Table of Contents

Tech Notes

참고자료

외부링크

Tech Notes

OSX Tip

.dev 도메인 접속 안될때

```
sudo rm /etc/resolver/dev
```

cli로 애플리케이션 실행하기

```
do shell script  "open -na ShellCraft"
```

스크린샷 디렉토리 경로 변경

```
defaults write com.apple.screencapture location folder_name_here
```

Back-end

Database

MySQL

레거시 테이블 정보 문서화를 돕는 쿼리

```
SELECT TABLE_SCHEMA,
       TABLE_NAME,
       COLUMN_NAME,
       REPLACE(CONCAT(DATA_TYPE, '(', IFNULL(CHARACTER_MAXIMUM_LENGTH, IFNULL(
       COLUMN_KEY,
       IS_NULLABLE,
       COLUMN_DEFAULT,
       COLUMN_COMMENT
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA NOT IN ('information_schema', 'performance_schema', 'mysql'
ORDER BY TABLE_SCHEMA, TABLE_NAME, ORDINAL_POSITION;
```

Infra

Shell Script

MySQL loadavg 모니터링

/proc/loadavg\ **이 파일의 처음 세 필드는 1, 5, 15분 동안 평균 실행 대기열 또는 디스크 I/O에 있는 로드 평균 수치를 가르킨다.

```
#!/bin/sh

mysql_loadavg=`sshpass -f /root/db_password.txt ssh 123.123.123.123 cat /proc/

if [ "$mysql_loadavg" -ge "10" ]; then
    curl "https://example.com/push" --data "mesg=slave1:123.123.123.123 loadavg:$m
fi
```

Front-end

튜토리얼 링크 모음

- **모던 JavaScript 튜토리얼** : https://ko.javascript.info/**
- **Learning JS** : https://april.gitbook.io/learning-js/

JavaScript 기초

스코프 (Scope)

1. 스코프란?

- 스코프(*scope*)는 **변수에 접근할 수 있는 범위**를 말한다.
- 범위는 중괄호(블록) 또는 함수에 의해 나뉜다.
- 자바스크립트는 새로운 함수가 생성 될 때마다 새로운 스코프가 발생한다.

1-1. 스코프의 주요 규칙

1. 안쪽 스코프에서 바깥쪽 스코프로 접근할 수 있지만 반대는 불가능하다.
2. 스코프는 중첩이 가능하다.
3. 전역/지역 스코프\ - 가장 바깥족의 스코프를 전역 스코프라고 부른다.\ - 전역이 아닌 다른 스코프는 모두 지역 스코프이다.
4. 지역 변수는 전역 변수보다 우선 순위가 높다.\ - 전역 스코프에서 선언한 변수는 전역 변수\ - 지역 스크포에서 선언한 변수는 지역 변수\ - 지역 변수는 전역변수보다 더 높은 우선 순위를 가진다.

> `전역 스코프` (*global*)는 어디에서든 해당 변수에 접근 가능
>
> `지역 스코프` (*local*)는 한정적인 범위에서 해당 변수에 접근이 가능

2. 함수 스코프 (Function scope)

- 함수가 선언되면 하나의 스코프가 발생하는데 이걸 함수 스코프라고 한다.
- 함수 스코프는 **함수에서 선언한 변수는 해당 함수 내에서만 접근이 가능**하다.

```
# 함수 스코프 예시
function fn1() {
  var aa = '12';  // 함수 스코프, 지역 변수 (local)
}
console.log(aa);  // Uncaught ReferenceError: aa is not defined

# 전연 스코프 예시
var abc = '123';  // 전역 스코프, 전역 변수 (global)
function fn2() {
  console.log(abc);
}
fn2(); // 123
```

> `var` 키워드는 함수 내에서만 지역 변수로 유지된다.

3. 블록 스코프 (Block scope)

- 블록 스코프는 **블록(`{}`)내부에서 선언된 변수는 해당 블록에서만 접근 가능**한 걸 말한다.

```
// var 예시
if (2 > 1) {
    var aa = 1;
}
console.log(aa); // 1

// let, const 예시
if (2 > 1) {
    let bb = 1;
}
console.log(bb); // Uncaught ReferenceError: aa is not defined
```

> ES6 이전 전통적인 JavaScript에는 함수 스코프와 전역 스코프 두 가지만 존재했다.
>
> `var` 로 선언한 변수는 함수 내부 또는 외부에서 선언되었는지에 따라 함수 스코프 또는 전역 스코프를 가진다.
>
> 이때, 중괄호로 표시된 블록이 스코프를 생성하지 않는다는 점에서 혼란을 일으킬 수 있다.
>
> C나 Java와 같이 블록이 스코프를 생성하는 언어의 경우 let, const 예시처럼 에러가 발생한다.\ 그러나 **블록은 var로 선언한 변수에 대해 스코프를 생성하지 않기 때문에 var 명령문은 전역 변수로 생성한다.**\ \ 요약하자면, ES6부터 블록은 스코프로 취급되기 시작했지만, 이는 `let` 과 `const` 로 변수를 선언했을 때만 유효하다.

scope	const	let	var
global	N	N	Y
function	Y	Y	Y
block	Y	Y	N
reassigned	N	Y	Y

4. 렉시컬 스코프 (Lexical scope : 어휘적 범위)

- 함수를 어디서 **선언** 하였는지에 따라 상위 스코프를 결정한다.
- JavaScript를 비롯한 대부분의 프로그래밍 언어는 렉시컬 스코프 (정적 스코프 = Static scope)이다.

```
function init() {
    var name = "Mozilla"; // name은 init에 의해 생성된 지역 변수이다.
    function displayName() { // displayName() 은 내부 함수이며, 클로저다.
      alert(name); // 부모 함수에서 선언된 변수를 사용한다.
    }
    displayName();
  }
  init();
```

5. 동적 스코프 (Dynamic scope)

- 함수를 어디서 호출 하였는지에 따라 상위 스코프를 결정한다.

```
function foo() {
  console.log(x);
}

function bar() {
  x = 15;
  foo();
}

var x = 10;
foo(); // 10
bar(); // 15
```

즉시 호출하는 함수 표현식 (IIFE)

IIFE란, "Immediately Invoked Function Expression"의 줄임말로, **정의되자마자 즉시 실행되는 함수 표현식**을 말한다.

