28 MARCH 2022

Introduction to system/backend programming

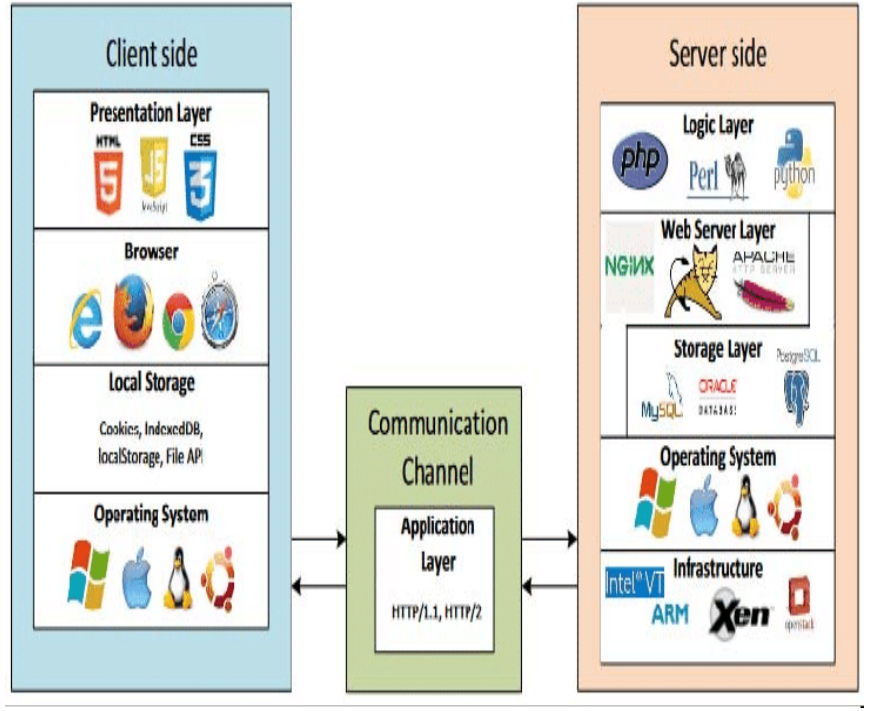
Agenda

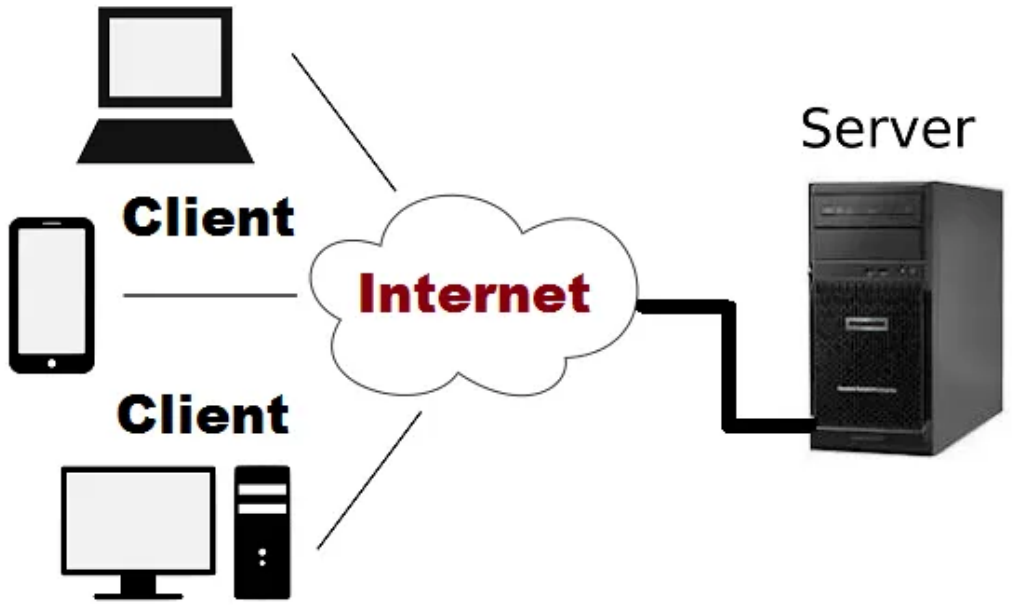
- Introduction to backend/frontend/system programming.
- Modern application architecture – client-server, microservices, message passing
- Introduction to the concept of an API
- Examples of real-world architecture using APIs/microservices/client-server architectures
- Introduction to gRPC and Rest
- Introduction to networking programming
- Introduction to systems programming

Backend/Frontend/System programming - WTF?

