

학습 내용 정리 - 2월 4주차

박시준

Table of content

- Algorithm
 - Algorithm – DFS by stack
 - Algorithm – Dijkstra + iteration
 - Algorithm – BFS + DP
- Micro Service Architecture
 - API gateway
 - Message Queue
- AWS
 - AWS – ELB
 - AWS – Load balancer vs API gateway and about VPC
 - AWS – EC2, Fargate, Lambda

Algorithm – DFS with stack

[프로그래머스] 여행경로(DFS)

여행경로

문제 설명

주어진 항공권을 모두 이용하여 여행경로를 짜려고 합니다. 항상 "ICN" 공항에서 출발합니다.

항공권 정보가 담긴 2차원 배열 tickets가 매개변수로 주어질 때, 방문하는 공항 경로를 배열에 담아 return 하도록 solution 함수를 작성해주세요.

입출력 예

tickets	return
[["ICN", "JFK"], ["HND", "IAD"], ["JFK", "HND"]]	["ICN", "JFK", "HND", "IAD"]
[["ICN", "SFO"], ["ICN", "ATL"], ["SFO", "ATL"], ["ATL", "ICN"], ["ATL", "SFO"]]	["ICN", "ATL", "ICN", "SFO", "ATL", "SFO"]

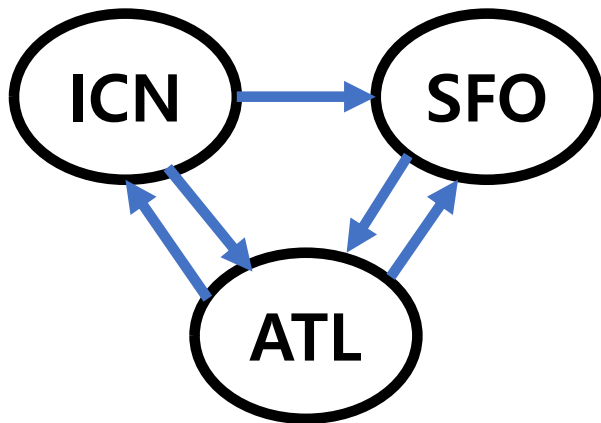
- DFS 문제
- 주어진 배열은 항공권을 의미, 출발지와 도착지가 기재되어 있음
- 모든 항공권(배열)을 사용해서 이동해야 함
- 가능한 경로를 반환.
- 가능한 경로가 여러 개면 알파벳 순으로 가장 앞서는 경로를 반환한다.

[프로그래머스] 여행경로(DFS)

입출력 예

tickets	return
[["ICN", "JFK"], ["HND", "IAD"], ["JFK", "HND"]]	[["ICN", "JFK", "HND", "IAD"]]
[["ICN", "SFO"], ["ICN", "ATL"], ["SFO", "ATL"], ["ATL", "ICN"], ["ATL", "SFO"]]	[["ICN", "ATL", "ICN", "SFO", "ATL", "SFO"]]

예시 2번 해석

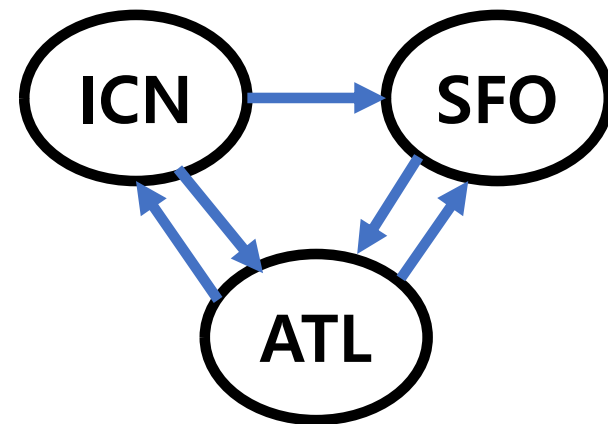


- 세 가지 경우의 수 발생.
 - ICN -> SFO -> ATL -> ICN -> ATL -> SFO
 - ICN -> ATL -> SFO -> ATL -> ICN -> SFO
 - ICN -> ATL -> ICN -> SFO -> ATL -> SFO
-
- 이 중에서 알파벳 순으로 앞서는 ICN -> ATL -> ICN -> SFO -> ATL -> SFO 경로가 정답

1차 시도 – hash+ backtracking(DFS)

- 일단 hash 사용해서 초기화하는 게 유리하다. 그래야 출발지 입력했을 때 도착지를 배열 형태로 꺼내 쓸 수 있음
- 백트래킹을 쓰기 위해 visited도 hash 형태로 만들어줬다. 같은 노드를 여러 번 방문할 수 있으므로 배열이 아닌 hash를 사용해서 방문 지점을 기억해야한다. 그래야 같은 노드를 몇 번째 방문하는 건데 기억할 수 있음

```
from collections import defaultdict
def solution(tickets)::
    # initialization
    answer = []
    candidates = []
    path = ["ICN"]
    routes = defaultdict(list)
    visited = defaultdict(list)
    for ticket in tickets:
        routes[ticket[0]].append(ticket[1])
        visited[ticket[0]].append(False)
```



1차 시도 – hash+ backtracking(DFS)

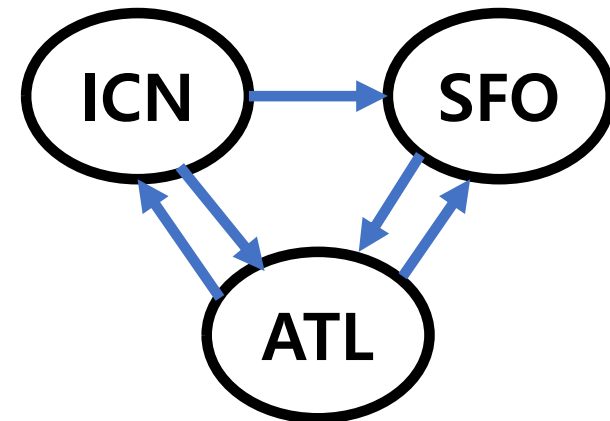
- 미리 알파벳으로 정렬하고 들어간다.
- 알파벳 순으로 정렬하면 모든 항공권을 사용했을 때 가장 먼저 나오는 경로가 정답인 경로가 된다.

```
# sorted by alphabet
for departure in routes.keys():
    routes[departure].sort()
print(routes)
```

- 주어진 항공권은 모두 사용해야 합니다.
- 만일 가능한 경로가 2개 이상일 경우 알파벳 순서가 앞서는 경로를 return 합니다.

[주어진 티켓으로 만들 수 있는 경로]

ICN -> SFO -> ATL -> ICN -> ATL -> SFO
ICN -> ATL -> SFO -> ATL -> ICN -> SFO
ICN -> ATL -> ICN -> SFO -> ATL -> SFO



1차 시도 – hash+ backtracking(DFS)

- 백트래킹을 해서 조건을 충족시켰을 때(path의 길이가 모든 항공권을 썼을 때 만들 수 있는 path의 길이 일 때) return 하도록 만든다.

```
#dfs loop
def dfs(departure, path):
    if len(path) == len(tickets) + 1:
        candidates.append(list(path))
        return

    for i in range(len(routes[departure])):
        arrival = routes[departure][i]
        if visited[departure][i] == False:
            path.append(arrival)
            visited[departure][i] = True
            dfs(arrival, path)
            path.pop()
            visited[departure][i] = False

dfs("ICN", path)
```


1차 시도 - hash + backtracking(DFS)

- 이 방법에는 문제가 있음.
- 앞서 Sort를 했기 때문에 제일 처음 나오는 경로가 정답 경로라서 나머지 경로는 탐색 안 해도 된다.
- 근데 현재의 백트래킹 방식에서는 나머지 모든 경로도 탐색을 진행한다.
- 계산할 필요도 없는 경우의 수까지 계산하므로 자원 낭비.
- 만약 이 문제가 항공권을 전부 사용하는 모든 경로를 구하는 것이었다면 본 풀이가 맞다. (하지만 그러면 sort도 필요 없음)

```
['ICN', 'ATL', 'ICN', 'SFO', 'ATL', 'SFO']  
['ICN', 'ATL', 'SFO']  
['ICN', 'ATL', 'SFO', 'ATL']  
['ICN', 'ATL', 'SFO', 'ATL', 'ICN']  
['ICN', 'ATL', 'SFO', 'ATL', 'ICN', 'SFO']  
['ICN', 'SFO']  
['ICN', 'SFO', 'ATL']  
['ICN', 'SFO', 'ATL', 'ICN']  
['ICN', 'SFO', 'ATL', 'ICN', 'ATL']  
['ICN', 'SFO', 'ATL', 'ICN', 'ATL', 'SFO']
```

너무 많은 시간을 소요한다.

정확성	테스트	
	테스트 1	통과 (215.19ms, 14.4MB)
	테스트 2	통과 (0.01ms, 10.2MB)
	테스트 3	통과 (0.02ms, 10.2MB)
	테스트 4	통과 (0.02ms, 10.2MB)

2차 시도 – hash + stack + dic.pop()

(1) 반복문 + stack으로 구현

- 재귀 함수로 구현된 DFS는 모든 경로를 탐색하게 된다. 문제는 이렇게 재귀함수로 구현할 경우 정답을 구했을 때, 빠져나오기가 마땅치 않음. (return을 사용해도 다른 경로 탐색을 진행함)
- DFS는 재귀함수 뿐만 아니라 반복문과 stack을 섞어서 구현할 수 있다. 재귀함수보다 유리한 부분은 반복문은 답안 하나 구했으면, break 사용해서 빠져나올 수 있다. 다른 경로 탐색 안 해도 됨.

(2) Dictionary의 pop()

- 이미 사용한 노드를 체크하는 것에 Visited hash를 사용하는 것 보다 그냥 routes hash에서 이미 방문한 목적지 노드를 제거해버리는 게 낫다. (어차피 다시는 볼 일 없음)
- **Dictionary의 값도 pop()할 수 있다는 점을 이용한다.**

2차 시도 - hash + stack + dic.pop()

- sorting을 할 때 역순으로 해야 한다. 그래야 routes에서 pop()을 할 때 알파벳 정순으로 들어간다.
- Pop()한 것을 stack에 집어 넣으면 stack에는 정순으로 값이 들어온다. 이것을 출발지로 사용한다.
- 다음 목적지가 없으면 pop해서 answer 경로에 포함시킨다.
- 다음 목적지가 있으면 그 목적지를 stack에 넣어서 출발지로 활용한다.
- 다시 answer를 reverse해준다.

```
for departure in routes.keys():
    routes[departure].sort(reverse=True)

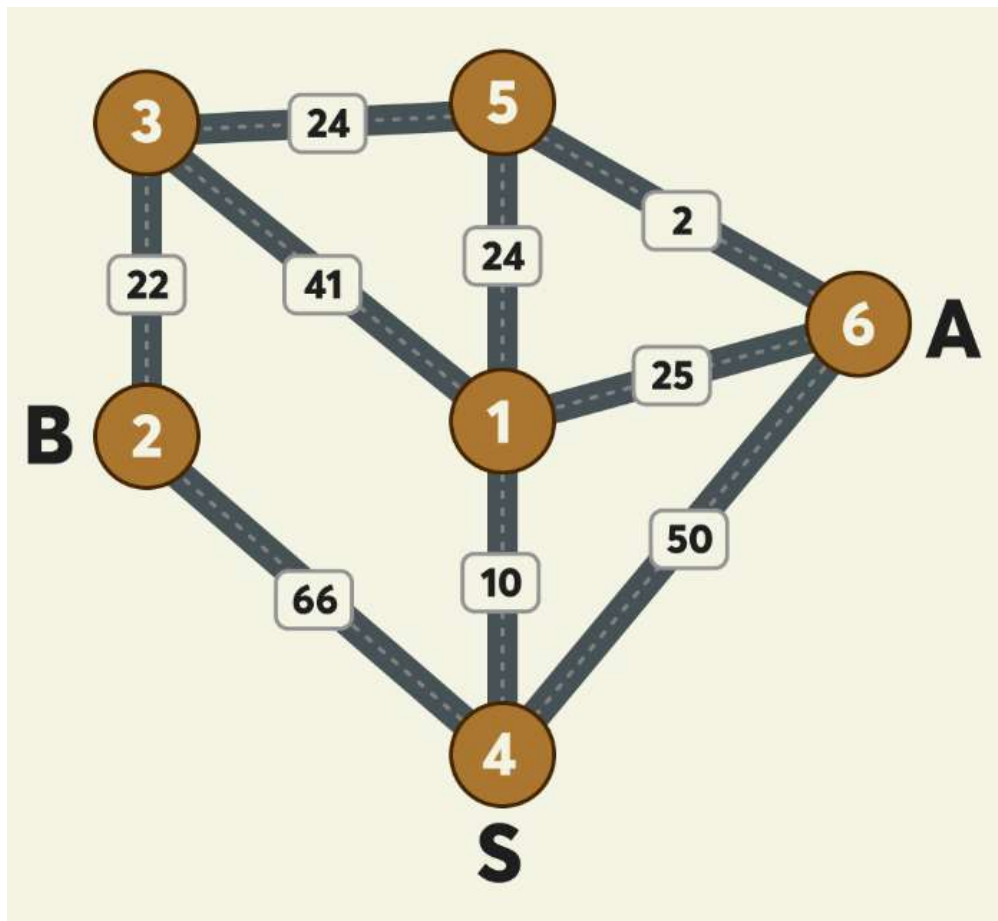
stack = ["ICN"]
while stack:
    if stack[-1] not in routes or not routes[stack[-1]]:
        answer.append(stack.pop())
    else:
        stack.append(routes[stack[-1]].pop())

return answer[::-1]
```

정 -> 역(sort reverse) -> 정(stack) ->
역(answer) -> 정(answer[::-1])

Algorithm – Dijkstra with iteration

[프로그래머스] 합승 택시 요금



[문제]

지점의 개수 n , 출발지점을 나타내는 s , A 의 도착지점을 나타내는 a , B 의 도착지점을 나타내는 b , 지점 사이의 예상 택시요금을 나타내는 $fares$ 가 매개변수로 주어집니다. 이때, A , B 두 사람이 s 에서 출발해서 각각의 도착 지점까지 택시를 타고 간다고 가정할 때, 최저 예상 택시요금을 계산해서 return 하도록 solution 함수를 완성해 주세요.

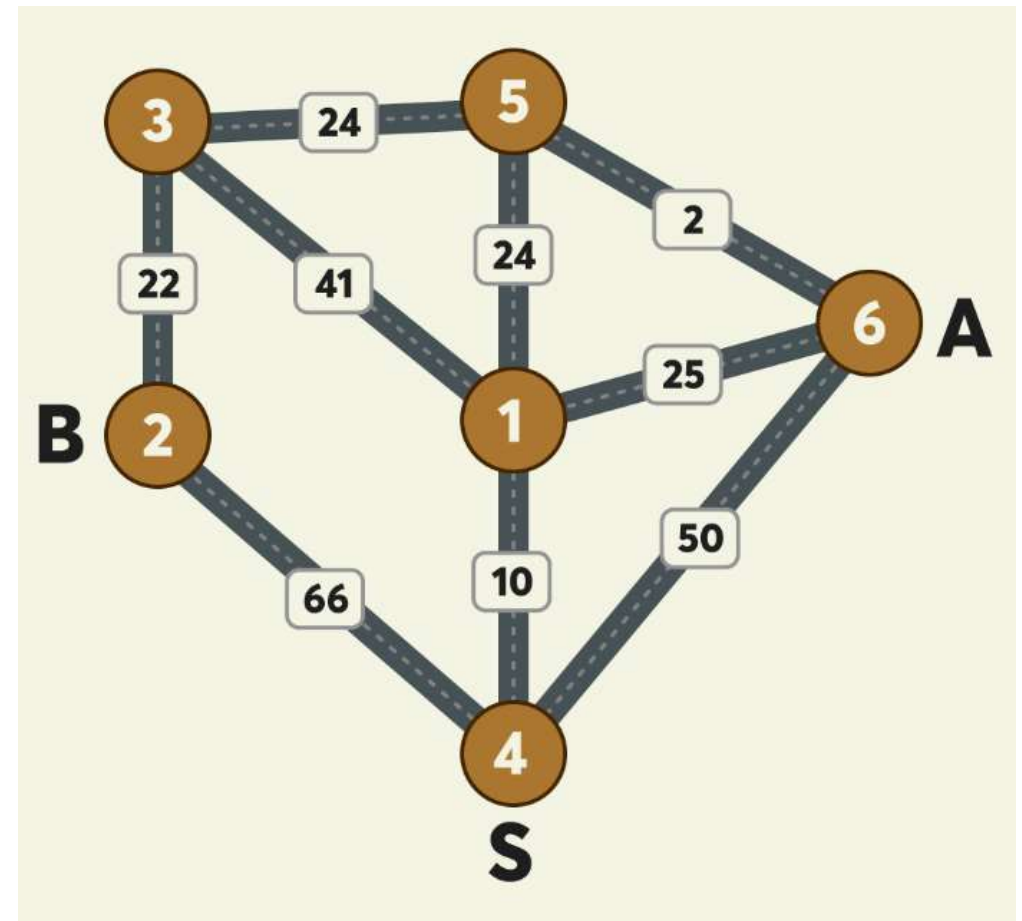
만약, 아예 합승을 하지 않고 각자 이동하는 경우의 예상 택시요금이 더 낮다면, 합승을 하지 않아도 됩니다.

- 예상되는 최저 택시요금은 다음과 같이 계산됩니다.

- 4→1→5 : A , B 가 합승하여 택시를 이용합니다. 예상 택시요금은 $10 + 24 = 34$ 원입니다.
- 5→6 : A 가 혼자 택시를 이용합니다. 예상 택시요금은 2 원입니다.
- 5→3→2 : B 가 혼자 택시를 이용합니다. 예상 택시요금은 $24 + 22 = 46$ 원입니다.
- A , B 모두 귀가 완료까지 예상되는 최저 택시요금은 $34 + 2 + 46 = 82$ 원입니다.

