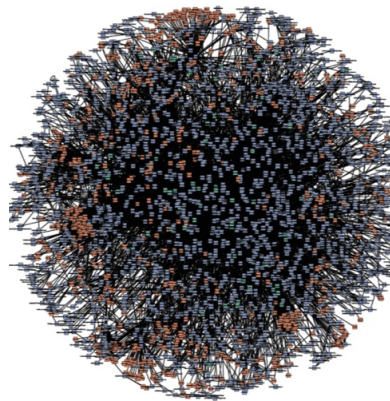


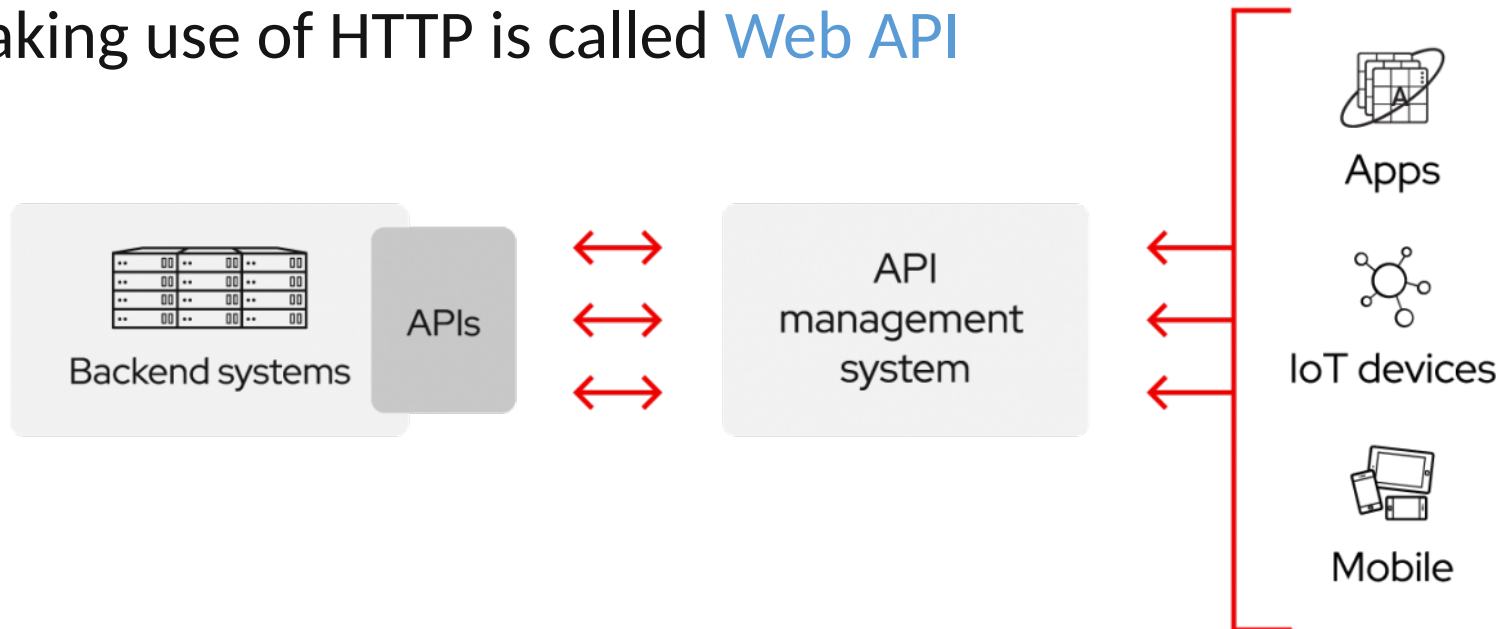
# CS-552/452 Introduction to Cloud Computing

