Google Cloud

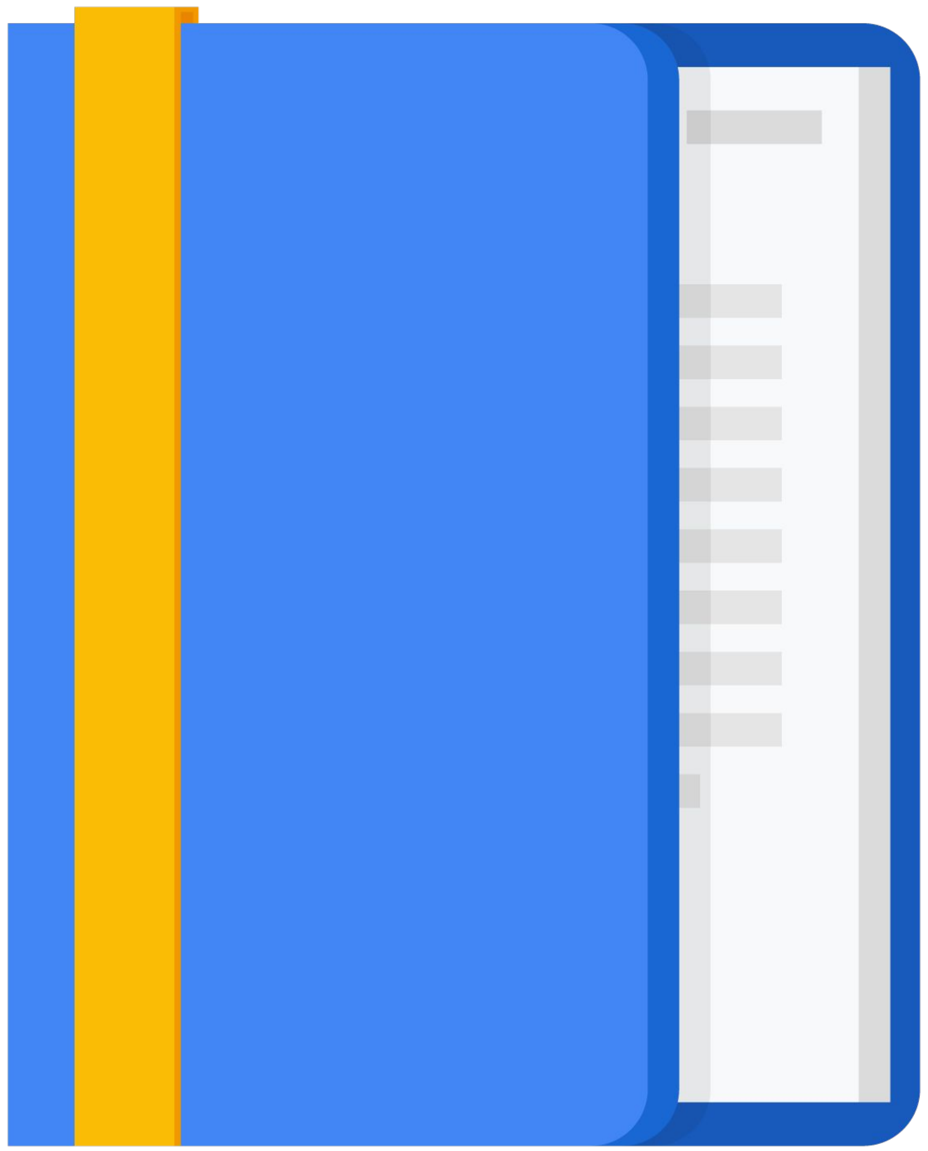
Microservice Design and Architecture

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.

