

|  |
| --- |
|  |
|  |
| **Global Payments Business View Architecture Document** |
|  |

Table of Contents

[System Architecture Overview 4](#_Toc482002264)

[System Context Diagram 5](#_Toc482002265)

[Technology Overview 6](#_Toc482002266)

[Platform 6](#_Toc482002267)

[Languages, Libraries, and Frameworks 6](#_Toc482002268)

[Tooling 7](#_Toc482002269)

[Data 7](#_Toc482002270)

[Application Security 8](#_Toc482002271)

[Authentication 8](#_Toc482002272)

[Authorization 8](#_Toc482002273)

[Front End Application 9](#_Toc482002274)

[Overview 9](#_Toc482002275)

[Description 9](#_Toc482002276)

[Diagram 10](#_Toc482002277)

[Development Environment Setup 10](#_Toc482002278)

[Build Process 10](#_Toc482002279)

[Testing 10](#_Toc482002280)

[Services / RESTful APIs 10](#_Toc482002281)

[Overview 10](#_Toc482002282)

[Details 10](#_Toc482002283)

[Development Environment Setup 11](#_Toc482002284)

[Common Libraries 12](#_Toc482002285)

[Overview 12](#_Toc482002286)

[Details 12](#_Toc482002287)

[Version Control / Continuous Integration / Build & Deploy Process 12](#_Toc482002288)

[Description 12](#_Toc482002289)

[Source Control 12](#_Toc482002290)

[Automated Build and Deploy Process 14](#_Toc482002291)

[Service Versioning 15](#_Toc482002292)

[Testing 16](#_Toc482002293)

[Unit / Functional / Integration Tests 16](#_Toc482002294)

[Load Testing 16](#_Toc482002295)

[Vulnerability Testing 16](#_Toc482002296)

[Data 17](#_Toc482002297)

[Description 17](#_Toc482002298)

[Data Diagram 17](#_Toc482002299)

[Deployment 17](#_Toc482002300)

[Bad Stuff 17](#_Toc482002301)

[Appendix 17](#_Toc482002302)

[A. Business View Non-Functional Requirements 17](#_Toc482002303)

[1. General Cloud Architectural Requirements 17](#_Toc482002304)

[2. Website Requirements 17](#_Toc482002305)

[3. API Requirements 18](#_Toc482002306)

[4. DevOps Requirements 18](#_Toc482002307)

[5. QA/Testing Requirements 18](#_Toc482002308)

[5.1. Automated Testing is required at multiple levels 18](#_Toc482002309)

[5.1.1. Unit Tests for compiled Business Logic 18](#_Toc482002310)

[5.1.2. JavaScript automated Tests for UI/UX Logic 18](#_Toc482002311)

[5.1.3. API Test Automation 18](#_Toc482002312)

[6. Logging Requirements 18](#_Toc482002313)

[7. High Level Coding Guidelines 19](#_Toc482002314)

# System Architecture Overview

Global Payments has chosen Google Cloud Platform (GCP) for this project and any references to *the cloud* below are a direct reference to the Google Cloud Platform. The decisions in this document are based technology available in GCP, coupled with the Non-Functional requirements [Appendix A], discussions had during the discovery phase of the project, and industry standards and best practices.

Cardinal Solutions has broken down the work into distinct development teams which facilitates quicker delivery by allowing multiple concurrent development streams, smaller testable units, and releases that can be done independent of other components. These major development streams include:

* Front-end application
  + Built using Angular2 and contain the nothing but the visual framework (html, css, JavaScript, etc.) to present a responsive application to the end user regardless of the device they are using.
* Services
  + RESTful set of CORS enabled [3.4] APIs built using Java 7 compartmentalized into bounded contexts and running on a series Google App Engine (GAE) standard instance, a platform as a service (PaaS) [1.2] [3.5] [3.1]
* Processes
  + Google Cloud functions written in node.js that can respond to a variety of triggers including actions on a message queue (Google Cloud PubSub) or a specific HTTP request
* Data
  + Stored in a variety of data stores based on volatility, security, caching and access speed requirements

Each environment in the hierarchy (Dev, QA, UAT, Prod, etc) will live in its own Google Cloud Project [4.1]. This approach helps reduce risk of cross-contaminating API versions, allows for different data to be stored thus protecting any confidential data, and provides for a more cost-effective design as entire environments can be scripted to be stopped started, spun up and torn down at will [4.9]. <<putt – is this truly possible>> A few exceptions to this may arise due to cost prohibitive nature of certain tools.

Being that Google App Engine (GAE) has an uptime SLA of 99.95[[1]](#endnote-1) and Global Payments has an uptime requirement of 99.999 [1.1] it is suggested that multiple production environments be implemented fronted by an external third-party load balancer that can manage traffic and handle failures of one region. Otherwise, Figure 1 below identifies a high level components that have been identified as necessary for each environment.

## System Context Diagram

The System Context diagram shows the interactions between the primary components comprising the Business View (BV) system.

