer*.

Write to Storage Server – This operation runs when the *encryptedBufferPointer* (actually the *encryptInputPointer* from the Encrypt Buffer operation) is updated, meaning encrypted data has been placed into the ByteBuffer and it can now be written out.
- Requires

- o   Storage Server Information
- o   That meta-data information has been written to the database
- o   That the HTTP Request has been successfully sent to the Storage Server.
- o   Encrypted Buffer
- o   The amount of data to write for the chunk to the Storage Server
- Produces
  - o   Boolean – Has entire chunk been written
  - o   ***writeToStorageServerPtr*** registered with **storageServerWriteBufferManger** (owned by the RequestContext) with a dependency on the ***encryptedBufferPointer***.
  - o   ***writeDonePointer*** registered with the **storageServerWriteBufferManager** with a dependency on ***writeToStorageServerPtr***.
- Events Generated
  - o

Compare Md5 to Storage Server Md5
- Requires
  - o   Storage Server Information
  - o   Entire chunk has been written to Storage Server (Event from **Write to Storage Server**) and response from the Storage Server has been received. The response from the Storage Server contains its computed Md5 value for the chunk.
  - o   Final Md5 value (Event/product from **Compute Md5**)
- Produces
  - o   Boolean – All chunks written to Storage Server
- Events Generated
  - o   Generates an event to **Update Meta-Data in Database** when all the writes to a chunk have completed

Update Meta-Data in Database – This is when the data and Md5 comparisons have completed to a Storage Server
- Requires
  - o   Chunk write completed
  - o   Number of chunks required for the client PUT object
- Produces
- Events Generated
  - o   Send an event to **Send Completion Status to Client** when all chunks have been written to the Storage Servers for the client object.

Read Response Buffer from Storage Server – This is the operation that runs to cause the HTTP Response data to be read in from the Storage Server.
- Requires
  - o   Empty ByteBuffer(s) which are allocated as part of the Setup Chunk Write operation.

- Produces
  - *readBufferPointer* registered with **storageServerResponseBufferManager** and has a dependency on *meterBufferPtr* (which is the *respBufferPointer* created in the Setup Chunk Write operation).
  - Registers the **storageServerResponseBufferManager** and the *readBufferPointer* with the NioSocket connection used to communicate with the Storage Server.
- Events Generated
  - Informs the NioSocket connection that there is an available read ByteBuffer that is expecting data.

Storage Server HTTP Response Handler – This operation handles checking that the HTTP Response returned by the Storage Server is valid and if the response was good or contained an error.
- Requires
  - At least one ByteBuffer filled in with the HTTP Response from the Storage Server
  - The Jetty HTTP Parser to perform the actual parsing of the buffer contents.
- Produces
  - *httpResponseBufferPointer* registered with **storageServerResponseBufferManger** with a dependency on *readBufferPointer*.
  - StorageServerResponseCallback that is called when the parsing of the HTTP Response completes. This is how the completion and cleanup of the Storage Server chunk write is triggered.
  - The status for the Storage Server chunk write operation.
- Events Generated
  - The StorageServerResponseCallback is triggered from the HttpResponseListener when the messageCompleted() callback is invoked by the Jetty Parser.

Validate Md5 Digest
- Requires
  - The calculated Md5 digest
- Produces
  - An error if the Md5 digest does not match the expected digest
- Events Generated
  - Generates an event to **Send Completion Status to Client**

Send Completion Status to Client
- Requires
  - Final Status – Meaning all the chunks have been written and meta-data all updated or an error occurred.
  - That the final Md5 digest compare was done or an error occurred
- Produces
  - Closes out the connection after the status has been sent and cleans up

- Events Generated
  - No events, just put the RequestContext back on the free listOperation Event Details for HTTP Parse

