Apple Pay™ iOS SDK Quick Start Guide

Quickly integrate Apple Pay™ into your iOS apps with Payeezy from First Data

If you want to enable secure and convenient in-app payments in your iOS app this guide will get you up and running in minutes. The Payeezy Service from First Data was created to simplify your integration with Apple Pay™. Payeezy handles all the heavy lifting of the complex cryptography that protects your customers' transactions. It also makes it super simple to create a developer test account and even apply for a merchant account all through our developer portal.

TABLE OF CONTENTS

Here's an overview of the steps you'll take to enable Apple Pay™ in your app:

- Register a free Payeezy developer account
- Download the Payeezy Apple Pay™ SDK
- Integrate the SDK into your app

Let's get started!

REGISTER YOUR FREE PAYEEZY DEVELOPER ACCOUNT

Start Sending Test Transactions Within Minutes

Hop over to the Payeezy developer registration page and get your own free account. We'll take care of setting up your own personal sandbox so you can start integrating and testing Apple Pay™ transactions in seconds. All you will need to give us is your name and a valid email address.

After you provide registration information an email is sent to you with further instructions. Here's an example.

Thank You for Registering

Before you can begin, you will need to activate your Payeezy[SM] developer account by clicking on the activation button below.

CLICK HERE TO ACTIVATE

Steps To Take

1. Follow the "Click Here To Activate" button where you will be prompted to set up your password
2. If the activation button does not work, you can cut-and-paste this link into your browser:
   https://developer.payeezy.com/user/reset[redacted]
3. For security purposes, this link can only be used once
4. After setting your password, you will be able to log into your account with:
   Username: [redacted]
   Password: [redacted]
5. After completing these steps, you can log in with your new password along with the username you set up at:

LOG IN

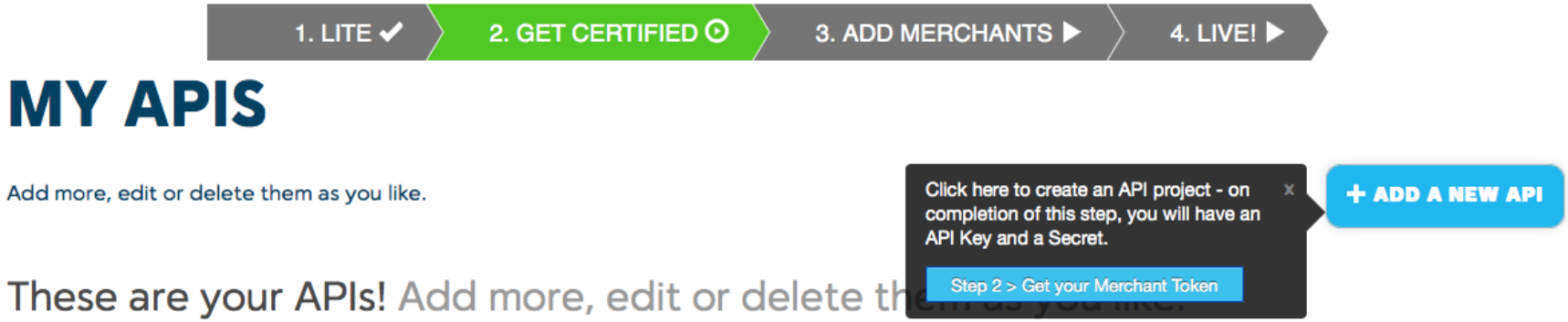
Or by using this link:

https://developer.payeezy.com/user

Thank you
Payeezy Dev Portal Team

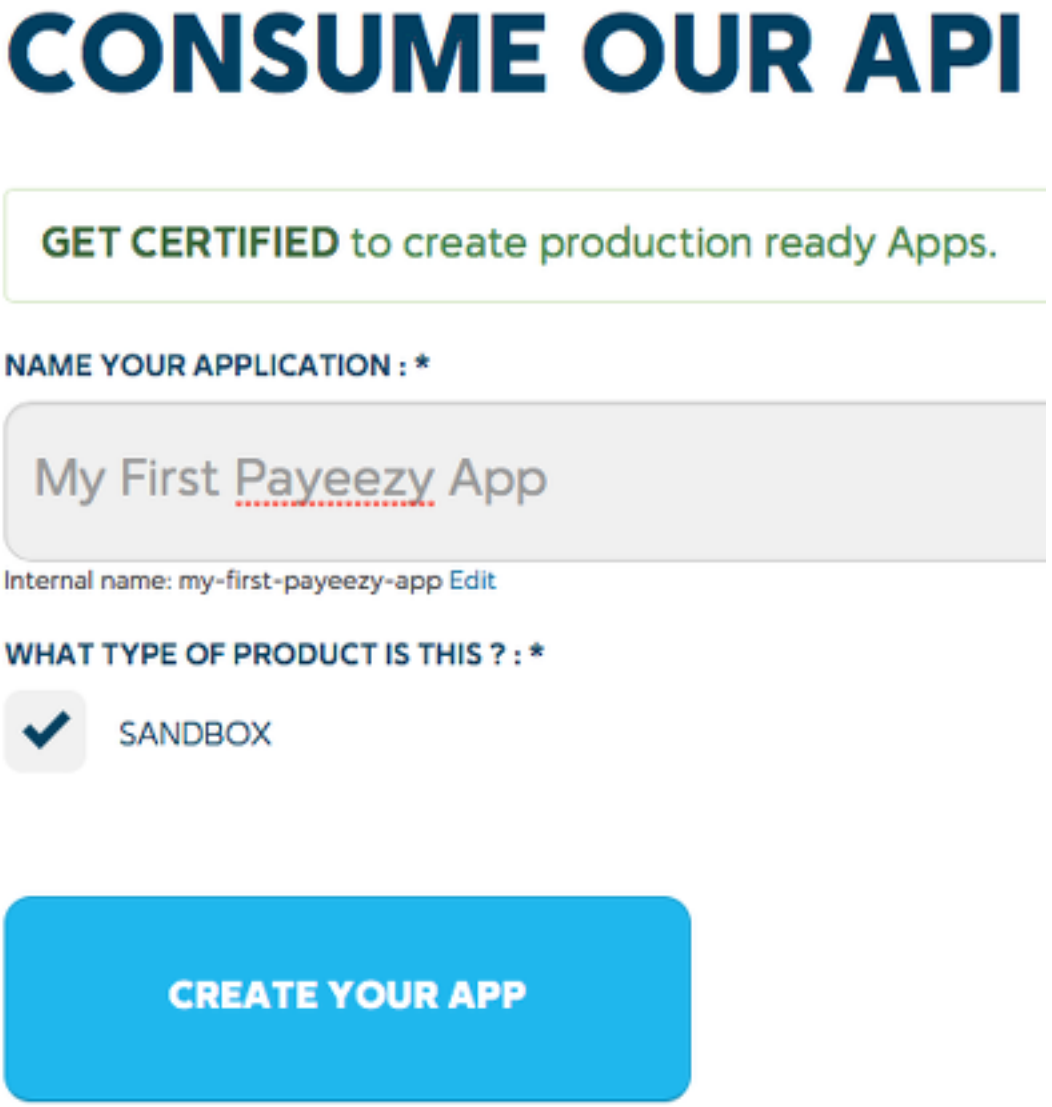
Be sure to complete your registration from the link, username and password provided in the email. When you process the initial registration you will be forced to select a new, more secure, password for your account. From this point you will be prompted for security questions that can be used if your password is forgotten.

The next step is to have you create an “API” on the Payeezy portal.
Think of this as a unique app name that is used to identify your application. The name is unrelated to the name you would use on Apple’s App Store but it can be the same. From the “Get Certified” tab on this page…



Click the “ADD A NEW API” button.

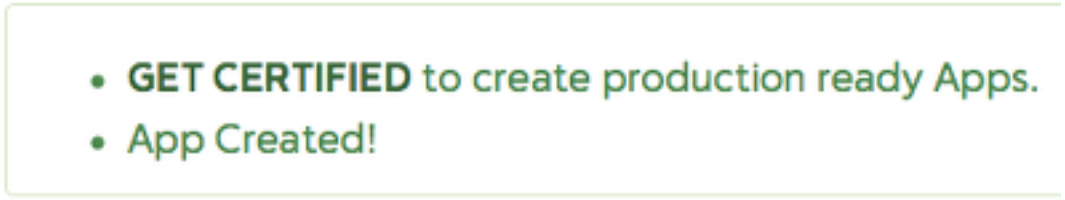
On the next screen you name your application.


CONSUME OUR API
GET CERTIFIED to create production ready Apps.
NAME YOUR APPLICATION : *
My First Payeezy App
Internal name: my-first-payeezy-app Edit
WHAT TYPE OF PRODUCT IS THIS ? : *
SANDBOX
CREATE YOUR APP

Spaces are allowed within the app name. For this example we are using `My First Payeezy App`. The checkbox “SANDBOX” must be selected at this time. This will allow you to test your app in a safe secure environment provided by Payeezy.

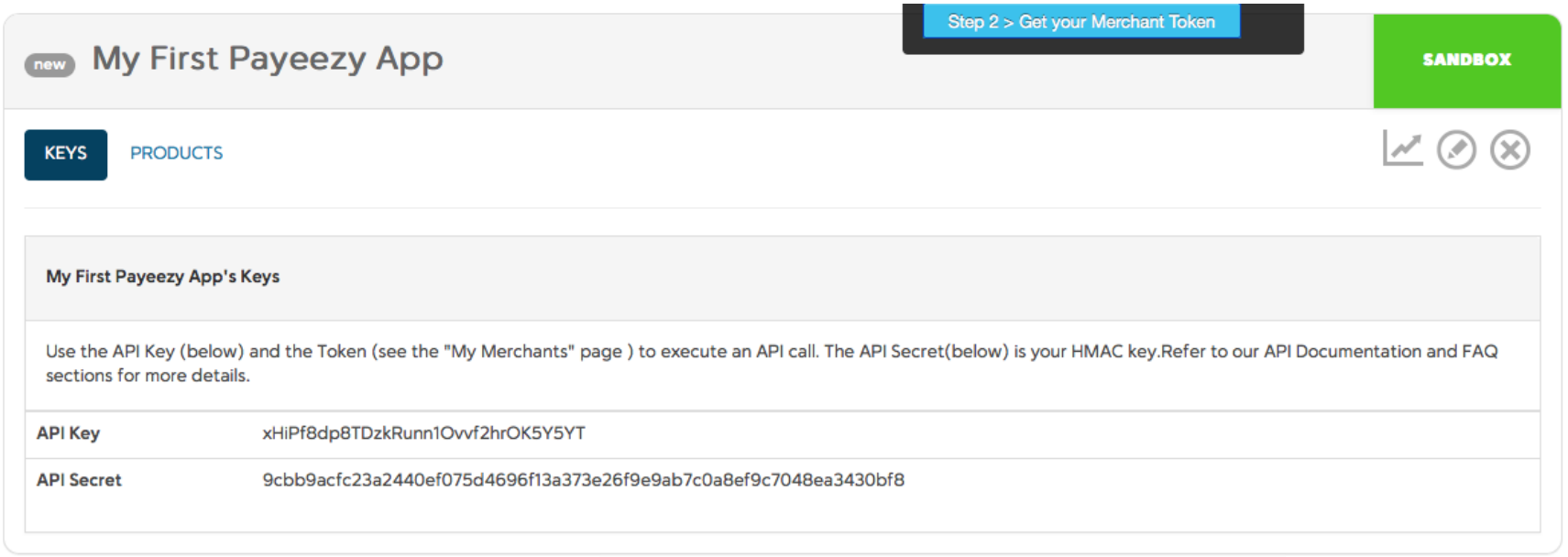
Click “CREATE YOUR APP” to continue.

You’re making progress!



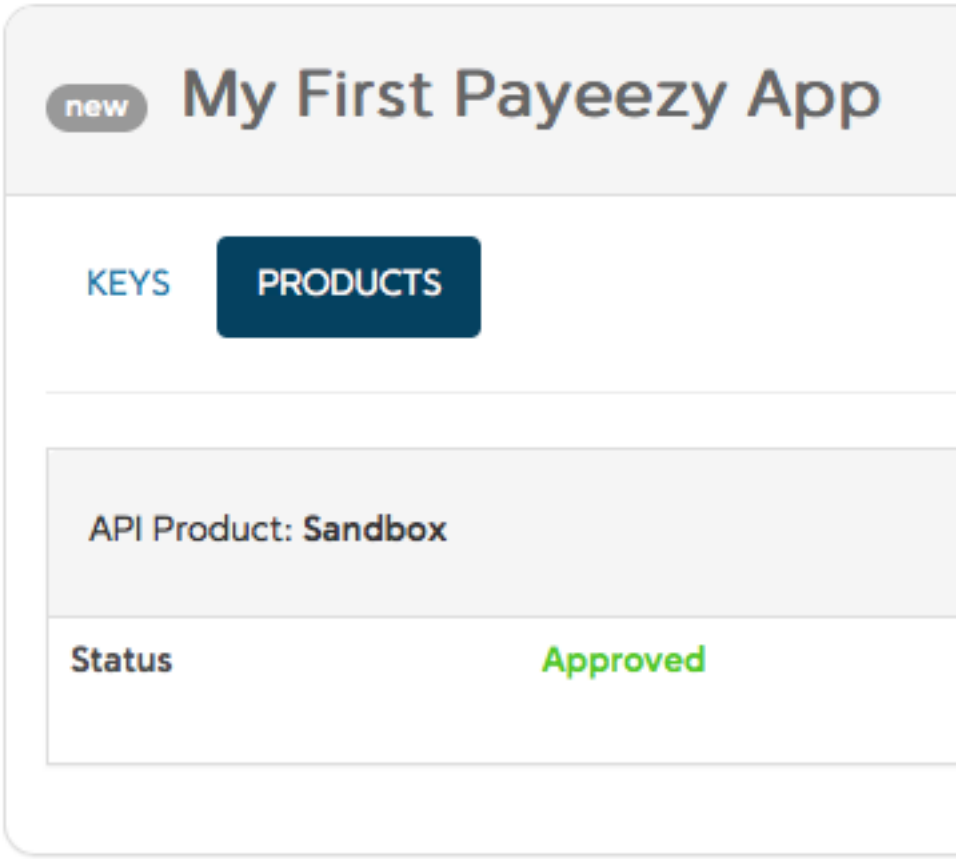
The page updates to show your newly created app entry and that it has a sandbox ready. We are done adding apps.

Let’s examine some key details about your new app that you will need as a developer. Click on your app name.



Details are revealed about your app. With the “KEYS” button selected (by default) you can see the `API Key` and `API Secret` that you will use later in your program.

If you tap on the “PRODUCTS” button you can confirm that your app is approved to be tested within the sandbox.

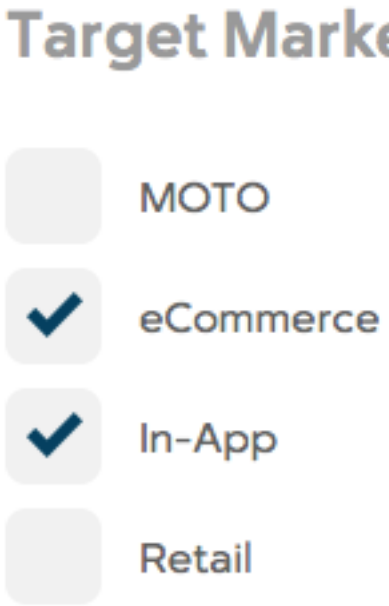


Next, you need a merchant token. To get the merchant token you need to complete the certification process as follows.

Click on the Get Certified button.

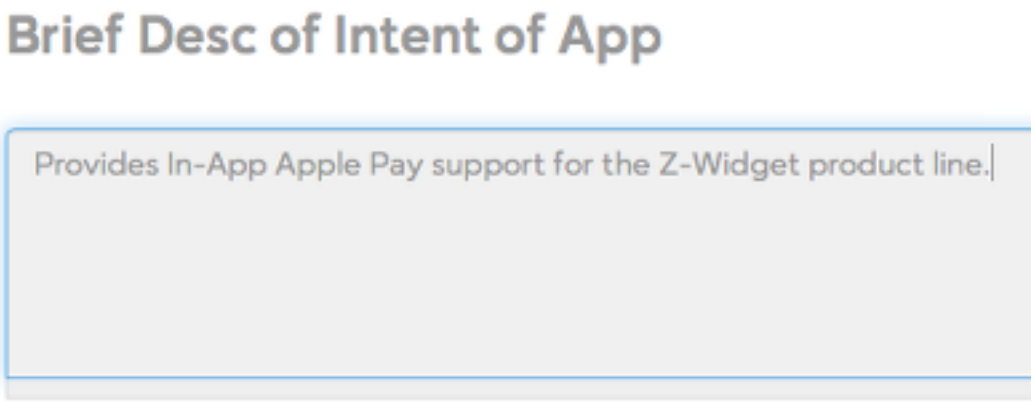
Fill in the fields on the next page that opens up. Some of the items will be pre-filled for you. The phone number you provide needs to be real but does not need to be one that can receive SMS messages.

When you get to the section `Target Market`



Be sure to check the boxes for `eCommerce` & `In-App`.

