Cordova - Guide - NoteTaker App

- Dr Nick Hayward

A general outline for developing a sample application with Cordova.

Contents

intro

We can now use the various options and features of IndexedDB to create a working Notes app, which we'll continue to expand and build out to test Cordova development. We can update the UI, add some extra plugins (existing and custom...), work with remote APIs and services, test, secure, and then publish.

basic requirements

Let us start by setting up our initial app. We're going to need the following to get us started,

- Cordova base app created

- add support for required platforms
  - Android & Browser (for testing...)
- add any initial plugins
- set icon and splashscreens in `config.xml`

Then, we can start to create the app itself

- add home screen
  - initial design and layout
  - initial widgets and elements
  - add some basic styling
- add IndexedDB support
  - create base **object stores**
  - load some notes
- add some more UI options
  - change view of notes - grid, list...
  - sort and filter notes
- view single note

Then, we can move on to v2. of the app...

icon and splashscreen

We've created our initial Cordova app using the CLI tool, added required platforms, and built the initial app to test it loads correctly. We'll now add a splashscreen and icon for the app,

- icon
- splashscreen

We'll add the splashscreen plugin,

```
cordova plugin add cordova-plugin-splashscreen
```

and then update our `config.xml` file to add support for the required splashscreens and icons.

We can also set a value for the splashscreen timeout, which controls the default *AutoHide* for the splashscreen.

```
<preference name="SplashScreenDelay" value="3000" />
```

If necessary, this option can be overridden either programmatically in our app's JS, or in the `config.xml` file as follows

```
<preference name="AutoHideSplashScreen" value="false" />
```

Default value is set to `value="true"`, which then hides the splashscreen at the specified value for the delay.

If the `AutoHideSplashScreen` is set to false, it will need to be hidden programmatically in the app's JS.

`config.xml` - splashscreens and icons

We can now add our splashscreens and icons to the `config.xml` file

```
<platform name="android">
        <icon density="ldpi" src="resources/android/icon/drawable-ldpi-icon.png" />
```

```
        <icon density="mdpi" src="resources/android/icon/drawable-mdpi-icon.png" />
        <icon density="hdpi" src="resources/android/icon/drawable-hdpi-icon.png" />
        <icon density="xhdpi" src="resources/android/icon/drawable-xhdpi-icon.png" />
        <icon density="xxhdpi" src="resources/android/icon/drawable-xxhdpi-icon.png"
/>
        <icon density="xxxhdpi" src="resources/android/icon/drawable-xxxhdpi-
icon.png" />
        <splash density="land-ldpi" src="resources/android/splash/drawable-land-ldpi-
screen.png" />
        <splash density="land-mdpi" src="resources/android/splash/drawable-land-mdpi-
screen.png" />
        <splash density="land-hdpi" src="resources/android/splash/drawable-land-hdpi-
screen.png" />
        <splash density="land-xhdpi" src="resources/android/splash/drawable-land-
xhdpi-screen.png" />
        <splash density="land-xxhdpi" src="resources/android/splash/drawable-land-
xxhdpi-screen.png" />
        <splash density="land-xxxhdpi" src="resources/android/splash/drawable-land-
xxxhdpi-screen.png" />
        <splash density="port-ldpi" src="resources/android/splash/drawable-port-ldpi-
screen.png" />
        <splash density="port-mdpi" src="resources/android/splash/drawable-port-mdpi-
screen.png" />
        <splash density="port-hdpi" src="resources/android/splash/drawable-port-hdpi-
screen.png" />
        <splash density="port-xhdpi" src="resources/android/splash/drawable-port-
xhdpi-screen.png" />
        <splash density="port-xxhdpi" src="resources/android/splash/drawable-port-
xxhdpi-screen.png" />
        <splash density="port-xxxhdpi" src="resources/android/splash/drawable-port-
xxxhdpi-screen.png" />
</platform>
<preference name="SplashScreenDelay" value="3000" />
```

n.b. we'll initially set `SplashScreenDelay` in the `config.xml` file...

initial home screen

Let us now consider the initial requirements for our app's home screen. We'll need the following places and functionality for our NoteTaker app,

Places

- header & navbar
  - title
  - icons for *create note*, *menu*...
- main content
  - heading
  - grid for notes
- footer

Functionality

- view all notes
  - single note with title, snippet, &c.
  - no. of notes
- switch layout of notes
  - grid, list, &c.

- filter and sort notes
- ...

