|  |  |
| --- | --- |
|  |  |
|  | Solution Design Document  <Title> |
|  |  |

Submitter Information

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | WIN ID | Name | Project | Role |
| 1. |  |  |  |  |
| 2. |  |  |  |  |
| 3. |  |  |  |  |
| 4. |  |  |  |  |

Table of Contents

[1. Problem Statement 4](#_Toc39738531)

[1.1. Objectives and Goals 4](#_Toc39738532)

[1.2. Acronyms and Definitions 4](#_Toc39738533)

[2. Architectural Requirements 5](#_Toc39738534)

[3. System Context 6](#_Toc39738535)

[3.1. System Context Diagram 6](#_Toc39738536)

[3.2. Interactions and Interfaces Descriptions 6](#_Toc39738537)

[4. Business Services 7](#_Toc39738538)

[4.1. Service Overview and Detail 7](#_Toc39738539)

[5. Logical Architecture 8](#_Toc39738540)

[5.1. Logical Architecture Overview 8](#_Toc39738541)

[6. Assumptions/Limitations/Constraints 9](#_Toc39738542)

[7. Appendix 10](#_Toc39738543)

# Problem Statement

This document describes the specific solution architecture of the [project] system for a particular solution/client, including architecture design principles and guidelines, and lays the foundation for design and construction the system.

## Objectives and Goals

<Typically, build a bulleted list of goals/objectives this document seeks to meet. The ideal length is 3 – 7 bullets. List architectural goals based on patterns within MicroServices.>

## Acronyms and Definitions

The tables below include acronyms, terms, and definitions specifically applicable to this document. The content must be consistent with the project-wide glossary located <in the project’s SharePoint site, or other location>.

<Include all acronyms used in the document. Most acronyms need terms but not definitions. Only add definitions that would be helpful to the reader. If a term has both an acronym and a definition, list it in both tables, and provide the acronym in parentheses next to the term name in the Definitions table.>

Acronyms

|  |  |
| --- | --- |
| Acronym | Term |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Architectural Requirements

< Detail out the MicroService design paterns proposed

Platform/Technolgy independent/Polyglot

Independent Deployement mode

Scalability/High Availability

Performance levels

Security

Maintainability

Monitoring

Fault Tolerance

Agility ,

Serverless

>

**Scalability/High Availability**

In order to build a scalable application, we need to design for concurrency and partitioning: concurrency allows each task to be broken up into smaller pieces, while partitioning is essential for allowing these smaller pieces to be processed in parallel. So, while scalability is related to how we divide and conquer the processing of tasks, performance is the measure of how efficiently the application processes those tasks.

In simplest terms, highly available architectures typically involve distribution of compute resources, load balancing, and replication of data

High Availability is feature you provide to your Service API. There are some fundamental principles/pattern which will ensure high availability.

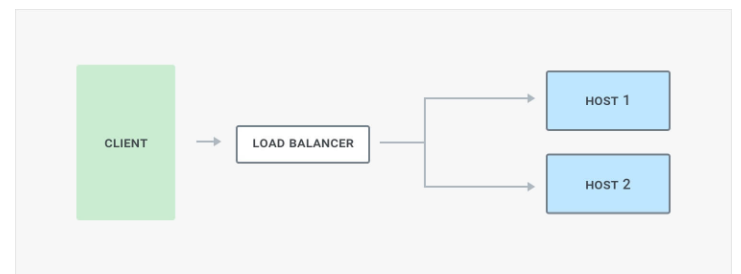
1. Circuit Breaker
2. Retry
3. Health Endpoint Monitoring
4. Throttling
5. Queue-Based Load Leveling pattern

Introduction To Load Balancer

Load balancing is the practice of distributing client request load across multiple application instances for improved performance. Load balancing distributes requests among healthy hosts so no single host gets overloaded.

