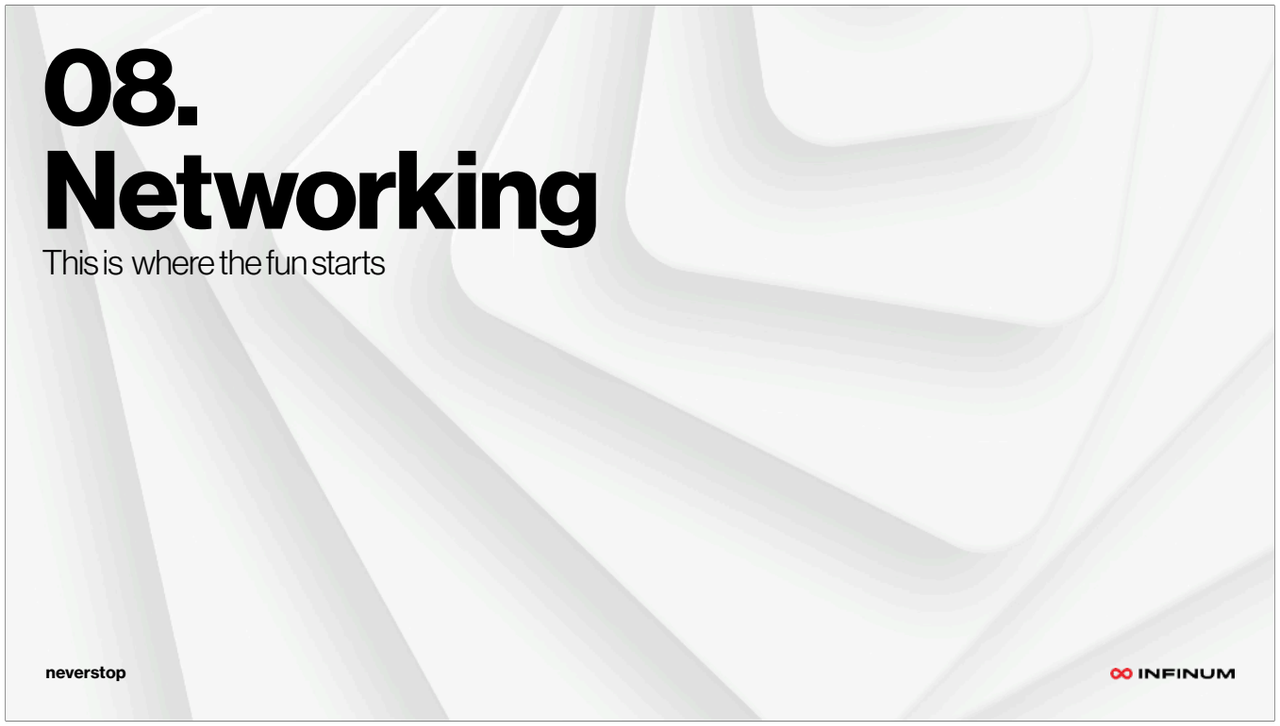


- First part of the lecture
    - presentation
- Second part of the lecture
    - live coding - networking and maybe some navigation

00

Closures

neverstop

- Will be used extensively through the networking
-

Closures

- self-contained blocks of functionality
- blocks of code that **can** be called later
  - **asynchronously**
  - **escaping closures**
- similar to lambdas in other languages
- capture and store references
  - constants and variables
  - context in which they're defined
- functions in Swift are special cases of closures

```
{ (parameters) -> return type in
    statements
}
```

neverstop

- definition of a closure
- input parameters
- statements
- and return value

```
let myClosure: (Int, Int) -> Void = { (int1: Int, int2: Int) -> Void in
    print("\(int1) \(int2)")
}
```

```
let myClosure: (Int, Int) -> Void = { (int1: Int, int2: Int) in
    print("\(int1) \(int2)")
}
```

```
let myClosure: (Int, Int) -> Void = { int1, int2 in
    print("\(int1) \(int2)")
}
```

```
let myClosure: (Int, Int) -> Void = {
    print("\($0) \($1)")
}
```

```
myClosure(1, 2)
```

neverstop ∞

- multiple ways for writing the same closure

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

let reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

```
let reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

