```
(
  (
    (
      (
        2016년에 자바스크립트를 배우는 기분
      )을 읽는다는 것
    )에 관한 세션
  )을 듣는다는 것
)
```

조승연

Knowre

play.node 2016

[Disclaimer]

이 세션에서는 새로운 기술에 대해 이야기하지 않습니다

13:30 ~ 14:10	How the node.js event loop works
	Bert Belder / StrongLoop

Bert Belder의 세션을 포기하고 들어와주셔서 감사합니다


자바스크립숲 게임

1. 4~6 정도의 글자수를 정합니다

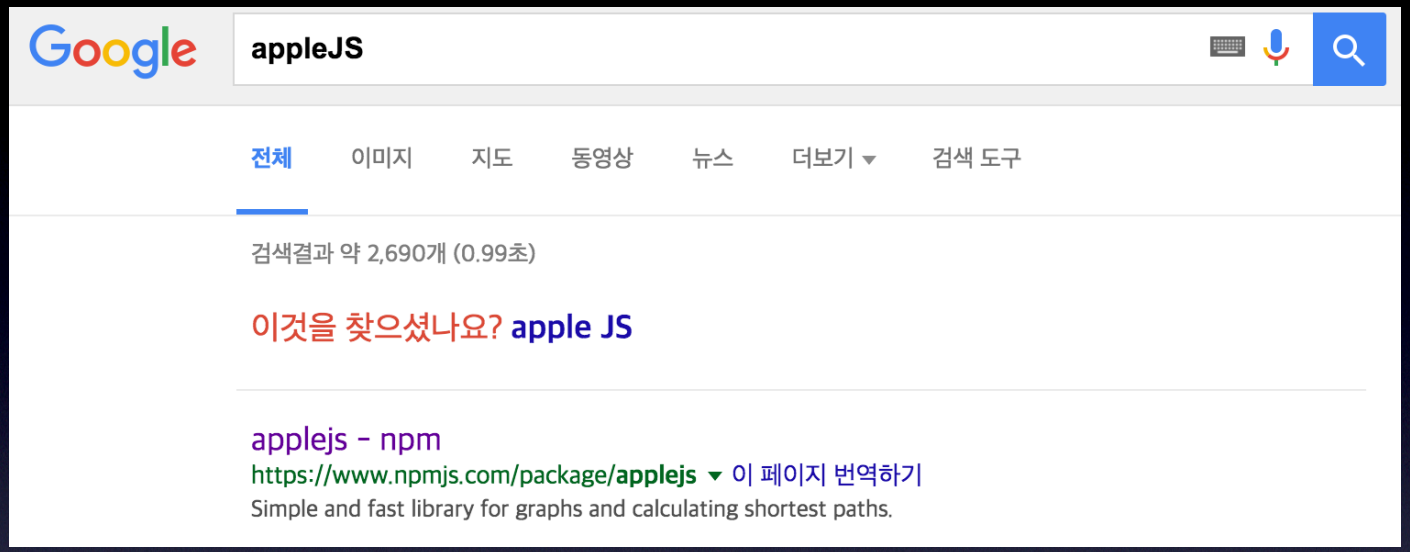
2. 해당 글자수로 이루어진 영어단어를 돌아가면서 말합니다

3. 그 단어(예: Apple) 뒤에 JS를 붙여서 구글링해 봅니다

4. 해당하는 라이브러리가 있으면 마십니다

신나고 재미나는~

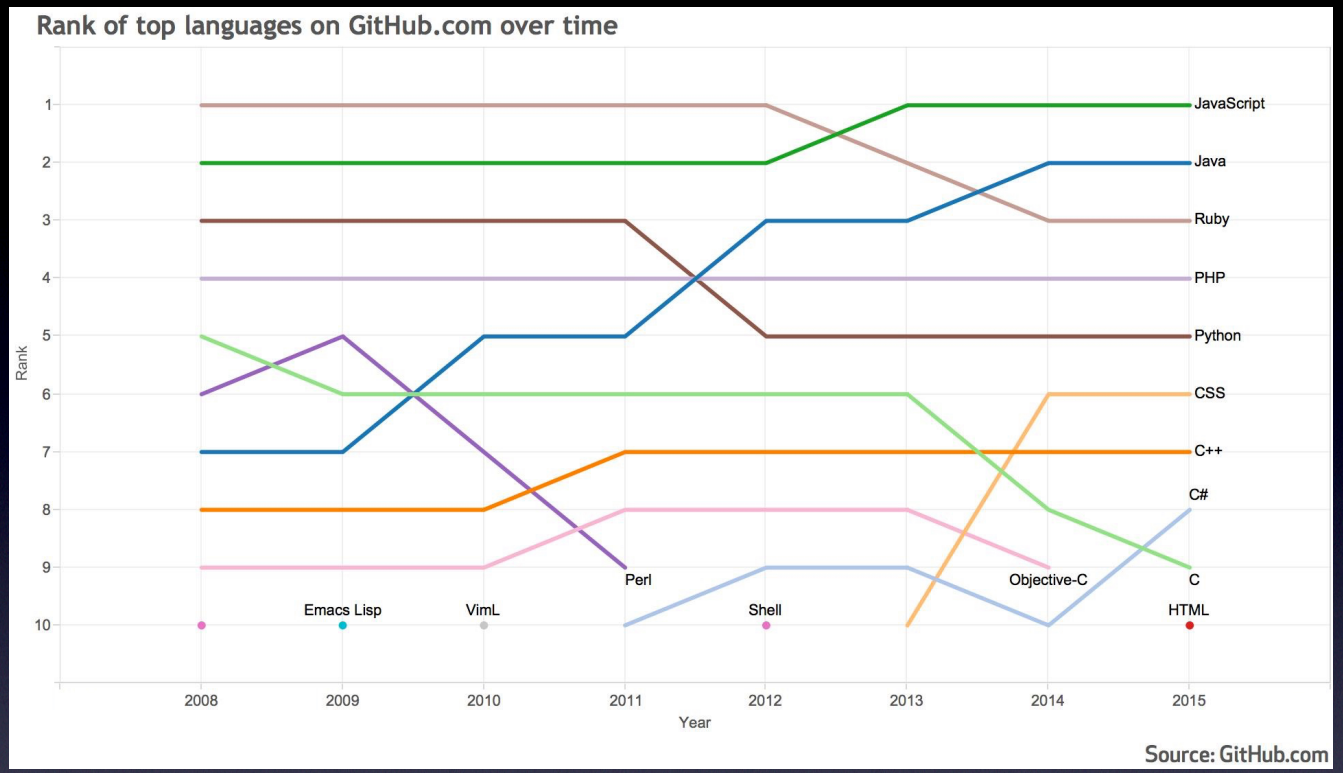
자바스크립트 게임



~~신나고 재미나는~~

자바스크립트 게임

교훈: 자바스크립트 라이브러리는 정말 정말 정말 많습니다



#1 language on Github

Usage

```
const leftPad = require('left-pad')

leftPad('foo', 5)
// => "  foo"

leftPad('foobar', 6)
// => "foobar"

leftPad(1, 2, '0')
// => "01"

leftPad(17, 5, 0)
// => "00017"
```

left-pad 사태

1. 지난 3월, Azer Koçulu라는 프로그래머가 npm과 마찰

2. 본인이 짠 모든 패키지를 npm에서 삭제

3. 그 중 left-pad라는 패키지에 의존하는 수많은 라이브러리들이 작동 불능에 빠짐