> **'Self-Executing Anonymous Function'** 이라고도 불리며,\ 전역 스코프에 불필요한 변수를 추가해서 오염시키는 것을 방지할 수 있을 뿐 아니라 IIFE 내부안으로 다른 변수들이 접근하는 것을 막을 수 있는 방법이다.

IIFE의 목적

IIFE는 **외부에서 접근할 수 없는 자체 Scope를 형성한다**.\ Parser는 JavaScript에서 변수의 Scope가 함수에 의해 정해진다는 것을 알고 있다. 그러므로 IIFE 함수는 **상위 Scope에 접근할 수 있으면서도, 내부 변수를 외부로부터 보호해 Privacy를 유지할 수 있다**.

따라서 IIFE 사용의 가장 큰 목적은 **데이터 프라이버시**와 **코드 모듈화**라고 할 수 있다.

```
// 예시
(function() {
    // IIFE 바디
    console.log("IIFE");
})();

// 화살표 함수로도 사용 가능하다
(() => {
    console.log("IIFE");
})();
```

```
const f = (function() {
    var count = 0; // 외부에서 접근 할 수 없는 내부 변수
    return function() {
        return `I have been called ${++count} time(s).`;
    }
})();

f(); // 'I have been called 1 time(s).'
f(); // 'I have been called 2 time(s).'
f(); // 'I have been called 3 time(s).'
f(); // 'I have been called 4 time(s).'

console.log(count); // ReferenceError: count is not defined

var f2 = (function (x) {
    return x * 2;
})(2);
console.log(f2); // 4
```

Map 과 Set

Map

맵(Map)은 키가 있는 데이터를 저장한다는 점에서 `객체` 와 유사하다. \ 다만, `맵` **은 키에 다양한 자료형을 허용**한다는 점에서 차이가 있다.

맵에는 다음과 같은 주요 메서드와 프로퍼티가 있다.

- `new Map()` – 맵 생성
- `map.set(key, value)` – `key` 를 이용해 `value` 를 저장
- `map.get(key)` – `key` 에 해당하는 값을 반환합니다. `key` 가 존재하지 않으면 `undefined` 를 반환
- `map.has(key)` – `key` 가 존재하면 `true` , 존재하지 않으면 `false` 를 반환
- `map.delete(key)` – `key` 에 해당하는 값을 삭제
- `map.clear()` – 맵 안의 모든 요소를 제거
- `map.size` – 요소의 개수를 반환

```
let map = new Map();

map.set('1', 'str1');   // 문자형 키
map.set(1, 'num1');     // 숫자형 키
map.set(true, 'bool1'); // 불린형 키

// 객체는 키를 문자형으로 변환한다.
// 맵은 키의 타입을 변환시키지 않고 그대로 유지한다.
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'
alert( map.size );     // 3
```

맵은 객체와 달리 키를 문자형으로 변환하지 않는다. 키엔 자료형 제약이 없다.

`map[key]` 는 `Map` 을 쓰는 바른 방법이 아닙니다.

`map[key] = 2` 로 값을 설정하는 것 같이 `map[key]` 를 사용할 수 있긴 합니다. 하지만 이 방법은 `map` 을 일반 객체처럼 취급하게 됩니다. 따라서 여러 제약이 생기게 되죠.

`map` 을 사용할 땐 `map` 전용 메서드 `set` , `get` 등을 사용해야만 합니다.

맵은 키로 객체를 허용한다.

```
let john = { name: "John" };
let visitsCountMap = new Map();
visitsCountMap.set(john, 123);
alert( visitsCountMap.get(john) ); // 123
```

객체를 키로 사용할 수 있다는 점은 `맵` 의 가장 중요한 기능 중 하나이다. \ Object에는 객체형 키는 사용할 수 없습니다. (문자형으로 변환하기 때문)

```
let john = { name: "John" };
let visitsCountObj = {};
visitsCountObj[john] = 123;
alert( visitsCountObj["[object Object]"] ); // 123
```

맵 이 키를 비교하는 방식

맵 은 SameValueZero라 불리는 알고리즘을 사용해 값의 등가 여부를 확인한다. \ 이 알고리즘은 일치 연산자 === 와 거의 유사하지만, NaN 과 NaN 을 같다고 취급하는 것에서 일치 연산자와 차이가 있습니다. 따라서 맵에선 NaN 도 키로 쓸 수 있습니다.

이 알고리즘은 수정하거나 커스터마이징 하는 것이 불가능합니다.

체이닝

map.set 을 호출할 때마다 맵 자신이 반환된다. 이를 이용하면 map.set 을 '체이닝(chaining)'할 수 있습니다.

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

맵의 요소에 반복 작업하기

다음 세 가지 메서드를 사용해 맵 의 각 요소에 반복 작업을 할 수 있다.

- map.keys() – 각 요소의 키를 모은 반복 가능한(iterable, 이터러블) 객체를 반환한다.
- map.values() – 각 요소의 값을 모은 이터러블 객체를 반환한다.
- map.entries() – 요소의 [키, 값] 을 한 쌍으로 하는 이터러블 객체를 반환합니다. 이 이터러블 객체는 for..of 반복문의 기초로 쓰인다.

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);

for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}

for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

for (let entry of recipeMap) { // recipeMap.entries()와 동일
  alert(entry); // cucumber,500 ...
}
```

맵은 삽입 순서를 기억한다.

`맵` 은 값이 삽입된 순서대로 순회를 실시합니다. `객체는` 프로퍼티 순서를 기억하지 못한다.

여기에 더하여 `맵` 은 `배열` 과 유사하게 내장 메서드 `forEach` 도 지원한다.

```
// 각 (키, 값) 쌍을 대상으로 함수를 실행
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 ...
});
```

Object.entries: 객체를 맵으로 바꾸기

각 요소가 키-값 쌍인 배열이나 이터러블 객체를 초기화 용도로 `맵` 에 전달해 새로운 `맵` 을 만들 수 있습니다.

아래와 같이 말이죠.

```
// 각 요소가 [키, 값] 쌍인 배열
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

평범한 객체를 가지고 `맵` 을 만들고 싶다면 내장 메서드 Object.entries(obj)를 활용해야 합니다. 이 메서드는 객체의 키-값 쌍을 요소(`[key, value]`)로 가지는 배열을 반환합니다.

예시:

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

`Object.entries` 를 사용해 객체 `obj` 를 배열 `[ ["name","John"], ["age", 30] ]` 로 바꾸고, 이 배열을 이용해 새로운 `맵` 을 만들어보았습니다.

