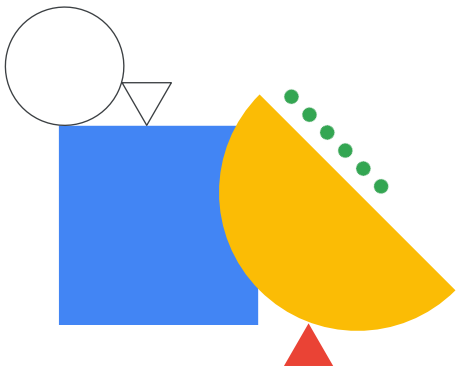
Google Cloud

Microservice Design and Architecture



This module introduces application architecture and microservice design.

Learning objectives

01 Decompose monolithic applications into microservices.

02 Recognize appropriate microservice boundaries.

03 Architect stateful and stateless services to optimize scalability and reliability.

04 Implement services using 12-factor best practices.

05 Build loosely coupled services by implementing a well-designed REST architecture.

06 Design consistent, standard RESTful service APIs.

This module introduces application architecture and microservices. The benefits of a microservice architecture for cloud-native applications is considered and contrasted with a monolith. The challenges of decomposing applications into clear microservice boundaries for independently deployable units and the challenges of state management on reliability and scalability are discussed.

Once a cloud-native microservice architecture has been chosen, the best practices for development and deployment and designing consistent loosely coupled service interfaces are introduced.

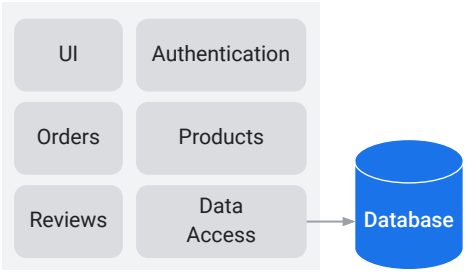
Agenda

01	Microservices
02	Microservice Best Practices
03	REST
04	APIs
05	Quiz
06	Review

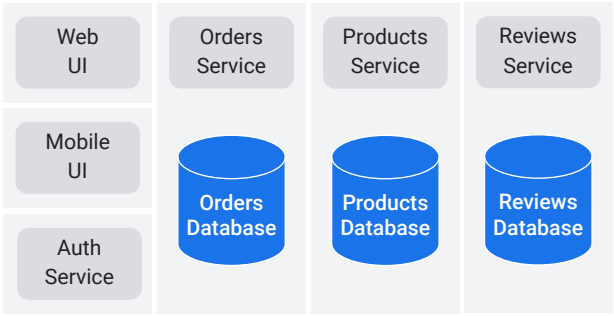
Microservices

Microservices divide a large program into multiple smaller, independent services

Monolithic applications implement all features in a single code base with a database for all data.



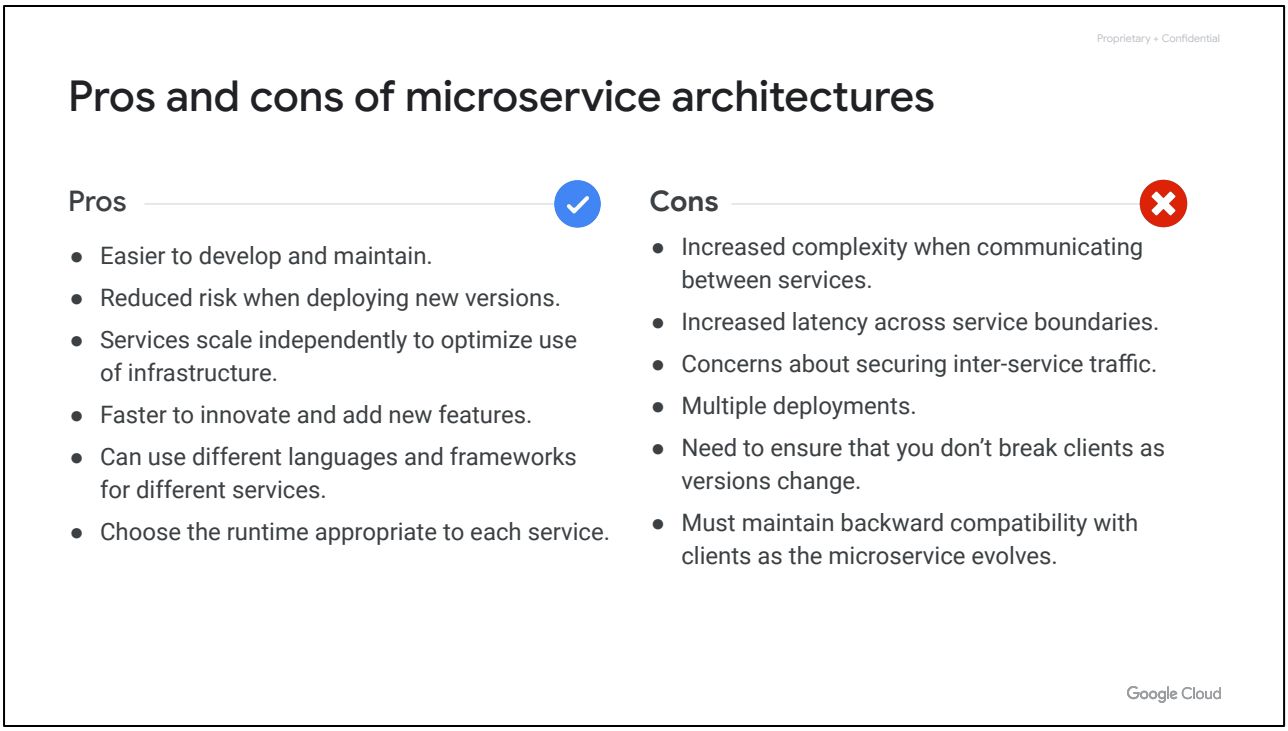
Microservice have multiple code bases, and each service manages its own data.



