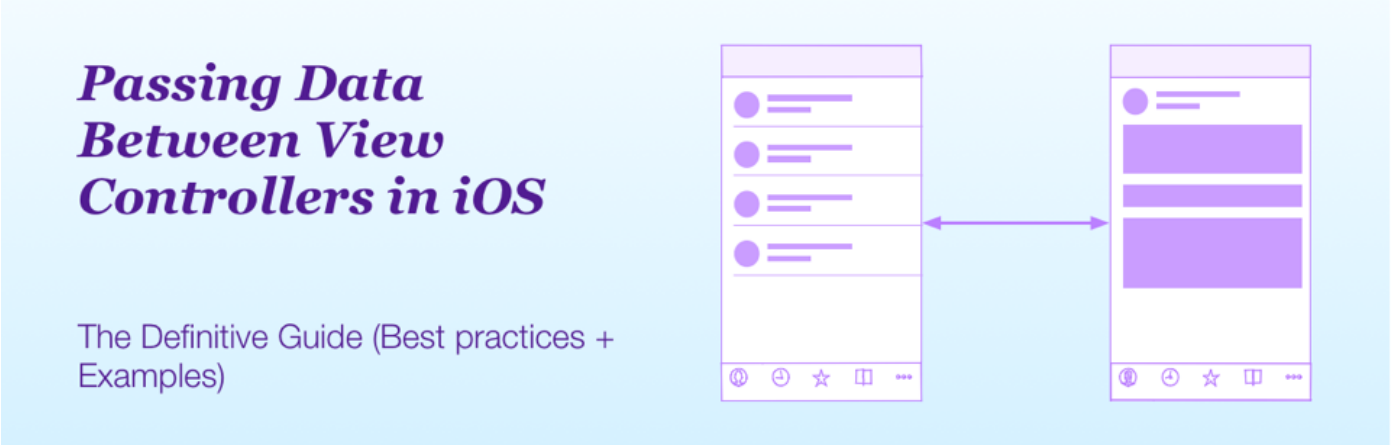
Passing Data Between View Controllers in iOS: the Definitive Guide (Best Practices + Examples)



One of the places where most iOS developers make architecture mistakes is passing data between view controllers.

There are many ways to actually do this, but how do you know which ones are the best ones?

Many developers use the wrong solutions because they are quick and easy.

Quick and easy solutions often mean you will get into problems later. I know it first hand from many client projects and even my own apps.

When Apple published the iOS SDK in 2008, I made a little game for the iPhone. I read some Apple guides and started coding my app from there.

Soon I found myself puzzled by how to pass data from one view controllers should to another.

I was coming from Mac development where there was no such thing as a view controller. It was a new concept for me, so I did what I could with my little experience.

Some years later, I decided to improve my game by adding some new features. At that point my experience in iOS development had grown a lot. When I opened the project and I looked at the architecture of the app, I wondered:

> *WTF was I thinking when I wrote this?*

You see, sometimes it is easy to look at someone else's code and think they are bad developers. The truth is that everybody goes through an initial phase of poor understanding of the platform.

But what is the fundamental difference between the few developers that will become highly skilled and all the other copy and paste developers?

- Good developers know they lack knowledge and take the time to grow it.
- Bad ones never bother, they go on with the little they know, trying to hack together something and calling it a day.

Do you want to be be a proficient iOS developer? Then you have to know the architecture ideas behind view controller communication.

I have already wrote about the MVC pattern in iOS and how an iOS app is structured. In those articles I have treated view controller as separate independent entities for simplicity, but they are not. They constantly need to communicate and pass data back and forth, for any non trivial app to work.

There are different ways to enable communication between view controllers. In fact, if you look at this question on Stack Overflow (which is among the most frequently asked about iOS), you can find all of them.

But are they all good?

Here are all the ways in which you can pass data between view controllers:

- when a segue is performed
- triggering transitions programmatically in your code
- through the state of the app
- using singletons
- using the app delegate
- assigning a delegate to a view controller
- through unwind segues
- referencing a view controller directly
- using the user defaults
- through notifications

A few of them are the best practices of view controller communication.

Others, though, are not so good. Like the ones I adopted in my little game (I was referencing view controllers directly).