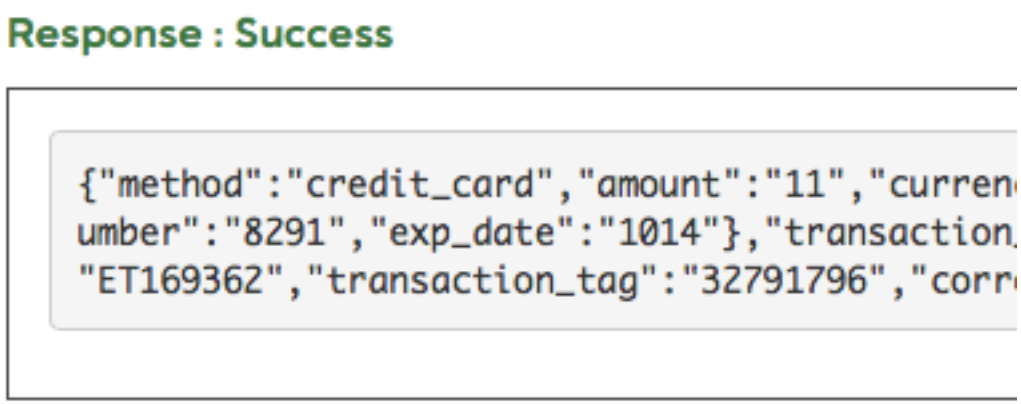
Add a brief description about your app.



The web page also advises you to ensure there is properly formatted content filled in for the Certification Test data to ensure that you are not a bot. The example cert will work fine for testing.

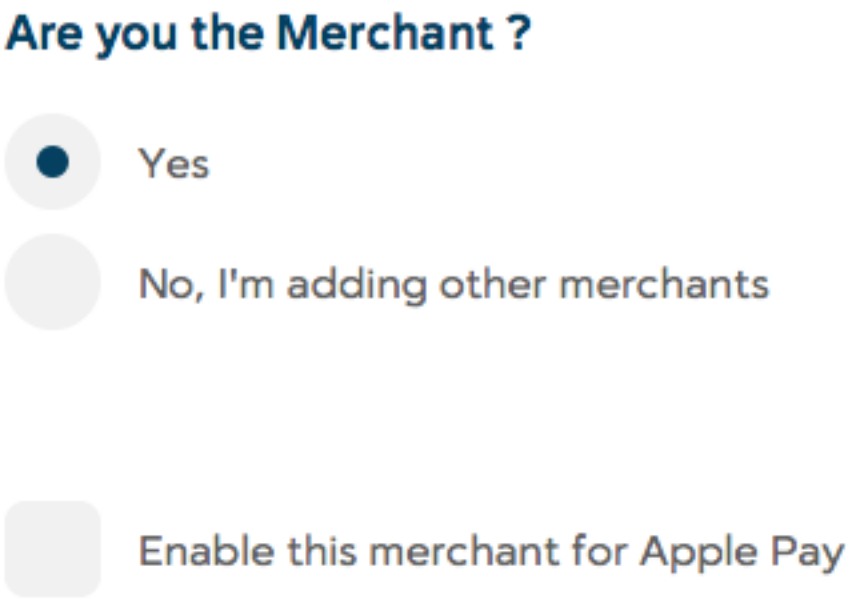
Click CERTIFY.

Within a few seconds you should see a JSON success response.



You can now select the “I Agree” checkbox for the terms and conditions and press SUBMIT.

You will now be taken to the ADD MERCHANT page.



There are 2 radio buttons to cover the 2 possible scenarios:

Select “Yes” if you are your own merchant and will be directly accepting payments. If you are a merchant in this scenario, only you will be looking at transactions. You will need to have valid tax information if you select this option.

Select “No, I’m adding other merchants” if, for example you are a developer working at a merchant or are a developer hired to do this work for a merchant.

The “Enable this merchant for Apple Pay” checkbox must be chosen for either scenario.

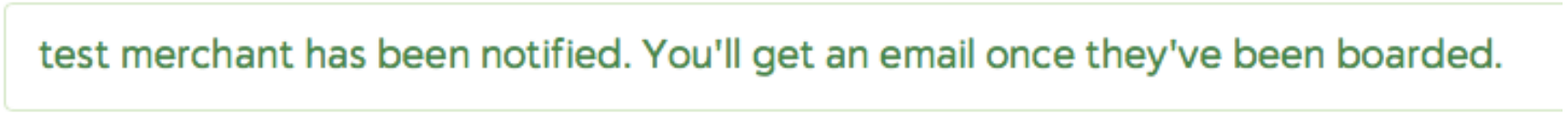
Press SUBMIT.

From here you are taken to the NOTIFY MERCHANT page. On this page you provide the information to notify the merchant of your boarding process.

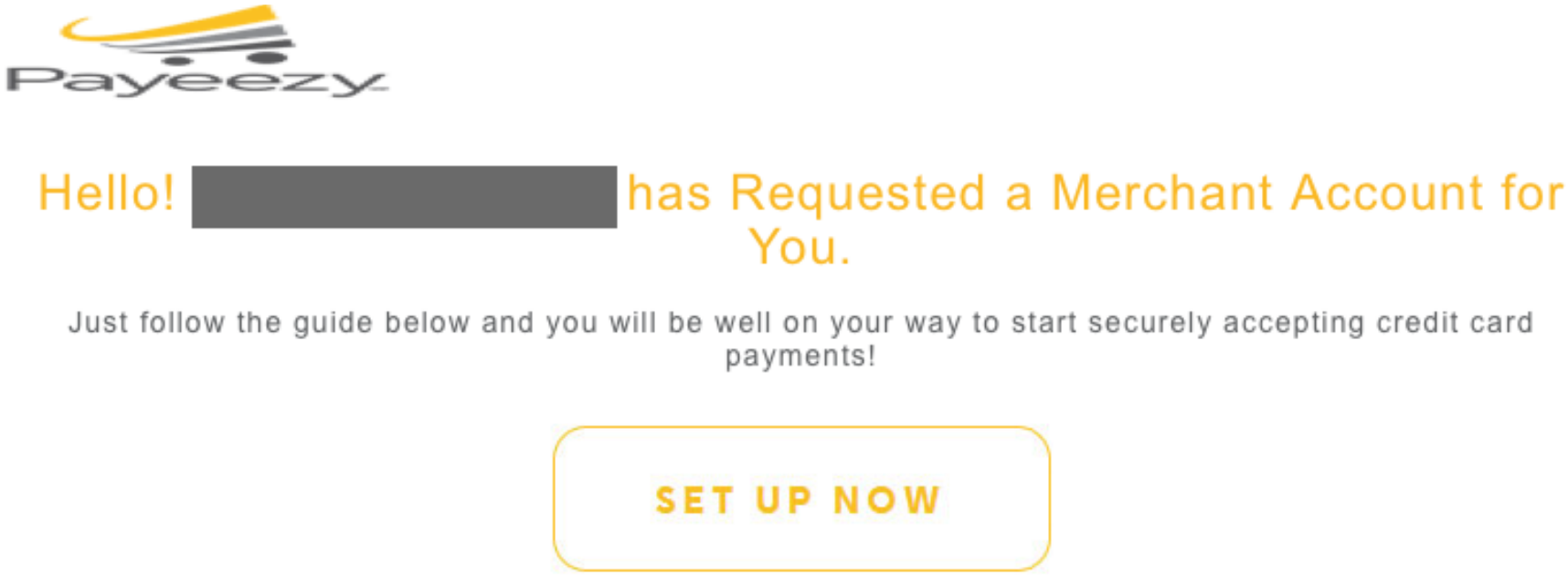
All fields are required.

After completing the form press the NOTIFY MERCHANT button.

A message notifying you that an email has been sent is presented.



The merchant will receive an email asking them to authorize the next step.

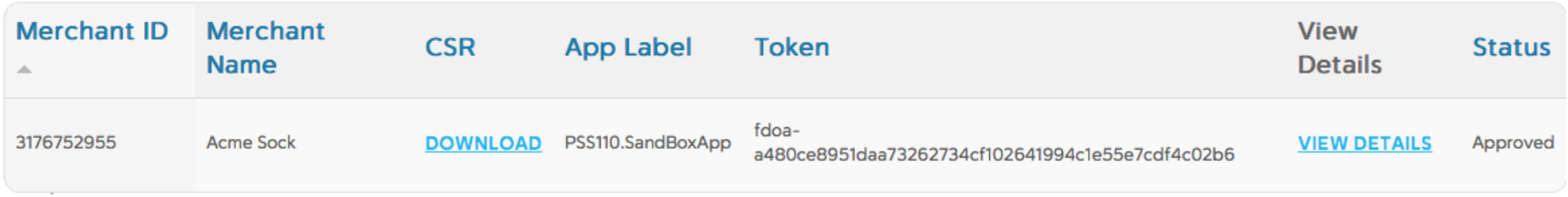


The merchant will be taken through steps where they can specify their tax and banking information as part of the authorization and boarding process.

You can select the SANDBOX button to view the application and token information we have setup for you automatically, enabling you to test your app before going live.

We've provisioned you with a merchant account in the Sandbox. You should provide this merchant token as the `token` header parameter in your API call. For integrating with Apple Pay™, your CSR is provided

as a download. Note that the CSR is only available for your download when the status shows you are approved.



Merchant ID	Merchant Name	CSR	App Label	Token	View Details	Status
3176752955	Acme Sock	DOWNLOAD	PSS110.SandBoxApp	fdoa-a480ce8951daa73262734cf102641994c1e55e7cdf4c02b6	VIEW DETAILS	Approved

The Acme Sock company is a test merchant we created for your use as soon as you are certified. The token is shown in the table.

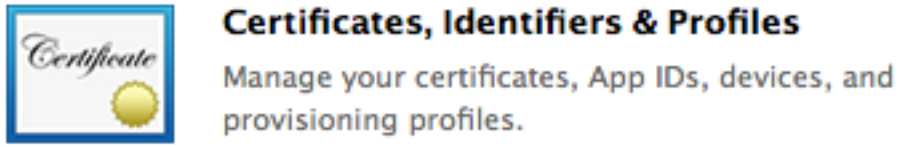
If you select the VIEW DETAILS link, you are taken to a page where you can provide an optional web-hook URL to be able to receive web push notifications.

The downloadable CSR is only available for your Apple developer account.

Click the LIVE button to continue.

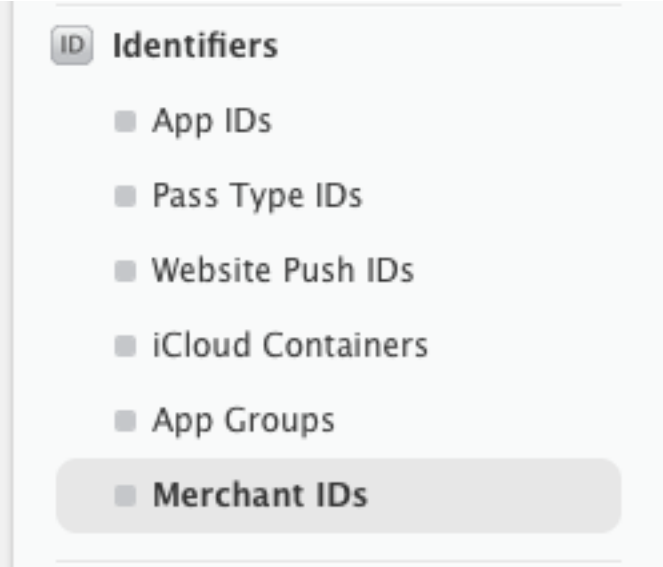
You now need to logon to the Apple developer portal through your account.

Navigate to the “Certificates, Identifiers & Profiles” link on Apple’s developer portal.

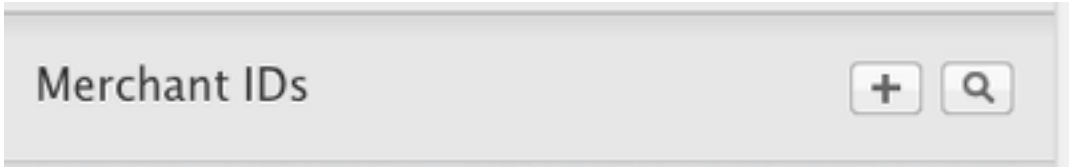


Continue to the Identifiers section.

Select “Merchant IDs” next.



You want to add a new Merchant ID, so select the “+” button.



The description you provide here can be anything you want.

The identifier you specify should follow the same reverse name domain scheme used for other app ids. Again, it does not need to agree with your bundle ID, however it needs to be unique.
Click Continue.

Merchant ID Description

Description: My First Payeezy Merchant

You cannot use special characters such as @, &, *, ', "

Identifier

Enter a unique identifier for your Merchant ID, starting with the string 'merchant'.

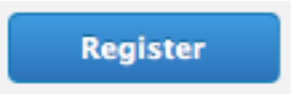
ID: merchant.com.firstpayeezy

We recommend using a reverse-domain name style string (i.e., merchant.com.example.merchantname).

Continue

Apple's portal presents you with a confirmation page…

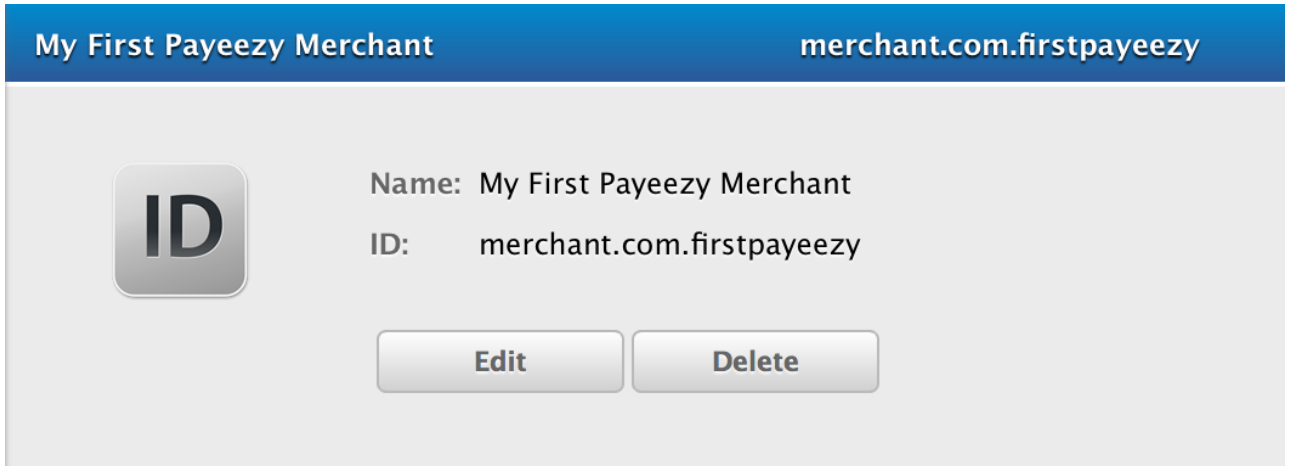
where you can verify your information and then click Register.



Apple processes your registration and presents you with confirmation that your registration of merchant ID is complete.

Click Done.

The newly registered Merchant ID is shown in a list with any other Merchant IDs you have defined. You can select your new Merchant ID from the list and see details, including an Edit and Delete button.



Click the Edit button.

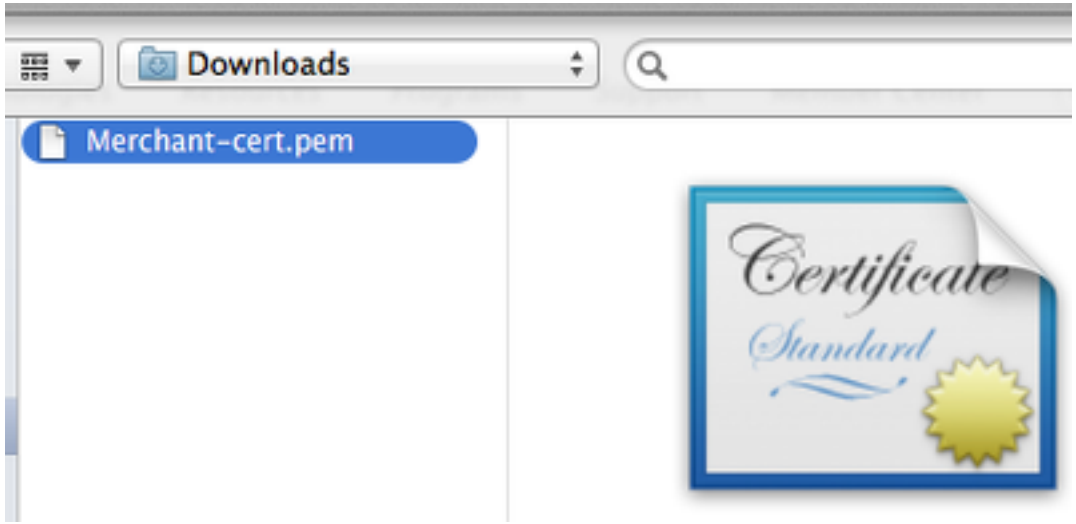
On the next screen, Apple shows you your registered Merchant ID details and provides the following information.