4. 해당 라이브러리들에 의존하는 서비스들 역시 자동화된 빌드시스템에서 작동 또는 배포 불능에 빠짐

left-pad 사태: 전개

1. 자바스크립트 커뮤니티는 저런 간단한 라이브러리도 불러와서 썼어야 하는가

2. isArray 패키지: 한 줄짜리 코드, 하루 110만 다운로드, 72개의 패키지가 여기에 의존

3. ES2017의 String.prototype.padStart(padEnd) 함수 추가로 훈훈한(?) 결말

left-pad 사태: 반성과 결말

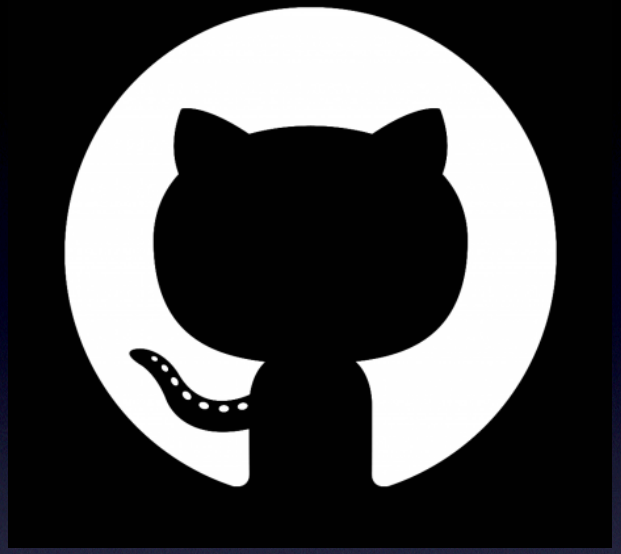
0. 간단한 웹 페이지를 만들고 싶다!
1. React를 배워야 한다. 2016년이니까!
2. React를 쓰려면 JSX를 배워야 한다. 2016년이니까!
3. React를 쓰려면 Babel을 써야 한다. 2016년이니까!
4. 라이브러리를 하나의 파일로 묶는게 좋다. 2016년이니까!
5. Gulp Grunt 대신 Webpack을 배워야 한다. 2016년이니까!

문제의 글:
“2016년에 자바스크립트를 배우는 기분”

6. TypeScript를 배워야 한다. 2016년이니까!
7. 함수형 프로그래밍을 하자. 2016년이니까!
8. XMLHttpRequest 대신 fetch()를 쓰자. 2016년이니까!
9. 상태 관리는 Redux로 하자. 2016년이니까!
10. 템플릿은 ES6 네이티브로 쓰자. 2016년이니까!
11. 비동기 관리는 async/await로 하자. 2016년이니까!
12. 이런게 싫다면 파이썬 3를 해라!

문제의 글:
“2016년에 자바스크립트를 배우는 기분”

자바스크립트 힙스터문화



Since RoR,

10분 데모를 위한 프레임워크

간단한 데모 이상을
해 보지 않고 소개하는 '전문가'