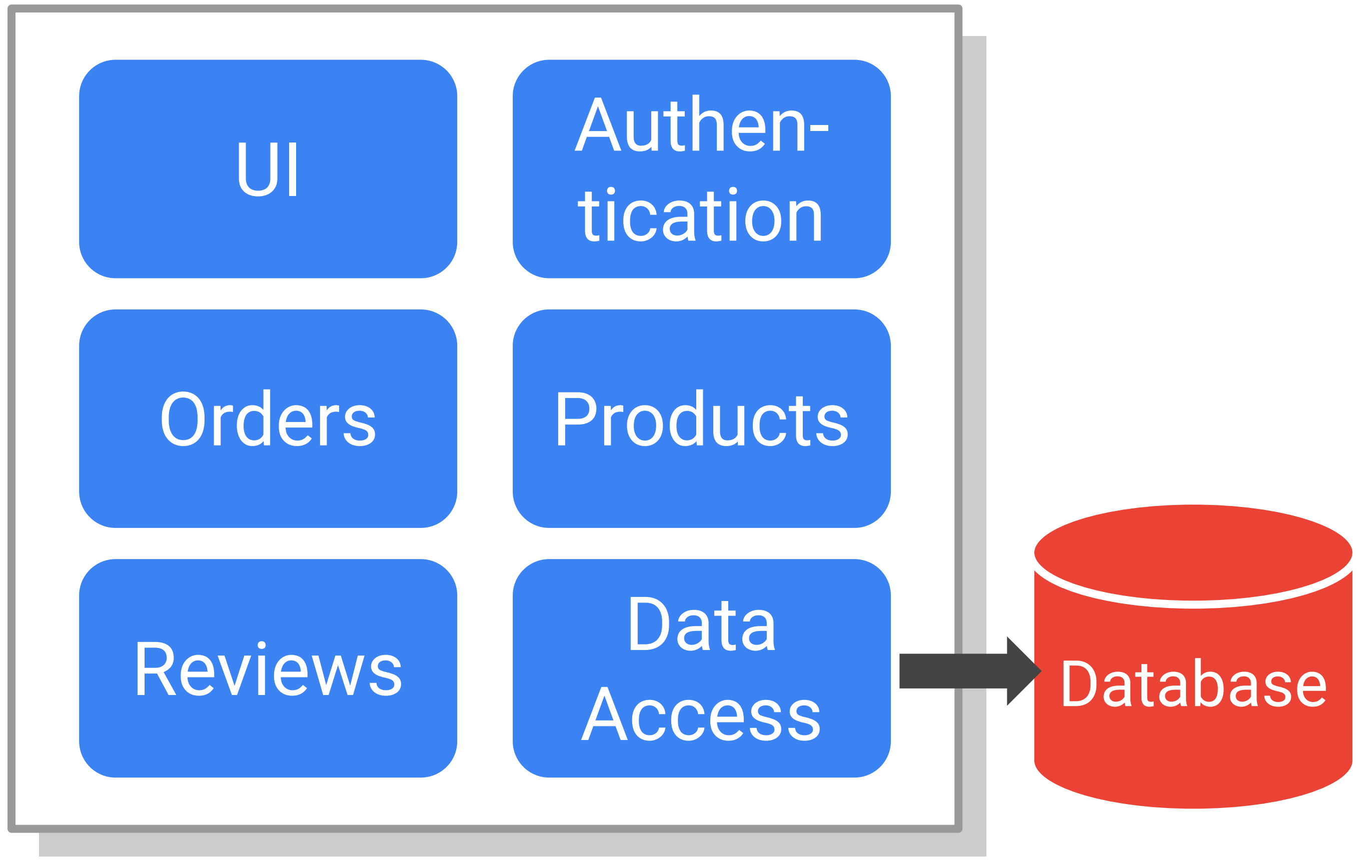
Agenda

- Microservices
- Microservice Best Practices
- Design Activity #4
- REST
- APIs
- Design Activity #5

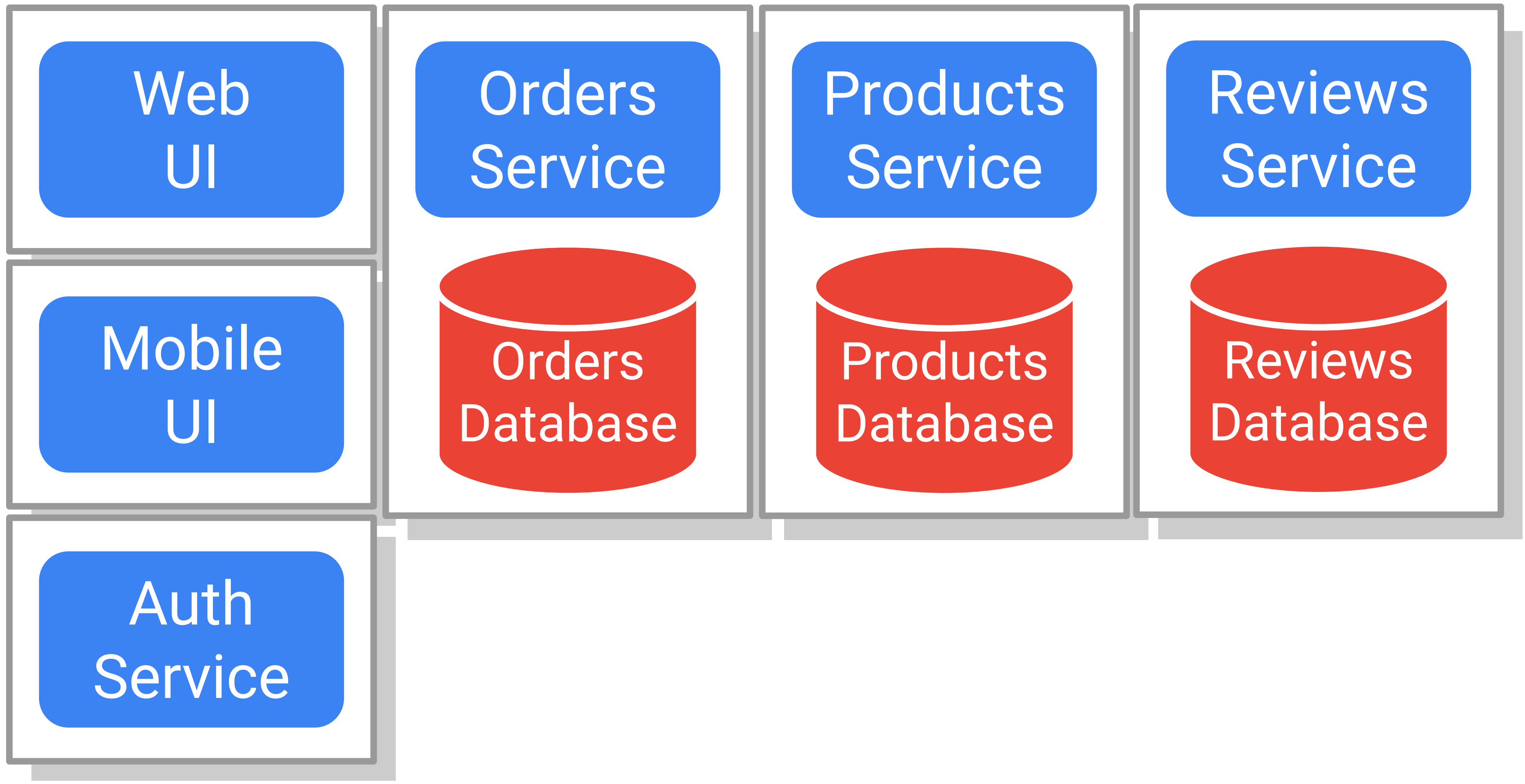


Microservices divide a large program into multiple smaller, independent services

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.



