이번 파트에서 할 이야기

1. API Specification - 무지성에서 지성까지
2. API 개발에 대한 몇가지 팁

일러두기

- 코드 예제는 Python (간결한 예제를 위해)
- API Specification이라는 게 감을 잡기 전까지는 어려움
  - 글을 읽는 것보다 직접 해보는 것이 더 빠름
  - 멘토님들 있을 때 API Spec 리뷰해달라고 하자!

API

- Application Programming Interface
- 어플리케이션 개발에서 사용하는 모든 '기능'은 API
  - Java의 Stream API
  - Android의 Widgets API
- 함수 ⊂ API
  - Java의 `System.out.println` 도 API라고 할 수 있음

웹 서비스에서의 API

- HTTP Networking을 통해 사용할 수 있는 API (HTTP API)
- 클라이언트와 DB의 중간 layer

```
from requests import get

for x in range(10):
    for y in range(10):
        nickname = f'{x}{y}'

        res = get(f'https://op.gg/ajax/autocomplete.json/keyword={nickname}')

        ...
```

API Design

쉽게 살기

- endpoint 하나만 뚫어놓고 code를 사용해 구분

```
POST /
{ "action": "login", "id": "planb", "password": "gohome" }

POST /
{ "action": "createPost", "title": "제목", "content": "내용" }

POST /
{ "action": "getPosts" }
```

