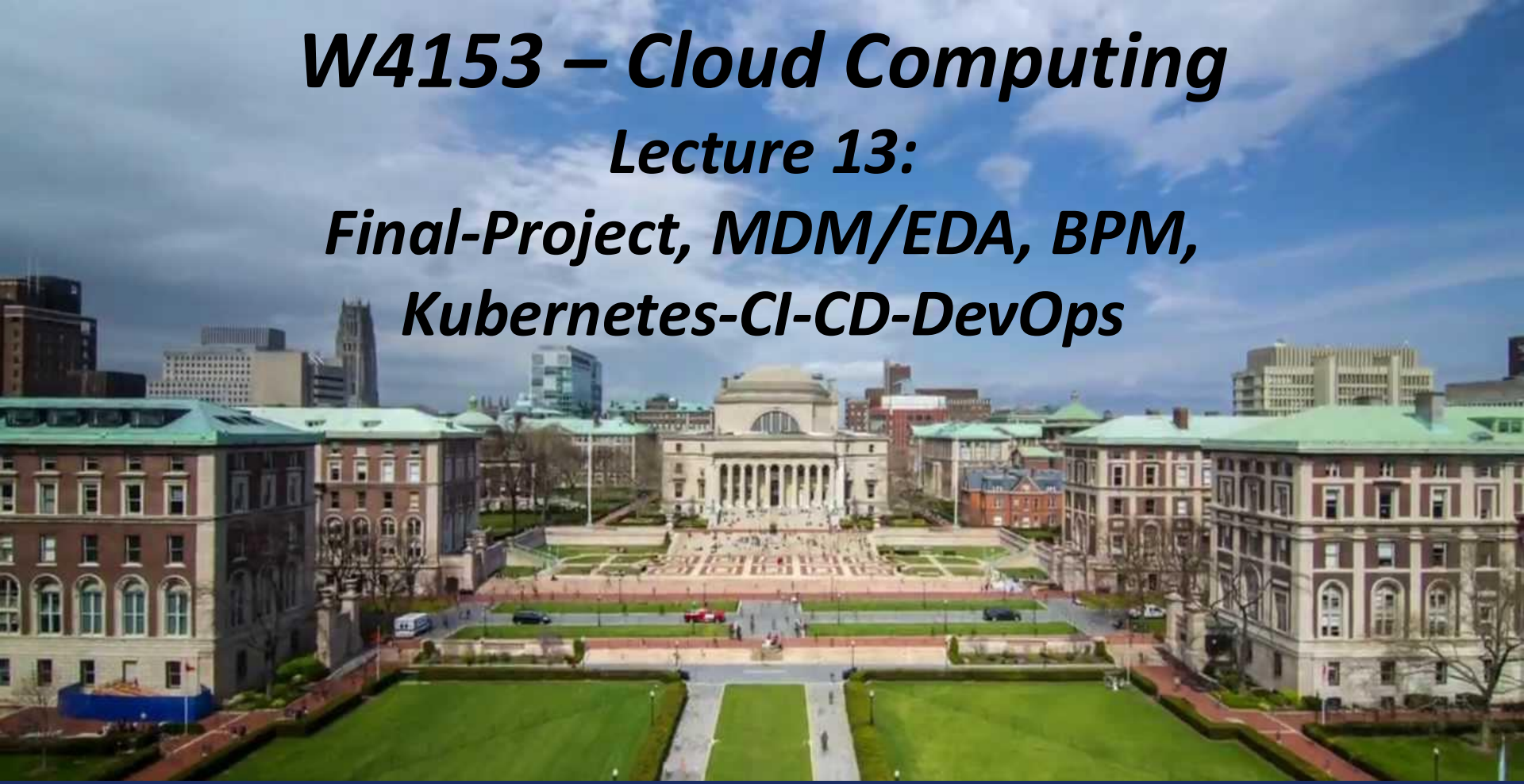
W4153 – Cloud Computing

Lecture 13:

Final-Project, MDM/EDA, BPM, Kubernetes-CI-CD-DevOps



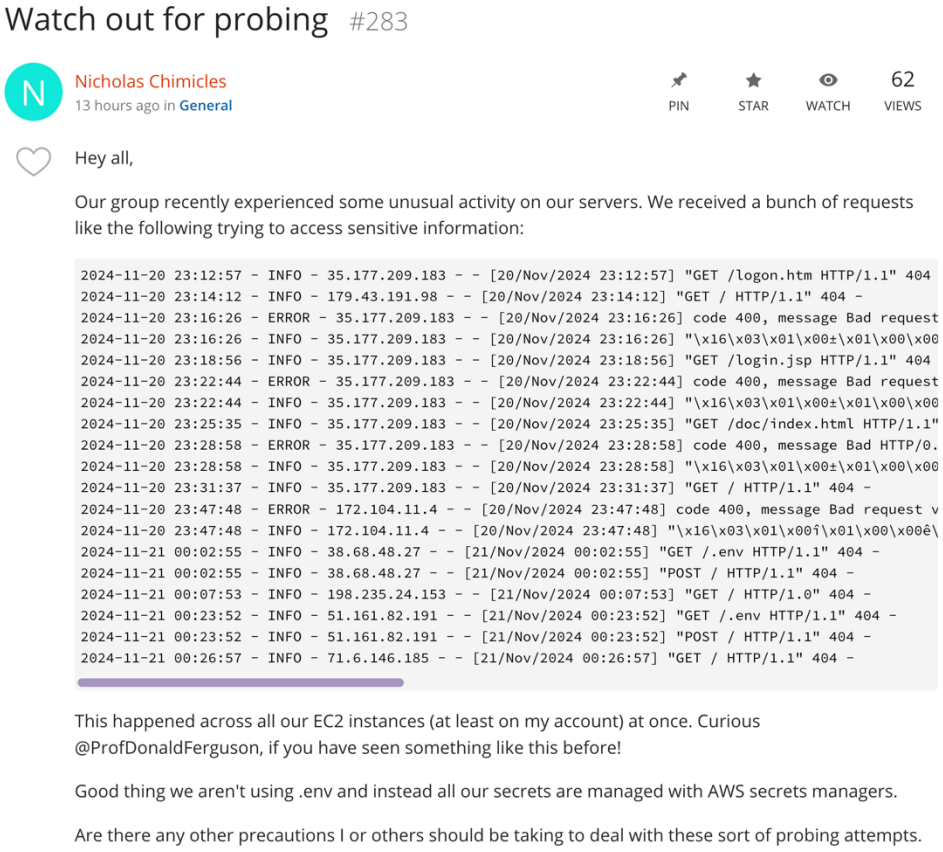
Contents

Contents

- Interesting questions on Ed
- A quick overview of multi-tenancy
- Initial final project checklist
- Advanced composition
  - Message Driven Processing
  - Event Driven Architecture
  - Workflow, service orchestration, workflow, BPM
- DevOps – A quick overview
  - Concepts
  - Infrastructure as code
  - CI/CD
- ~~Kubernetes~~
- ~~WebSocket and UI notification~~

Interesting Ed Questions

Penetration Attacks



Watch out for probing #283

Nicholas Chimicles
13 hours ago in General

PIN STAR WATCH 62 VIEWS

Hey all,

Our group recently experienced some unusual activity on our servers. We received a bunch of requests like the following trying to access sensitive information:

```
2024-11-20 23:12:57 - INFO - 35.177.209.183 - - [20/Nov/2024 23:12:57] "GET /logon.htm HTTP/1.1" 404
2024-11-20 23:14:12 - INFO - 179.43.191.98 - - [20/Nov/2024 23:14:12] "GET / HTTP/1.1" 404 -
2024-11-20 23:16:26 - ERROR - 35.177.209.183 - - [20/Nov/2024 23:16:26] code 400, message Bad request
2024-11-20 23:16:26 - INFO - 35.177.209.183 - - [20/Nov/2024 23:16:26] "\x16\x03\x01\x00±\x01\x00\x00
2024-11-20 23:18:56 - INFO - 35.177.209.183 - - [20/Nov/2024 23:18:56] "GET /login.jsp HTTP/1.1" 404
2024-11-20 23:22:44 - ERROR - 35.177.209.183 - - [20/Nov/2024 23:22:44] code 400, message Bad request
2024-11-20 23:22:44 - INFO - 35.177.209.183 - - [20/Nov/2024 23:22:44] "\x16\x03\x01\x00±\x01\x00\x00
2024-11-20 23:25:35 - INFO - 35.177.209.183 - - [20/Nov/2024 23:25:35] "GET /doc/index.html HTTP/1.1"
2024-11-20 23:28:58 - ERROR - 35.177.209.183 - - [20/Nov/2024 23:28:58] code 400, message Bad HTTP/0.
2024-11-20 23:28:58 - INFO - 35.177.209.183 - - [20/Nov/2024 23:28:58] "\x16\x03\x01\x00±\x01\x00\x00
2024-11-20 23:31:37 - INFO - 35.177.209.183 - - [20/Nov/2024 23:31:37] "GET / HTTP/1.1" 404 -
2024-11-20 23:47:48 - ERROR - 172.104.11.4 - - [20/Nov/2024 23:47:48] code 400, message Bad request v
2024-11-20 23:47:48 - INFO - 172.104.11.4 - - [20/Nov/2024 23:47:48] "\x16\x03\x01\x00î\x01\x00\x00ê\
2024-11-21 00:02:55 - INFO - 38.68.48.27 - - [21/Nov/2024 00:02:55] "GET /.env HTTP/1.1" 404 -
2024-11-21 00:02:55 - INFO - 38.68.48.27 - - [21/Nov/2024 00:02:55] "POST / HTTP/1.1" 404 -
2024-11-21 00:07:53 - INFO - 198.235.24.153 - - [21/Nov/2024 00:07:53] "GET / HTTP/1.0" 404 -
2024-11-21 00:23:52 - INFO - 51.161.82.191 - - [21/Nov/2024 00:23:52] "GET /.env HTTP/1.1" 404 -
2024-11-21 00:23:52 - INFO - 51.161.82.191 - - [21/Nov/2024 00:23:52] "POST / HTTP/1.1" 404 -
2024-11-21 00:26:57 - INFO - 71.6.146.185 - - [21/Nov/2024 00:26:57] "GET / HTTP/1.1" 404 -
```

This happened across all our EC2 instances (at least on my account) at once. Curious @ProfDonaldFerguson, if you have seen something like this before!

