TACO Finance: Adding an Azure App Services Back-end

TACO Finance App Recipe Workshop Lesson 4

Contents

[Introduction 1](#_Toc472955706)

[Requirements 1](#_Toc472955707)

[Adding Online Capabilities to the Finance App 1](#_Toc472955708)

[Clone the Github Repository 1](#_Toc472955709)

[Azure Mobile App Service Configuration 2](#_Toc472955710)

[Add the Azure Mobile Apps Plugin to the Project 2](#_Toc472955711)

[Implement Providers 3](#_Toc472955712)

[Implement Settings 6](#_Toc472955713)

[Implement Login 11](#_Toc472955714)

Figures

[Figure 1 – Visual Studio: Adding the Azure Mobile Apps Plugin to the Project 2](#_Toc472955715)

[Figure 2 – TACO Finance App: Start (Login) Page 16](#_Toc472955716)

[Figure 3 – TACO Finance App: Settings Page 17](#_Toc472955717)

[Figure 4 – TACO Finance App: Start Page 18](#_Toc472955718)

# Introduction

The TACO Finance application you created in *Creating the TACO Finance App* lesson leverages the browser or mobile device’s localstorage capabilities to store Client data. You learned in that lesson that you could change a configuration setting to use Cordova’s SecureStorage capabilities instead, but regardless, all you have so far is a stand-alone solution. In this lesson, you’ll add online capabilities to the application; using central storage and enabling users to access the same data across multiple systems.

The app’s online capabilities are provided through the [Azure Mobile App Service](https://azure.microsoft.com/en-us/services/app-service/mobile/). You’ll eventually use that service to deliver offline capabilities for the app as well (in a subsequent lesson).

# Requirements

To complete this lesion, you must install the following software components:

* Visual Studio 2017 – Download the IDE at <http://go.microsoft.com/fwlink/?LinkID=533794>.
* Visual Studio Tools for Apache Cordova (TACO) – Install TACO using the Visual Studio Installer; learn more at <https://taco.visualstudio.com/en-us/docs/vs-taco-2017-install>.
* Visual Studio Ionic 2 Templates.
* Ionic Template dependencies.
* [Google Chrome browser](https://www.google.com/chrome/browser/desktop/index.html) (used by Visual Studio’s Simulate in Browser option).

Additionally, you need to have completed Lesson 2: *Creating the TACO Finance Application* lesson.

# Adding Online Capabilities to the Finance App

## Clone the Github Repository

Several of the code listings for this application are quite long, so rather than make you copy large blocks of code from this document, you’ll start by cloning the project’s Github repository to your local system to streamline this exercise. You should already have a clone of the repository from the previous lesson, but if you don’t, open a terminal window and navigate to the folder where you want to store the code and execute the following command:

git clone https://github.com/Microsoft/cordova-app-recipe-finance

That’s it, you’ll have the complete code inside a Visual Studio solution if you want to bypass this lesson and just run and analyze the finished code directly.

## Azure Mobile App Service Configuration

The online version of the app requires access to an online data store the application can use to store its data. For this application, we implemented a cloud-based storage and authentication environment using the Azure mobile App Service. The data tables that the application needs have already been created, and the integration with Microsoft Active Directory has been set up. To use these capabilities, you’ll simply need to configure the application to access the implementation of the service.

1. Open Visual Studio, then open the TACO Finance app solution you created in a previous lesson.
2. Open the src\providers\config.ts file. Inside the file, look for the authEndpoint variable declaration:

//change this endpoint for your Azure project

readonly authEndpoint = '';

Populate this variable with the Azure Mobile app endpoint provided by the workshop instructor.

1. Press **Ctrl**+**S** to save your changes to the file.

The Config provider has code in its constructor that validates that this variable is populated. The user won’t see any warnings, but while debugging the application, you’ll see warnings in the console if this step isn’t complete.

