Scalability of Web Systems Exam

Thor Olesen (tvao@itu.dk)

January 2017

Contents

1 Reflecting on the assignments (25%)

1.1 A. Describe the challenges you faced working with satellite data.

During the three assignments, a web service was developed in Go to query satellite images stored as JPEG 2000 in a Google Cloud Storage Bucket. The final version is capable of querying satellite images based on either 1) a specific latitude and longitude, 2) a geographical area specified as two latitude and longitude pairs and 3) a country represented as a geometric region cover. In this regard, the main three challenges constituted understanding the Sentinel-2 satellite domain, leveraging scale in Go and supporting failure handling.

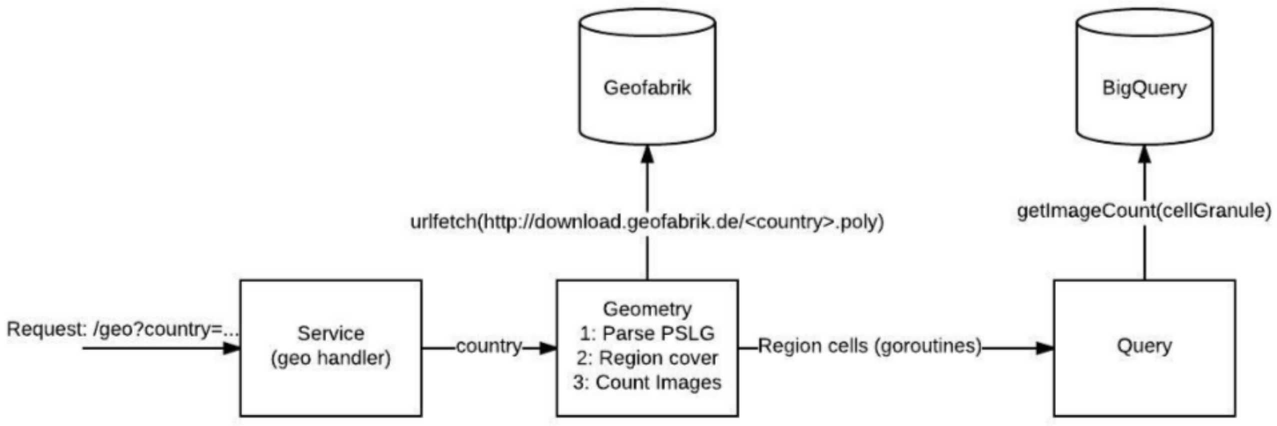
Firstly, the domain was not familiar to us, meaning a lot of time was spent understanding the satellite image format and how to query satellite images, using a RESTful web API for Google Bigquery and Google Storage in Go. Namely, concepts like tiles, granules and latitudes and longitudes had to be fully grasped, in order to build the satellite web service. In this regard, it was realized that the Sentinel-2 images are organized in a grid composed of tiles that represent 100km2 areas on earth, and a granule is a tile in a given grid at a given point in time. Consequently, the web service has to work with this data structure, using latitudes, longitudes and granules ids to query satellite images (see document on assignment 1 for more on this).

Secondly, in assignment two it was found that the web service was somewhat slow when having to query satellite images in geographical areas. Namely, the web service had to support querying images located in a geographical area of interest, specified by two latitude and longitude bands. This is done using Bigquery to retrieve a link to all relevant granules and using Google Storage to query image data located in the Sentinel-2 Google Cloud Bucket. Doing this sequentially resulted in a very high latency (i.e. HTTP request response time) and low throughput. By illustration, the web service spent 1 minute to query only 1500 image links, which was profiled, using the HTTP benchmarking tool "go-wrk". To investigate this, Go's profiling tools were used to identify any potential bottleneck. In this regard, it was found that a lot of memory allocations were spent during Bigquery and Storage queries and the images residing in different bucket folders are fetched sequentially. To overcome this challenge, a simple worker pool was implemented, using goroutines and channels (message passing), which is the idiomatic way to leverage concurrency in. As a result, the web service became capable of fetching the same 1500 images in 2 seconds - as opposed to the previous response time of 1 minute.

Finally, in assignment three, the web service was extended to support querying satellite images, based on a given country, which posed a tremendous scalability challenge. Namely, the worker pool abstraction was used to query thousands of images, located across different granule folders in the Sentinel-2 Google Cloud Bucket. However, this resulted in high delays and byzantine (i.e. arbitrary) failures, caused by the massive amounts of concurrent API calls to the Bigquery and Storage service. To counter this, a retry mechanism was implemented to make the web service more resilient by retrying failed API calls to the Google Cloud services. As a result, the web service behaved consistent and accurate but at a great cost of throughput and latency. The reason for this is was due to the retry mechanism that potentially runs every API call up to five times by default. Consequently, when querying a region cover of Denmark with 4070 granules, 4070 goroutines are spawned to fetch images concurrently, which may take up to 20.000 concurrent API requests to the Google Storage service. In order to address this, a compromise was between the trade offs of accuracy, latency and throughput. On one hand, one may wat to consistently find the correct amount of satellite images that match a country, which favors accuracy over throughput and latency.

On the other hand, one may compromise this by approximating the result to favor throughput and low latency. Initially, the web service did not scale well with the size if the country, since a bigger country is comprised of more granules, and each granule spawns a new goroutine to request images in the Sentinel-2 bucket. This was not due to the many thousand of goroutines, as they represent lightweight threads of execution that are cheap to create. Rather, the bottleneck was caused by the many granule storage bucket requests. In this regard, the performance and latency was improved by counting the satellite images directly via Bigquery, instead of querying the amount of images for each granule in the storage bucket. This was done by querying the amount of images for a single granule only and assuming that this number is consistent across all granule image folders in the sentinel-2 bucket. Although this is a strong assumption, and the query results may vary to a certain degree, it scales dramatically better in practice. For example, it took 120 seconds to query the amount of images of Denmark using the Storage client. By comparison, the modified web service spent less than 2 seconds. Interestingly, the same image count was yielded by both versions but with a huge performance gain in the latter. As shown below, the storage client is not used in the web service:

Figure 1: Satellite Web Service Request Flow


Geofabrik
BigQuery
urlfetch(http://download.geofabrik.de/<country>.poly)
getImageCount(cellGranule)
Request: /geo?country=...
Service (geo handler)
country
Geometry
1: Parse PSLG
2: Region cover
3: Count Images
Region cells (goroutines)
Query

Thus, this section clearly illustrates the challenges of scalability and errors in distributed systems.

