cscc01 (summer 2021)
introduction to software engineering

tutorial 7

microservices

tutorial outline

01 microservices

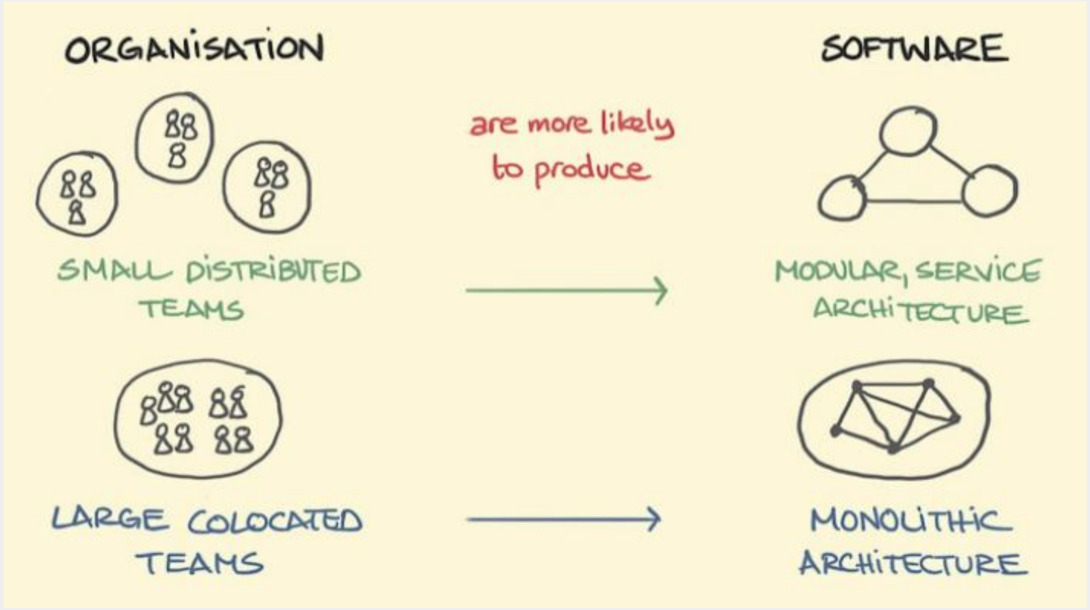
02 api gateways

03 graphql

04 event-driven architecture

conway's law

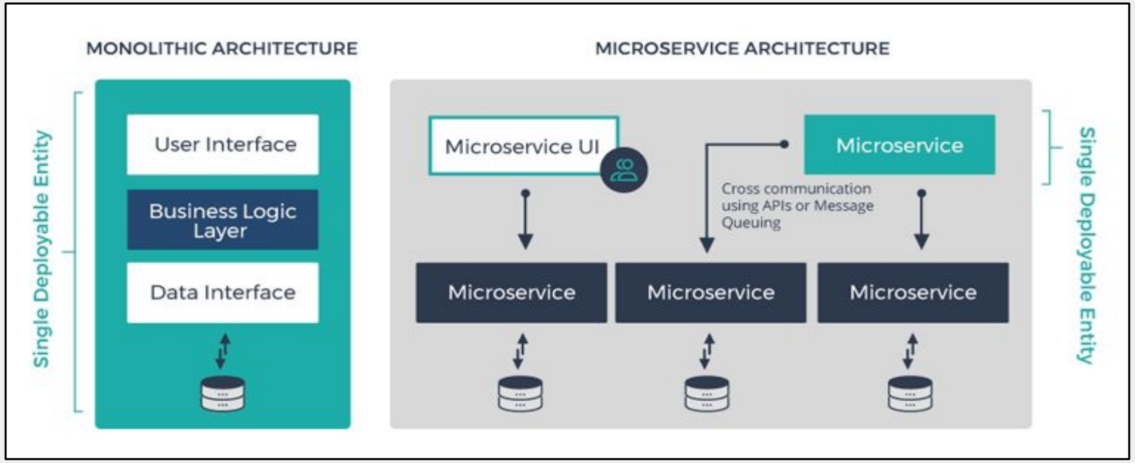
The structure of software will mirror the structure of the organization that built it



monoliths vs. microservices

Monolith: Application components (e.g. UI, Database, Business Logic) are coupled together under a single program. These are the types of programs you've been developing in courses, even A1!

