Introduction to React Native

We will first focus theoretically on React Native and later we will show how to create a very basic React Native App.

Later you will get to know also how we can integrate Google Vision API in our React Native application and get responses on image recognition. During these steps we will explain with details how to implement several features that are included in it like: Object Detection, Labeling Images, Text Recognition and Face Detection. After detecting the object we are going to use Google Translate API to translate it into another language.

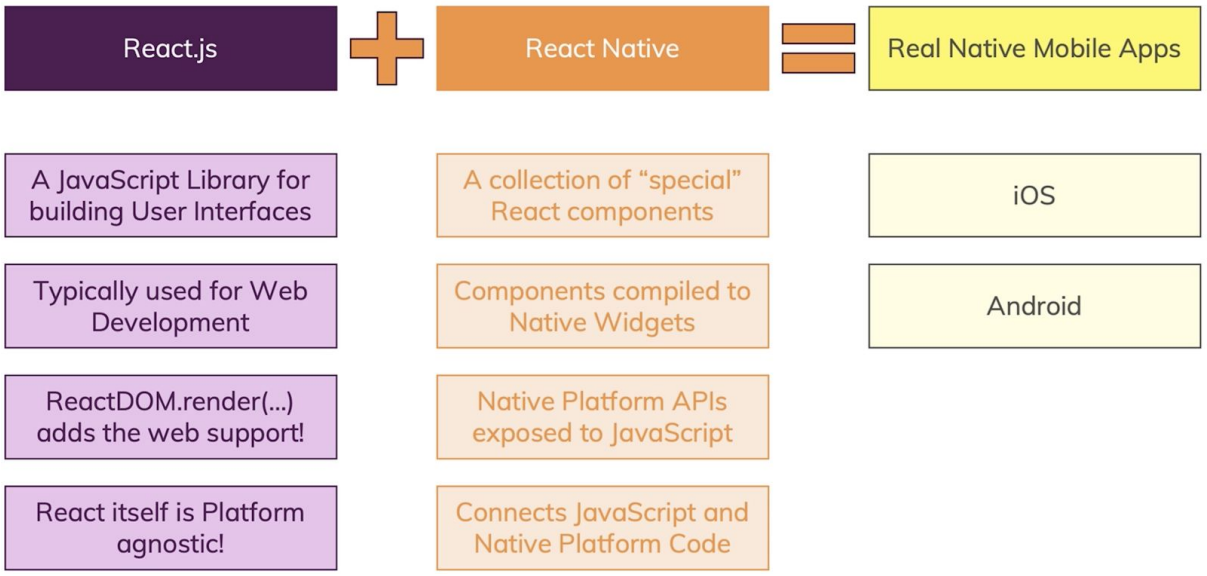
1. Developing Mobile Applications

Based on what you want to build, you can change your tech stack accordingly to your expertise and to the easiest choice. For this reason let's discuss how we should build mobile applications. Nowadays we have a lot of opportunities:

1. You can choose Kotlin, Java, Swift or ObjectiveC to build true nativescripts apps.
2. You can build a Progressive Web App (PWA are web applications that are regular web pages or websites, but can appear to the user like traditional applications or native mobile applications)
3. You can use React Native, Flutter or Ionic.

To understand more about the differences between them follow this link: *React Native, Flutter, Ionic vs Nativescript vs PWA*

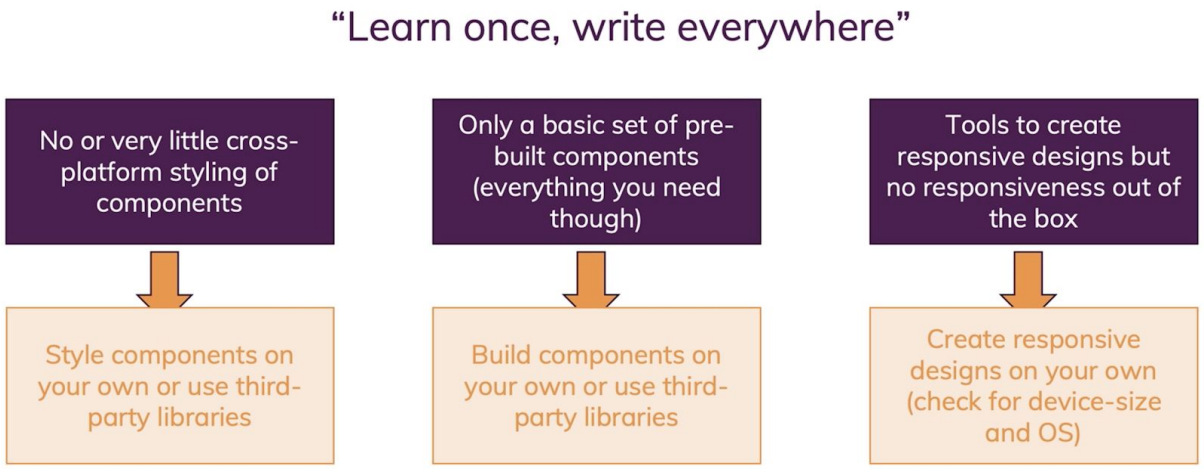
2. What is React Native?



React Native is a JavaScript framework for writing real, natively rendering mobile applications for *iOS and Android*.

It's based on React for building user interfaces, but instead of targeting the browser, it targets mobile platforms.

In other words: web developers can now write mobile applications that look and feel truly "native," all from the comfort of a JavaScript library that they already know.



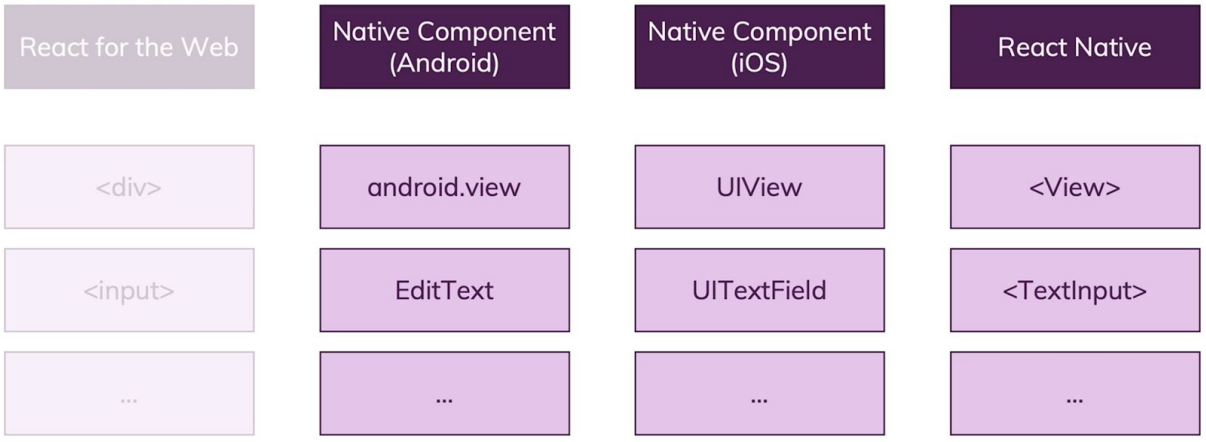
*__Agnostic__, in an information technology (IT) context, refers to something that is generalized so that it is interoperable among various systems.

3. RN Apps are Hard Work

React Native Applications are a fast moving target and that makes them very hard and demanding, because you always have to keep up with new versions and bugs related to third-party packages that may also change.

So now the question is, should you learn it? To answer this question, let's go through the React Native logic.

4. Behind the scenes

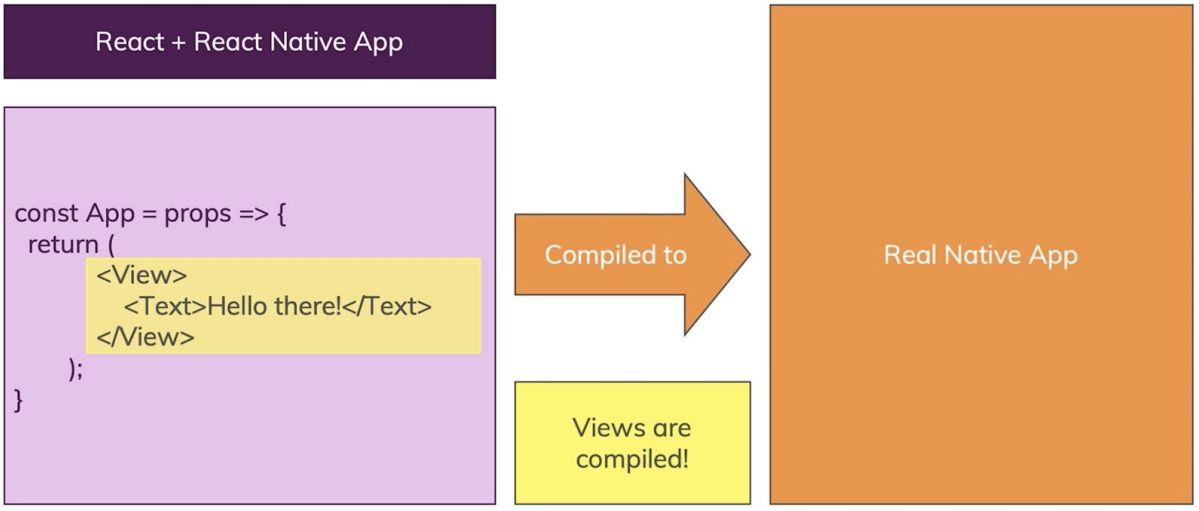


Similar to React for the Web, React Native applications are written using a mixture of JavaScript and XML-Esque markup, known as *JSX*.

You can develop in different ways to create mobile and web applications.

- If you want to create *Web Applications* you can use *ReactJS*, but we are now focusing only on creating Mobile Applications so we fade the web part.

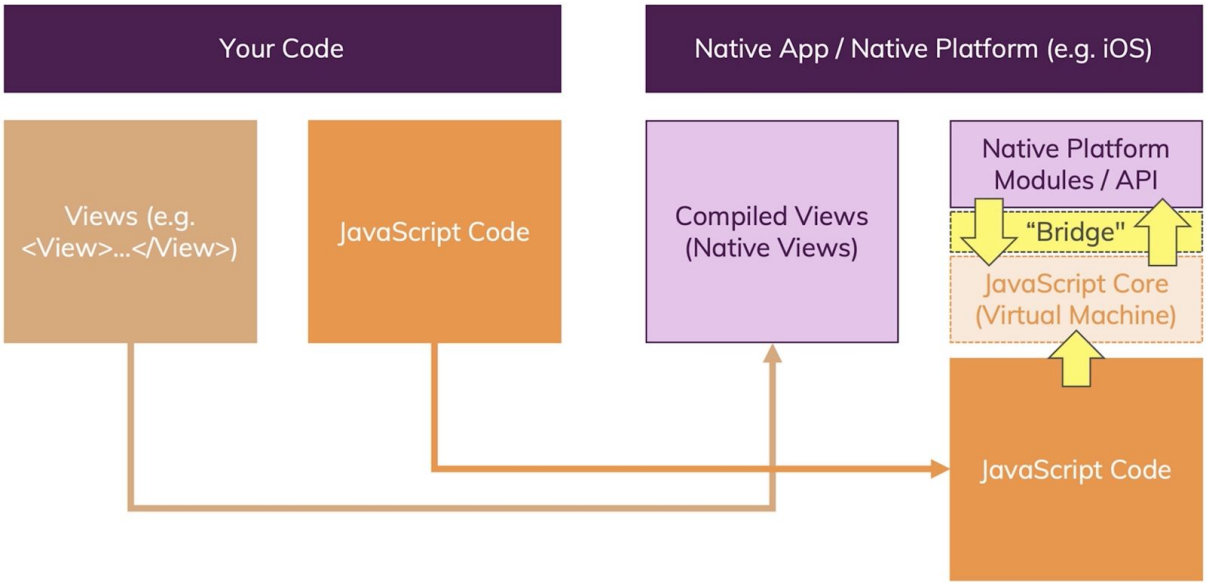
- If you want to create *Mobile Applications* you can use *Native Component (iOS or Android)* or *React Native*.



The most fundamental component for building a UI is **View**, a container that supports layout with flexbox, style, touch handling, and accessibility controls. View maps directly to the native view equivalent on whatever platform React Native is running on, whether that is a UIView, , android.view, etc.

We can see the application as divided into two parts, the first one is UI , which includes react components that are compiled like native views and the second one is Logic, which includes the javascript code we write and is not compiled as native view.

