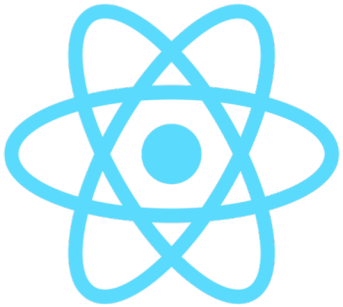


Reusable UI Components in

@ray_deck

element55

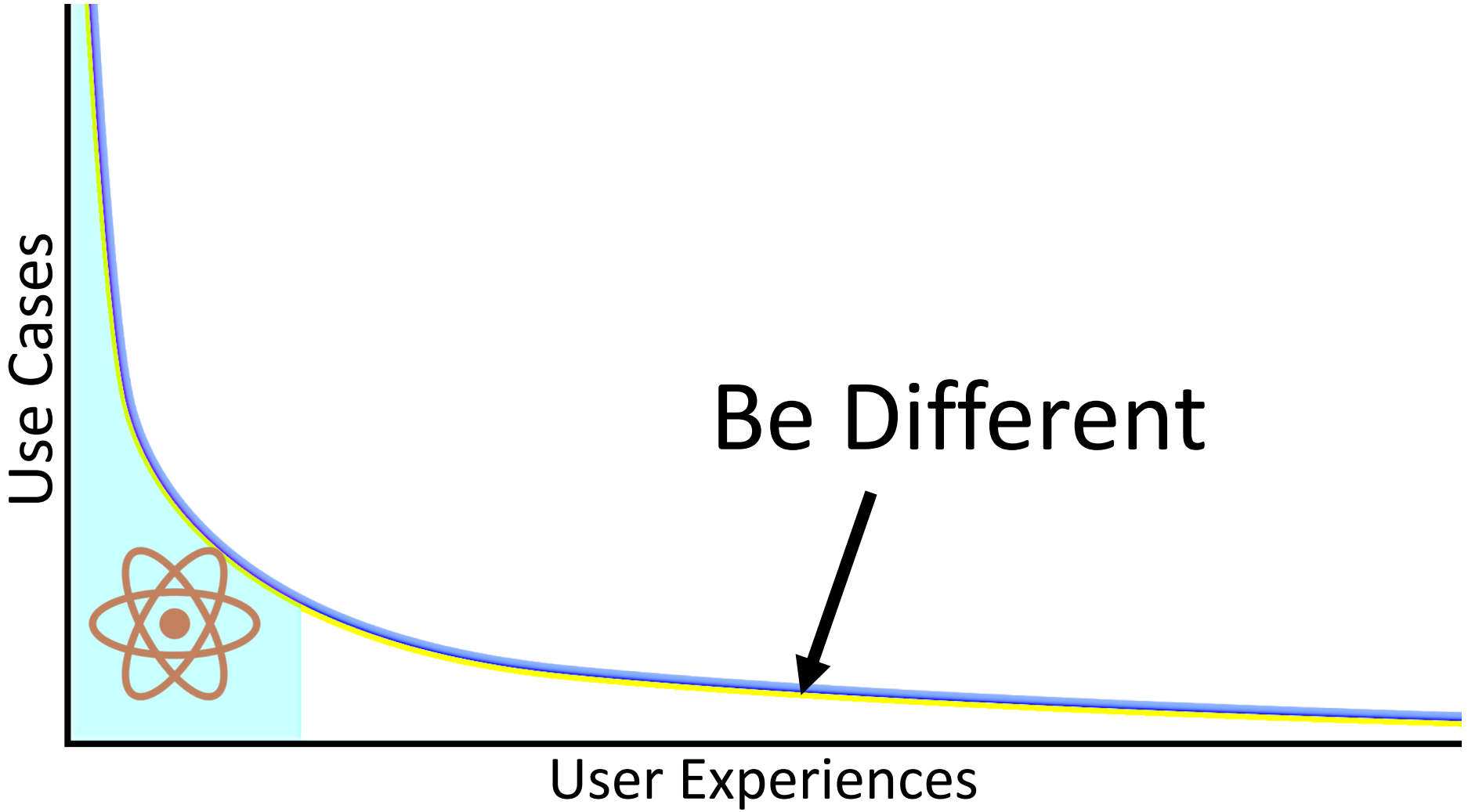
Why Would I Ever Do This?