- Due to increased complexity we need to fine-grained separation of responsibilities
- Mostly used in the context of web development (but not only)
- Frontend
  - Deals directly with the user visible experience – text style, layout, images, graphs, tables, buttons. Generally the user visible structure of the site aka GUI.
  - Main languages – HTML/CSS/Javascript
  - Main frameworks – AngularJS/React.js/jQuery/SASS and many more
- Backend / server side
  - Deal with everything else – data processing, storage, database.
  - Users don’t know it’s there (they don’t have to)
  - Almost any language can be used – PHP/Java/Python/JavaScript(Node.js)/C#/ Go…
- System programming
  - Low-level programming – device drivers, operating systems, antivirus software, all kinds of daemons, system libraries

Client-server architecture





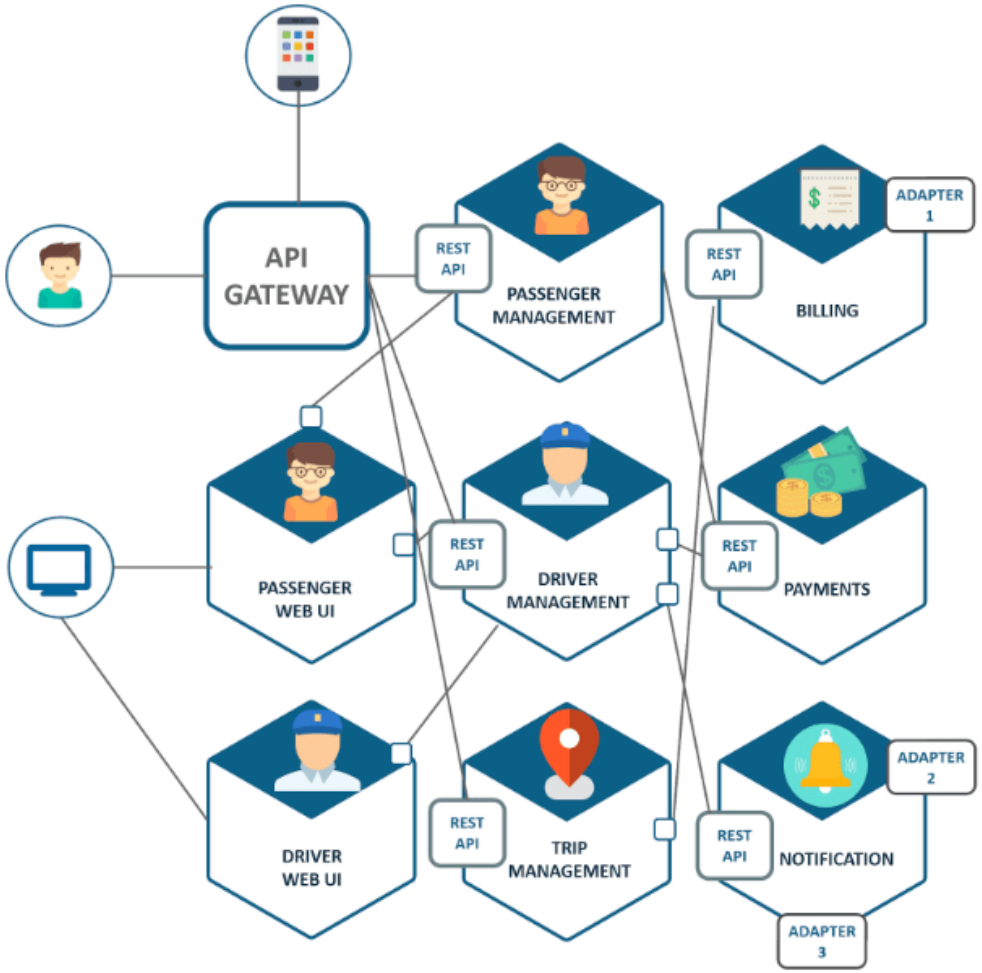
Microservices

- You got multiple **loosely coupled**, **independently deployable, highly maintainable** services
- Each service ***should*** provide a well-defined **API** that other can consume
- It provides encapsulation (it’s a black box for its users).
- Multiple services work together to bring the full application experience (i.e twitch.tv has at least 100s of microservices*)