Microservices APIs



# What is an API?

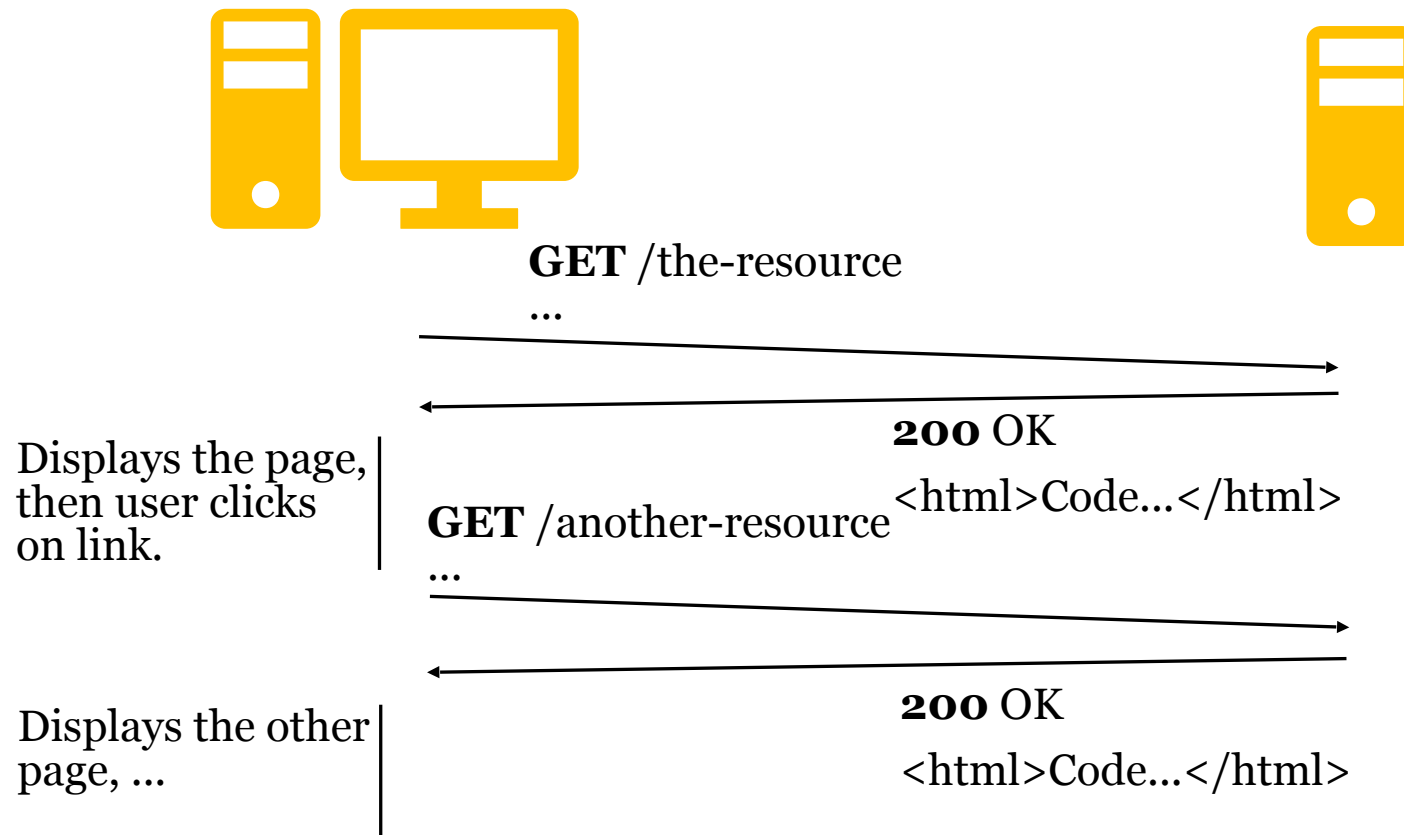
- A set of **definitions** and **protocols** for building and integrating application software.
  - The communication channel between services/components
- APIs let one product/service communicate with other products and services without having to know how they're implemented.
- An API making use of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



# Type of Messaging Systems (Recap)

- 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.

