# NativeScript: Mobile Applications for the Angular Developer

> In this chapter, we're going to walk through how to build your first NativeScript app. NativeScript is a huge topic that could warrant it's own book.
>
> Here we're going to explain NativeScript for the Angular Developer. By the end of this chapter you'll understand the differences between NativeScript and a 'regular' Angular web-app, and have the foundation to be creating your own native apps using NativeScript and Angular.

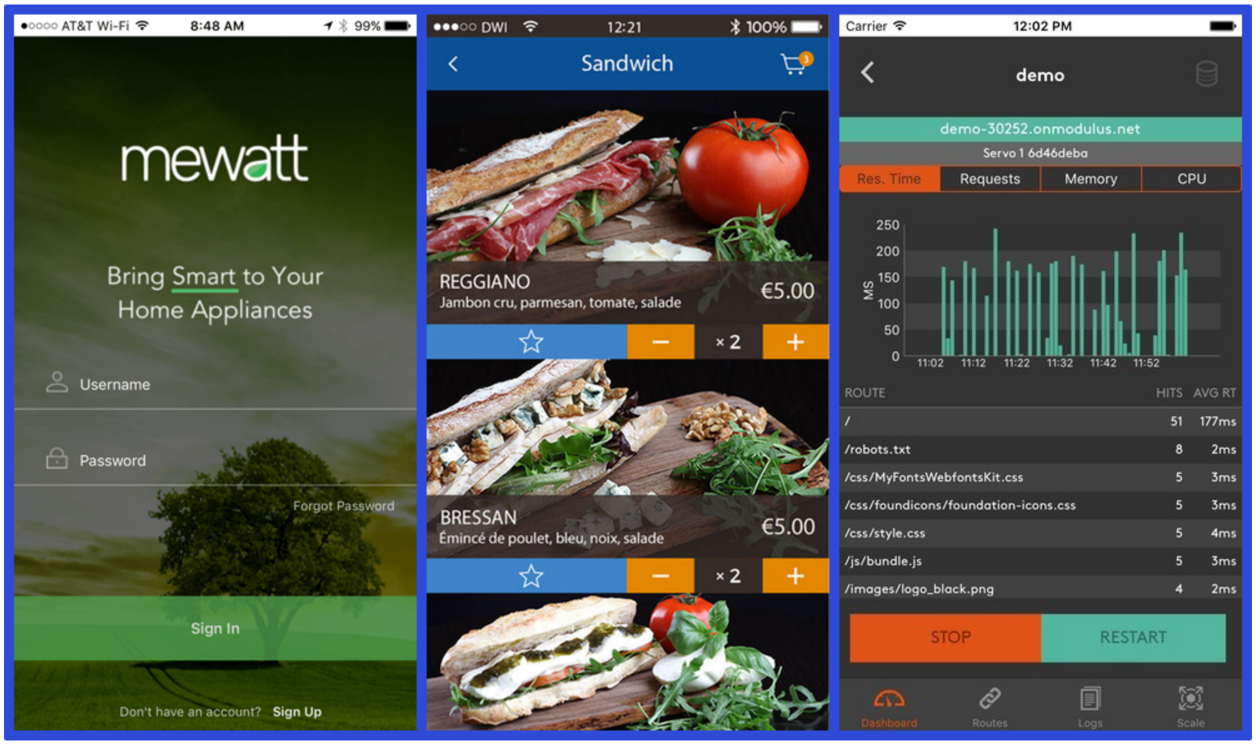
Being that Angular was designed to be unspecific to any particular deployment platform, you can take much of your web application code and reuse it beyond just the web.

It is the norm for businesses to have not only a fully functional web application, but a mobile application to compliment it as well. A few years back, companies would need to spend countless dollars to fund a team of iOS and Android developers to accomplish the same task of creating a mobile application.

With Angular, mobile development becomes not only cheaper, but more maintainable and efficient.

## What is NativeScript?

NativeScript is a cross platform mobile development framework that leverages technologies you already know: JavaScript, CSS, and of course, Angular.



| ROUTE | HITS | AVG RT |
|---|---|---|
| / | 51 | 177ms |
| /robots.txt | 8 | 2ms |
| /css/MyFontsWebfontsKit.css | 5 | 3ms |
| /css/foundicons/foundation-icons.css | 5 | 3ms |
| /css/style.css | 5 | 4ms |
| /js/bundle.js | 5 | 3ms |
| /images/logo_black.png | 4 | 2ms |

**NativeScript Showcase**

With NativeScript, developers can build native iOS and Android applications using a single shared code base.

## Where NativeScript Differs from Other Popular Frameworks

NativeScript isn't the first or only framework to make it easy to develop Android and iOS applications using a single code base. Mobile development frameworks can be separated into two: *hybrid* mobile and *native* mobile.

### Hybrid Mobile Applications

Hybrid mobile frameworks are those such as Ionic Framework[154], PhoneGap[155], Apache Cordova[156], and Onsen UI[157]. These are frameworks that allow you to develop mobile applications using web technologies, but render these mobile applications in what's called a *web view*. A web view is essentially a web browser and it allows you to use HTML with full DOM support for all your component rendering.

The conveniences of a web view is not without limitation. The number one flaw in using a web view to render mobile applications comes down to performance. Not all mobile devices are treated

[154]https://ionicframework.com/
[155]http://phonegap.com/
[156]https://cordova.apache.org/
[157]https://onsen.io/

as equal even if they have the same version of Android or iOS. There are thousands of different mobile handsets in existence all with varying hardware and processing power, not to mention all the custom flavors of Android. Because of this diversity, the consistency in web view performance is very poor, leaving some people with an amazing user experience and some with hardly useable applications.

### Native Mobile Applications

Native mobile applications built with frameworks such as NativeScript[158], React Native[159], and Xamarin[160] **do not render in a web view**. These are applications that use the **native UI components** that Google and Apple made available to developers and as a result don't suffer from performance instability.

So how does one choose between the available native mobile frameworks? The simple answer is to choose between each of their underlying development technologies. React Native uses ReactJS, a common JavaScript framework for web developers, and Xamarin uses C#, a common development language for .NET developers. NativeScript of course uses Angular.

As an Angular developer, it makes sense to go the NativeScript route because we'll get fantastic native performance while keeping our the familiar Angular development experience.

## What are the System and Development Requirements for NativeScript?

NativeScript doesn't have any system requirements beyond what you'd need when developing Objective-C based iOS applications or Java based Android applications.

For example, let's say you wanted to build and deploy an Android application developed with NativeScript. You would need at least the following:

- Windows, Linux, or Mac
- Java Development Kit (JDK) 8+
- 4GB of hard drive space
- 4GB of RAM

The above system and software requirements are what's necessary for installing and using the Android SDK.