Microservice: Application components are decoupled and modularized into services to perform a single piece of functionality. The services connect to form the final program



monoliths vs. microservices

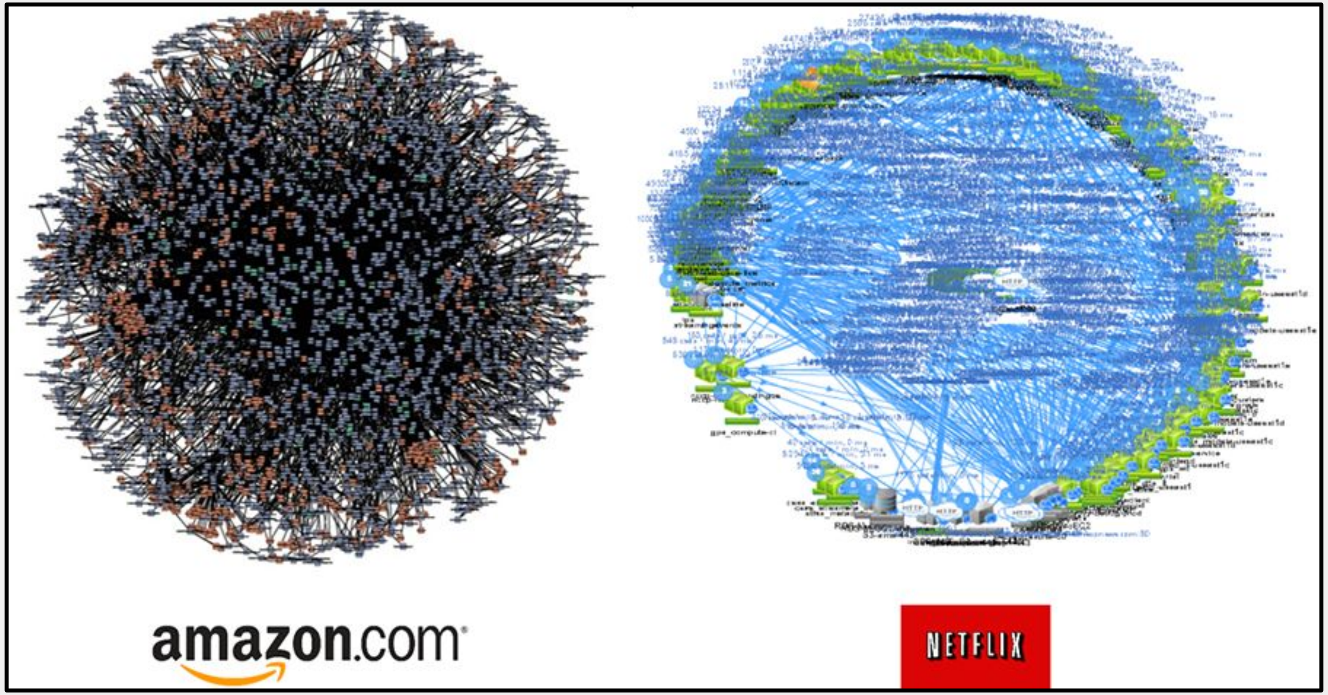
Microservices are an architectural design pattern, so they are not inherently better than monoliths. Don’t force the pattern, use it when it’s applicable! Most applications start out as monoliths.

Solves specific problems:

- Scaling
- Team Dependency
- Autonomy
  - Teams can use the right tool for the right job i.e. SQL vs. NoSQL

Large organizations opt for microservices because they tackle the issue of scaling very well (millions of requests, thousands of engineers).

they can get complex



Since they can get complex, we'll need the right tools for the job

backend requirements

When deploying backend services, what do we want to achieve?