# 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.

# 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

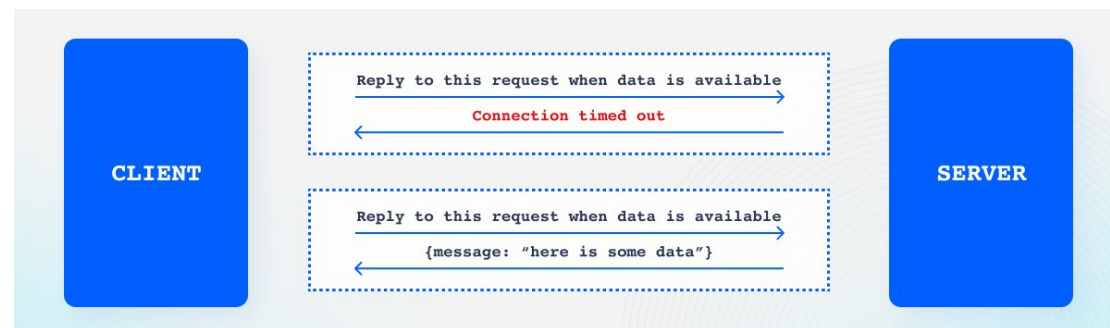
- Inefficient when messages need to be sent in real-time from the client to the server and vice versa
- E.g., if new information is available on the server that needs to be shared with the client, this transaction can only occur once the client initiates a request.

- 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.



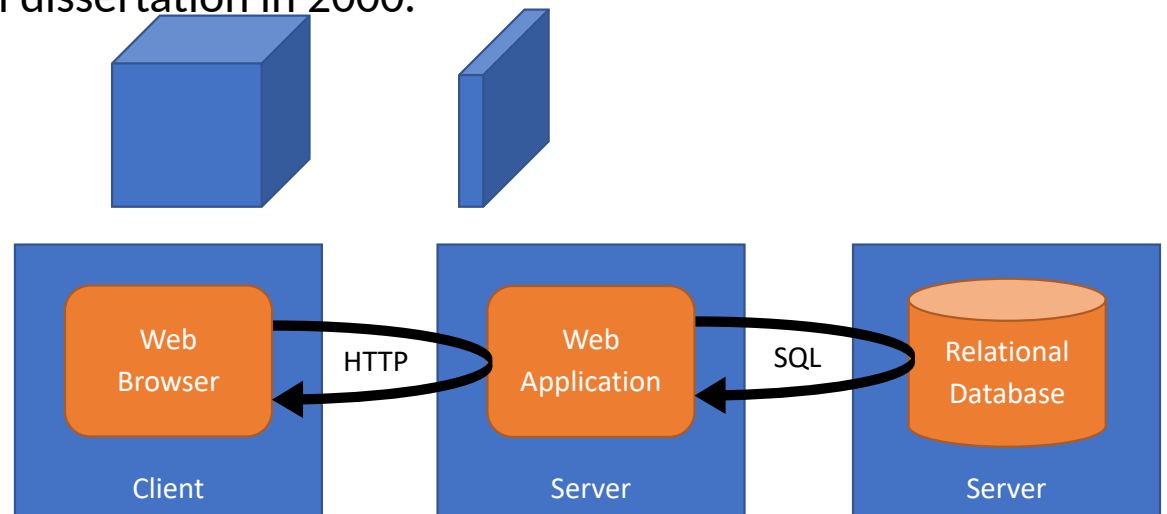
# Different Types of Web APIs

- 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)?