## Add the Azure Mobile Apps Plugin to the Project

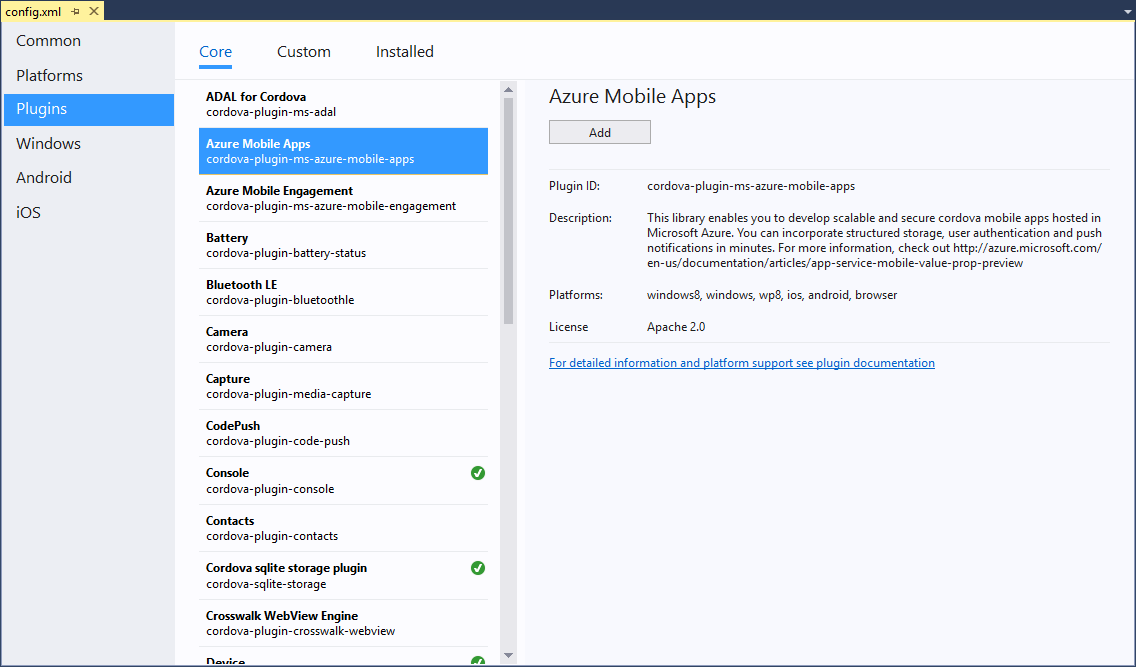
1. In Solution Explorer, double-the project’s config.xml file to open Visual Studio’s custom editor.
2. Switch to the plugins tab as shown in Figure 1 and search for the **Azure Mobile Apps** plugin as shown in the figure. 

Figure 1 – Visual Studio: Adding the Azure Mobile Apps Plugin to the Project

1. Click the **Add** button to add the plugin to your project.

## Implement Providers

You’re going to add two providers to the project. One is the ClientDataOnline provider that adds another data storage option to the application. The other is the UserData provider which handles all authentication-related tasks such as login and logout.

1. Add the UserData provider to the application using the Ionic CLI. Open a terminal window and navigate to the project’s Ionic/Cordova application folder (the one with the src and www folders) and execute the following command:

ionic g provider UserData

The CLI will create a new file in src\providers\user-data.ts.

1. You might as well create the other provider now as well, so add the ClientDataOnline provider to the application using the Ionic CLI. In the terminal window, execute the following command:

ionic g provider ClientDataOnline

1. With the two providers in place, you have to tell Ionic and the app about the providers. Open the project’s src\app\app.module.ts file, and add the following imports to the top of the file:

import { ClientDataOnline } from '../providers/client-data-online';

import { UserData } from '../providers/user-data';

Next, add the following lines to the providers array at the bottom of the file:

ClientDataOnline,

UserData,

When you’re done, the array should look like the following:

providers: [

ClientData,

**ClientDataOnline,**

ClientDataStorage,

Config,

Storage,

**UserData,**

{ provide: ErrorHandler, useClass: IonicErrorHandler }

]

Press **Ctrl**-**S** to save your changes to the file.

1. The UserData provider needs some code, so populate the src\providers\user-data.ts file with the contents of the same file from the cloned Github repository. Take a few moments to study the code.

The provider’s code implements the login and logout functions the application will use to manage the user’s authentication state within the application. These functions interact with the Microsoft.WindowsAzure.MobileServiceClient, an object exposed by the **Azure Mobile Apps** plugin your added to the project in step 6.

It also implements the isLoggedIn function which enables other parts of the application to easily determine whether the user is logged in to the Azure cloud.

Press **Ctrl**-**S** to save your changes to the file.

1. If you look at the provider’s code in Visual Studio, you may notice that Visual Studio is reporting some problems with the code. That’s because Visual Studio’s TypeScript compiler doesn’t understand the Azure Mobile Apps plugin.

To fix this, you must install the [TypeScript Declaration](https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html) file (typings file) for the plugin. The project’s Github project has the required typings file; copy src\typings\azureMobileServices.d.ts from the Github repository to the project’s typings folder (src\typings). With this in place, Visual Studio should recognize the plugin’s types and remove the errors.

