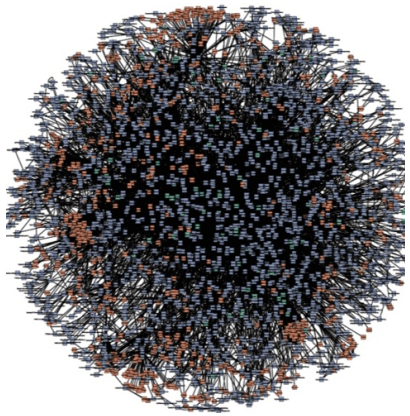
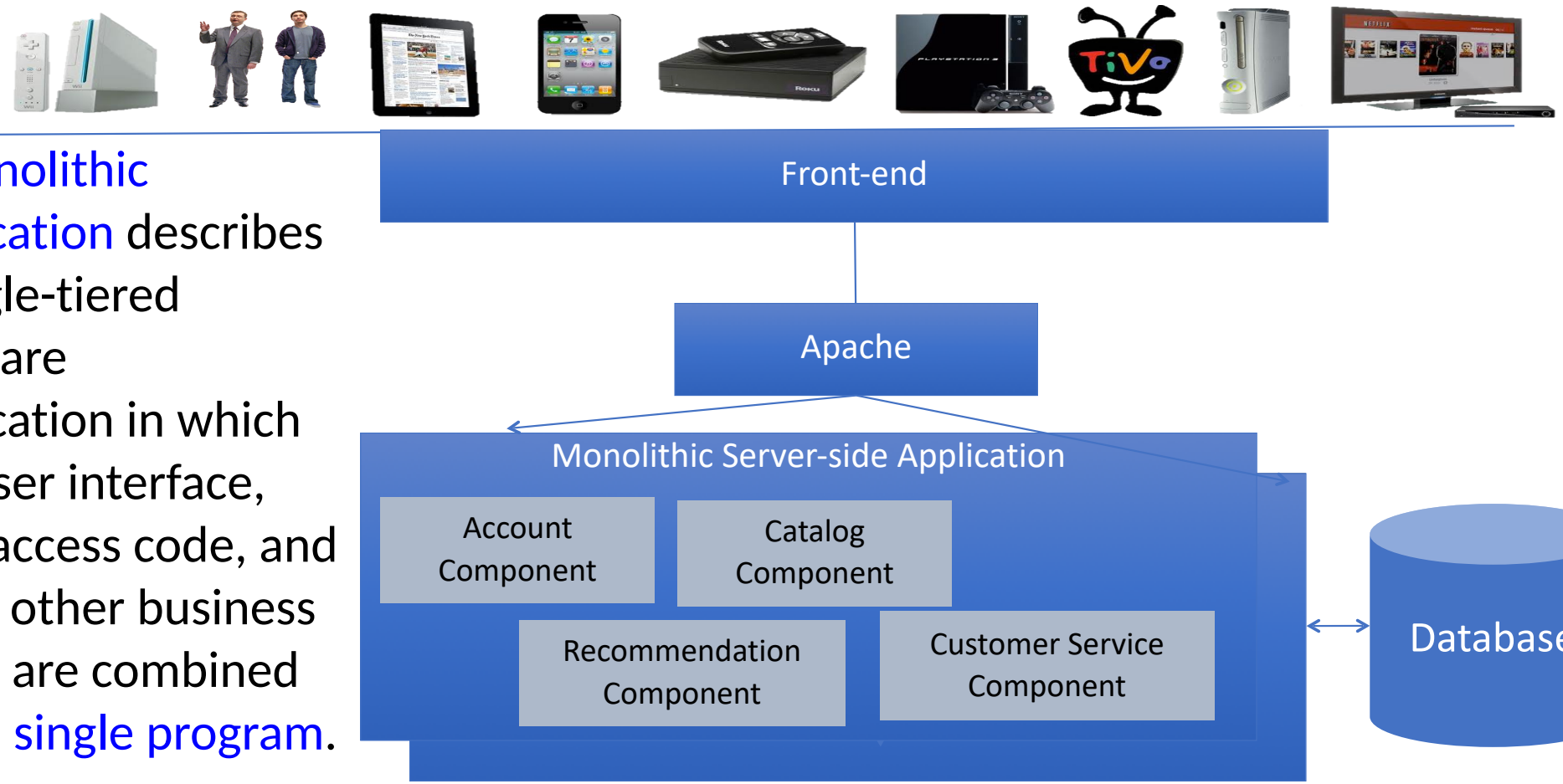
CS-552/452 Introduction to Cloud Computing

Microservices Overview



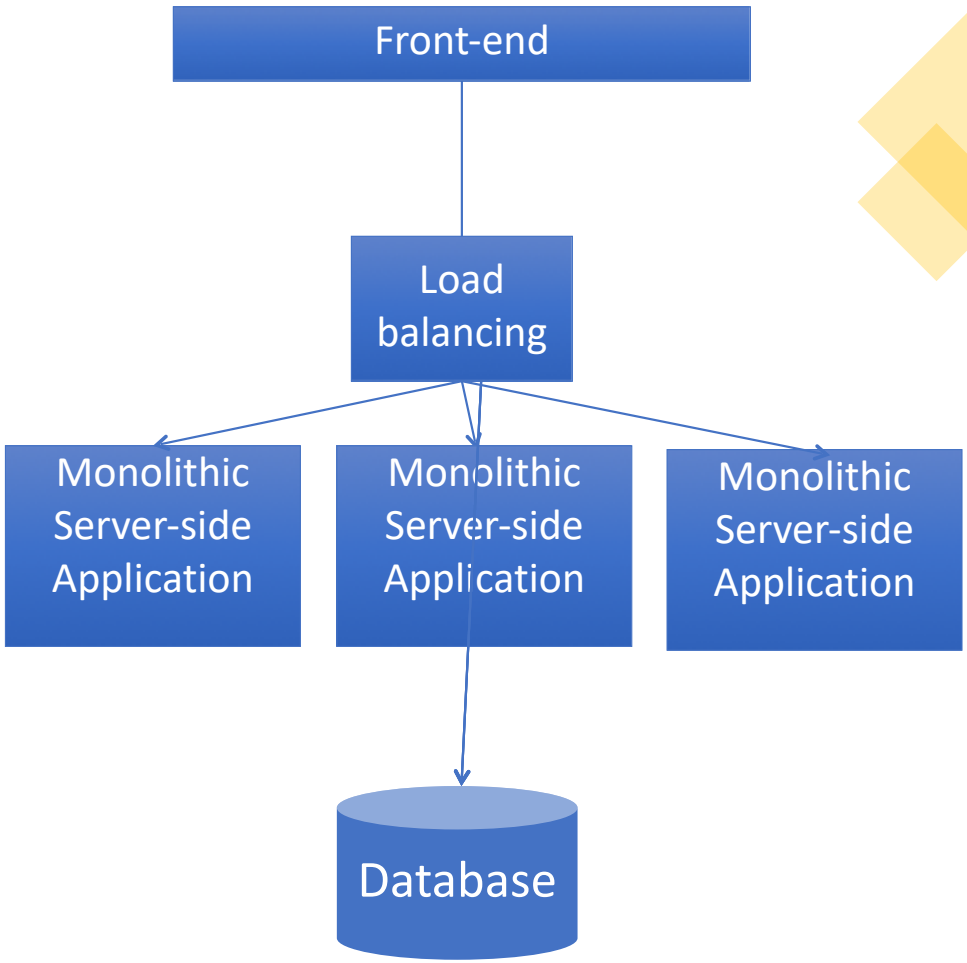
Monolithic Architecture

A monolithic application describes a single-tiered software application in which the user interface, data access code, and many other business logics are combined into a single program.



Characteristics of Monolithic Applications

- Single, large codebase
  - Good IDE support - easy to develop/debug/deploy (in the beginning)
  - Simple to deploy - a central ops team can efficiently handle
  - Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer

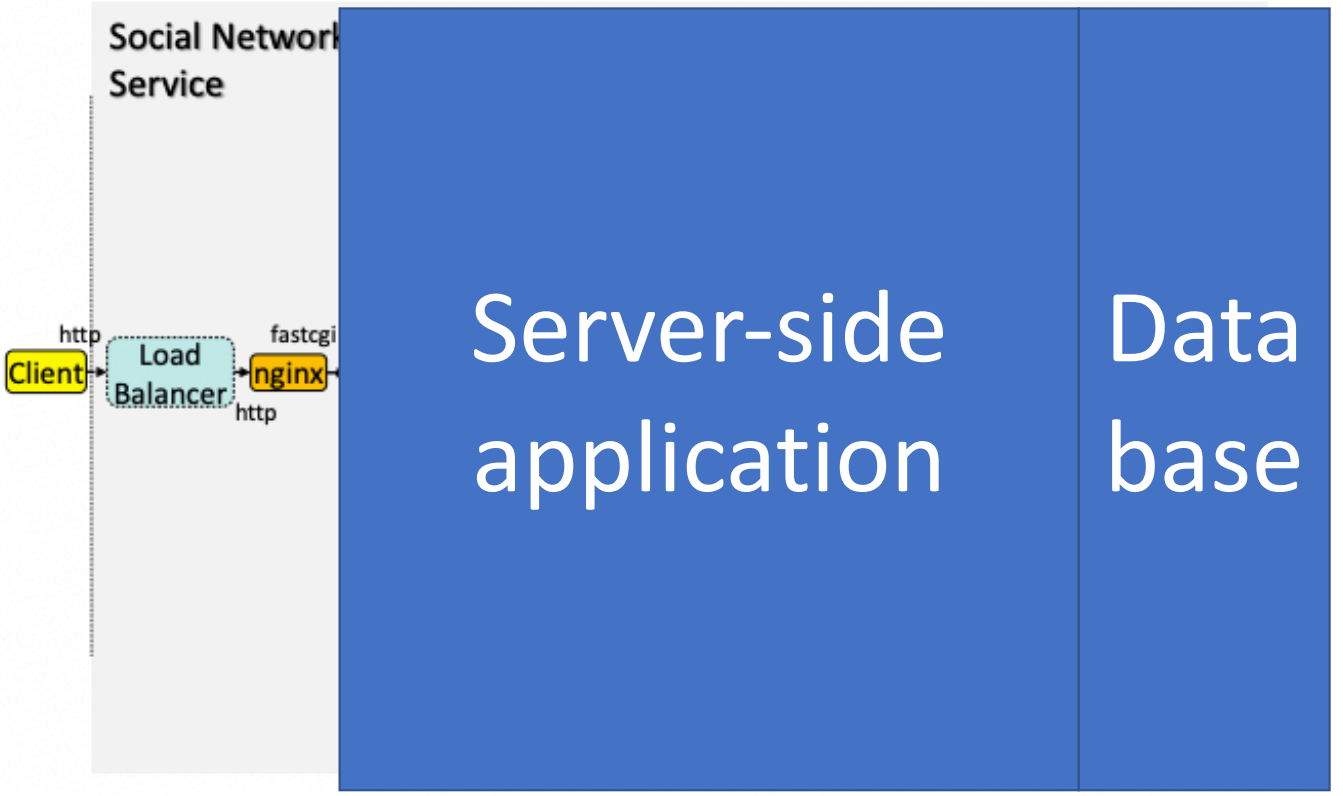


Key Challenges:

- Long deployment cycles
  - However, IT companies want to shorten production cycles, aiming to adapt to changing markets and always-on demands
- Scalability
  - Because of tight coupling, scaling is “undifferentiated”
  - Scaling can only be done horizontally
- Availability
  - A single missing “;” brought down the Netflix website for many hours (~2008)

Micro-Services Architecture

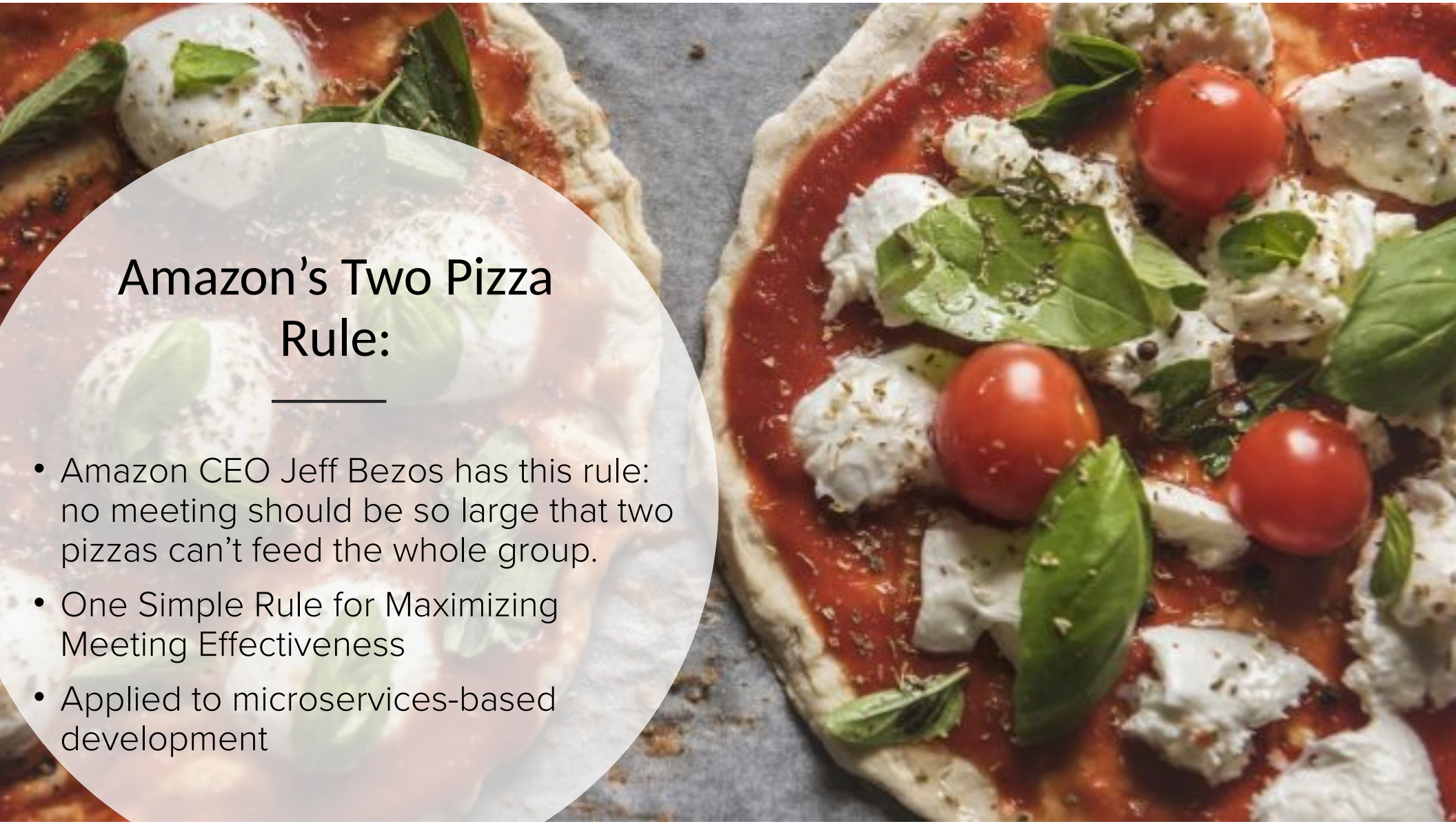
"Service-oriented architecture composed of small, loosely coupled elements that have bounded contexts"