Object.fromEntries: 맵을 객체로 바꾸기

방금까진 `Object.entries(obj)` 를 사용해 평범한 객체를 `맵` 으로 바꾸는 방법에 대해 알아보았습니다.

이젠 이 반대인 맵을 객체로 바꾸는 방법에 대해 알아보겠습니다. `Object.fromEntries` 를 사용하면 가능합니다. 이 메서드는 각 요소가 `[키, 값]` 쌍인 배열을 객체로 바꿔줍니다.

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

`Object.fromEntries` 를 사용해 `맵` 을 객체로 바꿔봅시다.

자료가 `맵` 에 저장되어있는데, 서드파티 코드에서 자료를 객체형태로 넘겨받길 원할 때 이 방법을 사용할 수 있습니다.

예시:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // 맵을 일반 객체로 변환 (*)

// 맵이 객체가 되었습니다!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

`map.entries()` 를 호출하면 맵의 `[키, 값]` 을 요소로 가지는 이터러블을 반환합니다. `Object.fromEntries` 를 사용하기 위해 딱 맞는 형태이죠.

`(*)` 로 표시한 줄을 좀 더 짧게 줄이는 것도 가능합니다.

```
let obj = Object.fromEntries(map); // .entries()를 생략함
```

`Object.fromEntries` 는 인수로 이터러블 객체를 받기 때문에 짧게 줄인 코드도 이전 코드와 동일하게 동작합니다. 꼭 배열을 전달해줄 필요는 없습니다. 그리고 `map` 에서의 일반적인 반복은 `map.entries()` 를 사용했을 때와 같은 키-값 쌍을 반환합니다. 따라서 `map` 과 동일한 키-값을 가진 일반 객체를 얻게 됩니다.

Set

`셋(Set)` 은 중복을 허용하지 않는 값을 모아놓은 특별한 컬렉션입니다. 셋에 키가 없는 값이 저장됩니다.

주요 메서드는 다음과 같습니다.

- `new Set(iterable)` – 셋을 만듭니다. `이터러블` 객체를 전달받으면(대개 배열을 전달받음) 그 안의 값을 복사해 셋에 넣어줍니다.
- `set.add(value)` – 값을 추가하고 셋 자신을 반환합니다.
- `set.delete(value)` – 값을 제거합니다. 호출 시점에 셋 내에 값이 있어서 제거에 성공하면 `true`, 아니면 `false`를 반환합니다.
- `set.has(value)` – 셋 내에 값이 존재하면 `true`, 아니면 `false`를 반환합니다.
- `set.clear()` – 셋을 비웁니다.
- `set.size` – 셋에 몇 개의 값이 있는지 세줍니다.

셋 내에 동일한 값(value)이 있다면 `set.add(value)`을 아무리 많이 호출하더라도 아무런 반응이 없을 겁니다. 셋 내의 값에 중복이 없는 이유가 바로 이 때문이죠.

방문자 방명록을 만든다고 가정해 봅시다. 한 방문자가 여러 번 방문해도 방문자를 중복해서 기록하지 않겠다고 결정 내린 상황입니다. 즉, 한 방문자는 '단 한 번만 기록'되어야 합니다.

이때 적합한 자료구조가 바로 `셋` 입니다.

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// 어떤 고객(john, mary)은 여러 번 방문할 수 있습니다.
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// 셋에는 유일무이한 값만 저장됩니다.
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // // John, Pete, Mary 순으로 출력됩니다.
}
```

`셋` 대신 배열을 사용하여 방문자 정보를 저장한 후, 중복 값 여부는 배열 메서드인 arr.find를 이용해 확인할 수도 있습니다. 하지만 `arr.find`는 배열 내 요소 전체를 뒤져 중복 값을 찾기 때문에, 셋보다 성능 면에서 떨어집니다. 반면, `셋`은 값의 유일무이함을 확인하는데 최적화되어있습니다.

셋의 값에 반복 작업하기

`for..of` 나 `forEach`를 사용하면 셋의 값을 대상으로 반복 작업을 수행할 수 있습니다.

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// forEach를 사용해도 동일하게 동작합니다.
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

흥미로운 점이 눈에 띄네요. `forEach` 에 쓰인 콜백 함수는 세 개의 인수를 받는데, 첫 번째는 `값` , 두 번째도 *같은 값*인 `valueAgain` 을 받고 있습니다. 세 번째는 목표하는 객체(셋)이고요. 동일한 값이 인수에 두 번 등장하고 있습니다.

이렇게 구현된 이유는 `맵` 과의 호환성 때문입니다. `맵` 의 `forEach` 에 쓰인 콜백이 세 개의 인수를 받을 때를 위해서죠. 이상해 보이겠지만 이렇게 구현해 놓았기 때문에 `맵` 을 `셋` 으로 혹은 `셋` 을 `맵` 으로 교체하기가 쉽습니다.

`셋` 에도 `맵` 과 마찬가지로 반복 작업을 위한 메서드가 있습니다.

- `set.keys()` – 셋 내의 모든 값을 포함하는 이터러블 객체를 반환합니다.
- `set.values()` – `set.keys` 와 동일한 작업을 합니다. `맵` 과의 호환성을 위해 만들어진 메서드입니다.
- `set.entries()` – 셋 내의 각 값을 이용해 만든 `[value, value]` 배열을 포함하는 이터러블 객체를 반환합니다. `맵` 과의 호환성을 위해 만들어졌습니다.

요약

`맵` 은 키가 있는 값이 저장된 컬렉션입니다.

주요 메서드와 프로퍼티:

- `new Map([iterable])` – 맵을 만듭니다. `[key,value]` 쌍이 있는 `iterable` (예: 배열)을 선택적으로 넘길 수 있는데, 이때 넘긴 이터러블 객체는 맵 초기화에 사용됩니다.
- `map.set(key, value)` – 키를 이용해 값을 저장합니다.
- `map.get(key)` – 키에 해당하는 값을 반환합니다. `key` 가 존재하지 않으면 `undefined` 를 반환합니다.
- `map.has(key)` – 키가 있으면 `true` , 없으면 `false` 를 반환합니다.
- `map.delete(key)` – 키에 해당하는 값을 삭제합니다.
- `map.clear()` – 맵 안의 모든 요소를 제거합니다.
- `map.size` – 요소의 개수를 반환합니다.

일반적인 `객체` 와의 차이점:

- 키의 타입에 제약이 없습니다. 객체도 키가 될 수 있습니다.
- `size` 프로퍼티 등의 유용한 메서드나 프로퍼티가 있습니다.

`셋` 은 중복이 없는 값을 저장할 때 쓰이는 컬렉션입니다.

주요 메서드와 프로퍼티:

- `new Set([iterable])` – 셋을 만듭니다. `iterable` 객체를 선택적으로 전달받을 수 있는데(대개 배열을 전달받음), 이터러블 객체 안의 요소는 셋을 초기화하는데 쓰입니다.

- `set.add(value)` – 값을 추가하고 셋 자신을 반환합니다. 셋 내에 이미 `value` 가 있는 경우 아무런 작업을 하지 않습니다.
- `set.delete(value)` – 값을 제거합니다. 호출 시점에 셋 내에 값이 있어서 제거에 성공하면 `true` , 아니면 `false` 를 반환합니다.
- `set.has(value)` – 셋 내에 값이 존재하면 `true` , 아니면 `false` 를 반환합니다.
- `set.clear()` – 셋을 비웁니다.
- `set.size` – 셋에 몇 개의 값이 있는지 세줍니다.

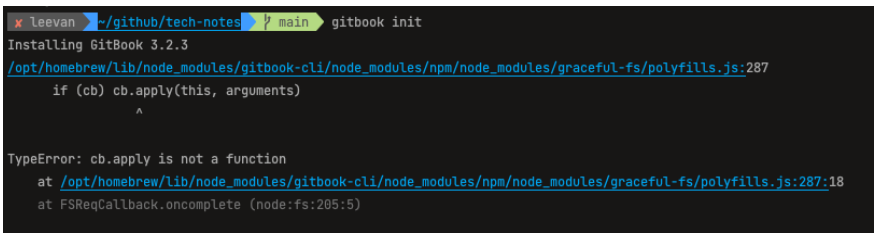
`맵` 과 `셋` 에 반복 작업을 할 땐, 해당 컬렉션에 요소나 값을 추가한 순서대로 반복 작업이 수행됩니다. 따라서 이 두 컬렉션은 정렬이 되어있지 않다고 할 수 없습니다. 그렇지만 컬렉션 내 요소나 값을 재 정렬하거나 (배열에서 인덱스를 이용해 요소를 가져오는 것처럼) 숫자를 이용해 특정 요소나 값을 가지고 오는 것은 불가능합니다.

React

GitBook

설치 방법