If you wanted to build and deploy and iOS application with NativeScript, the requirements are a bit different:

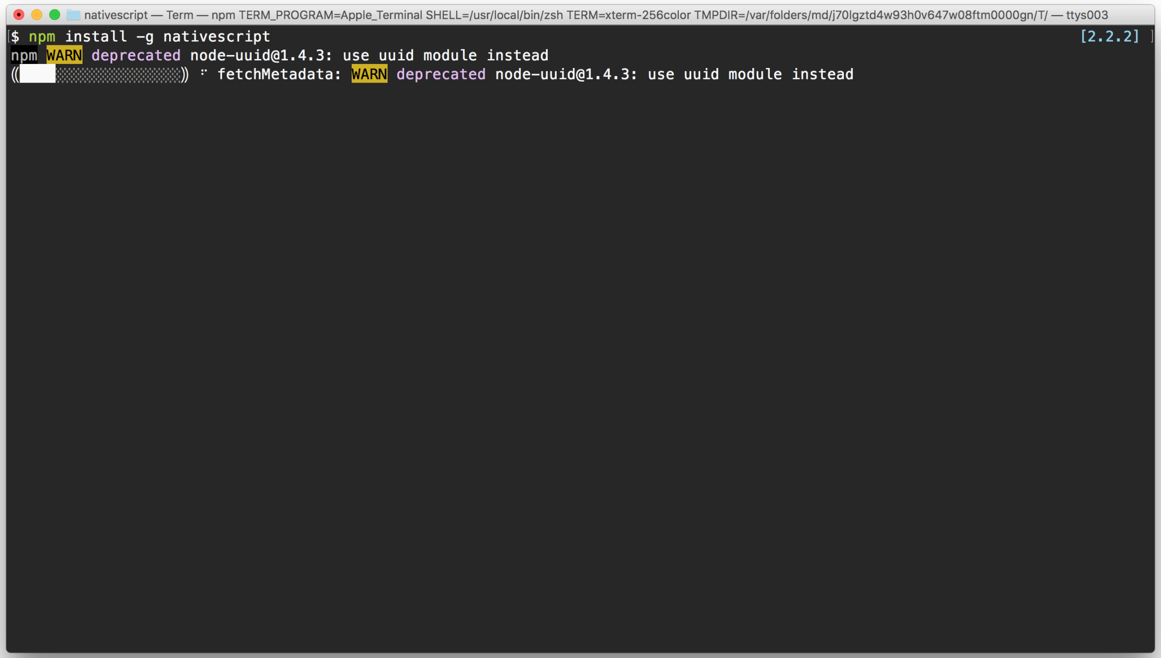
[158]https://www.nativescript.org/
[159]https://facebook.github.io/react-native/
[160]https://www.xamarin.com/

- Mac
- Xcode 7+
- 5GB of hard drive space
- 4GB of RAM

Notice the main difference here is that a Mac is required. While you can develop Android and iOS applications with NativeScript, you cannot actually build and deploy iOS applications unless you're using a Mac. This is a limitation that exists because of Apple.

From a development perspective, NativeScript uses the Node Package Manager (NPM) a tool that is part of Node.js and something you probably already have installed as an Angular developer. With NPM, the NativeScript CLI can be installed using the following command:

```
npm install -g nativescript
```


nativescript — Term — npm TERM_PROGRAM=Apple_Terminal SHELL=/usr/local/bin/zsh TERM=xterm-256color TMPDIR=/var/folders/md/j70lgztd4w93h0v647w08ftm0000gn/T/ — ttys003

```
$ npm install -g nativescript                                                    [2.2.2]
npm WARN deprecated node-uuid@1.4.3: use uuid module instead
((      ░░░░░░░░░░░░░░)) ⠈ fetchMetadata: WARN deprecated node-uuid@1.4.3: use uuid module instead
```

**Installing NativeScript**

A list of available commands can be found by running `tns --help` or `tns help` if you wish to view them in a web browser rather than the Command Prompt or Terminal.

For more information on installing NativeScript for Mac, Windows, and Linux, visit the NativeScript installation documentation[161].

[161]https://docs.nativescript.org/start/quick-setup.html

There are a significant number of tools to be installed to do native app development. Once everything is installed properly, NativeScript development is relatively painless, but make sure you visit the URL above if you run into any trouble getting the NativeScript build tools installed.

With the NativeScript CLI, native mobile applications can be developed with Angular.

## Creating your First Mobile Application with NativeScript and Angular

To be successful in developing NativeScript applications with Angular, you should already have the NativeScript CLI tool installed and either Xcode or the Android SDK installed, or both.

The goal here is to become familiar with the mobile application creation process and some of the UX and UI differences between an Angular web application and an Angular NativeScript application.

Using the Command Prompt (Windows) or Terminal (Mac and Linux), execute the following:

```
tns create NgProject --ng
```

The above command will create a project directory, `NgProject`, wherever your command line's active directory is located. The `--ng` flag indicates that we want to create an Angular with TypeScript project. It is necessary to use the `--ng` flag because NativeScript doesn't require Angular to build mobile applications. It is an option, one that we're going to take full advantage of.

### Adding Build Platforms for Cross Platform Deployment

While a project has been created and can be actively developed, there are no build platforms such as Android or iOS enabled for building and deployment.

To build for a specific platform, it must first be added. Using the NativeScript CLI, execute the following:

```
tns platform add [platform]
```

Just swap out `[platform]` with either `android` or `ios` depending on which you wish to add, remembering that iOS requires a Mac with Xcode installed.

### Building and Testing for Android and iOS

When the application is ready for testing or deployment to the app stores, we can make use of a few NativeScript CLI commands. Before deployment, you'll probably want to test the application on your device or emulator. Using the command line, execute the following to emulate the application:

```
tns emulate [platform]
```

Swapping `[platform]` with `android` or `ios` will launch the application in the specified emulator. To test the application on a device, swap out `emulate` with the word `run` while your device is connected to your development machine.

```
tns run [platform]
```

The emulation process can often take a bit of time because a lot of recompilation happens in the process. To make development more efficient, the NativeScript CLI offers live-reload functionality called live-sync. We can utilize this feature by executing the following command in our terminal:

```
tns livesync [platform] --emulator --watch
```

After swapping `[platform]` with either `android` or `ios`, changes made to TypeScript, CSS, or HTML files will be automatically deployed to the Android or iOS simulator, much faster than if you were to strictly emulate the application.

When it comes to deploying our app to the app store, we can use the following command:

```
tns build [platform]
```

After replacing `[platform]` with the appropriate platform, the binaries and build packages will be created.

## Installing JavaScript, Android, and iOS Plugins and Packages

Like with any Angular web application, there are external components available to make the development process easier. This applies to NativeScript applications as well.

Most JavaScript packages will work in a NativeScript application as long as there isn't a dependency on the DOM. As previously mentioned, NativeScript being a native framework, doesn't use a web view and has no concept of a DOM. JavaScript libraries can be included via NPM, for example:

```
npm install jssha --save
```

The above would install the JavaScript hashing library, jsSHA, to your Angular NativeScript project.

There are native plugins available strictly for NativeScript as well. These are typically plugins that make use of native device features or interface with Android or iOS directly in some fashion.

Take, for example, the NativeScript SQLite plugin:

```
tns plugin add nativescript-sqlite
```

The above command will install SQLite functionality for both Android and iOS.

## Understanding the Web to NativeScript UI and UX Differences

As a web developer you're probably very familiar with HTML and common design practices for building attractive, responsive, and overall great web applications. With NativeScript we're using Angular and CSS, but we're not using HTML. Instead we are using XML which won't have the same markup tags that you'd find in HTML.

So how do you take your UI and UX skills to mobile?

There are a few things that need to be taken into consideration when designing your mobile application. You need to worry about the screen layout and the screen components.

### Planning the NativeScript Page Layout

When designing a web application, common layout components include `<div>` tags and `<table>` tags. Generally if you want a grid of rows and columns you'd use a table and if you wanted a stack of components you'd use a div because it acted as a container.

In NativeScript, you don't have the `<div>` and `<table>` tags, but you have something similar. Instead you have the `<StackLayout>` and `<GridLayout>` tags.

So let's compare web and NativeScript.

Let's say we wanted to contain a bunch of HTML components on a website. You might do something like the following:

```
<div>
    <span>Nic Raboy was here</span>
    <span>https://www.thepolyglotdeveloper.com</span>
</div>
```

To accomplish the same in a NativeScript application, you'd do the following:

```
<StackLayout>
    <Label text="Nic Raboy was here"></Label>
    <Label text="https://www.thepolyglotdeveloper.com"></Label>
</StackLayout>
```

In both the web and NativeScript scenarios you can nest the `<div>` and `<StackLayout>` tags as appropriate to create more component groupings.