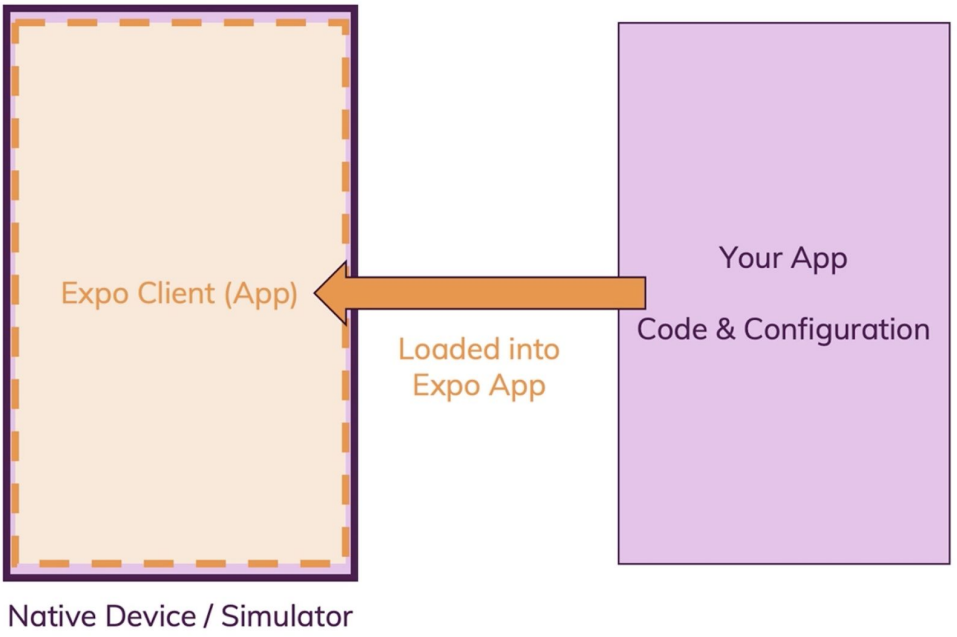
Then, under the hood, the React Native "bridge" invokes the native rendering APIs in Objective-C (for iOS) or Java (for Android) and connects them to our javascript core.



5. Expo CLI vs React Native CLI

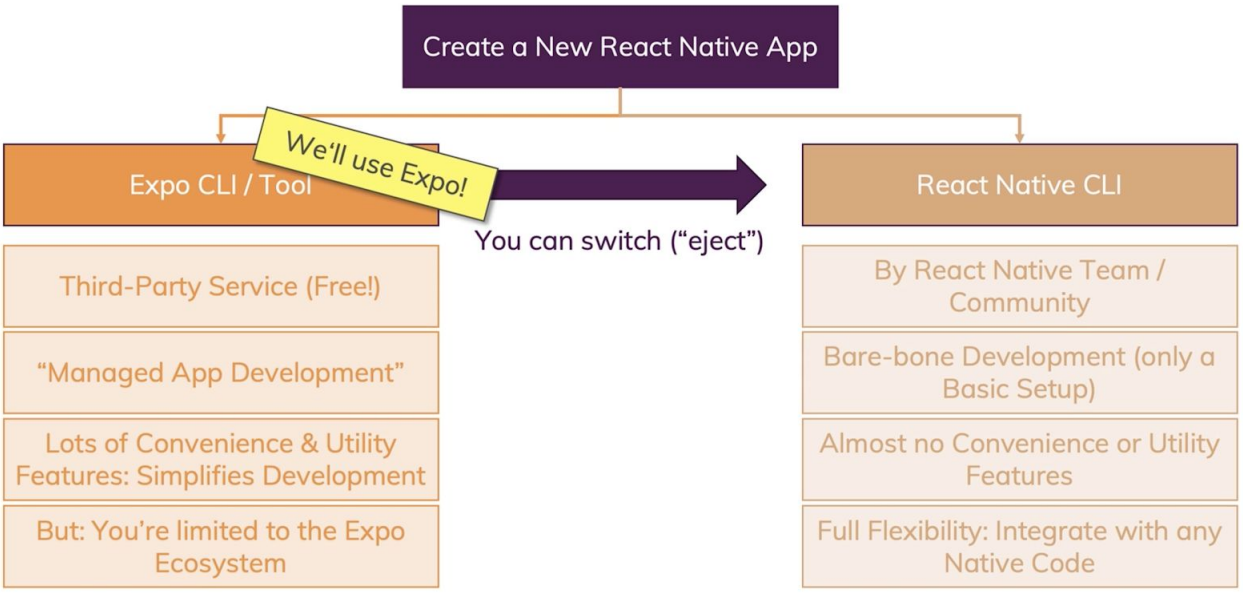
There are two different ways to create React Native Apps:

- If you are new to mobile development, the easiest way to get started is with **Expo CLI**. Expo is a set of tools built around React Native and while it has many features, the most relevant feature for us right now is that it can get you writing a React Native app within minutes. You will only need a recent version of Node.js and a phone or emulator.



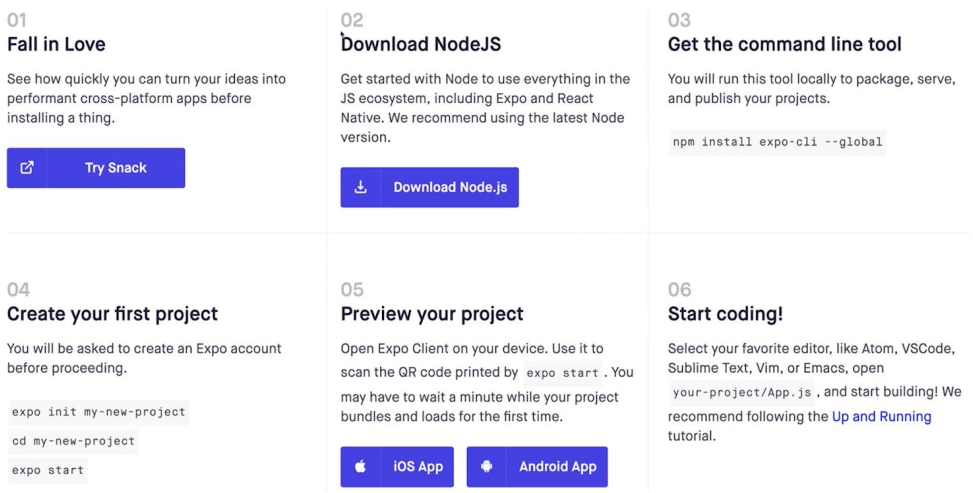
!If you'd like to try out React Native directly in your web browser before installing any tools, you can try out Snack.

- If you are already familiar with mobile development, you may want to use **React Native CLI**. It requires Xcode or Android Studio to get started. If you already have one of these tools installed, you should be able to get up and running within a few minutes. If they are not installed, you should expect to spend about an hour installing and configuring them.



6. Steps for using Expo

After we decided that we are going to use Expo, we go into their website and follow the steps above:



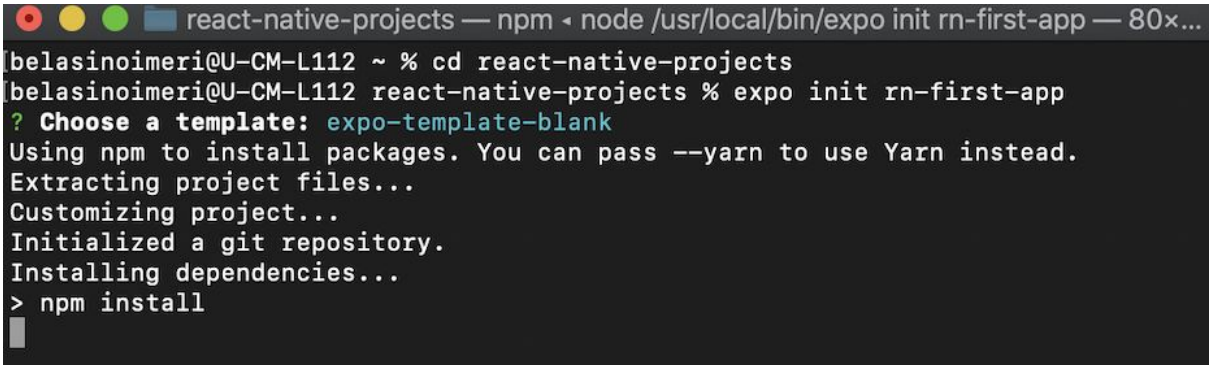
02: Download NodeJS if you already do not have it on your pc.

03: Get the expo cli tool with the command:

npm install expo-cli --global

04: Create your first project with name: rn-first-app

expo init rn-first-app



05: Run it locally with:

expo start or *npm start*

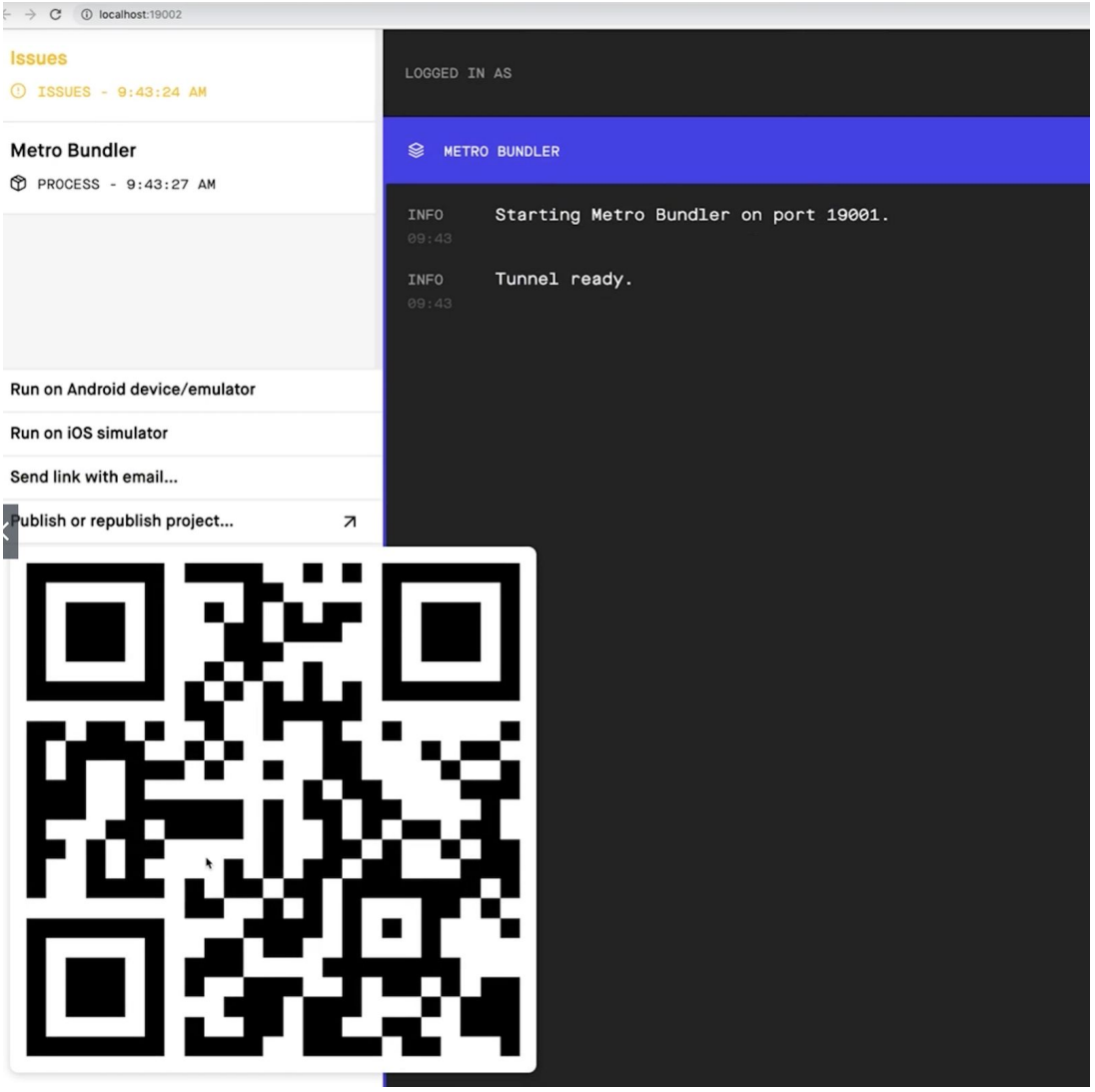
```
belasinoimeri@U-CM-L112 react-native-projects % cd rn-first-app
belasinoimeri@U-CM-L112 rn-first-app % npm start

> @ start /Users/belasinoimeri/react-native-projects/rn-first-app
> expo start

Starting project at /Users/belasinoimeri/react-native-projects/rn-first-app
Expo DevTools is running at http://localhost:19002
Opening DevTools in the browser... (press shift-d to disable)
Starting Metro Bundler on port 19001.
Tunnel ready.

  exp://192.168.178.56:19000

  To run the app with live reloading, choose one of:
  • Scan the QR code above with the Expo app (Android) or the Camera app (iOS).
  • Press a for Android emulator, or i for iOS simulator, or
    w to run on web.
  • Press e to send a link to your phone with email.
  • Press s to sign in and enable more options.

 Expo  Press ? to show a list of all available commands.
Logs for your project will appear below. Press Ctrl+C to exit.
```

CTRL + C cancels this running process. If you did, you can run npm start again to restart the process of course.