[프로그래머스] 합승 택시 요금

- 문제를 딱 보니 (그래프 구조 + 최소비용경로) => Dijkstra 알고리즘
- 하지만 다익스트라는 출발지와 목적지가 하나로 정해져있는 알고리즘이 아닌가?
- 고민하다가 결국 풀지 못함



해법: Dijkstra 사용한 후, 간선 별로 합산, 반복문

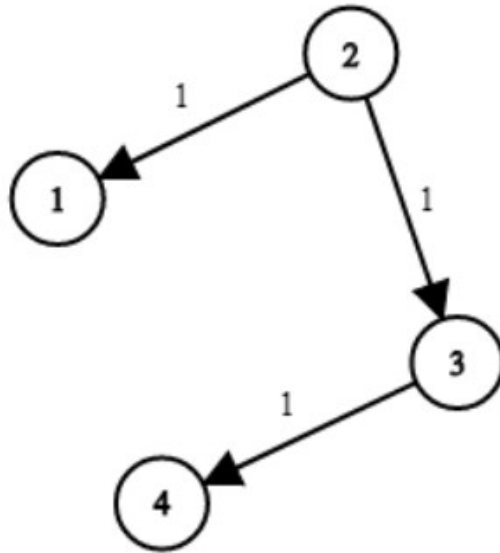
- 출발지에서 분열점까지, 분열점에서 도착지 A까지, 분열점에서 도착지 B까지 각각 dijkstra를 실행해서 합치면 된다.
- 다만 이 분열점이 어디인지 모르므로 모든 노드를 분열점이라고 가정하고 반복문을 돌려서 검사한다.
- 모든 경우의 수에서 가장 작은 min 값을 가져오면, 그것이 가장 적은 요금이 됨.

```
def dijkstra(start):  
    distance = [inf] * (n + 1)  
    distance[start] = 0  
    heap = [(start, 0)]  
    while heap:  
        node, cost = heapq.heappop(heap)  
        for arrival, weight in graph[node]:  
            alt = cost + weight  
            if alt < distance[arrival]:  
                distance[arrival] = alt  
                heapq.heappush(heap, (arrival, alt))  
    return distance
```

```
for i in range(1, n + 1):  
    route.append(dijkstra(i))  
  
for i in range(1, n + 1):  
    answer = min(answer, route[s][i] + route[i][a] + route[i][b])
```


문제를 못 풀었던 이유 – Dijkstra return 값

- 이전에 풀었던 최소 비용경로 문제가 network delay time
- 최단 경로에서 걸리는 Maximum 시간 값을 반환했던 문제
- Dijkstra는 단일 값을 반환한다고 기억하고 있었음
- 하지만 dijkstra의 결과물은 배열이므로 원하는 형태로 가공



```
while len(heap) > 1:
    root = dequeue(heap)
    if not root:
        continue
    cumulative = root[0]
    node = root[1]
    if node not in visited:
        visited[node] = cumulative
        for end, time in dic[node]:
            new_time = cumulative + time
            heap = enqueue(heap, (new_time, end))

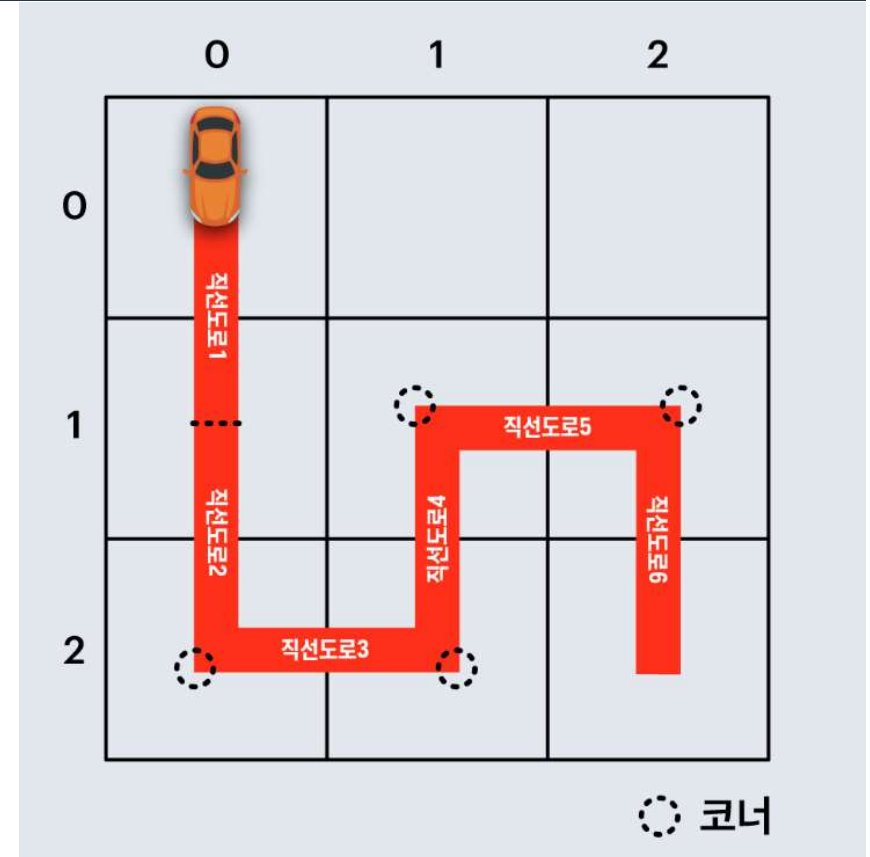
if len(visited) == n:
    return max(visited.values())
return -1
```


Algorithm – DFS

[프로그래머스] 경주로 건설

예를 들어, 아래 그림은 직선 도로 6개와 코너 4개로 구성된 임의의 경주로 예시이며, 건설 비용은 $6 \times 100 + 4 \times 500 = 2600$ 원 입니다.

- 직선도로를 짓는 데는 100원
- 코너를 짓는 데는 500원 소요됨
- 예시 그림은 직선도로 6개, 코너 4개 => 2600원
- 배열이 주어지고 경로와 비용이 주어지는 것을 보니 DFS 관련 문제라고 판단함



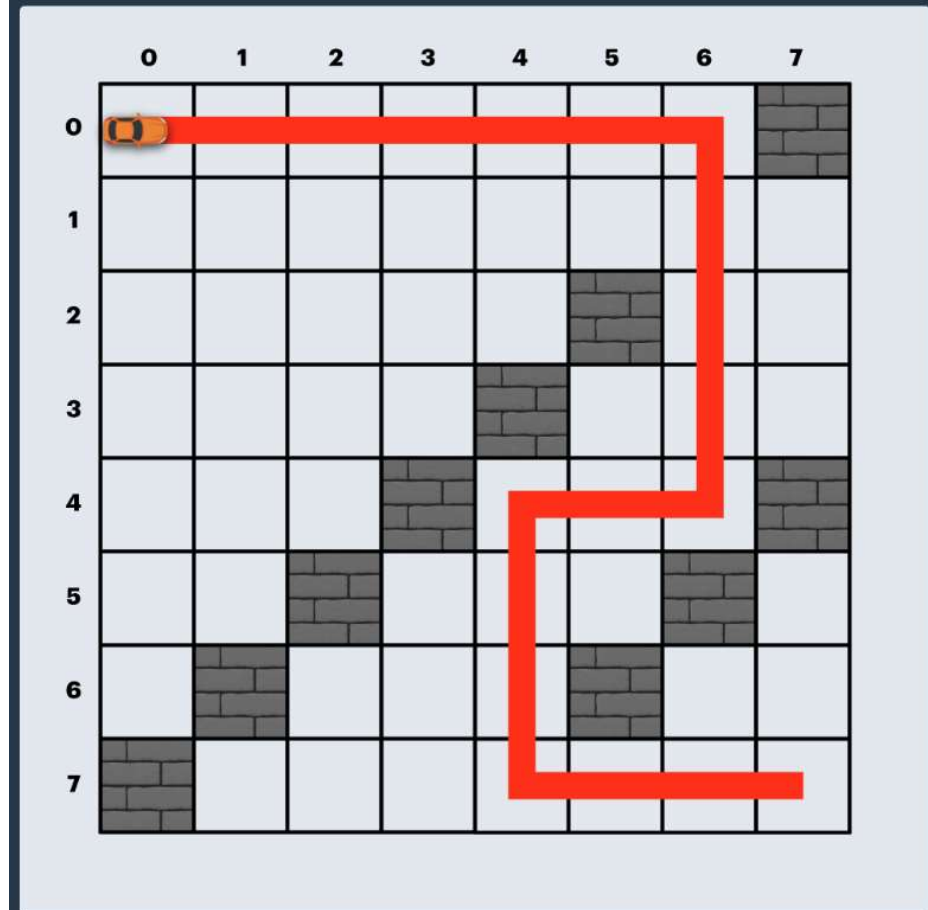
[프로그래머스] 경주로 건설

- 갈 수 있는 빈공간은 0으로 주어진다. (도로 건설 가능)
- 벽은 1로 주어진다. (건설 불가)
- 역방향 진행도 가능 => DFS에서 visited 배열 필요함

입출력 예

board	result
[[0,0,0],[0,0,0],[0,0,0]]	900
[[0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0],[0,0,0,0,0,1,0,0],[0,0,0,0,1,0,0,0], [0,0,0,1,0,0,0,1],[0,0,1,0,0,0,1,0],[0,1,0,0,0,1,0,0],[1,0,0,0,0,0,0,0]]	3800
[[0,0,1,0],[0,0,0,0],[0,1,0,1],[1,0,0,0]]	2100
[[0,0,0,0,0,0],[0,1,1,1,0],[0,0,1,0,0,0],[1,0,0,1,0,1],[0,1,0,0,0,1],[0,0,0,0,0,0]]	3200

입출력 예 #2



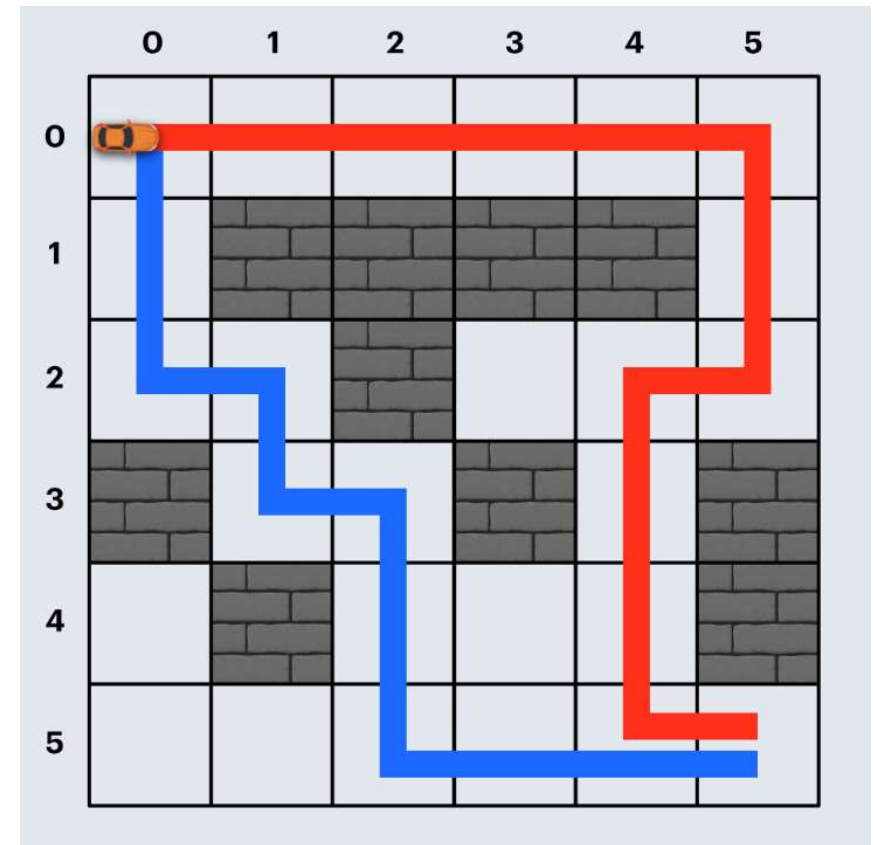
위와 같이 경주로를 건설하면 직선 도로 18개, 코너 4개로 총 3800원이 듅니다.

[프로그래머스] 경주로 건설

- 빨간색 길은 3200원
- 파란색 길은 3500원
- 코너가 많을수록 비용이 많이 올라감.
- 코너를 최대한 적게 사용하기 위해 각 DFS탐색할 때마다 direction을 기록해두자.

```
(new_row, new_col) not in visited:
```

```
if prev_direction == direction:  
    count += 100  
else:  
    count += 600
```



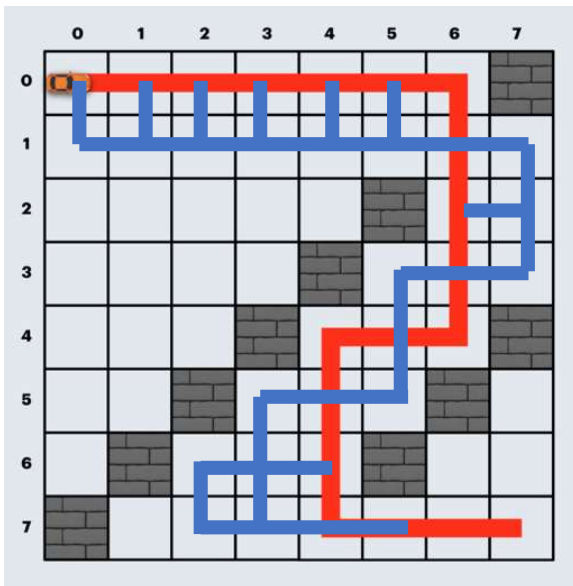
1차 시도 - 모든 경로 탐색하여 기록, min() 사용

- 모든 경로를 탐색하고 모든 경우의 수에서 발생하는 cost를 기록한다.

```
def dfs(row, col, prev_direction, visited, cost):
    if row == len(board) - 1 and col == len(board) - 1:
        candidate.append(cost)
    for direction in direction_list:
        new_row = row + direction[0]
        new_col = col + direction[1]
        if 0 <= new_row < len(board) and 0 <= new_col < len(board[0])
and board[new_row][new_col] == 0 and (new_row, new_col) not in visited:
            if prev_direction == direction:
                cost += 100
            else:
                cost += 600
            visited.append((new_row, new_col))
            dfs(new_row, new_col, direction, visited, cost)
            visited.pop()
            if prev_direction == direction:
                cost -= 100
            else:
                cost -= 600
```

1차 시도 - 모든 경로 탐색하여 기록, min() 사용

- 모든 경로를 탐색하고 모든 경우의 수에서 발생하는 cost를 전부 기록한다.



```
def dfs(row, col, prev_direction, visited, cost):
    if row == len(board) - 1 and col == len(board) - 1:
        candidate.append(cost)
    for direction in direction_list:
        new_row = row + direction[0]
        new_col = col + direction[1]
        if 0 <= new_row < len(board) and 0 <= new_col < len(board[0])
and board[new_row][new_col] == 0 and (new_row, new_col) not in visited:
            if prev_direction == direction:
                cost += 100
            else:
                cost += 600
            visited.append((new_row, new_col))
            dfs(new_row, new_col, direction, visited, cost)
            visited.pop()
            if prev_direction == direction:
                cost -= 100
            else:
                cost -= 600
```

1차 시도 - 모든 경로 탐색하여 기록, min() 사용

- 최초 출발 방향을 오른쪽 방향과 아래쪽 방향으로 두 번 보내서 DFS를 돌린다.
- 모든 경우의 수에서 가장 작은 비용 cost를 답으로 리턴하면 됨
- 일부 테스트는 통과하긴 하지만 너무 많은 경로를 탐색해야 해서 비효율적인 코드

```
dfs(0, 0, (1, 0), [], 0)
dfs(0, 0, (0, 1), [], 0)
print(candidate)
return min(candidate)
```

```
[[0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 1, 0], [0, 0, 1, 0, 0, 0], [1,
0, 0, 1, 0, 1], [0, 1, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0]]
```

3200

테스트를 통과하였습니다.

```
[3500, 5700, 4500, 4500, 4400, 5600, 5800, 3700, 4900, 5100, 5200,
7400, 6200, 6200, 4000, 6200, 5000, 5000, 4900, 6100, 6300, 3200,
4400, 4600, 4700, 6900, 5700, 5700]
```


1차 시도 - 모든 경로 탐색하여 기록, min() 사용

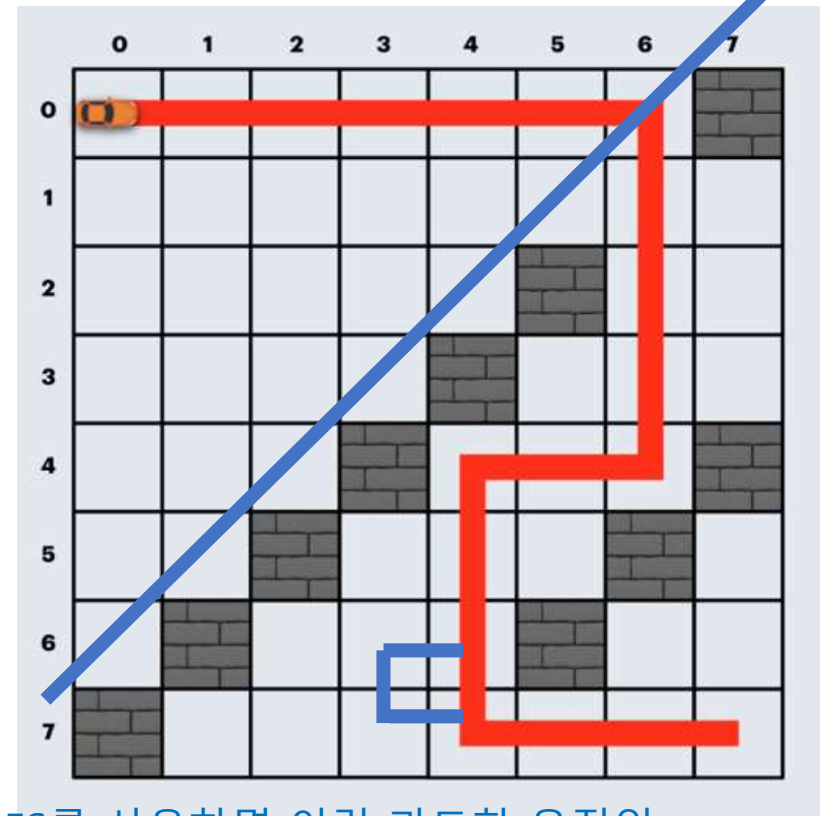
- 결국 시간 초과 발생
- 모든 경로 탐색으로는 풀 수 없음.

```
테스트 1 > 통과 (0.14ms, 10.3MB)
테스트 2 > 통과 (0.03ms, 10.4MB)
테스트 3 > 통과 (0.03ms, 10.4MB)
테스트 4 > 통과 (0.28ms, 10.3MB)
테스트 5 > 통과 (0.19ms, 10.4MB)
테스트 6 > 실패 (시간 초과)
테스트 7 > 실패 (시간 초과)
테스트 8 > 실패 (시간 초과)
테스트 9 > 실패 (시간 초과)
테스트 10 > 실패 (시간 초과)
테스트 11 > 실패 (시간 초과)
테스트 12 > 실패 (시간 초과)
테스트 13 > 통과 (94.69ms, 10.1MB)
테스트 14 > 통과 (4645.22ms, 15.9MB)
테스트 15 > 실패 (시간 초과)
```


못 풀었던 이유: 최단 거리는 BFS

- 최소 칸 움직임으로 목표지점에 도달해야 하므로 DFS가 아닌 BFS를 사용해야 한다.
- 각 움직임에 가중치는 없기 때문에 Dijkstra를 사용하지 않고 BFS만 써도 충분
- 각 그래프 칸에 도달하기까지 비용을 기록하기 위해 $N \times N$ 의 기록 공간을 만든다. => 즉 DP를 이용한다.
- 이런 최단거리 이동은 기본적으로 BFS를 사용해야 한다.