실제 '유의미한' 프로젝트를 수행할 때
고려되어야 하는 것들에 대한 간과

오늘, 그런 것들에 대해
이야기하고자 합니다

1. 빌더, 혹은 태스크 매니저

2. 트랜스파일러

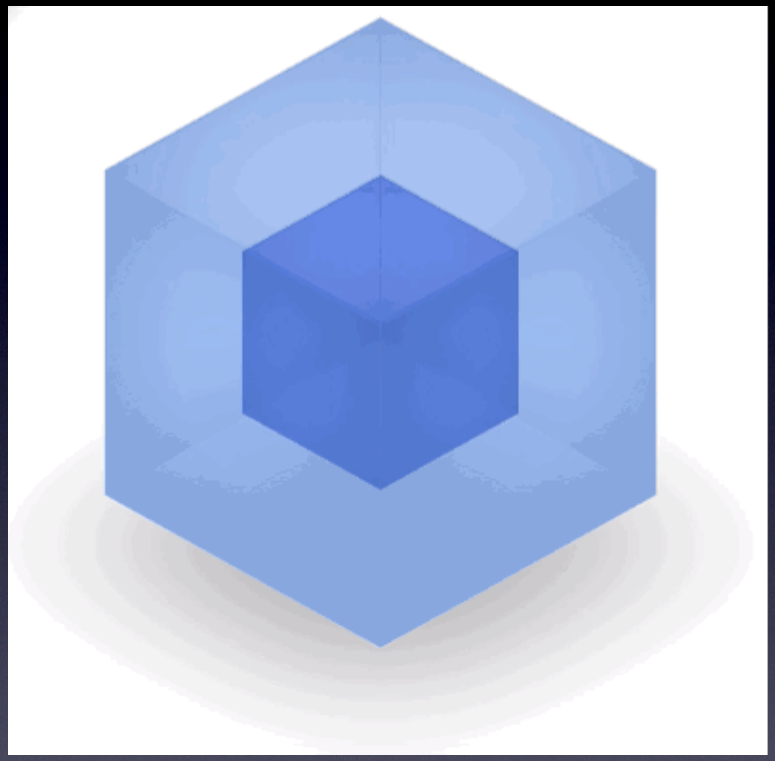
3. 프레임워크

1. 빌더, 혹은 태스크 매니저

2. 트랜스파일러

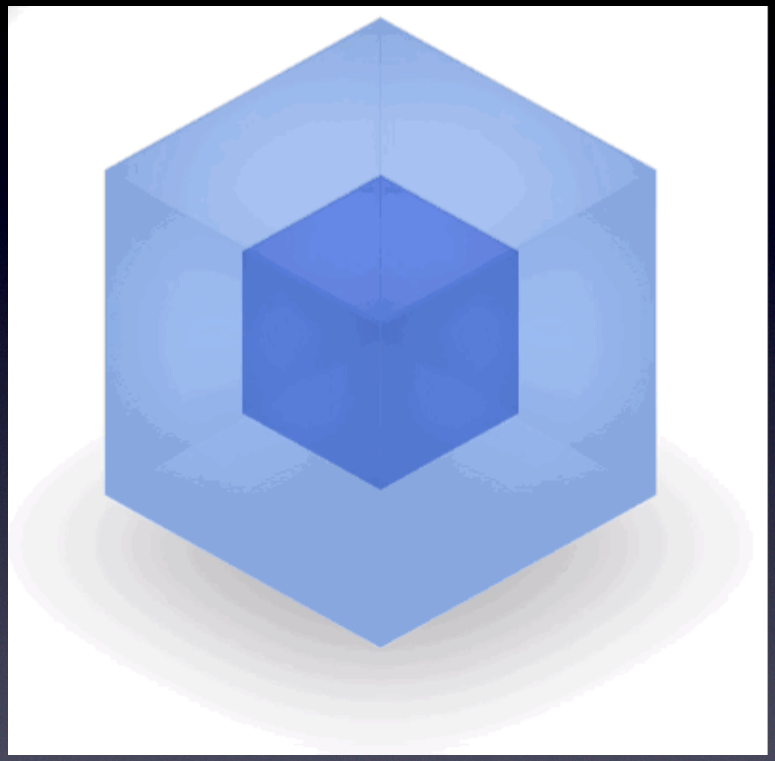
3. 프레임워크

태스크매니저가 하는 일들



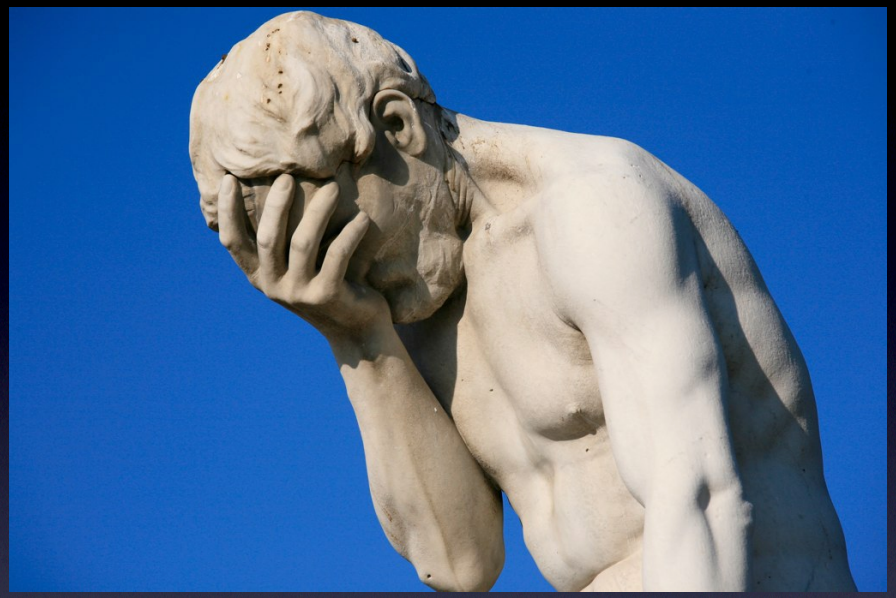
Transpile,
Concatenate,
Minify,
Lint,
Template process,
Image sprite generate,
Upload,
Purge,
Test,
...

그 밖에도..



실시간 리로딩
HMR(Hot module replacement)
실시간 트랜스파일

```
# gcc -o mytest.c mytest.out
```