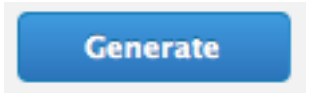
To configure Apple Pay for this Merchant ID, a Certificate that is used by Apple to encrypt transaction data, is required. Each Merchant ID requires its own Certificate. Manage and generate your certificates below.

Click on Create Certificate. 

The Apple portal shows you instructions for your creating your certificate. The Payeezy portal through steps you did earlier make this next part easy for you. You already have the Merchant-cert.pem file from earlier.

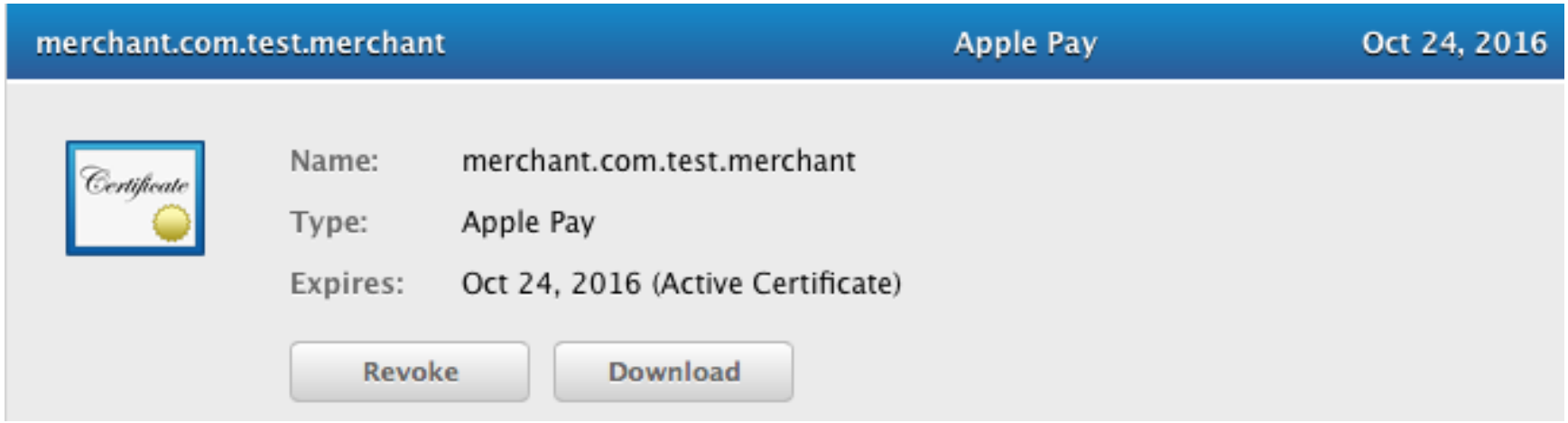
Navigate to your cert request file's location and select it.



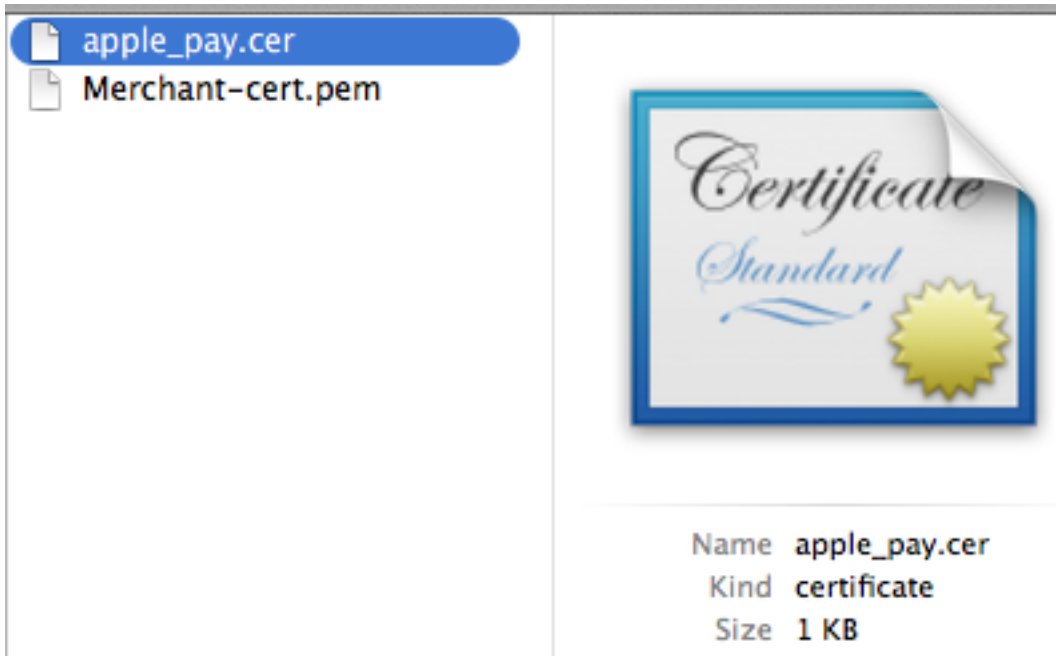
The Generate button 

is now active. Click on it to generate your certificate.

The remaining step on the Apple Developer Portal is to download your Apple Pay™ certificate.



You can download the new certificate for use in your development project. At this point you can log off the Apple portal.



DOWNLOAD THE PAYEEZY APPLE PAY™ SDK

1-Click to Harness the Power of Apple Pay™ Powered by Payeezy

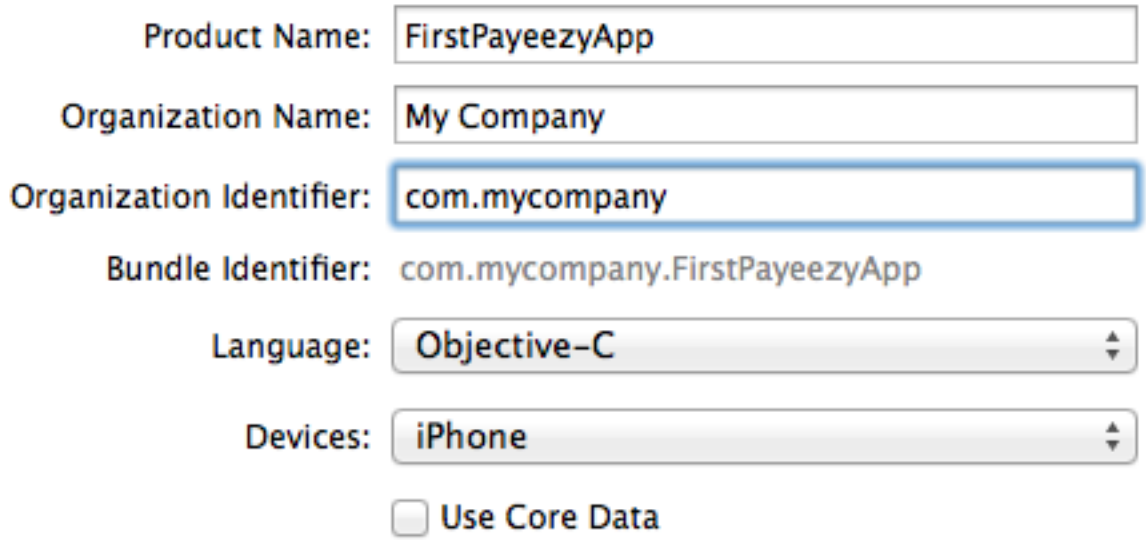
Download our Apple Pay SDK which contains all the bits and bytes you need to power your app with Apple Pay™ by Payeezy. It is in the orange box labeled “APPLE PAY™ SDK STARTER KIT”. Decompress the download zip file and store the included frameworks in a folder that you can reference from your app.

INTEGRATE THE SDK INTO YOUR APP

Let's Get Coding

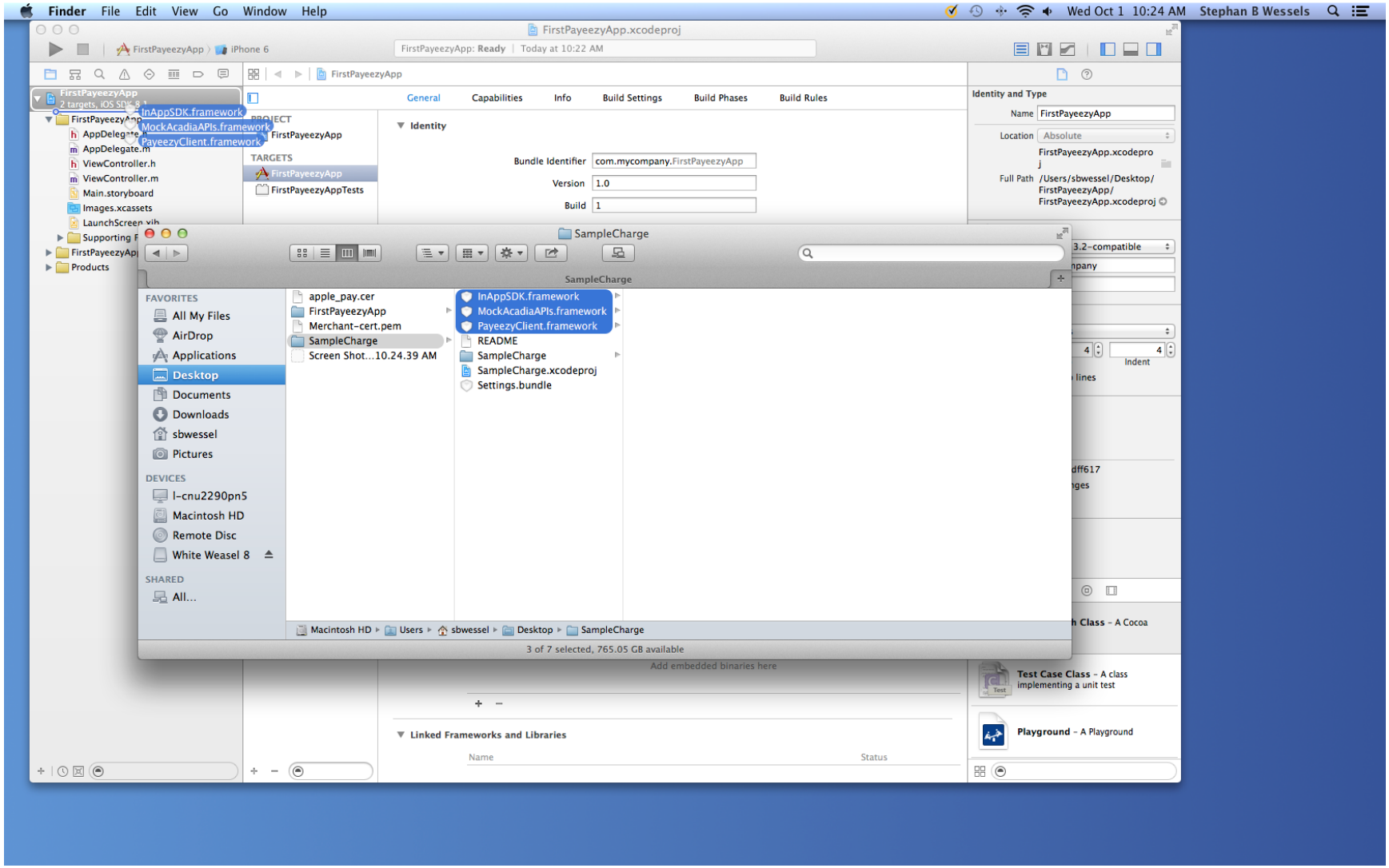
First make sure that you've installed Xcode version 6.0 or greater so you can access the new Apple Pay™ API's. If you're not sure which version you have just grab the latest version available on the Apple developer site or directly from the App Store.

Create a new project in Xcode. Use Single View Application as our template. For this example, here is how we filled out the project options sheet…



Add Our Frameworks To Your Project

With your project opened in Xcode and a Finder window opened to the folder containing the frameworks you downloaded in Step 2, drag and drop the frameworks onto your project.


Finder File Edit View Go Window Help
Wed Oct 1 10:24 AM Stephan B Wessels
FirstPayeezyApp.xcodeproj
InAppSDK.framework
MockAcadiaAPIs.framework
PayeezyClient.framework
SampleCharge
FAVORITES
All My Files
AirDrop
Applications
Desktop
Documents
Downloads
sbwessel
Pictures
DEVICES
Macintosh HD
Remote Disc
White Weasel 8
SHARED
apple_pay.cer
FirstPayeezyApp
Merchant-cert.pem
SampleCharge
README
SampleCharge.xcodeproj
Settings.bundle
3 of 7 selected, 765.05 GB available
Identity
Bundle Identifier
Version
Build
Linked Frameworks and Libraries

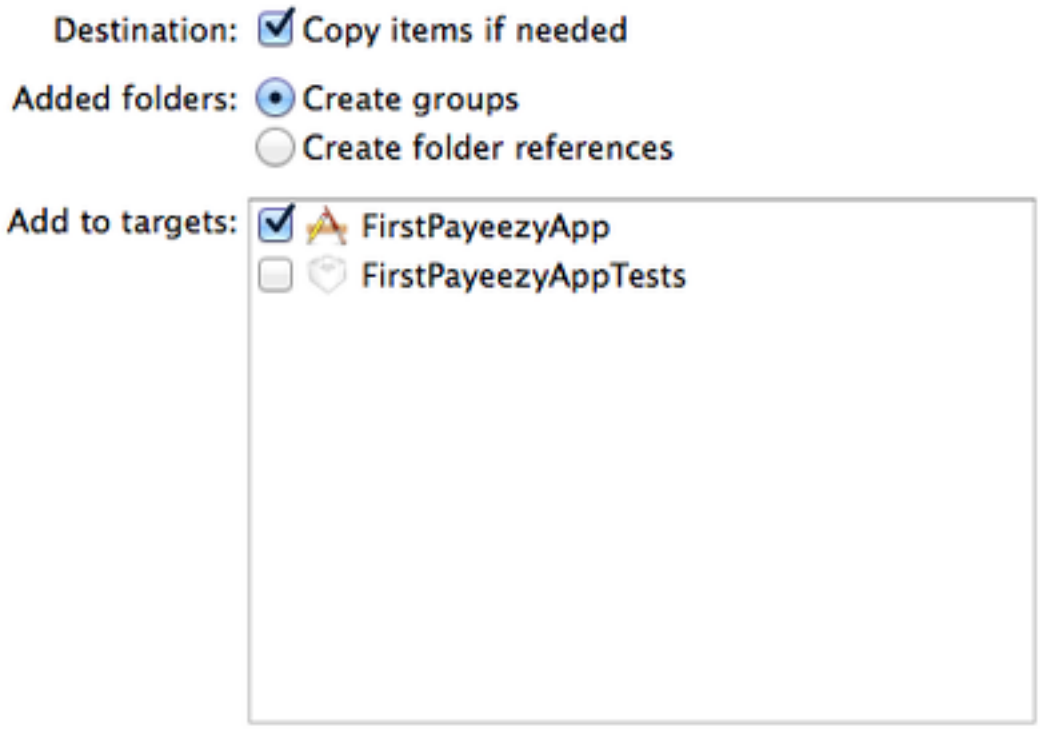
There are 3 frameworks included here:

1. `InAppSDK.framework` This is the main framework that supports ApplePay transactions.

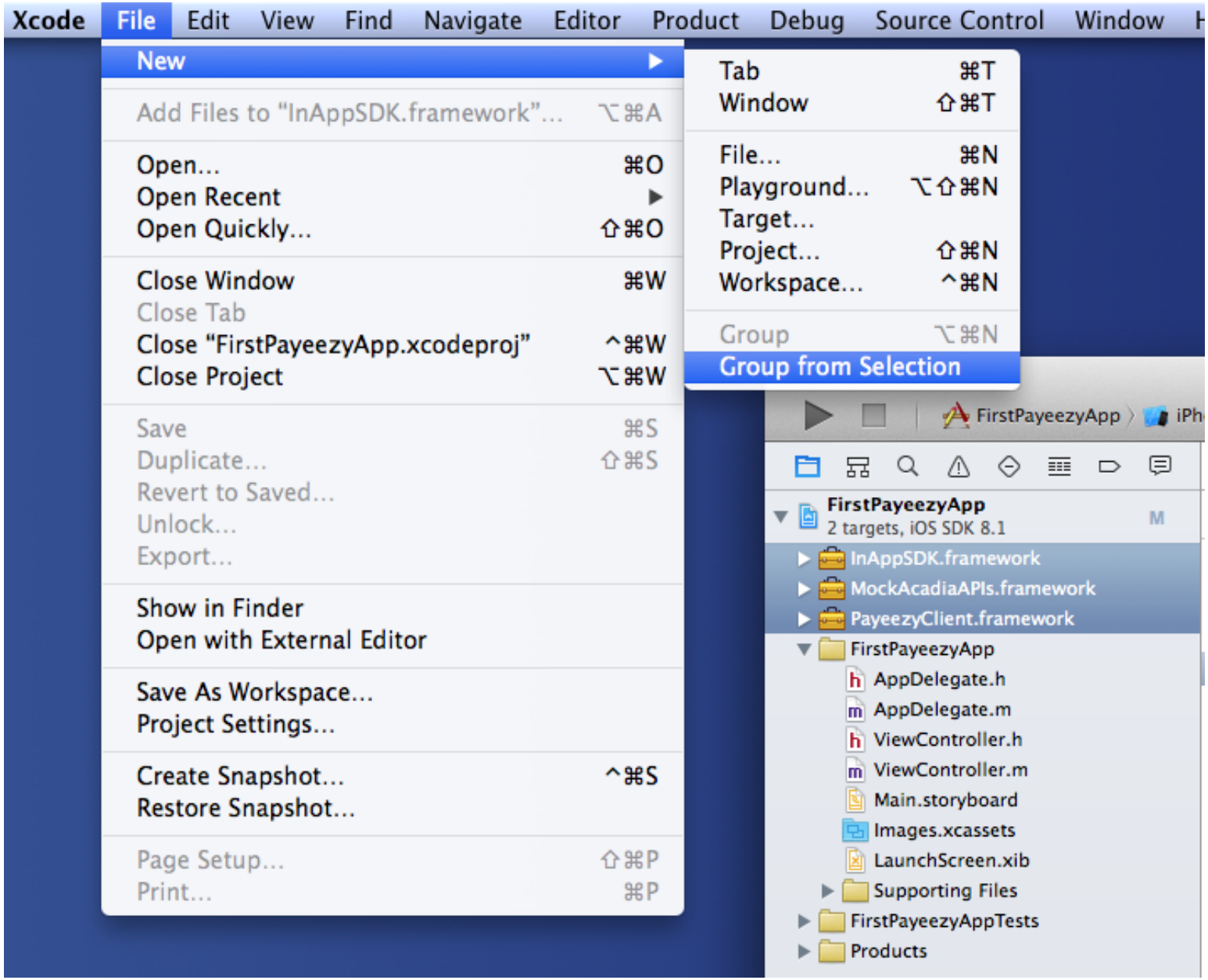
2. `PayeezyClient.framework` This is the iOS client for the Payeezy (RESTful) API backend services. The InAppSDK framework relies on this client for making secure service calls to the services. Visit http://developer.payeezy.com for documentation and an interactive sandbox for experimenting with the Payeezy API's.