Microservice architecture is the current trend. It is 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, a monolith or microservices should be modular components with clearly defined boundaries. With a monolith, the deployment is the grouping of the components, whereas with microservices, the individual components are deployable. Google Cloud provides several services for deploying microservices from App Engine, Cloud Run, GKE, and Cloud Run 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. Google Cloud is strong here, with a wide selection of datastores.



A properly implemented microservices-based application can achieve the following goals:

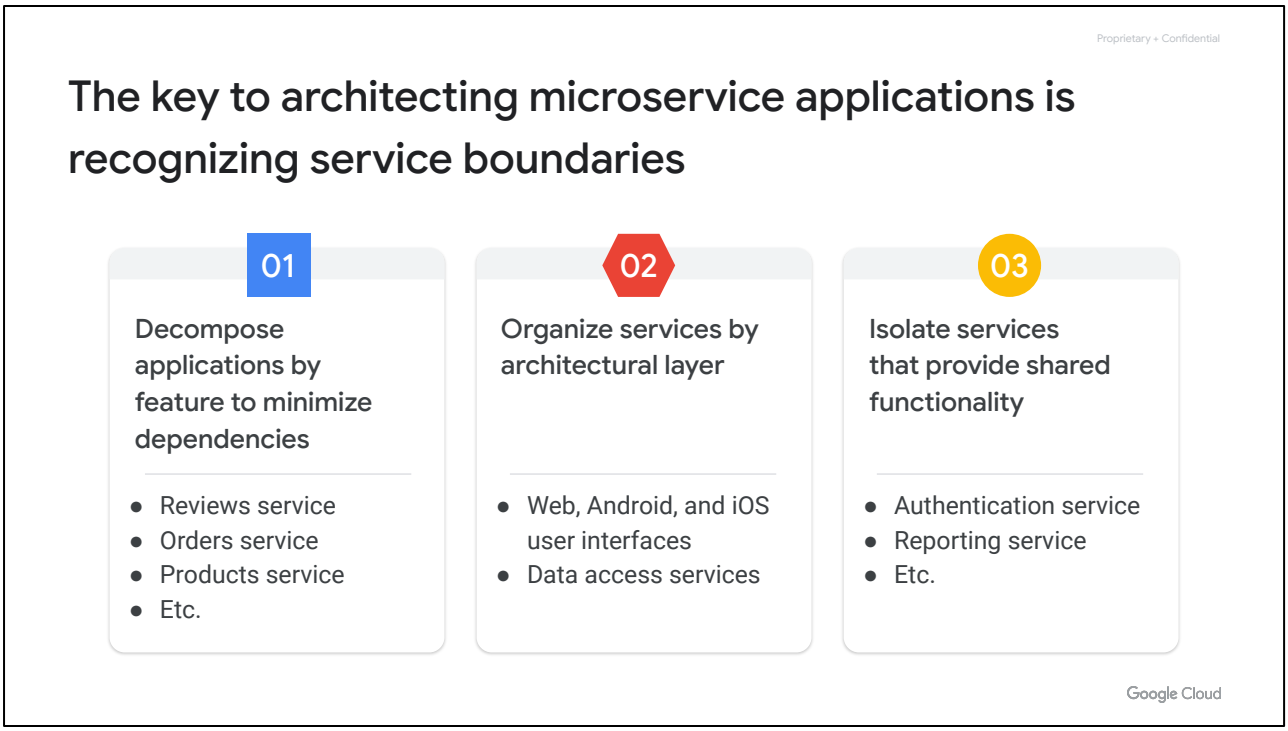
- Define strong contracts between the various microservices
- Allow for independent deployment cycles, including rollback
- Facilitate concurrent, A/B release testing on subsystems
- Minimize test automation and quality assurance overhead
- Improve clarity of logging and monitoring
- Provide fine-grained 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:

- 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.

More on the drawbacks can be found here
http://www.ptone.com/dablog/2015/07/microservices-may-be-the-new-premature-optimization/



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. https://en.wikipedia.org/wiki/Domain-driven_design

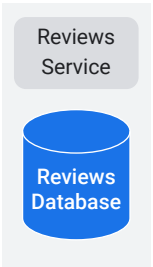
Consider, for example, an online retail application. Logical functional groupings could be product management, reviews, accounts, orders. These groupings then form mini applications which expose an API. Each of these mini applications will be implemented by potentially multiple microservices internally. Internally these microservices are organized by architectural layer, and each is independently deployable and scalable. Each in theory can be implemented in the language most suitable.

