



PROF. RENÊ XAVIER

DESENVOLVIMENTO PARA IOS 11 COM SWIFT 4

ONDE ENCONTRAR O MATERIAL?

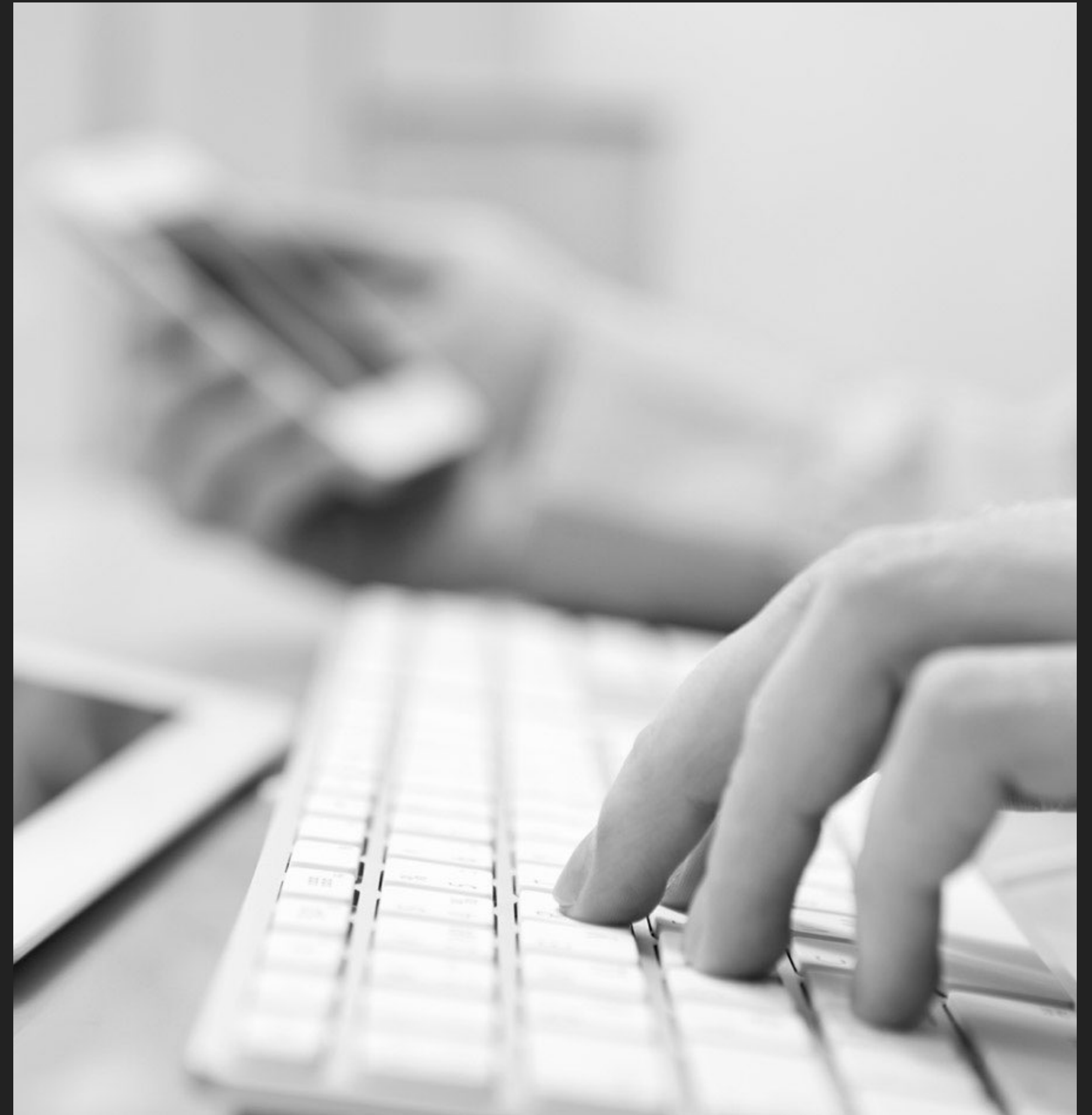
[HTTPS://GITHUB.COM/RENEFX/IOS-2018-01](https://github.com/RENEFX/IOS-2018-01)

GITHUB

ALÉM DO TRADICIONAL BLACKBOARD DO IESB

AGENDA

- ▶ Preenchendo objetos da nossa Model
- ▶ Organizando a requisição no MVC
- ▶ Como ajustar as requisições ao ciclo de vida da ViewController





**ENVIANDO
PARÂMETROS**

URL PARAMETERS

- ▶ Quando temos parâmetros que são enviados na url, podemos colocar tudo na String que irá gerar a URL:

```
guard let url = URL(string: "https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/price/BTCLTC/latest/?market_code=nex") else {  
    return  
}
```


HEADERS & BODY

- ▶ No exemplo da aula passada, conseguimos adicionar um parametro ao Header da nossa requisição da seguinte forma:

```
var request = URLRequest(url: url)
request.httpMethod = "GET"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
```

```
let headers = [
    "content-type": "application/json",
    "cache-control": "no-cache"
]
```

```
var request = URLRequest(url: url)
request.httpMethod = "GET"
request.allHTTPHeaderFields = headers
```

- ▶ Ou ainda criando um dicionário e preenchendo o allHTTPHeaderFields da request

HEADERS & BODY

- ▶ No exemplo da aula passada, conseguimos adicionar um parâmetro ao **Header** da nossa requisição da seguinte forma:

```
var request = URLRequest(url: url)
request.httpMethod = "GET"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
```

HEADERS & BODY

- ▶ Para passar parâmetros no **Body** preenchemos o `httpBody` da nossa request:

```
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.allHTTPHeaderFields = headers
request.httpBody = jsonData
```

- ▶ Porém, temos que preencher com um tipo **Data**

HEADERS & BODY

- ▶ Podemos criar o json como um dicionário

```
let bodyDicionario : [String : Any] = [  
    "idade": 20,  
    "nome": "Pedro",  
    "filmePreferido": [  
        "nome": "2001 – Uma odisséia no espaço",  
        "diretor": "Stanley Kubrick"  
    ]  
]
```

HEADERS & BODY

- ▶ Podemos criar o json como um dicionário
- ▶ Para transformar o dicionário em **Data** usamos o JSONSerialization

```
let bodyDicionario : [String : Any] = [  
    "idade": 20,  
    "nome": "Pedro",  
    "filmePreferido": [  
        "nome": "2001 – Uma odisséia no espaco",  
        "diretor": "Stanley Kubrick"  
    ]  
]  
  
let jsonData = try! JSONSerialization.data(withJSONObject: bodyDicionario, options: .prettyPrinted)
```

HEADERS & BODY

- ▶ Esse Data criado pelo JSONSerialization que passamos para o httpBody

```
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.allHTTPHeaderFields = headers
request.httpBody = jsonData
```



**ENVIANDO
PARÂMETROS – EXTRA**

HEADERS & BODY

- ▶ Se quiserem testar um POST ou um GET que sejam necessários passar headers e body para teste:

<https://nexchange2.docs.apiary.io/#reference/0/create-user-order/create-user-order?console=1>

Create User Order

Create User Order

Create order.