3. `MockAcadiaAPIs.framework` This framework contains mock versions of the ApplePay API's. Its purpose is to allow developers to start integrating In-App payments without having access to the ApplePay API's and/or an iPhone 6. When we are ready to deploy on a real Apple Pay™ enabled iPhone we will no longer include this framework.

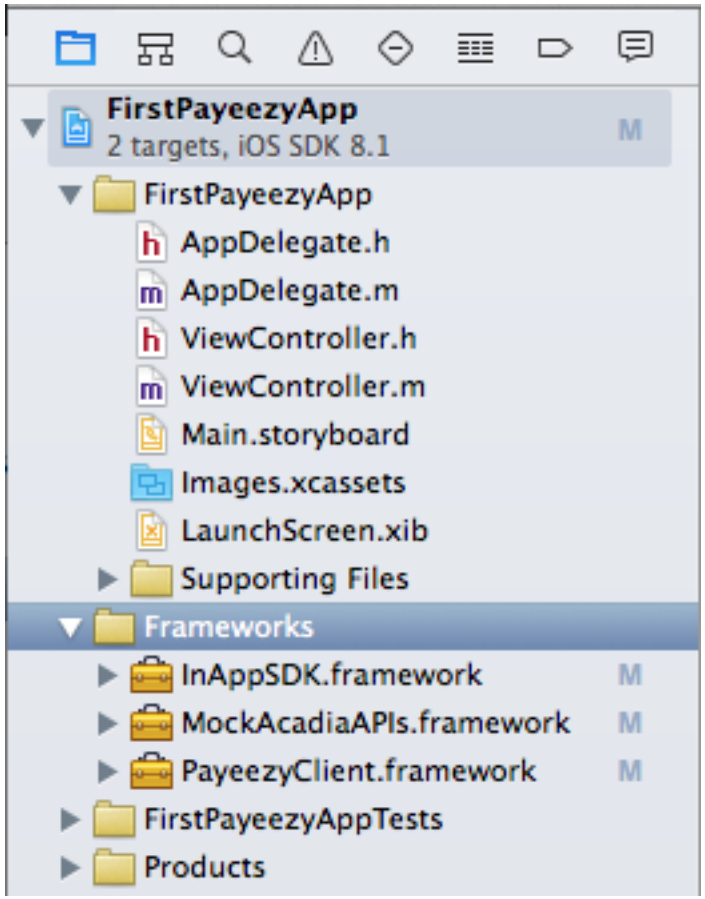
When prompted be sure to select the "Copy items if needed" option.



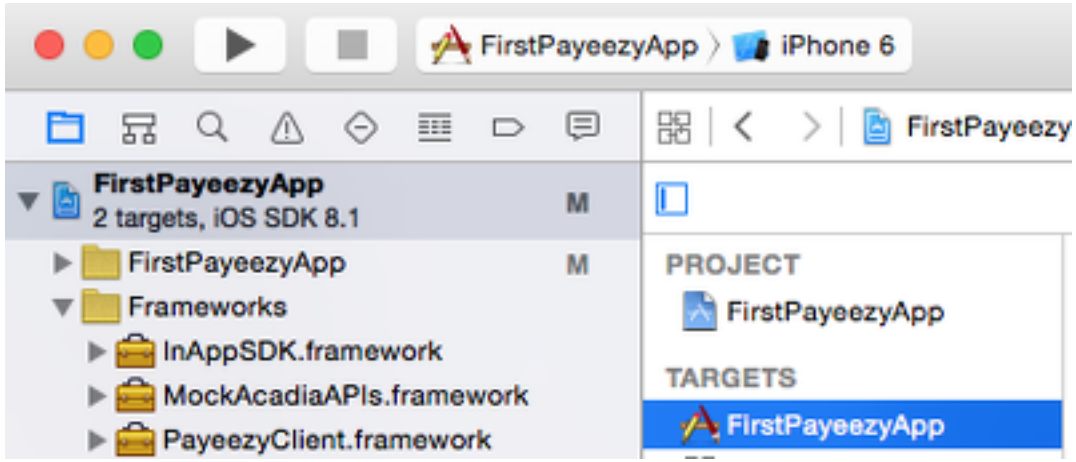
If you prefer to use a frameworks group when working with Xcode projects, it is a simple matter to select the frameworks and create a new Group.


Xcode
File
Edit
View
Find
Navigate
Editor
Product
Debug
Source Control
Window
New
Add Files to "InAppSDK.framework"... ⌥⌘A
Open... ⌘O
Open Recent
Open Quickly... ⇧⌘O
Close Window ⌘W
Close Tab
Close "FirstPayeezyApp.xcodeproj" ^⌘W
Close Project ⌥⌘W
Save ⌘S
Duplicate... ⇧⌘S
Revert to Saved...
Unlock...
Export...
Show in Finder
Open with External Editor
Save As Workspace...
Project Settings...
Create Snapshot... ^⌘S
Restore Snapshot...
Page Setup... ⇧⌘P
Print... ⌘P
Tab ⌘T
Window ⇧⌘T
File... ⌘N
Playground... ⌥⇧⌘N
Target...
Project... ⇧⌘N
Workspace... ^⌘N
Group ⌥⌘N
Group from Selection
FirstPayeezyApp
FirstPayeezyApp
2 targets, iOS SDK 8.1
InAppSDK.framework
MockAcadiaAPIs.framework
PayeezyClient.framework
FirstPayeezyApp
AppDelegate.h
AppDelegate.m
ViewController.h
ViewController.m
Main.storyboard
Images.xcassets
LaunchScreen.xib
Supporting Files
FirstPayeezyAppTests
Products

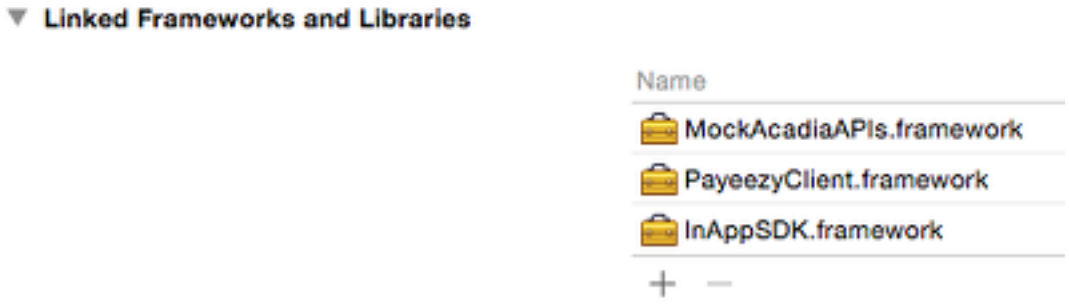
Name the new group `Frameworks` and drag it to its new location within the Project window.



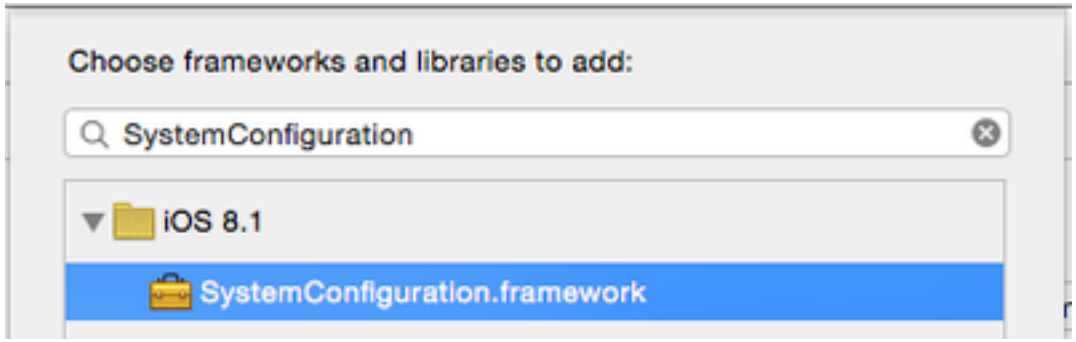
We will also need to add the `SystemConfiguration.framework` from Apple's available frameworks. The simplest way to do this is to first select the target.



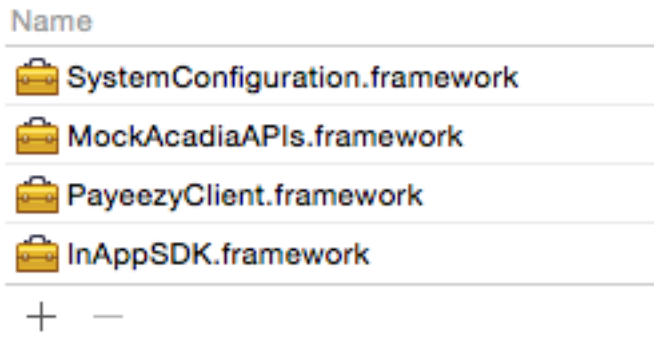
Under the General tab find the `Linked Frameworks and Libraries` section.



When you click on the “+” button Xcode will prompt you with a list of frameworks to choose. You can narrow down your search by typing in SystemConfiguration as a filter.



Click the “Add” button and Xcode will include the new framework.



About the Example Application

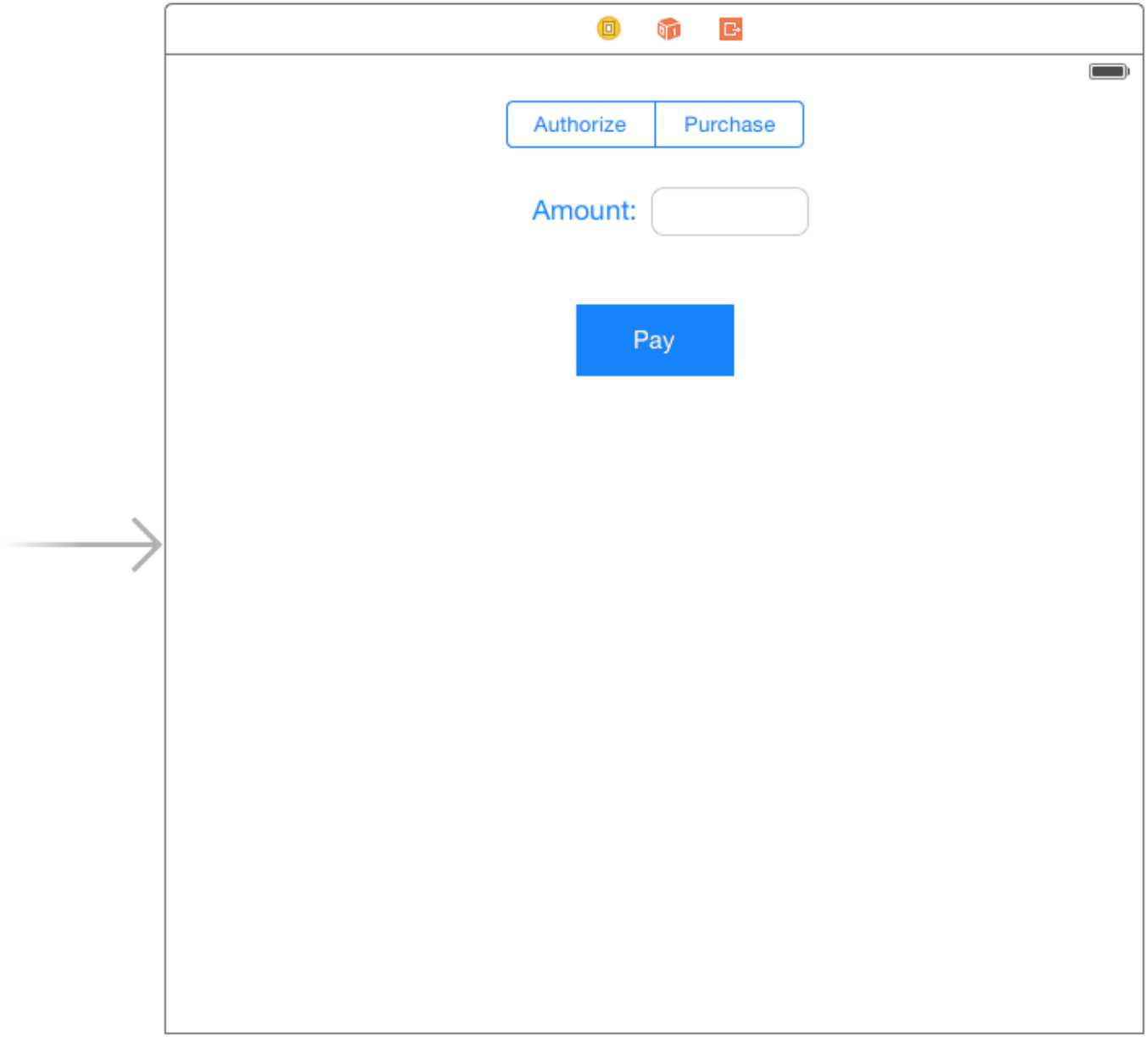
We will create an Application that demonstrates how easy it is to enable secure In-App payments via Apple's Apple Pay™ technology. Apple collaborated with First Data to be one of the first payment platforms that is certified to support this revolutionary payment method.

The application itself contains a single view controller that allows the user to specify a transaction amount and select whether to perform a pre-authorization for that amount or an instantaneous purchase. That transaction is routed through the First Data In-App payments SDK to perform an ApplePay transaction.

We will create the Storyboard contents next, then discuss and add the model objects you will need. Finally we will wire up the models and messages to your Storyboard.

Create the Storyboard for Our Example

The Storyboard we want to create will look like this.

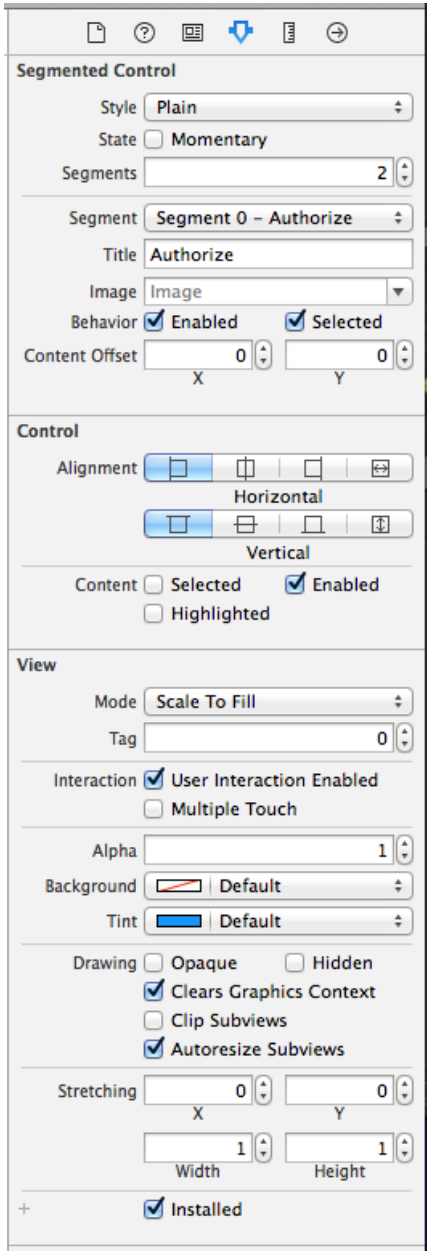


There are 4 GUI elements to add to the Storyboard. We'll also use the default "Any/Any" size so we can design an app that will work well on the support iPhone 6 & 6 Plus (as well as any potential other future supported devices).

