[Provider]

“Meta pattern”

Idea

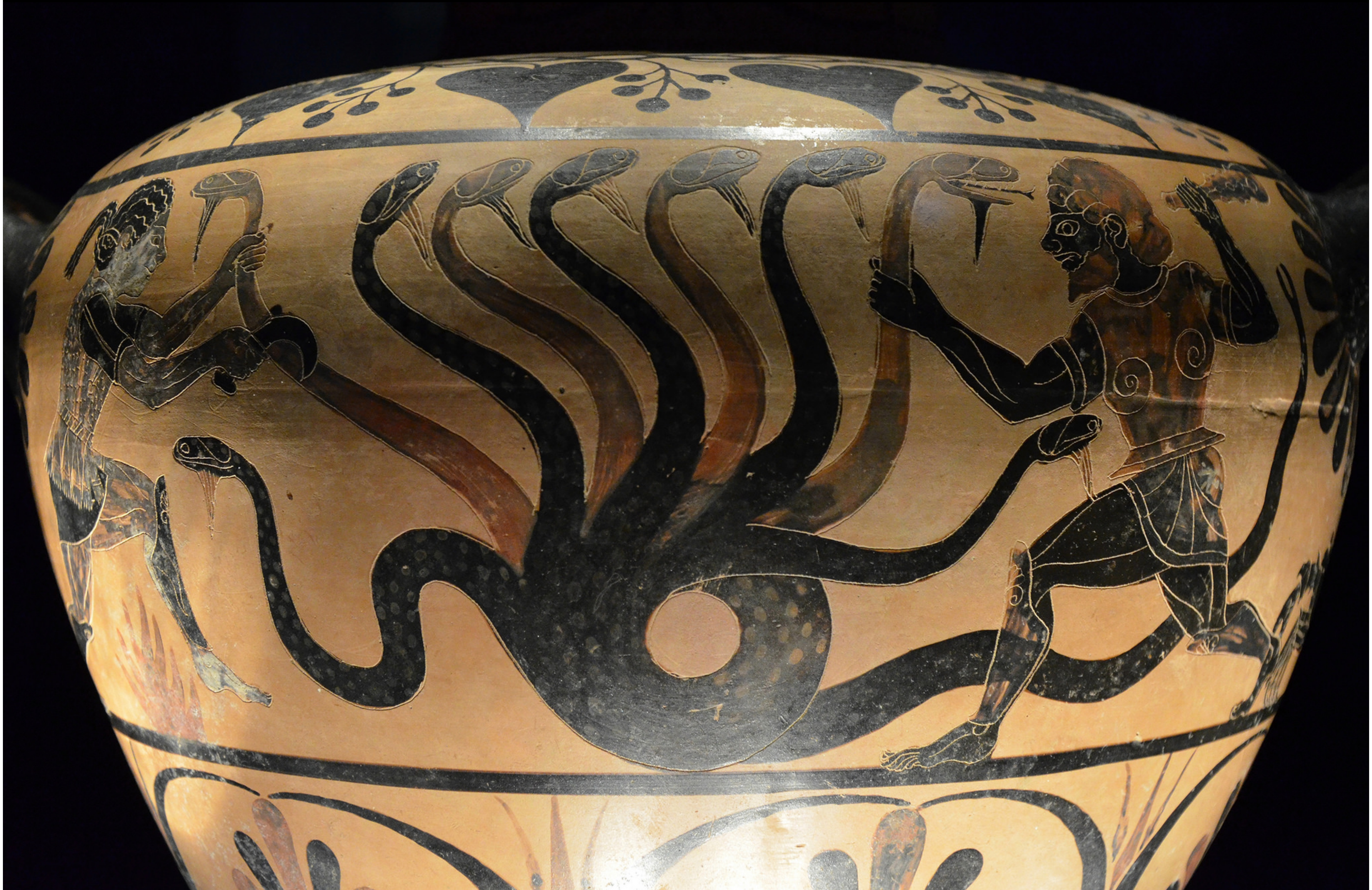
- Generate an object when it’s needed
- Provide access to the object
- Encapsulate logic for generating the object
- If needed, notify when the object is ready for use

Pseudo-interface

```
protocol Provider {

    associatedtype StoredObjectType
    associatedtype ReturnedObjectType
    
    var objectProvided: [ReturnedObjectType] { get }

    func addListener(lambda: (ReturnedObjectType) -> Void)
}
```