Metro is a JavaScript bundler which takes in options, an entry file, and gives you a JavaScript file including all JavaScript files back. Every time you run a react native project, a compilation of many javascript files are done into a single file. This compilation is done by a bundler which is called Metro and it will open a new window like below.

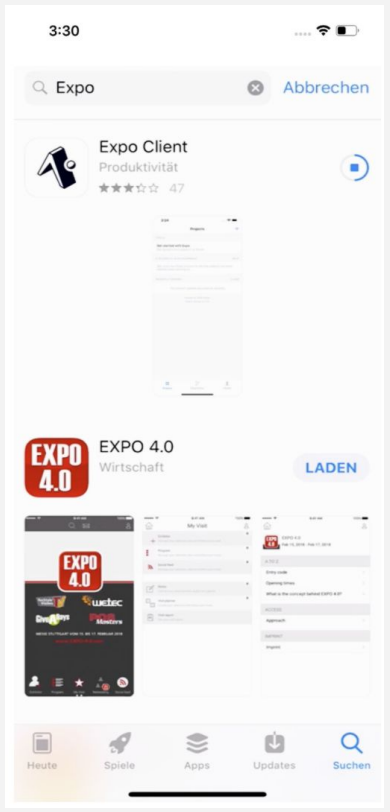
There are different ways you can run the application:

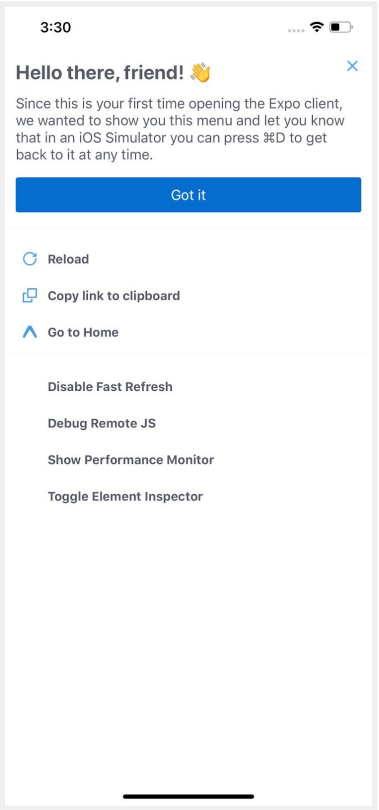
1. Directly on the screen at Metro Bundler you click into Run on Android device/emulator or Run on iOS simulator.

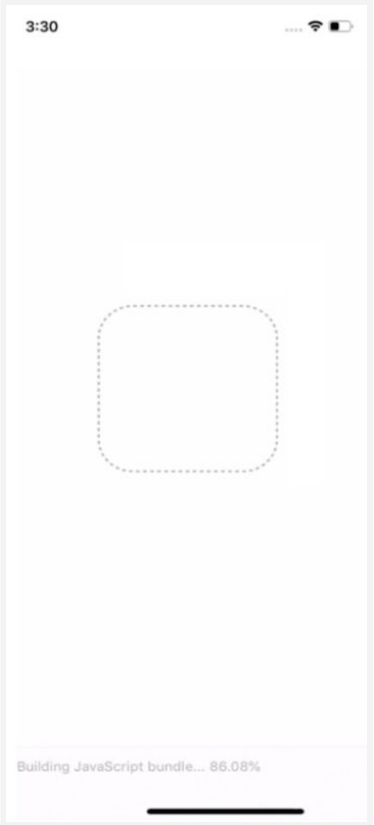


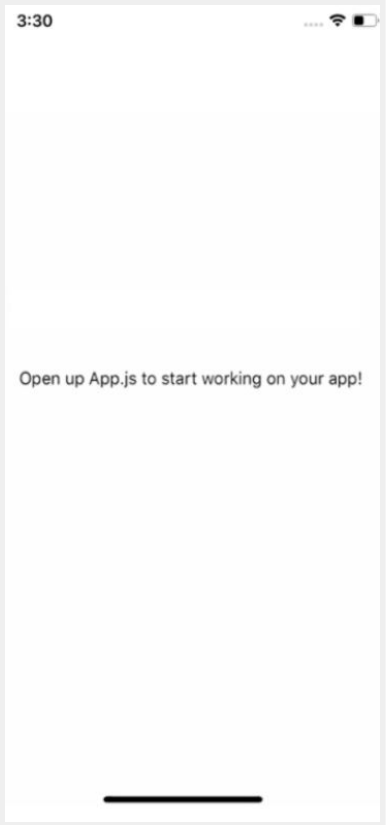
2. You can run it on your **own phone**.

   In order to do so, you need to install Expo on your phone. (Below is an example of downloading Expo on iPhone, AppStore). After downloading and having Expo, you can open your phone at the camera application and center the QR code. Then a modal will pop up saying open with Expo, click on it and you will see the JS bundler being loaded in your phone.


3:30
Expo
Abbrechen
Expo Client
Produktivität
47
EXPO 4.0
Wirtschaft
LADEN
Heute
Spiele
Apps
Updates
Suchen

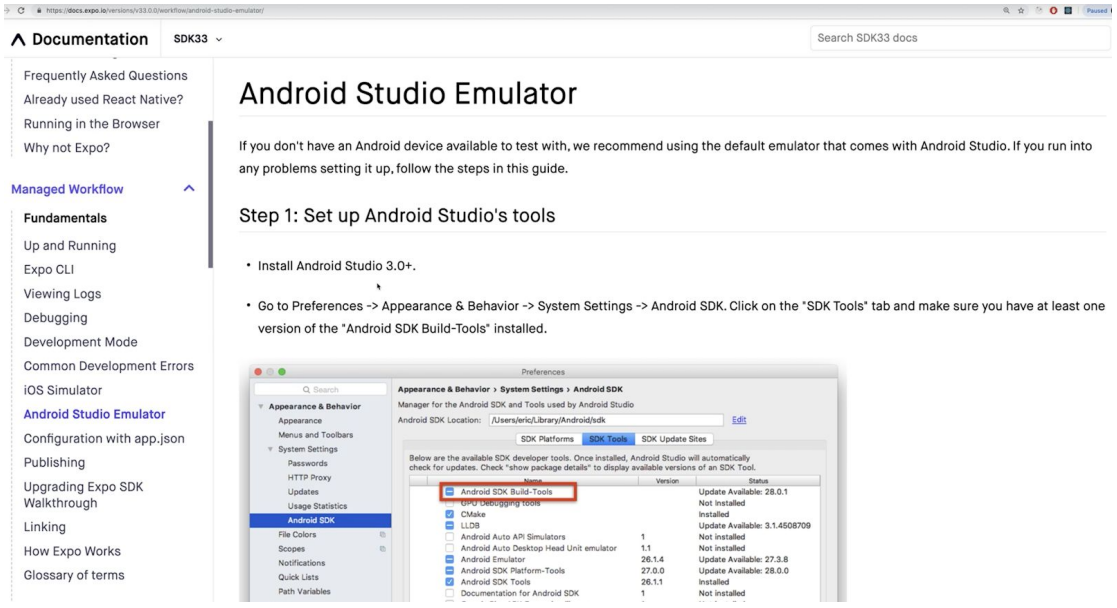

3:30
Hello there, friend! 👋
Since this is your first time opening the Expo client, we wanted to show you this menu and let you know that in an iOS Simulator you can press ⌘D to get back to it at any time.
Got it
Reload
Copy link to clipboard
Go to Home
Disable Fast Refresh
Debug Remote JS
Show Performance Monitor
Toggle Element Inspector

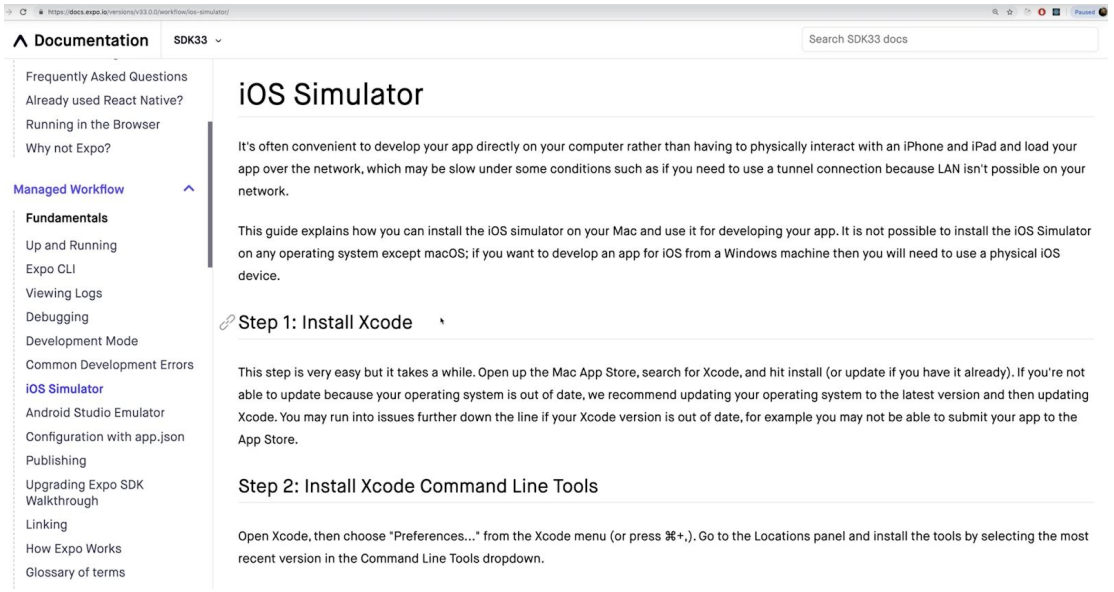

3:30
Building JavaScript bundle... 86.08%


3:30
Open up App.js to start working on your app!

3. You click **a** or **i** in the command line and automatically it can run it into a simulator.

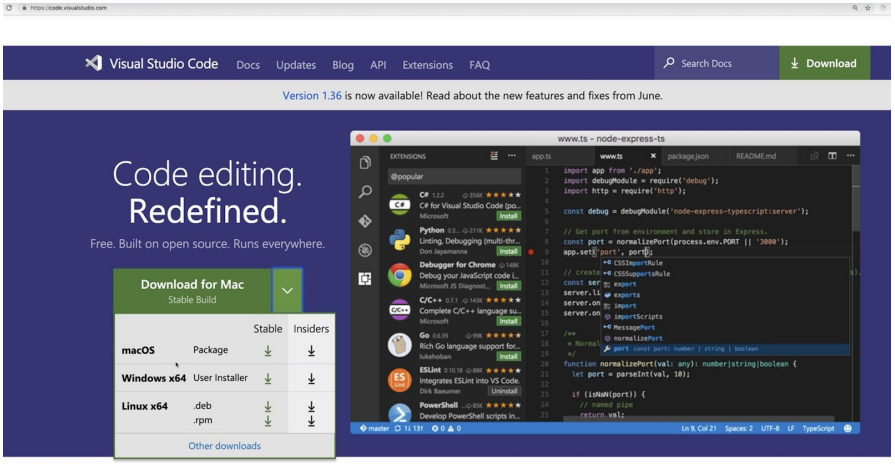
Why? It's often convenient to develop your app directly on your computer rather than having to physically interact with an iPhone or Android and load your app over the network, which may be slow under some conditions such as if you need to use a tunnel connection because LAN isn't possible on your network. For this reason you can download an Android Studio Emulator or an iOS Simulator:



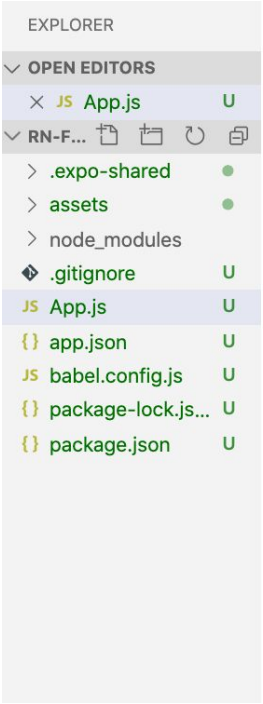


7. First Basic App

I use Visual Studio as a source-code editor. If you want to use VS too, you need to download it.