Why Build a UI Component?

1. **Deep Cut**
   Specific need in an otherwise-available component (visionkit)

2. **Native Capabilities**
   Access to previously-unavailable or new subsystems (AR, VR)

3. **Third Party Integration**
   External functionality to orchestrate through RN (CocoaPods)


Use Cases
Be Different
User Experiences

Familiar

Terse

Type-Safe

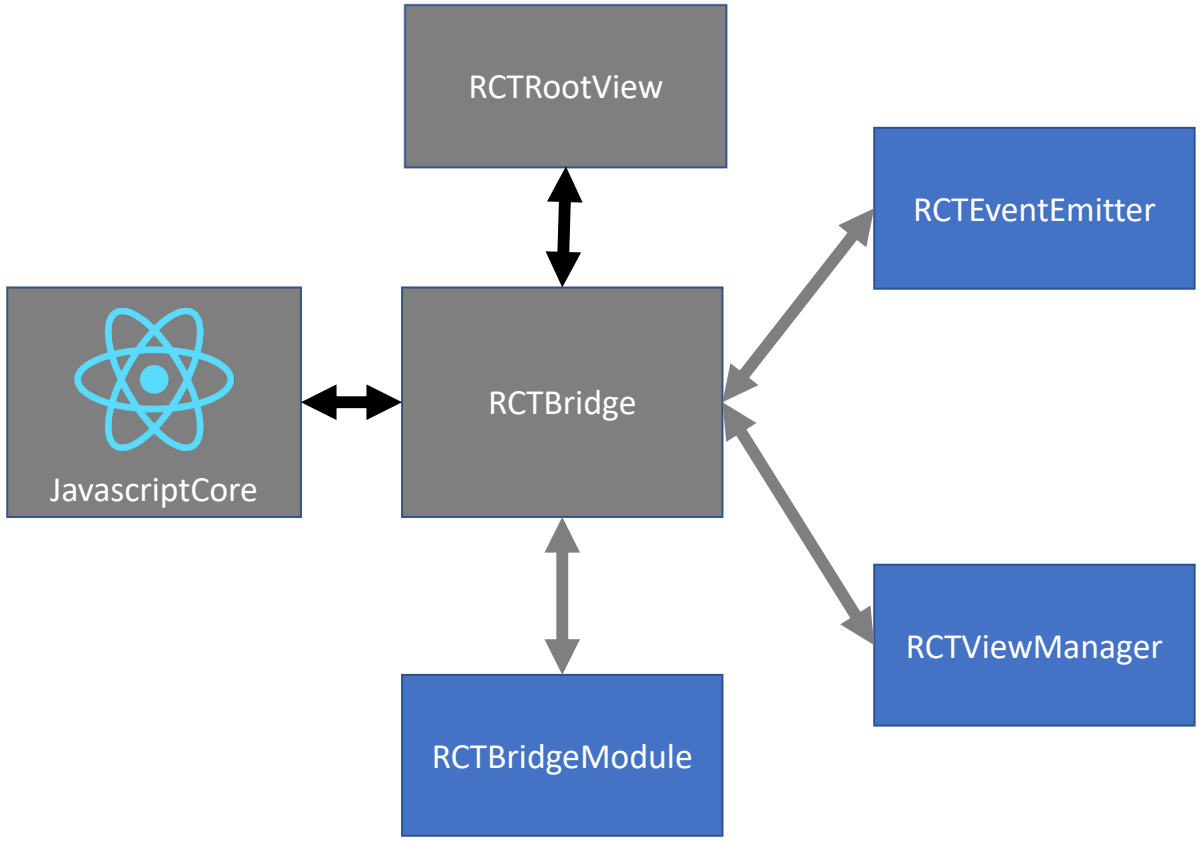
First-Class Support from Apple

Mature (v4.0)