1. Next, lets tackle the ClientDataOnline provider, populate the src\providers\client-data-online.ts file with the contents of the same file from the cloned Github repository. Take a few moments to study the file’s code.

The provider delivers create, read, update, and delete (CRUD) functions for the Client, Account, and Investment data just like the existing ClientDataLocalstorage provider the app uses today. The only difference is that it interacts with the Azure Mobile App service instead of the browser’s localstorage container.

1. With the new data provider in place, you need to replace the contents of the ClientData provider. If you remember from the previous lesson, we hacked this provider so it would force all data requests to use the localstorage data source. Now that we have another storage option available to the app, we need the ClientData provider to have the smarts it needs to switch in the correct provider as necessary.

Populate the src\providers\client-data-online.ts file with the contents of the same file from the cloned Github repository. Take a few moments to study the file’s code.

The provider doesn’t do much, it simply waits until the application container has completed initializing, then asks the Config provider for the storage location for the app’s data. Once it has that information, it makes a call to the setDataProvider function to swap in the selected provider as the source for all application data requests (leaving user data requests still processed by the UserData provider).

Since offline hasn’t been implemented yet (you’ll add it in a subsequent lesson), in the ClientData provider, comment out the following parameter to the constructor as shown in the following example:

//public offlineStore: ClientDataOffline,

When you’re done, the constructor should look like the following:

constructor(

private config: Config,

public events: Events,

public localStore: ClientDataStorage,

//public offlineStore: ClientDataOffline,

public onlineStore: ClientDataOnline,

public platform: Platform,

) {

In the setDataProvider switch statement, comment out the following lines:

//case 'offline':

// this.provider = this.offlineStore;

// this.provider.init();

// break;

Press **Ctrl**-**S** to save your changes to the file.

## Implement Settings

Now that the application has multiple options for where the application gets its data, lets implement the Settings form that users will use to make their selection.

1. Start by adding the Settings page to the application. In a terminal window pointing to the project’s Ionic/Cordova project folder (the one with src and www folders), execute the following command:

ionic g page Settings

This commands adds a new page in the project’s src\pages\settings\ folder.

1. With the page created, you have to tell the application about the new page. Open the project’s src\app\app.module.ts file and add the following import statement to the top of the file:

import { SettingsPage } from '../pages/settings/settings';

Next, add the following reference to the declarations and entryComponents arrays:

SettingsPage,

When you’re done, the arrays will look like the following:

declarations: [

AboutPage,

AccountDetail,

AccountForm,

ClientDetail,

ClientForm,

ClientList,

InvestmentDetail,

InvestmentForm,

MyApp,

**SettingsPage,**

],

entryComponents: [

AboutPage,

AccountDetail,

AccountForm,

ClientDetail,

ClientForm,

ClientList,

InvestmentDetail,

InvestmentForm,

MyApp,

**SettingsPage,**

],

Press **Ctrl**-**S** to save your changes to the file.

1. Now, lets setup the page. Open the project’s src\pages\settings\settings.html file, and replace the existing content with the following:

<ion-header>

<ion-navbar>

<ion-title>

{{config.appNameShort}}: Settings

</ion-title>

<ion-buttons start>

<button ion-button (click)="dismiss()">Cancel</button>

</ion-buttons>

<ion-buttons end>

<button ion-button (click)="save()">Done</button>

</ion-buttons>

</ion-navbar>

</ion-header>

<ion-content padding>

<ion-item>

Select a <strong>storage type</strong> for the application:

</ion-item>

<ion-list radio-group [(ngModel)]="storageType">

<ion-item>

<ion-label>Online</ion-label>

<ion-radio value="online" checked></ion-radio>

</ion-item>

<ion-item>

<ion-label>Offline</ion-label>

<ion-radio value="offline"></ion-radio>

</ion-item>

<ion-item>

<ion-label>Local Storage</ion-label>

<ion-radio value="localstorage"></ion-radio>

</ion-item>

<ion-item>

<ion-label>Secure Storage</ion-label>

<ion-radio value="securestorage"></ion-radio>

</ion-item>

</ion-list>

</ion-content>

This markup creates a simple input form, enabling users to select the storage location for the app’s data. The form’s header includes **Done** and **Cancel** buttons, and the form’s only input field is an Ionic **Radio Group** selector of storage locations.

