Anıl taşkıran

170 Followers About

Using Property Wrapper on Production iOS E-commerce App How did we find solutions to Decoding Errors?

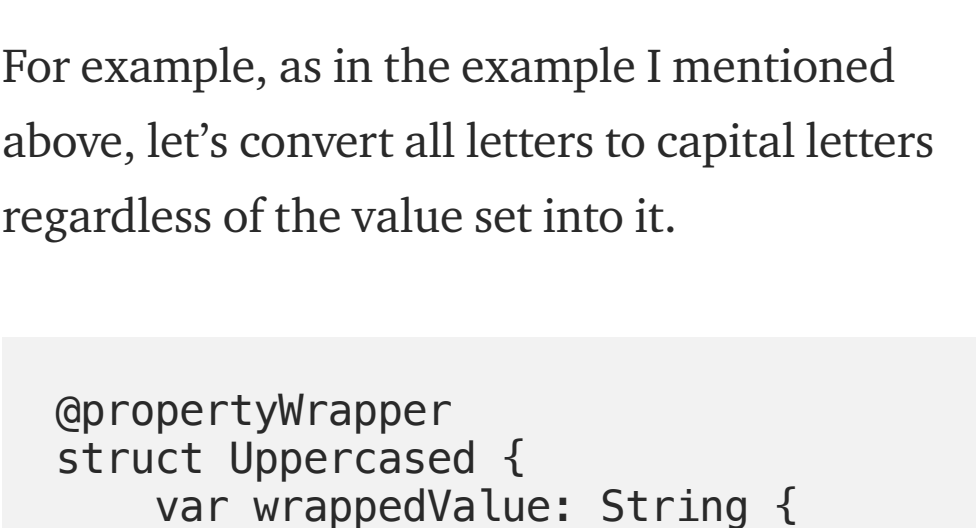
Anıl taşkıran Mar 20 · 6 min read ★

How did we get rid of a structure that we are repeatedly building by using the Property Wrapper in Production? In order not to get decoding errors, we started thinking about how we can assign a default value instead of defining an optional value, and the property wrapper did our job. You will find the answer to how we solved our needs at the end of this article.

Property Wrapper entered our lives as a backward compatible feature with swift 5.1. It is a structure that automatically enables new operations to be performed, which facilitates the addition of new features, filters or calculations, including the creation process of a feature and setter / getter stages. Let's picture it in our minds with a simple example.If you want the value that holds a string to be kept in uppercased characters when a new value is stored in it, you can easily do this with the property wrapper.

Doug Gregor and Joe Groff offered an alternative approach to this issue, which was first addressed in 2015–2016. It introduced it at Swift Evolution 0258 Proposal on September 16, 2020. Compared to the proposal presented in 2015, Doug and Joe's approach was simpler, more efficient to the compiler, and easier to understand for developers.

apple/swift-evolution
There are property implementation pattern...
github.com



· · ·

You can activate the Property wrapper by typing "@propertyWrapper" at the beginning of the enum, struct or class you want to create, and you can manage your logics on the wrappedValue property.

For example, as in the example I mentioned above, let's convert all letters to capital letters regardless of the value set into it.

```
@propertyWrapper
struct Uppercased {
    var wrappedValue: String {
        didSet { wrappedValue =
wrappedValue.uppercased() }
    }

init(wrappedValue: String) {
        self.wrappedValue =
wrappedValue.uppercased()
    }
}
```

In the above figure we created a struct named Uppercased. We showed that this is property Wrapper by typing @propertyWrapper. So he forced us to add wrappedValue. You can pass type according to the type of property you will use here. Now i will use string value to be capitalized.

```
struct Article {
    @Uppercased var title: String
}
```

Then you just need to write this wrapper at the beginning of which property you want to use. I want to capitalized the title of each article I created. That's why I added it to the title property.

```
let article = Article(title:
"property Wrapper")
print(article.title) // PROPERTY
WRAPPER
```

and now every time you call it, the article in title will be changed to a capital letter.