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 with clients as the microservice evolves

The key to architecting microservice applications is recognizing service boundaries

Decompose applications by feature to minimize dependencies

- Reviews service
- Orders service
- Products service
- Etc.

Organize services by architectural layer

- Web, Android, and iOS user interfaces
- Data access services

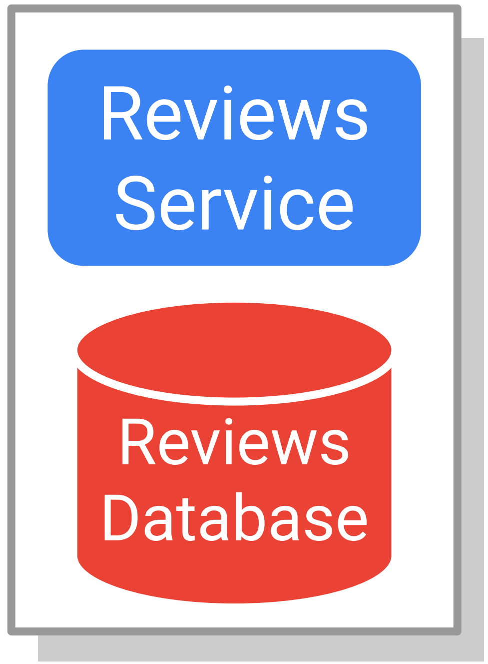
Isolate services that provide shared functionality

- Authentication service
- Reporting service
- Etc.

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

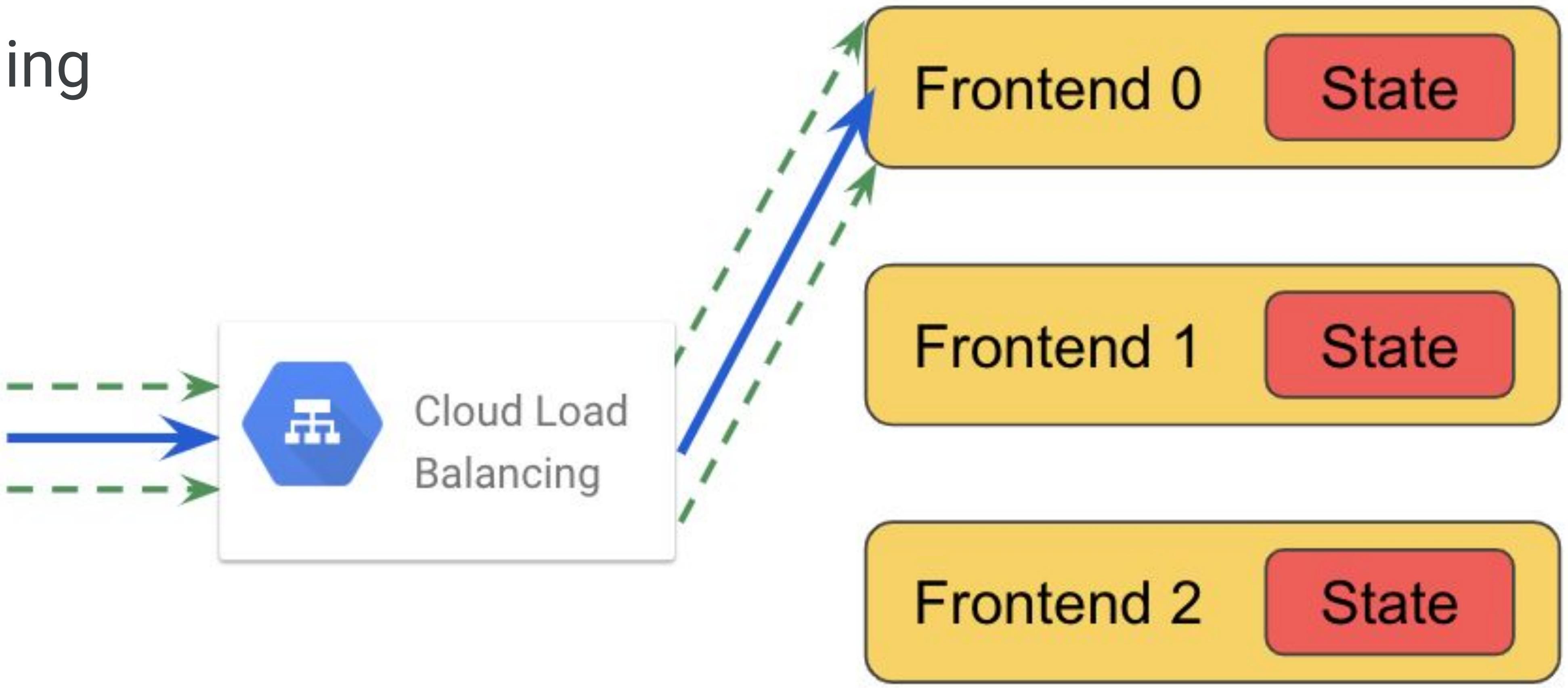


Stateless services get their data from the environment or other stateful services