Good thing we aren't using .env and instead all our secrets are managed with AWS secrets managers.

Are there any other precautions I or others should be taking to deal with these sort of probing attempts.

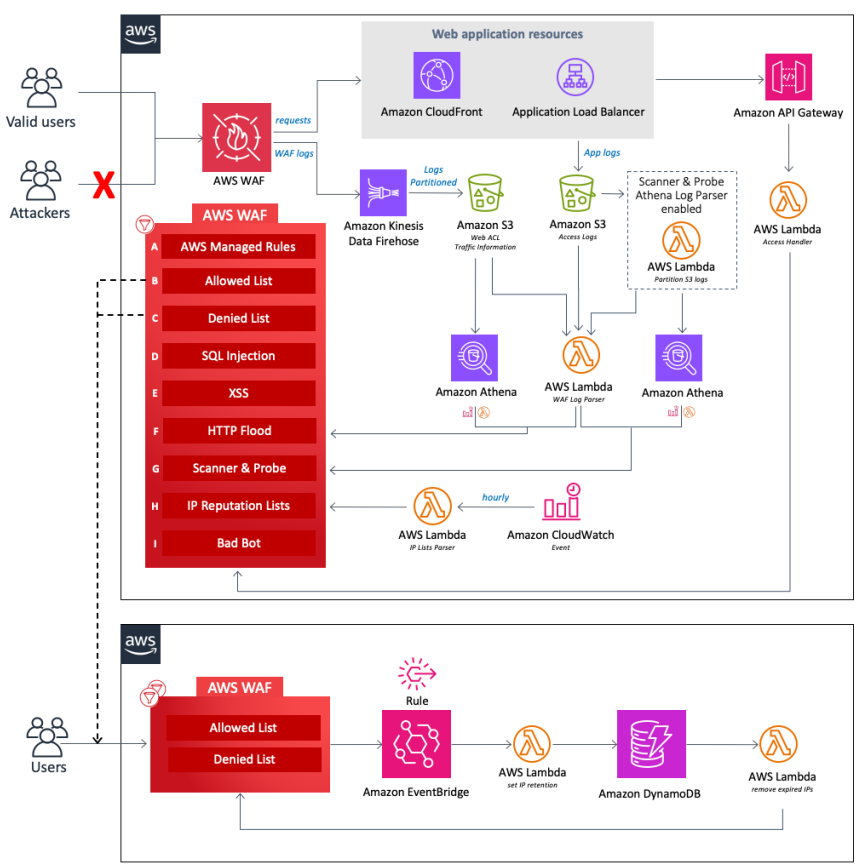
- Most organizations regularly use experts to perform a "pentest."
- If you provide an API or support a web browser application, you will be attacked. Period.
- There are many, many techniques and tools that teams use.
  - Virtual private network.
  - IP address filtering and blocking.
  - Firewalls with request analysis rules.
  - Secure software development lifecycle and tools.
  - Data encryption.
  - Governance and audit.
  - … …
- There are many documented best practices, standards, review mechanisms, etc. Independent 3rd parties perform audits.
- For now – shutdown VMs, and perhaps use IP address filtering in security groups.

Example AWS WAF

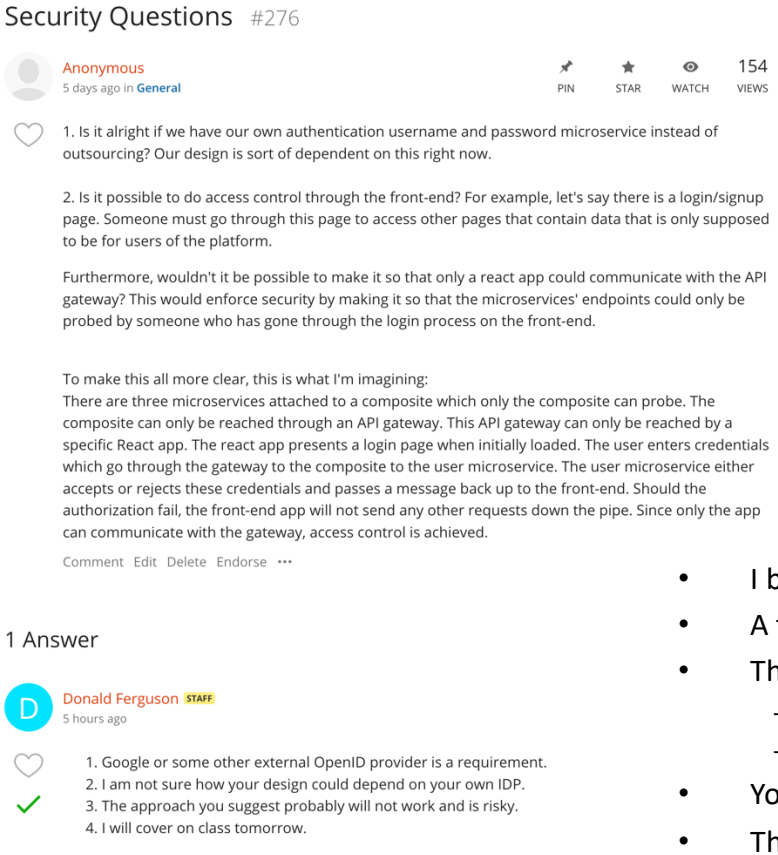
This architecture is deployed through a CloudFormation template, integrating various AWS resources to shield web applications from common attacks. Upon initial setup, users can select protective components to activate within the AWS WAF.

The AWS WAF architecture comprises several key components:

- **AWS managed rules**: Includes IP reputation rule groups and baseline rule groups designed to guard against common vulnerabilities and unwanted traffic as outlined in OWASP publications.
- **Manual IP lists**: Allows users to manually specify IPs for allowance or denial, with features for configuring IP retention and removing expired IPs via Amazon EventBridge rules and Amazon DynamoDB.
- **SQL injection and XSS protection**: Configures rules to prevent SQL injection and cross-site scripting attacks in request URIs, query strings, or bodies.
- **HTTP flood protection**: Utilizes rate-based rules or AWS Lambda functions for mitigating large volumes of requests from single IP addresses, characteristic of DDoS attacks or brute-force attempts.
- **Scanner and probe detection**: Analyzes application access logs for suspicious activities using Lambda functions or Athena queries, blocking sources that exhibit unusual error rates.
- **IP reputation lists**: Employs a Lambda function that checks third-party IP reputation lists hourly for new ranges to block, including those from Spamhaus DROP/EDROP lists, Proofpoint Emerging Threats list, and Tor exit nodes.
- **Bad bot mitigation**: Sets up a honeypot mechanism intended to attract and identify malicious bots or content scrapers, automatically blocking their source IPs upon detection.



Login and Identity Provider



Security Questions #276

Anonymous
5 days ago in General

PIN STAR WATCH 154 VIEWS

1. Is it alright if we have our own authentication username and password microservice instead of outsourcing? Our design is sort of dependent on this right now.

2. Is it possible to do access control through the front-end? For example, let's say there is a login/signup page. Someone must go through this page to access other pages that contain data that is only supposed to be for users of the platform.

Furthermore, wouldn't it be possible to make it so that only a react app could communicate with the API gateway? This would enforce security by making it so that the microservices' endpoints could only be probed by someone who has gone through the login process on the front-end.