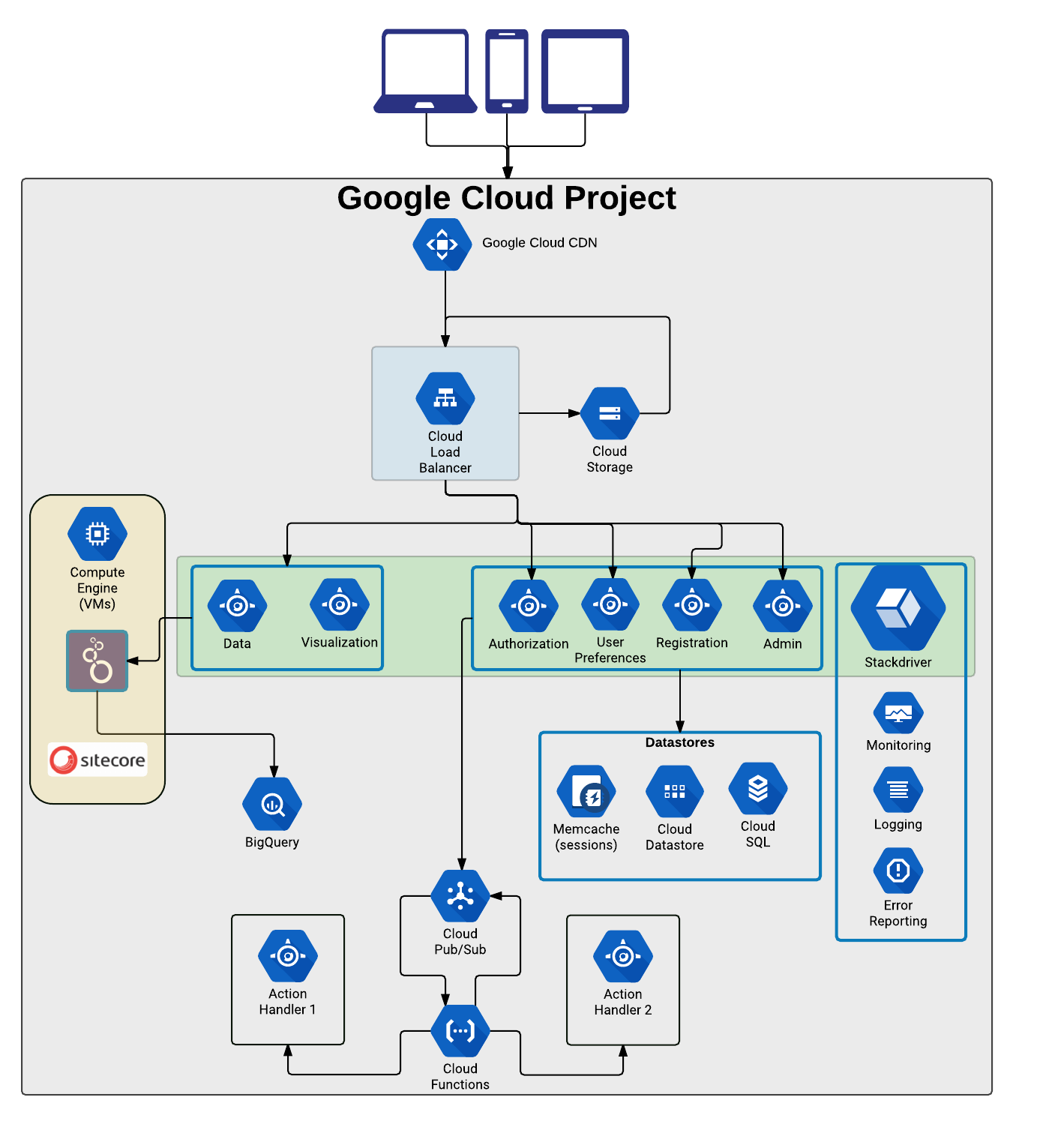


Figure : System Context Diagram

## Technology Overview

The following technology decisions were made based on a combination of guidelines put forth by Global Payments [Appendix A], industry standards, and constraints within the GAE.

### Platform

|  |  |
| --- | --- |
| Google App Engine Standard (GAE, App Engine) | A [cloud computing](https://en.wikipedia.org/wiki/Cloud_computing) platform with a uptime SLA of 99.95% [1.1, 1.2] used for hosting [web applications](https://en.wikipedia.org/wiki/Web_application) in Google-managed data centers. App Engine offers automatic scaling for web applications as the number of requests increases or resource usage rise beyond customer defined thresholds |
| Google Cloud Content Delivery Network (CDN) | Globally distributed network of proxy servers to serve content to end-users with high availability and high performance. |
| Google Cloud Load Balancer | Layer 7 load balancer to direct traffic based on URL patterns |
| Google Cloud Functions | Connective layer of logic that lets you write code to connect and extend cloud services all in a serverless architecture |
| StackDriver | Monitoring, logging, and diagnostics of GAE instances with metrics, dashboards, alerting, log management, reporting, and tracing capabilities. |
| Google PubSub | Fully-managed real-time messaging service that allows you to send and receive messages between independent applications |
| Google Cloud Functions | Cloud functions that allow for real-time processing of events against a Google PubSub in a serverless architecture |