Each time do one thing and do it well

Key Benefits of Micro-Services

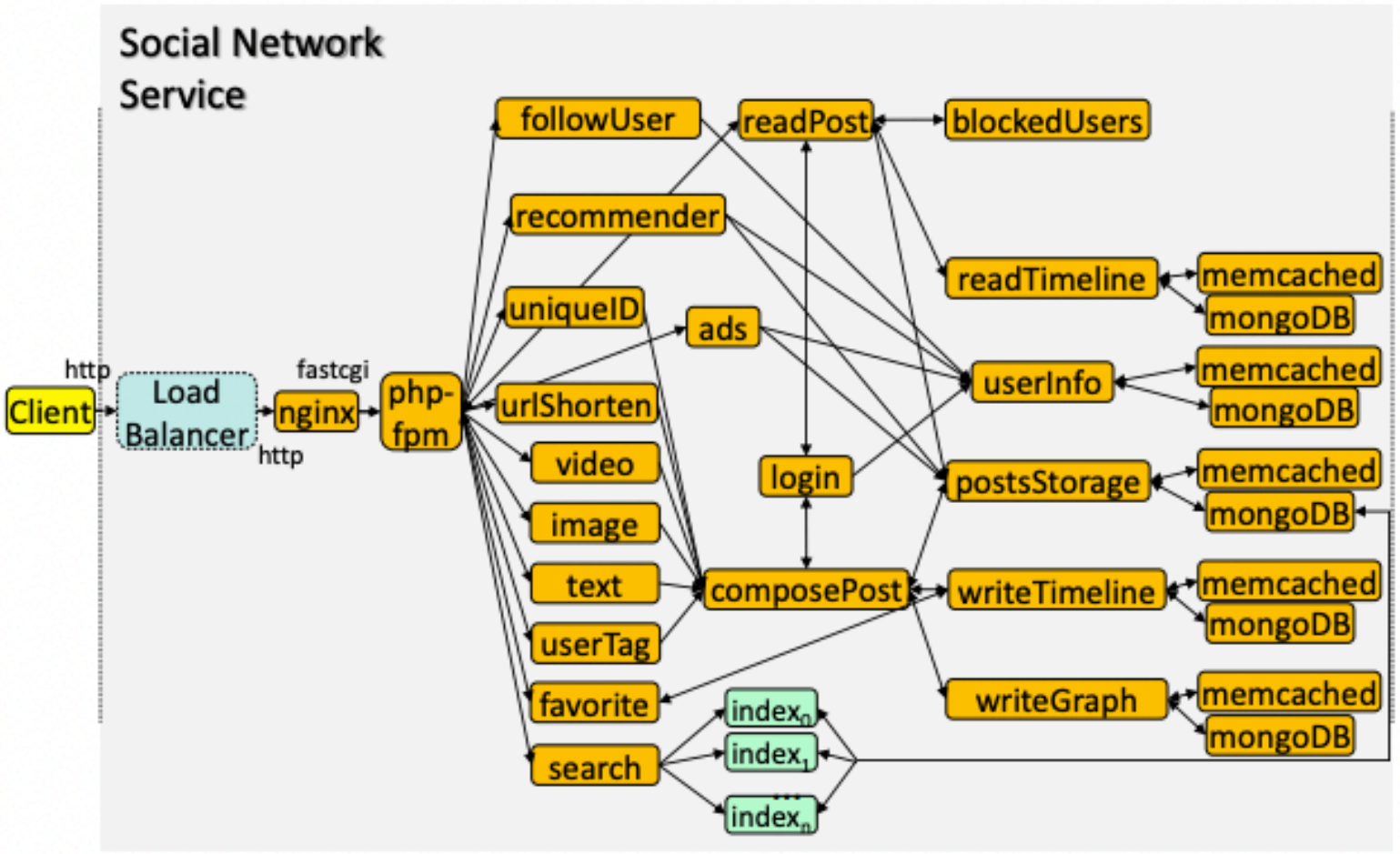
- Each micro-service is really small
  - Easier for a developer to understand
  - It makes developers more productive, and speeds up deployments
- Continuous delivery and deployment of large, complex applications
  - Improved maintainability
  - Better testability
  - Better deployability
  - Easy to organize the development effort around multiple, autonomous teams.
- Improved fault isolation
- Eliminates any long-term commitment to a technology stack
- Fine-grained scalability



Amazon's Two Pizza Rule:

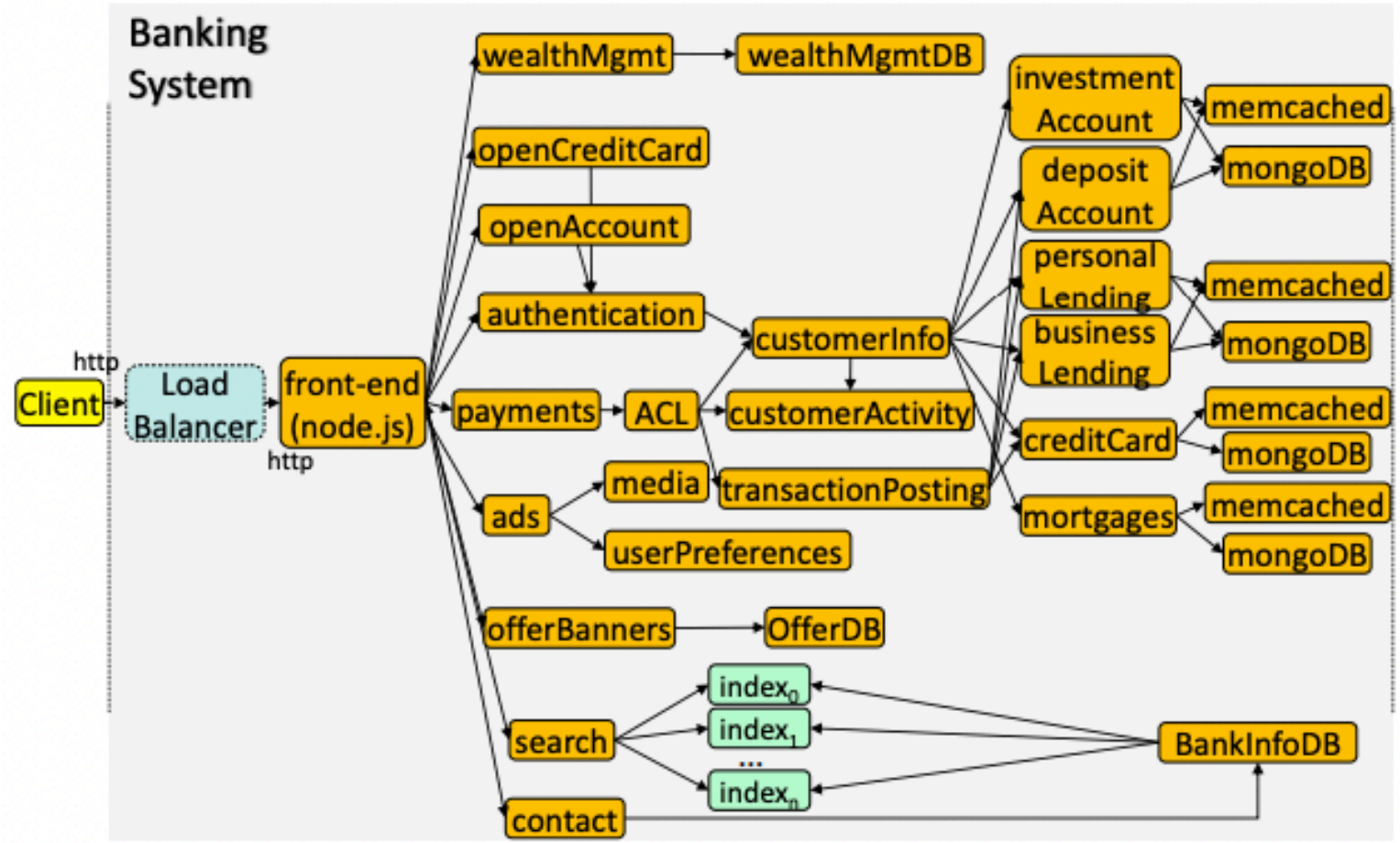
- Amazon CEO Jeff Bezos has this rule: no meeting should be so large that two pizzas can't feed the whole group.
- One Simple Rule for Maximizing Meeting Effectiveness
- Applied to microservices-based development

