Getting Started with iOS Mobile Development and a Sinatra API

Last updated 15 November 2017

Table of Contents

This quickstart will have you up and running with a native iOS application that consumes a web API with Sinatra (http://www.sinatrarb.com/). For additional resources about Sinatra, please see the Ruby quickstart (https://devcenter.heroku.com/articles/getting-started-with-ruby).

Sample code for the **Sinatra application (https://github.com/heroku/devcenter-ios-api)** and the **iOS Client (https://github.com/heroku/devcenter-ios-client)** is available on GitHub.

Prerequisites

- Basic Objective-C knowledge, including a development environment running Mac OS X with Xcode 4.2 (http://developer.apple.com/xcode/) installed.
- Basic Ruby knowledge, including an installed version of Ruby 1.9.2, Rubygems, Bundler, and Sinatra.
- A Heroku user account. Signup is free and instant (https://signup.heroku.com/devcenter).

Local workstation setup

Start by setting up the Heroku CLI (https://devcenter.heroku.com/articles/heroku-cli) and then logging into Heroku to upload your ssh public key. If you've used Heroku before and already have a working local setup, skip to the next section.

Once installed, you'll have access to the `heroku` command from your command shell. Log in using the email address and password you used when creating your Heroku account:

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/adam/.ssh/id_rsa.pub
```

Press enter at the prompt to upload your existing ssh key or create a new one, used for pushing code later on.

Write your web API app

Following the client-server model (http://en.wikipedia.org/wiki/Client%E2%80%93server_model), this guide implements both a Sinatra application on the server, and an iOS client that communicates with it.

Create a Sinatra application using the following example:

api.rb

```
require 'sinatra'
require 'json'

get '/sushi.json' do
  content_type :json
  return {:sushi => ["Maguro", "Hamachi", "Uni", "Saba", "Ebi", "Sake", "Tai"]}.to_json
end
```

Declare gem dependencies with Bundler

Heroku recognizes an app as Ruby by the existence of a `Gemfile`. Even if your app has no gem dependencies, you should still create an empty `Gemfile` in order that it appears as a Ruby app.

In local testing, you should be sure to run your app in an isolated environment (via `bundle exec` or an empty RVM gemset), to make sure that all gems your app depends on are in the `Gemfile`.

Here's an example `Gemfile` for the Sinatra app created above:

Gemfile

```
source :rubygems
gem 'sinatra'
gem 'unicorn'
gem 'json'
```

Run `bundle install` to set up you bundle locally

Declare process types with a Procfile

Cedar will recognize any app with a `config.ru` as a Rack app and generate a web process type for you. However, you'll get more control over how your app is executed if you skip `config.ru` and instead declare your own web process type in a Procfile (https://devcenter.heroku.com/articles/procfile).

Here's a `Procfile` for the sample app we've been working on:

```
web: bundle exec unicorn -p $PORT
```

Now that you have a `Procfile`, you can start your application locally:

```
$ heroku local
```

Your app will come up on port 5000. Test that it's working with `curl` or a web browser, then Ctrl-C to exit.

```
$ curl http://localhost:5000/sushi.json
{"sushi":["Maguro","Hamachi","Uni","Saba","Ebi","Sake","Tai"]}
```

Store your app in Git

The three major components of the web app are now in place: dependencies in `Gemfile`, process types in `Procfile`, and the application source in `api.rb`. Add everything into a new Git repository:

```
$ git init
$ git add .
$ git commit -m "init"
```

Deploy to Heroku

Create the app on the Cedar stack:

```
$ heroku create --stack cedar-14
```

Deploy your code:

```
$ git push heroku master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 660 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)

-----> Heroku receiving push
-----> Ruby app detected
...
-----> Compiled slug size is 6.3MB
-----> Launching... done, v4
       http://morning-wind-6672.herokuapp.com deployed to Heroku

To git@heroku.com:morning-wind-6672.git
 * [new branch]      master -> master
```

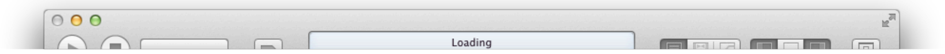
Test that your application is running with `curl` or a web browser:

```
$ curl http://myapp.herokuapp.com/sushi.json
{"sushi":["Maguro","Hamachi","Uni","Saba","Ebi","Sake","Tai"]}
```

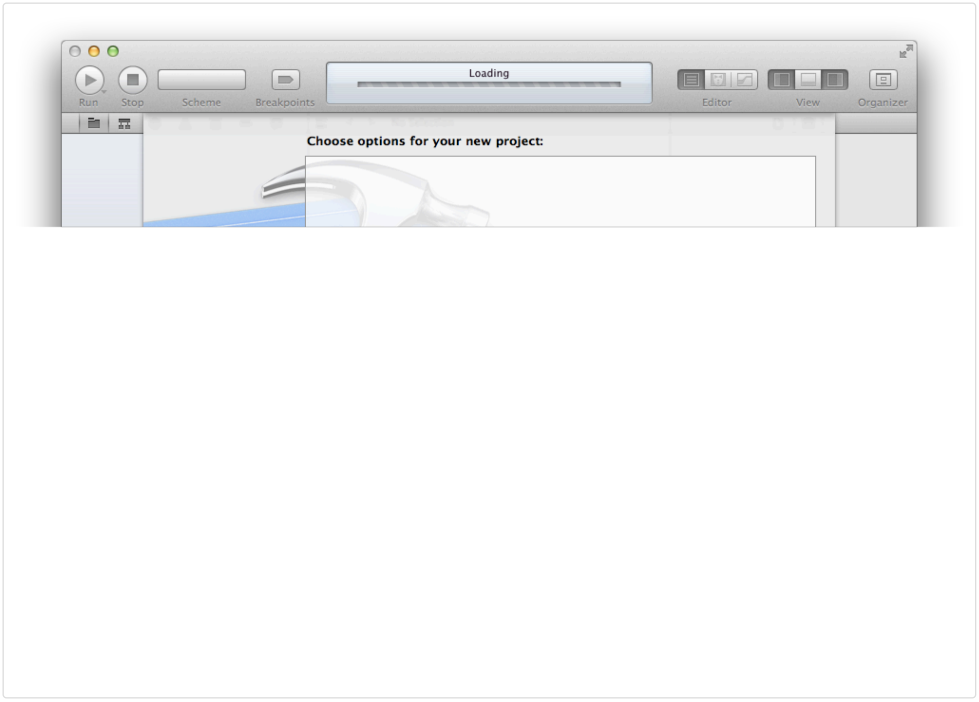
Create your iOS client app

With the server up and running, it's time to create the iOS client.

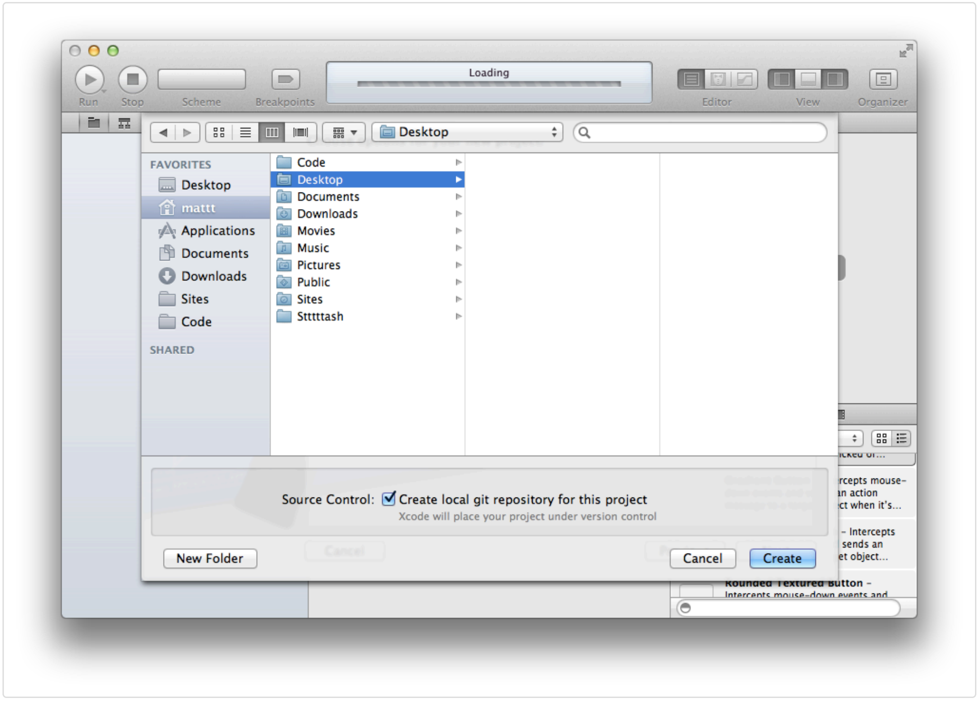
Open up Xcode and select "File > New ▶ > New Project…", or use the keyboard shortcut, `⇧⌘N` .



When prompted to choose a template for your new project, select iOS - Application on the sidebar, and choose the "Empty Application" template. Click "Next" to continue.



In the next step, enter your Product Name, Company Identifier, and Class Prefix (optional). For “Device Family”, select “iPhone”. Make sure that the checkboxes for “Use Core Data”, “Use Automatic Reference Counting”, and “Include Unit Tests” are unchecked. Click “Next” to continue.



Source Control: [✓] Create local git repository for this project
Xcode will place your project under version control

Finally, select a directory to save your new project to, check the box to create a local Git repository for this project, and click “Create”.

Declare iOS dependencies with CocoaPods

Instead of using CocoaPods to manage dependencies, you could also install AFNetworking manually. **Download the latest release (https://github.com/AFNetworking/AFNetworking/tarball/0.9.0)** and add the enclosed "AFNetworking" directory to your project folder.

CocoaPods (http://cocoapods.org/) manages library dependencies for your Xcode project, similar to the way Bundler manages Ruby gem dependencies.

Now install CocoaPods:

```
$ gem install cocoapods
$ pod setup
```

CocoaPods dependencies are declared in `Podfile` . In this case, AFNetworking (https://github.com/AFNetworking/AFNetworking) will be used to communicate with the web API. Copy the following `Podfile` in the root of your iOS project:

Podfile

```
platform :ios
dependency 'AFNetworking', '0.9'
```

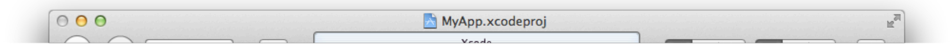
Run `pod install MyApp.xcodeproj` to setup the dependencies:

```
$ pod install MyApp.xcodeproj
Installing AFNetworking (0.9.0)
Generating support files
[!] From now on use `MyApp.xcworkspace' instead of `MyApp.xcodeproj'.
```

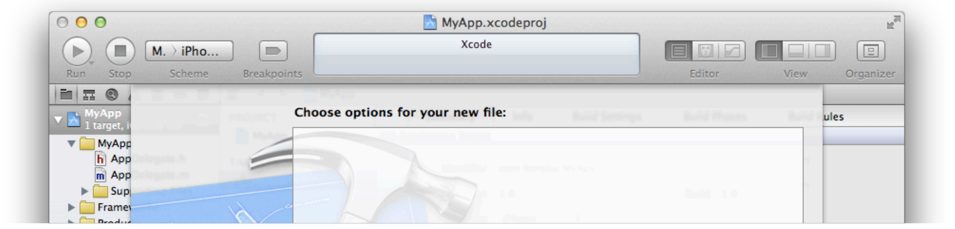
Following the instructions from CocoaPods, close `MyApp.xcodeproj` and open `MyApp.xcworkspace` . This workspace file contains the build targets for your project as well as its dependencies.

Write a table view controller

In Xcode, select "File > New ▶ > New File...", or use the keyboard shortcut, `⌘N` .



When prompted to choose a template for your new file, select iOS - Cocoa Touch on the sidebar, and choose the "UIViewController subclass" template. Click "Next" to continue.



`SushiTableViewController` will download and parse the JSON response from the API, and display the results in a `UITableView`.

In the header, add a `sushi` property to store the results received from the server.

SushiTableViewController.h

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

@interface SushiTableViewController : UITableViewController
@property (nonatomic, retain) NSArray *sushi;
@end
```