Event Flow for HTTP Parse

```
┌─────────────────┐
│ Initial Request │
│ Parsing Setup   │
└─────────────────┘
         │
      Event to
      Allocate
       Buffer
         ▼
┌─────────────────┐  ◄──────────────────┐
│ Allocate Buffer │                     │
└─────────────────┘                     │
         │                              │
       Buffer                           │
    available to                        │
   read data into                       │
         ▼                              │
┌─────────────────┐                     │
│ Read into Buffer│                     │
└─────────────────┘                     │
         │                              │
     Data Read              HTTP        │
     into Buffer           Request      │
         │              Parsing  has    │
         │              not finished    │
         ▼                              │
┌─────────────────┐ ────────────────────┘
│ Parse HTTP      │
│ Buffer          │
└─────────────────┘
         │
       HTTP
      Request
      Parsing
      finished
         ▼
┌─────────────────┐
│ Determine       │
│ Request Type    │
└─────────────────┘
```
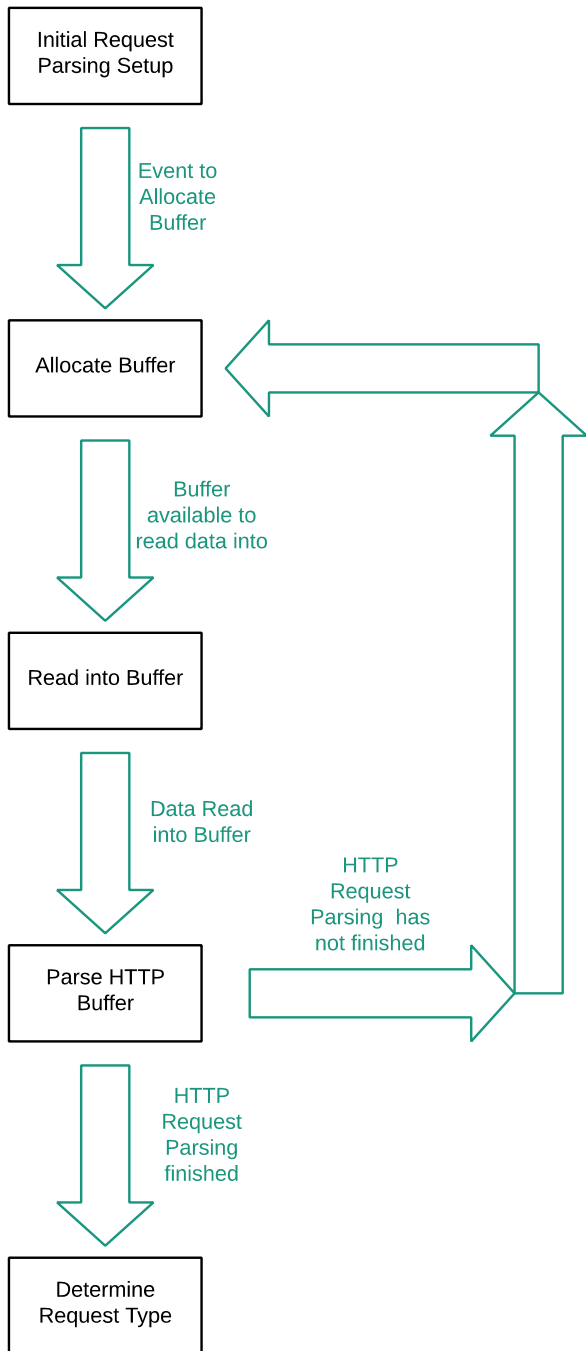
Event Flow for PutObject Request

Buffer Manager Design

The Buffer Manager provides an abstraction that allows operations to register to be told when information within the buffer changes. The idea is that there is at least one producer of data (meaning something that transfer data into a ByteBuffer or allows buffers to be used) and at least one consumer of the data. For example, a producer of data could be a SocketChannel read and the consumer of the data could be the HTTP Parser. The HTTP Parser only has work to perform when the SocketChannel read has completed and updated the byte position values for the ByteBuffer. The BufferManager is designed to allow clients (producers and consumers) to register and to setup dependencies between the different producers and consumers. When the dependency is met, then the consumer is then sent an event() to inform it that there is work that can be done. This allows the design of the operations to be event driven (a reactive design where operations are told to run when data is available) instead of a polling design (where the operations query if data is available and then perform work).