Since we’re not supporting offline with this version of the application, go ahead and comment out the option for offline:

<!-- <ion-item>

<ion-label>Offline</ion-label>

<ion-radio value="offline"></ion-radio>

</ion-item> -->

Press **Ctrl**-**S** to save your changes to the file.

1. Open the src\pages\settings\settings.ts file and replace:

import { NavController, NavParams } from 'ionic-angular';

with:

import { App, ViewController } from 'ionic-angular';

import { Config } from '../../providers/config';

This loads the Ionic App and ViewController components used by the page’s code as well as the Config provider.

Next, replace the following constructor paramaters:

public navCtrl: NavController,

public navParams: NavParams

with the following:

public app: App,

public config: Config,

public view: ViewController

When you’re done, the constructor will look like:

constructor(

public app: App,

public config: Config,

public view: ViewController

) { }

Add the following variable declarations to the beginning of the SettingsPage class:

storageType: String;

oStorageType: String;

The page needs to be able to tell if the user changed the storage type while the dialog is displayed, so these variables are key to the process. When the page opens, the application gets the currently configured storage location from the Config provider and stores it in both variables. The storageType variable maps to the page’s input form, so when the user taps the **Done** button, the values in storageType and oStorageType are compared; if they’re the same, nothing happens and the page is closed. If they’re different, then the selected storage type is passed back to the ClientList page where the application’s configuration and UI are updated accordingly.

To implement this logic, add the following functions to the class:

ionViewDidEnter() {

//Set the browser window title, just because we can

this.app.setTitle(this.config.appNameShort + ': Settings');

//Get the current setting for storage type

this.config.getStorageType().then(res => {

//this variable is used to drive the input form value(s)

this.storageType = res;

//keep the original setting as well (for comparison purposes later)

this.oStorageType = res;

});

}

dismiss() {

//since the user cancelled, don't return any data to the page

this.view.dismiss();

}

save() {

if (this.oStorageType === this.storageType) {

//return nothing to the calling page

this.view.dismiss();

} else {

console.log(`Settings Form: Storage type changed to ${this.storageType}`);

//Return the new value to the calling page

this.view.dismiss(this.storageType);

}

}

1. Finally, update the ClientList page to is loads the Settings page when the user taps the settings button:

showSettings() {

//Create the settings form in a modal dialog

let clientModal = this.modalCtrl.create(SettingsPage);

//display the modal form

clientModal.present();

//Do something with the returned data

clientModal.onDidDismiss(data => {

if (data) {

//Set the data provider based on the value returned by

//the settings page

this.config.setStorageType(data).then(res => {

//Change the data provider

this.clientData.setDataProvider(data);

});

} else {

//The user must have cancelled

console.log('No data returned from modal');

}

});

}

Press **Ctrl**-**S** to save your changes to the file.

## Implement Login

1. Add the Login page to the application using the Ionic CLI

ionic g page Start

1. With the page created, you have to tell the application about the new page. Open the project’s src\app\app.module.ts file and add the following import statement to the top of the file:

import { StartPage } from '../pages/start/start';

Next, add the following reference to the declarations and entryComponents arrays:

StartPage

When you’re done, the arrays will look like the following:

declarations: [

AboutPage,

AccountDetail,

AccountForm,

ClientDetail,

ClientForm,

ClientList,

InvestmentDetail,

InvestmentForm,

MyApp,

SettingsPage,

**StartPage**

],

entryComponents: [

AboutPage,

AccountDetail,

AccountForm,

ClientDetail,

ClientForm,

ClientList,

InvestmentDetail,

InvestmentForm,

MyApp,

SettingsPage,

**StartPage**

],

1. Now, lets setup the page. Open the project’s src\pages\settings\start.html file, and replace the generated content in the file with the following:

<ion-header>

<ion-navbar>

<ion-title>{{config.appNameLong}}</ion-title>

<ion-buttons end>

<button ion-button icon-only (click)="showIonicInfo()">

<ion-icon name="ionic"></ion-icon>

</button>

</ion-buttons>

</ion-navbar>

</ion-header>

<ion-content padding>

**Add some content here...**

<button ion-button (click)="doLogin()" block>Login</button>

**And, some content here...**

</ion-content>

<ion-footer>

<ion-toolbar>

({{clientData.storageType}})

</ion-toolbar>

</ion-footer>

This creates a simple login page to act as the start page for the application. The only critical part of the page is the **Login** button shown in the code. Add some content above and below the button as needed.