Microservices Architecture Example 1

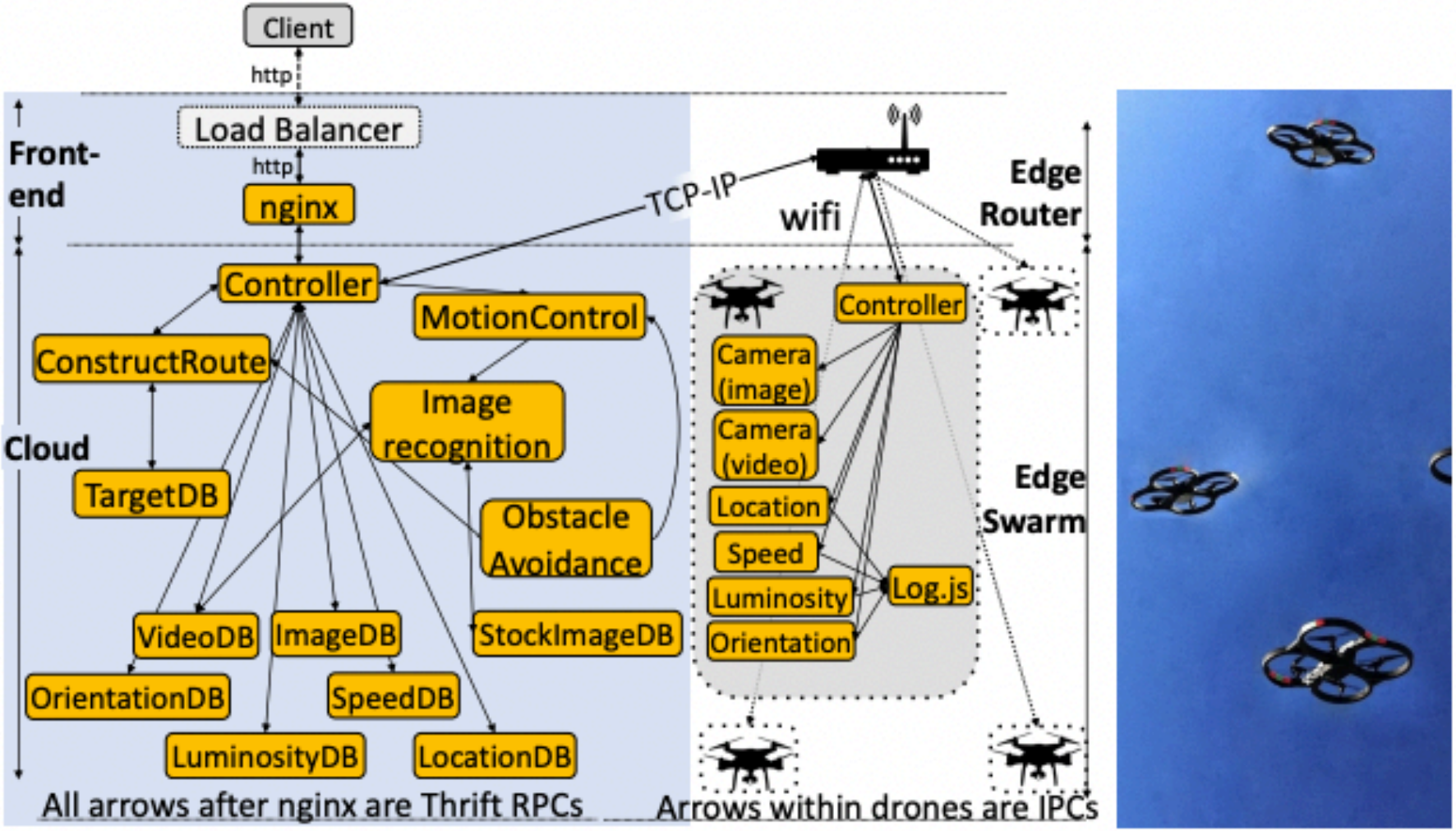


https://github.com/delimitrou/DeathStarBench/tree/master/hotelReservation

Microservices Architecture Example 2

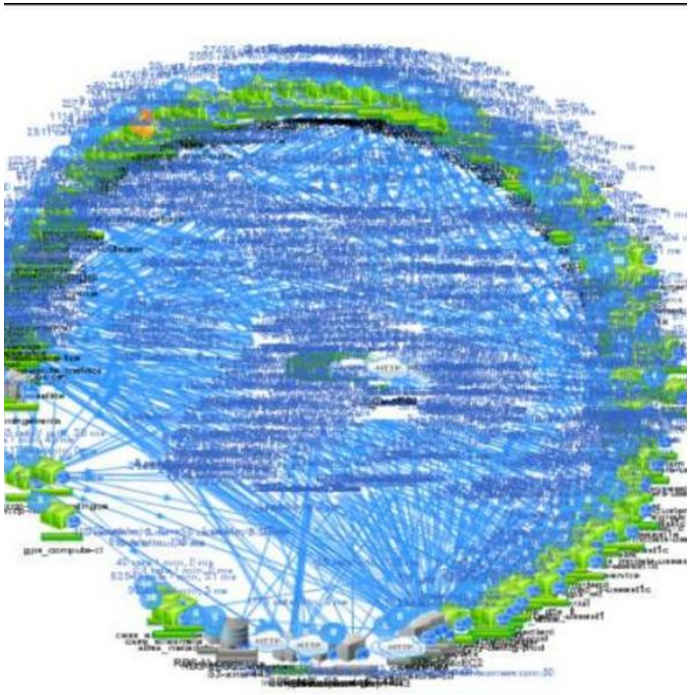


Microservices Architecture Example 3



Case Study 1: Netflix

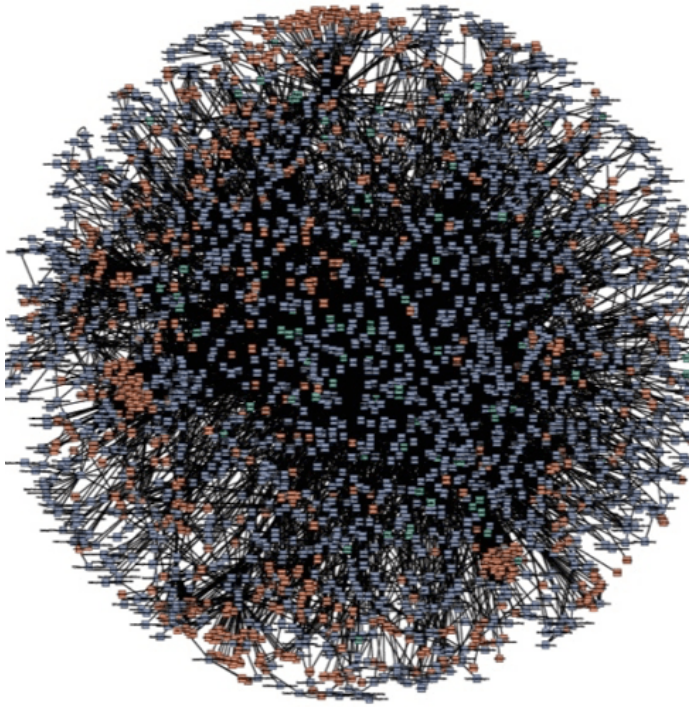
- Around 2009 started to move to micro-services based architecture on AWS.
  - First non-customer facing applications, e.g., encoding
  - Then customer facing elements, e.g., moving selection, device selection, and configuration
- Now (2019), 500+ microservices with 2 billion requests each day



Netflix Microservices Structure

Case Study 2: Amazon

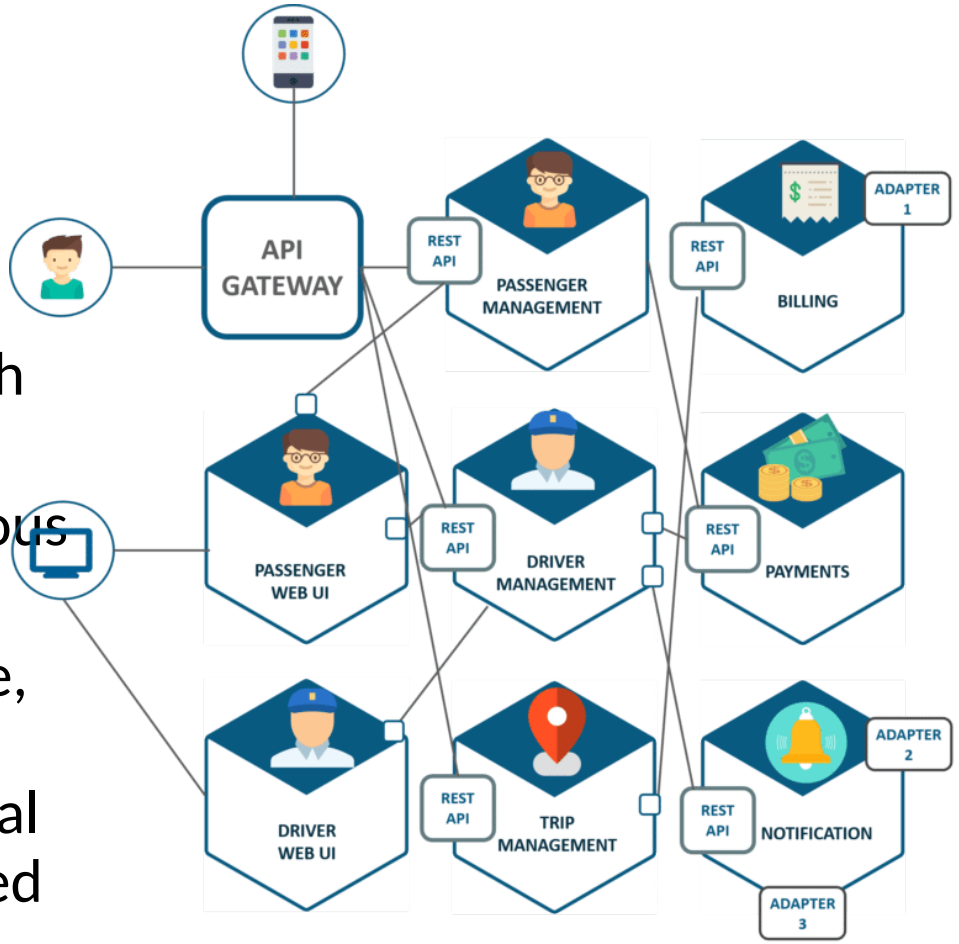
- Before 2001 – the “amazon.com” retail website was based on a monolithic architecture
  - Reconcile changes from hundreds of developers
  - Resolve all the conflicts
  - Merge into a single version and produce a master version waiting to be moved into production
- Now, still multiple tiers with each multiple components but loosely coupled
  - Decomposed the monolithic architecture gradually
  - Pulled out single-purpose functions and wrapped them with a web service interface
  - continuous deployment tools — CodeDeploy, CodePipeline and CodeCommit



Amazon Microservices structure

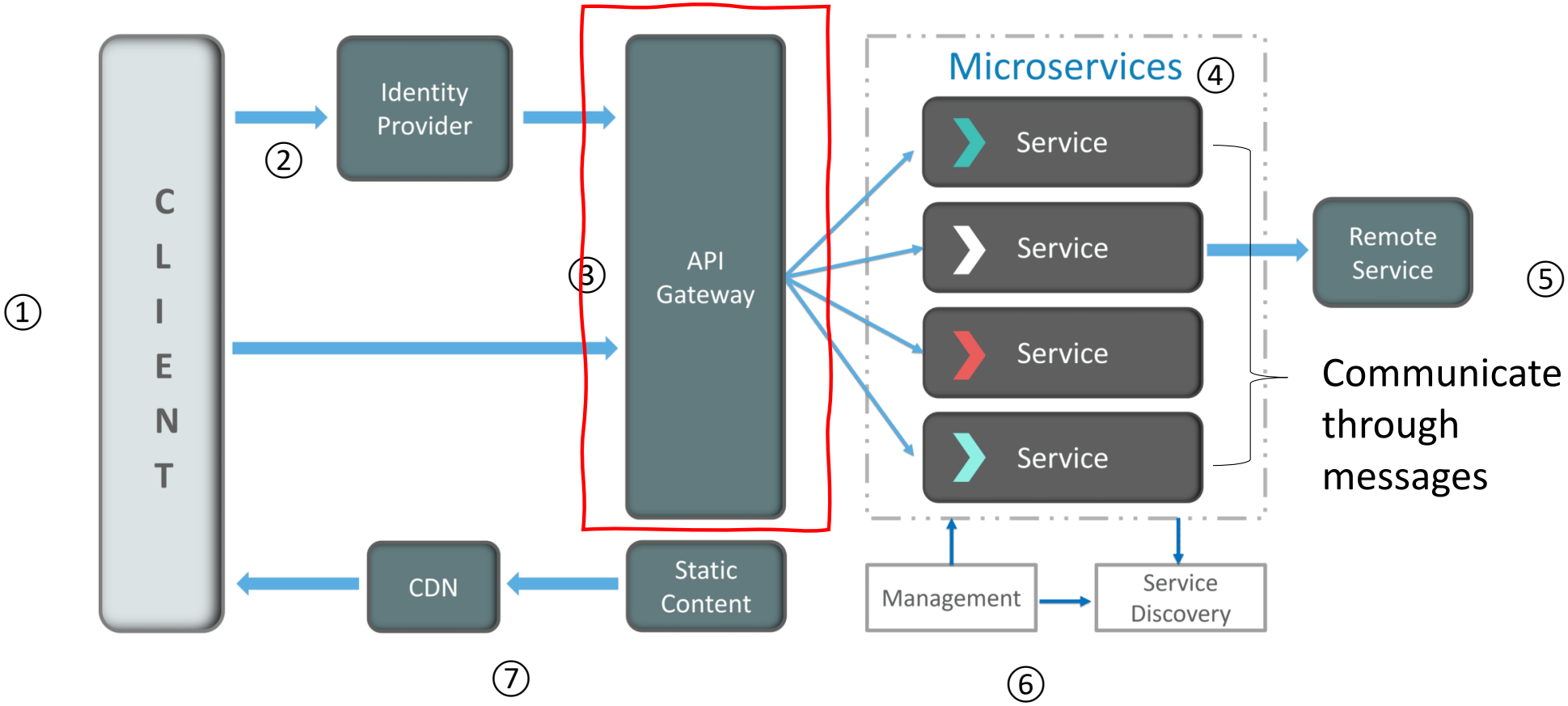
Case Study 3: Uber

- In the beginning, a single offering in each city
  - Challenges in scalability and continuous integration
- With the new microservices architecture, Uber introduced an API Gateway and independent services that have individual functions and can be deployed and scaled separately.