What makes the best practices what they are? They respect the principles of good software architecture and common design patterns.

I checked again recently Apple's guide on view controllers and the information was in there all the time. But Apple's documentation is a bit terse. When I read it years ago I could not absorb all the concepts.

So let's have a look at hot to pass data between view controllers in more detail.

There are two directions in which a view controller can pass data to another view controller. For each one of these there are different techniques:

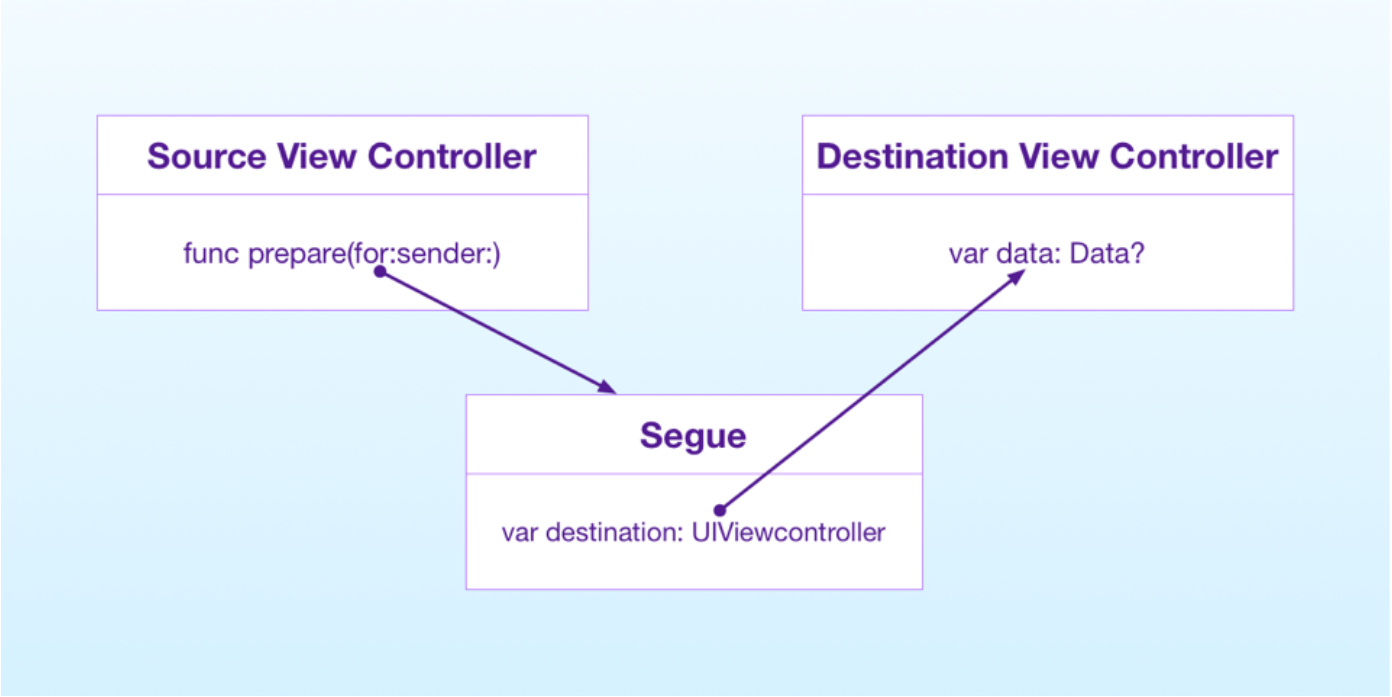
- **forward**, to the destination view controller of a transition
- **backwards**, to a view controller that was on the screen before and to which the user is will go back at some point.

So, how do you pass data between view controllers in Swift?

Let's explore all the examples.

Passing data between view controllers connected by a segue

The most common situation is when a transition happens through a storyboard segue.



Separate view controllers in a storyboards have no knowledge of each other. The only connection they have in the storyboard is the segue itself. This is exactly the mean through which they can communicate.

When a transition is triggered and a segue is performed, the `prepare(for:sender)` method is called on the source view controller.

The triggered segue, which is an object itself, is passed as a parameter to this method. Through its `destinationViewController` property the source view controller can access the destination view controller.