Static Library Support (Xcode 9.0)

react-native-swift

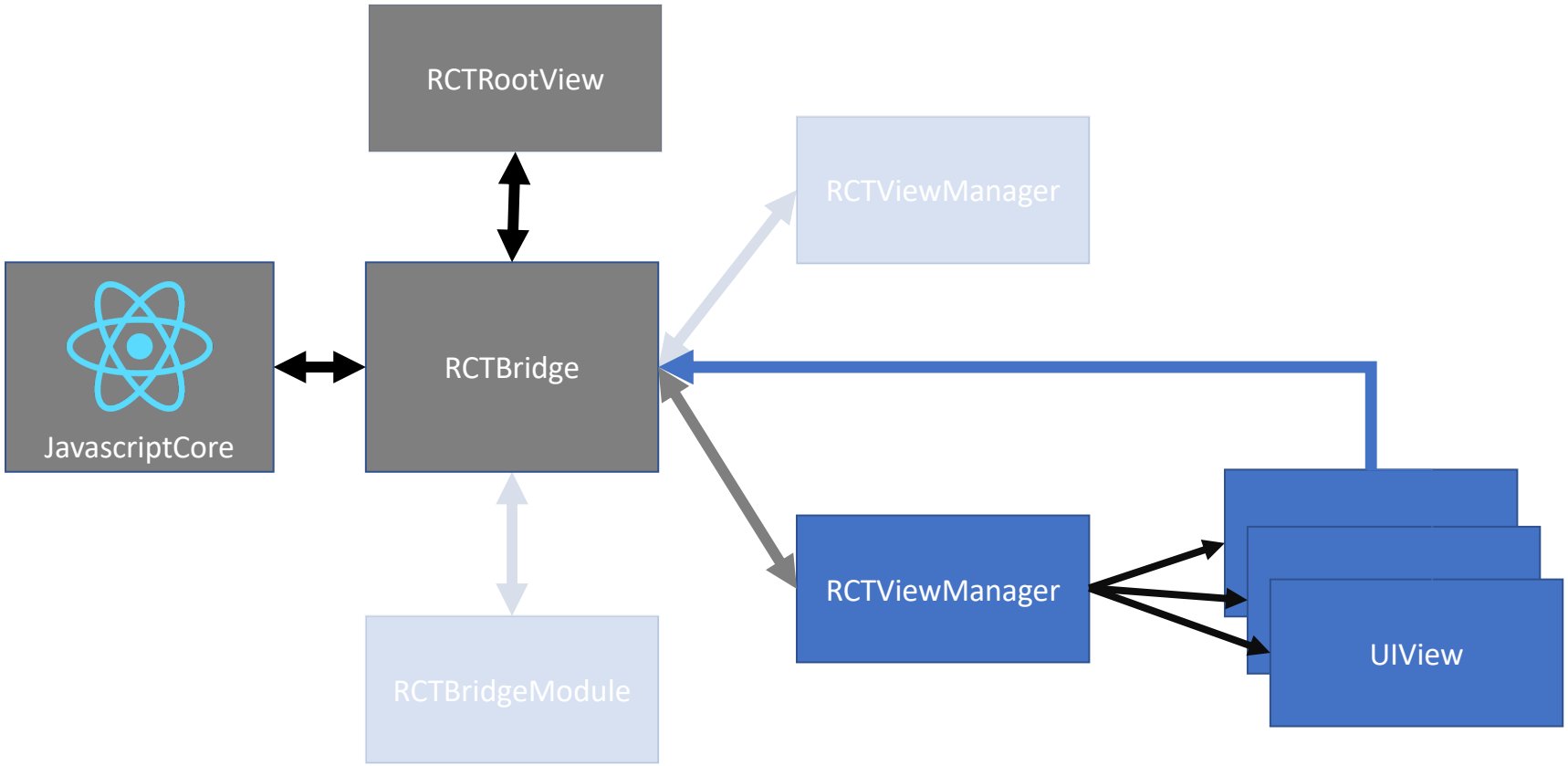
yarn add react-native-swift


RCTRootView
RCTEventEmitter
JavascriptCore
RCTBridge
RCTViewManager
RCTBridgeModule

```
#import <React/RCTViewManager.h>
#import <React/RCTEventEmitter.h>
#import <React/RCTBridgeModule.h>
@interface RCT_EXTERN_MODULE(RNSBoston, RCTEventEmitter)
RCT_EXTERN_METHOD(demo:(NSString *)message
success:(RCTPromiseResolveBlock)success
reject:(RCTPromiseRejectBlock)reject);
RCT_EXTERN_METHOD(delayedSend:(NSString *)message
ms:(NSInteger)ms);
@end
@interface RCT_EXTERN_MODULE(RNSBostonBasicViewManager,
RCTViewManager)
@end
@interface RCT_EXTERN_MODULE(RNSBostonViewManager,
```

