Hyperloop for iOS Programming Guide

> *This documentation is made available before final release and is subject to change without notice and comes with no warranty express or implied.*

Requirements

You'll need to have the following minimum requirements to use Hyperloop for iOS:

- Titanium 5.2.0
- iOS 9.0 SDK

Hyperloop only works with Titanium applications that are registered with the platform. If you are using a Titanium project that hasn't yet been registered, you can register it with the following command: `appc new --import`.

If you'd like to experience Hyperloop on your application before registering your application, you can use the following demo GUID in your tiapp.xml: `11111111-1111-1111-1111-111111111111`. However, when using a demo GUID, your application will only operate on the simulator.

Pre-release Installation

For pre-release, you'll need to update to the latest unreleased version of Titanium 5.2.0 by running `ti sdk install -b master -d`. Make sure you set the version of your application to use this version in your `tiapp.xml` `<sdk-version>`.

To run the examples application, you'll need to also install CocoaPods by running `sudo gem install cocoapods`. *NOTE: some users have reported problems with the built-in OSX Ruby version (1.9). CocoaPods seems to require a 2.0 or later version to install and work properly.*

Using the Hyperloop Examples project

If you're going to run the Hyperloop Example project, you do not need to do any additional installation to use Hyperloop. The project will allow you to run on the simulator using `appc ti`

`build -p ios`.

Using Hyperloop in your own project

If you'd like to use Hyperloop in a new or existing Titanium project, you will need to install the Hyperloop module and plugin. You can download the distribution (the distribution includes support for both Android and iOS) from https://s3-us-west-2.amazonaws.com/appc-labs-server/downloads/hyperloop-1.0.1b.zip.

You should extract the zip file at the root level of your Titanium (or Alloy) project. Once extracted, you will need to add the following to your `tiapp.xml`:

Configure the plugin

```
<plugins>
    <plugin>hyperloop</plugin>
</plugins>
```

Configure the module

```
<modules>
    <module>hyperloop</module>
</modules>
```

Configure for iOS

For iOS, you'll need to add the following two elements as children to the `ios` element:

```
<ios>
    <run-on-main-thread>true</run-on-main-thread>
    <use-jscore-framework>true</use-jscore-framework>
</ios>
```

Classes

Overview

Classes in Hyperloop map to the underlying classes defined in Objective-C. For example, if you have a class such as `UIView` defined in the `UIKit` framework, you would reference it using a standard require such as:

```
var UIView = require('UIKit/UIView');
```

This will return the `UIView` class object. Meaning, it's not an instance of a `UIView`, but the `UIView` class itself (or in Objective-C parlance, the interface defined with `@interface`).

Once you have a the Class reference returned from `require`, you can call normal JavaScript property and functions against it. Remember, at this point calling functions or properties against the class object above will be accessing Class level Objective-C methods (not instance level).

For example, you could get the `layerClass` of the `UIView` using the example:

```
var layerClass = UIView.layerClass;
```

This is because `layerClass` is defined as a Class method.

Instantiation

To instantiate a native Class and create an instance, you can use the normal `alloc init` style pattern from Objective-C:

```
var view = UIView.alloc().init();
```

Or, to simplify and make it more standard JavaScript convention, use `new` such as:

```
var view = new UIView();
```

This is the equivalent of the `alloc init` example above. When constructing an instance using `new`, it will always call the default initializer that is designated as `init`.

If you have a special initializer that takes arguments, you can use the following as you would in Objective-C:

```
var view = UIView.alloc().initWithFrame(CGRectMake(0,0,100,100));
```

Methods and Properties

Methods is Objective-C are mapped to JavaScript functions. Properties in Objective-C are mapped to JavaScript property accessors.