Let's look at an example of this.

Let's say we have a `Data` struct that we want to pass from an `SourceViewController` to a `DestinationViewController`

```
struct Data {}

class SourceViewController: UIViewController {}

class DestinationViewController: UIViewController {}
```

First of all we have to make the `DestinationViewController` able to receive such data

```
class DestinationViewController: UIViewController {
    var data: Data?
}
```

This property needs to be declared as optional because when the view controller is initialized it will not be able to receive this data in its initializer. I talked more about these initializers and other common methods in my article on the lifecycle of a view controller

We then pass the data from the `OriginViewController` to the `DestinationViewController` in the `prepare(for:sender:)` method of `SourceViewController`:

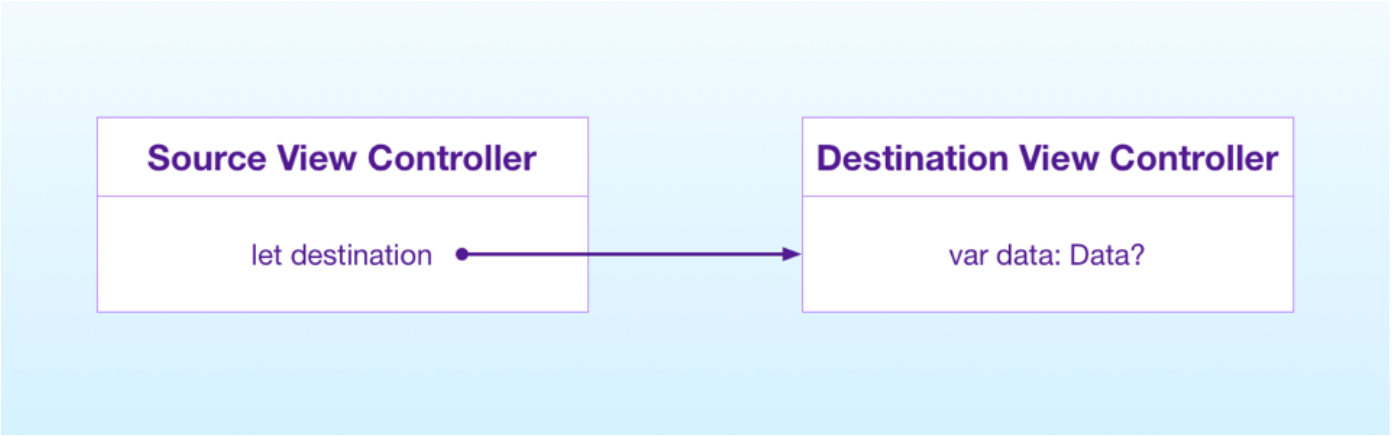
```
class SourceViewController: UIViewController {
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        let data = Data()
        if let destinationViewController = segue.destination as? DestinationV
            destinationViewController.data = data
        }
    }
}
```

Keep in mind that this method gets called for all segues that originate from a view controller.

To distinguish between different segues when there are more than one, you can either use an optional binding like in the example or the `identifier` property of the segue. The identifier is a string you can set for each segue in the storyboard.

Passing data between view controllers without a segue

There are some cases when a transition to a new view controller does not happen through a segue, but programmatically.



This might happen when the destination view controller:

- is in a different storyboard than the source view controller (although since Xcode 7 it is possible to use storyboard references, which enable segues between separate storyboards)
- is in a nib file
- does not have an interface file at all and creates its view hierarchy completely in code.

In this case, the passage of data is straightforward. Since the source view controller instantiates the destination view controller, it has a direct reference to the latter.

