NativeScript iOS Runtime

Deep dive

Agenda /red color means “not covered yet”/

1. iOS Runtime Intro
    a. Repositories
    b. Build Dependencies
    c. Processes
    d. Buildable Artefacts
2. Debugging Issues in Client Apps
    a. Metadata Related Issues
    b. Runtime Related Issues
3. What Does the Runtime Do on Application Start
4. Extending Native Classes
5. Data Marshaling

Agenda (2)

6. Garbage Collection and memory management
7. Exception Handling
8. LiveSync in the Context of the iOS Runtime
9. Debugging. Debugging Protocol.
10. Built-in JSC vs custom JSC build
11. Other

Feel free to contact the iOS Runtime team to add more bullets in the list. For example: Libffi Deep Dive, JavaScriptCore Deep Dive, Metadata Generator Deep Dive

iOS Runtime Intro

Repositories

1. ios-runtime
   a. ios-metadata-generator
   b. webkit - fork of the unofficial mirror of the WebKit repo in github
      i. JavaScriptCore
      ii. WebInspectorUI User Interface
   c. libffi - a portable foreign-function interface library
2. ios-runtime-docs

Build dependencies

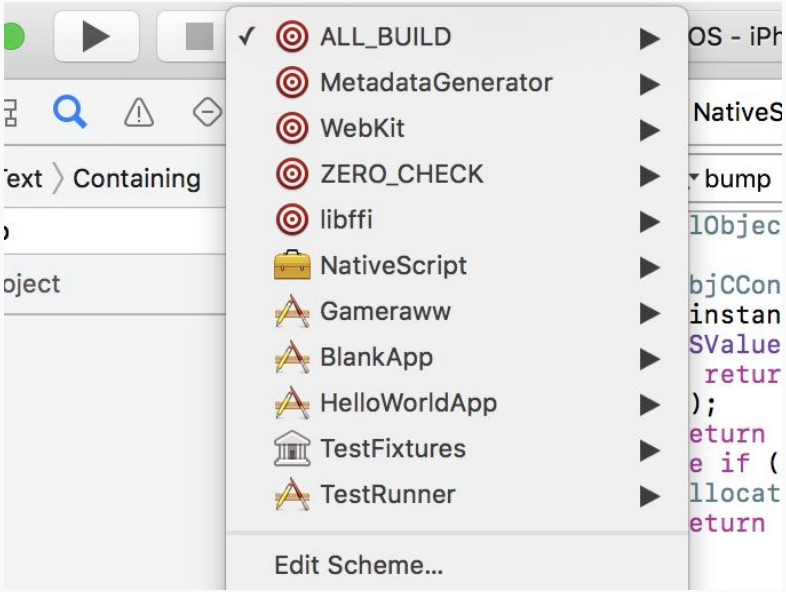
1. OS X 10.11+
2. Xcode 8+
3. CMake 3.1.3 with command line tools included
    a. We are currently stuck to using CMake 3.1.3 because the version of JSC we are using is not buildable with newer CMake versions. Once the WebKit guys introduce support for newer CMake and only after we update the WebKit/JSC, we can use newer version of CMake.
4. llvm 3.9 + Clang - linked in the metadata-generator
5. Automake - available in Homebrew as automake.
6. GNU Libtool - available in Homebrew as libtool.

Processes

1. Updating to newer JavaScriptCore version
    a. Updating the Webkit fork
    b. Updating the WebInspectorUI
    c. Possible improvements of the process?
2. Contributing
3. Release process
    a. Update the release branch
    b. Bump versions of the WebInspector package and the runtime package
    c. Wait for the packages to be released
    d. Create a release in github with appropriate tag
    e. Update the CHANGELOG.md
    f. Merge release branch back to master

Buildable Artefacts

1. Metadata Generator
2. WebKit
3. libffi
4. NativeScript
    a. Static library
    b. Dynamic framework
5. Gameraww App
6. BlankApp
7. HelloWorldApp
8. TestFixtures
9. TestRunner



Debugging Issues in Client Apps

Debugging Metadata Issues

1. Inspect the artefacts produced by the metadata generator at *platforms/ios/build/emulator* or *platforms/ios/build/device*

    a. *metadata-generation-stderr-{arch}.txt* – content logged by the MG on the std error stream
    b. *metadata-{arch}.bin* – binary metadata which is later embedded in the app binary as a __DATA section
    c. *umbrella-{arch}.h* – the umbrella header used as the only input of the AST generation API. It contains an *#import* clause for every single header in the header search paths which is part of a clang module