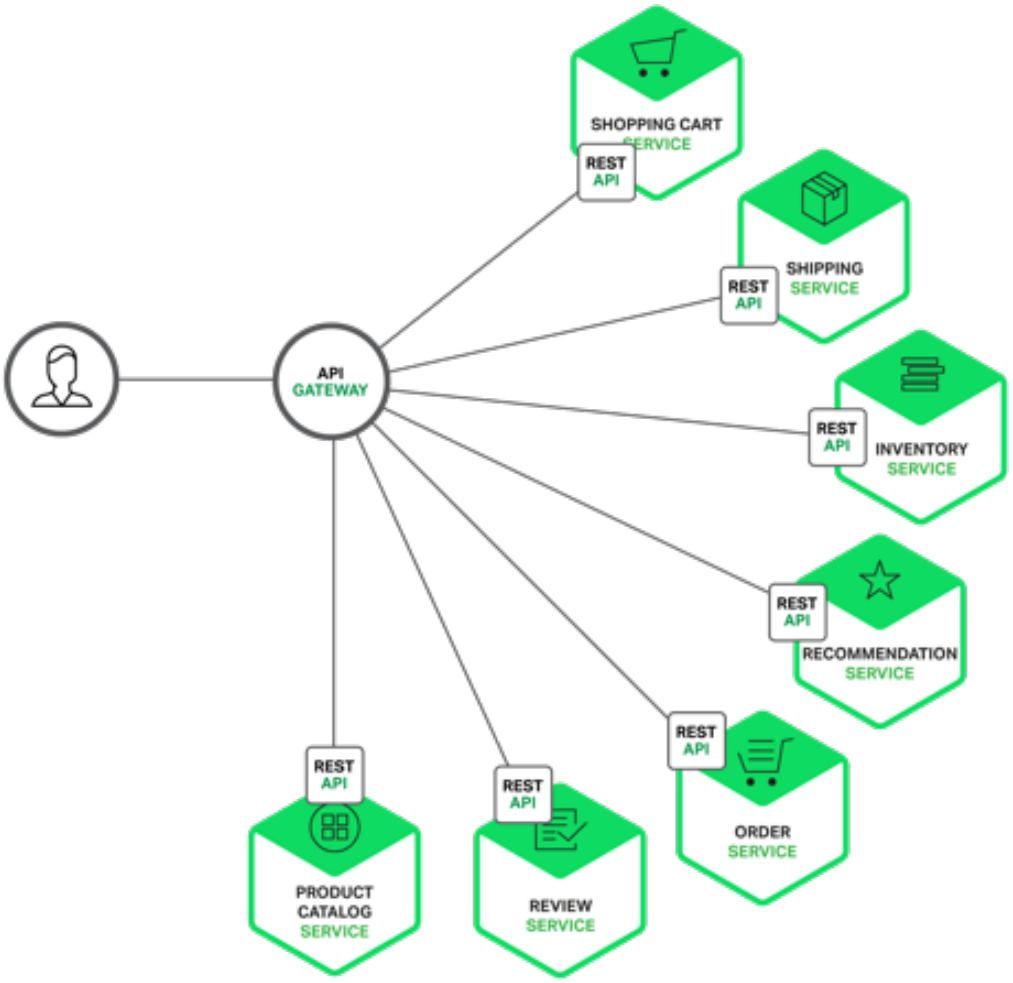
Uber Microservices Architecture

A Typical Microservice Architecture

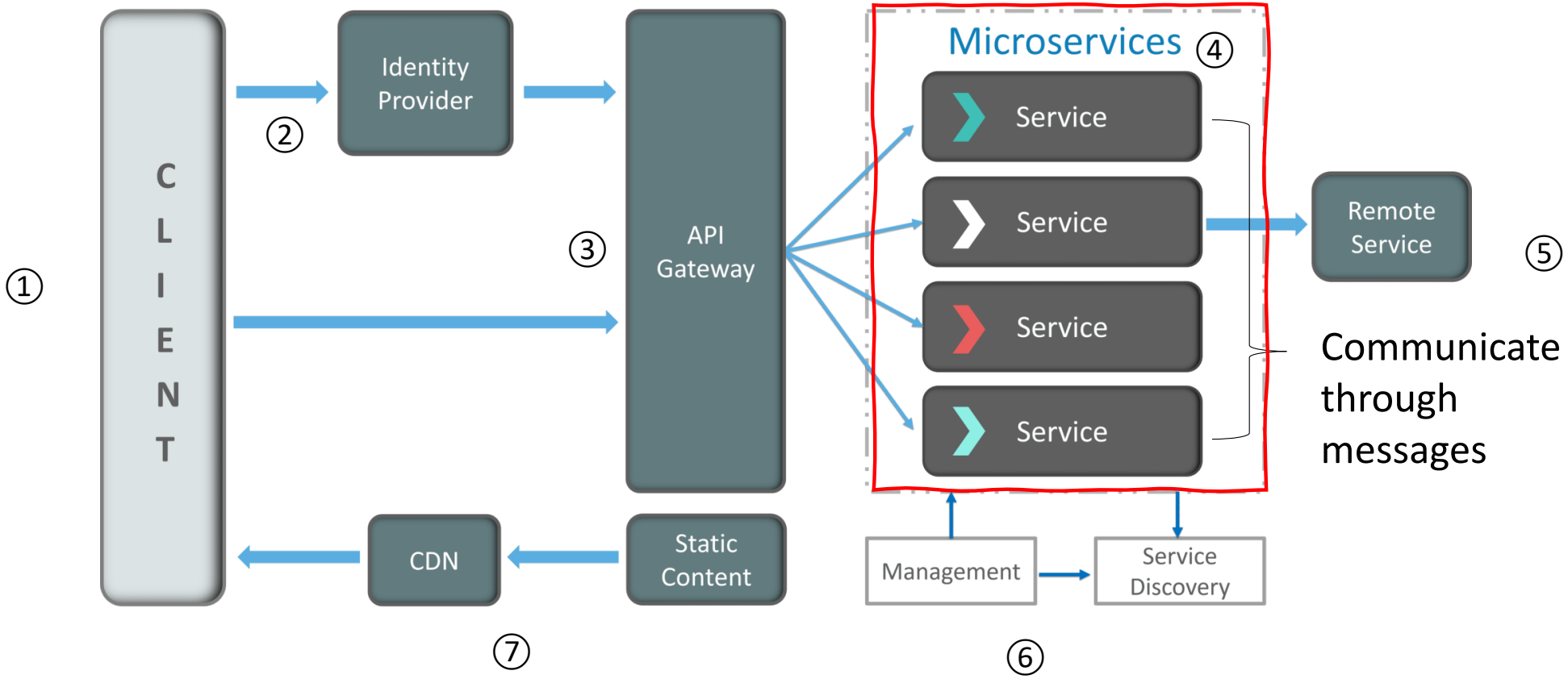


API Gateway

- An API Gateway is a server that is the single entry point into the system.
  - The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client.
- The API Gateway is responsible for request routing, composition, and protocol translation.
  - The API Gateway handles the request by invoking the various services – product info, recommendations, reviews, etc. – and combining the results.



A Typical Microservice Architecture

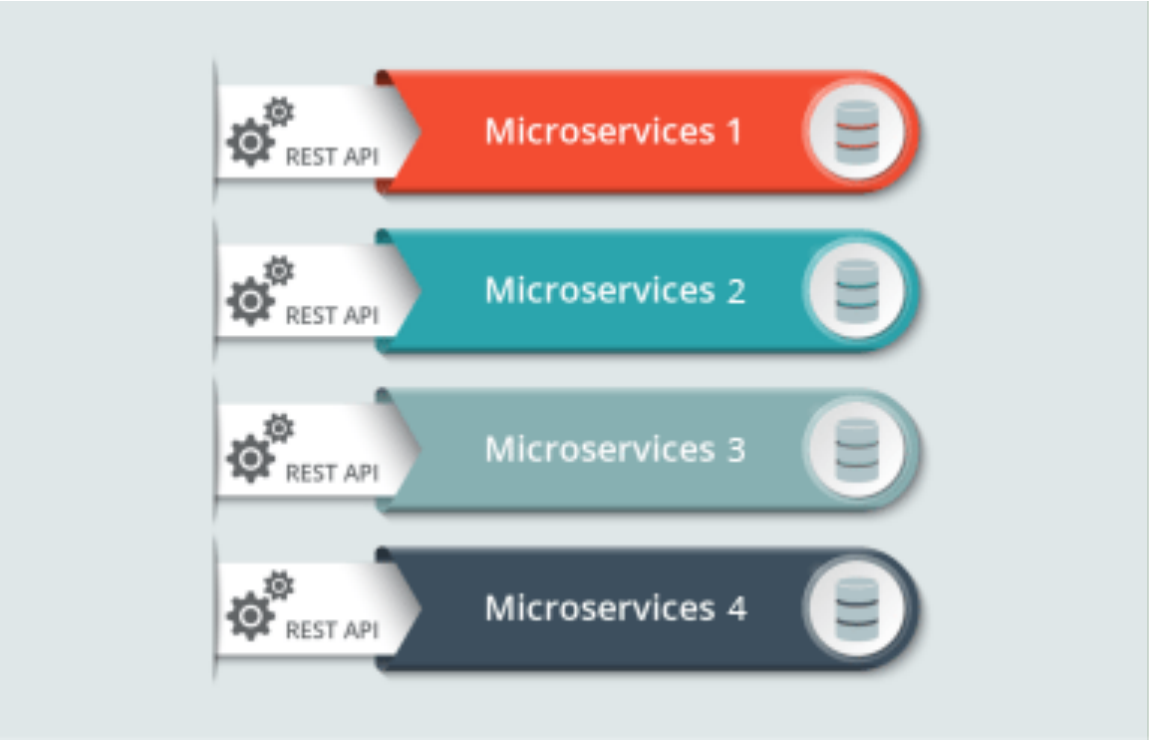


Type of Messages

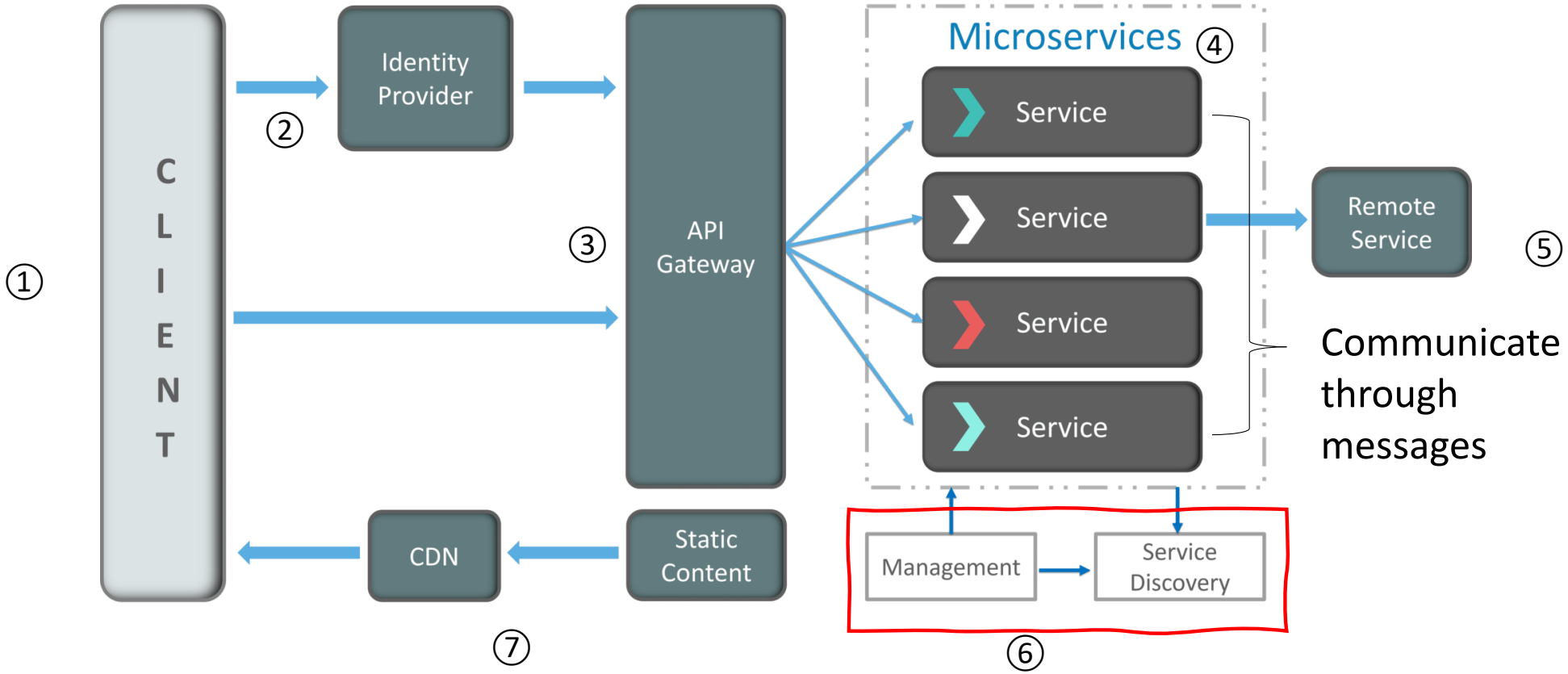
- There are two types of messages through which microservices communicate:
  - Synchronous messages:
    - clients wait for the responses from a service,
    - microservices usually use REST (Representational State Transfer) API
  - Asynchronous messages:
    - Clients do not wait for the responses from a service
    - microservices usually use communication protocols (message brokers) such as AMQP, STOMP, MQTT.
- Cloud provides many messaging services
  - AWS Firebase (instant messages)
  - Apache Kafka, AWS Kinesis (streaming analytics)
  - Amazon Simple Queue Service (messages)
  - RabbitMQ, Apache Pulsar, etc.

Data Handling

- Each microservice owns a private database to capture their data and implement the respective business functionality



A Typical Microservice Architecture



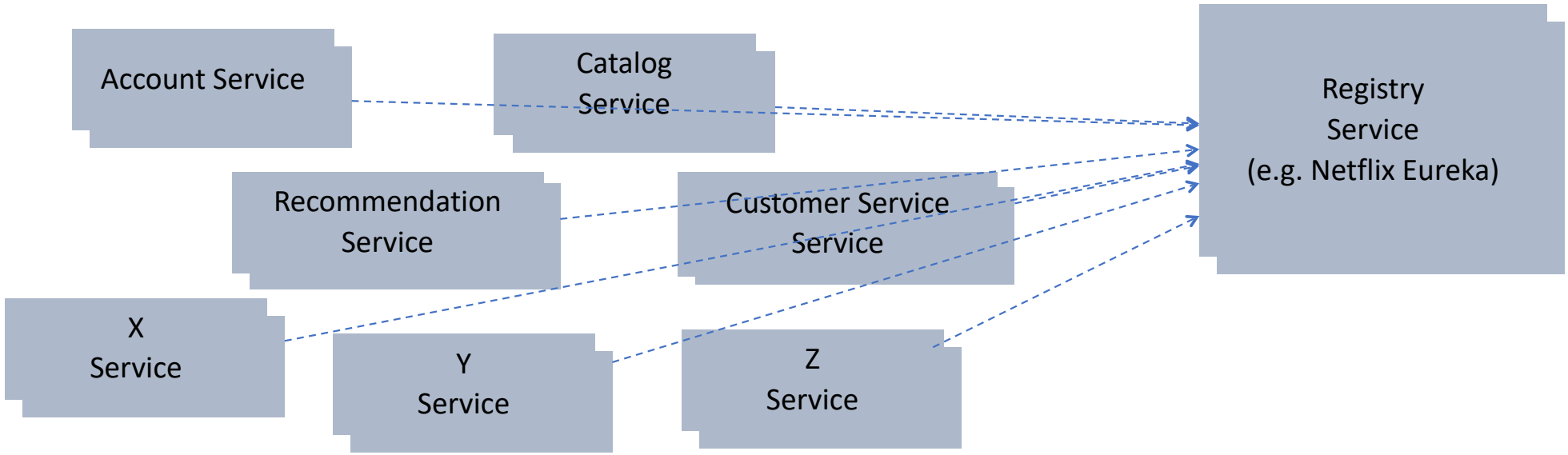
Challenges of Micro-services

- Distributed Systems are inherently Complex
  - N/W Latency, Fault Tolerance, Retry storms ..