API Design

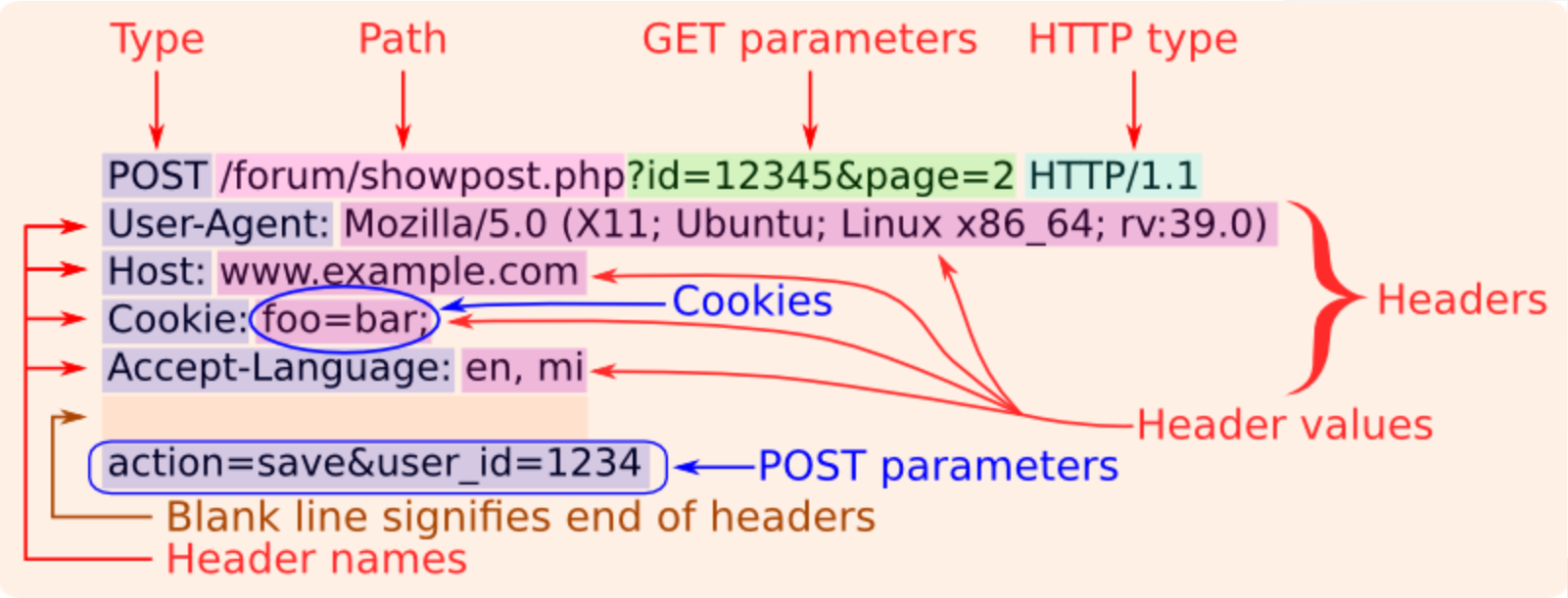
쉽게 살기

- 말이 되냐구요? 플랜비는 옛날에 진짜로 그랬음;;

```
10
11  /**
12   * @author JoMingyu
13   */
14  public class Commands {
15          /*
16           * Notice, Newsletter, Competition, Plan, Meal will parsed in
17           * dms-data-utilities
18           */
19          public static final int REGISTER_STUDENT_ACC = 100;
20          public static final int REGISTER_TEACHER_ACC = 101;
21
22          // notice, newsletter, competition : parsing
23          public static final int UPLOAD_RULE = 111;
24          public static final int UPLOAD_QNA_QUESTION = 112;
25          public static final int UPLOAD_QNA_ANSWER = 113;
26          public static final int UPLOAD_QNA_COMMENT = 114;
27          public static final int UPLOAD_FAQ = 115;
```

API Design

- HTTP message는 여러 area로 구성
- 이들을 잘 이용해서 여러 action들을 human friendly하고 예쁘게 정리하자



API Design

- HTTP Client 라이브러리들도 endpoint가 분리되어 있는 것을 당연하게 생각하고 개발함

```
1   public interface ExampleService {
2
3       @GET("/example/info")
4       Call<String> getExampleBySeq(@Query("seq") String param);
5   }
```

- 대충 만들어놓고 편하게 살 수도 있겠지만 세상은 그렇게 하도록 가만두지 않음

REST

- Roy Fielding은 HTTP가 뛰어난 프로토콜이라고 생각했고,
  network based application에서 HTTP를 더 잘 사용하고 싶어했음
- 논문 **Architectural Styles and the Design of Network-based Software Architectures** 에서 제안됨
- HTTP의 장점을 최대한 활용할 수 있는 아키텍처
- 사실상 HTTP API guideline 집합, HTTP API Design의 참고자료
- REST는 그러한 guideline에 포함된 추가적인 개념
- 이번 파트에선 REST에 언급된 내용은 일절 없음
  - '아 이게 REST구나' (x)
  - '아 이게 HTTP API 디자인 가이드라인이구나' (o)

API Specification - Request

URL

- collection과 single resource에 대해 두 개의 URL을 만들자
  - collection : 게시글들
  - single resource : 특정 게시글
- Use Consistently Plural Nouns
  - collection : `/posts`
  - single resource : `/posts/{post_id}`

API Specification - Request

Method

- URL이 지켜야 하는 룰 : Use Nouns instead of Verbs for Resources
  - /createPosts, /getPosts (X)
  - URL 구성은 simple하게 유지하고 HTTP method를 통해 세분화
- Read: `GET`
- Create: `POST` or `PUT`
- Update: `PUT` or `PATCH`
- Delete: `DELETE`
- /createPosts → **POST /posts**
- /getPosts → **GET /posts**

API Specification - Request

Method

- 다들 대충 느낌은 알고 있을 것
- int 타입의 `id` 를 가진 게시글(post) 리소스가 있다고 했을 때,
  - 게시글 생성은 `POST /posts`
  - 목록 조회는 `GET /posts`
  - 단일 게시글 조회는 `GET /posts/{post_id}` 등
- HTTP Method마다 정해진 semantic이 있어서, 맞춰주면 좋음
  - 예를 들어 `POST /posts/{post_id}` 같은 endpoint는 semantic에 맞지 않음

API Specification - Request

Method - Idempotence

- HTTP Method 각자는 Idempotence 제약사항을 지켜줘야 함
- RFC : Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the **side-effects of N > 0 identical requests is the same as for a single request.**
- single request의 결과와 2번 이상 요청했을 때의 side effect가 동일하면 Idempotent하다
  - `POST /posts` : single request는 데이터가 1번 생성, 2번 이상 요청하면 그만큼 더 생성되므로 not idempotent
  - `DELETE /posts/{post_id}` : single request던 2번 이상 요청했던 게시글이 삭제되는 것은 똑같으므로 idempotent

