Spring 2018

Comparing the Usage of React Native and Ionic

Sam Borick
sb205@zips.uakron.edu

Comparing the Usage of React Native and Ionic

Sam Borick

Since the iPhone launched with an App Store in 2008, business and organizations have been moving towards providing a high-quality experience for their users, on touch devices often smaller than a notecard. Not only does this in and of itself present challenges, but there was never a single dominant platform on the market. Currently in the US, 53% of smartphone owners have an Android device, while 44% of users have an iOS device.[1] Since the different platforms all have massively different implementation, languages, and design philosophies, for the majority of the history of mobile computing, a developer had to choose to specialize in one or the other, or attempt to master both.

However, recent efforts towards building useable cross-platform tools have finally begun to pay off. At the time of writing, a 'typical' structure for cross-platform applications has emerged. The application logic itself is created as one unit. When it is some sort of browser application, it is referred to as a hybrid app. Hybrid frameworks must provide a native application that contains a lightweight browser to run the web application. Typically the framework also provides a set of UI elements that are rendered differently on each platform, such that the UI is familiar to the user. This native application also includes mechanisms that allow this browser application to control native features, such as sensors or cameras. While this structure is not universal, it provides a useful concept to understand which parts of cross-platform applications run on both platforms, and which are different enough to require duplication and abstraction.

---

[1] "• US mobile smartphone OS market share 2012-2017 | Statistic - Statista." https://www.statista.com/statistics/266572/market-share-held-by-smartphone-platforms-in-the-united-states/. Accessed 14 Feb. 2018.

As a category, cross-platform tools all have several goals in common. The first is to leverage technologies that developers already possess. Obviously this would encourage developers to use the technology, and lower the barrier of entry to new developers by taking advantage of an existing pool of information. Another goal is to be able to target as many platforms as possible. A cross-platform framework that can build applications for every major platform would have an advantage over one that just targets a subset.

Of course, the advantages of cross-platform frameworks come at costs. One frequent issue is performance. Since these applications necessarily have a layer of abstraction, this can come in different forms. Frequently the process needed to generate the resulting application are longer than the build process for an equivalent native application. Additionally, the built application can also have poor performance. In the case of hybrid applications, if the embedded web application has poor performance, compounded with a poorly optimized bridge between the web component and the native component, the result can be an application that is noticeably show to the user. Additionally, since these applications almost always have components that have been re-done in some way, there are often small differences between the cross-platform application components and those from native applications. To take a simple example, many applications have a bar at the top or bottom with navigation components that take users between major pages within the application. In some frameworks, the transition between these different pages is not the same as in a native application. While this issue in and of itself is not significant, repeated instances of this kind of problem can result in an application that does not ‘feel’ native. The last major issue we will address here is access to native features. Since they must be abstracted, this means that not all features will be easily available, and some may not

be useable at all. Specifically looking at hybrid applications, the web portion cannot be run in background, meaning that all background functionality must be written in native code.

As web applications continue to grow in popularity, the popularity of JavaScript as a language has also grown. Subsequently, there has been a corresponding explosion of the variety of different frameworks and tools to aid in using JavaScript to fill many different needs. This has resulted in groups of developers gathering around particular frameworks and libraries, all of which promote their framework at being the best possible framework available. For the purposes of this case study, we will take an in-depth look at two major players: Ionic, and React Native.

For this case study, we will focus on answering these questions:

1. Is the ratio of switching costs to value gained of React Native vs Ionic as obviously low as the community contends?
2. What are the different strengths and weaknesses of the two platforms?
    a. If they share strengths and weaknesses, what are the root causes?
    b. If they don't, why?

In order to address these questions, we will have to understand the history of each, as well as their structure.

Methodology

Like any framework, each of these platform tackle these issues in their own ways. For the purpose of this discussion, we will focus on two, React Native, and Ionic. In an attempt to

compare apples to apples, we will attempt to compare these two platforms by examining how we would go about building the same application on both platforms.

In order to construct a simple application that is representative of a real application, we will look to a local business to provide a set of requirements for our sample application. This startup, HungerPerks, provides a service for creating and distributing virtual coupons, called SuperPerks. HungerPerks require an administrative application to create new coupons separate from their consumer application. This application needs to be able to, for each participating organization;

1. List existing coupons by fetching from an existing API
2. Create a new coupon
    a. Take an image of the item
    b. Crop the image
    c. Enter some data
        i. Title
        ii. Price
        iii. Etc

We believe that this is a good representation of what a ‘typical’ utility application will do, as it involves understanding the basics of dealing with external data, displaying that data in a structure, and interacting with native device functionality.