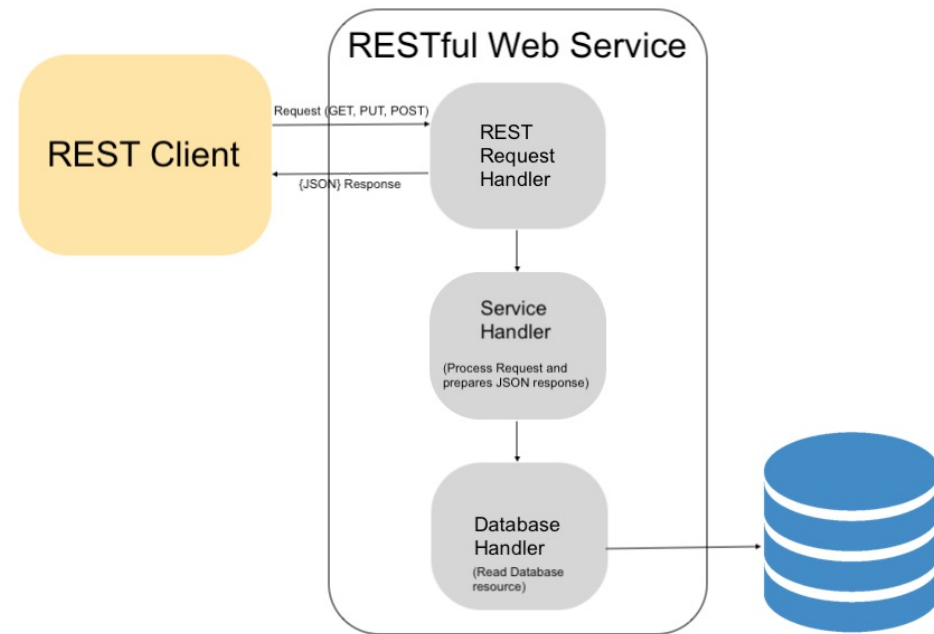
- REST APIs are typically used to access and manipulate data on a remote server.
- They can be used by any type of client application, including web browsers, mobile apps, and other software.
- An *architectural style for distributed hypermedia systems*
  - By Roy Thomas Fielding in his doctoral dissertation in 2000.

- Design Principles of REST:
  1. Client - Server separation
  2. Statelessness
  3. Cacheability
  4. Uniform Interface
  5. Layered System
  6. Code-On-Demand (optional)



# 1. Client-server separation

- 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
- A REST **client** accesses and modifies the REST resources

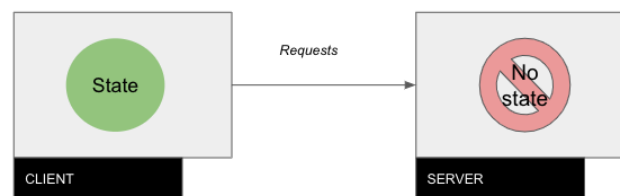


- HTTP vs. REST

- HTTP is a protocol that defines how messages are formatted and transmitted over the World Wide Web.
- REST API is an architectural style that defines how resources are accessed and manipulated. It can be used to design any type of API, but it is most commonly used with HTTP.