Can lead to chaos if not designed right …

Service Discovery

- One request may need to trigger 100s of MicroServices
  - Need a Service Metadata Registry (Discovery Service)
  - Single point of failure?

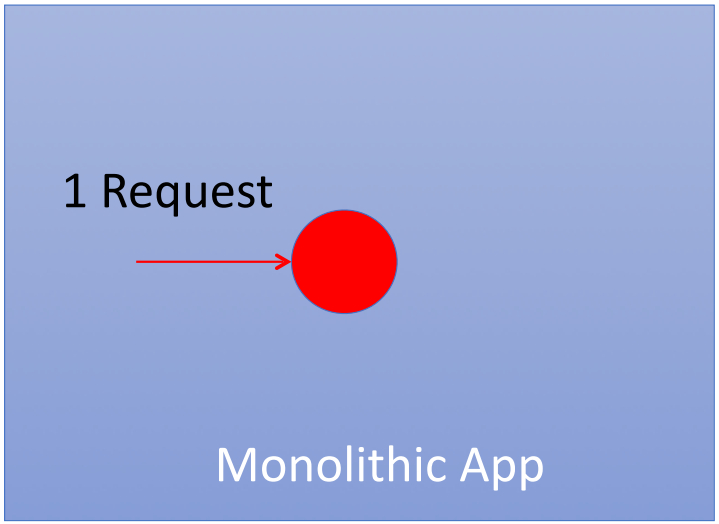


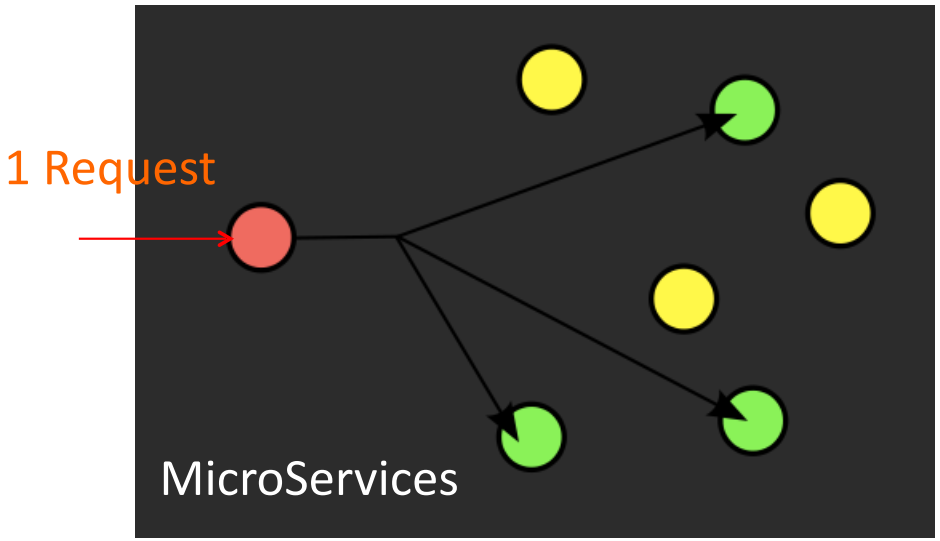
Service Dependency

- How many dependencies does my service have?
- What is the call (request) volume on my Service?
- Are any dependent services running Hot?
- What are the top N slowest “business transactions”?
- What are the sample HTTP Requests/Responses that had a 500 Error Code in the last 30 minutes?

Requests (and Fan Out)

~2 Billion Requests (e.g., Netflix) per day on Edge Service
Results in ~20 Billion Fan out requests in ~10 MicroServices





Main reasons to use microservices

- Zero-downtime independent deployability
  - Especially under SaaS model when application uptime is super important
- Isolation of processing around data
  - When handling sensitive customer data, so that cod ethat touches the data is isolated.
- Enable a higher degree of organizational autonomy
  - Distribute responsibilities into teams to reduce the amount of coordination with the rest of the organization
  - As team size increases, it is harder to coordinate across teams.

How to avoid a “distributed” monolithic microservice

- Distributed monolith
  - In theory, there are many distributed microservices
  - In practice, to deploy/change one service, you need to deploy/change N other services.
- Practice “Information Hiding”
  - Services must be self-contained in the data they touch.
  - Changing a service implementation should not change its interface to other services
- Combine services that change together
  - Review change logs regularly
-

Takeaways

- Monolithic apps
  - Good for small organizations and teams
  - Good for small self-contained applications

- Microservices
  - Consider adopting when your organization scales
  - Scales better, more uptime
  - Microservices will cost more money in the short term, but they will make more money in the long term
  - Fits better with a decentralized organization structure

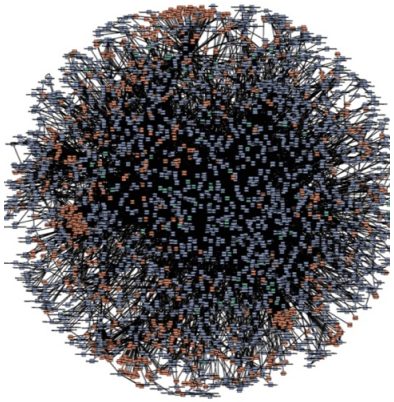
Sources

- What Led Amazon to its Own Microservices Architecture: https://thenewstack.io/led-amazon-microservices-architecture/
- Microservices Architectures: Become a Unicorn like Netflix, Twitter and Hailo: https://www.slideshare.net/gjuljo/microservices-architectures-become-a-unicorn-like-netflix-twitter-and-hailo
- Microservice Architecture — Learn, Build, and Deploy Applications, https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a
- Building Microservices: Using an API Gateway: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/#:~:text=A%20great%20example%20of%20an,API%20for%20their%20streaming%20service.
- https://www.youtube.com/watch?v=57UK46qfBLY

CS-552/452 Introduction to Cloud Computing

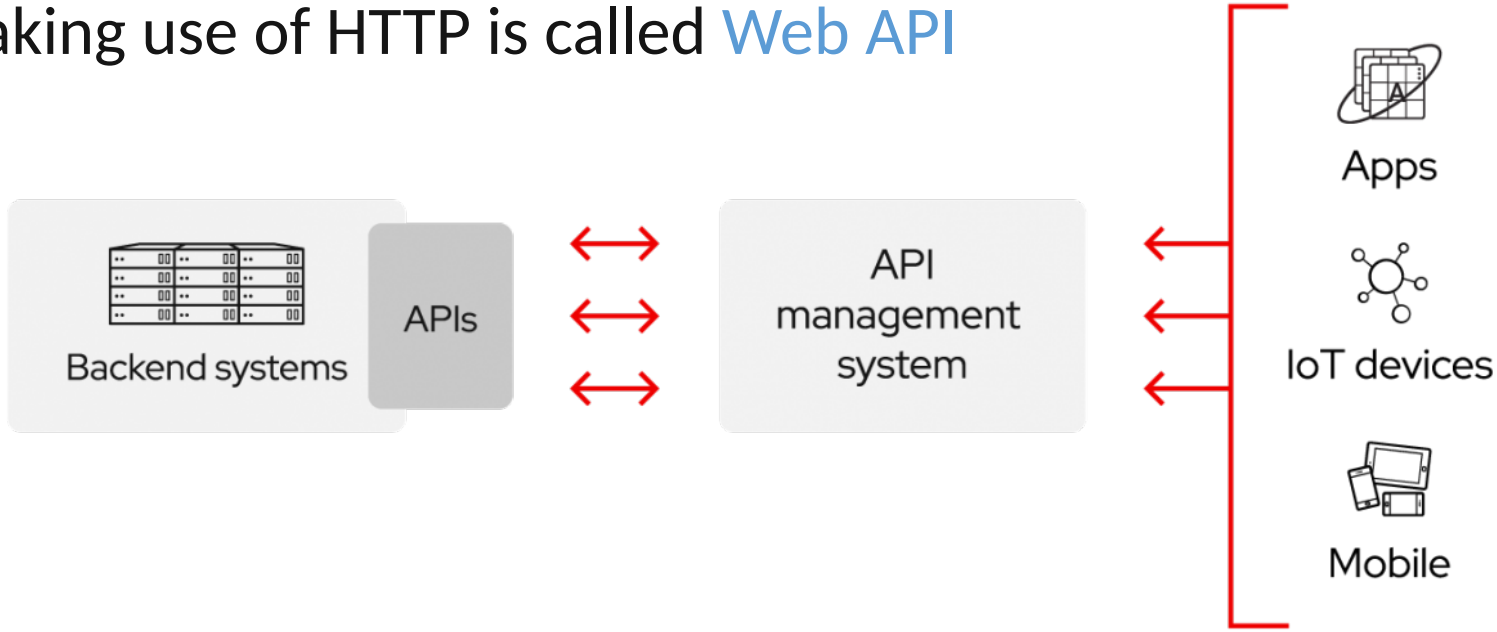
3. Micro-services (Part II)

Inter-connect and RESTful APIs

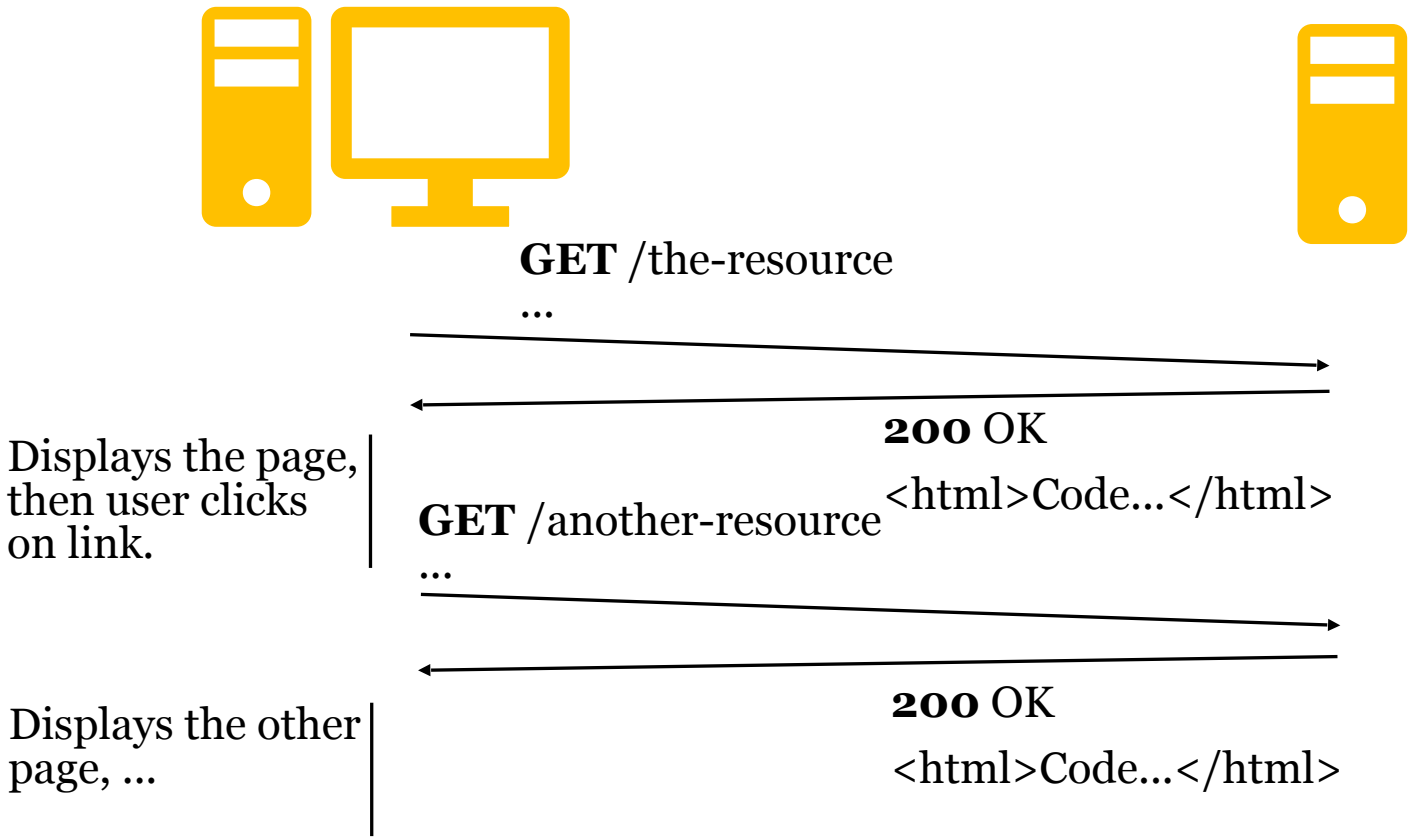


What is an API?

- A set of definitions and protocols for building and integrating application software.
  - The communication channel between services/components
- APIs let one product/service communicate with other products and services without having to know how they're implemented.
- An API making use of HTTP is called Web API



Traditional web applications (HTML and HTTP)



Traditional web applications

The interface is built on HTML & HTTP.

