
# Microservice Design and Architecture

Priyanka Vergadia
Developer Advocate, Google Cloud

Hi I'm Priyanka Vergadia, a developer advocate for Google Cloud. In this module, we introduce application architecture and microservice design.

# Learning objectives

- Decompose monolithic applications into microservices.
- Recognize appropriate microservice boundaries.
- Architect stateful and stateless services to optimize scalability and reliability.
- Implement services using 12-factor best practices.
- Build loosely coupled services by implementing a well-designed REST architecture.
- Design consistent, standard RESTful service APIs.

Specifically, you will learn about microservice architectures and how to decompose monolithic applications into microservices. The benefits of a microservice architecture for cloud-native applications is discussed and contrasted with a monolith.

The challenges of decomposing applications into microservices with clear boundaries to support independently deployable units are investigated.

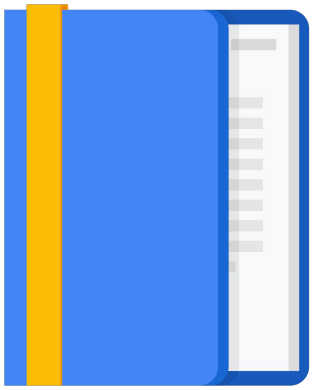
You will learn how to architect for some of the major technical challenges of microservice architectures such as state management, reliability, and scalability.

Once a cloud native microservice architecture has been chosen, the best practices for development and deployment are introduced based around the widely recognized 12-factor best practices.

At the end of the module, we'll go over a core component of a microservice architecture, which is the design of consistent, loosely coupled service interfaces.

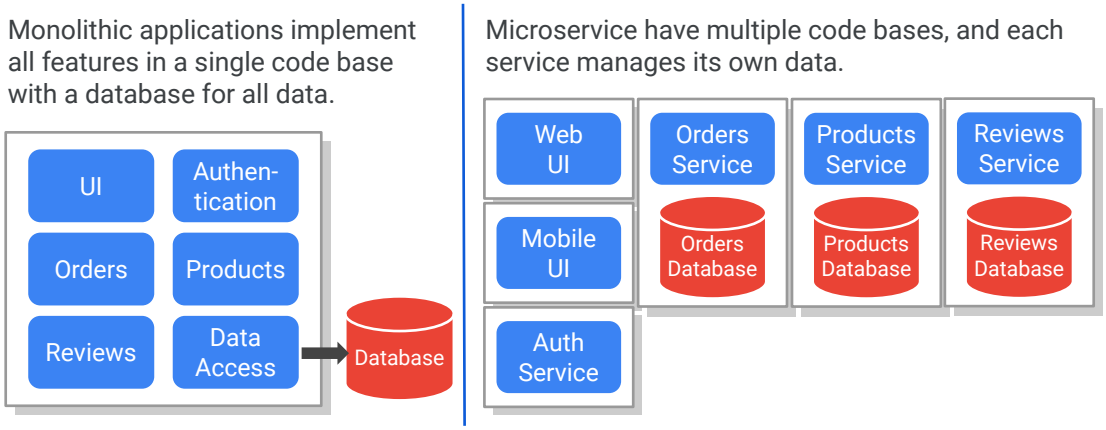
# Agenda

- Microservices
- Microservice Best Practices
- REST
- APIs



Let’s begin by looking at microservices in more detail.

## Microservices divide a large program into multiple smaller, independent services




Microservices divide a large program into a number of smaller, independent services, as shown on the right, unlike a monolithic application, which implements all features in a single code base with a database for all data, as shown on the left.

Microservices are the current industry trend; however, it's important to ensure that there is a good reason to select this architecture. The primary reason is to enable teams to work independently and deliver through to production at their own cadence. This supports scaling the organization: adding more teams increases speed. There is also the additional benefit of being able to scale the microservices independently based on their requirements.

Architecturally, an application designed as a monolith or around microservices should be composed of modular components with clearly defined boundaries. With a monolith, all the components are packaged at deployment time and deployed together. With microservices, the individual components are deployable. Google Cloud provides several compute services that facilitate deploying microservices. These include App Engine, Cloud Run, GKE, and Cloud Functions. Each offers different levels of granularity and control and will be discussed later in the course.

To achieve independence on services, each service should have its own datastore. This lets the best datastore solution for that service be selected and also keeps the services independent. We do not want to introduce coupling between services through a datastore.

## Pros and cons of microservice architectures...

- Easier to develop and maintain
- Reduced risk when deploying new versions
- Services scale independently to optimize use of infrastructure
- Faster to innovate and add new features
- Can use different languages and frameworks for different services
- Choose the runtime appropriate to each service

- Increased complexity when communicating between services
- Increased latency across service boundaries
- Concerns about securing inter-service traffic
- Multiple deployments
- Need to ensure that you don't break clients as versions change
- Must maintain backward compatibility when clients as the microservice evolves

A properly designed microservice architecture can help achieve the following goals:

- Define strong contracts between the various microservices
- Allow for independent deployment cycles, including rollback
- Facilitate concurrent, A/B release testing on subsystems
- Minimize test automation and quality assurance overhead
- Improve clarity of logging and monitoring
- Provide fine-grained cost accounting
- Increase overall application scalability and reliability through scaling smaller units

However, the advantages must be balanced with the challenges this architectural style introduces. Some of these challenges include:

- It can be difficult to define clear boundaries between services to support independent development and deployment
- Increased complexity of infrastructure, with distributed services having more points of failure
- The increased latency introduced by network services and the need to build in resilience to handle possible failures and delays
- Due to the networking involved, there is a need to provide security for service-to-service communication, which increases complexity of infrastructure
- Strong requirement to manage and version service interfaces. With independently deployable services, the need to maintain backward compatibility increases.

## The key to architecting microservice applications is recognizing service boundaries

| Decompose applications by feature to minimize dependencies | Organize services by architectural layer | Isolate services that provide shared functionality |
| --- | --- | --- |
| • Reviews service<br>• Orders service<br>• Products service<br>• Etc. | • Web, Android, and iOS user interfaces<br>• Data access services | • Authentication service<br>• Reporting service<br>• Etc. |

Now, decomposing applications into microservices is one of the biggest technical challenges of application design. Here techniques like domain-driven design are extremely useful in identifying logical functional groupings.