1.2 B. How would the absence of Bigquery impact your design in assignment 1?

This would impact the design, by having to use a different set of cloud services to query sentinel-2 satellite images. Namely, Google Bigquery and Storage is used in assignment 1. In the absence of Bigquery, I would consider using a different a cloud service platform that has made the sentinel data available. For example, Amazon AWS provides public access to Sentinel-2 data[1] and the AWS SDK is available for Go. Seemingly, the structure of the data would not change and the implementation would only need to be modified to use the AWS SDK, instead of the bigquery and storage libraries in Go. However, from a business perspective the pricing might be somewhat different, meaning it might be more expensive to transition from the Google Cloud platform to another Cloud service provider like e.g. Amazon.

[1]`http://sentinel-pds.s3-website.eu-central-1.amazonaws.com/` seen 4/1/18

2 Server-side programming (25%)

2.1 A. How can a service such as in Assignment 1 scale with a large number of users?

In general, a scalable web architecture in the cloud is achieved by using a stateless web architecture that avoids session state to leverage horizontal scaling, and handles failures through redundancy mechanisms. Thus, the web service should avoid any session state across user requests. Otherwise, one has to introduce means of synchronization to avoid race conditions and communicate session state across servers, ultimately making it hard to scale horizontally and leverage the performance benefits of distributed processing. Instead, the web service is made stateless with a minimum of session state and each user request is handled separately. Go provides this out of the box in the **net/http** package for making HTTP requests. Namely, each HTTP requests is handled concurrently in goroutines by default in Golang. This is done in the **Serve** function that accepts incoming HTTP connections, and creates a new service goroutine for each of them[2]. Ultimately, this helps scale the web service to handle a large amount of concurrent user connections.

Needless to say, the Google Cloud Platform further supports making the web service scalable to support a large number of users. Namely, the web service is hosted using the Google Cloud App service engine, and the Google Cloud platform exposes a HTTP load balancer that improves the distribution of workloads across multiple computing machines located in Google data centers. The HTTP load balancer uses an Autoscaler system component to add or remove Google App Engine instances of the web service in response to traffic, measured as the amount of requests per second (RPS).

Thus, the web service should work well for 1 user or 1,000,000 users, and its resources should increase of decrease, based on the amount of users. This provides a high system elasticity, meaning the system may adapt to increasing or decreasing workload changes by provisioning or deprovisioning resources automatically. For example, Bigquery is an IaaS cloud computing service used in assignment 1 to query images, without the developer having to worry about the infrastructure and resource management used to accommodate for scale but only the development of the web service). Again, it is important not to store any stateful data across user requests because it makes it hard to distribute app instances of the web service across multiple cluster machines. Also, it is still important to consider how to leverage a concurrent design to query images as a developer in the web service, but the scale of user connections is seemingly handled by the Google cloud ecosystem and the **net/http** package.

[2]`https://golang.org/pkg/net/http/` seen 4/1/18

2.2 B. Can a RPC-Based service scale better or worse than a REST-Based service?

As mentioned already, a scalable web architecture should design for scale by avoiding stateful session state, making it possible to leverage horizontal scaling and distributed processing.

In the general case, a RPC-based service is assumed to be stateful and thus scales worse than a REST-based service that is stateless. In other words, a RESTful stateless web service does not rely on session state on the server-side, meaning it is easier to leverage horizontal scale. On the other hand, a stateful RCP service relies on session state, stored on specific servers to process requests, ultimately making it hard to scale horizontally, since one has to potentially synchronize the session state across servers. However, REST based web services use HTTP and a standard data exchange format between applications like JSON or XML.

In terms of scale, HTTP is subject to the pitfall of using numerous TCP handshakes per client connection and using a data format that is not optimized for scale. A counter example of this is Google's RPC framework, gRPC, that uses the HTTP/2 request-reply protocol and protocol buffer mechanism for serializing structured data in a binary compact format. Interestingly, there is an increasing trend in using a micro-service architecture instead of a monolithic architecture (i.e. app as a single unit) to scale using independent, deployable, small modular services that are designed to communicate with each other. Ultimately, this architecture may be used to leverage horizontal scale across cluster machines using scaling techniques, such as cloning and partitioning.

However, the main challenge developers face in a modular architecture is to enable reliable and fast communication between these services through Remote Procedure Calls (RPCs), using a specific type of message format. As a reaction to this demand, Google invented the gRPC framework that uses HTTP/2 and protocol buffers to promote a request-response model, using a single TCP connection with a compact binary data exchange format. Ultimately, this is more scalable than using a REST JSON API, since HTTP/2 reduces latency be enabling request and response multiplexing (i.e. reusing a single TCP connection across client requests) and protocol buffers increase throughput by enabling binary serialization of transported data, leveraging faster encoding/decoding or marshalling/unmarshalling of data.

3 Topics from the lectures (25%)

3.1 A. An important aspect of any distributed system is eliminating single points of failures.

3.1.1 (a) What are the two main methods to eliminate single points of failure?

As recalled, distributed systems are subject to network errors and need to be designed with failure in mind. In this regard, this impacts the architecture that has to avoid single point of failures using scaling techniques, such as cloning and partitioning to help leverage redundancy across system components and make them tolerant to failure of cluster nodes without compromising availability. Thus, one may either clone or partition when scaling out to increase redundancy and improve availability in a distributed system.

The 1st method of cloning is used to scale out by replicating system components across cluster nodes. The components are stateless, meaning no consistency requirements are present. An example of this might be a stateless 3-tier web service that relies on minimum session state and has a clear separation between stateful and stateless system components. Thus, it may be cloned across multiple cluster nodes without having to immediately worry about communicating information about users (session state) between cluster servers. The 2nd method of partitioning is used to scale out system components components with state by partitioning them across nodes. Thus, the system components are stateful, meaning they may be subject to data loss failures.

3.1.2 (b) Discuss the pros and cons of each method.