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

Stateful services have different challenges than stateless ones

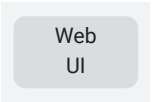
Stateful services manage stored data over time

- 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



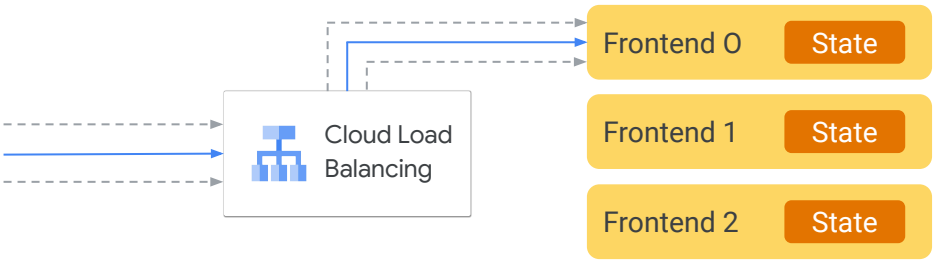
When you're designing microservices, services that do not maintain state but obtain their state from the environment or stateless services are easier to manage. That is, they are easy to scale, to administer, and to migrate to new versions because of their lack of state.

However, it is generally not possible to avoid using stateful 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

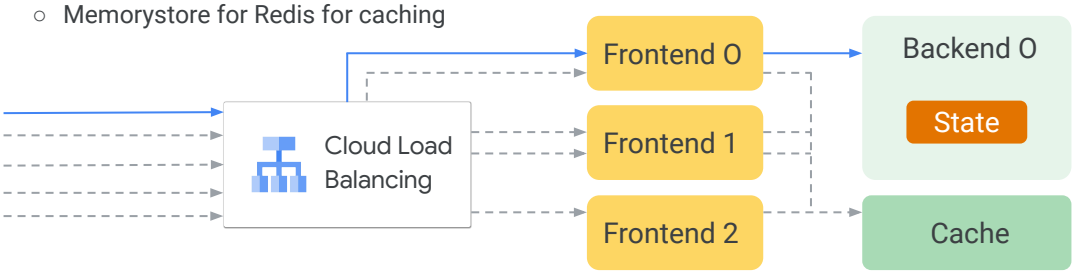
- Requires sticky sessions to be set up in the load balancer
- Hinders elastic autoscaling



In memory, shared state has implications that impact many of the benefits of a microservice architecture. The auto scaling potential 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

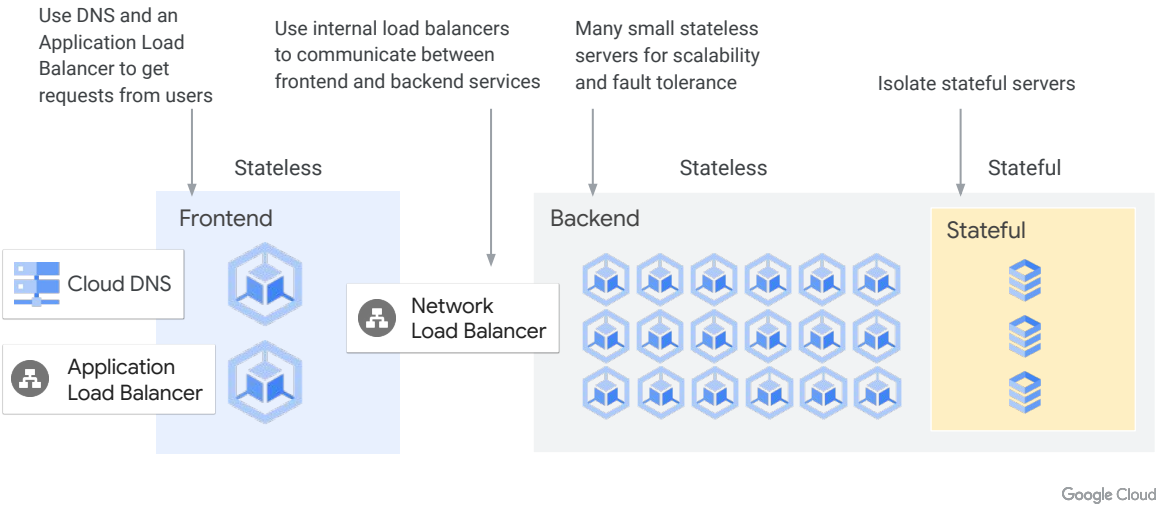
- Cache state data for faster access
- Take advantage of Google Cloud–managed data services
  - Firestore, Cloud SQL, etc. for state
  - Memorystore for Redis for caching



A recognized best practice for stateful services is to cache state using Google Cloud–managed data services.
For example, performance can be improved using Memorystore for Redis, which is a highly available Redis-based service:
https://cloud.google.com/memorystore/docs/redis/

The following repository shows an example of a microservice implementation of an online shop. Examining the cart service shows that a user's shopping cart is stored in a Redis cache and not on the cart microservice:
https://github.com/GoogleCloudPlatform/microservices-demo

A general solution for large-scale cloud-based systems



The slide 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 layout above 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.

Microservice Best Practices

The Twelve-Factor App is a set of best practices for building web or software-as-a-service applications