Let's see as an example, the modal presentation of a view controller that comes from a nib file:

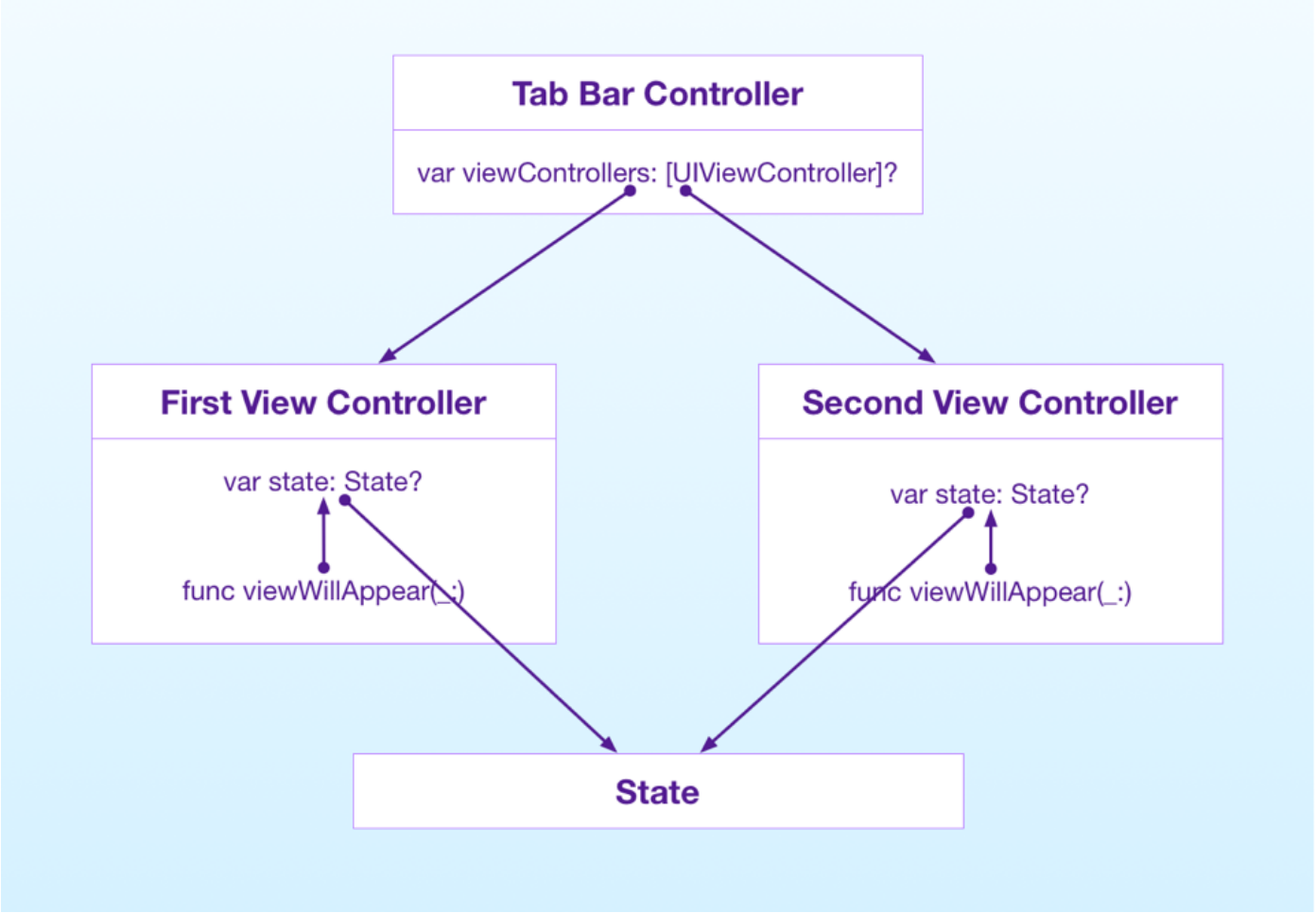
```
class SourceViewController: UIViewController {
    func presentDestinationViewController() {
        let data = Data()
        let destinationViewController = DestinationViewController(nibName: "D
        destinationViewController.data = data
        present(destinationViewController, animated: true, completion: nil)
    }
}
```

As you can see, the view controller creation and the passing of data happen at the same time.

Passing data between view controllers inside a tab bar controller

In the cases we saw above, the source view controller gets a reference to the destination view controller somehow.

There is a special case where this does not happen: when you use a tab bar controller.



When you switch between the tabs of a tab bar controller, the contained view controllers do not get a chance to be connected, and no segue gets triggered.