In general, the cloning method creates a duplicate copy version of the system components, where as the partition method splits the system components across cluster nodes directly. Thus, an immediate consequence of this is that the latter method requires a backup solution to support fault tolerance. Namely, if two server crash and their data is not replicated across other nodes, all data is lost. Further, partitioning replicates stateful data across cluster nodes, meaning the problem of consistency is introduced.

On the other hand, cloning a stateless distributed system does not pose this challenge. Namely, by avoiding state and using immutable data only, one does not have to address the challenge of linearizability. In other words, distributed systems with replicated copies of stateful data are subject to a compromise between transactions being linearizable or non-linearizable. This means that transactions should either occur at one point in time and block until its effects are visible (linearizable) or inconsistent replicas may be read (i.e. stale values) to help promote high availability. Thus, by using a stateless system architecture, one may use cloning without having to worry about linearizability and backup mechanisms. Ultimately, state makes it harder to enable horizontal scale in distributed systems.

3.2 B. What is the role of replication in distributed systems? What is the difference between master-slave and peer-to-peer replication? What are the problems associated to maintaining consistency across replicas?

As recalled, distributed systems are subject to network errors. In this regard, replication is a mechanism used in distributed systems to address failure handling, by making redundant copies of data across multiple cluster nodes, in order to leverage higher availability. In other words, replication helps build redundant system components that are tolerant to the failure of cluster nodes without necessarily compromising the availability of the system. An example of this might be a NoSQL store that is scaled horizontally be partitioning the data across distributed cluster machines to spread the loads. In this regard, the data may either be distributed using a master-slave approach or peer-to-peer approach as shown below:

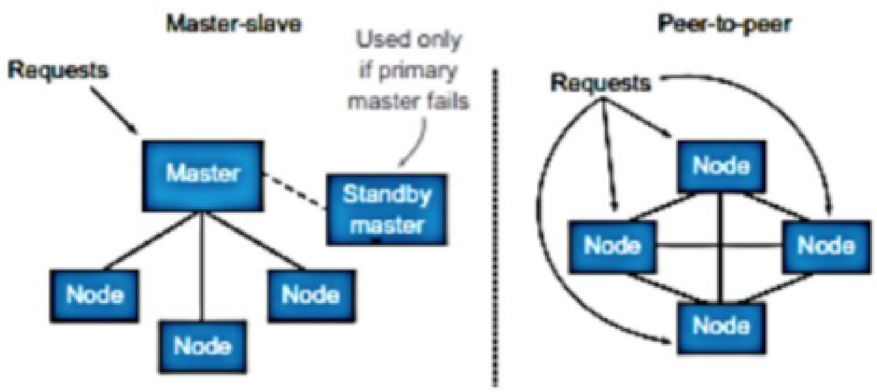


Figure: Master-slave versus peer-to-peer

In terms of their differences, the master-slave is simple to implement and its data locations are known, but it is subject to single point of failure. On the other hand, a peer-to-peer network is more complex with unknown data locations but more robust than the master-slave approach. In general, the master-slave structure is more effective and the peer-to-peer approach is more resilient, which should be considered when choosing a distribution model.

As a side effect of replication, a distributed system may achieve availability but not consistency, since the replicas are not guaranteed to be consistent. In this regard, one may consider using the CAP theorem proposition to help reason about the proposed system traits and understand the tradeoffs involved in distributed systems. Namely, distributed database systems may either allow reads before updating all nodes to ensure high availability or lock all nodes before allowing reads to ensure high consistency. Thus, the CAP theorem helps reason about the general trade off between consistency (C) and availability (A) in a distributed system. However, it is important to emphasize the fact that THE CAP theorem only applies due to the occurrence of failures, caused by having multiple network partitions that may introduce failures. In short, the replication of data introduces the possibility of failures that force the developer to decide between reducing either the desirable property of availability or consistency.

4 Go programming (25%)

4.1 A. Explain the Go calls that occur when a new HTTP request issued by a client reaches the indexHandler. How does Go enable scalability for multiple requests? How do Go compare to the Unix OS processes and threads.

The example program uses the built-in **net/http** and **encoding/json** package to create a web server that handle HTTP requests and encode and decode JSON data. The main method is used to register the HTTP handlers, indexHandler and imageHandler:

```
func main() {
        ...
        http.HandleFunc("/images", imageHandler)
        http.HandleFunc("/", indexHandler)
        ...
}
```

Upon a new HTTP client request, it is redirected to the indexHandler by default, since the **http.HandleFunc** tells the http package to handle all requests to the route ("/") with the indexHandler. The indexHandler function is of type **http.HandleFunc** that takes an **http.ResponseWriter** and **http.Request**, which satisfies the **http.Handler** interface used to serve responses to HTTP requests. Upon the request reaching the indexHandler, the content type of the HTTP response is set to plain text and it returns back a response containing the message "Welcome to the image service...".

As explained already, Go enable scalability for multiple requests by spawning a new service goroutine for each request in the **ServeHTTP** method exposed by the **http.Handler** interface. Compared to traditional threads and Unix OS processes, goroutines are lightweight threads managed by the Go runtime. Thus, you may run thousands or millions of goroutines on your machine depending on the hardware, whereas threads map directly to the low level OS threads and are thus relatively "heavyweight".

In general, the differences mainly concern the memory consumption, cost of context switching and setup and tear down costs. Namely, the creation of goroutines does not require much memory, where as threads start out with more. Consequently, a server handling incoming requests can create one goroutine per request without a proble, which would not be possible with any language using OS threads. Further, threads do pose a significant setup and teardown cost because resources have to be requested from the OS, which is the reason why worker pools are used (to promote reuse). In contrast, goroutines are created and destroyed by the runtime as cheap operations. Finally, goroutines are scheduled cooperatively, meaning less memory registers need to be saved and restored, as compared to threads. All together, go routines provide a high level abstraction on top of the threads on which they are multiplexed to, that is lightweight and easy to understand for the Go programmer to leverage concurrency without having to deal with threads [1].