1. Open the project’s src\pages\settings\start.ts file, and replace the generated content in the file with the following:

import { Component } from '@angular/core';

import { AlertController, App, NavController, Platform } from 'ionic-angular';

import { ClientData } from '../../providers/client-data';

import { Config } from '../../providers/config';

import { UserData } from '../../providers/user-data';

@Component({

selector: 'page-start',

templateUrl: 'start.html'

})

export class StartPage {

constructor(

public alertController: AlertController,

public app: App,

public clientData: ClientData,

public config: Config,

public navCtrl: NavController,

public platform: Platform,

public user: UserData

) { }

ionViewDidEnter() {

this.platform.ready().then(() => {

this.app.setTitle(this.config.appNameLong);

});

}

doLogin() {

//When this is successful, the user:login event is fired and

//the ClientList page is displayed from there

this.user.login();

}

showIonicInfo() {

let alert = this.alertController.create({

title: this.config.appNameShort,

message: 'This application was hand crafted by professional software developers, working in a carefree environment, using <a href="http://taco.visualstudio.com/" target="\_blank"><strong>Microsoft Visual Studio</strong></a> and the <a href="http://ionicframework.com/" target="\_blank"><strong>Ionic 2</strong></a> framework.',

buttons: [{ text: 'OK' }]

});

alert.present();

}

}

1. Finally, update the project’s src\app\app.component.ts file, and add the following import to the file:

import { StartPage } from '../pages/start/start';

Add the following to the parameters passed in the constructor:

public events: Events,

When you’re done, the constructor should look like the following:

constructor(

clientData: ClientData,

**public events: Events,**

public menu: MenuController,

platform: Platform

)

The Events component enables the application so fire and respond to events. As you’ll see in a little while, the application fires **login**, **logout** and **data source change events** so interested parts of the application can respond.

In the class, replace the following line of code:

rootPage: any = {};

With the following:

rootPage: any = StartPage;

This configures the application to launch the Start page by default when the application starts; keeping them from viewing any data until logged in, or until a data source is set that doesn’t require authentication. You’ll see where this happens in a minute.

In the constructor, replace:

this.menu.enable(true, 'loggedOutMenu');

this.menu.enable(false, 'loggedInMenu');

this.rootPage = ClientList;

With the following code:

//Start listening for the login event

this.events.subscribe('user:login', (data) => {

console.log('app.component: Processing user:login event');

//We're logging in, so we the appropriate provider needs access to the

//client object.

clientData.provider.setClientObject(data.client);

/\* Setting the client object (above) used to be handled by a login event

handler in the online provider. However, we ran into a timing issue where

the following line of code executed before the provider received the

event. This caused the app to open the page, but not display any data.

Setting the client object here fixed the problem, but violates some

separation of concerns rules.

Both online and offline modes need access to this client object, so the

logic is safe here as at this point, the app.component knows whether it

needs to support an authenticated user or not at this point and let the

provider merely manage access to data. \*/

//Open the client list page now that we're logged in

this.nav.setRoot(ClientList);

});

//Listen for the logout event

this.events.subscribe('user:logout', () => {

console.log('app.component: Processing user:logout event');

//We've logged out, so clear the provider's link to the Azure client.

clientData.provider.setClientObject(null);

//Then show the start page

this.nav.setRoot(StartPage);

});

//Whenever the data source changes, we need to update the

//app's UI accordingly. So, register an event listener

//for the data source changed event

this.events.subscribe('client-data:change', () => {

console.log('app.component: Data source change detected');

//Do we have an option that requires login?

let showLogin = clientData.showLogin();

//Set menu status (flip the menus)

this.menu.enable(!showLogin, 'loggedOutMenu');

this.menu.enable(showLogin, 'loggedInMenu');

//Now, which page do we need to go to next?

if (showLogin) {

//Show the start page (if login is required)

this.nav.setRoot(StartPage);

} else {

//Otherwise, open the client list page

this.nav.setRoot(ClientList);

}

});

This sets up event listeners for the following events:

* client-data:change – Fires whenever the application changes the selected data source. It happens when the application launches, and the current storage type value is read from the application’s configuration (in the Config provider) and whenever the user makes a change using the SettingsPage.
* user:login – Fires when the user successfully completes the login process.
* user:login – Fires when the user logs out of the application.

All of this code essentially makes sure the right page is displayed after one of the events occurs.

1. At this point, your changes are complete and you can run the application using the Simulate in Browser option, a device simulator or emulator, or on a physical device.

When you open the application, open the Settings page by tapping on the icon in the upper-right corner of Figure 2.