이 선에 해당하는 칸들은 모두 같은 시간이 걸린다.



BFS를 사용하면 이런 과도한 움직임은 애초에 끝에 도달하지 못함

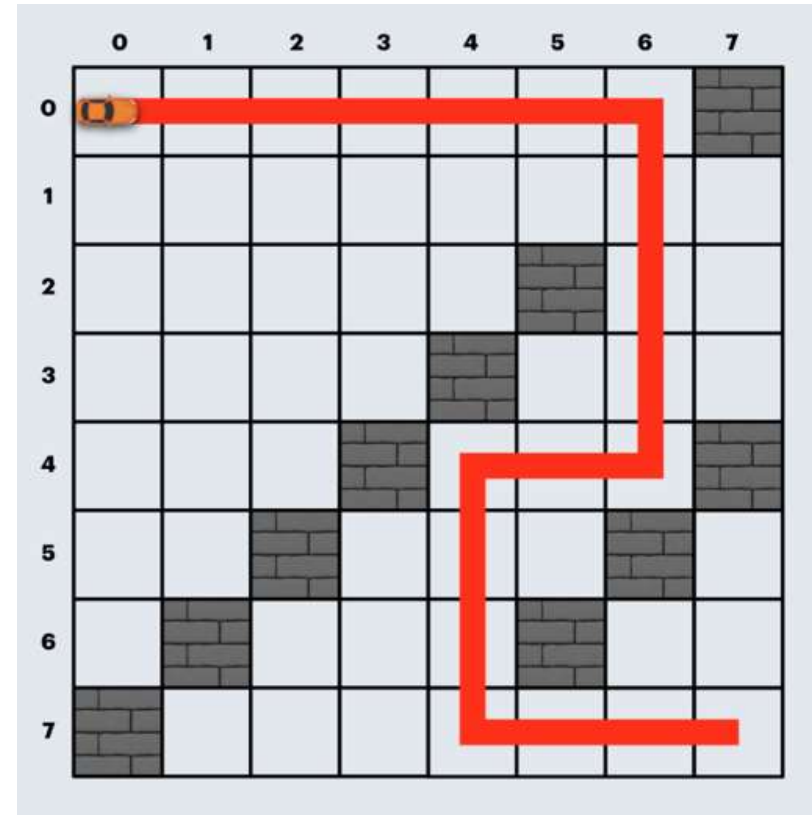
2차 시도 - BFS + DP

- BFS와 DP를 사용하여 모든 칸에 비용을 기록
- 그림에도 불구하고 오답을 뱉어냄.
- 두 방향 출발을 동시에 Queue에 넣는 것은 오답을 내놓을 수 있다.

```
dp = [[inf for _ in range(len(board))] for _ in range(len(board))]
dp[0][0] = 0
direction_list = [(1, 0), (0, 1), (-1, 0), (0, -1)]
q = deque()
q.append((0, 0, 0, direction_list[0]))
q.append((0, 0, 0, direction_list[1]))
while q:
    row, col, cost, prev_direction = q.popleft()
    for direction in direction_list:
        new_row = row + direction[0]
        new_col = col + direction[1]
        if 0 <= new_row < len(board) and 0 <= new_col < len(board[0]) \
        and board[new_row][new_col] == 0:
            if prev_direction == direction:
                new_cost = cost + 100
            else:
                new_cost = cost + 600
            if dp[new_row][new_col] > new_cost:
```

2차 시도 - BFS + DP

- 첫 출발지에서 오른쪽으로 갈 수도 있고 아래쪽으로 갈 수도 있다.
- 이것을 Queue에 한 번에 담아선 안 된다.
- 오른쪽 출발, 아래쪽 출발 각각 연산 한 다음 한 번에 가져와야 한다.
- min() 함수를 사용하여 둘 중 더 저렴한 비용을 가져와야 함.



2차 시도 - BFS + DP

- BFS 별도의 함수 정의
- 오른쪽 출발의 경우와 아래쪽 출발의 경우를 각각 구한 다음 min()으로 처리한다.

```
def bfs(i):  
    dp = [[inf for _ in range(len(board)) for _ in range(len(board))]  
          dp[0][0] = 0  
          q = deque()  
          q.append((0, 0, 0, direction_list[i]))  
          while q:  
              row, col, cost, prev_direction = q.popleft()  
              for direction in direction_list:  
                  new_row = row + direction[0]  
                  new_col = col + direction[1]  
                  if 0 <= new_row < len(board) and 0 <= new_col < len(board[0]) \ and board[new_row][new_col] == 0:  
                      if prev_direction == direction:  
                          new_cost = cost + 100  
                      else:  
                          new_cost = cost + 600  
                      if dp[new_row][new_col] > new_cost:  
                          dp[new_row][new_col] = new_cost  
                          q.append((new_row, new_col, new_cost, direction))  
          return dp[-1][-1]  
  
return min(bfs(0), bfs(1))
```

Micro Service Architecture

Monolithic architecture

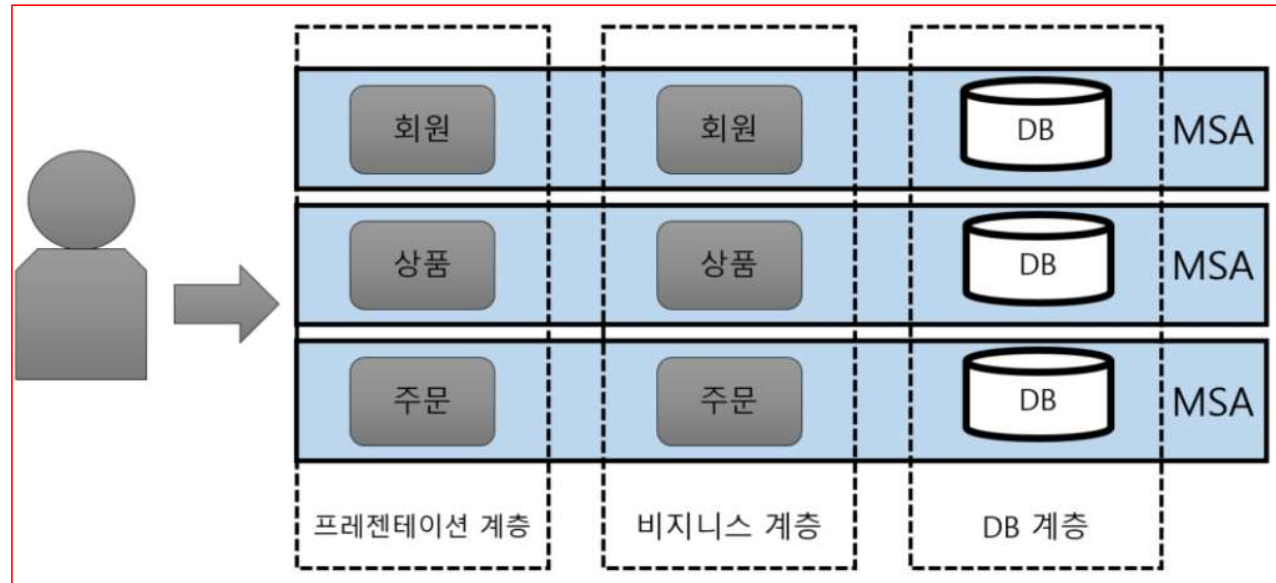
- 전체 어플리케이션이 하나로 되어있음
- 장점: 개발 및 환경설정이 간단. 테스트도 하나의 어플리케이션이 수행
- 단점: 시스템이 커지면 빌드/테스트 시간이 길어진다. 특정 기능을 선택적으로 scale out 할 수 없다. 하나의 서비스가 모든 서비스에 영향을 준다.



포인트 관리 기능에서 에러가 발생했는데, 상품 조회도 못 하게 되는 상황이 발생함.

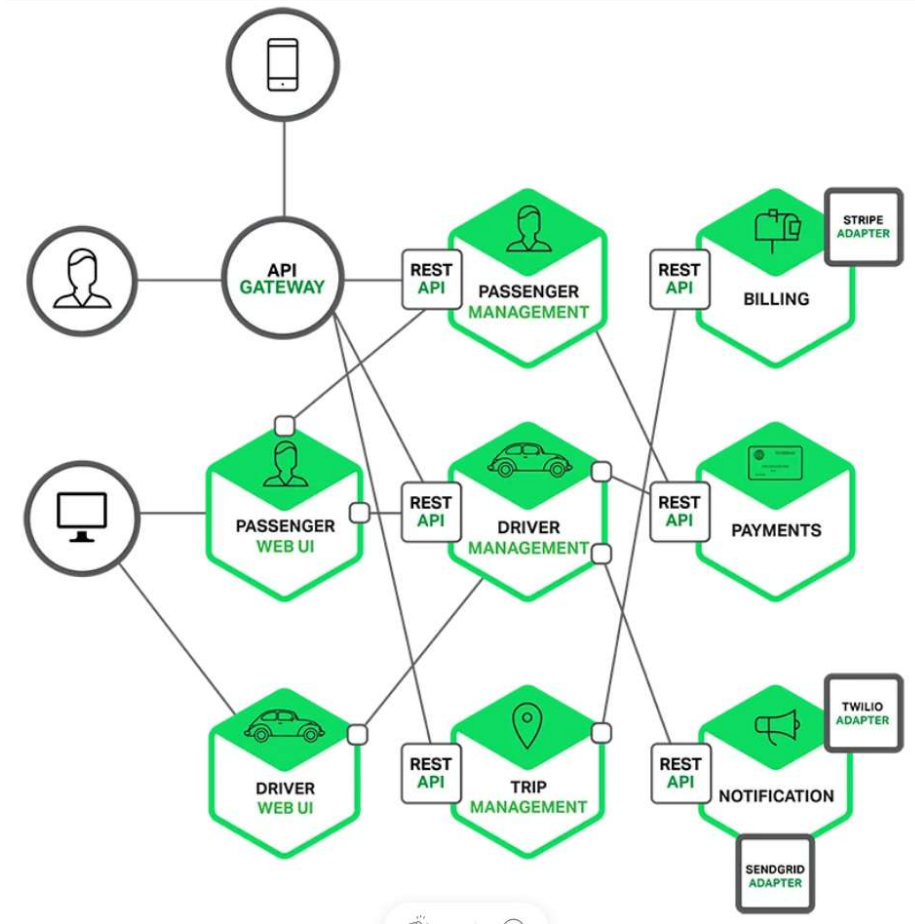
Micro Service Architecture (MSA)

- 하나의 어플리케이션을 컴포넌트 별로 나누어 작은 서비스의 조합으로 구축하는 방식
- 각 컴포넌트는 서비스 형태로 구현되고, API로 타 서비스와 통신하게 된다.
- 각각의 서비스는 독립적이다. 다른 서비스에 의존하지 않기 때문에 독립적으로 배포한다.



Micro Service Architecture (MSA)

- 장점: 각 컴포넌트가 독립적으로 개발되었기 때문에 성능이 중요한 서비스는 부분적인 확장이 가능함. 서비스 별로 별도의 DB를 사용할 수 있다. 하나의 서비스가 장애가 나도 다른 서비스는 멀쩡하게 작동된다.
- 단점: monolithic에 비해 latency가 증가한다. 통신을 사용하기 위해 데이터를 변환하는 작업에서 워크로드 추가. 각 컴포넌트의 데이터를 트랜잭션으로 묶을 수 없음. API 정의를 제대로 해야 혼란이 없음



Micro service architecture 고려사항

- 배달의 민족 예시
- 각 서비스에 필요한 성능은 어느 정도인가? (서비스마다 성능을 다르게 할 수 있음)
- 서비스-서비스 간의 장애 격리는 어떻게 구현할 것인가?
- 서비스-서비스 간 데이터 동기화는 어떻게 구현할 것인가?

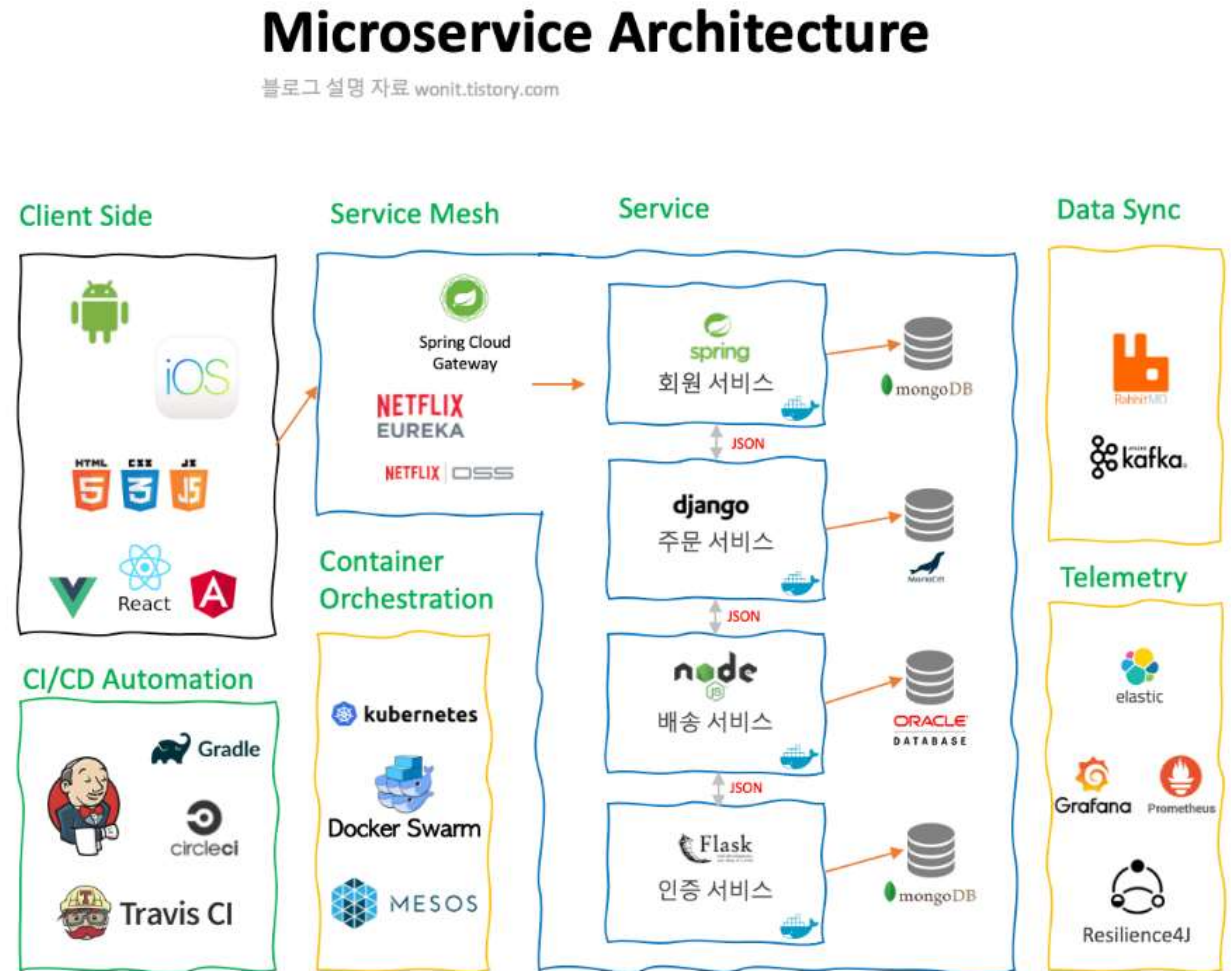
먼데이

먼데이 아키텍처 고려사항

- ▶ 성능
- ▶ 장애 격리
- ▶ 데이터 동기화

CNCF가 제시한 MSA 구성요소

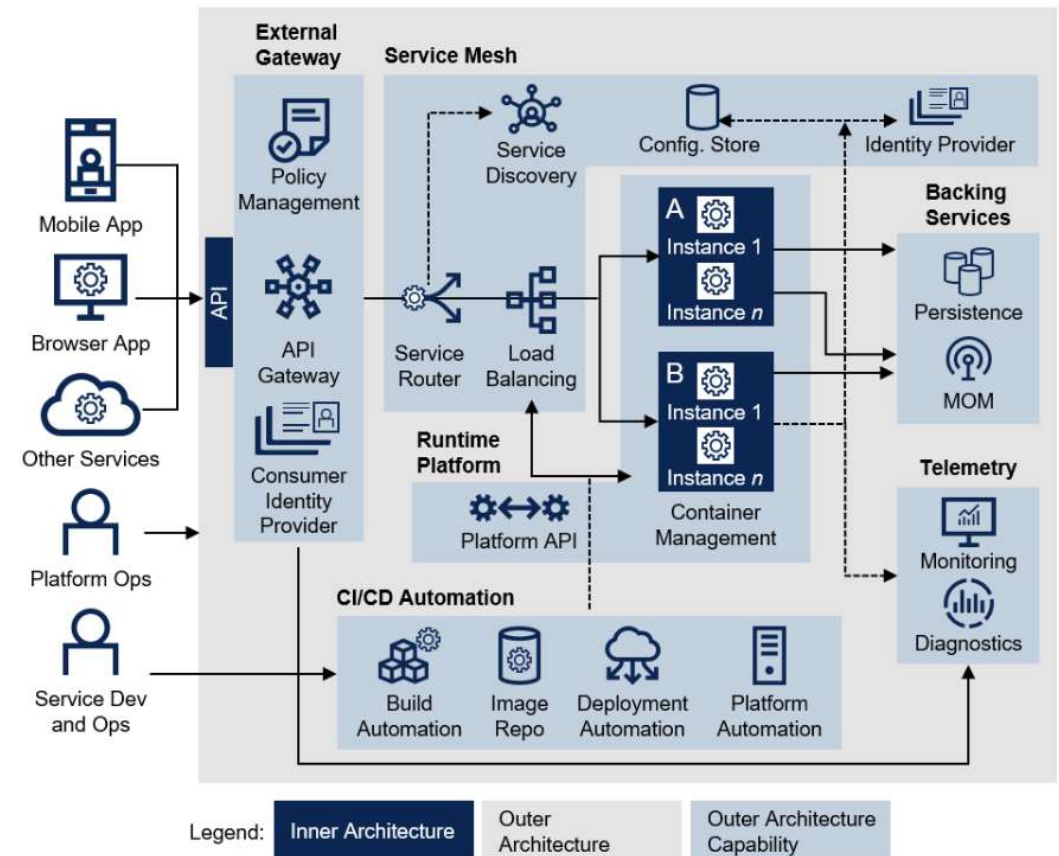
- **API Gateway**
- Service Mesh
- Container Runtime
- CI/CD
- Message Queuing
- RESTful API