4.2 B. Explain how Go's concept of interfaces can enable sharing common code across web services.

Go provides interfaces to make your code more flexible, modular and scalable by using abstractions. Formally in Go, interfaces are named collections of method signatures and to implement then, one just needs to implement all the methods in the interface. Thus, interfaces in Go are implemented implicitly and the developer does not have to specify that e.g. type T implements interface I., which is inferred by the Go compiler automatically. In addition to this, Go favors composition over inheritance to help build large programs from smaller parts.

Go's concept of interfaces can enable sharing common code across web services by making it possible to specify shared behavior across these services. For example, the **http.Handler** interface may be used to dictate how response headers and bodies should be responded by providing a custom **ServeHTTP** method with the following signature:

```go
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```

Interestingly, this was used in the assignments along with composition to make a handler that supports better error handling and reuses context resources across its different web services:

```go
// User friendly error representation with error, message and HTTP status code
type appError struct {
        Error   error
        Message string
        Code    int // Server (500 Internal Error) or Client (400 Bad Request Error)
}
// Implement ServeHTTP to comply with the http.Handler interface
// NB: fn is a first order function that invokes the underlying http request function
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        ctx := appengine.NewContext(r)
        ctxWithDeadline, cancel := context.WithTimeout(ctx, 5*time.Minute)
        if err := fn(w, r.WithContext(ctxWithDeadline)); err != nil {
                http.Error(w, err.Message, err.Code)
        }
        defer cancel() // Cancel ctx as soon as request returns
        defer r.Body.Close()
}
```

By using a common Go interface across the HTTP handlers and the satellite web services, one may now pass on the context object that encapsulates information across the different APIs and web services. Consequently, one no longer has to write repetitive code to create the context object across all the web services implemented in the server, geometry and query packages (see appendix). Finally, the context becomes scoped to each request, meaning it is never kept around more than strictly needed and it does not leak any resources. Thus, the interface abstraction helped pass along the context with each request across the services and runs during the lifetime of a request without having any repetitive code.

4.3 C

4.4 You should use these questions to (a) find potential bottlenecks in this Go program and (b) propose a solution that improves the program scalability.

The imageHandler may be improved dramatically by using goroutines and channels to fetch images concurrently. In the current version, it takes all of the urls in the Work slice and fetch data sequentially in downloadImage and getImage. Alternatively, one may spawn off one new goroutine per url work item to download and get the images and use channels to communicate back the results to the main thread in the imageHandler. In addition, one might choose not to analyse every pixel in all images and instead use approximation to reduce the total amount of computations (this was suggested in assignment 1 for the color feature). Ideally, one should use a worker pool abstraction to delegate the url work items across worker goroutines, as done in the satellite web service:

```
// Worker receives work on jobs channel and send back images links on results channel
func worker(id int, jobs <-chan string, results chan<- Links) {
        for imgFolder := range jobs {
                result, err := getImagesFromBucket(bucketName, imageObject, r)
} }
func area(w http.ResponseWriter, r *http.Request) *appError {
        // Create a set of worker jobs for each link
        numberOfJobs := len(links)
        jobs := make(chan string, numberOfJobs)
        results := make(chan Links, numberOfJobs)
        // Setup worker pool
        for i := 0; i <= numberOfJobs; i++ {
                go worker(r, jobs, results)
}
// Send jobs
        for _, imgLink := range links {
                jobs <- imgLink
}
close(jobs) // Close do indicate this is all work to be done
        // Collect worker results and write them back to client in JSON result
        imageResult := Links{}
        for i := 0; i <= numberOfJobs; i++ {
}
        imageResult = append(imageResult, <-results...)
close(results)
}
encode := json.NewEncoder(w).Encode(imageResult)
```

One would only have to adjust this setting to work with the []Work slice and call downloadImage and getImage concurrently.

References

[1] J.C. Corbett et al. *How goroutines work*. `https://blog.nindalf.com/posts/how-goroutines-work/`. [Online; accessed 4/1/18]. 2014.

[2] Sau Sheong Chang. *Go Web Programming*. Manning Publications, 2016. ISBN: 9781617292569.

[3] *Debunking the Myths of RPC & REST*. `http://etherealbits.com/2012/12/debunking-the-myths-of-rpc-rest/`. [Online; accessed 15/11/17. 2012.

[4] Jonathan Furst. "Concurrency in Go". In: Scalability of Web Systems (MSc), IT University of Copenhagen. 2017.

[5] Michal Lowicki. *Interfaces in Go*. `https://medium.com/golangspec/interfaces-in-go-part-i-4ae53a97479c`. 2017.

5 Appendix

5.1 service.go (satellite web service)

```go
// Custom HTTP appHandler that includes error value for better error handling
type appHandler func(http.ResponseWriter, *http.Request) *appError
// User friendly error representation with error, message and HTTP status code
type appError struct {
        Error   error
        Message string
        Code    int // Server (500 Internal Error) or Client (400 Bad Request Error)
}
// Implement ServeHTTP to comply with the http.Handler interface
// Go functional feature: fn is a first order function
// that invokes the underlying http request function (e.g. get)
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        if err := fn(w, r); err != nil {
                http.Error(w, err.Message, err.Code)
        }
}
//Returns JSON array with links to all satellite images based on a location
func images(w http.ResponseWriter, r *http.Request) *appError {
        if err := r.ParseForm(); err != nil {
                return &appError{err, "Cannot parse data", http.StatusInternalServerError}
        }
        address := r.Form.Get("address")
        lat, lng, err := convertAddressToCoords(address, r)
        if err != nil {
                lat, lng = r.Form.Get("lat"), r.Form.Get("lng")
        }
        validLat, validLng := regexp.MustCompile(Latitude).MatchString(lat),
                              regexp.MustCompile(Longitude).MatchString(lng)
        if !validLat && !validLng {
                return &appError{errors.New("Invalid coordinates"),
                                 "Please provide a valid latitude and longitude",
                                 http.StatusBadRequest}
        }
        projectID := os.Getenv("GOOGLE_CLOUD_PROJECT_ID")
        links, err := getLinks(lat, lng, projectID, r)
        if err != nil {
                return &appError{err, "Unable to retrieve links",
                                 http.StatusInternalServerError}
        }
        if err := json.NewEncoder(w).Encode(links); err != nil {
                return &appError{err, "Unable to map JSON to response",
                                 http.StatusInternalServerError}
        } return nil } // Success
```

