Advanced Software Engineering Course

SaaS Application Architecture: Microservices, APIs, and REST

Sadegh Sulaimany

info@Bioinfotmation.ir

دانشگاه کردستان
University of Kurdistan
زانکۆی کوردستان

SBNA
Social & Biological Network Analysis Laboratory

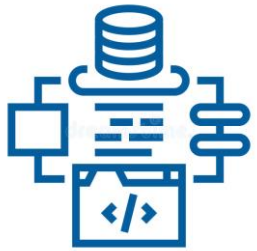
Initial assessment

1. What is RESTful URL?
2. What are the problems of SOA (Microservice Based) software development?

Agenda

- Software Archtectures
  - Client-Server Architecture
- SaaS communication Uses HTTP Routes
- Service-Oreinted Architecture
- RESTful APIs

Software architecture



- Definition
  - Software architecture is the high-level structural organization of a software system that defines:
    - The system's components
    - Their external properties
    - The relationships and interactions between these components
  - Key Characteristics of Software Architecture:
    - Provides a blueprint for the entire system
    - Focuses on the overall structure and system-level properties
    - Addresses quality attributes like performance, scalability, and reliability
    - Establishes constraints and patterns for system development

Software Architecture vs. Software Design

SOFTWARE ARCHITECTURE

- Strategic, big-picture view
- Defines system-wide structural patterns
- Focuses on high-level components and their interactions
- Concerned with non-functional requirements
- Makes fundamental design choices about the system
- Typically created early in the development process

SOFTWARE DESIGN

- Tactical, detailed implementation view
- Specifies how individual components work internally
- Focuses on module-level design and algorithms
- Addresses specific implementation details
- Translates architectural decisions into concrete solutions
- Occurs after architecture is established

Software Architecture vs. Software Design

- Analogy:
  - Software Architecture = City Planning (layout, infrastructure, zones)
  - Software Design = Building Design (interior details, room layouts, specific construction methods)

- Example:
  - Architecture Decision: Choosing a microservices architecture
  - Design Decision: Implementing a specific microservice's authentication mechanism

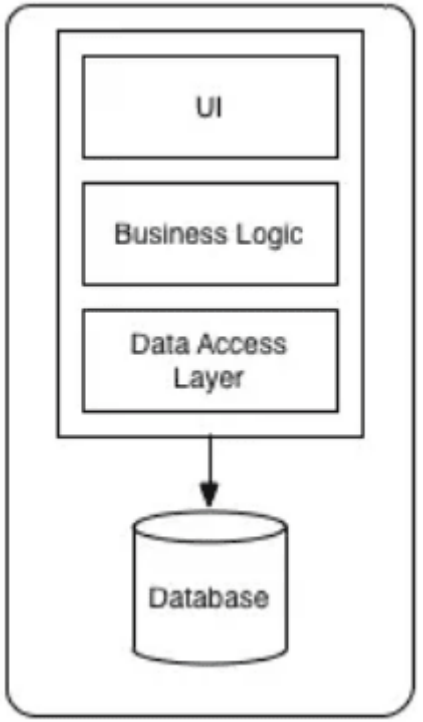
- Relationship:
  - Software design is essentially the implementation-level realization of the architectural blueprint.
  - Architecture provides the framework, while design fills in the specific details within that framework.

Popular software architectures

1. Monolithic Architecture
2. Client-Server Architecture
3. Peer-to-Peer (P2P) Architecture
4. Microservices Architecture
5. Event-Driven Architecture
6. Serverless Architecture

Monolithic Architecture

- Traditional approach where entire application is built as a single, unified unit
- All components are interconnected and interdependent
- Pros:
  - Simple to develop, easier initial deployment
- Cons:
  - Less flexible, harder to scale, more challenging to maintain



Client-Server Architecture

- One of the most common architectures where clients (user devices) request services or resources from centralized servers

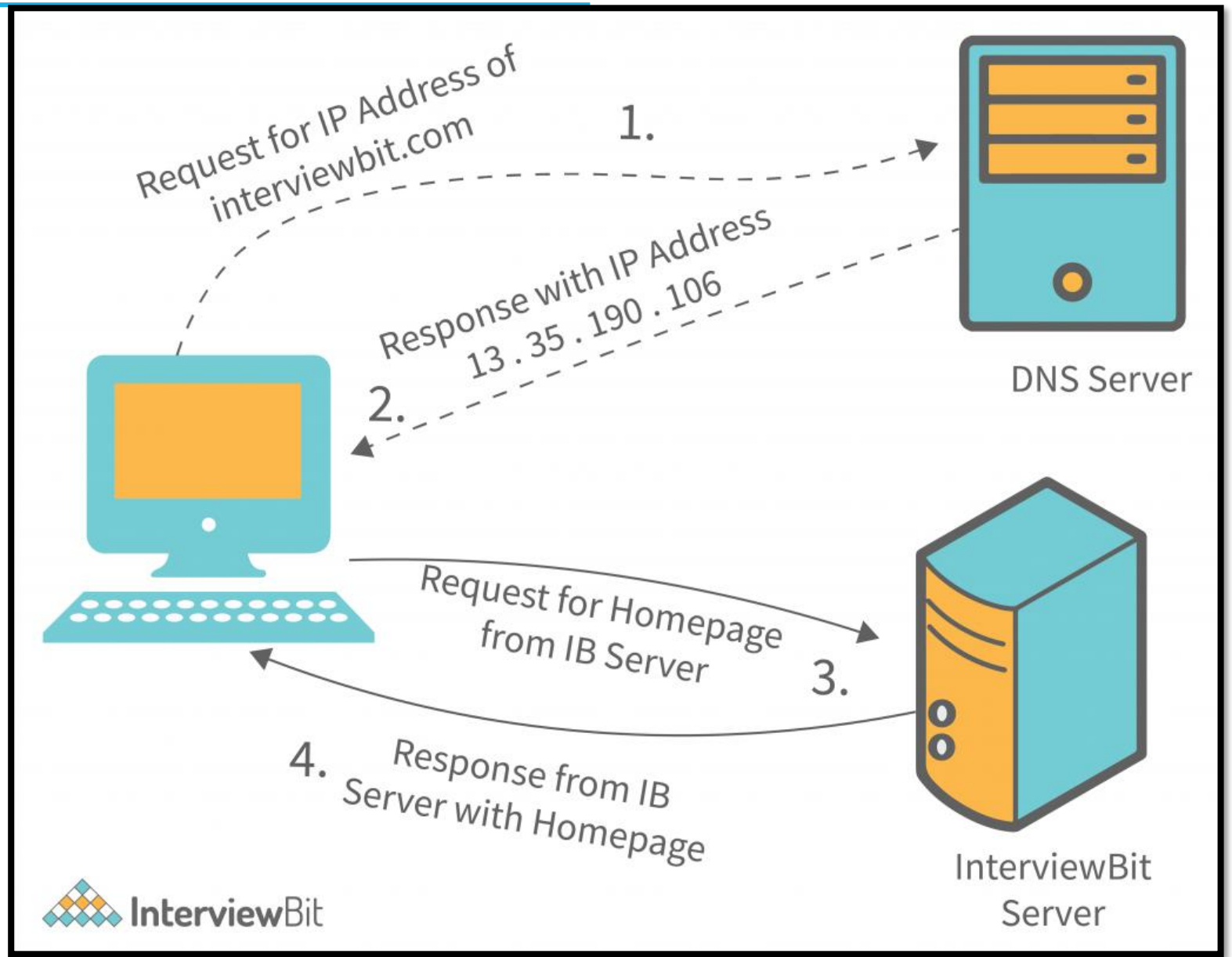
- Examples:
  - Web applications, email systems, database management systems