Takeaways

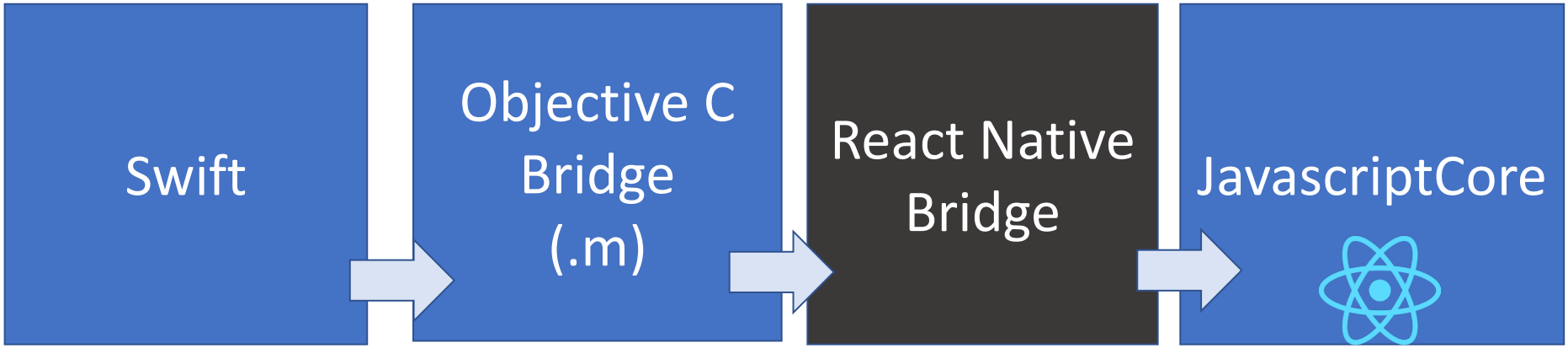
1. The RCTBridge is the core.
2. There is one bridge module instance per class per bridge
3. Modules are exposed to the bridge using objective-C macros (RCT_EXPORT_MODULE, RCT_EXTERN_MODULE, etc)


RCTRootView
RCTViewManager
JavascriptCore
RCTBridge
RCTViewManager
UIView
RCTBridgeModule

Takeaways

1. RCTViewManagers are bridge modules, and follow these rules
2. Views are generated from the ViewManagers
3. RN will control layout and lifecycle of views emitted from ViewManagers