The other advantage of having the ring buffer and its associated pointers contained within the BufferManager and BufferManagerPointer objects is the different behaviors (i.e. wrap around conditions, out of filled buffers and such) can easily be tested without requiring much additional code infrastructure. Once the dependencies and how they interact are debugged, the clients simply use the methods and the code should work. It prevents having to implement the checking logic in the different Operations.
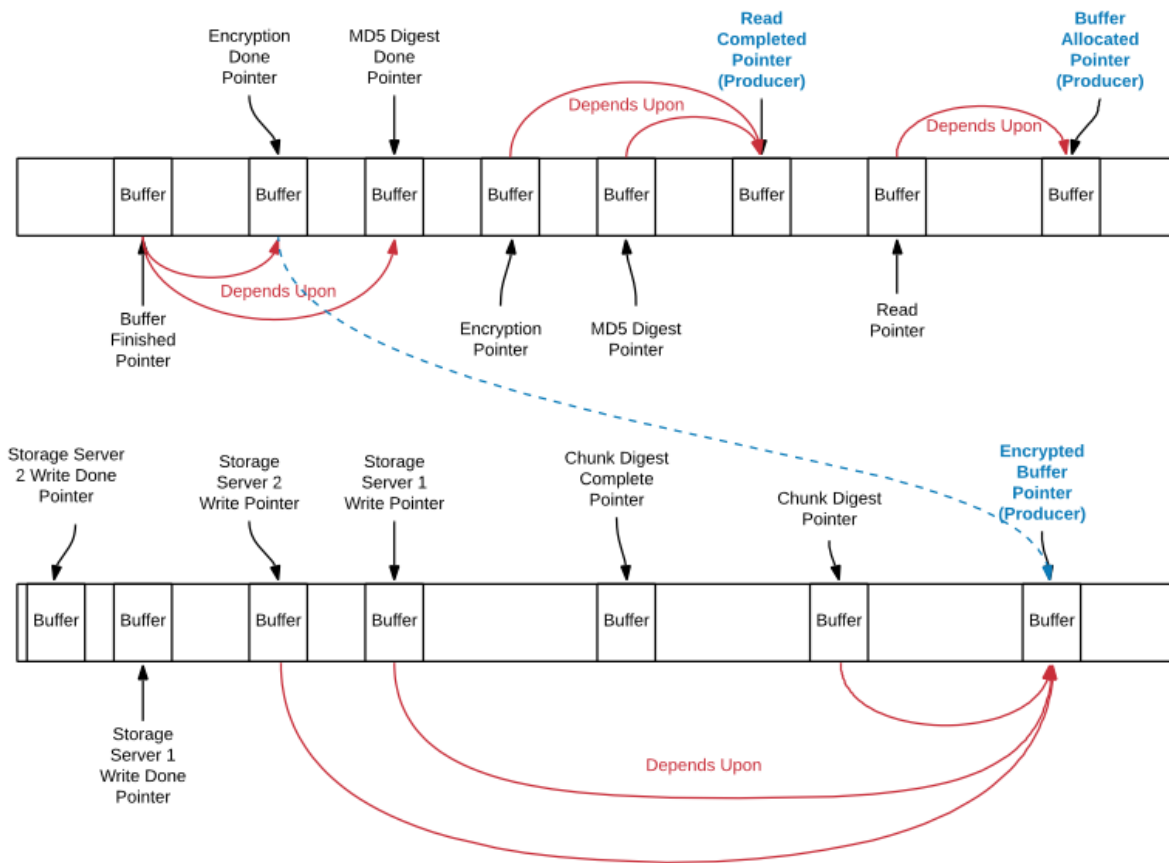
The two linear sequences of operations above represent a view into a two different circular buffer. The top circular buffer represents data coming off the wire from the client performing the PUT operation. The lower circular buffer represents data that was encrypted (encryption can be an an encrypt and put the result in a different buffer) and then needs to be operated on in a chunk to be sent to the Storage Servers. The "Depends Upon" represents a dependency upon a prior operation on the buffer completing prior to the next operation taking place. For example, when a "Read Completed" event takes place, that can be used to send events to the "MD5 Digest" and "Encryption" methods indicating they are ready to run. There are several features of the Ring Buffer to consider:
 - Anything that depends on something else, cannot "pass" the operation it depends on in the Ring Buffer.
 - If items do not have a dependency upon each other, they are free to operate at different rates. For example, "Storage Server 1 Write" and "Storage Server 2 Write" can both write to their targets at different rates.

Creation of JAR Files for the Services

To create the Docker images for use in the Kubernetes POD, the first requirement is to create a JAR file for each of the services. The following gives an example of how to create the JAR file from within the IntelliJ framework (the steps are pretty much the same with the module, directories and main class changing between services). Since this project is rather small, the JAR files are created manually for the time being, but a useful improvement at some point would be to automate their generation as code changes.