- We want our client to be able to send requests to the backend, performing the expected behaviour
- We want to protect our APIs from overuse & abuse
  - Authentication & rate limiting
- We want to understand how our APIs are being used
  - Analytics/monitoring tools allowing us to analyze user data
- We want rapid deployment

tools

As such, software engineering teams often choose to employ these tools accompanying a microservices infrastructure

- API gateway: api management service to handle requests
  - GraphQL
- CI/CD (see tut 4)
- Containerization (see tut 6)

api gateway

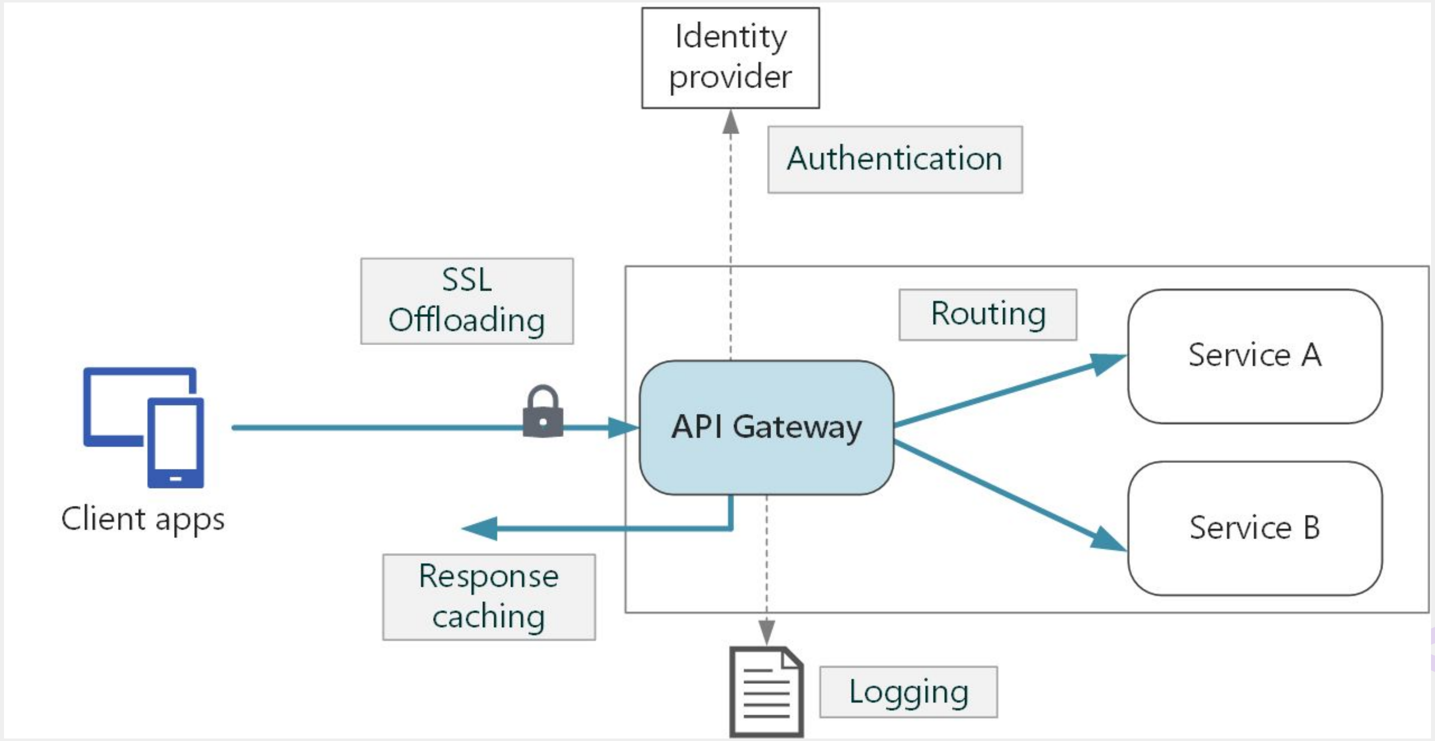
an **api gateway** is a service that sits between the client and a collection of backend services (service mesh).

it is responsible for calling other services, aggregating their result, and returning them to the client.

this simplifies the work from the client as it knows to send requests to one service rather than to all the required ones

in addition, since all requests hit one point (i.e. this part is decoupled), we can have majority of our security/analytics logic implemented here

api gateway


Identity provider
Authentication
SSL Offloading
Routing
Service A
Client apps
API Gateway
Service B
Response caching
Logging

scalability

Like all things in life, certain issues can arise with microservices.

- Latency
- Operational complexity
- Data integrity

Netflix employed microservices with an api gateway. However, given their magnitude of microservices, the api gateway ended up becoming a troublesome large monolith!