The structure of the folder is like below. It includes an **App.js**, which is the main js file that builds the static single-page app. This means that it keeps all page navigation within the session of the webpage.




```
import React from 'react';
import { Text, View, StyleSheet } from 'react-native';

export default function App() {
 return (
   <View style={styles.container}>
     <Text>
       Open up App.js to start working on your App!
     </Text>
   </View>
 );
}

const styles = StyleSheet.create({
 container: {
   flex: 1,
   backgroundColor: '#fff',
   alignItems:'center',
   justifyContent:'center',
 },
});
```

Just to be super clear: Styling is done via JavaScript here. React Native just offers CSS-like property names.

The **node_modules** folder contains all packages that are necessary for the app to work. Whenever you download a new package, a new folder inside node_module will be created with the name of the package and to load this package into the main module, you need to import or require("packagename").

Assets contain all images or static data you need for the application, while **expo-shared** is an automatic folder created by the expo init command.

A **gitignore** file specifies intentionally untracked files that Git should ignore.

.gitignore

```
node_modules/**/*
.expo/*
npm-debug.*
*.jks
*.p8
*.p12
*.key
*.mobileprovision
*.orig.*
web-build/
web-report/

# macOS
.DS_Store
```

app.json > ...

```
{
  "expo": {
    "name": "First React Native Application ",
    "slug": "rn-first-app",
    "privacy": "public",
    "sdkVersion": "36.0.0",
    "platforms": [
      "ios",
      "android",
      "web"
    ],
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "splash": {
      "image": "./assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "updates": {
      "fallbackToCacheTimeout": 0
    },
    "assetBundlePatterns": [
      "**/*"
    ],
    "ios": {
      "supportsTablet": true
    }
  }
}
```

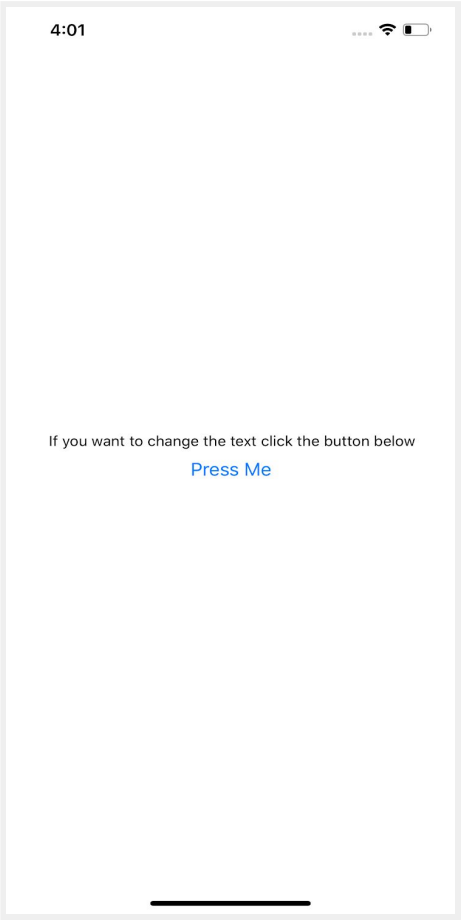
App.json is your go-to place for configuring parts of your app that don't belong in code. It is located at the root of your project next to your package.json. You can change properties like the name of your app, orientation, supported platforms, etc.

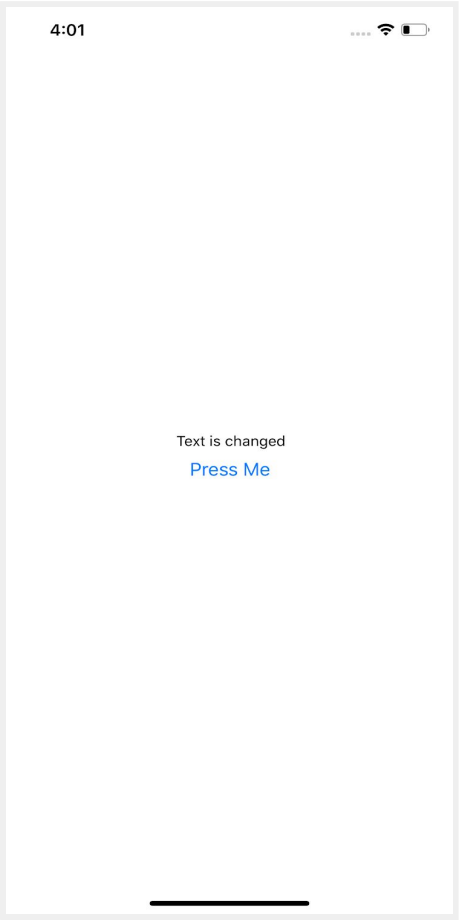
Package.json contains all the dependencies and the requirements for the app to run. They are automatically added when you install a new package.

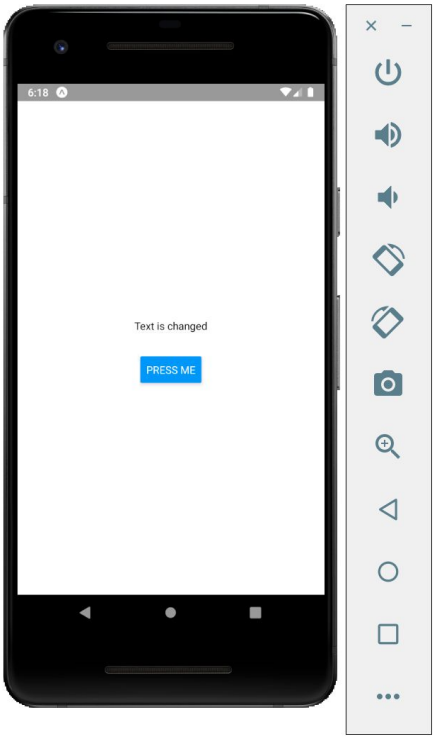
```
{} package-lock.json > ...
1  {
2    "requires": true,
3    "lockfileVersion": 1,
4    "dependencies": {
5      "@babel/code-frame": {
6        "version": "7.8.3",
7        "resolved": "https://registry.npmjs.org/@babel/code-frame/-/code-frame-7.8.3.tgz",
8        "integrity": "sha512-a9gxpmdXtZEInkCSHUJDLHZVBgb1QS0jhss4cPP93EW7s+uC5bikET2twEF3KV+7rDblJcmN
9        "requires": {
10         "@babel/highlight": "^7.8.3"
11       }
12     },
13     "@babel/compat-data": {
14       "version": "7.9.0",
15       "resolved": "https://registry.npmjs.org/@babel/compat-data/-/compat-data-7.9.0.tgz",
16       "integrity": "sha512-zeFQrr+284Ekvd9e7KAX954LkapWiOmQtsfHirhxqfdlX6MEC32iRE+pqUGlYIBchdevaCwv
17       "requires": {
18         "browserslist": "^4.9.1",
19         "invariant": "^2.2.4",
20         "semver": "^5.5.0"
21       }
22     },
23     "@babel/core": {
24       "version": "7.9.0",
25       "resolved": "https://registry.npmjs.org/@babel/core/-/core-7.9.0.tgz",
26       "integrity": "sha512-kWc7L0fw1xwvI0zi8OKVBuxRVefwGOrKSQMvrQ3dW+bIIavBY3/NpXmpjMy7bQnLgwgzWQZ8
27       "requires": {
```

Now let's change some code in App.js. The idea is to have a button and when you click it, the text above the button changes. We are going to use useState Hook and is exactly the same logic as in ReactJS.

```
import React, { useState } from 'react';
import { Text, View, StyleSheet, Button } from 'react-native';

export default function App() {
const [updatedText, setUpdatedText] = useState("If you want to change
the text click the button below");

 return (
   <View style={styles.container}>
     <Text> {updatedText}</Text>
     <Button title= "Press Me" onPress={()=> setUpdatedText("Text is
      changed")}>
     </Button>
   </View>
 );
}
```


4:01
If you want to change the text click the button below
Press Me

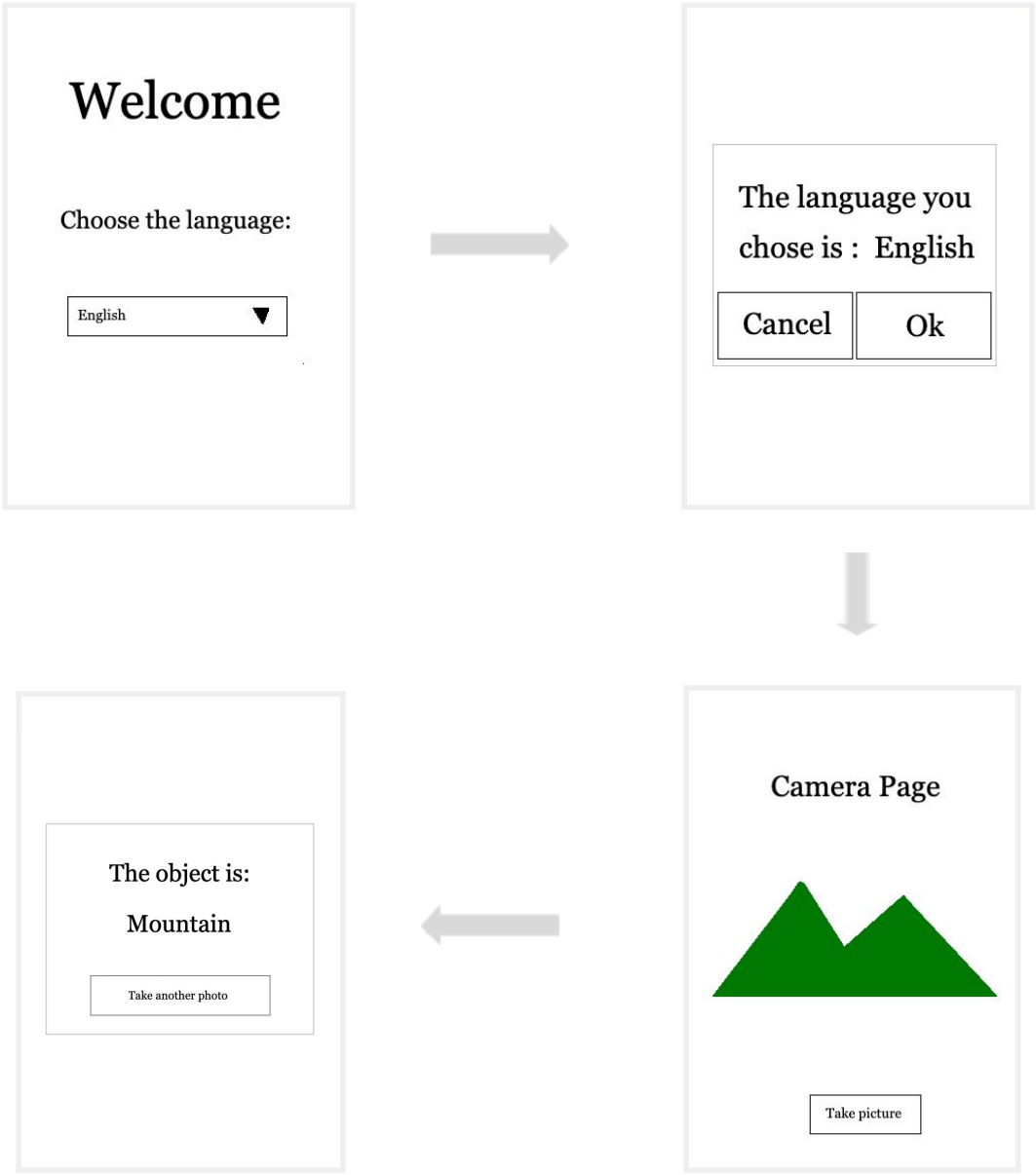

4:01
Text is changed
Press Me


6:18
Text is changed
PRESS ME

Welcome to Android Studio
Android Studio
Version 3.6.1
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Android Virtual Device Manager
Your Virtual Devices
Android Studio
Type
Name
Play Store
Resolution
API
Target
CPU/ABI
Size on ...
Actions
Pixe...
108...
28
Andr...
x86
8.9 ...
Create Virtual Device...