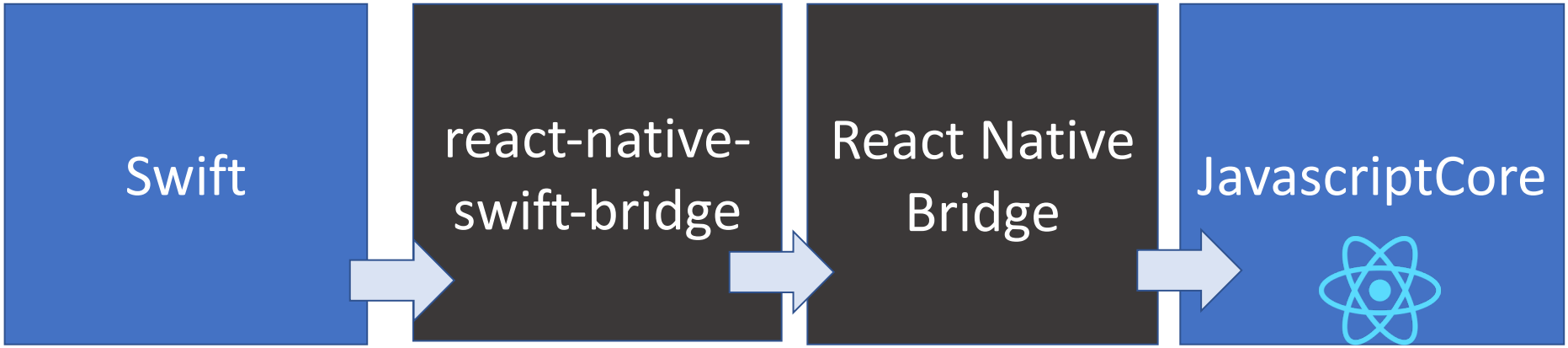
A Winding Road



react-native-swift-bridge

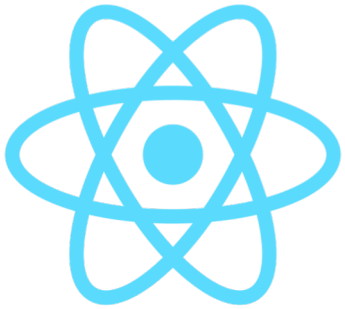
react-native-swift-bridge --watch

A Less-Winding Road



react-native-swift-cli

yarn global add react-native-swift-cli



1. Start with templates from **react-native-swift-cli**
2. Open your app as project for editing
3. Edit or add files in your static library – not the app proper
4. Use **yarn watch** to have your bridge keep up

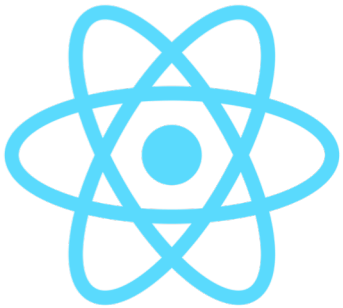
>_

```
yarn global add react-native-swift-cli

rns init RNBoston ; rns makeapp RNBostonApp RNBoston

cd RNBoston; code .

cd ../RNBostonApp; code . ; react-native xcode
```

```swift
import Foundation
@objc(RNBostonBasicViewManager)
class RNBostonBasicViewManager: RCTViewManager {
    //MARK: RCTViewManager key methods
    override func view() -> UIView {
        let newView = UIView()
        newView.backgroundColor = UIColor.green
        return newView
    }
    override class func requiresMainQueueSetup() -> Bool {
        return false;
    }
}
```

Takeaways

1. **@objc** attribute for code we want to expose to React native
2. **view()** is the only method that really matters
3. requiresMainQueueSetup() should return **false**



```
import { requireNativeComponent } from "react-native";
import React, { Component } from "react";

const NativeComponent = requireNativeComponent(
  ”RNBostonBasicView",
  BasicView
);

class BasicView extends Component {
  render() {
    return <NativeComponent {...this.props} />;
  }
}

export default BasicView;
```

```
import { BasicView } from “RNBoston”
…
<View style={{height: 90,…}} >
  <View style={{ height: 40 }}>
    <Text>
      Starting with a basic native view. That's the green
thing. Pretty boring.
    </Text>
  </View>
  <BasicView style={{ height: 50, width: "50%" }} />
</View>
```

Takeaways

1. **requireNativeComponent** exposes the native view for a React component wrapper
2. You must create a React component that has the specific job of wrapping the native view
3. React is kind of awesome

we can do better

```swift
import Foundation
@objc(RNBostonViewManager)
class RNBostonViewManager: RCTViewManager {

    var currentView:RNBostonView?

    //MARK: RCTViewManager key methods
    override func view() -> RNBostonView {
        let newView = RNBostonView()
        currentView = newView
        return newView
    }
    override class func requiresMainQueueSetup() -> Bool {
        return false;
    }
```