```go
// Project 2 : Image data in geographic location
// Returns JSON array with links to satellite images within marked area of interest
// Area is specified by pair of latitude and longitude coordinates as query parameters.
func area(w http.ResponseWriter, r *http.Request) *appError {
        if err := r.ParseForm(); err != nil {
                return &appError{err, "Cannot parse data", http.StatusInternalServerError}
        }
        lat1, lng1, lat2, lng2 := r.Form.Get("lat1"), r.Form.Get("lng1"), r.Form.Get("lat2")...
        if !regexp.MustCompile(Latitude).MatchString(lat1) ||
           !regexp.MustCompile(Latitude).MatchString(lat2) ||
           !regexp.MustCompile(Longitude).MatchString(lng1)||
           !regexp.MustCompile(Longitude).MatchString(lng2) {
                return &appError{errors.New("Invalid coordinates"),
                        "Please provide a valid pair of latitude and longitude bands",
                         http.StatusBadRequest}
        }
        links, err := getImageBaseURL(lat1, lng1, lat2, lng2, r)
        if err != nil {
                return &appError{err, "Unable to retrieve granulelinks",
                                                 http.StatusInternalServerError}
        }
        imageResult := pool(links, r)
        if err := imageResult.Error; err != nil {
                return &appError{err, "Could not fetch pictures from granules",
                                                 http.StatusInternalServerError}
        } // Encode JSON result
        encodeErr := json.NewEncoder(w).Encode(len(imageResult.Links))
        if encodeErr != nil {
                return &appError{err, "Unable to encode JSON",
                                                 http.StatusInternalServerError}
        }
        return nil // Success
}

// Project 3 : Fetch and parse PSLG data of country user inputs from Geofabrik
// Returns count of images associated with bounding box of country
func geo(w http.ResponseWriter, r *http.Request) *appError {
        if err := r.ParseForm(); err != nil || !(len(r.Form.Get("country")) > 0) {
                return &appError{err, "Could not parse specified country location.",
                                 http.StatusBadRequest}
        }
        country := r.Form.Get("country")
        continent := r.Form.Get("continent")
        coords, err := parse(r, country, continent)
        if err != nil {
                return &appError{err, "Could not fetch PSLG data",
                                                 http.StatusInternalServerError}}
        cover := regionCover(coords, 15, 100)
        imageCount, err := imagesByRegion(cover, r)
        if err != nil {
                return &appError{err, "Could not get granules",
                                                 http.StatusInternalServerError}
        }
        encodeErr := json.NewEncoder(w).Encode(imageCount)
        if encodeErr != nil {
                return &appError{encodeErr, "Unable to find region cover",
                                                 http.StatusInternalServerError}
        }
        return nil
}
// Result represents links and wraps errors that may occur
type Result struct {
        Links []string
        Error error
}
// Worker pool fetch images from subfolders in Google Cloud Bucket concurrently using goroutines
func pool(links Links, r *http.Request) Result {
        // Create a set of worker jobs for each link
        numberOfJobs := len(links)
        jobs := make(chan string)
        results := make(chan Result)
        imageResult := Result{}
        // Clients should be reused instead of created as needed.
        // The methods of Client are safe for concurrent use by multiple goroutines.
        client, err := storage.NewClient(r.Context())
        if err != nil {
                imageResult.Error = err
                return imageResult // Error propagated
        }
        // Start goroutine workers
        for i := 0; i <= numberOfJobs; i++ {
                go worker(client, r, jobs, results)
        }
        // Send jobs
        for _, imgLink := range links {
                jobs <- imgLink
        }
        close(jobs) // Close do indicate this is all work to be done
        // Collect worker results and write them to JSON result
        for i := 0; i <= numberOfJobs; i++ {
                result := <-results
                imageResult.Links = append(imageResult.Links, result.Links...)
        }
        close(results)
        return imageResult }
```

```
// Worker receives work on jobs channel and send images for each folder job to result
func worker(client *storage.Client, r *http.Request, jobs <-chan string, results chan<- Result) {
        folderImages := Result{}
        for imgLink := range jobs {
                linkAndGranule := strings.SplitAfter(imgLink, "gcp-public-data-sentinel-2")
                bucketName := linkAndGranule[0]
                imageObject := strings.Trim(linkAndGranule[1], "/")
                //bucketHandle := client.Bucket(bucketName)
                result, err := getImagesFromBucket(client, bucketName, imageObject, r)
                // Retry for better resilience
                if err != nil {
                        err := retry(DefaultRetry().MaxRetries,
                                                     DefaultRetry().Duration*time.Second,
                                                     func() (err error) {
                                                     result, err = getImagesFromBucket(client,
                                                                                       bucketName,
                                                                                       imageObject,
                                                                                       r)
                                                     return
                                                     })
                        if err != nil {
                                folderImages.Error = err
                        }
                }
                folderImages.Links = result
        }
        results <- folderImages
}

// Google Client API may fail in which we want to enforce a retry mechanism to improve the resilie
// Credits: https://blog.abourget.net/en/2016/01/04/my-favorite-golang-retry-function/
// http://sethammons.com/post/pester/
func retry(attempts int, sleep time.Duration, callback func() error) (err error) {
        for i := 0; ; i++ {
                err = callback()
                if err == nil {
                        return
                }
                if i >= (attempts - 1) {
                        break
                }
                /// Randomness to prevent Thundering Herd: upgear.io/blog/simple-golang-retry-func
                jitter := time.Duration(rand.Int63n(int64(sleep)))
                sleep = sleep + jitter/2
                time.Sleep(sleep)
        }
        return fmt.Errorf("after %d attempts, last error: %s", attempts, err) }
```

5.2 query.go (query images with BigQuery and Storage)