8. Design More Complex App

The idea is that we want to create an app that translates from pictures into a specific word in a language. We have a first page, with a welcome image and a dropdown with the chance to choose a language. I have decided some languages, but you can change them based on what the focus of your app is. After you choose the language a modal will pop up and it will redirect you to the camera view. Then when you will click the capture button, a picture will be captured and a modal will pop up showing what this picture is in the language you determined before.



9. Include Camera

expo-camera provides a React component that renders a preview for the device's front or back camera. The camera's parameters like zoom, auto focus, white balance and flash mode are adjustable. With the use of Camera, one can also take photos and record videos that are then saved to the app's cache.

Run the commands:

expo install expo-camera
expo install expo-permissions

```
belasinoimeri@U-CM-L112 rn-first-app % expo install expo-camera

   There is a new version of expo-cli available (3.15.5).
   You are currently using expo-cli 3.13.2
   Install expo-cli globally using the package manager of your choice; fo

Installing 1 SDK 36.0.0 compatible native module using npm.
> npm install
added 1 package from 1 contributor and audited 75014 packages in 4.045s

9 packages are looking for funding
  run `npm fund` for details

found 4 low severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

```
belasinoimeri@U-CM-L112 rn-first-app % expo install expo-permissions

   There is a new version of expo-cli available (3.15.5).
   You are currently using expo-cli 3.13.2
   Install expo-cli globally using the package manager of your choice;

Installing 1 SDK 36.0.0 compatible native module using npm.
> npm install
audited 75015 packages in 3.664s

9 packages are looking for funding
  run `npm fund` for details

found 4 low severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

10. Design Camera UI

Now we are going to make our camera page look interesting by adding icons that can be useful for the user. I am going to create a simple UI with some components I think are necessary, but you can change it based

on what is the focus of your app. I will create 6 main components while dividing the screen into 3 parts:

1. The Camera View
2. Above Camera View
3. Below Camera View

The interaction buttons divided into the 2nd and 3rd part will include:

- a button for taking a picture
- a button for switching from Back view to Front view
- a button to bring up the camera roll to check the gallery
- a button to go back to the previous page
- a button to switch flash mode on and off

To access a camera in React Native you need to import the expo-camera SDK and camera permissions from expo-permissions, before we can display the camera, or interact with it in any way.

```
import { Camera } from 'expo-camera';
import * as Permissions from 'expo-permissions';
```

Whenever you use an app that includes a camera, the app asks you for permission the first time you use it. This will be the same for our app, so the app must ask with a text message about the permission for camera.

In order to not show the user permission every time the user opens the app, but only the first time he uses it we need to create a state called hasPermission and declare it null in the beginning. In addition to the user's permission in the state, let's add a property to control the type of camera that's being used (Front or Back) and also a property to control the type of flash mode that's being used (on and off).

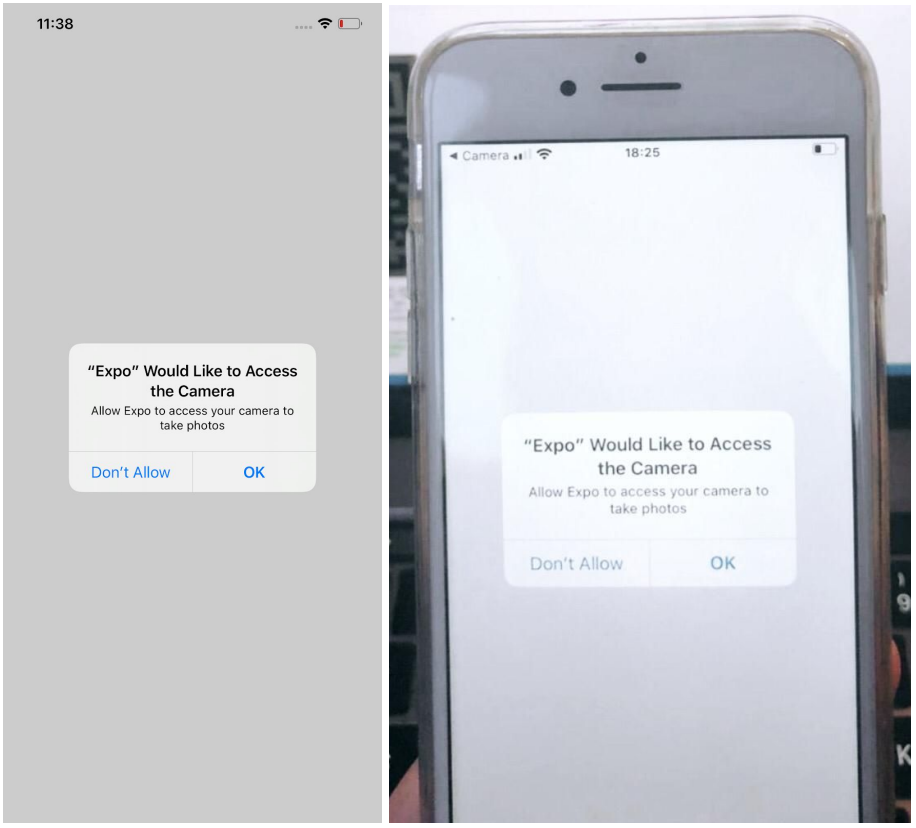
```
state = {
        hasPermission: null,
        cameraType: Camera.Constants.Type.back,
        flashmode: Camera.Constants.FlashMode.off
      }
```

We also need to update the state in an async function. If you are using class components the best place to ask for user permissions is inside the

componentDidMount lifecycle method, otherwise you need to use the useEffect hook.

```
async componentDidMount() {
const { status } = await Permissions.askAsync(Permissions.CAMERA);
  this.setState({ hasPermission: status === 'granted' });
}
```

This is the message the user will see, once he lands on the Camera screen:



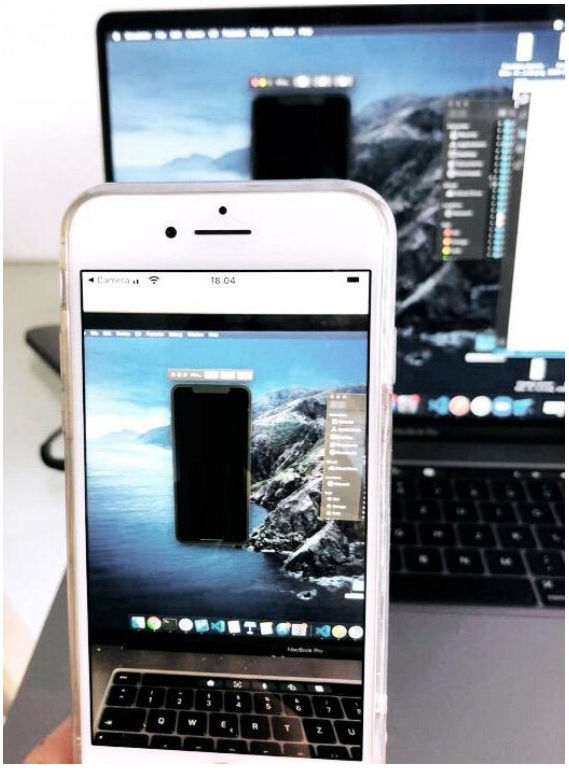
Let's head to the render method and add the Camera display logic:

```
render() {
   const { hasPermission } = this.state
   const { flashMode } = this.state
   const { cameraType } = this.state
   if (hasPermission === null) {
     return <View />;
   } else if (hasPermission === false) {
     return <Text>No access to camera</Text>;
   } else {
```

```
return (

  <View style={{ flex: 1 }}>
     <Camera style={{flex:1}} type={cameraType}
      flashMode={flashMode}>
     </Camera>
  </View>
        );
      }
    }
```

If the app has the permission, the Camera View will be shown, if not a text message asking for permission. While the property is null it will display an empty view and this usually happens at the start of the app while the componentDidMount is not yet called.

The **Camera View Takes the full screen** and this is what the app looks like so far. I am not displaying the iOS simulator because it will be just a black image.



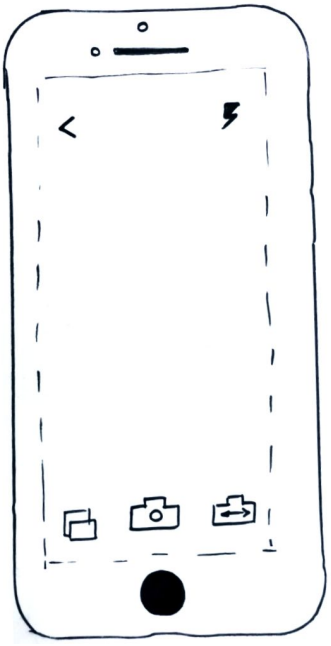
11. Add Camera UI icons

The Camera UI Interaction buttons will be using:

1. a TouchableOpacity component: a wrapper for making views respond properly to touches.

2. an Expo Icons component : a library of icons included in the expo package, where you can include all types of libraries like Font Awesome, Ionicons, MaterialCommunityIcons etc.

The app will have 5 buttons each with a different Icon representing an action. They will be divided into 2 Views, like in the low-fidelity design below:



1. The Above Camera View will contain 2 buttons:
   - Going back - using FontAwesome
   - Switching flash mode on and off - using Ionicons

2. The Below Camera View will contain 3 buttons:
   1. Displaying gallery - using Ionicons
   2. Taking a photo - using FontAwesome
   3. Switching back and front - using MaterialCommunityIcons

To use any Icon, all you need is to match its name and its type and add it to the Touchable component. First import the Type of Icons you want to use, in our case we are going to import the libraries below:

```
import {FontAwesome, Ionicons, MaterialCommunityIcons } from
'@expo/vector-icons';
```

And use it like this:

```
<FontAwesome name="camera" />
```

You can add styles to it, and transform it as you like, you will most likely need to either change its color and its size using these 2 properties. The functionalities will be included later, for now we are just doing the ui.

```
<FontAwesome name="camera" style={{ color: "#fff", fontSize: 40 }} />
```

The views will be added after the camera component in the render.

Above Camera View:

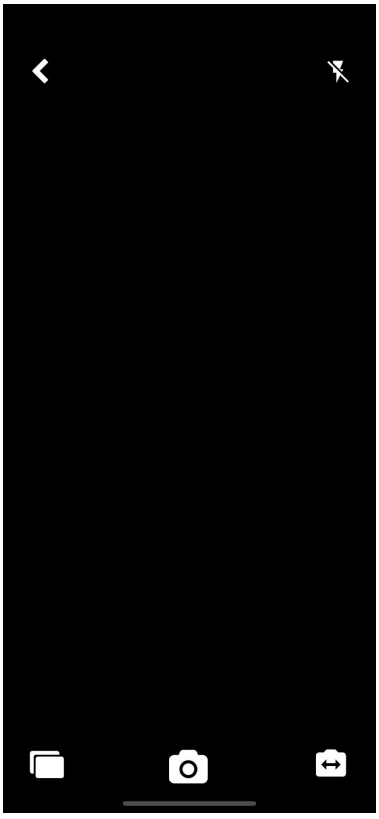
```
<View style={{flex: 1, flexDirection: "row", justifyContent:
"space-between", margin: 30, backgroundColor: 'transparent'}}>

                            // The Going Back icon
    <TouchableOpacity
              style={{
                marginTop: 30,
                alignItems: 'center',
                backgroundColor: 'transparent',
              }}>
              <FontAwesome
                name="chevron-left"
                color="white"
                size={30}
              />
     </TouchableOpacity>

                           // The Flash icon
     <TouchableOpacity
              style={{
                marginTop: 30,
                alignItems: 'center',
                backgroundColor: 'transparent',
              }}>
              <Ionicons
               name= "md-flash"
               color="white"
```

```
                size={30}
               />
      </TouchableOpacity>
   </View>
```

Below Camera View:

```
<View style={{flex: 1, flexDirection: "row", justifyContent:
"space-between", margin: 30, backgroundColor: 'transparent'}}>

                      // The Display Gallery icon
      <TouchableOpacity
               style={{
                 alignSelf: 'flex-end',
                 alignItems: 'center',
                 backgroundColor: 'transparent'
               }}>
               <Ionicons
                 name="ios-photos"
                 style={{ color: "#fff", fontSize: 40 }}
               />
       </TouchableOpacity>

                        // The Take Picture icon
       <TouchableOpacity style={{
               alignSelf: 'flex-end',
               alignItems: 'center',
               backgroundColor: 'transparent',
             }}>
               <FontAwesome
                 name="camera"
                 style={{ color: "#fff", fontSize: 40 }}
               />
       </TouchableOpacity>

                     // The Switch camera type icon
       <TouchableOpacity
               style={{
                 alignSelf: 'flex-end',
                 alignItems: 'center',
                 backgroundColor: 'transparent',
               }}>
               <MaterialCommunityIcons
                 name="camera-switch"
                 style={{ color: "#fff", fontSize: 40 }}
               />
        </TouchableOpacity>
 </View>
```

Now this is what the app looks like, in the iOS simulator and in my phone:



12. Camera interactions

In the section below we are going to show one by one how to make the functionalities of icons work.

