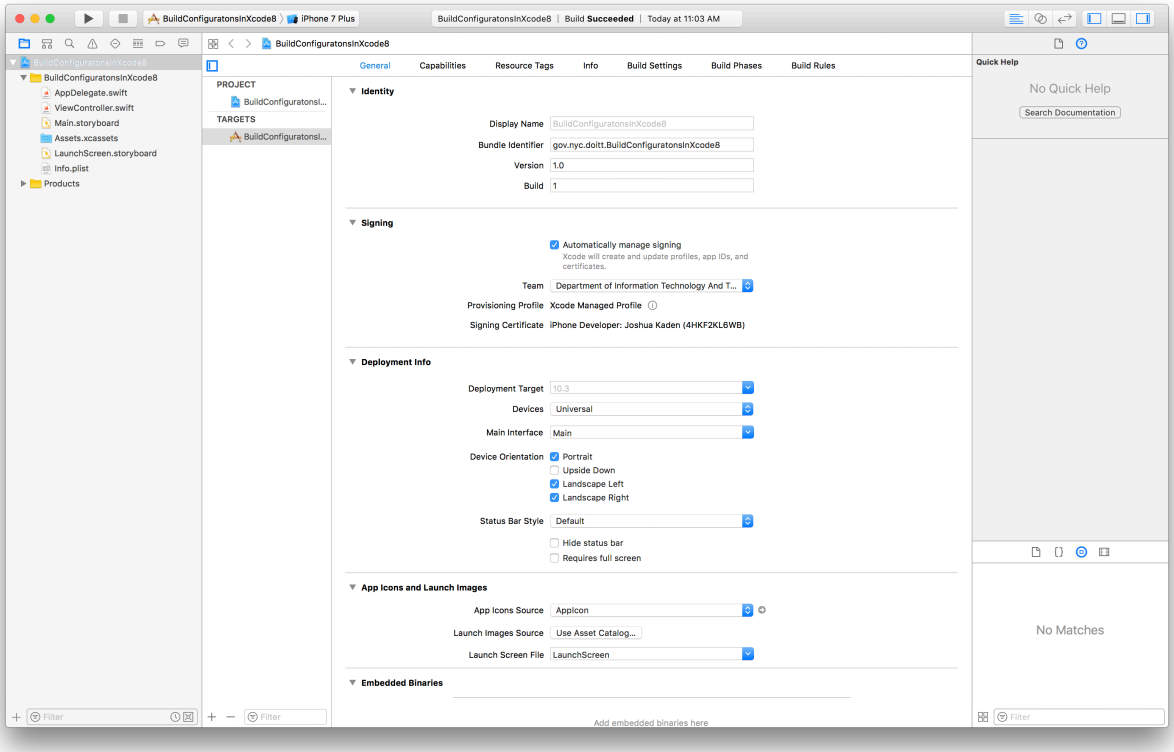
This article describes the technique of using schemes in Xcode to control configuration settings, for iOS mobile application development. This method allows for great flexibility in the configuration of an app, while minimizing the possibility of human error.

Step 1

NEW PROJECT

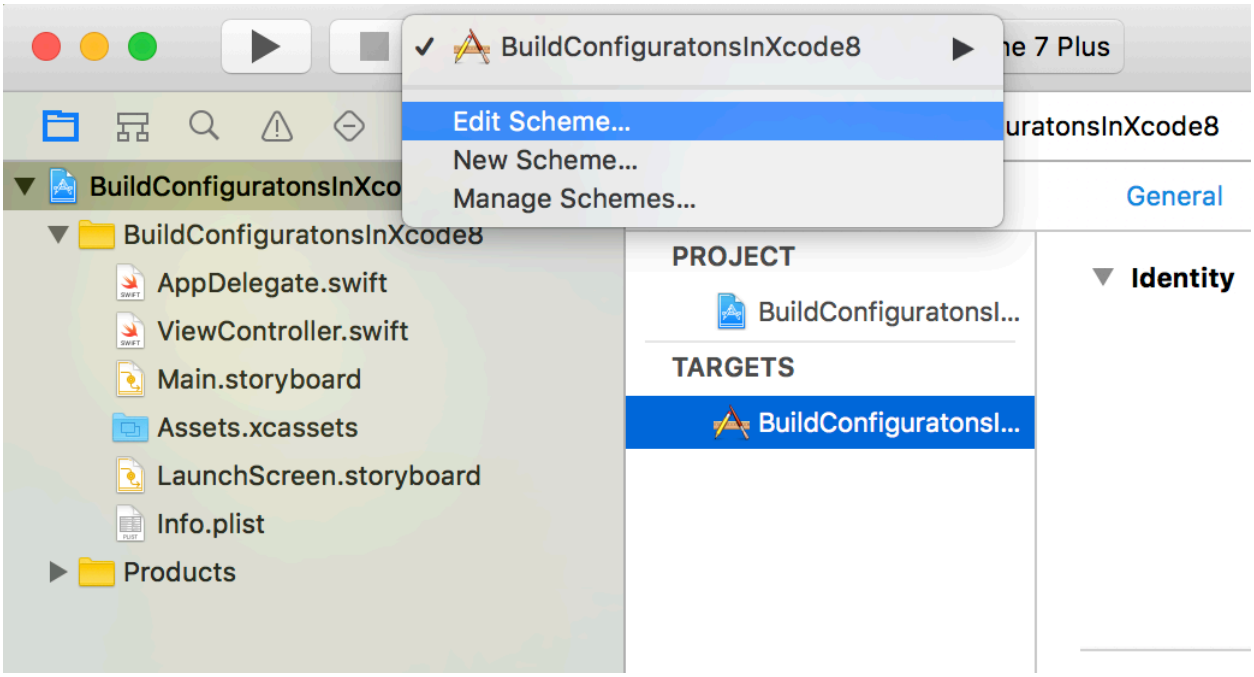
Open Xcode, and create a new iOS project. Select the Single View Application option.



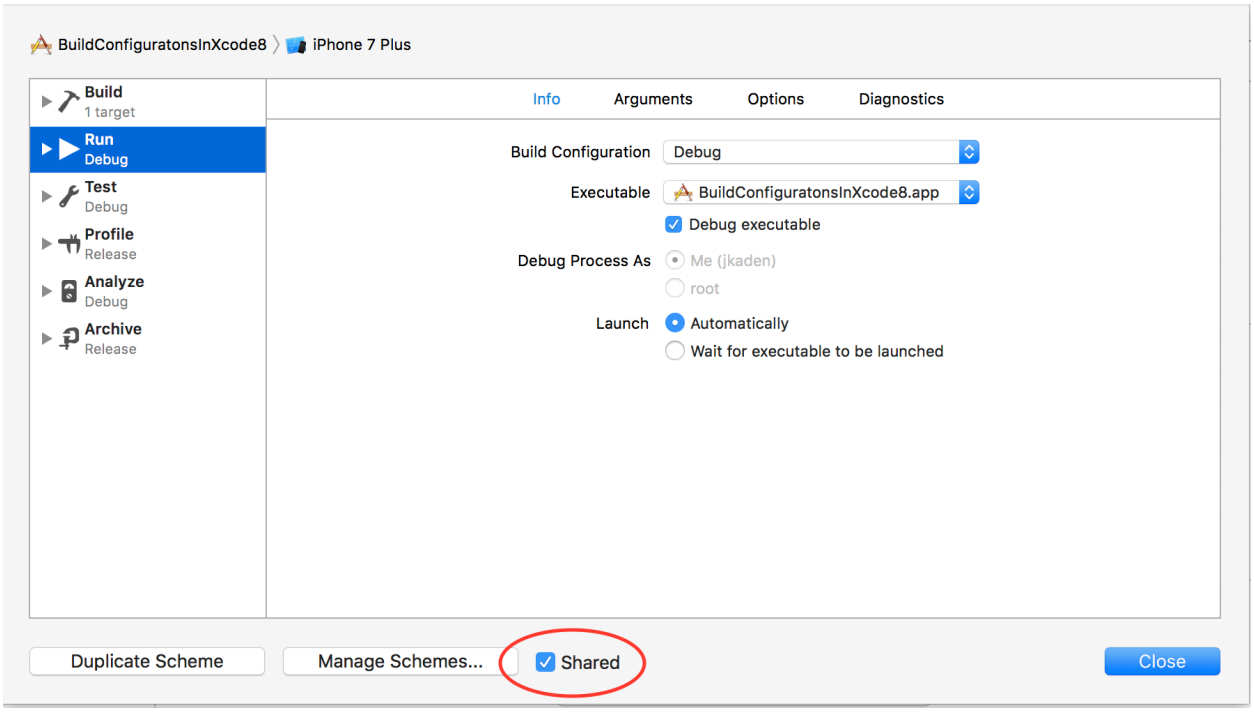
Step 2

SHARED SCHEME

Edit the default scheme by selecting it, and clicking “Edit Scheme”.