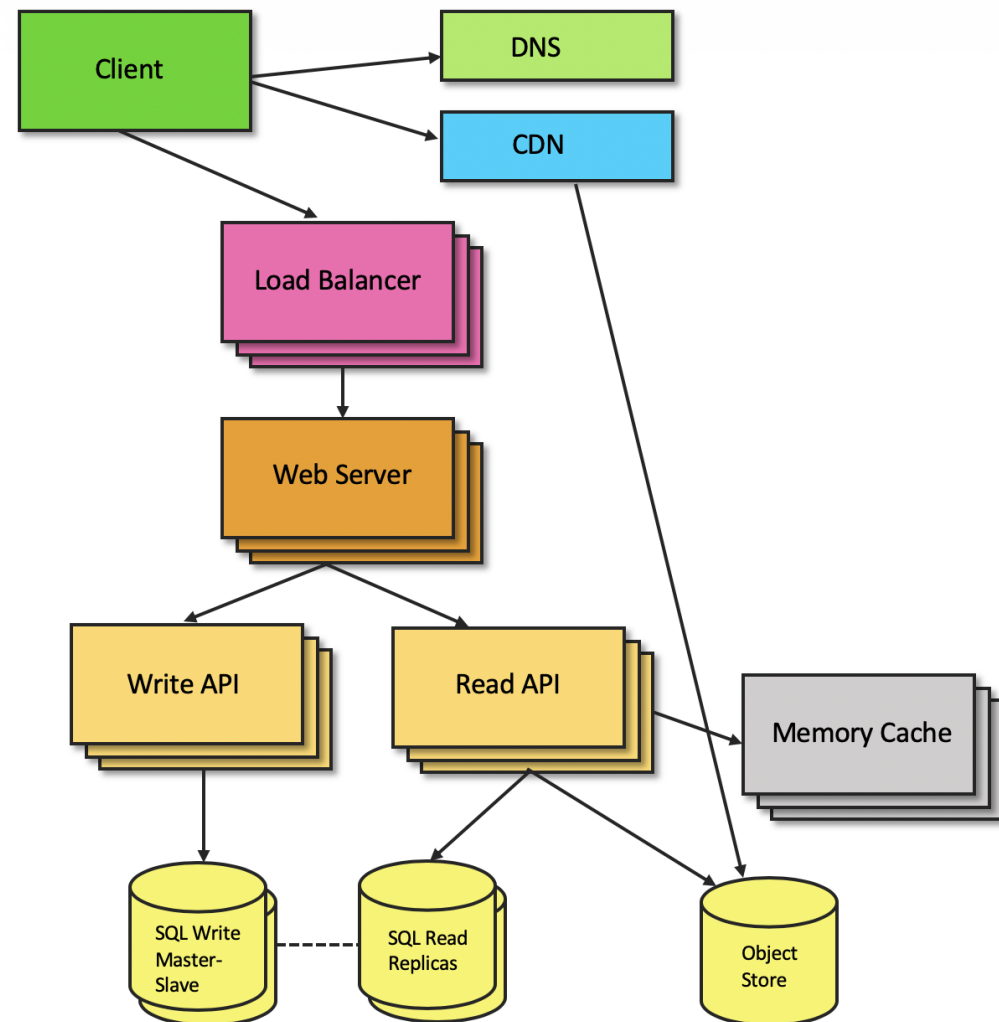
## 2. Statelessness

- 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 statelessness
  - **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 server does not save session context that can get 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.

# REST examples

A server with information about users.

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

Id	Name
1	Alice
2	Bob

Users

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

application/  
xml was popular  
before JSON.

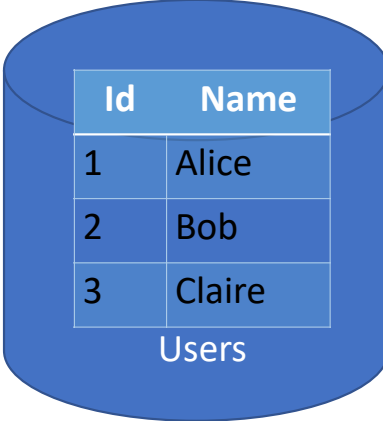
```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 66
```

```
[
  {"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!



Id	Name
1	Alice
2	Bob
3	Claire

Users