- Drawbacks:
  - The client must understand both HTTP and HTML.
    - Smart phones' GUIs are not based HTML
  - The entire webpage is replaced with another one.
    - No way to animate transitions between webpages.

Different Types of Web APIs

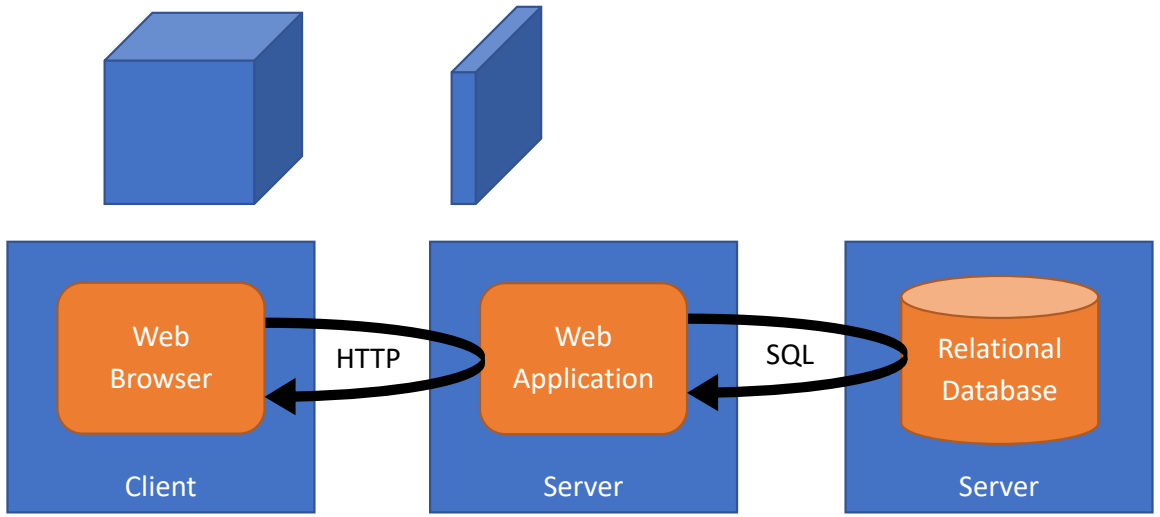
- Remote Procedure Call, RPC.
  - Clients can call functions on the remote server
  - e.g., gRPC, Apache Thrift, Apache Avro, etc.
- Remote Method Invocation, RMI.
  - Clients can call methods on objects on the remote server.
- Representational State Transfer, REST.
  - Clients can apply CRUD operations on resources on the server.
    - CRUD – create, read, update, and deletion
    - Major functions over “database” applications

What is REST (Representational State Transfer)?

An *architectural style* for *distributed hypermedia systems*

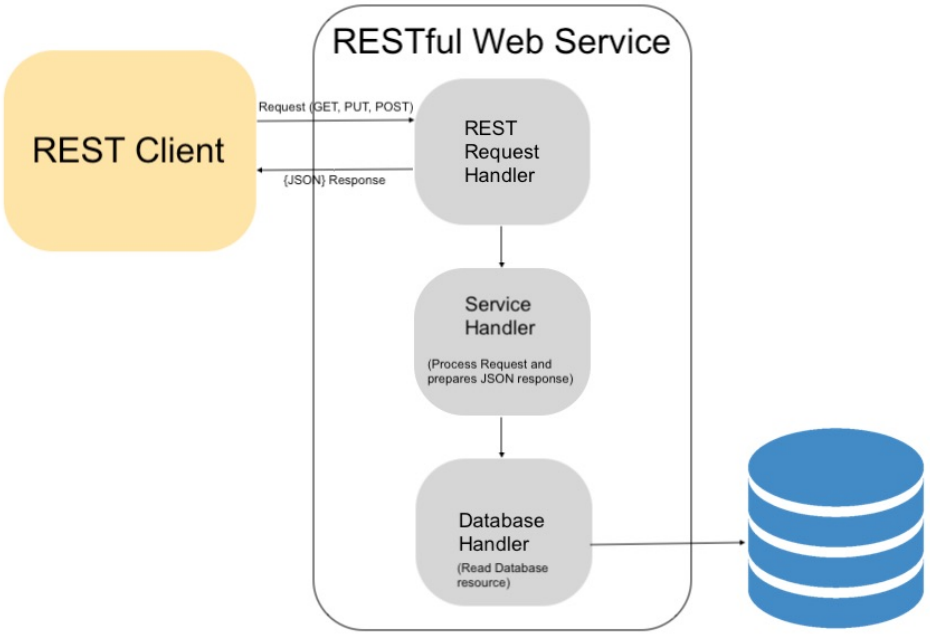
- By Roy Thomas Fielding in his doctoral dissertation in 2000.

- Consists of constraints:
  1. Client - Server
  2. Stateless
  3. Cache
  4. Uniform Interface
  5. Layered System
  6. Code-On-Demand (optional)



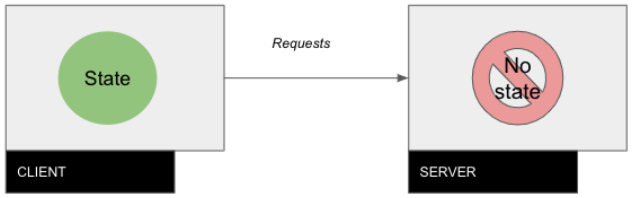
1. Client-server architecture

- REST is an architecture style based on web-standards and the HTTP protocol
- In a REST based architecture, *everything is a resource*
  - Identified using a global identifier (URL or URI)
  - Resources can use different formats, e.g., text, xml, json, etc.
- A resource is accessed via a common interface based on the HTTP standard methods
  - GET, POST, PUT, and Delete
- A REST server provides the access to the resources and a REST client which accesses and modifies the REST resources



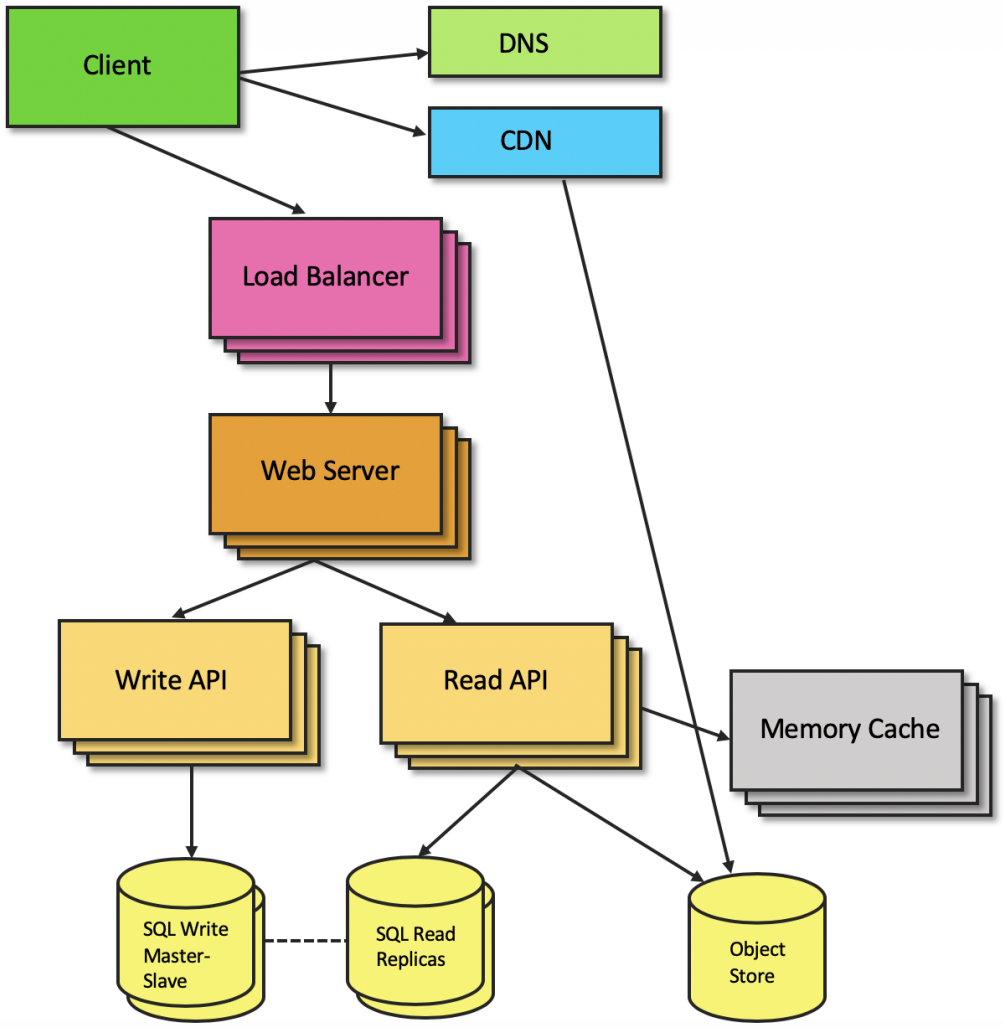
2. Stateless

- No client content is stored on the server between requests.
  - Information about the session state is, instead, held with the client
  - Excellent for distributed systems
  - Key benefits of stateless
    - Scalability:
      - The server no longer needs to keep track of client sessions or resources between requests and can quickly free resources after requests have been finished.
      - Any server can handle any request: essential to modern cloud computing and the internet
    - Reliability:
      - It will also be easier for the client to recover from failures, as the session context on the server has not suddenly gotten corrupted or out of sync with the client
    - Less complexity: No complex synchronization on the server side
    - Visibility:
      - Every request contains all context necessary to understand it. Therefore, looking at a single request is sufficient to visualize the interaction



3. Cacheability and 4. Layered System

- Responses are labeled as cacheable or non-cacheable and caching is restricted to the client or an intermediary between the server.
  - Caching can eliminate the need for some client-server interactions.
  - The same requests can be served by local cache
- Additional layers can mediate client-server interactions.
  - These layers could offer additional features like load balancing, proxies, shared caches, or security
  - They do not change the existing API



5. Uniform interface

- Use URLs/URIs to identify resources
  - URLs can map to a single resource e.g. movie/1234
- Use HTTP methods to specify the CRUD operations:
  - Create: POST
  - Retrieve: GET
  - Update: PUT
  - Delete: DELETE
- Use HTTP headers
  Content-Type and Accept
  to specify data format for the resources.
  - Text, XML, json, etc.
- Use HTTP status code to indicate success/failure.

5. Uniform interface

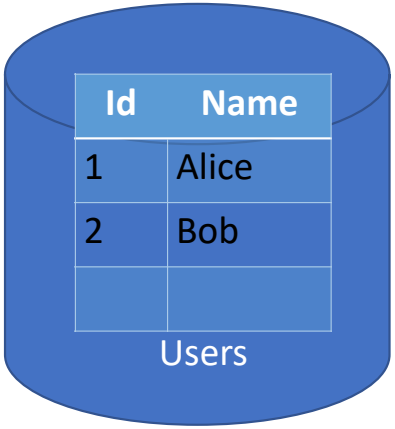
- Use nouns but not verbs in path or URI
- Use only plural nouns for all resources

Purpose	Method	Incorrect	Correct
Retrieves a list of users	GET	/getAllCars	/users
Create a new user	POST	/createUser	/users
Delete a user	DELETE	/deleteUser	/users/10
Get balance of user	GET	/getUserBalance	/users/11/balance

5. Uniform interface

- Make the API version mandatory and do not release an unversioned API
  - Use a simple ordinal number and avoid dot notation such as 2.5
  - For example: /blog/api/v1
- Use a unique query parameter or a query language for filtering
  - GET /cars?color=red (returns a list of red cars)
  - GET /users?name=tom (returns a list of users whose name matches tom)
- Allow ascending and descending sorting over multiple fields
  - GET /cars?sort=-manufacturer, +model
  - Returns a list of cars sorted by descending manufacturers and ascending models

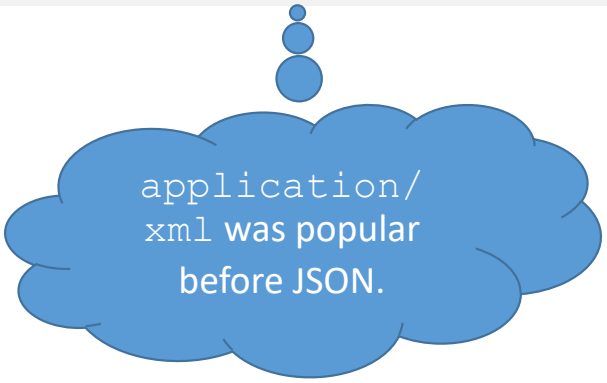
REST examples



A server with information about users.

- The GET method is used to retrieve resources.
  - Which data format? Specified by the Accept header!

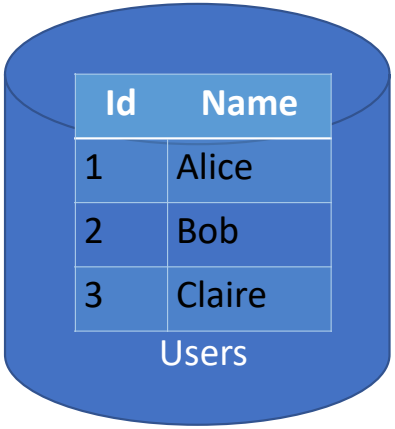
```
GET /users HTTP/1.1
Host: the-website.com
Accept: application/json
```




```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 66

[
  {"id": 1, "name": "Alice"},
  {"id": 2, "name": "Bob"}
]
```

REST examples



A server with information about users.