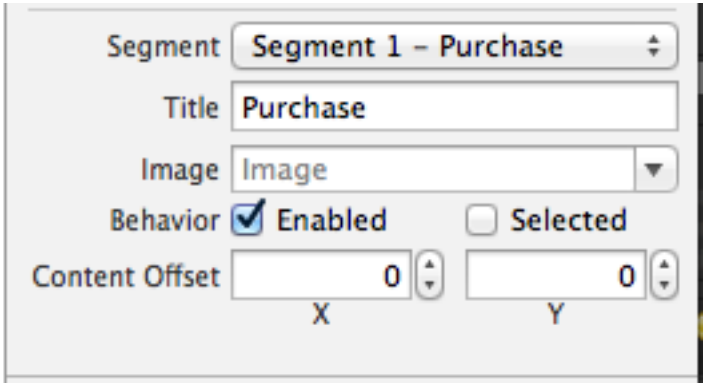
Authorize / Payment Control

Drag a Segmented Control onto the Storyboard and locate it near the top of the view and roughly centered. First we'll set the properties we need and the define the location constraints.

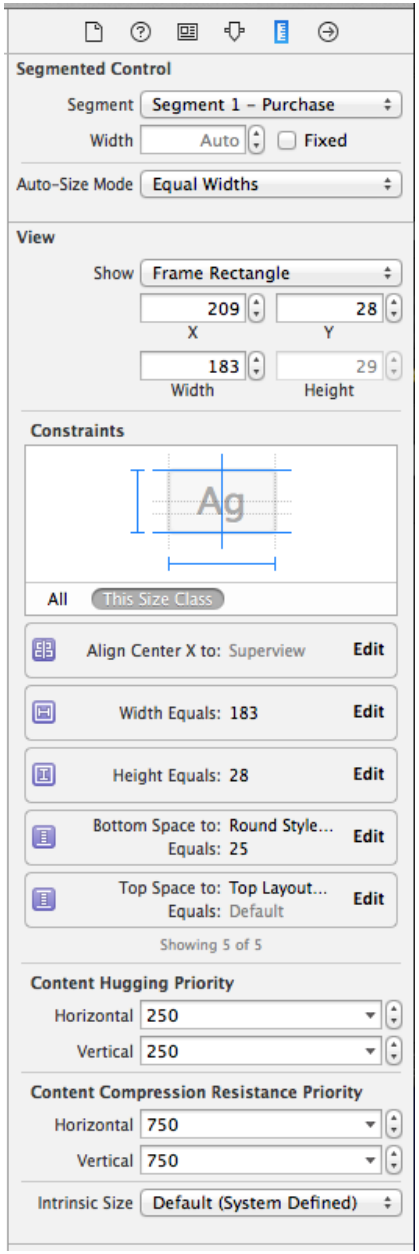
Set the Properties view for the control as follows.



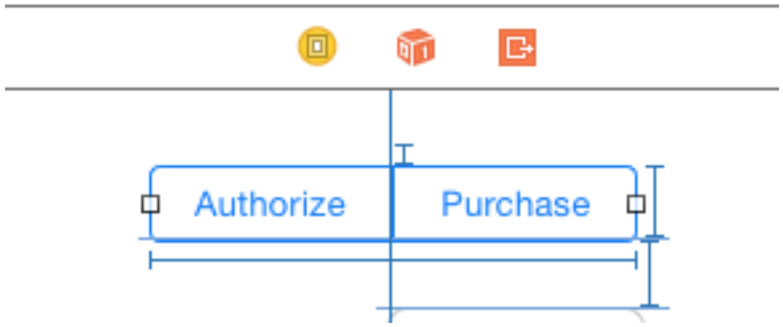
The details for segment–0, “Authorize”, are as shown above. Select the settings for segment–1, “Purchase”, as follows.



Next we must set the constraints for our segment control. The concept we are aiming for is to have the segment control to be centered horizontally inside the view and constrained against the top edge of the view. This will make it work regardless of orientation of size of iPhone. The constraints look as follows.


Segmented Control
Segment
Segment 1 – Purchase
Width
Auto
Fixed
Auto-Size Mode
Equal Widths
View
Show
Frame Rectangle
209
X
28
Y
183
Width
29
Height
Constraints
Ag
All
This Size Class
Align Center X to: Superview
Edit
Width Equals: 183
Edit
Height Equals: 28
Edit
Bottom Space to: Round Style...
Equals: 25
Edit
Top Space to: Top Layout...
Equals: Default
Edit
Showing 5 of 5
Content Hugging Priority
Horizontal
250
Vertical
250
Content Compression Resistance Priority
Horizontal
750
Vertical
750
Intrinsic Size
Default (System Defined)

Note that we have fixed the width and height of the control and aligned it to the center of the super-view. The Top Layout constraint defines how close we get to the top of the view. The Bottom Space constraint has no meaning until we add an object below it. If you have only defined the segmented control so far you will not see that constraint.

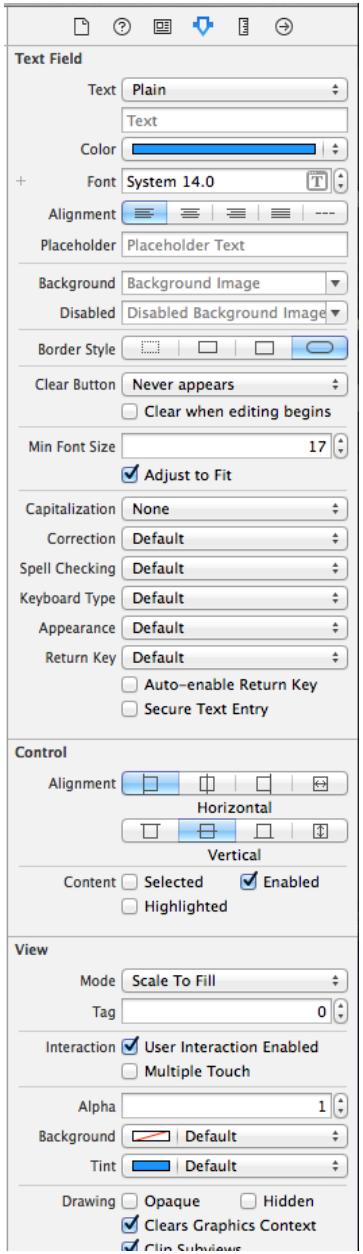


We will go back and deal with state and events, if needed, for this control later.

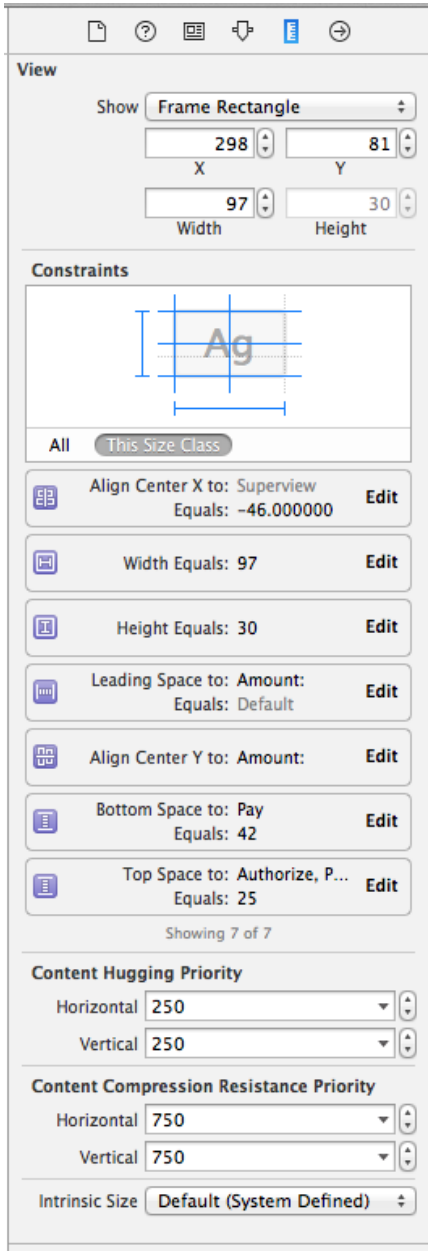
Amount Label and Input Text Field

Our next two visual objects will be the input text field and a companion label. We'll add the input field first.

Drag a Text Field object onto the view and locate it below the segmented control and slightly right-of-center in the view. We'll get the properties right for it first then fix the constraints. Here is the properties view.

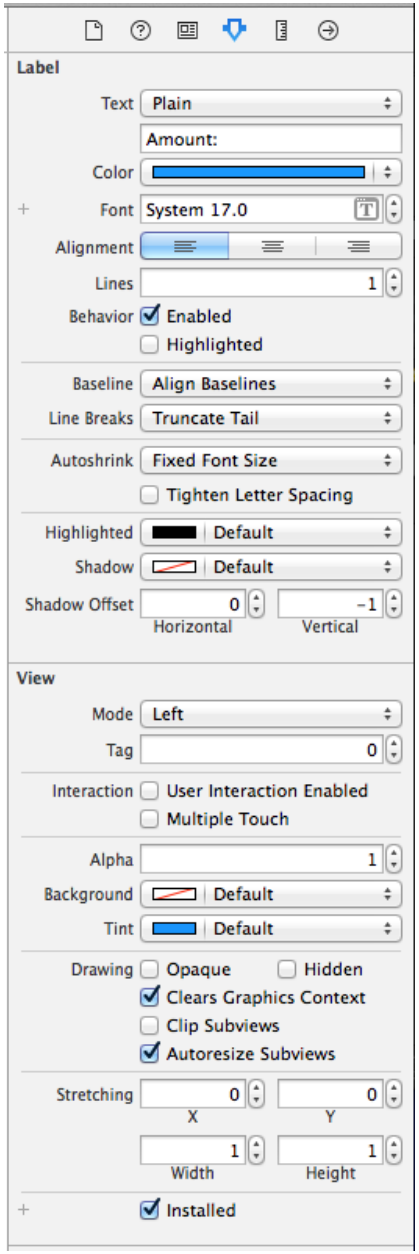

Text Field
Text
Plain
Text
Color
Font
System 14.0
Alignment
Placeholder
Placeholder Text
Background
Background Image
Disabled
Disabled Background Image
Border Style
Clear Button
Never appears
Clear when editing begins
Min Font Size
17
Adjust to Fit
Capitalization
None
Correction
Default
Spell Checking
Default
Keyboard Type
Default
Appearance
Default
Return Key
Default
Auto-enable Return Key
Secure Text Entry
Control
Alignment
Horizontal
Vertical
Content
Selected
Enabled
Highlighted
View
Mode
Scale To Fill
Tag
0
Interaction
User Interaction Enabled
Multiple Touch
Alpha
1
Background
Default
Tint
Default
Drawing
Opaque
Hidden
Clears Graphics Context

When defining the constraints we seek to align the object just below the segment and will use the horizontal center of the view as a reference and specify an X-offset. Here is the constraints & sizing information.

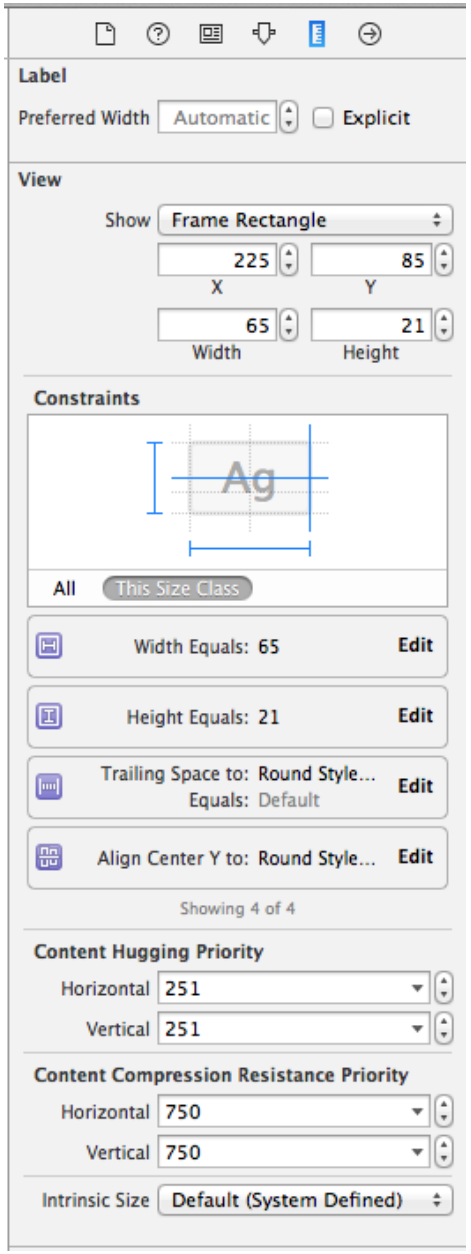


The constraints for Leading Space to Amount and Align Center Y to Amount as well as the Bottom Space constraints are not relevant until we add the label object.

Drag a label to the left of the input text. The properties are as follows.

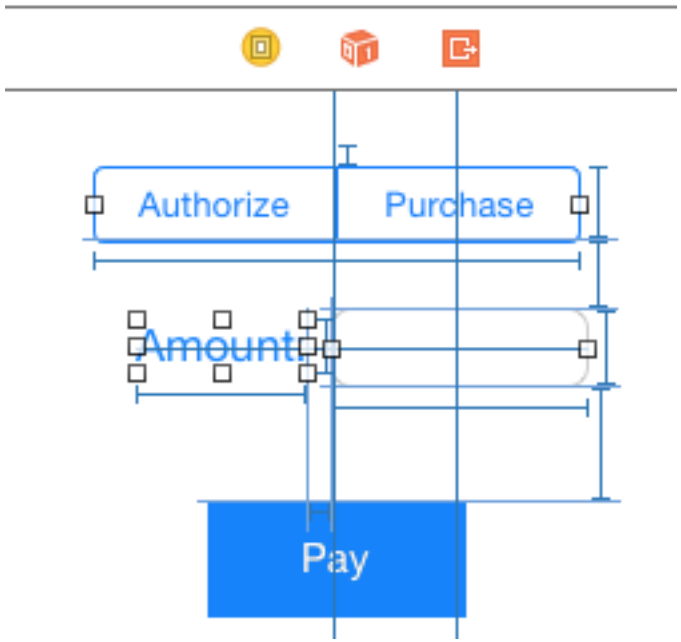


For the constrains we will locate this object relative to the input text field. Here are the sizing and constraint values.



Note that we have specified the size of the widget as well as the space to the right between it and the input field. Select both the input field and label and you can specify the Align Center Y value to neatly align them together.

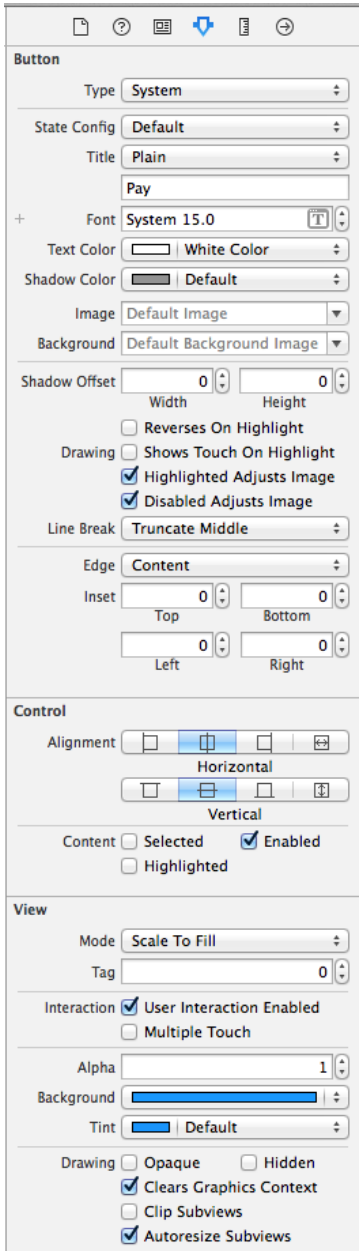
The visual elements you have located on your view should show constraints similar to this.



The only exception is that the view shown already has the “Pay” button added. That’s next.

