

iOS 세미나 2

2023 와플스튜디오 루키 세미나

2023.10.08, 박신홍

오늘 배울 내용

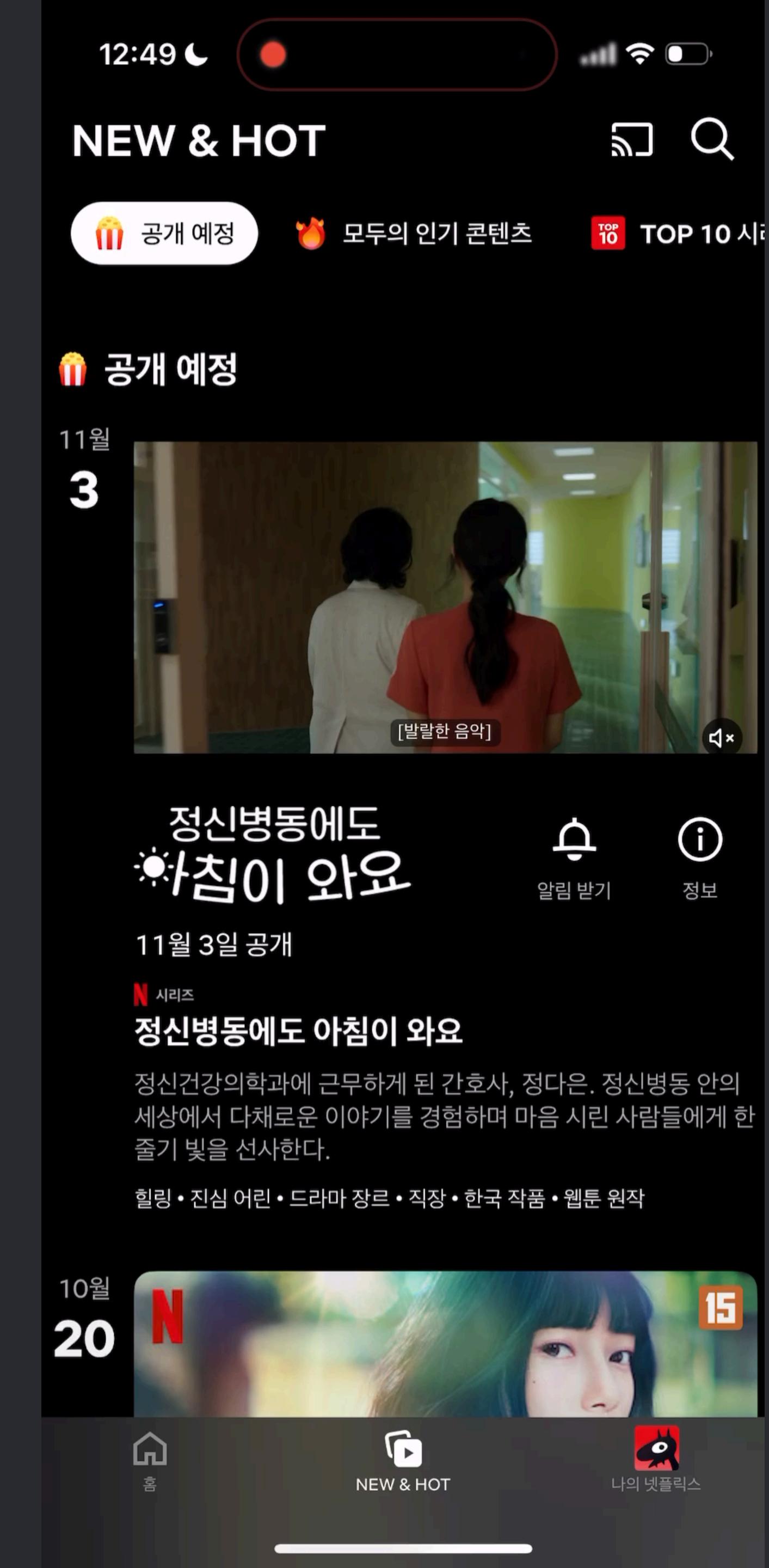
어제 손가락 풀기는 끝났다...