### Languages, Libraries, and Frameworks

|  |  |
| --- | --- |
| Java 7 | All RESTful APIs will be written in Java 7 with the Java Servlet 2.5 standard [[2]](#endnote-2) |
| Spring / Spring MVC 3.2.18.RELEASE | An industry standard library that provides an MVC framework for quickly creating RESTful API's while providing inversion of control (IoC), dependency injection, and a harness for a robust testing framework. |
| Swagger (springfox-swagger2) | An annotation driven library that assists in creating both human and machine readable documentation while providing a human usable test harness for RESTful endpoints |
| Project Lombock | Lombok is used to reduce boilerplate code (primarily in a domain model) by automatically generating accessor methods, as well as implementations for the toString, equals, and hashCode methods |
| Node.js v6.9.1 | Platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices. This is the only language available to write google cloud functions |
| Angular 4.0 | A structural framework by Google, built for dynamic web apps that uses HTML as the core template language and extends HTML's syntax to express application components clearly and succinctly. Additionally, AngularJS's data binding and dependency injection eliminate much of the code one would otherwise have to write |
| Other front end technologies | **TypeScript** – A JavaScript superset by Microsoft that allows for strong typing, advanced features not yet supported in major browsers, and many other features, all of which compile to clean, compatible JavaScript  **SCSS** – A powerful, feature rich preprocessor for CSS that adds improved syntax, functions, variables, and many other features  **Karma** – A powerful, framework agnostic environment for JavaScript unit tests that allows for easy debugging  **Webpack** – A module bundler that turns modular source code into efficient web-ready production code |

### Tooling

|  |  |
| --- | --- |
| Maven | A build automation tool that makes building the application uniform as well as managing the dependencies used in a project |
| Git | A distributed revision control system aimed at speed, data integrity, and support for distributed, non-linear workflows |
| Jenkins | Continuous integration tool used for automated build, testing, and deployment |
| Artifactory | Software repository for hosting shared dependencies |
| gcloud SDK | A set of command line tools to facilitate scripting of google cloud tasks such as deployments, traffic management, and managing cloud storage buckets |

### Data

|  |  |
| --- | --- |
| Cloud SQL | Fully managed MySQL instance for application data |
| Google BigQuery | Enables interactive analysis of massively large datasets |
| Memcache | Caching mechanism to be used for volatile data that can be recreated easily but is able to be cached for faster retrieval |
| Looker | Visualization engine and sematic layer to wrap BigQuery |

# Application Security

Authentication

Authentication for the application will be handled according to the OpenID Connect specification. Currently, Global payments is unsure of who will be the identity provider, but by building to this specification we will be able to build a generic process flow that will be able to be adapted to whichever provider is selected. Figure 2: Authentication process below shows the process of how the front-end client will authenticate and receive a JSON Web Token (JWT) to be used as a bearer token in subsequent requests.

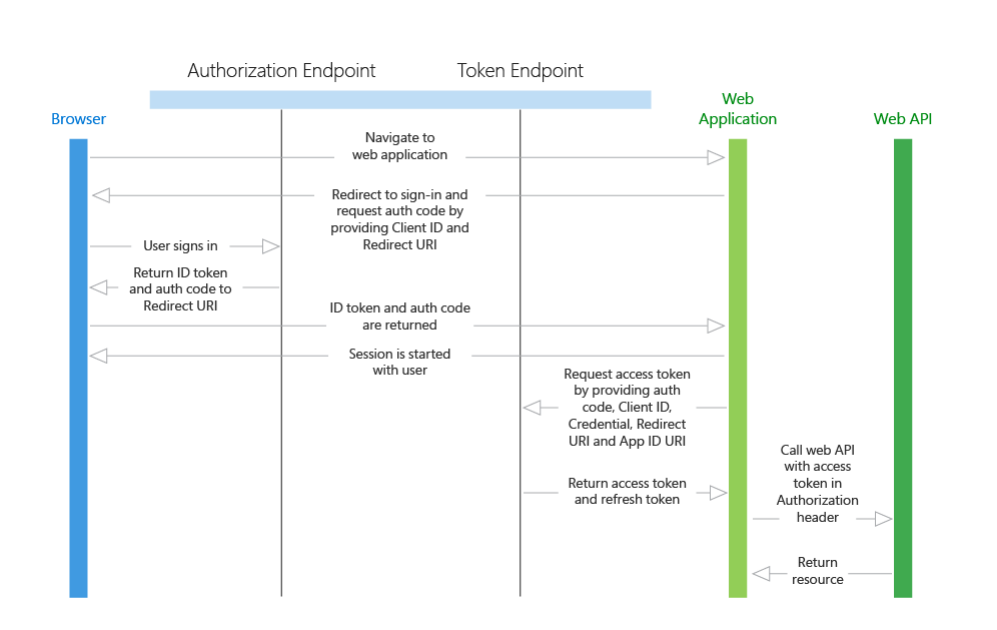


Figure : Authentication process

## Authorization

Once a front-end client has received a JWT, they will need to pass it as an *Authentication* header to each request where it will be immediately intercepted and validated. From here any number of processes can take place. One flow can be to look roles for the user in a memcache instance inside of the same GCP. If the key is found we’ll use the roles found in cache, otherwise we’ll look up the roles and permissions and cache them for the next call. This approach allows a user’s roles to change and the cache to be flushed by a different service as well as quick access to the roles without hitting a database on every request. Figure 3 shows this process as well as error codes returned at each step.

