이번 파트에서 할 이야기

- API 제공 방식
  - 디자인 방법론, 직렬화
  - 정답은 어느정도 정해져 있지만 대안과 옛날 이야기
- 기술스택
  - 언어, 프레임워크
  - 서버 인프라를 어디에 준비할지
  - 컴퓨팅 엔진
  - DB, DB 서버 어디 띄울지

API 제공 방식

API 디자인 방법론

- SOAP
- REST
- GraphQL

API 제공 방식

API 디자인 방법론 〉 SOAP

- SOAP : Simple Object Access Protocol
- 목적 : SOA (Service Oriented Architecture, 서비스 지향 아키텍처)
- 결론적으로는 XML 데이터를 주고받음
- HTTP 위에서 돌아가는 또 하나의 Protocol 개념
  - 보안 수준이 엄격하고 트랜잭션 등이 고려되어 있어 오버헤드가 큼
  - 구현 복잡도가 높고 느림
- 웹 서비스보다는 기업용 어플리케이션 등을 작업하는 데에 더 이상적
- 클라이언트 개발자들은 보통 익숙하지 않음

API 제공 방식

API 디자인 방법론 〉 REST

- REST : Representational State Transfer
- 목적 : ROA (Resource Oriented Architecture, 자원 지향 아키텍처)
- SOAP은 프로토콜, REST는 HTTP API 디자인 가이드라인 개념
  - REST API를 개발해야겠다 → 지식 빼고 준비할 것이 딱히 없음
- 대부분의 웹 서비스 개발 사례 및 API (Google Maps 등)에서 REST를 사용

API 제공 방식

API 디자인 방법론 〉 GraphQL

- DB에 SQL을 사용하듯, 웹 클라이언트가 서버에 GraphQL로 데이터를 쿼리
- 굳이 HTTP Endpoint를 뚫고 query parameter를 대응시킬 것 없이 GraphQL 인터페이스만 제공
- 단일 Endpoint만 열어두고 GraphQL 내용에 따라 데이터를 다르게 제공

```
{
  human(id: "1000") {
    name
    height
  }
}
```

API 제공 방식

API 디자인 방법론 〉 GraphQL

- 프론트엔드에게 더 할 일이 많아지고 힘이 실리는 느낌
  - 물론 resolver같은 것을 구현해야 하지만 개발 부담이 덜한 것은 맞음
- API 개발에 덜 투자하고 인프라 등 다른 일에 투자할 여유가 생김
- 그러나 실전에서 사용하기 쉽지 않음
  - 프론트엔드의 React+flux/redux 아키텍처는 GraphQL에 대응하기 어려움
  - 여타 다른 클라이언트도 GraphQL에 익숙하지 않은 개발자 + 라이브러리 대응이 미흡
  - 백엔드쪽도 아직 개발 환경 대응이 명쾌하게 되어 있진 않음

API 제공 방식

데이터 직렬화 포맷

- 데이터 직렬화
  - ArrayList, Map 등은 Java만 알아들을 수 있는 데이터
  - 네트워크를 타고 갈 수 있게 하려면 다른 런타임에서도 알아들을 수 있는 형태로 만들어줘야 함

API 제공 방식

데이터 직렬화 포맷

- XML
- YAML
- JSON
- HTTP Content-Type에 지원되는 것이라면 무엇이든

API 제공 방식

데이터 직렬화 포맷 〉 예제용 ArrayList

```
public final class User{
        private final int id;
        private final String nickname;
        private final String githubUrl;

        // ... 대충 생성자 ...
}

User planb = new User(1, "planb", "https://github.com/planb");
User spencer = new User(2, "spencer", "https://github.com/spencer");

ArrayList<User> users = new ArrayList<>(Arrays.asList(planb, spencer));
```

API 제공 방식

데이터 직렬화 포맷 〉 XML

```
<users>
        <user>
                <id>1</id>
                <nickname>planb</nickname>
                <github-url>https://github.com/planb</github-url>
        </user>
        <user>
                <id>2</id>
                <nickname>spencer</nickname>
                <github-url>https://github.com/spencer</github-url>
        </user>
</users>
```

- number/string 구분이 없음
- 실질적인 컨텐츠에 비해 데이터 양이 큼
- JSON을 쓰면 camelCase를 그대로 사용할 수 있으나 XML은 snake case가 표준

API 제공 방식

데이터 직렬화 포맷 〉 YAML

```yaml
users:
        - id: 1
          nickname: planb
          githubUrl: https://github.com/planb
        - id: 2
          nickname: spencer
          githubUrl: https://github.com/spencer
```

- 줄여서 표현할 수 없음 : 새로운 key-value 쌍은 무조건 new line과 indent가 필요
  - comma만 있으면 새로운 kv pair를 더 명시할 수 있는 JSON에 비해 용량을 많이 소비
- 보통 config에서 자주 사용

