Lecture 13

Microservices

Service-Oriented Architecture

REST API

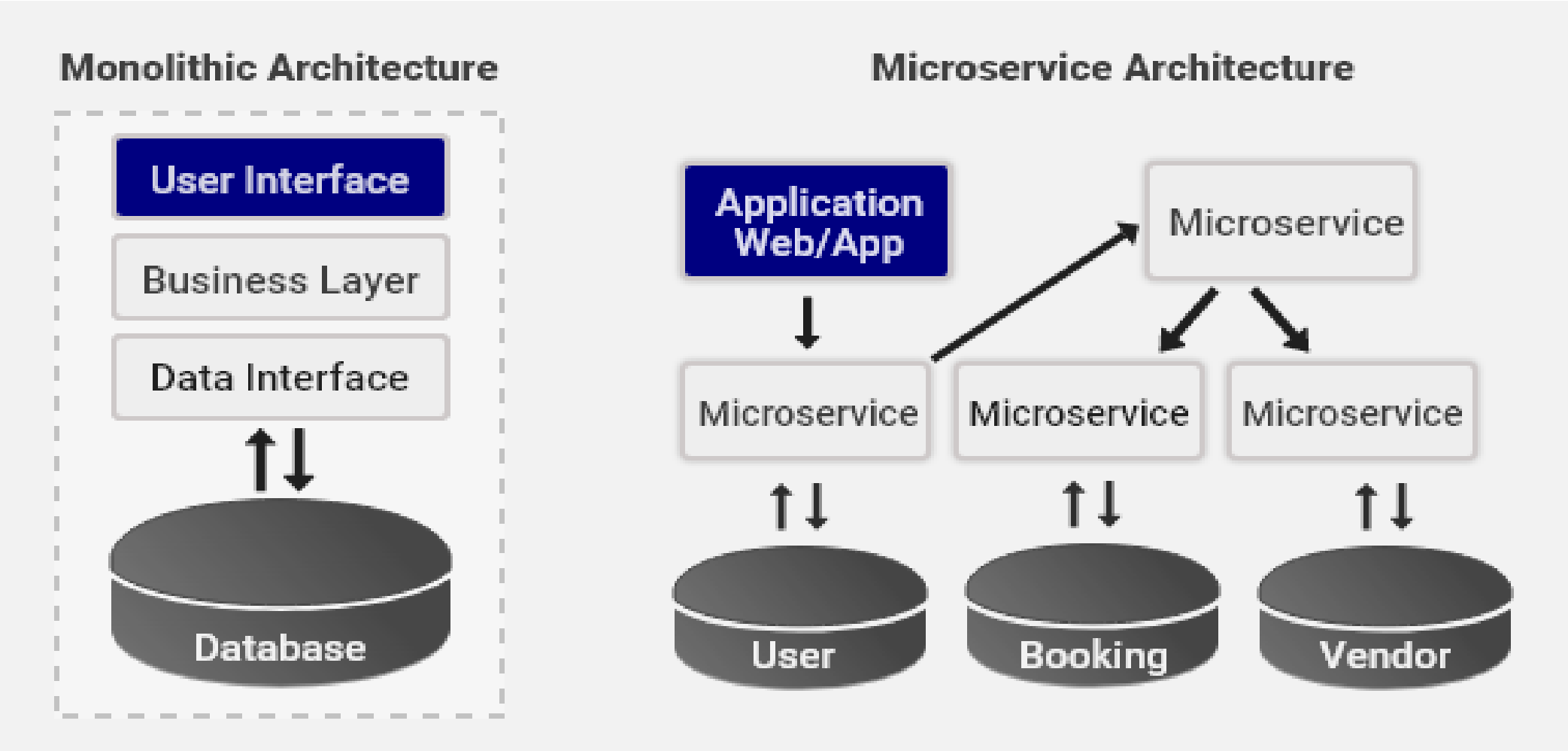
What are Microservices ?

Microservices

- Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs (Application Programming Interface). These services are owned by small, self-contained teams.

- Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

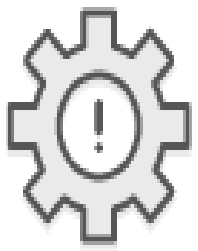
Monolithic vs. Microservices Architecture



Characteristics of Microservices

Autonomous: Each component service in a microservices architecture can be developed, deployed, operated, and scaled without affecting the functioning of other services. Services do not need to share any of their code or implementation with other services. Any communication between individual components happens via well-defined APIs.