The use of grids in NativeScript and on the web are a bit different in structure, but the same in concept. Take the following HTML:

```
<table>
    <tr>
        <td>Nic</td>
        <td>Raboy</td>
    </tr>
    <tr>
        <td>Burke</td>
        <td>Holland</td>
    </tr>
</table>
```

In NativeScript, instead of defining rows and columns with `<tr>` and `<td>` tags something a little different happens:

```
<GridLayout rows="auto, auto" columns="*, *">
    <Label text="Nic" row="0" col="0"></Label>
    <Label text="Raboy" row="0" col="1"></Label>
    <Label text="Burke" row="1" col="0"></Label>
    <Label text="Holland" row="1" col="1"></Label>
</GridLayout>
```

In the above `<GridLayout>` we define that we want two rows that take the height of their child components and two columns that stretch evenly to fill the screen.

But what about a flexbox, commonly found on the web?

When building websites, there is the opportunity to set `<div>` tags, or any other container, to have a CSS property of `display: flex`. This allows websites to behave appropriately for different screen sizes. Nearly the same can be used in NativeScript using the `<FlexboxLayout>` as a container, which is nearly the same as the web's implementation.

## Adding UI Components to the Page

When it comes to NativeScript there are many UI components available, each accomplishing something different. For example we already saw how to display static text on the screen through the use of the `<Label>` component, but what other options are available?

There are too many components to name, but some of the common components include buttons, images, lists, and inputs. These are all components that are common to what you'd find in a web application as well.

To add a button to our application, we'd add the following to one of our layouts:

```
<Button text="Submit Me" (tap)="myFunc()"></Button>
```

Notice the use of the `(tap)` attribute. This is not specific to the UI component, but more a mixture of Angular and NativeScript. In a web application these events are better known as `(click)` events, however, they both accomplish the same.

To include an image, local or remote, within an application, we can use the `<Image>` tag like so (similar to the `<img />` tag on the web):

```
<Image src="https://placehold.it/350x150"></Image>
```

Many mobile applications, like web applications, collect data from users. This data is collected through forms composed of text input fields. To accept text input in a NativeScript application, make use of the `<TextField>` tag like the following:

```
<TextField
    text="First Name"
    [(ngModel)]="firstname"></TextField>
```

The `[(ngModel)]` attribute seen above is identical to that which is found in an Angular web application. It allows the binding of data between the UI and the TypeScript paired to it.

It is often necessary to list large amounts of data within a mobile application. This data is presented in what is called a `<ListView>`. These lists are populated from arrays of strings or objects that are defined within the application TypeScript.

```
<ListView [items]="people">
    <Template let-person="item">
        <Label [text]="person.firstname"></Label>
    </Template>
</ListView>
```

The above snippet will create a list from an array of objects called `people`. Each object in the array will be called `person` and the `firstname` of each `person` will be displayed in a list row.

Again, there are many other components available, some not heard of in the land of web development. However, they are all similar by design.

Just like with web components, NativeScript UI components don't look attractive in their vanilla state. They need to be themed and styled with some artistic flair.

## Styling Components with CSS

There are a few options available when it comes to giving a NativeScript application a boost in the attractiveness department, just as there is in web design.

NativeScript allows UI components to be styled with a CSS subset. To be clear, most web CSS will work in NativeScript, but not everything. To change the font color of a `<Label>` component, the following is an option:

```
.title {
    color: #cc0000;
}
```

The class name can then be applied to the UI component in the same fashion as with HTML.

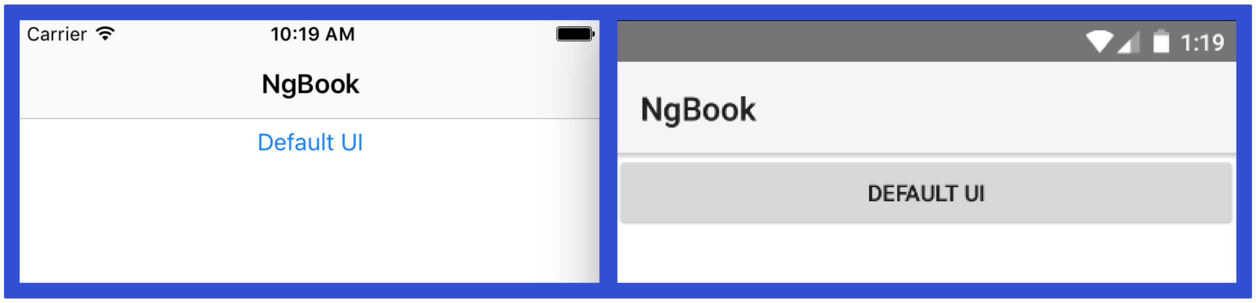
Creating a custom stylesheet isn't the only solution when it comes to making a NativeScript application more attractive. When building a website, there are frameworks such as Bootstrap that were designed to make life easier. We can translate this same concept with NativeScript.

There is what is called NativeScript Theme, which is a package of CSS styles designed to be easily added to any application.

Take the following action bar with button example:

```
<ActionBar title="NgBook"></ActionBar>
<StackLayout>
    <Button text="Default UI"></Button>
</StackLayout>
```

The above code would generate a native, but very plain looking action bar with a very plain looking button. On Android and iOS, it would look like the following:

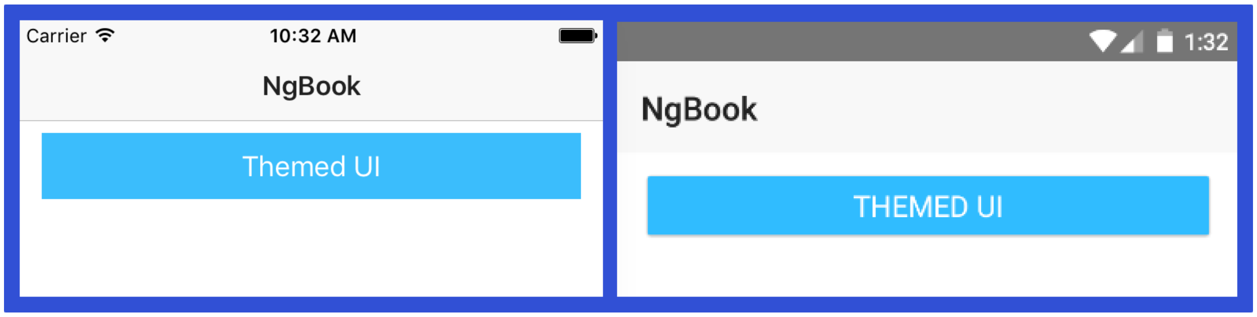


**NativeScript Basic CSS**

This simple UI can be significantly improved by using NativeScript Theme. For example, take the minor revisions to the code snippet found below:

```
<ActionBar title="NgBook" class="action-bar"></ActionBar>
<StackLayout>
    <Button text="Themed UI" class="btn btn-primary"></Button>
</StackLayout>
```

A few class names were applied to the components giving them a much more pleasant look and feel as demonstrated in the image below:



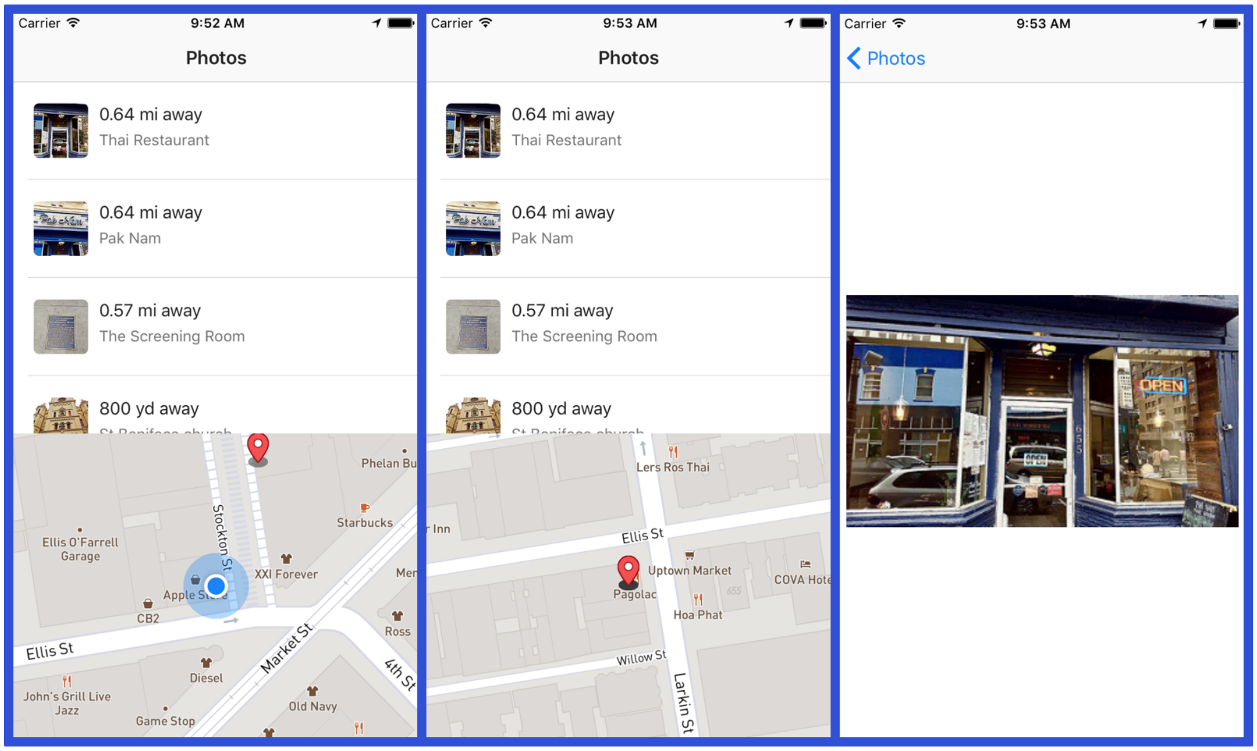
**NativeScript Theme CSS**

The naming conventions for the theme classes have a similar naming convention to those found in the popular web frameworks.

## Developing a Geolocation Based Photo Application

Taking what we know about Angular, web development, and the NativeScript mobile framework, we can apply it towards creating a native and functional mobile application for both iOS and Android.

Much of what comes next will be a review of the Angular skills you already have, but in a mobile example. The example application will use geolocation and the Flickr API to show images that were captured near you.



**NativeScript Photos Near Me**

The application will have two pages that act as a master-detail interface meaning the first page will list data and the second page will show more information about the data selected from the first page.

The completed project can be found in the sample code under `code/nativescript/photos-near-me`.

## Creating a Fresh NativeScript Project

To get the most out of this demo, it would be good to start with a new project. As a review to what was mentioned previously, a project can be created by executing the following:

```
tns create GeoPhotoProject --ng
cd GeoPhotoProject
tns platform add android
tns platform add ios
```

The above commands will create an Angular NativeScript project called **GeoPhotoProject** with the Android and iOS build platforms. To be able to build iOS applications we must be using a Mac with Xcode installed.

The default project template will be a single page application, so we'll have to add more pages and configure the Angular Router.

## Creating a Multiple Page Master-Detail Interface

The default project template uses the project's **app/app.component.html** file as the default page. This file will still be valuable in this project, but we're going to create two new pages.

Let's create a few of the new components we'll use by executing the following commands to create necessary files and directories:

```
mkdir -p app/components/image-component
mkdir -p app/components/imagesList-component
touch app/components/image-component/image.component.ts
touch app/components/image-component/image.component.html
touch app/components/imagesList-component/imagesList.component.ts
touch app/components/imagesList-component/imagesList.component.html
```

We can also create these directories manually in our Explorer window, if the `mkdir` or `touch` commands are not available in our command-line (or if we just feel more comfortable in the UI).

The first page in our application flow will be the **imagesList.component** page to display all the list of photos.

Let's open the project's **app/components/imagesList-component/imagesList-component.ts** file and include the following basic class code:

```
import { Component, NgZone } from "@angular/core";
import { Router } from "@angular/router";

@Component({
    selector: "ImagesListComponent",
    templateUrl: "components/imagesList-component/imagesList.component.html"
})
export class ImagesListComponent {

    public constructor(private zone: NgZone, private router: Router) { }

}
```

In the above code the `ImagesListComponent` class is being defined and various Angular components are being imported and injected into the `constructor` method in the usual method.

The UI that goes with the `ImagesListComponent` class is found in the **app/components/imageList-component/imagesList-component.html** file. For now, let's update the file to contain following HTML markup:

```
<ActionBar title="Photos" class="action-bar"></ActionBar>
<StackLayout>
</StackLayout>
```

Before we add useful functionality to the first page of our application, let's lay the foundation to the second page and link them together.

Open the project's **app/components/image-component/image-component.ts** file and include the following TypeScript code:

```
import { Component, OnInit } from "@angular/core";
import { ActivatedRoute } from "@angular/router";

@Component({
    templateUrl: "components/image-component/image.component.html"
})
export class ImageComponent implements OnInit {

    public constructor(private activatedRoute: ActivatedRoute) { }

    public ngOnInit() { }

}
```

In the above code the `ImageComponent` class is created and various Angular components are imported and injected in the `constructor` method. The core difference here, as of now, is the `ngOnInit` method which is going to be used to load data after the page loads.

The UI that goes with the TypeScript code is found in the **app/components/image-component/image-component.html** file and it will contain, for now, the following HTML markup:

```
<ActionBar></ActionBar>
<StackLayout>
</StackLayout>
```

With the pages available, they need to be brought together for Angular routing. This requires two things to happen. First, the routes need to be defined and second they need to be included in the project's `@NgModule` block.

Let's create an **app/app.routing.ts** file in our project and include the following routing configuration code:

**code/nativescript/photos-near-me/app/app.routing.ts**

```
import { ImagesListComponent } from "./components/imagesList-component/imagesLis\
t.component";
import { ImageComponent } from "./components/image-component/image.component";

export const routes = [
    { path: "", component: ImagesListComponent },
    { path: "image-component/:photo_id", component: ImageComponent },
];

export const navigatableComponents = [
    ImagesListComponent,
    ImageComponent
];
```

In the above code, both the `ImagesListComponent` and `ImageComponent` classes were imported. The routes define how to navigate to each of the classes and what data can be passed. The `ImagesListComponent` has an empty path which represents the default, or first page that loads when the application starts. The `ImageComponent` has a path with one URL parameter which represent a piece of data that can be passed from the `ImagesListComponent` page to the `ImageComponent` page.

Without getting too far ahead of ourselves, the `photo_id` represents the photo we wish to load in the second page. This is a piece to the Flickr API.

The **app/app.routing.ts** file needs to be imported and added to the project's `@NgModule` block. In our project's **app/app.module.ts** file and include the following TypeScript code:

```
import { NativeScriptModule } from "nativescript-angular/platform";
import { NgModule } from "@angular/core";
import { NativeScriptFormsModule } from "nativescript-angular/forms";
import { NativeScriptHttpModule } from "nativescript-angular/http";
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { registerElement } from "nativescript-angular/element-registry";

import { AppComponent } from "./app.component";
import { routes, navigatableComponents } from "./app.routing";

@NgModule({
    imports: [
        NativeScriptModule,
        NativeScriptFormsModule,
        NativeScriptHttpModule,
        NativeScriptRouterModule,
        NativeScriptRouterModule.forRoot(routes)
    ],
    declarations: [
        AppComponent,
        ...navigatableComponents,
    ],
    bootstrap: [AppComponent],
    providers: []
})
export class AppModule {}
```

There is more setup in this file than what you'll find in the default. To save us some time we're importing the `NativeScriptFormsModule`, `NativeScriptHttpModule`, and `NativeScriptRouterModule` along with the `routes` and `navigatableComponents` variables that were defined in the previous file.