API 제공 방식

데이터 직렬화 포맷 〉 JSON

```
{
        "users": [
                {
                        "id": 1,
                        "nickname": "planb",
                        "githubUrl": "https://github.com/planb"
                }
                {
                        "id": 2,
                        "nickname": "spencer",
                        "githubUrl": "https://github.com/spencer"
                }
        ]
}
```

- JavaScript Object Notation

API 제공 방식

데이터 직렬화 포맷 〉 정답은 JSON

- 일반적인 웹서비스에서 JSON을 많이들 사용하고 있기 때문임
  - 다른 사람들이 쓰기 때문에 따라서 쓰는 것 : X
  - 익숙함 덕분에 협업이 편해서 (클라이언트 개발자는 당연히 JSON을 생각하고 있을 것)
  - JSON이 시장을 지배하고 있으므로 생태계 지원이 잘 되어 있어서
    - 보통 http request 라이브러리에 JSON 파서가 빌트인으로 껴 있음
    - JSON response를 바로 VO로 변환해주는 경우도 꽤 있음
- XML, YAML보다 덜 장황하고 읽기 쉽다 – XML, YAML만큼 기능이 많지 않기 때문
- 축약에 대해 유연하다 : ndjson

```
{"some":"thing"}
{"foo":17,"bar":false,"quux":true}
{"may":{"include":"nested","objects":["and","arrays"]}}
```

API 제공 방식

데이터 직렬화 포맷 〉 Insight : Protobuf (Protocol Buffers)

- 참고자료 : 타다 시스템 아키텍처 , Online Protobuf encoder/decoder
- human readability를 포기하고 속도를 챙긴 포맷
- 어떤 느낌?

i. protobuf compiler를 설치

ii. model이 필요할 때마다 .proto 파일을 만든다. (지면 상 뭉개놓았으니 양해 바람)

```
message PostCreateRequest { required string content = 1; }

message PostListResponse {
    message Post { required int32 id = 1; required string content = 2; }
    repeated Post posts = 1;
}
```

iii. protobuf compiler로 컴파일

iv. 그럼 언어에 맞게 파일이 하나 생긴다. 여기엔 `.proto` 를 기준으로 만들어진 클래스 등이 들어감.

v. 클라이언트/서버 모두 이를 통해 request/response를 protobuf message로 처리

API 제공 방식

데이터 직렬화 포맷 〉 Insight : Protobuf (Protocol Buffers)

- 주장하는 upside
  - `.proto` 를 통해 API 문서화, DTO 자동 생성
  - API Spec이 변경된다면 `.proto` 파일 내용도 변경될 것이므로 스펙 문서와 실제 동작이 mismatch될 일이 없다
  - 속도가 빠르다
    - How are protocol-buffers faster than XML and JSON?
- 직접 써보고 느낀 downside
  - 바이너리 데이터라서 사람이 읽으려면 decoding을 거쳐야 함
    - 인코딩되면 `08 96 01` 처럼 되어있음
  - protobuf는 validation하라고 만들어진 라이브러리가 아님
    - min, max 등을 명시할 수 없으므로 결국 추가적인 validation 작업이 필요
    - 이러한 validation rule은 proto 파일과 별개의 것이 되므로 mismatch 문제가 생길 가능성 있음
  - 웹 서비스 개발에 주류로 쓰이는 기술이 아님
- 웹 서비스와 같은 client/server 시스템보다는 gRPC 시스템에 어울림

API 제공 방식

데이터 직렬화 포맷 〉 Insight : server-side rendering

- server-side rendering (SSR)
  - 웹페이지 전체를 서버에서 만들어서 뿌려주는 방식
  - placeholder가 뚫린 HTML을 서버가 가지고 있다가, 요청 들어오면 query 결과를 채워서 내려주는 식
  - 웹페이지 전체를 서버가 준비하기 때문에 server-side rendering
- client-side rendering (CSR)
  - placeholder가 뚫린 HTML을 먼저 받아서 웹페이지를 시작
  - 데이터는 별도로 API 요청을 통해 채움
  - 서버는 데이터만 제공하고 웹페이지를 만드는 것은 브라우저에서 동작하는 JavaScript가 하므로 client-side rendering
- response header의 Content-Type으로 생각하면 편함
  - SSR은 `text/html` , CSR은 `application/json` 위주

API 제공 방식

데이터 직렬화 포맷 〉 Insight : server-side rendering

- 요즘? → client-side rendering
  - 페이지의 일부를 변화시켜줄 일이 많은 현대의 웹서비스는 CSR이 매우 유리
  - CSR을 위해 만든 API는 모바일 어플리케이션 개발 시 재사용 가능