jQuery Mobile - index.html

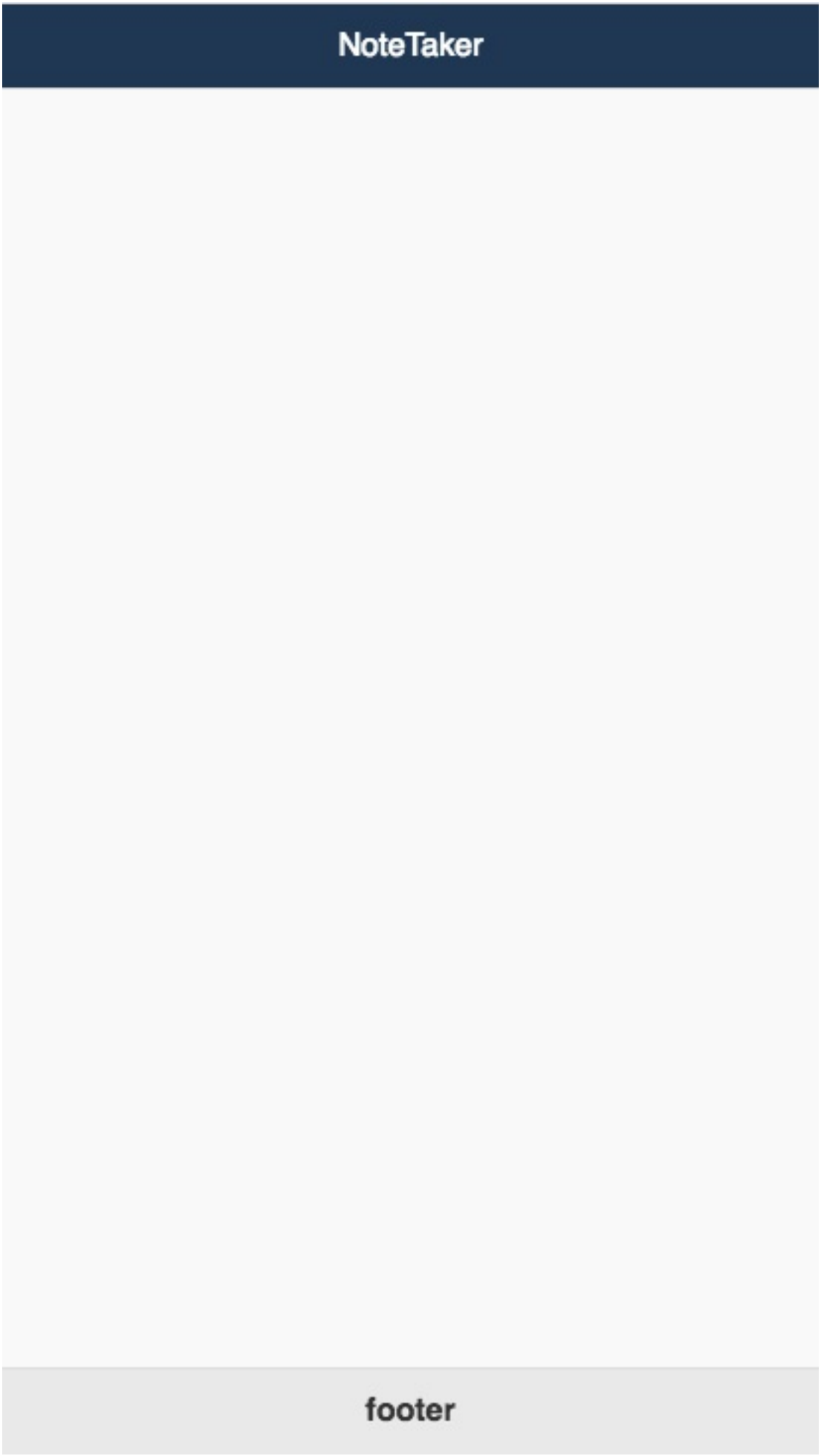
We'll start to customise our app's `index.html` page, adding our initial structure and custom CSS styles. Our first example uses the familiar jQuery Mobile structure as a useful point of reference,

```
<body>
        <!-- app pages -->
        <!-- page1 - home screen -->
        <div data-role="page" id="home">
                <div data-role="header">
                        <h3>NoteTaker</h3>
                </div><!-- /header -->
                <div role="main" class="ui-content">
                        ...
                </div><!-- /content -->
                <div data-role="footer" data-position="fixed" class="page-footer" >
                        <h5>footer</h5>
                </div><!-- /footer -->
        </div><!-- /page1 - home screen -->
</body>
```

We can then modify the initial CSS for the app to fit the requirements of the app and the specifics of jQuery Mobile,

```
/* remove default all uppercase...*/
body {
        text-transform: none;
}
/* customise page header */
.ui-page .ui-header {
        background-color: #1a3852;
}
/* customise header title */
.ui-header .ui-title {
        font-weight: normal;
        text-shadow: none;
        color: #fff;
}
```

Image - NoteTaker - Home Screen 1 - jQuery Mobile



OnsenUI setup

As a useful comparison, we can now start to introduce alternative UI frameworks for developing cross-platform apps.

Our first option is OnsenUI, which currently supports development with

- JavaScript
- Angular 1 & 2
- React

After creating our initial Cordova app, using the familiar pattern we've seen for jQuery Mobile based apps, we then need to install OnsenUI, and add to our project.

A quick and easy way to install and setup OnsenUI is using either NPM or Bower. Using NPM, we can install Bower using the following terminal command,

```
npm install -g bower
```

We'll use Bower to install UI components and dependencies for building our OnsenUI projects, updating as needed, and adding any other front-end dependencies.

So, we might install our OnsenUI bower components in the `www` directory of the Cordova project,

```
cd www
bower install onsenui
```

A new directory will be created for `bower_components` in the project's `www` directory.

Add OnsenUI files to project

Now, we need to add OnsenUI to our Cordova project. Specifically, we add links for the framework's CSS and JS files

```
...
<head>
...
<!-- setup css -->
<link rel="stylesheet" type="text/css" href="css/index.css">
<link rel="stylesheet" type="text/css"
href="bower_components/onsenui/css/onsenui.css">
<link rel="stylesheet" type="text/css" href="bower_components/onsenui/css/onsen-css-
components.css">
<link rel="stylesheet" type="text/css" href="css/style.css">
<!-- setup js -->
<script type="text/javascript" src="cordova.js"></script>
<script type="text/javascript" src="bower_compoents/onsenui/js/onsenui.js"></script>
<script type="text/javascript" src="js/app.js"></script>
<title>NoteTaker</title>
</head>
...
```

We're adding the required framework CSS and JavaScript to allow us to use the specific OnsenUI elements, structures, methods, &c. We can also update these files to support custom themes and styles, which we can create using the **OnsenUI Theme Roller**.

OnsenUI concepts - `ons` object, elements, page content...

working with `ons` object

The `ons` object is exposed as an integral part of working with OnsenUI. It forms part of the core OnsenUI library.

We can use this object with available methods to help us work with various options and structures within OnsenUI. For

example, methods exposed for a Tab Bar, Page, or associated utility functions.

Elements in OnsenUI

OnsenUI specific elements are all custom, and defined with tag prefix of `<ons-`. However, they still include attribute and class patterns, and provide properties and events as expected for standard HTML.

We can still traverse an OnsenUI created DOM as expected.

OnsenUI includes many UI components. e.g. we can add the `<ons-navigator>` component to provide management of a defined page stack and app navigation.

```
<ons-navigator id="navigator" page="page1.html"></ons-navigator>
```

We can then update our app's JS to handle the navigation.

add page content

We can also add many types of pre-built OnsenUI components, including

- buttons, dialogs, notifications...
- toolbars, pages, splitters, tabs...
- forms, lists...

and many more. There are also helper properties and functions such as *infinite scroll*, which provide pre-built ways to manage content, rendering, layout, and general development within an app.

OnsenUI - index.html

We can add a page using the `<ons-page>` element. This becomes an initial container for the whole page, and the root element for other page elements.

Then, we can add a toolbar to the top or foot of a page using the `<ons-toolbar>` element.

```
<!-- page root -->
<ons-page>
  <!-- page toolbar -->
  <ons-toolbar>
  ...
  </ons-toolbar>
</ons-page>
```

A toolbar can be divided into three sections using a class name, `left`, `center`, and `right`, which simply correspond to relative positioning.