Threats to Validity

It is necessary to note the challenges that are present in this case study and its methodology. The first of which is to what degree two applications can be said to be the same, and from there

what degree of similarity is necessary for comparisons of the two applications to be valid. In the process of implementing any specification, many specific details are unearthed, which should theoretically reduce the amount of effort needed to build the application again. Additionally we already has familiarity with the Ionic framework prior to this case study. This could result in a bias towards Ionic during comparisons to React. In an attempt to mitigate these factors, we will write the version of the application in Ionic first. Ideally, this will make implementation of the React version easier, and will somewhat mitigate the issue of prior knowledge. Finally we must recognize the possibility that the two technologies might not share identical goals. While at a very high level they do (i.e. to create mobile applications), each framework likely identifies different areas of importance. This must be accounted for in our analysis.

Discussion

In order to be able to discuss the Ionic and React Native, we must first understand the structures that underlie each framework. We will start by understanding Ionic.

Ionic did not come about in isolation, it is implemented on top of two pre-existing frameworks. The first is a popular web framework, called Angular. Angular is a full-featured web framework that includes several importants concepts such as components, bindings, providers, and routing. We will focus on the subset of those features that are relevant to Ionic applications.

The first Angular feature to consider is Binding. This simple feature allows easy connections between content displayed to users and the underlying code.

For example:

```
@Component({selector: 'example', template: '<p>My current hero is {{currentHero.name}}</p>'})
```

```
class Example {
  currentHero: any = {
        name: string = "Example"
  }
}
```

Will retrieve the ‘currentHero’ object and display its ‘name’ property. Angular then takes care of making sure the value displayed to the user is the same as the value held in memory by the underlying code. This concept can be used further with Angular’s NgIf and NgFor.

For example:

```
@Component({selector: 'example', template: `
        <button (click)="show = !show">{{show ? 'hide' : 'show'}}</button>
         show = {{show}}
         <br>
         <div *ngIf="show">Content to show</div>
`})
class Example {
  show: boolean = false;
}
```

Angular then can handle weather or not to show a portion of template depending on the underlying application state, in this case the boolean variable ‘show’. However, perhaps the most powerful application of binding is the NgFor operator. Given a set of strings with the names of ‘Heros’ we want to display, we can create an array ‘HEROS’ in our class.

We can then create a visual representation of this set:

```
@Component({selector: 'example', template: `
        <div *ngFor="let hero of HEROS">
                {{hero}}
```

```
                <br>
        </div>


`})
class Example {
        const HEROES = [
            'Superman',
            'Batman',
            'BatGirl',
            'Robin',
            'Flash'
        ];
}
```

In order to facilitate better encapsulation and re-use of visual and logical components, Angular provides a concept of Components. Components allow for the arbitrary nesting of components into other components, as well as certain performance improvements by only running updates on components and their sub-components. Angular also provides an interface for passing data back and forth between methods. All of this comes together to abstract away the browser DOM, in effect creating a fake dom that updates itself in the background.

The final major service Angular provides is Services. Services perform function that can be long-running, don't directly involve user interaction, and should be further separated from components.

Now that we have some basic understanding of how Angular works, we can consider Ionic. As a hybrid application, Ionic can leverage Angular for the web application component. In order to

make Angular applications appear more like mobile applications, Ionic adds extra angular components that will change appearance depending on what device they are running on.

A simple example is the Ion-Icon component

This icon category can be invoked with

```
<ion-icon name="star"></ion-icon>
```

And ionic will display the appropriate Icon depending on what system the application is running on.

In order to consider the native portion of a hybrid application structure, we have to understand the other major framework Ionic utilizes, Apache Cordova. Cordova is a much older hybrid application framework that is normally used solely with JavaScript. Cordova allows for the concept of a plugin, which includes native code for supported platforms, as well as a JavaScript bridge between them. Ionic then provides tools that allow for communication from within the web portion to the Cordova plugin, thus gaining access to the underlying native functionality.

Now we must understand the inner workings of React Native. In the same way that Ionic is based on Angular, React Native is based on React. React has a similar concept of components, and employs a similar treed structure. React includes concepts of components and bindings, but does not include routing or services. First let's look at the equivalent of binding in react:

```
class Example extends React.Component {
  render() {
    var test = 2;
    return (
      <div>
        {test}
      </div>
    );
  }
}
```

The key difference in React is that the template must be returned from a render function, which will be called when data is updated. Otherwise, apart from minor syntactic differences, at a high level this is the same as Angular.