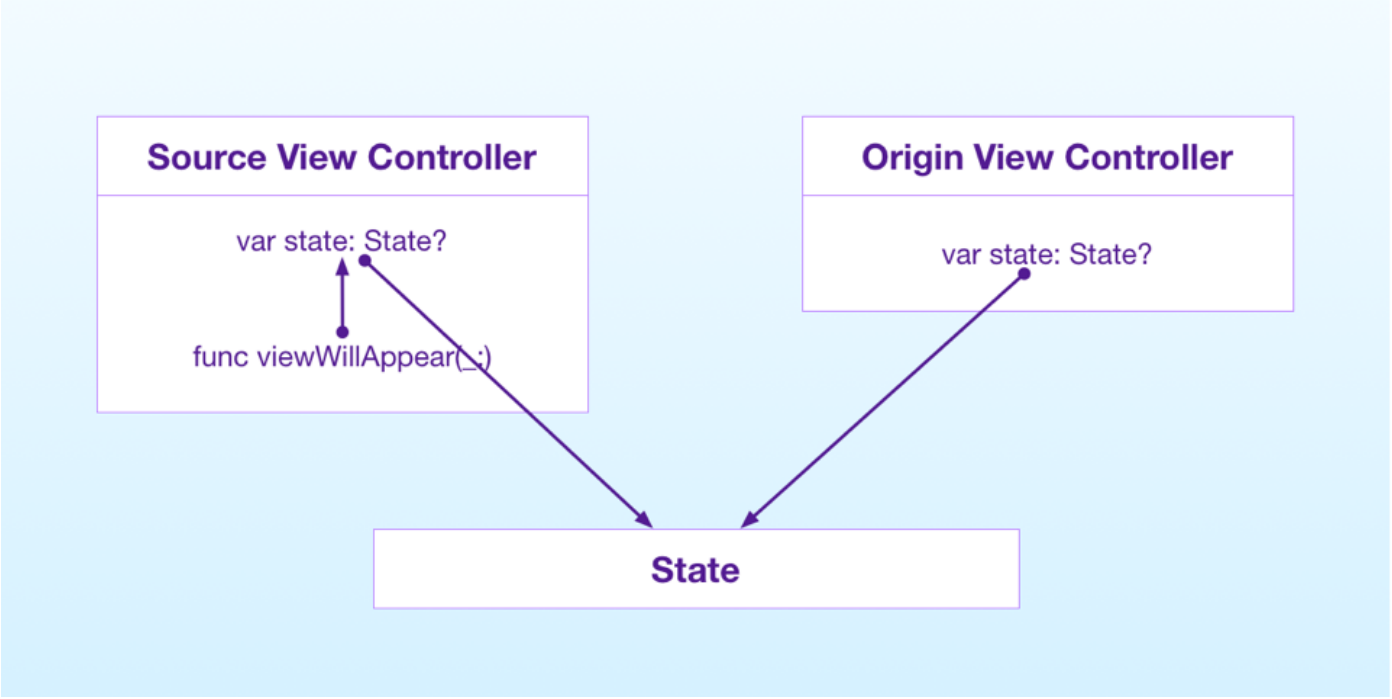
This is not necessary though. View controllers in a tab bar controller usually have no relationship, since they deal with independent parts of an app. There is no source and no destination.

This means that they don't need to pass data to each other.

What these view controllers do is read their data from the state of the app. You can refer to the next section to see how to read data from the state of an app.

Passing data backwards through the shared state of the app

The first way to send data back to a previous view controller is indirectly through the shared of the app.



View controllers usually share references to the model containing the current state of the app. The model of the app is usually passed forward between view controllers in the ways we have seen above.

To pass some data back the destination view controller simply updates the model, without needing any reference to the source view controller.

```
class State {}

class DestinationViewController: UIViewController {
    var state: State?
```

```
    func updateState() {
        // Updates the state
    }
}
```

When the previous view controller comes back on the screen, it just updates its user interface reading the model again, to which it also has a reference. This usually happens in the `viewWillAppear(_:)` method (read my article on the lifecycle of a view controller for more on this)