Each section can then hold an icon, `<ons-icon>`, or a toolbar button, `<ons-toolbar-button>`, or a `<ons-back-button>`. We can also simply add HTML content, such as a title for the app or page.

We can then update our toolbar with a menu icon, title, and search icon

```
<!-- page toolbar -->
<ons-toolbar>
        <div class="left">
    <ons-toolbar-button>
      <!-- hamburger menu -->
      <ons-icon icon="md-menu"></ons-icon>
    </ons-toolbar-button>
  </div>
  <div class="center">NoteTaker</div>
  <div class="right">
```

```
        <ons-toolbar-button>
      <!-- search option -->
      <ons-icon icon="md-search"></ons-icon>
    </ons-toolbar-button>
  </div>
</ons-toolbar>
```

Our current OnsenUI design is as follows.

Image - NoteTaker - add initial navbar - OnsenUI


4G
1:05
NOTETAKER

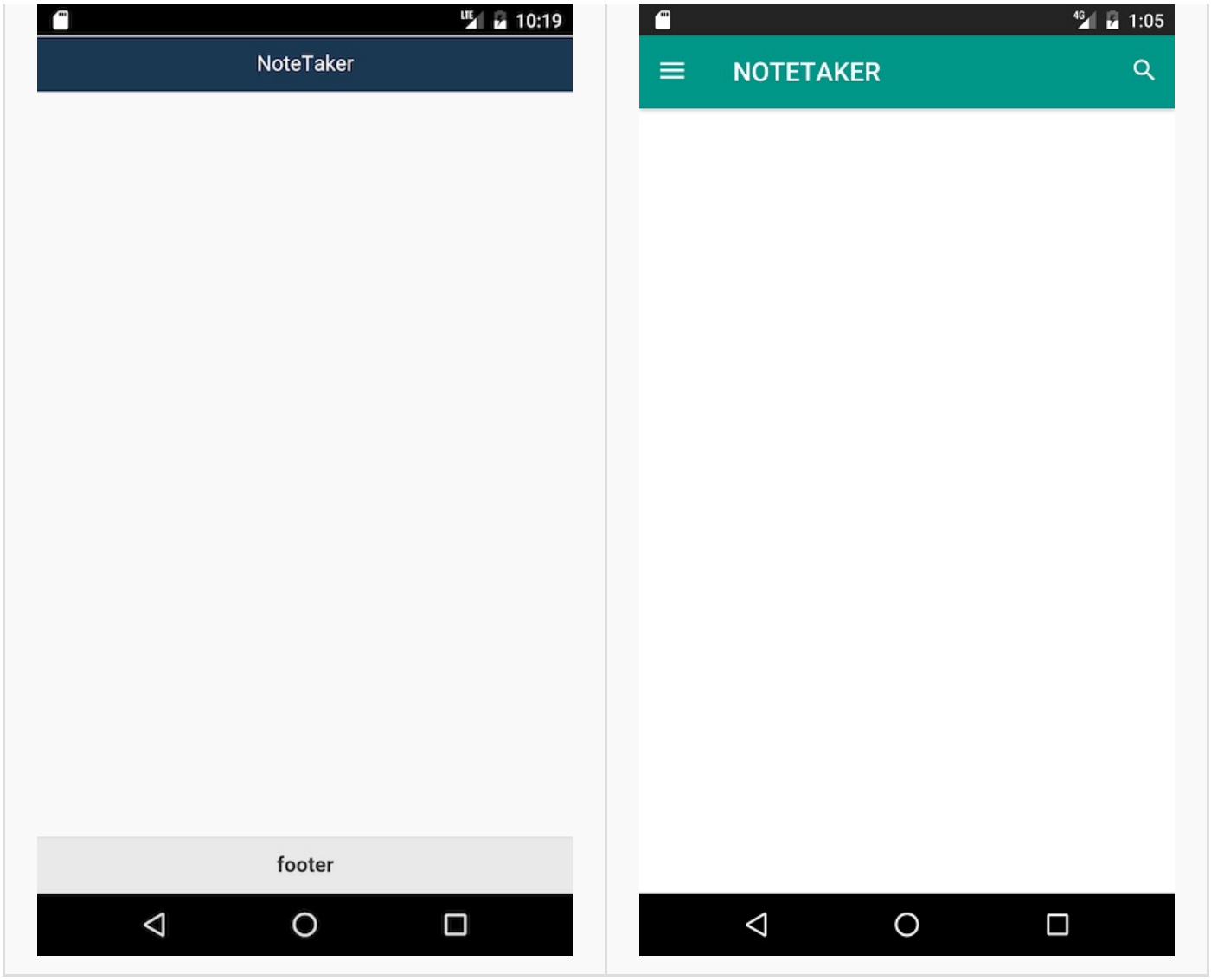
So, our initial homepage is as follows,

```
<ons-page id="home">
  <!-- page toolbar -->
  <ons-toolbar>
    <div class="left">
      <ons-toolbar-button>
        <!-- hamburger menu -->
        <ons-icon icon="md-menu"></ons-icon>
      </ons-toolbar-button>
    </div>
    <div class="center">NoteTaker</div>
    <div class="right">
      <ons-toolbar-button>
        <!-- search option -->
        <ons-icon icon="md-search"></ons-icon>
      </ons-toolbar-button>
    </div>
  </ons-toolbar>
  <!-- home page content -->
  <h4>notes</h4>
  ...
  <ons-button id="push-button">Create</ons-button>
</ons-page>
```

Image - NoteTaker - Statusbar - default

However, for both jQuery Mobile and OnsenUI, the statusbar at the top of the UI for Android still remains the default black. As you can see, aesthetically this is noticeably an issue. So, we'll have to fix the statusbar and update the colour to match our initial app design and colour scheme.

jQuery Mobile	OnsenUI



statusbar

We can now add Cordova's `statusbar` plugin, which will help us customise the Android statusbar at the top of the UI,

```
cordova plugin add cordova-plugin-statusbar
```

Then, we can update our app's `config.xml` file to include the following

```
<preference name="StatusBarBackgroundColor" value="#1a3852" />
```

which modifies the background colour of the status bar to match the aesthetics of our app's colour scheme.