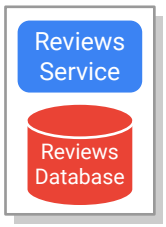
The first step is to decompose the application by feature or functional groupings to minimize dependencies. Consider, for example, an online retail application. Logical functional groupings could be product management, reviews, accounts, and orders. These groupings then form mini applications which expose an API. Each of these mini applications will be implemented by potentially multiple microservices internally. Internally, these microservices are then organized by architectural layer, and each should be independently deployable and scalable.

Any analysis will also identify shared services, such as authentication, which are then isolated and deployed separately from the mini applications.

## Stateful services have different challenges than stateless ones

Stateful services manage stored data over time

- Harder to scale
- Harder to upgrade
- Need to back up




Stateless services get their data from the environment or other stateful services

- Easy to scale by adding instances
- Easy to migrate to new versions
- Easy to administer

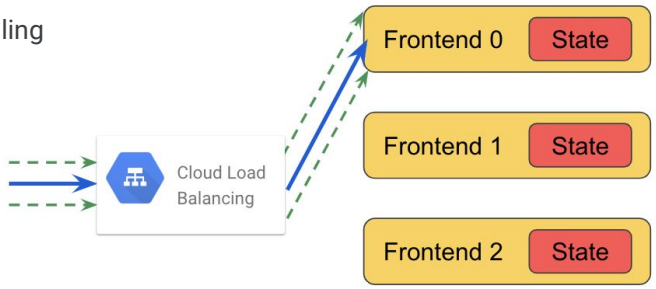

When you're designing microservices, services that do not maintain state but obtain their state from the environment or stateful services are easier to manage. That is, they are easy to scale, to administer, and to migrate to new versions because of their lack of state.

However, it is generally not possible to avoid using stateless services at some point in a microservice-based application. It is therefore important to understand the implications of having stateful services on the architecture of the system. These include introducing significant challenges in the ability to scale and upgrade the services.

Being aware of how state will be managed is important in the very early stages of microservice application design. Let me introduce some suggestions and best practices on how this can be achieved.

## Avoid storing shared state in-memory on your servers

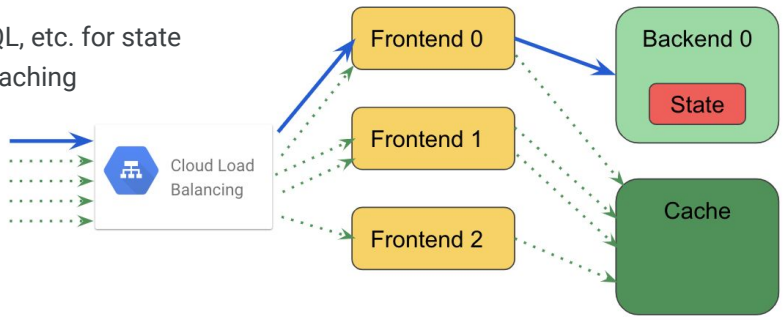
- Requires sticky sessions to be set up in the load balancer
- Hinders elastic autoscaling



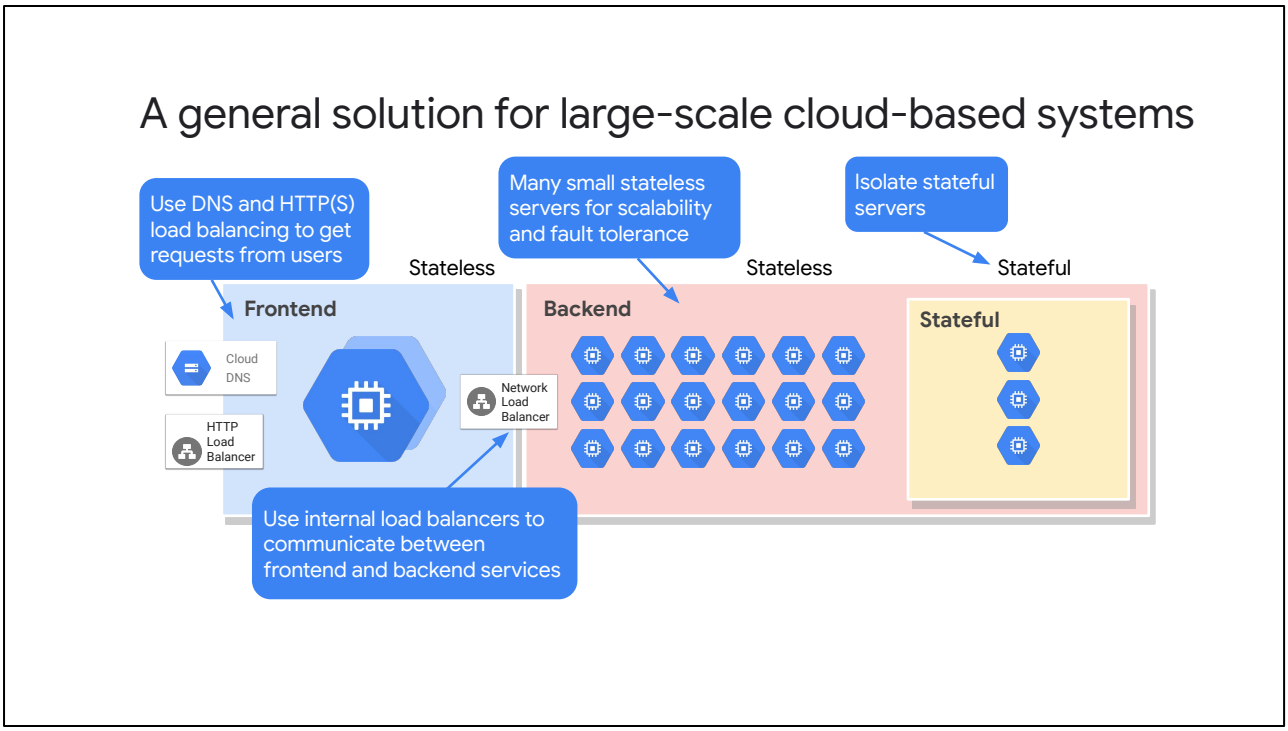

In memory, shared state has implications that impact and negate many of the benefits of a microservice architecture. The auto scaling potential of individual microservices is hindered because subsequent client requests have to be sent to the same server that the initial request was made to. In addition, this requires configuration of the load balancers to use sticky sessions. which in Google Cloud is referred to as session affinity.