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

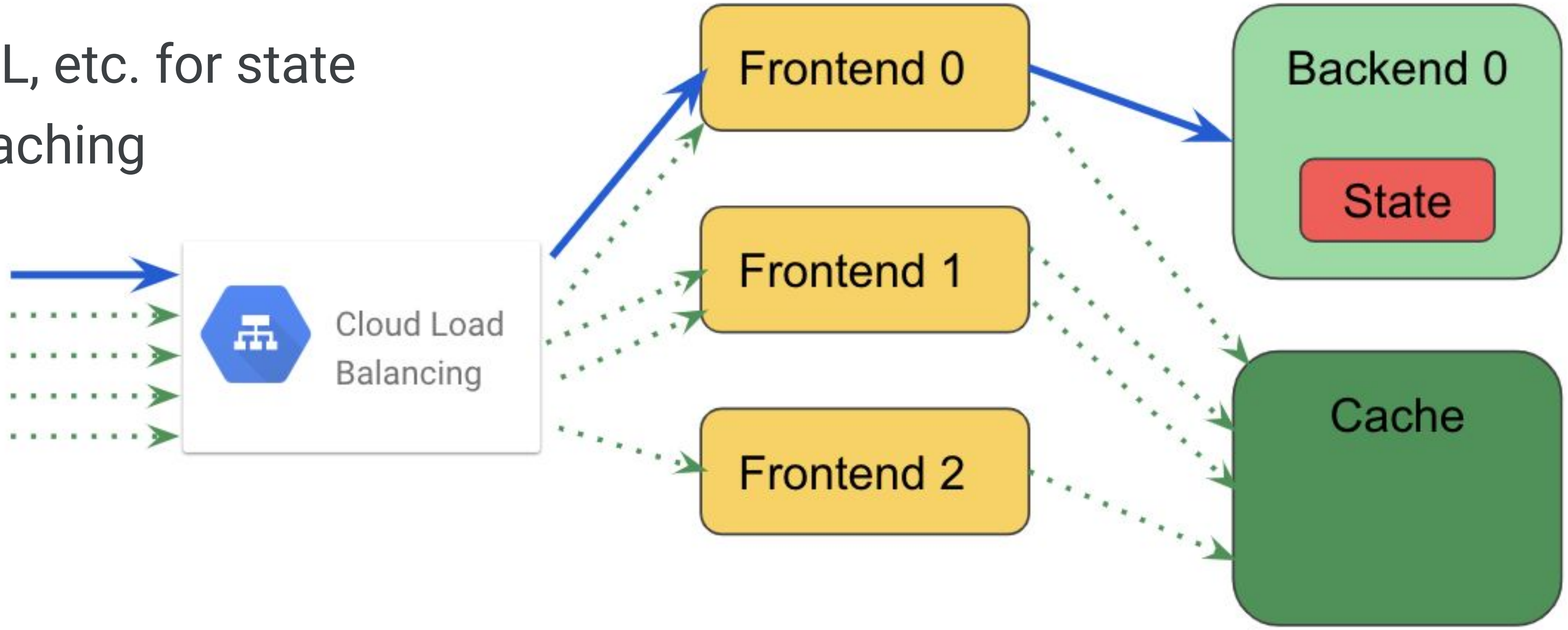
Avoid storing shared state in-memory on your servers