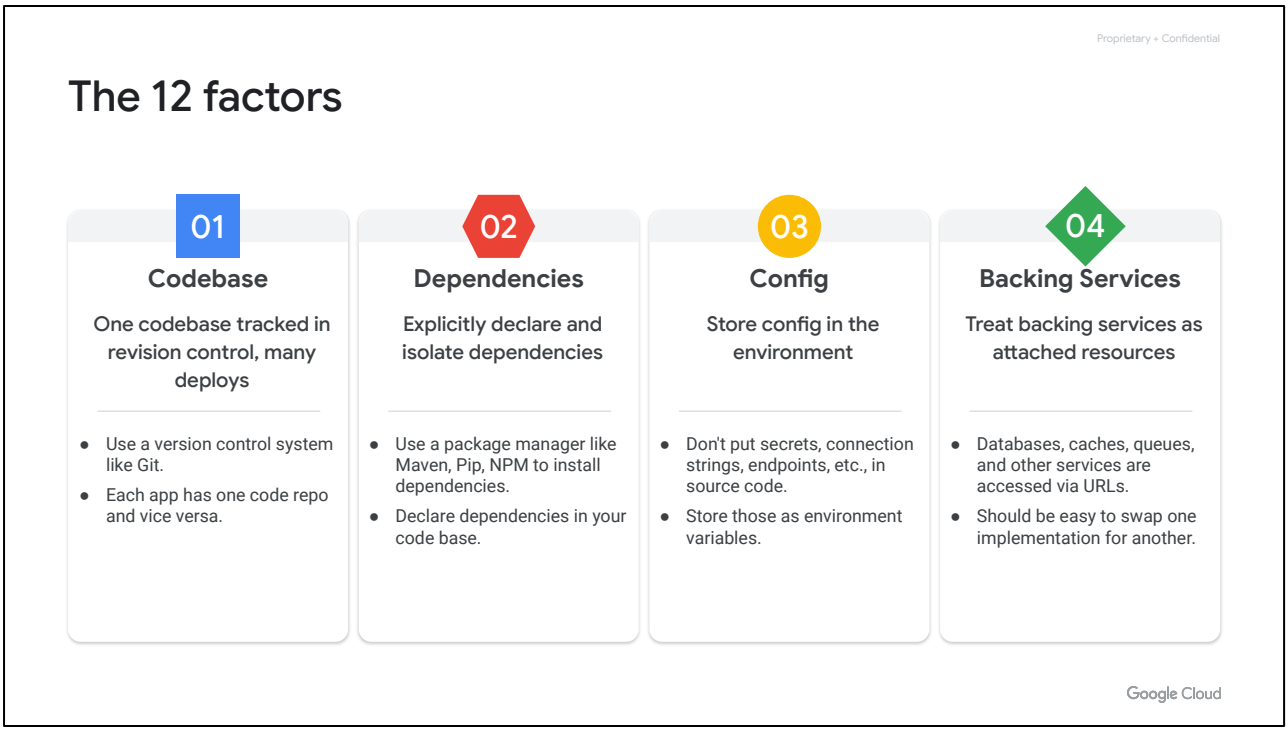
- Maximize portability
- Deploy to the cloud
- Enable continuous deployment
- Scale easily

THE TWELVE-FACTOR APP

https://12factor.net

Twelve-factor design also helps you decouple components of the application, so that each component can be replaced easily or scaled up or down seamlessly. Because the factors are independent of any programming language or software stack, 12-factor design can be applied to a wide variety of applications. Based on the experience of building cloud-native applications, extra factors above the 12 mentioned on the slide are now openly discussed in the community. For example, factors such as an API first strategy, stateless services, telemetry/observability, and security are often added.

An example article can be found here:
https://content.pivotal.io/blog/beyond-the-twelve-factor-app



Codebase
The codebase should be in version control such as Git. Cloud Source Repositories provides fully featured private repositories.
https://cloud.google.com/source-repositories/

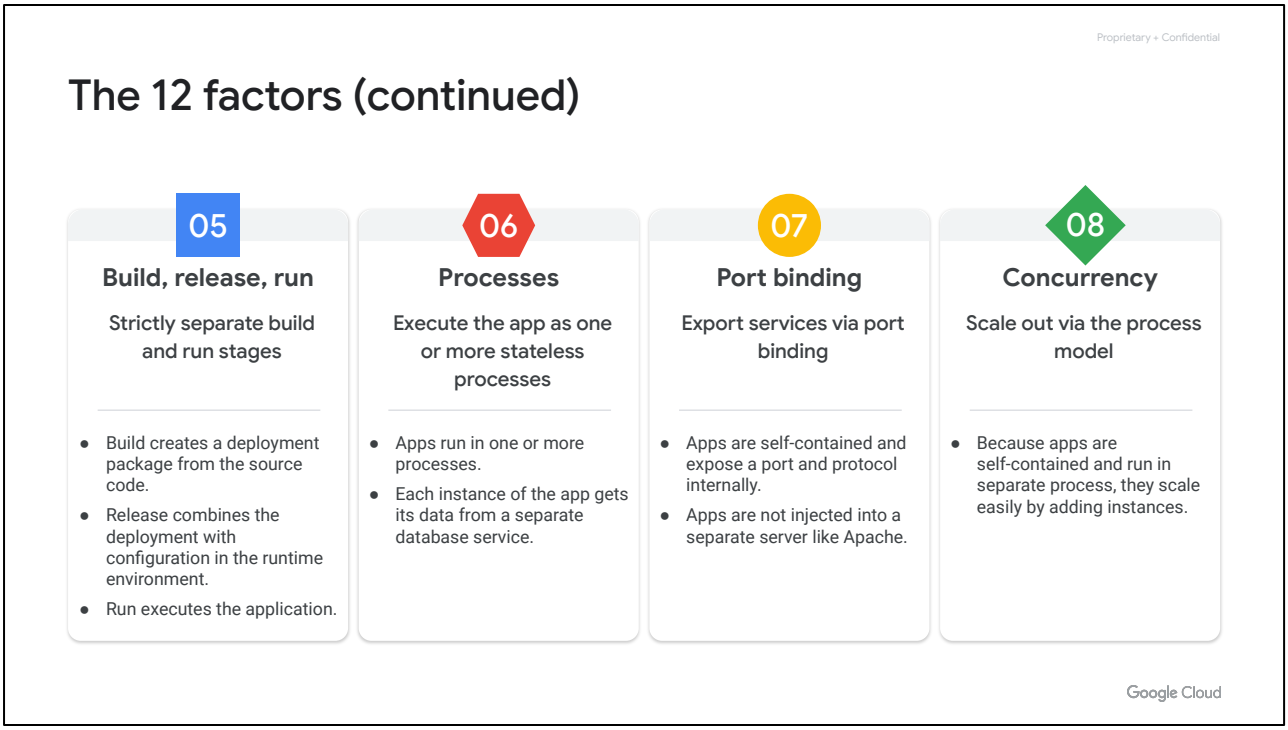
Dependencies
There are two considerations when it comes to dependencies for 12-factor apps: dependency declaration and dependency isolation.