Gartner가 제시한 MSA 구조

- **API Gateway**
- Load balancer

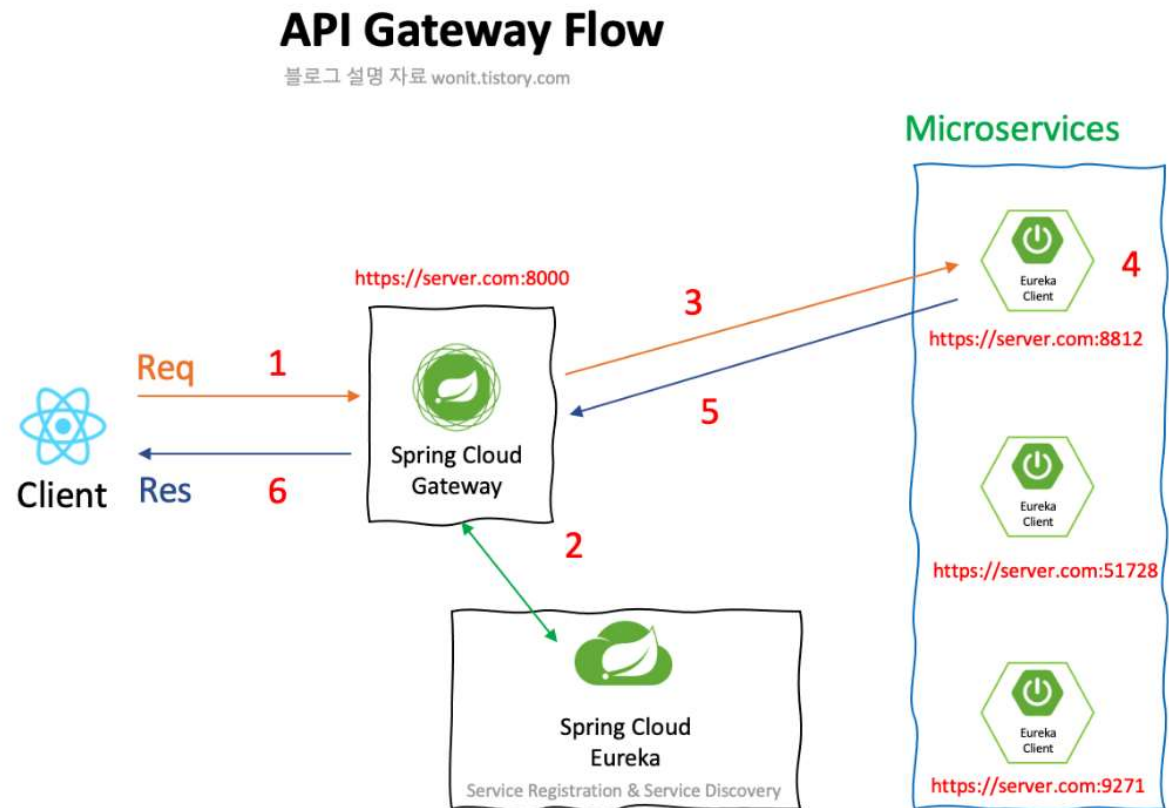
Microservices Architecture Components



API gateway

API Gateway

- 하나의 모든 클라이언트의 요청이 하나의 서버로 들어와 해당 서버에서 요청이 정제되어 각자 목적에 맞는 서비스를 보냄. 또한 각 서버에서 적절한 로직을 수행한 뒤 발생하는 응답 데이터를 모아 사용자에게 분배함.
- API gateway 역할:
 - Proxy 역할 (인증과 로깅)
 - Load balancing, routing



API Gateway vs Load Balancer

- API gateway
- Authentication, Authorization, Rate limiting, Logging에 특화되어 있음
- 서비스가 여러 서버로 분할되어 있어서 단일 액세스 포인트로 접근할 수 있게 만들려고 할 때 유용함
- Rate limiting – 제한 시간 안에 요청할 수 있는 개수가 정해져 있다.

- Load Balancer
- Content Caching, HTTP Compression, SSL offloading에 특화되어 있음
- 여러 개의 웹 서버에 부하를 분산하여 성능을 높이는데 초점이 맞춰져 있음
- SSL offloading – HTTPS 보안 접속을 할 때 사용하는 암호화 복호화를 프록시 서버에서 대신 해주는 것.

API Gateway with Load Balancer

- 물론 둘 다 써도 된다.
- 특정 서비스의 보안이 중요하다면 -> API gate way
- 응답 처리 성능이 중요하다면 -> load balancer
- 만약 서비스도 분산되어 있고, 각 서비스당 여러 서버를 사용한다면, API gateway도 사용하고 load balancer도 써야 한다.

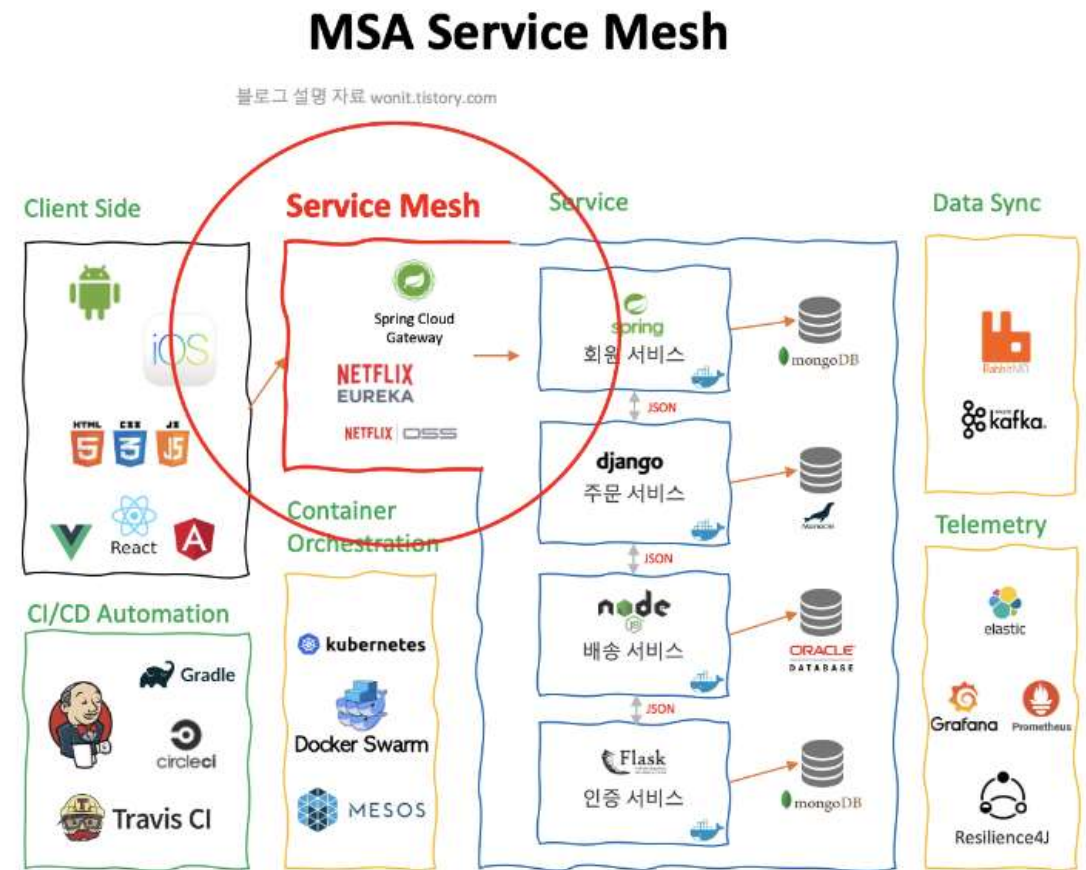


Knowledge Tip:

When choosing between an API gateway and a load balancer, it's essential to consider your application's specific security and performance needs. Also, if you're dealing with multiple servers or services, it's worth considering using both an API gateway and a load balancer to ensure maximum coverage.

Service Mesh

- CNCF가 제시한 MSA 구성요소 중 하나
- 마이크로 서비스 구조에서 서비스 간 통신이 일어날 때 어느 지점으로 전송이 될 지에 대해 추상화한 것.
- 역할: 서비스 등록, 서비스 검색, 프록시, 인증, 로드밸런싱, 암호화
- API gateway와 거의 같다.



Service Mesh vs API gateway

- 둘의 역할은 거의 같다.
- 하지만 외부에 노출되는 정도가 다름

이름	수행하는 일	적용 위치	외부 노출 여부
API Gateway	라우팅, 인증, 모니터링, 서비스 검색, 서비스 등록	Client-to-Server	Yes
Service Mesh	라우팅, 인증, 모니터링, 서비스 검색, 서비스 등록	Server-to-Server	No

적용되는 위치

API Gateway는 마이크로서비스 그룹의 외부 경계에 위치하여 역할을 수행하지만, ServiceMesh는 경계 내부에서 그 역할을 수행합니다. API Gateway의 주요 목적은 네트워크 외부의 트래픽을 수락하고 내부적으로 배포하는 것입니다. 서비스 메시의 주요 목적은 네트워크 내부에서 트래픽을 라우팅하고 관리하는 것입니다.

Service Mesh with API gateway

- 앞으로 service mesh가 API gateway의 기능을 흡수 할 것으로 전망됨

서비스 메시 기술은 빠르게 발전하고 있으며 API 게이트웨이의 일부 기능을 수행 하기 시작했습니다. 클라우드 네이티브 공간이 발전하고 더 많은 조직이 Docker 및 Kubernetes를 사용하여 마이크로 서비스 아키텍처를 관리하는 방식으로 이동함에 따라 서비스 메시와 API 게이트웨이 기능이 병합될 가능성이 높습니다. 앞으로 몇 년 안에는 독립형 API 게이트웨이의 많은 기능이 서비스 메시에 흡수되어 점점 더 적게 사용될 것으로 예측 됩니다.

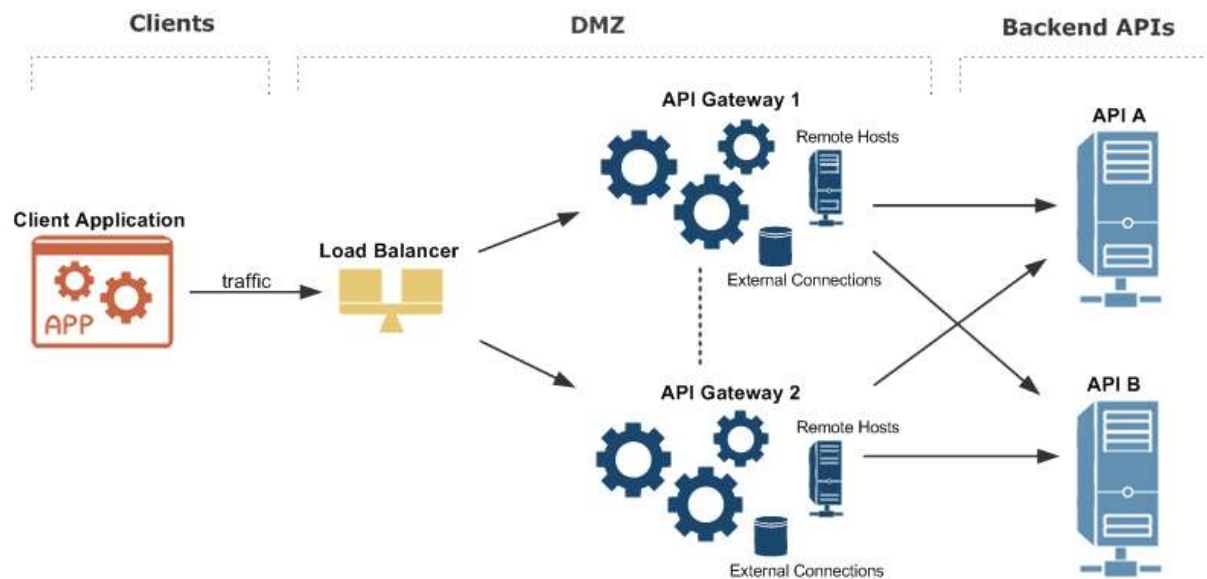
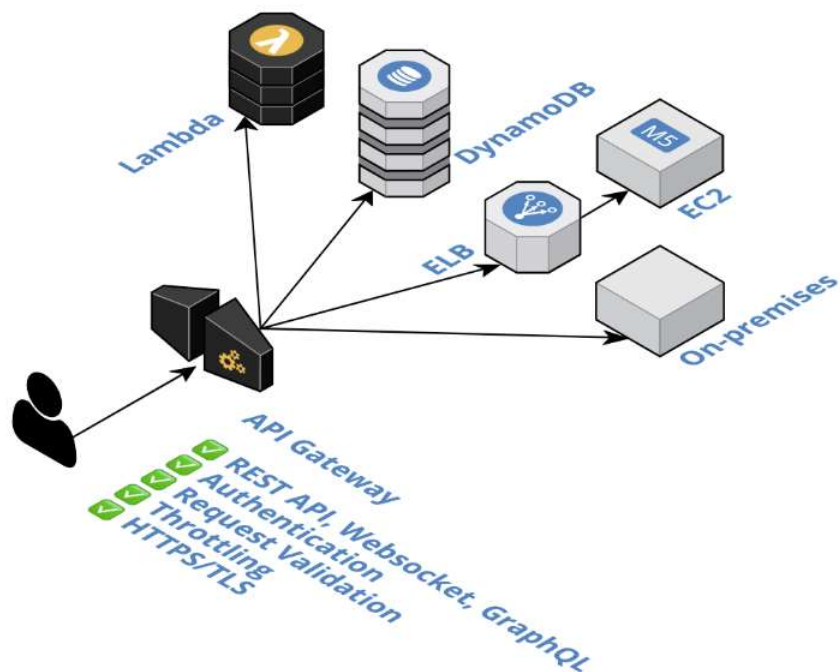
- 로드 밸런서도 마찬가지

Mark Church가 Kubernetes 및 애플리케이션 네트워킹의 미래에 대한 NGINX Sprint 2.0 기조 연설에서 "API Gateway, Load Balancer 및 Service Mesh는 계속해서 서로 점점 더 유사하게 보이고 유사한 기능을 제공할 것"이라고 들었습니다.

그렇다면 설계는 어떻게?

<Separated 방식>

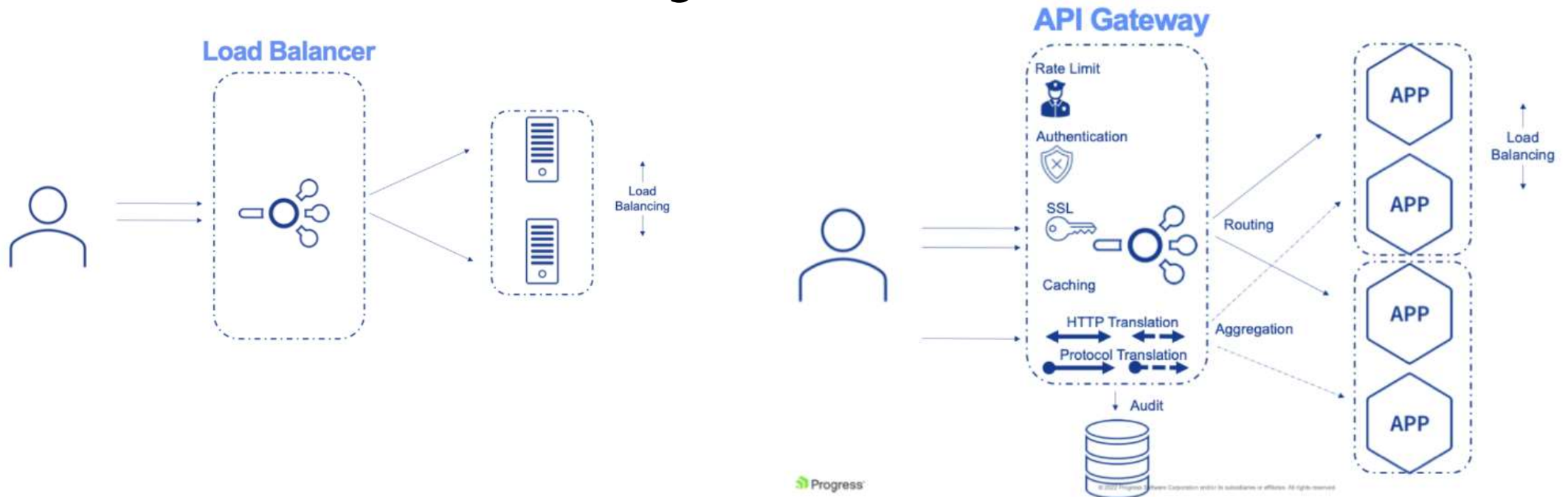
- Load balancer 를 앞 단에 배치하고 그 뒤에 API gateway를 배치하거나 Gateway를 먼저 배치하고 그 뒤로 load balancer 를 배치할 수도 있다.
- 순전히 설계 방향에 따라 결정하면 됨.



그렇다면 설계는 어떻게?

<Consolidated 방식>

- 기존 로드 밸런서 내에서 API gateway 를 구현할 수도 있다.



그렇다면 설계는 어떻게?

- 이처럼 설계에 API gateway를 몇 개를 사용할 것인지에 따라 API gateway 앞에 load balancer를 배치할 수도 있다.