## Store state using backend storage services shared by the frontend server

- Cache state data for faster access
- Take advantage of Google Cloud–managed data services
  - Firestore, Cloud SQL, etc. for state
  - Memorystore for caching




A recognized best practice for designing stateful services is to use backend storage services that are shared by frontend stateless services. For example, for persistent state, the Google Cloud-managed data services such as Firestore or Cloud SQL may be suitable. Then to improve the speed of data access, the data can be cached. Memorystore, which is a highly available Redis-based service, is ideal for this.

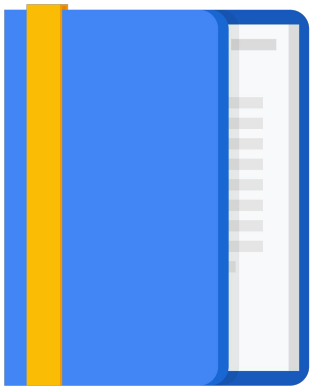



This diagram displays a general solution that shows the separation of the frontend and backend processing stages. A load balancer distributes the load between the backend and frontend services. This allows the backend to scale if it needs to keep up with the demand from the frontend. In addition, the stateful servers/services are also isolated. The stateful services can make use of persistent storage services and caching as previously discussed.

This layout allows a large part of the application to make use of the scalability and fault tolerance of Google Cloud services as stateless services. By isolation of the stateful servers and services, the challenges of scaling and upgrading are limited to a subset of the overall set of services.

# Agenda

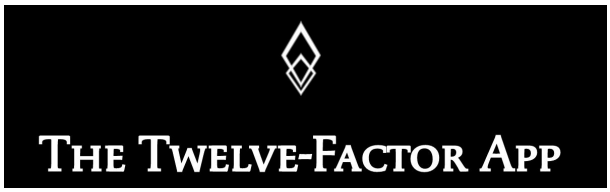
- Microservices
- **Microservice Best Practices**
- REST
- APIs



Let’s discuss microservice best practices.

The 12-factor app is a set of best practices for building web or software-as-a-service applications

- Maximize portability
- Deploy to the cloud
- Enable continuous deployment
- Scale easily




The 12-factor app is a set of best practices for building web or software-as-a-service applications. Twelve-factor design helps you decouple components of the application so that each component can be deployed to the cloud using continuous deployment and scaled up or down seamlessly. The design principles also help maximize portability to different environments. Because the factors are independent of any programming language or software stack, 12-factor design can be applied to a wide variety of applications. Let’s take a look at these best practices.

## The 12 factors

| | |
|---|---|
| 1. **Codebase**<br>One codebase tracked in revision control, many deploys | • Use a version control system like Git.<br>• Each app has one code repo and vice versa. |
| 2. **Dependencies**<br>Explicitly declare and isolate dependencies | • Use a package manager like Maven, Pip, NPM to install dependencies.<br>• Declare dependencies in your code base. |
| 3. **Config**<br>Store config in the environment | • Don't put secrets, connection strings, endpoints, etc., in source code.<br>• Store those as environment variables. |
| 4. **Backing services**<br>Treat backing services as attached resources | • Databases, caches, queues, and other services are accessed via URLs.<br>• Should be easy to swap one implementation for another. |

The first factor is codebase. The codebase should be tracked in a version control such as Git. Cloud Source Repositories provides fully featured private repositories.

The second factor is dependencies. There are two main considerations when it comes to dependencies for 12-factor apps: dependency declaration and dependency isolation. Dependencies should be declared explicitly and stored in version control. Dependency tracking is performed by language-specific tools such as Maven for Java and Pip for Python. An app and its dependencies can be isolated by packaging them into a container. Container Registry can be used to store the images and provide fine-grained access control.

The third factor is configuration. Every application has a configuration for different environments like test, production, and development. This configuration should be external to the code and usually kept in environment variables for deployment flexibility.

The fourth factor is backing services. Every backing service such as a database, cache, or message service, should be accessed via URLs and set by configuration. The backing services act as abstractions for the underlying resource. The aim is to be able to swap one backing service for a different implementation easily.

## The 12 factors (continued)

| | |
|---|---|
| 5. **Build, release, run**<br>Strictly separate build and run stages | • Build creates a deployment package from the source code.<br>• Release combines the deployment with configuration in the runtime environment.<br>• Run executes the application. |
| 6. **Processes**<br>Execute the app as one or more stateless processes | • Apps run in one or more processes.<br>• Each instance of the app gets its data from a separate database service. |
| 7. **Port binding**<br>Export services via port binding | • Apps are self-contained and expose a port and protocol internally.<br>• Apps are not injected into a separate server like Apache. |
| 8. **Concurrency**<br>Scale out via the process model | • Because apps are self-contained and run in separate process, they scale easily by adding instances. |

The 5th factor is build, release, run**.** The software deployment process should be broken into three distinct stages: build, release, and run. Each stage should result in an artifact that's uniquely identifiable. Build will create a deployment package from the source code. Every deployment package should be linked to a specific release that's a result of combining a runtime environment configuration with a build. This allows easy rollbacks and a visible audit trail of the history of every production deployment. The run stage then simply executes the application.

The 6th factor is processes. Applications run as one or more stateless processes. If state is required, the technique discussed earlier in this module for state management should be used. For instance, each service should have its own datastore and caches using, for example, Memorystore to cache and share common data between services used.

The 7th factor is port binding. Services should be exposed using a port number. The applications bundle the web server as part of the application and do not require a separate server like Apache. In Google Cloud, such apps can be deployed on platform services such as Compute Engine, GKE, App Engine, or Cloud Run.

The 8th factor is concurrency. The application should be able to scale out by starting new processes and scale back in as needed to meet demand/load**.**

## The 12 factors (continued)

| | |
|---|---|
| 9. **Disposability**<br>Maximize robustness with fast startup and graceful shutdown | • App instances should scale quickly when needed.<br>• If an instance is not needed, you should be able to turn it off with no side effects. |
| 10. **Dev/prod parity**<br>Keep development, staging, and production as similar as possible | • Container systems like Docker makes this easier.<br>• Leverage infrastructure as code to make environments easy to create. |
| 11. **Logs**<br>Treat logs as event streams | • Write log messages to standard output and aggregate all logs to a single source. |
| 12. **Admin processes**<br>Run admin/management tasks as one-off processes | • Admin tasks should be repeatable processes, not one-off manual tasks.<br>• Admin tasks shouldn't be a part of the application. |

The 9th factor is **disposability.** Applications should be written to be more reliable than the underlying infrastructure they run on. This means they should be able to handle temporary failures in the underlying infrastructure and gracefully shut down and restart quickly. Applications should also be able to scale up and down quickly, acquiring and releasing resources as needed.