For example:

```
@interface UIView : UIControl
@property UIColor * backgroundColor;
-(void)addSubview:(id)view;
@end
```

Would map to the following in JavaScript:

```
view.backgroundColor = UIColor.redColor();
view.addSubview(label);
```

Named methods

If you have a selector with multiple parameters, the name of the function will be slightly different since JavaScript cannot receive multiple parameters as part of a function call. For example, to send a message with the selector `addAttribute:value:range:` you would use the function named: `addAttributeValueRange` instead. Hyperloop will camel case each receiver name in the selector and remove the `:` character to formulate the name of the method. Arguments should then be passed to the function in the same order as you would in Objective-C.

Constants, Enumerations and Functions

Constants, enumerations and functions defined in the Framework are available in the Framework package. For example, to reference the enum UISemanticContentAttribute you would reference it such as:

```
var UISemanticContentAttributeUnspecified = require('UIKit').UISemanticContentAtt
ributeUnspecified;
view.semanticContentAttribute = UISemanticContentAttributeUnspecified;
```

The constants, enumerations and functions will be read-only properties of the `UIKit` framework.

Casting

Sometimes interfaces define generic return types such as `NSObject` or `id` and you will need to cast them to a different type to then reference methods and properties of the class. You can use the special class function `cast` on any Class to return a casted object.

For example, suppose the result of the function returned an `id` but you know the implementation is actually a `UIView`. You could use the following:

```
var view = UIView.cast(object);
view.backgroundColor = UIColor.redColor();
```

Be careful with casting: If you cast an object which is actually something different, you will experience an error and likely a crash.

You can also cast a Titanium UI Component into its equivalent. For example, this would work:

```
var tiView = Ti.UI.createView( { backgroundColor : "red" } );
var nativeView = UIView.cast(tiView);
console.log('color should be red', nativeView.backgroundColor);
```

Blocks

Blocks in Hyperloop are translated into JavaScript functions.

For example, to animate a view which normally takes a block:

```
UIView.animateWithDurationAnimationsCompletion(1.0, function () {
   view.layer.opacity = 0.0;
}, function (done) {
});
```

Function pointers

Currently, function pointers are not currently supported in the latest version.

Creating your own classes

Hyperloop provides you the ability to dynamically create your own Objective-C classes at runtime. Once created, this classes can be used as normal in either Hyperloop or passed to native calls.

Let's create a simple custom `UIView`:

```
var MyView = Hyperloop.defineClass('MyClass', 'UIView');
```

This will create a new class in the Objective-C runtime named `MyClass` which will extend `UIView` which is equivalent to the following code:

```
@interface MyClass : UIView
@end
```

You can also pass an Array or String as the third argument which are the protocols to implement for the new class.

You can now add methods:

```
MyView.addMethod({
   selector: 'drawRect:',
   instance: true,
   arguments: ['CGRect'],
   callback: function (rect) {
     // this code is executed when drawRect: is called
   }
});
```

Hyperloop supports the following set of properties for adding methods:

- `arguments` can be either an Array or String of argument types (which can be either Objective-C encoding types or general type names such as `float` or `int`).
- `returnType` can be a String return type (which can be either Objective-C encoding types or general type names such as `float` or `int`). If no return is required (a `void` return type), you can omit the `returnType` property altogether and `void` will be implied.

Another example with multiple arguments using simplified types:

```
MyView.addMethod({
   selector: 'foo:bar:hello:',
   instance: true,
   returnType: 'void',
   arguments: ['int', 'float', 'id'],
   callback: function (a, b, c) {
   }
});
```

Once you have defined your class, you would just instantiate it as normal.

```
var myview = new MyView();
```

Using Third-party libraries

You can use Third-party libraries in Hyperloop as if they were APIs defined in Cocoa.

CocoaPods

Hyperloop supports CocoaPods as a way to manage Third-party dependencies in your Hyperloop enabled project.

You must first install CocoaPods if you do not already have it installed. You can install using:

```
sudo gem install cocoapods
```

Once you have CocoaPods installed you can create a Podfile in your Titanium project directory such as:

```
platform :ios, '7.0'
target 'MyProject' do
end
```

Note that `MyProject` should be the name of your Titanium project.