Switch to Console

Request



Switch to Example

Create User Order / Create User Order

Console calls are routed via Apiary

[Use browser](#) ?

POST https://private-anon-fa4537ada1-nexchange2.apiary-mock.com/en/api/v1/users/me/orders/

URI Parameters

Headers

Body

Reset Values

+ Add a new query parameter

Show Code Example

Mock Server

Call Resource

Mock Server

Debugging Proxy

Production

> Request

POST https://private-anon-fa4537ada1-nexchange2.apiary-mock.com/en/api/v1/users/me/orders/

URI Parameters

Headers

Body

Reset Values

✓ Content-Type application/json

✓ Authorization Bearer 3HrghbVeDUQWaOriqrXYLZmCb4c...

+ Add new header

URI Parameters

Headers

Body

Reset Values

```
{
  "amount_base": 3,
  "is_default_rule": false,
  "pair": {
    "name": "LTCBTC"
  },
  "withdraw_address": {
    "address": "LYUoUn9ATCcxvkbHseBJyVZMkLonx7agXA"
  }
}
```

Show Code Example

Mock Server

Call Resource

HEADERS & BODY

- ▶ Para o seguinte GET precisamos passar a Authorization:

<https://nexchange2.docs.apiary.io/#reference/0/get-order/get-order?console=1>

<https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/orders/V08PD/>

GET

https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/orders/V08PD/

URI Parameters

Headers

Body

Reset Values

☒

Authorization

Bearer 3HrghbVeDUQWaOriqrXYLZmCb4...

+

Add new header

Show Code Example

Mock Server

▼

Call Resource

HEADERS & BODY

- ▶ Para o seguinte GET precisamos passar a Authorization:

<https://nexchange2.docs.apiary.io/#reference/0/get-order/get-order?console=1>

<https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/orders/V08PD/>

Conseguimos da seguinte forma:

Authorization : Bearer 3HrghbVeDUQWaOriqrXYLZmCb4cEXB

```
guard let url = URL(string: "https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/orders/V08PD/") else {  
    return  
}  
  
var request = URLRequest(url: url)  
request.httpMethod = "GET"  
request.setValue("Bearer 3HrghbVeDUQWaOriqrXYLZmCb4cEXB", forHTTPHeaderField: "Authorization")
```

HEADERS & BODY

- ▶ Para o seguinte POST precisamos passar no Header o content-type e o Authorization:

<https://nexchange2.docs.apiary.io/#reference/0/get-order/get-order?console=1>

[https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/
users/me/orders/](https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/users/me/orders/)

content-type : application/json

Authorization : Bearer 3HrghbVeDUQWaOriqrXYLZmCb4cEXB

URI Parameters	Headers	Body	Reset Values
	<input checked="" type="checkbox"/> Content-Type	application/json	
	<input checked="" type="checkbox"/> Authorization	Bearer 3HrghbVeDUQWaOriqrXYLZmCb4...	

HEADERS & BODY

- ▶ No body, podemos passar os seguintes parâmetros que são de teste da API:

URI Parameters Headers Body [Reset Values](#)

```
{
  "amount_base": 3,
  "is_default_rule": false,
  "pair": {
    "name": "LTCBTC"
  },
  "withdraw_address": {
    "address": "LYUoUn9ATCxvkbtHseBJyVZMkLonx7agXA"
  }
}
```

[Show Code Example](#) [Mock Server](#) [Call Resource](#)

HEADERS & BODY

► Assim ficaria a chamada da API:

```
guard let url = URL(string: "https://private-anon-54726724dc-nexchange2.apiary-mock.com/en/api/v1/users/me/orders/") else {
    return
}

let headers = [
    "content-type": "application/json",
    "Authorization": "Bearer 3HrghbVeDUQWaOriqrXYLZmCb4cEXB"
]

let bodyDicionario : [String : Any] = [
    "amount_base": 3,
    "is_default_rule": false,
    "pair": [
        "name": "LTCBTC"
    ],
    "withdraw_address": [
        "address": "LYUoUn9ATCxvkbtHseBJyVZMkLonx7agXA"
    ]
]

let jsonData = try! JSONSerialization.data(withJSONObject: bodyDicionario, options: .prettyPrinted)

var request = URLRequest(url: url)
request.httpMethod = "POST"
request.allHTTPHeaderFields = headers
request.httpBody = jsonData

let task = URLSession.shared.dataTask(with: request) { data, response, error in
```

O QUE É ESSE
JSONSERIALIZATION?