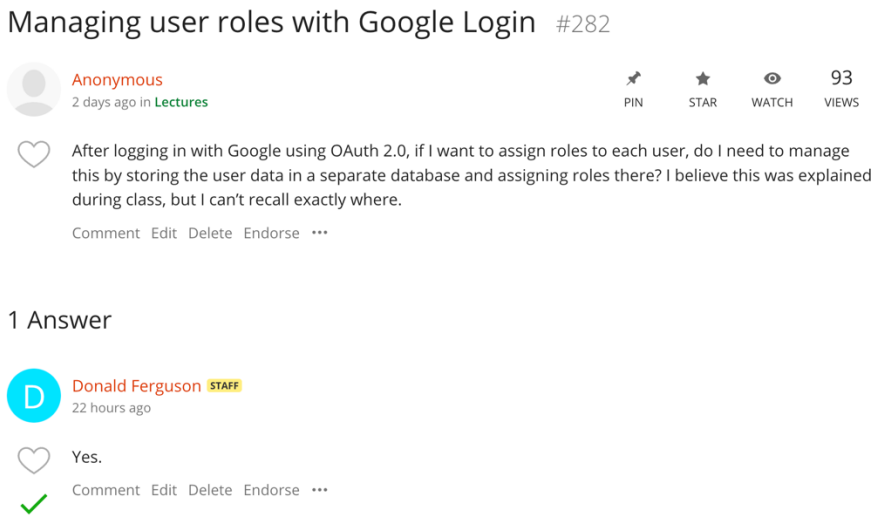
To make this all more clear, this is what I'm imagining:
There are three microservices attached to a composite which only the composite can probe. The composite can only be reached through an API gateway. This API gateway can only be reached by a specific React app. The react app presents a login page when initially loaded. The user enters credentials which go through the gateway to the composite to the user microservice. The user microservice either accepts or rejects these credentials and passes a message back up to the front-end. Should the authorization fail, the front-end app will not send any other requests down the pipe. Since only the app can communicate with the gateway, access control is achieved.

Comment Edit Delete Endorse •••

1 Answer

Donald Ferguson STAFF
5 hours ago

1. Google or some other external OpenID provider is a requirement.
2. I am not sure how your design could depend on your own IDP.
3. The approach you suggest probably will not work and is risky.
4. I will cover on class tomorrow.



Managing user roles with Google Login #282

Anonymous
2 days ago in Lectures

PIN STAR WATCH 93 VIEWS

After logging in with Google using OAuth 2.0, if I want to assign roles to each user, do I need to manage this by storing the user data in a separate database and assigning roles there? I believe this was explained during class, but I can't recall exactly where.

Comment Edit Delete Endorse •••

1 Answer

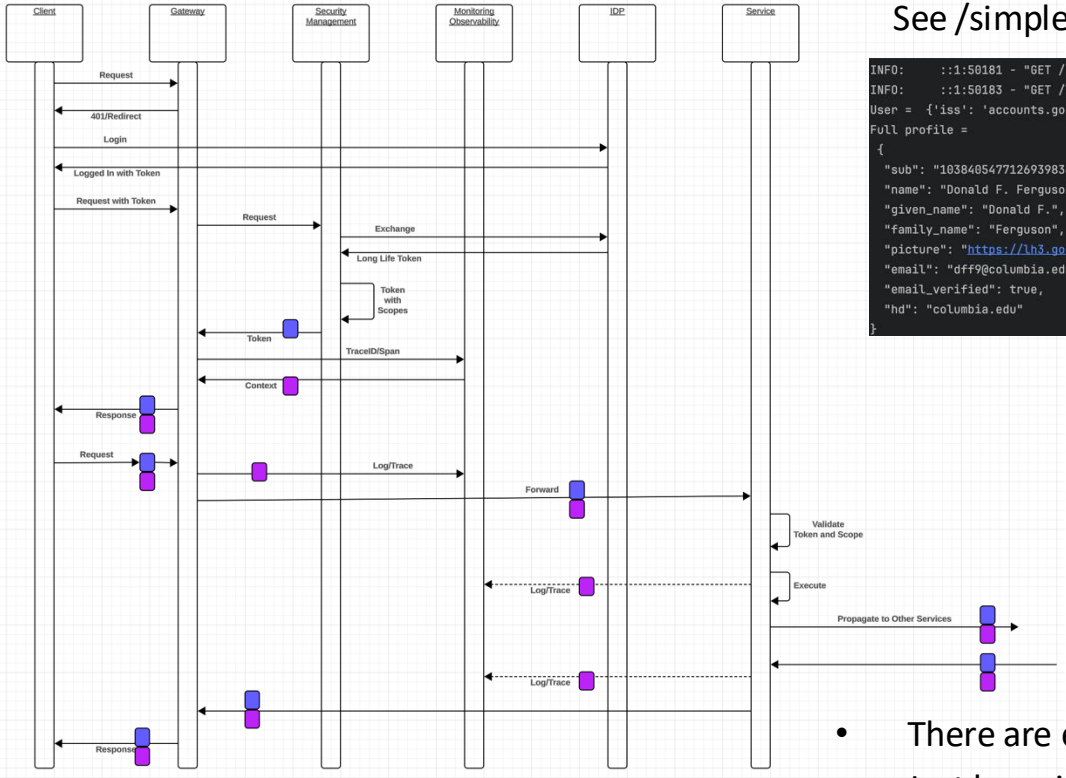
Donald Ferguson STAFF
22 hours ago

Yes.

Comment Edit Delete Endorse •••

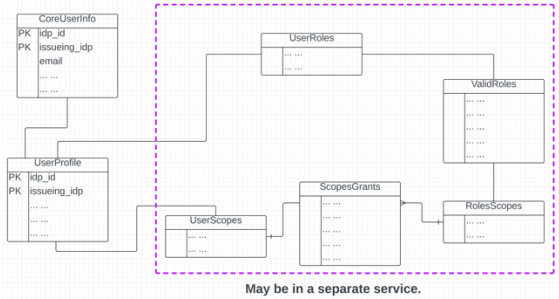
- I briefly covered this in class last week.
- A fundamental concept is *resource.*
- The fundamental concept in authorization is a set of rules of the form:
  - (role, resource, operation)
  - Remember the SQL GRANT/REVOKE from database class.
- You perform the check in middleware or layered service.
- The check can be a "database lookup" or generating signed claims during login.

Reminder



See /simple_examples/google_login in course repo.

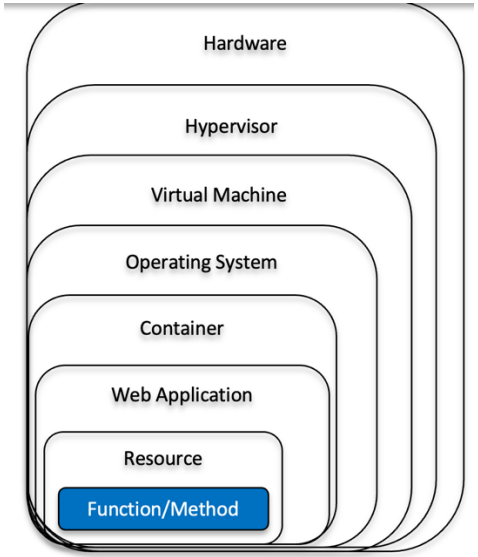
```
INFO:     ::1:50181 - "GET / HTTP/1.1" 200 OK
INFO:     ::1:50183 - "GET /login HTTP/1.1" 302 Found
User =  {'iss': 'accounts.google.com', 'azp': '669170329085-a4dic4vjrgmsrs62al
Full profile =
 {
  "sub": "103840547712693983622",
  "name": "Donald F. Ferguson",
  "given_name": "Donald F.",
  "family_name": "Ferguson",
  "picture": "https://lh3.googleusercontent.com/a/ACg8ocL1tN2xNaywLA0I0oVdQZHp
  "email": "dff9@columbia.edu",
  "email_verified": true,
  "hd": "columbia.edu"
}
```



- There are examples, including use of middleware, JWT, … …
- Just keep it simple.
- In general, each microservice performs the checks.

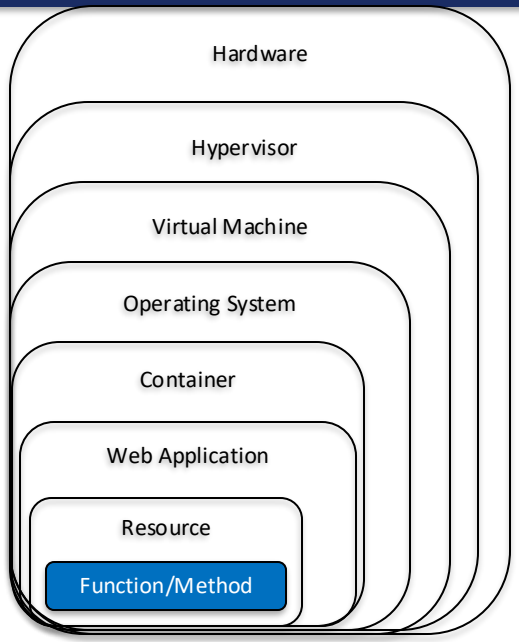
Bootstrapping Credentials

- Your code “runs” inside something, which often runs inside something else.
- A controller/manager/… launches the ”thing” that contains the code or other things.
- “Code” also has an identity, role and corresponding roles.
- There are various, “thing specific” ways of configuring, injecting and setting
  - Identity/credentials for an executing environment.
  - Environment variables.
  - … …
- You can use something simple, e.g. environment variables set for a VM in the console, … … I did something in the app for issuing your Google tokens.