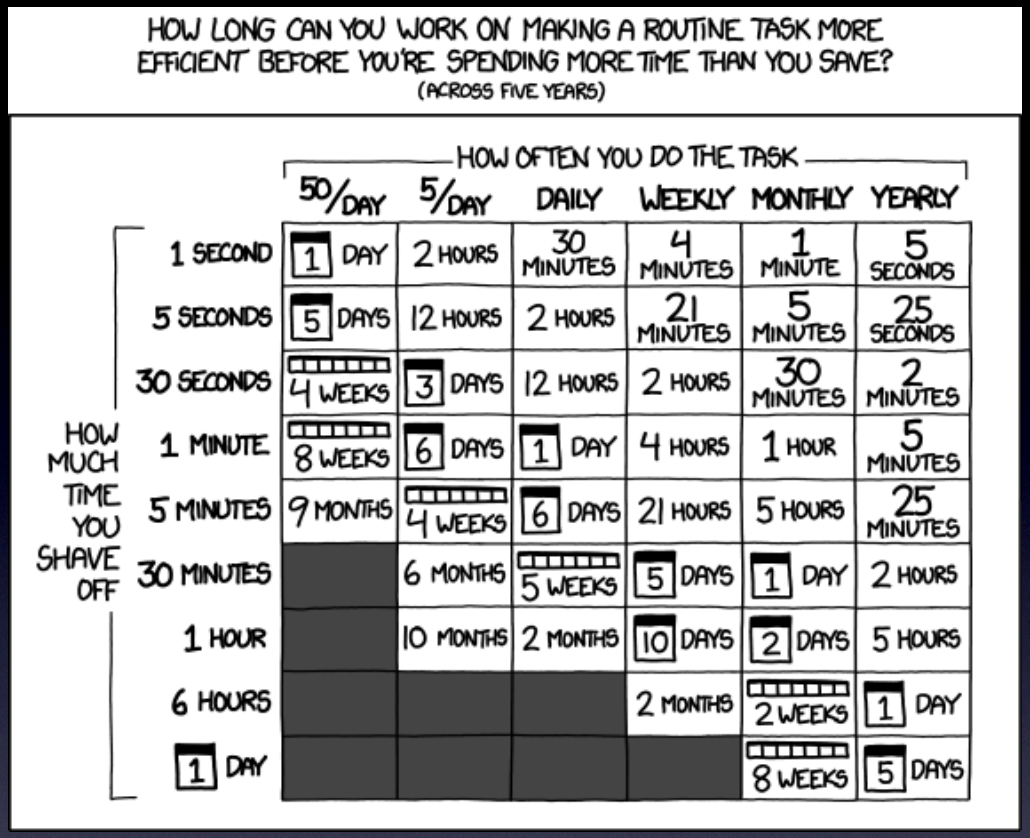
```
 1 #
 2 # makefile sample that uses phony ta
 3 #
 4
 5 # *** MACROS
 6
 7 INSTPATH=./bin/myapp
 8 INCPATH=./include
 9 OBJPATH=./obj
10 CC=cc
11 CFLAGS=-g -Wall -I$(INCPATH)
12 COND1=`stat app 2>/dev/null | grep M
13 COND2=`stat $(INSTPATH) 2>/dev/null
14
15 # *** Targets
16
17 all: getobj app install putobj
18
19 app: main.o mod_a.o mod_b.o
20         $(CC) $(CFLAGS) -o app main.
21
22 main.o: main.c $(INCPATH)/inc_a.h $(
23         $(CC) $(CFLAGS) -c main.c
24
25 mod_a.o: mod_a.c $(INCPATH)/inc_a.h
```

C 시절을 떠올려 봅시다

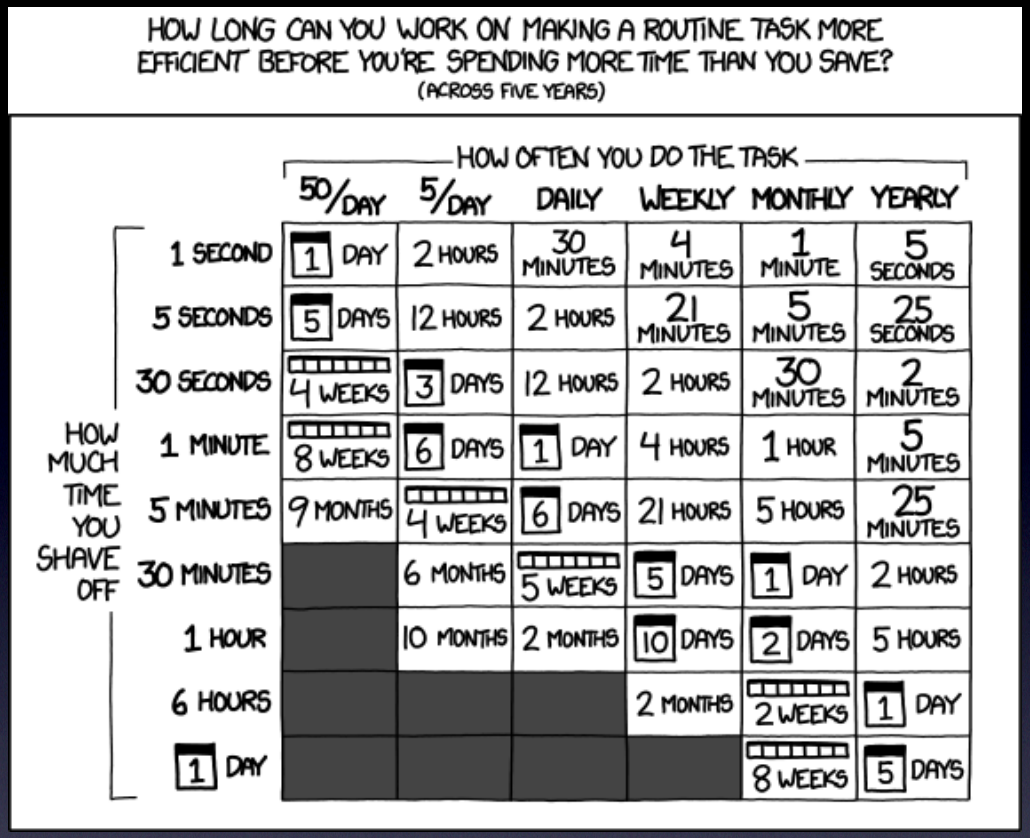
커맨드라인 빌드
→ Makefile을 이용한 빌드
→ Makefile을 만들어주는 프로그램
→ ……

빌드 툴들의 백가쟁명

IDE가 답일까?
외부에서 긴급히 개발환경을 세팅해야 한다면?