```
// Retrieves links (i.e. granules) of satellite images that match location
func getLinks(lat, lng, proj string, r *http.Request) (Links, error) {
        granuleQuery := strings.TrimSpace(fmt.Sprintf(
                `SELECT granule_id, base_url
                 FROM %[1]sbigquery-public-data.cloud_storage_geo_index.sentinel_2_index%[1]s
                 WHERE %[2]s < north_lat
                 AND south_lat < %[2]s
                 AND %[3]s < east_lon
                 AND west_lon < %[3]s;`, "`", lat, lng))

        ctx := appengine.NewContext(r)
        client, err := bigquery.NewClient(ctx, proj)
        if err != nil {
            return links, err
        }

        query := client.Query(granuleQuery)
        query.QueryConfig.UseStandardSQL = true
        rows, err := query.Read(ctx)
        for {
                var row []bigquery.Value
                err := rows.Next(&row) // No rows left
                if err == iterator.Done {
                        return links, nil
                }

                if err != nil {
                        return links, err
                }

                granuleID := row[baseGranuleColumn].(string)
                links = append(links, granuleID)
        }
}

// Project 2 : Image data in geographic location
// Fetches all sentinel-2 image folders that contain image data
// within the specified area of interest, using the Big Query Api
func getImageBaseURL(lat1, lng1, lat2, lng2 string, r *http.Request) (Links, error) {
        imageURLQuery := strings.TrimSpace(fmt.Sprintf(
                `SELECT base_url, granule_id
                FROM %[1]sbigquery-public-data.cloud_storage_geo_index.sentinel_2_index%[1]s
                WHERE %[2]s < north_lat
                AND south_lat < %[4]s
                AND %[3]s < east_lon
                AND west_lon < %[5]s;`, "`", lat1, lng1, lat2, lng2)) // Argument 2, 3, 4, 5
```

```go
        links := Links{}
        client, err := bigquery.NewClient(r.Context(), projectID)
        if err != nil {
                return nil, err
        }

        query := client.Query(imageURLQuery)
        query.QueryConfig.UseStandardSQL = true
        rows, err := query.Read(r.Context())
        if err != nil {
                return nil, err
        }

        row := []bigquery.Value{}
        imageBaseURL, granuleID, fullImageURL := "", "", ""
        for {
                err := rows.Next(&row) // No rows left
                if err == iterator.Done {
                        return links, nil // Returns result
                }
                if err != nil {
                        return nil, err
                }
                // Removes trailing gs:// from bucket name
                imageBaseURL = strings.Replace(row[0].(string), "gs://", "", 1)
                granuleID = row[1].(string)
                fullImageURL = imageBaseURL + "/GRANULE/" + granuleID + "/IMG_DATA/"
                links = append(links, fullImageURL)
        }
}

// Project 3 : Fetch all links to granules containing a subfolder of images
// that match specified area of interest, using Big query API
// This version works in parallel by using goroutines and channels
// TODO: refactor getImageBaseUrl to support setting concurrency level
func getImageCount(client *bigquery.Client, r *http.Request, channel chan int, errors chan error, 
        count := 0
        imageURLQuery := strings.TrimSpace(fmt.Sprintf(
                `SELECT COUNT(granule_id)
                FROM %[1]sbigquery-public-data.cloud_storage_geo_index.sentinel_2_index%[1]s
                WHERE %[2]s < north_lat
                AND south_lat < %[4]s
                AND %[3]s < east_lon
                AND west_lon < %[5]s;`, "`", lat1, lng1, lat2, lng2))

        query := client.Query(imageURLQuery)
        query.QueryConfig.UseStandardSQL = true
        rows, err := query.Read(r.Context())
        if err != nil {
                errors <- err
        }
        row := []bigquery.Value{}
        for {
                err := rows.Next(&row) // No rows left
                if err == iterator.Done {
                        channel <- count // Write image count to channel instead of returning
                        break
                }
                if err != nil {
                        errors <- err
                }
                imgCount := int(row[0].(int64))
                count += imgCount
        }
}

// Project 2 : Image data in geographic location
// Fetches a complete list of image ids from a specified image folder in the sentinel-2 folder, us
func getImagesFromBucket(client *storage.Client, bucketName, objectName string, r *http.Request) (
        query := storage.Query{Prefix: objectName, Versions: false}
        links := Links{}
        fullImageURL := bytes.Buffer{}
        it := client.Bucket(bucketName).Objects(r.Context(), &query)
        for {
                attrs, err := it.Next()
                if err == iterator.Done {
                        break
                }
                if err != nil {
                        return nil, err
                }
                fullImageURL.WriteString(bucketName + "/" + attrs.Name)
                links = append(links, fullImageURL.String())
                fullImageURL.Reset()
        }
        return links, nil
}
```

5.3 app.yaml (deployment configuration)

runtime: go api_version: go1
handlers: - url: /images script: service.images
- url: /.* script: service.index

5.4 converter.go (coordinates to human address)

```
// JSON result returned by Geolocation API
type geoResponse struct {
        Results []struct {
                Geometry struct {
                        Location struct {
                                Lat float64
                                Lng float64
                        }
                }
        }
}

// Converts human-like address to coordinates (latitude and longitude) via Geolocation API
// Request form: https://maps.googleapis.com/maps/api/geocode/json?address=<address>,
// where output is json and the required parameter is an address
func convertAddressToCoords(address string, r *http.Request) (string, string, error) {
        if address == "" {
                return "", "", errors.New("Invalid address input")
        }
        // Escapes string so it is safe to place inside URL query
        safeAddress := url.QueryEscape(address)

        // Geocoding API
        fullURL := fmt.Sprintf("http://maps.googleapis.com/maps/api/geocode/json?address=%s",
                                                  safeAddress)

        // App engine context to interact with external service via http client
        ctx := appengine.NewContext(r)
        client := urlfetch.Client(ctx)
        response, err := client.Get(fullURL)
        if err != nil {
                return "", "", err
        }
        defer response.Body.Close()

        // Generate latitude and longitude from address using Google Geocoding API
        // Use json.Decode or json.Encode for reading or writing streams of JSON data
        var res geoResponse
        if err := json.NewDecoder(response.Body).Decode(&res); err != nil {
                return "", "", err
        }
        lat := strconv.FormatFloat(res.Results[0].Geometry.Location.Lat, 'f', 6, 64)
        lng := strconv.FormatFloat(res.Results[0].Geometry.Location.Lng, 'f', 6, 64)
        log.Printf("Success: converted address '%s' into lat = '%s' and lng = '%s' \n", address, la
        return lat, lng, nil // Success
}
```

5.5 geometry.go - country region cover