API Specification - Request

Method - Understand the Semantics

- GET (SQL : `SELECT`)
  - Idempotent
  - Used for Read-only. GET은 side effect가 생기면 안됨
  - collection, single resource에 모두 사용
    - collection : `GET /posts`
    - single resource : `GET /posts/1`
- POST (SQL : `INSERT`)
  - Not idempotent
  - Used for creating
  - collection에만 사용
    - `POST /posts`
    - `POST /posts/1/comments`

API Specification - Request

Method - Understand the Semantics

- PUT (SQL : `DELETE` and `INSERT`)
  - Idempotent
  - Used for creating and updating (full update)
    - Upsert 라고 부름
  - single resource에만 사용
    - `PUT /posts/1`
      - 게시글 1번을 업데이트하거나 새로 생성
- PATCH (SQL : `UPDATE`)
  - Idempotent
  - Used for partial updating (full update도 당연 가능)
  - single resource에만 사용
    - `PATCH /posts/1`
      - 게시글 1번을 업데이트 (payload에 포함된 필드만)

API Specification - Request

Method - Understand the Semantics

- DELETE (SQL : `DELETE`)
  - Idempotent
  - Used for deletion
  - single resource에만 사용
    - `DELETE /posts/1`

API Specification – Request

Method – 특별히 주의할 점

- PUT
  - upsert를 지원해야 함 (없으면 create, 있으면 update)
  - ID가 서버에서 generate되는 것이라면?
    - `PUT /posts/{post_id}`
      - `post_id` 가 `AUTO_INCREMENT INT` 라면 지원해주기 모호함
      - auto increment로 여태 쌓인 것은 10번까지인데, `PUT /posts/30` 이 들어온다면?
    - → create action을 지원하기 모호하다면 아예 PUT을 사용하지 말아야 함
      - full update는 PATCH의 범위에서 처리

API Specification - Request

Endpoint Design Diagram

1. DB Table을 가지고 collection과 single resource를 뽑아내기
    - collection : `posts` , single resource : `posts/{id}`
2. 기능을 어디까지 지원해줄지에 따라, 테이블에 들어가는 DML을 HTTP Method로 표현
    - 생성 지원 : `POST /posts` , 조회 지원 : `GET /posts` , `GET /posts/{post_id}`
    - 수정 지원 : `PATCH /posts/{post_id}` , 삭제 지원 : `DELETE /posts/{post_id}`
3. PUT을 지원할 수 있는 resource라면 `POST` 와 `PATCH` 를 `PUT` 하나로 퉁칠 수 있음
    - 보통 어느 테이블의 nullable 컬럼을 대상으로 했을 때 잘 들어맞음
    - 게시글마다 해시태그를 달 수 있다면 `PUT /posts/{post_id}/hashtags`
    - → `PATCH /posts/{post_id}` 에서 `hashtags` 필드를 지원하는 것으로 대체할 수도 있음

API Specification – Request

Use Verbs for Operations

- 행동을 명시할 때 동사를 사용하는 것이 요즘 세상에서 일반적으로 여겨짐

```
10  const internal = new Router();
11
12  internal.get('/flush', async (ctx) => {
13    const { INTERNAL_KEY } = process.env;
14    if (ctx.query.key !== INTERNAL_KEY) {
15      ctx.status = 403;
16      return;
```

```
25
26  internal.get('/check-file-size', async (ctx) => {
27    try {
28      const images = await UserImage.findAll({
29        where: {
```

- RPC-style API라고 얘기함
- 서버 재시작을 위해 `PATCH /server` 에 `{"restart": true}` 보다 `POST /restartServer` 가 직관적
- Roy Fielding의 논문에서 REST를 이야기할 때, 이렇게 하라는 이야기는 따로 없음

API Specification - Request

Tip - Magic Resource

- `GET /me`, `PATCH /me`
- 의외로 리소스 관점에서 완벽하게 괜찮다
- 논문 중 5.2.1.1 Resources and Resource Identifiers 단락
  - Any information that can be named can be a resource
  - → 이름을 지정할 수 있는 모든 정보는 resource가 될 수 있다

