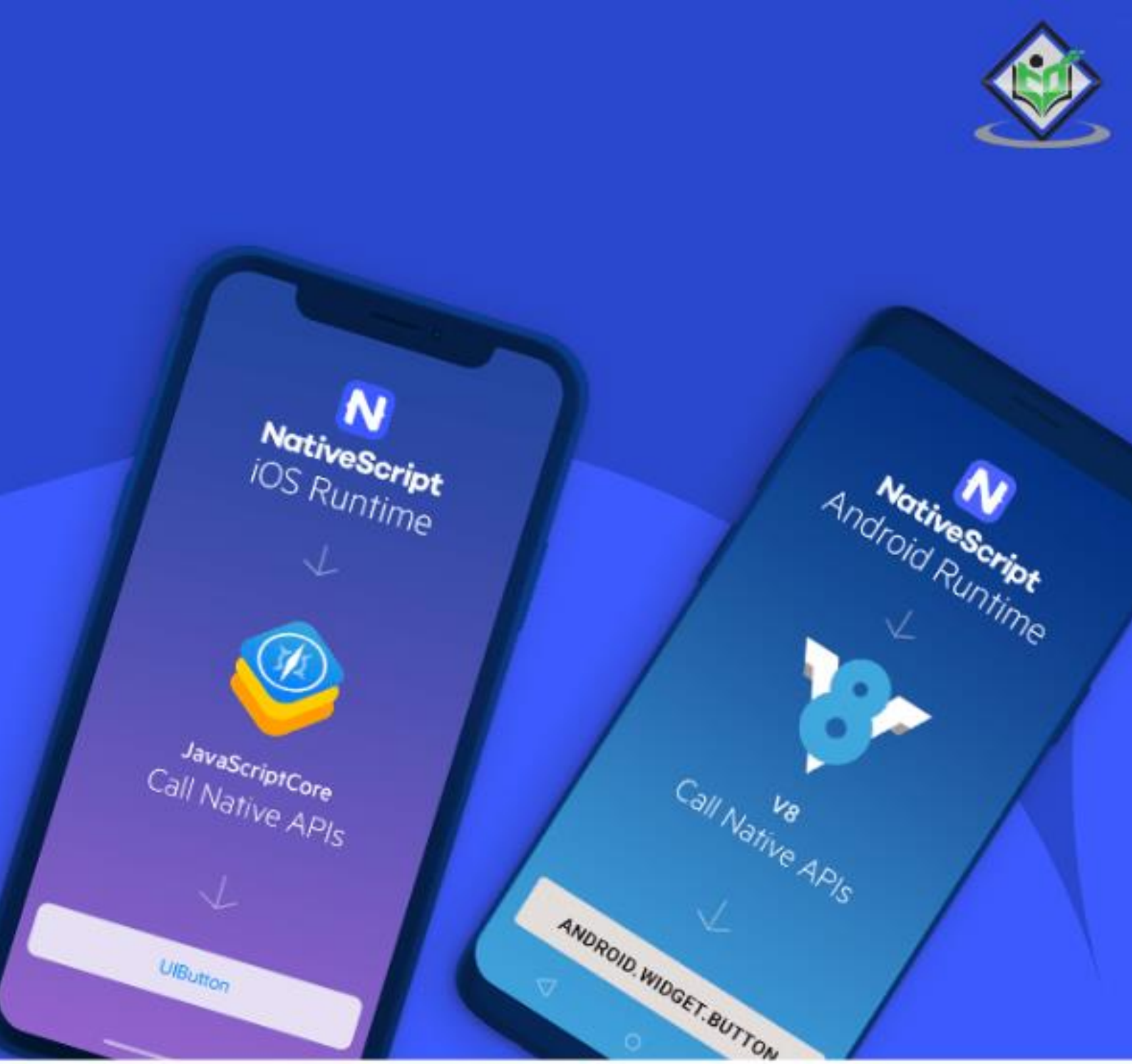


tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

NativeScript is an open source framework for creating native iOS and Android apps in Angular, TypeScript, or JavaScript. It was developed by *Progress Telerik*. NativeScript allows you to build web and mobile apps from a single code-base. This tutorial walks through the basics of NativeScript framework, installation of NativeScript CLI, setting up Android Studio and iOS to develop NativeScript based application, architecture of NativeScript framework and finally conclude with developing all type of web and mobile apps.

Audience

This tutorial is prepared for professionals who are aspiring to make a career in the field of mobile applications. This tutorial is intended to make you comfortable in getting started with NativeScript framework and its various functionalities.

Prerequisites

This tutorial is written assuming that the readers are already aware about what a Framework is and that the readers have a sound knowledge on Object Oriented Programming and basic knowledge on Android and iOS development, JavaScript and Angular. If you are a beginner to any of these concepts, we suggest you to go through tutorials related to these first, before you start with NativeScript.

Copyright & Disclaimer

Table of Contents

1. NativeScript — Introduction

Generally, developing a mobile application is a complex and challenging task. There are many frameworks available to develop a mobile application. Android provides a native framework based on Java language and iOS provides a native framework based on Objective-C/Shift language. However, to develop an application that support both operating systems, we need to code in two different languages using two different frameworks.

To overcome this complexity, mobile frameworks supports this feature. The main reason behind to use cross-platform or hybrid framework is easier to maintain a single code base. Some of the popular frameworks are NativeScript, Apache Cordova, Xamarin, etc.

Overview of JavaScript Frameworks

JavaScript is a multi-paradigm language. It supports functional programming, object-oriented and prototype based programming. JavaScript was initially used for the client-side. Nowadays, JavaScript is used as a server-side programming language as well. JavaScript frameworks are a type of tool that makes working with JavaScript easier and smoother.

Using this framework, programmers can easily code the application as a device responsive. Responsiveness is one of the reasons behind why this framework is becoming very popular.

Let us have a look at some of the popular JS frameworks:

Angular

One of the most powerful, efficient, and open-source JavaScript frameworks is Angular. We can build mobile and desktop applications. Google uses this framework. It is used for developing a Single Page Application (SPA).

Vue.js

VueJS is a progressive JavaScript framework used to develop interactive web interfaces. It is one of the famous frameworks used to simplify web development. It can be easily integrated into big projects for front-end development without any issues. It is dual integration mode is one of the most attractive features for the creation of high-end SPA or Single Page Application.

React

ReactJS is JavaScript library used for building reusable UI components. It is developed by Facebook. It is currently one of the most popular JavaScript libraries and has a strong foundation and large community behind it.

Node.js

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. It is built on Google Chrome's JavaScript Engine (V8 Engine). Node.js applications are written in JavaScript, and can be run on OS X, Microsoft Windows,

and Linux. It provides a rich library of various JavaScript modules which simplifies the development of web applications.

Overview of NativeScript

NativeScript is an open source framework used for creating native iOS and android mobile applications. It is a JIT compiled framework. NativeScript code runs on JS virtual machine. It uses V8 engine runtime for both Android and iOS platforms. NativeScript uses XML, JS and CSS for development. It has a WebIDE known as PlayGround. This PlayGround supports easy working interface, easy to manage projects, hot reload and debug on devices.

NativeScript allows developers to create native, cross-platform apps quickly and efficiently and save on the costs of development, testing, and training. Hence, Native apps will continue to be rich and strong for years to come to make better and easier to use.

Features

NativeScript has vibrant community support. Some of the salient features of NativeScript listed below:

- Extensible
- Hot Module Replacement
- Easy to setup
- We can build rich animations, graphs, charts and lists
- Developers can use any view as the root of an application
- Lazy coding

Benefits

NativeScript helps small or large scale companies to build cross-platform mobile apps. Some of the key benefits are:

- Developers can reuse existing tools and code
- Easy to modify, troubleshoot and update newer versions
- Development experience is good so we don’t have to spend time to learn new tools
- Platform-specific APIs from JavaScript, eliminating the need to learn Cordova plugins
- Ease authentication with different sign-on providers

2. NativeScript — Installation

This section explains about how to install NativeScript on your machine.

Prerequisites

Before moving to installation, we need the following prerequisites:

- Node.js
- Android
- iOS

Verify Node.js

Node.js is a JavaScript runtime engine build on top of Google Chrome's internal JavaScript engine, *v8*. NativeScript uses *Node.js* extensively for various purpose like creating the starter template application, compiling the application, etc., It is mandatory to have *Node.js* on your machine.

Hopefully, you have installed *Node.js* on your machine. If it is not installed then visit the link, https://nodejs.org/ and download the latest LTS package and install it.

To verify if *Node.js* is properly installed, type the below command on your terminal:

```
node --version
```

You could see the version. As of now, the current stable "LTS" version of node is 12.14.0.

CLI setup

NativeScript CLI is a terminal/command line based application and allows you to create and develop NativeScript application. Node.js package manager npm is used to install NativeScript CLI on your machine.

Use the below command to install NativeScript CLI:

```
npm install -g nativescript
```

After executing this command, we could see the following output:

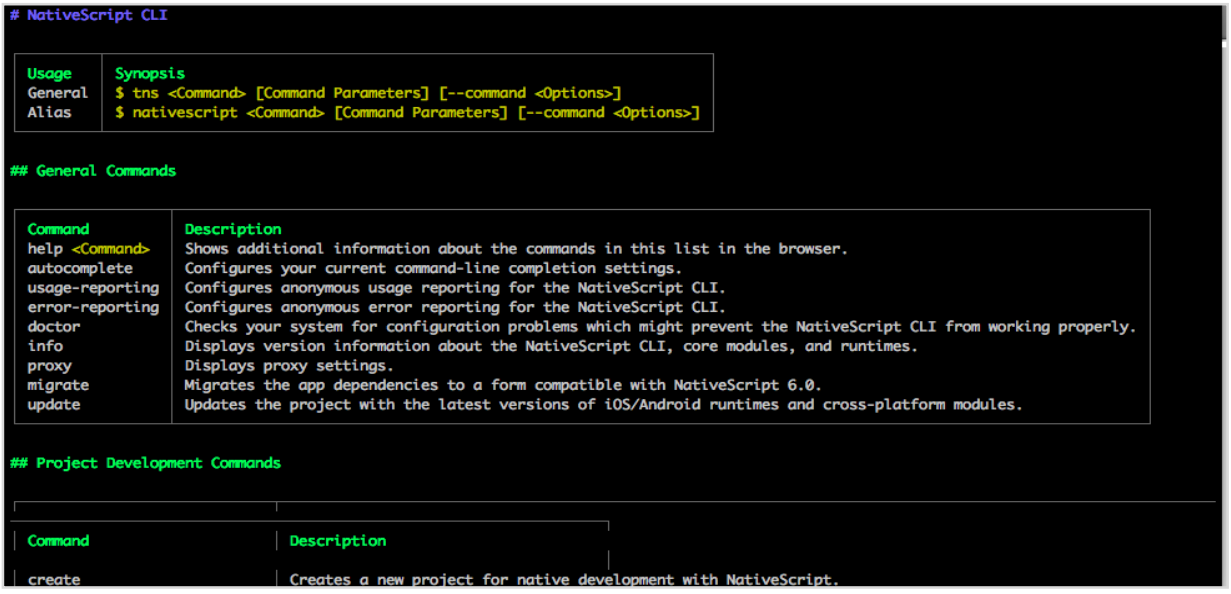
```
> node postinstall.js

Autocompletion is already enabled

You have successfully installed the NativeScript CLI!

Your next step is to create a new project:
tns create

New to NativeScript? Try the tutorials in NativeScript Playground: https://play.nativescript.org


If you have any questions, check Stack Overflow: https://stackoverflow.com/questions/tagged/nativescript and our public
s://nativescriptcommunity.slack.com/

+ nativescript@6.2.2
added 1058 packages from 1385 contributors in 26.593s
```

setupcli

We have installed the latest NativeScript CLI, `tns` in our system. Now, type the below command in your terminal:

```
tns
```

This will list out quick-start guide. You could see the following output:



cli

We can use tns to create and develop application even without any additional setup. But, we could not able to deploy the application in real device. Instead we can run the application using *NativeScript PlayGround* iOS / Android application. We will check it in the upcoming chapters.

Installing NativeScript playground App

Go to your iOS App store or Google Play Store and search NativeScript Playground app. Once the application is listed in the search result, click the install option. It will install the *NativeScript Playground* app in our device.

NativeScript Playground application will be helpful for testing your apps in Android or iOS device without deploying the application in the real device or emulator. This will reduce the time to develop the application and easy way to kick-start the development of our mobile application.

Android and iOS setup

In this chapter, let us learn how to setup the system to build and run iOS and Android apps either in emulator or in real device.

Step 1: Windows dependency

Execute the below command in your windows command prompt and run as administrator:

```
@powershell -NoProfile -ExecutionPolicy Bypass -Command "iex
((new-object
net.webclient).DownloadString('https://www.nativescript.org/setup/win'))"
```

After this command, Scripts being downloaded then install the dependencies and configure it.

Step 2: macOS dependency

To install in macOS, you must ensure that Xcode is installed or not. Xcode is mandatory for NativeScript. If Xcode is not installed, then visit the following link https://developer.apple.com/xcode/ and download; then install it.

Now execute the following command in your terminal:

```
sudo ruby -e "$(curl -fsSL https://www.nativescript.org/setup/mac)"
```

After executing the above command, script will install the dependencies for both iOS and Android development. Once it is done, close and restart your terminal.

Step 3: Android dependency

Hopefully, you have configured the following prerequisites:

- JDK 8 or higher
- Android SDK
- Android Support Repository
- Google Repository
- Android SDK Build-tools 28.0.3 or higher
- Android Studio

If the above prerequisites are not configured, then visit the following link https://developer.android.com/studio/install and install it. Finally, Add JAVA_HOME and ANDROID_HOME in your environment variables.

Step 4: Verify dependencies

Now everything is done. You can verify the dependency using the below command:

```
tns doctor
```

This will verify all the dependency and summarize the result as below:

```
√ Getting environment information

No issues were detected.
√ Your ANDROID_HOME environment variable is set and points to correct
directory.
√ Your adb from the Android SDK is correctly installed.
√ The Android SDK is installed.
√ A compatible Android SDK for compilation is found.
√ Javac is installed and is configured properly.
√ The Java Development Kit (JDK) is installed and is configured properly.
√ Local builds for iOS can be executed only on a macOS system. To build for iOS
on a different operating system, you can use th
e NativeScript cloud infrastructure.
√ Getting NativeScript components versions information...
√ Component nativescript has 6.3.0 version and is up to date.
√ Component tns-core-modules has 6.3.2 version and is up to date.
√ Component tns-android has 6.3.1 version and is up to date.
√ Component tns-ios has 6.3.0 version and is up to date.
```

If you find any issues, please correct the issues before proceeding to develop the application.

3. NativeScript — Architecture

NativeScript is an advanced framework to create mobile application. It hides the complexity of creating mobile application and exposes a rather simple API to create highly optimized and advanced mobile application. NativeScript enables even entry level developers to easily create mobile application in both Android and iOS.

Let us understand the architecture of the NativeScript framework in this chapter.

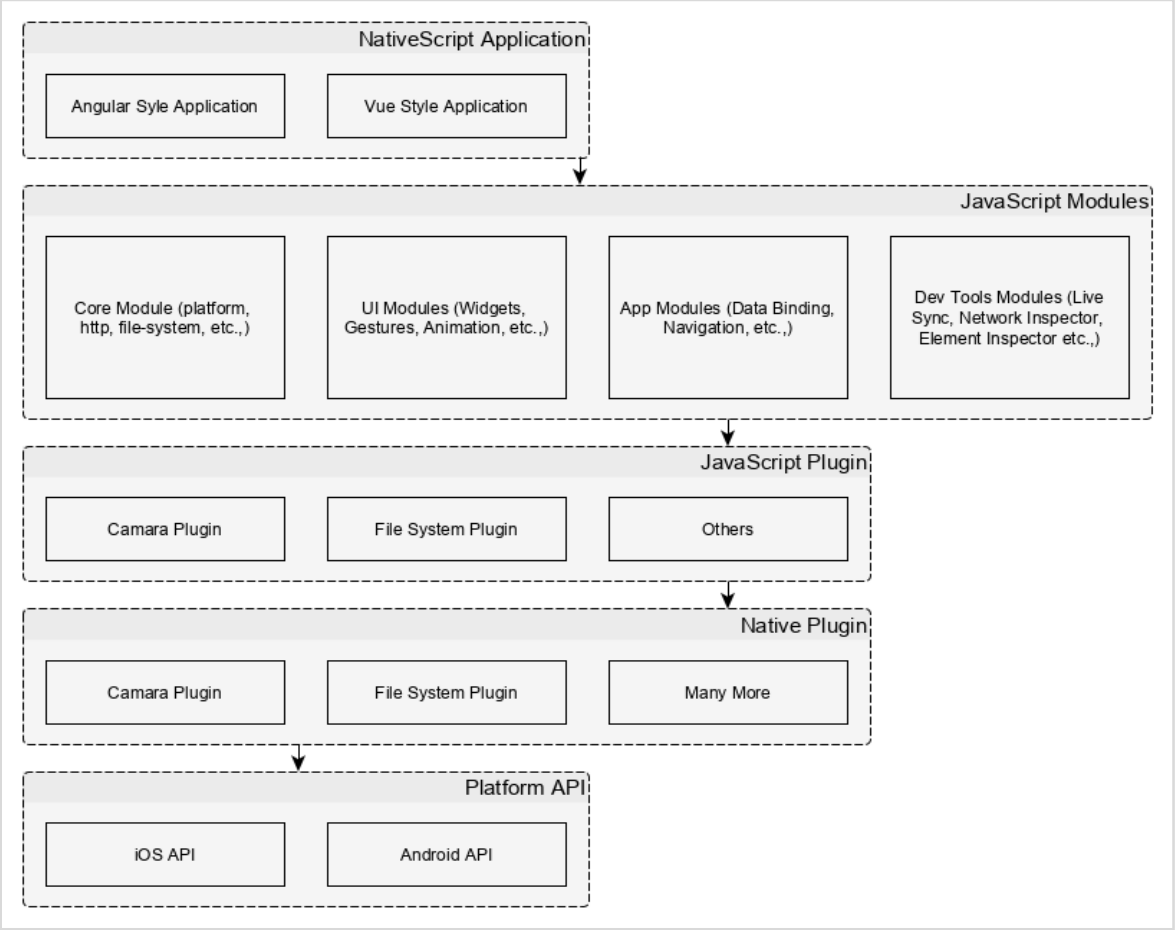
Introduction

The core concept of NativeScript framework is to enable the developer to create hybrid style mobile application. Hybrid application uses the platform specific browser API to host a web application inside a normal mobile application and provides system access to the application through JavaScript API.

NativeScript invests heavily on the **JavaScript language** to provide an efficient framework for developers. Since **JavaScript** is de-facto standard for client side programming (Web development) and every developer is well aware of the JavaScript language, it helps developers to easily get into the NativeScript framework. At the low level, NativeScript exposes the *native API* through a collection of JavaScript plugins called **Native plugins**.

NativeScript builds on the foundation of *Native plugins* and provides many high level and easy to use *JavaScript Modules*. Each module does a specific functionality like accessing a camera, designing a screen, etc. All these modules can be combined in multiple ways to architect a complex mobile application.

Below diagram shows the high level overview of the NativeScript framework:



NativeScript Application: NativeScript framework allows developer to use either Angular style application or Vue Style application.

JavaScript Modules: NativeScript framework provides a rich set of JavaScript modules clearly categorized as UI Modules, Application Modules, Core Modules, etc. All modules can be accessed by application at any time to write any level of complex application.

JavaScript plugins: NativeScript framework provides a large collection of JavaScript plugins to access the platform related functionality. Modules uses the JavaScript plugins to provide platform specific functionality.

Native plugins: Native plugins are written in platform specific language to wrapper the system functionality which will be further used by JavaScript plugin.

Platform API: API provided by platform vendors.

In short, NativeScript application is written and organized using modules. Modules are written in pure JavaScript and the modules access the platform related functionality (whenever needed) through plugins and finally, the plugins bridge the platform API and JavaScript API.

Workflow of a NativeScript Application

As we learned earlier, NativeScript application is composed of modules. Each and every module enables a specific feature. The two important categories of module to bootstrap a NativeScript application are as follows:

- Root Modules
- Page Modules

Root and Page modules can be categorized as application modules. The application module is the entry point of the NativeScript application. It bootstraps a page, enables the developer to create user interface of the page and finally allows execution of the business logic of the page. An application module consists of below three items:

- User interface design coded in XML (e.g. page.xml/page.component.html)
- Styles coded in CSS (e.g. page.css/page.component.css)
- Actual business logic of the module in JavaScript (e.g. page.js/page.component.ts)

NativeScript provides lot of UI components (under UI Module) to design the application page. UI Component can be represented in XML format or HTML format in Angular based application. Application module uses the UI Component to design the page and store the design in separate XML, *page.xml/page.component.html*. The design can be styled using standard CSS.

Application modules stores the style of the design in separate CSS, *page.css/page.component.css*. The functionality of the page can be done using JavaScript/TypeScript, which has full access to the design as well as the platform features. Application module uses a separate file, *page.js/page.component.ts* to code the actual functionality of the page.

Root Modules

NativeScript manages the user interface and user interaction through UI containers. Every UI container should have a *Root Module* and through which the UI container manages UI. NativeScript application have two type of UI containers:

Application Container: Every NativeScript application should have one application container and it will be set using application.run() method. It initializes the UI of the application.

Model View Container: NativeScript manages the Modal dialogs using model view container. A NativeScript application can have any number of model view container.