There are many other options, including transparency, various styles &c., which are detailed in the Cordova API,

- Cordova Plugin Statusbar

n.b. Cordova Plugin Statusbar

Image - NoteTaker - Statusbar - custom

Our app's updated custom status bar, which now matches our app's colour scheme.

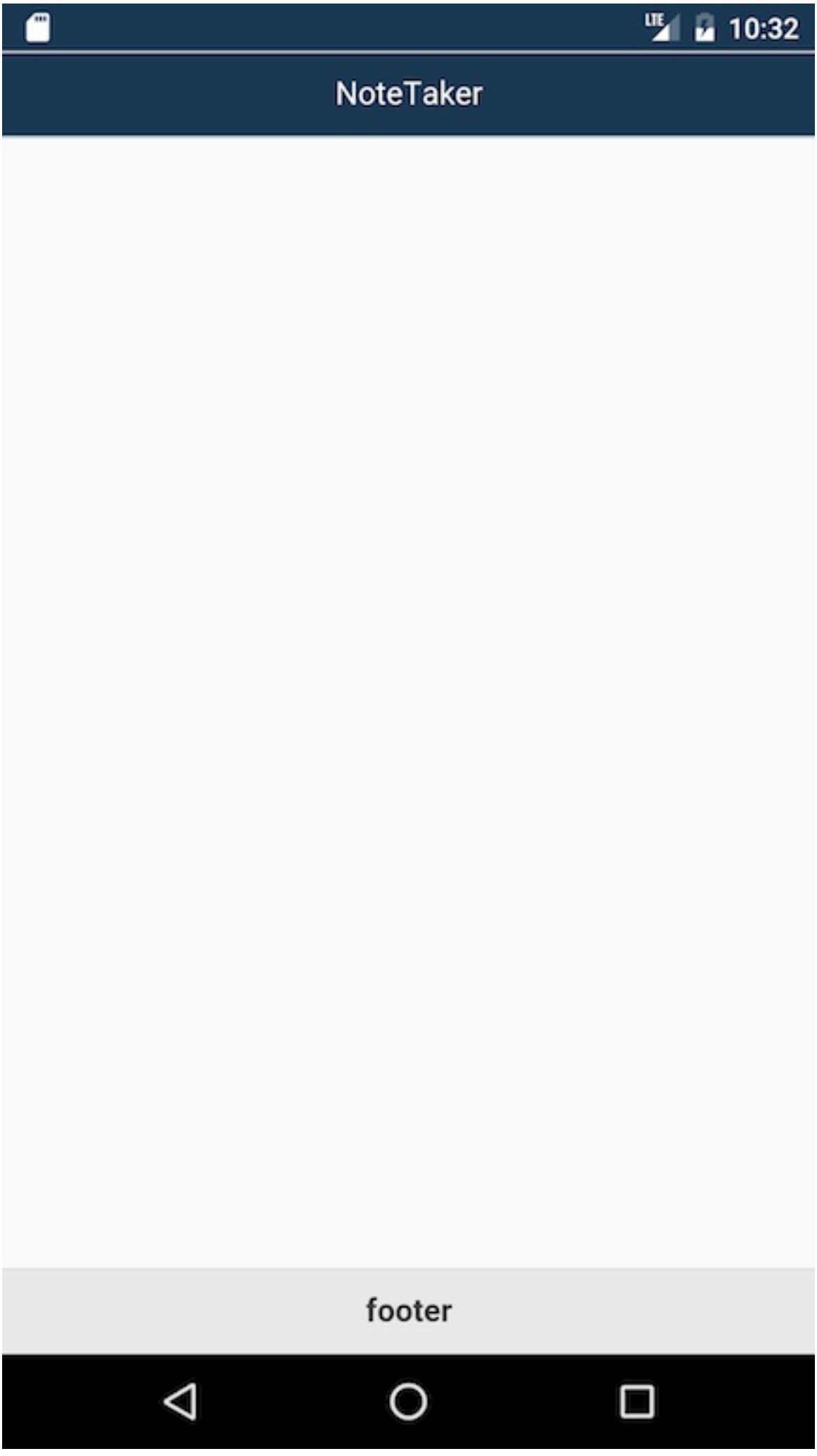
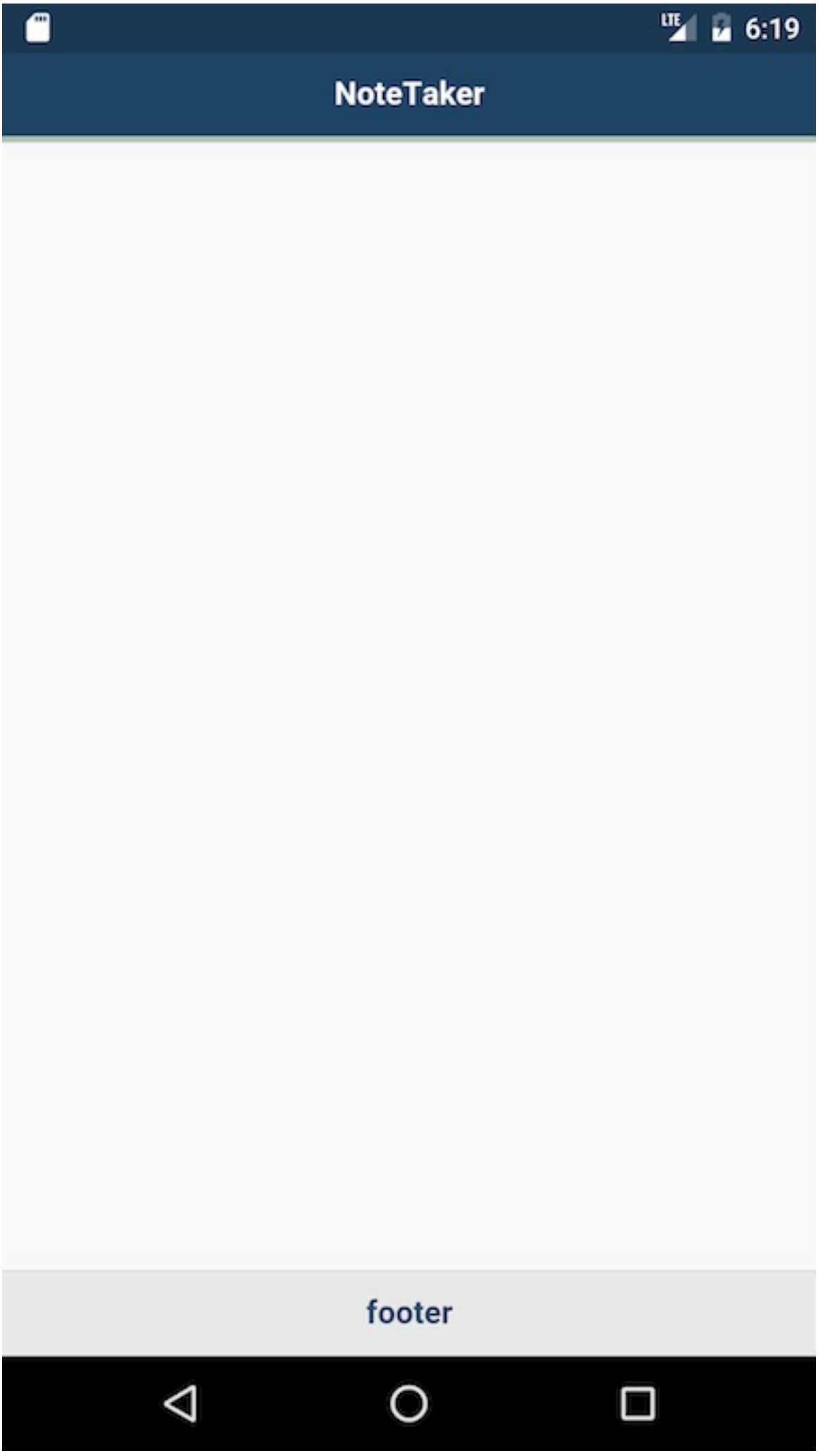