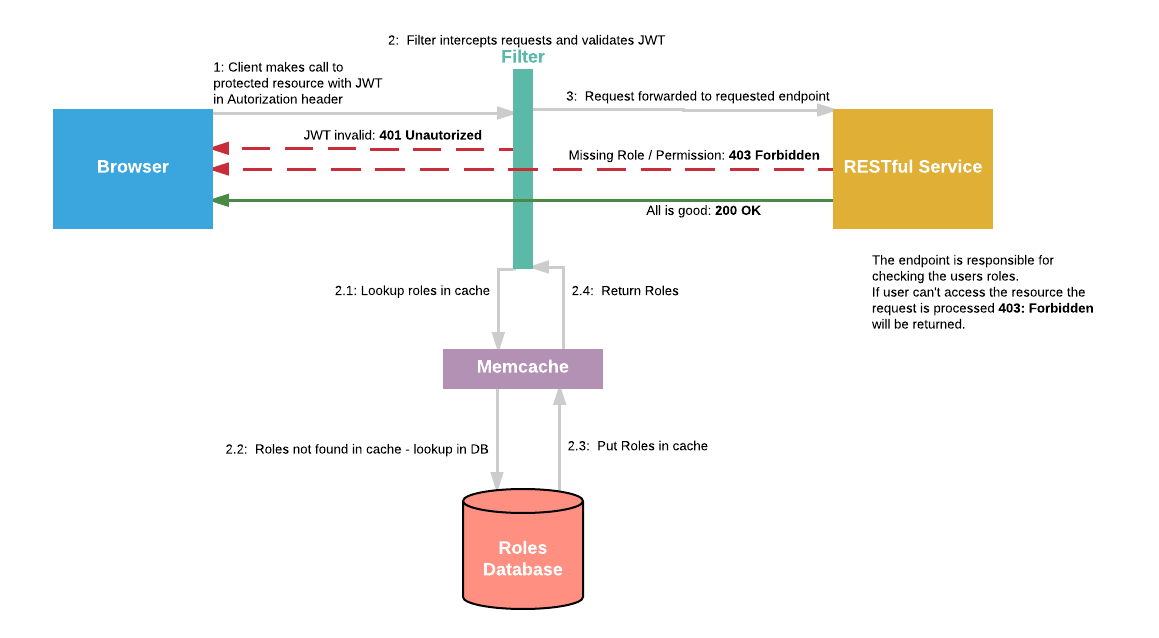


Figure : Authorization process flow

# Front End Application

## Overview

## Description

<<UI/UX>>

Specifically address the following from the non functional requirements:

* Initial Page Response time < 500ms
* All resources (css,js, images) optimized for speed and minimum latency
* Minify and Bundle css and js when appropriate
* Content that requires > 500 MS should be loading via AJAX calls
* All content to load in <5000 MS
* Friendly URLS (i.e. no .jsp or .html)
* Should work on all Modern Browsers (including phone, tablet)
* Custom Javascript written in Typescript if possible

To reduce cost and page load times all front-end resources will be stored in a Google Cloud Storage bucket fronted by a load balancer [1.4] and CDN [2.1.2]. This approach will distribute cached versions of static resources across the world directing users the to the closest version of the resource.

## Diagram

<<UI/UX>>

## Development Environment Setup

## Build Process

The build process for the front-end application will be handled primarily by the Angular CLI.

* Components and unit tests are written, primarily using TypeScript, HTML, SCSS
* All source code is placed into source control.
* Code is then compiled using the Angular CLI, resulting in production ready web assets
* Components are documented and integrated into component library
* Finally, code is deployed to web application

## Testing

# Services / RESTful APIs

## Overview

While we won’t be implementing a microservices architecture in the truest sense – we will be breaking our APIs into groups of smaller functional units, or bounded contexts. This approach allows us the flexibility to make a change in one module then test and deploy it separate from the rest of the application greatly reducing the amount of time spent fixing bugs and regression testing before a deployment can be initiated. In addition, this allows a more granular scaling model as each service can scale up and down based on its own load and resource usage which will reduce overall cost [1.5].

## Details

Each distinct service will reside in a Google App Engine Standard java project - Googles out of the box implementation of its web traffic serving java runtime. While this environment has some restrictions (Java 7, Java Servlet 2.5, only certain libraries whitelisted) it should be adequate for most purposes. Should the need arise to break out of this environment we’ll step up to a Google App Engine flexible environment that allows for greater customization of the runtime container.

As mentioned above we will be breaking down and grouping our services into related functionality. The services listed below are not meant to be exhaustive, but to demonstrate the level of granularity in which our services will be implemented.