11.1 Change Expo Camera Type

Switching between camera Type is straight forward. If you remember we have already set up a cameraType state. All you need to do is handle the type prop in the camera component. To do this we need to create a function that checks whether the camera View is using the back camera, if so it changes it into the front one and the other way around.

```
handleCameraType = () => {
  const { cameraType } = this.state

  this.setState({
    cameraType:
      cameraType === Camera.Constants.Type.back
        ? Camera.Constants.Type.front
        : Camera.Constants.Type.back
  })
```

```
 }
```

Now add this function to the onPress props on the TouchableOpacity component and the functionality works.

```
onPress={() => this.handleCameraType()
```

11.2 Take Pictures

To take a picture, the Expo Camera already has a function called takePictureAsync. First you need to expose a ref to the camera component by just adding the following prop to the camera in the render:

```
ref={ref => { this.camera = ref }}
```

Next, create a function to make use of the camera ref and it's takePictureAsync function. We also need to declare some arguments and to understand what this function returns for later use.

```
takePicture = async () => {
   if (this.camera) {
   let photo = await this.camera.takePictureAsync();
 }
 }
```

Argument: **options**

- quality (*number*) -- Specify the quality of compression, from 0 to 1. 0 means compress for small size, 1 means compress for maximum quality.
- base64 (*boolean*) -- Whether to also include the image data in Base64 format.
- exif (*boolean*) -- Whether to also include the EXIF data for the image.

```
takePicture = async () => {

   const options = { quality: 0.5, base64: true };
   if (this.camera) {
   let photo = await this.camera.takePictureAsync(options);
 }
 }
```

Returns: **a Promise that resolves to an object** { uri, width, height, exif, base64 }

- uri is a URI to the local image file
- width, height specify the dimensions of the image.
- base64 is included if the base64 option is set true, and is a string containing the JPEG data of the image in Base64
- exif is included if the exif option is set true, and is an object containing EXIF data for the image

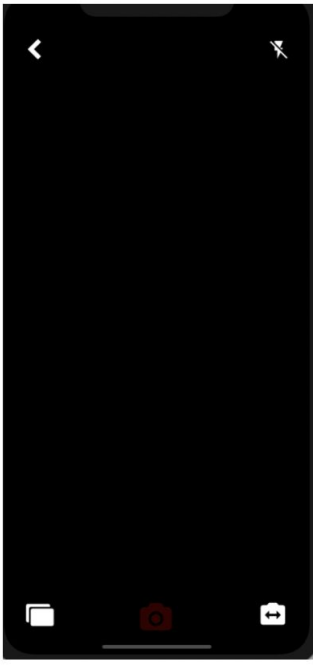
! The local image URI is temporary. Use FileSystem.copyAsync to make a permanent copy of the image.

We also want to add another interaction into this step just to make it a little bit more fancy. When the user is holding the icon it should turn red, just to make it more obvious that it is taking a photo. To do this we need a state named capturing set to false in the beginning.

```
state = {
    ...
  capturing: false,
}
```

We should update this state in two other functions connecting to this logic, at the takePicture we set the state to true because it is recording a photo and at the handleCaptureOut we set the state to false, because we stop recording. Later on we call both of them with onPressIn and Out at the render.

```
takePicture = async () => {
…
   this.setState({
     capturing: true
   });
}

 handleCaptureOut = () => {
   this.camera.stopRecording();
   this.setState({
     capturing: false
   });
 };

onPressIn={() => this.takePicture()}
onPressOut={() => this.handleCaptureOut()}
```



11.3 Displaying Gallery

To access the photos we have taken using the camera, we can use the Expo ImagePicker Component. This component is used for selecting Images without the need to write code from scratch and it's pretty handy and easy to use. First we need to install it:

expo install expo-image-picker

```
belasinoimeri@U-CM-L112 rn-first-app % expo install expo-image-picker

   There is a new version of expo-cli available (3.15.5).
   You are currently using expo-cli 3.13.2
   Install expo-cli globally using the package manager of your choice; for example:

Installing 1 SDK 36.0.0 compatible native module using npm.
> npm install
added 1 package from 1 contributor and audited 75017 packages in 4.515s

9 packages are looking for funding
  run `npm fund` for details

found 4 low severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

Then we need to import it at camera.js:

```
import * as ImagePicker from 'expo-image-picker';
```

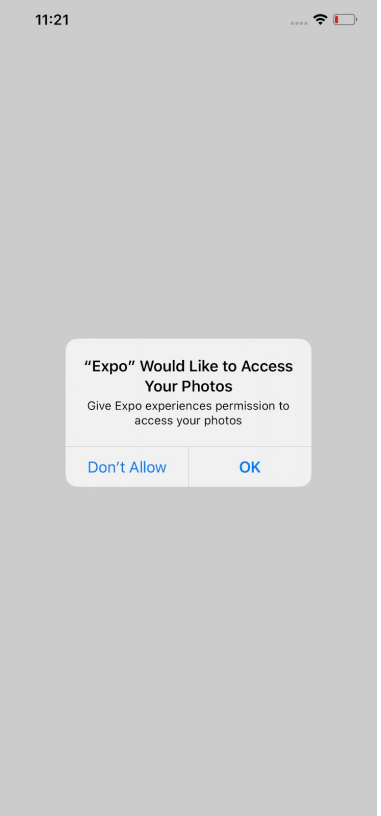
Again, we will need permissions to access the camera roll of the device, but this time only for IOS devices and not for Android. The message that will be like in the take picture permission above, only this time asking for permission to access the photos.

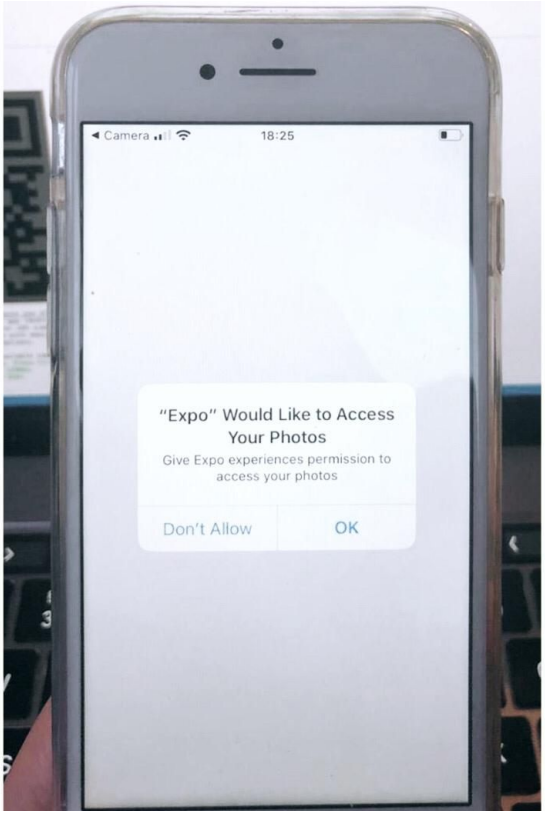
To make the code clean and easier to read, let's make a function for permissions and include both of them, camera and camera roll permissions at once. We call this function into the componentDidMount Method.

```
async componentDidMount() {
   this.getPermissionAsync()
 }

 getPermissionAsync = async () => {
   // Camera roll Permission
   if (Platform.OS === 'ios') {
const { status } = await Permissions.askAsync(Permissions.CAMERA_ROLL);
     if (status !== 'granted') {
    alert('Sorry, we need camera roll permissions to make this work!');
```

```
    }
  }
  // Camera Permission
  const { status } = await Permissions.askAsync(Permissions.CAMERA);
  this.setState({ hasPermission: status === 'granted' });
}
```

Now the app will show also another permission for accessing photos, in the iOS simulator and in my phone:



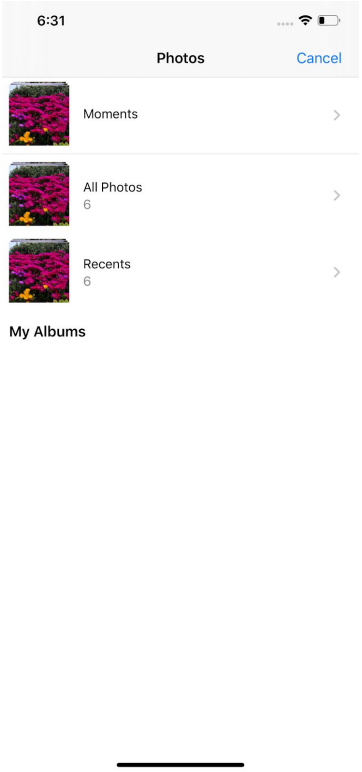


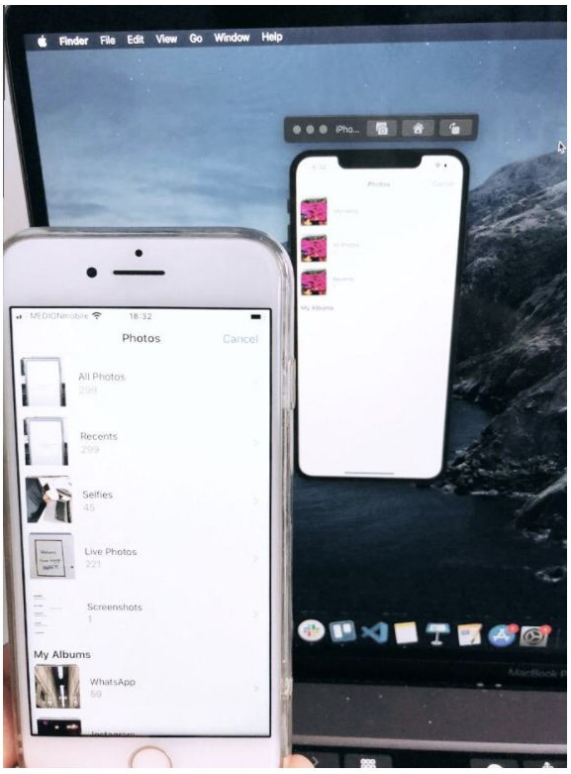
Now, let's write a function to handle the camera roll image list:

```
pickImage = async () => {
  let result = await ImagePicker.launchImageLibraryAsync({
    base64: true,
    mediaTypes: ImagePicker.MediaTypeOptions.Images
  })
  }
```

Once the user invokes this function, the screen will display all the saved photos in the gallery and the user can check and select one of the photos already taken before.

```
onPress={() => this.pickImage()}
```





11.4 Switching Flash Mode

We are going to do the same logic as with the cameraType, we are going to create a function to check if the flash is on and turn it off and the other way around.

```
handleFlashMode = () => {
  const { flashMode } = this.state

  this.setState({
    flashMode:
      flashMode === Camera.Constants.FlashMode.off
        ? Camera.Constants.FlashMode.on
        : Camera.Constants.FlashMode.off
  })
}

onPress={() => this.handleFlashMode()}
```

We also need to change the icon and to do so we just need to change the name of the icon.