기술스택

언어와 프레임워크

- 정해진 답 : Java + Spring
- 기술에 silver bullet은 없다
    - Spring을 중심으로 한 생태계는 매우 우수하지만 게임 서버를 개발하기에는 부적합하듯

기술스택

언어와 프레임워크

- 지금과 같은 과제를 해결하는 데에 일반적으로 사용되는 기술인지
  - 요즘 어떤지 생각해 봐야 함 (PHP도 2015년에는 일반적인 기술이었음)
  - → 인력 pool에도 영향을 줌
- 팀에 익숙한지
  - 일정에 맞춰 생산성을 보장할 수 있는지
  - 익숙하지 않다면 러닝커브를 감당할 수 있는지
- 플랜비의 사례
  - Java + vert.x 조합으로 프로젝트를 진행했으나 vert.x를 배우는 사람이 없어 물려줄 수가 없었음
  - 이후 Python + Flask로 다시 개발

기술스택

인프라 〉 서버 하드웨어

- on-premise
  - PC, 서버용 워크스테이션 등
  - OP.GG 오픈부터의 1년을 되돌아보며
- **클라우드**
  - 비용 절감
    - 여러 클라우드 서비스에서 free-tier 지원 (초기 투자 비용이 0원)
    - pay as you go (리소스를 무한정 늘릴 수 있지만 쓴만큼 과금)
  - 관리 용이성
    - 서버룸 관리는 생각보다 매우x1000 어려움
      - OSI 7계층 각각에 해당하는 하드웨어를 서로 연결시키고 뭐하고..
    - 몇 분만에 서버를 띄우고, 지우고, 트래픽 분산시키고, 도메인 달고, 스토리지 백업 등
    - 서버룸을 운영하는 것보다 훨씬 쉽고 빠르고 정확

기술스택

인프라 〉 vendor

- AWS (Amazon Web Service)
- GCP (Google Cloud Platform)
- MicroSoft Azure
- Naver Cloud

기술스택

인프라 〉 vendor

- 글로벌은 AWS 33%, Azure 18%, GCP 7% 정도
  - Market share(클라우드 시장에서의 매출) 개념이라 이 셋의 사용 사례는 이보다 더 많을 것
- 한국의 경우
  - 체감 상 AWS 65%, GCP 20%, Azure 10%
  - 점유율이 AWS → GCP로 조금씩 이동 중
  - 키워드 : `cloud service market share`
- 이동 중인 이유? → 쓰기 편하고, AI 개발하기 좋아서
  i. GCP가 AWS보다 더 예쁘고 일관성 있는 UI/UX를 지원
  ii. AI 산업의 최전선에 있는 Google이라 AI 엔진을 쉽고 싸고 빠르게 이용 가능
  iii. AWS와 비슷하거나 더 많은 수준의 free-tier 지원

기술스택

인프라 〉 vendor

- 결정 : AWS
- 아직은 Azure, GCP 경험이 풍부한 인력 〈〈〈 AWS 경험이 풍부한 인력
- 클라우드 경험이 풍부하지 않은 경우 AWS가 공부하기도 더 좋음
    - e.g. AWS로 시작하는 클라우드 ~

기술스택

인프라 〉 compute engine

- EC2 (Elastic Compute Cloud)
- ECS (Elastic Container Service)
- ECS+Fargate
- Beanstalk
- Lightsail
- Lambda

기술스택

인프라 〉 compute engine

- 결정 : Lambda
- 1개월 당 100만 request(정확히는 100만 초의 코드 실행 시간) 무료
  - 하루 약 3.3만 request, 거의 3초당 request가 하나씩 들어와도 비용 지불을 할 것이 없음
  - serverless : 코드 실행이 없다면 과금도 없음
- 관리포인트가 줄어드는 효과
  - raw한 compute engine인 EC2의 경우 사실상 컴퓨터를 빌리는 개념이라 직접 관리해줘야 할 것이 많음
    - EC2는 트래픽이 많이 치는 경우 서버를 늘리도록 설정(Auto Scaling)해줘야 하지만 Lambda는 자동 탑재
  - Lambda도 서버 하드웨어는 있지만 매우 추상화되어 있으므로 직접 리눅스 명령어를 칠 일이 없음
- 더 알아보기 : 백엔드가 이 정도는 해줘야 함#컴퓨팅 엔진

기술스택

인프라 〉 DB

- RDB (Relational DataBase)
- NoSQL (Not only SQL)