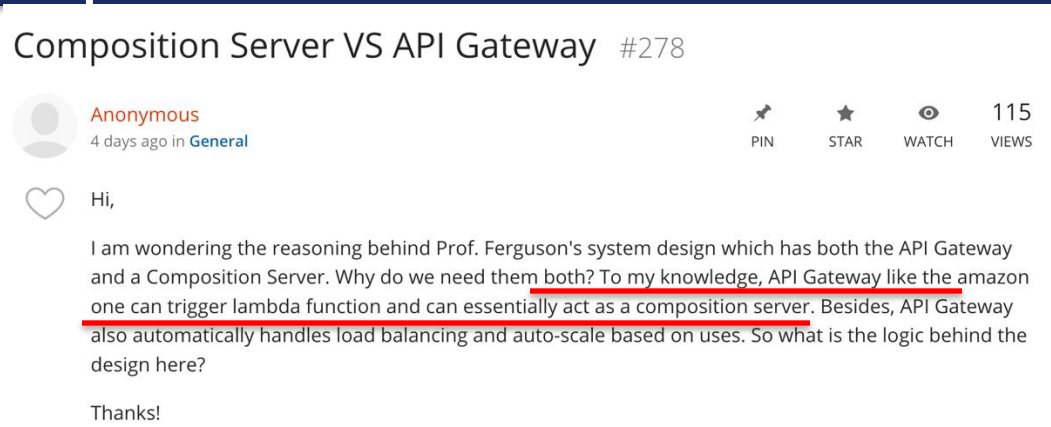
Computer Models

- At the very core, you write a function or method.
- A set of functions or methods become a resource.
- A resource runs in a web application and uses required software, e. g. packages, libraries.
- A web application run in some kind of container, which provides isolation, ensures pre-reqs, etc.
- Containers run in an operating system.
- An operating systems runs on HW, which may be a virtual machine and virtual resources.
- Virtual resources run on hypervisors, which
- Run on hardware.
- This is like a *matryoshka doll*. Ultimately, I decide:
  - Which level of nesting is the package I am willing to/want to produce and deploy.
  - I provide that as a package/bundle and provide a declaration/manifest of what I want created and managed automatically for me.
- This is over simplified and the boundaries blur.



API Gateway versus Composition

- First, in your scenario,
  - The gateway is implementing request routing.
  - You are implementing the composition service as a bunch of Lambda functions.



Composition Server VS API Gateway #278

Anonymous
4 days ago in General

PIN STAR WATCH 115 VIEWS

Hi,

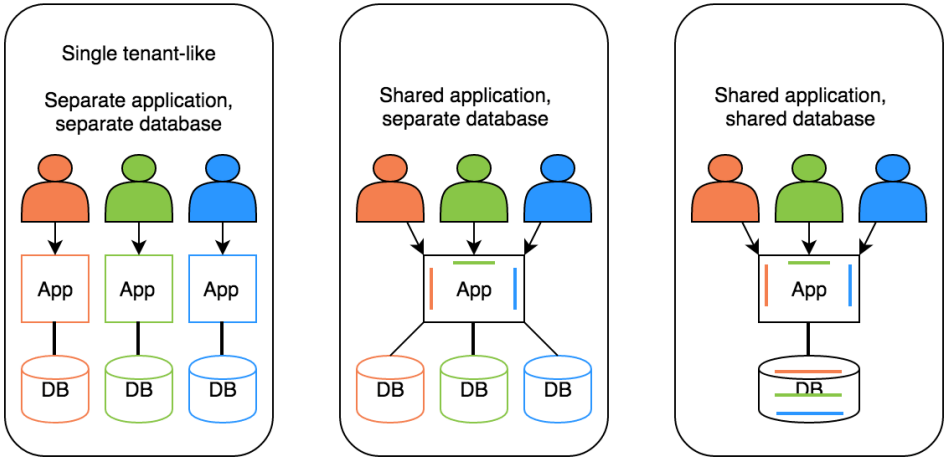
I am wondering the reasoning behind Prof. Ferguson's system design which has both the API Gateway and a Composition Server. Why do we need them both? To my knowledge, API Gateway like the amazon one can trigger lambda function and can essentially act as a composition server. Besides, API Gateway also automatically handles load balancing and auto-scale based on uses. So what is the logic behind the design here?

Thanks!

- Ferguson's 7th Law of SW Architecture: "A runtime infrastructure must be a floor wax or a mouth wash. Nothing is good at both."
- The functionality/requirements for something that governs and controls a large set of API endpoints and applies complex policies versus something that is good at running complex application logic are vastly different ➔ No "thing" will be good at both.
- Composition is a statement about *what functions* the microservice implements relative to other microservices and is not a statement about "what it runs in." Composite logic can be relatively complex, especially when errors occur.

Multitenant Applications

Multitenant Applications



A. SaaS concepts and terminology

In the Software as a Service (SaaS) model, your company doesn't sell *licenses* to your software. Instead, each customer makes rent payments to your company, making each customer a *tenant* of your company.

In return for paying rent, each tenant receives access to your SaaS application components, and has its data stored in the SaaS system.

The term *tenancy model* refers to how tenants' stored data is organized:

- *Single-tenancy:* Each database stores data from only one tenant.
- *Multi-tenancy:* Each database stores data from multiple separate tenants (with mechanisms to protect data privacy).
- Hybrid tenancy models are also available.

- Just be aware.
- How to build a multitenant system is a complete semester itself.
- No need to include in the project.

B. How to choose the appropriate tenancy model

In general, the tenancy model doesn't affect the function of an application, but it likely impacts other aspects of the overall solution. The following criteria are used to assess each of the models:

- **Scalability:**
  - Number of tenants.
  - Storage per-tenant.
  - Storage in aggregate.
  - Workload.
- **Tenant isolation:** Data isolation and performance (whether one tenant's workload impacts others).
- **Per-tenant cost:** Database costs.
- **Development complexity:**
  - Changes to schema.
  - Changes to queries (required by the pattern).
- **Operational complexity:**
  - Monitoring and managing performance.
  - Schema management.
  - Restoring a tenant.
  - Disaster recovery.
- **Customizability:** Ease of supporting schema customizations that are either tenant-specific or tenant class-specific.

https://learn.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns?view=azuresql

Final Project Checklist

Initial Final Project Checklist

We will finalize in discussion with TAs, student feedback, etc.
Finalizing will include being more precise about requirements.

1. API gateway with basic routing and authorization check.
2. Security:
    1. Google login or equivalent.
    2. Issue own JWT tokens with “grants.”
    3. Validate tokens in middleware and propagate on calls to other services.
3. Observability:
    1. Basic log and trace in microservices.
    2. Correlation ID and propagation.
    3. Middleware.
    4. Demonstrate in a cloud’s logging tools.
4. At least 3 basic/atomic microservices. There must be at least one microservice deployed:
    1. Directly on a VM.
    2. In a Docker container running on a VM.
    3. Running in PaaS.
5. REST API:
    1. OpenAPI documents for all microservices.
    2. Documented and followed a basic REST API best practices guidance.
    3. HATEOAS and support for links.
    4. Pagination for at least one microservice.
    5. For POST, at least one microservice should implement 201 created with a link header.
    6. At least one path must implement an asynchronous execution pattern, e.g. 202 Accepted.
6. At least one composite service that composes the behavior of the atomic services. At least one use of
    1. Composition using code and synchronous API calls.
    2. Composition using code and asynchronous API calls.
    3. Service choreography using pub/sub.
    4. Service orchestration using StepFunctions, workflow, … …
7. Calling/using at least one external cloud service.

Initial Final Project Checklist

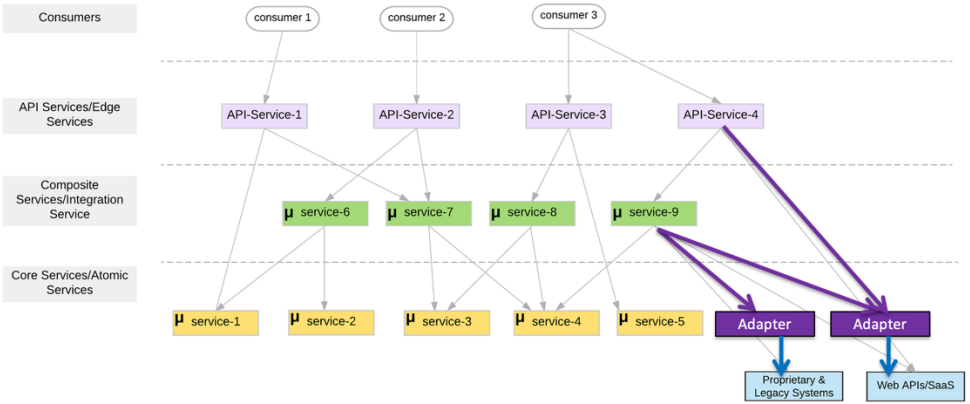
We will finalize in discussion with TAs, student feedback, etc.
Finalizing will include being more precise about requirements.