```
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

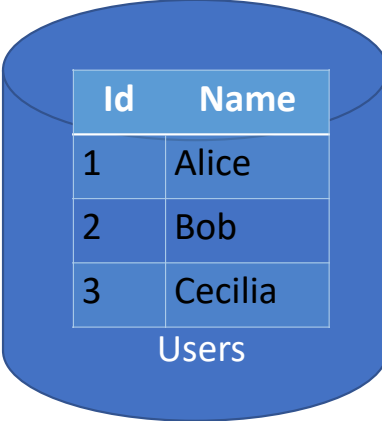
{"id": 3, "name": "Claire"}
```



# REST examples

A server with information about users.

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



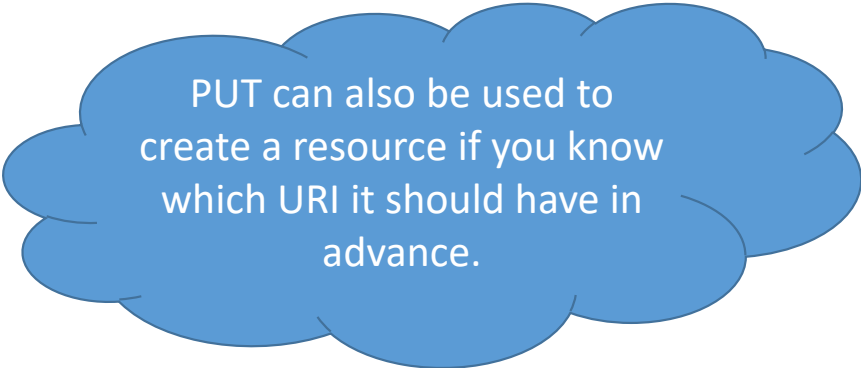
Id	Name
1	Alice
2	Bob
3	Cecilia

Users

```
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
```

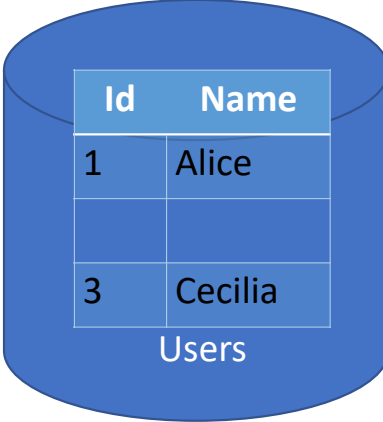


PUT can also be used to create a resource if you know which URI it should have in advance.

# REST examples

A server with information about users.

- The DELETE method is used to delete a resource.



Id	Name
1	Alice
3	Cecilia

Users