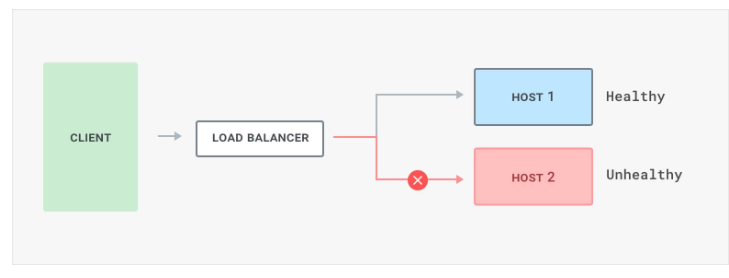
A typical load balancing architecture showing that clients make requests to a load balancer, which then passes (or proxies) requests to the upstream hosts. Clients can be a real person or a service calling another service, and they can be external or internal to your company.

Below Is Round Robin Load Balancer



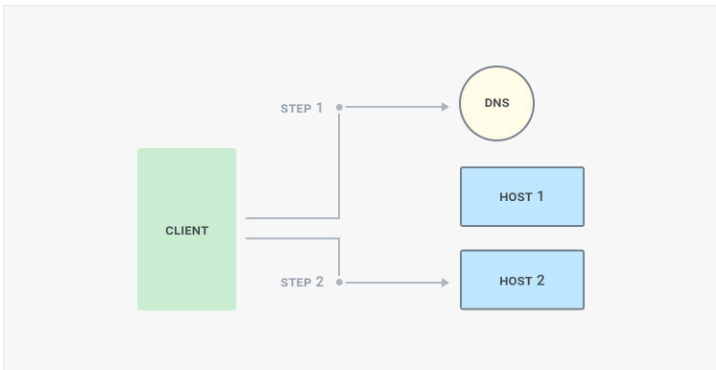
Circuit Breaker

When you know that a given host is unhealthy, its best to “break the circuit” so that traffic flows to healthy hosts instead. This provides a better experience for end-users because they will encounter fewer errors and timeouts. It’s also better for your host because diverting traffic will prevent it from being overloaded, and give it a chance to recover. It may have too many requests to handle, the process or container may need to be restarted, or your team may need to investigate.



Load Balancer Types

1.DNS (Domain Name Server) load Balancing



2.Round Robin Load balancing

3.Ring Balancer

Optional way

[Kong](https://getkong.org/) is the most popular open source API gateway for microservices. It’s very fast with sub-millisecond latency, runs on any infrastructure, and is built on top of reliable technologies like NGINX. It has a rich plug-in ecosystem that allows it to offer many capabilities including rate limiting, access control and more.

Performance

1. Provide batch APIs

In addition to CRUD-style APIs, you can still provide good microservice performance for groups of entities by providing batch APIs. For example, rather than only exposing a GET API method that retrieves a single user, provide an API that takes a set of user IDs and returns a dictionary of corresponding users:

**Request:**

/user-service/v1/?**userId=**ABC123&**userId=**DEF456&**userId=**GHI789

**Response:**

{

"ABC123": {

"userId": "ABC123",

"firstName": "Jake",

… },

"DEF456": {

"userId": "DEF456",

"firstName": "Sue",

… },

"GHI789": {

"userId": "GHI789",

"firstName": "Ted",

… }

}

The App Engine SDK supports many batch APIs, such as the ability to fetch many entities from Cloud Datastore through a single RPC, so servicing these types of batch APIs can be very efficient.

1. Use asynchronous requests

Often, you will need to interact with many microservices to compose a response. For example, you might need to fetch the logged-in user's preferences as well as their company details. Frequently, these pieces of information are not dependent on one another and you could fetch them in parallel. The Urlfetc

App Engine SDK supports asynchronous requests, allowing you to call microservices in parallel

3.Use Shortest Route

Depending on how you invoke Urlfetch, you can cause different infrastructure and routes to be used. In order to use the best-performing route, consider the following recommendations:

**Use**[***REGION\_ID***](https://cloud.google.com/appengine/docs/standard/java/microservice-performance#appengine-urls).r.appspot.com**, not a custom domain**

**Set**follow redirects**to**False

4.Prefer services within a project over multiple projects

There are good reasons to use multiple projects when building a microservices-based application, but if performance is your primary goal, use services within a single project. Services of a project are hosted in the same datacenter and even though throughput on Google's inter-datacenter network is excellent, local calls are faster.