```
# 설치
$ npm install gitbook-cli -g
# 생성
$ gitbook init
# 서버 실행
$ gitbook serve
```



graceful-fs error

```
# 설치중 에러 발생시 아래와 같이 4.2.0 버전으로 매뉴얼 설치한다.
cd /opt/homebrew/lib/node_modules/gitbook-cli/node_modules/npm/node_modules &&
```

book.json 파일 생성

```
{
  "plugins": [
    "disqus",
    "theme-api",
    "hide-published-with",
    "multipart",
    "collapsible-chapters"
  ],
  "pluginsConfig": {
    "disqus": {
      "shortName": "baboleevan"
    },
    "theme-api": {
      "theme": "dark"
    }
  }
}
```

```
# E-book을 만들어주는 소프트웨어
$ brew install --cask calibre

# 플러그인 설치
$ gitbook install
```

용어사전

용어에 대한 전발적인 개념만 명시하고, 깊은 내용은 Tech Notes에 정리한다.

마일스톤 (Mailestone)

마일스톤이란 프로젝트 진행 과정에서 특정할 만한 건이나 표를 말한다. \ 예를 들어, 프로젝트 계약, 착수, 인력투입, 선금 수령, 중간보고, 감리, 종료, 잔금 수령 등 프로젝트 성공을 위해 반드시 거쳐야 하는 중요한 지점을 말한다.

마일스톤은 프로젝트 일정관리를 위해 반드시 필요한 지점을 체크하기 위해 사용한다. \ 프로젝트 성공을 위해 필수적인 사항들을 각 단계별로 체크함으로써 전체적인 일정이 늦춰지지 않고 제 시간 안에 과업이 종료될 수 있도록 관리하는데 도움을 준다.

다만, 마일스톤은 프로젝트 진행 과정에서 결정적으로 중요한 핵심적인 사항들만 체크하기 때문에, 그다지 중요하지는 않더라도 프로젝트 진행에 꼭 필요한 다양한 요소들을 상세하게 파악하기 힘들다는 단점이 있다.

제안요청서 (RFP : Request for proposal)

발주자가 특정 과제의 수행에 필요한 요구사항을 체계적으로 정리하여 제시함으로써 제안자가 제안서를 작성하는데 도움을 주기 위한 문서

우선 본인이 원하는 것이 무엇인지 구체화해서 나열하고, 그 결과로 얻을 수 있는 내용들(기대효과)과 작업에 필요한 기간도 산정해야 할 것이다. 무엇보다도 적정한 예산계획을 수립해야 할 것이다. 예산이 너무 적다면, 낮은 품질의 결과물이 나올 확률이 높기 때문이다. 이러한 내용을 정리하며 체계화된 문서로 정리한다면 RFP 된다. 즉, RFP는 "수요자"가 특정 과제의 수행에 필요한 내용을 체계적으로 명시한 문서이다.

이렇게 수요기관이 RFP(제안요청서)를 공고하면 공급자(사업자)는 RFP의 내용을 토대로 "제안서"를 작성한다.

"내가 원하는 것을 이렇게 이렇게 만들어 주세요"라고 운을 띄우는 것이 제안요청서(RFP)라면, \ "네, 내용을 봤는데 이렇게 하는 것은 어떻까요?"라고 제시하는 것이 "제안서"의 역할이다.

기본적으로 RFP에는 해당 업무(프로젝트)에 대한 자세한 정보, 추진 일정, 예산, 그리고 제안서의 목차, 제안 평가 기준 등을 구체적으로 명시해야 한다. \ RFP를 얼마나 체계적으로 작성했는가에 따라 제안서의 품질이 결정되고, 업무를 의뢰한 업체와 외주업체 간의 의견 충돌을 미연에 예방할 수 있기 때문에 RFP 작성은 해당 프로젝트의 성공 여부를 결정하는 가장 중요한 단계라고 할 수 있다.

모멘텀 (Momentum)

모멘텀은 본래 물리학 용어로 동력을 말하며, 추진력·여세·타성이라고 한다. \ \ '추진을 위한 동력'

기하학에서는 곡선 위에 있는 한 점의 기울기를 나타내며, 경제학에서는 **한계변화율**을 뜻한다. \ \ 특히 주식시장에서 주가가 상승추세를 형성했을 경우, 얼마나 가속을 붙여 움직일 수 있는지를 나타내는 지표가 된다.\ 주가를 움직일 수 있는 자극이 있느냐를 나타내는데, 예를 들면 증자발표나 신사업 진출, 액면분할, 정부의 정책발표 등을 말한다. 주가가 상승하더라도 이 수치가 부족하면 향후 상승추세가 꺾여 하락할 가능성이 크고, 반대로 주가가 하락하더라도 이 수치가 높으면 주가는 상승할 가능성이 크다. 따라서 주가의 변동을 알아내는 기준이 되며, 특히 단기투자에서 유용하게 사용된다. 사전적 의미로는 '(일의 진행에 있어서의) 탄력, 가속도, 운동량(물체의 질량과 속도의 곱으로 나타내는 물리량)'을 말한다.