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. Artifact Registry can be used to store the images and provide fine-grained access control.

Config
Every app has configuration for different environments: test, production, development. This configuration should be external to the code and usually kept in environment variables.

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.

For more details see:
https://cloud.google.com/solutions/twelve-factor-app-development-on-gcp

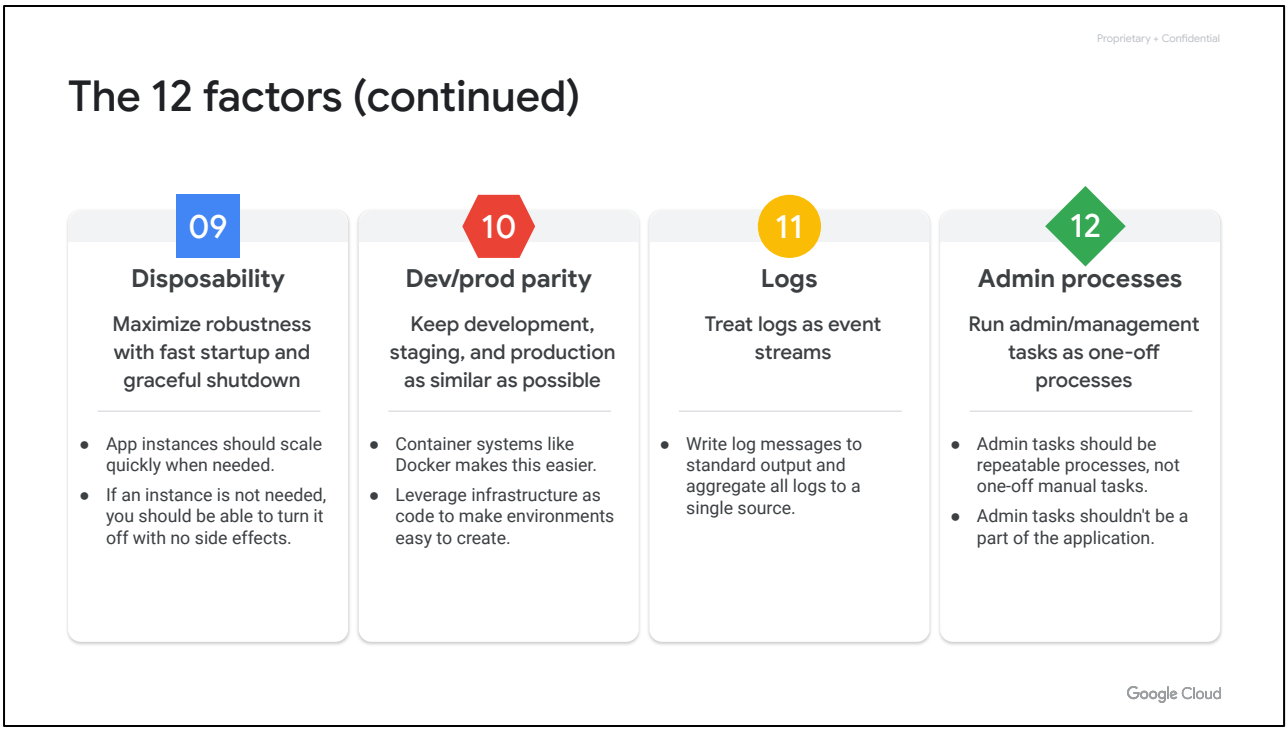


Build, release, run
The software deployment process should be broken into three distinct stages: build, release, and run. Each stage should result in an artifact that's uniquely identifiable. Every deployment should be linked to a specific release that's a result of combining an environment's configuration with a build. This allows easy rollbacks and a visible audit trail of the history of every production deployment.

Processes
Apps run as one or more stateless services. If state is required, then the technique discussed earlier in this module can be used. For example, use Memorystore to cache and share common data between services:
https://cloud.google.com/memorystore/

Port Binding
Services should be exposed using a port number. The apps bundle the web server as part of the app. In Google Cloud, such apps can be deployed on platform services such as Compute Engine, GKE, App Engine, or Cloud Run.

Concurrency
The app should be able to scale out by starting new processes and scale back in as needed to meet demand/load.



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.

Dev/production parity
The aim should be to have the same environments used in development and test/staging as are used in production. 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, Artifact Registry, and Terraform. Terraform uses the underlying APIs of each Google Cloud service to deploy your resources.

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. Decoupling logging 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.