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

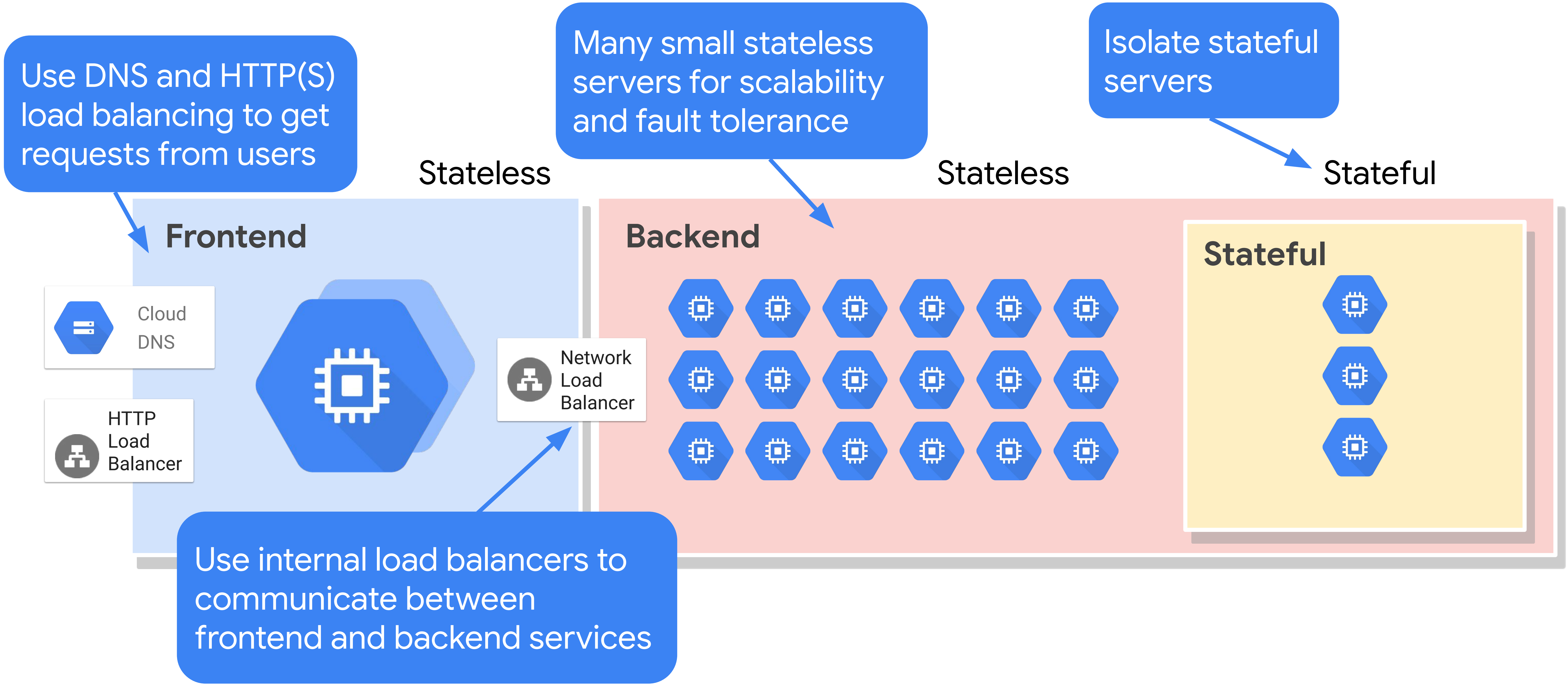


Store state using backend storage services shared by the frontend server

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



A general solution for large-scale cloud-based systems



Agenda

Microservices

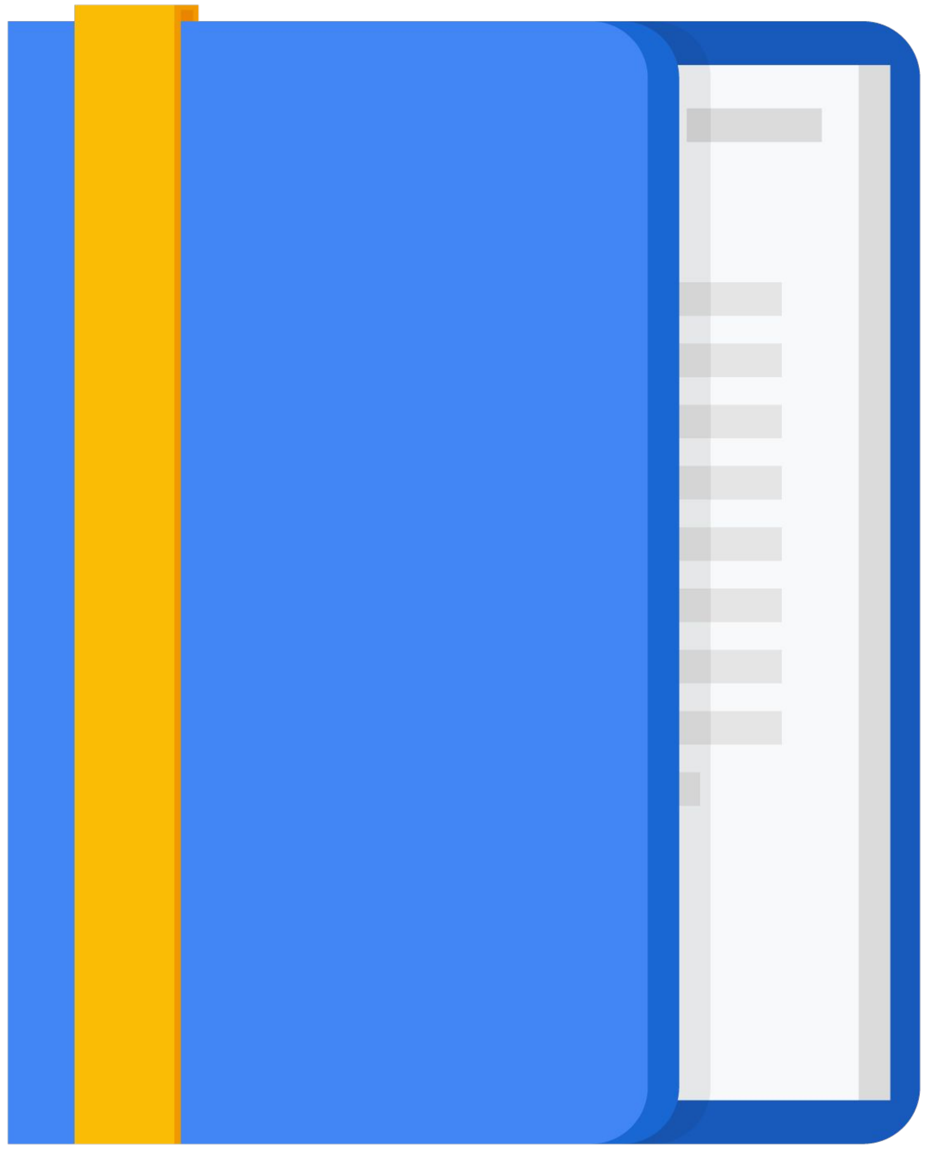
Microservice Best Practices

Design Activity #4

REST

APIs

Design Activity #5



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 TWELVE-FACTOR APP

The 12 factors

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

The 12 factors (continued)

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

The 12 factors (continued)