Each module is added to the `imports` array of the `@NgModule` block and the two page classes found in the `navigatableComponents` variable are added to the `declarations` array.

Even though the application doesn't do much at the moment, it is linked together and ready to go. Adding UI components and functionality will be explored later on.

Finally, we'll need to add a place for our pages to render via our routes. In our main app component in `app/app.component.html`, let's add the `<page-router-outlet/>` markup to tell Angular where to render our subroutes. Since we don't have any common views between views, can replace all of the content with this markup:

```
<page-router-outlet></page-router-outlet>
```

## Creating a Flickr Service for Obtaining Photos and Data

Flickr will be a critical part of this application. Instead of calling the Flickr API directly in each of the pages we wish to use it, the better approach would be to create an Angular service, also known as a provider.

In a Flickr provider we can add logic to query for photos based on latitude and longitude information as well as get information about particular photos.

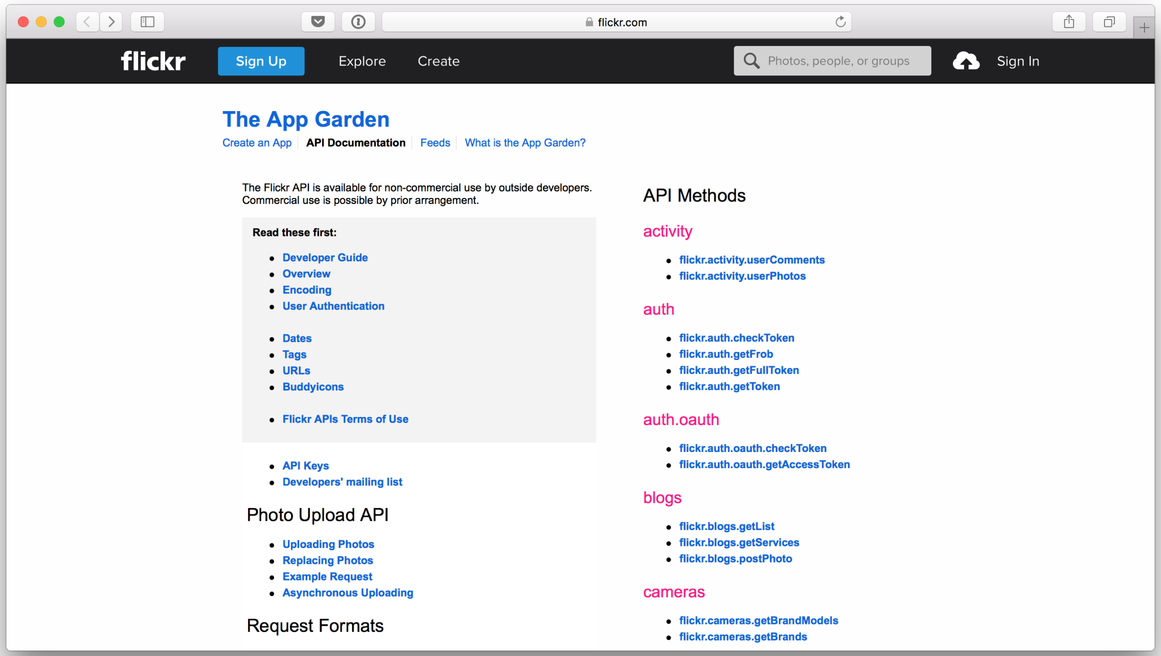
Before designing this provider it is a good idea to create a global configuration file for the application. This will prevent hard coded URL values, amongst other things, in the application.

Let's create a **app/config.ts** file and include the following:

```
export const Config = {
    Flickr: {
        CLIENT_ID: "FLICKR_CLIENT_ID_HERE",
        API_URL: "https://api.flickr.com/services/rest/?"
    }
};
```

Before using the Flickr API, an account needs to be created to obtain a client id. We'll head to https://www.flickr.com/services/api/[162] and create an account.

[162] https://www.flickr.com/services/api/



The App Garden

Create an App | API Documentation | Feeds | What is the App Garden?

The Flickr API is available for non-commercial use by outside developers. Commercial use is possible by prior arrangement.

**Read these first:**

- Developer Guide
- Overview
- Encoding
- User Authentication

- Dates
- Tags
- URLs
- Buddyicons

- Flickr APIs Terms of Use

- API Keys
- Developers' mailing list

Photo Upload API

- Uploading Photos
- Replacing Photos
- Example Request
- Asynchronous Uploading

Request Formats

API Methods

activity

- flickr.activity.userComments
- flickr.activity.userPhotos

auth

- flickr.auth.checkToken
- flickr.auth.getFrob
- flickr.auth.getFullToken
- flickr.auth.getToken

auth.oauth

- flickr.auth.oauth.checkToken
- flickr.auth.oauth.getAccessToken

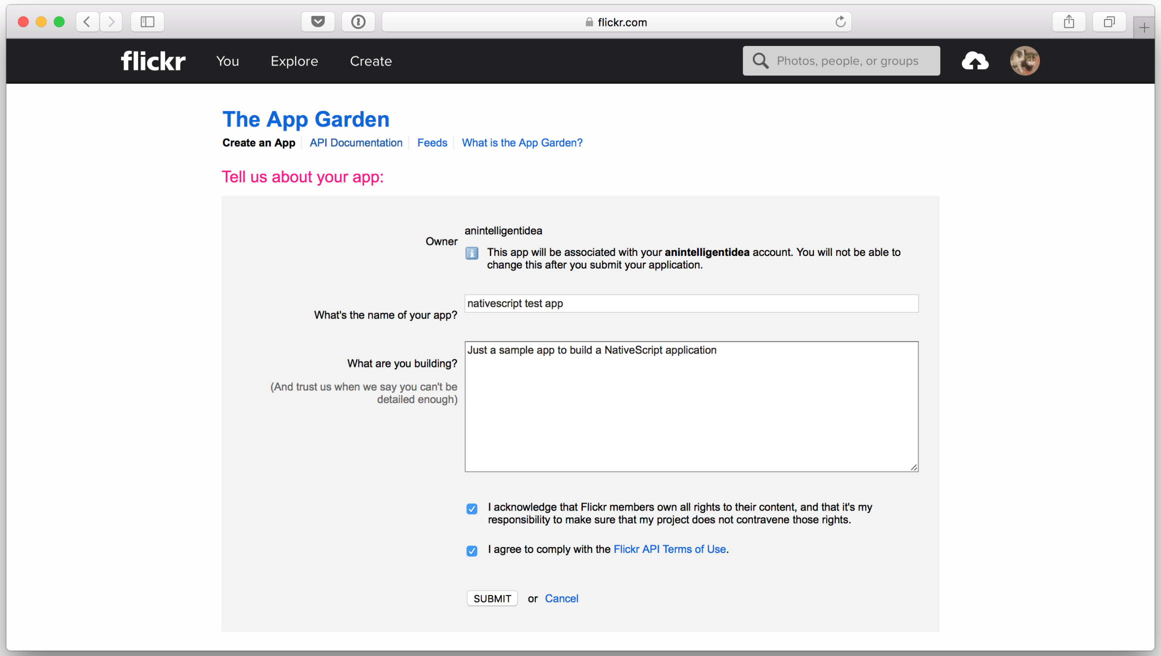
blogs

- flickr.blogs.getList
- flickr.blogs.getServices
- flickr.blogs.postPhoto

cameras

- flickr.cameras.getBrandModels
- flickr.cameras.getBrands

Flickr will create a `client_id` that will be unique to our application. The value of the `FLICKR_-CLIENT_ID_HERE` in our `app/config.ts` file.



The App Garden

Create an App | API Documentation | Feeds | What is the App Garden?

Tell us about your app:

Owner: anintelligentidea

This app will be associated with your **anintelligentidea** account. You will not be able to change this after you submit your application.