Example: “Translate” Data Types

Preliminary Concepts



s News Agency

“Translate” data types

```
// Animal:
// General type coming from a service.
// We don't control this service and its data 
types.

struct Animal {

    let name: String
    let numberOfLegs: UInt
    let numberOfHeads: UInt
}

// MyAnimal:
// This data type is in our domain.

public enum MyAnimal {

    case tiger
    case cat
    case bird
}

// MyMythologicalAnimal:
// This data type is in our domain.
public enum MyMythologicalAnimal {

    case kerberos
    case hydra
    case medusa
}
```

- We are dealing with an external service or a library in a different “business domain”
- e.g. SDKs, APIs, Bluetooth accessories…
- We want to decouple the release cycle of the two products: app and external data source
- It’s a good idea to translate data types

Animal Provider

```swift
struct AnimalProvider<T> {

    var myAnimals: [T] {
        return animals.flatMap { transformer($0) }
    }

    private let animals: [Animal]
    private let transformer: ((Animal) -> T?)

    init(animals: [Animal], transformer: @escaping ((Animal) -> T?)) {
        self.animals = animals
        self.transformer = transformer
    }
}
```

- Given a bunch of “generic” animals coming from the underlying service…
- …I can transform them on demand into a specific “something” expressed by the generic type T
- Note the flatMap: if a transformer fails to transform an Animal, it gets “filtered out”

Animals

```
// External data types

let tiger = Animal(name: "tiger", numberOfLegs: 4, numberOfHeads: 1)
let cat = Animal(name: "cat", numberOfLegs: 4, numberOfHeads: 1)
let bird = Animal(name: "bird", numberOfLegs: 2, numberOfHeads: 1)
let kerberos = Animal(name: "kerberos", numberOfLegs: 4, numberOfHeads: 3)
let hydra = Animal(name: "hydra", numberOfLegs: 4, numberOfHeads: 7)

let animals = [tiger, cat, bird, kerberos, hydra, bird]
```

- Simulate a bunch of Animals…
- …coming from an external service

Mythological (animal)

```
let mythologicalAnimalTransformer = { (animal: Animal) -> MyMythologicalAnimal? in

    if animal.name == "cerberus" {
        return MyMythologicalAnimal.kerberus
    } else if animal.name == "hydra" {
        return MyMythologicalAnimal.hydra
    } else {
        return nil
    }
}
```

```
let mythologicalAnimalProvider =

                        AnimalProvider(animals: animals,
                                       transformer: mythologicalAnimalTransformer)
```

Regular (animal)

```
let myNormalAnimalTransformer = { (animal: Animal) -> MyAnimal? in

    if animal.name == "tiger" {
        return MyAnimal.tiger
    } else if animal.name == "cat" {
        return MyAnimal.cat
    } else if animal.name == "bird" {
        return MyAnimal.bird
    } else {
        return nil
    }
}


let myNormalAnimalProvider = 

                            AnimalProvider(animals: animals, 
                                           transformer: myNormalAnimalTransformer)
```