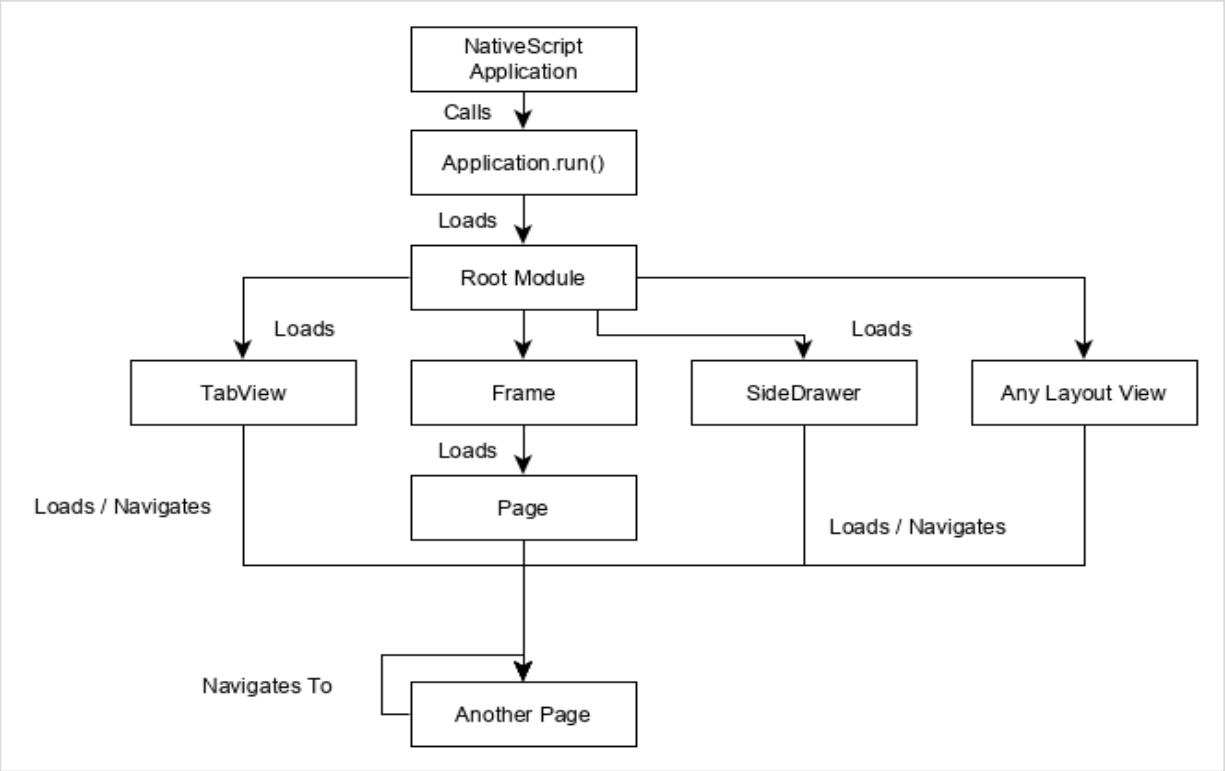
Every root module should have only one UI Component as its content. The UI component in turn can have other UI components as its children. NativeScript provides a lot of UI component like TabView, ScrollView, etc., with child feature. We can use these as root UI component. One exception is *Frame*, which does not have child option but can be used as root component. *Frame* provides options to load *Page Modules* and options to navigate to other page modules as well.

Page Modules

In NativeScript, each and every page is basically a *Page Module*. Page module is designed using the rich set of UI components provided by NativeScript. Page modules are loaded

into the application through *Frame* component (using its defaultPage attribute or using navigate() method), which in turn loaded using *Root Modules*, which again in turn loaded using application.run() while the application is started.

The work flow of the application can be represented as in the below diagram:



The above diagram is explained in detail in the following steps:

1. NativeScript Application starts and calls application.run() method.
2. application.run() loads a *Root module.*
3. *Root Module* is designed using any one of the UI component as specified below:
   - Frame
   - TabView
   - SideDrawer
   - Any Layout View
4. Frame component loads the specified page (Page module) and gets rendered. Other UI components will be rendered as specified in the *Root Module*. Other UI component also has option to load *Page Modules* as its main content.
5. Finally, user can navigate to other pages using UI component's navigation option.

Workflow of Angular based NativeScript Application

As we learned earlier, NativeScript framework provides multiple methodologies to cater different category of developers. The methodologies supported by NativeScript are as follows:

- NativeScript Core: Basic or core concept of NativeScript Framework

- Angular + NativeScript: Angular based methodology
- Vuejs + NativeScript: Vue.js based methodology

Let us learn how Angular framework is incorporated into the NativeScript framework.

Step 1

NativeScript provides an object (platformNativeScriptDynamic) to bootstrap the Angular application. platformNativeScriptDynamic has a method, bootstrapModule, which is used to start the application.

The syntax to bootstrap the application using Angular framework is as follows:

```
import { platformNativeScriptDynamic } from "nativescript-angular/platform";
import { AppModule } from "./app/app.module";

platformNativeScriptDynamic().bootstrapModule(AppModule);
```

Here,

AppModule is our Root module.

Step 2

A simple implementation (below specified code) of the app module.

```
import { NgModule } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";

import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";

@NgModule({
    bootstrap: [
        AppComponent
    ],
    imports: [
        NativeScriptModule,
        AppRoutingModule
    ],
    declarations: [
        AppComponent
    ]
})
export class AppModule { }
```

Here,

AppModule starts the application by loading AppComponent component. Angular components are similar to pages and are used for both design and programming logic.

Step 3

A simple implementation of AppComponent (app.component.ts) and its presentation logic (app.component.css) is as follows:

app.component.ts

```
import { Component } from "@angular/core";

@Component({
    selector: "ns-app",
    templateUrl: "app.component.html"
})
export class AppComponent { }
```

Here,

templateUrl refers the design of the component.

app.component.html

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

Here,

page-router-outlet is the place where the Angular application get attached.

In summary, Angular framework is composed of modules similar to NativeScript framework with slight differences. Each module in the Angular will have an Angular component and a router setup file (page-routing.mocdule.ts). The router is set per module and it takes care of navigation. Angular components are analogues to pages in NativeSctipt core.

Each component will have a UI design (page.component.html), a style sheet (page.component.css), and a JavaScript/TypeScript code file (page.component.ts).

4. NativeScript — Angular Application

Let us create a simple bare bone application to understand the work flow of the NativeScript application.

Creating the Application

Let us learn how to create simple application using NativeScript CLI, tns. tns provides a command create to used to create a new project in NativeScript.

The basic syntax to create a new application is as below:

```
tns create <projectname> --template <template_name>
```

Where,

- ***Projectname*** is the Name of the project.
- ***template_name*** is Project template. NativeScript provides lot of startup template to create different type of application. Use Angular based template.

Let us create a new directory named NativeScriptSamples to work on our new application. Now, open a new terminal then move to our directory and type the below command:

```
tns create BlankNgApp --template tns-template-blank-ng
```

Where, ***tns-tempalte-blank-ng*** refers a blank mobile application based on AngularJS.

Output

```
.....
.....
.....

Project BlankNgApp was successfully created.

Now you can navigate to your project with $ cd BlankNgApp

After that you can preview it on device by executing $ tns preview
```

Now, our first mobile application, *BlankNgApp* is created.

Structure of the Application

Let us understand the structure of the NativeScript application by analyzing our first application BlankNgApp in this chapter. NativeScript application is organized into multiple sections and they are as follows:

- Configuration section
- Node modules

- Android sources
- iOS Sources
- Application source code

The general structure of the application is as follows:

```
│   angular.json
│   LICENSE
│   nsconfig.json
│   package-lock.json
│   package.json
│   tsconfig.json
│   tsconfig.tns.json
│   tsfmt.json
│   webpack.config.js
│
├───App_Resources
│   ├───Android
│   │
│   └───iOS
│
├───hooks
│
├───node_modules
│
└───src
    │   app.css
    │   main.ts
    │   package.json
    │
    └───app
        │   app-routing.module.ts
        │   app.component.html
        │   app.component.ts
        │   app.module.ts
        │
        └───home
                home-routing.module.ts
                home.component.html
                home.component.ts
                home.module.ts
```

Let us understand each section of the application and how it helps us to create our application.

Configuration section

All the files in the root of the application are configuration files. The format of the configuration files are in JSON format, which helps the developer to easily understand the configuration details. NativeScript application relies on these files to gets all available configuration information. Let us go through all the configuration files in this section.

package.json

package.json files sets the identity (id) of the application and all the modules that the application depends on for its proper working. Below is our package.json:

```
{
  "nativescript": {
    "id": "org.nativescript.BlankNgApp",
    "tns-android": {
      "version": "6.3.1"
    },
    "tns-ios": {
      "version": "6.3.0"
    }
  },
  "description": "NativeScript Application",
  "license": "SEE LICENSE IN <your-license-filename>",
  "repository": "<fill-your-repository-here>",
  "dependencies": {
    "@angular/animations": "~8.2.0",
    "@angular/common": "~8.2.0",
    "@angular/compiler": "~8.2.0",
    "@angular/core": "~8.2.0",
    "@angular/forms": "~8.2.0",
    "@angular/platform-browser": "~8.2.0",
    "@angular/platform-browser-dynamic": "~8.2.0",
    "@angular/router": "~8.2.0",
    "@nativescript/theme": "~2.2.1",
    "nativescript-angular": "~8.20.3",
    "reflect-metadata": "~0.1.12",
    "rxjs": "^6.4.0",
    "tns-core-modules": "~6.3.0",
    "zone.js": "~0.9.1"
  },
  "devDependencies": {
    "@angular/compiler-cli": "~8.2.0",
    "@ngtools/webpack": "~8.2.0",
    "nativescript-dev-webpack": "~1.4.0",
    "typescript": "~3.5.3"
  },
  "gitHead": "fa98f785df3fba482e5e2a0c76f4be1fa6dc7a14",
  "readme": "NativeScript Application"
}
```

Here,

Identity of the application (nativescript/id): Sets the id of the application as org.nativescript.BlankNgApp. This id is used to publish our app to the Play Store or iTunes. This id will be our Application Identifier or Package Name.

Dependencies (dependencies): Specifies all our dependent node modules. Since, the default NativeScript implementation depends on Angular Framework, Angular modules are included.

Development dependencies: Specifies all the tools that the application depends on. Since, we are developing our application in TypeScript, it includes typescript as one of the dependent modules.

angular.json: Angular framework configuration information.

nsconfig.json: NativeScript framework configuration information.

tsconfig.json, tsfmt.json & tsconfig.tns.json: TypeScript language configuration information

webpack.config.js: WebPack configuration written in JavaScript.

Node modules

As NativeScript project are node based project, it stores all its dependencies in the node_modules folder. We can use npm (npm install) or tns to download and install all the application dependency into the node_moduels.

Android source code

NativeScript auto-generates the android source code and place it in App_Resources\Android folder. It will be used to create android application using *Android SDK*

iOS source code

NativeScript auto-generates the iOS source code and place it in App_Resources\iOS folder. It will be used to create iOS application using *iOS SDK* and *XCode*

Application source code

The actual application code is placed in src folder. Our application has below files in src folder.

```
└──src
    │   app.css
    │   main.ts
    │   package.json
    │
    └──app
        │   app-routing.module.ts
        │   app.component.html
        │   app.component.ts
        │   app.module.ts
        │
        └──home
                home-routing.module.ts
                home.component.html
                home.component.ts
                home.module.ts
```

Let us understand the purpose of all files and how they are organized in this section:

Step 1

main.ts - Entry point of the application.

```
// this import should be first in order to load some required settings (like
globals and reflect-metadata)
import { platformNativeScriptDynamic } from "nativescript-angular/platform";

import { AppModule } from "./app/app.module";

platformNativeScriptDynamic().bootstrapModule(AppModule);
```

Here, we have set the AppModule as the bootstrapping module of the application.

Step 2

app.css - Main style sheet of the application is as shown below:

```
@import "~@nativescript/theme/css/core.css";
@import "~@nativescript/theme/css/brown.css";

/* Place any CSS rules you want to apply on both iOS and Android here.
This is where the vast majority of your CSS code goes. */
```

Here,

app.css imports the core style sheet and brown color themes style sheet of the NativeScript framework.

Step 3

app\app.module.ts - Root module of the application.

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";

import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";

@NgModule({
    bootstrap: [
        AppComponent
    ],
    imports: [
        NativeScriptModule,
        AppRoutingModule
    ],
    declarations: [
        AppComponent
    ],
    schemas: [
        NO_ERRORS_SCHEMA
    ]
})
export class AppModule { }
```

Here,

AppModule is created based on NgModule and sets the components and modules of the application. It imports two modules NativeScriptModule and AppRoutingModule and a component, AppComponent. It also set the AppComponent as the root component of the application.

Step 4

app.component.ts - Root component of the application.

```
import { Component } from "@angular/core";

@Component({
    selector: "ns-app",
    templateUrl: "app.component.html"
})
export class AppComponent { }
```

Here,

AppComponent sets the template and style sheet of the component. Template is designed in plain HMTL using NativeScript UI components.

Step 5

app-routing.module.ts - Routing module for the AppModule

```
import { NgModule } from "@angular/core";
import { Routes } from "@angular/router";
import { NativeScriptRouterModule } from "nativescript-angular/router";

const routes: Routes = [
    { path: "", redirectTo: "/home", pathMatch: "full" },
    { path: "home", loadChildren: () =>
import("~/app/home/home.module").then((m) => m.HomeModule) }
];

@NgModule({
    imports: [NativeScriptRouterModule.forRoot(routes)],
    exports: [NativeScriptRouterModule]
})
export class AppRoutingModule { }
```

Here,

AppRoutingModule uses the NativeScriptRouterModule and sets the routes of the AppModule. It basically redirects the empty path to /home and the points the /home to HomeModule.

Step 6

app\home\home.module.ts - Defines a new module, HomeModule.

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptCommonModule } from "nativescript-angular/common";

import { HomeRoutingModule } from "./home-routing.module";
import { HomeComponent } from "./home.component";

@NgModule({
    imports: [
        NativeScriptCommonModule,
        HomeRoutingModule
    ],
    declarations: [
        HomeComponent
    ],
    schemas: [
        NO_ERRORS_SCHEMA
    ]
})
export class HomeModule { }
```

Here,

HomeModule imports two modules, HomeRoutingModule and NativeScriptCommonModule and one component HomeComponent

Step 7

app\home\home.component.ts - Defines the Home component and used as home page of the application.

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

@Component({
    selector: "Home",
    templateUrl: "./home.component.html"
})
export class HomeComponent implements OnInit {

    constructor() {
        // Use the component constructor to inject providers.
    }

    ngOnInit(): void {
        // Init your component properties here.
    }
}
```

Here,

HomeComponent sets the template and selector of the home component.

Step 8

app\home\home-routing.module.ts - Routing module for HomeModule and used to define routing for home module.

```
import { NgModule } from "@angular/core";
import { Routes } from "@angular/router";
import { NativeScriptRouterModule } from "nativescript-angular/router";

import { HomeComponent } from "./home.component";

const routes: Routes = [
    { path: "", component: HomeComponent }
];

@NgModule({
    imports: [NativeScriptRouterModule.forChild(routes)],
    exports: [NativeScriptRouterModule]
})
export class HomeRoutingModule { }
```

Here,

HomeRoutingModule set the empty path to HomeComponent.

Step 9

app.component.html and home.component.html - They are used to design the UI of the application using NativeScript UI components.

Run your app

If you want run your app without using any device, then type the below command:

```
tns preview
```

After executing this command, this will generate QR code to scan and connect with your device.

Output

QRCode

Now QR code is generated and connect to PlayGround in next step.

NativeScript PlayGround

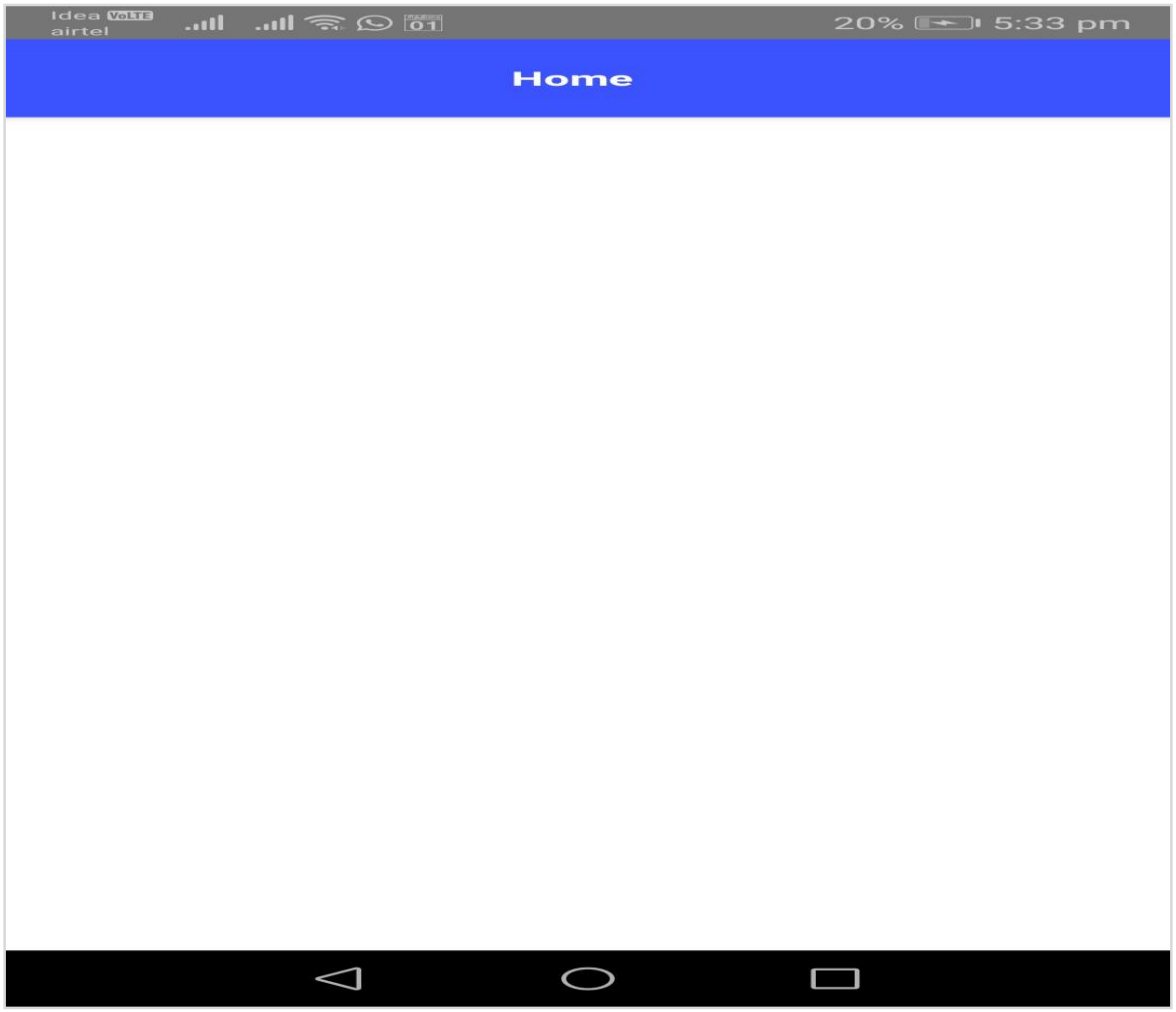
Open NativeScript PlayGround app on your iOS or Android mobile then choose *Scan QR code* option. It will open the camera. Focus the QR code displayed on the console. This will scan the QR Code. Scanning the QR Code will trigger the application build and then sync the application to the device as given below:

```
Copying template files...
Platform android successfully added. v6.3.1
Preparing project...
File change detected. Starting incremental webpack compilation...

webpack is watching the files…

Hash: 1f38aaf6fcda4e082b88
Version: webpack 4.27.1
Time: 9333ms
Built at: 01/04/2020 4:22:31 PM
```

```
              Asset       Size   Chunks             Chunk Names
               0.js   8.32 KiB        0  [emitted]
          bundle.js   22.9 KiB   bundle  [emitted]  bundle
       package.json  112 bytes           [emitted]
         runtime.js     73 KiB  runtime  [emitted]  runtime
tns-java-classes.js    0 bytes           [emitted]
          vendor.js    345 KiB   vendor  [emitted]  vendor
Entrypoint bundle = runtime.js vendor.js bundle.js
[../$$_lazy_route_resource lazy recursive] ../$$_lazy_route_resource lazy
namespace object 160 bytes {bundle} [built]
[./app.css] 1.18 KiB {bundle} [built]
[./app/app-routing.module.ts] 688 bytes {bundle} [built]
[./app/app.component.html] 62 bytes {bundle} [built]
[./app/app.component.ts] 354 bytes {bundle} [built]
[./app/app.module.ts] 3.22 KiB {bundle} [built]
[./app/home/home.module.ts] 710 bytes {0} [built]
[./main.ts] 1.84 KiB {bundle} [built]
[@angular/core] external "@angular/core" 42 bytes {bundle} [built]
[nativescript-angular/nativescript.module] external "nativescript-
angular/nativescript.module" 42 bytes {bundle} [built]
[nativescript-angular/platform] external "nativescript-angular/platform" 42
bytes {bundle} [built]
[tns-core-modules/application] external "tns-core-modules/application" 42 bytes
{bundle} [built]
[tns-core-modules/bundle-entry-points] external "tns-core-modules/bundle-entry-
points" 42 bytes {bundle} [built]
[tns-core-modules/ui/frame] external "tns-core-modules/ui/frame" 42 bytes
{bundle} [built]
[tns-core-modules/ui/frame/activity] external "tns-core-
modules/ui/frame/activity" 42 bytes {bundle} [built]
    + 15 hidden modules
Webpack compilation complete. Watching for file changes.
Webpack build done!
Project successfully prepared (android)
Start sending initial files for device Bala Honor Holly (ff5e8622-7a01-4f9c-
b02f-3dc6d4ee0e1f).
Successfully sent initial files for device Bala Honor Holly (ff5e8622-7a01-
4f9c-b02f-3dc6d4ee0e1f).
LOG from device Bala Honor Holly: HMR: Hot Module Replacement Enabled. Waiting
for signal.
LOG from device Bala Honor Holly: Angular is running in the development mode.
Call enableProdMode() to enable the production mode.
```

Output

After scanning, you should see your BlankNgApp on your device. It is shown below:



Run your app on device

If you want to test the connected device in your application, you can verify it using the below syntax:

```
'tns device <Platform> --available-devices'
```

After that, you can execute your app using the below command:

```
tns run
```

The above command is used to build your apps locally and install on Andriod or iOS devices. If you want to run your app on an Android simulator, then type the below command:

```
tns run android
```

For iOS device, you can follow the below command:

```
tns run ios
```

This will initialize the app in an Android/iOS device. We will discuss this more in detail in the upcoming chapters.

LiveSync

NativeScript provides real time syncing of changes in the application to the preview application. Let us open the project using any of your favourite editor (Visual Studio Code would be the ideal choice for better visualization). Let us add some changes in our code and see how that will be detected in LiveSync.

Now open the file `app.css` and it will have below content:

```
@import "~@nativescript/theme/css/core.css";
@import "~@nativescript/theme/css/blue.css";

/* Place any CSS rules you want to apply on both iOS and Android here.
This is where the vast majority of your CSS code goes. */
```

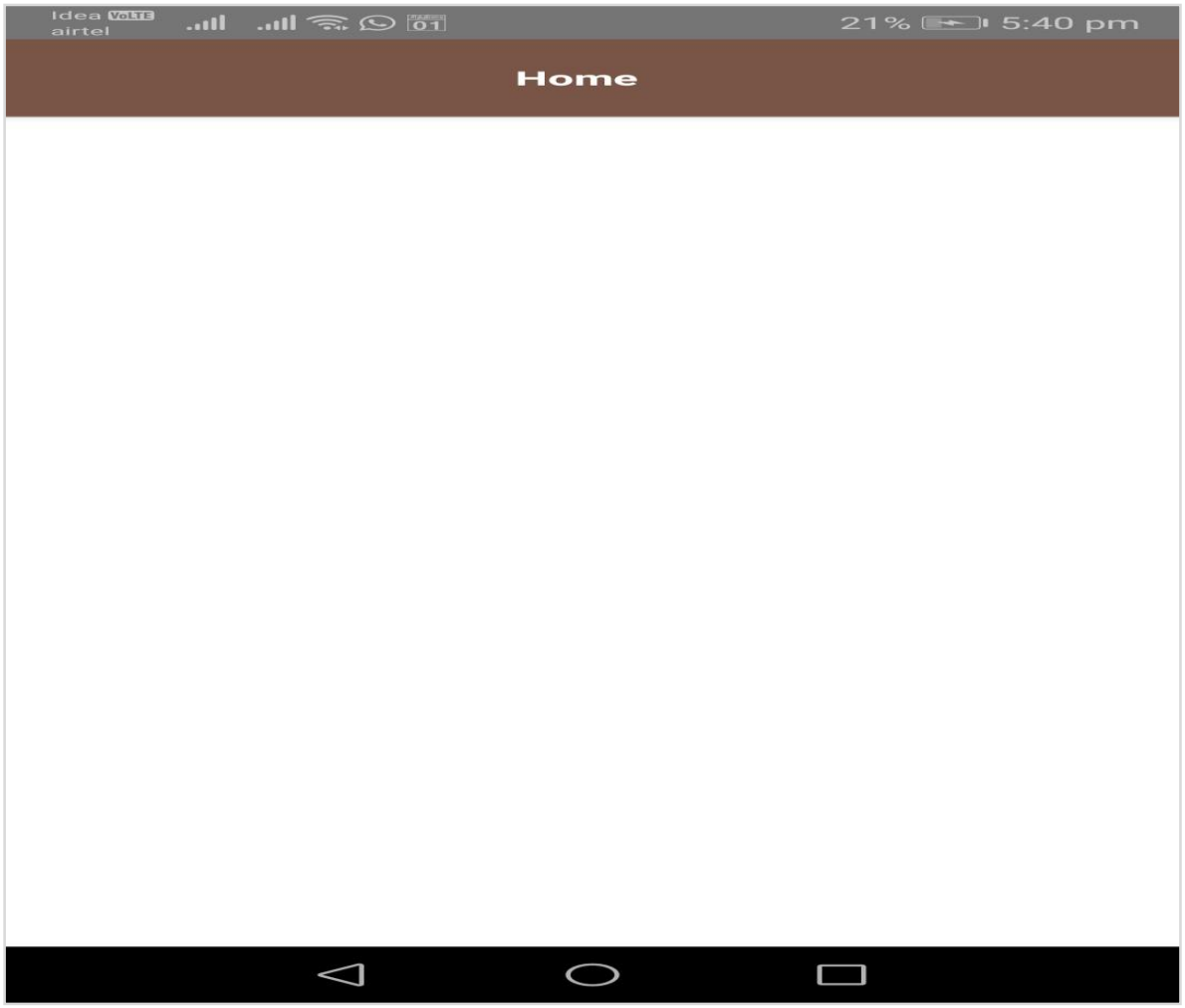
Here, import statement tells the color scheme of our app. Let's change the blue color scheme to the **brown** color scheme as specified below:

```
@import "~@nativescript/theme/css/core.css";
@import "~@nativescript/theme/css/brown.css";

/* Place any CSS rules you want to apply on both iOS and Android here.
This is where the vast majority of your CSS code goes. */
```

The application in our device refreshes and you should see a brown color ActionBar as shown below:

Output

Below is the BlankNgApp Home Page - Brown Theme.



5. NativeScript — Templates

NativeScript provides lot of readymade templates to create simple blank but fully functional application to complex Tab based application.

Using template

As learned earlier, new application can be created using **create** subcommand of `tns` command.

```
tns create <app-name> --template <tns-template-name>
```

Here,

tns-template-name is the name of the template.

If you want to create a template with one page and without any custom styles using JavaScript, use the below command:

```
tns create <app-name> --template tns-template-blank
```

The above same template can be created using TypeScript as follows:

```
tns create <app-name> --template tns-template-blank-ts
```

Navigation template

Navigation template is used to create moderate to complex application. It comes with pre-configured ***SideDrawer*** component with several pages. ***SideDrawer*** component contains a hidden view for navigation UI or common settings. Use the below command to create navigation based application:

```
tns create <app-name> --template tns-template-drawer-navigation
```

Tab navigation template

Tab navigation template is used to create tab based application. It comes with pre-configured **TabView** component with several pages. Use below command to create tab based application:

```
tns create <app-name> --template tns-template-tab-navigation
```

Master-Detail template

Master-Detail template is used to create list based application along with detail page for every item in the list.

```
tns create <app-name> --template tns-template-master-detail
```

Custom templates

To create simple customized template, we need to clone blank templates. As you know already, NativeScript supports JavaScript, TypeScript, Angular and Vue.js templates so you can choose any language and create your customized one.

For example, clone simple and customized template from git repository using the below command:

```
git clone https://github.com/NativeScript/template-blank-ts.git
```

Now, it will create mobile app structure so you can do any changes and run your android/iOS device. This structure based on list of guidelines. Let us see the guidelines in brief.

Structure

Your customized template must meet the following requirements:

- Don't place your code inside your app root folder.
- Create a separate folder and add `feature area` inside.
- Page, View models and service should be placed in feature area. This helps to create neat and clean code.
- Create page folder and place inside *.ts, .xml, .scss/css, etc.,* files.

package.json

Place ***package.json*** file in the root folder of your app template. Provide a value for the name property using the format:

```
{
  "name": "tns-template-blank-ts",
  "displayName": "template-blank",
}
```

Assign a value for the version property. It is defined below:

```
"version": "3.2.1",
```

Assign a value for the main property specifying the primary entry point to your app. It is defined below:

```
 "main": "app.js",
```

Assign a value for the android property. It is defined below:

```
"android": {
    "v8Flags": "--expose_gc"
  },
```

The **repository** property should be specified inside your code as follows:

```
  "repository": {
    "type": "git",
    "url": "https://github.com/NativeScript/template-master-detail-ts"
  },
```

Style

Import styles and themes in your app template using the below syntax:

```
@import '~nativescript-theme-core/scss/light';
```

We can also assign custom background color using the below code:

```
/*  Colors */
$background: #fff;
$primary: lighten(#000, 13%);
```

6. NativeScript — Widgets

NativeScript provides a large set of user interface components and are called as 'widgets'. Each widget does a special task and comes with a set of methods. Let's understand NativeScript widgets in detail in this section.

Button

Button is a component to execute tap event action. When a user taps the button it performs the corresponding actions. It is defined below:

```
<Button text="Click here!" tap="onTap"></Button>
```

Let us add the button in our *BlankNgApp* as below:

Step 1

Open the **src\app\home\home.component.html**. This is the UI design page of our home component.

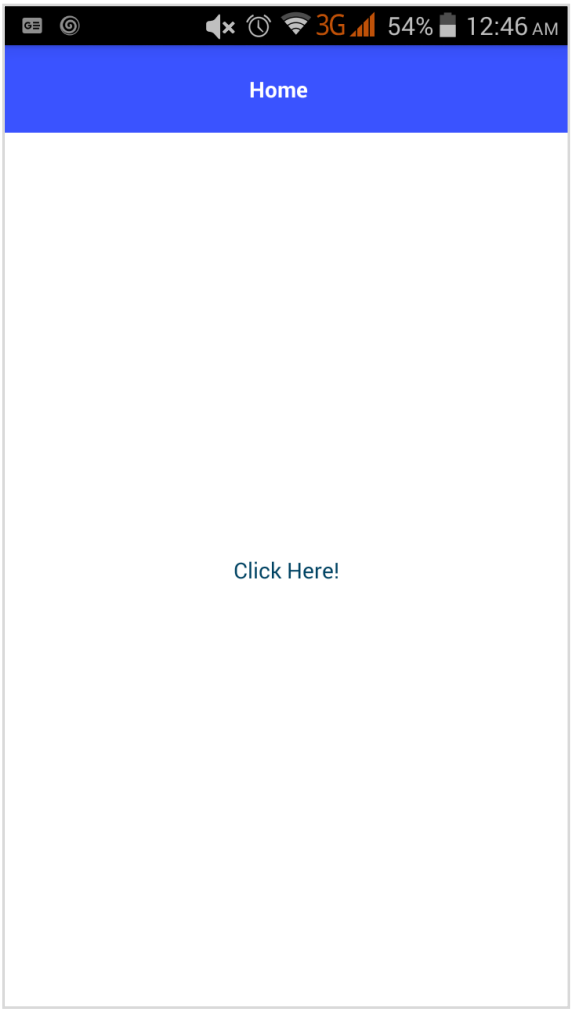
Step 2

Add a button inside the ***GirdLayout*** component. The complete code is as follows:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<GridLayout>
    <button text="Click Here!"></button>
</GridLayout>
```

Output

Below is the output of the button:



Step 3

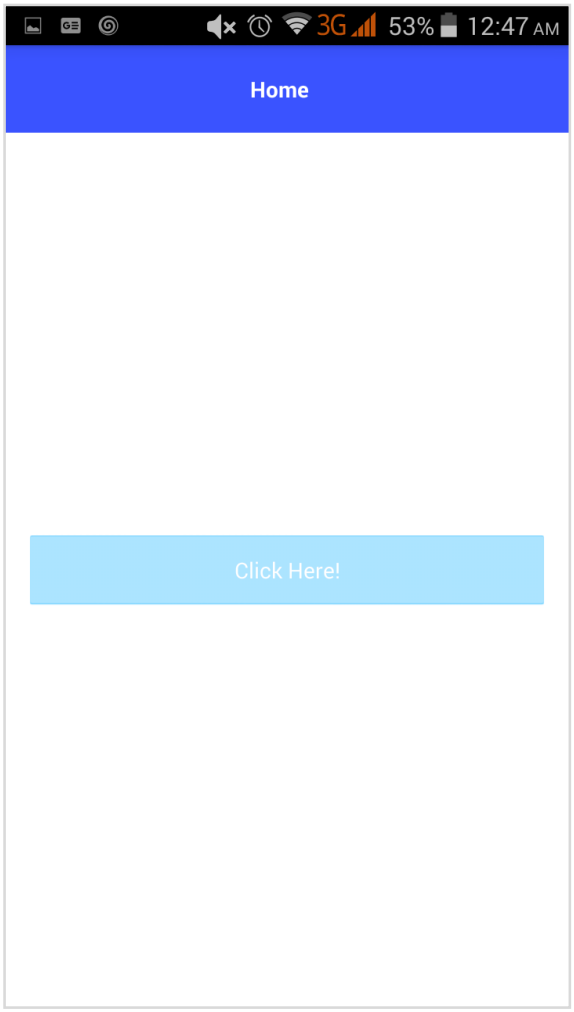
We can style the button using CSS as specified below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<GridLayout>
    <button text="Click Here!" class="-primary"></button>
</GridLayout>
```

Here, **-primary** class is used to represent the primary button.

Output

Below is the output of ***ButtonPrimary:***



Step 4

NativeScript provides formatted option to provide custom icons in the button. The sample code is as follows:

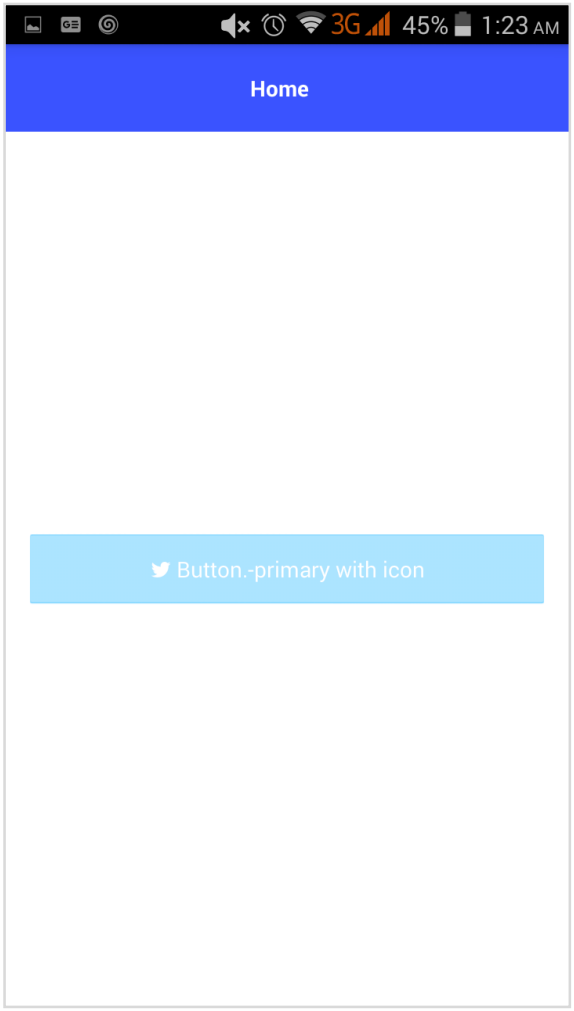
```
<GridLayout>
    <Button class="-primary">
        <FormattedString>
            <Span text="&#xf099;" class="fa"></Span>
            <Span text=" Button.-primary with icon"></Span>
        </FormattedString>
    </Button>
</GridLayout>

.fa {
    font-family: "FontAwesome", "fontawesome-webfont";
}
```

Here,

&#xf099 specifies the location of the icon in the font, FontAwesome. Download the latest Font Awesome font and place the fontawesome-webfont.ttf in src\fonts folder.

Output

Below is the output of ***ButtonPrimary***:



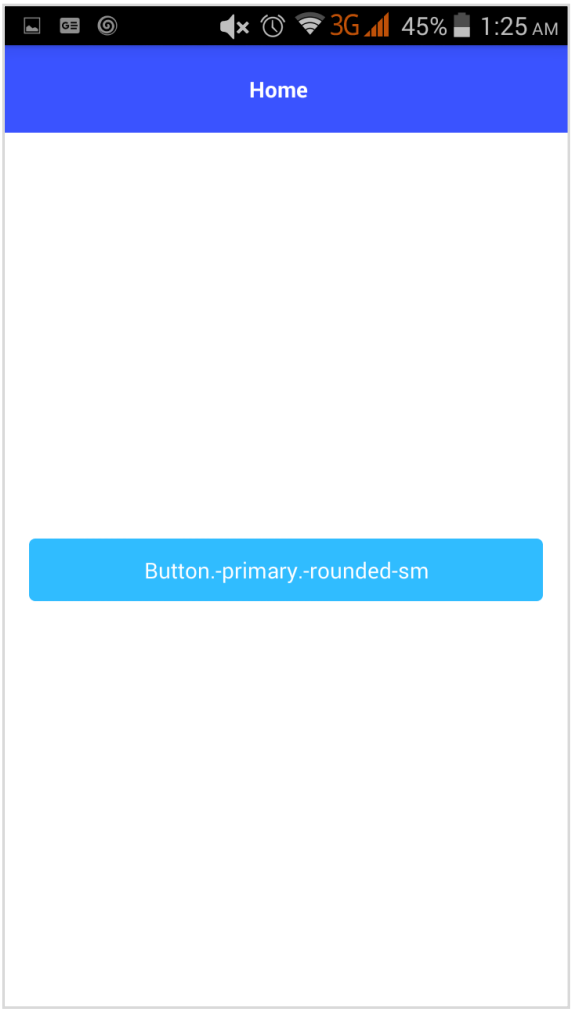
Step 5

Rounded button can be created using the below syntax:

```
<Button text="Button.-primary.-rounded-sm" class="-primary -rounded-
sm"></Button>
```

Output

Below is the output of ButtonPrimary:



Label

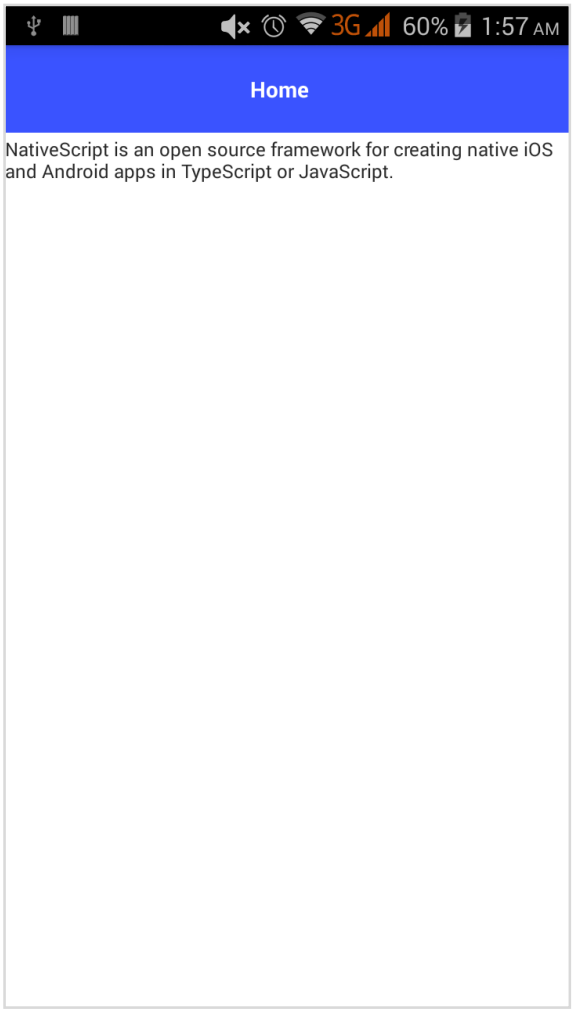
Label component is used to display static text. Change the home page as below:

```
<GridLayout>
    <Label text="NativeScript is an open source framework for creating native
iOS and Android apps in TypeScript or JavaScript."
           textWrap="true"></Label>
</GridLayout>
```

Here, textWrap wraps the content of the label, if the label extends beyond the screen width.

Output

Below is the output of Label:



TextField

TextField component is used to get information from user. Let us change our home page as specified below:

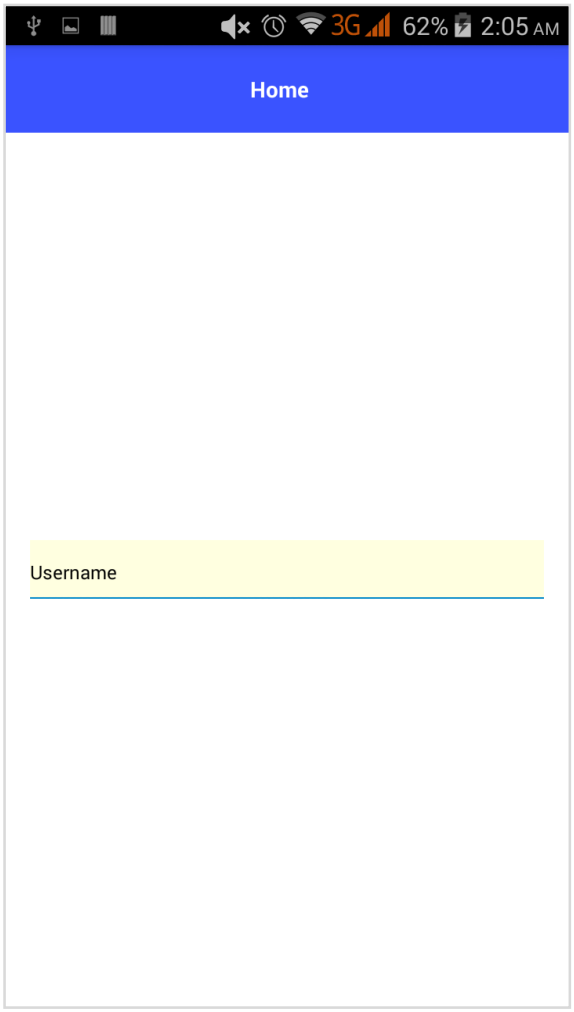
```
<GridLayout>
    <TextField hint="Username"
        color="lightblue"
        backgroundColor="lightyellow"
        height="75px"></TextField>
</GridLayout>
```

Here,

- `color` represent text color
- `backgroundColor` represent background of the text box
- `height` represent the height of the text box

Output

Below is the output of Text Field:



TextView

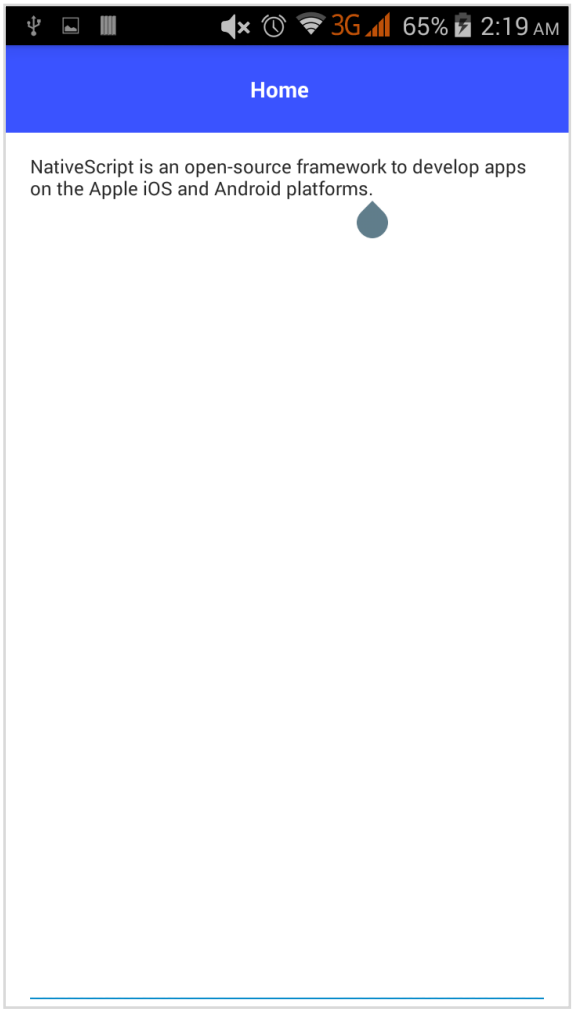
TextView Component is used to get multi-line text content from the user. Let us change our home page as specified below:

```
<GridLayout>
    <TextView loaded="onTextViewLoaded" hint="Enter text" returnKeyType="done"
autocorrect="false" maxLength="100">
    </TextView>
</GridLayout>
```

Here, maxLength represent maximum length accepted by *TextView.*

Output

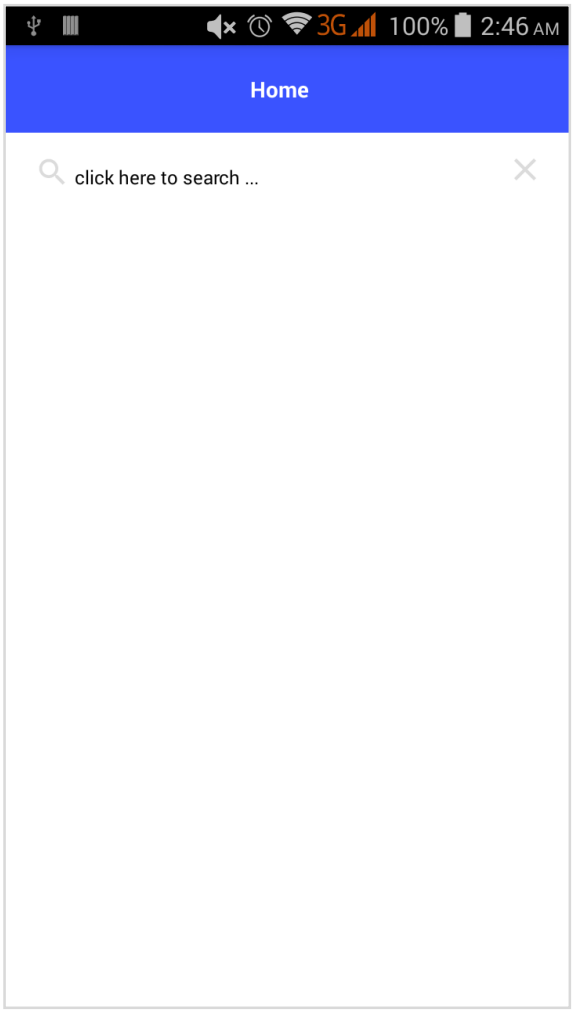
Below is the output of TextView:



SearchBar

This component is used for search any queries or submit any request. It is defined below:

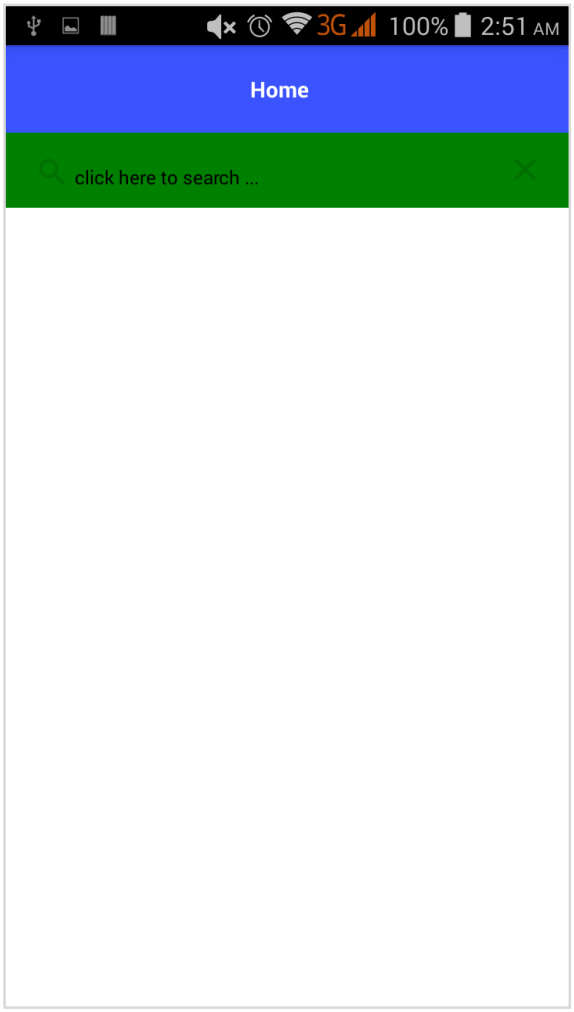
```
<StackLayout>
    <SearchBar id="bar" hint="click here to search ..."></SearchBar>
</StackLayout>
```



We can apply styles:

```
<StackLayout>
    <SearchBar id="bar" hint="click here to search ..." color="green"
backgroundColor="green"></SearchBar>
</StackLayout>
```

Below is the output of SearchBarStyle:

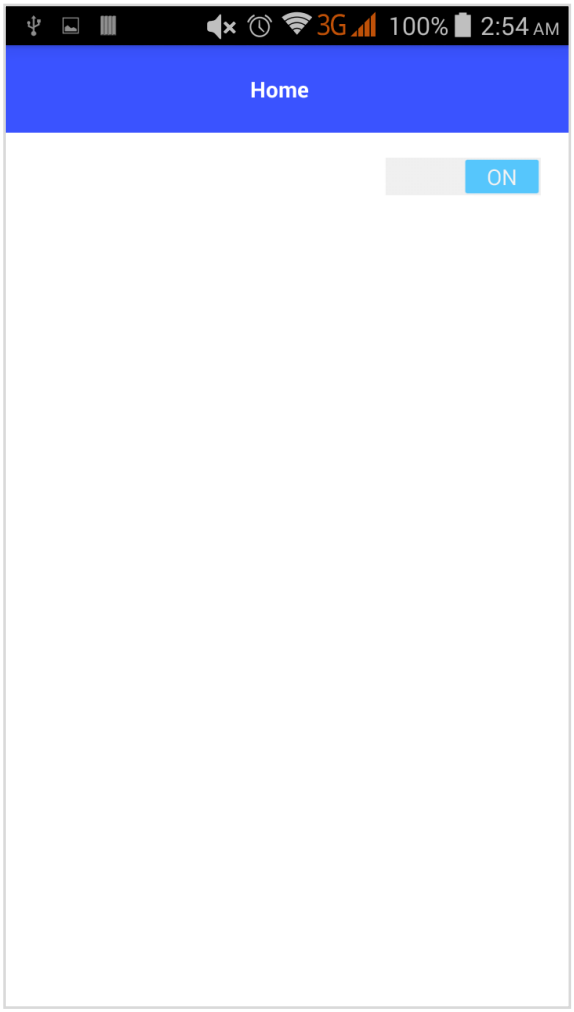


Switch

Switch is based on toggle to choose between options. Default state is false. It is defined below:

```
<StackLayout>
    <Switch checked="false" loaded="onSwitchLoaded"></Switch>
</StackLayout>
```

The output for the above program is shown below:

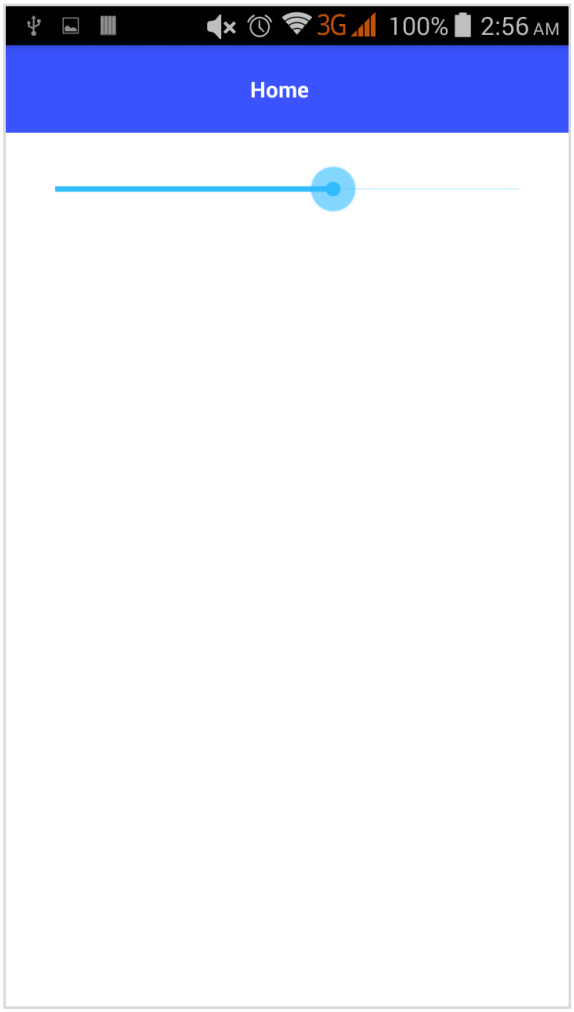


Slider

Slider is a sliding component to pick a numeric range. It is defined below:

```
<Slider value="30" minValue="0" maxValue="50"
loaded="onSliderLoaded"></Slider>
```

The output for the above program is given below:

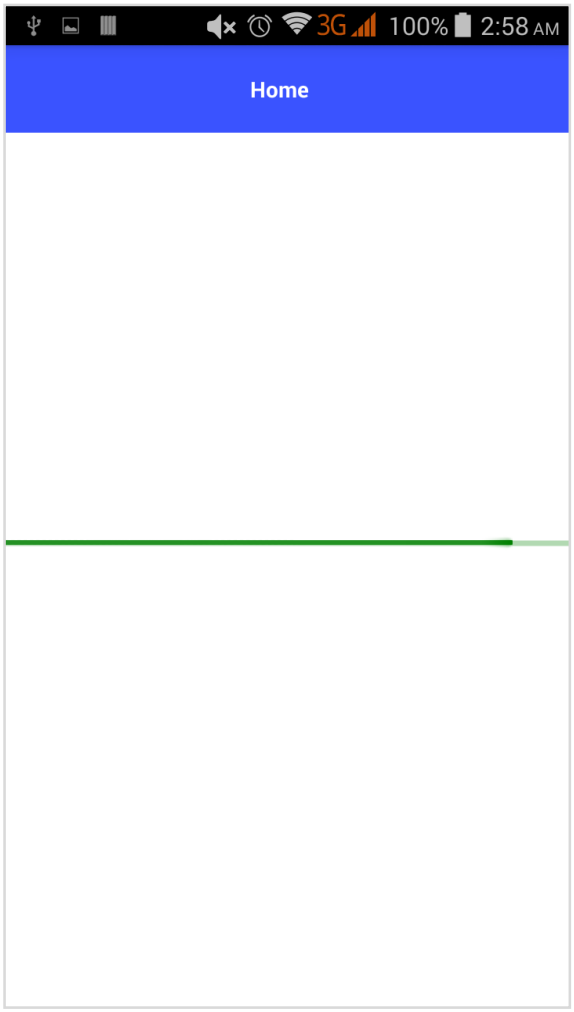


Progress

Progress widget indicates progress in an operation. Current progress is represented as bar. It is defined below:

```
<StackLayout verticalAlign="center" height="50">
    <Progress value="90" maxValue="100" backgroundColor="red" color="green"
row="0"></Progress>
</StackLayout>
```

Below is the output of Progress widget:

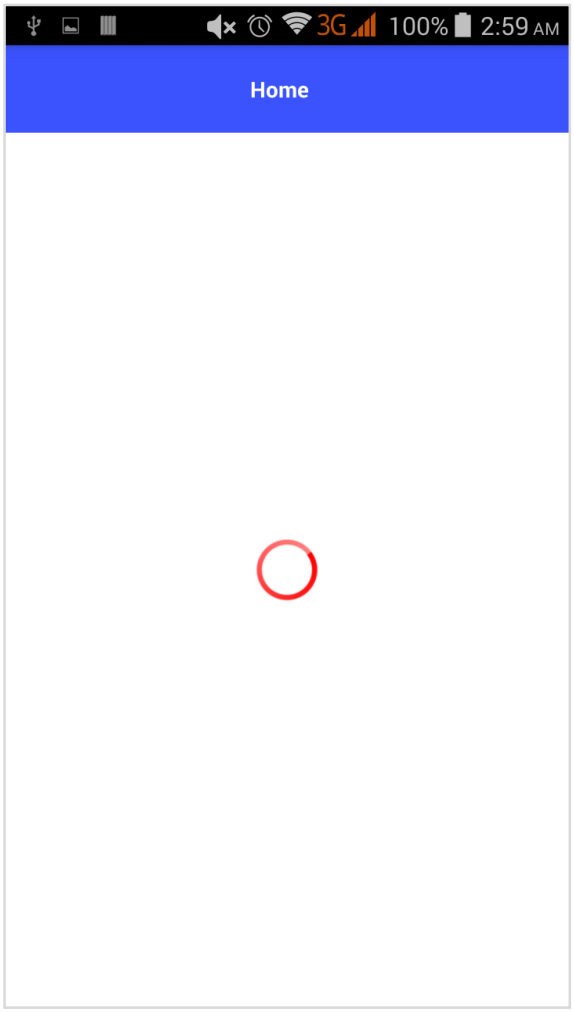


ActivityIndicator

ActivityIndicator shows a task in a progress. It is defined below:

```
<StackLayout verticalAlign="center" height="50">
    <ActivityIndicator busy="true" color="red" width="50"
height="50"></ActivityIndicator>
</StackLayout>
```

Below is the output for ActivityIndicator:



Image

Image widget is used to display an image. It can be loaded using 'ImageSource' url. It is defined below:

```
<StackLayout class="m-15" backgroundColor="lightgray">
    <Image src="~/images/logo.png" stretch="aspectFill"></Image>
</StackLayout>
```

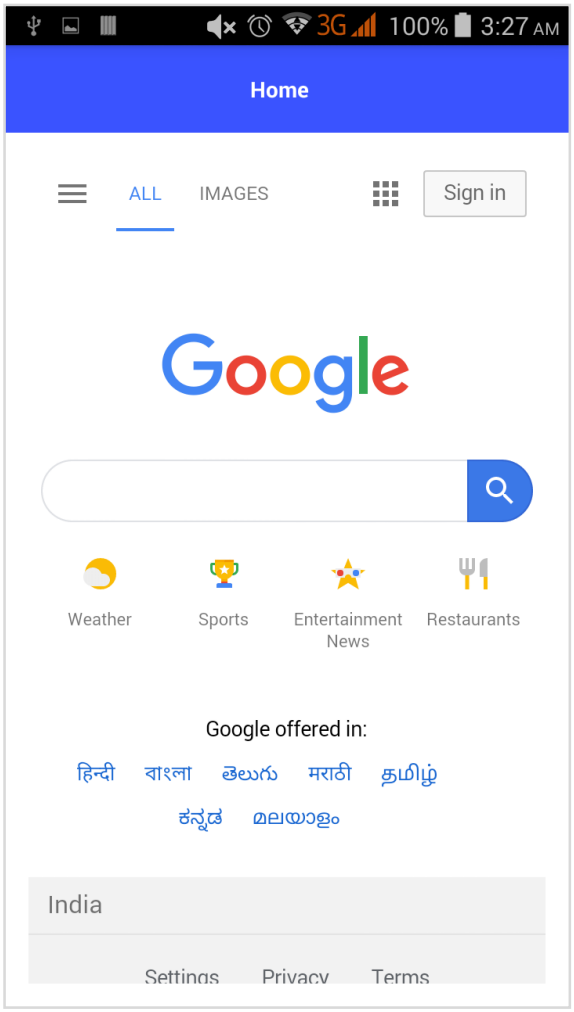
The output for Image Widget is as shown below:

WebView

WebView shows web pages. Web pages can be loaded using URL. It is defined below:

```
<WebView row="1" loaded="onWebViewLoaded" id="myWebView"
src="http://www.google.com"></WebView>
```

The output for the above code is as shown below:

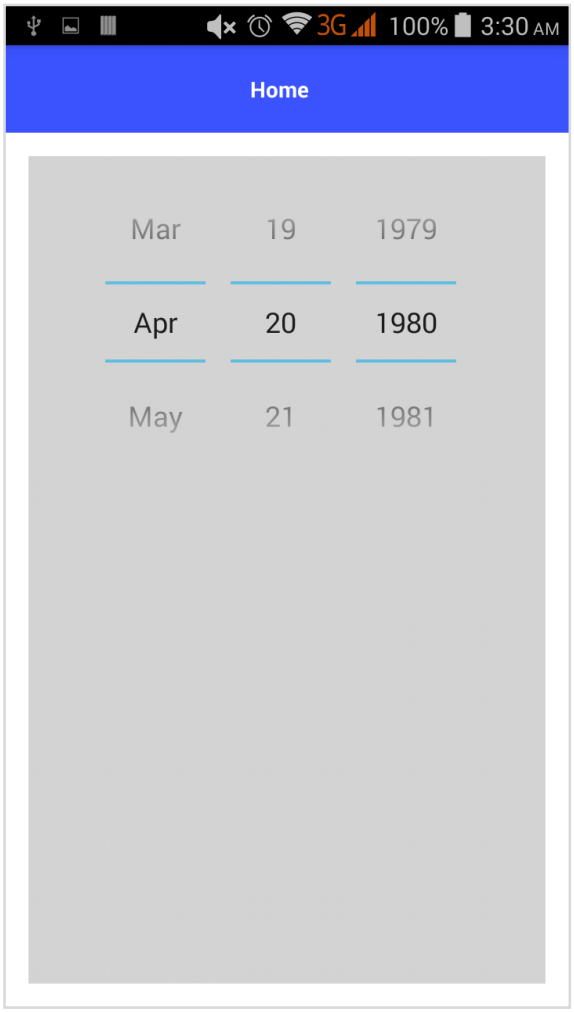


DatePicker

DatePicker component is used to pick date. It is defined below:

```
<StackLayout class="m-15" backgroundColor="lightgray">
<DatePicker year="1980" month="4" day="20"
                verticalAlignment="center">
</DatePicker>
</StackLayout>
```

The output of DatePicker component is as shown below:

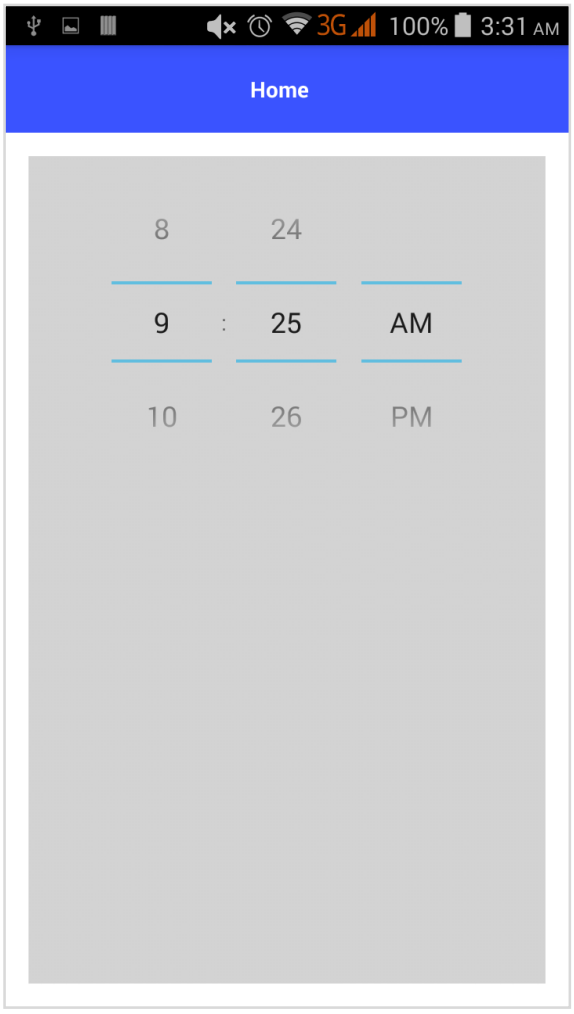


TimePicker

TimePicker component is used to pick the time. It is defined below:

```
<StackLayout class="m-15" backgroundColor="lightgray">
<TimePicker hour="9"
            minute="25"
            maxHour="23"
            maxMinute="59"
            minuteInterval="5">
</TimePicker>
</StackLayout>
```

Below is the output of TimePicker component:


100%
3:31 AM
Home
8
24
9
:
25
AM
10
26
PM

7. NativeScript — Layout Containers

NativeScript provides collection of container component for the sole purpose of laying out UI widget component. Layout containers act as the parent component and can have one or more child components. All child components of a layout container can be arranged based on the technique provided by its parent layout container.

NativeScript supports six layouts containers and they are as follows:

- Absolute layout container
- Dock layout container
- Grid layout container
- Stack layout container
- Wrap layout container
- FlexBox layout container

Let us learn all the layout container concepts in detail in this chapter.

Absolute Layout

AbsoluteLayout container is the simplest layout container in NativeScript. AbsoluteLayout does not enforce any constraint on its children and will place its children inside it using 2-dimensional coordinate system with top-left corner as origin.

AbsoluteLayout uses four properties of its children to position them and they are as follows:

top: Defines the placement of the child from origin moving downwards in y direction.

left: Defines the placement of the child from origin moving sidewards in x direction.

width: Defines the width of the child.

height: Defines the height of the child.

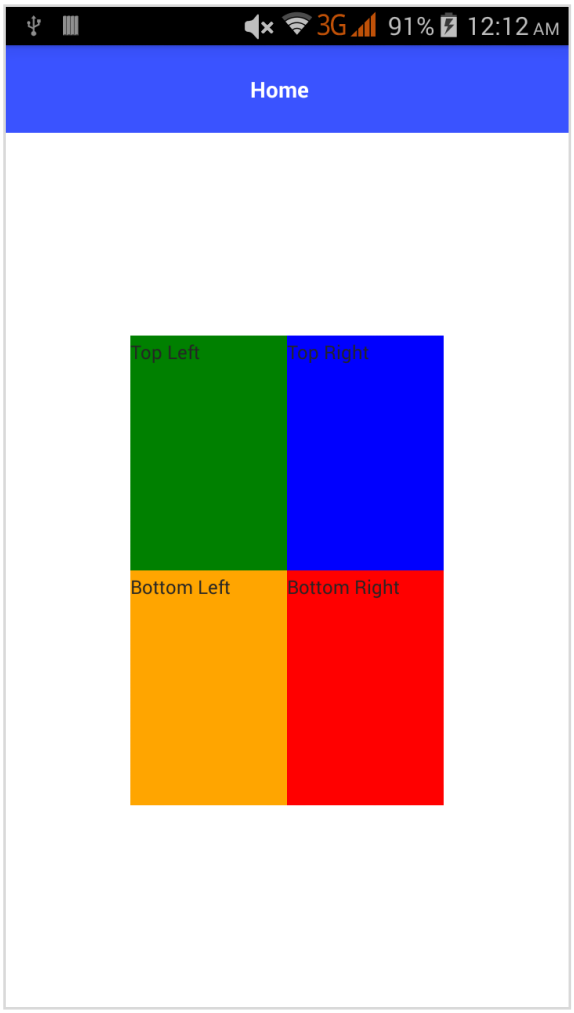
Let us add AbsoluteLayout container in the home page of our application as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<AbsoluteLayout width="200" height="300" backgroundColor="blue">
    <Label text="Top Left" left="0" top="0" width="100" height="150"
backgroundColor="green"></Label>
    <Label text="Top Right" left="100" top="0" width="100" height="150"
backgroundColor="blue"></Label>
    <Label text="Bottom Left" left="0" top="150" width="100" height="150"
backgroundColor="orange"></Label>
```

```
    <Label text="Bottom Right" left="100" top="150" width="100" height="150"
backgroundColor="red"></Label>
</AbsoluteLayout>
```

Output

The output of the AbsoluteLayout is as given below:



DockLayout

Docklayout container component enables its children to dock inside it. Each side of the container (top, bottom, left, right) can dock a child component. DockLayout container uses the dock property of its children to correctly dock them.

The possible values of the dock property is as follows:

top: Layout container dock the child component at the top corner.

bottom: Layout container dock the child component at the bottom corner.

left: Layout container dock the child component at the left corner.

right: Layout container dock the child component at the right corner.

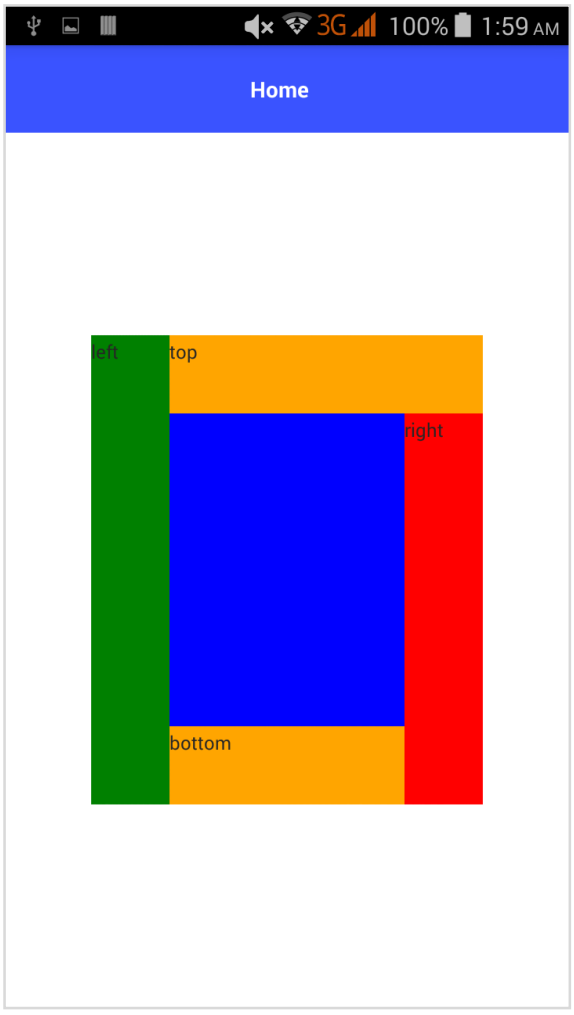
By default, **DockLayout** container docks its last child component. It can override by setting its stretchLastChild property to zero.

Let us add **DockLayout** container in the home page of our application as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<DockLayout width="250" height="300" backgroundColor="blue"
stretchLastChild="false">
    <Label text="left" dock="left" width="50" backgroundColor="green"></Label>
    <Label text="top" dock="top" height="50" backgroundColor="orange"></Label>
    <Label text="right" dock="right" width="50" backgroundColor="red"></Label>
    <Label text="bottom" dock="bottom" height="50"
backgroundColor="orange"></Label>
</DockLayout>
```

Output

Below is the output for DockLayout:



GridLayout

GridLayout container component is one of the complex layout container and arranges child elements in tabular format with rows and columns. By default, it has one row and one column. It has the following properties:

columns: Used to represent the default width of each column separated by ,. The possible values are number, * and auto keyword.

Where,

- number indicates an absolute column width.
- indicates the width of a column relative to other columns. It can be preceded by number to indicate how many times the column width should be relative to other column. For example, 2* indicate the width the column should be 2 times the width of the smallest column.
- auto indicates the width of the column as wide as its widest child.

For example, *,2* means two columns and second will be twice the size of first column.

rows: Used to represent the default height of each row separated by ,. Value representation is similar to columns.

GridLayout uses the below specified properties of its children to layout them:

row: Row number

col: Column number

rowSpan: total number of rows that child content spans within a layout.

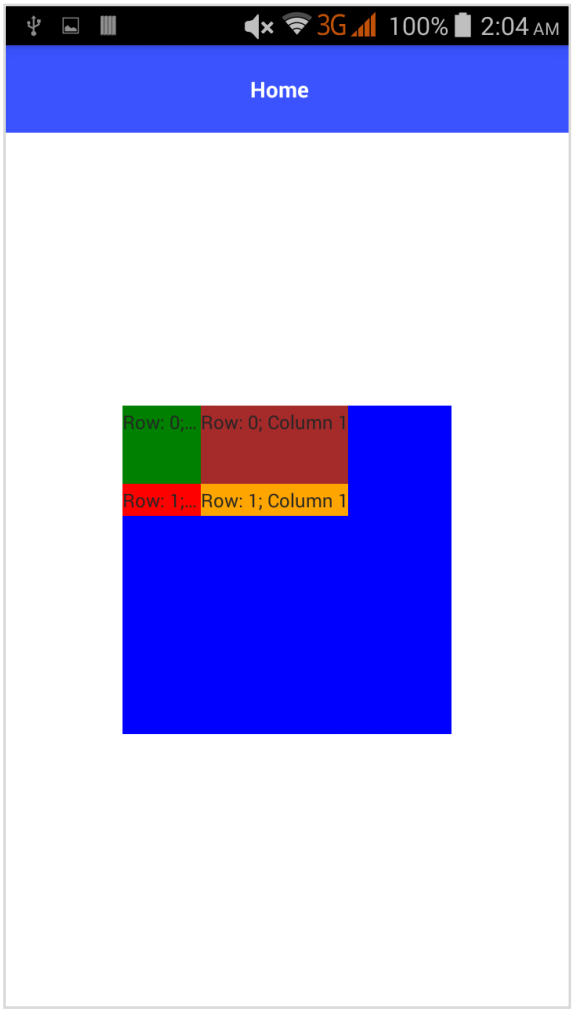
colSpan: total number of columns that child content spans within a layout.

Let us add GridLayout container in the home page of our application as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<GridLayout columns="50, auto, *" rows="50, auto, *" width="210" height="210"
backgroundColor="blue">
    <Label text="Row: 0; Column 0" row="0" col="0"
backgroundColor="green"></Label>
    <Label text="Row: 0; Column 1" row="0" col="1" colSpan="1"
backgroundColor="brown"></Label>
    <Label text="Row: 1; Column 0" row="1" col="0" rowSpan="1"
backgroundColor="red"></Label>
    <Label text="Row: 1; Column 1" row="1" col="1"
backgroundColor="orange"></Label>
</GridLayout>
```

Output

Below is the output for GridLayout:



StackLayout

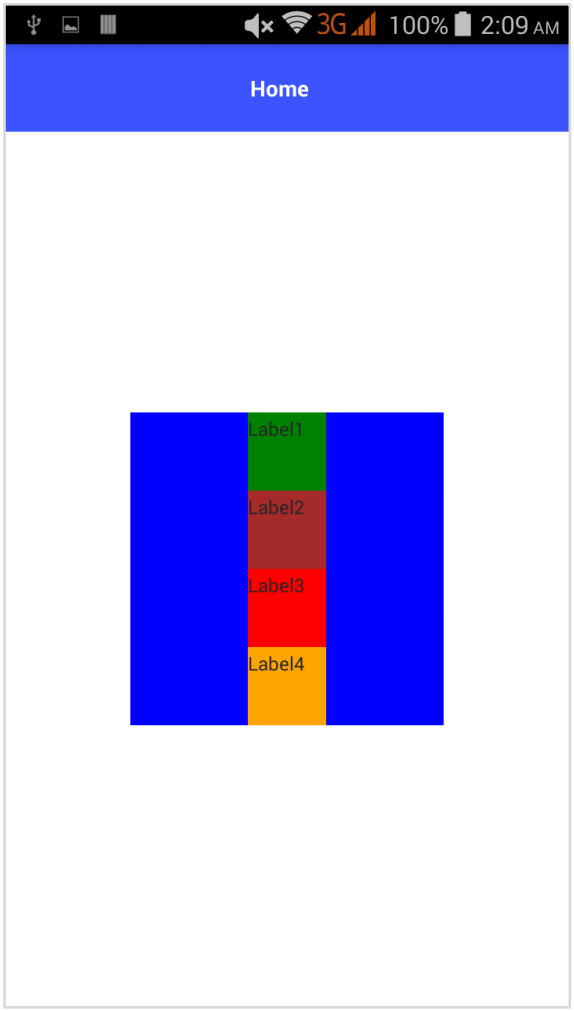
StackLayout organizes its children in a one-dimensional line either horizontally or vertically. It can be sized based on the space in the layout using layout options. It has orientation property that can be used to specify direction, horizontal or vertical.

Let us add StackLayout container in the home page of our application as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<StackLayout orientation="vertical" width="200" height="200"
backgroundColor="blue">
    <Label text="Label1" width="50" height="50"
backgroundColor="green"></Label>
    <Label text="Label2" width="50" height="50"
backgroundColor="brown"></Label>
    <Label text="Label3" width="50" height="50" backgroundColor="red"></Label>
    <Label text="Label4" width="50" height="50"
backgroundColor="orange"></Label>
</StackLayout>
```

Output

The output for StackLayout is as shown below:



WrapLayout

WrapLayout is used to wrap contents on new rows or columns.

It has the following three properties:

orientation: display either horizontally or vertically.

itemWidth: layout width for each child.

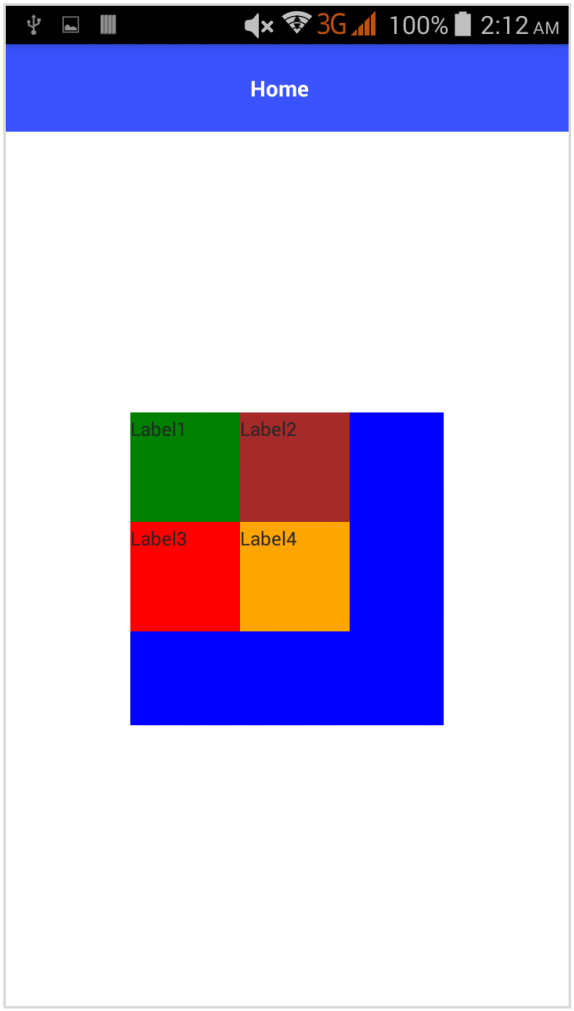
itemHeight: layout height for each child.

Let us add WrapLayout container in the home page of our application as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<WrapLayout orientation="horizontal" width="200" height="200"
backgroundColor="blue">
```

```
    <Label text="Label1" width="70" height="70" backgroundColor="green"</Label>
    <Label text="Label2" width="70" height="70"
backgroundColor="brown"></Label>
    <Label text="Label3" width="70" height="70" backgroundColor="red"></Label>
    <Label text="Label4" width="70" height="70"
backgroundColor="orange"></Label>
</WrapLayout>
```

Output



Flexbox Layout

FlexboxLayout container component is one of the advanced layout container. It provides option to render simple layout to very complex and sophisticated layouts. It is based on the *CSS Flexbox*.

FlexboxLayout component has lot of properties and they are as follows:

flexDirection

It represents the direction in which the child components are arranged. The possible values of flexDirection are as follows:

row: Child components are arranged side by side.

row-reverse: Child components are arranged side by side but in reverse direction.

column: Child components are arranged one below the another.

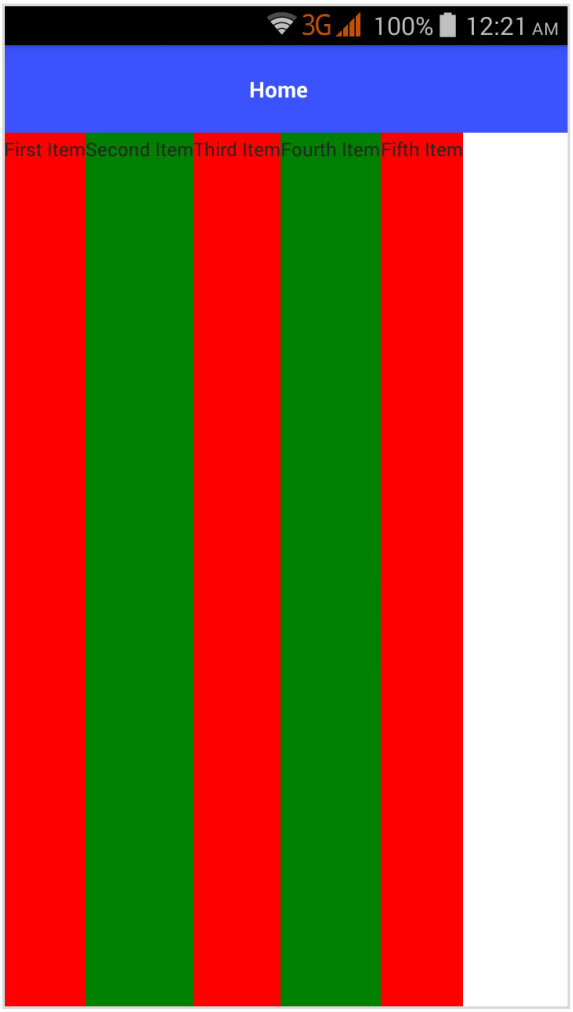
column-reverse: Child components are arranged one below the another but in reverse direction.

Let us add FlexLayout container in the home page of our application as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<FlexboxLayout flexDirection="row">
    <Label text="First Item" backgroundColor="red"></Label>
    <Label text="Second Item" backgroundColor="green"></Label>
    <Label text="Third Item" backgroundColor="red"></Label>
    <Label text="Fourth Item" backgroundColor="green"></Label>
    <Label text="Fifth Item" backgroundColor="red"></Label>
</FlexboxLayout>
```

Output

Below is the output of FlexLayout – Row:

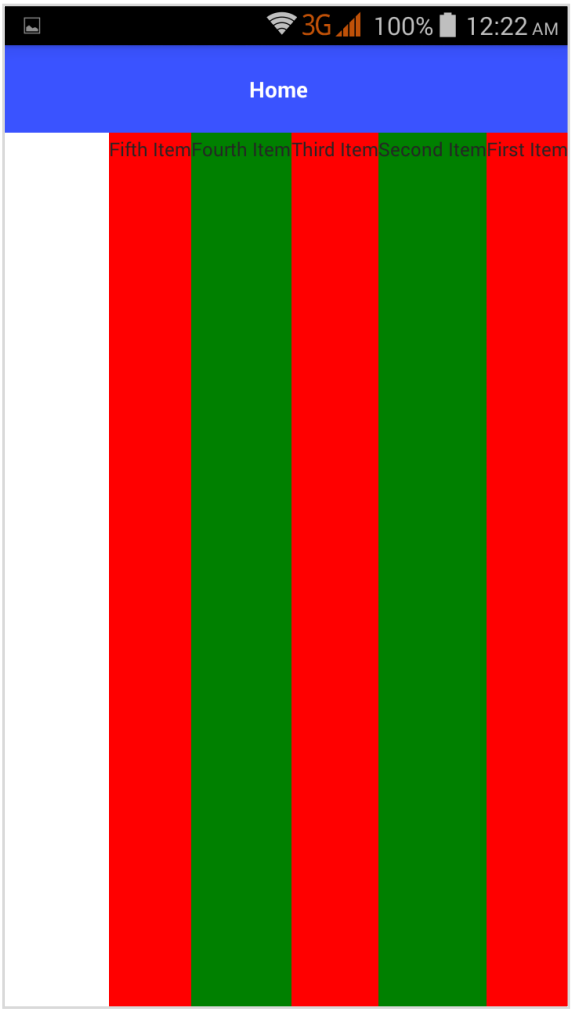


Now, let us change the flexDirection value from row to row-reverse and check how it affects the layout.

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<FlexboxLayout flexDirection="row-reverse">
    <Label text="First Item" backgroundColor="red"></Label>
    <Label text="Second Item" backgroundColor="green"></Label>
    <Label text="Third Item" backgroundColor="red"></Label>
    <Label text="Fourth Item" backgroundColor="green"></Label>
    <Label text="Fifth Item" backgroundColor="red"></Label>
</FlexboxLayout>
```

Output

Below is the output of Flex Layout - Row Reverse:



Let us change the flexDirection value from row-reverse to column and check how it affects the layout.