- Networking
 - Server & Client
 - HTTP + REST API
- Asynchronous / Concurrent Programming
 - Threading
 - Grand Central Dispatch
 - Swift Concurrency
 - Synchronization
- UITableView w/ Asynchronous Data

Spoiler

과제 2,3: 넷플릭스 같은거 만들기

- 2개의 탭
 - (홈 탭: 과제 3에서 구현)
 - NEW & HOT 탭
 - Upcoming & Popular 영화 목록 보여주기
 - 키워드 로딩 or 검색 기능
- 넷플릭스 API 대신 TMDB API 사용



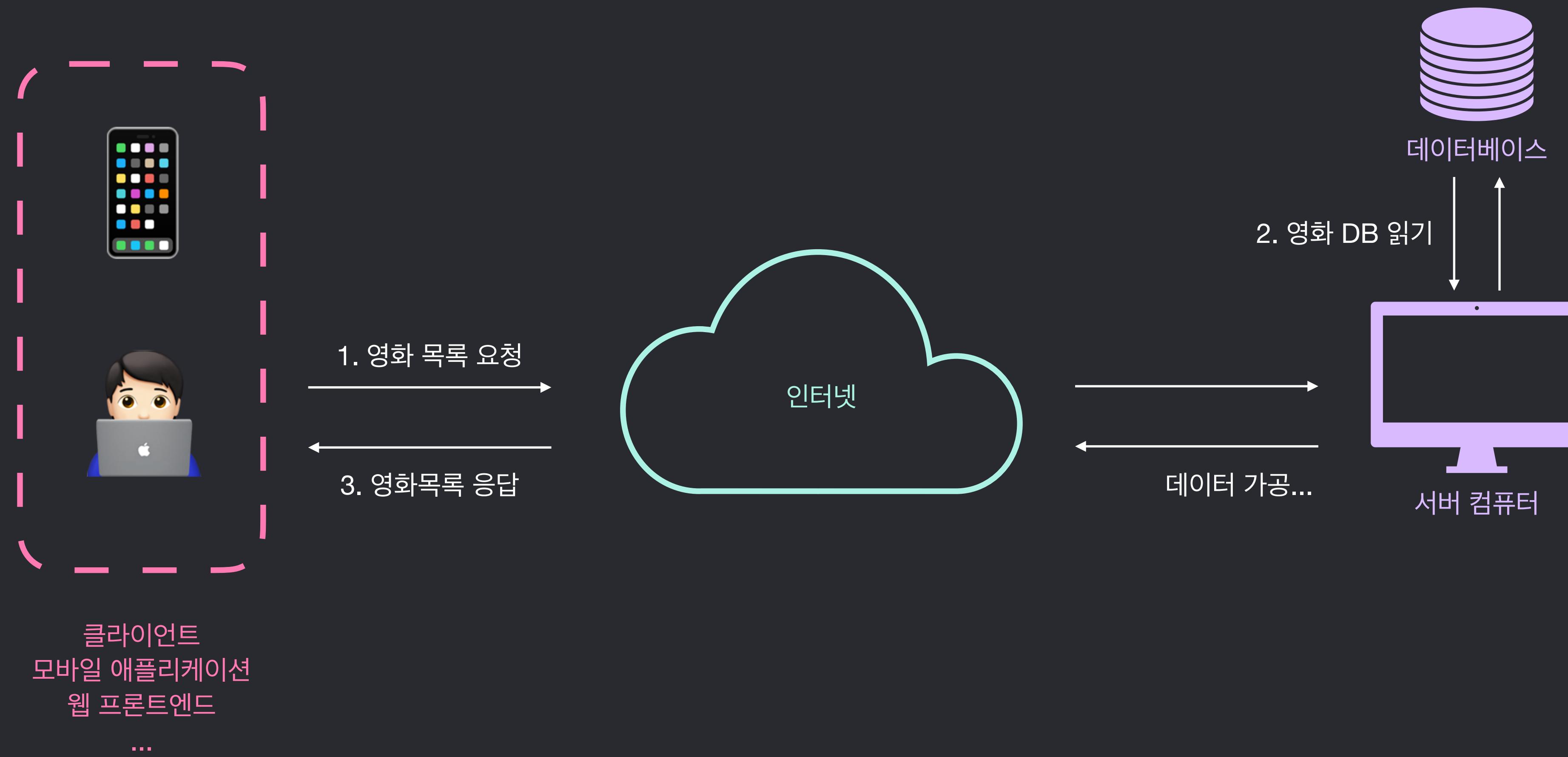
Networking

서버와 클라이언트

- 지금까지 UserDefaults를 사용해서 유저가 생성한 데이터를 저장하고 보여줌
- 근데 앱을 재설치하면? → 전부 사라짐..
- 서버에 데이터를 저장하고, 필요할 때 받아서 보여주어야 함
- 어떻게?