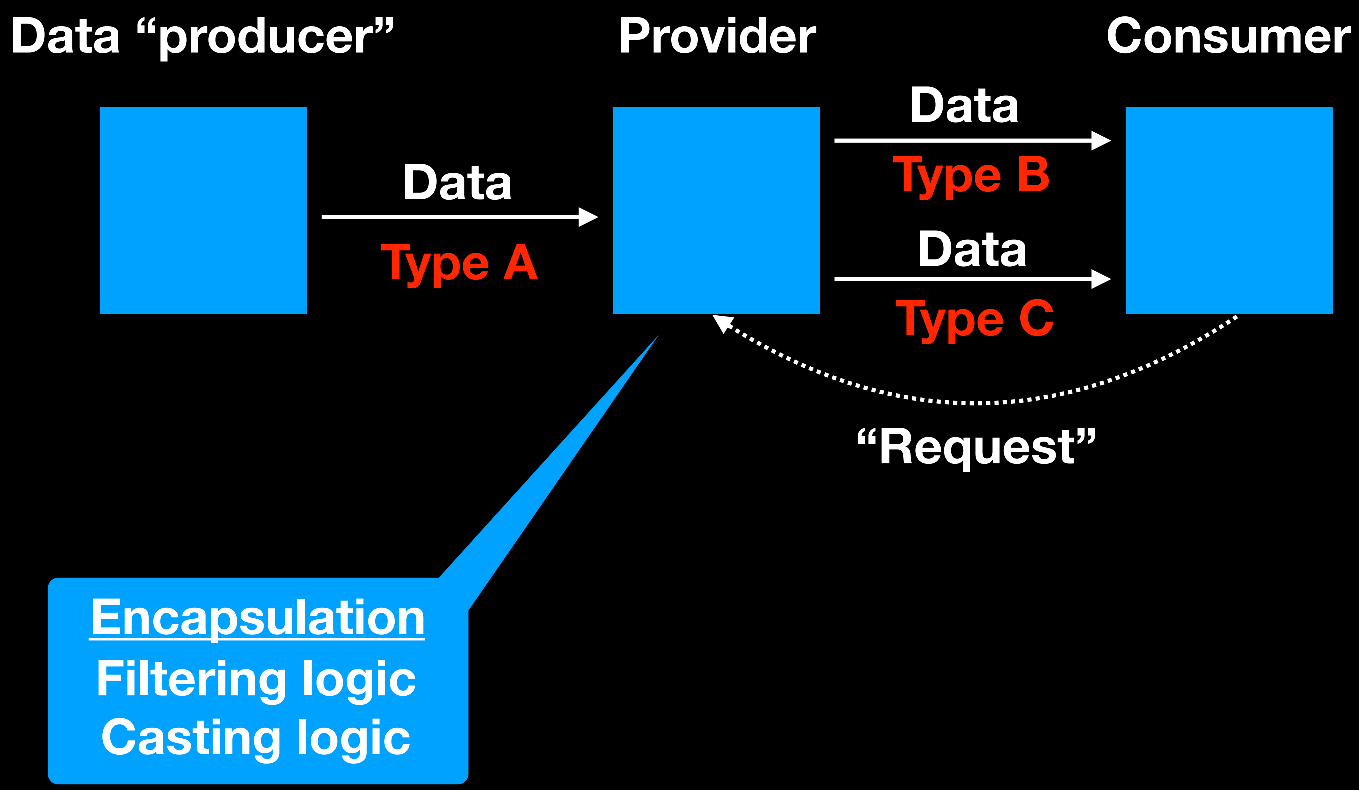
Usage

```swift
// Getting some mythological animal

let myKerberos = mythologicalAnimalProvider.myAnimals.filter { $0 == .kerberos }
let myMedusa = mythologicalAnimalProvider.myAnimals.filter { $0 == .medusa }

// Getting some normal animals

let myBirds = myNormalAnimalProvider.myAnimals.filter { $0 == .bird }
```

- Conclusion: we have generic models coming from an underlying service

- We use the provider to split those types into two “domain specific” types that we use throughout the app

Flow


Data “producer”
Provider
Consumer
Data
Type A
Data
Type B
Data
Type C
“Request”
Encapsulation
Filtering logic
Casting logic

Real World Usage: Sane Refactoring

- A low level SDK provides general information
- The SDK had its own objectives and release cycles
- App needed to insure that the SDK logic and data types did not creep into every single view
- Decoupling app domain from the SDK’s own domain was a good idea. Data types must be “translated”
- With a major refactoring in the SDK, a team can keep working without interruption while someone imports and tests the new SDK

Example: Provide Delayed Data

Image Provider (IP)

```swift
class ImageProvider {

    private(set) var imageData: Data? {
        didSet {
            guard let data = imageData else { return }
            listeners.forEach { $0(data) }
        }
    }
    private var listeners: [ ((Data) -> Void) ] = []
    private let url: URL

    init(url: URL) {
        self.url = url
    }

    func fetch() {
           // Logic for fetching the image
    }

    func addListener(_ listener: @escaping ((Data) -> Void)) {
           // Logic for adding a listener
   }
}
```

As soon as new data arrives, listeners are notified

IP: Data Fetch

```swift
class ImageProvider {

    // …

    func fetch() {

        print("===> Fetch called")
        let task = URLSession.shared.dataTask(with: url) {data, response, error in

            print("===> Response received")
            guard let data = data, error == nil else {
                print(error ?? "Unknown error")
                return
            }
            print("===> Just wasting some time...")
            sleep(5)
            self.imageData = data
        }
        task.resume()
    }

    // …
}
```