**COMO
TRANSFORMAR O
JSON NA MINHA
MODEL?**



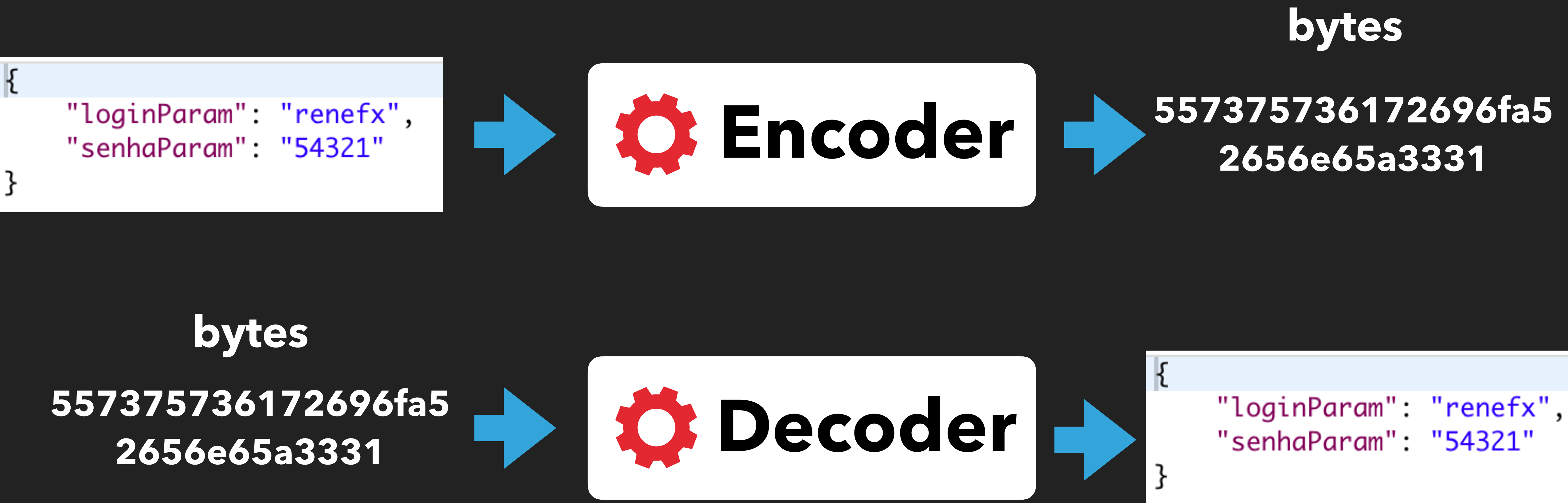
PREENCHENDO A MODEL

PREENCHENDO A MODEL

- ▶ Nós aprendemos a fazer alguns tipos de requisições
- ▶ Em todos os casos o retorno vinha por meio de um dicionário
- ▶ Não é prático
- ▶ Vamos fazer o **Encoding** e o **Decoding** do JSON para uma classe
 - ▶ Vimos um pouco com o UserDefaults
 - ▶ Não podemos enviar um objeto diretamente pela rede, temos que enviar os seus bytes

PREENCHENDO A MODEL

- ▶ O JSON não é diferente, não podemos enviar o objeto pela rede, mas sim seus bytes



PREENCHENDO A MODEL

- ▶ Swift nos provê uma forma de fazer isso sem muito esforço
- ▶ Implementando o protocolo **Codable**
- ▶ O protocolo Codable executa tanto o Encoding quanto o Decoding
- ▶ O que é preciso fazer?
 - ▶ Preparar nossa Model
 - ▶ Transformar em bytes - Encode
 - ▶ Ou transformar os bytes para a nossa Model - Decode

PREPARANDO A MODEL

PREENCHENDO A MODEL

- ▶ Para a maioria dos tipos que você usar: String, Int, Double, Date, Data e URL

Na model você não tem que fazer "nada"

```
class Login: Codable {  
    var loginParam: String?  
    var senhaParam: String?  
}
```

```
class LojaX: Codable {  
    var nome: String?  
    var iconePequeno: String?  
    var iconeGrande: String?  
    var favorita: Bool?  
    var vendeComputador: Bool?  
    var vendeJogos: Bool?  
    var produtos: [ProdutoX]?  
  
}
```

```
class ProdutoX: Codable {  
    var titulo: String?  
    var imagem: String?  
    var descricao: String?  
  
}
```

Se sua model possuir uma lista de outra classe que implementa o Codable, também não é preciso mudá-la

ENCODE
TRANSFORMANDO
EM BYTES

PREENCHENDO A MODEL

- ▶ O Encoder é usado quando queremos transformar o Objeto para bytes de JSON
- ▶ Esses bytes de JSON são trabalhados pelo tipo Data
- ▶ Já vimos um Encoding em ação

```
let jsonData = try! JSONSerialization.data(withJSONObject: bodyDicionario, options: .prettyPrinted)

var request = URLRequest(url: url)
request.httpMethod = "POST"
request.allHTTPHeaderFields = headers
request.httpBody = jsonData
```

PREENCHENDO A MODEL

- ▶ Usando o JSONEncoder mais novo, fica como a forma abaixo
- ▶ Isso só foi possível, pois nossa classe implementa o protocolo Codable

```
do {  
    let user = Login()  
    user.login = "renefx"  
    user.senha = "321"  
    let jsonEncoder = JSONEncoder()  
    request.httpBody = try jsonEncoder.encode(user)  
} catch {}
```

DECODER

PREENCHENDO A MODEL

- ▶ Aquela configuração vale para ambos (Encode e Decode). Agora temos de preenchê-la com os valores do JSON
- ▶ Quem é responsável por transformar os bytes em uma classe que segue o protocolo Codable é o JSONDecoder
- ▶ Temos o JSONEncoder

```
do {  
    let jsonDecoder = JSONDecoder()  
    let loginRetorno = try jsonDecoder.decode(Login.self, from: data)  
    print("\(loginRetorno.loginParam ?? "vazio")")  
} catch let error {  
    print(error.localizedDescription)  
}
```


PREENCHENDO A MODEL

- ▶ Como existe a possibilidade de falha da conversão, executamos dentro de um try catch
- ▶ Usamos o data que recebemos da nossa requisição
- ▶ Nele temos os bytes que irão virar nosso objeto

```
do {  
    let jsonDecoder = JSONDecoder()  
    let loginRetorno = try jsonDecoder.decode(Login.self, from: data)  
    print("\(loginRetorno.loginParam ?? "vazio")")  
} catch let error {  
    print(error.localizedDescription)  
}
```

PREENCHENDO A MODEL

- ▶ Dentro do jsonDecoder que informamos para qual objeto queremos transformar aqueles bytes.
- ▶ Nesse caso é a Login, mas se fosse um Array de Login?

```
do {  
    let jsonDecoder = JSONDecoder()  
    let loginRetorno = try jsonDecoder.decode(Login.self, from: data)  
    print("\(loginRetorno.loginParam ?? "vazio")")  
} catch let error {  
    print(error.localizedDescription)  
}
```

PREENCHENDO A MODEL

- ▶ Se fosse um Array de Login?
- ▶ Simples, indicamos como **[Login]**
- ▶ Então a variável loginRetorno é um array de objetos Login

```
do {  
    let jsonDecoder = JSONDecoder()  
    let loginRetorno = try jsonDecoder.decode([Login].self, from: data)  
} catch let error {  
    print(error.localizedDescription)  
}
```

```
guard let url = URL(string: "http://localhost:3333/urlParam?login=renefx&senha=54321") else {
    return
}

var request = URLRequest(url: url)
request.httpMethod = "GET"

let task = URLSession.shared.dataTask(with: request) { data, response, error in
    guard let data = data, error == nil else {
        print("error=\(error.debugDescription)")
        return
    }

    if let httpStatus = response as? HTTPURLResponse, httpStatus.statusCode != 200 {
        print("\(httpStatus.statusCode)")
        print("Mensagem = \(HTTPURLResponse.localizedString(forStatusCode: httpStatus.statusCode))")
    }

    do {
        let jsonDecoder = JSONDecoder()
        let loginRetorno = try jsonDecoder.decode(Login.self, from: data)
        print("\n")
    } catch let error {
        print(error.localizedDescription)
    }
}
```

CUSTOMIZAÇÕES

DESSES

PROCESSOS

CUSTOMIZAÇÃO DO ENCODER

PREENCHENDO A MODEL

- ▶ A forma de fazer o **Encode** é implementando a seguinte function na sua Model
- ▶ Usamos o container de **CodingKeys** como no **Decode**
- ▶ Depois encodificamos as nossas properties

```
public func encode(to encoder: Encoder) throws {  
    var container = encoder.container(keyedBy: CodingKeys.self)  
    try container.encode(login, forKey: .login)  
    try container.encode(senha, forKey: .senha)  
}
```