API Specification - Request

Tip - Toggle Action

어떤 게시글의 공개 여부를 설정하는 API가 있다고 가정,

1. 이름을 지정할 수 있는 것이 무엇인지 파악
    - 게시글 → 공개 여부 : `/posts/{post_id}/public`
2. create behavior가 필요한지 파악
    - 활성화 여부가 별도 테이블로 만들어져있거나 해서 DB Insert할 필요가 있다 → `PUT`
    - 활성화 여부가 해당 리소스의 컬럼 정도 수준이라 DB Insert할 필요가 없다 → `PATCH`
        - 오히려 `PATCH /posts/{post_id}` 에 `public` 필드를 지원하는 것도 방법
    - `PATCH` 는 create behavior를 지원하지 않는 것이 semantic이기 때문

API Specification - Request

Tip - Toggle Action

- 켠다/끈다 엔드포인트를 두는 것도 방법
  - `POST /posts/{post_id}/make-public`
  - `POST /posts/{post_id}/make-private`
- 이름을 부여할 수 있는 resource가 있다면 굳이 RPC-Style API를 작성할 필요가 없음

API Specification – Request

Query Parameter

- `GET /posts`, `GET /olderFirstPosts`, `GET /publicPosts`
  - → Query parameter를 통해 `GET /posts` 선에서 정리
  - `GET /posts`
  - `GET /posts?sort_by=created_at.desc`
  - `GET /posts?public=1`

API Specification - Request

Query Parameter - Best Practice

- 어딘가 확실한 표준은 없고, 보통 이렇게 쓰더라~ 하는 관례 수준의 practice들만 있음
- 중점은 SELECT 쿼리의 여러 요소들을 어떻게 query parameter라는 k/v pair로 표현할 것인지
- 대표적으로 3가지
  - filtering (WHERE query)
  - sorting (ORDER BY query)
  - limiting, pagination (LIMIT query)

API Specification - Request

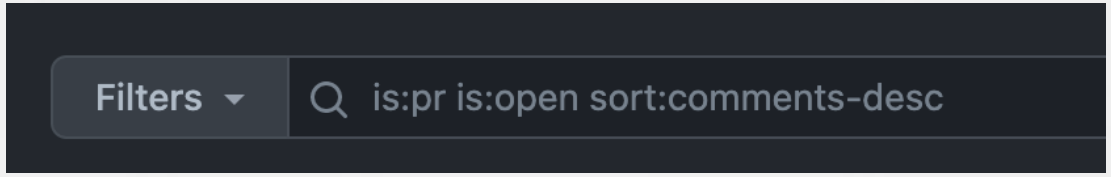
Query Parameter - Filtering

- exact match
  - key=value
  - `GET /posts?public=1&author=planb`
- IN query
  - key=value1,value2
  - `GET /posts?public=1,0`
- 더 많은 operator 지원
  - LHS Brackets
    - field[operator]=value
    - `GET /posts?created_at[gte]=2021-10-15&title[regex]=.+백엔드.+`
  - RHS Colon
    - field=operator:value
    - `GET /posts?created_at=gte:2021-10-15&title=regex:.+백엔드.+`
  - 직접 구현하긴 그렇고 보통 라이브러리가 있음 : nodejs 진영의 qs

API Specification - Request

Query Parameter - Sorting

- 파라미터 이름은 `sort` 나 `sort_by` 이용. field와 asc/desc를 명시해야 함. 여러 스타일 존재
  - `GET /posts?sort_by=asc(created_at)` , `GET /posts?sort_by=desc(created_at)`
  - `GET /posts?sort_by=+created_at` , `GET /posts?sort_by=-created_at`
  - `GET /posts?sort_by=created_at.asc` , `GET /posts?sort_by=created_at.desc`
- multi-column sort
  - `GET /posts?sort_by=desc(created_at),asc(upvotes)`
  - `GET /posts?sort_by=-created_at,+upvotes`
  - `GET /posts?sort_by=created_at.desc,upvotes.asc`
- GitHub의 예