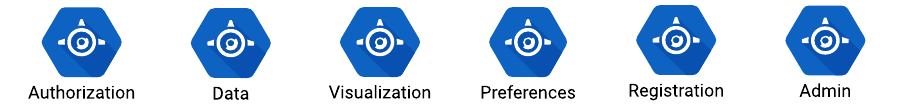


Figure : Example Services

* **Authorization –** Given an authenticated user (session id) lookup roles and permissions
* **Data** – The services will be used for returning raw data to the front end to display as it needs. This service will interact with variety of data storage mechanisms including Google Cloud SQL, Spanner, BigQuery, and Looker Query.
* **Visualization** – This service will be used to interact with the looker visualization tool. It will be used as a pass through to embed looks/dashboards into the front-end iframe as well as run LookML queries to return raw data utilizing the built-in caching strategies of Looker.
* **Preferences** – Used for user specific preferences (language, table / column layout)
* **Registration –** Endpoints for registration process
* **Admin –** Group of APIs used to administer the application as well as users.

Each service will be designed to serve the needs of the Business View application, but they could be exposed publicly through an API gateway such as Apigee for general enterprise consumption. Regardless of whether an API will be exposed externally it will be documented using the Swagger standard which allows for documentation that is easily readable by API consumers as well as provide a quick framework to manually validate and test endpoints [3.3, 3.4]. To facilitate quick adoption all APIs will be RESTful [3.1], follow the Microsoft REST API guidelines [3.5], and be CORS enabled by default [3.4].

All java assets will utilize the built in GCP Stackdriver tools to facilitate logging. [6] With minimal configuration Stackdriver provides error reporting, uptime monitoring, alerts, logging and dashboards all available through the Google Cloud Platform console and third party tools such as Splunk [6.1.1, 6.1.2, 6.1.3, 6.3].

<<runtime configuration>>

As environment variables are stored within the deployment description of an App Engine project, changing these values takes a complete redeploy of the application. Overcoming this limitation can be addressed a couple different ways.

faTo overcome this limitation, we can store configuration items in a file in a cloud storage instance and load them into memcache for future quick reference lookup. The project itself will provide a set of failback or default values should the resource not be available. By taking advantage of this mechanism we can meet the

## Development Environment Setup

All services and common libraries will be developed and deployed as separate modules on each individual developer’s machine. Each module will be able to be run independently on a locally simulated GAE sandbox using the Google App Engine maven plugin.

To keep code standardized across services and common libraries we will create Maven archetypes to assist in scaffolding out new code [7.1]. By creating these archetypes, or project templates, we will be able to easily create a new service and common libraries with all the necessary configuration, libraries, and frameworks built in allowing us to quickly implement new functionality. In addition to allowing for quicker code creation this keeps the format of each module similar so that developers can immediately be familiar with the structure of the code.

# Common Libraries

## Overview

In addition to compartmentalizing the services we will also create a core set of common libraries that can be maintained in a private artifactory instance. Abstracting common functionality reduces the amount of repeated code which limits places a defect can occur and provides consistency across service modules.

## Details

The list of common libraries will grow as we begin to write code, but examples of common libraries could include items such as:

* Caching
* Currency Conversion
* Session lookup
* Permissions / Authorization

Much like the API / Services libraries, each common library will contain its own suite of unit tests that will automatically execute when built on the developer laptop as well as part of the automated build process [5.1.1]. In order to ensure

# Version Control / Continuous Integration / Build & Deploy Process

## Description

For the success of this project we will create a single stand-alone Google Cloud Project to control all aspects of versioning, testing, build and deployment. This standalone project would contain the following standalone Google Compute Engine services used to facilitate a devops pipeline:

* git
* Jenkins
* Jfrog Artifactory

While Cardinal will follow industry standards and best practices by making the process as flexible as possible we will only focus on building a process that best fits the Business View project solely. We are not attempting to build a one size fits all solution for Global Payments devops. Should another project outside of Business View be able to fit into the designed process with no change it can do so. Furthermore, should Global Payments develop a dev ops workflow separate from the proposed it may change the estimation.

### Source Control

Through the course of many projects Cardinal has settled on a git branching model [Figure 5] that is easy enough for novices, yet robust enough to handle the extreme conditions of a large-scale enterprise application. This paradigm can be easily extended to as many branches / environments as needed.

The following git workflow would apply to both front-end code/resources, shared common libraries, and service API’s.