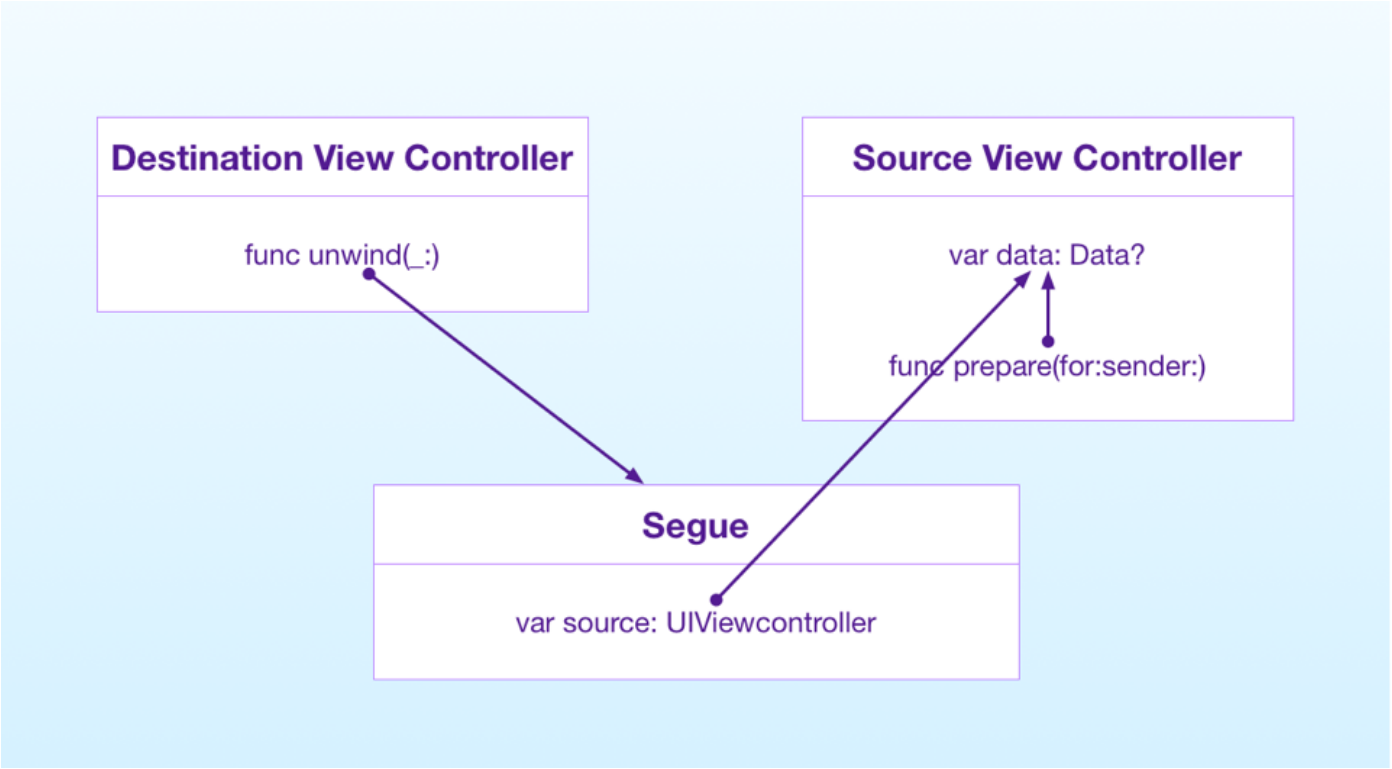
```
class SourceViewController: UIViewController {
    var state: State?

    override func viewWillAppear(_ animated: Bool) {
        // Updates the view controller interface using the updated state
    }
}
```

This is a common and easy way of passing data backwards, since it requires little code and no connection between the view controllers.

Passing data to the previous view controller through an unwind segue

If you use storyboards in your app, it makes sense to go back to previous view controllers through unwind segues.



If you are using a navigation controller, there is no unwind segue. The user goes back to the previous view controller by tapping on the back button and the navigation controller takes care of everything. In this case you use either the state of the app as we have seen above, or you use delegation, as we will see below.

You use unwind segues usually when you present a view controller modally.

Unwind segues work a bit differently than forward segues. To connect two view controllers through an unwind segue, you need to create and unwind action in the view controller you are going back to. You can see how that works here.