Admin processes

These are usually one-off processes and should be decoupled from the application. Depending on your deployment on Google Cloud, there are many options for this including:
Cron jobs in GKE https://cloud.google.com/kubernetes-engine/docs/how-to/cronjobs
Cloud tasks on App Engine https://cloud.google.com/tasks/docs/
Cloud Scheduler https://cloud.google.com/scheduler/

Activity 4

30 min

Designing microservices for your application

Refer to your Design and Process Workbook.

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

REST

A good microservice design is loosely coupled

- Clients shouldn't 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. Keep in mind 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. API specifications using OpenAPI and test tools such as the Atlassian Request Response validator (https://bitbucket.org/atlassian/swagger-request-validator/src/master/) can be hugely beneficial, as can contract testing tools such as Pact. https://docs.pact.io/

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 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.

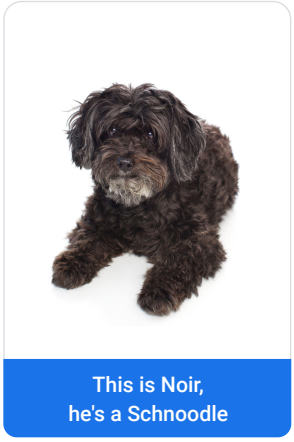
RESTful services communicate over the web using HTTP(S)

- 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

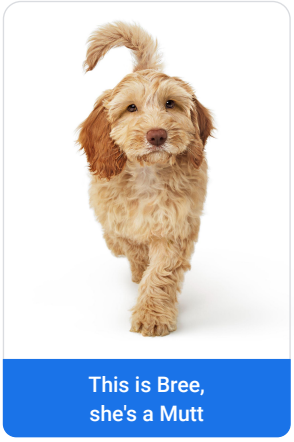
It is important that API design is part of the development process. Ideally, a set of API design rules is in place that helps the REST APIs provide a uniform interface; for example, each service reports errors consistently, the structure of the URLs is consistent, and the use of paging is consistent.

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



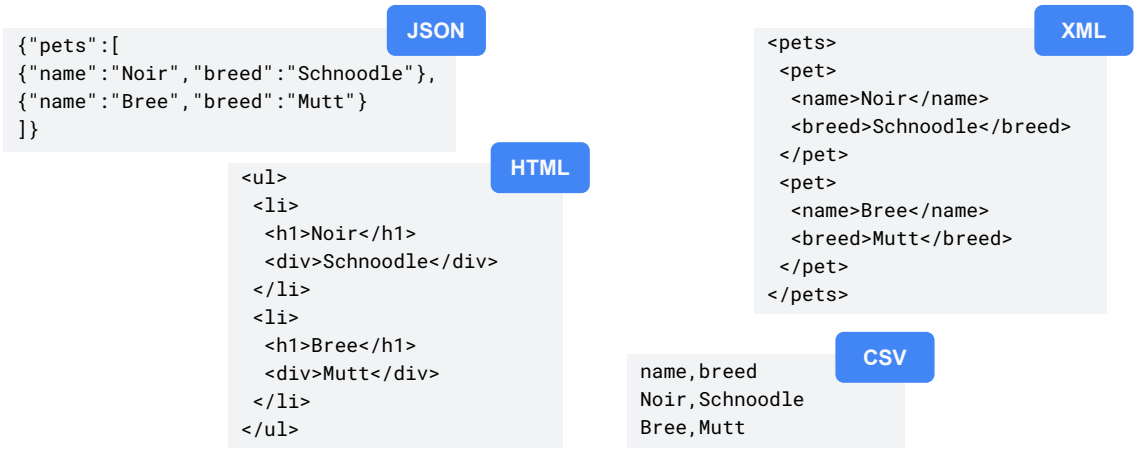
This is Noir,
he's a Schnoodle



This is Bree,
she's a Mutt

In REST, a client and server exchange 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*.

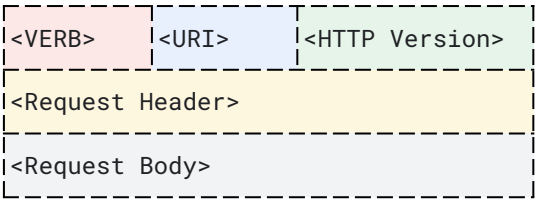
Passing representations between services is done using standard text-based formats