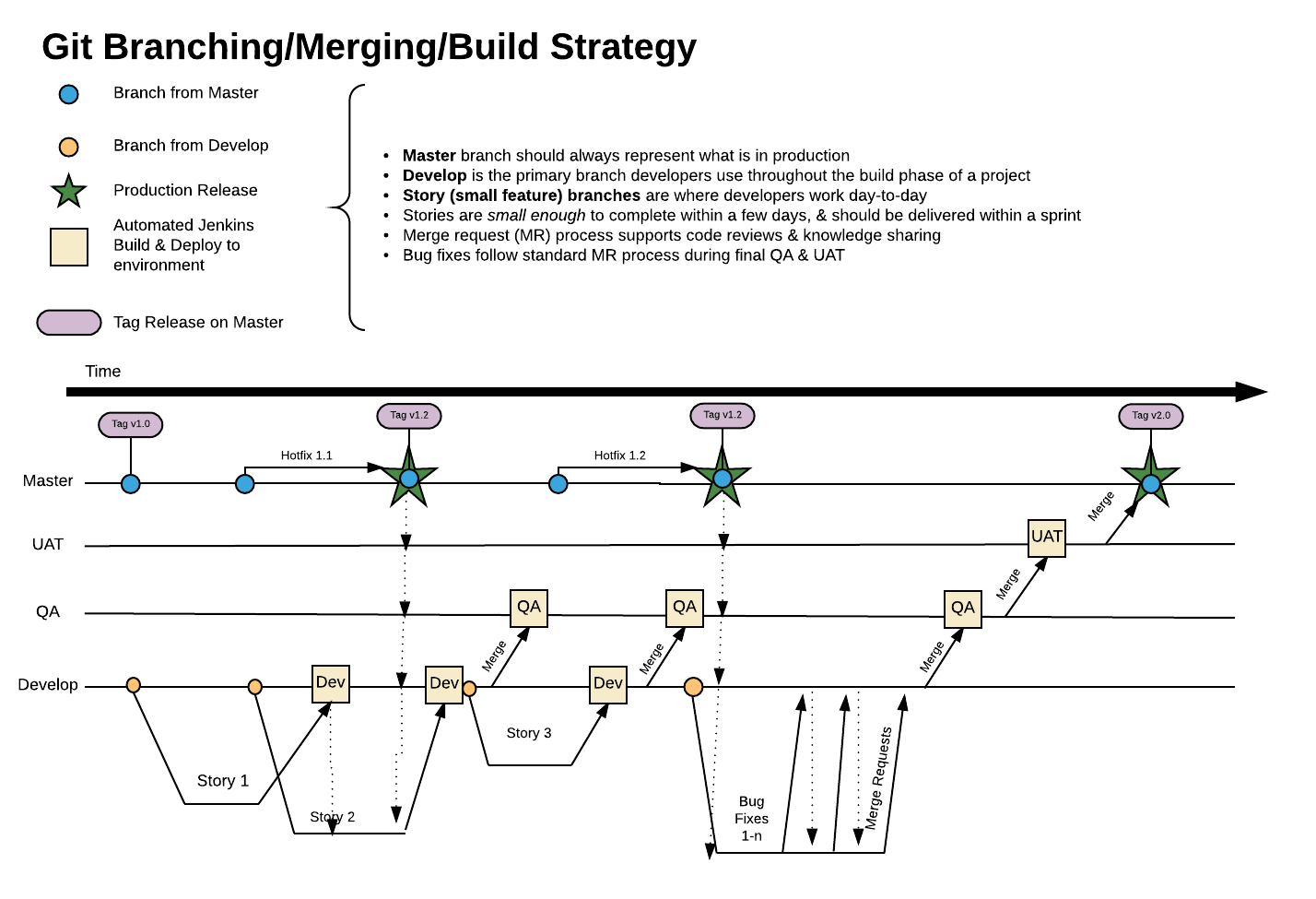


Figure : Git Branching Model

The git flow is as follows:

1. Each feature / story is a new branch off the develop branch
   1. Periodically the development branch should be merged into the feature branch to minimize future merge conflicts and keep the code up to date with the most recent accepted code.
2. As a story is completed a merge request is created
   1. Dev manager / team reviews code prior to merge
3. Feature is merged into the develop branch \*
4. When a feature(s) is ready for QA the development branch is merged into the QA branch \*
5. As the QA process progresses all bugs are fixed against a “bug fix” branch allowing for new concurrent development of new features and bug fixes
6. Once again bug fix branch is merged into Dev
7. Code is again merged into QA \*
8. Code is Merged to UAT \*
9. Code is merged to the Master branch \* where another new branch is created and tagged with a version number

If at any point a critical production bug is found we can:

1. Make the fix on the master branch \*
   1. Create a new branch from master and tag with version number
2. Merge code back through environments sequentially:
   1. UAT\*
   2. QA\*
   3. Dev\*
   4. Feature branch(es) \*

\* Denotes a trigger of the automated build and deploy process to the appropriate environment

## Automated Build and Deploy Process

Part of developing quality code is having a good continuous integration process with automated testing and code coverage checks. A simple, yet robust, approach to this is to use a Jenkins service running in a Google Cloud compute engine (VM) to monitor distinct branches in git. When a new commit arrives, the automated process will execute all unit tests [5.3], code coverage checks, and build the project. Upon successful passing of all the quality gates a combination of Jenkins plugins and *gcloud* scripts will execute to promote the code to the correct environment. [4.4, 4.5]

To prevent developers from deploying code manually to any environment we’ll utilize GCPs Identity and Access Management (IAM) to lock down most aspects of deployments and the platform as a whole. [4.3] The Jenkins server will have a service account associated to allow artifact releases and file copying while preventing developers direct access to change deployed code in any environment. [4.2]

Especially important for shared common libraries is a repository that can handle all versions so that dependent code can quickly access them. An industry standard for managing artifacts is artifactory – this tool creates a centralized, yet private, repository for all versions of libraries used.

To As mentioned in section <<where did I talk about traffic manager>> the code loaded to app engine will, by default, be done in “dark mode” allowing for manual testing and confirmation prior to switch some or all the traffic to use the newly deployed service.