When the segue gets triggered, `prepare(for:sender:)` gets called in the source view controller as usual. Here you save the data you want to pass backwards in a property that the destination view controller will access to read the data.

```
class SourceViewController: UIViewController {
    var data: Data?

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        data = Data()
    }
}
```

Then the unwind action gets called in the destination view controller. Here, the

destination view controller reads the data from the property of the source view controller.

```
class DestinationViewController: UIViewController {
    @IBAction func unwind(_ segue: UIStoryboardSegue) {
        if let origin = segue.source as? SourceViewController {
            let data = origin.data
            // Do something with the data
        }
    }
}
```

You don't necessarily need to pass data back this way. You can pass data backwards through the state of the app even when you use an unwind segue. In that case you would:

1. Save the data in the state inside of the `prepare(for:sender:)` method of the source view controller
2. Read it in the `viewWillAppear(_:)` method of the destination view controller, as we did above

Passing data between view controllers using a delegate

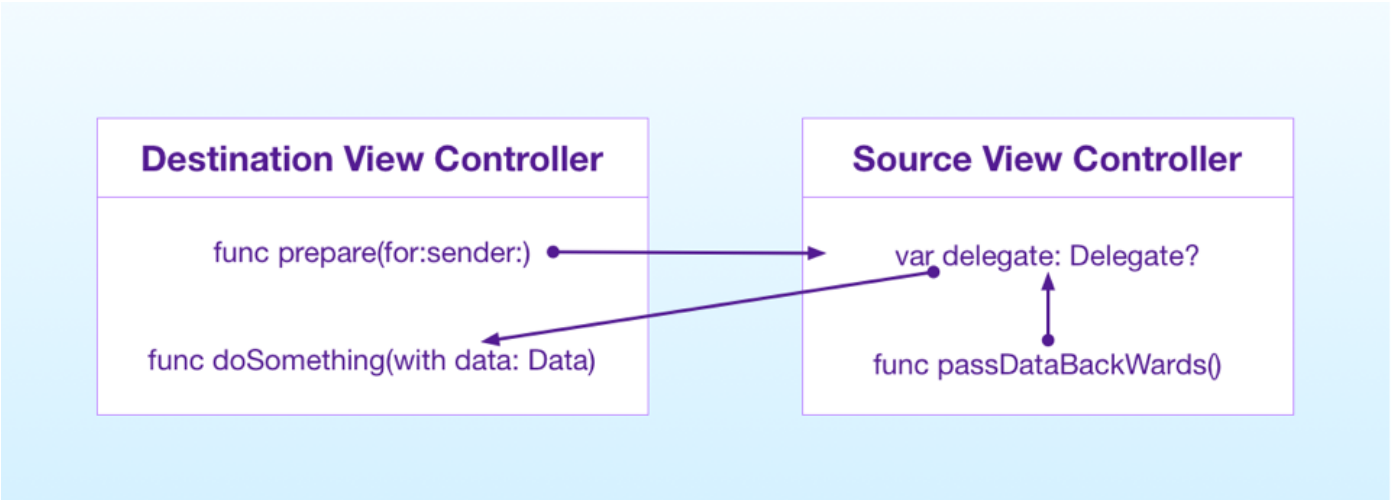
Sometimes the methods we have seen above are not enough.

- The data to pass back might be temporary and thus not belong to the shared state of the app
- When going back inside of a navigation controller, no unwind segue is triggered
- We might to pass data backwards at a different moment than the one of the transition

We need to create a connection between the source and the destination view controller.

The solution could be to pass a reference forward using one of the methods we have seen above and then use that to communicate backwards.

But it's not so simple.



The problem is that we cannot reference the previous view controller directly. Read the section below to see why.

The solution here is delegation.

What we need is a delegate protocol for the destination view controller. When the destination view controller needs to pass data back, it does so to its delegate.

```
protocol Delegate {
    func doSomething(with data: Data)
}

class DestinationViewController: UIViewController {
    weak var delegate: Delegate?

    func passDataBackwards() {
        let data = Data()
        delegate?.doSomething(with: data)
    }
}
```

The `delegate` property needs to be declared `weak` to avoid strong reference cycles. See this article for more details on weak and strong references.