```go
// Normalize (i.e. remove exponent) in parsed coordinates
// Credits: https://gobyexample.com/collection-functions
func normalizeCoords(vs []string, f func(string, int) (float64, error)) ([]float64, error) {
        vsm := make([]float64, len(vs))
        for i, v := range vs {
                f, err := f(v, -1)
                if err != nil {
                        return nil, err
                }
                vsm[i] = f //strconv.FormatFloat(f, 'f', -1, 64)
        }
        return vsm, nil
}
// Fetch and parse PSLG data from Geofabrik, based on a country specified by the user
func parse(r *http.Request, country, continent string) ([]float64, error) {
        client := urlfetch.Client(r.Context())
        request := ""
        if len(continent) > 0 {
                request = fmt.Sprintf("http://download.geofabrik.de/%s/%s.poly", continent, countr
        } else {
                request = fmt.Sprintf("http://download.geofabrik.de/%s.poly", country)
        }
        resp, err := client.Get(request)
        // Retry if error
        if err != nil {
                err := retry(DefaultRetry().MaxRetries, DefaultRetry().Duration*time.Second, func(
                        resp, err = client.Get(request)
                        return
                })
                if err != nil {
                        return nil, err
                }
        }
        defer resp.Body.Close()
        regex := regexp.MustCompile(floatExponentPattern)
        bytes, err := ioutil.ReadAll(resp.Body)
        if err != nil {
                return nil, err
        }
        data := regex.FindAllString(string(bytes), -1)
        countryCoords, err := normalizeCoords(data, strconv.ParseFloat)
        if err != nil {
                return nil, err
        }
        return countryCoords, nil
}
```

```go
// Construct region cover from polygon, based on country coords (a union of cells, i.e. CellUnion)
// MaxLevel determines the granularity of cells covering regions, where 30 = 0,48 cm^2
// MaxCells determines how many cells are used to cover the given region
func regionCover(coords []float64, maxLevel, maxCells int) s2.CellUnion {
	points := []s2.Point{} // Parse coordinates into points
	for len(coords) > 0 {
		lat, lng := coords[0], coords[1]
		p := s2.PointFromLatLng(s2.LatLngFromDegrees(lat, lng))
		points = append(points, p)
		coords = coords[2:] // Rest coords
	} // Construct loop representing spherical polygon and polygon from loop
	l1 := s2.LoopFromPoints(points)
	loops := []*s2.Loop{l1}
	poly := s2.PolygonFromLoops(loops)
	rc := &s2.RegionCoverer{MaxLevel: maxLevel, MaxCells: maxCells} // Construct region cover
	cover := rc.Covering(poly)
	return cover
}
// Count satellite images associated to a country based on its polygon representation
// Use region cover data in combination with query.go to query relevant images with Storage API
func imagesByRegion(cover s2.CellUnion, r *http.Request) (int, error) {
	numberOfJobs := len(cover)
	results := make(chan int, numberOfJobs)
	errChan := make(chan error)
	imageCount := 0
	client, err := bigquery.NewClient(r.Context(), projectID)
	if err != nil {
		return 0, err
	} // Fetch image base links in parallel
	for i := 0; i < len(cover); i++ {
		c := s2.CellFromCellID(cover[i])
		go getImageCount(client, r, results, errChan,
			c.RectBound().Lo().Lat.String(),
			c.RectBound().Lo().Lng.String(),
			c.RectBound().Hi().Lat.String(),
			c.RectBound().Hi().Lng.String())
	}
	for range cover { // Await concurrent results on channel
		select {
		case err := <-errChan:
			return 0, err
		case count := <-results:
			imageCount += count
		}
	}
	close(results)
	return imageCount * bucketGranuleSize, nil}
```

5.6 pool.go - worker pool abstraction

```go
// Worker pool abstraction that uses goroutines and channels for concurrency
// Credits: https://brandur.org/go-worker-pool

// Task encapsulates a work item that should go in a work pool.
type Task struct {
        // Err holds an error that occurred during a task. Its
        // result is only meaningful after Run has been called
        // for the pool that holds it.
        Err error

        f func() error
}

// NewTask initializes a new task based on a given work function.
func NewTask(f func() error) *Task {
        return &Task{f: f}
}

// Run runs a Task and does appropriate accounting via a given sync.WorkGroup.
func (t *Task) Run(wg *sync.WaitGroup) {
        t.Err = t.f()
        wg.Done()
}

// Pool is a worker group that runs a number of tasks at a configured concurrency.
type Pool struct {
        Tasks []*Task

        concurrency int
        tasksChan   chan *Task
        wg          sync.WaitGroup
}

// NewPool initializes a new pool with the given tasks and at the given concurrency.
func NewPool(tasks []*Task, concurrency int) *Pool {
        return &Pool{
                Tasks:       tasks,
                concurrency: concurrency,
                tasksChan:   make(chan *Task),
        }
}
```

```go
// Run runs all work within the pool and blocks until it's finished.
func (p *Pool) Run() {
	for i := 0; i < p.concurrency; i++ {
		go p.work()
	}

	p.wg.Add(len(p.Tasks))
	for _, task := range p.Tasks {
		p.tasksChan <- task
	}

	// all workers return
	close(p.tasksChan)

	p.wg.Wait()
}

// The work loop for any single goroutine.
func (p *Pool) work() {
	for task := range p.tasksChan {
		task.Run(&p.wg)
	}
}

// HOW TO USE IT
// tasks := []*Task{
//     NewTask(func() error { return nil }),
//     NewTask(func() error { return nil }),
//     NewTask(func() error { return nil }),
// }

// p := pool.NewPool(tasks, conf.Concurrency)
// p.Run()

// var numErrors int
// for _, task := range p.Tasks {
//     if task.Err != nil {
//         log.Error(task.Err)
//         numErrors++
//     }
//     if numErrors >= 10 {
//         log.Error("Too many errors.")
//         break
//     }
// }
```

5.7 service_test.go - integration tests

```go
// Test actual retrieval of images granules based on invalid lat/lng, should return error
func TestImageHandler_BadRequest(t *testing.T) {
	inst, err := aetest.NewInstance(nil)
	if err != nil {
		t.Fatalf("Failed to create instance: %v", err)
	}
	defer inst.Close()