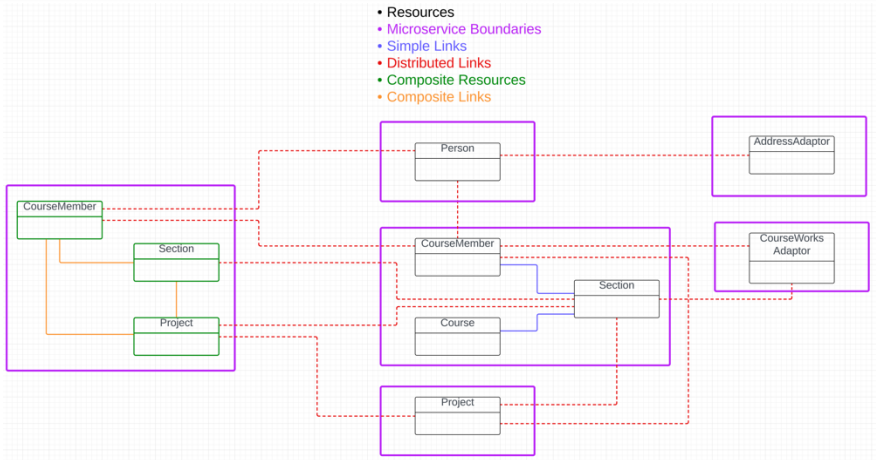
8. End-user notification – At least one example of
    1. A FaaS reacting to an event.
    2. Some form of end-user notification, e.g.
        1. Discord message
        2. Email
        3. … …
9. GitHub
    1. A GitHub repository for each microservice.
    2. A project that shows the subordinate microservices.
10. CI/CD – At least one example of GitHub or other actions that deploys a microservice on commit.
11. UI/UX
    1. Simple browser web UI
    2. Deployed and delivered from a cloud “blob store.”
12. CQRS
    8. Simple example of a GraphQL endpoint querying a read only copy of the combined data model.

Advanced Service Composition

Concepts
Reminder and Overview

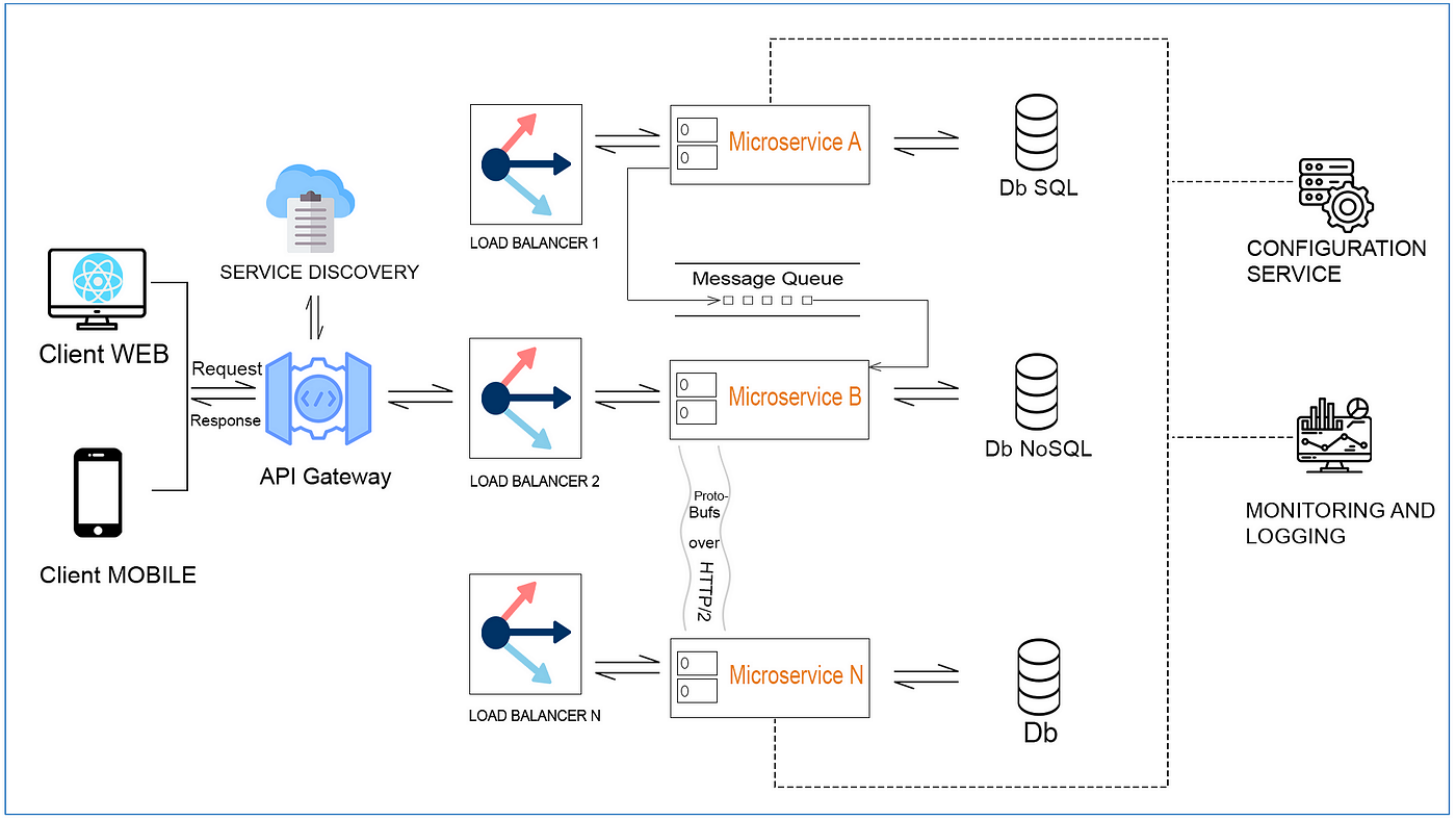
Composition Concepts and Reminder





- There are two aspects to "composition:"
  - Structure: How do I "link things together" to "find services" that a microservice needs?
  - Behavior: How do I implement the application of logic of the composite?
    - Two models: *Orchestration* and *Choreography.*
    - Several approaches to implementing the service methods: synchronous code, asynchronous code, event driven architecture, message driven architecture, state machines/workflows, … …
    - Use pattern like CQRS to "optimize" queries?

One Approach

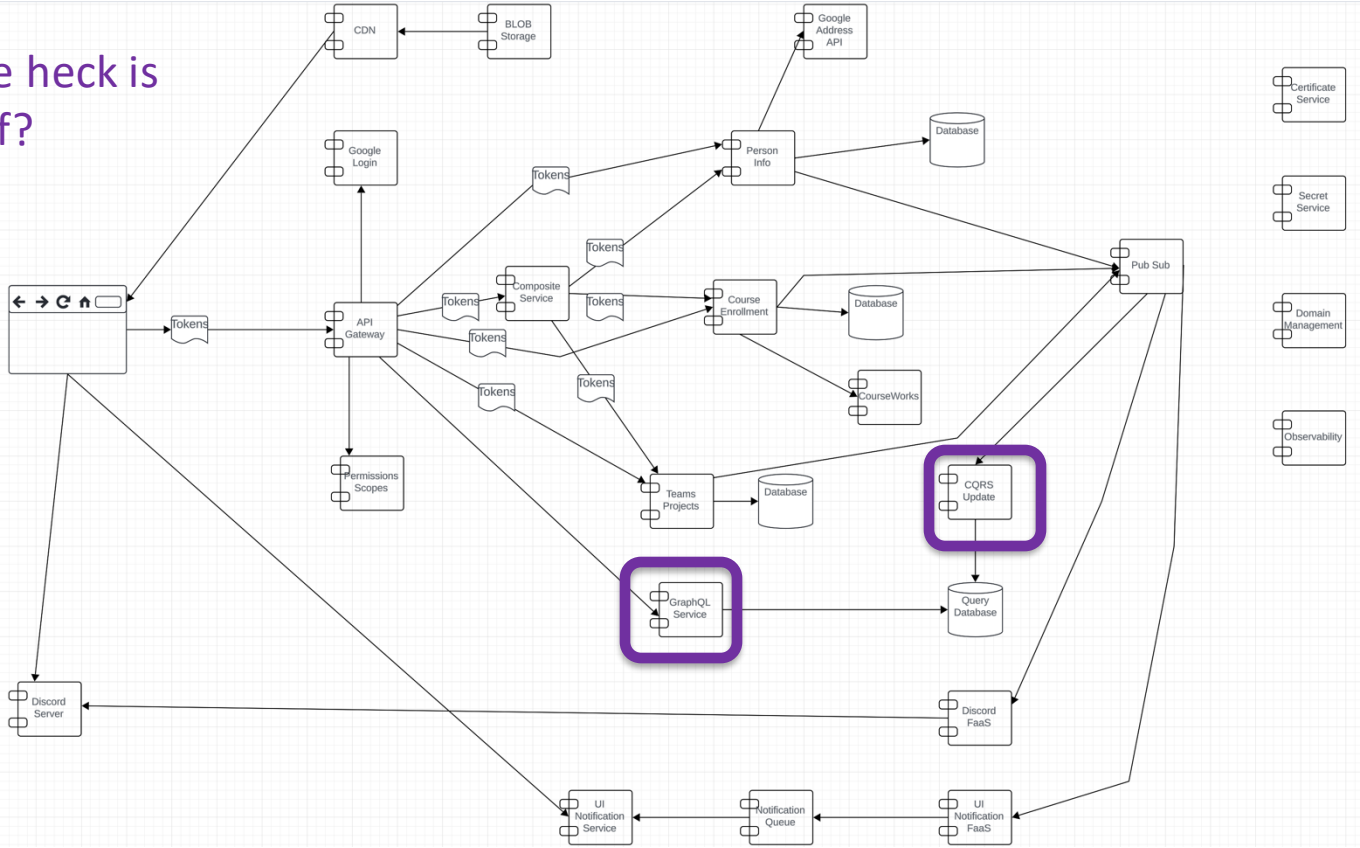


https://medium.com/technology-hits/microservices-simplified-8f19b7990fa3

CQRS
and
GraphQL

CQRS and GraphQL

What the heck is that stuff?