What's the name of your app? nativescript test app

What are you building? (And trust us when we say you can't be detailed enough) Just a sample app to build a NativeScript application

✓ I acknowledge that Flickr members own all rights to their content, and that it's my responsibility to make sure that my project does not contravene those rights.

✓ I agree to comply with the Flickr API Terms of Use.

SUBMIT or Cancel

With the configuration file created, we need to define a data model for the Flickr responses. While not absolutely necessary, it does create a more maintainable TypeScript application.

Let's create the `app/models` directory, if it does not already exist:

```
mkdir app/models
```

Create an **app/models/getInfoResponse.ts** file and include the following TypeScript code:

**code/nativescript/photos-near-me/app/models/getInfoResponse.ts**

```
interface Owner {
    username: string;
    realname: string;
}

export class GetInfoResponse {
    owner: Owner;
    farm: number;
    server: number;
    secret: string;
    id: number;
    url: string;
}
```

The above represents the data that is returned from the Flickr `flickr.photos.getInfo` RESTful endpoint. The data makes it possible to obtain an image file along with holding information about that image file.

The second model we need is for Flickr search data. Create an **app/models/photosSearchResponse.ts** file with the following TypeScript code:

**code/nativescript/photos-near-me/app/models/photosSearchResponse.ts**

```
export class PhotosSearchResponse {
    id: string;
    owner: string;
    secret: string;
    server: number;
    title: string;
    latitude: string;
    longitude: string;
    datetaken: string;
    url_t: string;
    url_m: string;
    url_q: string;
    url_n: string;
    distance: string;

    constructor() {
        this.url_n = " ";
    }
}
```

The above model holds useful information such as the photo id, the owner, and geolocation information all useful when discovering images and displaying them on the second page of the application.

With the data models created, we can now create the Flickr service. Let's create a file at **app/services/flickr.service.ts** in the project.

```
mkdir app/services
touch app/services/flickr.service.ts
```

We'll start with this foundation, in the `flickr.service.ts` file:

```
import { Component, Injectable } from "@angular/core";
import { Http, Response } from "@angular/http";
import { Observable } from "rxjs/Rx";
import { Config } from "../app.config";
import { PhotosSearchResponse } from "../models/photosSearchResponse";
import { GetInfoResponse } from "../models/getInfoResponse";
import "rxjs/add/operator/map";

@Injectable()
export class FlickrService {

    public constructor(private http: Http) { }

    public photosSearch(lat: number, lon: number): Observable<PhotosSearchRespon\
se[]> { }

    public getPhotoInfo(photoId: number): Observable<GetInfoResponse> { }

}
```

Both the `photosSearch` and `getPhotoInfo` functions return observables which are streams of data obtained by HTTP requests to the Flicker API.

The `photosSearch` function will take a latitude and longitude and apply it towards Flickr's API like follows:

**code/nativescript/photos-near-me/app/services/flickr.service.ts**

```
    public photosSearch(lat: number, lon: number): Observable<PhotosSearchRespon\
se[]> {
        let url = `${Config.Flickr.API_URL}method=flickr.photos.search&api_key=$\
{Config.Flickr.CLIENT_ID}&content_type=1&lat=${lat}&lon=${lon}&extras=url_q,geo&\
format=json&nojsoncallback=1`;

        return this.http.get(url)
            .map(response => response.json().photos.photo)
            .catch(error => Observable.throw(error));
    }
```

An HTTP request is made per the Flickr API documentation. Using RxJS, the response of the request is transformed using the `map` operator to be of type `PhotosSearchResponse`. If there is an error in the response, it will be caught through the normal http promise error chain. Just like normal Angular, our HTTP request won't execute until the observable is subscribed.

The `getPhotoInfo` method will take a photo id, probably from the result returned in the previous `photosSearch` function:

**code/nativescript/photos-near-me/app/services/flickr.service.ts**

```
    public getPhotoInfo(photoId: number): Observable<GetInfoResponse> {
        let url = `${Config.Flickr.API_URL}method=flickr.photos.getInfo&api_key=\
${Config.Flickr.CLIENT_ID}&photo_id=${photoId}&format=json&nojsoncallback=1`;

        return this.http.get(url)
            .map(response => response.json().photo)
            .catch(error => Observable.throw(error));
    }
```

Like with the `photosSearch` function, the `getPhotoInfo` function makes a HTTP request against the Flickr API and parse the response using RxJS.

Before the Flickr provider can be used throughout the application, it must be added to the `@NgModule` block similarly to how the application pages were added.

Inside the project's **app/app.module.ts** file, we need to import the Flickr service must be imported and then add it to the `providers` array in the `@NgModule` block:

```
import { FlickrService } from "./services/flickr.service";

@NgModule({
  // ...
  providers: [FlickrService]
})
...
```

The Flickr provider can now be used in the various pages of the application.

## Creating a Service for Calculating Device Location and Distance

Up until now, all the TypeScript has been general to Angular and with nothing to do with NativeScript. This geolocation application will have dependence on the location of the Android or iOS device so NativeScript must be used to natively interface with the GPS components.

Because GPS will be used throughout the application, it is a good idea to create an Angular provider for it. This will keep the code clean and maintainable.

Before creating the provider, a JavaScript library must be installed into the project.

```
npm install humanize-distance --save
```

The `humanize-distance` library allows us to calculate the distance between two latitude and longitude locations. This will be particularly useful when checking our user's device location versus that of a photo returned from Flickr.

We'll also need to include a nativescript library called `nativescript-geolocation` using the `tns` plugin command:

```
tns plugin add nativescript-geolocation
```

Let's create another service called the `geolocation.service`:

```
touch app/services/geolocation.service.ts
```

In this new file, let's include the following foundation code:

```
import { Injectable } from "@angular/core";
import * as geolocation from "nativescript-geolocation";
var humanizeDistance = require("humanize-distance");

@Injectable()
export class GeolocationService {

    public latitude: number;
    public longitude: number;

    public getLocation(): Promise<any> { }

    public getDistanceFrom(latitude: number, longitude: number): string { }

    private _getCurrentLocation(): Promise<any> { }

}
```

This provider will be injectable into the application pages. It will use the `nativescript-geolocation` plugin which interfaces with native Android and iOS GPS code. The `humanize-distance` library is imported differently because it is JavaScript rather than TypeScript.

**code/nativescript/photos-near-me/app/services/geolocation.service.ts**

```
    private _getCurrentLocation(): Promise<any> {
        return new Promise(
            (resolve, reject) => {
                geolocation.getCurrentLocation({
                    desiredAccuracy: Accuracy.high,
                    timeout: 20000
                })
                .then(location => {

                    this.latitude = location.latitude;
                    this.longitude = location.longitude;

                    resolve();
                })
                .catch(error => {
                    reject(error);
                })
            }
        );
    }
```

Using the geolocation plugin we can get the current longitude and latitude of the device GPS. This is an asynchronous request and must be added to a JavaScript promise or observable. The result of `_getCurrentLocation` will be a promise of any data.

Not all devices have GPS hardware and both Android and iOS require permissions to use location services. Because of this a few checks must be put into place.

**code/nativescript/photos-near-me/app/services/geolocation.service.ts**

```
    public getLocation(): Promise<any> {
        return new Promise(
            (resolve, reject) => {
                if (!geolocation.isEnabled()) {
                    geolocation.enableLocationRequest(true).then(() => {
                        this._getCurrentLocation()
                            .then(resolve)
                            .catch(reject);
                    });
                }
                else {
                    this._getCurrentLocation()
                        .then(resolve)
                        .catch(reject);
                }
            }
        );
    }
```

Using the `getLocation` method a check to see if the geolocation service is enabled is made. If it is not enabled, a request to enable it will be made. Provided that everything checks out, a call to the other `_getCurrentLocation` function will be made. This also applies if the geolocation service is enabled already.

With the device location in hand, a distance can be calculated from a different location, more than likely the picture distance.