IP: Events

```
class ImageProvider {

    // …

    func addListener(_ listener: @escaping ((Data) -> Void)) {

        listeners.append(listener)
        guard let data = imageData else { return }
        listener(data)
    }
}
```

- While adding a listener, you also want to call it immediately if data is already there
- This makes life easier for the consumer

Usage

```
class ImagePresenter {

    private let provider: ImageProvider

    init(provider: ImageProvider) {

        self.provider = provider
        provider.addListener { (data) in
            let image = UIImage(data: data)
            print("===> New image.")
            let view = UIImageView(image: image)
        }
    }
}
```

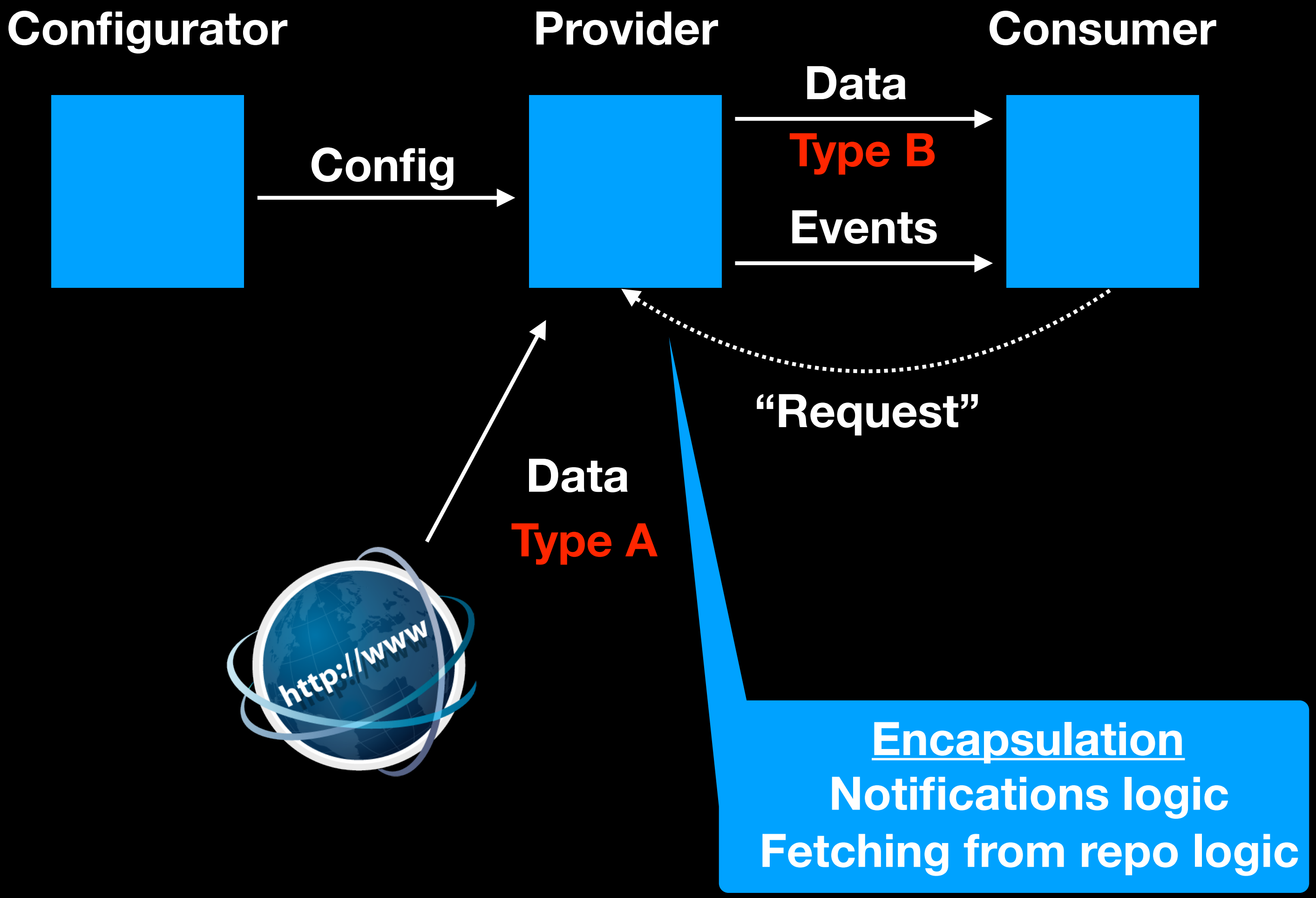
- This class does not do much, it instantiates (and immediately destroys) a UIImageView