CQRS

- We have seen that microservices leads to little bits of “data” all over the place.
- Something that a user thinks of as a “thing” is scattered all over the place.
- We have seen how to handle this issue using various patterns for composition.
- But,
  - Most of the time things are not changing.
  - The vast majority of requests are reads.
  - I would like something more view like and with better query support than just query string in URLs.

Pattern: Command Query Responsibility Segregation (CQRS)

pattern | service collaboration | implementing queries

Context

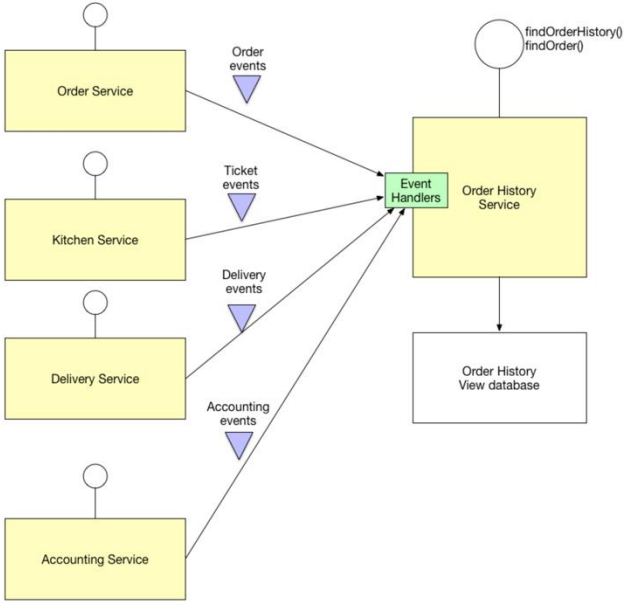
You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services. Also, if you have applied the Event sourcing pattern then the data is no longer easily queried.

Problem

How to implement a query that retrieves data from multiple services in a microservice architecture?

Solution

Define a view database, which is a read-only ‘replica’ that is designed specifically to support that query, or a group related queries. The application keeps the database up to date by subscribing to Domain events published by the service that own the data. The type of database and its schema are optimized for the query or queries. It’s often a NoSQL database, such as a document database or a key-value store.



REST vs GraphQL in a Nutshell

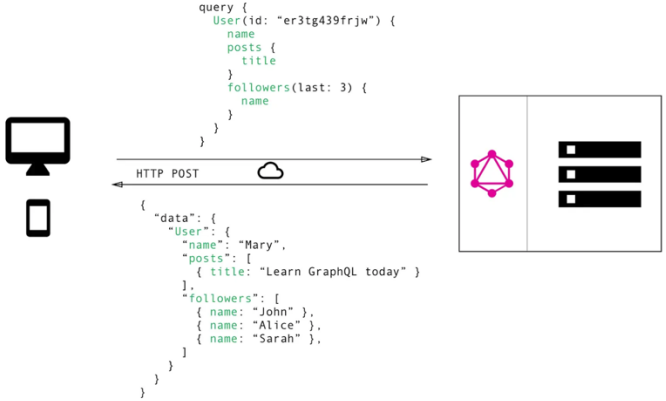
Data fetching with REST vs GraphQL

With a REST API, we'll probably gather data through multiple endpoints. These endpoints could be

- **/users/<id>** — to fetch the initial users data
- **/users/<id>/posts** — to fetch all the posts of the user
- **/users/<id>/followers** — to fetch list of followers of the user

With REST, we will have to make three requests to different endpoints to get the required data. Also, there will be a problem of overfetching as excess data will be received than what is needed. So the data has to be formatted on the client-side before it can be rendered.

In GraphQL, a single query will be sent to the GraphQL server that specifies the exact data requirements. The server sends back the response with the desired data.



Client can specify exactly the data it needs in the query. And the server responds in the exact format as the nested query.

GraphQL Concepte

Here are some core concepts in GraphQL:

- Schema: GraphQL uses schemas to define the types of operations or queries that can be performed. The schema acts as an interface between the client and the server to fetch or modify data.
- Types: GraphQL, like REST, has various data types in an API. These include String, Int, Boolean, or custom object types that the developer defines. Each field of a GraphQL type has its own type. For example, in your GraphQL, the book_name field of the Book type is a String, and the publish_date field is an Int. Is that clear?
- Queries: GraphQL is a query language that lets clients specify the shape and content of the data they want from the server. Clients can use GraphQL to request specific fields and nested data in their queries, and the server will respond with data that matches the same structure as the queries. This enables clients to fetch only the data they need, avoiding unnecessary or excessive requests to the server.
- ~~Mutations: Mutations are a way of modifying data with GraphQL. Unlike queries, which are mainly for retrieving data from the server, mutations are for creating, updating, or deleting data on the server. Mutations ensure that the data written to the server is predictable.~~
- ~~Subscriptions: Subscriptions in GraphQL enable real-time updates such as notifications. Clients can use subscriptions to listen for specific events or actions on the server. When those events or actions occur, the server sends notifications to the subscribed clients.~~
- Resolvers: A resolver is a function that connects a schema field to a data source, where it can fetch or modify data according to the query. Resolvers are the bridge between the schema and the data, and they handle the logic for data manipulation and retrieval.

GraphQL

- Show the GraphQL Overviews
  - REST vs. GraphQL: A Detailed Comparison of API Architectures for Developers
  - GraphQL: An Introduction
  - FastAPI with GraphQL: Delving Deep into Modern API Development
  - Lectures/W4153-2024F-12-Sample-Project-EDA:MDP-AdvancedComposition-APIGW-GraphQL-Kubernetes/generalpresentation1-180129061029.pdf

- How this comes together:
  - A script (or scripts) creates the query schema and database, and loads the data.
  - Microservices emit a resource change event via middleware to a topic, e.g. “person_change_event.”
  - A FaaS instance reacts to the event and updates “the copy.”
  - Simple, very simple, GraphQL service for query. You only need to do resolvers.
  - Links in query enable update of the underlying data through the composites.

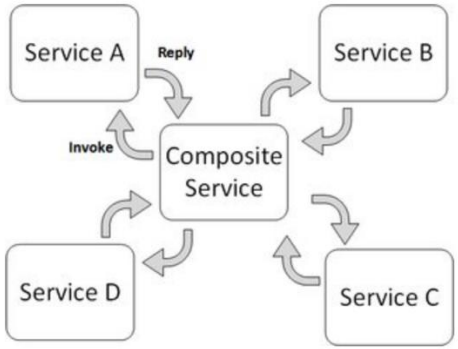
Updates and Sagas

Concepts

Service orchestration

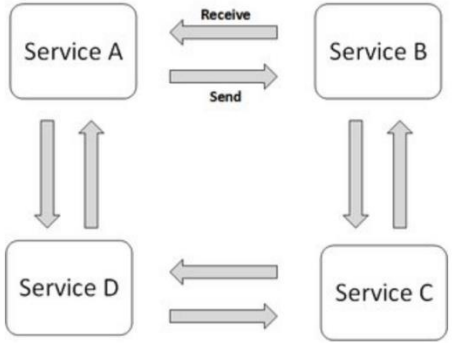
Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

The relationship between all the participating services are described by a single endpoint (i.e the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.



Service Choreography

Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. Choreography employs a decentralized approach for service composition.

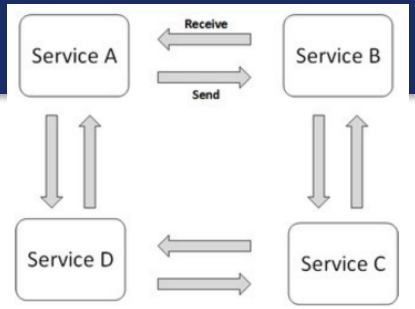


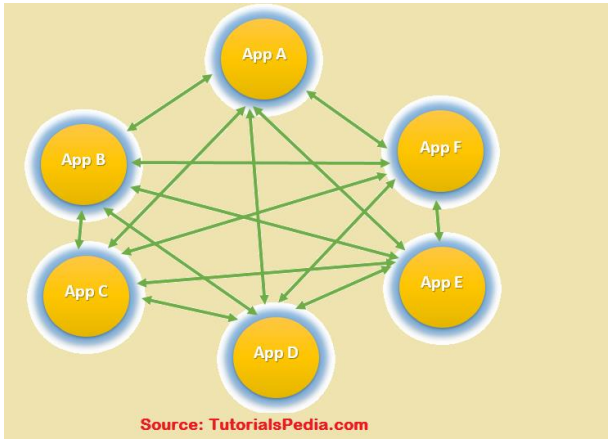
The choreography describes the interactions between multiple services, where as orchestration represents control from one party's perspective. This means that a **choreography** *differs* from an **orchestration** with respect to where the logic that controls the interactions between the services involved should reside.