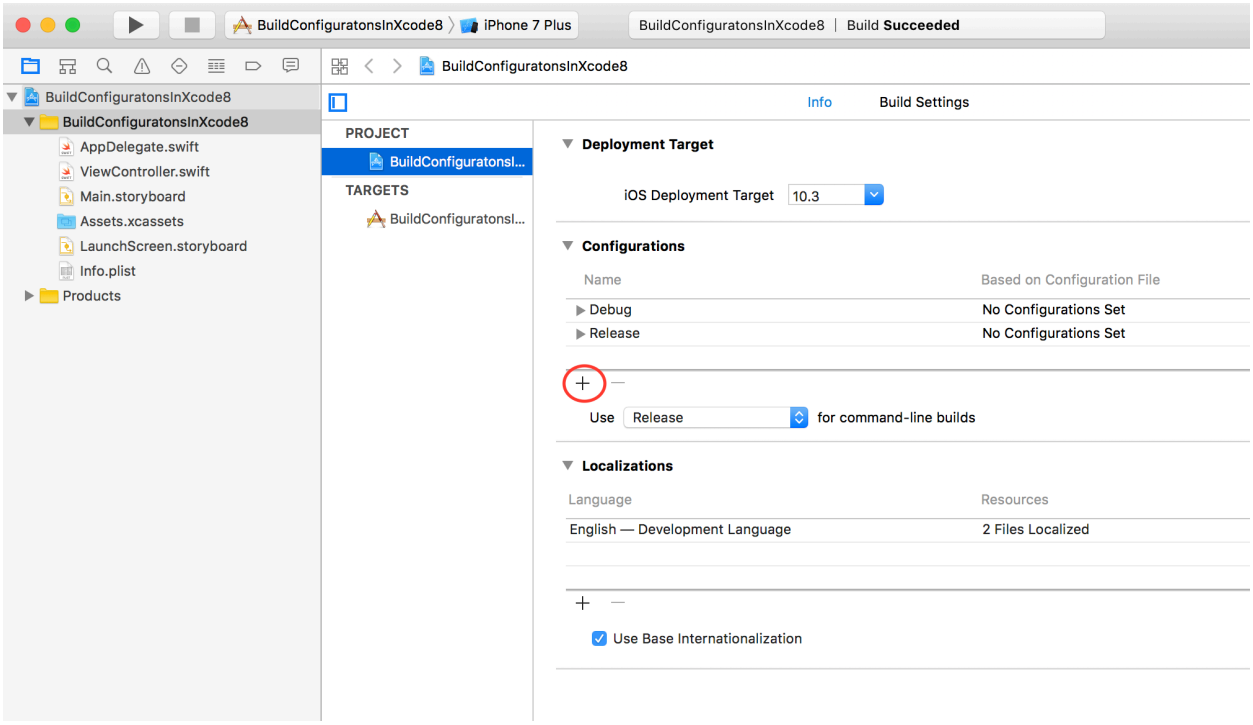
Check the “Shared” checkbox.



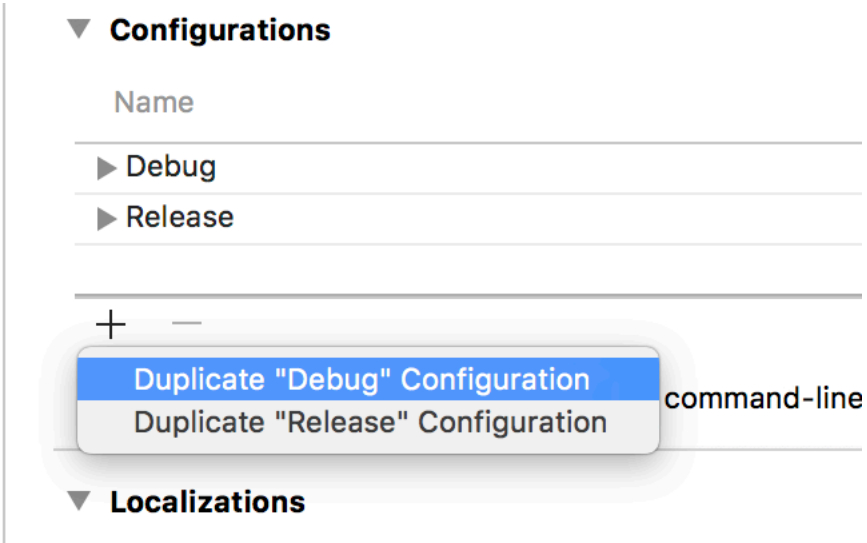
Step 3

DUPLICATE CONFIGURATIONS

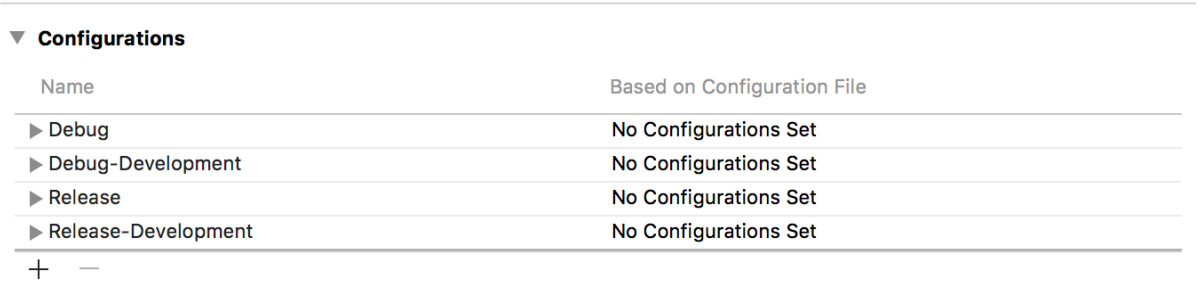
Select the project in the Project Navigator. On the Info tab of the project, click the “+” symbol at the bottom left of the Configurations section.



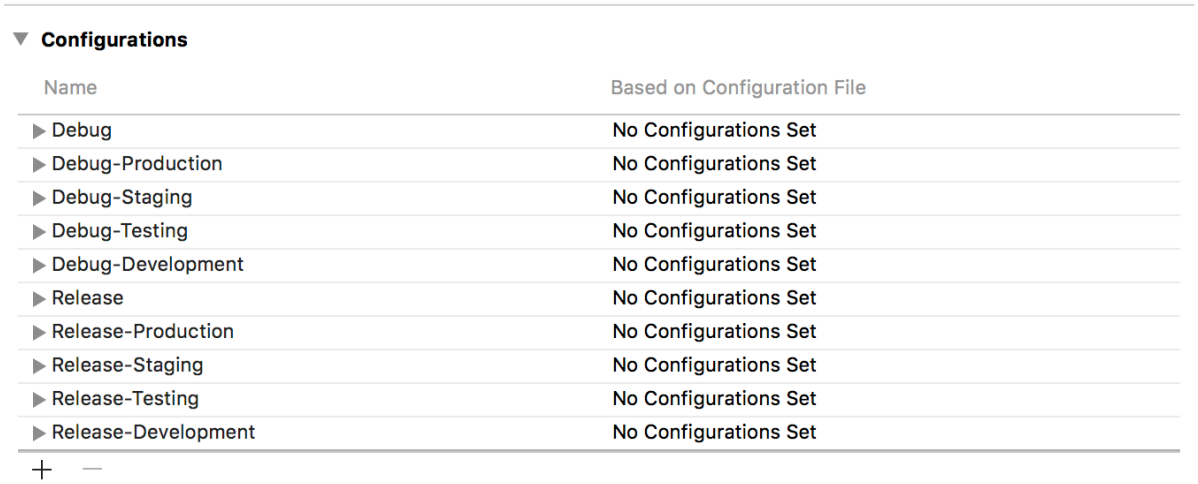
On the pop-up menu, click “Duplicate ‘Debug’ Configuration”.