Scenario 1: You have a cluster of API Gateways

User ---> Load Balancer (provided by Cloud Providers like AWS or your own) ---> API Gateway Cluster ---> Service Discovery Agent (like *eureka*) ---> Microservice A ---> Client Side Load Balancer ---> Microservice B

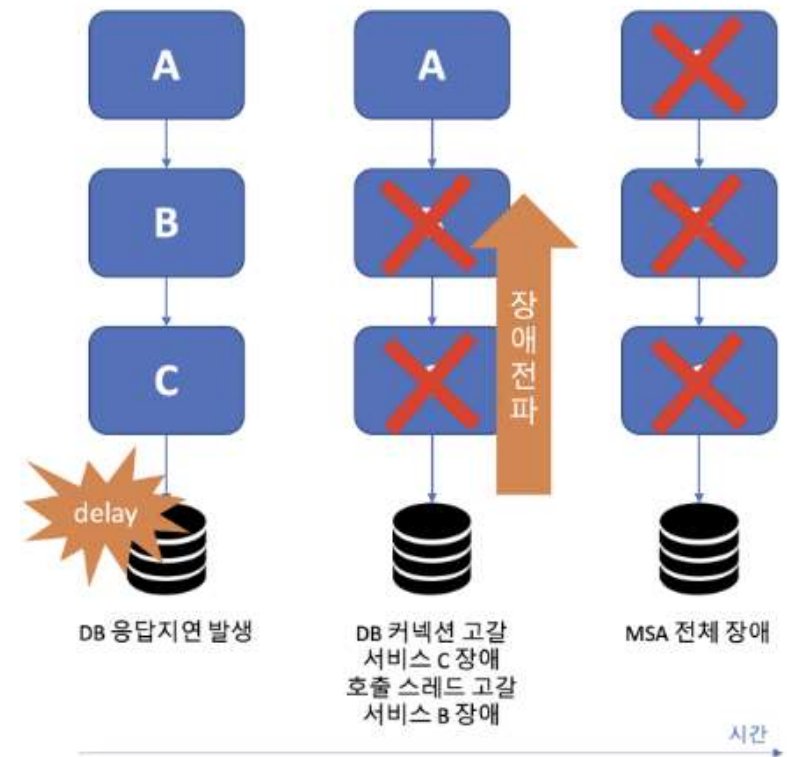
Scenario 2: You have a *single* API Gateway

User ---> API Gateway ---> Service Discovery Agent (like *Eureka*) ---> Microservice A ---> Client Side Load Balancer -> Microservice B

Message Queue

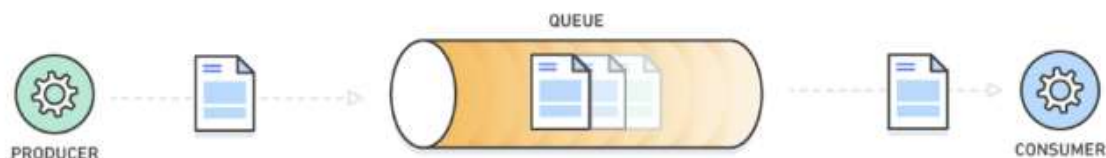
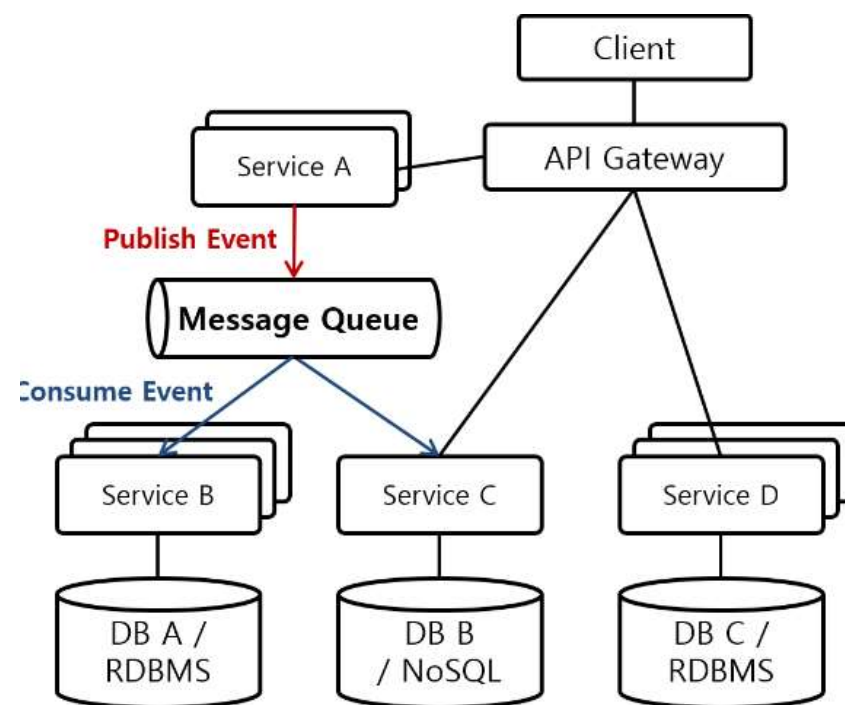
MSA 문제점

- 첫번째 단점은 성능이 떨어진다는 것.
Network latency 때문에 시간이 더 걸릴 수 있음 하지만 이것은 API gateway의 캐시 기능을 적절히 활용하면 충분히 커버칠 수 있다고 봄
- 두번째 단점은 하나의 서비스가 다른 서비스에 요청을 무한히 반복할 수 있다는 것. 호출이 동기식으로 이루어지는 경우 특정 DB에 응답이 지연되면, DB 커넥션이 고갈되고, 연쇄적으로 스레드 풀도 고갈되면서 전체 서비스가 죽어버릴 수도 있다.



메시지 큐 역할

- 마이크로 서비스 아키텍처가 가지는 느슨한 결합을 구성하기 위해서는 메시지 큐를 활용한다.
- Consumer는 producer의 로직을 이해할 필요 없이 이벤트 방식으로 데이터만 받는다. 장애가 서로 분리된다.
- 비동기적인 통신이므로 즉각적인 응답을 줘야하는 기능에는 사용하는 것이 부적합하다.

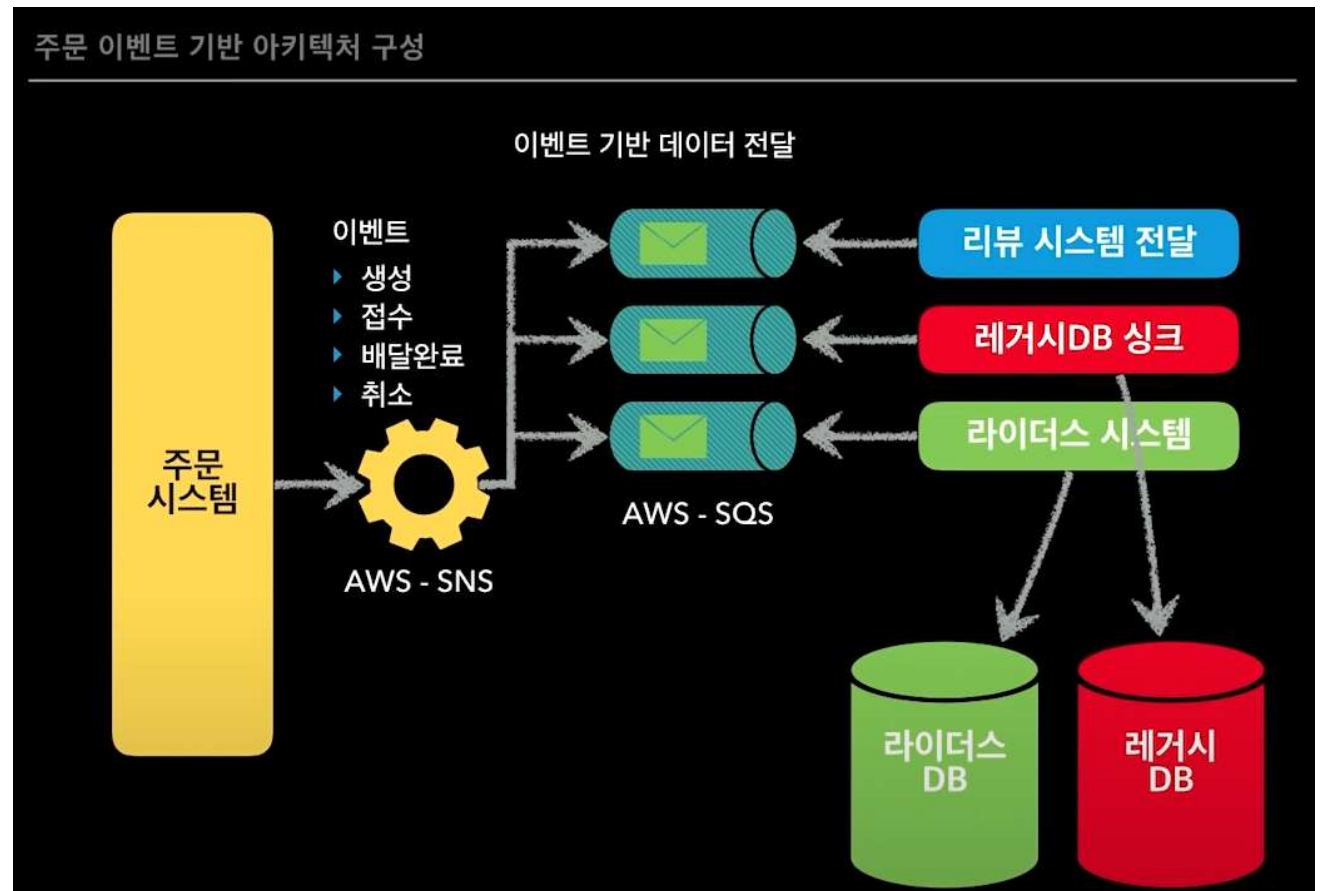


메시지 큐 장점

- 비동기(Asynchronous) : 데이터를 수신자에게 바로 보내지 않고 큐에 넣고 관리하기 때문에 나중에 처리 가능
- 낮은 결합도(Decoupling) : 생산자 서비스와 소비자 서비스가 독립적으로 행동하게 됨으로써 서비스 간 결합도가 낮아진다.
- 탄력성(Resilience) : 일부가 실패하더라도 전체에 영향을 주지 않음
- 보증(Guarantees) : 작업이 처리된 걸 확인할 수 있음
- 확장성(Scalable) : N:1:M 구조로 다수의 프로세스들이 큐에 메시지를 보낼 수 있음
- 과잉(Redundancy) : 실패할 경우 재실행 가능

메시지 큐 구성 예시

- 배달의 민족 메시지 큐 구성 예시
- 주문 시스템은 다른 서비스가 필요로 하는 인자를 신경 쓸 필요가 없다.
- 그냥 본인들에게 사전에 정의된 형식을 지켜서 이벤트를 내보내면 됨.



메시지 큐 인터페이스 종류

- **RabbitMQ**

- AMQP 구현한 오픈소스 MQ. At-Most-Once, At-Least-Once, Exactly-Once 전달 방식 선택. Message Queue가 도착하기 전에 라우팅, 플러그인을 통해 더 복잡한 라우팅도 가능

- **ActiveMQ**

- Spring 애플리케이션에 쉽게 임베딩 + XML 설정이 쉬움

- **Kafka**

- 대용량의 실시간 로그 처리에 특화되어 설계된 메시징 시스템
- 다른 메시징 시스템은 메시지 큐에 적재된 메시지 양이 많을수록 성능이 크게 감소하였지만, Kafka는 메시지를 파일 시스템에 저장하기 때문에 성능이 크게 감소하지 않음. 기존 메시지 큐의 다양한 기능을 포기하고 대용량 메시지를 처리하기 위한 기능을 제공하는 메시지 큐

AWS 메시지 큐 - SQS

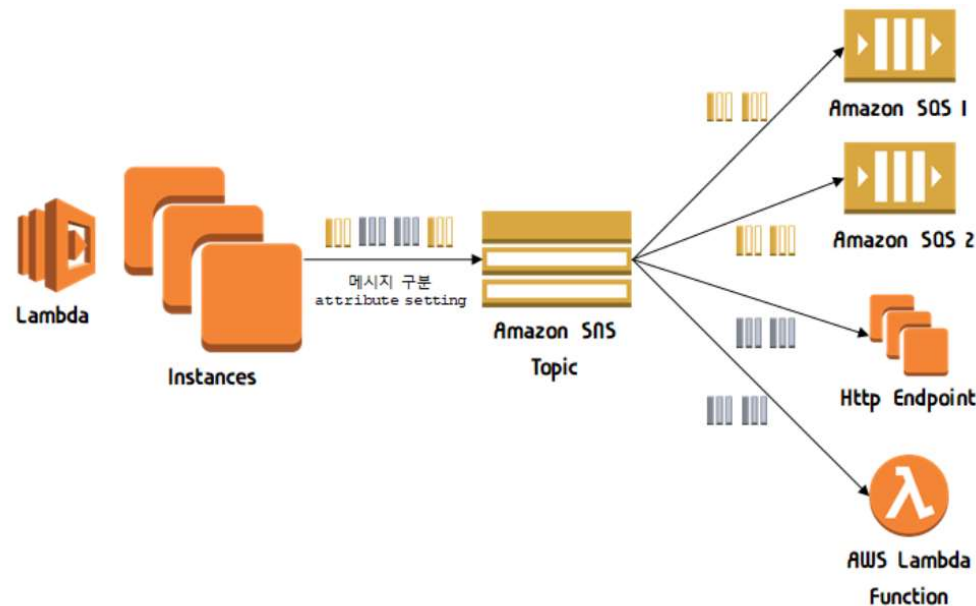
- 서버리스 간 통신을 위한 완전관리형 메시지 큐
- 여러 AZ에 걸쳐서 메시지 전송할 수 있음
- Pull 형 queue 서비스

a. Amazon SQS (Simple Queue Service)

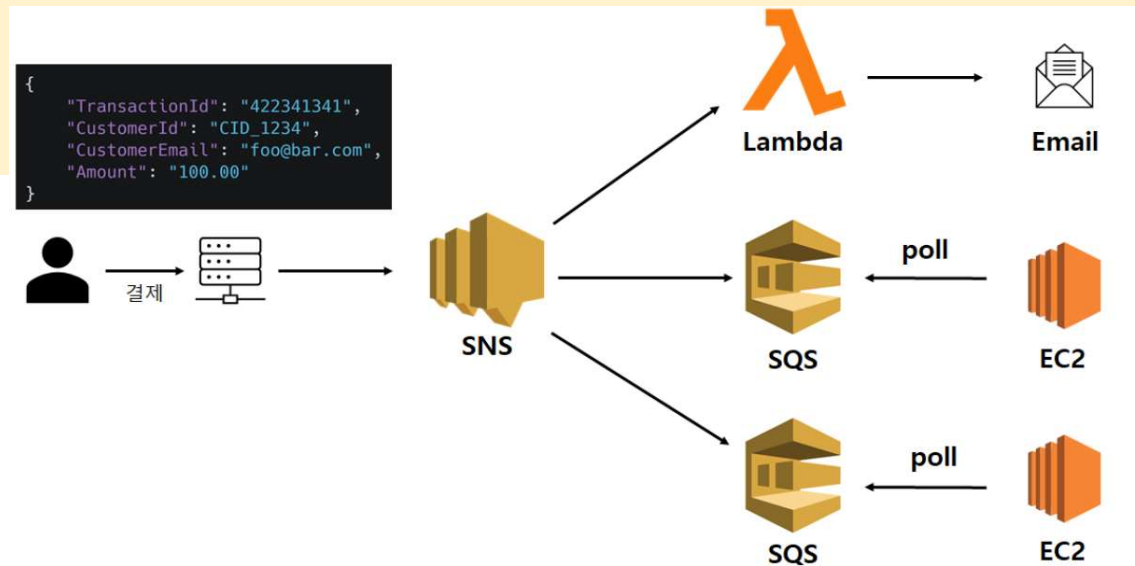


AWS 메시지 큐 - SNS

- Pub/Sub 메시지 알림 서비스.
- Push형 queue 서비스. 수신 측이 별도의 polling을 할 필요가 없다.
- 하나의 메시지는 여러 수신자에게 보낼 수 있다. 모든 consumer에게 보낼 수 없으므로 송신 측에서 topic 별로 필터링해야 한다.



SNS vs SQS

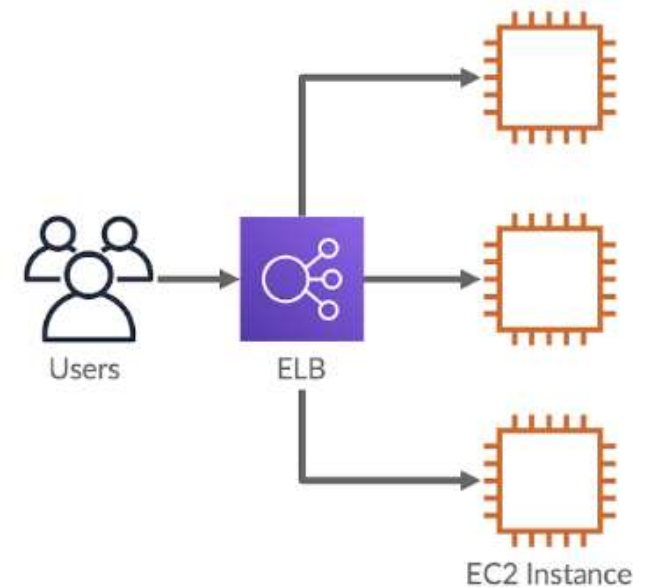


SNS	SQS
Topic(Pub/Sub)	Queue
Push: 사용자에게 메시지 전송	Pull: 사용자가 메시지를 가져온다
Fanout: 같은 메시지를 여러 방향으로 발행	Decouple: 여러 서비스로 분리하여 병렬식 비동기 처리
메세지를 받는 사용자가 없으면 몇 번 시도하다가 삭제된다.	메세지는 일정 기간동안 유지된다.
하나의 메시지를 사용자들이 각각 다른 방식으로 활용	하나의 메시지를 사용자들이 동일한 방식으로 활용

AWS - ELB

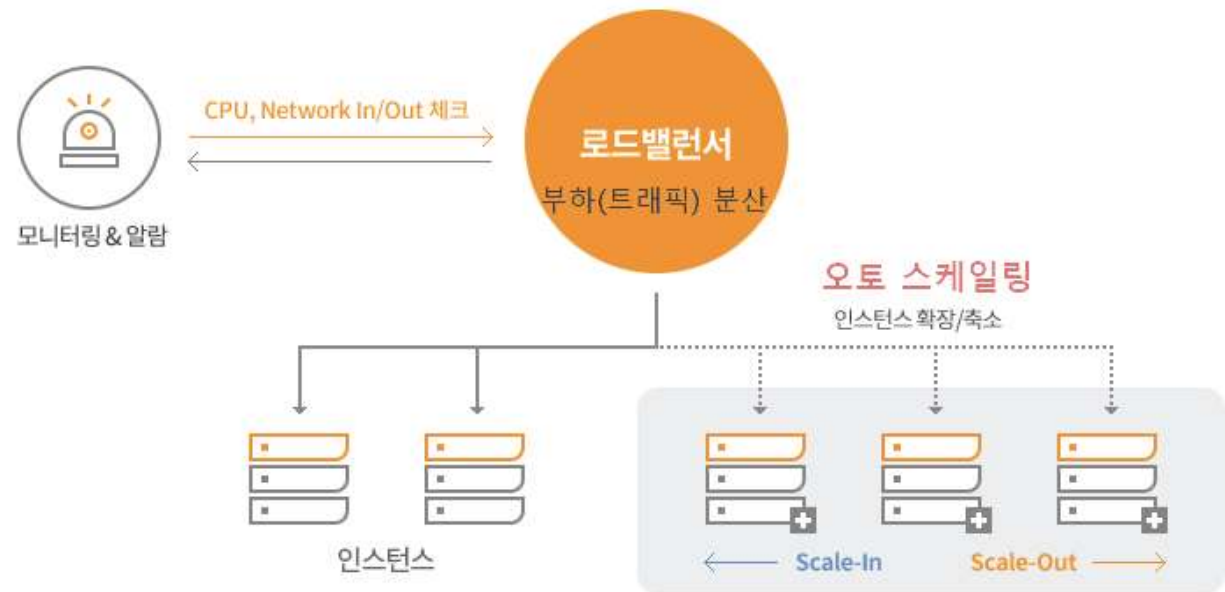
ELB의 역할

- ELB는 AWS에서 load balancing 하는 서비스를 통칭하는 것.
- 사용자는 인스턴스에 신경 쓰지 않고, 동일한 end point로 요청을 계속 날릴 수 있다.
- 부하 분산 뿐만 아니라, SSL offloading, sticky session, health check를 통한 down 서버 제외 등을 수행할 수 있다.