Characteristics of Microservices



Specialized: Each service is designed for a set of capabilities and focuses on solving a specific problem. If developers contribute more code to a service over time and the service becomes complex, it can be broken into smaller services.

Benefits of Microservices

- **Agility:** Microservices foster an organization of small, independent teams that take ownership of their services. Teams act within a small and well understood context, and are empowered to work more independently and more quickly. This shortens development cycle times. You benefit significantly from the aggregate throughput of the organization.
- **Flexible Scaling:** Microservices allow each service to be independently scaled to meet demand for the application feature it supports. This enables teams to right-size infrastructure needs, accurately measure the cost of a feature, and maintain availability if a service experiences a spike in demand.

Benefits of Microservices

- **Easy Deployment:** Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn't work. The low cost of failure enables experimentation, makes it easier to update code, and accelerates time-to-market for new features.
- **Technological Freedom:** Microservices architectures don't follow a "one size fits all" approach. Teams have the freedom to choose the best tool to solve their specific problems. As a consequence, teams building microservices can choose the best tool for each job.

Benefits of Microservices

- **Reusable Code:** Dividing software into small, well-defined modules enables teams to use functions for multiple purposes. A service written for a certain function can be used as a building block for another feature. This allows an application to bootstrap off itself, as developers can create new capabilities without writing code from scratch.
- **Resilience:** Service independence increases an application's resistance to failure. In a monolithic architecture, if a single component fails, it can cause the entire application to fail. With microservices, applications handle total service failure by degrading functionality and not crashing the entire application.

Service-Oriented Architecture (SOA)

SOA

Service-oriented architecture (SOA) is a method of software development that uses software components called services to create business applications. Each service provides a business capability, and services can also communicate with each other across platforms and languages. Developers use SOA to reuse services in different systems or combine several independent services to perform complex tasks.

SOA

For example, multiple business processes in an organization require the user authentication functionality. Instead of rewriting the authentication code for all business processes, you can create a single authentication service and reuse it for all applications. Similarly, almost all systems across a healthcare organization, such as patient management systems and electronic health record (EHR) systems, need to register patients. These systems can call a single, common service to perform the patient registration task.

Benefits of SOA

- **Faster time to market:** Developers reuse services across different business processes to save time and costs. They can assemble applications much faster with SOA than by writing code and performing integrations from scratch.

- **Efficient maintenance:** It's easier to create, update, and debug small services than large code blocks in monolithic applications. Modifying any service in SOA does not impact the overall functionality of the business process.

- **Greater adaptability:** SOA is more adaptable to advances in technology. You can modernize your applications efficiently and cost effectively. For example, healthcare organizations can use the functionality of older electronic health record systems in newer cloud-based applications.

Basic Principle SOA

- **Interoperability:** Each service in SOA includes description documents that specify the functionality of the service and the related terms and conditions. Any client system can run a service, regardless of the underlying platform or programming language. For instance, business processes can use services written in both C# and Python. Since there are no direct interactions, changes in one service do not affect other components using the service.

Basic Principle SOA

- **Loose coupling:** Services in SOA should be loosely coupled, having as little dependency as possible on external resources such as data models or information systems. They should also be stateless without retaining any information from past sessions or transactions. This way, if you modify a service, it won’t significantly impact the client applications and other services using the service.
- **Abstraction:** Clients or service users in SOA need not know the service's code logic or implementation details. To them, services should appear like a black box. Clients get the required information about what the service does and how to use it through service contracts and other service description documents.

Basic Principle SOA

- **Granularity:** Services in SOA should have an appropriate size and scope, ideally packing one discrete business function per service. Developers can then use multiple services to create a composite service for performing complex operations.

Components of SOA

- Service
- Service Provider
- Service Consumer
- Service Registry

Working of SOA

In service-oriented architecture (SOA), services function independently and provide functionality or data exchanges to their consumers. The consumer requests information and sends input data to the service. The service processes the data, performs the task, and sends back a response. For example, if an application uses an authorization service, it gives the service the username and password. The service verifies the username and password and returns an appropriate response.

Communication protocols

Services communicate using established rules that determine data transmission over a network. These rules are called communication protocols. Some standard protocols to implement SOA include the following:

- Simple Object Access Protocol (SOAP)
- RESTful HTTP
- Apache Thrift
- Apache ActiveMQ
- Java Message Service (JMS)

You can even use more than one protocol in your SOA implementation.

Limitations of SOA

- Limited scalability: System scalability is significantly impacted when services share many resources and need to coordinate to perform their functionality.