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

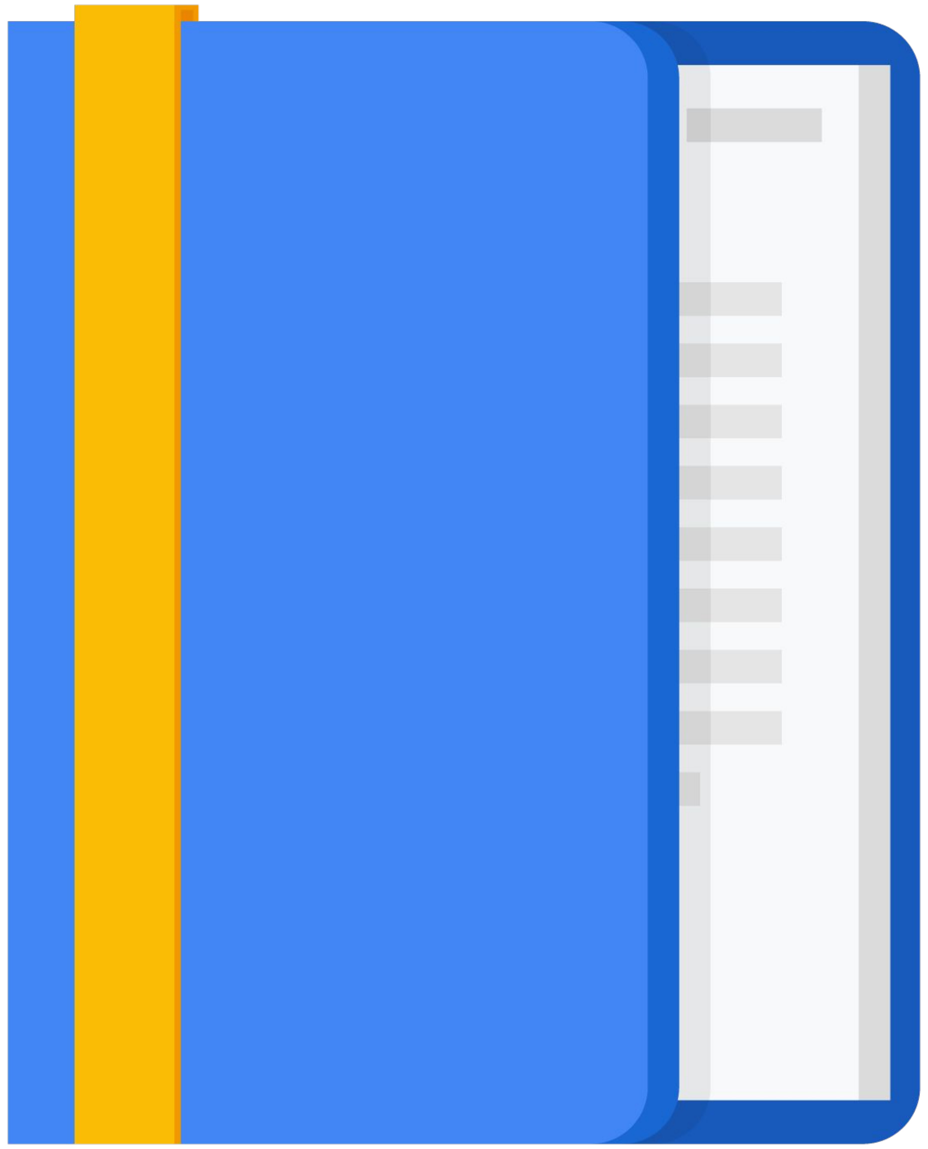
Activity 4: Designing microservices for your application

Refer to your Design and Process Workbook.

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

Agenda

- Microservices
- Microservice Best Practices
- Design Activity #4
- **REST**
- APIs
- Design Activity #5



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.

REST architecture supports loose coupling

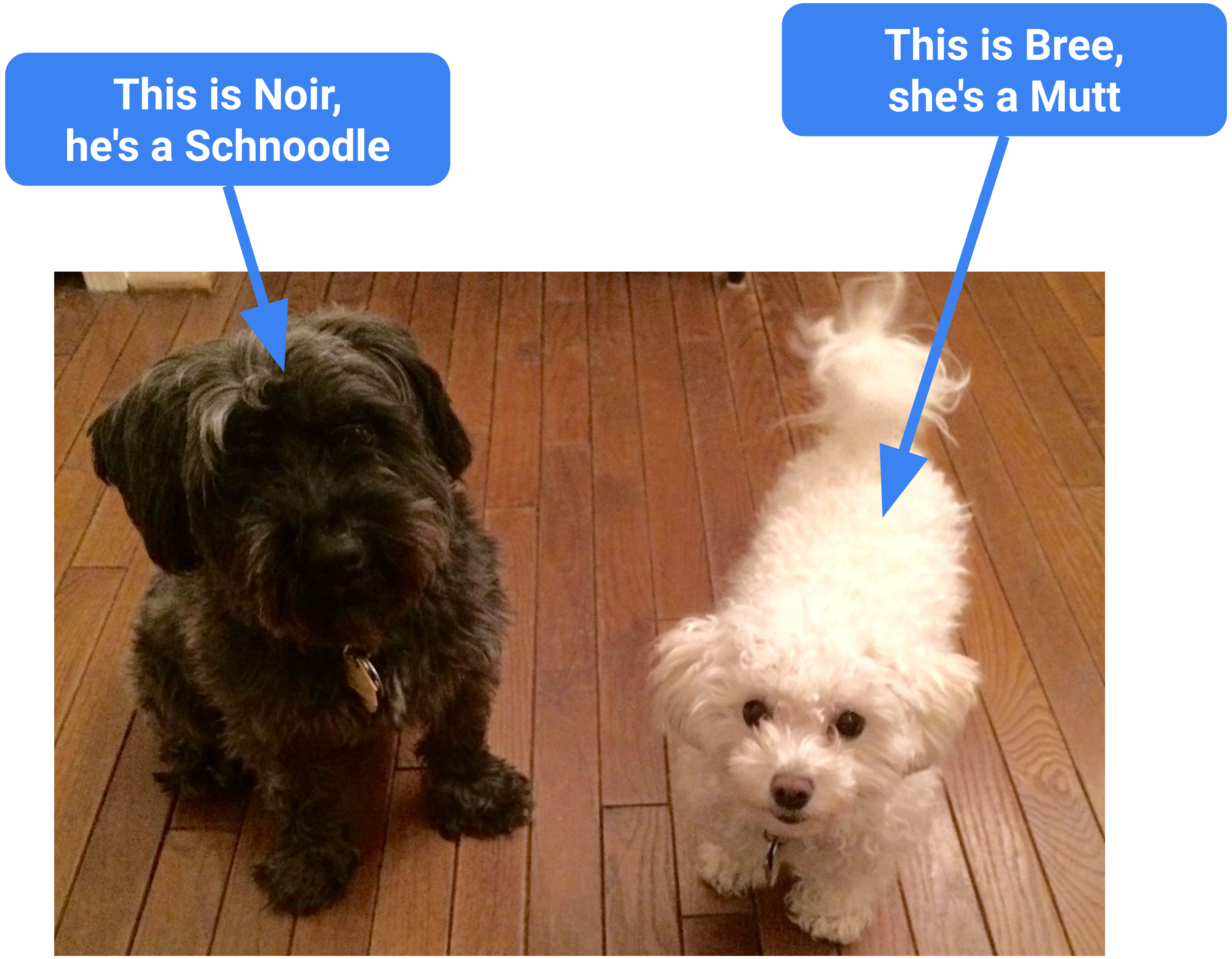
- REST stands for *Representational State Transfer*
- Protocol independent
  - HTTP is most common
  - Others possible like gRPC
- Service endpoints supporting REST are called *RESTful*
- Client and Server communicate with Request – Response processing