PREENCHENDO A MODEL

- ▶ Isso é feito para caso seja necessário sempre adicionar algo ou codificar uma variável de uma forma específica antes de enviar ela no Body da requisição.
- ▶ Não faz sentido ficar colocando e removendo uma camada de segurança a cada vez que você usa uma variável no seu App. Só colocamos quando ela vai ser enviada.
- ▶ Para isso alteramos a variável da nossa classe que será Encodificada

```
public func encode(to encoder: Encoder) throws {  
    var container = encoder.container(keyedBy: CodingKeys.self)  
    try container.encode(login, forKey: .login)  
    try container.encode(senha, forKey: .senha)  
}
```

PREENCHENDO A MODEL

- ▶ Nesse caso, estamos encodificando nossa senha em Base64 antes de enviá-la
- ▶ O base64 é só uma forma de encoding, não é uma camada de segurança da sua informação - usamos só para exemplificar como alterar o encoding
- ▶ Para colocar uma camada de segurança, é interessante dar uma olhada na biblioteca CommonCrypto com o algoritmo AES256

```
public func encode(to encoder: Encoder) throws {
    var container = encoder.container(keyedBy: CodingKeys.self)
    try container.encode(login, forKey: .login)

    if let senha = senha {
        let senhaBase64 = Data(senha.utf8).base64EncodedString()
        try container.encode(senhaBase64, forKey: .senha)
    }
}
```

CUSTOMIZAÇÃO DO DECODER

PREENCHENDO A MODEL

- ▶ Certas ocasiões, só implementar as CodingKeys do protocolo **Codable** não decodificam o JSON:
 - ▶ Quando não queremos que nossa model tenha o mesmo nome que a key do JSON - Ex.: JSON "usuario_id_unica" enquanto queremos na Model "id"
 - ▶ Quando queremos colocar alguma lógica ao receber um valor do JSON - Ex.: Concatenar um texto; Quando o JSON retornar a string "vazio" preencher o objeto com nil
 - ▶ Quando usamos tipos que não seguem o protocolo Codable - Ex.: RealmOptional
- ▶ Para esses casos conseguimos resolver com o protocolo **Codable** mesmo, mas só fazendo alguns ajustes.

CASO 1

PREENCHENDO A MODEL

Server



JSON

App



Objeto



```
{  
  "loginParam": "renefx",  
  "senhaParam": "54321"  
}
```

```
class Login: Codable {  
  var loginParam: String?  
  var senhaParam: String?  
}
```




JSONDecoder

JSON

Objeto

```
{  
  "loginParam": "renefx",  
  "senhaParam": "54321"  
}
```

```
class Login: Codable {  
  var loginParam: String?  
  var senhaParam: String?  
}
```




E se eu não quiser usar o nome que vem do JSON?

JSON

```
{  
  "loginParam": "renefx",  
  "senhaParam": "54321"  
}
```

Objeto

```
class Login: Codable {  
  var loginParam: String?  
  var senhaParam: String?  
}
```

Two red arrows originate from the JSON object. The first arrow points from the "loginParam" key to the "loginParam" property in the Kotlin class. The second arrow points from the "senhaParam" key to the "senhaParam" property in the Kotlin class.

E se eu não quiser usar o nome que vem do JSON?

JSON

```
{  
  "loginParam": "renefx",  
  "senhaParam": "54321"  
}
```

Objeto

```
class Login: Codable {  
    var login: String?  
    var senha: String?  
}
```

E se eu não quiser usar o nome que vem do JSON?

JSON

```
{  
    "loginParam": "renefx",  
    "senhaParam": "54321"  
}
```

Objeto

```
class Login: Codable {  
    var login: String?  
    var senha: String?  
}
```

PREENCHENDO A MODEL

- ▶ Tanto para Encode quanto para Decode, os parâmetros que ligam o JSON com o nosso objeto são definidos por Keys - EncodingKeys
- ▶ EncodingKeys são enums

```
class Login: Codable {  
    var login: String?  
    var senha: String?  
  
    enum CodingKeys: String, CodingKey {  
        case login = "loginParam"  
        case senha = "senhaParam"  
    }  
}
```


Agora o JSONDecoder vai olhar para as Keys

JSON

```
{  
  "loginParam": "renefx",  
  "senhaParam": "54321"  
}
```

Objeto

```
class Login: Codable {  
  var login: String?  
  var senha: String?  
  
  enum CodingKeys: String, CodingKey {  
    case login = "loginParam"  
    case senha = "senhaParam"  
  }  
}
```



CASO 2

Mas e se eu quisesse fazer algum cálculo ao preencher meu objeto?

JSON

```
{  
  "loginParam": "renefx",  
  "senhaParam": "54321"  
}
```

Objeto

```
class Login: Codable {  
    var login: String?  
    var senha: String?  
  
    enum CodingKeys: String, CodingKey {  
        case login = "loginParam"  
        case senha = "senhaParam"  
    }  
}
```

Digamos que eu queira colocar um ";" após o login que recebo do JSON

JSON

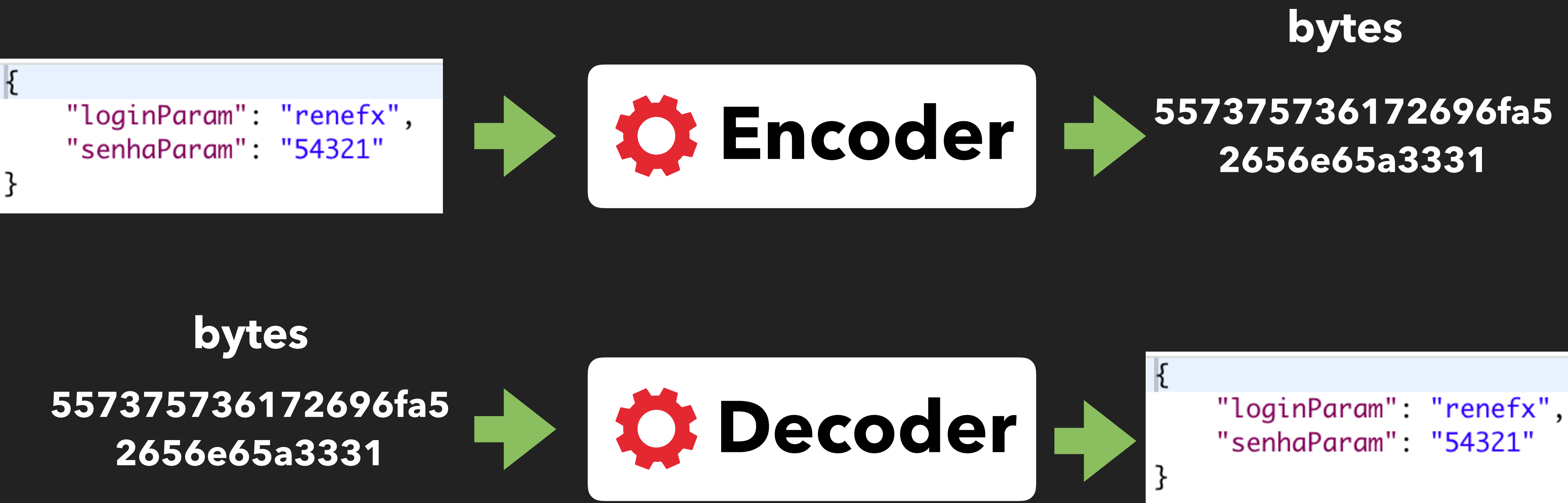
```
{  
    "loginParam": "renefx",  
    "senhaParam": "54321"  
}
```