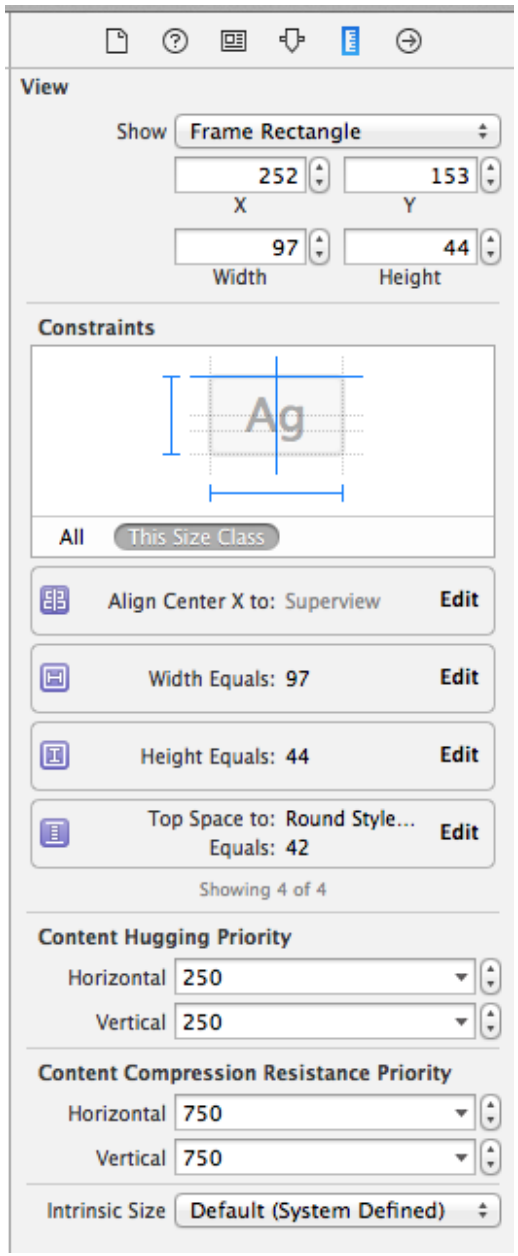
Pay Button

Our last visual component is the “Pay” button. Drag a button onto the view and locate it below the input field and label and center it in the view. Here is the properties pane for the new button.

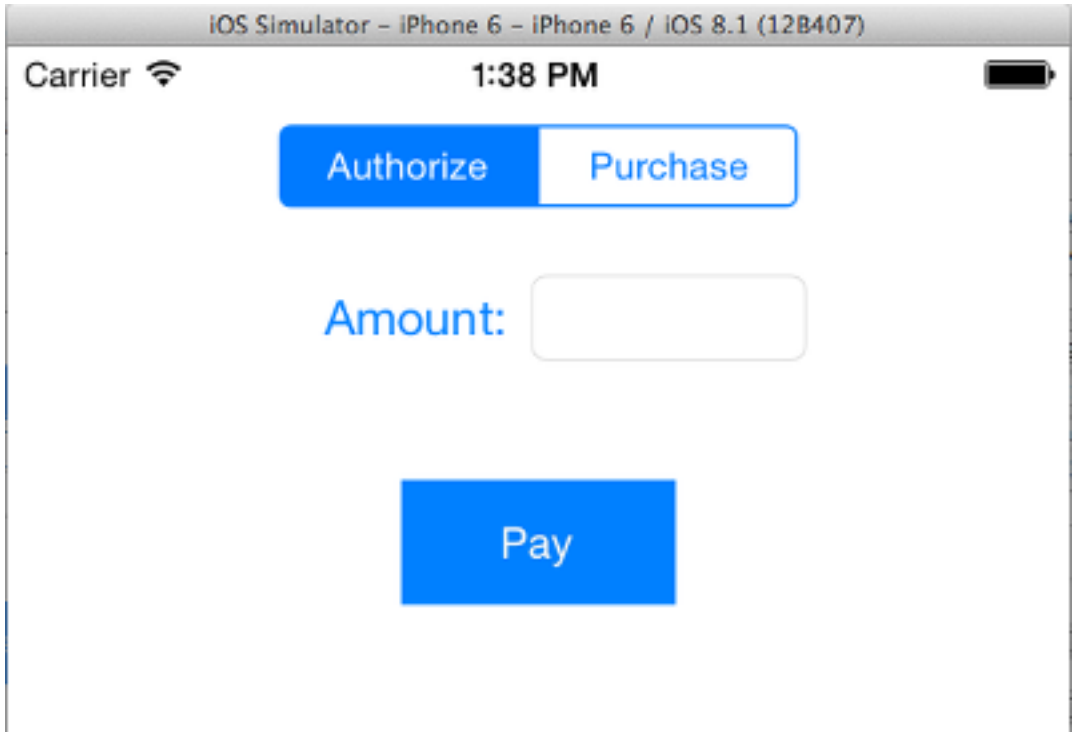

Button
Type System
State Config Default
Title Plain
Pay
Font System 15.0
Text Color White Color
Shadow Color Default
Image Default Image
Background Default Background Image
Shadow Offset 0 0
Width Height
Reverses On Highlight
Drawing Shows Touch On Highlight
Highlighted Adjusts Image
Disabled Adjusts Image
Line Break Truncate Middle
Edge Content
Inset 0 0
Top Bottom
0 0
Left Right
Control
Alignment
Horizontal
Vertical
Content Selected Enabled
Highlighted
View
Mode Scale To Fill
Tag 0
Interaction User Interaction Enabled
Multiple Touch
Alpha 1
Background
Tint Default
Drawing Opaque Hidden
Clears Graphics Context
Clip Subviews
Autoresize Subviews

Note that we set the button text to white and the background color to a shade of blue.

The constraints for our “Pay” button are very simple.


View
Show
Frame Rectangle
252
X
153
Y
97
Width
44
Height
Constraints
Ag
All
This Size Class
Align Center X to: Superview
Edit
Width Equals: 97
Edit
Height Equals: 44
Edit
Top Space to: Round Style...
Equals: 42
Edit
Showing 4 of 4
Content Hugging Priority
Horizontal
250
Vertical
250
Content Compression Resistance Priority
Horizontal
750
Vertical
750
Intrinsic Size
Default (System Defined)

To test out our layout let's run the project using the iPhone 6 as the target in the simulator. You should see the visual objects remain nicely aligned in portrait or landscape orientations.

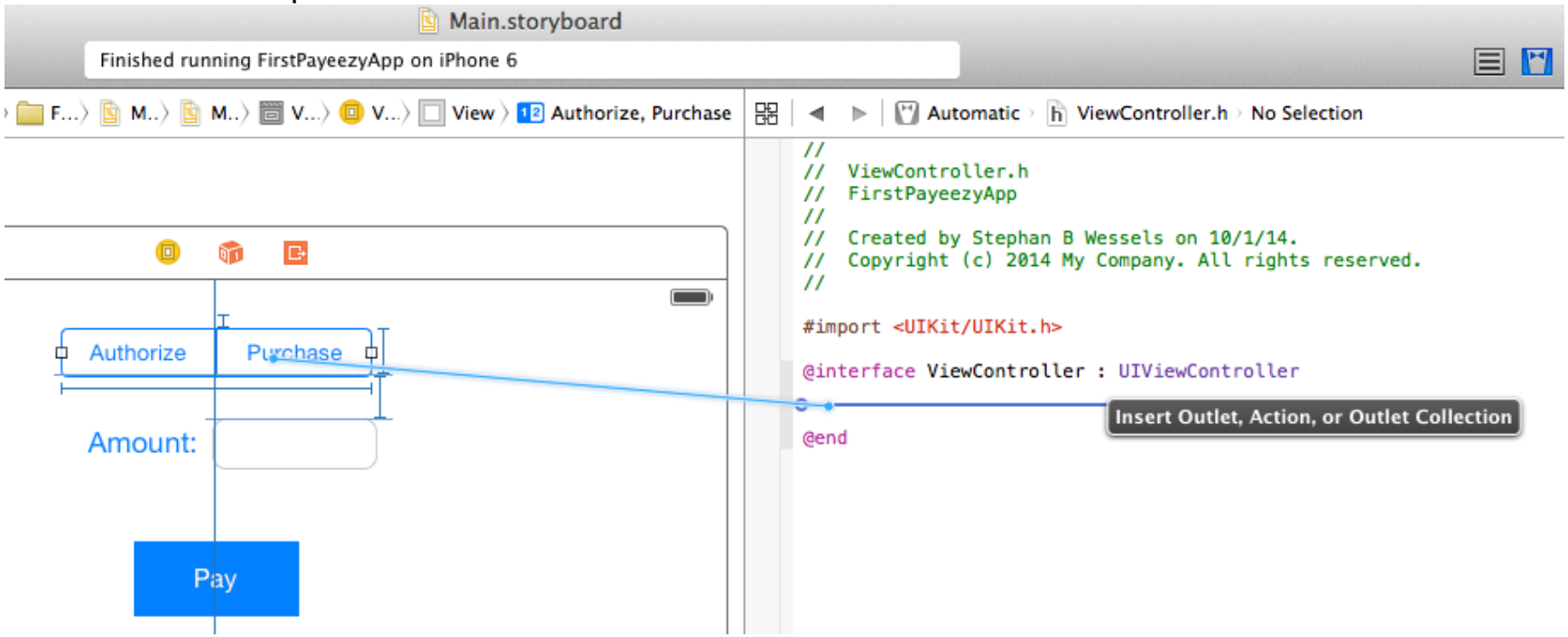


Create Outlets and an Action for Our View Controller

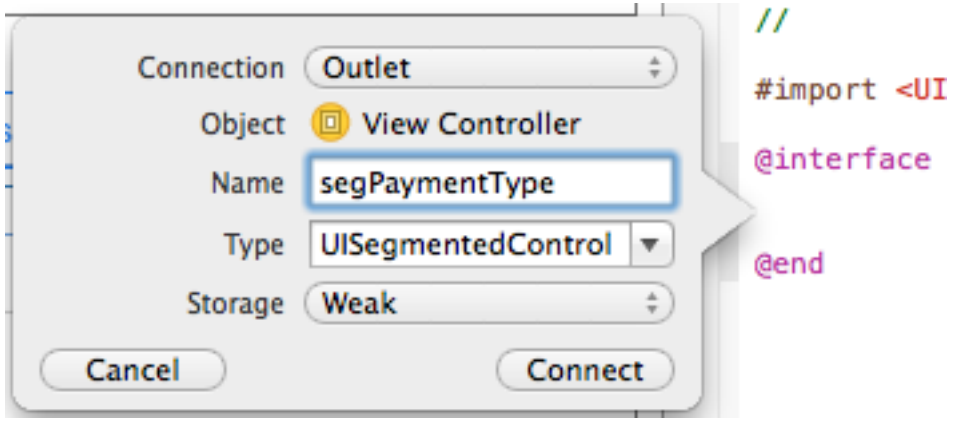
It's time to hook a few things up. With the Storyboard still selected you can use Xcode's Assistant Editor and pair it with the ViewController.h file.

We will need to make outlets for the segment control and the test input field. We will also need to create an action related to pressing the "Pay" button.

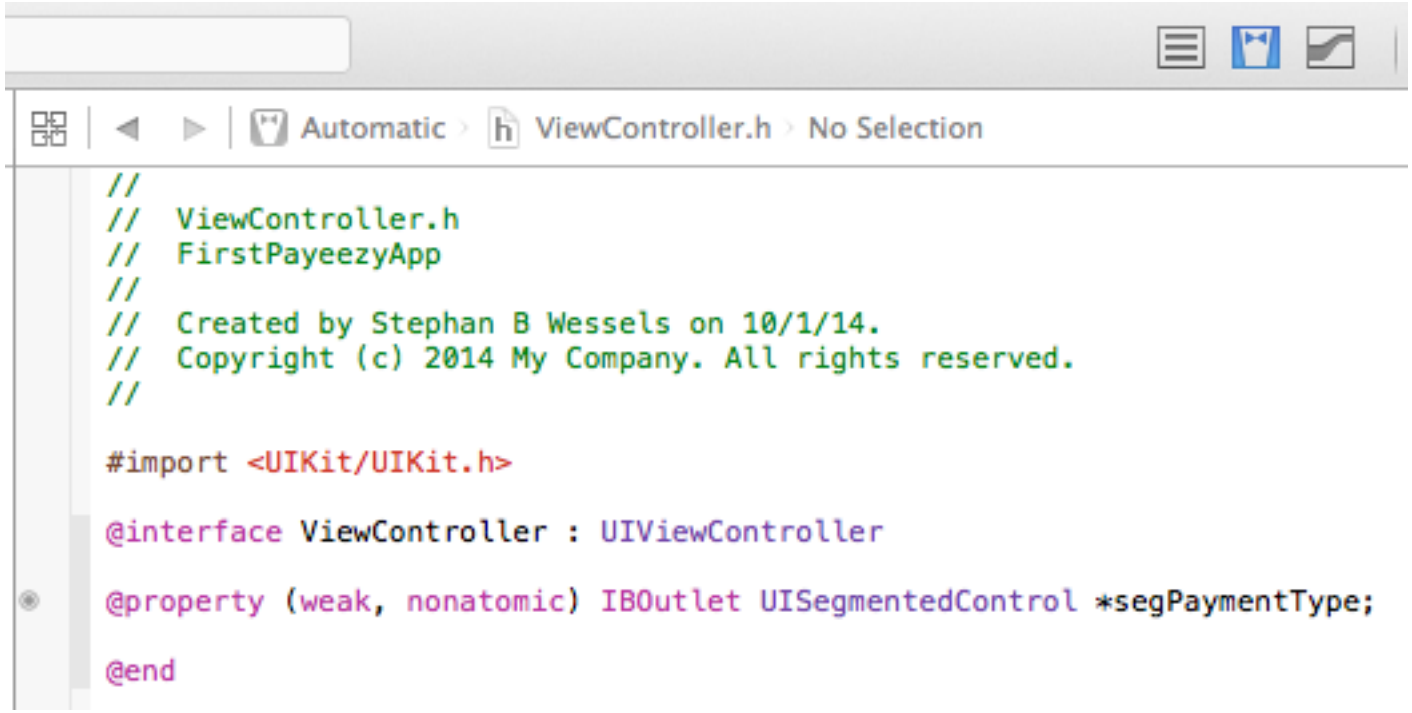
With the Storyboard open and the ViewController open, Control-Click and drag from the segmented control to the code pane.



Xcode will present a pop-up where you can define the name of the instance variable to associate with the visual object's outlet. Fill out the pop-up.



Name the property `segPaymentType` and click "connect". The following line of code will be added to your `ViewController.h`.

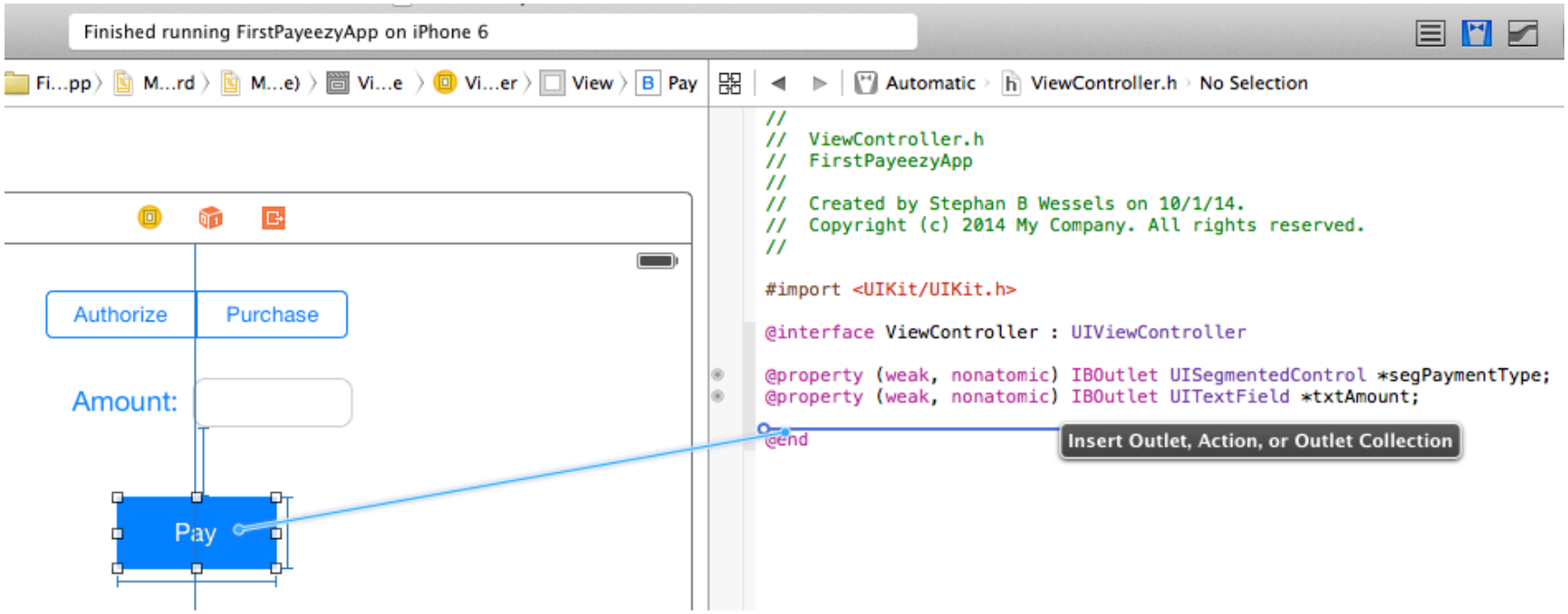



```
//
//  ViewController.h
//  FirstPayeezyApp
//
//  Created by Stephan B Wessels on 10/1/14.
//  Copyright (c) 2014 My Company. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segPaymentType;

@end
```

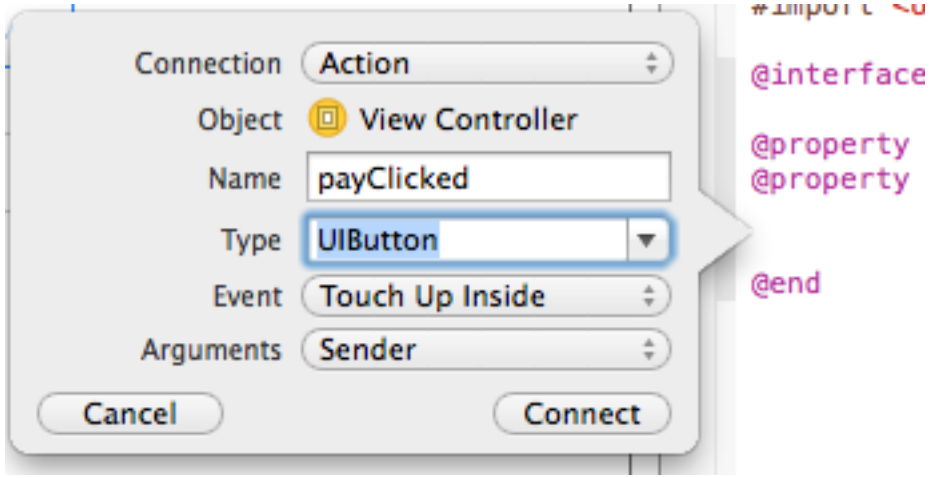
Do the same thing for the Text Field and name the property `txtAmount`. Your header file should now look like this.