```
let reversedNames = names.sorted(by: { $0 > $1 })
```

```
let reversedNames = names.sorted { $0 > $1 }
```

```
let reversedNames = names.sorted(by: >)
```

neverstop ∞

- synchronous closures (non-escaping)

Trailing closure syntax

```
func someFunctionThatTakesAClosure(closure: (Int) -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: { (int1: Int) in
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() { int1 in
    // trailing closure's body goes here
}
```

neverstop ∞

- in case if a closure is the last argument in function definition

Asynchronous execution

```
// This will be executed immediately
MBProgressHUD.showAdded(to: view, animated: true)

// Here we are scheduling some work which will be executed after 4 seconds.
// Scheduling will be executed immediately.
// Work is defined through closure - in our case, hiding the spinner.
// Closure will be executed asynchronously, after 4 seconds.

DispatchQueue.main.asyncAfter(deadline: .now() + 4, execute: {
    // This is a closure which will be executed after 4 seconds.
    // Here we are capturing current context (self) where we will
    // hide the spinner.
    MBProgressHUD.hide(for: self.view, animated: true)
})

// Or shortly, using trailing closure syntax
DispatchQueue.main.asyncAfter(deadline: .now() + 4) {
    MBProgressHUD.hide(for: self.view, animated: true)
}
```

Capture list

- closure captures reference to variables
  - **strong** by default
- helpful when dealing with retain cycles & memory leaks

Example - API call

```
MBProgressHUD.showAdded(to: view, animated: true)

Alamofire
    .request(...)
    .validate()
    .responseDecodable(of: LoginResponse.self) { [weak self] response in
        guard let self = self else { return }
        MBProgressHUD.hide(for: self.view, animated: true)
        ...
    }
```

Threading/Concurrency

- available through closures and Grand Central Dispatch (**GCD**)
- you don't work directly with threads, rather queues
- **DispatchQueue**
- schedule work (**closure**) on some of queues (**DispatchQueue** instances)
- **main queue**
  - queue used for UI updates
  - only one thread - always the same one
  - **main thread**