Recommended viewing: **How Netflix Scales Its API with GraphQL Federation**

graphql

GraphQL was originally developed internally by Facebook in 2012. It was inspired during Facebook's shift to mobile. While creating a native mobile app, they found that their web app and APIs business logic was too tightly coupled.

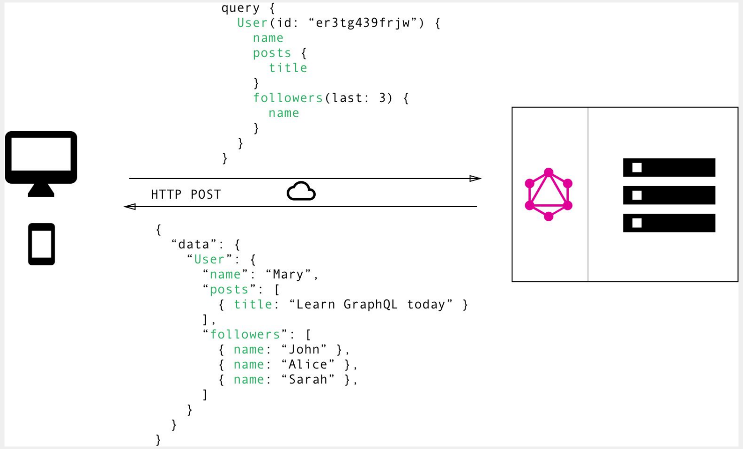
GraphQL is a query language for API requests. It provides a complete and understandable description of the data, gives clients the power to ask for exactly what they need and nothing more, and makes it easier to evolve APIs over time, and enables powerful developer tools.

what's the difference?

The concept of endpoints and queries are decoupled.

Whereas an API request to a REST API might look like:
GET http://localhost:3000/api/rovers/1

An API request to a GraphQL API might look like:
POST http://localhost:3000/graphql?query={rover(id: “1”)}



```
query {
  User(id: “er3tg439frjw”) {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