Security

1.Use OAuth for user identity and access control

2.Use 'defence in depth' to prioritize key services

the Defense in Depth mechanism is basically a technique through which you can apply layers of security countermeasures to protect the sensitive services. So, as a developer, you just have to identify the services with the most sensitive information and then apply a number of security layers to protect them. In this way, you can make sure that any potential attacker cannot crack the security on a single go, and has to go forward and try to crack the defense mechanism of all the layers.

3.Use automatic security updates

4. Use a distributed firewall with centralized control.

5. Get your containers out of the public network

6. Use security scanners for your containers.

**Maintainability**

Maintainability is all about how easy it is to fix a bug found in the application.we call maintainability is typically two things•

Maintainability

Extensibility

#### Maintainability

Maintainability is all about how easy it is to fix a bug found in the application.

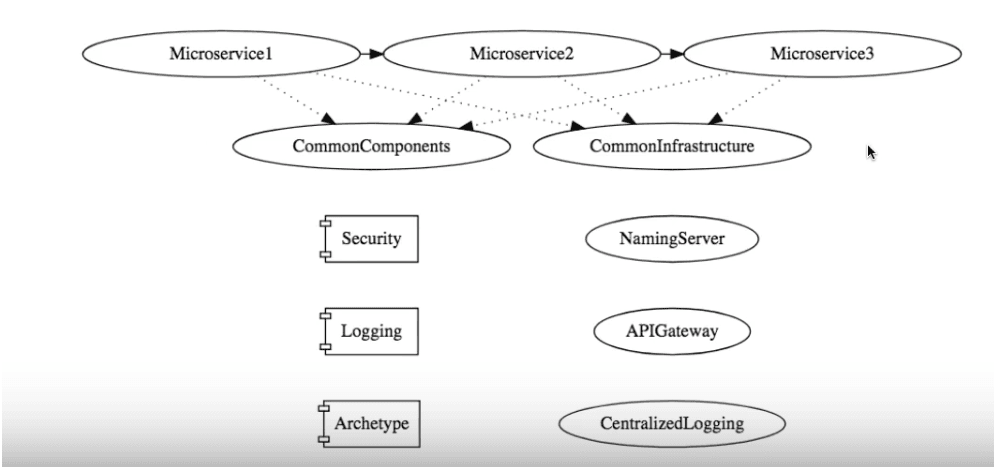
#### Extensibility

Extensibility of a system describes how easy it is to incorporate new features into it.

Both these concepts are quite deeply related, and are clubbed together under a single umbrella - maintainability. That’s how we address them as well

Adhering to 4 Principles of Simple Design is a great starting point.

1.Ensure Modular Design

****

2.Ensure Great Automation Tests

#### 3. Automate The Deployment Pipeline

With an automated deployment pipeline, the build moves through different stages such as development, QA, staging and production in a systematic and efficient manner.

#### 4. Greater Visibility

Greater visibility of code across the modules in the system makes it easier to detect bugs, and shorten the time line in addressing them.

**Monitoring**

Monitoring systems over time produces valuable data that can be used to improve service performance. Failure and performance data can be analyzed to look for patterns in system failures, which can be correlated with events. For example, consider a case

where data indicates 25 percent of total system failures occur within an hour of a new deployment. This would thus be a strong indicator that deployment processes need attention and improvement.

Monitoring technology and tools come in two broad categories: libraries and platforms. Some tools include both, providing a platform for collection and a library to instrument code.

Examples include open-source libraries like [**AppMetrics**](https://www.app-metrics.io/) for .NET and [**SPF4J**](http://www.spf4j.org/) for Java.

* [**Zipkin**](https://zipkin.io/): Zipkin is an open-source tracing system designed specifically to trace calls between microservices. It is especially useful for analyzing latency problems. Zipkin includes both instrumentation libraries and the collector processes that gather and store tracing data.
* [**Apache Kafka**](https://kafka.apache.org/): Kafka is a streams-processing system. It uses a “publish/subscribe” methodology for reading and writing data to a logical “stream,” which is similar in concept to a messaging system such as **RabbitMQ**. **Kafka** can be combined with other tools such as Zipkin to provide a robust solution for transmitting and storing metrics data.