Name the new configuration: “Debug-Development”. Repeat the process by duplicating the “Release” configuration, and use the new name “Release-Development”.



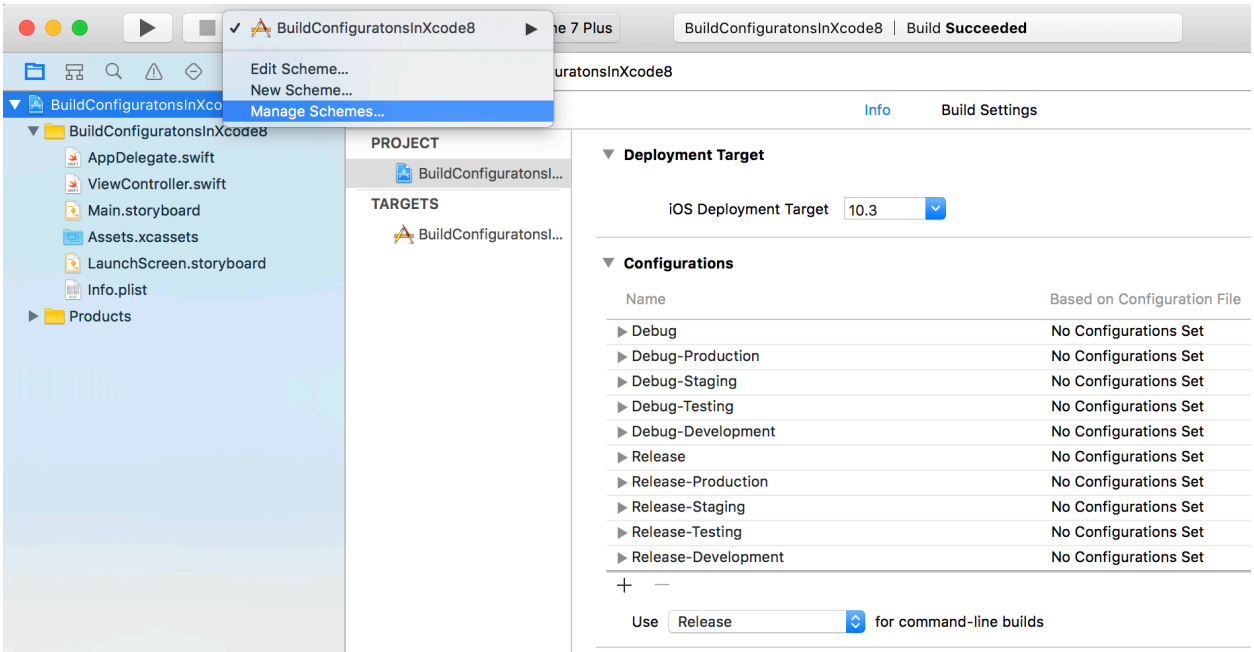
Repeat these two steps for the other three environments: Testing, Staging, and Production.