```
{
  “data”: {
    “User”: {
      “name”: “Mary”,
      “posts”: [
        { title: “Learn GraphQL today” }
      ],
      “followers”: [
        { name: “John” },
        { name: “Alice” },
        { name: “Sarah” },
      ]
    }
  }
}
```

why graphql?

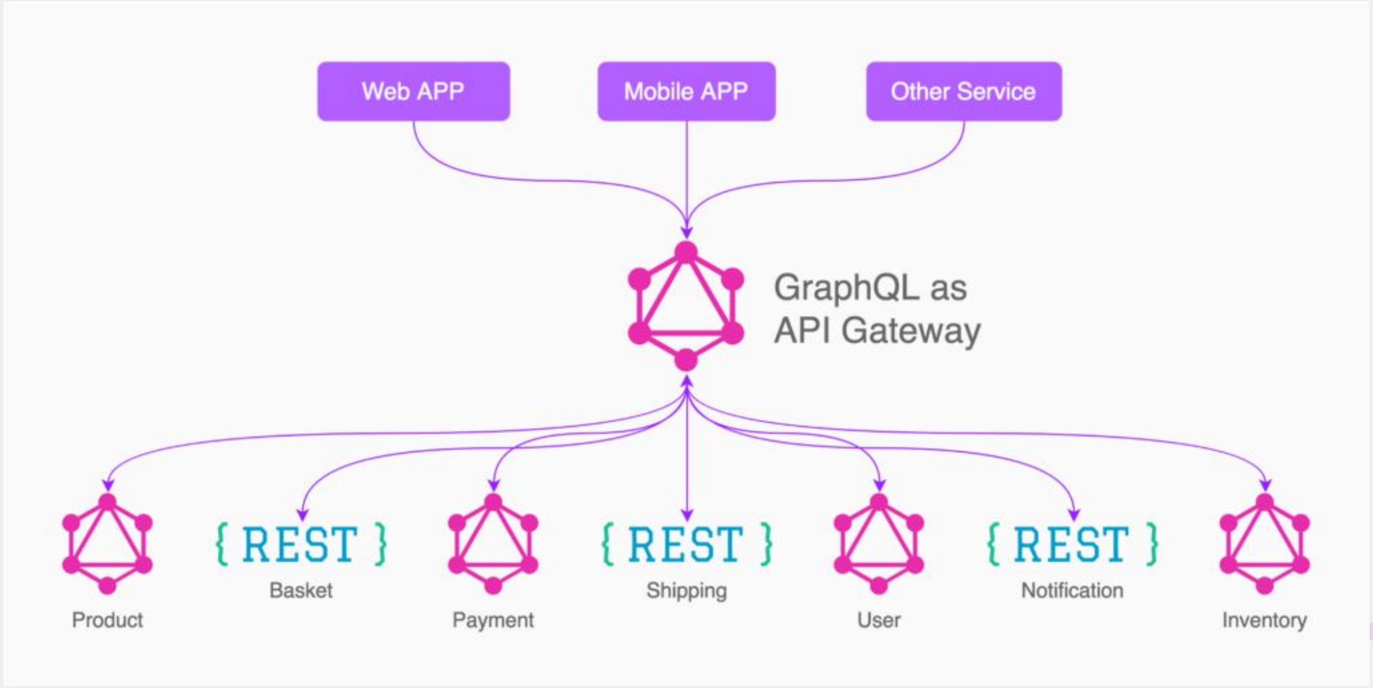
Flexible interface:

- Since endpoints are not tailored to a specific interface (mobile, IoT) the graphql endpoint can service multiple platforms

No more over/under fetching:

- You get exactly what you request
- Bandwidth is more manageable
- No more versioning APIs

graphql in microservices



event-driven architecture

Event-driven architecture is a software architecture that promotes the production, detection, and consumption of, and reaction to events.

It is typically enabled by technologies such as Kafka, RabbitMQ which are event streaming platforms enabling events to be shared and distributed amongst services.

If you've used React, you have some experience with events in the form of event handlers e.g. onClick, onSubmit, etc.

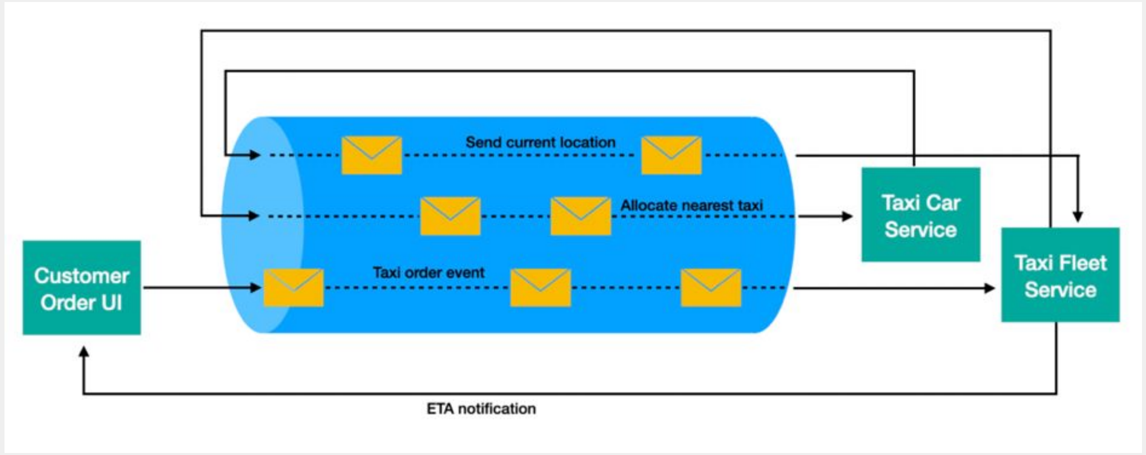
why EDA?

Paradigm shift: client/server -> producer/consumer
API: stateless handling of repeatable, independent requests to a stateful awareness

- **Consumers can subscribe to and receive events asynchronously, yet in real-time** if both are online
- **Client no longer has to keep asking for updates or state changes** (through requests/polling)
- **Loose coupling between API producers/consumers:** consumers are only interested in the message received and not the servers they were produced from. Producers do not care about the identity of their consumers.
- **Optimal network resource usage:** continuous polling is taxing and wasteful in nature, but can be solved with EDA.

eda example

The arrows within the diagram represent the flow of events (and thus state changes) within the system.



resources

Recommended viewing: **The Many Meanings of Event-Driven Architecture** (Martin Fowler)