빌드 자동화: 생산성을 위해

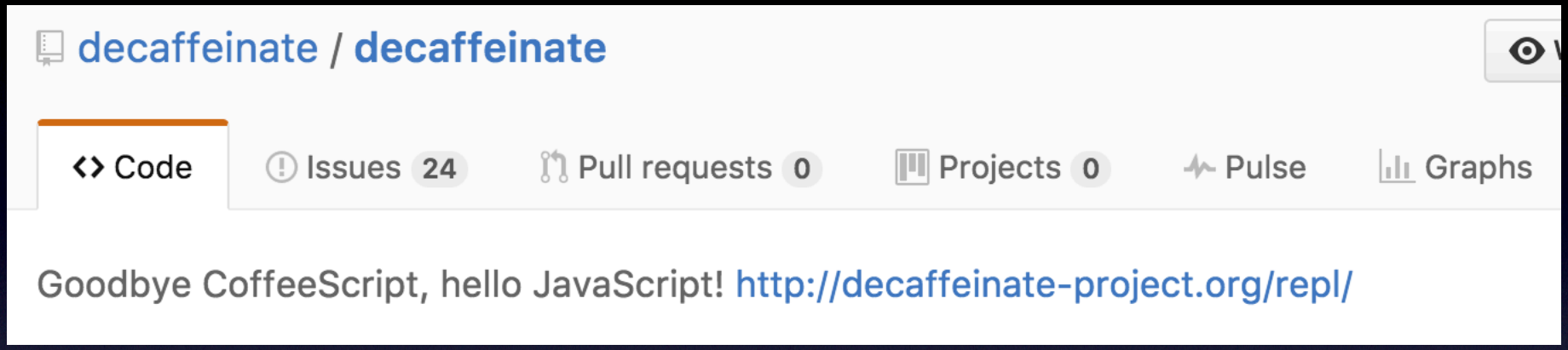


정말로 세팅하는 시간 이상의 생산성 효과가 있을까요?

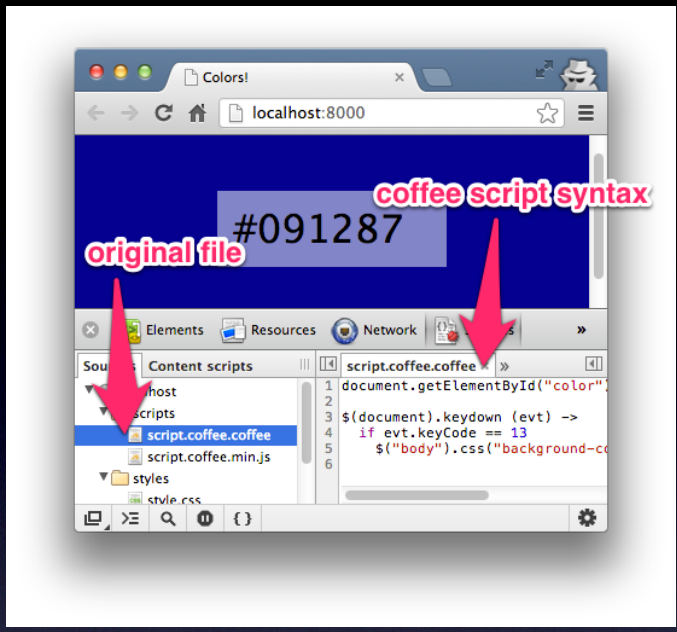
1. 빌더, 혹은 태스크 매니저

2. 트랜스파일러

3. 프레임워크



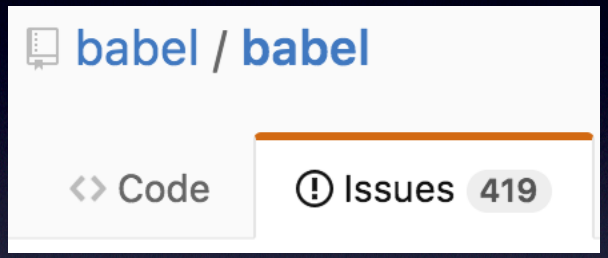
CoffeeScript의 몰락



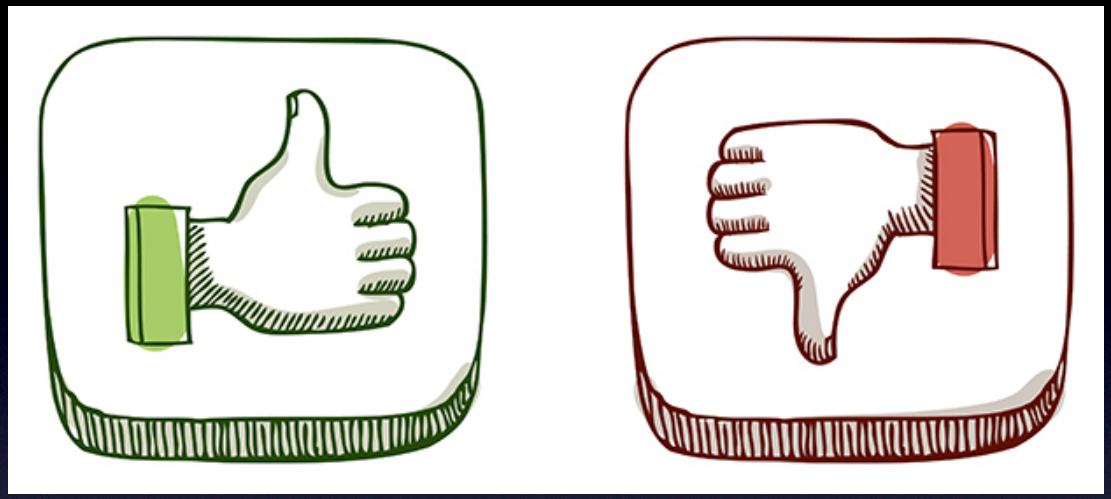
트랜스파일은 생각보다 큰 작업들을 수반합니다

```
function foo() {
  return typeof null === "undefined";
}

for(var i=0; i<10000; ++i) console.log(foo())
```



레이어가 추가된다는 것
: 버그의 여지가 추가된다는 것



장점: 약간 더 완성도 있는 문법
단점: 세팅에 걸리는 시간

```coffeescript
yearsOld = max: 10, ida: 9, tim: 11

ages = for child, age of yearsOld
  "#{child} is #{age}"
```

```javascript
for (var [child, age] of yearsOld)
  alert(child + ' is ' + age);
```