- Increasing interdependency: Service-oriented architecture (SOA) systems can become more complex over time and develop several interdependencies between services. They can be hard to modify or debug if several services are calling each other in a loop. Shared resources, such as centralized databases, can also slow down the system.

- Single point of failure: For SOA implementations with an ESB, the ESB creates a single point of failure. It is a centralized service, which goes against the idea of decentralization that SOA advocates. Clients and services cannot communicate with each other at all if the ESB goes down.

ESB

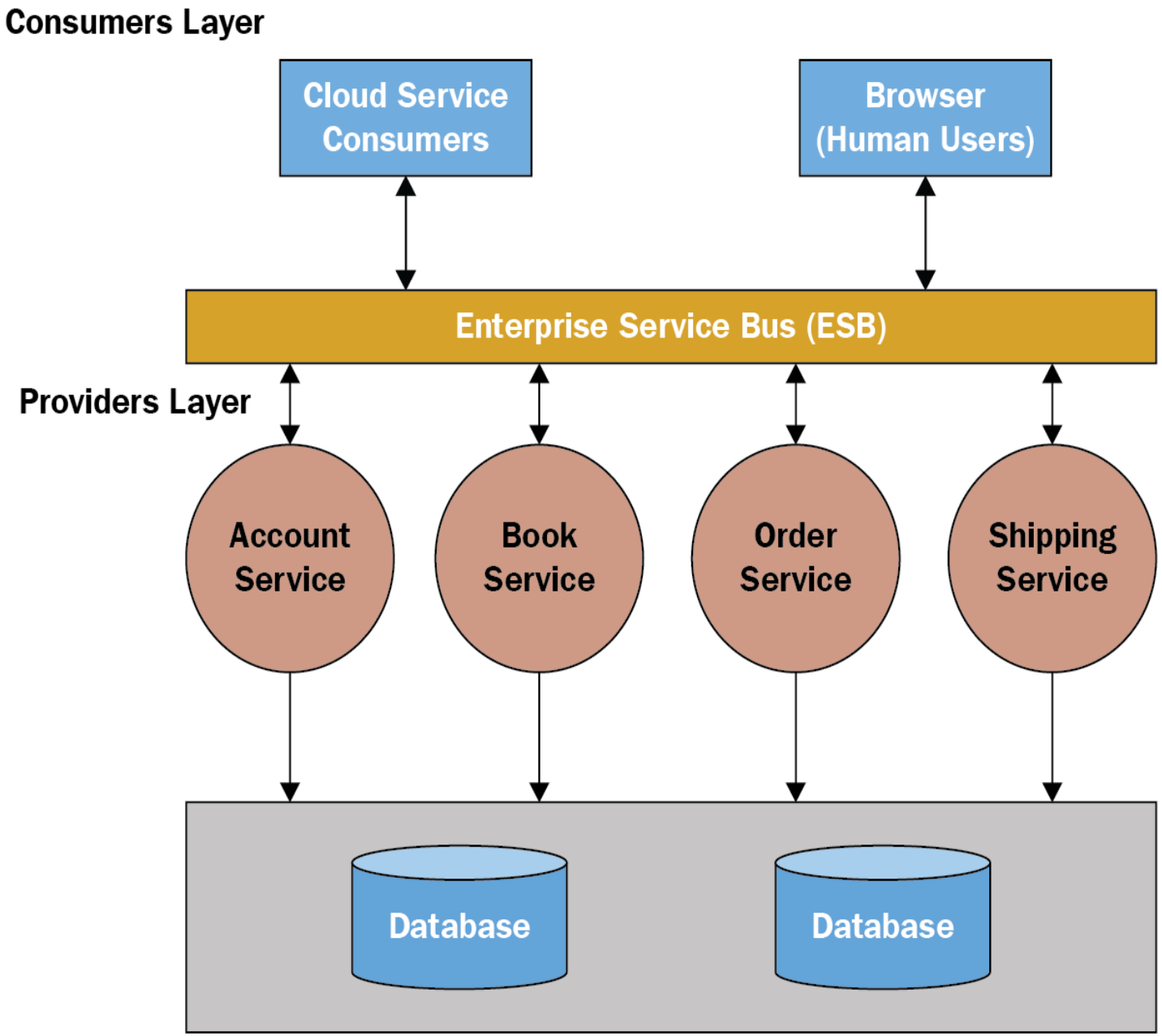
What is an ESB in service-oriented architecture?

An enterprise service bus (ESB) is software that you can use when communicating with a system that has multiple services. It establishes communication between services and service consumers no matter what the technology.