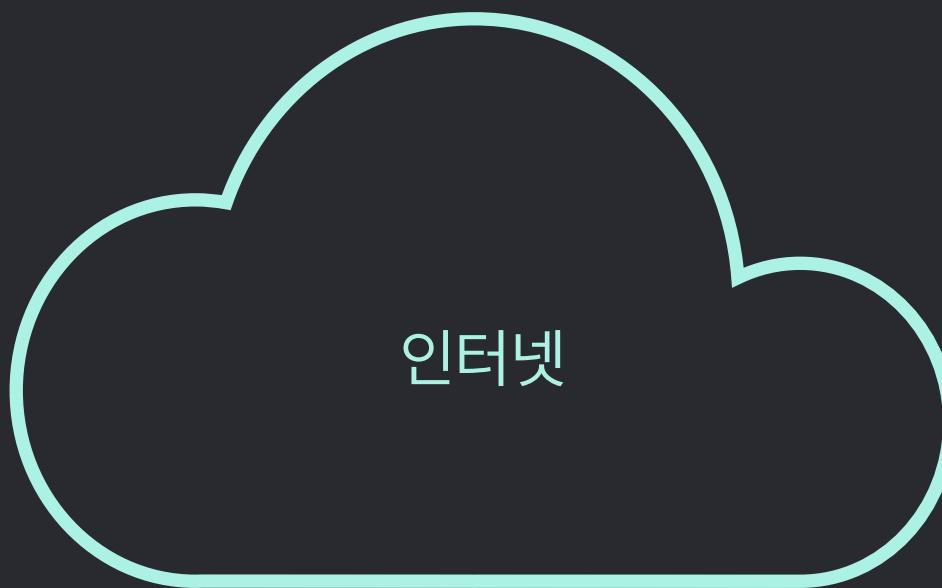
서버와 클라이언트의 작동 방식

매우 간소화된 설명



서버와 클라이언트의 작동 방식

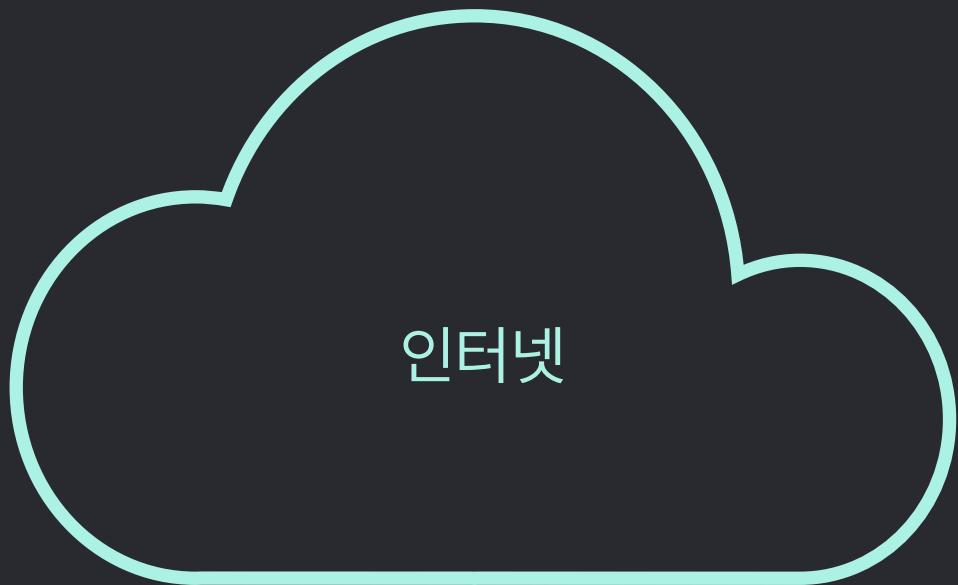
매우 간소화된 설명



- 데이터 전송 방식이 서버마다, 클라이언트마다 중구난방이면 개발이 매우 고통스러워질 것
- 그래서 HTTP (HyperText Transfer Protocol) 이라는 것이 나옴
 - Protocol = 규약
 - HTTPS는 여기에 보안 레이어를 끼얹은 것
- 모든 브라우저와 서버 프레임워크, 네트워킹 라이브러리는 HTTP(S) 통신을 주로 사용함 (e.g. <https://naver.com>)
- 모든 HTTP 요청에는 Method라는 필드가 있음
 - GET, POST, PUT, DELETE ...

REST API

HTTP만으로는 부족했다



- 모두가 HTTP라는 규약을 따르기 때문에 개발이 한결 편해짐
- 그런데 HTTP 자체도 자유도가 꽤 있어서, 잘못하면 API를 중구난방으로 설계하게 됨
 - API를 설계할 때 사용할만 할 관습/패턴/스타일이 없을까?
- **REST API** (Representational State Transfer Application Programming Interface)