CoffeeScript - 불투명한 미래
새 문법 지원의 딜레마

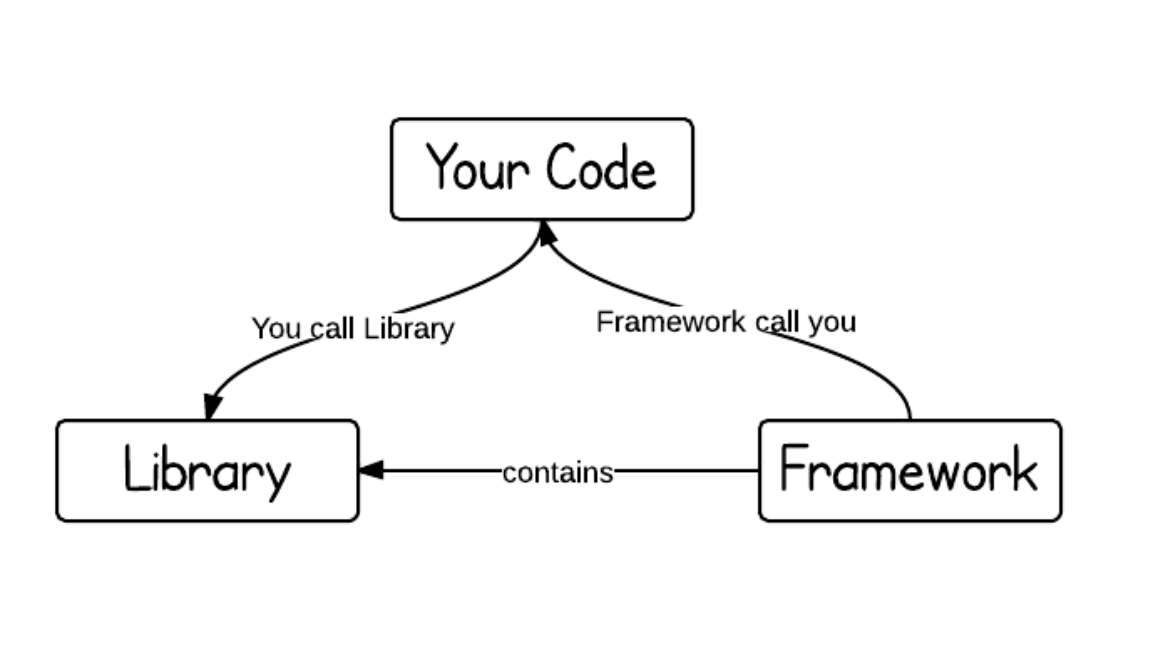
걷어내고자 할 때
고스란히 기술부채가 됩니다

과연 10분짜리 데모와 소개들이
이런 잠재적 문제를 이야기해 줄까요?