Benefits of ESB

An ESB provides communication and transformation capabilities through a reusable service interface. You can think of an ESB as a centralized service that routes service requests to the appropriate service. It also transforms the request into a format that is acceptable for the service's underlying platform and programing language.

SOA


Consumers Layer
Cloud Service Consumers
Browser (Human Users)
Enterprise Service Bus (ESB)
Providers Layer
Account Service
Book Service
Order Service
Shipping Service
Database
Database

QUIZ

What is a microservice?

a) A design used primarily in functional programming and object-oriented programming
b) A small program that represents discrete logic that executes within a well-defined boundary on dedicated hardware
c) A style of design for enterprise systems based on a loosely coupled component architecture
d) A very small piece of code that never gets any bigger than 10 lines

QUIZ

When would developers use microservices?

a) When they want to write cell phone applications that run quickly
b) When they work with ephemeral nano technology
c) When they need to create large, enterprise-level applications that are subject to changes on a frequent basis
d) When they create applications specifically for scientific test equipment

QUIZ

Applications that are accessed through browser are

a) Web services
b) Web applications
c) Desktop applications
d) None of these

QUIZ

Services minimize dependencies on each other.
(Choose one principle of SOA design).

a) Service Abstraction
b) Loose coupling
c) Tight coupling
d) None of these

QUIZ

Services hide the logic they encapsulate from the outside world. (Choose one principle of SOA design).

a) Loose coupling
b) Service Abstraction
c) Service reusability
d) None of these

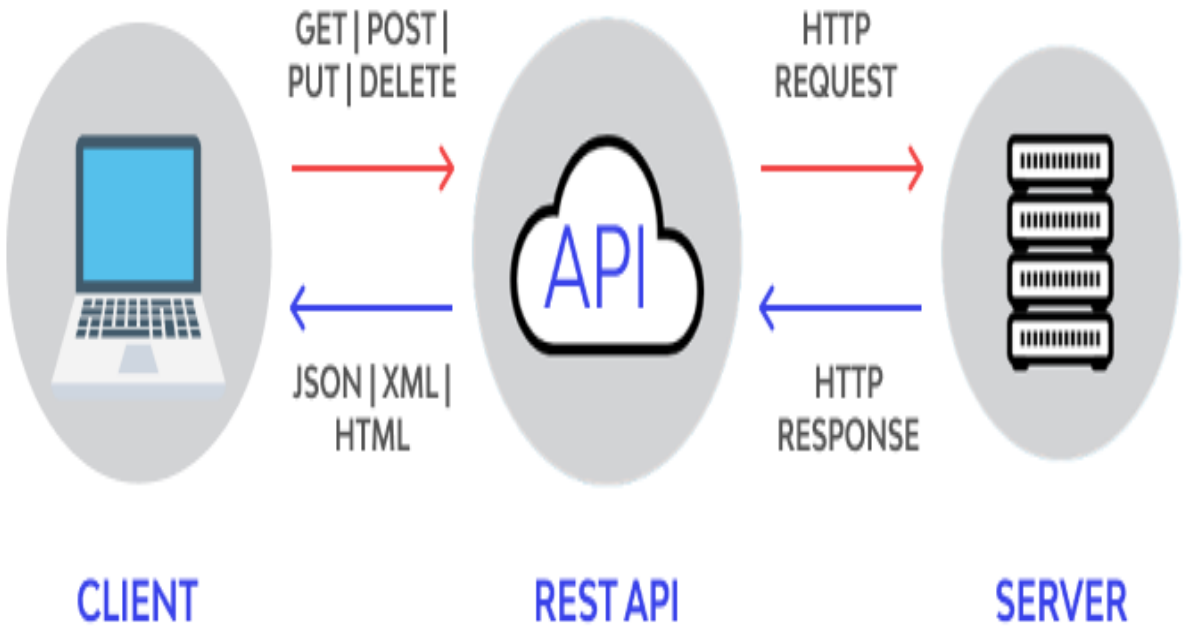
REST API

API

APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols. For example, the weather bureau's software system contains daily weather data. The weather app on your phone "talks" to this system via APIs and shows you daily weather updates on your phone.

REST API

REST stands for Representational State Transfer. REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP.