Again in React we can leverage binding for making decisions in what we show our user. For example:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      show: false,
    };
  }

  render() {
    const result = this.state.show ? (
      <div>Content to show</div>
    );
```

```
    return (
      <button onClick={() => this.setState({show: !show})}>
        {this.state.show}
      </button>
      {result}
    );
  }
}
```

Since the render function is called on a component when setState is called, the value in the variable result will change when the button is toggled, and the display will update. This pattern of the UI not being updated until requested continues throughout the React architecture. This allows for tighter developer control over performance bottlenecks.

For React Native, instead of the updates to the cloned DOM updating a browser DOM, the changes in the DOM are reflected in real application components. So instead of sending commands to a browser, React Native sends these commands to the device itself. This means that the React portion of the React Native application essentially 'describes' the native application output. Since the cross-platform and the native components are so tightly coupled together, the native applications output is included in the codebase, not as build output as in Ionic.

Comparison

Ionic and React Native both have very different goals as systems. We believe that these goals stem from the goals of their parent frameworks (Angular and React, respectively).

When considering the major design decisions of each parent framework, It becomes clear that Angular tends to obfuscate what action is really occurring in the browser in favor of it's own events, while React tends to make the actual DOM interactions as clear as possible. These core design decisions can be thought of as trending towards abstraction or away from abstraction. This trend of abstraction can be seen very clearly in the case of Ionic, as it extends to the point where the native device itself is essentially abstracted out, to the extent where the application is build output. The trend away from abstraction can be seen in React Native, evidenced by how tightly coupled it is with the underlying native implementation.

Results

When considering the results of this case study, the two main components we will look at are the List component, and the camera component. The list component will simply be the method of displaying an arbitrary number of coupons, and the camera will be the mechanism of capturing images from the device. First we will look at the list in both ionic and react.

As Angular separates components into html files and ts files, shown in Fig 1 and Fig 2, respectively.

```
<ion-content>
  <ion-list>
    <ion-item *ngFor="let SuperPerk of SuperPerks" >
        <ion-card class="col-12 large-card">
          <ion-card-header>
            {{SuperPerk.Title}}
          </ion-card-header>

          <div class="row">
              <img style="width:125px; height:125px;" src={{SuperPerk.ImageResource}} />
            <div class="col">
              Value: ${{SuperPerk.Value}}
              <br>
              Start: {{SuperPerk.Start | date:'medium'}}
```

```
              <br>
              End: {{SuperPerk.End | date:'medium'}}
              <br>
              Is Indefinite: {{SuperPerk.IsIndefinite}}
              <br>
              {{SuperPerk.NumLeft}} Remaining / {{SuperPerk.StartMax}}
              <br>
              Survey ID: {{SuperPerk.SurveyID}}
            </div>

            <div class="col">
              Geofence:
              <br>
                Latitude: {{SuperPerk.GeoFence.Latitude}}
                <br>
                Longitude: {{SuperPerk.GeoFence.Longitude}}
                <br>
                Radius: {{SuperPerk.GeoFence.Radius}}
            </div>
          </div>
        </ion-card>
    </ion-item>
  </ion-list>
</ion-content>
```

Fig 1

```
import { Component } from '@angular/core';
import { SuperPerkService } from './../../services/superperk-service';
import { SuperPerk } from '../../models/SuperPerk';

@Component({
  selector: 'page-contact',
  template: 'manage.html'
})
export class ManagePage {

       SuperPerks: Array<SuperPerk>;

  constructor(
       public SuperPerkSer: SuperPerkService,
       ) {
              SuperPerkSer.GetSuperPerks().subscribe((superperks) => {
                     this.SuperPerks = superperks
              })
   }

   ionViewWillEnter(){
       this.SuperPerkSer.GetSuperPerks().subscribe((superperks) => {
              this.SuperPerks = superperks
       })
   }

}
```

Fig 2

This simple component retrieves the data it needs via a service and saves it in a list. Then we use an ngFor to display the list of objects.

Now we can examine the React version in Fig 3.

```
import React from 'react';
import PropTypes from 'prop-types';
import { FlatList, TouchableOpacity, RefreshControl, Image, View } from 'react-native';
import { Container, Content, Card, CardItem, Body, Text, Button } from 'native-base';
import { Actions } from 'react-native-router-flux';
import Loading from './Loading';
import Error from './Error';
import Header from './Header';
import Spacer from './Spacer';

const SuperPerkPage = ({
  error,
  loading,
  superPerks,
  reFetch,
}) => {
  // Loading
  if (loading) return <Loading />;

  // Error
  if (error) return <Error content={error} />;

  const keyExtractor = item => item.id;

  const onPress = item => Actions.recipe({ match: { params: { id: String(item.id) } } });

  return (
    <Container>
      <Content padder>
        <FlatList
          numColumns={1}
          data={superPerks}
          renderItem={({ item }) => (
            <Card  style={{ paddingHorizontal: 6, paddingVertical:6 }}>
              <View style={{flex: 1, flexDirection: 'row'}}>
                <View style={{width: 150, height: 110}} >
                  <Image
                    source={{ uri: item.image }}
                    style={{
                      height: 100,
                      width: null,
                      flex: 1,
                      borderRadius: 5,
```