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segPaymentType;
@property (weak, nonatomic) IBOutlet UITextField *txtAmount;

@end
```

We need to create an action related to pressing the “Pay” button. Control-drag from the “Pay” button to the header file.



When Xcode presents the property dialog, select `Action` in the popup and then set the other values as follows.

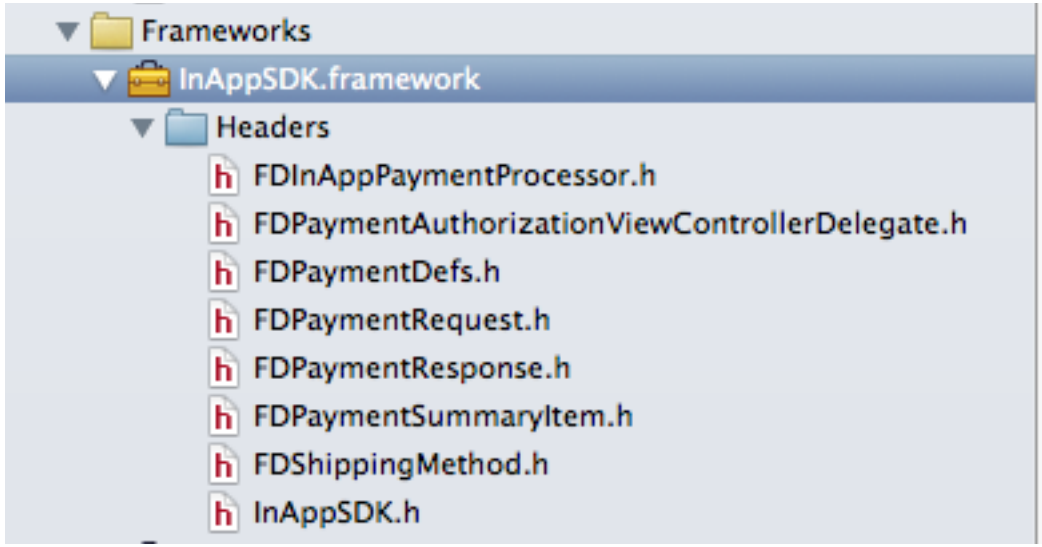


After clicking on “Connect” your header code should now look like this.

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segPaymentType;
@property (weak, nonatomic) IBOutlet UITextField *txtAmount;

- (IBAction)payClicked:(UIButton *)sender;

@end
```

Design Considerations

The frameworks you installed includes one named `InAppSDK.framework`. If you expand the Headers for the framework you can see the various `.h` files that define the developer accessible SDK.



The `FDInAppPaymentProcessor` is an instance you will create, typically in your App Delegate, that is used to send messages to Payeezy. The headers for the following…

```
FDPaymentRequest
FDPaymentResponse
FDPaymentSummaryItem
FDShippingMethod
```

are for the object models you will create as you interact with Payeezy. You will also assign the `FDPaymentAuthorizationViewControllerDelegate` protocol for callback messages as you interact with Apple Pay™ and Payeezy.

The `InAppSDK.h` is a convenience that pulls all the pieces together for you.

To begin to bolt all this together, let's go back to the `AppDelegate.h` file. We need to define constants that were obtained during the registration process and declare them in the app delegate.

Change your `AppDelegate.h` by adding these define statements as shown.

```
#import <UIKit/UIKit.h>

#define kApiKey             @"xHiPf8dp8TDzkRunn1Ovvf2hrOK5Y5YT"
#define kApiSecret
@"9cbb9acfc23a2440ef075d4696f13a373e26f9e9ab7c0a8ef9c7048ea3430bf8"
#define kMerchantToken      @"fdoa-a480ce8951daa73262734cf102641994c1e55e7cdf4c02b6"
#define kOsloMerchantId     @"3176752955"

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
```

The values used can be found from the Payeezy portal and/or in this document as part of the work you did earlier.

Use your own numbers/keys when you develop this example application.

It's time to add the `FDInAppPaymentProcessor` to the header. Change the header file to have the forward class definition and the new property for the processor we will create. When you are done your header should look like this (with your own keys substituted).

```
#import <UIKit/UIKit.h>

@class FDInAppPaymentProcessor;

#define kApiKey             @"xHiPf8dp8TDzkRunn1Ovvf2hrOK5Y5YT"
#define kApiSecret
@"9cbb9acfc23a2440ef075d4696f13a373e26f9e9ab7c0a8ef9c7048ea3430bf8"
#define kMerchantToken      @"fdoa-a480ce8951daa73262734cf102641994c1e55e7cdf4c02b6"
#define kOsloMerchantId     @"3176752955"

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) FDInAppPaymentProcessor *fdPaymentProcessor;

@end
```

Switch to the `AppDelegate.m`. Add the import statement to use the In App SDK. Include one new line as follows.

```
#import "AppDelegate.h"
#import <InAppSDK/InAppSDK.h>

@implementation AppDelegate
```

With this change all of the import statements we will need are available to us.

Add a new worker method to instantiate the payment processor for us.

```
- (void)instantiateFDLibService
{
    self.fdPaymentProcessor = [[FDInAppPaymentProcessor alloc] initWithApiKey:kApiKey
                                                                    apiSecret:kApiSecret

merchantToken:kMerchantToken

merchantIdentifier:kOsloMerchantId];

    // The app can choose which mode to send transactions in: pre-auth only or purchase
    // The default is purchase
    self.fdPaymentProcessor.paymentMode = FDPreAuthorization;
}
```

We will instantiate our payment process when the application launches. Change the `application:didFinishLaunchingWithOptions:` method.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    [self instantiateFDLibService];
    return YES;
}
```

It's time to work on the ViewController. We've created a button that initiates an action via the `payClicked:` method. For reasons that will become clear as we progress, we need to declare our ViewController to be an `FDPaymentAuthorizationViewControllerDelegate`. In the `ViewController.h` file change the header to import and declare the delegate protocol.

```
#import <UIKit/UIKit.h>
#import <InAppSDK/FDPaymentAuthorizationViewControllerDelegate.h>

@interface ViewController : UIViewController
<FDPaymentAuthorizationViewControllerDelegate>

@property (weak, nonatomic) IBOutlet UISegmentedControl *segPaymentType;
@property (weak, nonatomic) IBOutlet UITextField *txtAmount;

- (IBAction)payClicked:(UIButton *)sender;

@end
```

As we interact with the payment processing system we will receive specific callback messages through this delegate protocol. We'll cover details on those soon. For now, we have completed the edits to the `ViewController.h` file.

In the `ViewController.m` file we will begin to add behavior to our `payClicked:` method. We'll need a convenient way to interact with our App Delegate so we add that first. Imports for the App delegate and our new SDK will be needed too. Update the code as follows:

```
#import "ViewController.h"
#import "AppDelegate.h"
#import <InAppSDK/InAppSDK.h>

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

- (IBAction)payClicked:(UIButton *)sender
{
    AppDelegate *appDelegate = (AppDelegate *)[[UIApplication sharedApplication]
delegate];
}

@end
```

There are a few warnings now and they are related to our new unused variable and the fact that we have not implemented the required `FDPaymentAuthorizationViewControllerDelegate` methods yet.

To proceed we get to make a design decision. When a request is made of Apple Pay™, the application does not know which kinds of payment cards the user has installed in their Passbook App. However we do know which payment networks our application will support. We need to declare those now. Whether the user actually has a matching card for one of the networks we seek is not known.

Add the following to `payClicked:`.

```
- (IBAction)payClicked:(UIButton *)sender
{
    AppDelegate *appDelegate = (AppDelegate *)[[UIApplication sharedApplication]
delegate];
    NSArray *supportedNetworks = @[
                                   FDPaymentNetworkVisa,
                                   FDPaymentNetworkMasterCard,
                                   FDPaymentNetworkAmericanExpress
                                   ];
}
```

Here we are declaring an array to contain the payment networks our application supports.

There are two critical checks we need to make next. First it is possible someone may have installed our new application on an Apple device that does not support Apple Pay™. So we need to check that first. Second, even if the device can support Apple Pay™ the ability to make payments on that device is a user controlled setting. We need to also pass that check. This is easily done as follows in our method. Once we know using Apple Pay™ is possible we check if the user has at least one card in our supported networks.

```
- (IBAction)payClicked:(UIButton *)sender
{
    AppDelegate *appDelegate = (AppDelegate *)[[UIApplication sharedApplication]
delegate];
    NSArray *supportedNetworks = @[
                                   FDPaymentNetworkVisa,
                                   FDPaymentNetworkMasterCard,
                                   FDPaymentNetworkAmericanExpress
                                   ];
    // Does this device support In-App payments?
    if ([FDInAppPaymentProcessor canMakePayments])
    {
        // Is a card registered on the device for one of the merchant's supported card
networks?
        if ([FDInAppPaymentProcessor canMakePaymentsUsingNetworks:supportedNetworks])
        {
        }
        else
        {
            NSLog(@"Ability to make payments of merchant-supported network types was
rejected by FD SDK");
        }
    }
    else
    {
        NSLog(@"Ability for device to make payments was rejected by FD SDK");
    }
}
```

For this example we'll just log the error condition. You'll want better user feedback in your own application.

If we can proceed then next step is to create an `FDPaymentRequest` object.

```
- (IBAction)payClicked:(UIButton *)sender
{
    AppDelegate *appDelegate = (AppDelegate *)[[UIApplication sharedApplication]
delegate];
    NSArray *supportedNetworks = @[
                                   FDPaymentNetworkVisa,
                                   FDPaymentNetworkMasterCard,
                                   FDPaymentNetworkAmericanExpress
                                   ];
    // Does this device support In-App payments?
    if ([FDInAppPaymentProcessor canMakePayments])
    {
        // Is a card registered on the device for one of the merchant's supported card
networks?
        if ([FDInAppPaymentProcessor canMakePaymentsUsingNetworks:supportedNetworks])
        {
            // Populate the payment request
            FDPaymentRequest *pmtRqst = [[FDPaymentRequest alloc] init];
            pmtRqst.merchantIdentifier = kOsloMerchantId;
            pmtRqst.supportedNetworks = supportedNetworks;
            pmtRqst.countryCode = @"US";
            pmtRqst.currencyCode = @"USD";
            pmtRqst.merchantCapabilities = FDMerchantCapability3DS |
FDMerchantCapability3EMV;
            pmtRqst.requiredShippingAddressFields = FDAddressFieldNone;
        }
        else
        {
            NSLog(@"Ability to make payments of merchant-supported network types was
rejected by FD SDK");
        }
```

```
    }
    else
    {
        NSLog(@"Ability for device to make payments was rejected by FD SDK");
    }
}
```

Next we look at the GUI we made and determine whether we are performing an Authorize or Purchase action. Our `segPaymentType` property holds the chosen option.

We tell the payment processor about it by setting a property. Add one line of code inside where we have been building up our payment request.

```
        {
            // Populate the payment request
            FDPaymentRequest *pmtRqst = [[FDPaymentRequest alloc] init];
            pmtRqst.merchantIdentifier = kOsloMerchantId;
            pmtRqst.supportedNetworks = supportedNetworks;
            pmtRqst.countryCode = @"US";
            pmtRqst.currencyCode = @"USD";
            pmtRqst.merchantCapabilities = FDMerchantCapability3DS |
FDMerchantCapability3EMV;
            pmtRqst.requiredShippingAddressFields = FDAddressFieldNone;

            // Set the payment type
            appDelegate.fdPaymentProcessor.paymentMode =
(self.segPaymentType.selectedSegmentIndex==0 ? FDPreAuthorization : FDPurchase);
        }
```

The next step is to declare an `FDShippingMethod`. Add the following code so your conditional code looks as follows:

```
        {
            // Populate the payment request
            FDPaymentRequest *pmtRqst = [[FDPaymentRequest alloc] init];
            pmtRqst.merchantIdentifier = kOsloMerchantId;
            pmtRqst.supportedNetworks = supportedNetworks;
            pmtRqst.countryCode = @"US";
            pmtRqst.currencyCode = @"USD";
            pmtRqst.merchantCapabilities = FDMerchantCapability3DS |
FDMerchantCapability3EMV;
            pmtRqst.requiredShippingAddressFields = FDAddressFieldNone;

            // Set the payment type
            appDelegate.fdPaymentProcessor.paymentMode =
(self.segPaymentType.selectedSegmentIndex==0 ? FDPreAuthorization : FDPurchase);

            FDShippingMethod *shipping = [[FDShippingMethod alloc] init];
            shipping.identifier = @"Two Day Shipping";
            shipping.detail = @"Two day shipping to the Continental US";
            pmtRqst.shippingMethods = @[shipping];
        }
```

Now we have to add the details about what is being purchased. If we have more than one item in our order we would add more lines. Here, we'll add one item and purchase some "large shoes". Not really very descriptive, but…

The amount is pulled from the interface again in our `txtAmount` property.

```
        {
            // Populate the payment request
            FDPaymentRequest *pmtRqst = [[FDPaymentRequest alloc] init];
            pmtRqst.merchantIdentifier = kOsloMerchantId;
            pmtRqst.supportedNetworks = supportedNetworks;
            pmtRqst.countryCode = @"US";
            pmtRqst.currencyCode = @"USD";
            pmtRqst.merchantCapabilities = FDMerchantCapability3DS |
FDMerchantCapability3EMV;
            pmtRqst.requiredShippingAddressFields = FDAddressFieldNone;

            // Set the payment type
            appDelegate.fdPaymentProcessor.paymentMode =
(self.segPaymentType.selectedSegmentIndex==0 ? FDPreAuthorization : FDPurchase);

            FDShippingMethod *shipping = [[FDShippingMethod alloc] init];
            shipping.identifier = @"Two Day Shipping";
            shipping.detail = @"Two day shipping to the Continental US";
            pmtRqst.shippingMethods = @[shipping];

            // Create a sample order
            FDPaymentSummaryItem *item1 = [[FDPaymentSummaryItem alloc] init];
            item1.label = @"Large Shoes";
            item1.amount = [NSDecimalNumber decimalNumberWithString:self.txtAmount.text];
        }
```

We're building up an array of individual items on the order. In this example there is just one.

The last item in the array is the total line.

```
        {
            // Populate the payment request
            FDPaymentRequest *pmtRqst = [[FDPaymentRequest alloc] init];
            pmtRqst.merchantIdentifier = kOsloMerchantId;
            pmtRqst.supportedNetworks = supportedNetworks;
            pmtRqst.countryCode = @"US";
            pmtRqst.currencyCode = @"USD";
            pmtRqst.merchantCapabilities = FDMerchantCapability3DS |
FDMerchantCapability3EMV;
            pmtRqst.requiredShippingAddressFields = FDAddressFieldNone;

            // Set the payment type
            appDelegate.fdPaymentProcessor.paymentMode =
(self.segPaymentType.selectedSegmentIndex==0 ? FDPreAuthorization : FDPurchase);

            FDShippingMethod *shipping = [[FDShippingMethod alloc] init];
            shipping.identifier = @"Two Day Shipping";
            shipping.detail = @"Two day shipping to the Continental US";
            pmtRqst.shippingMethods = @[shipping];

            // Create a sample order
            FDPaymentSummaryItem *item1 = [[FDPaymentSummaryItem alloc] init];
            item1.label = @"Large Shoes";
            item1.amount = [NSDecimalNumber decimalNumberWithString:self.txtAmount.text];

            // Total line
            FDPaymentSummaryItem *item2 = [[FDPaymentSummaryItem alloc] init];
            item2.label = @"FD Test Merchant1";
            item2.amount = [NSDecimalNumber decimalNumberWithString:self.txtAmount.text];
            NSArray *itemArray = [NSArray arrayWithObjects: item1, item2, nil];
            pmtRqst.paymentSummaryItems = itemArray;
        }
```

The `itemArray` contains the items for the order. Apple recommends your total line should contain the merchant name as the description. After creating the order items array we set the property on our payment request (`pmtRqst`).

There's one more value we can use in the `pmtRqst`. Modify the last few lines of our conditional branch by adding the new code shown.