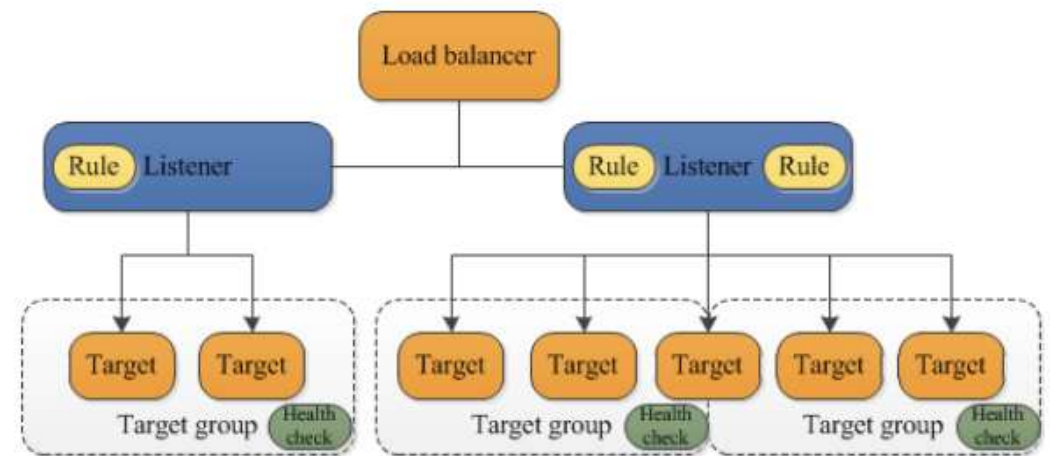
ELB with auto-scaling

- 오토 스케일링은 트래픽이 몰릴 때 자동으로 서버 수를 늘려주는 서비스. 트래픽이 감소하면 서버도 감소시킨다.
- ELB는 오토 스케일링이 감지한 서버 대수를 고려하여 부하를 분산한다.



ELB 구성 요소

- load balancer: 클라이언트가 요청할 수 있는 단일 접점
- Listener: 프로토콜 및 포트를 사용하여 클라이언트의 연결 요청을 확인, 이를 적절한 타겟으로 전달하는 기능.
- Target group: 리스너 규칙 조건이 충족된 요청에 대해 분산 받을 그룹.

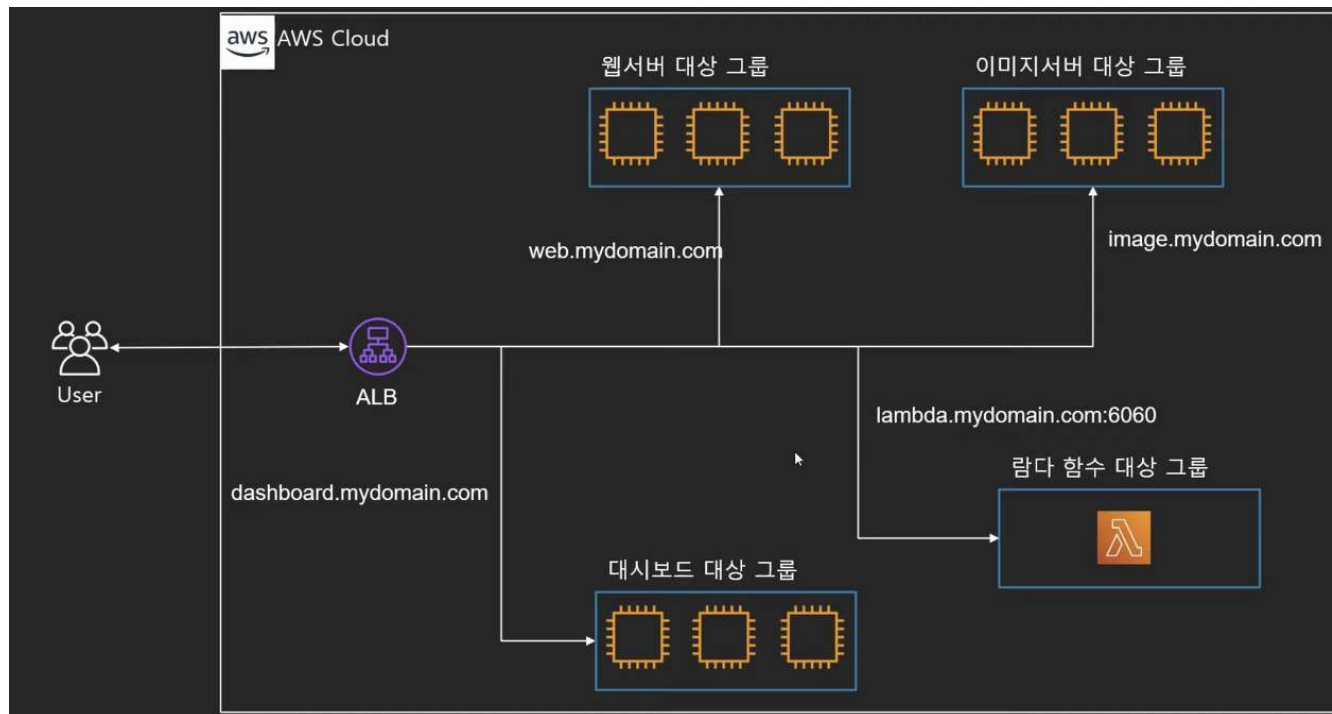


Listener

- 모든 로드 밸런서는 최소 1개 최대 10개의 리스너를 가질 수 있다.
- SSL 인증서도 게시해서 SSL offloading도 할 수 있다.
- Listener Rule: 룰은 우선순위, 액션, 조건 등의 정보가 있음. 조건에 부합할 때 지정된 액션을 수행하는 방식으로 작동됨
- 조건: path, host, HTTP header, source IP, query parameter 등등

Target Group

- 리스너가 전달한 요청을 처리할 대상들의 모임
- 요청 처리가 가능한 EC2가 몇 개인지 확인하는 모니터링 기능이 제공된다.



ELB가 요청을 처리하는 방법

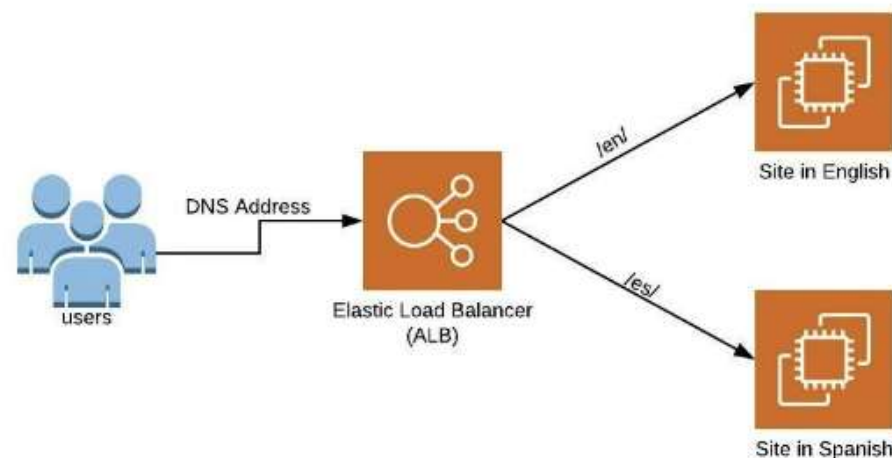
1. 클라이언트가 DNS 서버에 로드밸런서의 도메인을 질의한다.

- ELB는 `http://amazonaws.com` 도메인을 가지고 있다. 이 DNS 질의에 의해 Amazon DNS 서버가 하나 또는 여러 개의 IP 주소를 반환하게 된다. 전달받은 IP가 바로 로드밸런서의 IP이다.

2. 로드밸런서를 향해 전달된 트래픽은 리스너를 통해 전달될 대상 그룹을 결정하게 된다.

3. 전달될 대상 그룹이 결정되면, 정해진 라우팅 프로토콜에 의해 대상 그룹 내의 리소스(인스턴스 등..)에게 전달된다.

[ELB의 요청 처리 과정]

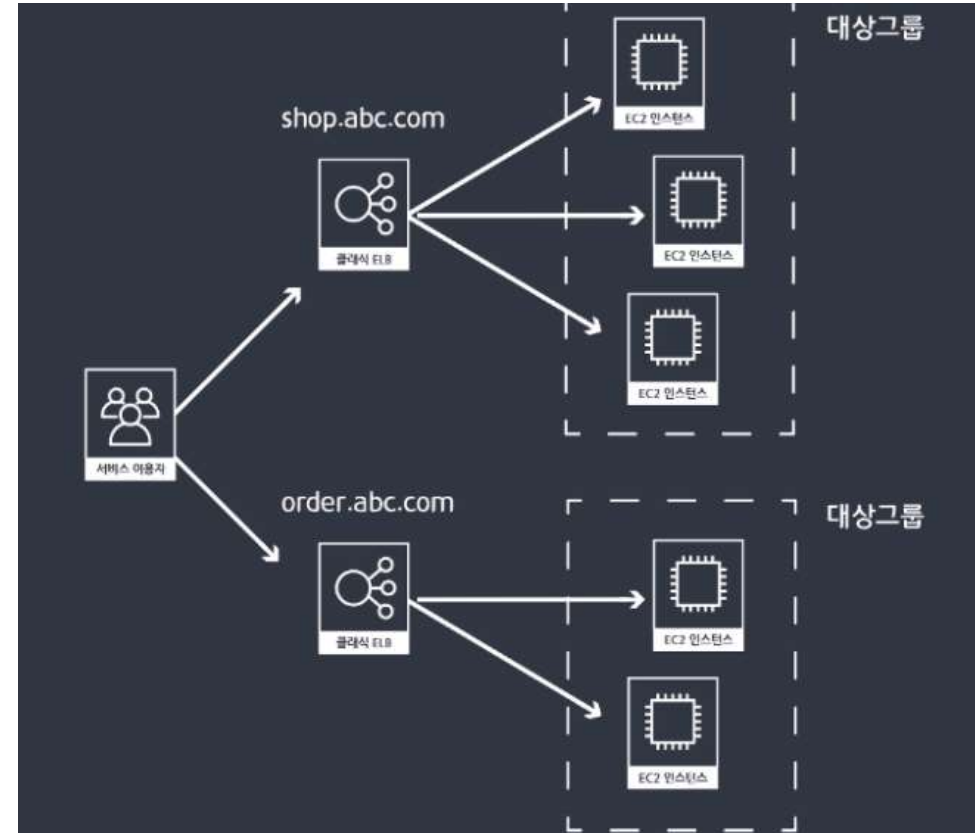


ELB 종류

- Classic Load Balancer(CLB)
- Application Load Balancer(ALB)
- Network Load Balancer(NLB)
- Gateway Load Balancer(GLB)

Classic Load Balancer

- 가장 초기에 서비스 되던 Load balancer
- 서버의 기본 주소가 바뀌면 로드 밸런서를 새로 생성해야한다.
- 하나의 주소에 하나의 대상그룹으로만 보낼 수 있음
- 대상 그룹 주소가 여러 개면 모든 주소마다 로드 밸런서를 뒤편야 한다.
- 로드 밸런서 여러 개를 써야 해서 비용 증가한다.

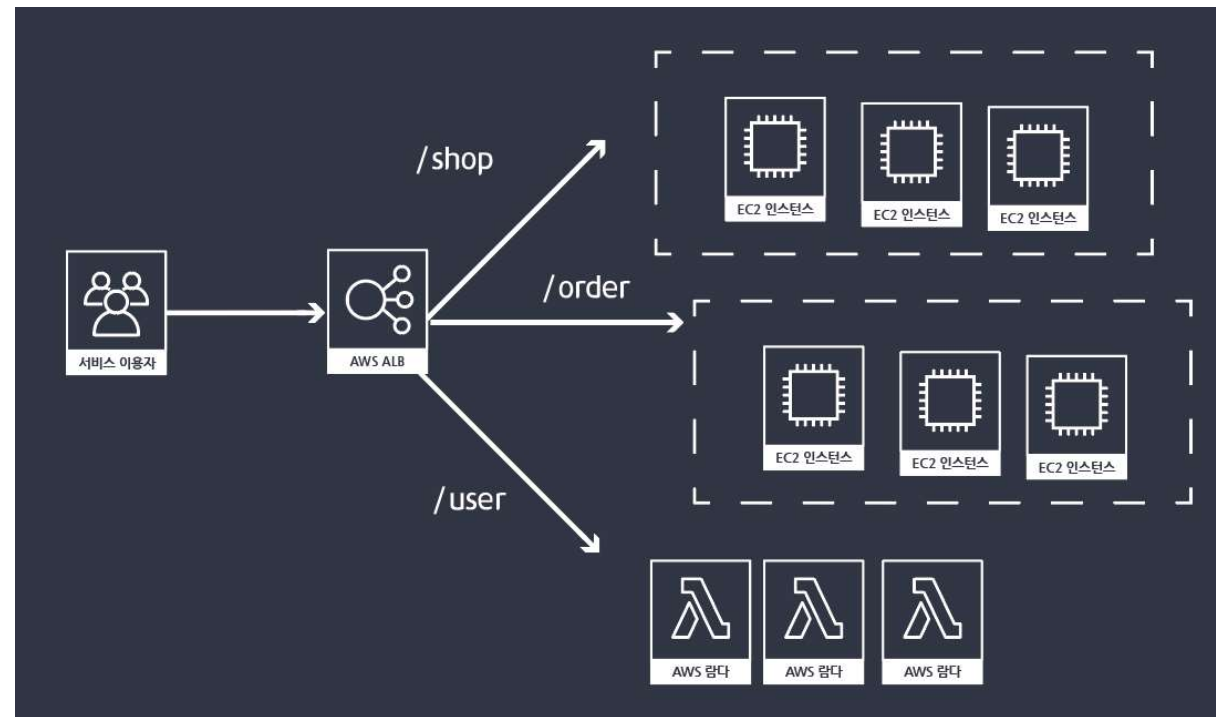


쿠팡에서 회원관리(shop) 인스턴스와 주문(order) 인스턴스가 따로 존재한다고 하자.

로그인 후 주문을 하기 위해서는 그림 처럼 각각 다른 Load Balancer를 거쳐 해당 인스턴스로 접속해야 하므로 서버 개수, 비용이 증가하게 된다.

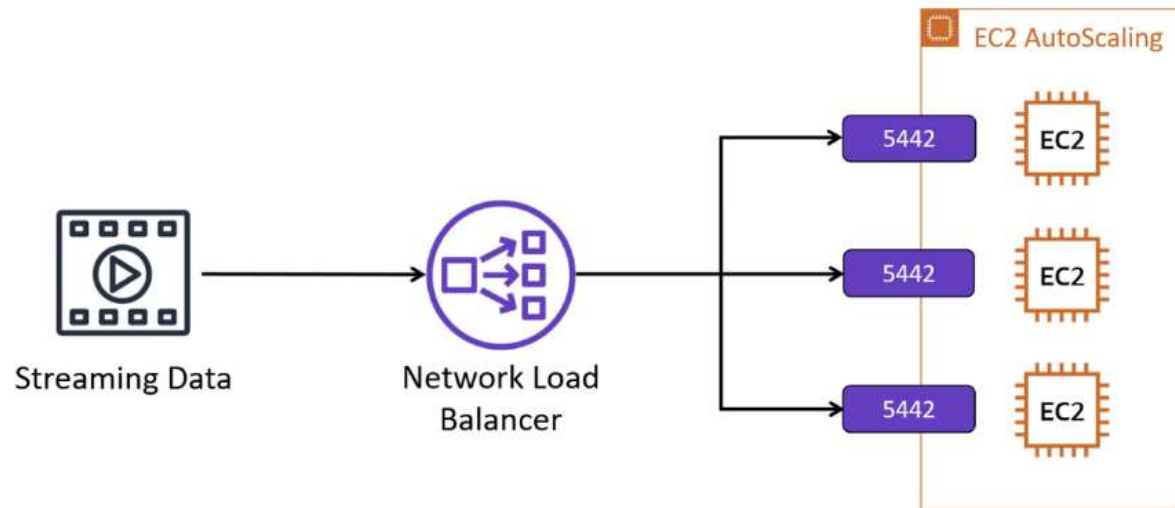
Application Load Balancer

- OSI application layer(L7)
- HTTP/HTTPS 헤더 정보를 이용해 부하분산 할 수 있다. (지능적 라우팅)
- 하나의 로드밸런서로 모든 주소에 라우팅 할 수 있다.
- /user 경로로 오면 람다로 보내고, /shop으로 오면 회원 관리 그룹으로 보낸다.
- Path 뿐만 아니라 port에 따라 다른 대상그룹으로도 매핑 할 수 있다.






Network Load Balancer

- OSI transport layer(L4)
- TCP와 UDP 계층에서 요청 부하를 분산한다. HTTP 헤더를 해석하지 못함
- Latency가 낮아서 고성능을 요구하는 환경에 적합함
- 고정 IP 주소를 사용할 수 있다
- 주소 별로 라우팅 하지 못함.
- L4라서 SSL 적용할 수 없다.



ALB vs NLB

- ALB가 기능이 더 많다.
- NLB가 더 빠르다.