10:32
NoteTaker
footer

Image - NoteTaker - Statusbar - custom

And then, update the custom status bar to better match Android design guidelines.


6:19
NoteTaker
footer

jQuery Mobile - index.html - part 2

We can start to add some content to the home screen, including a create button for a new note, grid layout for the notes, and a second page for the *create note* form. e.g.

```
<!-- header -->
<div data-role="header" class="ui-nodisc-icon">
        <h3>NoteTaker</h3>
        <a href="#create" data-transition="slideup"
        class="ui-btn ui-icon-plus ui-btn-icon-notext ui-btn-right">create</a>
</div><!-- /header -->
```

This updates our header, adding a *plus* icon for the *create note* option. To match the aesthetics of the app, we'll also need to modify svg properties for the underlying icon,

```
/* custom svg for button - plus - custom fill colour = 1f4463 */
.ui-icon-plus:after {
        background-image: url("...polygon%20fill%3D%22%231f4463...");
}
```

The interesting part is the *polygon fill*, which we can modify to fit our app's colour scheme,

```
  polygon%20fill%3D%22%231f4463%
```

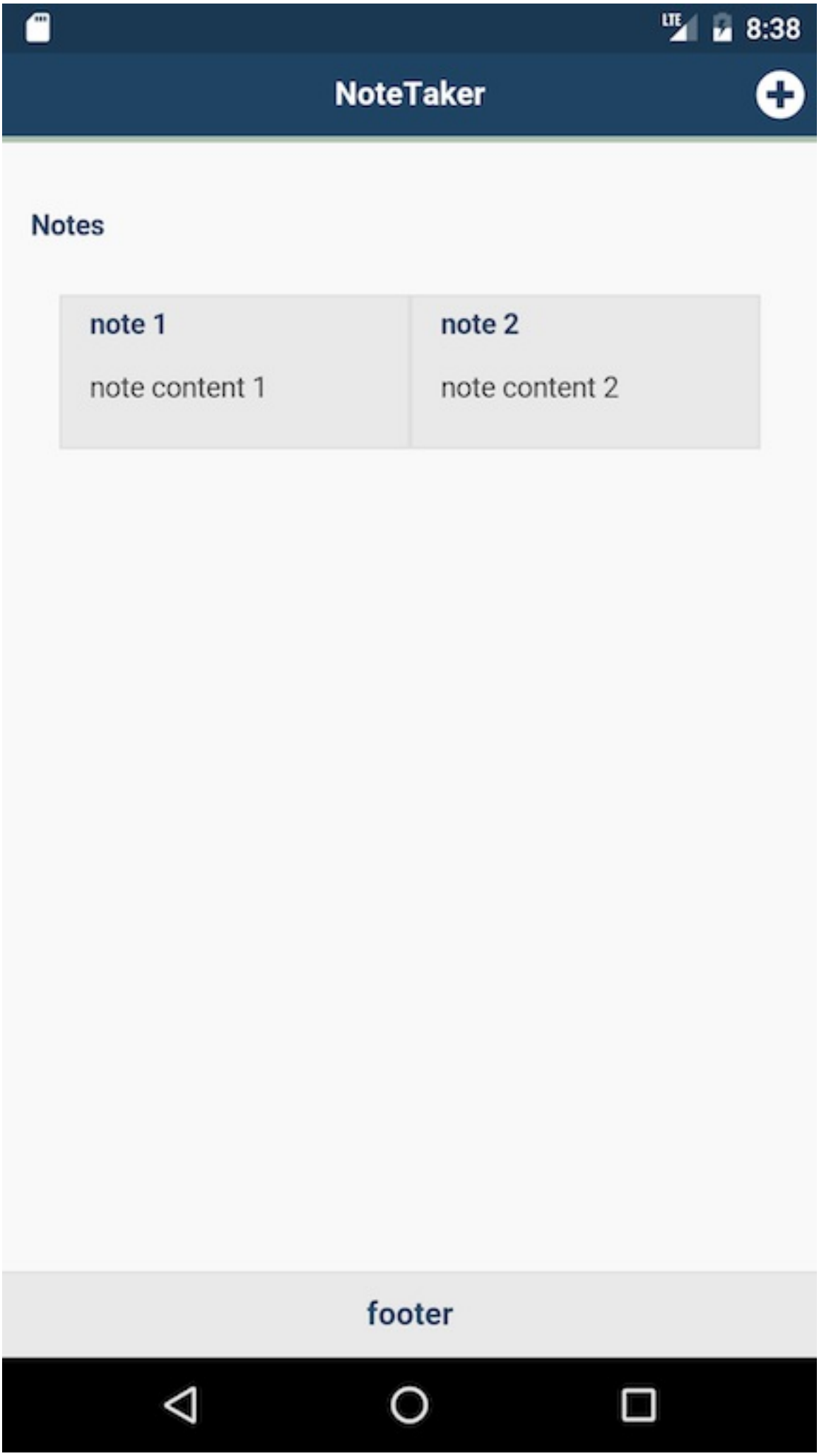
Our notes will be rendered by default using a grid pattern, which contains a note's title and snippet of content.