- Pros:
  - Centralized data management, easier security control

- Cons:
  - Potential performance bottlenecks, single point of failure

Client-Server Architecture

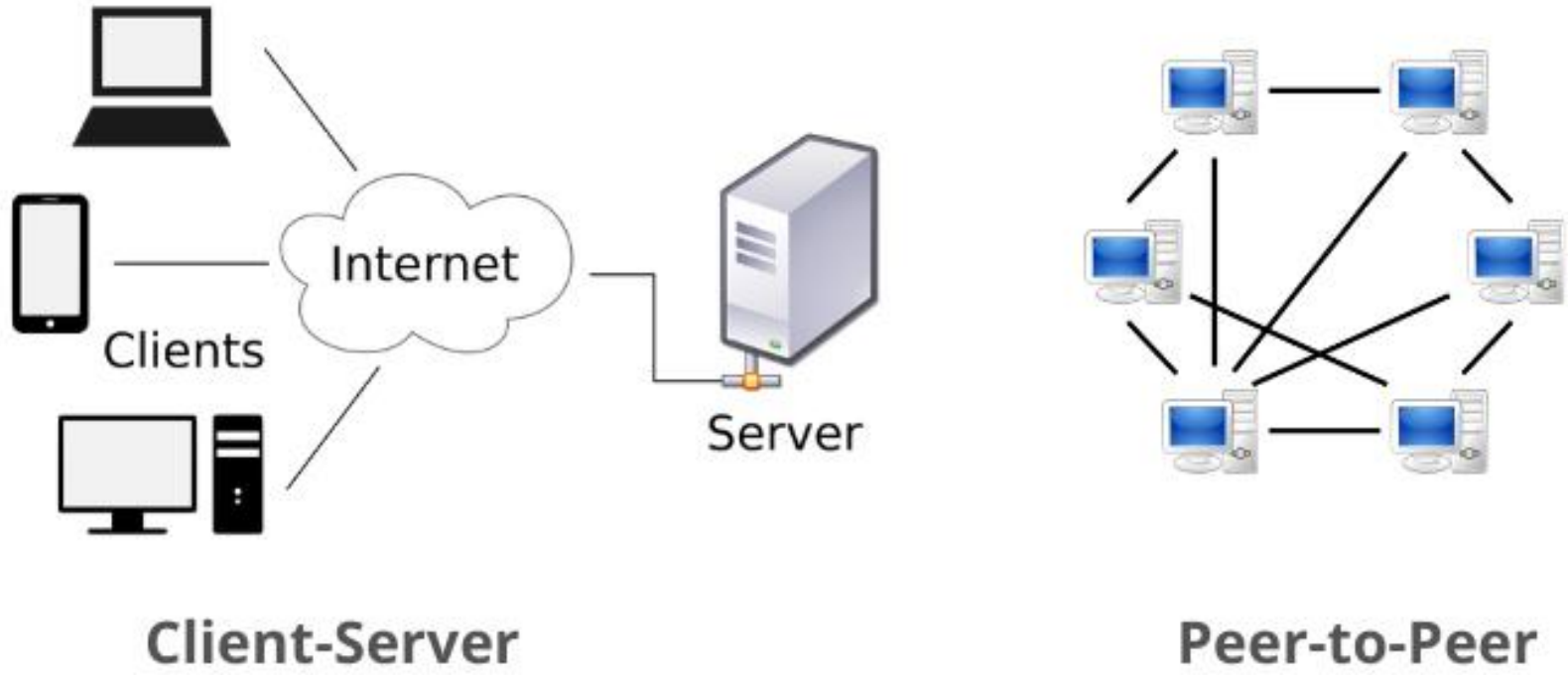
- Example



Peer-to-Peer (P2P) Architecture

- Decentralized model where each node (peer) can act as both client and server
- Peers directly share resources without a central coordination point
- Examples:
  - BitTorrent, blockchain networks, cryptocurrency systems
- Pros:
  - Scalability, resilience, reduced infrastructure costs
- Cons:
  - Harder to manage, potential security challenges

Peer-to-Peer vs. Client-Server Architecture



Microservices Architecture

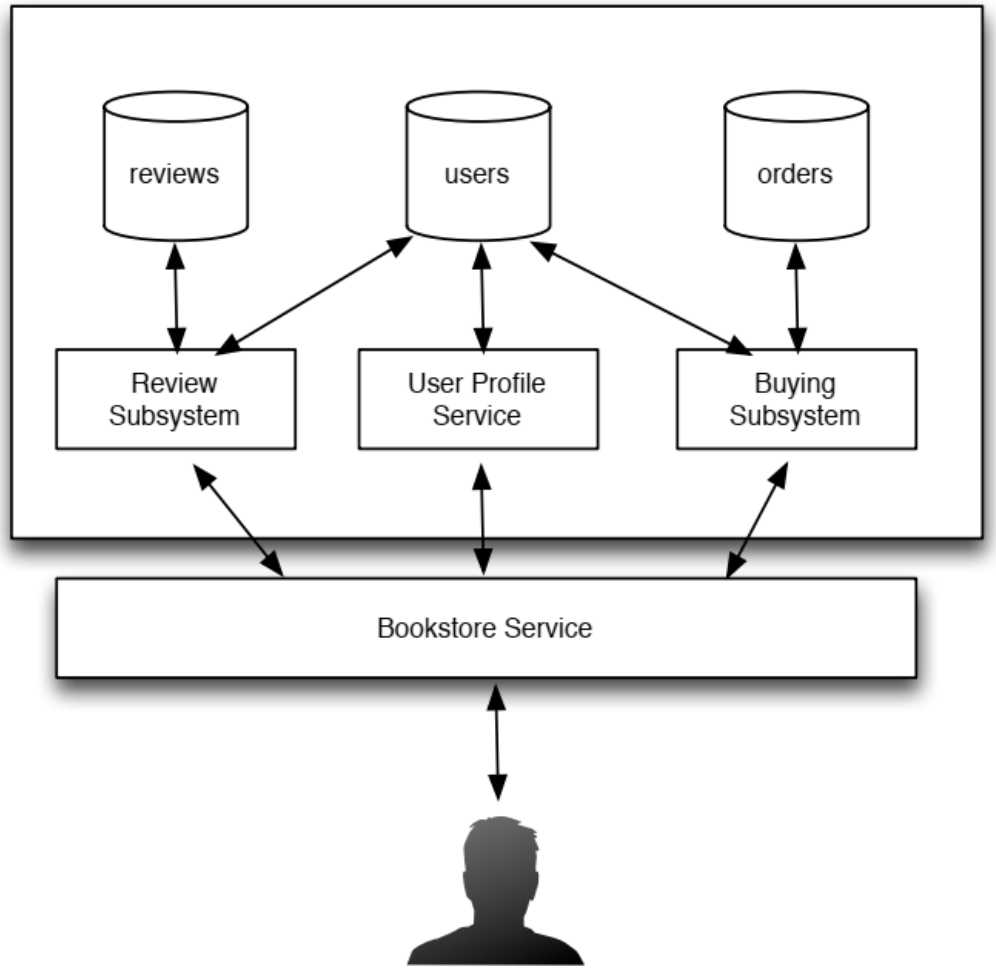
- Application built as a collection of small, independent services
- Each service runs a unique process and can be deployed independently