```swift
// Start the image download on global queue. Everything UI related
// runs on the main queue. You don't want to block the main queue,
// otherwise users will experience jitter in your app.
DispatchQueue.global().async { [weak self] in
    let url = URL(string: "https://www.apple.com/apple.jpg")!
    // Fetch the image from internet
    let data = try? Data(contentsOf: url)
    // Check if data exists, if not return
    guard let data = data else {
        return
    }
    // Return back to main queue since all UI updates
    // should happen only on main queue (main thread).
    // Otherwise you'll get into an undefined state or even a crash.
    DispatchQueue.main.async { [weak self] in
        let image = UIImage(data: data)
        self?.myImageView.image = image
    }
}
```

01

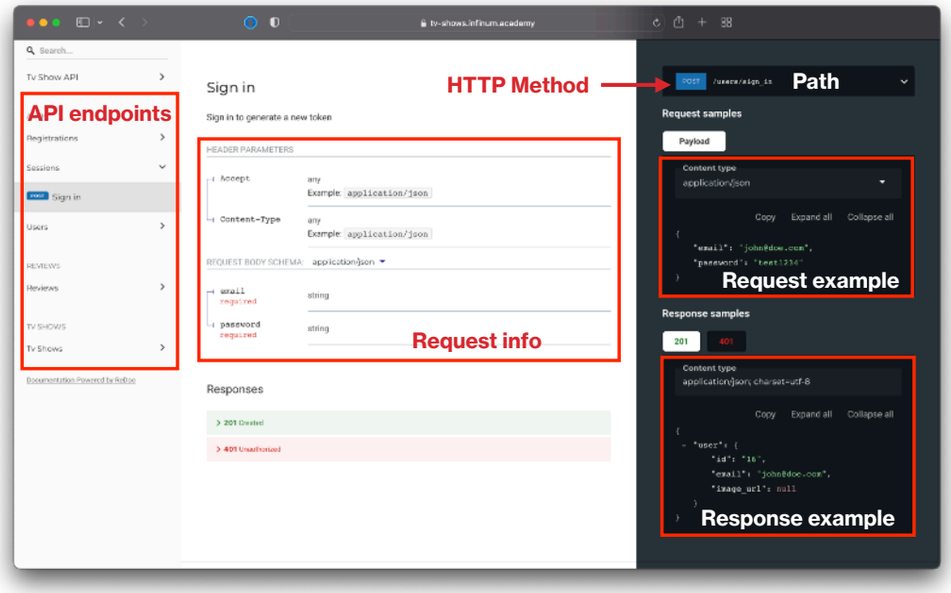
Backend API

TV Shows API Documentation

- https://tv-shows.infinum.academy/api/v1/docs/index.html
- documentation above has much more stuff than what we need for the app