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<FlexboxLayout flexDirection="column">
    <Label text="First Item" backgroundColor="red"></Label>
    <Label text="Second Item" backgroundColor="green"></Label>
    <Label text="Third Item" backgroundColor="red"></Label>
    <Label text="Fourth Item" backgroundColor="green"></Label>
    <Label text="Fifth Item" backgroundColor="red"></Label>
</FlexboxLayout>
```

Output

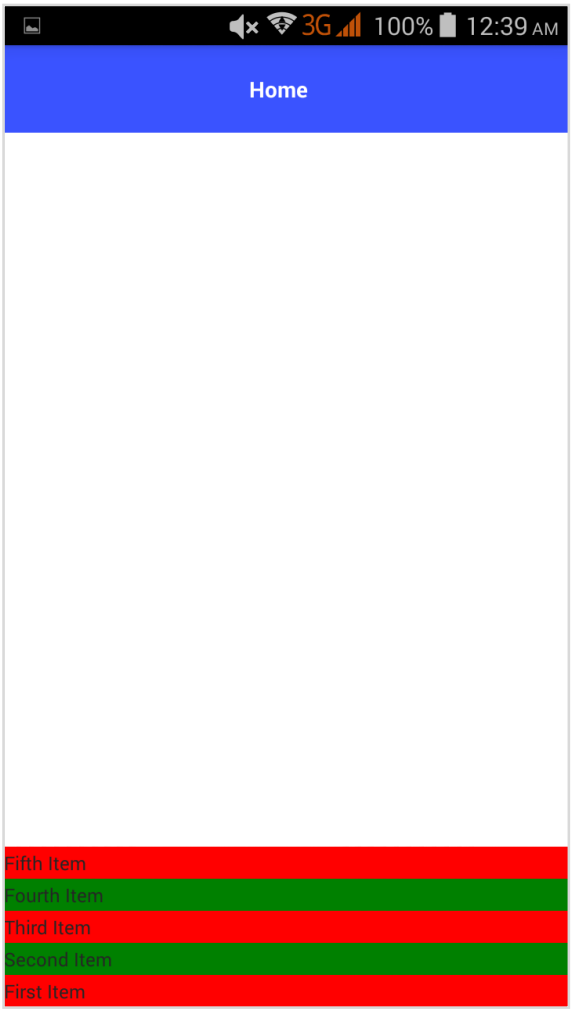
The output for FlexLayout – Column is given below:

Let us change the flexDirection value from column to column-reverse and check how it affects the layout.

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<FlexboxLayout flexDirection="column-reverse">
    <Label text="First Item" backgroundColor="red"></Label>
    <Label text="Second Item" backgroundColor="green"></Label>
    <Label text="Third Item" backgroundColor="red"></Label>
    <Label text="Fourth Item" backgroundColor="green"></Label>
    <Label text="Fifth Item" backgroundColor="red"></Label>
</FlexboxLayout>
```

Output

Below is the output of FlexLayout – Column Reverse:



flexWrap

It represents whether the child components will be rendered in a single row/column or flow into multiple rows by wraping in the direction set by flexDirection.

The possible values are as follows:

wrap: Wraps the children components, if no space is available in the given direction (flexDirection).

wrap-reverse: Same as wrap except the component flow in opposite direction.

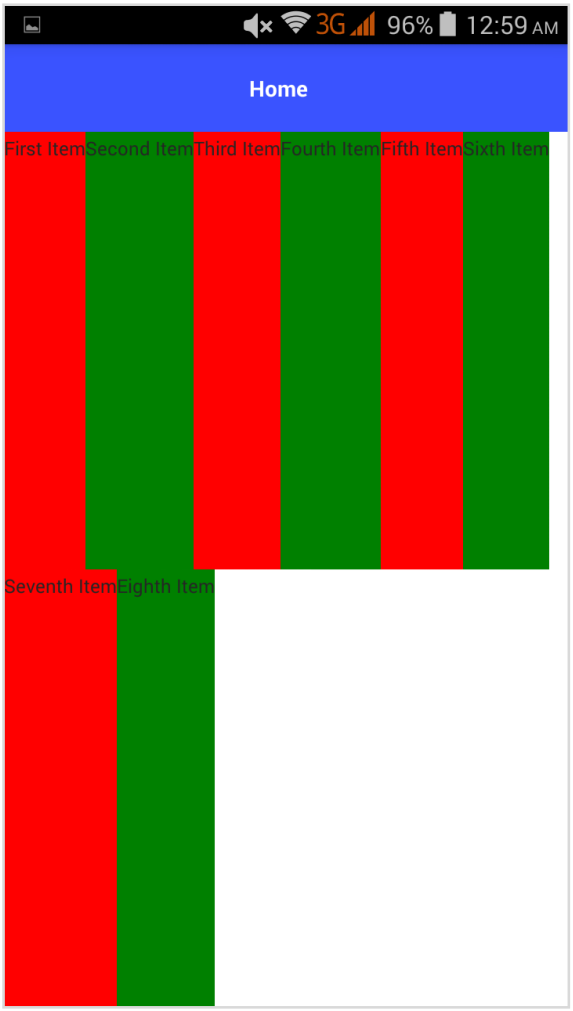
Let us add the flexWrap property and then set its value as wrap. Also add three more children as specified below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<FlexboxLayout flexDirection="row" flexWrap="wrap">
    <Label text="First Item" backgroundColor="red"></Label>
    <Label text="Second Item" backgroundColor="green"></Label>
    <Label text="Third Item" backgroundColor="red"></Label>
    <Label text="Fourth Item" backgroundColor="green"></Label>
    <Label text="Fifth Item" backgroundColor="red"></Label>
```

```
    <Label text="Sixth Item" backgroundColor="green"></Label>
    <Label text="Seventh Item" backgroundColor="red"></Label>
    <Label text="Eighth Item" backgroundColor="green"></Label>
</FlexboxLayout>
```

Output

Below is the output for flexWrap:



JustifyContent

It represents how child components are arranged with respect to each other and the overall structure. It has three properties as specified below:

flex-end: It packs the child component toward the end line.

space-between: It packs the child component by evenly distributing in line.

space-around: Similar to space-between except it packs the child component by evenly distributing in line as well as equal space around them.

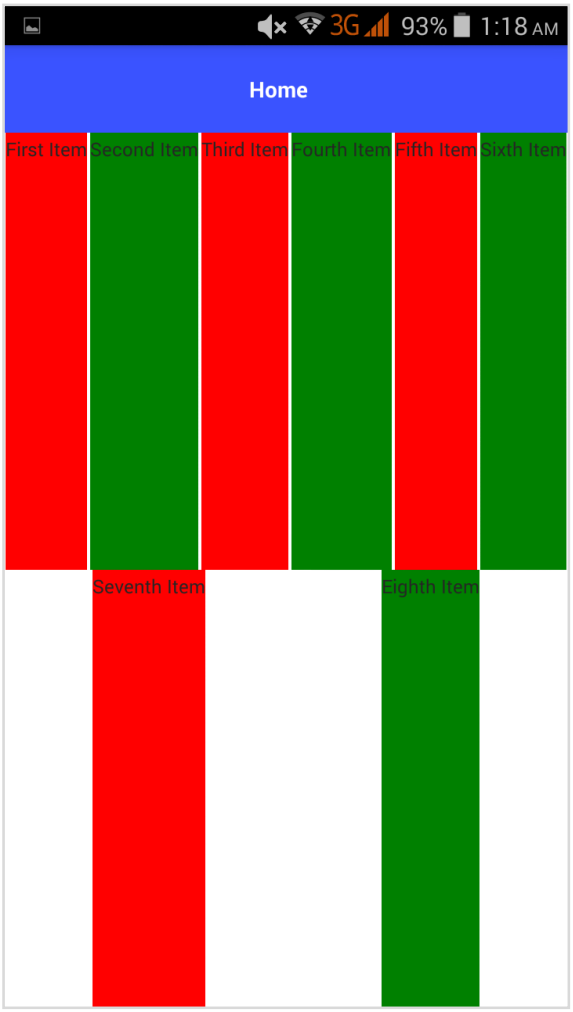
Let us add justifyContent as well and check how it behaves:

```
<ActionBar>
    <Label text="Home"></Label>
```

```
</ActionBar>

<FlexboxLayout flexDirection="row" flexWrap="wrap" justifyContent="space-
around">
    <Label text="First Item" backgroundColor="red"></Label>
    <Label text="Second Item" backgroundColor="green"></Label>
    <Label text="Third Item" backgroundColor="red"></Label>
    <Label text="Fourth Item" backgroundColor="green"></Label>
    <Label text="Fifth Item" backgroundColor="red"></Label>
    <Label text="Sixth Item" backgroundColor="green"></Label>
    <Label text="Seventh Item" backgroundColor="red"></Label>
    <Label text="Eighth Item" backgroundColor="green"></Label>
</FlexboxLayout>
```

Output

Below is the output of Flex Layout – JustifyContent:



FlexLayout container provides two more properties for its children to specify the order and ability to shrink. They are as follows:

order: It determines that order in which the children of the FlexLayout container will be rendered.

flexShrink: It determines the ability of the children to shrink to level 0.

8. NativeScript — Navigation

Navigation enables users to quickly swipe in to their desired screen or to navigate through an app or to perform a particular action. Navigation component helps you implement navigation using simple button clicks to more complex patterns.

Navigation differ substantially between core and angular version of NativeScript. While core framework navigation is foundation for navigation process, NativeScript's Angular model adopts the core navigation concept and extends it to make it compatible with Angular framework.

Let us see both core navigation concept and angular adoption of navigation in this chapter.

Core Concepts

Let us understand how the navigation works in core NativeScript in this chapter.

In NativeScript, navigation is split into four different categories based on the direction it applies as specified below:

- Forward navigation
- Backward navigation
- Lateral navigation
- Bottom navigation

Forward Navigation

Forward Navigation refers to navigating users to the screen in the next level of hierarchy. It is based on two NativeScript components, **Frame** and **Page**.

Frame

Frame is the root level component for navigation. It is not a visible container but acts as a container for transitions between pages.

A simple example is as follows:

```
<Frame id="featured" defaultPage="featured-page" />
```

Here,

Frame navigates to (or loads) the featured-page page component and renders it.

Page

Page is next to Frame component and it acts as a container for UI component. Simple example is defined below:

```
<Page loaded="onPageLoaded">
    <ActionBar title="Item" class="action-bar"></ActionBar>
```

```
    <AbsoluteLayout>
        <Label text="Label"/>
    <Button text="navigate('another-page')" tap="onTap"/>
    </AbsoluteLayout>
</Page>
```

Here,

- Initially, Page loads all the UI component of the screen and renders it.
- When user clicks the button, it will navigate the user to **another-page** page.

Backward Navigation

Backward navigation method enables backward movement through screens within one app or across different apps. It is the opposite direction of forward navigation. Simple goBack() method is used to navigate back to the previous page.

It is defined below:

```
<Page class="page" loaded="onPageLoaded">
    <ActionBar title="Item" class="action-bar"></ActionBar>

    <StackLayout class="home-panel">
        <Button class="btn btn-primary" text="Back" tap="goBack"/>
    </StackLayout>
</Page>
```

Here,

goBack() method will be triggered when user taps the button. **goBack()** navigates the users to the previous page, if one is available.

Lateral Navigation

Lateral navigation refers to the navigation between screens at same levels of hierarchy. It is based on hub pattern. It is enabled through specific navigation components such as BottomNavigation, Tabs, TabView, SideDrawer and Modal View.

A simple example is defined as below:

```
<Page class="page" xmlns="http://www.nativescript.org/tns.xsd">
    <ActionBar title="Hub" class="action-bar">
    </ActionBar>

    <StackLayout class="home-panel">
        <Button class="btn btn-primary" text="navigate('featured-page')"
tap="navigateToFeatured" />
        <Button class="btn btn-primary" text="navigate('search-page')"
tap="navigateToSearch" />
    </StackLayout>
</Page>
```

Here,

- **navigateToFeatured** function uses navigate() method to navigate the user to featured page.
- Similarly, **navigateToSearch** function will navigate the user to search page.

The hub page can also be reached using navigate method available in page screen and one can move out of hub page using goBack() method.

A simple example is as follows:

```
<Page class="page">
    <ActionBar title="Item" class="action-bar"></ActionBar>

    <StackLayout class="home-panel">
        <Button class="btn btn-primary" text="navigate('hub-page')"
tap="navigateToHub" />
        <Button class="btn btn-primary" text="goBack()" tap="goBack" />
    </StackLayout>
</Page>
```

Bottom and Tab Navigation

The most common style of navigation in mobile apps is tab-based navigation. The Tab Navigation is arranged at the bottom of the screen or on the top below the header. It is achieved by using *TabView* and *BottomNavigation* component.

Angular based navigation

NativeScript extends its navigation concept to accommodate the Angular routing concept. NativeScript provides a new module, NativeScriptRouterModule by extending Angular RouterModule.

NativeScript angular navigation concept can be categorized into section as below:

- page-router-outlet tag
- nsRouterLink attractive
- *RouterExtension* class
- Custom *RouterReuseStrategy*

Let us learn all the above angular navigation in this section.

Page Router Outlet

As learned earlier, page-router-outlet is the replacement of Angular's router-outlet. page-router-outlet wraps the Frame and Page strategy of Nativescript core navigation framework. Each page-router-outlet creates a new Frame component and each configured components in the outlet will be wrapped using Page component. Then, the native navigate method is used to navigate to another page / route.

Router Link (nsRouterLink)

nsRouterLink is the replacement of Angular's RouterLink. It enables UI component to link to another page using route. nsRouterLink also provides below two options:

pageTransition: It is used to set page transition animation. true enables default transition. false disables the transition. Specific values like slide, fadein, etc., set the particular transition.

clearHistory: true clears the navigation history of nsRouterLink.

A simple example code is as follows:

```
<Button text="Go to Home" [nsRouterLink]="['/home']"
    pageTransition="slide"
    clearHistory="true"></Button>
```

Router Extension

NativeScript provides RouterExtensions class and exposes the navigation function of the core NativeScript.

The methods exposed by RouterExtensions are as follows:

- navigate
- navigateByUrl
- back
- canGoBack
- backToPreviousPage
- canGoBackToPreviousPage

A simple example code using RouterExtensions is as follows:

```
import { RouterExtensions } from "nativescript-angular/router";

@Component({
    // ...
})
export class HomeComponent {

    constructor(private routerExtensions: RouterExtensions) {
    }
}
```

Custom Route Reuse Strategy

NativeScript uses a custom route reuse strategy (RouterReuseStrategy) to accommodate the architecture of a mobile application. A mobile application differs in certain aspects in comparison to a web application.

For example, the page can be destroyed in a web application when user navigates away from the page and recreates it when the user navigates to the page. But, in mobile application, the page will be preserved and reused. These concepts are taken into consideration while designing the routing concept.

Routes

A simple routing module in NativeScript Angular application will be as below:

```
import { NgModule } from "@angular/core";
import { Routes } from "@angular/router";
import { NativeScriptRouterModule } from "nativescript-angular/router";

import { HomeComponent } from "./home.component";
import { SearchComponent } from "./search.component";

const routes: Routes = [
    { path: "", redirectTo: "/home", pathMatch: "full" },
    { path: "home", component: HomeComponent },
    { path: "search", component: SearchComponent },
];

@NgModule({
    imports: [NativeScriptRouterModule.forRoot(routes)],
    exports: [NativeScriptRouterModule]
})
export class AppRoutingModule { }
```

Here,

Routing module is very similar to Angular version except very few exceptions. In reality, NativeScript uses its core navigation strategy by exposing it in a way similar to Angular framework.

9. NativeScript — Events Handling

In every GUI application, events play a very important role of enabling user interaction. Whenever user interact with the application, an event fires and a corresponding action will be executed.

For example, when user clicks the Login button in the login page of an application, it triggers the login process.

Events involve two actors:

- **Event sender:** object, which raise the actual event.
- **Event listener:** function, which listen for a particular event and then executed when an event is fired.

Observable Class

It is a pre-defined class to handle events. It is defined below:

```
const Observable = require("tns-core-modules/data/observable").Observable;
```

In NativeScript, almost every object derives from Observable class and so every object support events.

Event Listener

Let us understand how to create an object and add an event listener to the object in this chapter.

Step 1

Create a button that is used to generate an event as specified below:

```
const Button = require("tns-core-modules/ui/button").Button;
const testButton = new Button();
```

Step 2

Next add text to the button as specified below:

```
testButton.text = "Click";
```

Step 3

Create a function, onTap as specified below:

```
let onTap = function(args) {
  console.log("you clicked!");
};
```

Step 4

Now attach tap event to the onTap function as specified below:

```
testButton.on("tap", onTap, this);
```

An alternate way to add an event listener is as follows:

```
testButton.addEventListener("tap", onTap, this);
```

Step 5

An alternative way to attach event is through UI itself as specified below:

```
<Button text="click" (tap)="onTap($event)"></Button>
```

Here,

$event is of type EventData. EventData contains two property and they are follows:

Object: Observable instance that is used to raise an event. In this scenario, it is Button object.

EventName: It is the event name. In this scenario, it is tap event.

Step 6

Finally, an event listener can be detached / removed at any time as specified below:

```
testButton.off(Button.onTap);
```

You can also use another format as shown below:

```
testButton.removeEventListener(Button.onTap);
```

Modifying BlankNgApp

Let us modify the BlankNgApp application to better understand the events in NativeScript.

Step 1

Open the home component's UI, src/app/home/home.component.html and add below code:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>
```

```
<StackLayout>
    <Button text='Fire an event' class="-primary" color='gray'
        (tap)='onButtonTap($event)'></Button>
</StackLayout>
```

Here,

- tap is the event and Button is event raiser.
- onButtonTap is the event listener.

Step 2

Open the home component's code, 'src/app/home/home.component.ts' and update the below code:

```
import { Component, OnInit } from "@angular/core";
import { EventData } from "tns-core-modules/data/observable";
import { Button } from "tns-core-modules/ui/button"

@Component({
    selector: "Home",
    templateUrl: "./home.component.html"
})
export class HomeComponent implements OnInit {

   constructor() {
    // Use the component constructor to inject providers.
    }

    ngOnInit(): void {
    // Init your component properties here.
    }

    onButtonTap(args: EventData): void {
        console.log(args.eventName);

        const button = <Button>args.object;
        console.log(button.text);
    }
}
```

Here,

- Added new event listener, onButtonTap.
- Print the event name, tap and button text, Fire an event in the console.

Step 3

Run the application and tap the button. It prints the below line in the console.

```
LOG from device <device name>: tap
LOG from device <device name>: Fire an event
```

10. NativeScript — Data Binding

Data binding is one of the advanced concepts supported by NativeScript. NativeScript follows Angular data binding concept as closely as possible. Data binding enables the UI component to show/update the current value of the application data model without any programming effort.

NativeScript supports two type of data binding. They are as follows:

One-Way data binding: Update the UI whenever the model is changed.

Two-Way data binding: Sync the UI and model. Whenever the model is updated, UI is updated automatically and also whenever the UI gets data from user (UI gets updated), the model will be updated.

Let us learn both the concepts in this section.

One-Way Data Binding

NativeScript provides a simple option to enable one-way data binding in a UI component. To enable one-way data binding, just add square bracket in the property of the target UI and then assign it the necessary model's property.

For example, to update the text content of a Label component, just change the UI code as below:

```
<Label [text]='this.model.prop' />
```

Here,

this.model.prop refers to the property of the model, this.model.

Let us change our BlankNgApp to understand the one-way data binding.

Step 1

Add a new model, User (src/model/user.ts) as follows:

```
export class User {
    name: string
}
```

Step 2

Open UI of our component, src/app/home/home.component.html and update the code as below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>
```

```
<GridLayout columns="*" rows="auto, auto, auto">
    <Button text="Click here to greet" class="-primary" color='gray'
(tap)='onButtonTap($event)' row='1' column='0'></Button>
    <Label [text]='this.user.name'
        row='2' column='0'
        height="50px" textAlignment='center'
    style='font-size: 16px; font-weight: bold; margin: 0px 32px 0
25px;'></Label>
</GridLayout>
```

Here,

- Label's text is set to the user model's property name.
- Button tap event is attached to onButtonTap method.

Step 3

Open code of the home component, src/app/home/home.component.ts and update the code as below:

```
import { Component, OnInit } from "@angular/core";
import { User } from "../../model/user"

@Component({
    selector: "Home",
    templateUrl: "./home.component.html"
})
export class HomeComponent implements OnInit {
    public user: User;

    constructor() {
    // Use the component constructor to inject providers.
    this.user = new User();
    this.user.name = "User1";
    }

    ngOnInit(): void {
    // Init your component properties here.
    }

    onButtonTap(args: EventData) {
        this.user.name = 'User2';
    }
}
```

Here,

- user model is imported
- User object is created in component's constructor
- onButtonTap event is implemented. Implementation of onButtonTap updates the User object and set name of the property as User2

Step 4

Compile and run the application and click the button to change the model and it will automatically change the ***Label*** text.

The initial and final state of the application is as follows:

Initial State

One Way Data Binding Initial State is shown below:

Final State

One Way Data Binding Final State is shown below:

Two-way Data Binding

NativeScript also provides two-way data binding for advanced functionality. It binds the model data to UI and also binds the data updated in UI to model.

To do two-way data binding, use ngModel property and then surround it with [] and () as below:

```
<TextField [(ngModel)] = 'this.user.name'></TextField>
```

Let us change the BlankNgApp application to better understand the two-way data binding.

Step 1

Import NativeScriptFormsModule into the HomeModule (src/app/home/home.module.ts) as specified below:

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptCommonModule } from "nativescript-angular/common";

import { HomeRoutingModule } from "./home-routing.module";
import { HomeComponent } from "./home.component";
import { NativeScriptFormsModule } from "nativescript-angular/forms";
```

```
@NgModule({
    imports: [
    NativeScriptCommonModule,
    HomeRoutingModule,
    NativeScriptFormsModule
    ],
    declarations: [
    HomeComponent
    ],
    schemas: [
    NO_ERRORS_SCHEMA
    ]
})
export class HomeModule { }
```

Here,

NativeScriptFormsModule enables the two-way data binding. Otherwise, the two-way data binding will not work as expected.