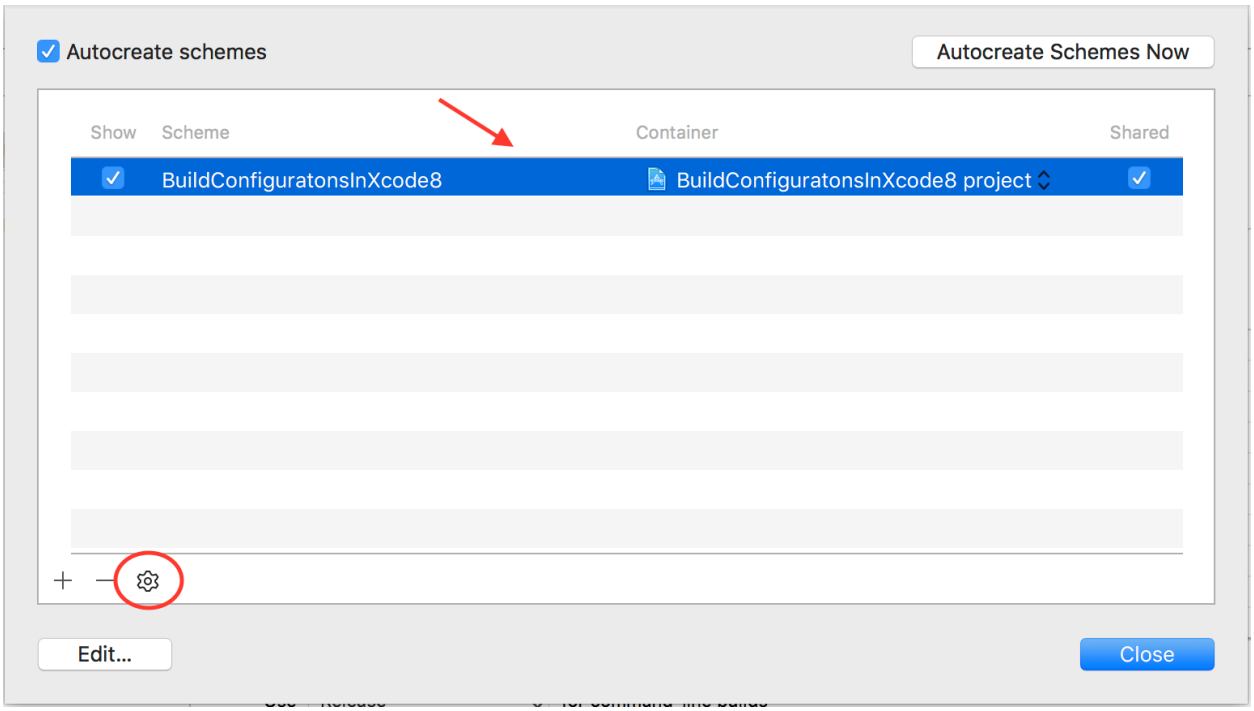
Step 4

DUPLICATE SCHEMES

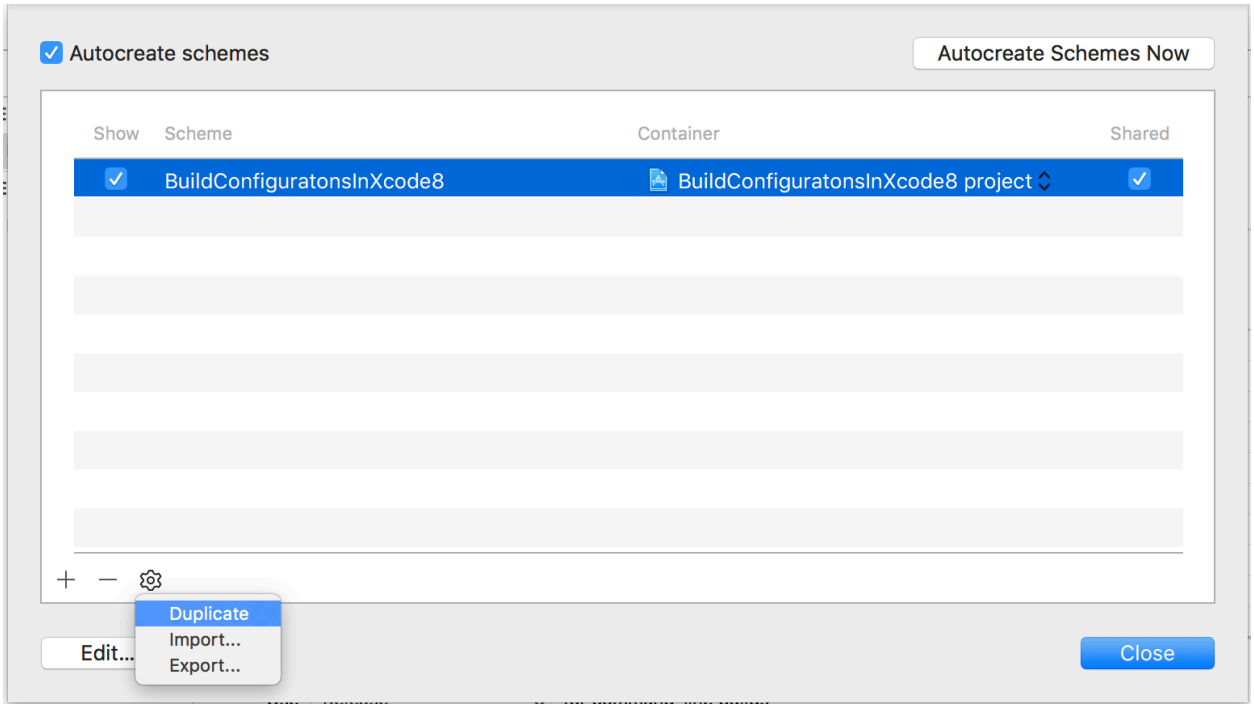
Launch the list of schemes by selecting the current scheme, and clicking “Manage Schemes”.



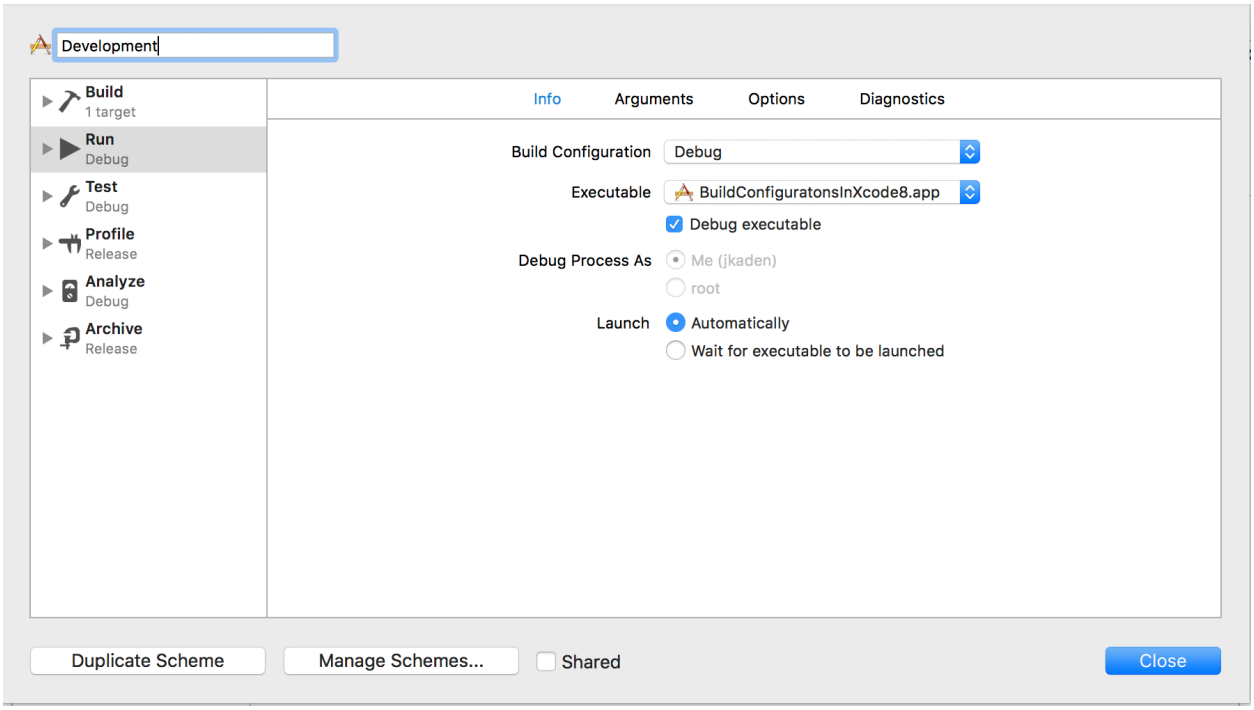
Select the listed scheme, and click the gear icon, at the window's lower left.



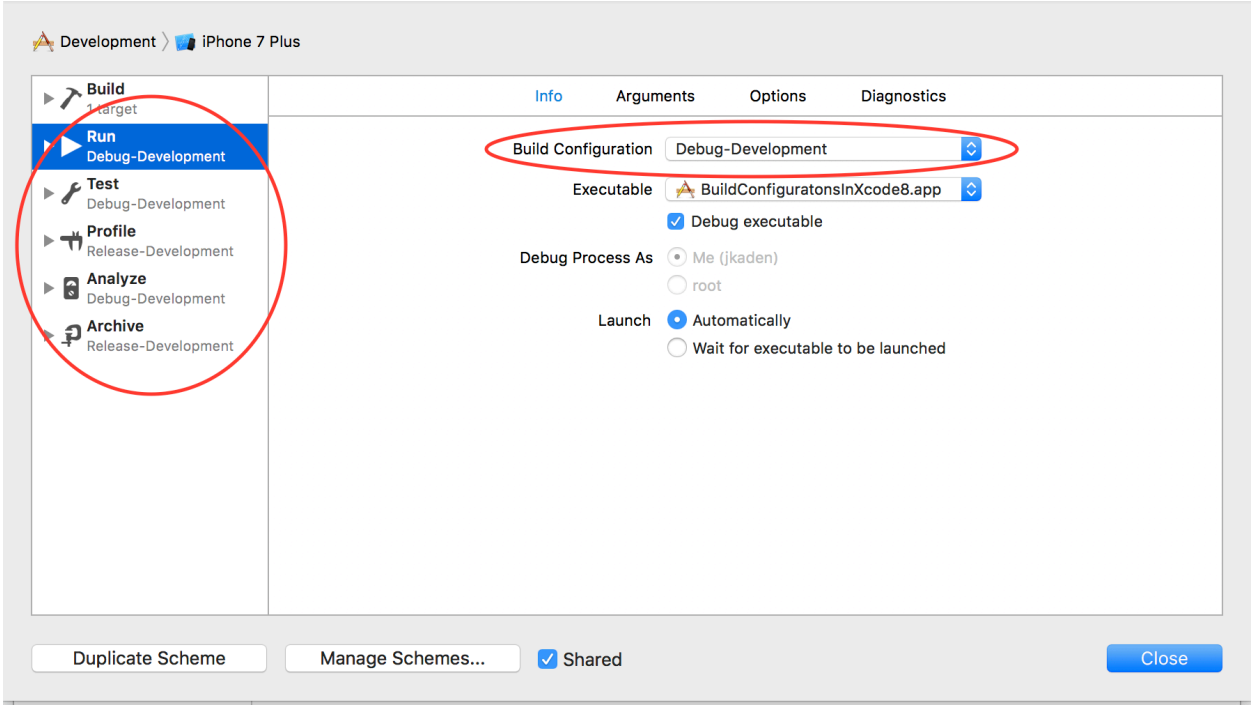
On the pop-up menu, click "Duplicate".