- The POST method is used to create resources.
  - Which data format? Specified by the Accept and Content-Type header!

```
POST /users HTTP/1.1
Host: the-website.com
Accept: application/json
Content-Type: application/xml
Content-Length: 49

<user>
  <name>Claire</name>
</user>
```

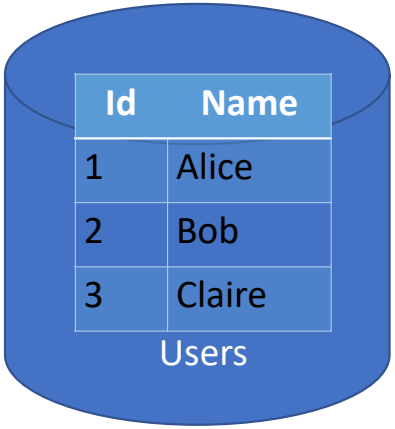
```
HTTP/1.1 201 Created
Location: /users/3
Content-Type: application/json
Content-Length: 28

{"id": 3, "name": "Claire"}
```

REST examples

A server with information about users.

- The PUT method is used to update an entire resource.



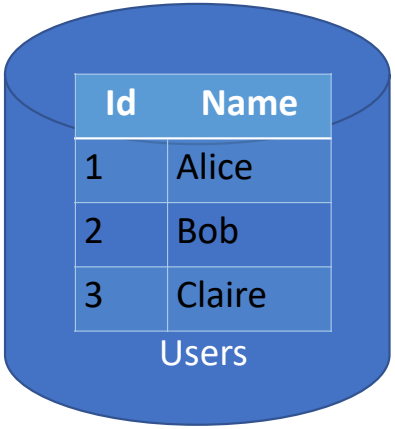

```
PUT /users/3 HTTP/1.1
Host: the-website.com
Content-Type: application/xml
Content-Length: 52

<user>
  <id>3</id>
  <name>Cecilia</name>
</user>
```

```
HTTP/1.1 204 No Content
```

PUT can also be used to create a resource if you know which URI it should have in advance.

REST examples



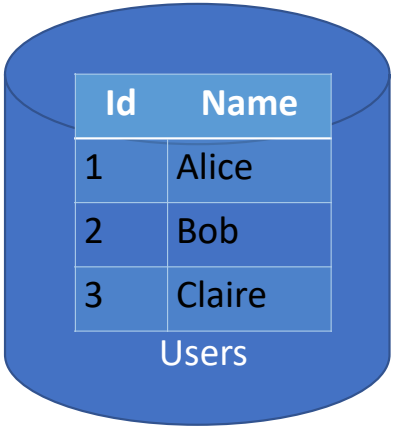
A server with information about users.

- The DELETE method is used to delete a resource.

```
DELETE /users/2 HTTP/1.1
Host: the-website.com
```

```
HTTP/1.1 204 No Content
```

REST examples



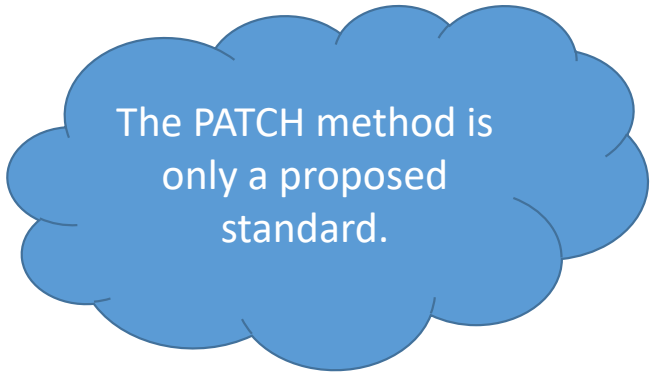
A server with information about users.

- The PATCH method is used to update parts of a resource.

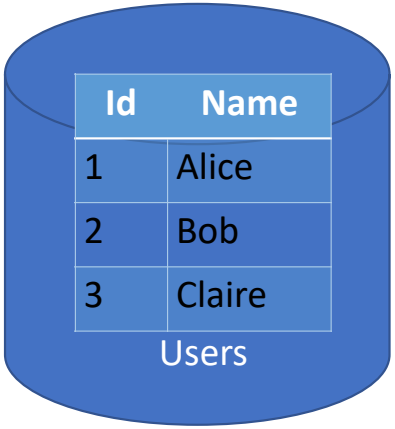
```
PATCH /users/1 HTTP/1.1
Host: the-website.com
Content-Type: application/xml
Content-Length: 37

<user>
  <name>Amanda</human>
</user>
```

```
HTTP/1.1 204 No Content
```



REST examples



A server with information about users.

- What if something goes wrong?
  - Use the HTTP status codes to indicate success/failure.

```
GET /users/999 HTTP/1.1
Host: the-website.com
Accept: application/json
```

```
HTTP/1.1 404 Not Found
```

Summary:

REST is an architectural style, not a specification. We can check how RESTful is an API from Fielding's definition with some questions.

- The client-server constraint is a given.
- Does the API use caching either by shared caches e.g. Redis, or browser caches with HTTP headers? *(caching)*
- Does the API have a single high-level purpose that contributes to (or able to support) a layered system? *(layered system)*
- Are resource URL paths labeled as nouns and structured hierarchically, increasing in specificity left to right? *(uniform interface)*
- If multiple representations of a resource are available, are they selected by HTTP headers e.g. Content-Type? *(uniform interface)*
- Is the protocol implemented to it's specification e.g. HTTP? This includes effective use of request methods like GET, POST, and PUT. *(uniform interface)*
- Does the server provide links (hypermedia) to the client for application state changes? *(uniform interface)*
- Are servers stateless between requests with session state stored on the client? *(stateless)*

Good recommendations:

- Web API Design - Crafting Interfaces that Developers Love
  - https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf

Designing a REST api

How should you think?

- Make it as easy as possible to use by other programmers.

Facebook:

- Always return 200 OK.
- GET /v2.7/{user-id}
- GET /v2.7/{post-id}
- GET /v2.7/{user-id}/friends
- GET /v2.7/{object-id}/likes

Designing a REST api

How should you think?

- Make it as easy as possible to use by other programmers.

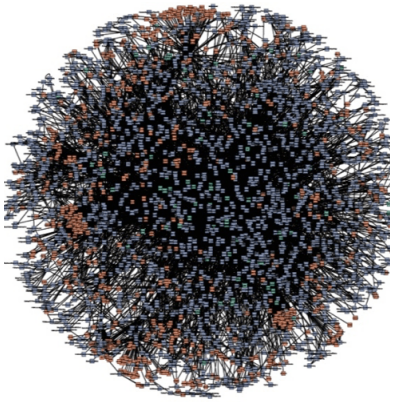
Twitter:

- Only use GET and POST.
- GET /1.1/users/show.json?user_id=2244994945
- POST /1.1/favorites/destroy.json?id=24313812895991398 6

CS-552/452 Introduction to Cloud Computing

3. Micro-services (Part III)

Other Inter-connect APIs

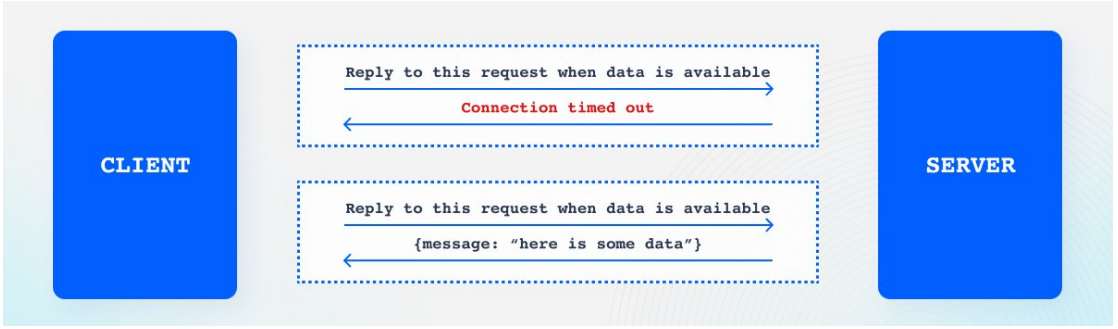


HTTP/1

- With HTTP/1, clients and servers communicate by exchanging individual messages.
- These messages are sent as regular text messages over a TCP connection.
  - Can be encrypted via HTTPs
- HTTP requests can only flow in one direction
  - From a client to a server: There is no way for the server to initiate communication with the client; it can only respond to requests.
- HTTP is perfect for traditional web and client applications, where information is fetched on an as-needed basis.

Limitations of HTTP/1

- Real-time
  - Inefficient when messages need to be sent in real-time from the client to the server and vice versa
  - E.g., if new information is available on the server that needs to be shared with the client, this transaction can only occur once the client initiates a request.
- Short polling
  - The client repeatedly sends requests to the server until it responds with new data.
- Long polling
  - A single request is made from the client, and then the server keeps that connection open until new data is available and a response can be sent.

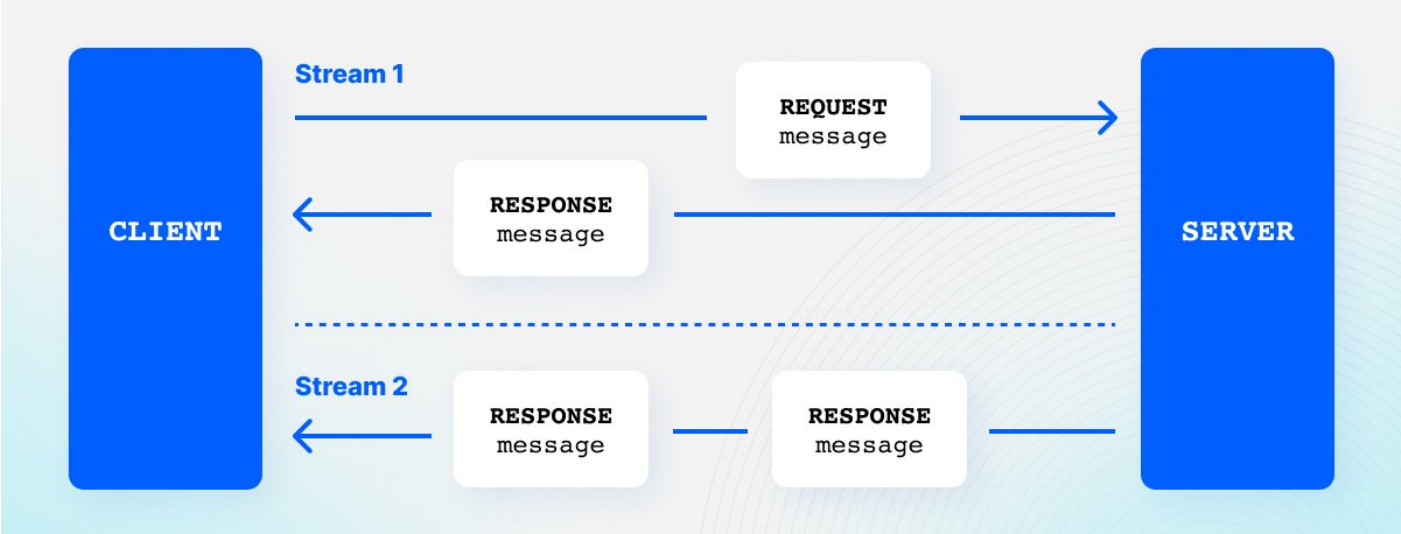


HTTP/1.1 and above

- HTTP/1.0 -- Opening a new TCP connection for each request became a problem as the web evolved.
  - Challenge: The number of media and files a browser needed to retrieve became more.
- HTTP/1.1
  - A persistent TCP connection for multiple requests/responses
    - Better performance than one connection per request
  - But not at the same time
    - Serialized protocol with where one must send a request and wait for the response
  - Each request contains a full header
    - Message size bloated
- HTTP/2 (41% of top websites support HTTP/2 in 2022)
  - Multiplexing -- simultaneously sends and receives multiple HTTP requests and responses over a single TCP connection.
  - Header compression -- avoids sending the same plain text headers over and over.
  - Prioritization -- allowing the client (developer) to specify the priority of the resources it needs
  - It uses server push to send data to the client before it requests it. This can be used to improve loading times by eliminating the need for the client to make multiple requests.