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



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. 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"}
```


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"}

The HTTP verb tells the server what to do

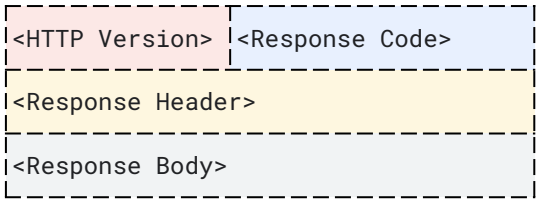
- **GET** is used to retrieve data
- **POST** is used to create data
  - Generates entity ID and returns it to the client
- **PUT** is used to create data or alter existing data
  - Entity ID must be known
  - *PUT should be* idempotent*, which means that whether the request is made once or multiple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

HTTP provides nine verbs, but only the four above are used in REST.

Another is HEAD, which asks for a response the same as for a GET but without a response body. For completeness, the others are CONNECT, OPTIONS, TRACE, and PATCH. Details can be found here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

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 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.

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

`GET /pet/1` should fetch a pet with ID 1
`POST /pet` inserts a pet

`GET /pets` fetches all the pets

Do not use `GET /getPets`

The above guidelines are about getting consistency on the API.

As an example, consider the following URI /pet.

Then:
GET /pet/1 should fetch a pet with id 1,
POST /pet inserts a pet

GET /pets fetches all pets

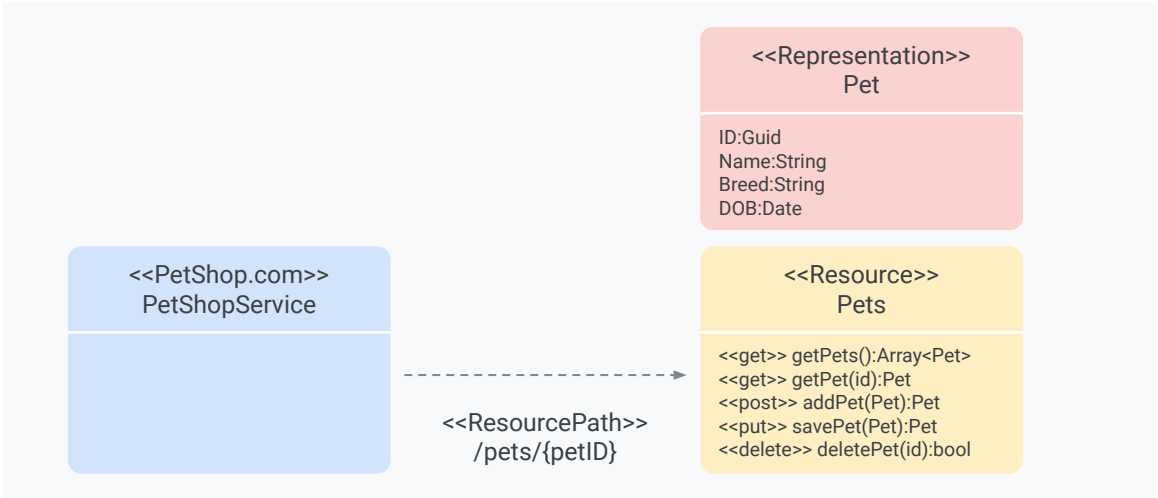
Do not use URIs such as:

GET /getPets

The URI should refer to the resource, not the action on the resource—that is the role of the verb.

Some details on versioning can be found here:
https://cloud.google.com/appengine/docs/standard/java/designing-microservice-api

Diagramming an example service



The above 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.

APIs

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

Google provides an API design guide (https://cloud.google.com/apis/design/) with recommendations on items such as names, error handling documentation, versioning, and compatibility.

There is also the API stylebook:
http://apistylebook.com/design/guidelines/google-api-design-guide

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

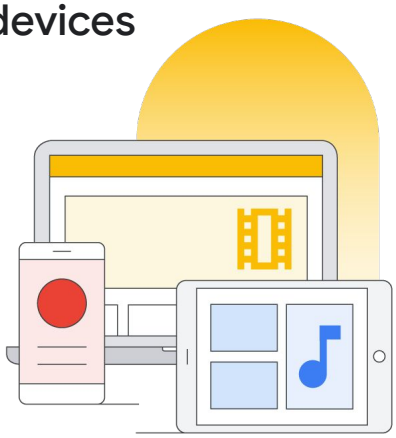
```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
         - pets
```


OpenAPI is an industry standard. Version 2.0 of the specification was known as Swagger. Swagger is now a set of open source tools built around Open API that 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 (discussed in upcoming slide).

See: https://github.com/OAI/OpenAPI-Specification

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
  - Application Load Balancer (HTTP/2)
  - Cloud Endpoints
  - Can expose gRPC services using an Envoy Proxy in GKE



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 Application Load Balancer and Cloud Endpoints for microservices, as well as on GKE by using an envoy proxy.

For more details see the following resources:

- https://www.grpc.io/
- https://cloud.google.com/endpoints/docs/grpc/about-grpc
- https://cloud.google.com/solutions/exposing-grpc-services-on-gke-using-envoy-proxy

Google Cloud provides three tools for managing APIs: Cloud Endpoints, Apigee, and API Gateway

They provide tools for:

- User authentication
- Monitoring
- Securing APIs
- Etc.

They support OpenAPI and gRPC

Apigee API Platform

API Gateway

Cloud Endpoints

Apigee, API Gateway, and Cloud Endpoints are built with a goal to manage all your APIs.

Endpoints is an API management gateway that 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.).

API Gateway enables you to provide secure access to your backend services through a well-defined REST API that is consistent across all of your services, regardless of the service implementation.

For more details, see the following resources:

- https://cloud.google.com/endpoints/
- https://cloud.google.com/apigee/
- https://cloud.google.com/api-gateway

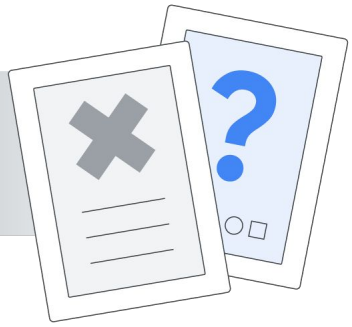
Activity 5

25 min

Designing REST APIs

Refer to your Design and Process Workbook.

- Design the APIs for your case study microservices.



Quiz

Question #1

Question

List some pros and cons of microservice architectures.

List some pros and cons of microservice architectures.

Question #1

Answer

List some pros and cons of microservice architectures.

Pros	Cons
• Easier to program and test	• Communication between services
• Scale independently	• Multiple deployments
• Less risky deployments	• Latency
• Easier to add new features	• Versioning
• Etc.	• Etc.

Items that could be considered include:

Pros:
Scale the organization; add more teams and increase development speed because they can work independently; contrast with a monolith where often adding more teams slows down development.

Let microservices use the most relevant technology and even different versions of same technology. For example, one service uses Java 8; another uses Java 11.

Cons:
Difficulty in identifying clear boundaries for independently deployable units.
Many more potential points of failure, and therefore cascading failures are possible.
Strong need for monitoring and observability to help with rapidly diagnosing faults: there are many more moving parts.

Question #2

Question

You've re-architected a monolithic web application so state is not stored in memory on the web servers, but in a database instead. This has caused slow performance when retrieving user sessions though. What might be the best way to fix this?