This describes the steps used to build the clienttest module JAR file from within the IntelliJ IDE. From the "File" -> "Project Structure" window, click on "Artifacts". The click on the "+" and select "JAR" -> "From modules with dependencies...". This will bring up a dialog box that needs to be filled in.

Module: clienttest
Main Class: com.webutils.webserver.manual.TestMain

Select "copy to the output directory and link via manifest"

Directory for META-INF/MNIFEST.MF:
  /Users/notterness/WebServer/webserver/clientest/src

NOTE: If the directory for the META-INF is not set to the above (meaning ./clienttest/src or something similar for the other JAR files), it will cause an error when trying to run the JAR. Error below.

```
Noels-MacBook-Pro:objectserver_jar notterness$ java -jar
objectserver.jar 5001 kubernetes
Error: Invalid or corrupt jarfile objectserver.jar
```

Then hit "OK" to move to the next step. This brings up another dialog box with a few more items to fill in.

Name: clienttest:jar

Type: other

Output directory: /Users/notterness/WebServer/webserver/clienttest/src/out/clienttest_jar

Then hit "OK" to move to finish. To build the actual JAR file, go to "Build" -> "Build Artifacts" and then use the popup dialog to select which Artifact to build.

To validate the JAR file runs.

```
/Users/notterness/WebServer/webserver/clienttest/src/out/
clienttest_jar

java -jar clienttest.jar 5010
```


Building the Docker Images for the Services

To build the artifacts that will be pulled in for the Kubernetes Pod, each JAR file needs to be built into a Docker image. For each service (ChunkMgr, ClientTest, ObjectServer and StorageServer) there is a *Dockerfile* used to create the Docker image located under ./{*service directory*}/src. For this project the *service directory* can be chunkmgr, clienttest, objectserver or storageserver.

From the directory where the Object Server code is located (for my system it is located in the directory below, but it is really relative to where the git repository is pulled to):

```
Noels-MacBook-Pro:src notterness$ pwd
/Users/notterness/WebServerNew/ObjectServer/objectserver/src
```

To build the image for the Object Server, the following command line is used.

```
docker build -f Dockerfile -t objectserver:1 .
```

For the Kubernetes Pod, it currently is setup to use version 1 (i.e. objectserver:1) of the Docker image that is created. The Kubernetes Pod currently requires the following Docker images to be built:

- chunkmgr:1
- clienttest:1 - Not actually required for the Pod, but used to test the services running within the Pod.
- objectserver:1
- storageserver:1
- authorizationmgr:1 - Not currently implemented, but is needed to properly handle the user validation and creating/deletion of Tenancies.

Building the Docker Images for the fluent-bit Log Pushers

There is a fluent-bit Docker image per service. This is done to allow a different *.conf file to be used. Within the *.conf file there are three parameters, *Index*, *Include_Tag_Key* and *Tag_Key* that are unique to a service. These provide additional information to the Elasticsearch engine where the logs are pushed and make it easier to differentiate where the logs originated.
The four log pushers that are currently build are:

- authorization-mgr-logger - Not currently used
- chunk-mgr-logger
- object-server-logger
- storage-server-logger

To build the various loggers, the steps are as follows (see the README in the directories for exact details):

```
Noels-MacBook-Pro:chunk-mgr-logger notterness$ pwd
/Users/notterness/WebServerNew/ObjectServer/kubernetes/fluent-bit/
chunk-mgr-logger

docker build -f Dockerfile -t fluent-chunk-logger:1 .
```

The above will build the fluent-chunk-logger Docker image. The steps are pretty much the same for the other loggers, just in different directories and with a different Docker image output.

Detail about the fluent-bit .conf Files

This provides a bit more information about how the *.conf files are setup for the fluent-bit Docker images.