```
name={flashMode === Camera.Constants.FlashMode.on ? "md-flash" : 'md-flash-off'}
```

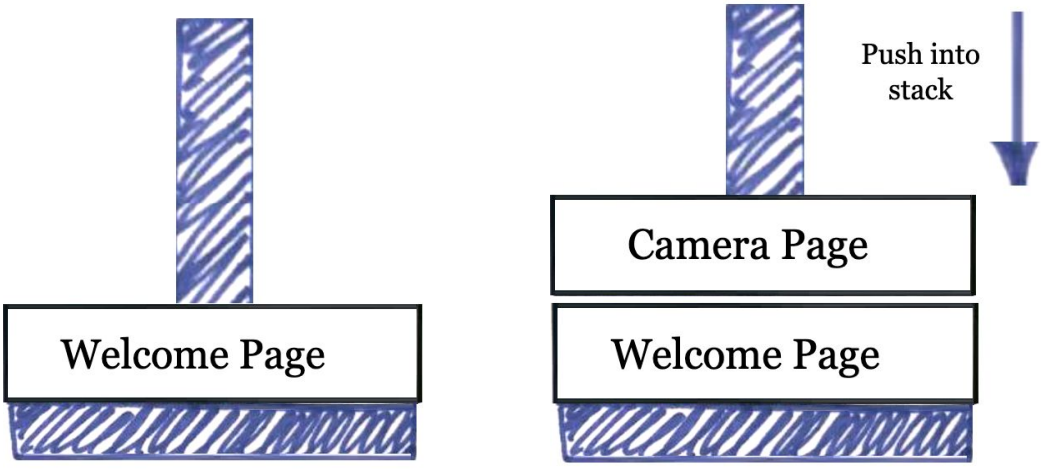
11.5 Going Back

We are going to implement the functionality for this button **later after the stack navigator explanation**, but just to give you an idea, it will go back into the previous page by just adding one line of code:

```
onPress={() => this.props.navigation.navigate("Welcome")}
```

13. Stack Navigator

Provides a way for your app to transition between screens where each new screen is placed on top of a stack.
By default the stack navigator is configured to have the familiar iOS and Android look & feel: new screens slide in from the right on iOS, fade in from the bottom on Android.



To install this package you need to follow the documentation and add these three commands in the terminal:

npm install @react-navigation/native

```
belasinoimeri@U-CM-L112 rn-first-app % npm install @react-navigation/native

npm WARN @react-native-community/masked-view@0.1.5 requires a peer of react-nati
ve@^0.57 but none is installed. You must install peer dependencies yourself.

+ @react-navigation/native@5.1.4
added 1 package, removed 2 packages, updated 3 packages, moved 1 package and aud
ited 74135 packages in 9.253s

9 packages are looking for funding
  run `npm fund` for details

found 4 low severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

npm install @react-navigation/stack

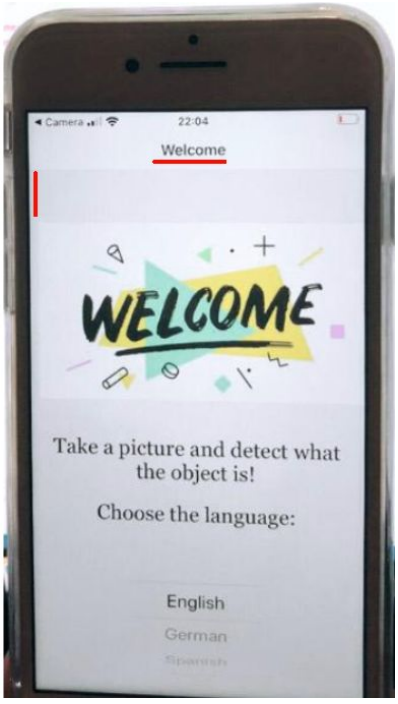
```
belasinoimeri@U-CM-L112 rn-first-app % npm install @react-navigation/stack
npm WARN @react-native-community/masked-view@0.1.5 requires a peer of react-nati
ve@^0.57 but none is installed. You must install peer dependencies yourself.

+ @react-navigation/stack@5.2.9
updated 1 package and audited 74135 packages in 4.917s

10 packages are looking for funding
  run `npm fund` for details

found 4 low severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

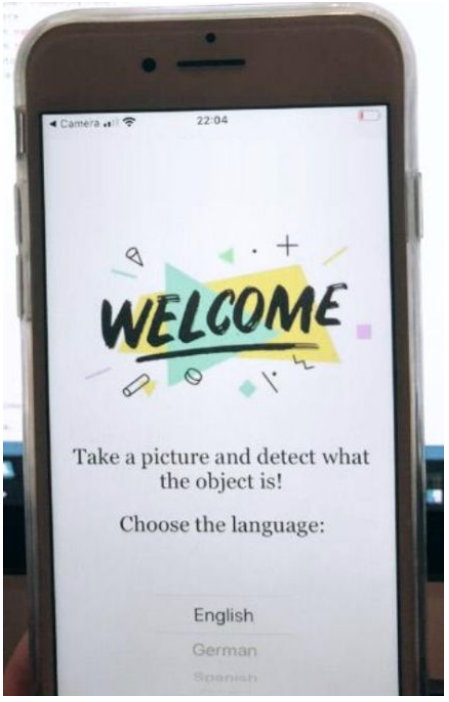
expo install react-native-gesture-handler react-native-reanimated react-native-screens react-native-safe-area-context @react-native-community/masked-view

```
belasinoimeri@U-CM-L112 rn-first-app % expo install react-native-gesture-handler reac
t-native-reanimated react-native-screens react-native-safe-area-context @react-native
-community/masked-view
┌──────────────
```

```
    background: 'rgb(255, 255, 255)',
  },
};

function App() {
  return (
    <NavigationContainer theme={MyTheme} >
      <Stack.Navigator initialRouteName="Welcome" screenOptions={{
        headerShown: false
      }}>
        <Stack.Screen name="Welcome" component={WelcomePage} />
         <Stack.Screen name="CameraComponent" component={CameraComponent}
/>
      </Stack.Navigator>
    </NavigationContainer>
  );
}
export default App;
```

We can also decide on the **theme** and **headerShown** to make our style a little bit cleaner like below:





14. Welcome Page

We first need to import everything we need for the app like Stylesheet, View, Tex, Image, Alert and last but not least Picker, which will be used for the dropdown part.

```
import React, { Component } from 'react';
import { StyleSheet, View, Text, Image, Picker, Alert } from
'react-native';
```

Than we need to structure the code into views based on the UI concept, we add an image, some text and the picker:

```
 render() {
   return (
     <View style={styles.container}>
       <View>
         <Image
           source={require("./assets/welcome.jpg")}
           style={styles.image}>
         </Image>

         <Text style={styles.text}>Take a picture and detect what the
          object is!</Text>
         <Text style={styles.text}>Choose the language:</Text>

         <View style={styles.picker}>
           <Picker
             selectedValue={this.state.selectedValue}
             style={{ height: 50, width: 150 }}
             onValueChange={(itemValue) => this.label(itemValue)}
           >
             <Picker.Item label="English" value="en" />
             <Picker.Item label="German" value="de" />
             <Picker.Item label="Spanish" value="es" />
             <Picker.Item label="French" value="fr" />
             <Picker.Item label="Italian" value="it" />
             <Picker.Item label="Albanian" value="sq" />
           </Picker>
         </View>

       </View>
     </View>
   )
 }
}
```

We create a function called label that takes itemValue as an argument and sets the state of selected value from English (the pre default state) to the one we select in the dropdown. We create an itemLabel for every itemValue and so in the moment we select the language an alert will return a pop up saying: *The language you chose is: itemLabel*

```
 state = {
   selectedValue: "English",
   itemLabel: null,
 }
```

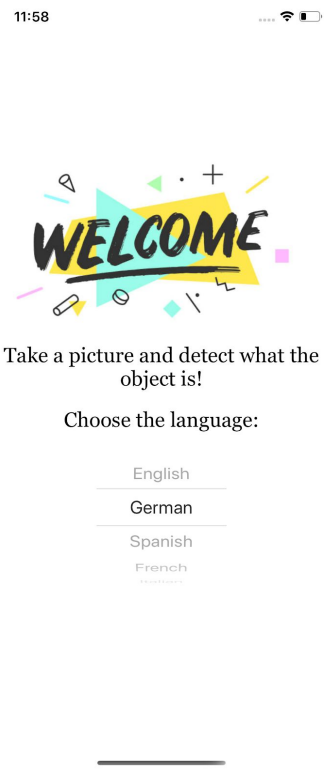
```
 label = (itemValue) => {
   this.setState({
     selectedValue: itemValue
   })

   if (itemValue===="en") {itemLabel= "English"}
   else if (itemValue===="es") {itemLabel= "Spanish"}
   else if (itemValue===="de") {itemLabel= "German"}
   else if (itemValue===="fr") {itemLabel= "French"}
   else if (itemValue===="it") {itemLabel= "Italian"}
   else if (itemValue===="sq") {itemLabel= "Albanian"}

   Alert.alert(
     'The language you chose is: ' + itemLabel,
     '',
     [
 { text: 'Cancel', onPress: () => console.log('Cancel Pressed!') },
 { text: 'Continue', onPress: () =>
{this.props.navigation.navigate('CameraComponent',{selectedValue:itemVa
lue})}},
     ],
     { cancelable: false }
   )
 }
```

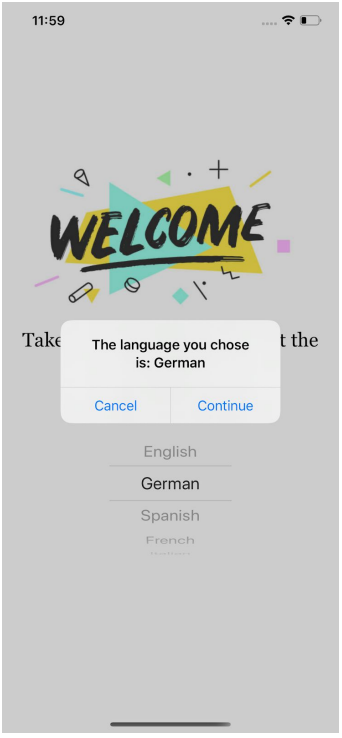
Below I attached the stylesheet I used for this page. It is very basic style, clean and very simple, all based on my personal taste, but feel free to change things on it.

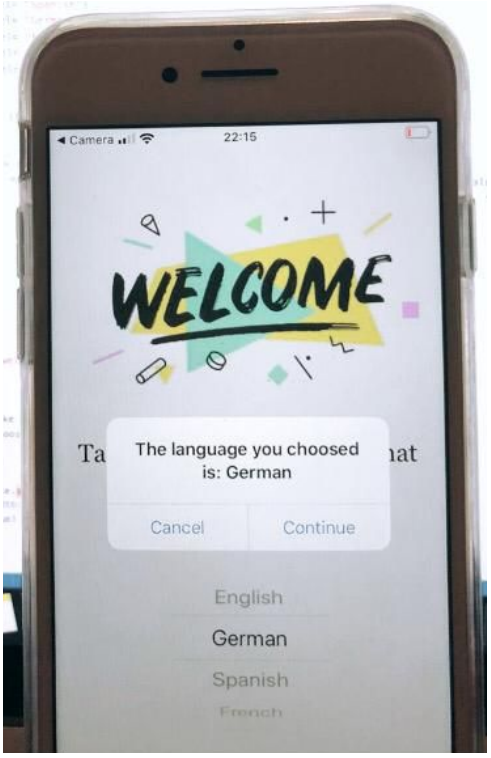
```
const styles = StyleSheet.create({
container: {
   flex: 1,
   margin: 10,
 },
 text: {
   color: "#000",
   fontSize: 24,
   textAlign: "center",
   fontFamily: "Georgia",
   top: 370,
   marginTop: 20
 },
 image: {
   top: 170,
   width: 370,
   height: 200,
   position: "absolute"
 },
 picker: {
   top: 350,
   alignItems: "center",
   fontFamily: "Georgia"
 }
```