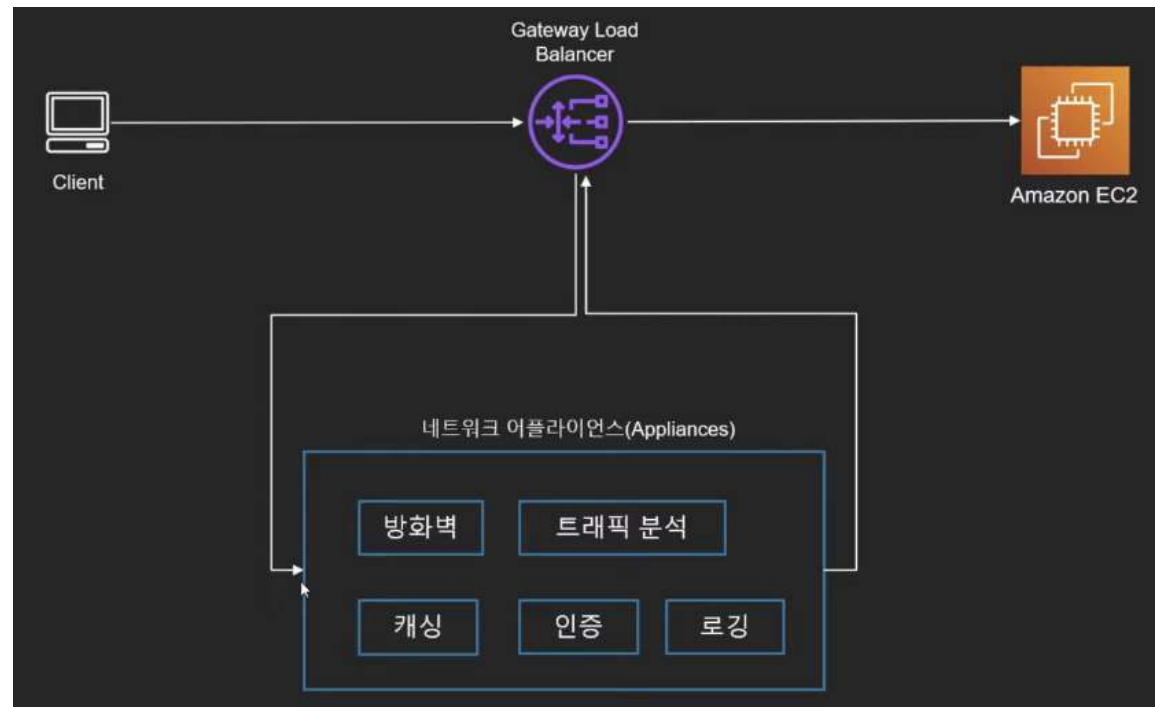
	 Application Load Balancer	 Network Load Balancer	 Classic Load Balancer
프로토콜	HTTP, HTTPS, HTTP/2	TCP, TLS	HTTP, HTTPS, TCP
EC2-Classic 지원	X	X	O
쿠키기반 스티키 세션 지원	O	X	O
타깃그룹 지원	O	O	X
WebSocket 지원	O	X	X
타깃으로 IP주소 지원	O	O	X
Content 기반 routing 지원	O	X	X
고정IP 지원	X	O	X

ALB : 클라이언트가 웹화면을 요청하는 상황일때 (HTTP,HTTPS 프로토콜을 사용해서 어플리케이션 레벨 접근할때

NLB : 내부로 들어온 트래픽을 처리하고, 내부의 인스턴스로 트래픽을 전송할때

Gateway Load Balancer

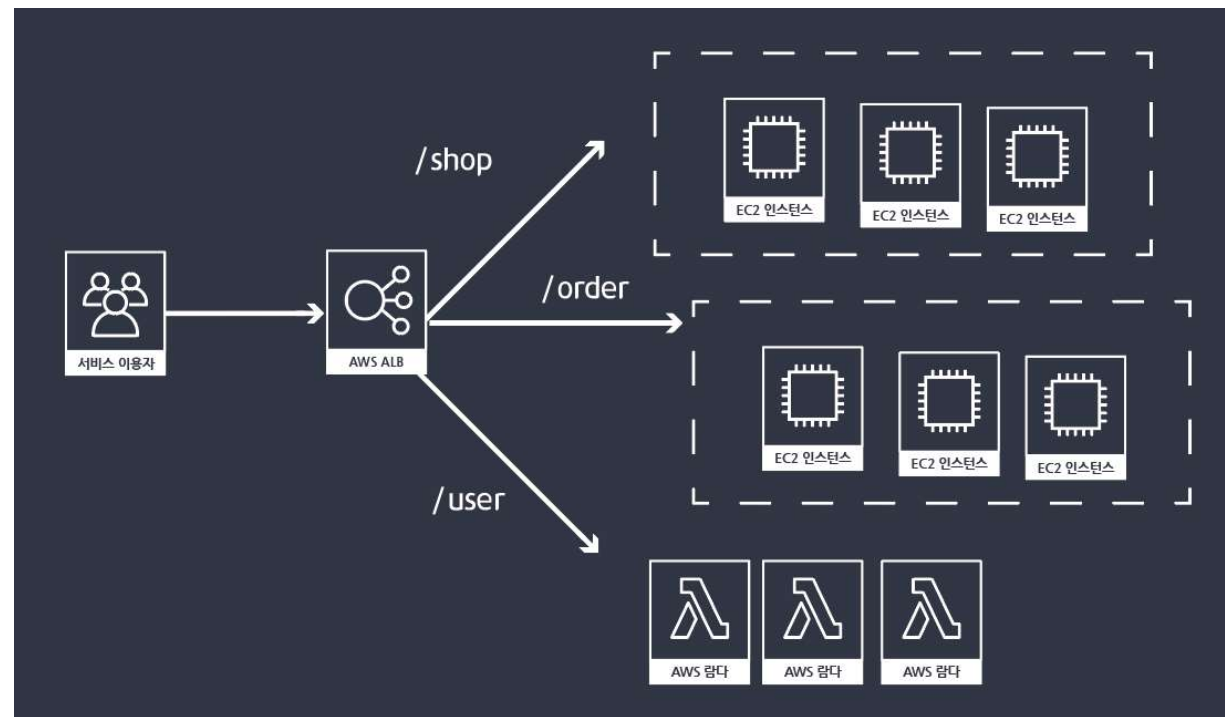
- OSI network layer(L3)
- 일반적인 로드밸런스 역할과는 다르다. 트래픽을 체크하는 기능을 한다고 보면 됨
- 방화벽, 침입 탐지
- 트래픽이 EC2에 도달하기 전에 먼저 트래픽을 검사하거나 인증하거나 로깅하는 작업을 먼저 수행하는 서비스



AWS – Load balancer vs API
gateway 그리고 VPC에 대해

API gateway 꼭 써야 할까?

- ALB만 사용해도 라우팅은 처리할 수 있다
- API gateway를 사용해서 얻는 이점은 있을까?



ALB에는 없는 API gateway만의 기능

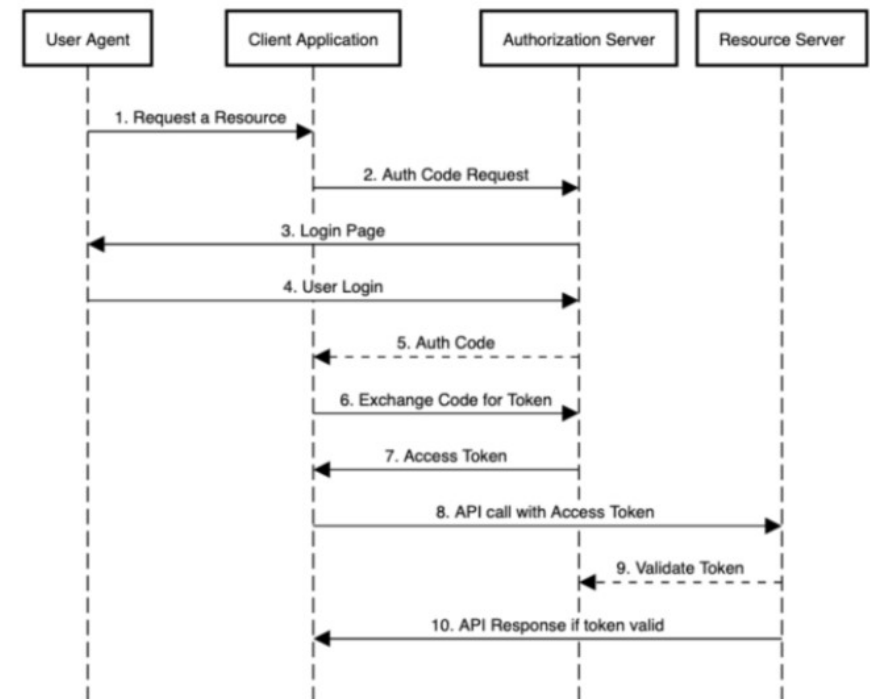
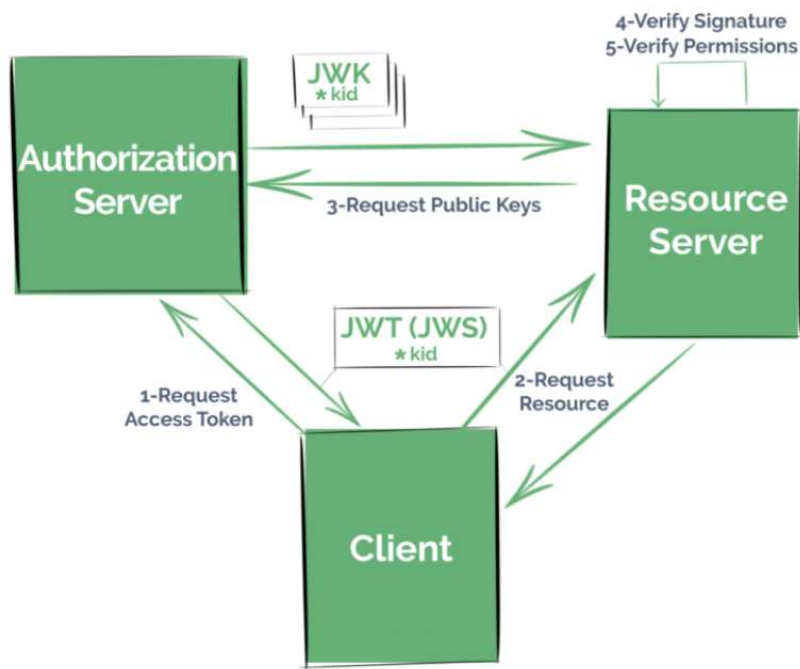
- API gateway를 사용해서 얻는 이점:
 1. Rate limiting: 각 micro service 에 대한 요청에 limit를 걸 수 있음
DoS 공격을 막을 수 있다.
 2. Caching: API gateway는 캐싱 기능도 제공한다. 각 micro service 에 접근 부하를 상당히 줄일 수 있음. 물론 API gateway 혼자 감당하는 요청이 클 테지만, API gateway는 10,000 RPS 까지 견딘다.
 3. Monitoring: API gateway는 metric 수집도 한다. 각 micro service의 성능을 측정하고 진단을 내리도록 도와준다.
 4. Request and response validation: 요청과 응답의 형식이 올바른지 검사해준다. Error를 사전에 예방해준다. 물론 validation은 spring에서 구현할 수 있지만, 이것 앞 단에서 한 방에 처리해주니 편리함

ALB에는 없는 API gateway 이점

5. Authentication and Authorization: 권한이 있는 사용자만 각 micro service에 접근할 수 있도록 한다. 비인가 접근을 방지하여 시스템의 보안을 향상시킴. 사실 API gateway를 사용하지 않아도 spring 내부적으로 모든 서버에서 request filter를 사용하면 되지만, API gateway를 사용하면 global filter를 사용하여 모든 요청에 대해 앞 단에서 처리할 수 있다.
6. Circuit breaker: 하나의 micro service fail이 전체 시스템을 죽이는 것을 방지한다. 각 micro service의 health를 체크해서 하나의 micro service가 죽으면 백업으로 되돌리는 기능을 함

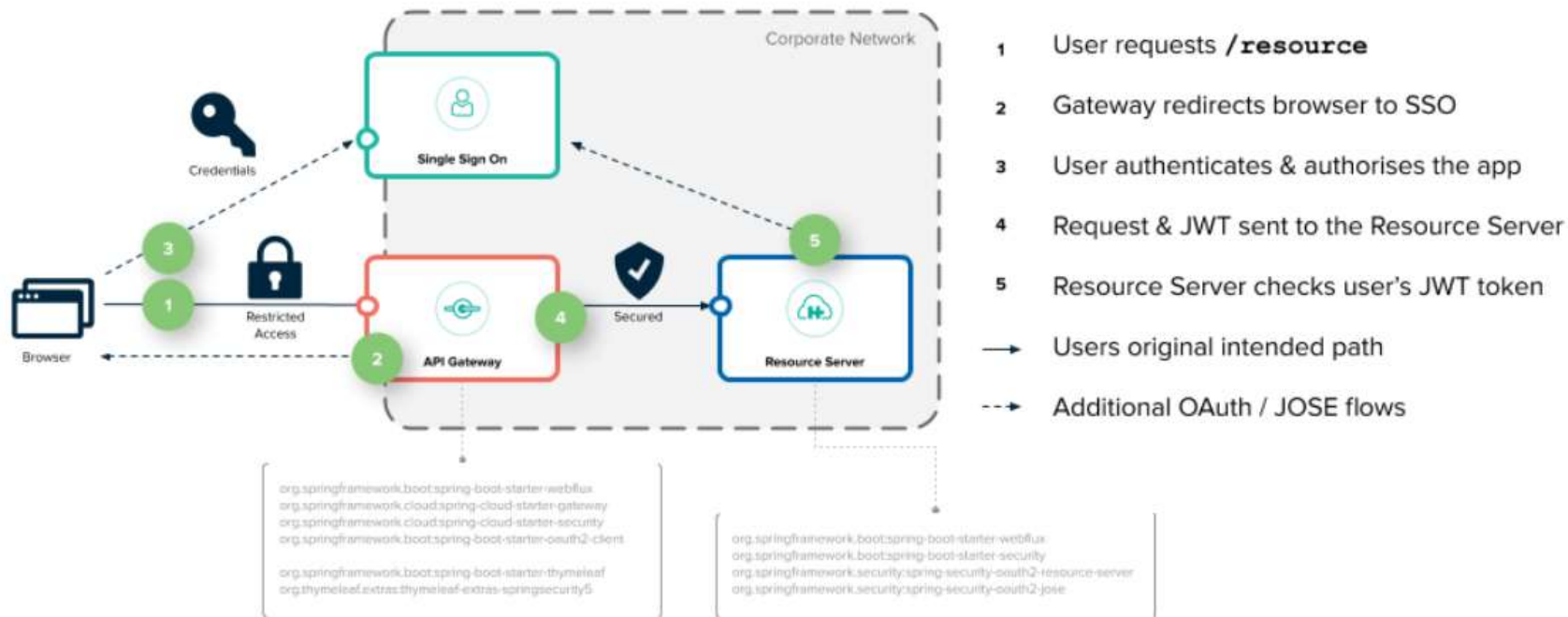
Authentication and Authorization

- 보통 인증/인가를 구현할 때 아래와 같은 구조를 많이 사용한다. 리소서 서버와 인증/인가 서버를 따로 두고 Auth token을 client가 받으면 그 토큰으로 resource server에 접근하게 한다.



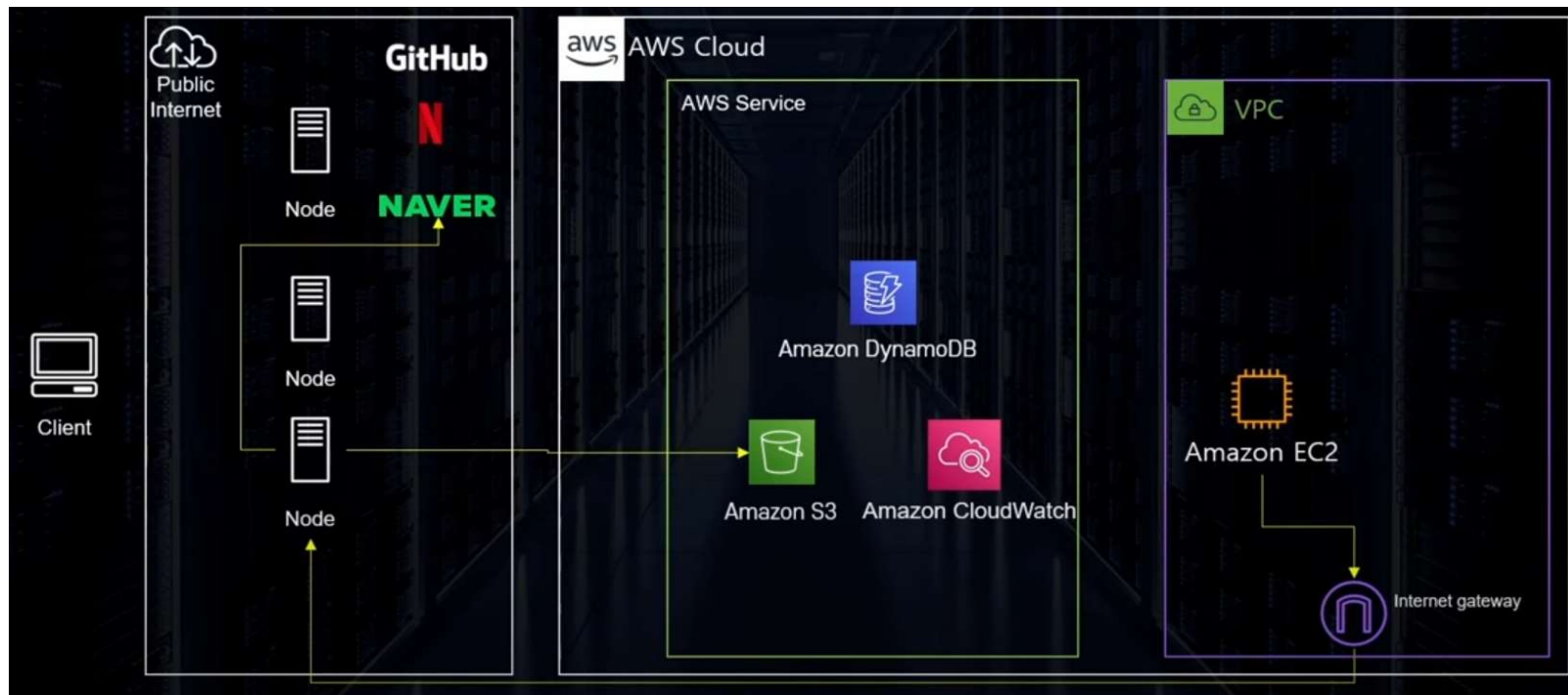
Authentication and Authorization

- API gateway를 사용하면 아래와 같이 API gateway가 SSO 서버와 연동되어 동작하면서 요청을 자동으로 Resource server에 보내준다.