Using the following storage-server-logger.conf file:

```
[SERVICE]
    Flush         5
    Daemon        Off
    Log_Level     debug
    Parsers_File /fluent-bit/etc/parser.conf

[INPUT]
    Name    tail
    Path    /var/log/storageserver.*

[OUTPUT]
    Name  es
    Match *
    Host  host.docker.internal
    Port  9200
    Index storageserver
    Type docker
    Include_Tag_Key On
    Tag_Key Storage-Server
```

The _index will be "storageserver" and to search for all the entries in Elasticsearch, the following URL is used:

```
$ curl -X GET 'http://localhost:9200/storageserver/_search'
```

To delete all the records from Elasticsearch for the log entries tagged with "storageserver":

```
$ curl -X DELETE 'http://localhost:9200/storageserver/'
```

Some Details About the Kubernetes .yaml File

Once the Docker images are all built, the next step is building the Kubernetes deployment. The deployment-webutils.yaml file is located at (again this will be different depending upon where the git pull/clone of the repository takes place):

```
Noels-MacBook-Pro:kubernetes notterness$ pwd
/Users/notterness/WebServerNew/ObjectServer/kubernetes
```

The yaml file contains the following (this is not the complete yaml file and is being used for reference):

```
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: webutils-site
  spec:
    selector:
      matchLabels:
```

```yaml
      app: webutils-site
      tier: backend
      track: stable
replicas: 1
template:
  metadata:
    labels:
      app: webutils-site
      tier: backend
      track: stable
  spec:
    containers:
      -
        name: object-server
        image: objectserver:1
        imagePullPolicy: Never
        ports:
          -
            containerPort: 5001
        volumeMounts:
          -
            name: object-log-volume
            mountPath: /usr/src/myapp/logs
          -
            name: kube-config-volume
            mountPath: /usr/src/myapp/config

      -
        name: fluent-object-logger
        image: fluent-object-logger:1
        imagePullPolicy: Never
        volumeMounts:
          -
            name: object-log-volume
            mountPath: /var/log
      -
        name: socat
        image: socat:1
        imagePullPolicy: Never
        ports:
          -
            containerPort: 1234
        volumeMounts:
          -
            name: docker-sock
            mountPath: /var/run/docker.sock
    volumes:
      -
        name: object-log-volume
        emptyDir: {}
      -
        name: kube-config-volume
        hostPath:
          path: /Users/notterness/.kube
```

The critical pieces are the shared mount point between the object-server and fluent-object-logger containers. This is the object-log-volume volume.

The object-server container will write its logs to */usr/src/myapp/logs* within the Docker container.

The fluent-object-logger container will read (tail) the logs at */var/log*. This is associated with the object-server volume through the *name* fields in the yaml file

In the "volumes:" section of the yaml file, the object-log-volume volume is defined as "emptyDir: {}".
The "emptyDir: {}" indicates that the connection between the two containers only exists as long as the deployment is running.

The other piece to look at is the kube-config-volume as that is needed to access the Kubernetes API from within the object-server Docker image. The socat Docker image is required to allow access from the object-server to the outside world for the Kubernetes API.

Socat as a Docker Image

The Dockerfile (located at ./ObjectServer/kubernetes/socat) is:

```
FROM alpine
RUN apk --no-cache add socat
VOLUME /var/run/docker.sock
ENTRYPOINT ["socat"]
ENTRYPOINT ["socat", "TCP-LISTEN:1234,reuseaddr,fork","UNIX-CLIENT:/
var/run/docker.sock"]
EXPOSE 1234
```

To build the docker image:

```
docker build -f Dockerfile -t socat:1 .
```

To run just the docker image:

```
docker run -v /var/run/docker.sock:/var/run/docker.sock -d --
restart=always --name docker-sock-proxy -it socat:1
```