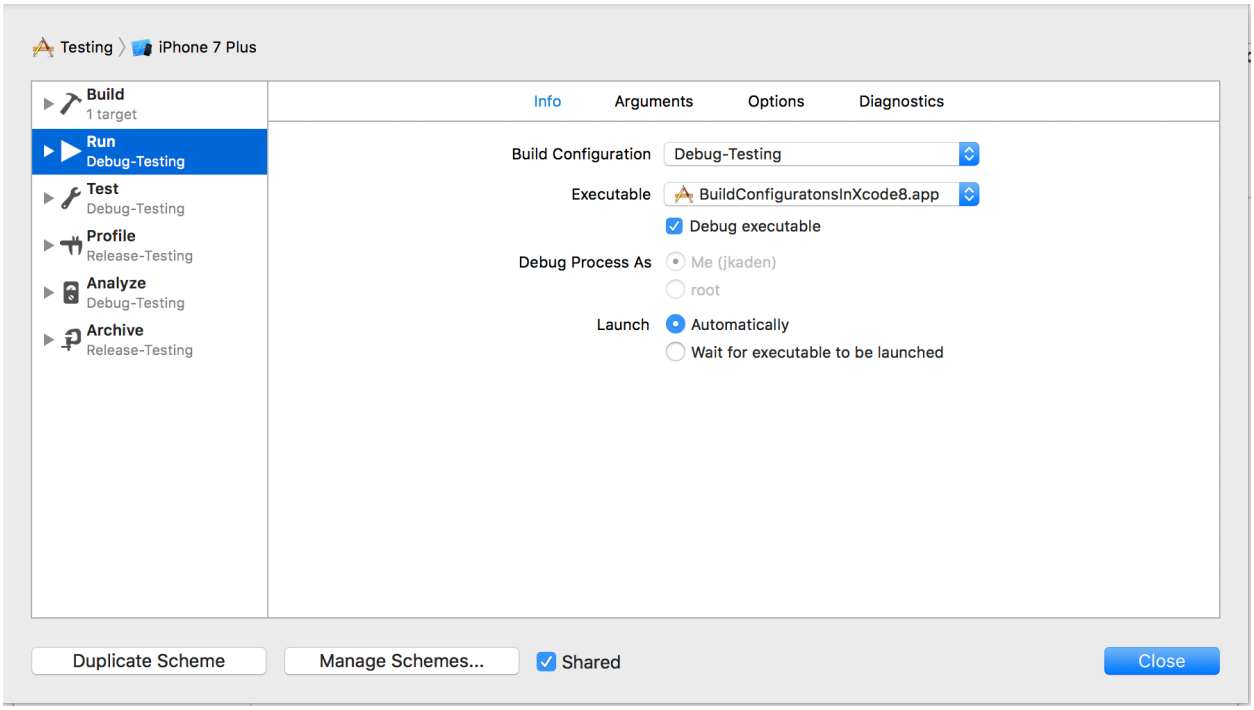
Name the new scheme “Development”.



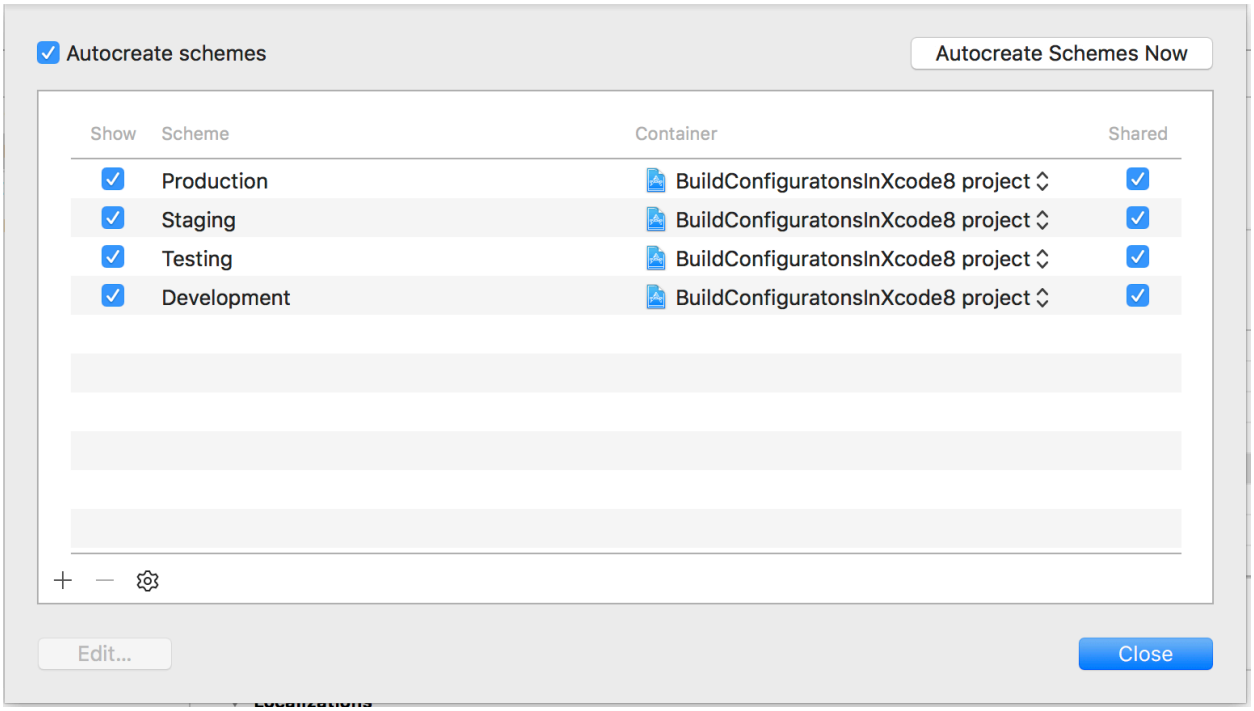
Select the Run step, and, on the Info tab, change the Build Configuration to “Debug-Development”. Do the same for Test and Analyze. For Profile and Release, change the Build Configuration to “Release-Development”. Also, make sure that “Shared” is checked.



Click Close to get back on the list of schemes. Select the default scheme, and duplicate it again, this time using the new name “Testing”. Change the Testing scheme’s Build Configurations by following the same pattern as before, as shown. Note that this time, we use the “Debug-Testing” and “Release-Testing” configurations.



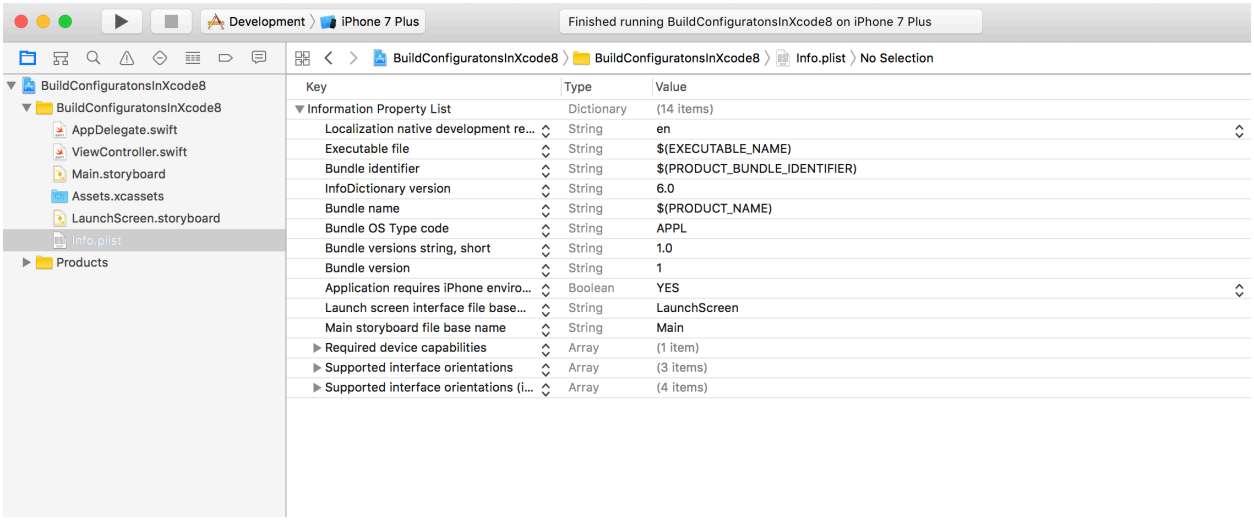
Create a new scheme named Staging, and one named Production, both by duplicating the default scheme. Set the proper Build Configurations for each of the new schemes. When you’re done, you may delete the default scheme, if you wish.