```
                  }}
                />
              </View>
              <View style={{width: 150, height: 110}} >
                <Body>
                  <Spacer size={10} />
                  <Text style={{ fontWeight: '800' }}>{item.title}</Text>
                  <Spacer size={15} />
                  <Spacer size={5} />
                </Body>
              </View>
            </View>
          </Card>
        )}
        keyExtractor={keyExtractor}
        refreshControl={
          <RefreshControl
            refreshing={loading}
            onRefresh={reFetch}
          />
        }
      />

      <Spacer size={20} />
    </Content>
  </Container>
 );
};

SuperPerkPage.propTypes = {
  error: PropTypes.string,
  loading: PropTypes.bool.isRequired,
  superPerks: PropTypes.arrayOf(PropTypes.shape()).isRequired,
  reFetch: PropTypes.func,
};

SuperPerkPage.defaultProps = {
  error: null,
  reFetch: null,
};

export default SuperPerkPage;
```

Fig 3

This page is structured as a react component, and takes four inputs: whether or not it is currently loading, whether or not there is an error, the SuperPerks to display, and the function to call when a reload is needed.

At this stage we can see that the syntactic differences are minor between these platforms.

Now we will look at the camera component in Ionic and React native.

```
import { Component, Output, Input, EventEmitter } from '@angular/core';
import { Camera, CameraOptions } from '@ionic-native/camera';

@Component({
  selector: 'camera',
  template: `
  <a (click)="TakePhoto()" >
    <ion-icon style="font-size: 70px;" name="camera"></ion-icon>
    <ion-icon *ngIf="Photo" style="font-size: 70px;" name="checkmark"></ion-icon>
  </a>
`
})
export class CameraComponent {

       @Input() Photo: any
  @Output() PhotoChange = new EventEmitter<any>();

  options: CameraOptions = {
    correctOrientation: true,
  }

  constructor(
    private camera: Camera,
    ) {}

  TakePhoto() {
    this.camera.getPicture(this.options).then(imageData => {
     this.PhotoChange.emit(imageData);
    }, (err) => console.error(err));
  }

}
```

Fig 4

Here in Fig 4 we can see that we have a simple camera button, which then calls a library that handles taking photos on the native device. This plugin than returns the data to our component, who reflects that fact visually for the user, and then returns the image data to the caller.

```
export default class App extends Component<{}> {
  render() {
    return (
      <View style={styles.container}>
        <Camera
```

```
          ref={(cam) => {
              this.camera = cam;
           }}
           aspect={Camera.constants.Aspect.fill}>
              <Text style={styles.capture} onPress={this.takePicture.bind(this)}>
                 [CAPTURE]
              </Text>
        </Camera>
      </View>
    );
  }

  takePicture() {
     this.camera.capture()
        .then((data) => console.log(data))
        .catch(err => console.error(err));
  }
}
```

Fig 5

Again, we can see in Fig 5 that the structure is similar between both platforms. React has a button that calls a function, which then calls a library to take a photo. In the case of react we do not pass up the image data to the parent.

So we can see that on a syntactic component level, there is little major difference in the usage of React Native and Ionic. Are there other differences to consider? These components do not exist in isolation. They must both be integrated into a larger system of files and program structures, and those structures are so widely different it is difficult to objectively compare them. However, it is worth noting that React does not consider itself to be a web framework. This means that it does not include many of the larger structures that Angular provides, and thus other tools must fill those voids. This fracturing steepens the learning curve.

Conclusion

The first research question we addresses was “Is the ratio of switching costs to value gained of React Native vs Ionic as obviously low as the community contends?” The answer to this question depends on the goals of the developer in question. If the goals of the application are to be simple to build, relatively easy to maintain, and be easy for those who are new to front-end development in general, Ionic may be a good choice compared to React Native. This is evidenced by the tendency of completeness of features in Angular. This results in not having to leave the angular ecosystem, easing the learning curve. Additionally, the tendency in the Angular ecosystem to abstract the DOM extends into the Ionic framework in abstracting the native device itself, further easing the learning curve.

However, if the goals are to build a high-quality application that leverages the benefits of cross platform applications while being simpler than developing native application, than React may be a better choice compared to Ionic. This is evidenced by the tendency to be tightly connected with the resulting application, as shown by the fact that the applications are not build outputs.

Ionic separates developers from the native side as much as possible, just as Angular does with the DOM. This results in build processes bring very intensive. This also results in a single, all-in-one tool that can produce apps of lower quality. React Native ends up doing a much better job of creating excellent applications, but does so via a much more complicated and diverse toolset, which raises barrier to entry. It's tight coupling with the resulting application also adds to this. To conclude, Ionic and React Native, while sharing high-level goals, have different intended audiences, and serve slightly different needs.