Debugging Metadata Issues (2)

2. Generate additional artefacts for more advanced debugging

    a. Yaml metadata:
    TNS_DEBUG_METADATA_PATH="$(pwd)/app/yaml" tns build ios (the path must exist)
    b. TypeScript declarations:
    TNS_TYPESCRIPT_DECLARATIONS_PATH="$(pwd)/app/typings" tns build ios
    c. Modulemap files for every clang module included in the metadata
    -output-modulemaps flag of the metadata generator

Debugging Metadata Issues (3)

3. Debugging metadata generator:

    - Get all arguments passed to the MG from the app build log dumped on the command line
    - Build the MG from source (Xcode/CLion)
    - Run the MG with attached debugger with the exact same arguments passed on app build

Debugging The iOS Runtime in Existing App

Video Demo: Debug the iOS Runtime in existing app

1. Make sure the native app template is ready for build
   a. Run *tns prepare ios*
   b. Some additional steps if any(in most of the cases there is no such steps)
2. Open the native app project in Xcode. You can find it at *platforms/ios/{AppName}.xcodeproj*. In case of CocoaPods are used - *platforms/ios/{AppName}.xcworkspace*.
3. Generate Xcode project for the ios runtime

```
mkdir "cmake-build" && cd "cmake-build"
cmake .. -G "Xcode" -D"BUILD_SHARED_LIBS=ON"
```

Debugging The iOS Runtime in Existing App

4. Add *NativeScript.xcodeproj* as subproject of the app project (drag & drop)

5. Link the app target against the NativeScript product of the subproject instead of the pre-built framework in the platforms folder

   A. Remove *–framework NativeScript* linker flag from the app build settings

   B. Add *NativeScript.framework* in the *Link Binary With Libraries* list of the app target

Application start

What does the runtime do on application start

1. Initialize the metadata - since the metadata is embedded as a __DATA section in the app binary it is memory-mapped by the system. This means that loading it is pretty fast operation.
2. Initialize the runtime
    a. Create and initialize the JavaScript VM
    b. Create and initialize the Global Object
        i. Create and cache internal structures
        ii. Attach JS APIs to the Global Object provided by the runtime: __collect, __extends, require, __runtimeVersion, __onUncaughtError, WeakRef and Worker constructors etc
        iii. Executes the so-called inline functions
        iv. Executes the TS extend

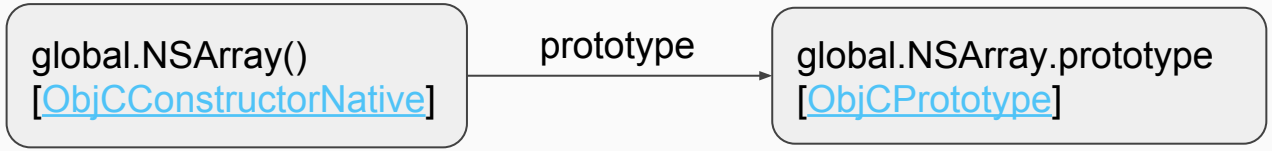
What does the runtime do on application start (2)

3. Schedule the VM tasks to be queued on the system event loop
4. Initialize the inspector (only in debug)
5. Executes the entry module

Extending Native Classes

Objective-C Classes

- Objective-C class is exposed as JavaScript class - a pair of JavaScript constructor function and a prototype object.



- The NSObject constructor has an extend function
- Exposing Objective-C methods:
  - Static - on the JavaScript constructor function (ObjCConstructorNative.getOwnPropertySlot())
  - Instance - on the JavaScript prototype object (ObjCPrototype.getOwnPropertySlot())
- Objective-C properties are exposed as JavaScript property descriptors. Properties are materialized eagerly. (ObjCPrototype.materializeProperties())

Objective-C Classes

- The prototype chain of the JavaScript objects matches the inheritance chain of the represented Objective-C classes.
- alloc, init or new

```
var view1 = UIView.alloc().init();
// Or with the short-cut
var view2 = UIView.new();
```