```
DELETE /users/2 HTTP/1.1
```

```
Host: the-website.com
```

```
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
```

# REST Summary:

REST is an architectural style, not a specification. We can check how RESTful is an API 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>

# 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.

# When to Use WebSockets

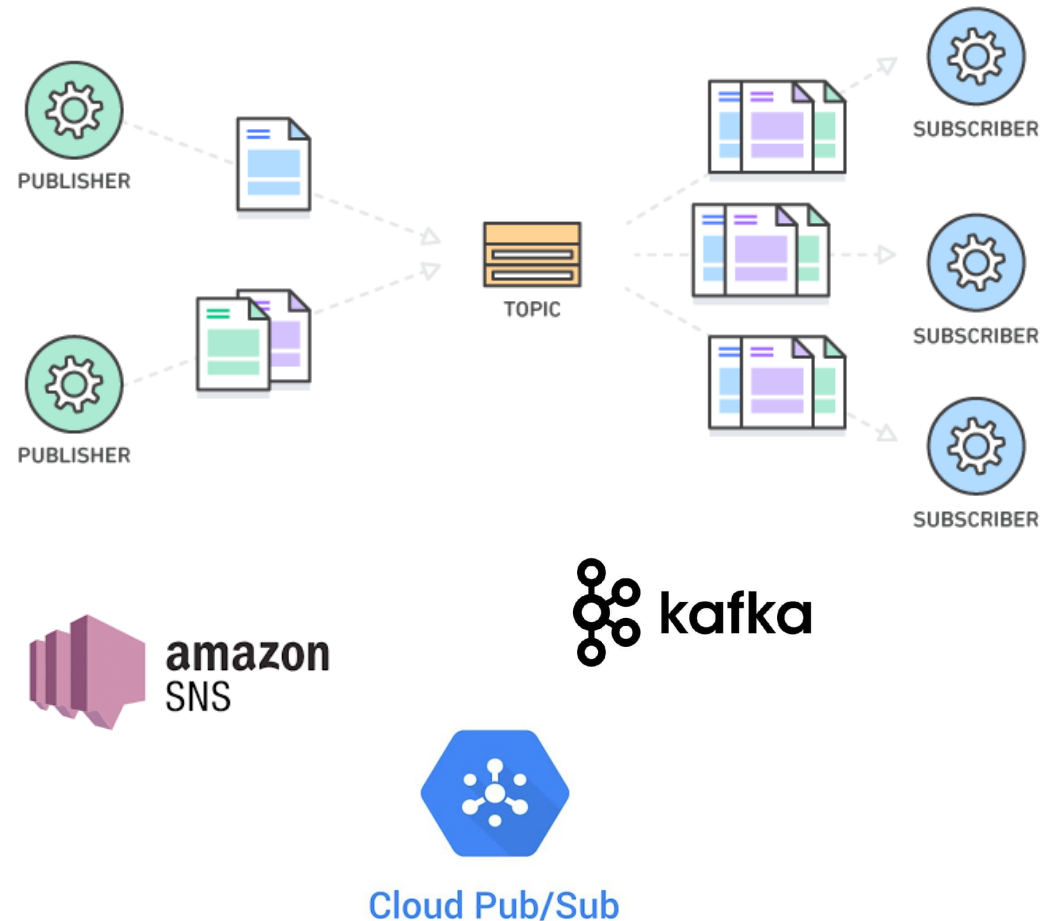
- Websockets are best suited for applications that need **two-way communication** in **real-time** and when small pieces of data need to be transmitted quickly, for example:
  - Chat applications
  - Multiplayer games
  - Collaborative editing applications
  - Live sports ticker
  - Stock trading application
  - 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
    - Serialize and deserialize structured data to communicate via [binary](#)
  - Built on HTTP 2 (instead of HTTP 1.1)
    - Does not use plain text, but binary format encapsulation.
    - Faster than HTTP 1.1
  - Support of better streaming and bidirectional communication

# Cloud Messaging Services – Pub/sub

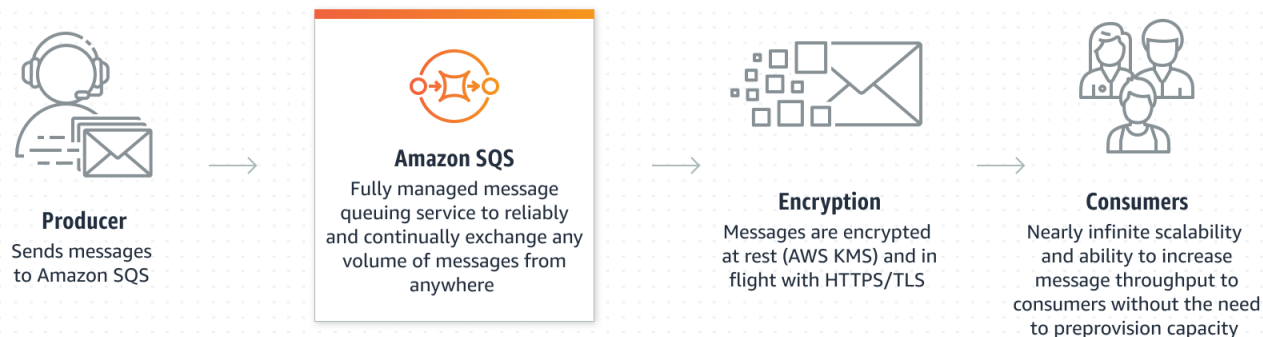
- **Publish/subscribe** messaging, or pub/sub messaging, is a form of asynchronous service-to-service communication used in serverless and microservices architectures.
  - AWS Simple Notification Service (SNS), Google Cloud Pub/sub (GCM), Apache Kafka
- Publisher broadcasts a message by pushing it to 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](#)