- You can easily plug the code in playground

Usage

```swift
// Configure (externally) the provider
let url = URL(string: "http://www.esta-rohr.de/html/js/SuperBGImage/img/1066378_36549393.jpg")!
let imageProvider = ImageProvider(url: url)

// Inject the provider in the consumer
let presenter = ImagePresenter(provider: imageProvider)

// Start the provider
imageProvider.fetch()
```

Flow



Encapsulation

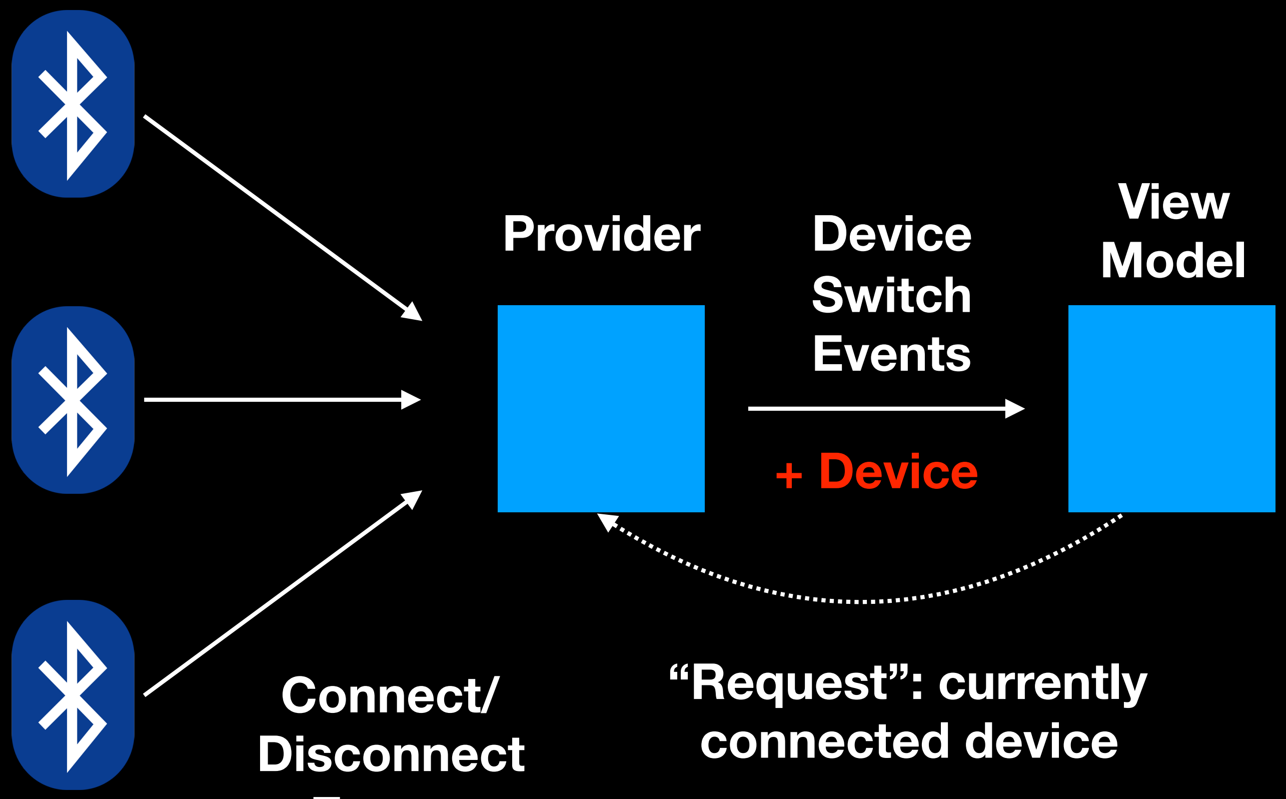
Notifications logic

Fetching from repo logic

Real World Usage

- I have used a version of this “provider”, with events, to connect and sort multiple Bluetooth accessories
- In the External Accessory Framework, you can have multiple Bluetooth devices physically linked to an iPhone (physically connected)
- I needed to operate only on one, chosen according to a certain logic: the “logically connected” device.
- The BluetoothDeviceProvider helped providing the one device according to the same login in every view
- I avoided using a singleton