- link on top of the materials repository as well
- **an interface between database and your app**

neverstop

- accompanying backend for out app
- holds all data about shows and users
- an interface between database and your app



neverstop

- go over documentation
- HTTP method
- path
- JSON
- sign in and access token, stay signed in, cookies

02

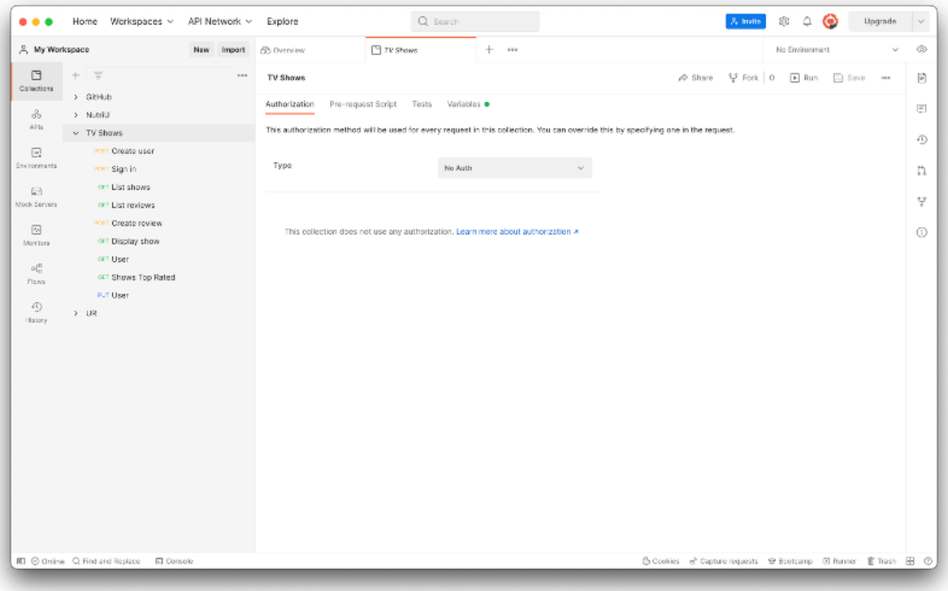
Debugging

Debugging the API

- **Postman**
  - https://www.postman.com/downloads/
  - select Mac Intel Chip
- **Paw**

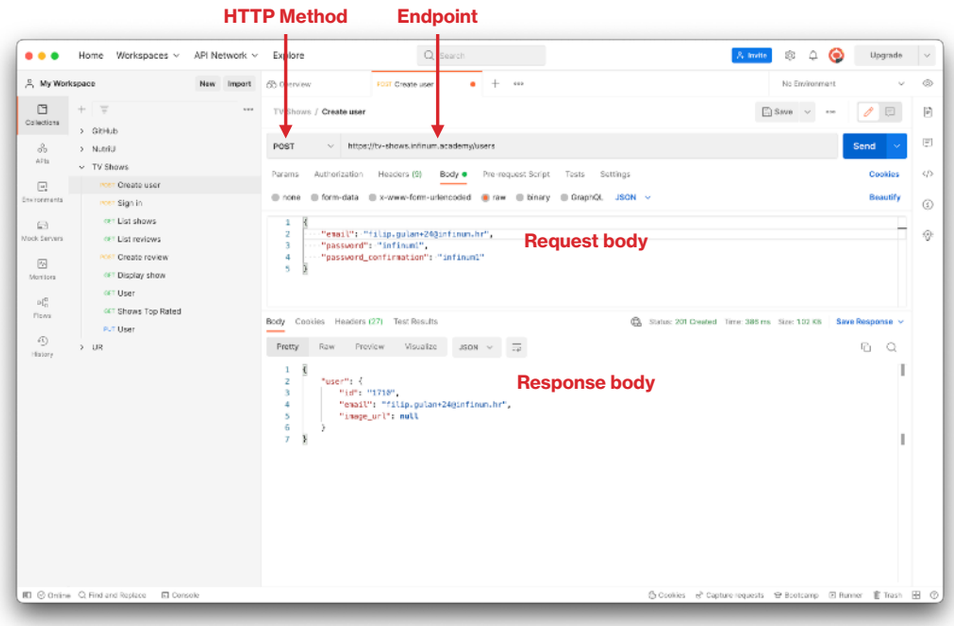
neverstop

- Postman - free, standalone version or Chrome extension
- Paw - paid, much better, native GUI



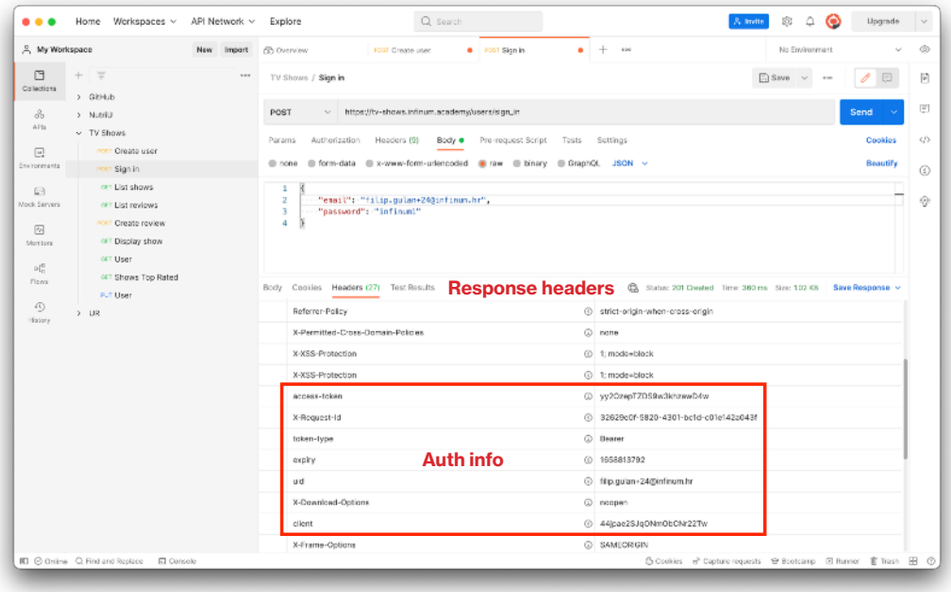
neverstop

- Postman demo time



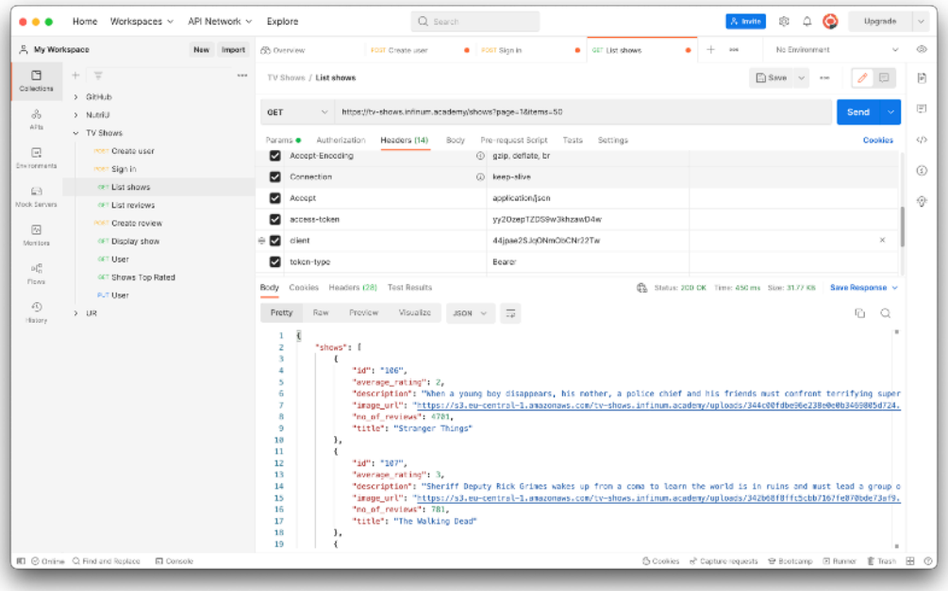
neverstop

- Create User



neverstop

- Sign In



neverstop

- Get all Shows

03

JSON

JSON

- **J**ava**S**cript **O**bject **N**otation
- request, response
- commonly used for APIs worldwide
  - easy to read
  - easy to parse
  - supported by a bunch of libraries