Objeto

```
class Login: Codable {  
    var login: String?  
    var senha: String?  
  
    enum CodingKeys: String, CodingKey {  
        case login = "loginParam"  
        case senha = "senhaParam"  
    }  
}
```

PREENCHENDO A MODEL

- ▶ O processo de Encoding e Decoding tem sido feito de forma abstrata para nós
- ▶ Antes de fazer a alteração, temos que analisar qual dos dois processos que será alterado:

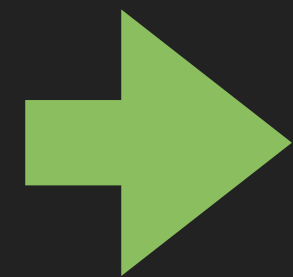


PREENCHENDO A MODEL

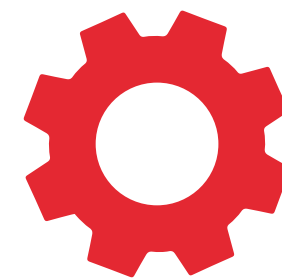
- ▶ Até agora temos visto somente o processo de Decoder
- ▶ Consiste em pegar os bytes do JSON e transformar em um objeto
- ▶ O processo contrário transformar o objeto em bytes que correspondem a um JSON, veremos já já
- ▶ O que será feito é:

**bytes do JSON
recebido**

**557375736172696fa5
2656e65a3331**



Adiciona o ;



Decoder



**objeto no
Xcode**

```
{  
  "login": "renefx;",  
  "senha": "54321"  
}
```

PREENCHENDO A MODEL

- ▶ Na nossa Model precisamos criar um construtor da classe - um init
- ▶ Esse init deve ser informado que será feito pelo decoder

```
init(from decoder: Decoder)
```


PREENCHENDO A MODEL

- ▶ A declaração segue dessa forma abaixo
- ▶ Criamos um init com o modificador convenience para não perdermos o init puro - init()
- ▶ Por ser um init diferente do padrão chamamos o init() e depois que iremos mudar o objeto que estamos construindo

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
}
```

PREENCHENDO A MODEL

- Nós criamos as CodingKeys e veremos como elas atuam no processo de decoding e encoding

```
class Login: Codable {  
    var login: String?  
    var senha: String?  
  
    enum CodingKeys: String, CodingKey {  
        case login = "loginParam"  
        case senha = "senhaParam"  
    }  
  
    convenience required init(from decoder: Decoder) throws {  
        self.init()  
    }  
}
```

PREENCHENDO A MODEL

- Precisamos das CodingKeys para definir os nomes das properties que o JSON irá trazer

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
}
```

PREENCHENDO A MODEL

- ▶ Precisamos das CodingKeys para definir os nomes das properties que o JSON irá trazer
- ▶ A partir disso, passamos a preencher nosso objeto com os bytes decodificados

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login)  
}
```

PREENCHENDO A MODEL

- ▶ Precisamos das CodingKeys para definir os nomes das properties que o JSON irá trazer
- ▶ A partir disso, passamos a preencher nosso objeto com os bytes decodificados

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login)  
    self.senha = try container.decode(String.self, forKey: .senha)  
}
```


PREENCHENDO A MODEL

- ▶ Agora sim podemos adicionar o ;

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login)  
    self.senha = try container.decode(String.self, forKey: .senha)  
}
```

PREENCHENDO A MODEL

- ▶ Agora sim podemos adicionar o ;

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login) + ";"  
    self.senha = try container.decode(String.self, forKey: .senha)  
}
```

Nesse init from **decoder** faremos as customizações que queremos ao transformar do JSON para a nossa model

ATENÇÃO

PREENCHENDO A MODEL

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login) + ";"  
    self.senha = try container.decode(String.self, forKey: .senha)  
}
```

!!!

Quando customizamos o decoder, temos
que reescrever o decoder de **TODAS** as
properties.

PREENCHENDO A MODEL

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login) + ";"  
    self.senha = try container.decode(String.self, forKey: .senha)  
}
```

!!!

Se deixarmos uma variável sem preencher achando que
ela irá seguir o fluxo normal,

**ELA NÃO SERÁ PREENCHIDA COM O VALOR DO
JSON.**

PREENCHENDO A MODEL

```
convenience required init(from decoder: Decoder) throws {  
    self.init()  
    let container = try decoder.container(keyedBy: CodingKeys.self)  
    self.login = try container.decode(String.self, forKey: .login) + ";"  
}
```

!!!

Nessa alteração acima (removi a atribuição da senha), a senha não será preenchida com o valor vindo do JSON, mas **SEMPRE** será preenchida com o valor default nil (pois é do tipo String?).

Já o login será preenchido com o valor vindo do JSON.

**COMO VALIDAR
A STRING
"VAZIO"?**

PREENCHENDO A MODEL

- ▶ Vamos fazer isso com o login
- ▶ Primeiro vamos remover a adição do ";"

```
self.login = try container.decode(String.self, forKey: .login) + ";"
```

```
self.login = try container.decode(String.self, forKey: .login)
```

PREENCHENDO A MODEL

- ▶ A parte sublinhada é onde recebemos o que veio do JSON
- ▶ Temos só que avaliar se ela corresponde a String "vazio" e atribuir nil ao self.login
- ▶ Lembrando que já tínhamos colocado o Login como Optional

```
self.login = try container.decode(String.self, forKey: .login)
```

```
var login: String?
```


PREENCHENDO A MODEL

- ▶ Temos só que avaliar se ela corresponde a String "vazio"
- ▶ Para isso vamos colocar ela em uma variável

```
let jsonLogin = try container.decode(String.self, forKey: .login)  
self.login =
```

PREENCHENDO A MODEL

- ▶ Agora avaliamos e atribuímos
- ▶ Lembrando que isso abaixo é equivalente ao uso do ternário

```
let jsonLogin = try container.decode(String.self, forKey: .login)
if jsonLogin == "vazio" {
    self.login = nil
} else {
    self.login = jsonLogin
}
```

```
let jsonLogin = try container.decode(String.self, forKey: .login)
self.login = jsonLogin == "vazio" ? nil : jsonLogin
```