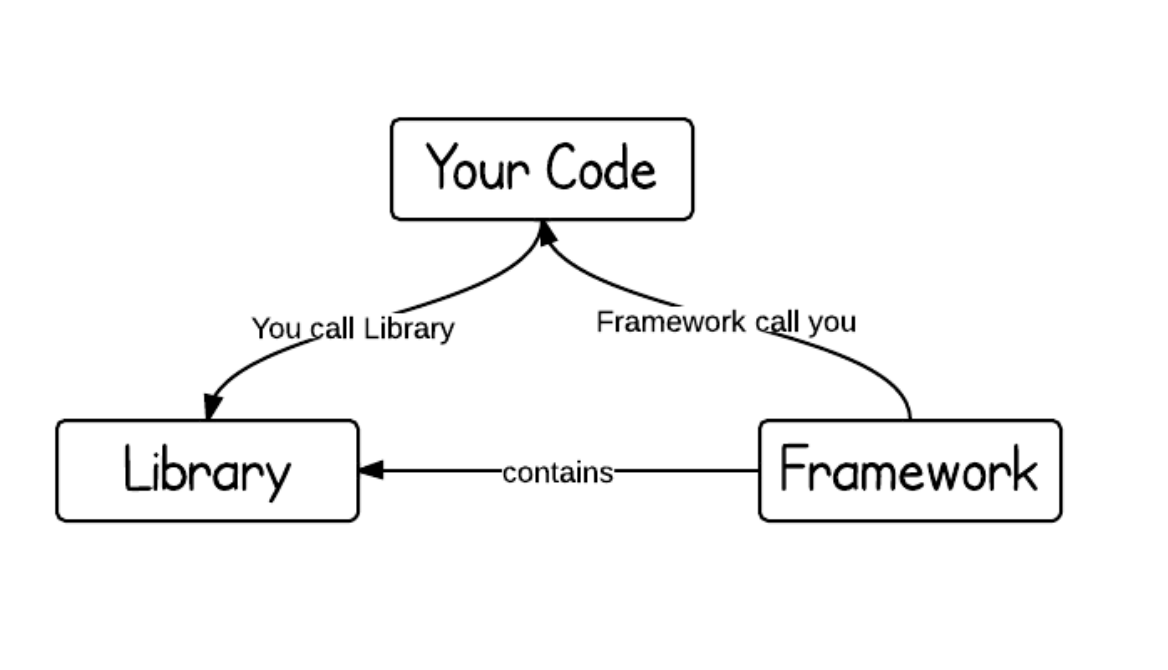
1. 빌더, 혹은 태스크 매니저

2. 트랜스파일러

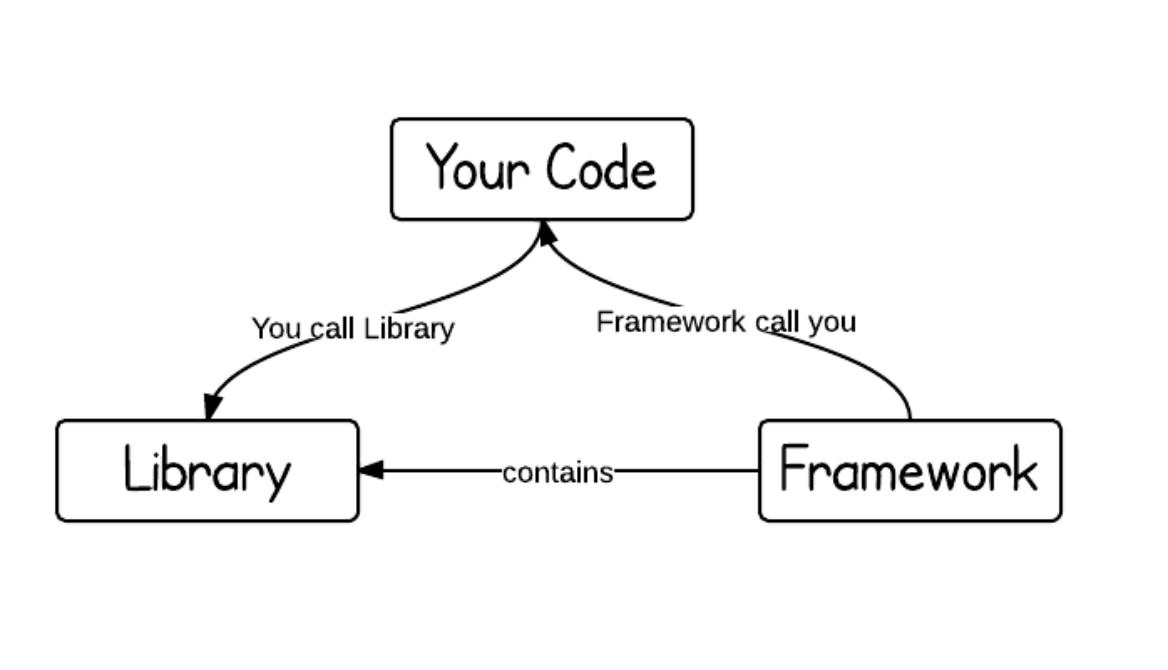
3. 프레임워크



프레임워크와 라이브러리의 차이



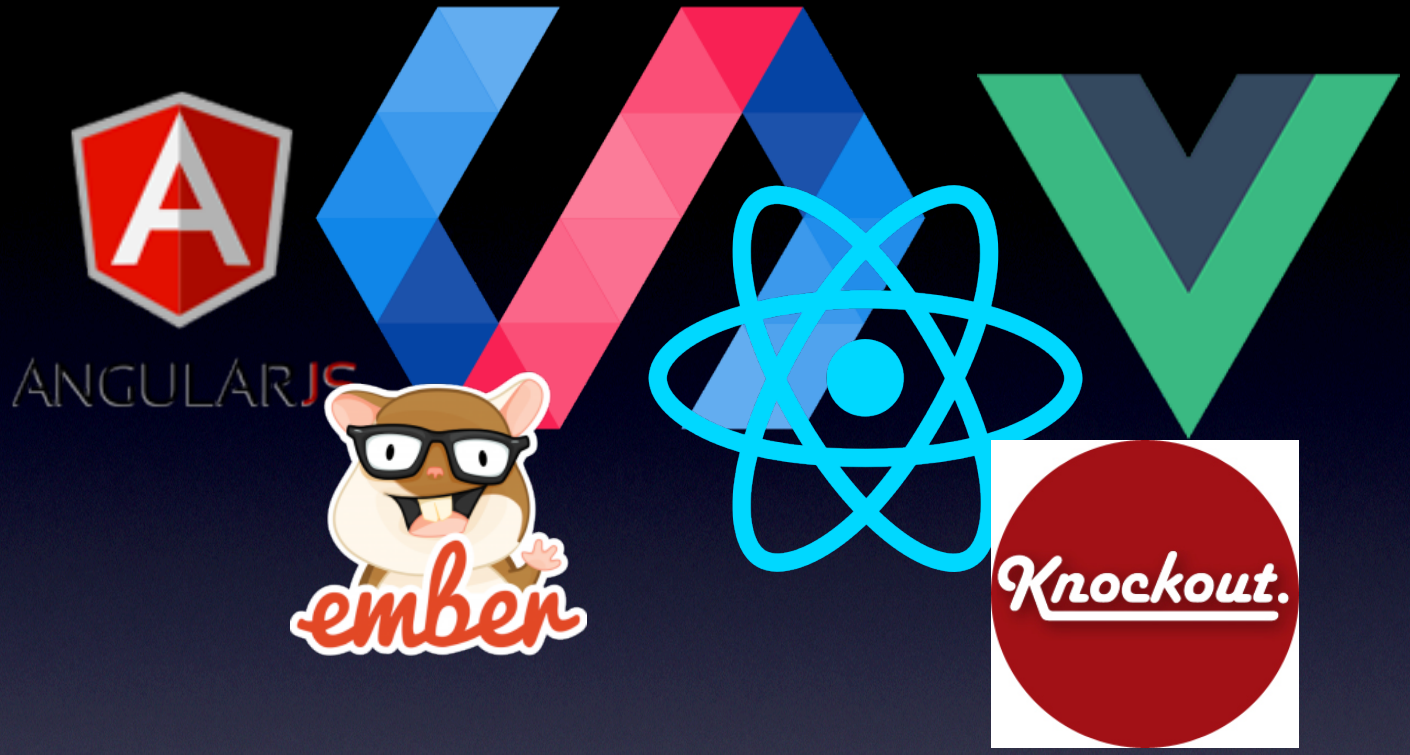
당신의 코드가
라이브러리를 컨트롤합니다



프레임워크가
당신의 코드를 컨트롤합니다

```
1.  <!doctype html>
2.  <html ng-app>
3.    <head>
4.      <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.8/
    angular.min.js"></script>
5.    </head>
6.    <body>
7.      <div>
8.        <label>Name:</label>
9.        <input type="text" ng-model="yourName" placeholder="Enter a name
    here">
10.       <hr>
11.       <h1>Hello {{yourName}}!</h1>
12.     </div>
13.   </body>
14. </html>
```