```
});
```

This is how our app look like, in the simulator and in my phone:







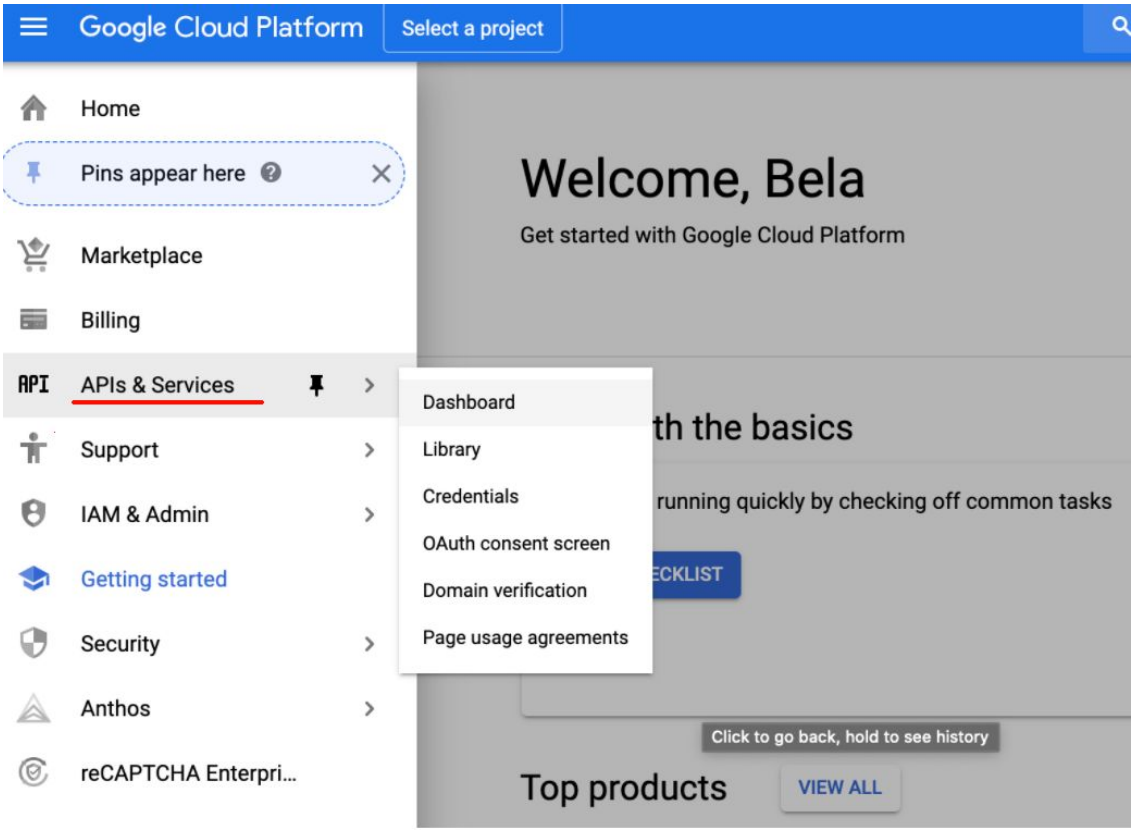
15. Google Vision API

Cloud Vision API allows developers to easily integrate vision detection features including image labeling, face, and landmark detection, optical character recognition (OCR), and tagging of explicit content, within applications.

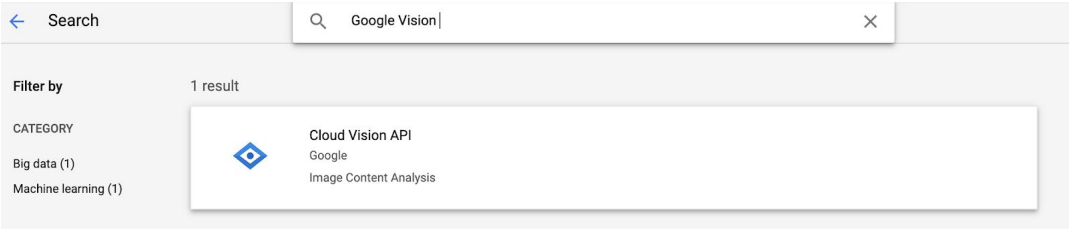
Google Cloud's Vision API offers robust pre-trained machine learning models through REST and RPC (remote procedure call) APIs. It assigns labels to images and quickly classifies them into millions of predefined classes or categories. For implementing Google Vision in your app follow the steps below:

1. Select or Create a Google Cloud Platform Project.
   Please make sure that billing is enabled for your Google Cloud Platform project. You will not be able to get any response from API until you enable the billing. ! For your DPS project everything is already setted up.

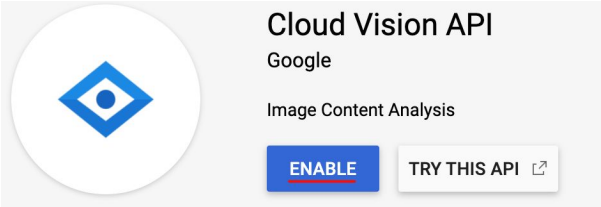
2. Click on "APIs and Services"



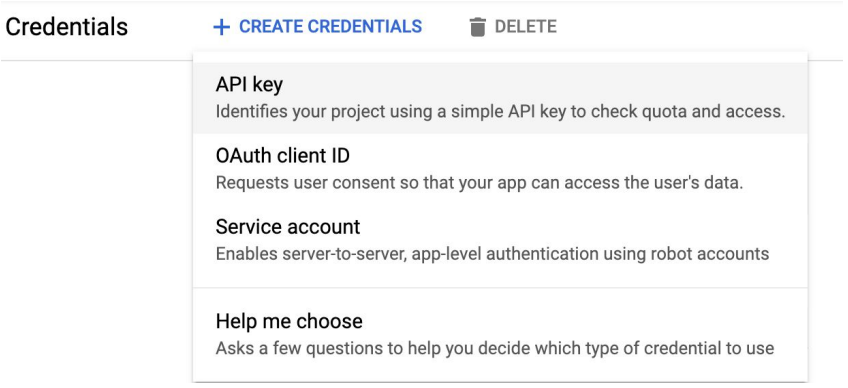
3. Search for Cloud Vision API.

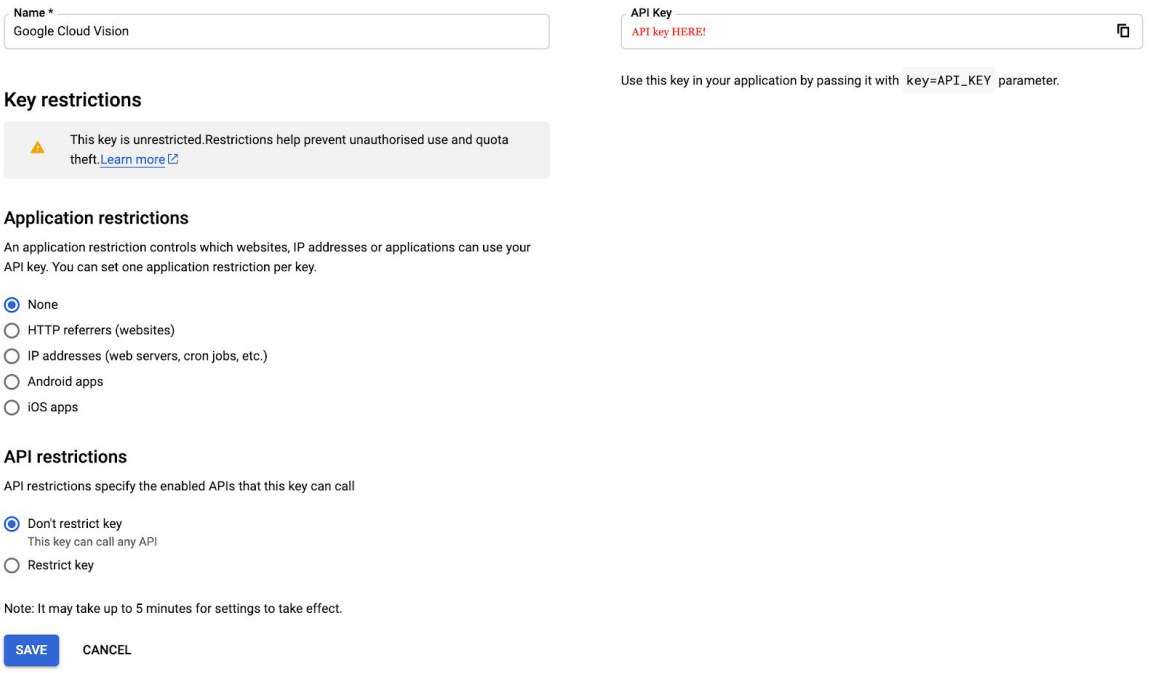


4. Click on the enable button.



5. To use this API, click on Create Credentials - API key





Congrats your API key has been generated. ! Please keep your key secret. Do not push it on Github. Save the key into a config.json file inside your folder like this:

```
{
   "googleCloud": {
     "api": "https://vision.googleapis.com/v1/images:annotate?key=",
     "apiKey": "..."
   }
 }
```

Let us code now. First we need to understand how to make a post request based on this Google API json file, like it is explained in this link and in this official documentation.

```
handlePress = async (base64) => {
  let googleVisionRes = await fetch(config.googleCloud.api +
  config.googleCloud.apiKey, {
    method: 'POST',
    body: JSON.stringify({
      "requests": [
        {
          "image": {
            "content": base64
          },
          "features": [
            {
              "maxResults": 10,
              "type": "OBJECT_LOCALIZATION"
            }
          ],
        }
      ]
    })
  });

await googleVisionRes.json().then(googleVisionRes => {

console.log(googleVisionRes.responses[0].localizedObjectAnnotations[0].
name)

return googleVisionRes.responses[0].localizedObjectAnnotations[0];})
  .catch((error) => { console.log(error) })
 };
```

Now we need to call this function into two other functions, so when we take a picture or we choose a picture from the gallery we can detect with google vision api what this object is.

```
takePicture = async () => {
…
   this.handlePress(photo.base64).then(googleVisionRes => {
     if (googleVisionRes) {
       console.log(googleVisionRes)
     }
   })
 }
 pickImage = async () => {
 …
   this.handlePress(result.base64).then(googleVisionRes => {
     if (googleVisionRes) {
       console.log(googleVisionRes);
       return googleVisionRes;
     }
   });
 }
```

16. Google Translate API

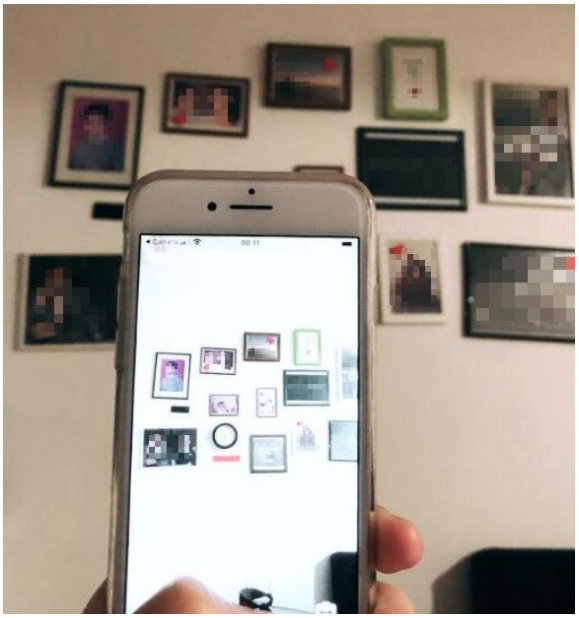
Cloud Translation can dynamically translate text between thousands of language pairs. We create a key in the same way we did with Google Vision API. You can follow the official documentation about the json files and also how to implement this API.

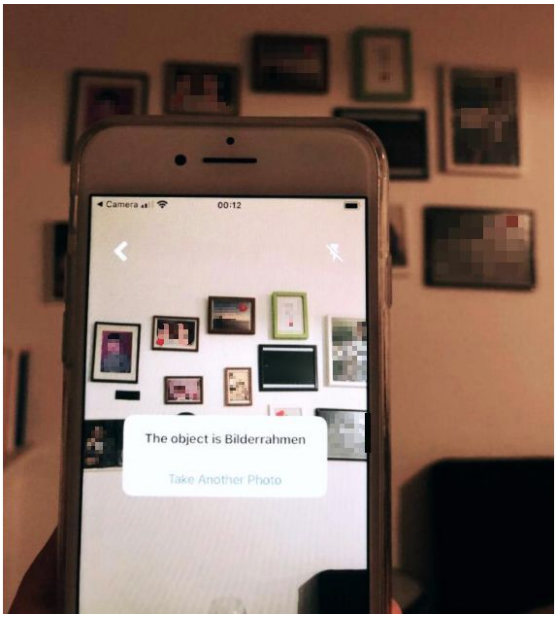
```
 translate = async (term) => {
const { selectedValue } = this.props.route.params
var url = "https://translation.googleapis.com/language/translate/v2";
   //Strings requiring translation
   url += "?q=" + term;
   //Target language
   url += "&target=" + selectedValue;
   //Replace with your API key
   url += "&key=...";

let googleTranslateRes = await fetch(url);
googleTranslateRes.json().then(googleTranslateRes => {

console.log(googleTranslateRes.data.translations[0].translatedText);
Alert.alert('The object is ' +
googleTranslateRes.data.translations[0].translatedText,
         '',
         [
{ text:'Take Another Photo', onPress: () => console.log('Next Photo')},
         ],
         { cancelable: false }
       )
       return googleTranslateRes;
     }).catch((error) => { console.log(error) })

 }
```

We should also call this function inside handlePress:

```
this.translate(googleVisionRes.responses[0].localizedObjectAnnotations[
0].name)
```

After selecting German, going to the next page and taking a picture an alert will pop up returning the name of that object in German. In the example below if we console.log all the steps, first we will get a photo in base64 format, then a "Picture Frame", then a "Bilderrahmen".





In the simulator we can not show anything because we can not use a camera feature and it will return: The object is undefined.

Github Link: https://github.com/bsinoimeri/ReactNativeTranslateDetect

17. Homework

- Understand the whole process of Mobile Development by following this link.
- Try to publish your app in AppStore or GooglePlay.