The 10th factor is **Dev/production parity.** The aim should be to have the same environments used in development and test/staging as are used in production. Infrastructure as code and Docker containers make this easier. Environments can be rapidly and consistent provisioned and configured via environment variables. Google Cloud provides several tools that can be used to build workflows that keep the environments consistent. These tools include Cloud Source Repositories, Cloud Storage, Container Registry, and Cloud Deployment Manager. Deployment Manager allows the writing of templates to create deployments using Google Cloud services.

The 11th factor is **Logs.** Logs provide an awareness of the health of your apps. It's important to decouple the collection, processing, and analysis of logs from the core logic of your apps. Logging should be to standard output and aggregating into a single source. This is particularly useful when your apps require dynamic scaling and are running on public clouds because it eliminates the overhead of managing the storage location for logs and the aggregation from distributed (and often ephemeral) VMs or containers. Google Cloud offers a suite of tools that help with the collection, processing, and structured analysis of logs.

The 12th factor is **Admin processes,** These are usually one-off processes and should be decoupled from the application. These should be automated and repeatable, not manual, processes. Depending on your deployment on Google Cloud, there are many options for this, including: cron jobs in GKE, cloud tasks on App Engine, and Cloud Scheduler.

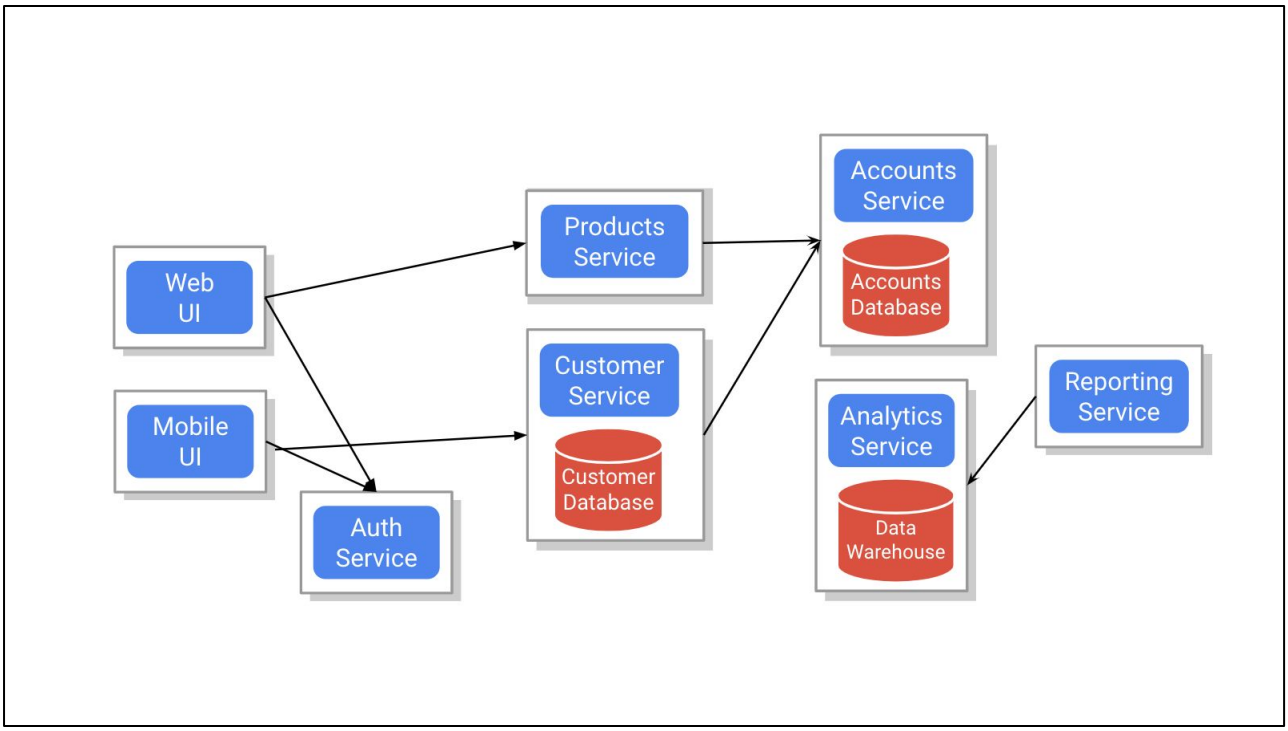
## Activity 4: Designing microservices for your application

Refer to your Design and Process Workbook.

- Diagram the microservices required by your case-study application.

In this design activity, you are going to work on activity 4 of the design workbook.

There you will design microservices for your application. The primary aim is to diagram the microservices required by your case study application. Some of the things to consider are the microservice boundaries and state management as well as common services. Use the principles we have discussed in this module so far.




Here's an example diagram for microservices for the website and the mobile phone application of an online banking service.

Draw a diagram similar to the one shown for your case study.