기술스택

인프라 〉 DB 〉 Main DB에 NoSQL은 제대로 고민하자

- 옛날에 MEAN Stack이라는 것이 있었다..
  - MongoDB, Express.js (서버 프레임워크), AngularJS (프론트엔드 프레임워크), Node.js(JS 런타임)
- 오 힙한데? 싶어서 어떤 프로젝트던 그냥 Node.js, MongoDB 사용
  - 그러나 NoSQL의 가장 큰 특장점인 schemaless를 이용하지 않음
- 대부분의 도메인은 schema가 정해진 데이터를 다루기 때문에 그냥 RDB를 쓰는 것이 맞다
  - 동적인 schema? RDB에 nullable 컬럼 정도 선에서 해결 가능
- MongoDB를 메인 DB로 사용하고 있는 조직 중 일부는 초기에 잘못 의사결정한 것일 수 있다
- NoSQL은 RDB의 한계를 극복하기 위해 사용하는 것

기술스택

인프라 〉 DB 〉 RDB

- MySQL
- PostgreSQL
- Oracle

기술스택

인프라 〉 DB 〉 RDB

- 보통은 MySQL을 선택
  i. MySQL은 이미 globally 많이 쓰이고 있고, 대부분의 워크로드에 어울리고, 다들 익숙해함
  ii. PostgreSQL은 MySQL에 없는 고급 쿼리가 몇몇 지원된다는 점이 advantage
    - 집계함수가 MySQL보다 조금 더 잘 구현되어있고, 지도 관련 postgis 기능 좋다
    - 공간 정보(gis)를 사용하는 것이 아니라면 쓸 이유가 잘 없음
    - 고급 집계 함수는 대용량 데이터에서 이득이 큼 – 이 때가 되면 Druid같은 OLAP로 관리하게 될 것
  iii. Oracle : PostgreSQL과 비슷한 이유
    - MySQL, PostgreSQL이 발전하면서 메리트가 조금은 줄어든 편
    - 엔터프라이즈용 대규모 DBMS? → 카카오뱅크 메인 DB는 MySQL

기술스택

인프라 〉 DB 〉 Insight : SQLite

- DBMS 전체 용량이 600KB밖에 되지 않는 굉장히 가벼운 DB
- DB 서버가 뜨고 여기에 붙어 SQL을 날리는 것이 아니라, 로컬에 있는 `.sqlite` 파일 I/O 방식
- 사용 사례
  - 인터넷 없이 동작하는 시간 관리 앱, 싱글플레이 게임
- Production 수준에는 어울리지 않음
  - 생각보다 나쁘지 않다
    - 그러나 서비스가 성장해 DB Engine을 교체하게 된다면 번거롭고 귀찮음
  - 클라우드 지원이 되지 않음
    - DB 서버를 지원하는 Amazon RDS는 SQLite를 지원하지 않음
- SQLite는 언제 사용하면 좋을까?

기술스택

인프라 〉 DB 〉 Insight : Aurora

- Amazon RDS로 DB를 띄우려다 보면 보이는 Aurora
- AWS에서 만든 DBMS
- MySQL 호환
- 자기들 서버 인프라에 딱 맞게 DB engine을 개발
    - DB Engine들은 당연히 범용적으로 사용할 수 있도록 구현
    - AWS는 자기들이 쓰고 있는 하드웨어에 맞춰 커스텀하면 성능 이득을 더 얻을 수 있음
    - 평균 MySQL보다 20% 비용 절감
    - 실무에서 많이 사용 중

기술스택

인프라 〉 DB 〉 RDB 〉 서버

- 웬만한 클라우드 vendor에서 DB 전용 서버를 띄울 수 있게 준비해둠
  - Amazon RDS
  - Google Cloud Databases
- 백업, 재해 복구, 스토리지 관리 등 DB 인프라에 대한 관리 작업들을 알아서 수행
- 사용할 수 있는 DB가 한정적
  - 쓰고자 하는 RDBMS가 지원되지 않는다면 비주류라는 뜻
    - 확실한 근거가 없다면 쓰지 말자
  - DB 서버를 직접 띄우는 것은 관리하기 매우 어려우므로 많은 지식이 필요

정리

- REST와 JSON이 정답이나, 이게 굳어지게 된 배경을 아는 것이 좋다
- 언어, 프레임워크는 현재 프로덕트에 어울리는 것인지가 먼저
- 웹 서비스를 위해 기본적으로 컴퓨팅, DB 서버가 필요
- 직접 서버 하드웨어를 관리하기보다 클라우드를 추천
- 일반적으로 AWS, GCP도 옛날에 비해 많이들 쓰고 있다
- NoSQL은 RDB의 한계를 극복하기 위해 사용하는 것. 특수 목적의 DB!
- RDB로는 MySQL이 많이들 쓰이며 다양한 클라우드 벤더에서 MySQL 호환 DBMS를 개발함
- EC2같은 곳에 직접 DB를 띄우는 것보다 전용 서비스를 이용하자