Step 2

Change the UI of the home component as given below:

```
<ActionBar>
    <Label text="Home"></Label>
</ActionBar>

<GridLayout columns="*" rows="auto, auto">
    <TextField hint="Username" row='0' column='0'
           color="gray"
           backgroundColor="lightyellow"
           height="75px"
           [(ngModel)]='this.user.name'></TextField>
    <Label [text]='this.user.name'
        row='1' column='0'
        height="50px" textAlignment='center'
    style='font-size: 16px; font-weight: bold; margin: 0px 32px 0
25px;'></Label>
</GridLayout>
```

Here,

- Label component's text property is set with one-way data binding. If the model user is updated, then its text property will automatically get updated.

- TextField component sets the ngModel as this.user.name. If the model user is updated, then it's text property will automatically get updated. At the same time, if user changes the TextField's value, then the model gets updated as well. If the model gets updated, it will trigger Label's text property changes as well. So, if user changes data, then it will show in Label's text property.

Step 3

Run the application and try to change the value of text box.

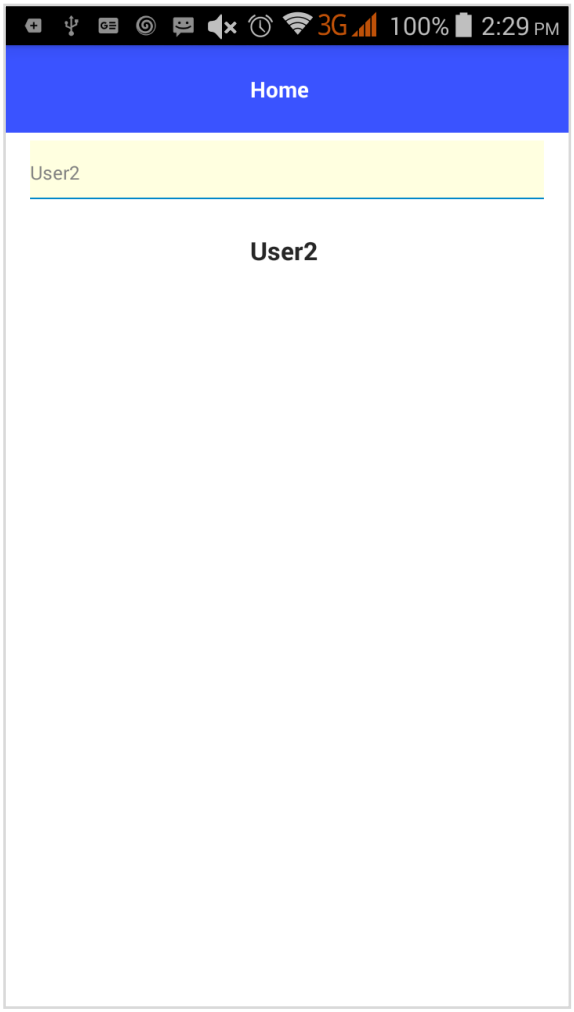
The initial and final state of the application will be similar as specified below:

Initial State

Two-way data binding – Initial state is given below:

Final State

Two-way data binding – Final state is shown below:



11. NativeScript — Modules

A *NativeScript Module* contains a set of related functionalities packaged as single library. Let us learn the modules provided by NativeScript framework.

It contains core functionalities of the NativeScript framework. Let us understand the core modules in this chapter.

Application

Application contains platform specific implementation of mobile application. Simple core module is defined below:

```
const applicationModule = require("tns-core-modules/application");
```

Console

Console module is used to log message. It has the following methods:

```
console.log("My FirstApp project");
console.info("Native apps!");
console.warn("Warning message!");
console.error("Exception occurred");
```

application-settings

application-settings module contains method to manage application settings. To add this module, we need to add the following code:

```
const appSettings = require("tns-core-modules/application-settings");
```

Few methods available in the *application-setting* are as follows:

- setBoolean(key: string, value: boolean) - set boolean object
- setNumber(key: string, value: number) - set number object
- setString(key: string, value: string) - sets string object
- getAllKeys() - Contains all stored keys
- hasKey(key: string) - check whether a key present or not
- clear - clears stored values
- remove - remove any entry based on key.

A simple example using application setting is as follows:

```
function onNavigatingTo(args) {
    appSettings.setBoolean("isTurnedOff", false);
```

```
    appSettings.setString("name", "nativescript");
    appSettings.setNumber("locationX", 54.321);

    const isTurnedOn = appSettings.getBoolean("isTurnedOn");
    const username = appSettings.getString("username");
    const locationX = appSettings.getNumber("locationX");

    // Will return "not present" if there is no value for "noKey"
    const someKey = appSettings.getString("noKey", "not present");
}

exports.onNavigatingTo = onNavigatingTo;

function onClear() {
    // Removing a single entry via its key name
    appSettings.remove("isTurnedOff");

    // Clearing the whole settings
    appSettings.clear();
}
```

http

This module is used for handling ***http*** request and response. To add this module in your application, add the following code:

```
const httpModule = require("tns-core-modules/http");
```

We can send data using the following methods:

getString: It is used to make request and downloads the data from URL as string. It is defined below:

```
httpModule.getString("https://.../get").then((r) => {
    viewModel.set("getStringResult", r);
}, (e) => {
});
```

getJSON: It is used to access data from JSON. It is defined below:

```
httpModule.getJSON("https://.../get").then((r) => {
}, (e) => {
});
```

getImage: downloads the content from specified URL and return ImageSource object. It is defined below:

```
httpModule.getImage("https://.../image/jpeg").then((r) => {
}, (e) => {
});
```

getFile: It has two arguments URL and file path.

- URL - downloads the data.
- File path - save URL data into the file. It is defined below:

```
httpModule.getFile("https://").then((resultFile) => {

}, (e) => {
});
```

request: It has options argument. It is used to request options and return HttpResponse object. It is defined below:

```
httpModule.request({
    url: "https://.../get",
    method: "GET"
}).then((response) => {
}, (e) => {
});
```

Image-source

image-source module is used save image. We can add this module using the below statement:

```
const imageSourceModule = require("tns-core-modules/image-source");
```

If you want to load images from resource, use the below code:

```
const imgFromResources = imageSourceModule.fromResource("icon");
```

To add image from local file, use the below command:

```
const folder = fileSystemModule.knownFolders.currentApp();
const path = fileSystemModule.path.join(folder.path, "images/sample.png");
const imageFromLocalFile = imageSourceModule.fromFile(path);
```

To save image to the file path, use the below command:

```
const img = imageSourceModule.fromFile(imagePath);
const folderDest = fileSystemModule.knownFolders.documents();
const pathDest = fileSystemModule.path.join(folderDest.path, "sample.png");

const saved = img.saveToFile(pathDest, "png");
if (saved) {
    console.log(" sample image saved successfully!");
}
```

Timer

This module is used to execute code at specific time intervals. To add this, we need to use ***require***:

```
const timerModule = require("tns-core-modules/timer");
```

It is based on two methods:

setTimeout: It is used to delay the execution. It is represented as milliseconds.

setInterval: It is used to apply recurring at specific intervals.

Trace

This module is useful for debugging. It gives the logging information. This module can be represented as:

```
const traceModule = require("tns-core-modules/trace");
```

If you want to enable in your application then use the below command:

```
traceModule.enable();
```

ui/image-cache

image-cache module is used to handle image download requests and caches downloaded images. This module can be represented as shown below:

```
const Cache = require("tns-core-modules/ui/image-cache").Cache;
```

connectivity

This module is used to receive the connection information of the connected network. It can be represented as:

```
const connectivityModule = require("tns-core-modules/connectivity");
```

Functionality Modules

Functionality modules include lot of system/platform specific modules. Some of the important modules are as follows:

platform: Used to display the information about your device. It can be defined as given below:

```
const platformModule = require("tns-core-modules/platform");
```

fps-meter: Used to capture frames per second. It can be defined as given below:

```
const fpsMeter = require("tns-core-modules/fps-meter");
```

file-system: Used to work with your device file system. It is defined below:

```
const fileSystemModule = require("tns-core-modules/file-system");
```

ui/gestures: Used to work with UI gestures.

UI module

UI module includes the UI component and its related functionality. Some of the important UI modules are as follows:

- frame
- page
- color
- text/formatted-string
- xml
- styling
- animation

12. NativeScript — Plugins

The npm package is used for adding native functionality. Using this package, we can install or search or delete any plugins. This section explains about plugins in detail.

Commands

Following commands are used to manage plugins for your project. They are listed below:

add: It is used to install plugin.

update: Updates specified plugin and modify its dependencies.

remove: Removes the plugin.

build: It is used to build plugin for iOS or android projects.

create: Creates a plugin for your project.

Adding plugin

Below syntax is used to add a new plugin:

```
tns plugin add <plugin-name>
```

For example, if you want to add nativescript-barcodescanner, you can use the following code:

```
tns plugin add nativescript-barcodescanner
```

You could see the following response:

```
+ nativescript-barcodescanner@3.4.1
added 1 package from 1 contributor and audited 11704 packages in 8.76s
```

You can also use npm module to add the above plugin:

```
npm install nativescript-barcodescanner
```

Now, NativeScript CLI downloads the plugin from npm and add inside your node_modules folder.

If you want to add the plugin directly to your ***package.json*** and resolve all the dependency issues, you can use the below command instead of the previous one:

```
npm i nativescript-barcodescanner
```

If you want to install a developer dependencies during development, use the below code:

```
npm i tns-platform-declarations --save-dev
```

Here,

tns-platform-declarations is a developer dependency required only for intelliSense during the development process.

Importing plugins

Now, we have installed ***nativescript-barcodescanner*** plugin. Let us add inside your project using the below command:

```
const maps = require("nativescript-barcodescanner");
maps.requestPermissions();
```

Updating plugins

This method is used to update a specified plugin so it uninstalls previous one and installs new version and modify its dependencies. It is defined below:

```
tns plugin update <Plugin name version>
```

Removing plugin

If you want remove the plugin, if not required, you can use the below syntax:

```
tns plugin remove <plugin-name>
```

For example, if you want to remove the above installed nativescript-google-maps-sdk, then use the below command:

```
tns plugin remove nativescript-barcodescanner
```

You could see the following response:

```
Successfully removed plugin nativescript-barcodescanner
```

Building plugins

It is used to build the plugin's Android-specific project files located in platforms/android. Let us build the nativescript-barcodescanner pugin using the below command:

```
tns plugin build nativescript-barcodescanner
```

Creating plugins

NativeScript plugins are simple JavaScript modules. It is defined inside your application src\package.json file. This module is used to create a new project for NativeScript plugin development. It is defined below:

```
tns plugin create <Plugin Repository Name> [--path <Directory>]
```

13. NativeScript — Native APIs Using Javascript

This section explains about the overview of accessing Native APIs using JavaScript.

Marshalling

The NativeScript Runtime provides implicit type conversion for both android and iOS platforms. This concept is known as marshalling. For example, NativeScript- iOS paltform can implicitly convert JavaScript and Objective-C data types similarly, Java/Kotlin can easily be mapped to JavaScript project types and values. Let us understand how to perform marshalling in each type one by one briefly.

Numeric values

We can easily convert iOS and android numeric data types into Javascript numbers. Simple numeric conversion for iOS into Javascript is defined below:

```
console.log(`max(7,9) = ${max(7,9)}`);
```

Here,

The native max() function is converted into JavaScript number.

Android Environment

Java supports different numeric types such as byte, short, int, float, double and long. JavaScript has only number type.

Consider a simple Java class shown below:

```
class Demo extends java.lang.Object
{

    public int maxMethod(int a,int b)
    {
        if(a>b)
        {
        return a;
        }
        else
        {
        return b;
        }
    }
}
```

Here,

The above code contains two integer arguments. We can call the above code object using JavaScript as shown below:

```
//Create an instance for Demo class
var obj = new Demo();

//implicit integer conversion for calling the above method
obj.maxMethod(7,9);
```

Strings

Android strings are defined in java.lang.string and iOS strings are defined in NSSring. Let us see how to perform marshalling in both platforms.

Android

Strings are immutable but String buffers support mutable strings.

Below code is an example for simple mapping:

```
//Create android label widget
var label = new android.widget.Label();

//Create JavaScript string
var str = "Label1";

//Convert JavaScript string into java
label.setText(str); // text is converted to java.lang.String
```

Boolean class is defined in java.lang.Boolean. This class wraps a value of boolean in an object. We can easily convert boolean to String and vice-versa. Simple example is defined as given below:

```
//create java string
let data = new java.lang.String('NativeScript');

//map java String to JavaScript string,
let result = data.startsWith('N');

//return result
console.log(result); // true
```

iOS environment

NSString class is immutable but its subclass NSMutableString is immutable. This class contains a collection of methods for working with strings. It is declared as below:

```
class NSString : NSObject
```

Consider a simple objective-c declaration as shown below:

```
NSString *str = @"nativescript";

//convert the string to uppercase
```

```
NSString *str1;
str1 = [str uppercaseString];

NSLog(@"Uppercase String :  %@\n", str1 );
```

NSStrings can easily be mapped to JavaScript strings.

Array

This section explains about how to perform marshalling in arrays. Let's take an example of iOS environment first.

Array declaration

```
class NSArray : NSObject
```

Here,

NSArray is used to manage ordered collection of objects called arrays. It is used to create static array. Its sub class ***NSMutableArray*** is used to create dynamic arrays.

Consider NSArray objects can be created using array literals as shown below:

```
let array: NSArray = ["React","Vue","TypeScript"]
```

Now, we can map this array into JavaScript as shown below:

```
//create native array
let nsArr = NSArray.arrayWithArray("React","Vue","TypeScript"]);

//create simple javascript array
let jsArr = ["Hello,World","NativeScript"];


//Now compare the two arrays,
let compare = nsArr.isEqual(jsArr);

console.log(comapre);
```

This will return the output as false.

Andriod array declartion

Java arrays are defined in ***java.util.Arrays***. This class contains various methods for manipulating arrays. An example is shown below:

```
//javascript array

let data = [12,45,23,56,34,78,50];

//create java array

let result = ns.example.Math.maxElement(data);
```

```
console.log(result);
```

Classes and Objects

Classes and Objects are basic concepts of Object Oriented Programming. Class is a user defined prototype. Object is an instance of class. Class represents the set of properties or methods that are common to all objects of one type. Let us understand native classes and objects for both mobile development environments.

Android environment

Java and Kotlin classes have unique identifiers denoted by the full package name.

For example,

android.view.View: It is a basic user interface class for screen layout and interaction with the user. We can access this class in JavaScript as shown below:

```
const View = android.view.View;
```

First, we import the class using the below statement:

```
import android.view.View;
```

Next create a class as given below:

```
public class MyClass
 {
    public static void staticMethod(context)
    {

        //create view instance
        android.view.View myview = new android.view.View(context);
    }
}
```

In the above same class, we can access JavaScript function using the below code:

```
const myview = new android.view.View(context);
```

Similarly, we can access interfaces, constants and enumerations within java.lang packages.

iOS environment

Objective - C classes are defined in two sections @interface and @implementation. Class definition starts with the keyword **@interface** followed by the **interface(class)** name. In Objective-C, all classes are derived from the base class called NSObject.

It is the superclass of all Objective-C classes. Simple Circle class is defined as shown below:

```
@interface Circle:NSObject
{
   //Instance variable
   int radius;

}
@end
```

Consider a class with one method as shown below:

```
@interface MyClass : NSObject

+ (void)baseStaticMethod;

@end
```

This class can be converted to javascript using the below code:

```
function MyClass() { /* native call */ };
Object.setPrototypeOf(MyClass, NSObject);
BaseClass.baseStaticMethod = function () { /* native call */ };
```

JavaScript ***instanceof*** operator is used to verify, if an object inherits from a given class. This can be defined as:

```
var obj = MyClass.alloc().init();  // object creation

console.log(obj instanceof NSObject); //return true
```

Here,

Objective - C instances are created using alloc, init or new methods. In the above example, we can easily create object initialization using new method as below:

```
var obj = MyClass.new();
```

Similarly, you can access static methods and properties.

14. NativeScript — Creating an Application in Android

Create and publish your app makes your Android application available to all users. Google Play is a robust publishing platform. It helps you to publish and distribute your Android applications to all the users around the whole world. This chapter explains about how to publish your Native app in Google Play.

NativeScript Sidekick

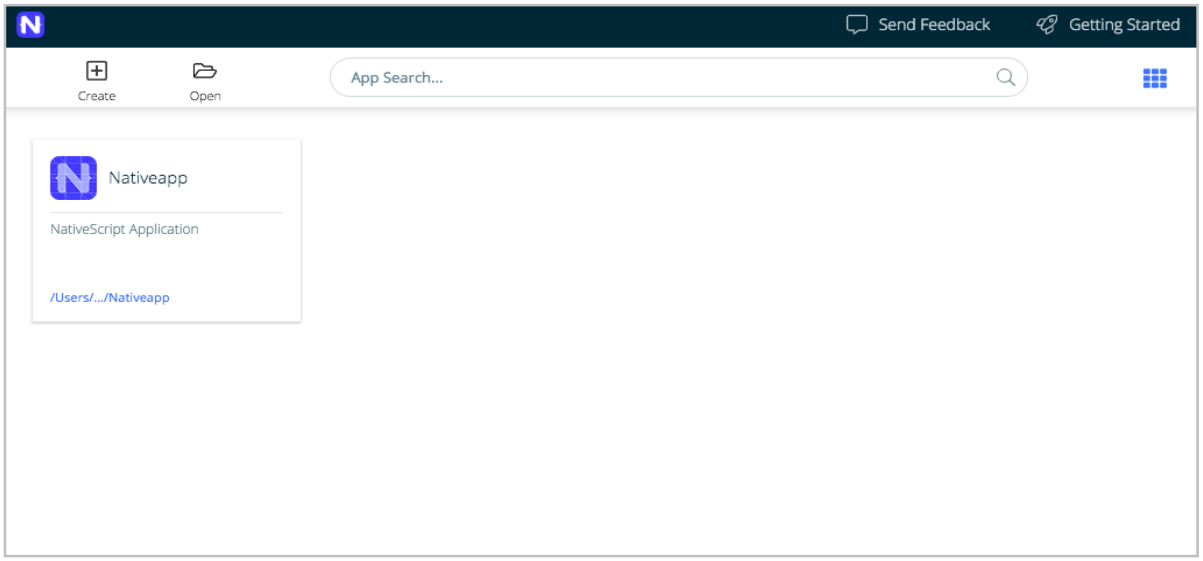
SideKick is a GUI client and supports all kind of OS. It simplifies NativeScript CLI process and helps to create mobile application.

Publish your app from Sidekick to Google Play Console

Downloading and installing sidekick depends on your OS. Follow the below steps to run your app in Sidekick.