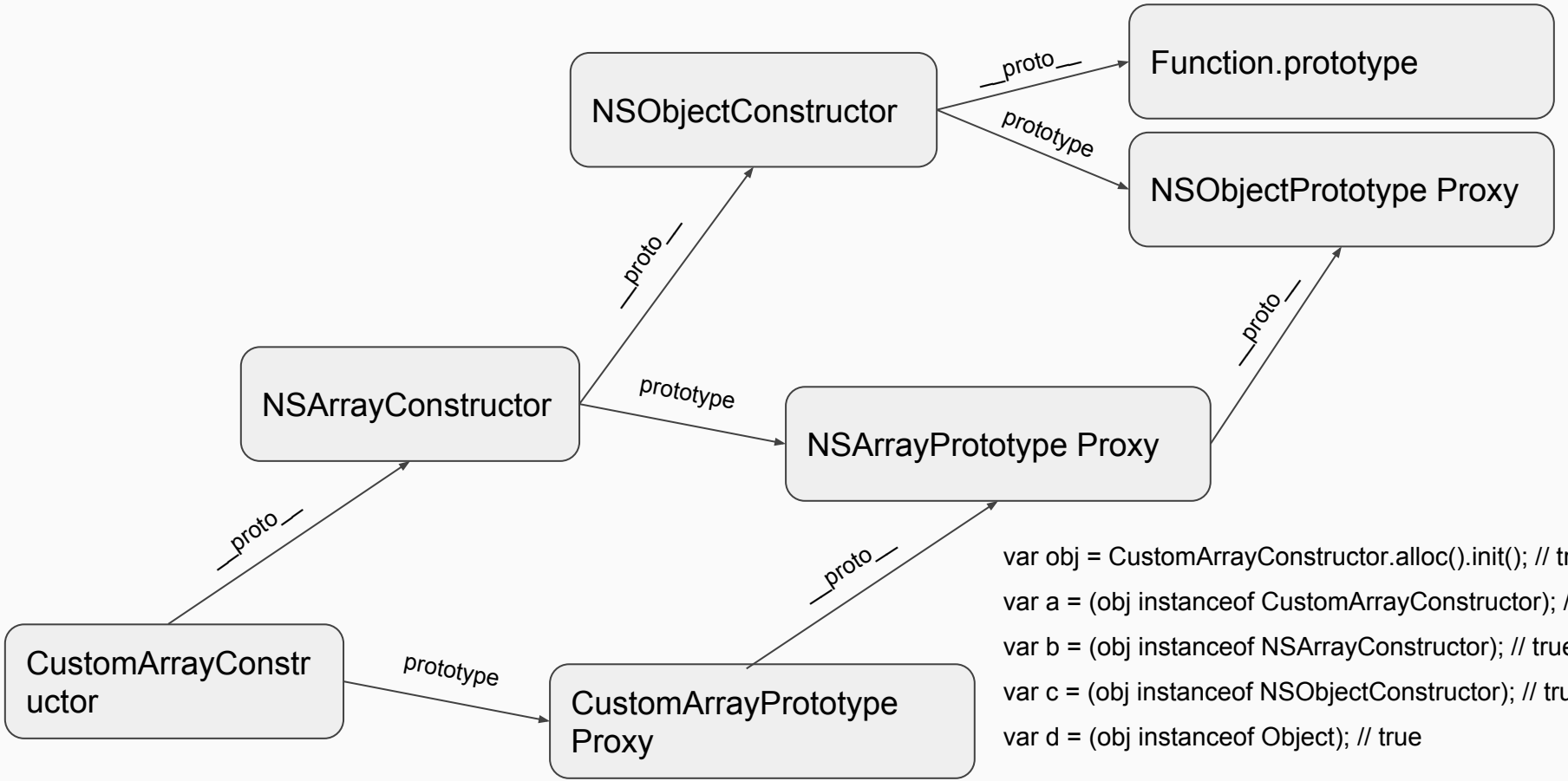
- JavaScript new operator
  - Called with constructor parameters - will try to match an appropriate initializer based on the number and types of the arguments.

```
var view1 = new UIView(); // Will call UIView.alloc().init();
var view2 = new UIView(CGRectMake(10, 10, 200, 100)); // Will call UIView.alloc().initWithFrame(...)
```

  - Called with object literal (Swift-style initializers)

```
NSURL.alloc().initWithString(aString) becomes new NSURL({ string: aString })
NSURL.alloc().initFileURLWithPath(aString) becomes new NSURL({ fileURLWithPath: aString })
```