The source view controller then conforms to such protocol to receive the data. Then, when passing data forward it has to set itself as the delegate for the destination view controller.

```
class SourceViewController: UIViewController, Delegate {
```

```
    func doSomething(with data: Data) {
        // Uses the data passed back
    }

    override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
        if let destinationViewController = segue.destination as? DestinationV
            destinationViewController.delegate = self
        }
    }
}
```

The advantage of this approach is that the destination view controller does not need to know which object is the delegate. As long as such object conforms to the protocol, it's fine.

Do not reference directly the previous view controller

The protocol approach could be substituted by direct knowledge about the previous view controller.

```
class SourceViewController: UIViewController {
    var data: Data?

    override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
        if let destinationViewController = segue.destination as? DestinationV
            destinationViewController.source = self
        }
    }
}

class DestinationViewController: UIViewController {
    var source: SourceViewController?

    func passDataToSourceViewController() {
        let data = Data()
        source?.data = data
    }
}
```

In this case we would lose the advantage of the protocol. The destination view controller now has specific knowledge of the source view controller.

What happens when we change the interface of this controller? Also in the flow of

an app, a view controller can be reached through different paths and thus through different source view controllers.

This clearly does not work. A view controller would need references to all the view controllers that could possibly come before itself.

Our destination view controller's code breaks and we have to change it. This is because we are introducing to much coupling between the two classes.

As Apple says in its own view controllers guide:

> *Always use a delegate to communicate information back to other controllers. Your content view controller should never need to know the class of the source view controller or any controllers it doesn't create.*

Do not use a singleton to share the app state

Singletons are an abused programming pattern that, especially in iOS, gets used for anything.

The reason a singleton is very easy to create and use. It looks like a perfect solution.

But it's not.

On the surface this approach looks similar to the example of passing the model between controllers we have seen above.

The problem is that singletons can be accessed directly from anywhere in the app. You have no control. Singletons introduce a lot of coupling in your code and make your objects hard to test in the future.

You can search on Google for “singleton anti-pattern” for more information about why singletons are bad. For example, you can read this Stack Overflow question.

Do not use the app delegate

Using the app delegate is the same as using a singleton.

Although technically speaking, the app delegate is not a singleton itself, it can be considered one.

This is because you access the app delegate instance through the shared `UIApplication` instance, which is itself a singleton. So this creates the same problems discussed above.

This line should never appear anywhere in your code:

```
let appDelegate = UIApplication.shared.delegate
```

Do not use UserDefaults

The purpose of `UserDefaults` is to store user preferences that persist between app executions.

Anything stored there will stay there unless explicitly removed, so it is not a mechanism to pass data between objects.

Also, you can only write simple data types in the user defaults and there is no compiler check on the data you write there. This means that some code at some point might change the format of such data and break code situated elsewhere in your app.

Finally, it’s again like using a singleton, because you usually access the user

defaults through the `shared` instance of the `UserDefaults` class.

In general, your app should have a single access point to the user defaults, through a custom class that every other part of your app uses. Refer to the app state example above to do this.

Do not use notifications

Notifications are a way to spread information to multiple objects to which you might not have direct access.

As such it can work also for passing data between view controllers. That does not mean you should use it for this reason.

Notifications create too much indirection and make your code hard to follow. When you post a notification you cannot be sure which objects will intercept it, which might lead to unexpected behavior in your app.

I have seen codebases haunted by bugs caused by objects receiving a notification when they should have not.

Notifications can be useful sometimes, but in general communication between view controllers should be done using the mechanisms we have seen above.

Should view controllers know about each other at all?

What we have seen are all the standard solutions used in iOS to pass data between view controllers. Many of these solutions require view controllers to know about other ones.

But should view controllers know about each other?

Since the flow of an app can change at any moment and many paths can lead to a single view controller, it might seem that we are introducing too much coupling between our view controller classes.

After all each view controller should only take care of their own screen, shouldn't it?

That is indeed true. You can use a more advanced and non standard architecture that includes coordinator objects that take care about the flow of an app.

That is exactly how the VIPER design pattern works, when you look behind all its jargon.

But although they are not so advanced, the techniques shown in this article are solid and widely used among iOS developers, so there is nothing bad in using them if you are not ready to adopt more advanced practices in your project.

I used myself in many projects. You can switch to a more complex architecture when you really start understanding their need from experience.