	// Create a request to pass to handler with query parameters
	req, err := inst.NewRequest("GET", "/images", nil)
	if err != nil {
		t.Fatalf("Failed to create req: %v", err)
	}

	// Create a ResponseRecorder (which satisfies http.ResponseWriter) to record response
	rr := httptest.NewRecorder()
	handler := http.Handler(appHandler(images))

	// Our handlers satisfy http.Handler, so we can call their ServeHTTP method
	// directly and pass in our Request and ResponseRecorder.
	handler.ServeHTTP(rr, req)

	// Check the status code is what we expect.
	if status := rr.Code; status != http.StatusBadRequest {
		t.Errorf("handler returned wrong status code: got %v want %v",
			status, http.StatusBadRequest)
	}

	//Check the response body is what we expect.
	expected := "Please provide a valid latitude and longitude"
	if strings.TrimSpace(rr.Body.String()) != strings.TrimSpace(expected) {
		t.Errorf("handler returned unexpected body: got '%v' want '%v'",
			rr.Body.String(), expected)
	}
}

// Integration test, testing actual retrieval of images granules based on valid lat/lng
func TestImageHandler_ValidRequest(t *testing.T) {
	inst, err := aetest.NewInstance(nil)
	if err != nil {
		t.Fatalf("Failed to create instance: %v", err)
	}
	defer inst.Close()

	// Create a request to pass to handler with query parameters
	req, err := inst.NewRequest("GET", "/images", nil)
```

```go
        req.Form = url.Values{"lat": {"55.660797"}, "lng": {"12.5896"}}
        if err != nil {
                t.Fatalf("Failed to create req: %v", err)
        }

        // Create a ResponseRecorder (which satisfies http.ResponseWriter) to record response
        rr := httptest.NewRecorder()
        handler := http.Handler(appHandler(images))

        // Our handlers satisfy http.Handler, so we can call their ServeHTTP method
        // directly and pass in our Request and ResponseRecorder.
        handler.ServeHTTP(rr, req)

        // Check the status code is what we expect.
        if status := rr.Code; status != http.StatusOK {
                t.Errorf("handler returned wrong status code: got %v want %v",
                        status, http.StatusOK)
        }
}

// Integration test, testing actual retrieval of images in geographic area of interest
func TestAreaHandler_ValidRequest(t *testing.T) {
        inst, err := aetest.NewInstance(nil)
        if err != nil {
                t.Fatalf("Failed to create instance: %v", err)
        }
        defer inst.Close()

        // Create a request to pass to handler with query parameters
        req, err := inst.NewRequest("GET", "/area", nil)
        req.Form = url.Values{"lat1": {"55.660797"}, "lng1": {"12.5896"}, "lat2": {"55.663369"}, "
        if err != nil {
                t.Fatalf("Failed to create req: %v", err)
        }
        // Create a ResponseRecorder (which satisfies http.ResponseWriter) to record the response
        rr := httptest.NewRecorder()
        handler := http.Handler(appHandler(area))

        // Our handlers satisfy http.Handler, so we can call their ServeHTTP method
        // directly and pass in our Request and ResponseRecorder.
        handler.ServeHTTP(rr, req)

        // Check the status code is what we expect.
        if status := rr.Code; status != http.StatusOK {
                t.Errorf("handler returned wrong status code: got %v want %v",
                        status, http.StatusOK)
        }
}
```

5.8 benchmarks_test.go - performance profiling

```go
// Benchmark image query that returns granules related to specified latitude and longitude
func BenchmarkImages(b *testing.B) {
        inst, err := aetest.NewInstance(nil)
        if err != nil {
                b.Fatalf("Failed to create instance: %v", err)
        }
        defer inst.Close()
        // Create a request to pass to handler with query parameters
        req, err := inst.NewRequest("GET", "/images", nil)
        req.Form = url.Values{"lat": {"55.660797"}, "lng": {"12.5896"}}
        if err != nil {
                b.Fatalf("Failed to create request: %v", err)
        }
        // Create a ResponseRecorder (which satisfies http.ResponseWriter) to record response
        rr := httptest.NewRecorder()
        //handler := http.Handler(appHandler(images))
        // Repeat operation  to benchmark (in this case service.images) b.N times.
        // Value will be changed by go test until resulting times are statistically significant.
        for i := 0; i < b.N; i++ {
                //handler.ServeHTTP(rr, req)
                images(rr, req)
        }
}
// Benchmark spatial query that returns all image links within geographical area of interest
func BenchmarkArea(b *testing.B) {
        inst, err := aetest.NewInstance(nil)
        if err != nil {
                b.Fatalf("Failed to create instance: %v", err)
        }
        defer inst.Close()
        // Create a request to pass to handler with query parameters
        req, err := inst.NewRequest("GET", "/area", nil)
        req.Form = url.Values{"lat1": {"55.660797"}, "lng1": {"12.5896"}, "lat2": {"55.663369"}, "
        if err != nil {
                b.Fatalf("Failed to create request: %v", err)
        }
        // Create a ResponseRecorder (which satisfies http.ResponseWriter) to record response
        rr := httptest.NewRecorder()
        //handler := http.Handler(appHandler(area))
        // Repeat operation  to benchmark (in this case service.area) b.N times.
        // Value will be changed by go test until resulting times are statistically significant.
        for i := 0; i < b.N; i++ {
                //handler.ServeHTTP(rr, req)
                area(rr, req)
        }
}
```

```go
// Benchmark geo query that returns all image links within country
func BenchmarkGeo(b *testing.B) {
        inst, err := aetest.NewInstance(nil)
        if err != nil {
                b.Fatalf("Failed to create instance: %v", err)
        }
        defer inst.Close()

        // Create request to pass to geo handler with query parameters
        req, err := inst.NewRequest("GET", "/area", nil)
        req.Form = url.Values{"country": {"Denmark"}, "continent": {"europe"}}
        if err != nil {
                b.Fatalf("Failed to create request: %v", err)
        }

        // Create ResponseRecorder (satisfies http.ResponseWriter) to record response
        rr := httptest.NewRecorder()

        // Run benchmark b.N times until resulting times are statistically significant
        for i := 0; i < b.N; i++ {
                geo(rr, req)
        }
}
```