API Specification - Request

Query Parameter - Pagination

- 대표적인 2가지 방식
  - Offset Pagination (SQL : LIMIT, OFFSET)
    - `GET /posts?limit=20&offset=20`
    - `GET /posts?size=20&skip=20`
    - Downside: 새로운 item이 추가되는 경우 다음 페이지에서 아이템이 중복 등장
    - page간 화면이 분리되어 있을 때만 사용하는 것이 좋음
      - (o) 게시글 목록
      - (x) 스크롤 형태의 SNS 서비스
  - Keyset Pagination (SQL : WHERE, LIMIT)
    - `GET /posts?limit=20&created_at[lt]=2021-10-15T10:00:00.000Z`
    - `GET /posts?limit=20&continuation_token=1613485859303.8572`

API Specification - Request

Header - 알아두기

- **Accept-Language: ko**
  - value에 해당하는 언어에 맞추어 컨텐츠를 제공
  - 참고 : i18n
- **Accept-Encoding: gzip, deflate**
  - response payload를 그대로 내려주지 않고 압축해서 내려주면 network 비용을 아낌
  - Accept-Encoding 헤더에 들어있는 알고리즘으로만 압축해줘야 함
  - 보통 프레임워크 수준에서 지원됨 (참고 : `server-compression-enabled` 설정)
- HTTP Header 찍히는 것 보고 각자가 어떤 용도인지 알아보면 좋은 배경지식이 될 수 있음
  - Request Header로는 Accept, Content-Type, Content-Length 등
  - Response Header로는 Content-Encoding, Vary 등

API Specification - Response

Status Code

- request message에서 method, url을 빼고 status code를 넣으면 response message structure
- 요청의 응답은 적절한 status code와 함께
  - 200 OK, 400 Bad Request, 404 Not Found, 500 Internal Server Error, ...
- 각 status code는 명확한 사용 기준이 있음
  - https://httpstatuses.com/
- 어떤 status code를 사용할지 결정하기 어려운 경우
  - https://github.com/for-GET/http-decision-diagram
- 제공하는 status code의 종류를 적게 유지하는 것이 좋음
  - 플랜비 : 200, 201, 400, 403, 404 정도만 사용하는 편
- 알아둬야 하는 것
  - `POST` 성공한 경우 `200 OK` 이 아닌 `201 Created` 사용
  - 권한이 없는 사용자의 리소스 접근에 `403 Forbidden` 대신 `404 Not Found` 사용 고려

API Specifiation - Response

Error Message

- status code 범위에서 구분하기 어려운 경우가 생길 수 있음
- 회원가입 API
  - user email이 중복됐을 때와 nickname이 중복됐을 때의 구분이 필요하다면?
  - status code 400 + error message

```
{
    "errors": [
        {
            "code": 101,
            "description": "User email duplicated.",
            "links": { "about": "https://bit.ly/api-errors/101" }
        },
        {
            "code": 102,
            "description": "User nickname duplicated.",
            "links": { "about": "https://bit.ly/api-errors/102" }
        }
    ]
}
```

API Specifiation - Response

Body

- JSON Response modeling에 대한 여러 컨벤션 존재
  - HAL
  - UBER
  - Siren
  - Collection+json
  - JSON:API
  - JSON-LD
- 그러나 너무 장황해서 response payload가 매우 고도화된 경우에야 고민해볼만 함
  - 참고 : https://jsonapi.org/

API Specifiation - Response

Body

- 플랜비가 그나마 사용하는 것
  - JSON:API에서,
  - 실제 데이터를 `data` 에 포함시키기
  - 에러는 `errors` 에 포함시키기