This grid will use the following pattern, which we can later abstract in JS as we read the notes from the specified data store.

```
<div role="main" class="ui-content">
        <h4>Notes</h4>
        <!-- output available notes -->
        <div id="notes" class="ui-body">
                <!-- grid layout for notes -->
                <div class="ui-grid-a">
                        <!-- left column -->
                        <div class="ui-block-a">
                                <div class="ui-bar ui-bar-a">
                                        <h5>note 1</h5>
                                        <p>note content 1</p>
                                </div>
                        </div>
                        <!-- right column -->
                        <div class="ui-block-b">
                                <div class="ui-bar ui-bar-a">
                                        <h5>note 2</h5>
                                        <p>note content 2</p>
                                </div>
                        </div>
                </div>
        </div>
</div><!-- /content -->
```

Image - NoteTaker - update `index.html` - jQuery Mobile

We've now modified the header and the statusbar, added a custom icon and the initial grid for notes, and created a second page for the *create note* form.


8:38
NoteTaker
Notes
note 1
note content 1
note 2
note content 2
footer

OnsenUI - index.html - part 2

We can now add a second page to allow a user to create their notes. We follow the same pattern as the home page,

```
<ons-page id="create">
  <ons-toolbar>
    <div class="left"><ons-back-button>Back</ons-back-button></div>
    <div class="center"></div>
  </ons-toolbar>
  <!-- create note page -->
  <h4>create note</h4>
  ...
</ons-page>
```

The toolbar has been modified to fit the requirements of a temporary page, including the substitution of the *menu* icon for a *back* button, and the removal of the search icon from the right section.

OnsenUI - page lifecycle

The OnsenUI `<ons-page>` element provides the following events at different points during the lifecyle of a page,

- `init`
  - fired after page is added to DOM
- `destroy`
  - fired prior to page being removed from the DOM, and just before the page is destroyed
- `show`
  - fired every time `<ons-page>` comes into the view
- `hide`
  - fired every time `<ons-page>` disappears from the view

As we start to work with the underlying logic of our app, add navigation &c. we're initially interested in the `init` event provided by `<ons-page>`.

OnsenUI - JS and initial listeners

Using OnsenUI with Cordova, for example, we need to consider the following initial events as an app starts and loads. This includes checks for Cordova loading with required plugins, and the UI for the app.

As the DOM loads, we can attach a listener for Cordova's `deviceready` event,

```
//add listener to DOM - check deviceready event...continue with app logic
document.addEventListener('deviceready', onDeviceReady, false);
```

and then call the onDeviceReady function. JS checks require that the DOM has loaded otherwise we can not attach listeners for such events. However, the DOM does not need to include the OnsenUI components before completing such Cordova checks.

Now, as mentioned before, checking this `deviceready` event is particularly important. However, if this Cordova event has not completed successfully, we will not be able to attach other listeners to the DOM. We can certainly try, but they won't work. In essence, this is because Cordova itself modifies the default event listener for an app's webview. These modifications include handling of special events such as attaching listeners &c. As noted in the Cordova JS library,

> Intercept calls to addEventListener + removeEventListener and handle deviceready,
> resume, and pause events.

Once the `deviceready` event has returned successful, the next listener needs to check that the OnsenUI `init` event has fired,

```
        document.addEventListener('init', function(event) {
    //check defined initial page for app...
        if (event.target.matches('#home')) {
      ...
    }
  }, false);
```

OnsenUI - notetaker.js

Let's start to add the logic for our OnsenUI based app. We'll start by checking for the `deviceready` event, and then check the `init` event that OnsenUI provides. We can use this to check for our home page, and then set up various checks, properties, and so on.

```
...
//cordova - add listener to DOM & check deviceready event
document.addEventListener('deviceready', function(event) {
        //prevent any bound defaults
        event.preventDefault();
        console.log("cordova checked...device ready");

        //call as ons-page added to DOM...
        function onsInit(event) {
                //properties - initial page load
                var page = event.target;
                //check IndexedDB
    //set navigation
                //check for home page
                if (page.matches("#home")) {
        //ons.notification.alert('init checked...homepage ready');
                        console.log("home page is now attached to the DOM...");
          } else {
                        console.log("away from home page...");
                }
        }
        //onsen - init event is fired after ons-page attached to DOM...
        document.addEventListener('init', onsInit, false);
}, false);
...
```

OnsenUI - navigation structure

We now have two initial pages for our NoteTaker app, so we need to add navigation from one page to another, one place to another place.

For this, we need to update our `index.html` page's structure, and then our app's logic in the `notetaker.js` file.

To provide support for multi-page navigation, we need to use one of three available navigation patterns. We can add a *navigator*, *tabbar*, or *splitter* component to help us structure our navigation blocks. Each component provides a **frame** for a defined place within our app. We can then dynamically update the inner content of each frame. These frames normally contain a `<ons-page>` component, but we can also nest multiple navigation components, if necessary.

As we add a navigator component, we can also define each individual page within our single page app's `index.html` . We define each page using the `<ons-template>` component. For a SPA, this helps us add multiple pages to a single `index.html` file.

Our updated structure is as follows,

```
<ons-navigator id="navigator" page="home.html"></ons-navigator>
<!-- home page -->
<ons-template id="home.html">
  <ons-page id="home">
   ...
  </ons-page>
</ons-template>
<!-- create note page -->
<ons-template id="create.html">
  <ons-page id="create">
    ...
  </ons-page>
</ons-template>
```

For a SPA, each `<ons-page>` component's ID attribute replaces a full URL to a separate page.