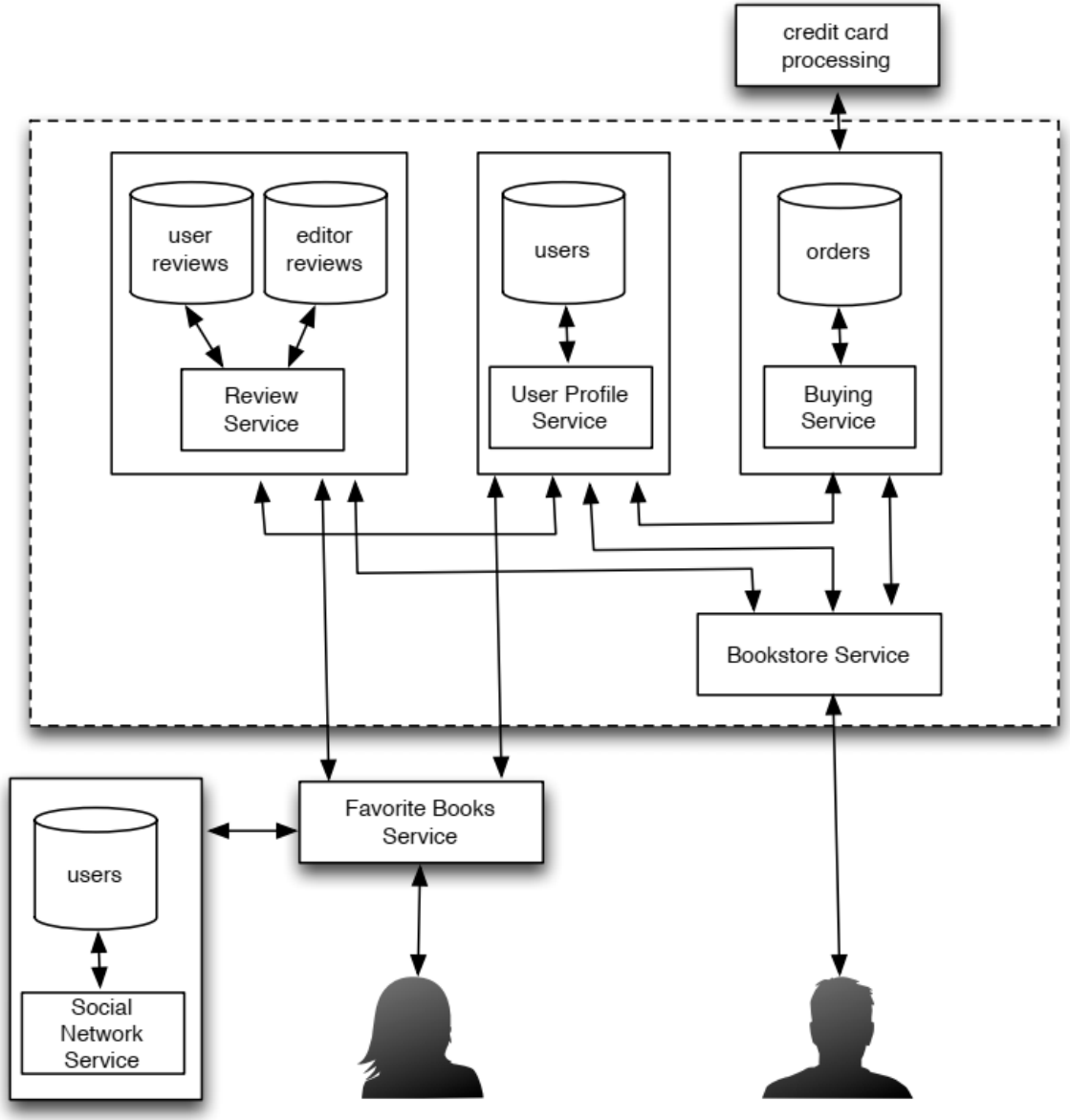
- Examples:
  - Netflix, Amazon, Uber

- Pros:
  - Flexibility, easier scaling, technology diversity

- Cons:
  - Complex inter-service communication, increased operational overhead

Monolithic vs. Microservices Architecture

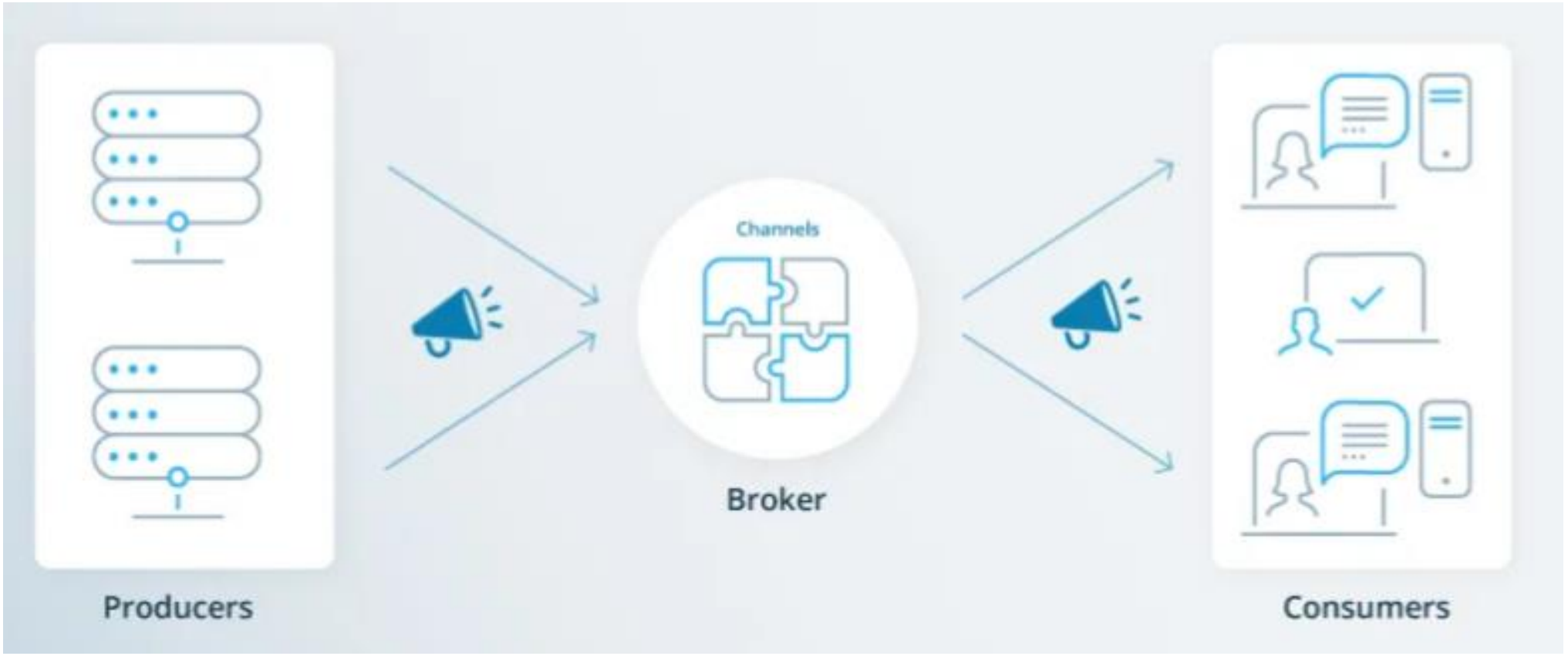




Event-Driven Architecture

- System components communicate through events
- Producers generate events, consumers react to them
- Examples:
  - Real-time analytics, IoT systems
- Pros:
  - Loose coupling, scalability, reactive design
- Cons:
  - Complexity in event tracking, potential performance overhead