```
            // Total line
            FDPaymentSummaryItem *item2 = [[FDPaymentSummaryItem alloc] init];
            item2.label = @"FD Test Merchant1";
            item2.amount = [NSDecimalNumber decimalNumberWithString:self.txtAmount.text];
            NSArray *itemArray = [NSArray arrayWithObjects: item1, item2, nil];
            pmtRqst.paymentSummaryItems = itemArray;

            // Send a sample application data payload
            NSString *appDataString = @"RefCode:12345; TxID:
342340892409823048230948234 32";
            pmtRqst.applicationData = [appDataString
dataUsingEncoding:NSUTF8StringEncoding];

        }
        else
```

The `NSString *appDataString` put into `pmtRqst.applicationData` after encoding, is an optional parameter. Apple recommends this can be used for additional data as appropriate for your app—for example, a shopping cart identifier or an order number. See Apple's docs about this parameter here. This is an optional value that you control. We are putting the string `RefCode:12345; TxID: 342340892409823048230948234 32` in our payment request for this example. The data will be encrypted before it gets sent to the server.

The payment request object is ready. At this point the application must present the payment request object to the `FDInAppPaymentProcessor`.

Add the lines shown below after we set the optional `applicationData`.

```
            BOOL bPaymentOK = [appDelegate.fdPaymentProcessor
presentPaymentAuthorizationViewControllerWithPaymentRequest:pmtRqst
presentingController:self delegate:self];

            if( !bPaymentOK ) {
                UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error"
                                                                message:@"Payment request was
rejected by Apple Pay server" delegate:self
                                                      cancelButtonTitle:@"Dismiss"
                                                      otherButtonTitles:nil];
                [alert show];
                NSLog(@"Payment request was rejected by Apple Pay server");
            }

            // Delegate methods are driving from here…
```

When we interact with the payment processor we are turning over control to Apple's Passbook code. Apple will present the next view controller and handle interaction with the customer.

If there was a problem the boolean value will be `NO` and your application can present an alert dialog.

If the payment request was accepted by the iPhone all following operations are sent as messages to your delegate. We will add those delegate messages soon. To help you make sure there were no errors made as you built-up the previous section of code the entire `payClicked:` method is repeated here for reference.

```
- (IBAction)payClicked:(UIButton *)sender
{
    AppDelegate *appDelegate = (AppDelegate *)[[UIApplication sharedApplication]
delegate];
    NSArray *supportedNetworks = @[
                                   FDPaymentNetworkVisa,
                                   FDPaymentNetworkMasterCard,
                                   FDPaymentNetworkAmericanExpress
                                   ];
    // Does this device support In-App payments?
    if ([FDInAppPaymentProcessor canMakePayments])
    {
        // Is a card registered on the device for one of the merchant's supported card
networks?
        if ([FDInAppPaymentProcessor canMakePaymentsUsingNetworks:supportedNetworks])
        {
            // Populate the payment request
            FDPaymentRequest *pmtRqst = [[FDPaymentRequest alloc] init];
            pmtRqst.merchantIdentifier = kOsloMerchantId;
            pmtRqst.supportedNetworks = supportedNetworks;
            pmtRqst.countryCode = @"US";
            pmtRqst.currencyCode = @"USD";
            pmtRqst.merchantCapabilities = FDMerchantCapability3DS |
FDMerchantCapability3EMV;
            pmtRqst.requiredShippingAddressFields = FDAddressFieldNone;

            // Set the payment type
            appDelegate.fdPaymentProcessor.paymentMode =
(self.segPaymentType.selectedSegmentIndex==0 ? FDPreAuthorization : FDPurchase);

            FDShippingMethod *shipping = [[FDShippingMethod alloc] init];
            shipping.identifier = @"Two Day Shipping";
            shipping.detail = @"Two day shipping to the Continental US";
```

```
            pmtRqst.shippingMethods = @[shipping];

            // Create a sample order
            FDPaymentSummaryItem *item1 = [[FDPaymentSummaryItem alloc] init];
            item1.label = @"Large Shoes";
            item1.amount = [NSDecimalNumber decimalNumberWithString:self.txtAmount.text];

            // Total line
            FDPaymentSummaryItem *item2 = [[FDPaymentSummaryItem alloc] init];
            item2.label = @"FD Test Merchant1";
            item2.amount = [NSDecimalNumber decimalNumberWithString:self.txtAmount.text];
            NSArray *itemArray = [NSArray arrayWithObjects: item1, item2, nil];
            pmtRqst.paymentSummaryItems = itemArray;

            // Send a sample application data payload
            NSString *appDataString = @"RefCode:12345; TxID:
342340892409823048230948234 32";
            pmtRqst.applicationData = [appDataString
dataUsingEncoding:NSUTF8StringEncoding];

            BOOL bPaymentOK = [appDelegate.fdPaymentProcessor
presentPaymentAuthorizationViewControllerWithPaymentRequest:pmtRqst
presentingController:self delegate:self];

            if( !bPaymentOK ) {
                UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error"
                                                                message:@"Payment request was
rejected by Apple Pay server" delegate:self
                                                      cancelButtonTitle:@"Dismiss"
                                                      otherButtonTitles:nil];
                [alert show];
                NSLog(@"Payment request was rejected by Apple Pay server");
            }
```

```
            // Delegate methods are driving from here...

        }
        else
        {
            NSLog(@"Ability to make payments of merchant-supported network types was
rejected by FD SDK");
        }
    }
    else
    {
        NSLog(@"Ability for device to make payments was rejected by FD SDK");
    }
}
```

Don't forget you will want to handle your `else` conditions with something more helpful than `NSLog`.

The FDPaymentAuthorizationViewControllerDelegate Methods

There are four delegate methods we need to provide. After the `payClicked:` method add a pragma statement to the code to make these delegate methods easier to find.

```
#pragma mark - FDPaymentAuthorizationViewControllerDelegate
```

Add the first of our delegate methods.å

```
- (void)paymentAuthorizationViewController:(UIViewController *)controller
                       didAuthorizePayment:(FDPaymentResponse *)paymentResponse
{

    NSString *authStatusMessage = nil;

    if (paymentResponse.validationStatus != FDPaymentValidationStatusSuccess)
    {
        authStatusMessage = @"Transaction Validation or communication failure. Please try
again.";
    }
    else if (paymentResponse.authStatus == FDPaymentAuthorizationStatusFailure)
    {
        authStatusMessage = [NSString stringWithFormat:@"Transaction was validated but
authorization failed with reason: %@", paymentResponse.transStatusMessage];
    }
    else if (paymentResponse.authStatus == FDPaymentAuthorizationStatusSuccess)
    {
        authStatusMessage = [NSString stringWithFormat:@"Transaction Successful\rType:%@
\rTransaction ID:%@\rTransaction Tag:%@",
                             paymentResponse.transactionType,
                             paymentResponse.transactionID,
```

```
                                paymentResponse.transactionTag];
    }

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"First Data Payment
Authorization"
                                                    message:authStatusMessage
delegate:self
                                          cancelButtonTitle:@"Dismiss"
                                          otherButtonTitles:nil];
    [alert show];

    // App can send order to the merchant's back-end server now
}
```

The `paymentAuthorizationViewController:didAuthorizePayment:` delegate method is called when the customer has authorized the payment.

The `paymentResponse` can be checked by the techniques shown to know if the response was both valid and authorized.

Your application can notify the merchant's back-end server at this point.

The next delegate method is called as the SDK dismisses the View Controller it used to interact with the customer. Add this delegate method now.

```
- (void)paymentAuthorizationViewControllerDidFinish:(UIViewController *)controller
{
    // Nothing to do here - the SDK handles all cleanup

    NSLog(@"ViewController:paymentAuthorizationViewControllerDidFinish invoked");
}
```

The only thing we do in here is log that the delegate was invoked.

The two remaining delegate methods are related to customer choices involving shipping information.

Add this delegate method.

```
- (void)paymentAuthorizationViewController:(UIViewController *)controller
                   didSelectShippingMethod:(FDShippingMethod *)shippingMethod
{
    // The customer has asked the merchant to use a specific shipping method.  We need to
update the amount.
    // We can also get here if the customer said the heck with it and canceled.

    NSLog(@"ViewController:didSelectShippingMethod invoked");
}
```

When the customer selects a shipping method your application is called by the SDK. This allows you to update the total amount if shipping method is impactful.

Add our final delegate method.

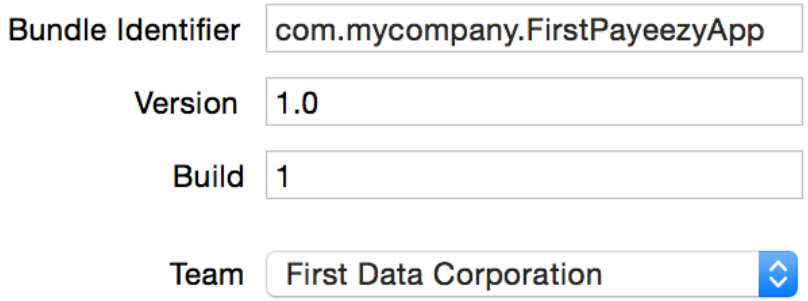
```
- (void)paymentAuthorizationViewController:(UIViewController *)controller
                  didSelectShippingAddress:(ABRecordRef)address
{
    // The customer is telling the merchant shipping address information.  We need to
update the amount.
    // We can also get here if the customer said the heck with it and canceled.

    NSLog(@"ViewController:didSelectShippingAddress invoked");
}
```

Your application is also called through this delegate method if the customer has changed the shipping address. This mechanism provides for a way to change the amount if necessary.

Managing Entitlements

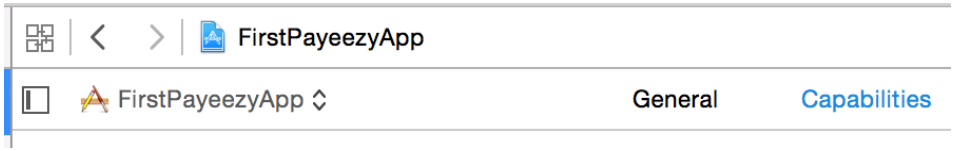
Entitlements are automatically added to your Xcode project when you turn on Apple Pay settings. However there are a few things we must check before we proceed.

Select your project in Xcode and go to the General tab. Look up the **Identity** information for your project.

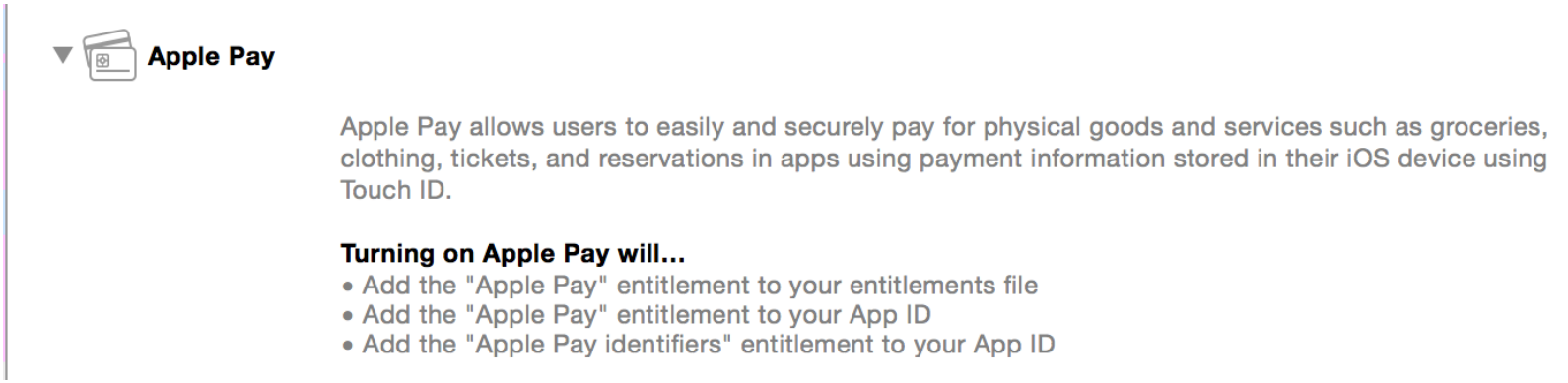


Ensure you have the proper Team selected and that there are no issues to resolve.

Select the Capabilities tab.

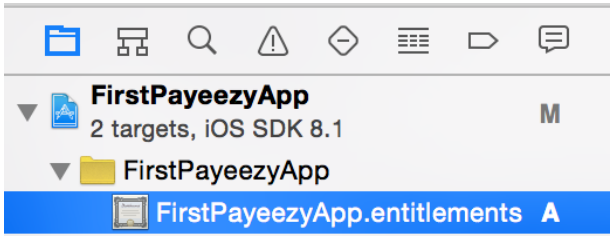


Toggle the disclosure triangle on Apple Pay. You should see a description of what will happen when you enable this capability.

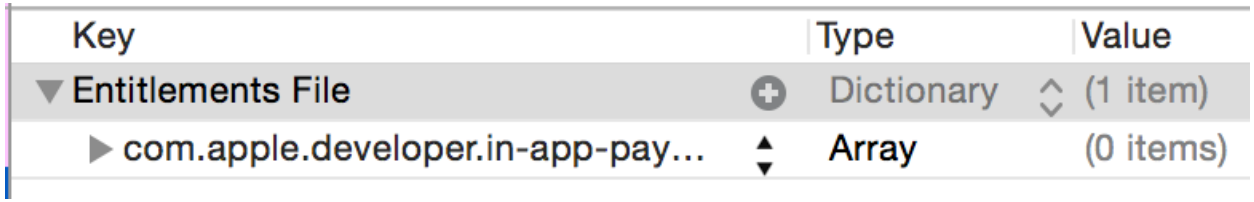


Turn the capability on.

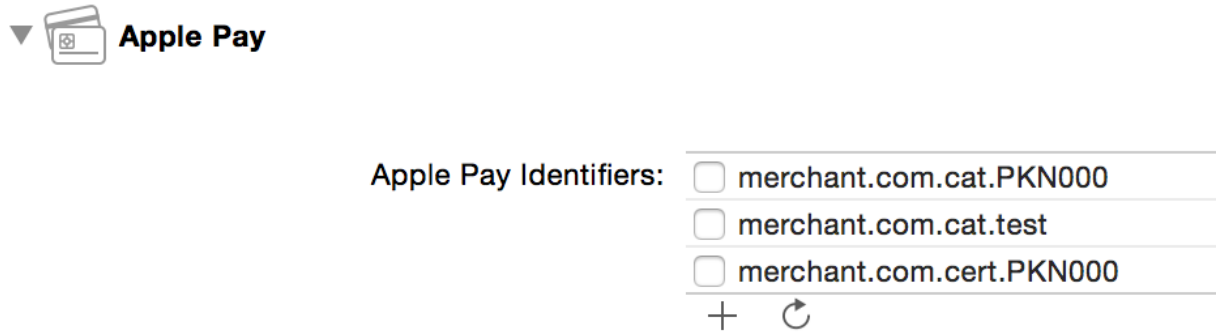
Several things will happen automatically. You should notice a new entitlements file has been added to your project.



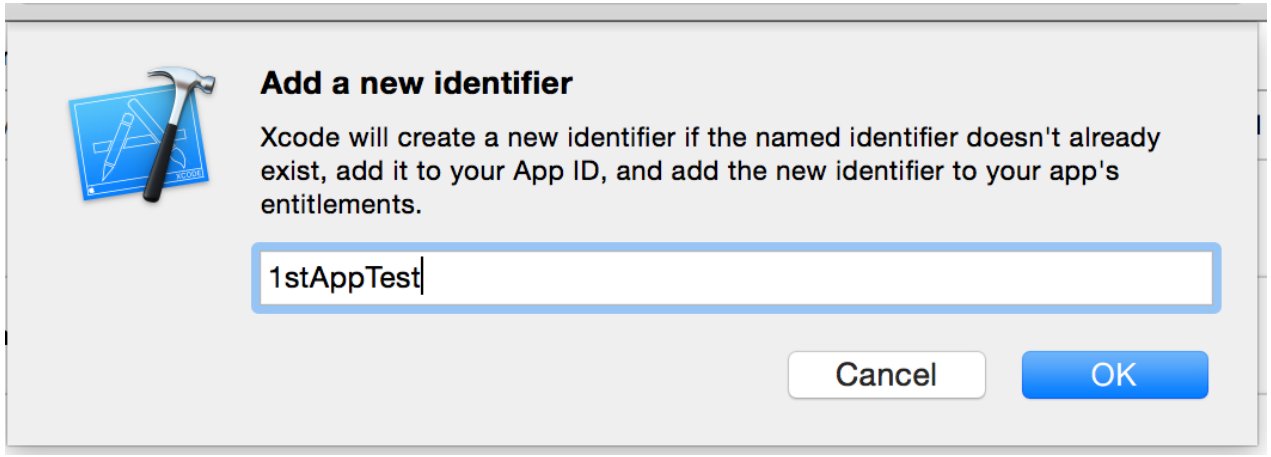
When you select the entitlements file it will contain a Dictionary containing an empty Array.



Under the Apple Pay capability a list of known Apple Pay Identifiers is shown.



Select one or more of the Merchant IDs from this list. You can also add a new one by using the + button below the table.



Once you have Apple Pay™ enabled you will also need to enable In-App Purchase.

You may find errors as you proceed. Xcode will identify what you need to do.