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 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



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

JSON

```json
{"pets":[
{"name":"Noir","breed":"Schnoodle"},
{"name":"Bree","breed":"Mutt"}
]}
```

HTML

```html
<ul>
 <li>
  <h1>Noir</h1>
  <div>Schnoodle</div>
 </li>
 <li>
  <h1>Bree</h1>
  <div>Mutt</div>
 </li>
</ul>
```

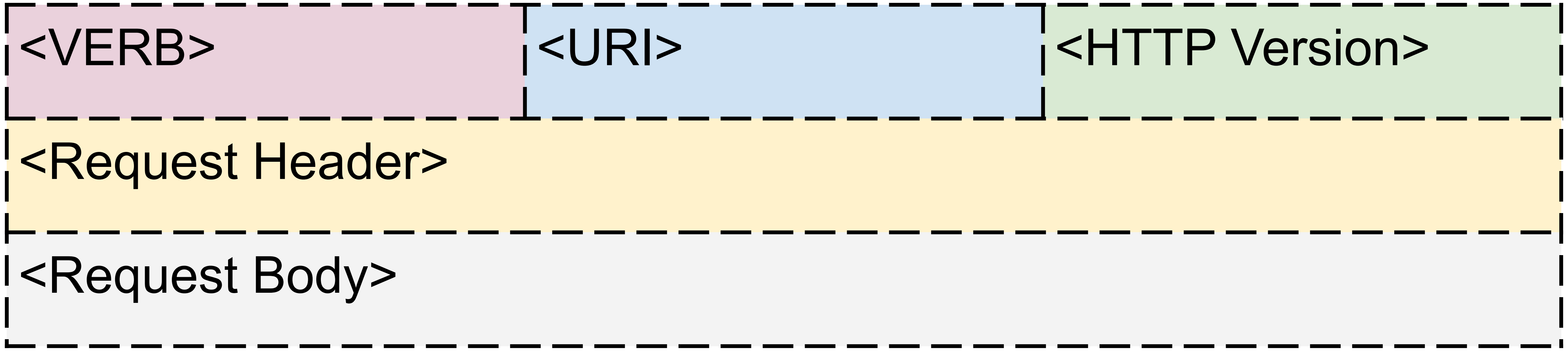
XML

```xml
<pets>
 <pet>
  <name>Noir</name>
  <breed>Schnoodle</breed>
 </pet>
 <pet>
  <name>Bree</name>
  <breed>Mutt</breed>
 </pet>
</pets>
```

CSV

```csv
name,breed
Noir,Schnoodle
Bree,Mutt
```

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 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 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

Services return HTTP responses

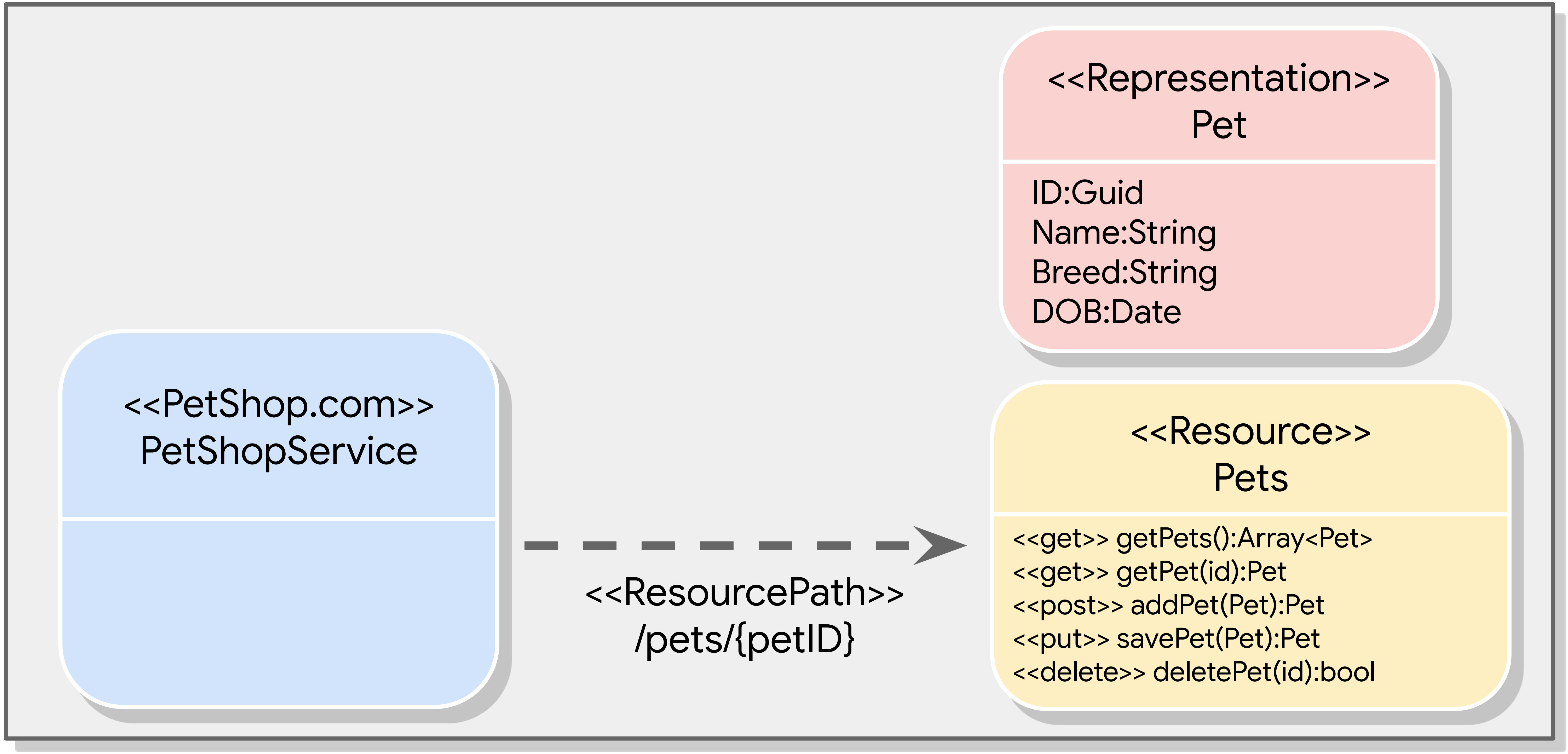
<HTTP Version>	<Response Code>
<Response Header>	
<Response Body>	