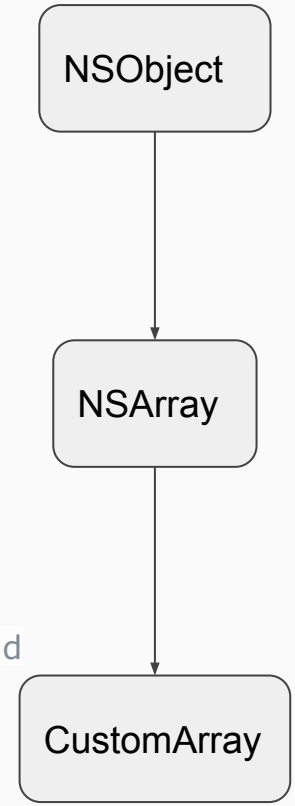
Prototype Chain




```
var obj = CustomArrayConstructor.alloc().init(); // true
var a = (obj instanceof CustomArrayConstructor); // true
var b = (obj instanceof NSArrayConstructor); // true
var c = (obj instanceof NSObjectConstructor); // true
var d = (obj instanceof Object); // true
```

Prototype Chain

```
function NSObject() { /* native call */ };
// Object.getPrototypeOf(NSObject) === Function.prototype
NSObject.alloc = function () { /* native call */ };

// Object.getPrototypeOf(NSObject.prototype) === Object.prototype
NSObject.prototype.init = function () { /* native call */ };

function NSArray() { /* native call */ };
Object.setPrototypeOf(NSArray, NSObject);
NSArray.arrayWithObjectsCount = function () { /* native call */ };

NSArray.prototype = Object.create(NSObject.prototype, { constructor: NSArray});
NSArray.prototype.lastObject = function () { /* native call */ };

function CustomArray() { /* native call */ };
Object.setPrototypeOf(CustomArray, NSArray);
CustomArray.customStaticMethod = function () { }; // you can't override static methods and
properties

CustomArray.prototype = Object.create(NSArray.prototype, { constructor: CustomArray});
CustomArray.prototype.customInstanceMethod = function () { /* native call */ };
```



Subclassing Objective-C Classes

```
var <DerivedClass> = <BaseClass>.extend(classMembers, [nativeSignature]);
```

- `classMembers`
  - methods - define or override instance methods
  - properties - define or override instance properties
- `nativeSignature`
  - `name` - optional, string with the derived class name
  - `protocols` - optional, array with the implemented protocols
  - `exposedMethods` - optional, dictionary with method names and native method signature objects
- `ObjCExtendFunction`, `ObjCClassBuilder`
- Subclass Example

Subclass Example

```
var MyViewController = UIViewController.extend({
    // Override an existing method from the base class.
    // We will obtain the method signature from the protocol.
    viewDidLoad: function () {
        // Call super using the prototype:
        UIViewController.prototype.viewDidLoad.apply(this, arguments);
        // or the super property:
        this.super.viewDidLoad();

        // Add UI to the view here...
    },
    shouldAutorotate: function () { return false; },

    // You can override existing properties
    get modalInPopover() { return this.super.modalInPopover; },
    set modalInPopover(x) { this.super.modalInPopover = x; },

    // Additional JavaScript instance methods or properties that are not accessible from Objective-C code.
    myMethod: function() { },

    get myProperty() { return true; },
    set myProperty(x) { },
}, { name: "MyViewController"});
```

Protocol Implementation Example

- You can implement only some methods of the protocol. If a not implemented method is called an exception will be raised at runtime.

```
var MyAppDelegate = UIResponder.extend({
    // Implement a method from UIApplicationDelegate.
    // We will obtain the method signature from the protocol.
    applicationDidFinishLaunchingWithOptions: function (application, launchOptions) {
        ...
    }
}, {
    // The name for the registered Objective-C class.
    name: "MyAppDelegate",
    // Declare that the native Objective-C class will implement the UIApplicationDelegate Objective-C
protocol.
    protocols: [UIApplicationDelegate]
});
```

Exposed Method Example

Runtime Types

```
var MyViewController = UIViewController.extend({
    viewDidLoad: function () {
        // ...
        var aboutButton = UIButton.buttonWithType(UIButtonType.UIButtonTypeRoundedRect);
        // Pass this target and the aboutTap selector for touch up callback.
        aboutButton.addTargetActionForControlEvents(this, "aboutTap", UIControlEvents.UIControlEventTouchUpInside);
        // ...
    },
    // The aboutTap is a JavaScript method that will be accessible from Objective-C.
    aboutTap: function(sender) {
        var alertWindow = new UIAlertView();
        alertWindow.title = "About";
        alertWindow.addButtonWithTitle("OK");
        alertWindow.show();
    },
}, { name: "MyViewController",
    exposedMethods: {
      // Declare the signature of the aboutTap. We can not infer it, since it is not inherited from base class or protocol.
        aboutTap: { returns: interop.types.void, params: [ UIControl ] }
    } });
```