순익분기점 (BEP)

손익분기점(B.E.P)은 일정한 기간의 매출액과 그것을 위하여 지출된 총 비용이 일치되는 매출액을 말한다. \ 따라서 그 점 이상으로 매출을 올리게 되면 점차 이익이 발생하고, 반면 기 이하의 매출이 나타나면 손실이 발생하게 된다.

손익 분기점은 손해와 이익의 경계점으로 손해도 이익도 나지 않는 지점을 의미한다. 손익분기점 보다 많은 매출이 일어나면 이익이 생기지만, 그 보다 적게 팔면 손해를 보게 된다. 손익분기점이 100만원이라면, 매출이 100만원 이상일때 이익이 창출되고, 100만원 이하이면 손해를 보게 되는 것이다.

- 손익 분기점 = 고정비 / (1-(변동비/ 매출액))
- 변동비 : 매출액이 증가함에 따라 변동하는 비용으로 직접 재료비, 원가 등이 해당 된다
- 고정비 : 매출액의 변동에 관계없이 고정적으로 발생하는 비용으로 임대료, 감가상각비, 관리비, 이자비용 등을 의미 한다 월 평균 매출이 1,200 만원이고 고정비 550만원, 변동비 430만원인 점포의 손익분기점을 한번 계산해 보자. 550만원 / (1-(430만원 / 1200만원)) = 859만원으로 이점포의 손익 분기점은 859만원이다. 따라서 859만원 이상 부터 이익이 발생하기 시작한다. 따라서 손익분기점을 계산 할 때 가장 유의해야 할 점은 변동비와 고정비를 적용할 때 최소비용 보다는 예측할 수 있는 최대비용에 가깝게 적용하는 것이 바람직하다는 것이다.

리텐션 (Retention)

리텐션이란 한번 획득한 유저들이 서비스를 이탈하지 않고 계속 서비스를 이용하는 것을 의미한다.

리텐션이 높은 서비스는 리텐션이 낮은 서비스보다 획득비용에 투자한 비용을 빠르게 회수할 수 있으며 이렇게 회수한 비용으로 다시금 빠르게 획득에 투자할 수 있도록 하여 성장을 촉진한다.\ \ 리텐션이 낮다는 것은 획득 이후 다시 돌아오지 않고 이탈하는 유저가 많다는 뜻으로, 리텐션이 낮은 서비스는 성장이 더딜 뿐만 아니라 한번 획득했다가 이탈한 유저는 다시 획득하기에도 더 많은 비용과 시간이 소요되므로 악순환을 만들어낸다. 따라서 리텐션은 사업 성장에 있어서 반드시 지켜보아야 할 지표 중 하나이다.

TIL (Today I Learned)

그날 그날 본인이 공부한 것을 정리하는 것을 말한다.

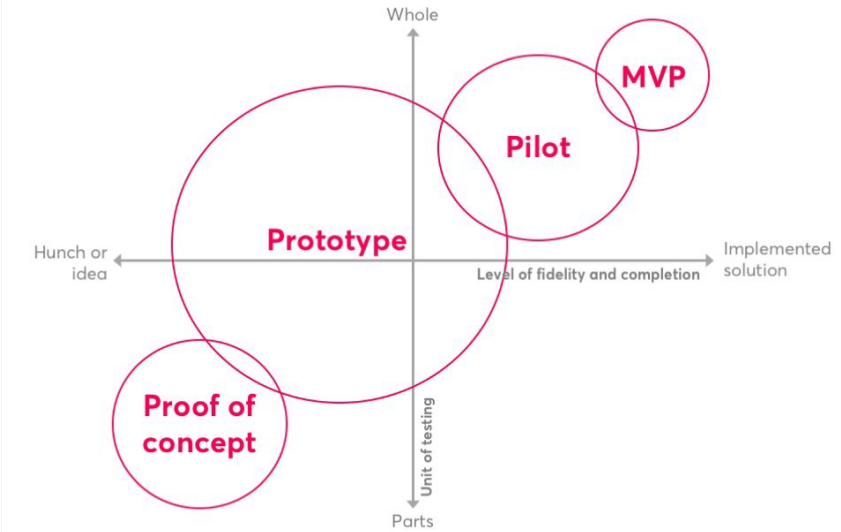
MVP (Minimum Viable Product)

최소 기능 제품(Minimum Viable Product, MVP)는 고객의 피드백을 받아 최소한의 기능(features)을 구현한 제품이다.

MVP와 PoC, Prototype, Pilot 차이

MVP와 PoC, Prototype, Pilot 차이

Agile, Lean Startup의 MVP(최소 실행가능 제품, Minimum Viable Product를 이해하기 위해 PoC(개념증명), Prototype(프로토타입), Pilot(시범적용)과 비교해보겠습니다. 이들 모두는 검증이라는 공통점이 있습니다. 특히 MVP는 제품을 완전히 개발하지 않고도 제품에 대한 고객의 관심을 초기에 이해하고 개선/검증합니다. 따라서 고객의 반응에 따라 방향성을 개선 할 수 있으며, 시장에서 성공하지 못할 제품에 대한 시간, 노력, 비용을 줄일 수 있습니다.



1. 설명

PoC(개념 증명, Proof of Concept)

개념 증명(POC, Proof of Concept)은 (기존 시장에 없었던) **신기술을 도입하기 전에 이를 검증하기 위해 사용**한다. 특정 방식이나 아이디어를 실체화하여 타당성을 증명하는 것을 의미한다.

- 의미 : 새로 도입되는 기술이나 제품이 도입 목적에 부합되는지 검증
- 검증 : 그 기술이 생각한대로 동작 돼? 도입해도 돼?
- MVP와 차이점 : 실사용자인 고객이 사용/피드백 안함

Prototype(프로토타입)

프로토타입(prototype)은 시제품이 나오기 전의 제품의 원형이며, 개발검증과 양산 검증을 거쳐야 시제품이 될 수 있다. **프로토타입은 정보시스템의 미완성 버전 또는 중요한 기능들이 포함되어 있는 시스템의 초기모델**이다.