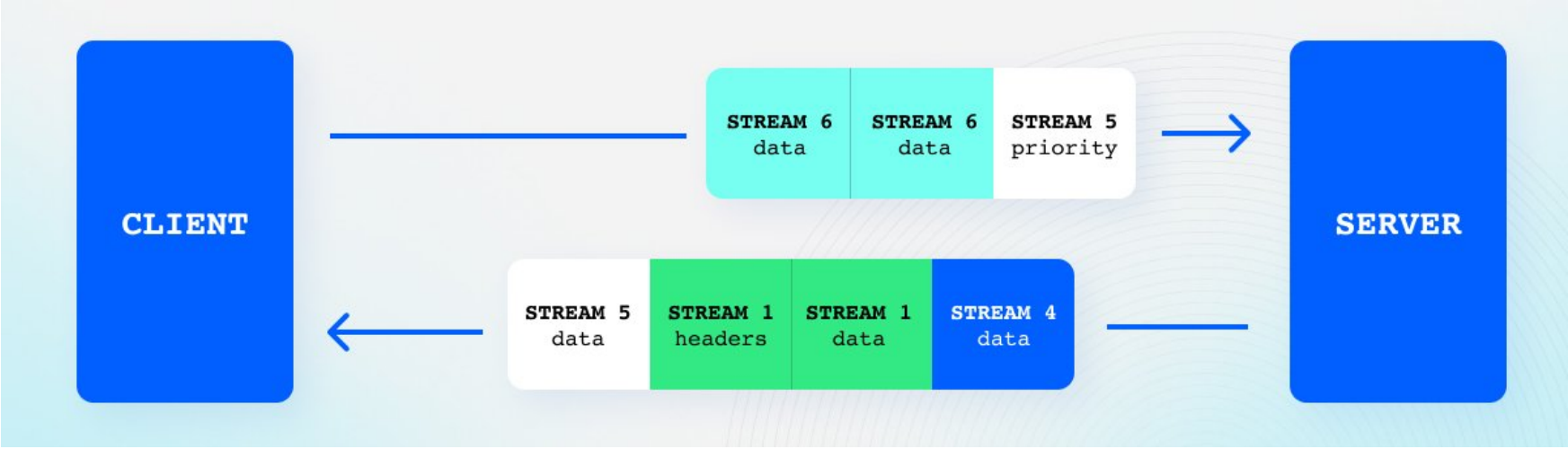
More Details about HTTP/2

- The basic protocol unit in HTTP/2 is a frame.
  - Header frame, data frame, priority frame, etc.
- Frames are combined to form a message
  - E.g., a header frame + a data frame forms a “request” from the client
  - E.g., a header frame + multiple data frames forms a “response” from the server



More Details about HTTP/2

- A series of messages can be part of a stream, allowing for a bidirectional data flow between the client and server.
- A “stream” is an independent, bidirectional sequence of frames exchanged between the client and server within an HTTP/2 connection.
  - Multiple streams share the same TCP connection
  - Frames of different streams are interleaved

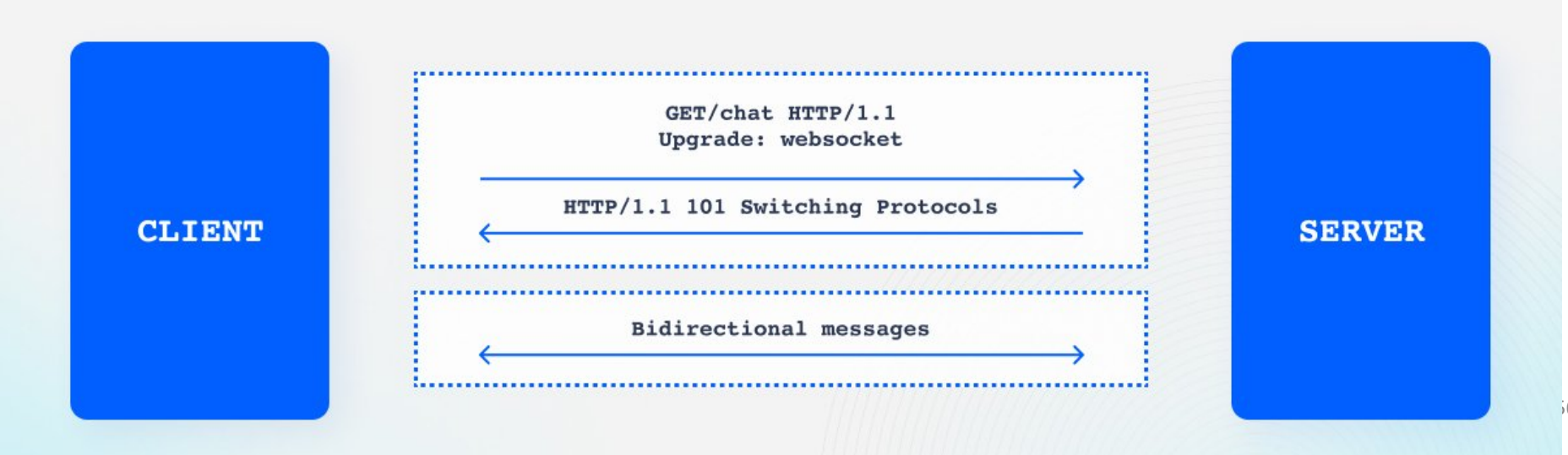


WebSockets

- Long before HTTP/2.
- The goal of this technology is to provide a mechanism for browser-based applications that need efficient two-way communication with servers that do not rely on opening multiple HTTP connections.
- WebSockets were invented to enable full-duplex communication between a client and server, which allows for data to travel both ways through a single open connection immediately.
- This improves speed and real-time capability compared to the original HTTP/1
  - Faster than RESTful APIs
- WebSocket does not have a format it complies to. You can send any data, text, or bytes – this flexibility is one of the reasons why WebSockets are popular.

WebSockets

- To establish a WebSocket connection the client and server first need to perform a handshake over a normal HTTP/1 connection.
  - With an upgrade header to switch from HTTP over to WebSockets
- Once the two-way communication channel is established, the client and server can send messages back and forth
  - WebSockets require a uniform resource identifier (URI) with a ws:// or wss://



When to Use WebSockets

- Websockets are best suited for applications that need two-way communication in real-time and when small pieces of data need to be transmitted quickly, for example:
  - Chat applications
  - Multiplayer games
  - Collaborative editing applications
  - Live sports ticker
  - Stock trading application
  - Real-time activity feeds

gRPC

- gRPC was created by Google in 2015 to speed up data transmission between microservices and other systems that need to interact.
- Key differences from REST APIs
  - A more compact data format, Protobuf
    - Language-neutral and platform-neutral
    - Serialize and deserialize structured data to communicate via binary
    - Faster data transmission due to lightweight data format
  - Built on HTTP 2 (instead of HTTP 1.1)
    - Published in 2015
    - Not use plain text, but binary format encapsulation.
    - Faster than HTTP 1.1 – due to multiplexing

7-10 x faster
than REST APIs

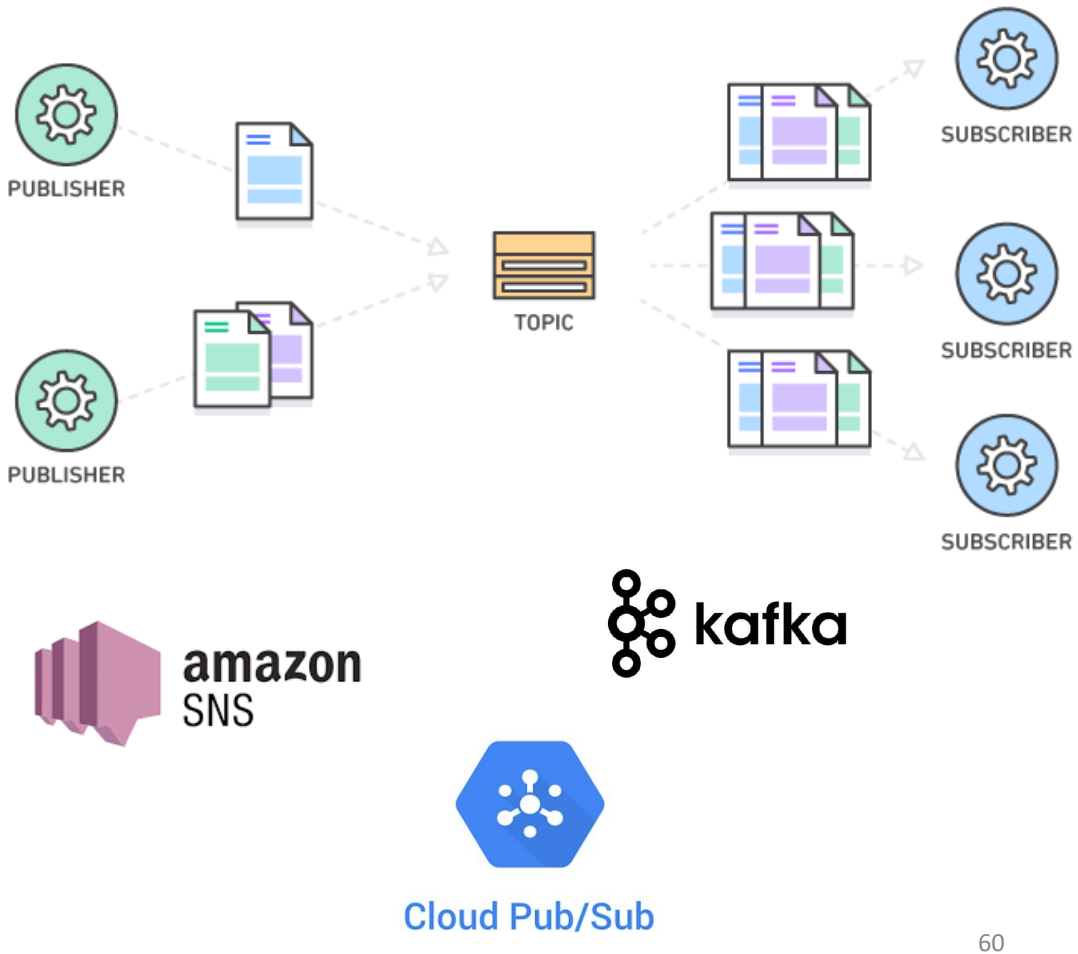
  - Support of better streaming
    - One client request vs. a stream of responses with a status termination response
    - A stream of client requests with a status message vs. one server response
    - Bidirectional -- A client and server transmit data to one another in no particular order.

Key Characteristics of gRPC

Characteristic	**gRPC**	**REST API**
HTTP Protocol	*HTTP 2*	*HTTP 1.1*
Messaging Format	*Protobuf (Protocol Buffers)*	*JSON (usually) or XML and others*
Code Generation	*Native Protoc Compiler*	*Third-Party Solutions Like Swagger*
Communication	*Unary Client-Request or Bidirectional/Streaming*	*Client-Request Only*
Implementation Time	*45 Minutes*	*10 Minutes*

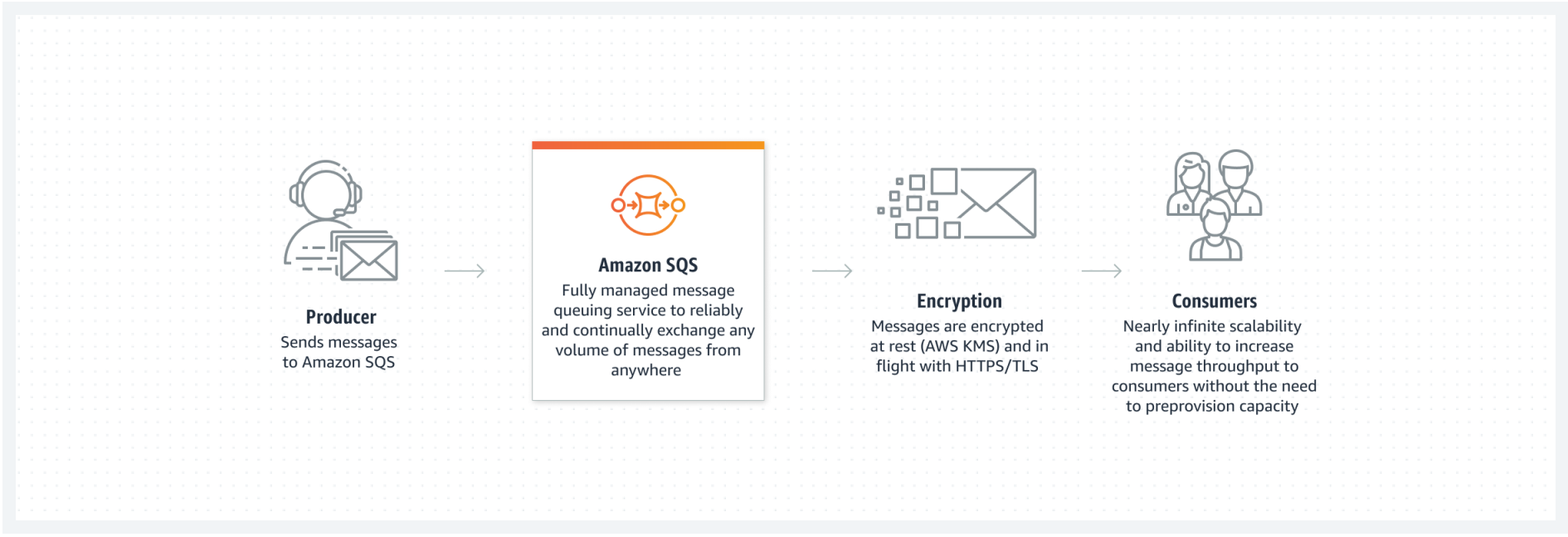
Cloud Messaging Services – Pub/sub

- Publish/subscribe messaging, or pub/sub messaging, is a form of asynchronous service-to-service communication used in serverless and microservices architectures.
  - AWS Simple Notification Service (SNS), Google Cloud Pub/sub (GCM), Apache Kafka
- Publisher broadcast a message by pushing it to a “topic”, while all subscribers subscribing to that topic will receive the message
- Used to enable event-driven architectures.
  - Mobile Push Messaging – notifications, chat applications, etc.



Cloud Messaging Services – Queuing

- Messages are sent by the producer and queued in a queueing service
  - AWS Simple Queue Service (SQS)
- Messages are retrieved by the consumer explicitly
  - E.g. pipelined data processing



Sources

- An Introduction To REST API: https://www.slideshare.net/AniruddhBhilvare/an-introduction-to-rest-api
- HTTP, WebSocket, gRPC or WebRTC: Which Communication Protocol is Best For Your App? https://getstream.io/blog/communication-protocols/
- gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs? https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/#:~:text=%E2%80%9CgRPC%20is%20roughly%207%20times,HTTP%2F2%20by%20gRPC.%E2%80%9D
- Exploring REST API Architecture