 - URL을 특정 리소스에 대한 Identifier로 사용
 - HTTP method를 사용하여 action을 정의
 - 응답은 (일반적으로) JSON

REST API 예시

- GET /users : 모든 유저 불러오기
- GET /users/123 : 아이디가 123인 유저의 상세 정보 불러오기
- POST /users : 유저 생성하기
- PUT /users/123 : 아이디가 123인 유저의 상세 정보 업데이트하기
- DELETE /users/123 : 아이디가 123인 유저 삭제하기

REST API 예시

- Query Parameter
 - GET /users?page=2 : 유저 목록의 두 번째 페이지 불러오기
 - GET /users?name=John&page=2 : John이라는 이름을 가진 유저 목록에서 두 번째 페이지 불러오기
- Request Body
 - POST /users : 아래 Request Body 정보를 가지고 유저 생성하기

```
{  
  "name": "John Doe",  
  "email": "john.doe@example.com",  
  "password": "securepassword123"  
}
```



- TMDB에서 영화 API 생김새 구경하기

네트워크 통신과 비동기

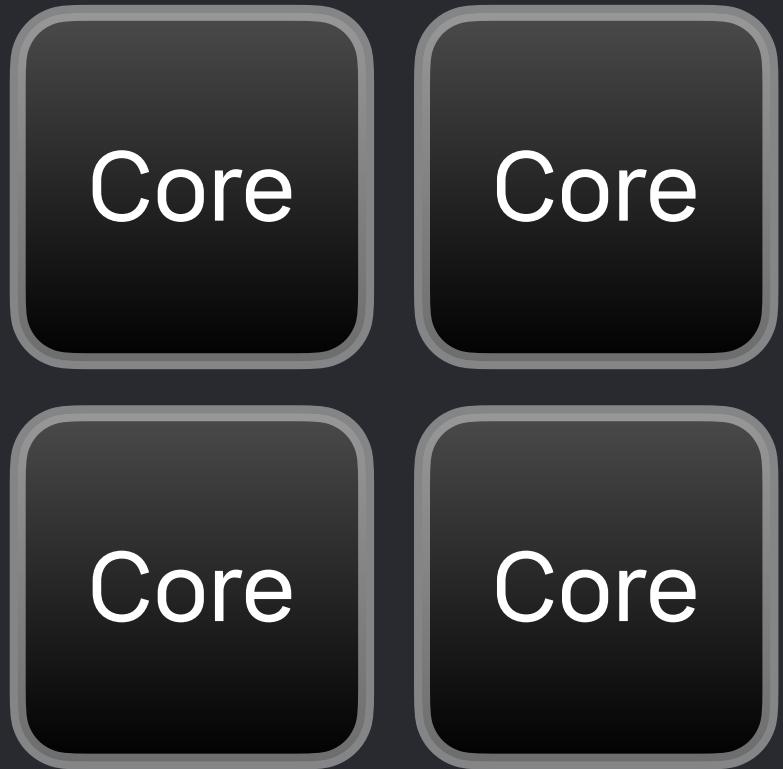
무슨 상관?