프로토타입은 사용자의 모든 요구사항이 정확하게 반영할 때까지 계속해서 개선/보완 된다. 실제로 많은 애플리케이션들이 지속적인 프로토타입의 확장과 보강을 통해 최종 승인되고 개발에 들어간다.

- 의미 : 시스템이나 제품들의 중요한 기능들이 포함된 초기모델(SW 개발 착수전 검증/승인)
- 검증 : 이렇게 설계하여 개발/양산(생산)해도 돼?
- MVP와 차이점 : 실사용자인 고객이 사용/피드백 안할 수도 있음

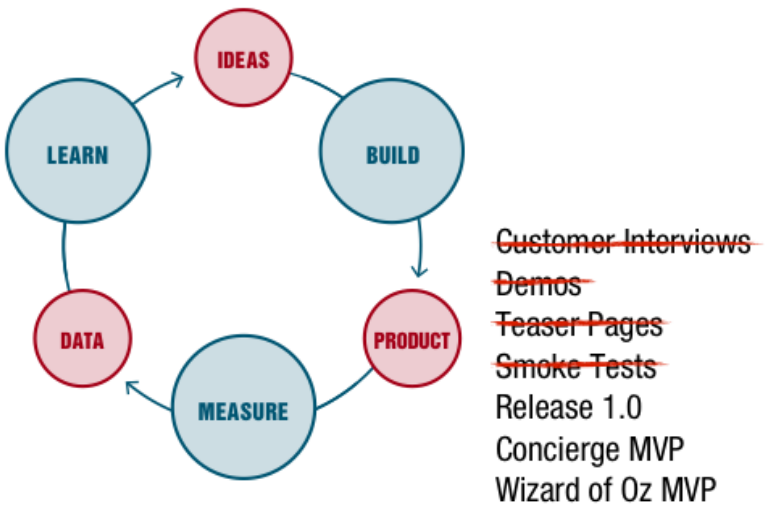
Pilot (시범 적용)

Pilot은 전체 확대 적용하기전에 소규모로 테스트해서 추후 발생할 수 있는 여러 문제의 원인을 미리 파악하고 수정 보완하기 위해 모의로 시행해 보는 활동입니다. 종종 새로운 공공 정책 또는 서비스 출시 전 사용됩니다.

- 의미 : 새로운 정책 또는 서비스 확산의 첫 번째 시범 적용
- 검증 : 더 넓은 그룹으로 확장해서 적용해도 돼?
- MVP와 차이점 : 실사용자인 고객이 사용/피드백은 유사하나 성공 또는 실패를 확인하기 위해 사용하며 지속적으로 검증하며 사용하지는 않음

MVP(최소 실행가능 제품, Minimum Viable Product)

최소 실행 가능 제품(Minimum Viable Product, MVP)은 고객에게 가치를 제공해야 하며, 고객 피드백을 받아 **생존하기 위한** 최소한의 노력을 들여 만든 기능(features)을 구현한 제품이다. 최소 실행 가능 제품(MVP)은 신제품 개발에서 학습의 영향을 강조하는 Lean Startup의 개념이다.



Eric Ries는 MVP를 최소한의 노력으로 고객에 대한 검증 된 정보를 최대한 수집 할 수있는 새로운 제품 버전으로 정의했다. 사람들이 제품과 관련하여 실제로 무엇을하는지 보는 것이 사람들에게 무엇을 원하는지 묻는 것보다 훨씬 더 신뢰할 수 있다.

- 의미 : 출시 후 제품이 생존하기 위해 최소한의 노력(개발범위, 시간)으로 고객에 대한 검증 된 정보를 최대한 수집 할 수있는 제품 버전
- 검증 : 고객에게 Value를 주는가? 고객이 원하는 것이 맞아? 가설대로 고객이 변화 돼?
- PoC, Prototype과 차이점 : 실사용자인 고객이 사용/피드백 해야 함
- Pilot과 차이점 : 성공/실패 확인 보다는 지속적인 실험/개선 목적
- MVP는 최소한의 노력으로 고객에게 Value를 주며 가설을 검증하며 개선

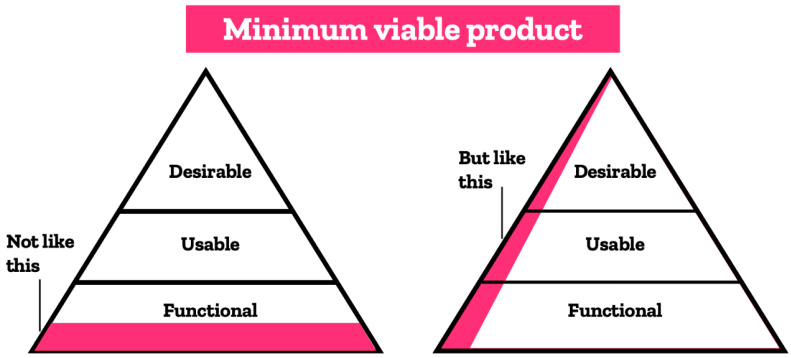
2. MVP에 대한 부연 설명

MVP의 주요 이점은 제품을 완전한 전체를 개발하지 않고도 제품에 대한 고객의 관심을 이해하고 제품에 대한 의사결정을 할 수 있습니다. 따라서 시장에서 성공하지 못할 제품에 대한 노력, 시간, 비용이 줄어 듭니다. (극단적으로는 10억으로 망할 Product를 5억만 쓰고 망할 수 있습니다.)

또한 PoC나 Prototype은 일부 버그나 완성도가 낮을 수도 있지만, MVP는 시장에 출시 및 고객 검증 받을 수 있는 수준의 완성도는 필요합니다. 즉 전체 기능 완성이 아닌 품질은 완성되어야 합니다. (만약 정상 동작하지 않는 홈쇼핑 상품을 본다면 개선을 기대하고 구매하겠습니까?)

개인적으로 Dropbox의 MVP(영상)는 최소라는 목적에는 부합되지만 실제 존재하지 않는 광고 수준의 베이퍼웨어이므로 일반적 MVP 예에서는 논외라고 생각됩니다.

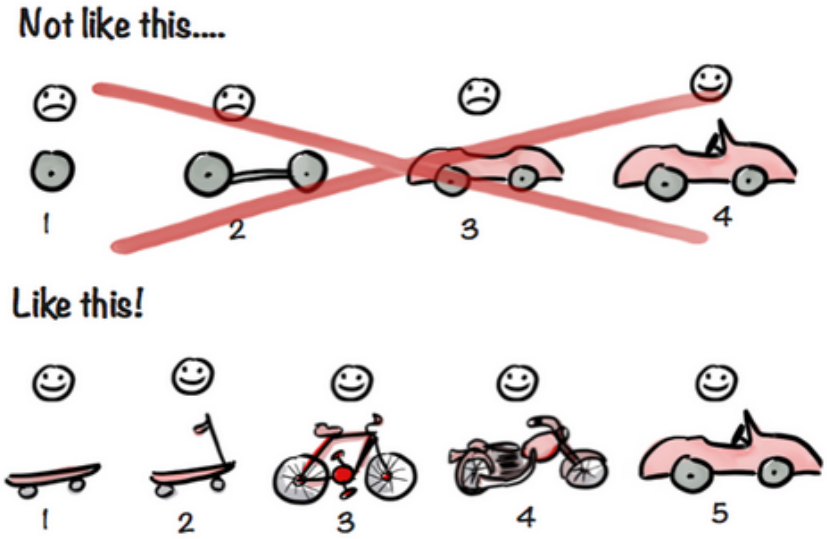
결론적으로, 4가지 모두 검증이라는 공통점은 있지만, 단순한 업무 결과물(Output)을 넘어 MVP가 좀 더우리의 가설대로 고객이 실제 그렇게 변화되는지 성과(Outcome) 검증에 더 효과적이라고 생각됩니다.



3. 접근 방식의 예

3.1. MVP

절대로 자동차를 만들기위해 킥보드, 자전거, 오토바이, 자동차를 순차적으로 만들라는 의미가 아닙니다. MVP는 고객의 피드백을 받아 **생존하기 위한** 최소한의 기능(features)을 구현한 제품입니다. 특히 아래 그림은 동작(실제 사용해 볼 수 있는)되는 결과물을 강조한 것입니다.