CASO 3

PREENCHENDO A MODEL

- ▶ Caso tivéssemos um tipo RealmOptional em nossa Model, como tratar?

```
var login: String?  
var senha: String?  
let token = RealmOptional<Int>()
```

```
enum CodingKeys: String, CodingKey {  
    case login = "loginParam"  
    case senha = "senhaParam"  
    case token  
}
```

PREENCHENDO A MODEL

- ▶ Caso tivéssemos um tipo RealmOptional em nossa Model, como tratar?
- ▶ O RealmOptional não implementa o **Codable**
- ▶ Então ele não pode receber diretamente o objeto do JSON

```
var login: String?  
var senha: String?  
let token = RealmOptional<Int>()
```

```
self.token = try container.decode(Int.self, forKey: .senha)
```


PREENCHENDO A MODEL

- ▶ Lembrando que o RealmOptional não pode ter um valor atribuído diretamente.
- ▶ É necessário preencher a property **value** dele
- ▶ Essa property **value** é do tipo Int, pois criamos um RealmOptional<Int>

```
self.token = 1  'let' property 'token'
```

```
self.token.value = 1
```

PREENCHENDO A MODEL

- ▶ Lembrando que o RealmOptional não pode ter um valor atribuído diretamente.
- ▶ É necessário preencher a property **value** dele
- ▶ Essa property **value** é do tipo Int, pois criamos um RealmOptional<Int>

```
self.token = 1  'let' property 'token'
```

```
self.token.value = 1
```

PREENCHENDO A MODEL

- ▶ O trecho que faz o decode do JSON retorna um Int

```
self.token.value = try container.decode(Int.self, forKey: .senha)
```

PREENCHENDO A MODEL

- ▶ O trecho que faz o decode do JSON retorna um Int
- ▶ O **token.value** recebe um Int
- ▶ Agora, essa atribuição dá certo

```
self.token.value = try container.decode(Int.self, forKey: .senha)
```



ORGANIZANDO NO MVC

ORGANIZANDO NO MVC

```
class ViewController: UIViewController {
    var controller = LoginController()

    override func viewDidLoad() {
        super.viewDidLoad()

        guard let url = URL(string: "http://localhost:3333/urlParam?login=renefx&senha=54321") else {
            return
        }

        var request = URLRequest(url: url)
        request.httpMethod = "POST"

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            guard let data = data, error == nil else {
                print("error=\(error.debugDescription)")
                return
            }

            if let httpStatus = response as? HTTPURLResponse, httpStatus.statusCode != 200 {
                print("\(httpStatus.statusCode)")
                print("Mensagem = \(HTTPURLResponse.localizedString(forStatusCode: httpStatus.statusCode))")
            }

            do {
                let jsonDecoder = JSONDecoder()
                let loginRetorno = try jsonDecoder.decode(Login.self, from: data)
                print(loginRetorno.senha)
            } catch let error {
                print(error.localizedDescription)
            }
        }
        task.resume()
    }
}
```

ISSO TUDO É
SÓ A NOSSA
REQUISIÇÃO!!!

E nem colocamos um
loading nem validação
se está conectado com
a Internet

ORGANIZANDO NO MVC

```
class ViewController: UIViewController {
    var controller = LoginController()

    override func viewDidLoad() {
        super.viewDidLoad()

        guard let url = URL(string: "http://localhost:3333/urlParam?login=renefx&senha=54321") else {
            return
        }

        var request = URLRequest(url: url)
        request.httpMethod = "POST"

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            guard let data = data, error == nil else {
                print("error=\(error.debugDescription)")
                return
            }

            if let httpStatus = response as? HTTPURLResponse, httpStatus.statusCode != 200 {
                print("\(httpStatus.statusCode)")
                print("Mensagem = \(HTTPURLResponse.localizedString(forStatusCode: httpStatus.statusCode))")
            }

            do {
                let jsonDecoder = JSONDecoder()
                let loginRetorno = try jsonDecoder.decode(Login.self, from: data)
                print(loginRetorno.senha)
            } catch let error {
                print(error.localizedDescription)
            }
        }
        task.resume()
    }
}
```

Vamos tirar isso
daqui!

ORGANIZANDO NO MVC

- ▶ Vamos criar uma Controller que irá trabalhar com a Model
- ▶ Dentro dela vamos criar um método para executar o login
- ▶ Além de criar a Controller, vamos instanciar ela na ViewController

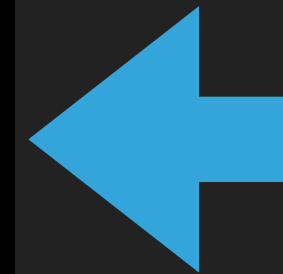
```
class LoginController {  
    func executarLogin() {  
  
    }  
}
```

```
class ViewController: UIViewController {  
    var controller = LoginController()  
}
```