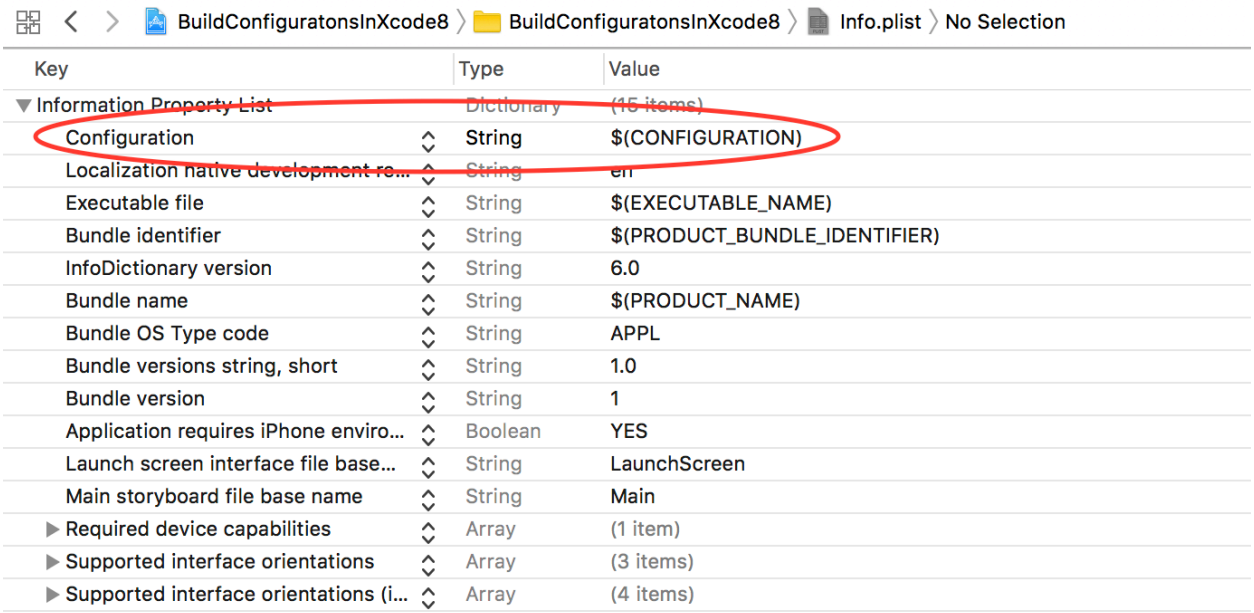
Step 5

MODIFY INFO PLIST

Select Info.plist on the project navigator.



Create a new key named “Configuration”. For its value, enter “$(CONFIGURATION)”. Now the system-supplied configuration value (as determined by the active scheme) is



accessible to your code via the info plist.

Step 6

READ CONFIGURATION AND DISPLAY IT

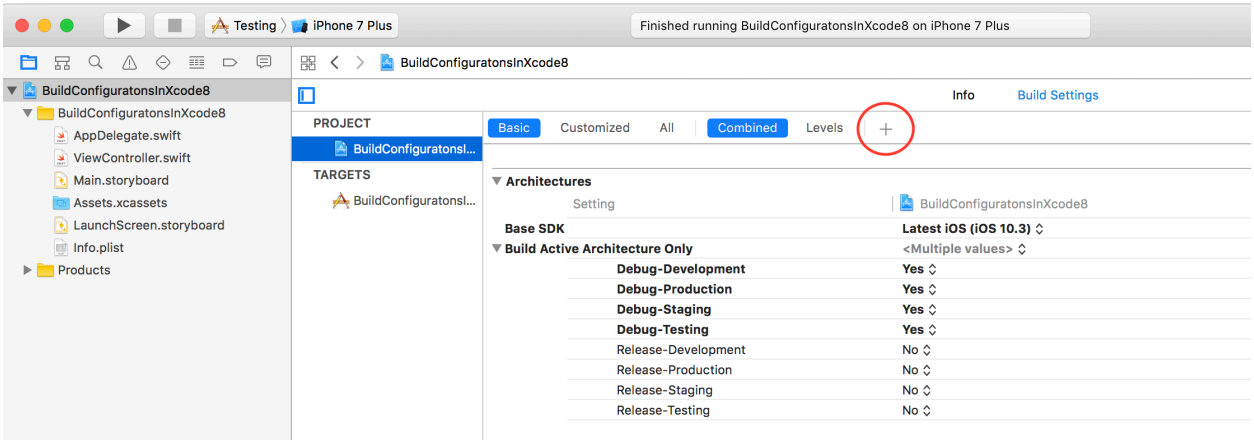
Info.plist is exposed as a dictionary via the `Bundle.main.infoDictionary` property.

Modify the ViewController class as shown below. This code reads the active configuration, and displays it on the screen.