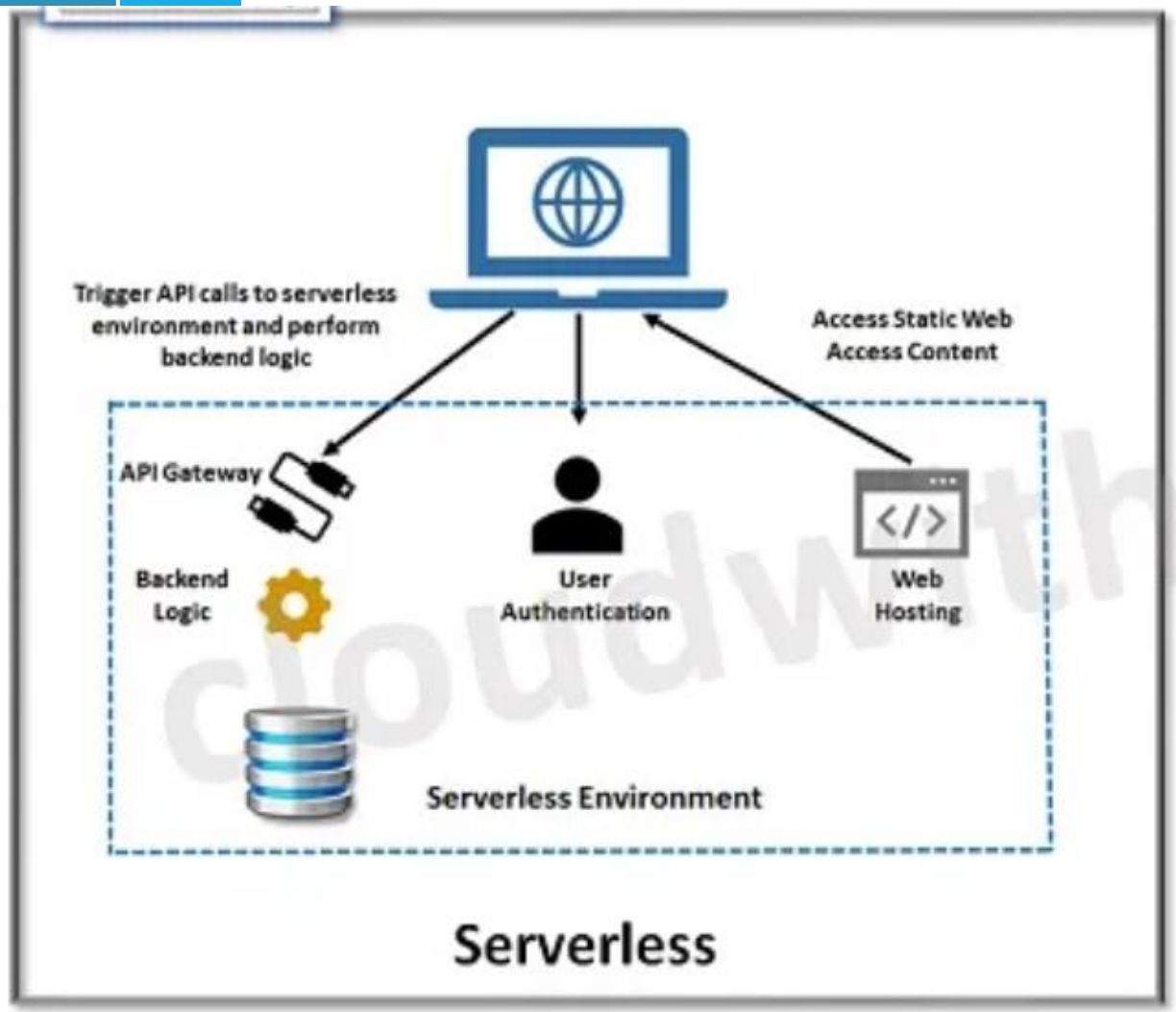
Event-Driven Architecture

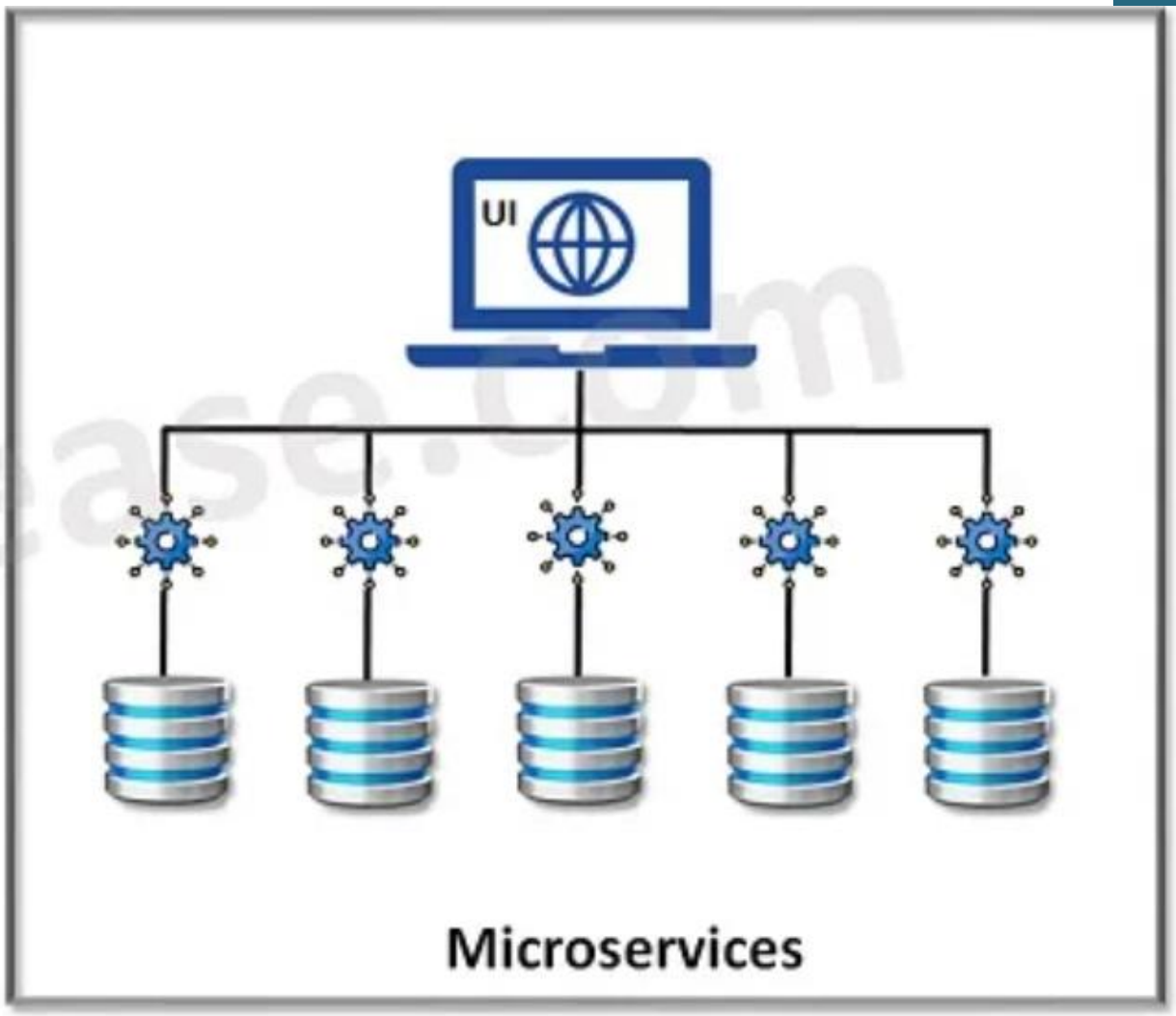


Serverless Architecture

- Cloud-based model where cloud provider manages server infrastructure
- Developers focus on writing code that runs in response to events
- Examples:
  - AWS Lambda, Azure Functions
- Pros:
  - Cost-effective, automatic scaling, reduced operational complexity
- Cons:
  - Potential cold start latency, vendor lock-in

Serverless vs. Microservice Architecture





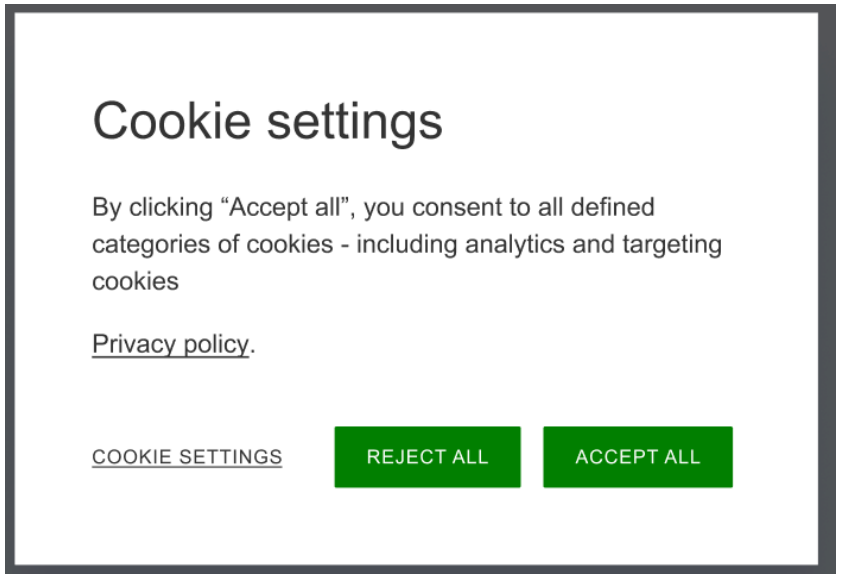
Web's Client-Server Architecture

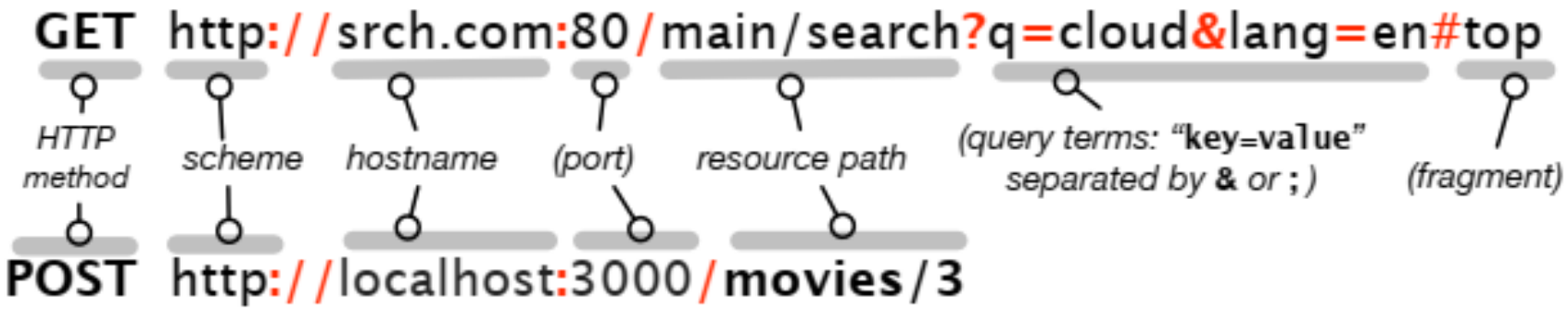
– History

Year	System	Client	Server	Protocol(s)
1960	Sabre, airline reservations system for American Airlines	Custom electromechanical terminals installed at travel agencies	Two IBM 7090 mainframes	Custom *FM*-based protocol over leased telephone lines
1971	*FTP* (File Transfer Protocol), which allowed clients to download files from servers	Originally, command-line client `ftp`; today, command-line clients (cURL, NcFTP, WinSCP), GUI apps (Cyberduck, Fetch), and all Web browsers	Various server software packages, including Unix `ftpd`, FileZilla, `Vsftpd`	ASCII-based FTP protocol over TCP/IP
1983	Novell NetWare, which allowed PCs running the *CP/M* or *MS-DOS* operating systems to share files on a server	Custom client software compatible with MS-DOS	Custom Novell file server appliance based on Motorola 68000 microprocessor	Custom protocols over custom PC-compatible network interface
1984	POP (Post Office Protocol), which allowed separation of email clients from servers	Various PC apps, including Eudora, Thunderbird, Apple Mail, Microsoft Outlook, Elm, Pine, Eureka	Various server software packages, including Apache James, Nginx, Eudora, Qpopper	ASCII-based POP protocol over TCP/IP; largely superseded by IMAP
1990	World Wide Web	Various PC apps, including NCSA Mosaic, Netscape Navigator, Microsoft Internet Explorer, Mozilla Firefox, Google Chrome	Various server software packages, including Apache Httpd, Microsoft Internet Information Server, Nginx	ASCII-based HyperText Transfer Protocol (HTTP) over TCP/IP