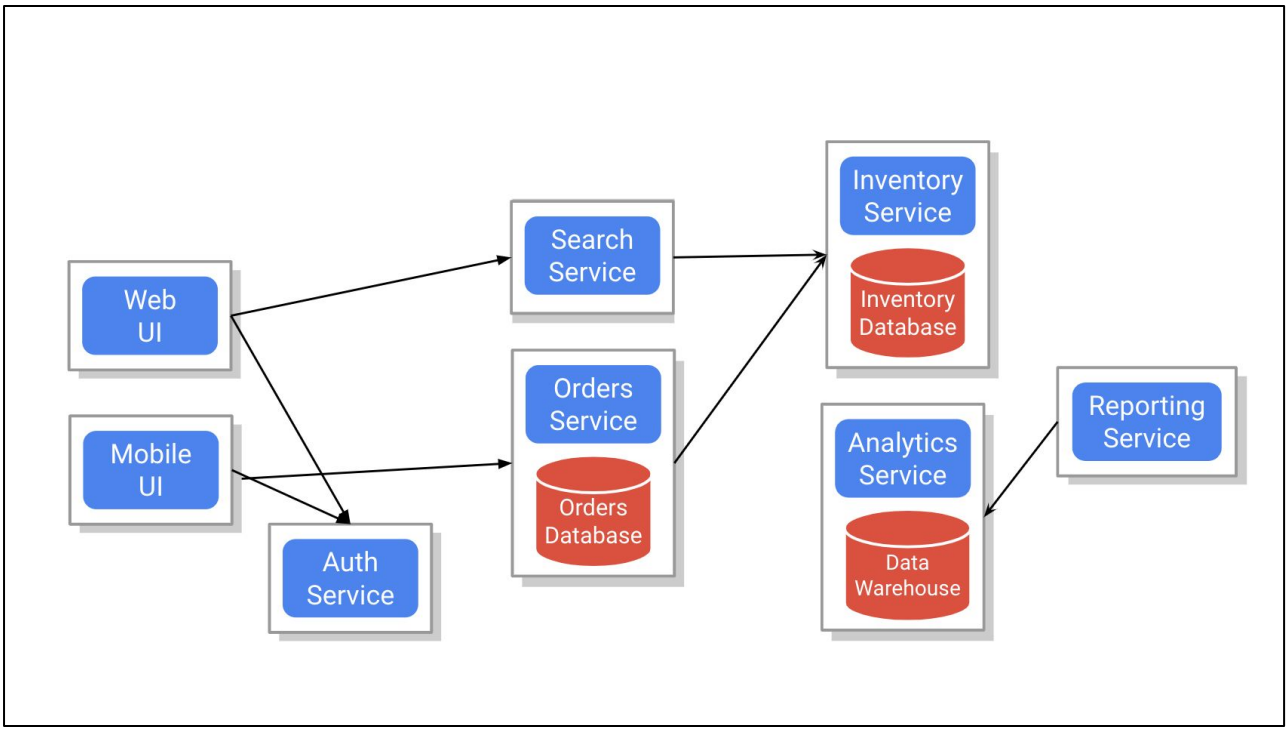
## Review Activity 4: Designing microservices for your application

- Diagram the microservices required by your case-study application.

In this activity, you were asked to diagram your case study application using a microservice-style architecture. The number of microservices appropriate for your application and recognizing the microservice boundaries is not obvious. Two programs might look similar, but their architectures might be considerably different based on number of users, size of data, security, and many other factors.

Fewer services might make deployment and communication between services easier but also make development and adding new features harder. Having more, smaller services makes each individual service easier to understand and implement, but may make the overall architecture of your program more complicated.

Like many things in life, you are looking for the right balance, trading one type of complexity for a different type, hoping to make the system overall as simple and as maintainable as possible.




Here is a sample diagram depicting the microservices of our online travel portal. I suppose we could lay this out many different ways. There isn't really one and only one right way to design an application.

Notice, we have separate services for our web and mobile UIs. There's a shared authentication service and we have microservices for search, orders, inventory, analytics and reporting. Remember, each of these services will be deployed as a separate application. Where possible we want stateless services, but the orders and inventory services will need databases, and the analytics service will provide a data warehouse.

This might make a good starting point, and we could adjust as needed when we starting implementing the application.

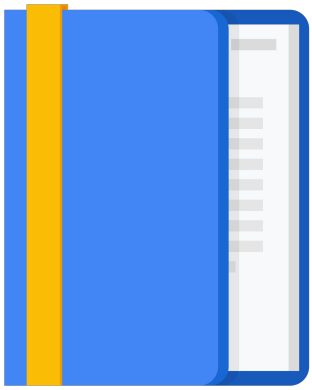
# Agenda

Microservices

Microservice Best Practices

REST

APIs



Let’s talk about the design of microservices based on REST and HTTP to achieve loosely coupled independent services.

## A good microservice design is loosely coupled

- Clients should not need to know too many details of services they use
- Services communicate via HTTPS using text-based payloads
  - Client makes GET, POST, PUT, or DELETE request
  - Body of the request is formatted as JSON or XML
  - Results returned as JSON, XML, or HTML
- Services should add functionality without breaking existing clients
  - Add, but don't remove, items from responses

*If microservices aren't loosely coupled, you'll end up with a really complicated monolith.*

One of the most important aspects of microservices-based applications is the ability to deploy microservices completely independent of one another. To achieve this independence, each microservice must provide a versioned, well-defined contract to its clients, which are other microservices or applications. Each service must not break these versioned contracts until it's known that no other microservice relies on a particular, versioned contract. Remember that other microservices may need to roll back to a previous code version that requires a previous contract, so it's important to account for this fact in your deprecation and turn-down policies.
A culture around strong, versioned contracts is probably the most challenging organizational aspect of a stable, microservices-based application.

At the lower level of detail, services communicate using HTTPS with text-based payloads, for example JSON or XML, and use the HTTP verbs such as GET and POST to provide meaning for the actions requested. Clients should just need to know the minimal details to use the service: the URI, the request, and the response message formats.

## REST architecture supports loose coupling

- REST stands for *Representational State Transfer*
- Protocol independent
  - HTTP is most common
  - Others possible like gRPC
- Service endpoints supporting REST are called *RESTful*
- Client and Server communicate with Request – Response processing

REST architecture supports loose coupling. REST stands for *Representational State Transfer,* and is protocol independent. HTTP is the most common protocol, but gRPC is also widely used.

REST supports loose coupling but still requires strong engineering practices to maintain that loose coupling. A starting point is to have a strong contract. HTTP-based implementations can use a standard like OpenAPI, and gRPC provides protocol buffers. To help maintain loose coupling, it is vital to maintain backward compatibility of the contract and to design an API around a domain and not particular use cases or clients. If the latter is the case, each new use case or application will require another special-purpose REST API, regardless of protocol.

While request-response processing is the typical use case, streaming may also be required and can influence the choice of protocol. gRPC supports streaming, for example.

## RESTful services communicate over the web using HTTP(S)

- URIs (or endpoints) identify resources
  - Responses return an immutable representation of the resource information
- REST applications provide consistent, uniform interfaces
  - Representation can have links to additional resources
- Caching of immutable representations is appropriate

Resources are identified by URIs or endpoints, and responses to requests return an immutable representation of the resource information.