Like this의 MVP Approach는 “사용자가 효율적으로 이동 할 수 있는 Value를 제공”하기 위한 과정을 소개하고 있습니다. MVP는 고객에게 실현(실행) 가능한 Value를 지속적으로 제공하기 위한 가설/검증하며 제품을 만들어 나갑니다.(부연 : 강의때 아래 그림만 보면 오해하시는 분들이 많습니다. 고객에게 이동 편의성 같은 가치를 줘야 한다는 의미입니다.)

- 가설/검증1 : 바퀴가 있으면 사용자에게 Value를 제공할 것이다.
- 가설/검증2 : 손잡이가 있으면 사용자에게 더 큰 Value를 제공할 것이다.
- 가설/검증3 : 바퀴가 크면 사용자에게 더 큰 Value를 제공할 것이다.
- 가설/검증4 : 모터가 있으면 사용자에게 더 큰 Value를 제공할 것이다.
- 가설/검증5 : 바퀴가 4개면 사용자에게 더 큰 Value를 제공할 것이다.

물론 집에서 가까운 지하철역까지 이동하기 위해서는 자동차보다 공유 킥보드나 자전거가 더 큰 Value를 줄 수도 있습니다. 이 처럼 상황에 따라서 전략과 Outcome이 달라질 수 있습니다.

3.2. 다른 용어들

- 모터를 킥보드에 탑재해서 전동 킥보드가 되는지 PoC해보겠습니다.
- 이것이 우리 전동킥보드의 Prototype입니다.
- 이 지역에 우리 공유 전동 킥보드가 효율적일지 Pilot 적용하겠습니다.
- 우리의 가설(전동 킥보드에 추가 feature)이 실제 사용자의 Needs를 충족하는지 반복적으로 실험, 검증하며 개선하겠습니다.

4. MVP 용어를 잘못 사용하는 경우

최근에 MVP라는 용어를 주변에서 많이 사용 하지만, 내부 데모용 초기 버전을 MVP로 칭하는 경우가 있습니다.

4.1. 순차적 오픈

서브 시스템 10개중 우선 4개를 1단계 프로젝트로 개발하고 우선 오픈하는 순차적 오픈을 MVP라고 부르기\ (부연 : 프로젝트 계획서에는 1단계 프로젝트 종료 후 개선없이 바로 2단계 프로젝트 시작)

- 사용자에게 검증/개선하기 위해 4개를 먼저 개발/오픈했다면 MVP가 맞음
- 만약, 4개를 오픈하고 검증/개선 계획 없이 바로 나머지를 개발한다면 MVP가 아닌 순차적 오픈임
- 즉, 원래 일정대로 가는 과정일 뿐 가설을 검증/개선하기 위한 목적이 아님

4.2. 베타 버전

아직은 시장에 출시할 수준이 되지 않은 초기 버전을 MVP라고 부르기\ (부연:알파,베타, 사용성 테스트처럼 출시전 문제확인을 위한 테스트 목적임)

- 검증 목적으로 기능/컨셉을 시장에 출시하여 사용자의 Feedback을 받기 위한 목적이라면 MVP가 맞음
- 만약, 제품이 출시되기 전에 “내부인”이 아닌 “외부인”에게 정식 배포전에 테스트와 오류 수정을 위한 버전이라면 베타 버전임
- 즉, 가설을 검증하는 목적보다는 출시전 품질을 높이는 용도

4.3. Demo용

막연하게 제3자에게 보여주기 위한 용도를 MVP라고 부르기

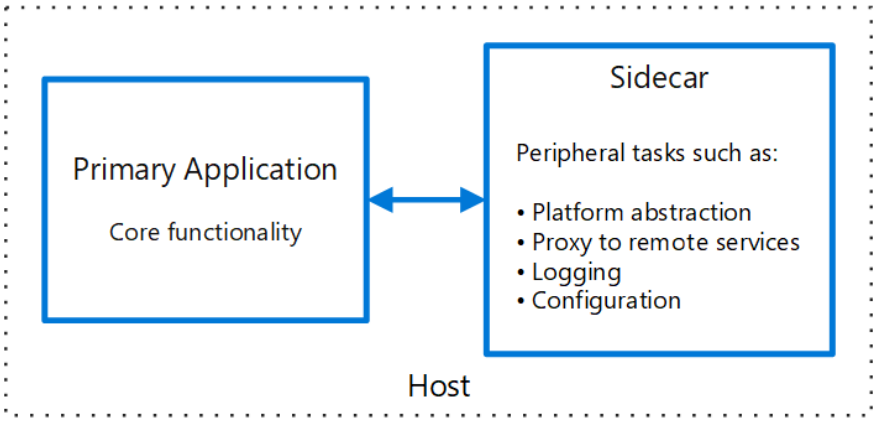
- PoC, Prototype, Pilot, MVP 모두 제3자에게 demo 할 수 있음

참고자료

- PoC : https://ko.wikipedia.org/wiki/%EA%B0%9C%EB%85%90_%EC%A6%9D%EB%AA%85
- Prototype : https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%ED%86%A0%ED%83%80%EC%9E%85
- MVP : https://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html
- Vaporware(**베이퍼웨어)** : https://ko.wikipedia.org/wiki/%EB%B2%A0%EC%9D%B4%ED%8D%BC%EC%9B%A8%EC%96%B4
- Dropbox MVP 동영상 (실제 제품은 없고 컨셉으로 증명 후 투자 받음) https://youtu.be/xy9nSnalvPc

원문링크 : https://medium.com/dtevangelist/mvp%EC%99%80-poc-prototype-pilot-%EC%B0%A8%EC%9D%B4-1f525cc4a218

사이드카 패턴 (Sidecar Pattern)



일반적으로 사이드카 패턴은 다음과 같이 구성된다. 사이드카 서비스는 어플리케이션의 일부일 필요는 없으며 애플리케이션에 연결되어 있다. 사이드카는 기본 어플리케이션을 사용해서 배포되는 프로세스 또는 서비스를 지원한다.

예시

- 정적 파일 컨텐츠를 처리하도록 NGINX Proxy를 node.js 서비스 인스턴스 앞에 배치한다.