ORGANIZANDO NO MVC

- Podemos pegar todo esse código e colocar na função executarLogin()

```
class LoginController {  
    func executarLogin() {  
  
    }  
}
```



```
class ViewController: UIViewController {  
    var controller = LoginController()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        guard let url = URL(string: "http://localhost:3333/urlParam?login=renefx&senha=54321") else {  
            return  
        }  
  
        var request = URLRequest(url: url)  
        request.httpMethod = "POST"  
  
        let task = URLSession.shared.dataTask(with: request) { data, response, error in  
            guard let data = data, error == nil else {  
                print("error=\(error.debugDescription)")  
                return  
            }  
  
            if let httpStatus = response as? HTTPURLResponse, httpStatus.statusCode != 200 {  
                print("\(httpStatus.statusCode)")  
                print("Mensagem = \(HTTPURLResponse.localizedString(forStatusCode: httpStatus.statusCode))")  
            }  
  
            do {  
                let jsonDecoder = JSONDecoder()  
                let loginRetorno = try jsonDecoder.decode(Login.self, from: data)  
                print(loginRetorno.senha)  
            } catch let error {  
                print(error.localizedDescription)  
            }  
        }  
        task.resume()  
    }  
}
```



```
class LoginController {
    func executarLogin() {
        guard let url = URL(string: "http://localhost:3333/urlParam?login=renefx&senha=54321") else {
            return
        }

        var request = URLRequest(url: url)
        request.httpMethod = "POST"

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            guard let data = data, error == nil else {
                print("error=\(error.debugDescription)")
                return
            }

            if let httpStatus = response as? HTTPURLResponse, httpStatus.statusCode != 200 {
                print("\(httpStatus.statusCode)")
                print("Mensagem = \(HTTPURLResponse.localizedString(forStatusCode: httpStatus.statusCode))")
            }

            do {
                let jsonDecoder = JSONDecoder()
                let loginRetorno = try jsonDecoder.decode(Login.self, from: data)
                print(loginRetorno.senha)
            } catch let error {
                print(error.localizedDescription)
            }
        }

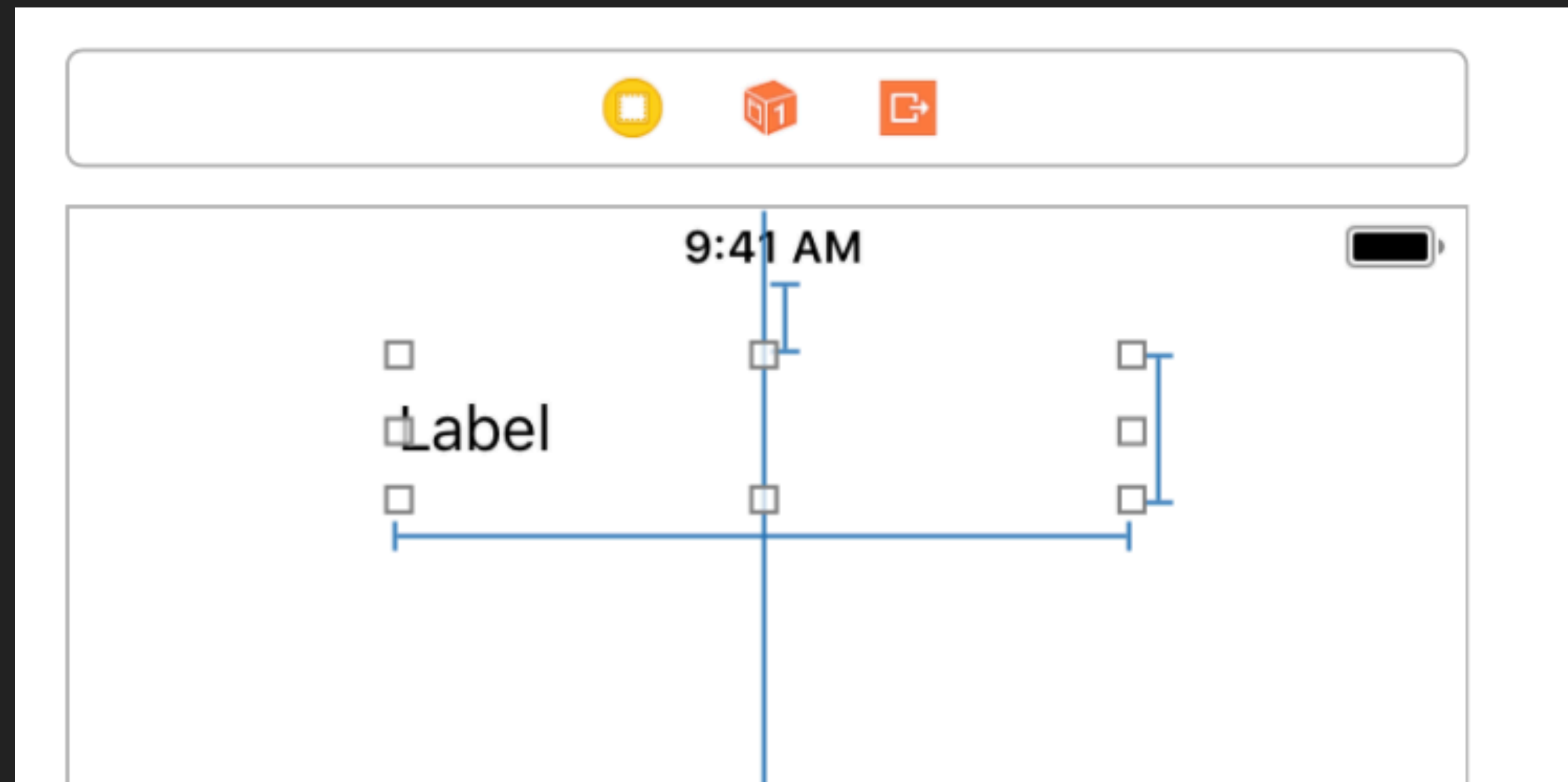
        task.resume()
    }
}
```


ORGANIZANDO NO MVC

Vamos chamar a
nossa controller

```
class ViewController: UIViewController {  
    var controller = LoginController()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        controller.executarLogin()  
    }  
}
```

ORGANIZANDO NO MVC

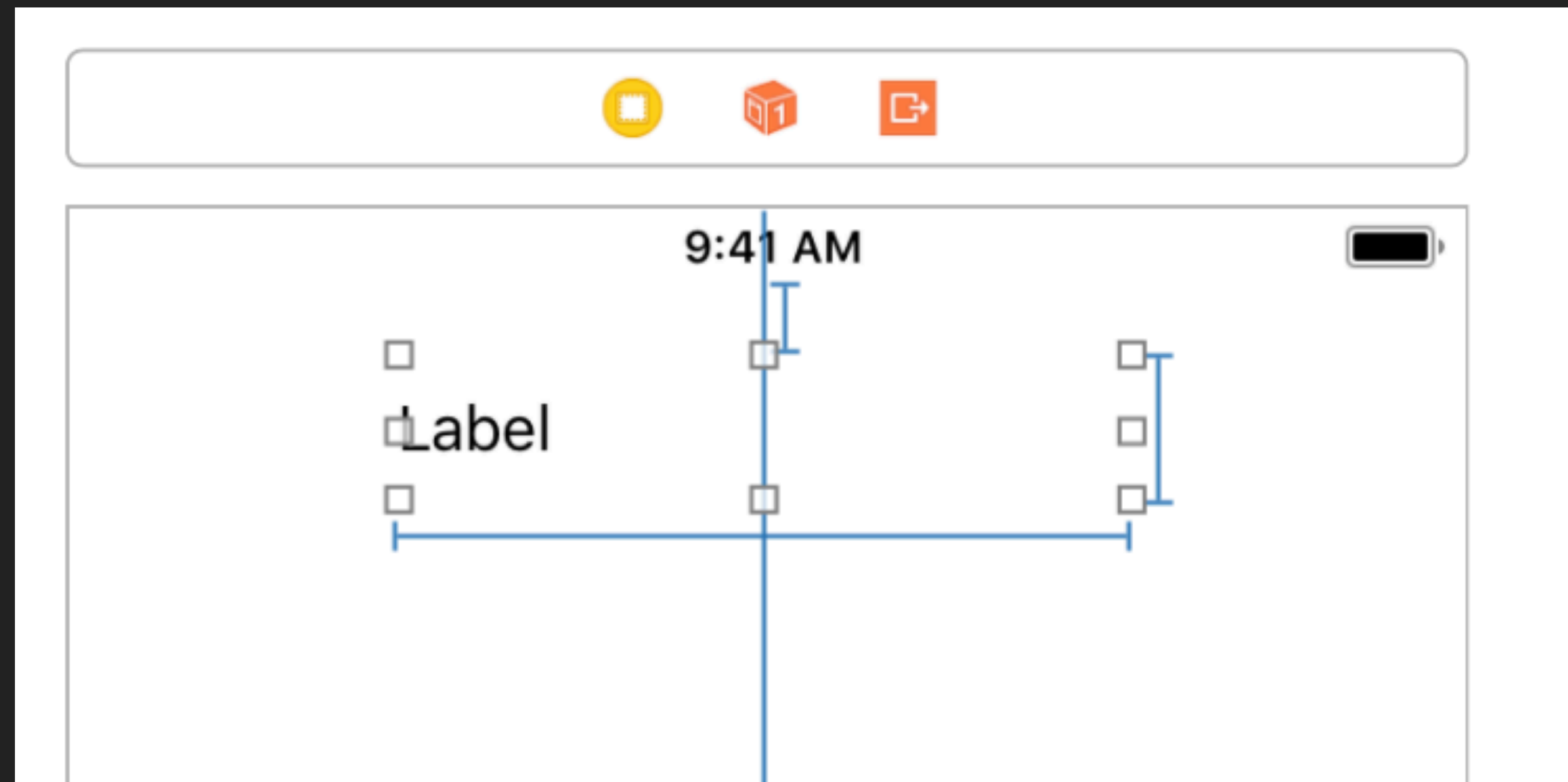


```
61
62 class ViewController: UIViewController {
63     var controller = LoginController()
64
65     @IBOutlet weak var loginLabel: UILabel!
66
67     override func viewDidLoad() {
68         super.viewDidLoad()
69         controller.executarLogin()
70     }
71 }
```