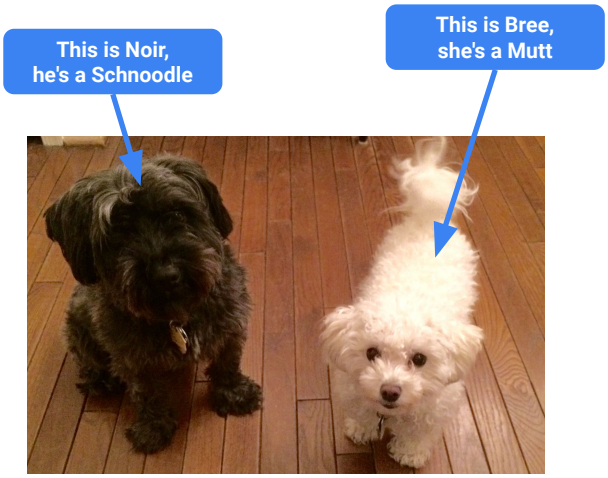
REST applications should provide consistent, uniform interfaces and can link to additional resources. Hypermedia as the Engine of Application State (or HATEOS) is a component of REST that allows the client to require little prior knowledge of a service because links to additional resources are provided as part of responses.

It is important that API design is part of the development process. Ideally, a set of API design rules is in place that helps the REST APIs provide a uniform interface; for example, each service reports errors consistently, the structure of the URLs is consistent, and the use of paging is consistent.

Also, consider caching for performance and resource optimization for immutable resources.

## Resources and representations

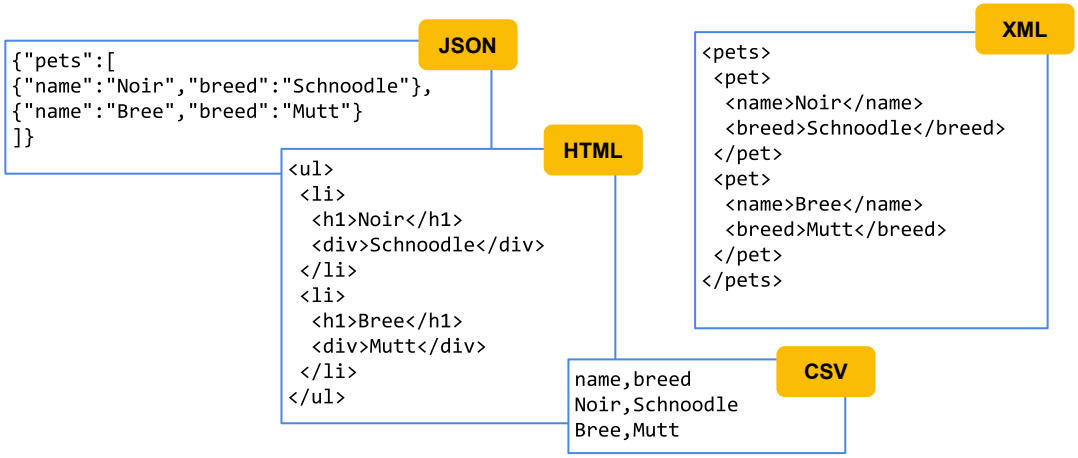
- Resource is an abstract notion of information
- Representation is a copy of the resource information
  - Representations can be single items or a collection of items




In REST, a client and server exchange representations of a resource. A resource is an abstract notion of information. The representation of a resource is a copy of the resource information. For example, a resource could represent a dog. The representation of a resource is the actual data for a particular dog; for example, Noir who is a schnoodle, or Bree who is a mutt. Two different representations of a resource.

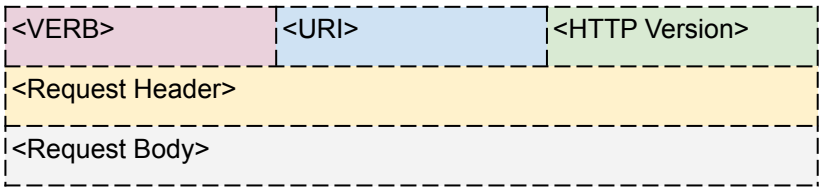
The URI provides access to a resource. Making a request for that resource returns a representation of that resource, usually in JSON format. The resources requested can be single items or a collection of items. For performance reasons, returning collections of items instead of individual items can be beneficial. These types of operations are often referred to as *batch APIs*.

## Passing representations between services is done using standard text-based formats




Representations of a resource between client and service are usually achieved using text-based standard formats. JSON is the norm for text-based formats, although XML can be used. For public-facing or external-facing APIs, JSON is the standard. For internal services, gRPC may be used, in particular if performance is key.

## Clients access services using HTTP requests




- VERB: GET, PUT, POST, DELETE
- URI: Uniform Resource Identifier (endpoint)
- Request Header: metadata about the message
  - Preferred representation formats (e.g., JSON, XML)
- Request Body: (Optional) Request state
  - Representation (JSON, XML) of resource

A client accessing HTTP services forms an HTTP request. HTTP requests are built in three parts: the request line, header variables, and request body.

The request line has the HTTP verb—GET, POST, PUT etc.—the requested URI, and the protocol version.

The header variables contain key-value pairs. Some of these are standard, such as User-Agent, which helps the receiver identify the requesting software agent. Metadata about the message format or preferred message formats is also included here for HTTPS-based REST services. You can add custom headers here.

The request body contains data to be sent to the server and is only relevant for HTTP commands that send data, such as POST and PUT.

## HTTP requests are simple and text-based

```
GET / HTTP/1.1
Host: pets.drehnstrom.com
```

```
POST /add HTTP/1.1
Host: pets.drehnstrom.com
Content-Type: json
Content-Length: 35

{"name":"Noir","breed":"Schnoodle"}
```

Here we see two examples of HTTP client text-based messages. The first example shows an HTTP GET request to the URL /  using HTTP version 1.1

There is one request header variable named Host with the value pets.drehnstrom.com

The second example shows an HTTP POST request to the URL /add using HTTP version 1.1

There are three request header variables:
Host:
Content-Type: set to json
Content-Length: set to 35 bytes

There is the request body which has the JSON document: {"name":"Noir","breed":"Schnoodle"}. This is the representation of the pet being added.

## The HTTP verb tells the server what to do

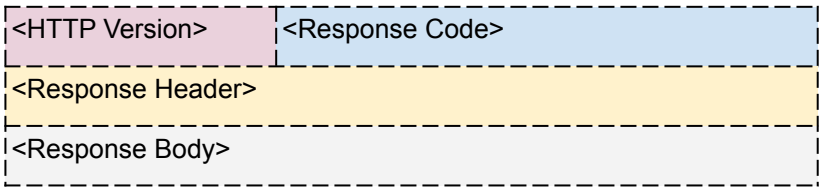
- **GET** is used to retrieve data
- **POST** is used to create data
  - Generates entity ID and returns it to the client
- **PUT** is used to create data or alter existing data
  - Entity ID must be known
  - *PUT should be* idempotent*, which means that whether the request is made once or multiple times, the effects on the data are exactly the same*
- **DELETE** is used to remove data

As part of a request, the HTTP verb tells the server the action to be performed on a resource.