Takeaways

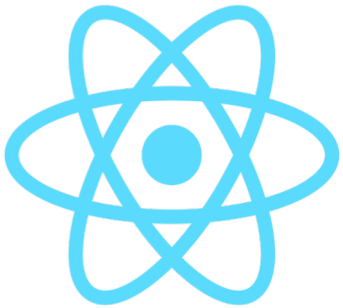
1. Using a custom view class to expose props and manage a nontrivial UX
2. Connect your view to the view manager through a reference at creation time
3. Use Caution!

```swift
import UIKit
import AVKit
@objc(RNBostonView)
class RNBostonView: UIView {
  //MARK: Private (non-RN-managed) properties
  var thisSession = AVCaptureSession?
  var previewLayer = AVCaptureVideoPreviewLayer?
  var isFront:Bool = false
  //MARK: React-native exposed props
  @objc var onStart:RCTBubblingEventBlock?
  @objc var cameraFront:Bool {
    get { return isFront }
    set(b) {
      isFront = b
      AVCaptureDevice.requestAccess(forMediaType: AVMediaTypeVideo) { success in
        guard success else { return }
        guard
          let device = AVCaptureDevice.defaultDevice(withDeviceType: .builtInWideAngleCamera, mediaType:
AVMediaTypeVideo, position: b ? AVCaptureDevice.Position.front : AVCaptureDevice.Position.back),
          let input = try? AVCaptureDeviceInput(device: device)
          else { return }
        let s = AVCaptureSession()
        s.addInput(input)
        s.startRunning()
        guard let pl = AVCaptureVideoPreviewLayer(session: s)  else { return }
        DispatchQueue.main.async(){
          pl.frame = self.bounds
          pl.videoGravity = AVLayerVideoGravityResizeAspectFill
          self.layer.addSublayer(pl)
          self.previewLayer = pl
          if let o = self.onStart { o([:]) }
        }
      }
    }
  }
```

```swift
@objc(RNBostonView)
class RNBostonView: UIView {

  //MARK: Private (non-RN-managed) properties
  var thisSession = AVCaptureSession?
  var previewLayer = AVCaptureVideoPreviewLayer?
  var isFront:Bool = false

  //MARK: React-native exposed props
  @objc var onStart:RCTBubblingEventBlock?
  @objc var cameraFront:Bool {
    get { return isFront }
    set(b) {
      isFront = b
      AVCaptureDevice.requestAccess…
```

Takeaways

1. Properties, not methods, are exposed.
2. Properties can be simple “var” declarations or use get/set pattern
3. Cheat lifecycle events with prop setters
4. Function props are RCTBubblingEventBlocks
5. Declare RCTBubblingEventBlocks as optional (using ?)



```
import { requireNativeComponent } from "react-native";
import React, { Component } from "react";

const NativeVision = requireNativeComponent(”RNBostonView",
CameraView);

class CameraView extends Component {
  render() {
    return <NativeVision {...this.props} />;
  }
}
CameraView.defaultProps = {
  onStart: () => {
    console.log("I am starting for reals");
  },
  cameraFront: true
};

export default CameraView;
```

Takeaways

1. Still Easy
2. Default props make your life easier
3. React – still awesome

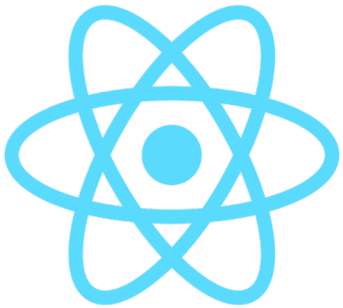
One More Thing

```swift
import Foundation
@objc(RNBostonViewManager)
class RNBostonViewManager: RCTViewManager {

    var currentView:RNBostonView?

    //MARK: RCTViewManager key methods
    override func view() -> RNBostonView {
        let newView = RNBostonView()
        currentView = newView
        return newView
    }
    override class func requiresMainQueueSetup() -> Bool {
        return false;
    }
```

```swift
@objc(RNBostonViewManager)
class RNBostonViewManager: RCTViewManager,
AVCapturePhotoCaptureDelegate {

…
    @objc func takePicture(_ resolve:@escaping
RCTPromiseResolveBlock, reject:RCTPromiseRejectBlock) {
        guard let view = currentView else { reject("no_view",
"No view loaded", nil); return }
        guard let session = view.thisSession else {
reject("no_session", "No AV capture session running", nil);
return }
        if let p = self.photoOutput {
            session.removeOutput(p)
            self.photoOutput = nil
        }
```

Takeaways

1. The ViewManager is a Native Module!
2. Add native module methods to add imperative logic to your view
3. Use promises (RCTPromiseResolveBlock and RCTPromiseRejectBlock) to return data
4. Give your app superpowers!