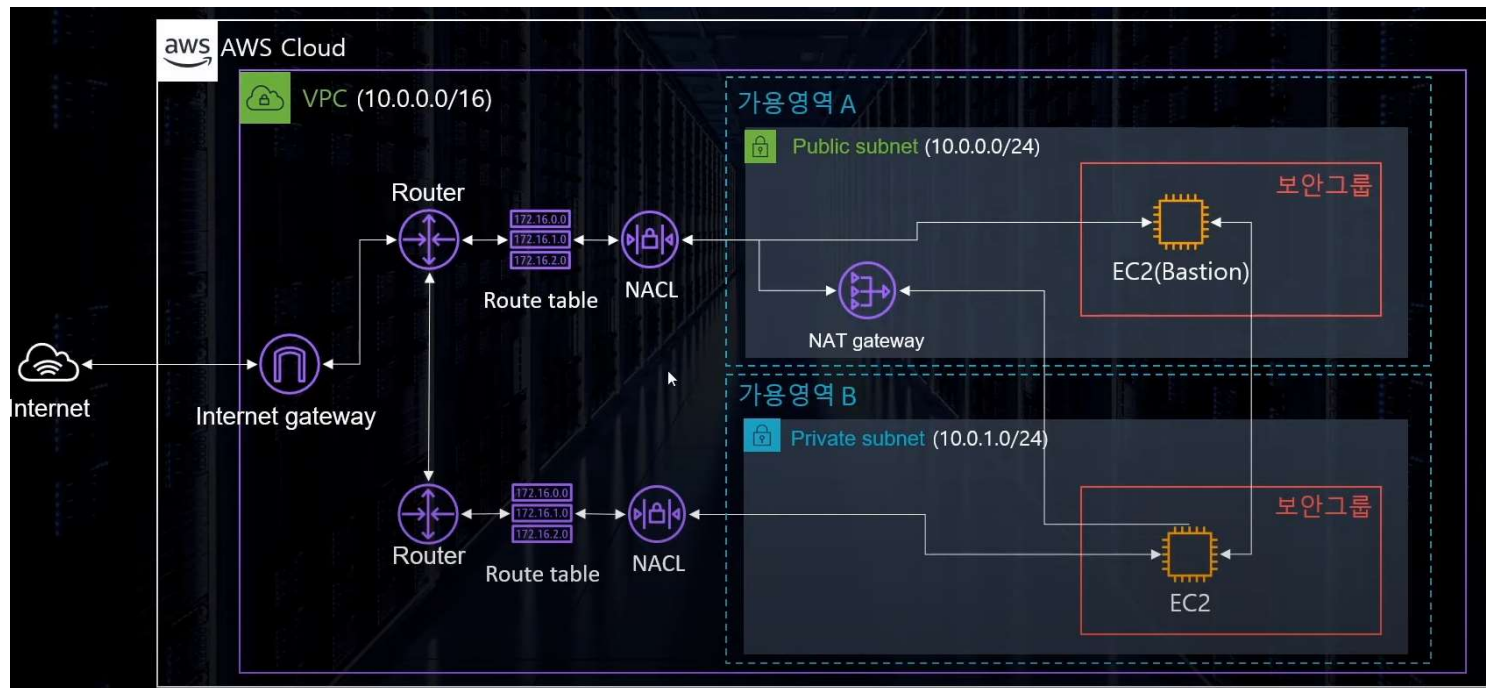
VPC

- VPC: 가상으로 존재하는 데이터 센터
- 대부분 AWS 서비스는 public internet으로 접근 가능
- VPC를 사용하면 외부와 격리된 서비스를 구축할 수 있다.



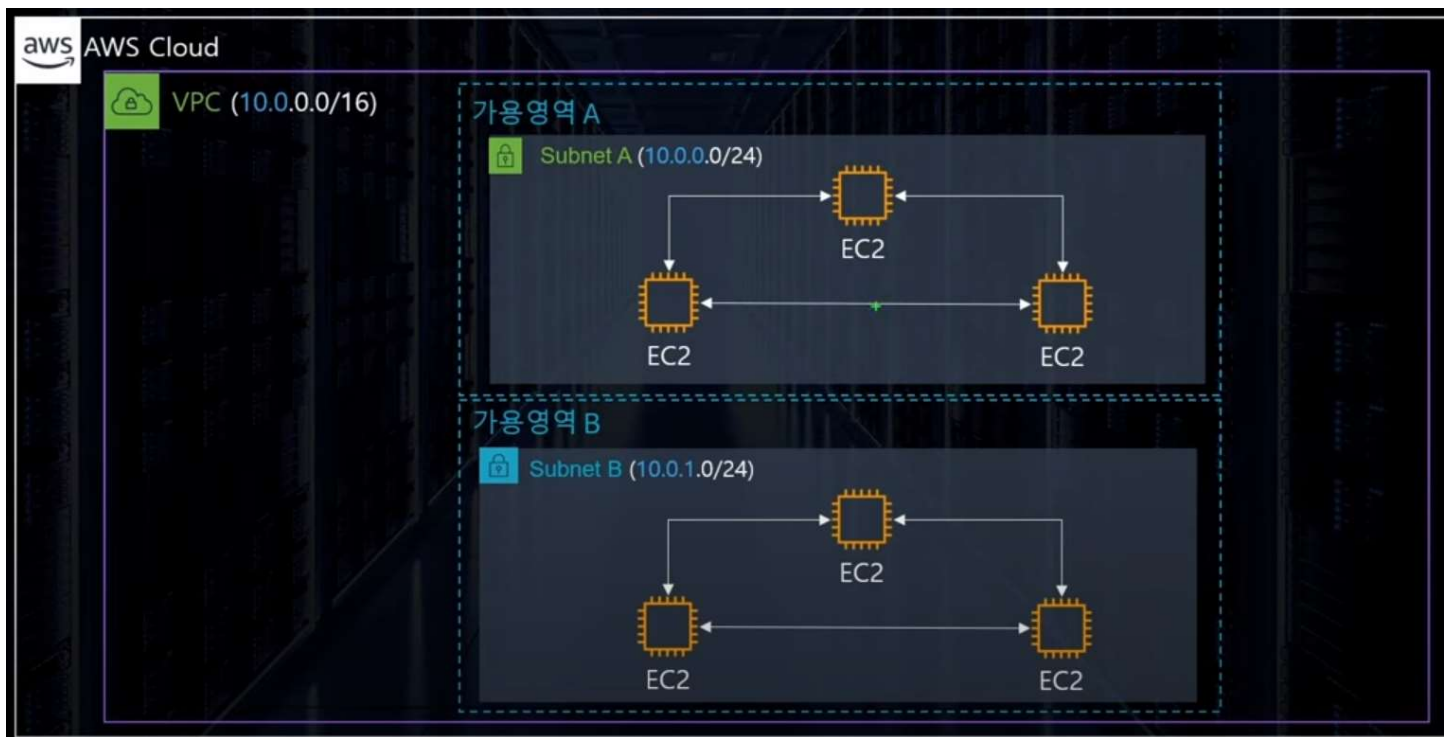
VPC와 서브넷

- VPC 내부에 원하는 대로 사설망을 구축할 수 있음
- 하나의 VPC에는 여러 서브넷이 들어갈 수 있음
- 서브넷에는 주로 EC2, lambda와 같은 컴퓨팅 서비스가 들어감



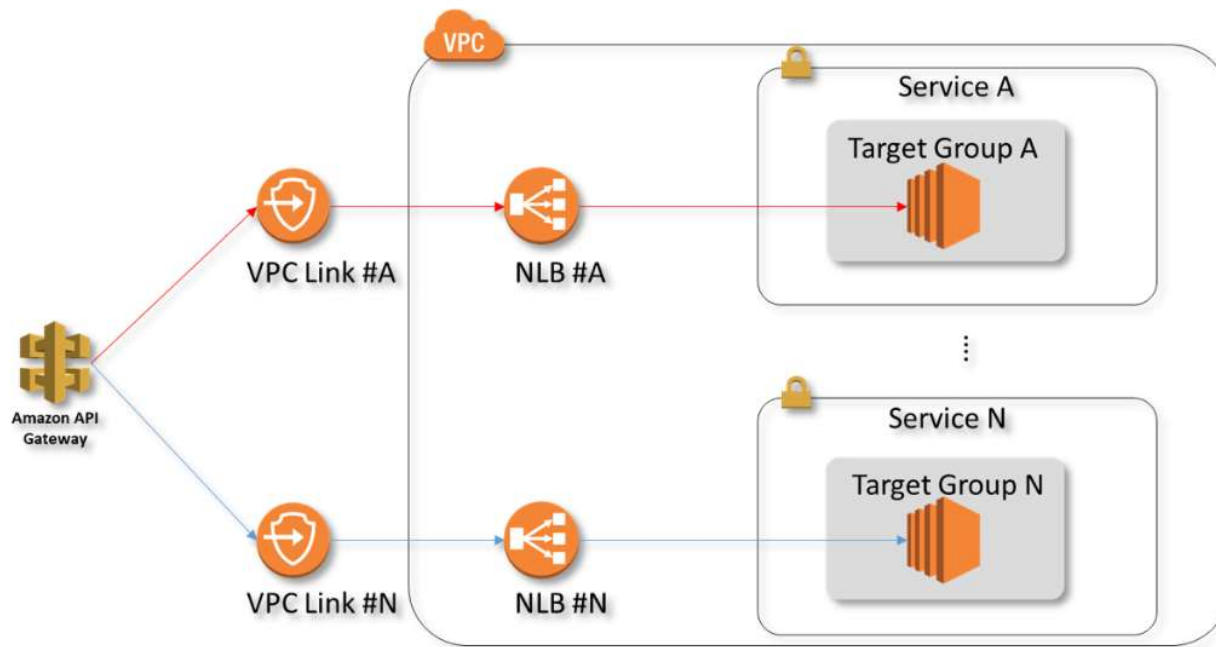
서브넷

- 서브넷이란 VPC의 하위 단위
- 하나의 서브넷에는 하나의 AZ(가용영역)안에 위치한다.



VPC 관점에서 Load balancer vs API gateway

- API gateway는 VPC 밖에 있고, Load balancer는 VPC 안에 있다.
- API gateway는 VPC link를 통해 VPC 내부와 통신한다.



AWS – EC2, Fargate, Lambda

EC2, Fargate, Lambda

셋 다 AWS에서 지원하는 컴퓨팅 서비스

- EC2: 독립된 컴퓨터를 임대해주는 서비스. 인스턴스를 설정해야 한다. 인스턴스는 컴퓨터의 사양을 의미한다.
- Lambda: 별도의 서버 세팅 없이 코드를 곧바로 함수로 실행하게 해주는 서비스. serverless
- Fargate: 기본 인프라를 관리할 필요 없이 컨테이너를 배포하고 관리할 수 있는 서비스. serverless

EC2(Elastic compute cloud)

- EC2는 정말 가상의 컴퓨터 한 대를 빌린다고 보면 된다.
- 초기 구입비와 설정비용이 전혀 없고, 사용한 만큼만 내면 됨.
- 사용 시간에 비례하여 요금을 책정함. 초 단위로 책정
- AMI(이미지)기능을 지원한다. 새로운 컴퓨터를 추가 구입할 때 AMI를 가져오면 AMI에 저장된 설정을 그대로 가져와서 번거롭게 프로그램을 새로 설치하지 않아도 됨.
- 인스턴스를 선택해야한다. CPU, memory 등 하드웨어 스펙을 취사선택할 수 있다. 인스턴스에는 유형과 사이즈가 나뉘어져 있어서 각각 선택할 수 있다.

EC2 인스턴스 유형

- 인스턴스 타입마다 특화된 기능이 다르다.

	범용				컴퓨팅 최적화				메모리 최적화					저장 최적화		
타입	t	m	A1	Mac	c	f	Inf	g	r	x	p	z	u-6tb1	h	i	d
설명	저렴한 범용	범용	ARM 기반	맥 기반	컴퓨팅 최적화	하드웨어 가속	머신러닝용	그래픽 최적화	메모리 최적화	메모리 최적화	그래픽 최적화	고주파수 컴퓨팅 워크로드	대용량 메모리 인스턴스	디스크쓰루풋 최적화	디스크 속도 최적화	디스크 최적화
예시	웹서버, DB	어플리케이션 서버	ARM 기반 에코시스템 워크로드/웹서버 등		CPU 성능이 중요한 어플리케이션/DB	유전 연구, 금융 분석, 빅데이터 분석	머신러닝	3D 모델링/인코딩	메모리 성능이 중요한 어플리케이션/DB	Spark	머신러닝, 비트코인	EDA에 플레키에션	가상화 오버헤드를 줄여주는 베어메탈	하둡/맵리듀스	NoSql/데이터 웨어하우스	파일서버/데이터 웨어하우스/하둡

- T가 저렴한 범용 서버이므로 연습용으로 사용하기 적절함

EC2 인스턴스 사이즈

- 원하는 성능과 용량을 고려해서 선택하면 된다.

인스턴스	vCPU*	시간당 CPU 크레딧	메모리 (GiB)	스토리지	네트워크 성능
t2.nano	1	3	0.5	EBS 전용	낮음
t2.micro	1	6	1	EBS 전용	낮음에서 중간
t2.small	1	12	2	EBS 전용	낮음에서 중간
t2.medium	2	24	4	EBS 전용	낮음에서 중간
t2.large	2	36	8	EBS 전용	낮음에서 중간
t2.xlarge	4	54	16	EBS 전용	중간
t2.2xlarge	8	81	32	EBS 전용	중간



serverless

- EC2는 하드웨어만 빌리는 것 뿐이므로 그 안에 들어갈 소프트웨어 설정은 모두 개발자가 해줘야 한다.
- 서버리스는 SW 설정이 필요 없음. 모든 백엔드를 함수 별로 쪼개서 작동시킨다. 유저가 들어오는 수만큼 함수를 작동시킴
- 서버리스는 24시간 동작하지 않는다. 평소에는 sleep 모드로 되어있다가 요청이 들어올 때만 깨어나서 작동한다. 대기 비용이 들지 않는다. 함수를 사용한 만큼만 비용을 지불하면 됨.
- 서버 운영 자율성은 좀 떨어진다. 하나의 서버리스를 사용하다가 다른 서버리스로 갈아타는 게 좀 어려움
- 빠르게 프로토타입을 만들기에는 유리함.

lambda

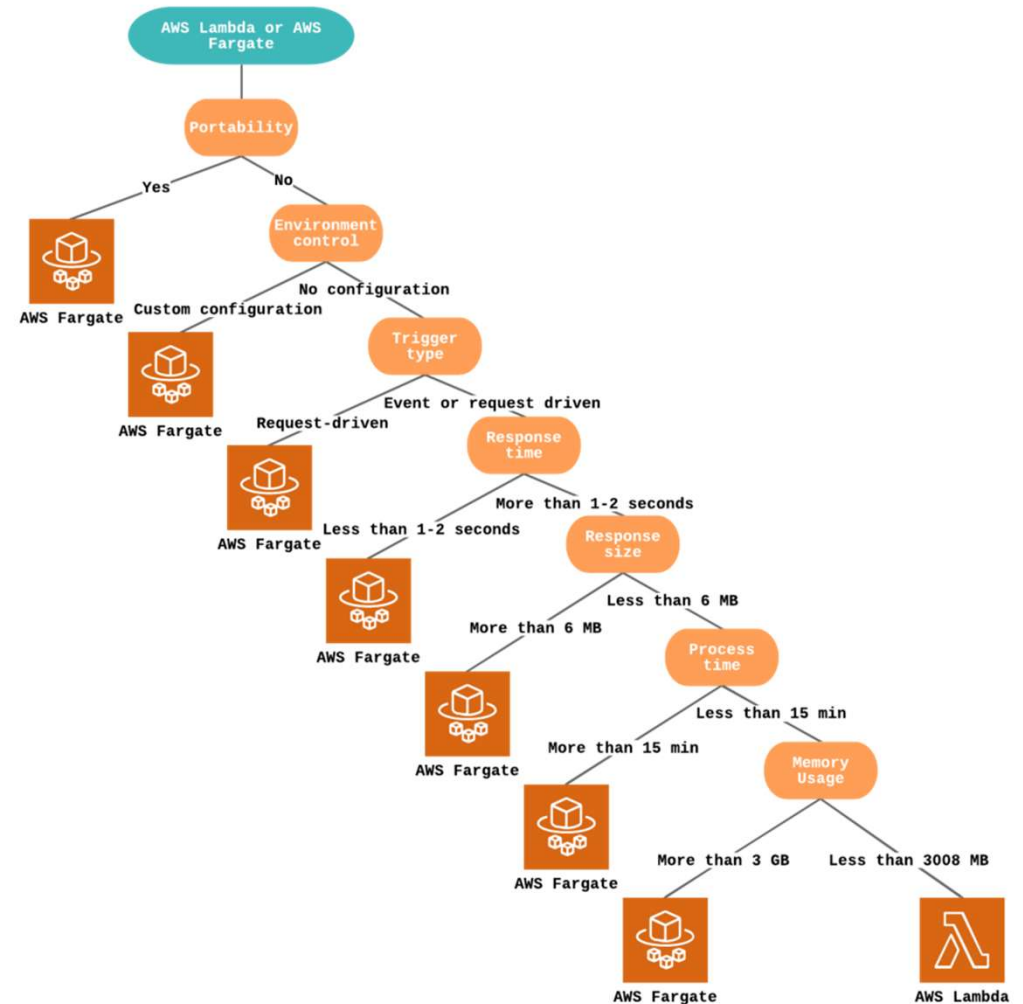
- EC2와는 다르게 복잡한 서버 설정 없이 바로 사용할 수 있음
- 사용 시간에 비례하여 요금을 책정함. 초 단위로 책정
- 이벤트 기반 작업을 처리하는데 유용함.
- 어플리케이션 배포 크기가 50MB이하, 실행시간이 15분 이하인 경우에 사용에 적절함. 대규모 데이터 처리에는 부적절함
- Cold start 문제가 있다. 평소에는 sleep 모드로 있기 때문.
- 물론 cold start 문제를 해결하기 위해 자주 사용하는 함수는 sleep하지 않도록 하는 방법도 있음

fargate

- Serverless. 기본 인프라를 관리할 필요 없이 컨테이너를 배포하고 관리
- EC2처럼 세밀한 설정은 못 하지만, CPU와 메모리 구성 정도는 선택 할 수 있다.
- Lambda 보다 응답 속도가 빠르다.
- Lambda 보다 대규모 데이터 작업 처리에 능하다.

Fargate vs Lambda

- Fargate와 Lambda 비교 선택
- 환경설정이 필요하다면 Fargate
- 응답 속도가 중요하다면 Fargate
- 데이터 처리 부하가 크다면 Fargate
- 메모리 사용량이 3GB 넘어간다면 Fargate
- 그 외에 간단한 데이터를 이벤트성으로 처리하는 것에는 lambda가 유리함



EC2 vs Fargate vs Lambda

- 작업 부하와 유연성 관점에서 비교한 것