Microservices

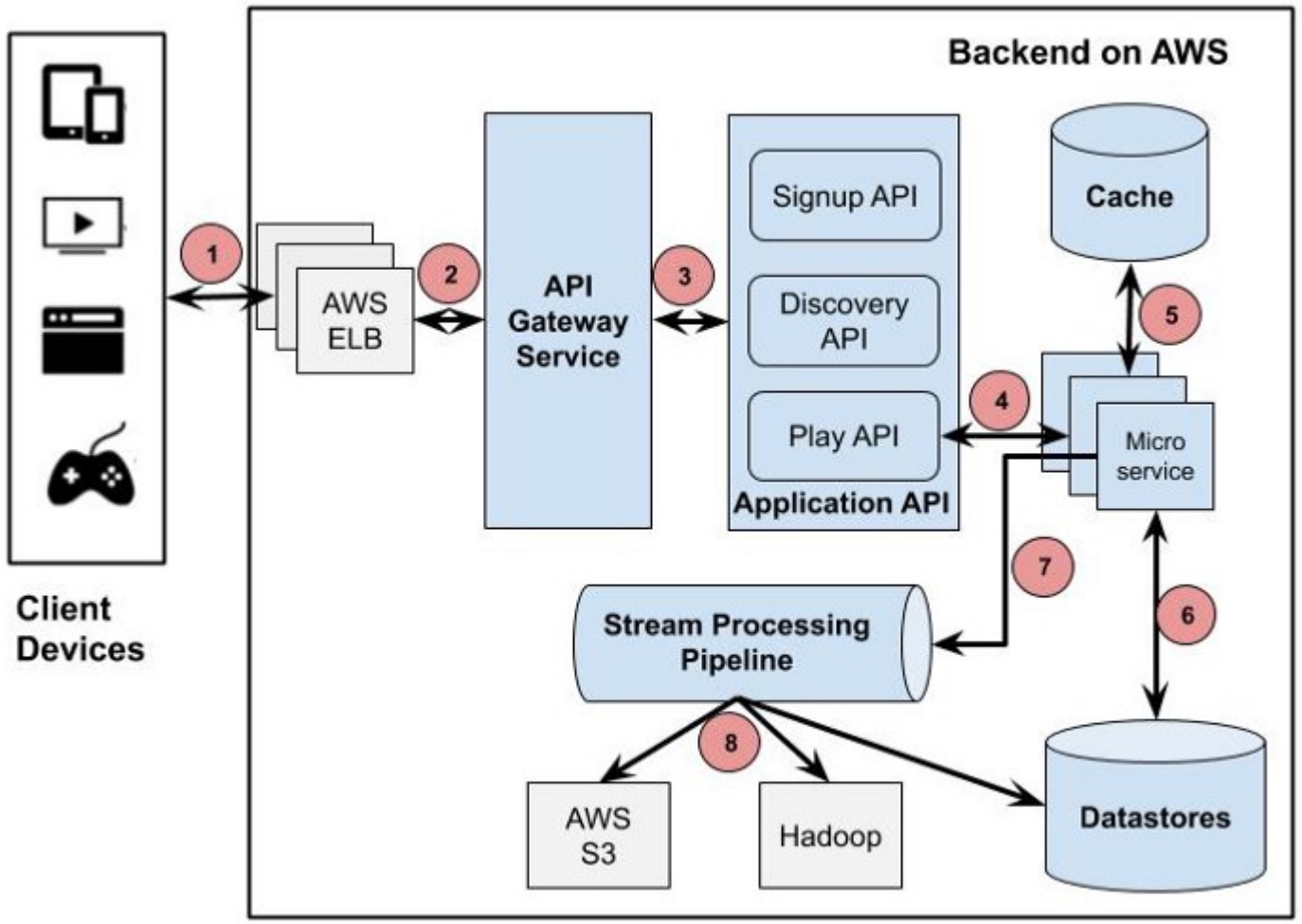
- You got multiple **loosely coupled**, **independently deployable, highly maintainable** services
- Each service ***should*** provide a well-defined **API** that other can consume
- It provides encapsulation (it’s a black box for its users).
- Multiple services work together to bring the full application experience (i.e twitch.tv has at least 100s of microservices*)
- Not the silver bullet
  - Care should be taken to tame complexity
  - Multiple services == **distributed system** (all pitfalls and risks of distribute systems apply)
  - Testing becomes more complex due to outside factors I.e network and communication failures
  - You suddenly have to start thinking about network latency, load balancing
  - Doing full system monitoring more complex
  - Testing **might** be more complex
  - Added deployment complexity

Uber architecture



https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a

Netflix architecture



So how is this achieved?