- 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.

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

Diagramming an example service



Google Cloud

Agenda

Microservices

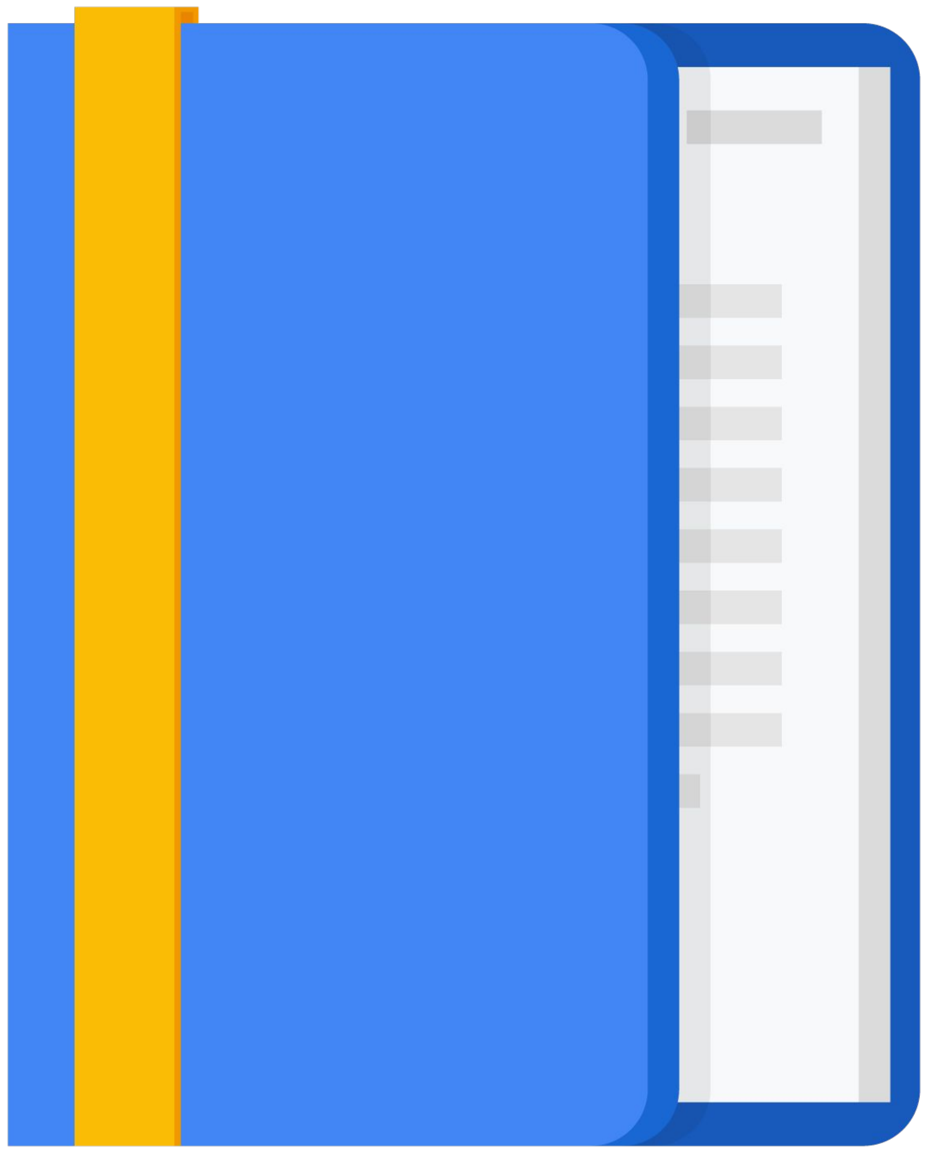
Microservice Best Practices

Design Activity #4

REST

APIs

Design Activity #5



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`

OpenAPI is an industry standard for exposing APIs to clients

- Standard interface description format for REST APIs
  - Language agnostic
  - 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
```

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

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

Activity 5: Designing REST APIs

Refer to your Design and Process Workbook.

- Design the APIs for your case study microservices.

Quiz

List some pros and cons of microservice architectures.

Quiz

List some pros and cons of microservice architectures.

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

Quiz

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 Redis or Memorystore.

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

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

Quiz

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 Redis or Memorystore.

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

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

Quiz

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.

Quiz

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.

Quiz

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

Quiz

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

Quiz

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

Quiz

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

Review

Microservice Design and 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

Google Cloud