A. Move session state back onto the web servers and use sticky sessions in the load balancer.

B. Use a caching service like Memorystore for Redis.

C. Increase the number of CPUs in the database server.

D. Make sure all web servers are in the same zone as the database.

You've re-architected a monolithic web application so state is not stored in memory on the web servers, but in a database instead. This has caused slow performance when retrieving user sessions though. What might be the best way to fix this?

A. Move session state back onto the web servers and use sticky sessions in the load balancer.

B. Use a caching service like Memorystore for Redis.

C. Increase the number of CPUs in the database server.

D. Make sure all web servers are in the same zone as the database.

Question #2

Answer

You've re-architected a monolithic web application so state is not stored in memory on the web servers, but in a database instead. This has caused slow performance when retrieving user sessions though. What might be the best way to fix this?

A. Move session state back onto the web servers and use sticky sessions in the load balancer.

B. Use a caching service like Memorystore for Redis. 

C. Increase the number of CPUs in the database server.

D. Make sure all web servers are in the same zone as the database.

A. Answer is not correct. For scalability and high availability, services should be stateless.

B. This answer is correct. Services should be stateless, and a service like Memorystore for Redis provides a fast caching service to store state. They enable services to be stateless and support scale and high availability.

C. Answer is not correct. There is no indication where the problem is. It could be the network latency, but the solution should be to scale out, not scale up.

D. This answer is not correct. Having resources in the same zone makes a potential impact on availability but will also affect the user experience with possible unnecessary delays in accessing the service.

Question #3

Question

Which below would ***violate*** 12-factor app best practices?

A. Store configuration information in your source repository for easy versioning.

B. Treat logs as event streams and aggregate logs into a single source.

C. Keep development, testing, and production as similar as possible.

D. Explicitly declare and isolate dependencies.

Which below would ***violate*** 12-factor app best practices?
A. Store configuration information in your source repository for easy versioning
B. Treat logs as event streams and aggregate logs into a single source
C. Keep development, testing, and production as similar as possible
D. Explicitly declare and isolate dependencies

Question #3

Answer

Which below would ***violate*** 12-factor app best practices?

A. Store configuration information in your source repository for easy versioning. 

B. Treat logs as event streams and aggregate logs into a single source.

C. Keep development, testing, and production as similar as possible.

D. Explicitly declare and isolate dependencies.

A. This answer is correct. Code and config should be separated, because config varies across deployments but code does not. The true test is whether the repository could be open-sourced without compromising any credentials.

Answers B, C, and D are all explicitly listed as factors at https://12factor.net/ and so are not correct.

Question #4

Question

You're writing a service, and you need to handle a client sending you invalid data in the request. What should you return from the service?

A. An XML exception

B. A 200 error code

C. A 400 error code

D. A 500 error code

You're writing a service and you need to handle a client sending you invalid data in the request. What should you return from the service?

A. An XML exception
B. A 200 error code
C. A 400 error code
D. A 500 error code

Question #4

Answer

You're writing a service, and you need to handle a client sending you invalid data in the request. What should you return from the service?

A. An XML exception

B. A 200 error code

C. **A 400 error code**

D. A 500 error code

A. This answer is not correct. With REST, exceptions/errors should be reported with error codes, not exceptions.

B. This answer is not correct. 200 is a status code saying all is OK.

C. This answer is correct. 400 is a HTTP status code indicating that a request could not be processed due to an apparent client error.

D. This answer is not correct. 500 is a HTTP status code indicating an internal server error.

Question #5

Question

You're building a RESTful microservice. Which would be a valid data format for returning data to the client?

A. JSON

B. XML

C. HTML

D. All of the above

You're building a RESTful microservice. Which would be a valid data format for returning data to the client?

A. JSON
B. XML
C. HTML
D. All of the above

Question #5

Answer

You're building a RESTful microservice. Which would be a valid data format for returning data to the client?

A. JSON

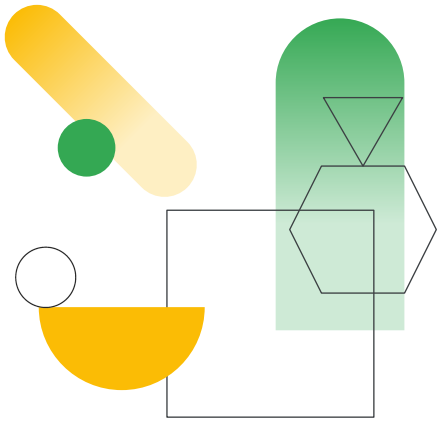
B. XML

C. HTML

D. **All of the above**

All of the above answers are correct. They have a standard Content-Type that can be set on the response header and are text-based. It is usual to use JSON, but both XML and JSON are valid.

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.

More resources

API Design Guide

https://cloud.google.com/apis/design/

Authenticating service-to-service calls with Google Cloud Endpoints

https://youtu.be/4PgX3yBJEyw

These two resources are useful to consult for API design and securing APIs.


Google Cloud