- Through a lot of communication
  - In order to establish communication you need 2 main things:
    - A shared understanding of the ‘language’ communicating parties will use (API)
    - Transferring data – aka sound waves (in case of verbal communication) or messages (very broadly defined, in case of written communication)

So how is this achieved?

- Through a lot of communication
  - In order to establish communication you need 2 main things:
    - A shared understanding of the ‘language’ communicating parties will use (API)
    - Transferring data – aka sound waves (in case of verbal communication) or messages (very broadly defined, in case of written communication)
  - **A**pplication **P**rogramming **I**nterface
    - Application – used for communication between programs (as opposed to computer<>humans or User Interface)

So how is this achieved?

- Through a lot of communication
  - In order to establish communication you need 2 main things:
    - A shared understanding of the ‘language’ communicating parties will use (API)
    - Transferring data – aka sound waves (in case of verbal communication) or messages (very broadly defined, in case of written communication)
  - **A**pplication **P**rogramming **I**nterface
    - Application – used for communication between programs (as opposed to computer<>humans or User Interface)
    - Programming – the interface is used for programmatic access to the software

So how is this achieved?

- Through a lot of communication
  - In order to establish communication you need 2 main things:
    - A shared understanding of the ‘language’ communicating parties will use (API)
    - Transferring data – aka sound waves (in case of verbal communication) or messages (very broadly defined, in case of written communication)
  - **A**pplication **P**rogramming **I**nterface
    - Application – used for communication between programs (as opposed to computer<>humans or User Interface)
    - Programming – the interface is used for programmatic access to the software
    - Interface – the point where 2 things communicated

API

- Some examples of an api
  - “The Instagram Graph API allows Instagram Professionals — Businesses and Creators — to use your **app** to manage their presence on Instagram. The **API can be used to get and publish their media, manage and reply to comments on their media**….”

```
# curl -i -X GET "https://graph.facebook.com/v13.0/me/accounts?access_token={access-token}"
response:
{
  "data": [
    {
      "access_token": "EAAJjmJ...",
      "category": "App Page",
      "category_list": [
        {
          "id": "2301",
          "name": "App Page"
        }
      ],
      "name": "Metricsaurus",
      "id": "134895793791914",  // capture the Page ID
      "tasks": [
        "ANALYZE",
        "ADVERTISE",
        "MODERATE",
        "CREATE_CONTENT",
        "MANAGE"
      ]
    }
  ]
}
```

API

- Olx partner API
- “OLX Europe shares public API for its partners. It allows to easily integrate with local sites by posting, managing adverts and communicating with OLX users via internal messaging system.”
  - https://developer.olx.bg/api/doc

```
curl -d @req.txt -H "Content-Type: application/json" -X POST https://www.olx.bg/api/partner/adverts
{
        "title": "string",
        "description": "string",
        "category_id": 0,
        "advertiser_type": "private",
        "external_url": "string",
        "external_id": "string",
        "contact": {
                "name": "string",
                "phone": "string"
        },
        "location": {
                "city_id": 0,
                "district_id": 0,
                "latitude": 0,
                "longitude": 0
        },
}
```

REST

- Previously looked at APIs are an example of RESTful apis.
- **R**epresentation **S**tate **T**ransfer (coined by Roy Fielding's PHD thesis in 2000)
- Rest guidelines (constraints) to achieve scalability/flexibility
  - Client-Server separation
  - Uniform Interface (Http, POST/GET/PUT/DELETE = CRUD)
  - Stateless
  - Layered System
  - Cacheable

Conclusion

- The problems we are solving are more complex → our solutions are more complex
  - Architectures and API guidelines are codified good practices how to solve **some** of the problems
  - REST is one of the well known API building guideline (SOAP/GraphQL/RPC)
  - Everything revolves around client-server models of communication
  - Microservices is an example of an architectural pattern (serverless/FaaS, monolithic or a frankenmix)

Questions so far?