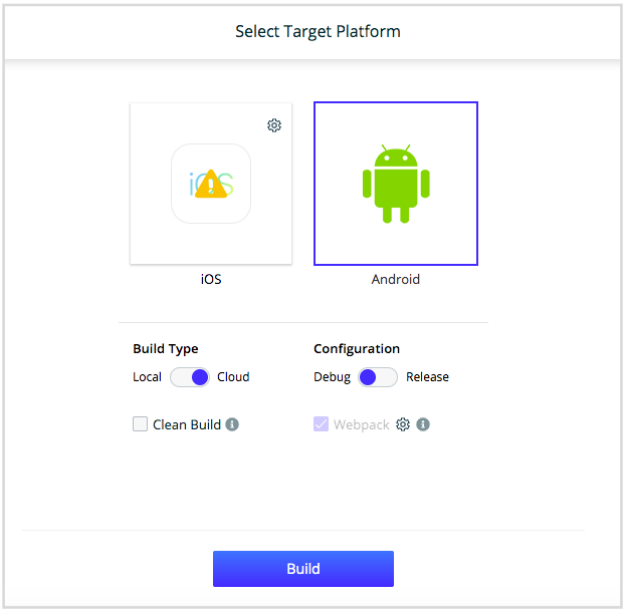
Step 1: Launch Sidekick

Let us Launch Sidekick. It looks similar to the below image:



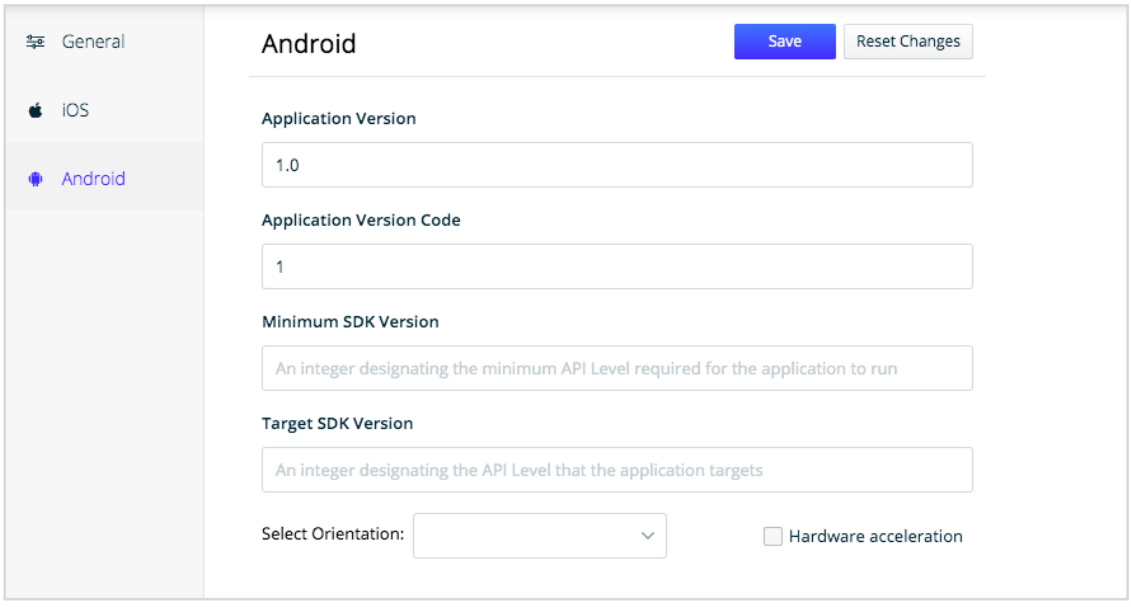
Step 2: Build your device

Now, open your app from your device and select build option from the toolbar and select Android. You will get a response similar to the below image:



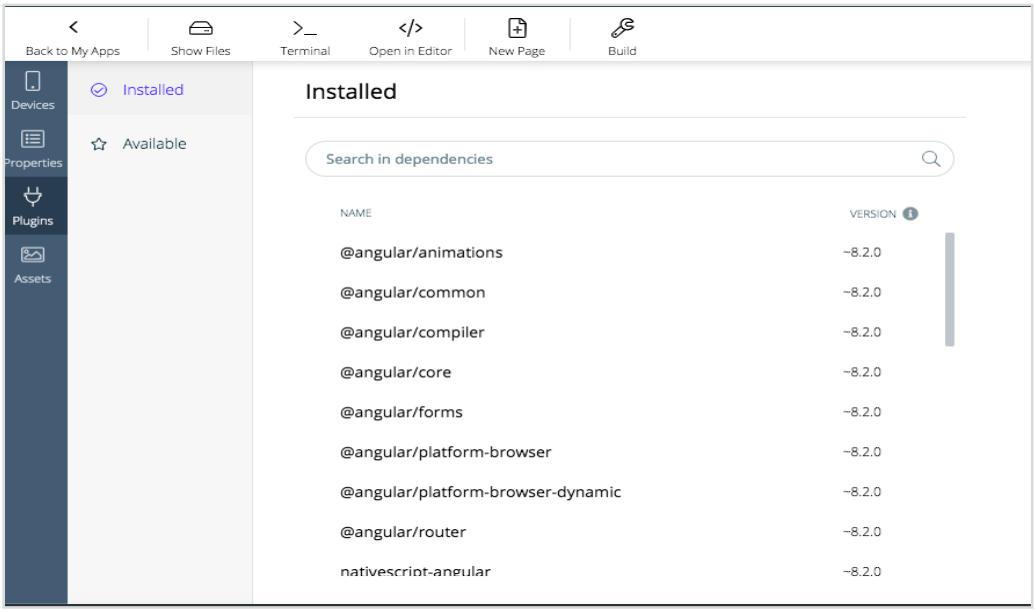
Step 3: Properties

Click properties tab and add your android configuration. Screen looks similar to the below one:



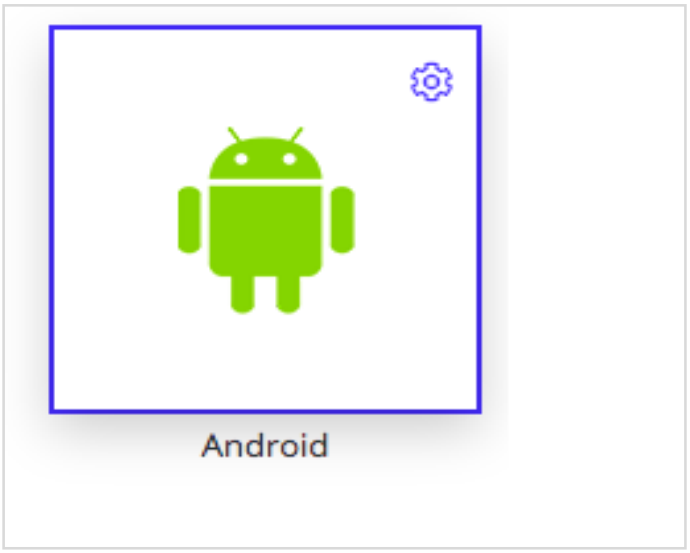
Step 4: Plugins

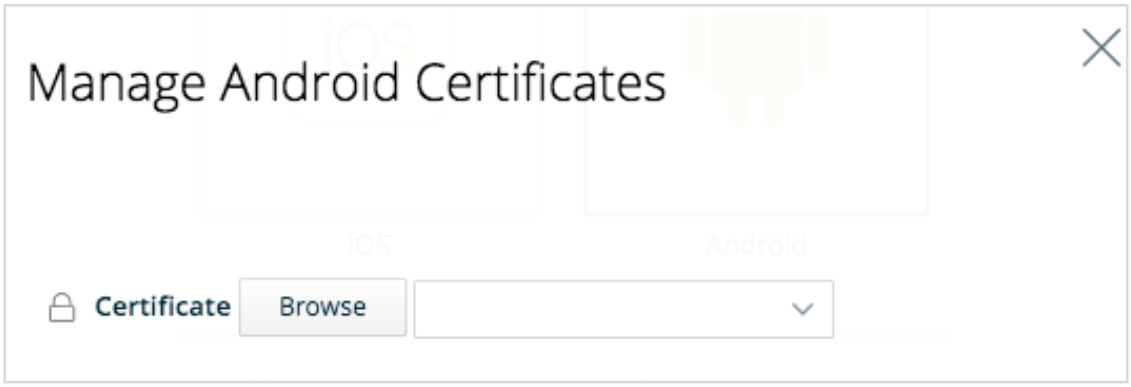
Sidekick helps to find which plugins you depend on for your application. Click on plugins tab and it will list out the following:



Step 5: Android Certificates

Click cogwheel icon from android and choose browse option, then select a certificate stored on your file system. It is shown below:





After selecting that, close the dialog box.

Step 6: Build your application

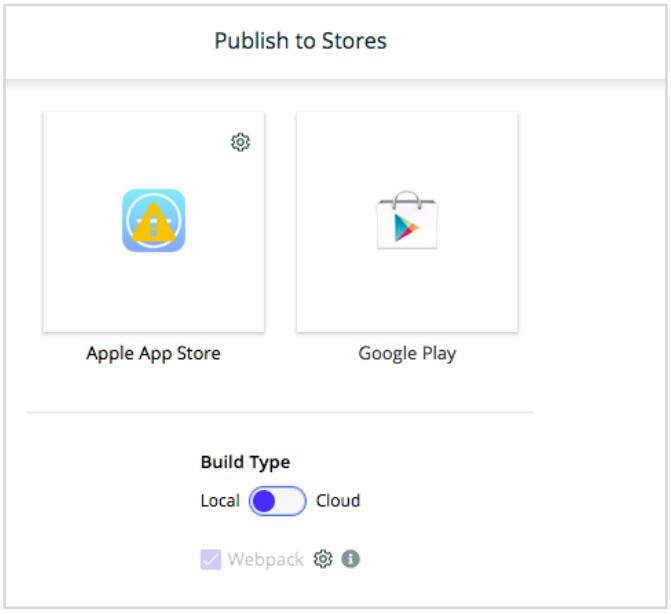
Finally click local build option from build type and select release option from configuration. After that build your application.

Step 7: Application package

Once build is completed, it will generate a path and ***apk*** file. Save the location of the application package. This apk file is used to upload it to the Google Play store.

Step 8: Publish in Google Play

Select publish option from the toolbar and select Google Play. Then, add Manage Android Certificates for Google Play Store dialog. It is shown below:



After that, select Build type and provide Service Account JSON key then select Alpha, Beta or Production tracks finally click upload.

Publish your app in Google Play

To publish your app in Google Play console, you must meet the following prerequisites.

Prerequisites

- You must be registered in Google Play
- You have a valid Google Play self-signed code signing identity

Procedure for publish your app

Below steps are helpful to understand how to release your app in Google Play store.

Step 1: Login Google Play console

Open Google Play console and login with your account.

Step 2: Create an app

Go to the All applications tab and click Create Application and create a new app. Now, add default language,application title finally click proceed to go further.

Step 3: Fill required fields

Move to store listing tab and fill the required fields, then complete the needed assets and save all the changes.

Step 4: Price and distribution

Go to Pricing & distribution tab, complete all the settings and save all the changes.

Step 5: Release your app

Choose App releases tab and select Alpha, Beta. It is used for testing your application. And, select Production tracks. It is used for publishing your app to Google Play. Finally add the application package (apk).

Step 6: Review your app

This is your final step. In the Review, verify if there are any issues. If no issues, then confirm rollout to publish your app.

15. NativeScript — Creating an Application in iOS

This chapter explains about how to publish your Native app in App Store. Go through the below steps to publish your app.

Prerequisites

To perform this, you must need the following prerequisites:

- Certificate for distribution
- Distribution provisioning profile
- Registered bundle ID in iOS Dev center
- App record in iTunes Connect

Steps to publish your app

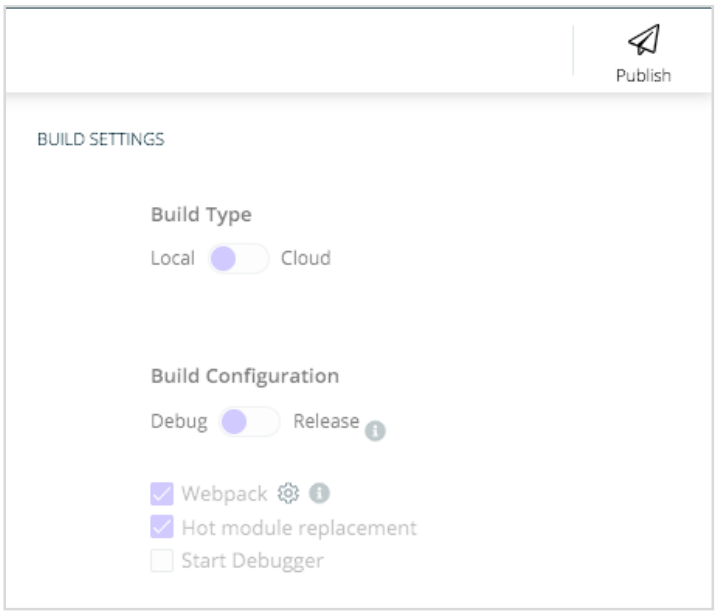
Below are the steps to publish your app:

Step 1: Open NativeScript Sidekick

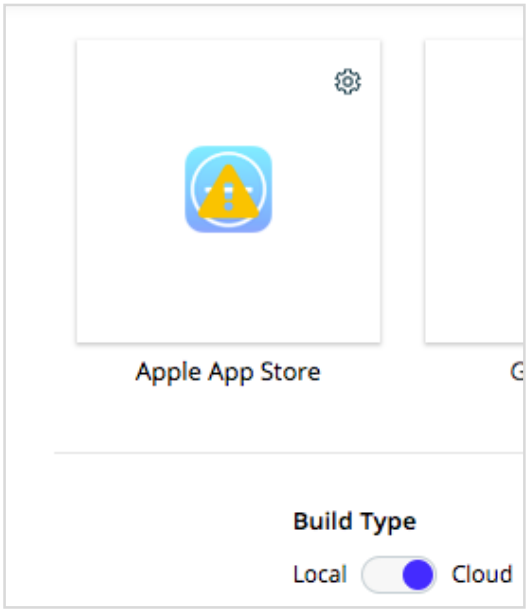
Launch NativeScript Sidekick and open your app in Sidekick.

Step 2: Select publish

Go to toolbar and select ***publish*** option from the toolbar. It is shown below:

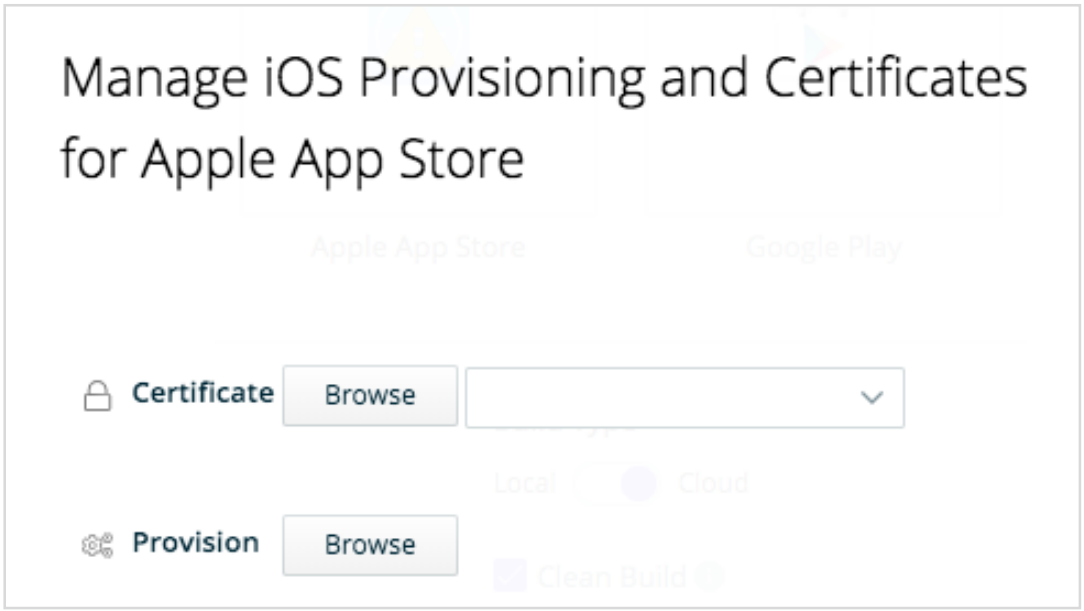


Now, select Apple App Store option. It looks similar to the below image:



Step 3: Manage iOS provision and certificates

Click the Apple App Store cogwheel icon and choose browse option and add the details.



Step 4: Build your app

Next, click build option and build your app and wait till the process to complete.

Step 5: Provide credentials

This is your final step. Specify Apple Username and Password in your account and click upload and check the confirmation message. If you want to submit your app for review, then go to iTunes Connect and submit it.

16. NativeScript — Testing

Testing is a very important phase in the development life cycle of an application. It ensures an application quality. It needs careful planning and execution. It is also most time consuming phase of the development. NativeScript framework provides an extensive support for the automated testing of an application.

Types of Testing

Generally, three types of testing processes are available to test an application. They are as follows:

Unit Testing

Unit testing is the easiest method to test an application. It is based on ensuring the correctness of a piece of code (a function, in general) or a method of a class. But, it does not reflect the real environment and subsequently. It is the least option to find the bugs.

Generally, NativeScript uses Jasmine, Mocha with Chai and QUnit unit testing frameworks.

To perform this, first you need to configure in your project using the below command:

```
tns test init
```

Now, you get the following response:

```
? Select testing framework: (Use arrow keys)
❯ jasmine
  mocha
  qunit
```

Now, select ***jasmine*** framework and your screen looks similar to this:

```
? Select testing framework: jasmine
+ karma@4.4.1
added 90 packages from 432 contributors and audited 11944 packages in 8.753s

+ karma-nativescript-launcher@0.4.0
added 2 packages from 1 contributor and audited 11946 packages in 7.299s

> core-js@2.6.11 postinstall
/Users/workspace/NativeScript/NativeApp/node_modules/core-js
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for
polyfilling JavaScript standard library!

The project needs your help! Please consider supporting of core-js on Open
Collective or Patreon:
```

```
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a
good job -)

npm WARN karma-webpack@3.0.5 requires a peer of webpack@^2.0.0 || ^3.0.0 but
none is installed. You must install peer dependencies yourself.

+ karma-webpack@3.0.5
added 19 packages from 52 contributors and audited 12012 packages in 9.368s


+ karma-jasmine@2.0.1
added 2 packages from 35 contributors and audited 12014 packages in 6.925s

+ karma@4.4.1
updated 1 package and audited 12014 packages in 7.328s

+ @types/jasmine@3.4.6

> nativescript-unit-test-runner@0.7.0 postinstall
/Users/deiva/workspace/NativeScript/NativeApp/node_modules/nativescript-unit-
test-runner
> node postinstall.js

+ nativescript-unit-test-runner@0.7.0
added 1 package from 1 contributor and audited 12032 packages in 7.14s

Successfully installed plugin nativescript-unit-test-runner.

Example test file created in src/tests
Run your tests using the "$ tns test <platform>" command.
```

Now, the test file is created inside src\tests\example.ts.

Create your tests

Let us add a simple test inside example.ts file as shown below:

```
describe("NativeApp test:", function() {
  it("Check counter.", function() {
    expect(mainViewModel.createViewModel().counter).toEqual(10);
  });
  it("Check message.", function () {
    expect(mainViewModel.createViewModel().message).toBe("10 taps left");
  });
});
```

Here,

First, check if the counter equals 10 and check if the message is 10 taps left.

Let us run the test in next step.

Run your tests

Now, run the test in either android or iOS connected device using the below command:

```
tns test andriod
```

This will return the following status:

```
? To continue, choose one of the following options:  (Use arrow keys)
❯ Configure for Cloud Builds
  Configure for Local Builds
  Configure for Both Local and Cloud Builds
  Skip Step and Configure Manually
```

Then choose the below option:

```
? To continue, choose one of the following options:  Configure for Local Builds
Running the setup script to try and automatically configure your environment.
These scripts require sudo permissions
.....
```

To execute your test suite in the android simulator, run the following command:

```
tns test android --emulator
```

Now, karma server prepares builds and deploy your project.

End To End (E2E) Testing

Unit tests are small, simple and fast process whereas, E2E testing phase multiple components are involved and works together which cover flows in the application. This could not be achieved by unit and integration tests.

NativeScript Appium plugin is used to perform E2E automation testing. Well, Appium is an open source testing framework for mobile app. To add this framework in your project, you must have either latest version of XCode or Android SDK above 25.3.0.

Install Appium

Let us install Appium globally using npm module:

```
npm install -g appium
```

Now, you could see the following response:

```
npm install -g appium

/Users/.npm-global/bin/authorize-ios -> /Users/.npm-global/lib/node_modules/
appium/node_modules/.bin/authorize-ios

> appium-windows-driver@1.8.0 install /Users/.npm-global/lib/node_modules/
appium/node_modules/appium-windows-driver
```

```
> node install-npm.js

Not installing WinAppDriver since did not detect a Windows system

> core-js@2.6.11 postinstall /Users/.npm-
global/lib/node_modules/appium/node_modules/core-js
> node -e "try{require('./postinstall')}catch(e){}"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for
polyfilling JavaScript
standard library!

The project needs your help! Please consider supporting of core-js on Open
Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a
good job -)


> appium-chromedriver@4.19.0 postinstall /Users/.npm-
global/lib/node_modules/appium/node_modules
/appium-chromedriver
> node install-npm.js

.........................................

.........................................

+ appium@1.16.0
added 671 packages from 487 contributors in 28.889s
```

Add plugin

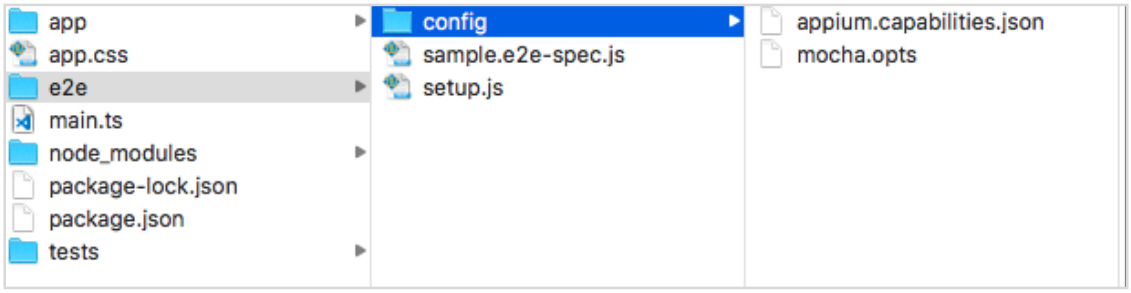
Let us add ***nativescript-dev-appium*** plugin as a devDependency to your project using the below command:

```
$ npm install -D nativescript-dev-appium
```

After executing this, choose ***mocha*** framework and you will get a response similar to this:

```
> node ./postinstall.js

? What kind of project do you use? javascript
? Which testing framework do you prefer? mocha

+ nativescript-dev-appium@6.1.3
```

Now, files are stored inside your project folder.



Build your device

Let us build android device using the below command:

```
tns build android
```

The above command will run the tests should specify the targeted capabilities. If you have iOS device, you can build using ***iOS*** device.

Run test

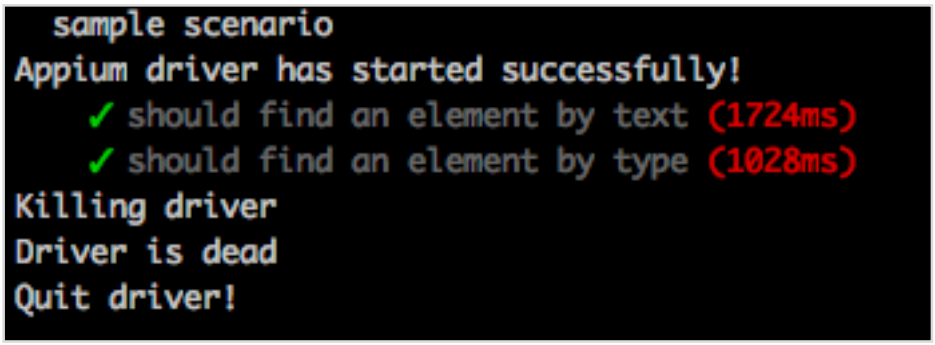
Now, we have configured the device. Let us run our test using the below command:

```
npm run e2e -- --runType <capability-name>
```

Here,

capability-name is defined inside your application e2e/config/appium.capabilities.json.

Output



NativeScript — Conclusion

NativeScript is a great mobile app for web developers to test their application completely in a very easy way without putting extra efforts. Developers can confidently develop a great looking as well as a successful application without any issues in a short period of time.