Mas e se quisermos preencher nossa View?

Como retornamos o valor da nossa Model
preenchida?

ORGANIZANDO NO MVC



```
61
62 class ViewController: UIViewController {
63     var controller = LoginController()
64
65     @IBOutlet weak var loginLabel: UILabel!
66
67     override func viewDidLoad() {
68         super.viewDidLoad()
69         controller.executarLogin()
70     }
71 }
```

Usamos uma Closure

pois não é sequencial, tem de aguardar o retorno do servidor

ORGANIZANDO NO MVC

```
func executarLogin(jsonResult: @escaping (String?) -> ()) {
```

- ▶ jsonResult é um parâmetro enviado para a função
- ▶ (String?) -> () indica que o parâmetro passado para executarLogin é uma função
 - ▶ Nessa função, vamos retornar a String? que é o username
 - ▶ Podemos chamar essa função que virá da ViewController assim:
jsonResult("stringParaRetorno")
- ▶ O escaping informa que sempre que chamarmos a jsonResult("stringParaRetorno") essa função será executada na classe que a enviou

ORGANIZANDO NO MVC

```
func executarLogin(senha: String, jsonResult: @escaping (String?) -> ()) {
```

- ▶ Ter uma closure não nos impede de receber outros parâmetros
- ▶ No exemplo acima recebemos a senha e a função que será executada ao obtermos o JSON

ORGANIZANDO NO MVC

- ▶ Ainda em nossa controller na função executarLogin:
- ▶ Ao receber o JSON e fazer o decode para nossa Model, chamamos o `jsonResult` passando o `username (login)` do nosso objeto
- ▶ Caso seja "erro" ou nulo, retornamos `nil`

```
do {  
    let jsonDecoder = JSONDecoder()  
    let loginRetorno = try jsonDecoder.decode(Login.self, from: data)  
  
    if let username = loginRetorno.login, username != "erro" {  
        jsonResult(loginRetorno.login)  
        return  
    }  
} catch let error {  
    print(error.localizedDescription)  
}  
jsonResult(nil)
```

ORGANIZANDO NO MVC

```
class LoginController {

    func executarLogin(jsonResult: @escaping (String?) -> ()) {

        guard let url = URL(string: "http://localhost:3333/urlParam?login=renefx&senha=54321") else {
            return
        }

        var request = URLRequest(url: url)
        request.httpMethod = "POST"

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            guard let data = data, error == nil else {
                print("error=\(error.debugDescription)")
                return
            }

            if let httpStatus = response as? HTTPURLResponse, httpStatus.statusCode != 200 {
                print("\(httpStatus.statusCode)")
                print("Mensagem = \(HTTPURLResponse.localizedString(forStatusCode: httpStatus.statusCode))")
            }

            do {
                let jsonDecoder = JSONDecoder()
                let loginRetorno = try jsonDecoder.decode(Login.self, from: data)

                if let username = loginRetorno.login, username != "erro" {
                    jsonResult(loginRetorno.login)
                    return
                }
            } catch let error {
                print(error.localizedDescription)
            }
            jsonResult(nil)
        }
        task.resume()
    }
}
```

Agora nossa
controller está
pronta

Mas como funciona
essa função que
enviamos para a
controller?

ORGANIZANDO NO MVC

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    controller.executarLogin(jsonResult: { username in  
    })  
}
```

ISSO JÁ ATÉ
VIMOS UM
POUCO

- ▶ Chamamos a executarLogin normalmente
- ▶ Declaramos que vamos enviar o parâmetro jsonResult
- ▶ O jsonResult é uma função, uma função em linha ou mais conhecida como closure
 - ▶ Sendo uma função, a sua limitação são os colchetes {}
 - ▶ Entre o primeiro colchete { e o in estão os parâmetros que a controller irá enviar
 - ▶ Entre o in e o último colchete } está o código que será executado quando a controller chamar a sua execução

ORGANIZANDO NO MVC

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    controller.executarLogin(jsonResult: { username in  
    })  
}
```

Vamos
preencher a
execução

- ▶ Temos que lembrar que estamos em uma Thread em background e assim como carregamos a imagem da raposa, vamos ter de chamar a main thread para alterar o visual

ORGANIZANDO NO MVC

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    controller.executarLogin(jsonResult: { username in  
        DispatchQueue.main.async {  
            self.loginLabel.text = username  
        }  
    })  
}
```

Pronto!

Separamos a View
da Controller

Carrier 

7:40 PM



renefx



REQUISICÕES NO CICLO DE VIDA

REQUISIÇÕES NO CICLO DE VIDA

- ▶ Ao fazer uma requisição temos de analisar onde chamamos
 - ▶ Ao chamar com a ação de um **botão**, sabemos que a requisição só será chamada com a ação do usuário
 - ▶ Ao chamar no **ViewDidLoad** temos que lembrar que essa chamada só será feita uma única vez
 - ▶ Se o nosso objetivo é que a request seja executada toda vez que a tela apareça, colocamos a chamada no **ViewWillAppear**
 - ▶ Só temos de lembrar que a Master-Detail e a TabBar quando temos uma ViewController para cada Tab, é como se todas as ViewControllers (das Tabs e a Master e Detail) estivessem ativas no mesmo momento - diferente de uma navigation (uma ViewController ativa por vez)