OnsenUI - navigation logic

We now need to update our app's logic to listen for navigation events. We'll add a custom function, which we'll call to set the required **stack-based** navigation that OnsenUI uses to keep track of page push and pop requests within an app.

```
//onsen - set stack-based navigation
function onsNav(page) {
  if (page.id === 'home') {
    page.querySelector('#push-button').onclick = function() {
      document.querySelector('#navigator').pushPage('create.html', {data: {title:
'Create Note'}});
    };
  } else if (page.id === 'create') {
    page.querySelector('ons-toolbar .center').innerHTML = page.data.title;
          console.log("page title = "+ page.data.title);
  }
}
```

Image - NoteTaker - check `init` event - OnsenUI

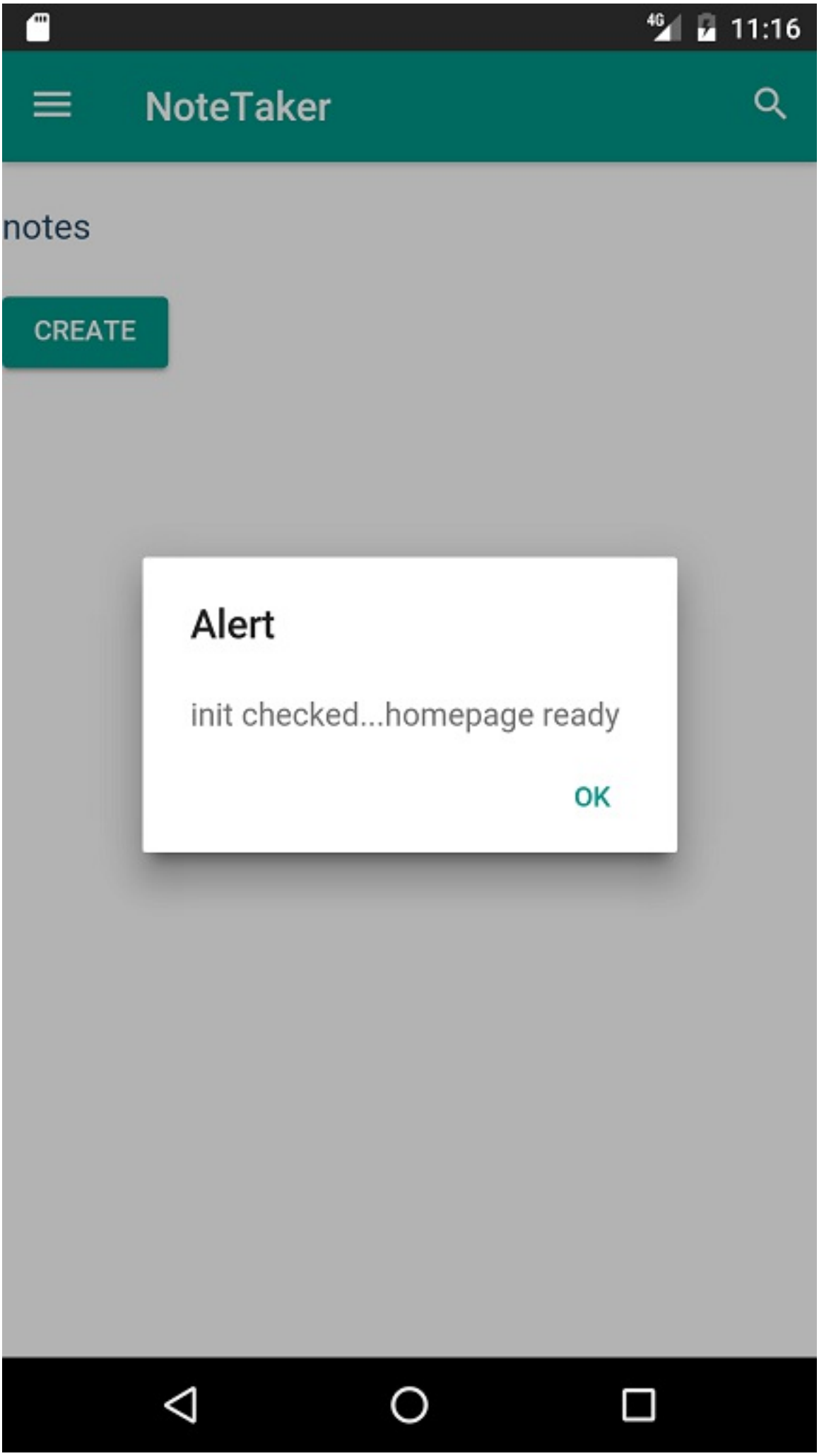

11:16
NoteTaker
notes
CREATE