TypeScript Support

```
// A native class with the name "JSObject" will be registered, so it should be unique
class JSObject extends NSObject implements NSCoding {
    public encodeWithCoder(aCoder) { /* ... */ }

    public initWithCoder(aDecoder) { /* ... */ }

    public "selectorWithX:andY:"(x, y) { /* ... */ }

    // An array of protocols to be implemented by the native class
    public static ObjCProtocols = [ NSCoding ];

    // A selector will be exposed so it can be called from native.
    public static ObjCExposedMethods = {
        "selectorWithX:andY:": { returns: interop.types.void, params: [ interop.types.id, interop.types.id ] }
    };
}
```

- Custom TypeScript __extend function

Data Marshaling

Data Marshaling

- Objective-C Class <-> JS Constructor Function with an associated prototype
- Instances of Objective-C classes <-> Special “wrapper” objects
  - There is only one JavaScript wrapper around an Objective-C object, always. This means that Objective-C wrappers maintain JavaScript identity equality:

    ```
    tableViewController.tableView === tableViewController.tableView
    ```
  - Primitive Exceptions:
    - `NSNull <-> null`
    - `NSNumber <-> number or boolean`
    - `NSString <-> string`
    - `NSDate <-> Date`
  - Primitive Exceptions. Exceptions :)
    The exception to this are the methods on those classes declared as returning `instancetype` - init methods and factory methods e.g. `NSString.stringWithString` will return a wrapper around an `NSString` instance, rather than a JavaScript string.

Data Marshaling

- Objective-C Protocols <-> JS Objects with an associated prototype

  ObjC: `BOOL isCopying = [NSArray conformsToProtocol:@protocol(NSCopying)];`

  JavaScript: `var isCopying = NSArray.conformsToProtocol(NSCopying);`
- Objective-C selector <-> JavaScript string

  ObjC: `[aString respondsToSelector:@selector(appendString:)];`

  JavaScript: `aString.respondsToSelector("appendString:");`
- Objective-C Block <-> JavaScript function

Data Marshaling

- CoreFoundation Toll-Free Bridged Types
  - CFDictionaryRef <-> NSDictionary
  - CFArrayRef <-> NSArray
  - CFStringRef <-> NSString
- Numeric Types
  - char, int, long, float, double, NSInteger and their unsigned variants <-> JavaScript number
  - Integer values larger than ±2^53 will lose their precision because the JavaScript number type is limited in size to 53-bit integers.
- Struct Types <-> JavaScript wrapper objects (example)
  - For each structure there exists a constructor object with the name of the structure (more)
- Enums

Data Marshalling

- interop.Pointer and interop.Reference
  - interop.alloc(size: number) // returns a pointer that will free the memory when garbage collected
  - interop.free(ptr: interop.Pointer); // releases the memory of a pointer
  - interop.sizeof(type: any); // returns the size of the provided type.
  - types

```
var nsstring = NSString.stringWithString("test");

// Calls the native C malloc function. You must call free when finished using it.
var buffer = malloc(4 * interop.sizeof(interop.types.unichar)); // interop.Pointer

nsstring.getCharacters(buffer); // Fill the buffer

// Reinterpret the void* buffer as unichar*. The reference variable doesn't retain the allocated buffer.
var reference = new interop.Reference(interop.types.unichar, buffer);
console.log(reference[0], reference[1], reference[2], reference[3]); // "t" "e" "s" "t"

free(buffer); // Same as interop.free(buffer)
```

Garbage Collection and memory management

Garbage Collection and memory management

- global.__collect() - triggers garbage collection
- ARC is enabled in NativeScript iOS applications
- JS wrapper objects have a strong reference to their native counterparts
- Custom release and retain methods

Exception Handling

NSErrors and NSExceptions

- NSErrors. To catch or not to catch.
  - Receive NSError in out parameter

```
var errorRef = new interop.Reference();
fileManager.contentsOfDirectoryAtPathError('/not-existing-path', errorRef);
console.log(errorRef.value); // NSError: "The folder '/not-existing-path' doesn't exist."
```

  - Try/catch NSErrors

```
try {
    fileManager.contentsOfDirectoryAtPathError('/not-existing-path');
} catch (e) {
    console.log(e); // NSError: "The folder '/not-existing-path' doesn't exist."
}
```

  - NSExceptions - __onUncaughtError

LiveSync in the Context of the iOS Runtime

- TODO...