Switching to the implementation, now `@synthesize` the `sushi` property, request the JSON in `-viewDidLoad`, and implement the required methods in `UITableViewDataSource`:

SushiTableViewController.m

```
#import "SushiTableViewController.h"

#import "AFJSONRequestOperation.h"

static NSString * const kMyAppBaseURLString = @"http://morning-wind-6672.herokuapp.com/";

@implementation SushiTableViewController
@synthesize sushi = _sushi;

#pragma mark - UIViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    self.title = NSLocalizedString(@"Sushi", nil);

    NSURL *url = [NSURL URLWithString:[kMyAppBaseURLString stringByAppendingPathComponent:@"sushi.json"]];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    AFJSONRequestOperation *operation = [AFJSONRequestOperation JSONRequestOperationWithRequest:request succe
        self.sushi = [JSON valueForKey:@"sushi"];
        [self.tableView reloadData];
    } failure:^(NSURLRequest *request, NSHTTPURLResponse *response, NSError *error, id JSON) {
        NSLog(@"Error: %@", error);
    }];
    [operation start];
}

#pragma mark - UITableViewDataSource

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    return [self.sushi count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
    if (!cell) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdenti
    }

    cell.textLabel.text = [self.sushi objectAtIndex:indexPath.row];

    return cell;
}

@end
```

`-viewDidLoad` is a method called at the beginning of the view controller lifecycle. Set the view controller's title and begin a network request to the API to get the list of sushi to display.

`AFJSONRequestOperation` will asynchronously load data from the API and execute the block in the `success` parameter when the request completes. In this case, the success block stores the results and tells the table view to reload its content.

In order to have the table view display the results from the API, implement the two required `UITableViewDataSource` delegate methods. `-tableView:numberOfRowsInSection:` is simply the `count` of the `sushi` property. `tableView:cellForRowAtIndexPath:` either dequeues a reusable table view cells from the table view's cache or creates a new one, and sets the text to the element that corresponds to that row.

Configure AppDelegate

With the table view controller finished, hook it up to the App Delegate to be displayed when the app is launched.

Add a `navigationController` property to `AppDelegate.h` :

AppDelegate.h

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

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) UINavigationController *navigationController;

@end
```

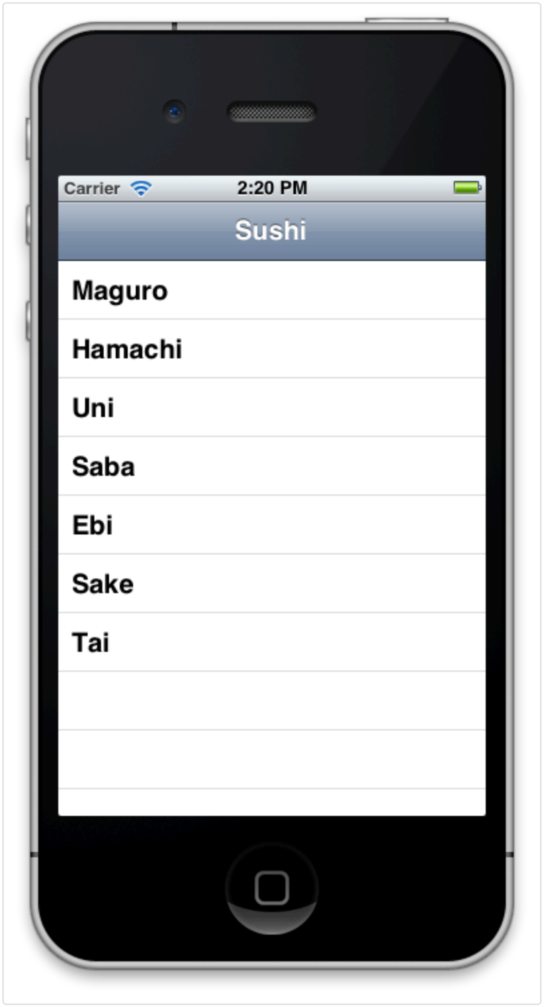
In the implementation, import `SushiTableViewController.h` . Then in `application:didFinishLaunchingWithOptions` , instantiate an instance of the table view controller, set it as the root view controller for a navigation controller, and add the navigation controller view to the application window.

AppDelegate.m