Okay, but what kind of problem did we encounter, and the property wrapper solved our problem? Our only concern was not to make a properties uppercased. Let's take a look at this issue now.

Let's say you will keep the userType in your DTO. And now there are 3 types. You can keep it as int or string equivalents, or you can even write enum if you know all the future equivalents. But when you use enum, and a new type is added, your project will start getting decoding errors, unless you do not assign a default value on your custom init method. In order to avoid decoding, we wrote our custom inits for values that could be enum equivalent throughout the project.

```
enum UserType: String, Codable {
    case admin, user, none

    public init(from decoder:
Decoder) throws {
        self = try
UserType(rawValue:
decoder.singleValueContainer().deco
de(RawValue.self)) ?? .none
    }
}
```

For example, looking at the example above, if a new UserType is added, the old client will not be able to decode it. But it will be set to the default value you will continue your life without getting decoding error.

So how about using a wrapper that can set the default value when decoding is received?

```
public protocol
DefaultCodableInterface {
    associatedtype RawValue:
Codable

    static var defaultValue:
RawValue { get }
}

@propertyWrapper
public struct DefaultCodable<T:
DefaultCodableInterface>: Codable {
    public var wrappedValue:
T.RawValue

    public init(wrappedValue:
T.RawValue) {
        self.wrappedValue =
wrappedValue
    }

    public init(from decoder:
Decoder) throws {
        let container = try
decoder.singleValueContainer()
        self.wrappedValue = (try?
container.decode(T.RawValue.self))
?? T.defaultValue
    }

    public func encode(to encoder:
Encoder) throws {
        try wrappedValue.encode(to:
encoder)
    }
}

extension DefaultCodable: Equatable
where T.RawValue: Equatable { }
extension DefaultCodable: Hashable
where T.RawValue: Hashable { }
```

First, let's create our Property Wrapper. Let us have an interface for this and it will easily do our operations in a generic structure. If you want, make a structure that will assign the last case of enum as the Default value, or create a structure that will set false / true when a bool value when null.
In the example above, the type of our wrappedValue comes from the interface. In this way, we give it to the decoder and if it does not return a value, we pass the default value we have.

When we look at the UserType example above, we said that we can actually make an development to give the last case by default. First of all, we started by creating a protocol called *EnumDefaultValueSelectable*. This protocol will determine the action to be taken. For example let's create a structure that will select the last value in the Enum case when we get decoding error. For this example; We have created a protocol conforming to *Codable & CaseIterable & RawRepresentable*. And we returned lastCase by browsing through allCases we have.

```
public protocol
EnumDefaultValueSelectable: Codable
& CaseIterable & RawRepresentable
where RawValue: Decodable,
AllCases: BidirectionalCollection {
}

public struct LastCase<T>:
DefaultCodableInterface where T:
EnumDefaultValueSelectable {
    public static var defaultValue:
T { T.allCases.last! }
}
```

Imagine clearing it all over the project and decorating it with a property wrapper. So clean :)

```
enum UserType: String,
EnumDefaultValueSelectable {
    case admin, user, none
}

struct UserResponse: Codable {
    @DefaultCodable<LastCase> var
user: UserType
}
```

You may be making your properties optional in order not to get decoding errors in the responses you create, but a bool value that comes in null must have a meaning for you.

In addition to the example above, we also have an isAdmin property. If the value of isAdmin comes to null, we can actually assign it false by default. So let's create a struct called DefaultFalse. And all we have to do is create a struct that conforms to our interface and return our default value.

```
public struct DefaultFalse:
DefaultCodableInterface {
    public static var defaultValue:
Bool { return false }
}
```

Then we can construct our Boolean property to be false when it is null.

```
struct UserResponse: Codable {
    @DefaultCodable<DefaultFalse>
var isAdmin: Bool
}
```

If you want to use this Property Wrapper, you can easily access it from the link below.

Trendyol/ios-components
iOS Components for iOS by Trendyol application...
github.com

By using the Property Wrapper in the Production code in this way, we got rid of the custom inits we made in many places. If there are structures you use, you can specify them in the comments section.

If you like it, you can share it to more people with your applause. 👏