Once you have a `Podfile`, you can add dependencies. For example:

```
platform :ios, '7.0'
target 'Hyperloop_Sample' do
    pod 'JBChartView'
end
```

Which will import the JBChartView framework as a dependency.

That's it! The Hyperloop compiler will do the rest – managing pulling down the required dependencies, compiling them and integrating them into the Xcode build.

Let's now use the imported project:

```
var JBBarChartView = require('JBChartView/JBBarChartView');
var chart = new JBBarChartView();
chart.minimumValue = 1;
chart.maximumValue = 100;
```

Custom

In addition to CocoaPods, you can include third-party or first-party custom code by including a reference in `appc.js` under the `thirdparty` property.

For example, to include custom objective-c from the project's `src` directory you could provide:

```
module.exports = {
    hyperloop: {
        ios: {
            thirdparty: {
                'MyFramework': {
                    source: ['src'],
                    header: 'src',
                    resource: 'src'
                }
            }
        }
    }
};
```

- The `source` property can be either an Array or String of source directories to include in the compile. This is optional.
- The `header` property can be either an Array or String of header directories to include in the compile. This is required.
- The `resource` property can be either an Array or String of resource directories to search for resources or files to compile (images, story boards, xibs, etc). These files will be flattened and copied directly into the root of the application and can be loaded with

`NSBundle`.

To provide additional compiler flags, you can add them in the `xcodebuild` property of `ios` such as:

```
module.exports = {
    hyperloop: {
        ios: {
            xcodebuild: {
                flags: {
                    LIBRARY_SEARCH_PATHS: 'src',
                    OTHER_LDFLAGS: '-lMyLibrary'
                }
            },
            thirdparty: {
                'MyFramework': {
                    source: ['src'],
                    header: 'src',
                    resource: 'src'
                }
            }
        }
    }
};
```

Using Swift

In addition to Objective-C, you can import third-party libraries written in Swift. Any `*.swift` files found in your `source` directories will automatically be compiled and available to use in your JavaScript just like Objective-C APIs.

Customizing your Xcode build

You can further customize the Xcode build by defining an `appc.js` file in the root of your project. This project will provide some additional configuration which the Hyperloop compiler

will read when invoking Xcode.

For example:

```
module.exports = {
    hyperloop: {
        ios: {
            xcodebuild: {
                flags: {
                    GCC_PREPROCESSOR_DEFINITIONS: 'foo=bar'
                },
                frameworks: [
                    'StoreKit'
                ]
            }
        }
    }
};
```

Any `flags` added to the `xcodebuild` property will be passed to `xcodebuild`. Any `frameworks` in the array provided will be automatically added to the xcode project. *Note: any referenced frameworks in your Hyperloop code are automatically added for you. However, this gives you even more control to custom your compile environment.*

Adding a third-party framework

If you'd like to add a third-party framework which isn't distributed with source code or available in CocoaPods, you can use this example configuration:

```
module.exports = {
    hyperloop: {
        ios: {
            xcodebuild: {
                flags: {
                    FRAMEWORK_SEARCH_PATHS: '/path/to/framework'
                },
                frameworks: [
                    'FrameworkName'
                ]
            }
        }
    }
};
```

In the above example, `/path/to/framework` should point the the location of the `.framework` file and `FrameworkName` should be the name of the Framework.

Resources

Any Xcode resources that are found in your `Resources` or `app` (for Alloy projects) will be automatically be compiled and/or copied to your application root directory. Any intermediate folders will be flattened automatically.

The following resources will be compiled automatically:

- Storyboard (*.storyboard)
- XC Data Model (*.xcdatamodel, *.xcdatamodeld, *.xcmappingmodel)
- Interface Builder (*.xib)

The other resources will be copied such as PNG files.

It is recommended a Best Practice that you place any platform specific files under `Resources/iphone` (Titanium classic) or `app/assets/iphone` (Alloy). *You can also use `ios` instead.*