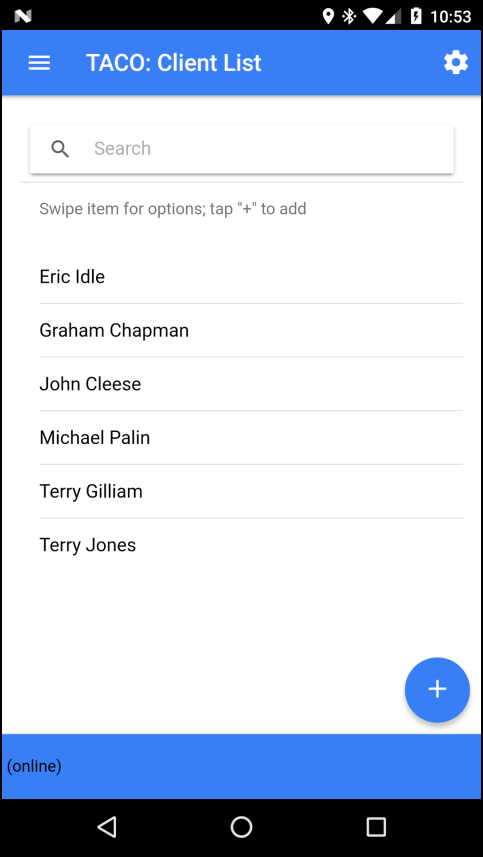


Figure – TACO Finance App: Start (Login) Page

The application will open the Settings page. Select the **Online** option as shown in Figure 3, then tap the **Done** button to save your changes.

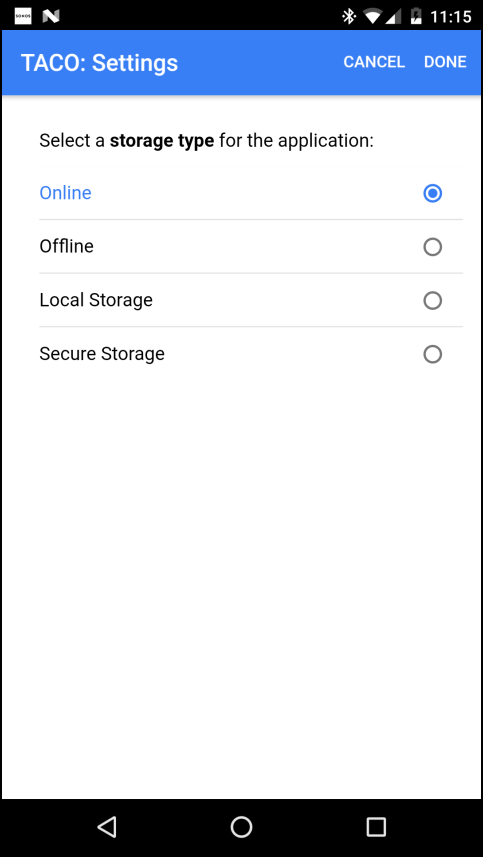


Figure – TACO Finance App: Settings Page

The application will then open the Start page, prompting you to login to the Azure Mobile App Service using the credentials provided by the workshop leader.

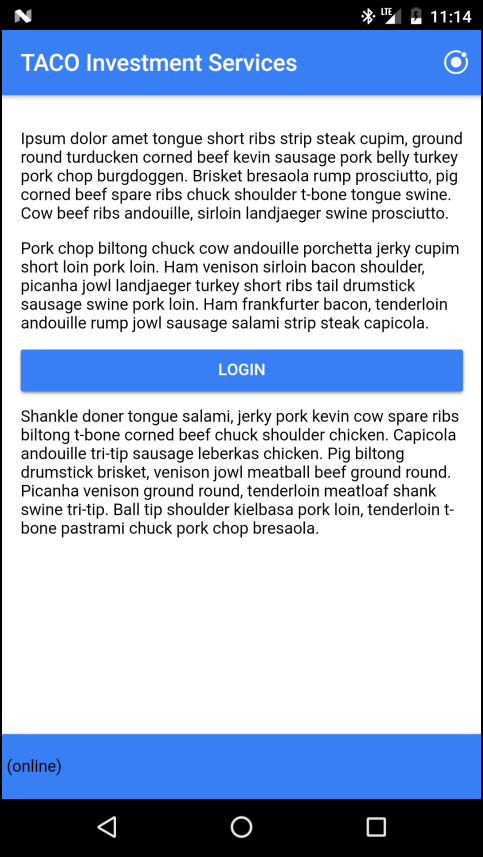


Figure – TACO Finance App: Start Page