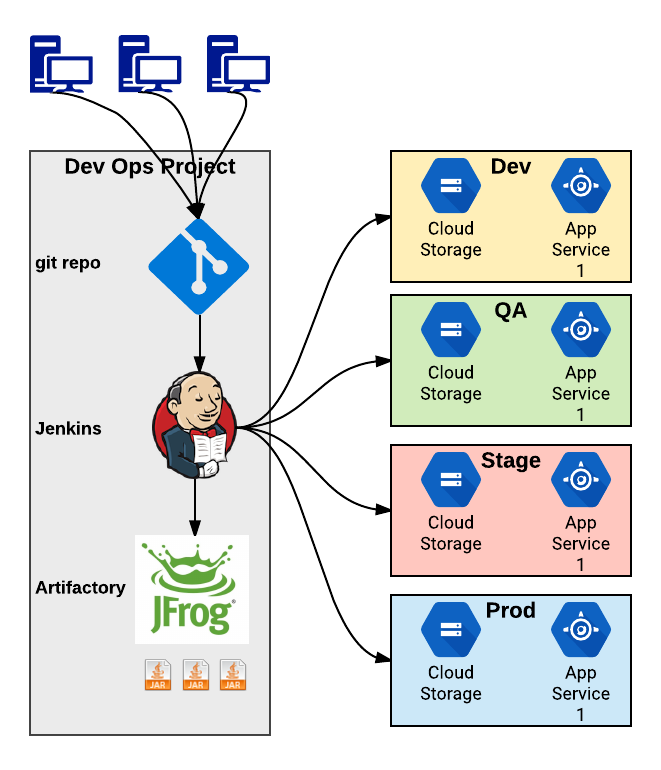


Figure : Overview of automated deploy process

## Service Versioning

Thanks to GAE’s service versioning capabilities we can deploy a new service version to a hidden staging area that can only be accessed via *hidden* URL that follows the pattern[[3]](#endnote-3):  
**https://**<<*version>>*-**dot**-<<*service>>*-**dot-<<***app-id>*.**appspot.com**

With a Google cloud load balancer utilizing layer 7 routing, coupled with the google cloud traffic manager we should be able to redirect requests in a manner that is seamless to the front end. This allows the deployment / devops team to manage traffic to the new endpoint prior to making them fully consumable to the outside world.

Migrating traffic from one version to another can be done through the Google Cloud Console or through a series of commands using the gcloud SDK. This traffic manager allows us to *hide* an API endpoint until a desired time, split traffic between multiple versions to facilitate blue/green testing, and allow for zero downtime deployments [4.8]. This versioning and traffic routing pattern also allows for easy rollback [4.7] to a previous version in real-time should something go wrong. Figure 7 below shows this environmental segregation as well as how the traffic manager can be used to route users between versions of each individual service.

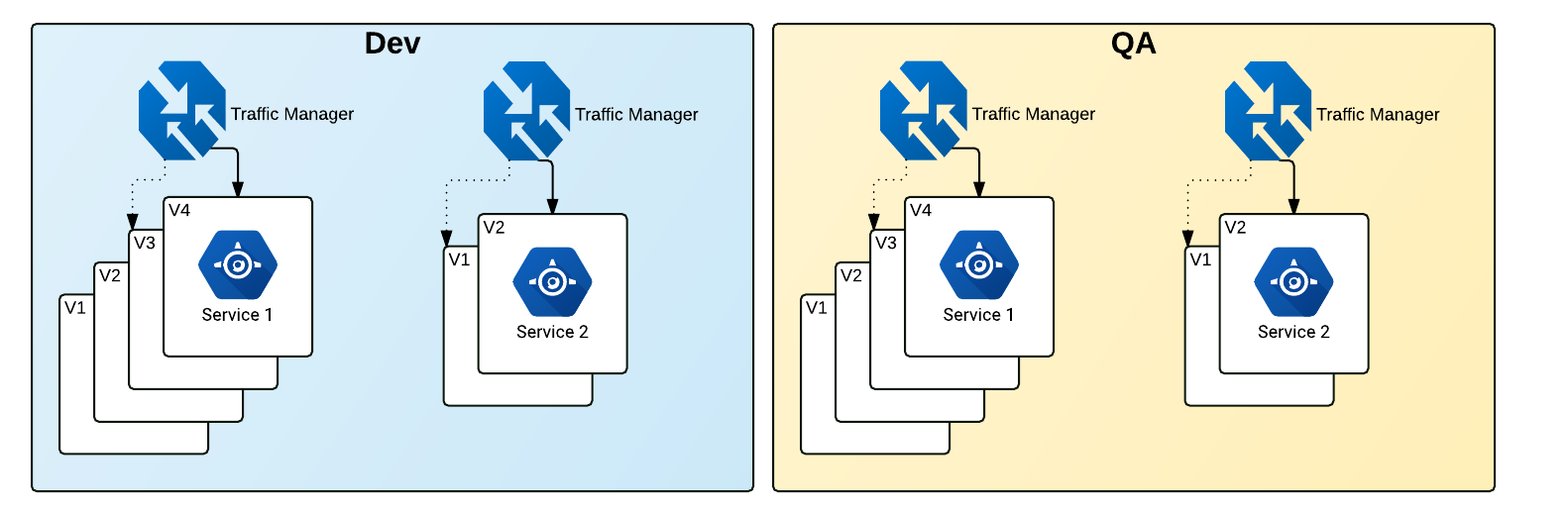


Figure : Environment segregation and traffic management

The traffic manager serves traffic to the live service through HTTPS [2.4] by default and can be switched to route traffic to multiple versions to provide a slow, well managed, rollout.