```
@implementation AppDelegate
@synthesize window = _window;
@synthesize navigationController = _navigationController;

- (void)dealloc {
    [_window release];
    [_navigationController release];
    [super dealloc];
}

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
    self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] autorelease];
    self.window.backgroundColor = [UIColor whiteColor];

    SushiTableViewController *viewController = [[[SushiTableViewController alloc] initWithStyle:UITableViewSt
    self.navigationController = [[[UINavigationController alloc] initWithRootViewController:viewController] a
    [self.window addSubview:self.navigationController.view];

    [self.window makeKeyAndVisible];

    return YES;
}

@end
```

Build and run

In Xcode, click the “Run” play button, or use the keyboard shortcut, ⌘R . If everything worked, your app should be displaying the results fetched from your API.



Maguro
Hamachi
Uni
Saba
Ebi
Sake
Tai

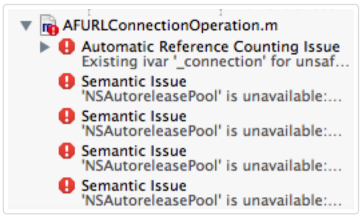
Troubleshooting

If you push up your app and it crashes (`heroku ps` shows state `crashed`), check your logs with `heroku logs` to find out what went wrong.

Failed to require a source file

If your Sinatra app failed to require a source file, chances are good you're running Ruby 1.9.1 or 1.8 in your local environment. The load paths have changed in Ruby 1.9. Port your app forward to Ruby 1.9.2 making certain it works locally before trying to push to Cedar again.

Unable to build Xcode project


AFURLConnectionOperation.m
Automatic Reference Counting Issue
Existing ivar '_connection' for unsaf…
Semantic Issue
'NSAutoreleasePool' is unavailable:…
Semantic Issue
'NSAutoreleasePool' is unavailable:…
Semantic Issue
'NSAutoreleasePool' is unavailable:…
Semantic Issue
'NSAutoreleasePool' is unavailable:…

If your Xcode project uses ARC (http://clang.llvm.org/docs/AutomaticReferenceCounting.html) and fails to "Build and Run", and the Issue Navigator reports errors such as "ARC forbids explicit message send of 'release'" or "'release' is unavailable", then your project includes source files that cannot be compiled with ARC.

To fix this, select your project file at the top of the Source Navigator, select your active target under the "Targets" section, and click the "Build Phases" tab in the editor screen. Expand the "Compile Sources" phase, and for each source file that does not support ARC, add the following compiler flag: `-fno-objc-arc` . You should now be able to successfully build your project.

Advanced configuration

In `SushiTableViewController.m` , the URL endpoint for the server is stored in the string constant `kMyAppBaseURLString`. Rather than manually switching between `localhost` and the Heroku URL when going between development and production, you can use a compiler macro to define the value dynamically:

```
#if TARGET_IPHONE_SIMULATOR
static NSString * const kMyAppBaseURLString = @"http://localhost:5000/";
#else
static NSString * const kMyAppBaseURLString = @"http://morning-wind-6672.herokuapp.com/";
#endif
```

`kMyAppBaseURLString` is still a string constant, but now it's value will be determined at compile-time, according to which build target is active: `localhost:5000` if on the simulator, and the Heroku URL when on the device.

Next steps

At this point you have a iPhone app communicating with a web API deployed to Heroku. Using this architecture, you can build out a fully-featured application, ready to be sold and distributed on the App Store.