HTTP as a protocol provides nine verbs, but usually only the four listed here are used in REST.

- GET is used to retrieve resources.
- POST is used to request the creation of a new resource. The service then creates the resource and usually returns the unique ID generated for the new resource to the client.
- PUT is used to create a new resource or make a change to an existing resource. PUT requests should be idempotent, which means that no matter how many times the request is made by the client to a service, the effects on the resource are always exactly the same.
- Finally, a DELETE request is used to remove a resource.

## Services return HTTP responses




- Response Code: 3-digit HTTP status code
  - 200 codes for success
  - 400 codes for client errors
  - 500 codes for server errors
- Response Body: contains resource representation
  - JSON, XML, HTML, etc.

HTTP services return responses in a standard format defined by HTTP.
These HTTP responses are built in three parts: the response line, header variables, and response body.

The response line has the HTTP version and a response code. The response codes are broken on boundaries around the 100s.

- The 200 range means ok. For example, 200 is OK, 201 means a resource has been created.
- The 400 range means the client request is in error. For example, 403 means “forbidden due to requestor not having permission.” 404 means “requested resource not found.”
- The 500 range means the server encountered an error and cannot process the request. For example, 500 is “internal server error.” 503 is “not available,” usually because the server is overloaded.

The response header is a set of key-value pairs, such as Content-Type, which indicates to the receiver the type of content the response body contains.

The response body has the resource representation requested in the format specified in the Content-Type header and can be JSON, XML, HTML, etc.

## All services need URIs (Uniform Resource Identifiers)

- Plural nouns for sets (collections)
- Singular nouns for individual resources
- Strive for consistent naming
- URI is case-insensitive
- Don’t use verbs to identify a resource
- Include version information

The guidelines listed here focus on achieving consistency on the API.

Singular nouns should be used for individual resources, and plural nouns for collections or sets.

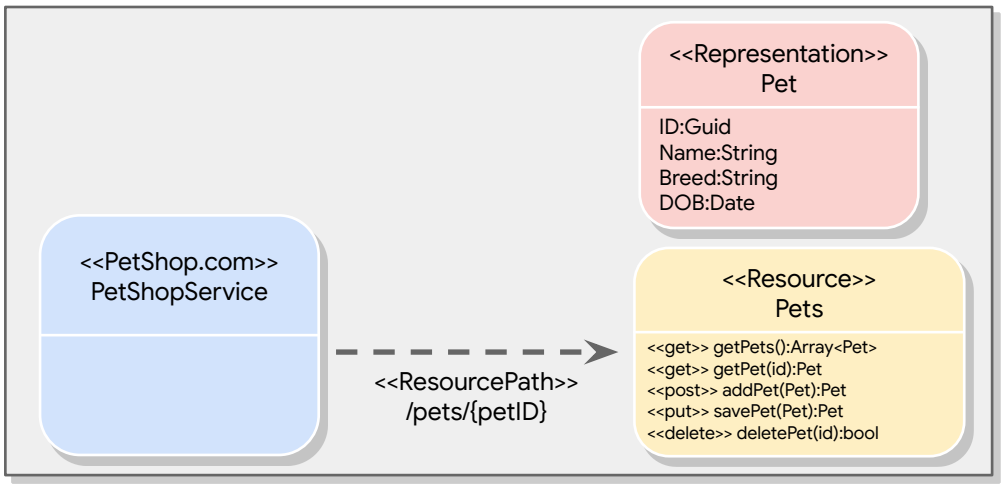
For an example, consider the following URI /pet.

Then:
A GET request for the URI  /pet/1 should fetch a pet with id 1, whereas GET /pets fetches all the pets.

Do not use URIs such as GET /getPets. The URI should refer to the resource, not the action on the resource—that is the role of the verb.

Remember that URIs are case-insensitive and that they include version information.

## Diagramming an example service




It is good practice to diagram services.

This diagram shows that there is a service that provides access to a resource known as Pets. The representation of the resource is the Pet. When a request is made for the resource via the service, one or more representations of a Pet are returned.

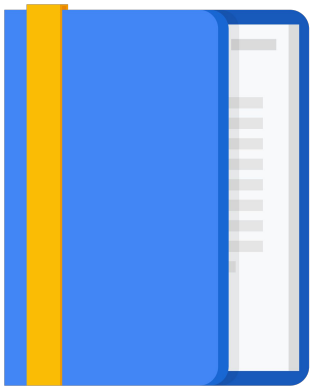
# Agenda

Microservices

Microservice Best Practices

REST

APIs



Let’s move on to API design.

## It's important to design consistent APIs for services

- Each Google Cloud service exposes a REST API
  - Functions are in the form: `service.collection.verb`
  - Parameters are passed either in the URL or in the request body in JSON format
- For example, the Compute Engine API has…
  - A service endpoint at: https://compute.googleapis.com
  - Collections include `instances`, `instanceGroups`, `instanceTemplates`, etc.
  - Verbs include insert, list, get, etc.
- So, to see all your instances, make a GET request to: `https://compute.googleapis.com/compute/v1/projects/{project}/zones/{zone}/instances`

It is important to design consistent APIs for services.

Google provides an API design guide with recommendations on items such as names, error handling documentation, versioning, and compatibility. This guide and the API stylebook are linked in the slides.
[https://cloud.google.com/apis/design/]
[http://apistylebook.com/design/guidelines/google-api-design-guide]

For examples of best practices, it is useful to examine the Google Cloud APIs.

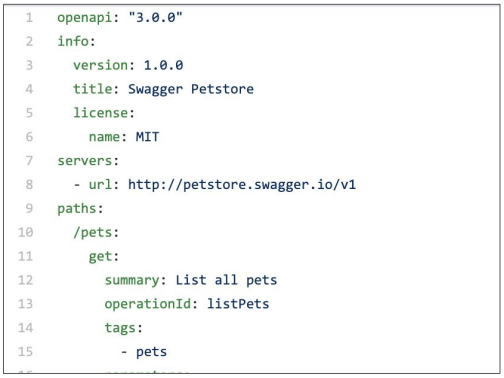
Each Google Cloud service exposes a REST API. Functions are defined in the form service.collection.verb. The service represents the service endpoint; e.g., for the Compute Engine API, the service endpoint is https://compute.googleapis.com.