Web's Client-Server Architecture

- Basic concepts
  - IP address
  - Port Number
  - DNS
  - HTTP protocol
    - Stateless protocol
      - Cookies!
  - URL





Self-Checks

Self-Check 3.2.1. *Is DNS a client–server protocol? Why or why not?*

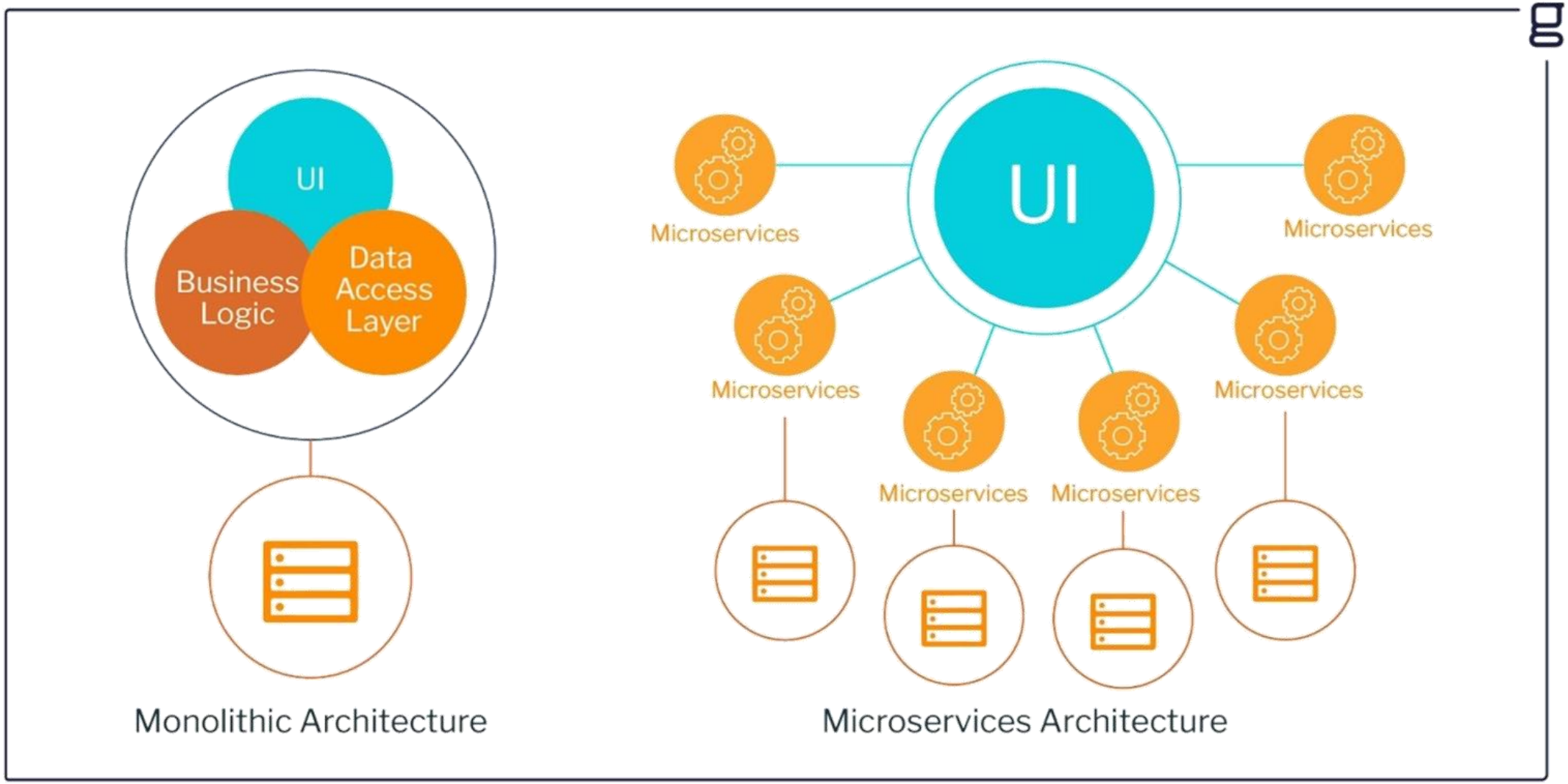
Self-Check 3.2.2. *Can you make a TCP connection without specifying a port number, and if so, what happens?*

■

Self-Check 3.2.3. *True or false: HTTP as a protocol has no concept of a "session" consisting of a sequence of related HTTP requests to the same site.*

Self-Check 3.2.4. *Many HTTP servers rely on using HTTP cookies to identify a client on repeated requests to the same site, for example, to track information such as whether that user has logged in. What happens if you complete disable cookies in your browser and try to visit such a site?*

From Websites to Microservices



Web history

1. 1990
   - Just display static content
2. 1995
   - Creating HTML pages “on the fly”
3. 2005
   - Making web apps similar to Desktop apps
   - making HTTP requests to the server without causing a page reload
   - AJAX
     - Asynchronous JavaScript And XML
     - Data Format: XML and JSON
4. Moving from Monolithic App to a set of independent services that could be composed to produce larger sites
   - Service Oriented Architecture (SOA)

SOA

› Start form Amazon

Yegge claims that Jeff Bezos broadcast an email to all employees along the following lines (we are paraphrasing the main points of Yegge's description for conciseness):

> All teams responsible for different subsystems of Amazon.com will henceforth expose their subsystem's data and functionality through service interfaces only. No subsystem is to be allowed direct access to the data "owned" by another subsystem; the only access will be through an interface that exposes specific operations on the data. Furthermore, every such interface must be designed so that someday it can be exposed to outside developers, not just used within Amazon.com itself.