## Conclusion

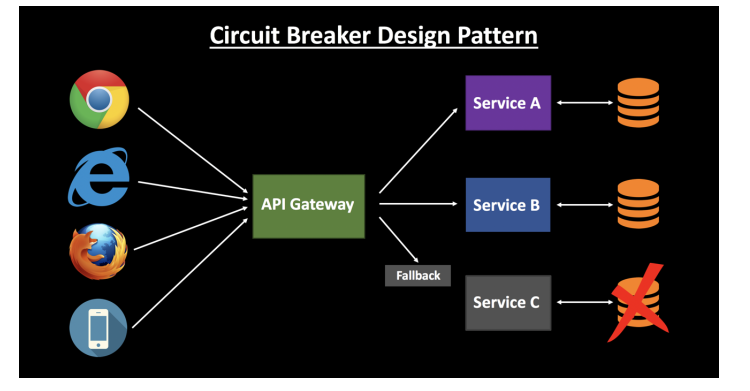
Monitoring requirements should be considered from the very beginning of an application’s lifecycle. Systems monitoring requires contributions from both development and operations.

**Fault Tolerance**

Design Patterns to Ensure Service Resiliency

### ****Circuit Breaker Pattern****

If there are failures in your microservices ecosystem, then you need to fail fast by opening the circuit. This ensures that no additional calls are made to the failing service, once the circuit breaker is open. So we return an exception immediately. This pattern also monitor the system for failures and once things are back to normal, the circuit is closed to allow normal functionality



This is a very common pattern to avoid cascading failure in your microservice ecosystem. You can use some popular third-party libraries to implement circuit breaking in your application, such as Polly and Hystrix.

### ****Retry Design Pattern****

This pattern states that you can retry a connection automatically which has failed earlier due to an exception. This is very handy in case of temporary issues with one of your services. A lot of times a simple retry might fix the issue. The load balancer might point you to a different healthy server on the retry, and your call might be a success.

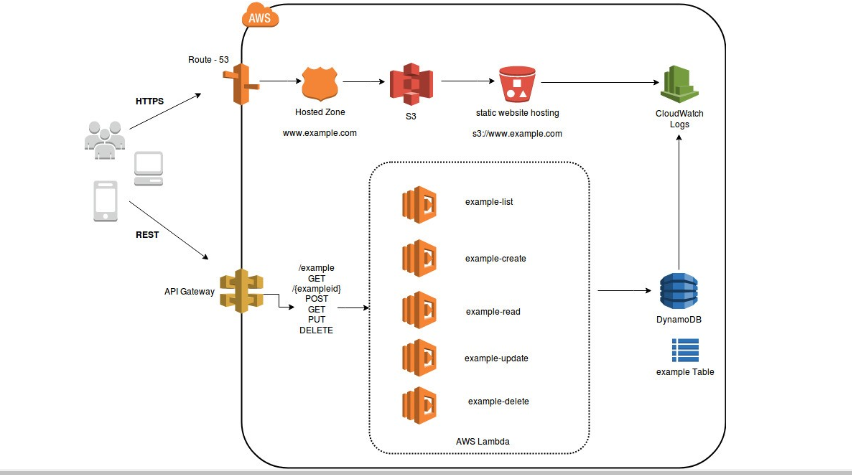
### ****Timeout Design Pattern****

This pattern states that you should not wait for a service response for an indefinite amount of time — throw an exception instead of waiting too long. This will ensure that you are not stuck in a state of limbo, continuing to consume application resources. Once the timeout period is met, the thread is freed up.

**Serverless**

The diagram below shows the server less micro services architecture where the complete solution is built without managing any server. This also eliminates the operational efforts of running and monitoring the servers.

Lambda will handle everything required to run and scale the execution to meet actual demand with high availability. Lambda supports several programming languages and it can be called directly from any web or mobile applications.