```
{
  "data": {
    "username": "myName",
     "email": "email@test.com",
     "password": "myPassword"
  }
}
```

Reading JSON

- `"key": "value"`
  - `"firstName": "John"`
- **`{}`**
  - marks a dictionary, hash-map or how you want to call it
- **`[]`**
  - marks an array

JSON in Swift

- JSON in Swift is represented as a dictionary of dictionaries
- in other languages known as a hash-map
- dictionary and arrays are represented with the same braces
  - `[key: value]`
  - `[value, value, value,…n]`

neverstop

- Array and Dictionary in Swift usually homogeneous
  - contains single type
- Array and Dictionary in ObjC heterogeneous
  - can contain multiple types
- [String: Any]
  - because we can’t know the type in advanced, we need to tell types

Dictionary - single type

```
// A simple `[String: String]` dictionary
let dictionary = [
    "milk"      : "cow",
    "egg"       : "chicken",
    "cheese"    : "hooman"
]

// Iterate through keys and values
dictionary.forEach { (key: String, value: String) in
    print("KEY: \(key), VALUE: \(value)")
}
```

Dictionary - heterogeneous

```
// Create a bit more complex dictionary
let complexDictionary: [String: Any] = [
    "products"          : dictionary,
    "author"            : "Ivan",
    "random_numbers"    : [1, 4, 166]
]

// Iterate through keys and values
complexDictionary.forEach { (key: String, value: Any) in
    print("KEY: \(key), VALUE: \(value)")
}
```

04

Networking in iOS

URLSession

- the base class for iOS networking
- used in combination with a few different classes
- **URLSession tutorial**
  - https://www.raywenderlich.com/3244963-urlsession-tutorial-getting-started
- abstracted through **Alamofire**

- we won’t cover it here now, we will use higher level abstraction called Alamofire

05

Alamofire

Alamofire

- https://github.com/Alamofire/Alamofire
- **HTTP** networking library for Swift
- probably the most popular third-party lib for iOS
- very easy JSON request/response
- **pod "Alamofire"**

- we won’t cover it here now, we will use higher level abstraction called Alamofire

```swift
let parameters: [String: String] = [
    "email": userNameField.text!,
    "password": passwordField.text!,
    "confirm_password": passwordField.text!
]

MBProgressHUD.showAdded(to: view, animated: true)

AF.request(
        "https://tv-shows.infinum.academy/users",
        method: .post,
        parameters: parameters,
        encoder: JSONParameterEncoder.default
    )
    .validate() // Status code in 200 ..< 300 range
    .responseJSON { dataResponse in
        MBProgressHUD.hide(for: view, animated: true)
        switch dataResponse.result {
        case .success(let response):
            print("Success: \(response)")
        case .failure(let error):
            print("Failure: \(error)")
       }
    }
```

06

JSON deserialization Codable

APIs suck - deserialization

- pretty much mostly
- dictionaries suck too (string based, loosely typed)
- we need to convert the JSON from the API into something we can easily read and interact with
- something = **struct** , **enum** or **class**

- [String: Any] -> a bit of an issue ;)

Codable

- https://developer.apple.com/documentation/swift/codable
- a type that can convert itself into and out of an external representation (**JSON**, **plist**...)
- used when parsing JSON into our custom models, and vice versa

neverstop

- we can use this to convert JSON string based API to our custom models, and back again when sending something to the API