**code/nativescript/photos-near-me/app/services/geolocation.service.ts**

```
    public getDistanceFrom(latitude: number, longitude: number): string {
        return humanizeDistance({ latitude: latitude, longitude: longitude }, { \
latitude: this.latitude, longitude: this.longitude }, 'en-US', 'us');
    }
```

The `getDistanceFrom` method will use the `humanize-distance` library to get us a better distance format like kilometers, miles, etc.

Like with the Flickr provider, the geolocation provider needs to be added the project's `@NgModule` block. Let's open our project's **app/app.module.ts** file and include the following lines:

```
import { GeolocationService } from "./services/geolocation.service";

@NgModule({
  // ...
  providers: [FlickrService, GeolocationService]
})
...
```

Essentially, we're importing the provider and adding it to the `providers` array of the `@NgModule` block. At this point the geolocation provider can be used throughout the application.

## Including Mapbox Functionality in the NativeScript Application

As of right now neither of the two application routes have any functionality that is particularly useful. The application has two very useful providers, but they aren't being used yet.

Since geolocation will be used, it makes sense to present a map. There are many options when it comes to maps. Two popular map solutions are Mapbox and Google Maps. For this example Mapbox renders itself the most convenient.

To install Mapbox in a NativeScript application, execute the following:

```
tns plugin add nativescript-mapbox
```

The Mapbox plugin for NativeScript has its own set of available HTML markup tags. To expose these tags in an Angular application, they must be registered in the project's **app/app.module.ts** file like so:

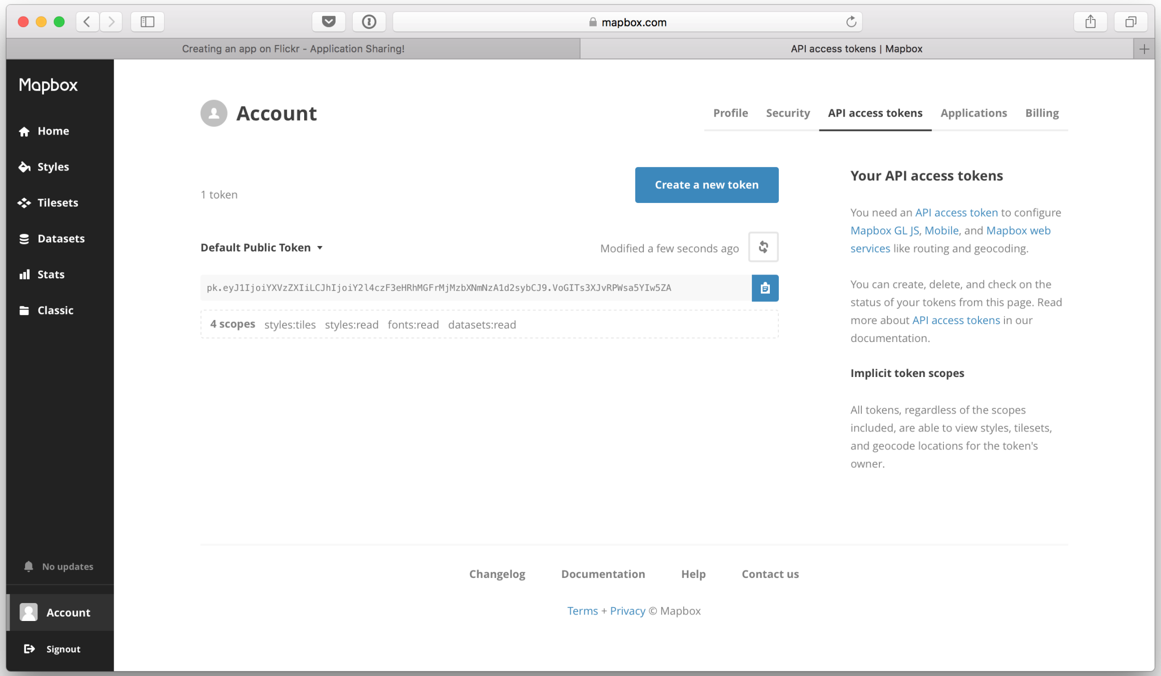
```
import { registerElement } from "nativescript-angular/element-registry";

var map = require("nativescript-mapbox");
registerElement("Mapbox", () => map.Mapbox);
```

Once registered, the `<Mapbox>` tag can be used within HTML files. However, Mapbox requires a valid API token in order to be used.

Register for an API token via the Mapbox Developers Page[163].

[163] https://www.mapbox.com/developers/



1 token

Default Public Token

Modified a few seconds ago

```
pk.eyJ1IjoiYXVzZXIiLCJhIjoiY2l4czF3eHRhMGFrMjMzbXNmNzA1d2sybCJ9.VoGITs3XJvRPWsa5YIw5ZA
```

4 scopes styles:tiles styles:read fonts:read datasets:read

**Your API access tokens**

You need an API access token to configure Mapbox GL JS, Mobile, and Mapbox web services like routing and geocoding.

You can create, delete, and check on the status of your tokens from this page. Read more about API access tokens in our documentation.

**Implicit token scopes**

All tokens, regardless of the scopes included, are able to view styles, tilesets, and geocode locations for the token's owner.

Let's store the value of the mapbox access_token open the project's **app/config.js** file. This is the same file where we added the Flickr API information. Modify this file to look like the following:

```
export const Config = {
    Flickr: {
        CLIENT_ID: "FLICKR_CLIENT_ID_HERE",
        API_URL: "https://api.flickr.com/services/rest/?"
    },
    MapBox: {
        ACCESS_TOKEN: "MAPBOX_ACCESS_TOKEN_HERE"
    }
};
```

The Mapbox API token will be obtained similarly to how the Flickr API token was obtained within the application. While Mapbox hasn't been added to the UI or the page logic, it is not able to be added.

## Implementing the First Page of the Geolocation Application

There was a lot of preparation that went into this project so far, but each of the pages are now ready to be crafted.

Open the **app/components/imageList-component/imageList.component.ts** file that was created earlier. We added foundation, but now it is time to finish it with functional logic.

```
import { Component, NgZone } from "@angular/core";
import { FlickrService } from "../../services/flickr.service";
import { PhotosSearchResponse } from "../../models/photosSearchResponse";
import { Router } from "@angular/router";
import { GeolocationService } from "../../services/geolocation.service";
import { Config } from "../../app.config";

@Component({
    selector: "ImagesListComponent",
    templateUrl: "components/imagesList-component/imagesList.component.html"
})
export class ImagesListComponent {

    private mapbox: any;
    public mapboxKey: string;
    public photos: PhotosSearchResponse[];

    public constructor(private flickrService: FlickrService, private geolocation\
Service: GeolocationService, private zone: NgZone, private router: Router) { }

    public onMapReady(args) { }

    public dropMarkers() { }

    public centerMap(args: any) { }

    public showPhoto(args: any) { }

    public loadPhotos() { }

}
```

In the above TypeScript file, each of the services and models that were previously created are now being imported into the page. The `ImagesListComponent` has a private variable which will hold the Mapbox and several public variables that will be bound to the UI.

In the `constructor` method each of the two providers are injected so they can be used throughout the current page of the application.

In a typical Angular application an `OnInit` would be used after the `constructor` method has executed. To prevent a race condition, this page will not make use of it. Instead, an `onMapReady` method will be created and used via the HTML markup. In other words, when the Mapbox thinks it's ready, this `onMapReady` method will trigger.