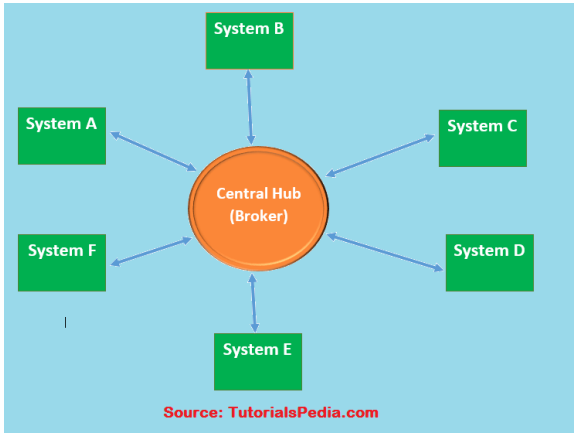
These are not formally, rigorously defined terms.

Choreography

- We saw the basic diagram, but …
  this is an anti-pattern and does not scale.

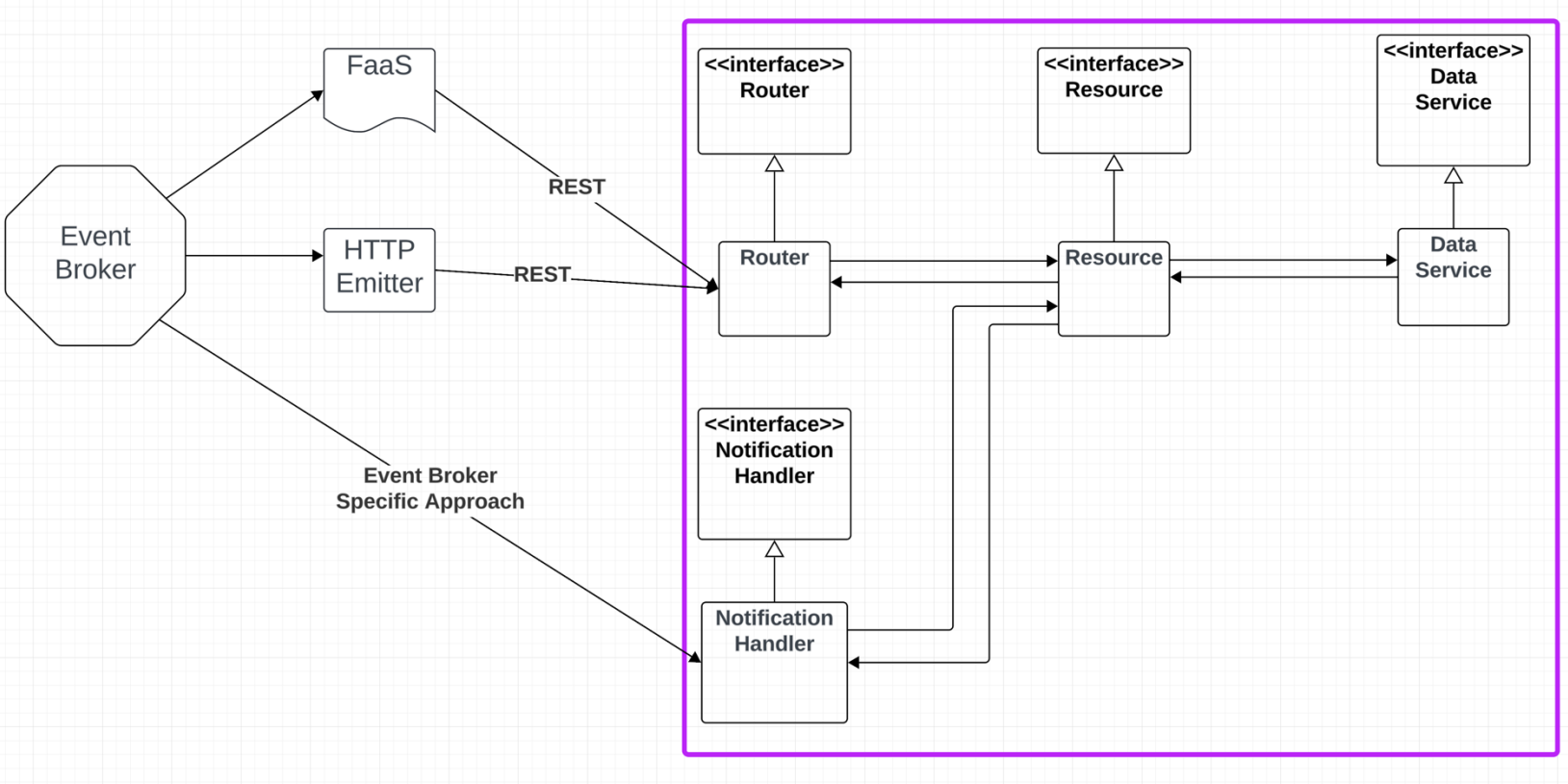




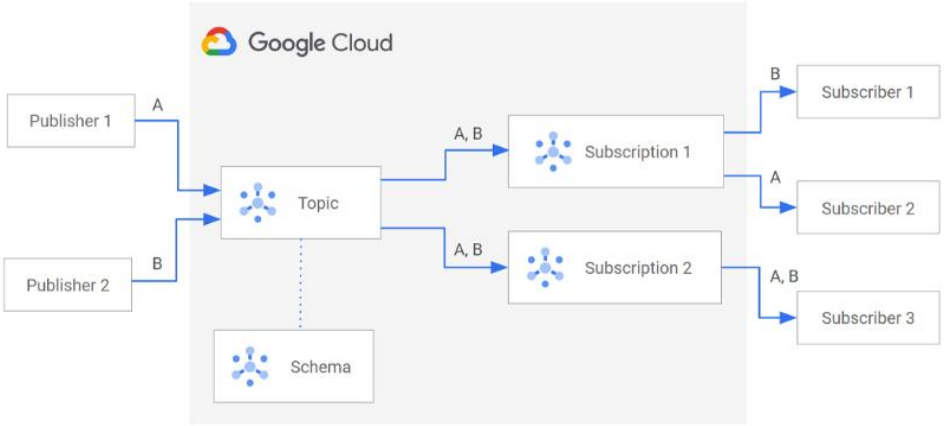


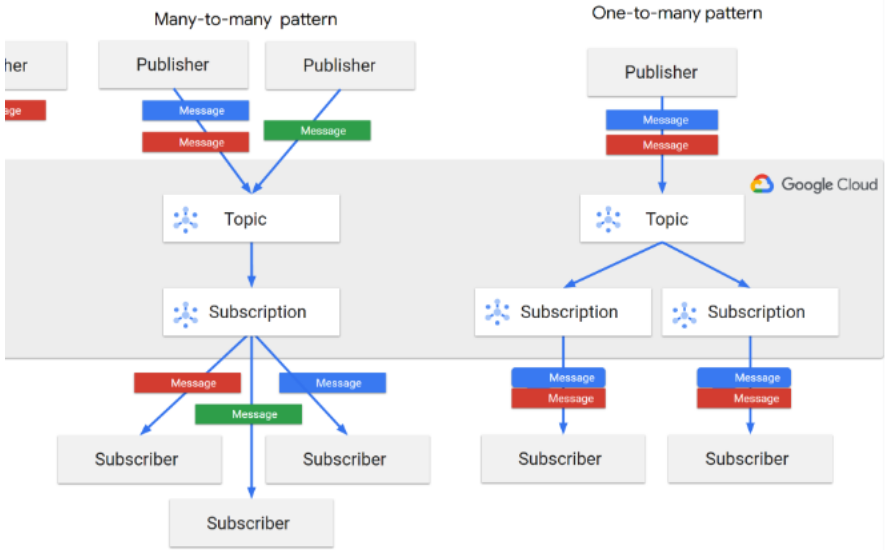
- But, how do you write the event driven microservices?
  - Well, you can just write code … …
  - Or use a state machines abstraction.
  - … …

Conceptual Approach



Google Pub Sub

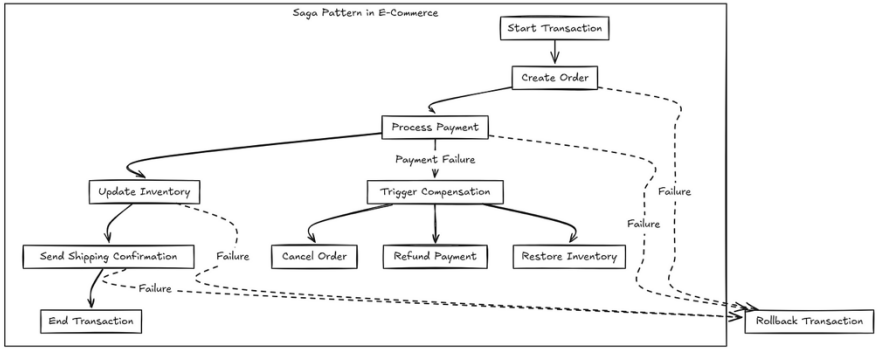


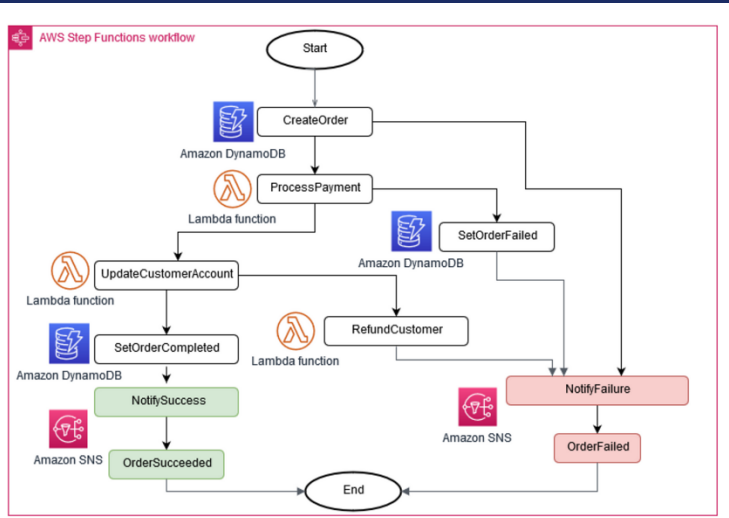


The following are the components of a Pub/Sub service:

- **Publisher** (also called a producer): creates messages and sends (put specified topic.
- **Message**: the data that moves through the service.
- **Topic**: a named entity that represents a feed of messages.
- **Schema**: a named entity that governs the data format of a Pub/Sub message.
- **Subscription**: a named entity that represents an interest in receiving messages on a particular topic.
- **Subscriber** (also called a consumer): receives messages on a specified subscription.