Real World Usage: Flow



Provider

Device Switch Events

+ Device

View Model

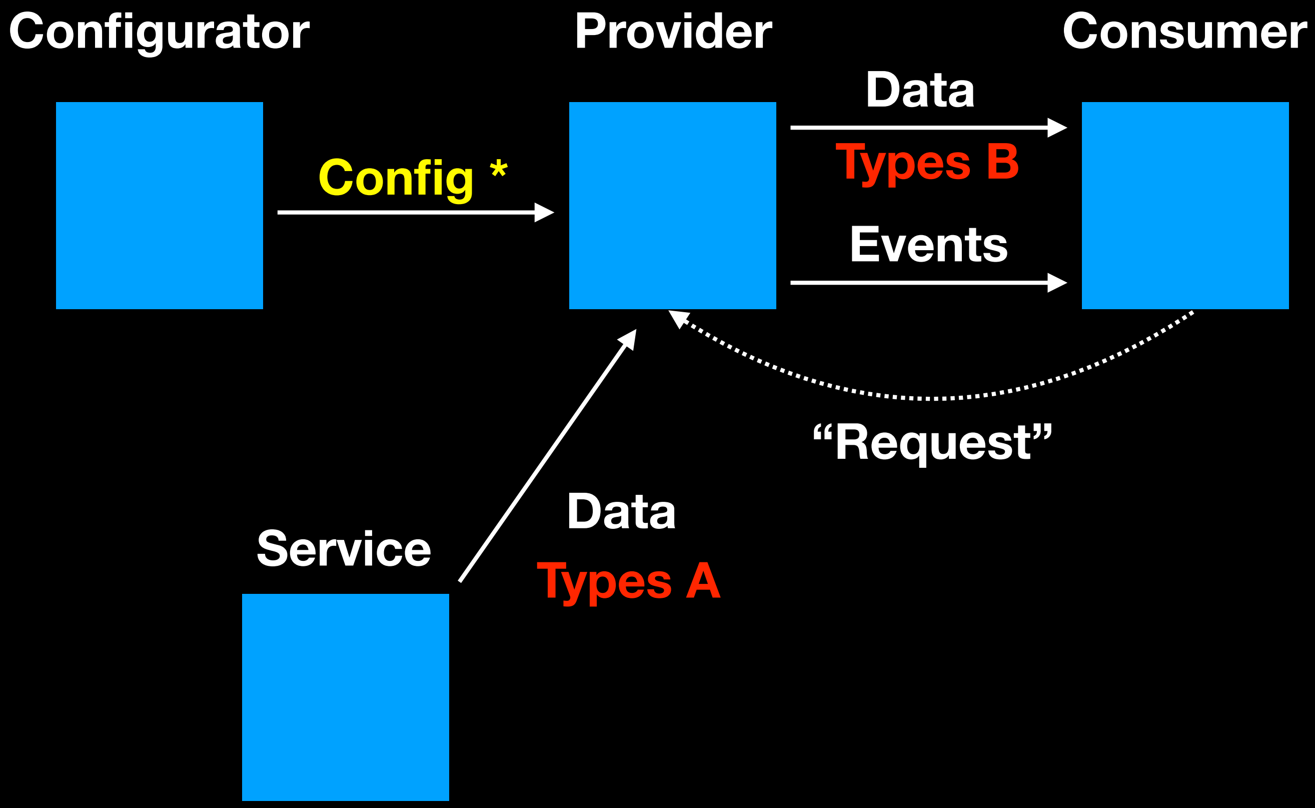
Connect/ Disconnect Events

+ Devices

“Request”: currently connected device

Conclusion

Flow


Configurator

Provider

Consumer

Config *

Data

Types B

Events

"Request"

Service

Data

Types A

(*) - Config examples: a bunch of data already "fetched", a URL, a "bluetooth connection manager"…

Summary

- Not really a pattern, more like a way of thinking
- Helps splitting responsibilities: [A] configuration, [B] data consumption, [C] data fetching and transformation
- Helps encapsulating data fetching logic and/or transformation logic, given a configuration of some kind
- Helps distributing the same data fetching/transformation logic across the app (no singleton)
- Can be used as a way to reactively provide new data to the consumer (home-made promise or stream)