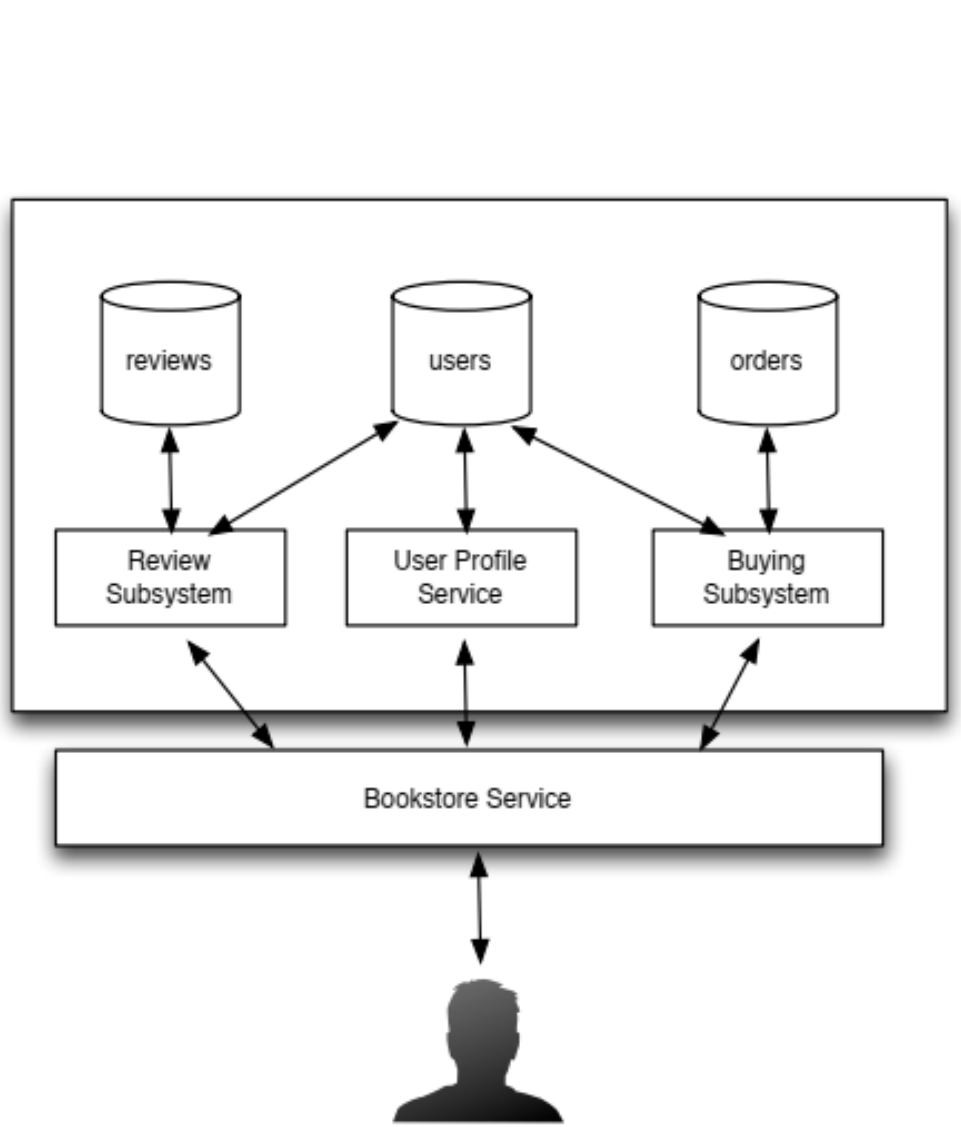
Microservice

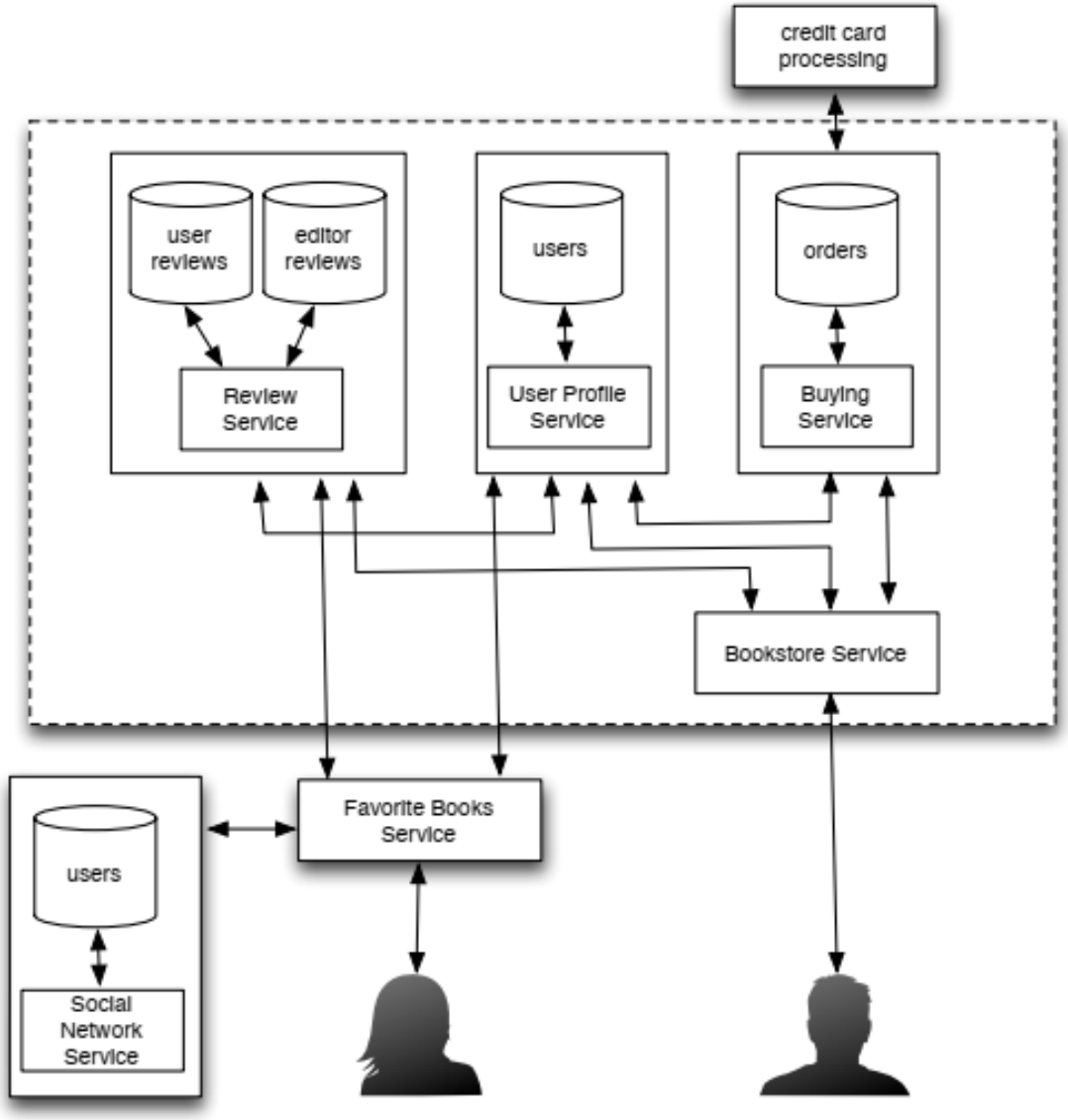
- an architectural style for developing software applications where a large application is decomposed into smaller, independently deployable services.
  - Each microservice is:
    - Focused on doing one specific business capability or function extremely well
    - Loosely coupled, meaning it can be developed, deployed, and scaled independently of other services
    - Typically communicating with other services through lightweight protocols like HTTP/REST or message queues
    - Capable of being developed, deployed, and maintained by small, autonomous teams
    - Often containerized (using technologies like Docker) and orchestrated using platforms like Kubernetes