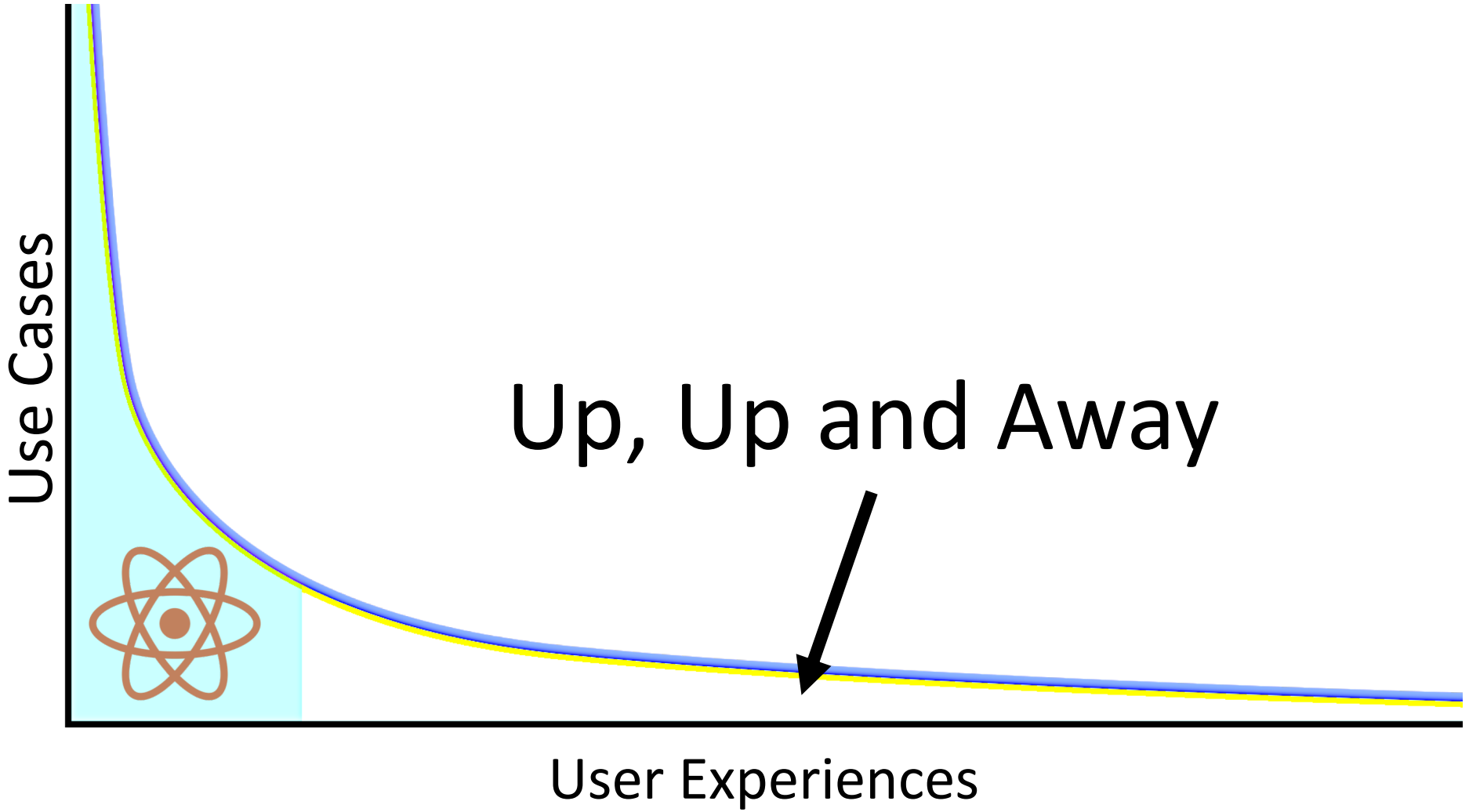


```
import { requiresNativeComponent, NativeModules } from "react-native";
import React, { Component } from "react";
const NativeView = requiresNativeComponent("RNBostonView", CameraView);
class CameraView extends Component {
…
CameraView.takePicture = async () => {
  try {
    return await NativeModules.RNBostonViewManager.takePicture();
  } catch (e) {
    return null;
  }
};
export default CameraView;
```

```
<TouchableOpacity
  onPress={async () => {
    const result = await CameraView.takePicture();
    const newText = result
    ? "Took a picture!"
    : "Error taking picture";
    this.setState({ cameraText: newText, imageURL: result.url })
  }}
>
  <CameraView
    style={{ width: "100%", height: "100%" }}
    cameraFront={this.state.cameraFront}
  />
</TouchableOpacity>
```

Takeaways

1. Add imperative functions as class methods for easy access
2. Return data via async/await for brevity
3. Superpowers.


Use Cases
Up, Up and Away
User Experiences

github.com/rhdeck @ray_deck

element55