- 보안을 위해 사이드카로 NGINX reverse proxy 등을 붙여서 HTTPS 통신을 한다.
- 성능을 위해 사이드카로 NGINX content cache 등을 붙인다.
- 컨테이너 외부로 로그를 모으기 위해 logstash, fluentd 등을 붙인다. (centralized logging)

장점

- 상호 의존성을 줄일 수 있다.
- 사이드카 장애 시 어플리케이션이 영향을 받지 않는다. (Isolation)
- 사이드카 적용/변경/제거 등의 경우에 어플리케이션은 수정이 필요 없다.
- 어플리케이션과 사이드카를 다른 언어로 만들 수 있다.
- 대부분 같은 스토리지를 공유할 수 있기 때문에 공유에 대한 고민이 적다.

단점

- 어플리케이션이 너무 작은 경우 배 보다 배꼽이 커질 수 있다.
- 프로세스간 통신이 많고 최적화 해야 한다면, 어플리케이션에서 함께 처리하는게 좋을 수 있다.

참고링크

- The Sidecar Pattern
- Sidecar pattern - Azure docs
- How Pods manage multiple Containers - kubernetes Pod Overview doc
- Deploying an NGINX Reverse Proxy Sidecar Container on Amazon ECS
- Centralized logging in Kubernetes

원문링크 \ https://blog.leocat.kr/notes/2019/02/16/cloud-sidecar-pattern\
https://azderica.github.io/00-design-pattern-sidecar/

Front-end

- **Defensive CSS** : 사이즈가 달라져도 깨지지 않도록 Flexbox나 백그라운드 이미지, 스크롤 등 CSS 팁을 정리한 사이트로 상황별로 어떤 문제가 있는지 정리하고 직접 재현해 볼 수 있게 예제도 제공하고 있다.(영어)
- **Satori** : HTML과 CSS를 SVG로 변환하는 Vercel에서 만든 라이브러리.
- **Slash libraries** ** : Toss에서 사용하는 TypeScript/JavaScript 패키지의 모음으로 한글 처리 등의 유틸리티나 React 관련 라이브러리가 포함되어 있다.
- **Cloudscape Design System** : AWS에서 만든 클라우드용 디자인 시스템으로 AWS 웹 콘솔에서 볼만한 UI가 컴포넌트로 정리되어 있다.
- **qwik** : 즉각적으로 웹 애플리케이션을 제공할 수 있도록 지원하는 웹 프레임워크
- **[용량을 줄인 NotoSans 및 NotoSerif](https://akngs.github.io/noto-kr-vf-distilled/)**
- **SUIT Font** : 본고딕을 기반으로 한 오픈소스 폰트
- **Introducing Signals** : React의 호환 라이브러리인 Preact에서 상태를 다루기 쉽게 해주는 Signals를 공개했다. 컴포넌트 안이든 밖이든 사용할 수 있고 hook이나 클래스 컴포넌트와도 잘 동작하는데 상태 관리보다 빠른 방법을 찾아서 Signals를 만들게 되었다고 한다. 가상 DOM을 사용한 방식보다 훨씬 빠르다고 하는데 Signals가 가지는 의미는 김태곤 님이 작성하신 Signals: React의 그늘에서 벗어나는 Preact?도 참고해보면 좋다.(영어)

정리중

- **Kodiak** : GitHub의 Pull Reqeust의 자동화를 도와주는 앱.
- **Bird Eats Bug** : 브라우저와 연동해서 문제상황을 개발자 도구와 함께 녹화해서 버그 리포팅을 할 수 있게 해주는 도구. 서지연 님이 작성한 소개 글 참고.
- **GraphQL Hive** : GraphQL 스키마 레지스트리 및 오퍼레이션 모니터링 시스템.
- **Nextra** : Next.js와 MDX를 이용한 정적 사이트 생성기.
- **tRPC** : TypeScript에서 서버 쪽 라우터의 코드를 클라이언트에서 임포트해서 클라이언트의 API 타입세이프를 맞출 수 있게 해주는 프로젝트.
- **Blitz** : Next.js용 풀스택 툴킷.
- **Code Catalog** : 유명 오픈소스의 설명을 추가해서 코드를 읽기 도와주는 사이트로 해당 프로젝트의 중요 부분을 설명하는 형식으로 되어 있다.