Microservice

- Key characteristics include:
  - Small, modular design
  - Technology agnostic (can be written in different programming languages)
  - Own its data storage and can have its own database
  - Supports horizontal scaling and rapid iteration
  - Resilient, with the ability to handle service failures without bringing down the entire system

Example of SOA





SOA Pros and Cons

Pro	Con
Reusability: Others can recombine existing services with others to create new apps, as in Figure 3.3, and each microservice can be implemented using the most appropriate language or framework, since its implementation is completely hidden behind its API.	
Easier testing: a microservice does only one thing, so testing each microservice is easier.	
More Agile-friendly: Chapter 1 reveals that Agile works best with small-to-medium projects and teams. SOA allows large services to be created by composing smaller ones, each of which can be built and operated by a small Agile team.	
"You build it, you run it" (as Amazon Web Services CTO Werner Vogels has said): the same tightly-knit team is responsible for developing, testing, and operating the microservice, allowing the microservice to be improved more quickly in response to customer re-	

Self Check

Self-Check 3.4.1. *Another take on SOA is that it is just a common sense approach to improving programmer productivity. Which productivity mechanism does SOA best exemplify: Clarity via conciseness, Synthesis, Reuse, or Automation and Tools?*

RESTful APIs: Everything is a Resource

- each service provides a well-defined set of operations on one or a few related types of resources
- clients need a way to name the server function to be called, pass arguments to it, consume return values, detect and handle server exceptions, …
  - All subject to the constraints of using HTTP for communication
- API (Application Programming Interface)
  - "contract" between a caller and callee,
    - whether these are a program calling a library function
      - or
    - a SaaS client invoking a service on a SaaS server,

API in applications vs. SaaS

	Python program (caller) calls a method in a Python package or library (callee)	**SaaS client (caller) invokes SaaS service (callee)**
1. How does the caller identify the callee?	Same computer and same process as caller; `import` makes a particular named library available to the caller, as in `import numpy`	
2. Which operation is called?	Method named in code, e.g. `numpy.array(...)`	
3. How does the caller pass required and optional parameters to the callee?	Passed as arguments to method call, e.g. `numpy.array([1,2,3])`	
4. How does the caller receive a return value?	Returned from function call and usually assigned to a variable, e.g. `n=numpy.array([1,2,3])`	
5. How does the callee signal an error?	Callee may return a "sentinel" error value (such as `None` in Python), or raise an exception (`try/except`)	

API for SaaS

- Challenge
  - HTTP does not prescribe a way to “name a remote function” or “pass parameters” since those tasks were never part of its original design.
  - HTTP and URI specifications offer no conventions regarding the semantics (implied meaning)
    - of how URIs are constructed or how these tasks should occur.
  - Conventions are articulated by REST
    - **RE**presentational **S**tate **T**ransfer
    - In 2000
    - Roy Fielding
      - Proposed REST in his Ph.D. dissertation as a way of mapping requests to actions that is particularly well suited to a service-oriented architecture.
      - REST is not a standard, but a design stance

REST

- It's a set of constraints and principles
  - that define how resources are identified, represented, and transferred between clients and servers **over HTTP**
  - **Key Principles of REST:**
    - Everything in REST is considered a resource
    - Each resource is uniquely identifiable via a URL (Uniform Resource Locator)
    - Resources can represent objects, data, or services
    - Resources can have multiple representations (JSON, XML, HTML)
  - **REST uses standard HTTP methods with specific semantic meanings:**
    - GET: Retrieve a resource
    - POST: Create a new resource
    - PUT: Update an existing resource (full update)
    - PATCH: Partially update a resource
    - DELETE: Remove a resource

RESTful API

– For any RESTful API operation,
it should be straightforward to answer the following questions:

1. What is the primary resource affected by the operation?
2. What is the operation to be done on that resource? What are the possible results? What are the possible side effects, if any?
3. What other data is necessary to complete the operation, if any, and how is it specified?

RESTful API

- Uniform Interface
  - Standardized way of communicating between components
  - Uses a consistent set of well-defined interaction rules
  - Simplifies and decouples the architecture

Example REST API Endpoint:

```
GET /users/123
POST /users
PUT /users/123
DELETE /users/123
```

Typical REST Response (JSON):

json

```
{
  "id": 123,
  "name": "John Doe",
  "email": "john@example.com",
  "status": "active"
}
```

RESTful API

- Practical Considerations:
  - Supports caching to improve performance
  - Uses standard HTTP status codes (200 OK, 404 Not Found, etc.)
  - Typically uses JSON as the primary data exchange format
  - Facilitates microservices and distributed system design
- Advantages:
  - Simple and lightweight
  - Scalable
  - Platform and language independent
  - Easy to understand and implement
  - Works well with HTTP infrastructure
- Limitations:
  - Can be chatty with complex data requirements
  - Overhead in transferring full resource representations
  - Requires careful design for complex interactions

Compare RESTful with Non-RESTful

	Non-RESTful site URI	RESTful site URI
Login to site	`POST /login/dave`	`POST /login/dave`
Welcome page	`GET /welcome`	`GET /user/301/welcome`
Add item ID 427 to cart	`POST /add/427`	`POST /user/301/add/427`
View cart	`GET / cart`	`GET /user/301/cart`
Checkout	`POST /checkout`	`POST /user/301/checkout`

????

Self-Check 3.5.1. *Which of these routes for updating the information of movie ID 35 follow good HTTP and REST practices: (a) `POST /movie/35`, (b) `POST /movies/35`, (c) `PUT /movies/35`, (d) `PUT /movies/35`, (e) `GET /movies/35`, (f) `GET /movies/35`,*

API calls and JSON

- There are three ways to pass parameters from an HTTP client to a
- service:
  1. in the URI,
  2. in the request body (for POST or PUT requests),
  3. and rarely, as the value of an HTTP header

- When the number of parameters is small,
- and when the parameters are simple types such as strings or numbers, they can often be passed as parameters embedded in the URI

```
GET /search/movies?query=Batman+Returns
```

- When the data to be passed is more complex, or when the API operation involves a state changing HTTP method such as POST or PUT,
  - the data is sent as part of the request body, as browsers do when submitting the values entered on a fill-in form

API calls and JSON

- How is this data presented to the server?
  - *JSON*
    - JavaScript Object Notation
    - common interchange format
    - its syntax is similar to JavaScript
    - a set of unordered key/value pairs

```
Request
GET /api/v1/resource HTTP/1.1
Host: example.com
Authorization: Bearer YOUR_API_KEY
```

  - Passing parameters in HTTP
    3. Via HTTP Header
       - pass very specialized types of parameters
         - Such as Passing an API Key in the Header
         - When making a request to an API, you often need to include an API key for authentication.
         - This is typically done using the Authorization header.

Question?

Bioinformation.ir

info@Bioinformation.ir