- 네트워크 통신은 아주 오래 걸리는 작업임
 - 와이파이가 아무리 빨라도 수십~ 수백 ms가 소요됨
 - 반면 iOS에서는 8ms 마다 새로운 화면을 그려주어야 함
 - (UI 작업이 실행되어야 하는) 메인 스레드에서 코드의 실행 속도가 8ms가 넘어가면 버벅임 발생
- 비동기(async)로 요청을 보내면 응답이 올때까지 기다리지 않아도 됨
 - 요청을 보내놓고, 응답이 돌아오기 전까지 할 일 하다가, 응답이 왔을 때 데이터를 가지고 UI를 새롭게 업데이트해주면 됨
 - e.g. 비동기 회의

비동기/동시성 프로그래밍

CPU Core / Process / Thread

백그라운드



- 싱글코어 CPU일 땐, 코어의 성능을 높이면 프로그램의 속도가 저절로 빨라짐
- 이제는 전력소비와 발열 때문에 코어의 성능을 무작정 올릴 수는 없음
- CPU에 코어 여러개를 땠을 때 려박으면 어떨까? → 멀티코어 CPU
- 이제 프로그램의 성능이 저절로 좋아지진 않음
- 개발자가 앱(=프로세스) 안에서 스레드를 활용해서 멀티코어를 효율적으로 활용할 수 있도록 프로그래밍을 해야 함
- 근데 Swift에서(뿐만 아니라 대부분의 프로그래밍 언어에서) 스레드를 직접 사용하는 건 매우 복잡하고 머리 아픈 일임

An aerial photograph of a large rail yard. The tracks curve from the bottom left towards the top right. Numerous freight cars in various colors—blue, red, green, white, yellow, and brown—are stacked along the tracks. Some cars have white roofs, while others have dark or light-colored sides. The ground between the tracks is a mix of dirt and gravel. Several overhead power lines and poles are visible, running across the yard.

Grand Central Dispatch

Grand Central Dispatch

Thread를 (그나마) 쉽게 사용할 수 있도록..

Main Dispatch Queue

```
{print("hello")}
```

```
{ ... }
```



```
DispatchQueue.main.async {  
    print("hello!")  
}
```

- 스레드 생성, 실행, 종료 등.. 직접 스레드를 관리할 필요가 없음
- DispatchQueue가 알아서 다 해줌

DispatchQueue?

Queue는 Thread가 아니다!

Main Dispatch Queue

```
{print("hello")}
```

```
{ ... }
```



```
DispatchQueue.main.async {  
    print("hello!")  
}
```

- 위 코드의 의미: "main 스레드를 관리하는 queue에다가 print 블록을 비동기적으로 enqueue한다."
- print 블록이 언제 실행될지는 전적으로 시스템이 관리함

DispatchQueue?

Queue는 Thread가 아닙니다!

```
let concurrentQueue = DispatchQueue(label: "CQ",  
                                    attributes: .concurrent)  
for i in 1...3 {  
    concurrentQueue.async {  
        print("hello \(i)")  
    }  
}
```



Concurrent Queue

{print("hello 1")}

{print("hello 2")}

{print("hello 3")}

Sync, Async, Serial, Concurrent

sync / async is about caller's execution

```
let queue = DispatchQueue(...)  
print("1 executed")  
queue.sync {  
    print("2 executed")  
}  
print("3 executed")
```

sync

```
let queue = DispatchQueue(...)  
print("1 executed")  
queue.async {  
    print("3 executed")  
}  
print("2 executed")
```

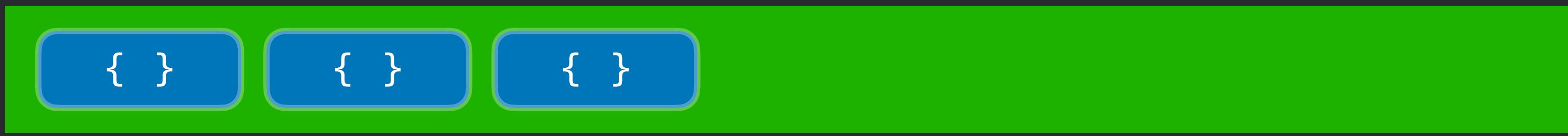
async

- Sync blocks the execution of caller thread, while async doesn't.
- Both sync / async operations are atomic; it's thread-safe.