API Specifiation - Response

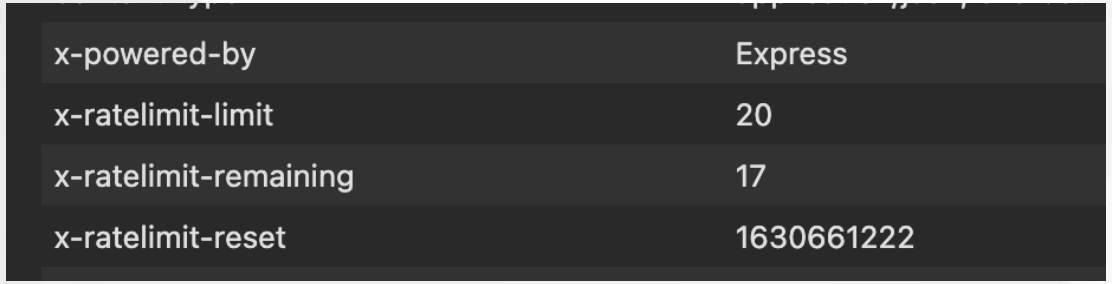
Header - 알아두기

- **Location: /posts/123**
  - POST 성공 시 response의 Location 헤더를 통해 생성된 리소스의 URL을 제공하는 것이 좋음
- **Content-Disposition: attachment; filename="filename.jpg"**
  - 다운로드되는 파일의 이름을 설정하는 헤더

API Specifiation - Response

Header - X-Header

- `x-` 또는 `X-` 로 시작하는 헤더
- 사용자 정의 header를 명시하기 위해 사용
- 사례) 공개 API에서 rate limiting 상태를 알려주기 위해 response에서 사용



x-powered-by	Express
x-ratelimit-limit	20
x-ratelimit-remaining	17
x-ratelimit-reset	1630661222

- body에 넣기는 좀 애매한 메타데이터를 X-Header에 끼는 경우가 종종 있음
- `x-` 를 제외하고 `ratelimit-limit` 로 쓸 수도 있지만, 표준 헤더에 포함되지 않은 것은 중간에 사라질수도 있음

API Documentation

API 문서화의 두 단계

1. API Spec DSL
    - API Blueprint
    - RAML
    - OpenAPI (aka Swagger)
    - example: https://petstore.swagger.io/v2/swagger.json
2. Visualize
    - Swagger
    - ReDoc 등
    - example: https://petstore.swagger.io/

API Documentation

UI로 한번에 하기

- GitBook API Documentation
- README.com
  - https://developers.intercom.com/intercom-api-reference/reference
  - https://docs.doppler.com/docs

API Documentation

몇가지 불만

1. 백엔드 : API Spec 작성하는 것 귀찮다.
2. 프론트엔드 : API Spec이 바뀌었는데 문서에서는 반영이 안 되어 있다.

- 두 문제를 한번에 해결하는 방법 : 코드와 문서를 강결합
  - 문서가 코드(request/response DTO)에서 자동으로 generate
  - 코드에서 API Spec을 변경하면 문서도 자동으로 바뀜
  - Spring REST Docs가 좋은 예

정리

- DB Table은 **resource**가 된다
- resource에 따라 collection, single resource로 분리하여 복수형 명사를 사용
- 여기에 HTTP Method들을 semantic에 맞추어 붙여주면 endpoint가 만들어진다
- GET에 디테일한 쿼리가 필요하다면 이런저런 query parameter 스펙을 추가하고
- 처리 결과는 status code로 표현한다
- 실질적인 데이터는 payload로 표현하고
- 메타데이터는 header로 표현한다
- `POST` 의 경우 `201 Created` 와 `Location` 헤더를 기억하자

정리

- API Docs는 플랜비의 경험 상 그냥 코드랑 문서가 강결합되어 있는 채로 자동 생성되는 게 좋더라
  - 어떤 필드의 설명을 바꾸겠다고 API 서버를 새로 배포하는게 좀 별로라고 생각했는데,
  - 그렇다고 강결합을 안 시키면 개발자들이 귀찮아서 + 기억을 못 해서 API 문서 최신화를 안 하더라
- 감이 안 잡힐텐데 GitHub API Docs 같은 것을 보면 느낌을 알 수 있다
- 개발자도구 열고 API call 어떻게 하는지 구경하는 것도 좋다(velog 등)

Bye!

- 4, 5일차 강의는 없습니다.
  - 데브코스의 장점은 비슷한 관심사와 비슷한 숙련도를 가진 사람들이 모여있다는 것
  - 무언가 더 전달해드리고 싶었지만,
  - 웬만한 강의보다 자습이 더 공부가 잘 될거라고 생각해요
  - 11주차 내용에 연계된, 공부에 도움되실만한 자료들을 올려드릴게요