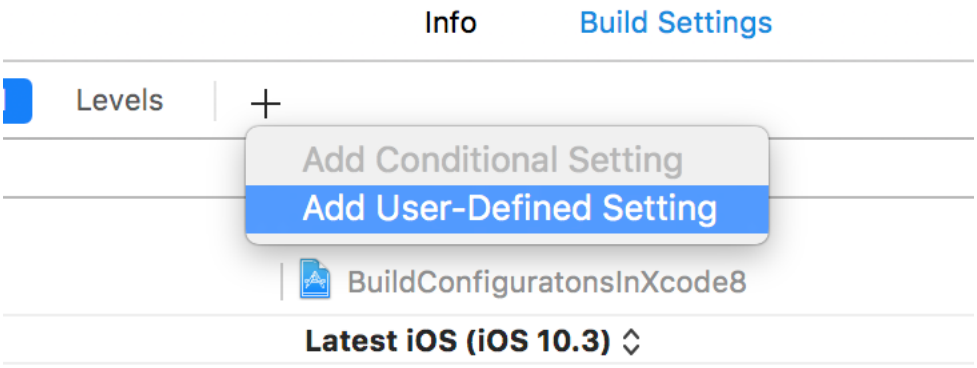
Step 7

USER-DEFINED BUILD SETTING

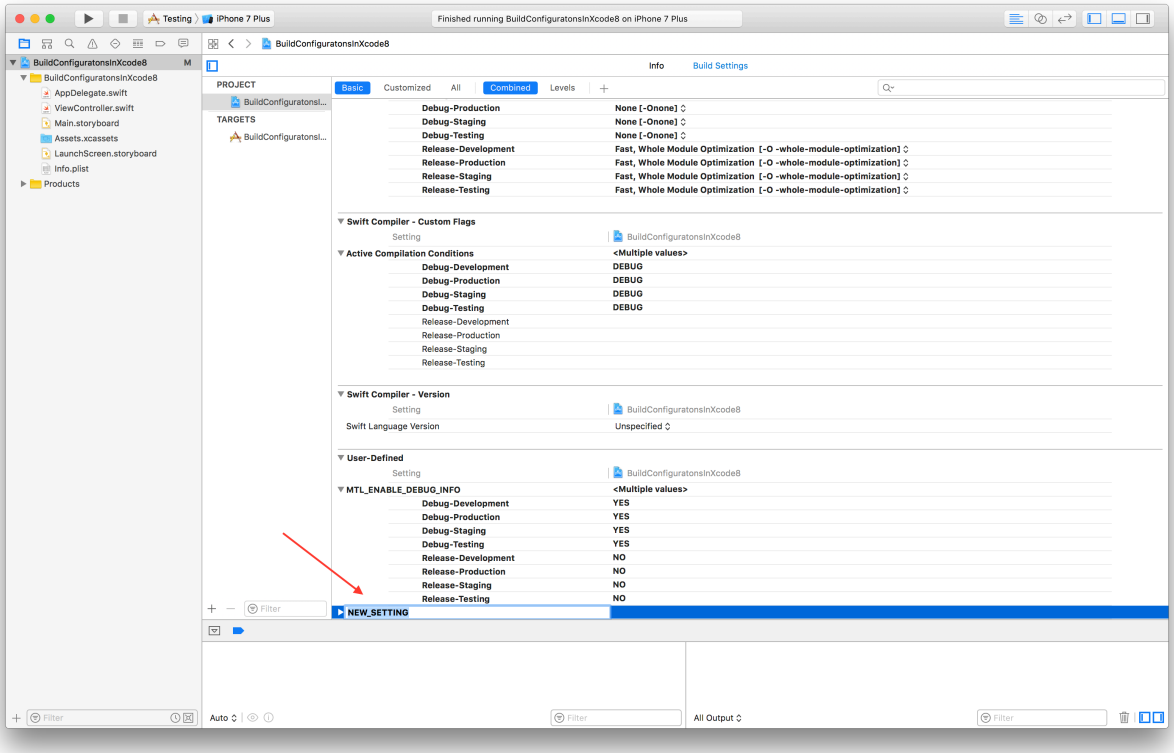
Select the project in the Project Navigator. On the Build Settings tab of the project, click the “+” symbol at the top.



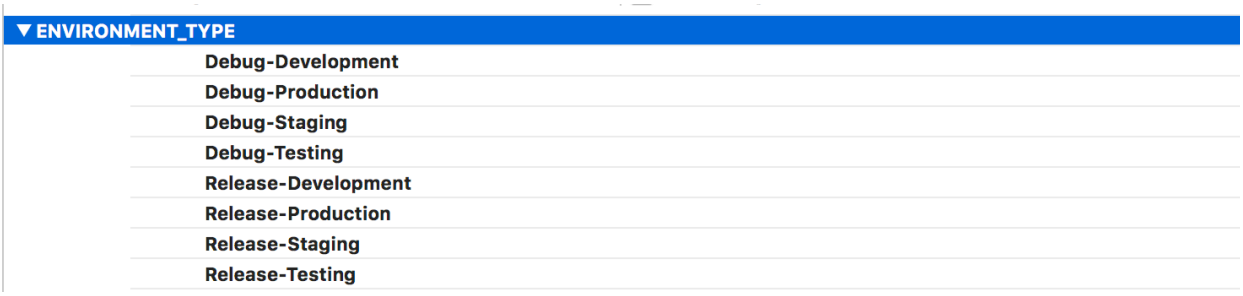
On the pop-up menu, click “Add User-Defined Setting”.



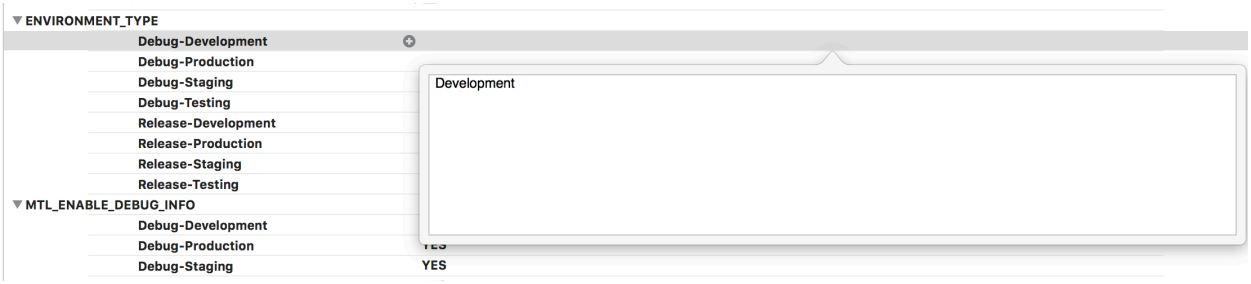
A new row will be added at the bottom of the list, in the User-Defined section.



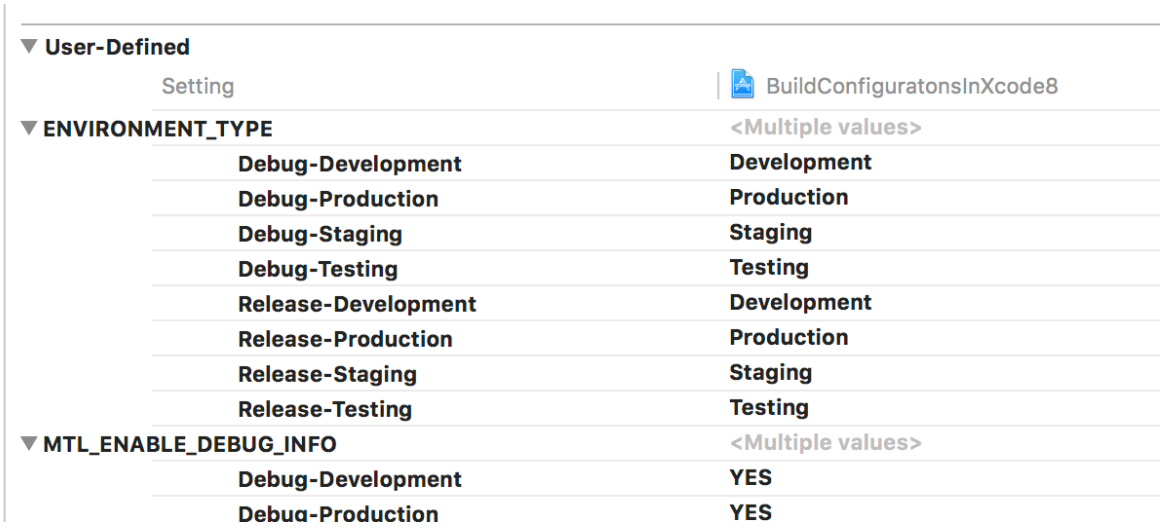
Name the new row “ENVIRONMENT_TYPE”. Your list will look like this:



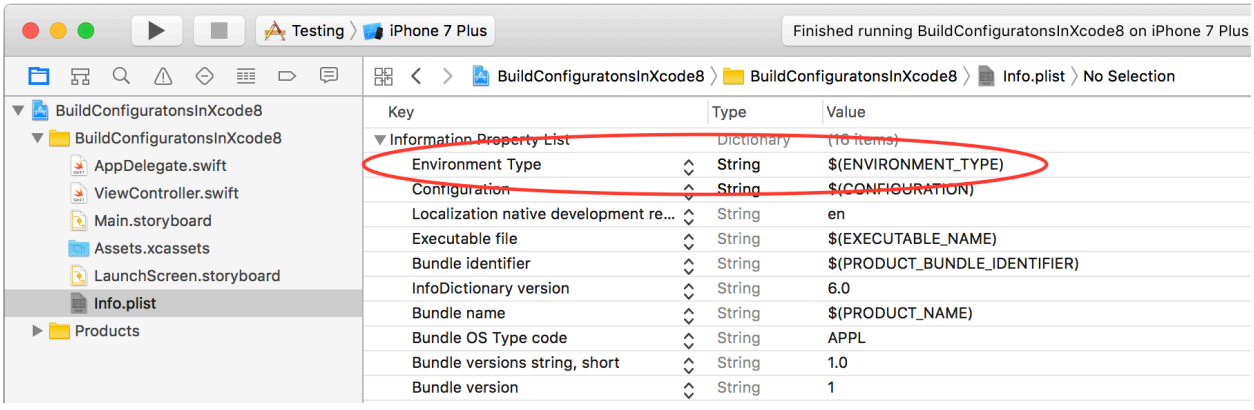
Double-click the Value for the Debug-Development row. In the pop-up editor, enter “Development”.