<<Assumption: Access to environments>>

<<Assumption: we are building dev ops for OUR project around best practices not for the company as a whole>>

# Testing

## Unit / Functional / Integration Tests

Each Service API and common library module will include a robust set of unit tests that will be executed as part of compilation on the developer’s machine as well as part of the automated build process through Jenkins. We plan to also implement a pre-commit hook in git that forces code to be compiled and tested prior to a git push to mitigate bad code making it into the repository

## Load Testing

Cardinal will create and automate load testing, but we’ll rely on Global Payments to provide metrics to be used as part of the test. Tests will be setup to run on demand or on schedule but will not happen on each check-in or build since scaling, even for a short amount of time repeatedly, would prove to be costly.

## Vulnerability Testing

Cardinal will test the services for any vulnerabilities as far as accessing endpoints or data without the proper authorization. The Cardinal team will stop short of doing a full penetration and network vulnerability tests as Global Payments has an internal team that will have to verify that it meets their guidelines.

# Data

## Description

Cardinal is targeting using a variety of different data storage options within google cloud. Each having its own purpose

## Data Diagram

## Deployment

# Bad Stuff

No ability to shutdown / turn off an entire project and resources without deleting it.

Multi-regional support requires manual setup and deployment – and out of the box GAE only has 99.95 SLA

No scripting of resources except for VMs

Looker does not elastically scale

# Appendix

# Business View Non-Functional Requirements

## General Cloud Architectural Requirements

* 1. High Availability with a 99.999% Uptime Target (or maximum available via cloud provider)
  2. Prefer Managed PaaS over IaaS
  3. Deployed to multiple Regions/Datacenters
  4. Load Balancing based on Geographic location for minimum latency
  5. Should leverage elastic auto-scale to minimize cloud cost and maximize availability and response time
  6. All dependent services (caching, messaging, etc.) should also be deployed with the same HA guidelines
  7. Configuration items (Service Endpoints, Connection Strings, App Settings) should be configurable by operations without the need to deploy code changes.
  8. Solution should not be coupled to a specific cloud vendor
     1. Should be able to deploy to any vendor or potentially all of them without coding changes

## Website Requirements

* 1. Initial Page Response time < 500ms
     1. All resources (css,js, images) optimized for speed and minimum latency
     2. Utilize CDNs when appropriate
     3. Minify and Bundle css and js when appropriate
  2. Content that requires > 500 MS should be loading via AJAX calls
     1. All content to load in <5000 MS
  3. Friendly URLS (i.e. no .jsp or .html)
  4. HTTPS via TLS 1.2
  5. Should work on all Modern Browsers (including phone, tablet)
     1. Specific Browser Versions TBD

## API Requirements

* 1. API design should be RESTful
  2. API should be transmitted over a secure protocol (TLS 1.2)
  3. API should have online documentation (i.e. Swagger)
  4. API should be CORS enabled
  5. Although exceptions can be made, APIs should follow the Microsoft REST API Guidelines (<https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md)>

## DevOps Requirements

* 1. Dev/QA/Staging/Prod Environments
  2. All Code Deployments must be automated from a Build Server or Release Pipeline Deployment Tool
  3. No Manual FTP or Command Line Deployments
  4. CI in the lower Environment if possible.
  5. Artifact Based Deployment throughout the Environments
  6. Build Once deploy many
  7. All Builds should be versioned and be able to rollback to a specific version on demand throughout the environments
  8. Production Deployments should be “Zero Down Time”
  9. Cloud Resources should be scriptable and deployable via the same automation tools (i.e. Azure ARM Templates or equivalent)

## QA/Testing Requirements

## Automated Testing is required at multiple levels

## Unit Tests for compiled Business Logic

## JavaScript automated Tests for UI/UX Logic

## API Test Automation

* 1. Automated Load Testing (Specific Throughput targets TBD)
  2. Automatic Testing should be part of the automated release process

## Logging Requirements

* 1. Application should log at several levels
     1. Security Events
     2. Applications Errors/Warnings
     3. Debug/Trace Events
  2. Logging Level should be configurable at runtime
  3. Custom Metrics as applicable should be created to monitor system performance and utilization (Metrics to be determined in the Design Phase)

## High Level Coding Guidelines

* 1. Code should be structured in a way to maximize maintainability
     1. Utilization of SOLID Design Principles
     2. Custom Javascript written in Typescript if possible

1. <https://cloud.google.com/appengine/sla> [↑](#endnote-ref-1)
2. <https://cloud.google.com/appengine/docs/standard/java/runtime> [↑](#endnote-ref-2)
3. <https://cloud.google.com/appengine/docs/standard/python/how-requests-are-routed> [↑](#endnote-ref-3)