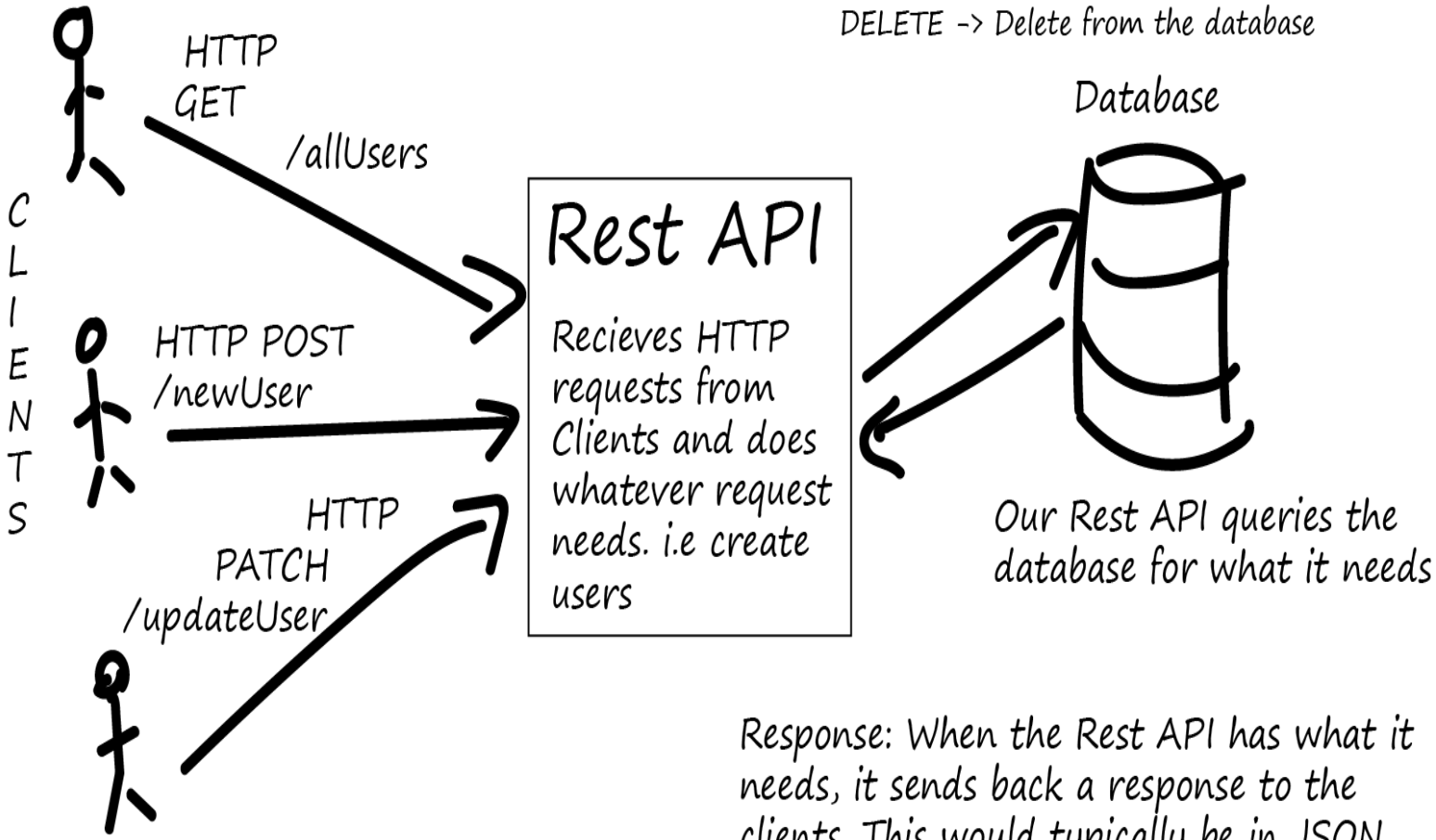
REST API

The main feature of REST API is statelessness. Statelessness means that servers do not save client data between requests. Client requests to the server are similar to URLs you type in your browser to visit a website. The response from the server is plain data, without the typical graphical rendering of a web page.

Working of REST API

Rest API Basics

Typical HTTP Verbs:
GET -> Read from Database
PUT -> Update/Replace row in Database
PATCH -> Update/Modify row in Database
POST -> Create a new record in the database
DELETE -> Delete from the database



Our Rest API queries the database for what it needs

Our Clients, send HTTP Requests and wait for responses

Response: When the Rest API has what it needs, it sends back a response to the clients. This would typically be in JSON or XML format.

REST API

- **GET:** The HTTP GET method is used to **read** (or retrieve) a representation of a resource. In the safe path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).
- **POST:** The POST verb is most often utilized to **create** new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

REST API

- **PUT:** It is used for **updating** the capabilities. However, PUT can also be used to **create** a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation.

REST API

- **PATCH:** It is used to **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource. This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.
- **DELETE:** It is used to **delete** a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body.

Thanks