더 이상 쓰고싶지 않을 때 가장 심각한 기술부채



하지만 우리는 무엇이 됐든 프레임워크를 쓰게 될 것입니다

프레임워크 선정의 고려요소

1. 얼마나 오래 쓸 것인가
→ 회사에서는 항상 과소평가됩니다

2. 어느 정도의 기술부채를 감수할 수 있는가
→ 이미 돌아가고 있는 라이브러리와의 충돌

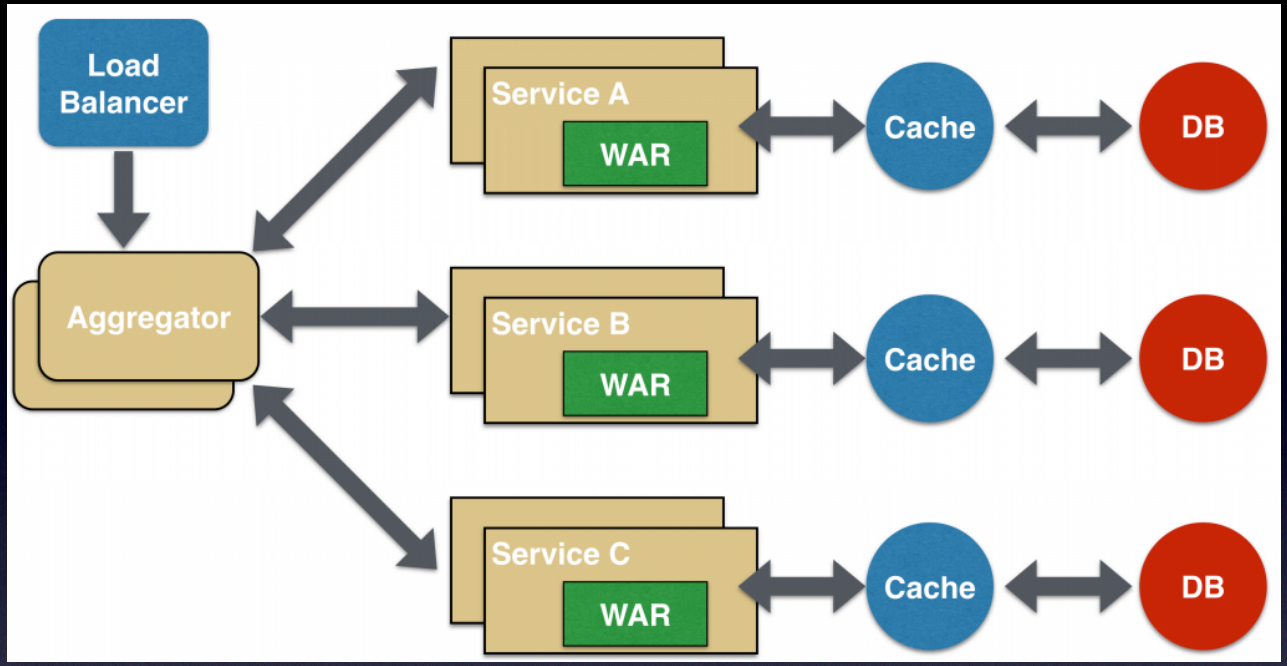
3. 팀원들 어느 정도가 만족하는가
→ 컨벤션과 코딩 스타일을 맞추려는 노력

결론

설계, 혹은 프로그래밍의 본질

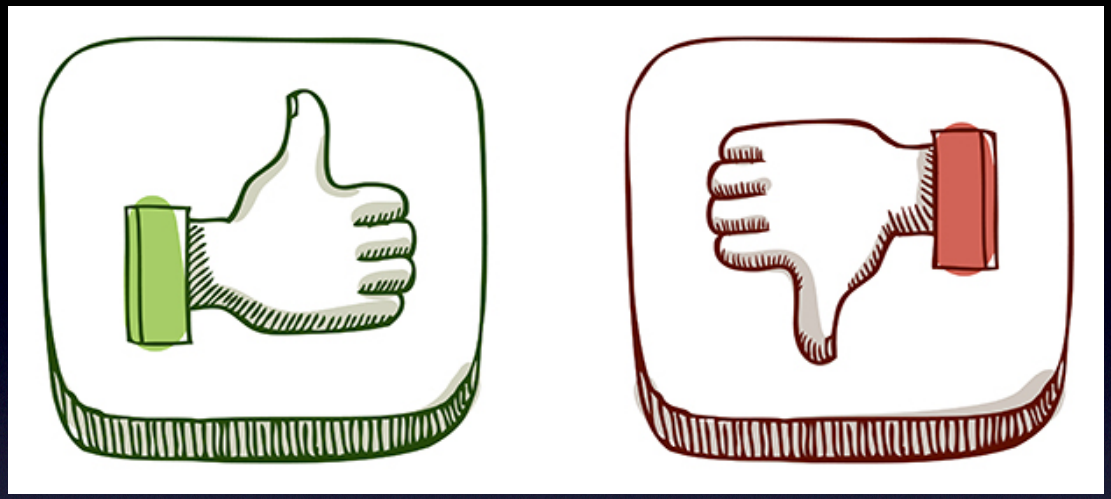
우리 모두는
손가락으로 폐기물을 만듭니다

하지만 중요한 것은
그 폐기물이 서로 섞이지 않게 하는 것



서버: 마이크로서비스 구조
→ 오후의 다른 세션들을 참고하세요!

클라이언트: 구성요소별 라이브러리
→ 독립적으로 기능할 수 있어야 합니다



잃는 것: 약간의 최적화, Eventual Consistency
얻는 것: 몸과 정신의 건강

0. 간단한 웹 페이지를 만들고 싶다!
1. React를 배워야 한다. 2016년이니까!
2. React를 쓰려면 JSX를 배워야 한다. 2016년이니까!
3. React를 쓰려면 Babel을 써야 한다. 2016년이니까!
4. 라이브러리를 하나의 파일로 묶는게 좋다. 2016년이니까!
5. Gulp Grunt 대신 Webpack을 배워야 한다. 2016년이니까!

문제의 글:
“2016년에 자바스크립트를 배우는 기분”

6. TypeScript를 배워야 한다. 2016년이니까!
7. 함수형 프로그래밍을 하자. 2016년이니까!
8. XMLHttpRequest 대신 fetch()를 쓰자. 2016년이니까!
9. 상태 관리는 Redux로 하자. 2016년이니까!
10. 템플릿은 ES6 네이티브로 쓰자. 2016년이니까!
11. 비동기 관리는 async/await로 하자. 2016년이니까!
12. 이런게 싫다면 파이썬 3를 해라!

문제의 글:
“2016년에 자바스크립트를 배우는 기분”

결국 무엇보다 중요한 것은

Hype에 휩쓸리지 않는 무게중심

eof