Collections include instances, instanceGroups and instanceTemplates. The verbs then include LIST, GET, and INSERT, for example.

To see all your Compute Engine instances, make a GET request to the link shown on the slide. Parameters are passed either in the URL or on the request body in JSON format.

## OpenAPI is an industry standard for exposing APIs to clients

- Standard interface description format for REST APIs
  - Language independent
  - Open-source (based on Swagger)
- Allows tools and humans to understand how to use a service without needing its source code



```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
```

OpenAPI is an industry standard for exposing APIs to clients. Version 2.0 of the specification was known as Swagger. Swagger is now a set of open source tools built around Open API that, with associated tooling, supports designing, building, consuming, and documenting APIs. OpenAPI supports an API-first approach. Designing the API through OpenAPI can provide a single source of truth from which, source code for client libraries and server stubs can be generated automatically, as well as API user documentation. OpenAPI is supported by Cloud Endpoints and Apigee.

The example document shows a sample of an OpenAPI specification of a petstore service. The URI is http://petstore.swagger.io/v1. Note the version in the URI here. The example then shows an endpoint, /pets, which is accessed using the HTTP verb GET and will provide a list of all pets.

## gRPC is a lightweight protocol for fast, binary communication between services or devices

- Developed at Google
  - Supports many languages
  - Easy to implement
- gRPC is supported by Google services
  - Global load balancer (HTTP/2)
  - Cloud Endpoints
  - Can expose gRPC services using an Envoy Proxy in GKE

Developed at Google, gRPC is a binary protocol that is extremely useful for internal microservice communication. It provides support for many programming languages, has strong support for loose coupling via contracts defined using protocol buffers, and is high performing because it's a binary protocol. It is based on HTTP/2 and supports both client and server streaming.

The protocol is supported by many Google Cloud services, such as the global load balancer and Cloud Endpoints for microservices, as well as on GKE by using an envoy proxy.

## Google Cloud provides two tools, Cloud Endpoints and Apigee, for managing APIs

Both provide tools for:

- User authentication
- Monitoring
- Securing APIs
- Etc.

Both support OpenAPI and gRPC

Apigee API Platform

Cloud Endpoints

Google Cloud provides two tools for managing APIs: Cloud Endpoints and Apigee.

Cloud Endpoints is an API management gateway which helps you develop, deploy, and manage APIs on any Google Cloud backend. It runs on Google Cloud and leverages a lot of Google's underlying infrastructure.

Apigee is an API management platform built for enterprises, with deployment options on cloud, on-premises, or hybrid. The feature set includes an API gateway, customizable portal for onboarding partners and developers, monetization, and deep analytics around APIs. You can use Apigee for any http/https backends, no matter where they are running (on-premises, any public cloud, etc.).

Both solutions provide tools for services such as user authentication, monitoring, and securing and also for OpenAPI and gRPC.

## Activity 5: Designing REST APIs

Refer to your Design and Process Workbook.

- Design the APIs for your case study microservices.

You will now design the APIs for the microservices identified for your application. The aim of this activity is to gain experience designing the APIs and considering aspects such as the API URL structure, the message request response formats, and versioning.

| Service name | Collections | Methods |
|---|---|---|
| *Review* | *reviews* | *list*<br>*add*<br>*get* |
| *Product* | *inventory* | *add*<br>*get*<br>*update*<br>*Delete*<br>*search* |
| *Web UI* | *products* | *list*<br>*get*<br>*review* |
| | *accounts* | *get*<br>*update*<br>*delete*<br>*add* |

Let’s say we were defining an API for an online store. The API may look similar to what is shown here.

Whether a service provides a user-interface or some backend functionality, it requires a programmatic interface that is callable via https. The only difference between a website and a web service is the format of the data that is returned. For a UI service we might return HTML, but a backend service might return JSON or XML.

Use the principles we have discussed in this module so far and refer to activity 5 in the design workbook.

## Review Activity 5: Designing REST APIs

- Design the APIs for your case study microservices.

In this activity, you were asked to design a RESTful API for the microservices used in your case study.

| Service name | Collections | Methods |
| --- | --- | --- |
| Search | Trips (flights + hotel combination) | find<br>save |
| Inventory | Items (flights + hotels) | add<br>search<br>get<br>remove |
| Analytics | sales | analyze<br>get<br>list |
| Order Processing | orders | add<br>get<br>list<br>update |

Here's an example for our online travel portal. Obviously, our API would be larger than this, but in a way the APIs are all more of the same. Each service manages and makes available some collection of data. For any collection of data there are a handful of typical operations we do with that data.

This is similar to Google Cloud APIs. For example in Google Cloud, we have a service called Compute Engine, which is used to create and manage virtual machines, networks, and the like. The Compute Engine API has collections like instances, instanceGroups, networks, subnetworks, and many more. For each collection, various methods are used to manage the data.

## REST Resource: v1.firewalls

| Methods | |
|---|---|
| delete | `DELETE /compute/v1/projects/{project}/global/firewalls/{resourceId}`<br>Deletes the specified firewall. |
| get | `GET /compute/v1/projects/{project}/global/firewalls/{resourceId}`<br>Returns the specified firewall. |
| insert | `POST /compute/v1/projects/{project}/global/firewalls`<br>Creates a firewall rule in the specified project using the data included in the request. |
| list | `GET /compute/v1/projects/{project}/global/firewalls`<br>Retrieves the list of firewall rules available to the specified project. |
| patch | `PATCH /compute/v1/projects/{project}/global/firewalls/{resourceId}`<br>Updates the specified firewall rule with the data included in the request. |
| update | `PUT /compute/v1/projects/{project}/global/firewalls/{resourceId}`<br>Updates the specified firewall rule with the data included in the request. |

For example, here are methods for adding, managing, and deleting firewalls.

When you're designing your APIs, you should strive to be as consistent as possible. This will make development easier, and it will also make it easier for clients to learn to use your APIs.

# Review

## Microservice Design and Architecture

In this module we focused on microservice design and architecture. We started out defining what a microservice architecture is and the advantages and disadvantages of using microservices. We also enumerated some microservice best practices. Then, we covered how to design service APIs and implement a REST-style architecture.