Service Orchestration and Saga

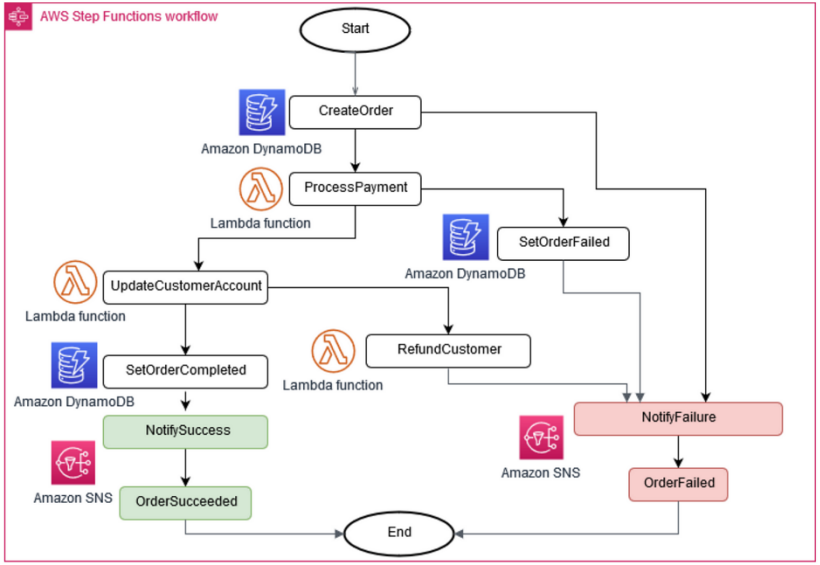


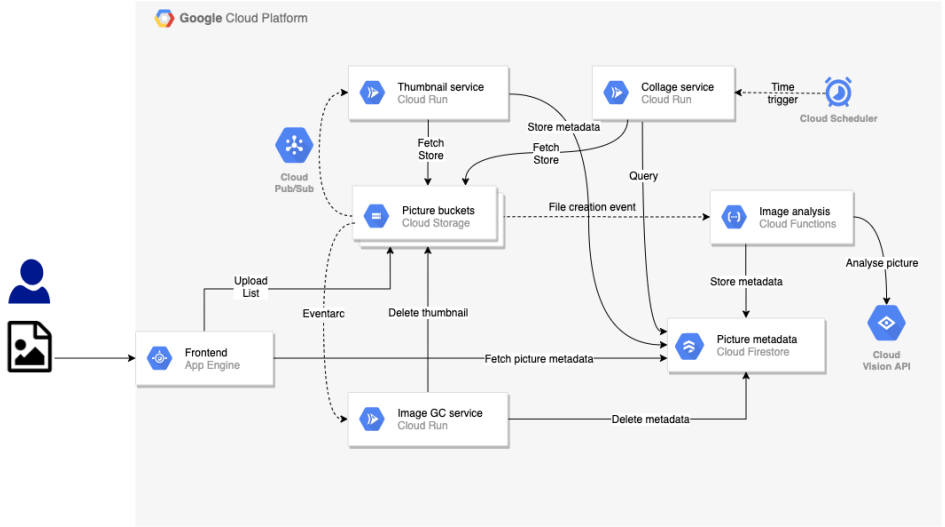


Saga (https://microservices.io/patterns/data/saga.html):

- You have applied the Database per Service pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services.
- Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.
- There are two ways of coordination sagas:
  - Choreography - each local transaction publishes domain events that trigger local transactions in other services
  - Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Orchestration





Both Google and AWS have approaches to “workflow” and “orchestration:”

- Development and management studios.
- High level language.
- Execution engine.

Summary

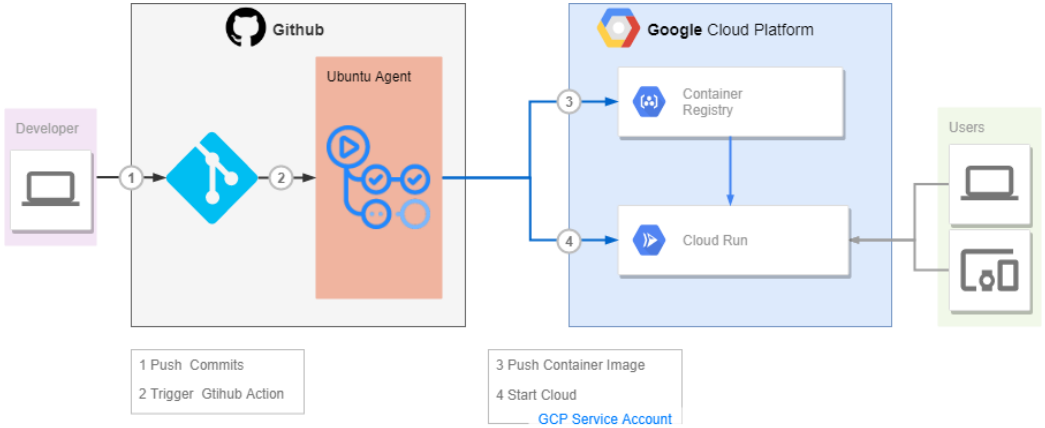
- You have implemented composition using code:
  - Synchronous
  - Asynchronous
- You will do something simple for some composite updates that implements
  - Choreography
    - Pub/Sub
    - Topics
    - Events and subscriptions
    - ”Do” and “Rollback” functions
  - Orchestration with compensation using one of the engines
  - Access via an API
  - Something very simple

DevOps

Some Terminology (From Wikipedia)

- **DevOps** is a methodology in the software development and IT industry. Used as a set of practices and tools, DevOps integrates and automates the work of software development (Dev) and IT operations (Ops) as a means for improving and shortening the systems development life cycle. DevOps is complementary to agile software development; several DevOps aspects came from the agile way of working.
- In software engineering, **CI/CD** or CICD is the combined practices of continuous integration (CI) and continuous delivery (CD) or, less often, continuous deployment. They are sometimes referred to collectively as continuous development or continuous software development. Components:
  - Continuous integration: Frequent merging of several small changes into a main branch.
  - Continuous delivery: Producing software in short cycles with high speed and frequency so that reliable software can be released at any time, with a simple and repeatable deployment process when deciding to deploy.
  - Continuous deployment: Automatic rollout of new software functionality.
- **Infrastructure as code (IaC)** is the process of managing and provisioning computer data center resources through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. The IT infrastructure managed by this process comprises both physical equipment, such as bare-metal servers, as well as virtual machines, and associated configuration resources.

GitHub Actions



Introduction

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that run tests whenever you push a change to your repository, or that deploy merged pull requests to production.

This quickstart guide shows you how to use the user interface of GitHub to add a workflow that demonstrates some of the essential features of GitHub Actions.

To get started with preconfigured workflows, browse through the list of templates in the actions/starter-workflows repository. For more information, see "Using workflow templates."

For an overview of GitHub Actions workflows, see "About workflows." If you want to learn about the various components that make up GitHub Actions, see "Understanding GitHub Actions."

```yaml
name: Deploy
on:
  push:
    branches:
      - master

jobs:
  build:
    name: Cloud Run Deployment
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@master

      - name: Setup GCP Service Account
        uses: google-github-actions/setup-gcloud@master
        with:
          version: "latest"
          service_account_email: ${{ secrets.GCP_SA_EMAIL }}
          service_account_key: ${{ secrets.GCP_SA_KEY }}
          export_default_credentials: true

      - name: Configure Docker
        run: |
          gcloud auth configure-docker
      - name: Build
        run: |
          docker build -t gcr.io/${{ secrets.GCP_PROJECT_ID }}/<image-name>:latest .
      - name: Push
        run: |
          docker push gcr.io/${{ secrets.GCP_PROJECT_ID }}/<image-name>:latest
      - name: Deploy Service
        run: |
          gcloud run deploy path \
          --region us-central1 \
          --image gcr.io/${{ secrets.GCP_PROJECT_ID }}/<image-name> \
          --platform managed \
          --allow-unauthenticated \
          --memory 256Mi \
          --port 443 \
          --project ${{ secrets.GCP_PROJECT_ID }}
```

action.yaml hosted with ❤ by GitHub view raw