—> The following provides the information by going directly to the Docker API endpoint

```
$ curl -XGET --unix-socket /var/run/docker.sock http://localhost/
containers/json
```

```
[{"Id":"d4e31b2e87ebde9b2fc6ec78a7a2c70c027338c8d91644f1b77910ede34a52
b6","Names":["/docker-sock-
proxy"],"Image":"socat:1","ImageID":"sha256:b616d992c8ed6b43eb57d8eaf5
559d37e03b068adcc0f11f1bdb1fbebae0b0a5","Command":"socat TCP-
LISTEN:1234,reuseaddr,fork UNIX-CLIENT:/var/run/
docker.sock","Created":1583273362,"Ports":
```

```
[{"PrivatePort":1234,"Type":"tcp"}],"Labels":
{},"State":"running","Status":"Up 5 seconds","HostConfig":
{"NetworkMode":"default"},"NetworkSettings":{"Networks":{"bridge":
{"IPAMConfig":null,"Links":null,"Aliases":null,"NetworkID":"c3d34e588c
d8d7d10b63865799b6defa59160dd6d10d1a0dd273a5c4a9a72ed3","EndpointID":"
2002a544d9ca7975963bd8df43ba7cc0f53639622b06e12ebea2b77bed38ca10","Gat
eway":"172.17.0.1","IPAddress":"172.17.0.2","IPPrefixLen":16,"IPv6Gate
way":"","GlobalIPv6Address":"","GlobalIPv6PrefixLen":0,"MacAddress":"0
2:42:ac:11:00:02","DriverOpts":null}}},"Mounts":
[{"Type":"bind","Source":"/var/run/docker.sock","Destination":"/var/
run/docker.sock","Mode":"","RW":true,"Propagation":"rprivate"}]}]
```

—> The following provides the information by going through the Docker image.

NOTE: To obtain the IP address of the docker container, use the following:

```
$ docker ps | grep socat
8d80277848b3          socat:1               "socat TCP-LISTEN:12…"   3
minutes ago      Up 3 minutes         1234/tcp                docker-sock-
proxy
```

The bold text, **8d80277848b3**, is the container ID to use in the docker inspect command.

```
$ docker inspect 8d80277848b3 | grep "IPAddress"
          "SecondaryIPAddresses": null,
          "IPAddress": "172.17.0.2",
                  "IPAddress": "172.17.0.2",
```

Elasticsearch and Kibana

The images for Elasticsearch and Kibana can be found the following link. There are also cloud providers, including Elastic, that provide versions of Elasticsearch that can be used to collect logs instead of using a local instance. For development purposes, having the entire environment able to run on a laptop simplifies the task and allows for quicker turnaround.

https://www.elastic.co

Download the Elasticsearch and Kibana tar balls. For the local environment (on a Mac) I downloaded them to the following (/Users/notterness/Kibana /Users/notterness/Elasticsearch) and and added the two search lines to *~/.bash_profile*.

```
export PATH=/Users/notterness/Kibana/kibana-7.5.2-darwin-x86_64/bin:
$PATH
export PATH=/Users/notterness/Elasticsearch/elasticsearch-7.5.2/bin:
$PATH
```

To run the two services (make sure the .bash_profile is correct):

```
elasticsearch
kibana
```

To check if Elasticsearch is up and running:

```
$ curl -X GET http://localhost:9200/
{
  "name" : "Noels-MacBook-Pro.local",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "on6fovJPSRORj0akxWVCfg",
  "version" : {
    "number" : "7.5.2",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "8bec50e1e0ad29dad5653712cf3bb580cd1afcdf",
    "build_date" : "2020-01-15T12:11:52.313576Z",
    "build_snapshot" : false,
    "lucene_version" : "8.3.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

To access Kibana from the browser:

http://localhost:5601

Once in Kibana, to view the logs, the different indexes will need to have their patterns created. From, Kibana, this is on the right hand side under the "gear" icon.

Running the Kubernetes POD

There is actually one addition piece in addition to the deployment-webutils.yaml file that is required to allow the POD to actually be functional. That is the service-webutils.yaml file that sets up the ports and the names for the various services within the POD to make them accessible.

To get everything running (from ./ObjectServer/kubernetes):

```
$ kubectl create -f deployment-webutils.yaml
$ kubectl create -f service-webutils.yaml
```

To shut everything down:

```
$ kubectl delete service webutils-service
$ kubectl delete deployment webutils-site
```

Code Path Overview

This describes the various pieces needed to implement a REST command from the test code all the way through to the service that actually implements the handler.
For this example the GET Object method will be used.

The services all use a sub-class of the RequestContext to handle requests. For the Object Server which implements the GET Object method, this is the ObjectServerRequestContext. These are allocated at the start of time and held in the ObjectServerContextPool. This allows the number of outstanding requests to be controlled and the resources required to be more closely managed.
At startup, WebServerMain in the main() function, first creates the ObjectServerRequestContextPool. It then passes that into the instantiation of the NioServerHandler class. The NioServerHandler provides the top level class that manages the NIO TCP socket handlers.

WebServerMain
   |——> NioServerHandler
         |——> NioEventPollBalancer
               |——> NioEventPollThread

When a new socket is accepted, a RequestContext is allocated in the NioEventPollThread via the registerClientSocket() method.
The various operations to handle the initial parsing of the data are created in the ObjectServerRequestContext class in the initializeServer() method. The BufferReadManager and ReadBuffer operations are instantiated in initializeServer() and ParseHttpRequest is instantiated in initializeHttpParsing() which is called from initializeServer(). Once the operations are created, then a ByteBuffer from the ring is provided to the NIO handler via the metering.event() call (metering is the local variable for the BufferReadMetering operation). At his point, the IoInterface is ready to read data into a ByteBuffer and the HTTP Parsing is ready to begin.

There are two Operations that are used to bring data from the NIO socket handler in the ByteBuffer(s). The first is the ReadBuffer Operation that registers the BufferManager with the IoInterface. In this case, the NioSocket class implements the IoInterface. The second is the BufferReadMetering Operation that is used to "allocate" a ByteBuffer that can be used to place data into by the NioSocket read (this actually advances an index that points to an available ByteBuffer in the ring).
The actual parsing of the ByteBuffer(s) is handled by the ParseHttpRequest Operation. This is provided a readBufferPtr (this is the pointer into the BufferManager that is used to obtain data that has been read in from the NIO socket handler). The httpBufferPointer is dependent upon the readBufferPointer (provided to the ParseHttpRequest class at instantiation time) and will be updated (meaning data is available to be parsed) when the readBufferPointer is

ParseHttpRequest

|——> ByteBufferHttpParser
                |——> listener = new HttpRequestListener
                |——> HttpParser (listener) - This is the Jetty HTTP Parser

The HTTP Parser is only responsible for parsing the HTTP headers and does not parse any of the content of the request. It is the request handler's responsibility to determine what to do with the content and how to parse it, if at all. For example, for the PUT Object, the content is directly written to the backing storage without any requirement to parse the data (the data is encrypted before being written and run through a checksum to insure it is received correctly).
The HttpRequestListener is responsible for saving away the data that is extracted through the Jetty parser. It provides the callbacks that required by the HttpParser.RequestHandler (it is the implementation of the interface).
The work of feeding data into the Jetty parser is done in the ByteBufferHttpParser operation. This takes a ByteBuffer and breaks it up into chunks (these are strings that are delineated by a CR of a LF) and then walks through the chunk by passing it to the HttpParser. The HttpParser is not guaranteed to parse everything within a chunk and the ByteBuffer may need to be feed into the HttpParser multiple times (the HttpParser is responsible for moving the pointers within the ByteBuffer). Since the headers must be terminated by either a CR or a LF, when a chunk is parsed, a check can then be made to determine if all the headers have been read in (this is a flag that is set by the Jetty parser).
Once the parsing of the headers is completed, then the DetermineRequest operation is run. This uses the Map<HttpMethodEnum, Operation> supportedHttpRequests to determine if a valid request was made and if so, which handler operation to execute. The handler operations are setup in the ObjectServerRequestContext initializeServer() method and added to the supportedHttpRequests map there. The HttpMethodEnum is used to provide an easy identifier between the HTTP Request and the operation that is used to handle the request. The operation that is used to handle the request usually is just a starting point for the request and often it will setup a chain of operations to perform different pieces of work (i.e. parsing and managing the content data for the request).
In the case of the SetupObjectGet operation, which handles the GET method for the Object Server (this is what reads the object data up from the backing storage and returns it to the client), it sets up two sub-operations: ReadObjectChunks and RetrieveObjectInfo. The RetrieveObjectInfo operation is used to retrieve the information about the object being requested from the MySQL database. The retrieved object information is how the Object Server knows where (which Storage Servers) to read the various object chunks from to create one object blob to return to the client. Once the information about the object is retrieved from the database, then the ReadObjectChunks operation is started.
The ReadObjectChunks operation uses a state machine within the execute() method to hopefully make the different phases of reading up chunks of the object clearer to understand. The first step is to determine the Storage Servers to read the chunk data in from. This is an ordered array and is determined by checking that the Storage Server has not been marked offline (due to some sort of error). Once the array of Storage Servers has been determined, then

the reads from each Storage Server are sent out in parallel. The read is performed by the SetupChunkRead operation.