**code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts**

```
    public onMapReady(args) {
        this.mapbox = args.map;
        this.geolocationService.getLocation().then(() => {
            this.loadPhotos().subscribe(
                photos => {
                    this.photos = photos.map((photo) => {
                        photo.distance = this.geolocationService.getDistanceFrom(
                            parseFloat(photo.latitude),
                            parseFloat(photo.longitude));
                        return photo;
                    });
                    this.dropMarkers();
                    this.mapbox.setCenter({
                        lat: this.geolocationService.latitude,
                        lng: this.geolocationService.longitude,
                        animated: true
                    });
                },
                error => console.log(error));
            });
    }
```

Once triggered, the `mapbox` variable will be set with the current Mapbox. Using the geolocation service, the device GPS location is obtained and Flickr photos near the location are queried. A humanized distance is calculated for each of the photos retrieved from the API call.

The photos obtained from the Flickr API are stored in the `photos` array at which point they are placed as markers on the map using the `dropMarkers` method. At the end of the initialization period the map is centered on the devices location.

The `dropMarkers` method called from the `onMapReady` looks like the following:

**code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts**

```
    public dropMarkers() {
        let markers = this.photos.map((photo: PhotosSearchResponse, index: numbe\
r) => {
            return {
                lat: photo.latitude,
                lng: photo.longitude,
                onTap: () => {
                    this.zone.run(() => {
                        this.showPhoto({ index: index });
                    });
                }
            }
        });
        this.mapbox.addMarkers(markers);
    }
```

In the above method, the `photos` array is recreated through a JavaScript `map` and stored as `markers`. The new objects found in the array include the longitude and latitude of the photo and a tap event, `showPhoto`, which will navigate to the next page. To keep everything in sync, the `showPhoto` method must be added within the Angular zone.

The `markers` array is added to the Mapbox for display on the soon to be created map component.

**code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts**

```
    public showPhoto(args: any) {
        let photo = this.photos[args.index];
        this.router.navigate(["/image-component", photo.id]);
    }
```

The route to the second page of the application requires a photo id. This information is obtained from a specific photo that was selected. Remember, the photo information was added within the `dropMarkers` method.

**code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts**

```
    public loadPhotos() {
      return this.flickrService.photosSearch(
        this.geolocationService.latitude,
        this.geolocationService.longitude);
    }
```

The `loadPhotos` method was used in the `onMapReady` method for subscribing to the Flickr observable. It was created to make the lines of the file shorter and easier to read.

The final method of the first application page, `centerMap`, will center the map on a particular photo:

**code/nativescript/photos-near-me/app/components/imagesList-component/imagesList.component.ts**

```
    public centerMap(args: any) {
        let photo = this.photos[args.index];
        this.mapbox.setCenter({
            lat: parseFloat(photo.latitude),
            lng: parseFloat(photo.longitude),
            animated: false
        });
    }
```

So what does the UI markup look like for the TypeScript logic that was just implemented? Open the project's **app/components/imageList-component/imageList.component.html** file. The UI is composed of two vertical sections, a list of pictures which resides on the upper level and a map which resides on the lower level.

```
<ActionBar title="Photos" class="action-bar"></ActionBar>
<StackLayout>
    <GridLayout columns="*" rows="*, 280">
        <ListView [items]="photos" row="0" col="0" class="list-group" (itemTap)=\
"centerMap($event)">
        </ListView>
        <ContentView row="1" col="0">
        </ContentView>
    </GridLayout>
</StackLayout>
```

In the above markup a `GridLayout` will allow for vertical sections, hence the two row values. The asterisk in the `columns` means that each row will take the full width of the screen. Since there is an asterisk and numeric value in the `rows`, the bottom row will have a height of `280` and the top row will take all remaining part of the screen.

The `ListView` is setup to iterate over each element in the public `photos` array. When a row is tapped, the `centerMap` method is called to position the map over the photo that was clicked. The `ContentView` is the second row and it will hold the map.

The `ListView` is incomplete though. It should really look like the following:

```
<ListView [items]="photos" row="0" col="0" class="list-group" (itemTap)="centerM\
ap($event)">
    <template let-item="item">
        <GridLayout columns="auto, *" rows="auto" class="list-group-item">
            <Image [src]="item.url_q" width="50" height="50" col="0" class="thum\
b img-rounded"></Image>
            <StackLayout row="0" col="1">
                <Label [text]="item.distance + ' away'" class="list-group-item-h\
eading"></Label>
                <Label [text]="item.title" class="list-group-item-text" textWrap\
="true"></Label>
            </StackLayout>
        </GridLayout>
    </template>
</ListView>
```

Each row of the `ListView` will have two columns and an automatically sized row height. The first column of the list row will be the image returned from the Flickr API and the second column will have stacked text which includes the photo title and the humanized distance.

```
<ContentView row="1" col="0">
    <Mapbox
        accessToken="{{ mapboxKey }}"
        mapStyle="streets"
        zoomLevel="17"
        hideLogo="true"
        showUserLocation="true"
        (mapReady)="onMapReady($event)">
    </Mapbox>
</ContentView>
```

The Mapbox calls the `onMapReady` function and uses the `mapboxKey` found in the configuration file. Other default properties are used as well.

The first, and default, page of the application is now complete. However, a page for showing the picture still needs to be completed. This is the page navigated to after tapping a marker on the map.

## Implementing the Second Page of the Geolocation Application

The second and final page of the application will show an image based on what was selected in the previous page. Open the project's **app/components/image-component/image.component.ts** file and include the following TypeScript code:

```
import { Component, OnInit } from "@angular/core";
import { ActivatedRoute } from "@angular/router";
import { FlickrService } from "../../services/flickr.service";

@Component({
    templateUrl: "components/image-component/image.component.html"
})
export class ImageComponent implements OnInit {

    public url: string;

    public constructor(private activatedRoute: ActivatedRoute, private flickrSer\
vice: FlickrService) { }

    public ngOnInit() { }

    public getPhoto(photoId: number) { }

}
```

The Flickr provider was imported to what was created previously and it is injected into the `constructor` method. The `url` variable will hold the image URL that will be bound to and presented in the UI.

**code/nativescript/photos-near-me/app/components/image-component/image.component.ts**

```
    public ngOnInit() {
        this.activatedRoute.params.subscribe(params => {
            let photoId = params["photo_id"];
            this.getPhoto(photoId);
        });
    }
```

When this page is initialized, the `ngOnInit` method will obtain the URL parameter and pass it to the `getPhoto` message.

**code/nativescript/photos-near-me/app/components/image-component/image.component.ts**

```
    public getPhoto(photoId: number) {
        this.flickrService.getPhotoInfo(photoId).subscribe(
            photo => {
                this.url = `https://farm${photo.farm}.staticflickr.com/${photo.s\
erver}/${photo.id}_${photo.secret}_n.jpg`;
            },
            error => console.log(error)
        );
    }
```

After making a request to the Flickr API with the Flickr provider, the public `url` variable will be filled.

With the logic in place, open the project's **app/components/image-component/image-component.html** file and include the following markup:

```
<ActionBar></ActionBar>
<StackLayout>
    <Image [src]="url" width="360" height="360"></Image>
</StackLayout>
```

The `Image` tag will present an image based on the `url` that was populated in the TypeScript code. Within the action bar, there will be a back button to navigate to the previous page.

## Try it out!

Now that we have the basic structure for our app in place, try running:

```
tns livesync android --emulator --watch
# or
tns livesync ios --emulator --watch
```

When you're ready to create a build call:

```
tns build android
# or
tns build ios
```

## NativeScript for Angular Developers

NativeScript makes it very easy for Angular developers to develop native mobile applications that use native device features, SDKs, and concepts. As technology evolves for the best, the need to know Java or Objective-C is dwindling in favor of these cross platform mobile development frameworks.

Obviously there is a lot more to learn about using NativeScript than we can cover in just this first chapter. Checkout these resources:

- Official NativeScript Site[164]
- Official NativeScript Docs[165]
- NativeScript App Examples[166]
- NativeScript on StackOverflow[167]

[164]https://www.nativescript.org/
[165]https://docs.nativescript.org/
[166]https://www.nativescript.org/app-samples-with-code
[167]http://stackoverflow.com/questions/tagged/nativescript