Add Values for the remaining configurations, so that your list looks like this:



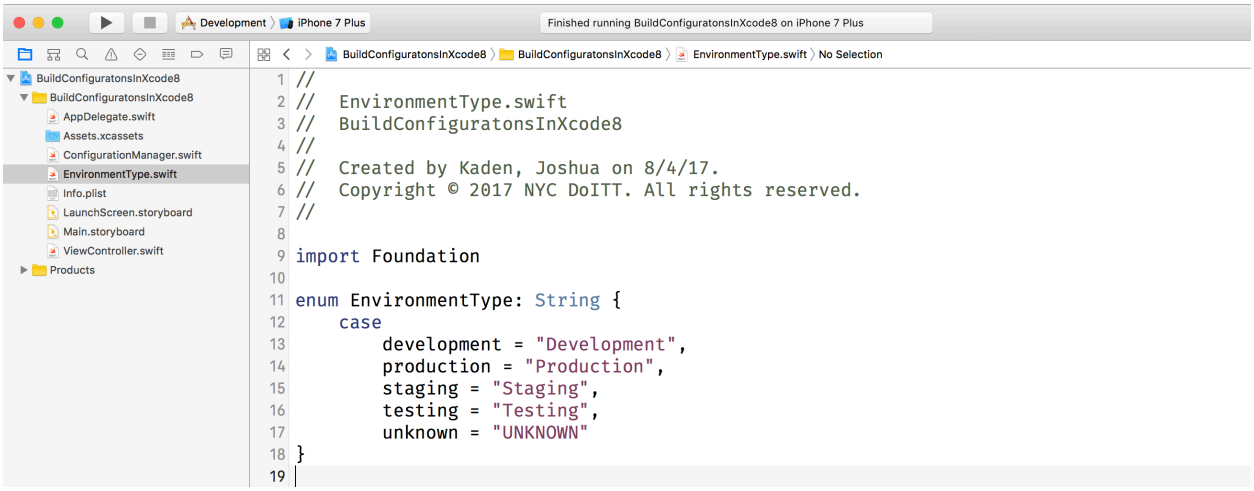
Back on Info.plist, create a new row named “Environment Type”. Give it a value of “$(ENVIRONMENT_TYPE)”.



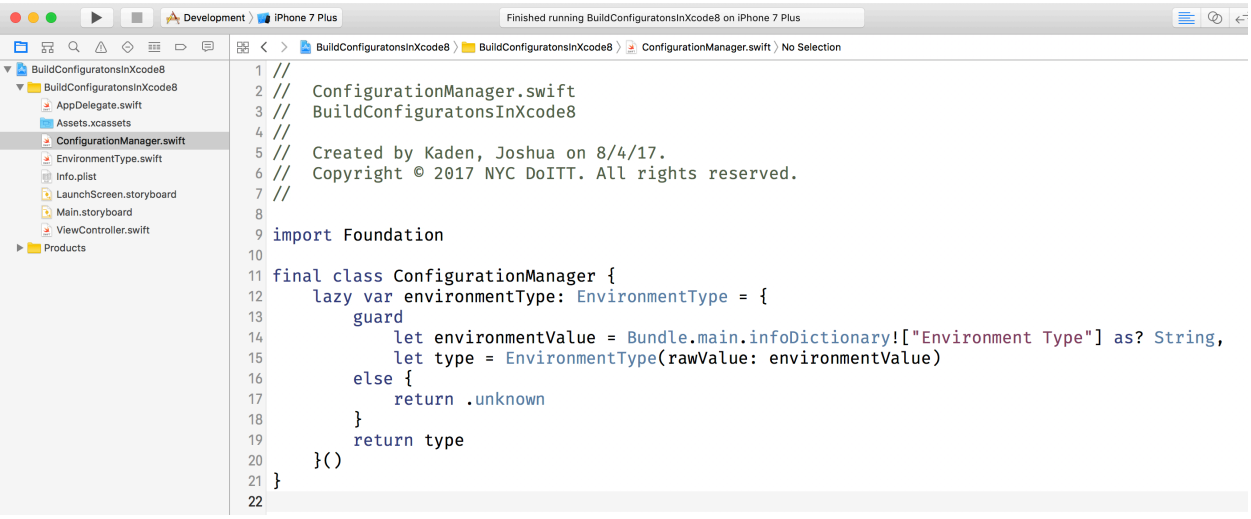
Step 8

CREATE ENUM AND MANAGER

Create an enum named “EnvironmentType”. This will act as a link between the string values in Info.plist and your code.



Create a class named “ConfigurationManager”. Its job is to read from Info.plist and supply the current EnvironmentType.



Modify ViewController to display the current EnvironmentType.

```swift
//
//  ViewController.swift
//  BuildConfiguratonsInXcode8
//
//  Created by Kaden, Joshua on 8/1/17.
//  Copyright © 2017 NYC DoITT. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    private let configurationManager = ConfigurationManager()
    private var environmentLabel = UILabel()

    override func viewDidLoad() {
        super.viewDidLoad()

        let environmentValue = configurationManager.environmentType.rawValue
        environmentLabel.text = "Environment Type: \(environmentValue)"
        view.addSubview(environmentLabel)
    }

    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()

        environmentLabel.sizeToFit()
        environmentLabel.center = view.center
    }
}
```

Step 9

MAKE DIFFERENT URLS FOR EACH ENVIRONMENT

Enhance the EnvironmentType enum to return a different URL for each value.

BuildConfiguratonsInXcode8 〉 BuildConfiguratonsInXcode8 〉 EnvironmentType.swift 〉 No Selection

```swift
//
//  EnvironmentType.swift
//  BuildConfiguratonsInXcode8
//
//  Created by Kaden, Joshua on 8/4/17.
//  Copyright © 2017 NYC DoITT. All rights reserved.
//

import Foundation

enum EnvironmentType: String {
    case
        development = "Development",
        production = "Production",
        staging = "Staging",
        testing = "Testing",
        unknown = "UNKNOWN"

    func serverURL() -> URL? {
        switch self {
        case .development:
            return URL(string: "https://dev-server.nyc.gov")
        case .production:
            return URL(string: "https://prod-server.nyc.gov")
        case .staging:
            return URL(string: "https://stg-server.nyc.gov")
        case .testing:
            return URL(string: "https://tst-server.nyc.gov")
        case .unknown:
            return nil
        }
    }
}
```

Here is code to simulate the launch of the URL associated with the current environment.

```swift
//
//  ViewController.swift
//  BuildConfiguratonsInXcode8
//
//  Created by Kaden, Joshua on 8/1/17.
//  Copyright © 2017 NYC DoITT. All rights reserved.
//

import UIKit

class ViewController: UIViewController {

    private let configurationManager = ConfigurationManager()
    private let urlButton = UIButton()

    override func viewDidLoad() {
        super.viewDidLoad()

        let environmentValue = configurationManager.environmentType.rawValue
        urlButton.setTitle("Open \(environmentValue) URL", for: .normal)
        urlButton.setTitleColor(.blue, for: .normal)
        urlButton.addTarget(self, action: #selector(didTapUrlButton(_:)), for: .touchUpInside)
        view.addSubview(urlButton)
    }

    func didTapUrlButton(_ sender: UIButton) {
        guard let url = configurationManager.environmentType.serverURL() else { return }
        let message = "URL is: \(url.absoluteString)"
        let alert = UIAlertController(title: "Button Tap", message: message, preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "Ok", style: .default, handler: { _ in }))
        present(alert, animated: true)
    }

    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()

        urlButton.frame.size = CGSize(width: 250, height: 60)
        urlButton.center = view.center
    }
}

```

The sample code for this article can be found at https://github.com/JoshuaKaden/BuildConfigurationsInXcode8.

Hopefully this will be enough to get you started with implementing build configurations in your project.