Sync, Async, Serial, Concurrent

serial / concurrent is about callee's execution

Concurrent Queue

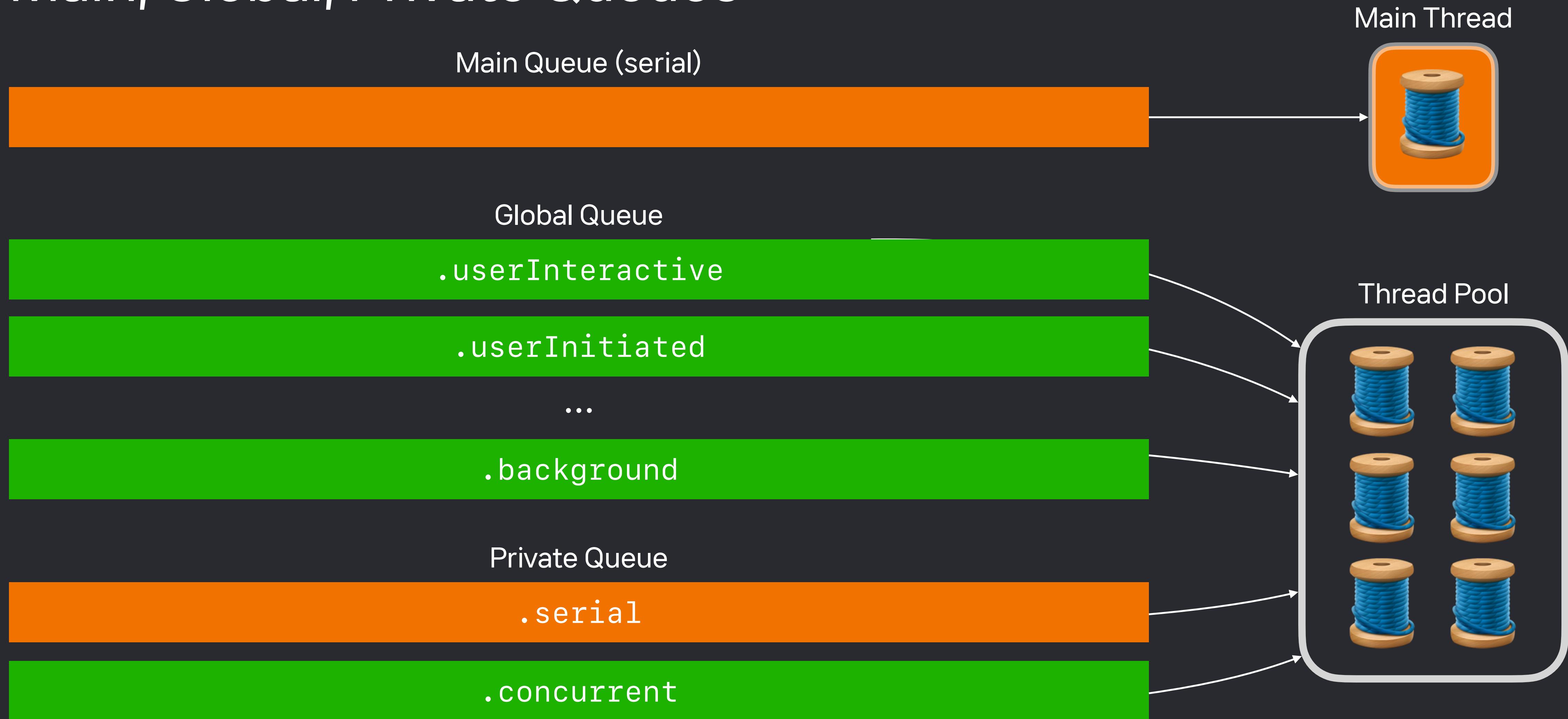


Serial Queue



Types of Queue

Main, Global, Private Queues



Swift Concurrency



Swift Concurrency

async / await

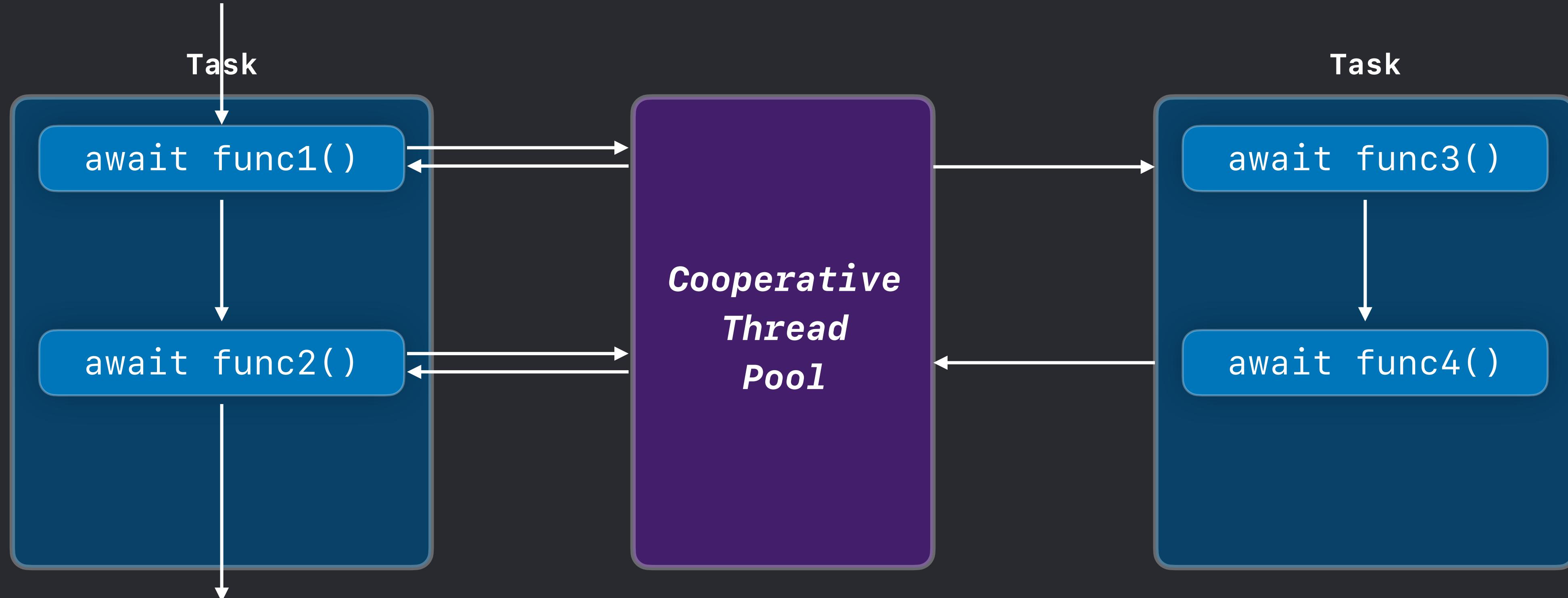
```
func processImageData(completionBlock: (_ result: Image) -> Void) {
    loadWebResource("dataprofile.txt") { dataSource in
        loadWebResource("imagedata.dat") { imageSource in
            decodeImage(dataSource, imageSource) { imageTmp in
                dewarpAndCleanupImage(imageTmp) { imageResult in
                    completionBlock(imageResult)
                }
            }
        }
    }
}

func processImageData() async throws -> Image {
    let dataSource = try await loadWebResource("dataprofile.txt")
    let imageSource = try await loadWebResource("imagedata.dat")
    let imageTmp = try await decodeImage(dataSource, imageSource)
    let imageResult = try await dewarpAndCleanupImage(imageTmp)
    return imageResult
}
```

Swift Concurrency

Task

```
Task {  
    await func1()  
    await func2()  
}
```



- A task is the basic unit of concurrency, that provides async context in a synchronous code.
- Every asynchronous function is executing in a task.

사용할 라이브러리

- Alamofire: 네트워크 라이브러리
 - 내장된 URLSession을 사용해도 과제는 충분히 가능하지만, 각종 편의 기능을 제공하므로 사용하면 편함
- Kingfisher: 이미지 비동기 다운로드 / 캐싱
 - 마찬가지로 이미지 관련 풍부한 편의 기능. 캐싱하는 것 꽤 복잡하고 귀찮은 일이지만 알아서 해줌



Putting things together