In the architecture diagram, Lambda is integrated with API Gateway. Synchronous calls from API gateway to AWS Lambda enables the application to operate as serverless. AWS Lambda will store all the data in a fully managed NoSQL database called DynamoDB and all the static data will be stored in S3 Bucket.

It can be said that microservices architecture is designed to overcome the challenges of traditional monolithic architectures seen in enterprise applications. It allows collaboration between operations and development teams of any organization leading to DevOps and is a preferred choice nowadays.

AWS offers multiple managed services that can help engineers build microservice architectures and minimize architectural and operational complexity.

## ***Challenges of Microservices Architecture***

**Quick Setup needed** : You cannot spend a month setting up each microservice. You should be able to create microservices quickly.

**Automation** : Because there are a number of smaller components instead of a monolith, you need to automate everything - Builds, Deployment, Monitoring etc.

**Visibility** : You now have a number of smaller components to deploy and maintain. Maybe 100 or maybe 1000 components. You should be able to monitor and identify problems automatically. You need great visibility around all the components.

**Bounded Context** : Deciding the boundaries of a microservice is not an easy task. Bounded Contexts from Domain Driven Design is a good starting point. Your understanding of the domain evolves over a period of time. You need to ensure that the microservice boundaries evolve.

**Configuration Management** : You need to maintain configurations for hundreds of components across environments. You would need a Configuration Management solution

**Dynamic Scale Up and Scale Down** : The advantages of microservices will only be realized if your applications can scaled up and down easily in the cloud.

**Pack of Cards** : If a microservice at the bottom of the call chain fails, it can have knock on effects on all other microservices. Microservices should be fault tolerant by Design.

**Debugging** : When there is a problem that needs investigation, you might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.

**Consistency** : You cannot have a wide range of tools solving the same problem. While it is important to foster innovation, it is also important to have some decentralized governance around the languages, platforms, technology and tools used for implementing/deploying/monitoring microservices.

**Solutions to Challenges with Microservice Architectures**

**Spring Boot**

Provide non-functional features

* embedded servers (easy deployment with containers)
* metrics (monitoring)
* health checks (monitoring)
* externalized configuration

**Spring Cloud**

Spring Cloud provides solutions to cloud enable your microservices. It leverages and builds on top of some of the Cloud solutions opensourced by Netflix (Netflix OSS)

# System Context

## System Context Diagram

<Insert a picture of the system context diagram. The diagram is at the system level and does not detail every interface that may come from one external entity. For example, there may be ten different interfaces from one of the state systems, but the state system appears once on the diagram.>

## Interactions and Interfaces Descriptions

<Describe the pattern that you have for interface descriptions. By default, show what Interface Control Documents govern each of the interfaces that appeared in the system context diagram. Below are the descriptions for the interactions and interfaces between the system under development and the external entities:

# Business Services

<According to ITIL (v3), a service is “a means of delivering value to Customers by facilitating Outcomes that Customers want to achieve without the ownership of specific Costs and Risks.”

This section describes how the solution architecture provides business services.

The Enterprise Architecture is comprised of several business services. Each service is depicted with workflow diagrams that show a collection of interactions (via the interfaces in the previous section) and the back-end business process steps that occur. >

## Service Overview and Detail

<Describe Business Service steps and which/how each subsystem described in the logical architecture responds for each including the events it will trigger and the responses. A workflow or activity diagram are an excellent mechanism for describing the service steps and is highly recommended as an amendment to a narrative text.>

# Logical Architecture

## Logical Architecture Overview

<Provide one or two logical architecture overview diagrams (AOD). Describe the logical function of the system given the context diagram

Some types of AODs are:

* Logical subsystems – the diagram calls out the major subsystems and datastores showing their relationship to each other and to external systems.
* Business subsystems – the diagram shows business functions and how they interact. The idea is to provide a view of the system as it relates to the business units and their use of services.

# Assumptions/Limitations/Constraints

List out the assumptions you have made to design the solution

List out what you feel where the constraints/limitations you had while designing this solution.

# Appendix