```json
{
    "user": {
        "id": "1713",
        "email": "filip.gulan+26@infinum.hr",
        "image_url": null
    }
}
```

```swift
struct LoginResponse: Codable {
    let user: LoginUser
}

struct LoginUser: Codable {
    let email: String
    let imageUrl: String?
    let id: String

    enum CodingKeys: String, CodingKey {
        case email
        case imageUrl = "image_url"
        case id
    }
}
```

```swift
let parameters: [String: String] = [
    "email": "filip.gulan@infinum.hr",
    "password": "infinum1"
]

MBProgressHUD.showAdded(to: view, animated: true)

AF.request(
    "https://tv-shows.infinum.academy/users/sign_in",
    method: .post,
    parameters: parameters,
    encoder: JSONParameterEncoder.default
)
.validate() // Status code in 200 ..< 300 range
.responseDecodable(of: LoginResponse.self) { [weak self] response in
    guard let self = self else { return }
    MBProgressHUD.hide(for: self.view, animated: true)
    switch response.result {
    case .success(let response):
        print("Success: \(response)")
    case .failure(let error):
        print("Failure: \(error)")
    }
}
```

07

User/Session API request

Things to consider

- **Register**
  - **POST**
  - https://tv-shows.infinum.academy/users
- **Login**
  - **POST**
  - https://tv-shows.infinum.academy/users/sign_in

08

Callback hell

Callback hell

- closure inside closure inside closure inside closure
- unreadable
- error propagation
- maintenance
- impossible to test
- very hard to debug

neverstop

- IMPORTANT:
- How to combat that?
    - well we have couple of levels of abstraction that we can use
        - Future/Promise
        - Rx
    - We want to achieve some sort of function composition
    - We want to minimise “code indentation” to the right
- Both *Futures* and *Rx* will allow us to write code that looks like synchronous code, but will actually be asynchronous
    - In let’s say C# we have async/await, a concurrency model to help, Swift still doesn’t have that, yet

Problem

```
// First API call with closure a.k.a callback
registerUserAPI(parameters) { [weak self] registerResponse in
    guard let self = self else { return }

    switch registerResponse.result {
    case .success(let registerData):

        // Second API call with closure a.k.a callback
        loginUserAPI(registerData) { [weak self] loginResponse in
            guard let self = self else { return }

            switch loginResponse.result {
            case .success(let user):
                self.navigateToHomeScreen(user)
            case .failure(let error):
                self.handleError(error)
            }
        }

    case .failure(let error):
        self.handleError(error)
    }
}
```

neverstop

PROBLEM

Solution

```
func registerUserAPI() -> Observable<RegisterData> {
    ...
}

func loginUserAPI() -> Observable<User> {
    ...
}

registerButtonEvent
    .withLatestFrom(emailAndPasswordEvent)
    .flatMap(registerUserAPI())
    .flatMap(loginUserAPI())
    .subscribe(
        onNext: { [weak self] user in
            self?.navigateToHomeScreen()
        },
        onError: { [weak self] error in
            self?.handleError(error)
        }
    )
    .disposed(by: diposeBag)
```

neverstop ∞

- **POSSIBLE SOLUTION**
- So this is written using Rx, we will not explain it now, but trust me when I say that you want to write code like that ;)

09

Appendix

Links

- **TV Shows API documentation**
  - https://tv-shows.infinum.academy/api/v1/docs/index.html
- **Alamofire**
  - https://github.com/Alamofire/Alamofire
- **Codable**
  - https://developer.apple.com/documentation/swift/codable
- **Multithreading and DispatchQueue**
  - https://fluffy.es/help-my-app-freezes/

Thank you!

neverstop

Linkedin Instagram Twitter Facebook

∞ INFINUM