Alert
init checked...homepage ready
OK

Image - NoteTaker - load home page - OnsenUI

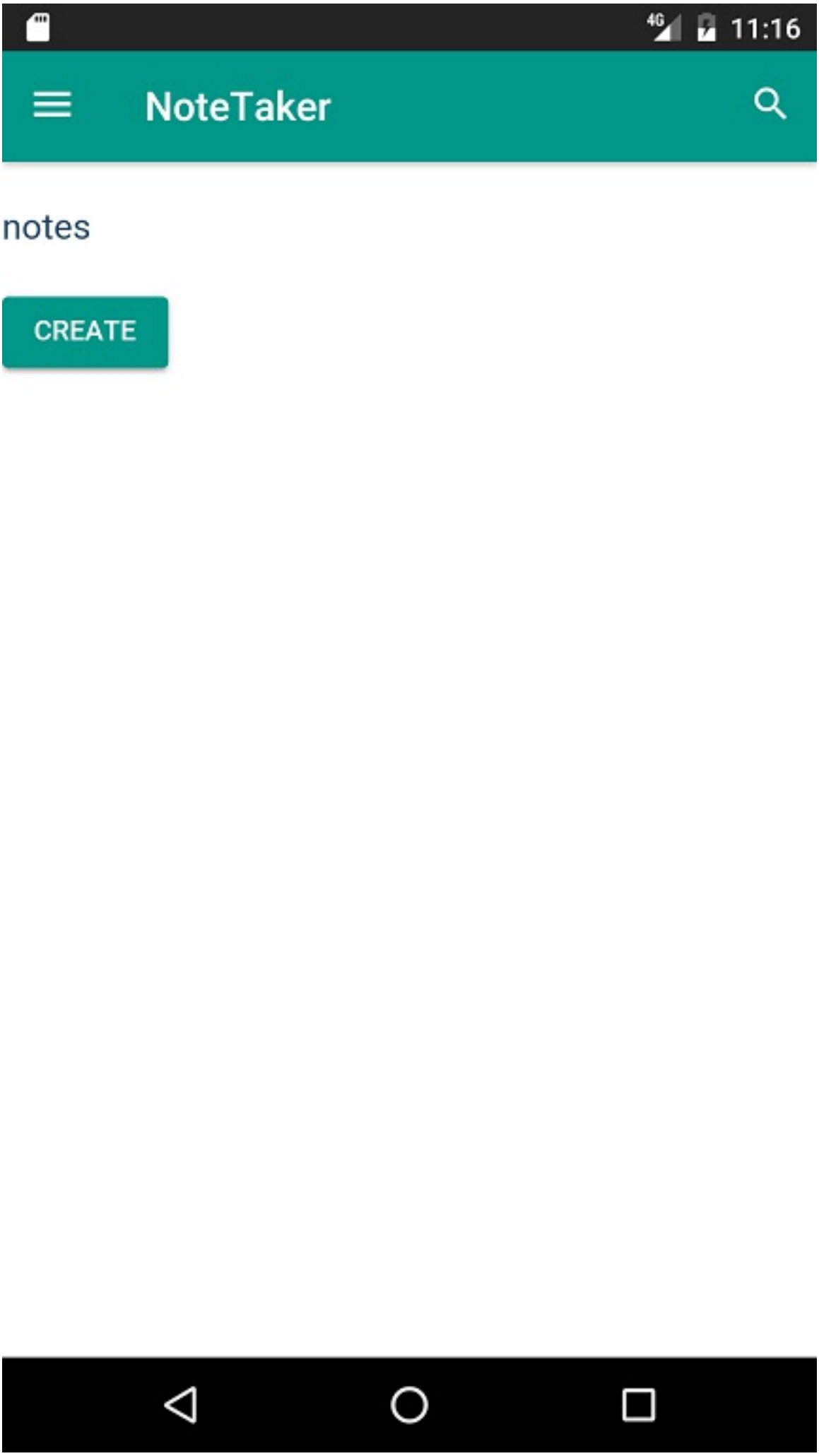

11:16
NoteTaker
notes
CREATE

Image - NoteTaker - add navigation - OnsenUI


11:16
Create Note
create note

OnsenUI - recap

So far we've added the following features and structures to our OnsenUI based app,

- home page and create note page
- initial navigation stack
- statusbar customisation and titles
- splashscreens and icon
- initial page elements
- checked loading of
  - `deviceready` for Cordova
  - `init` for OnsenUI ons-page component

and a few updates to the general aesthetics...

Image - NoteTaker - check `init` event - OnsenUI

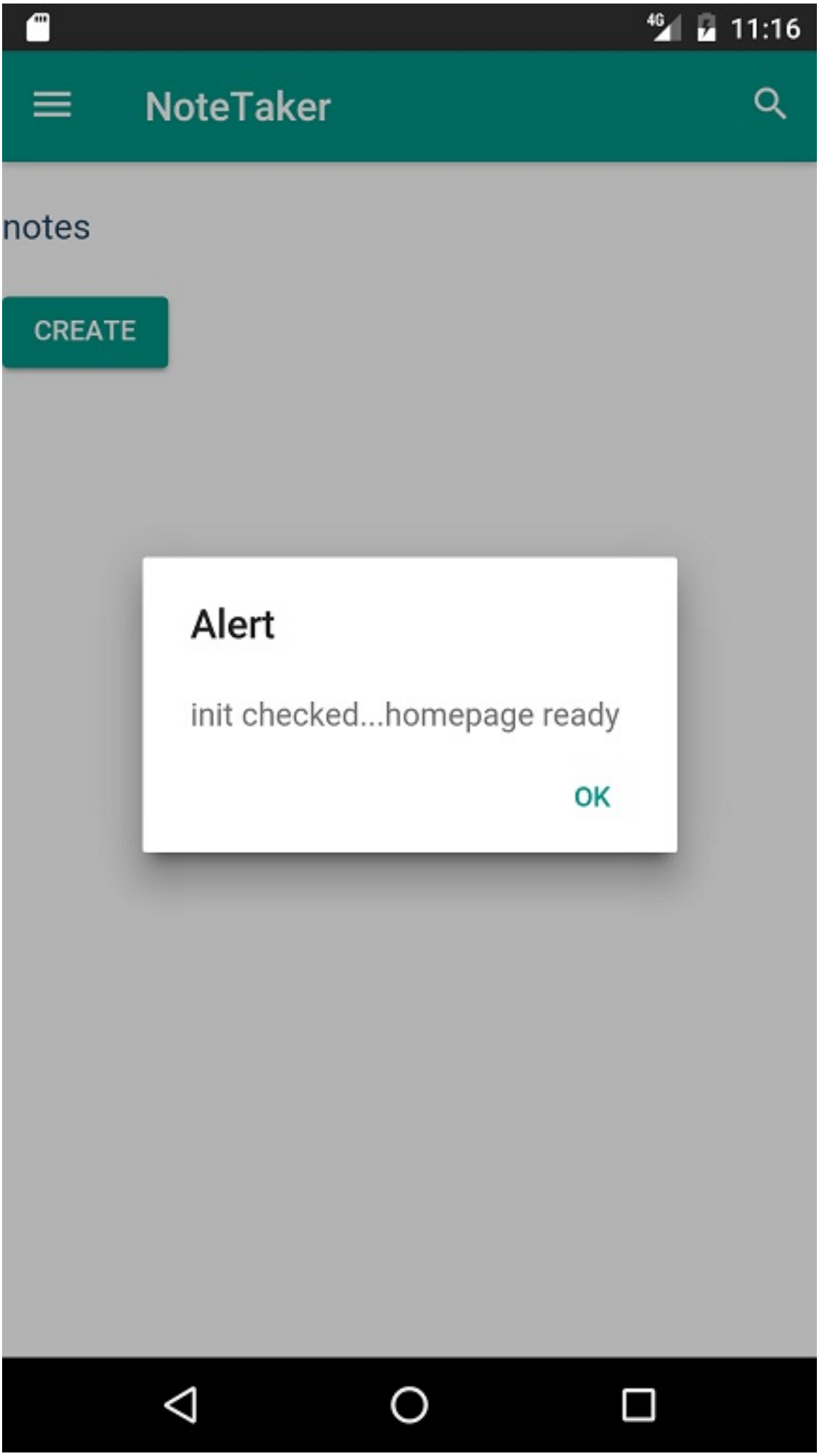

11:16
NoteTaker
notes
CREATE
Alert
init checked...homepage ready
OK

Image - NoteTaker - load home page - OnsenUI

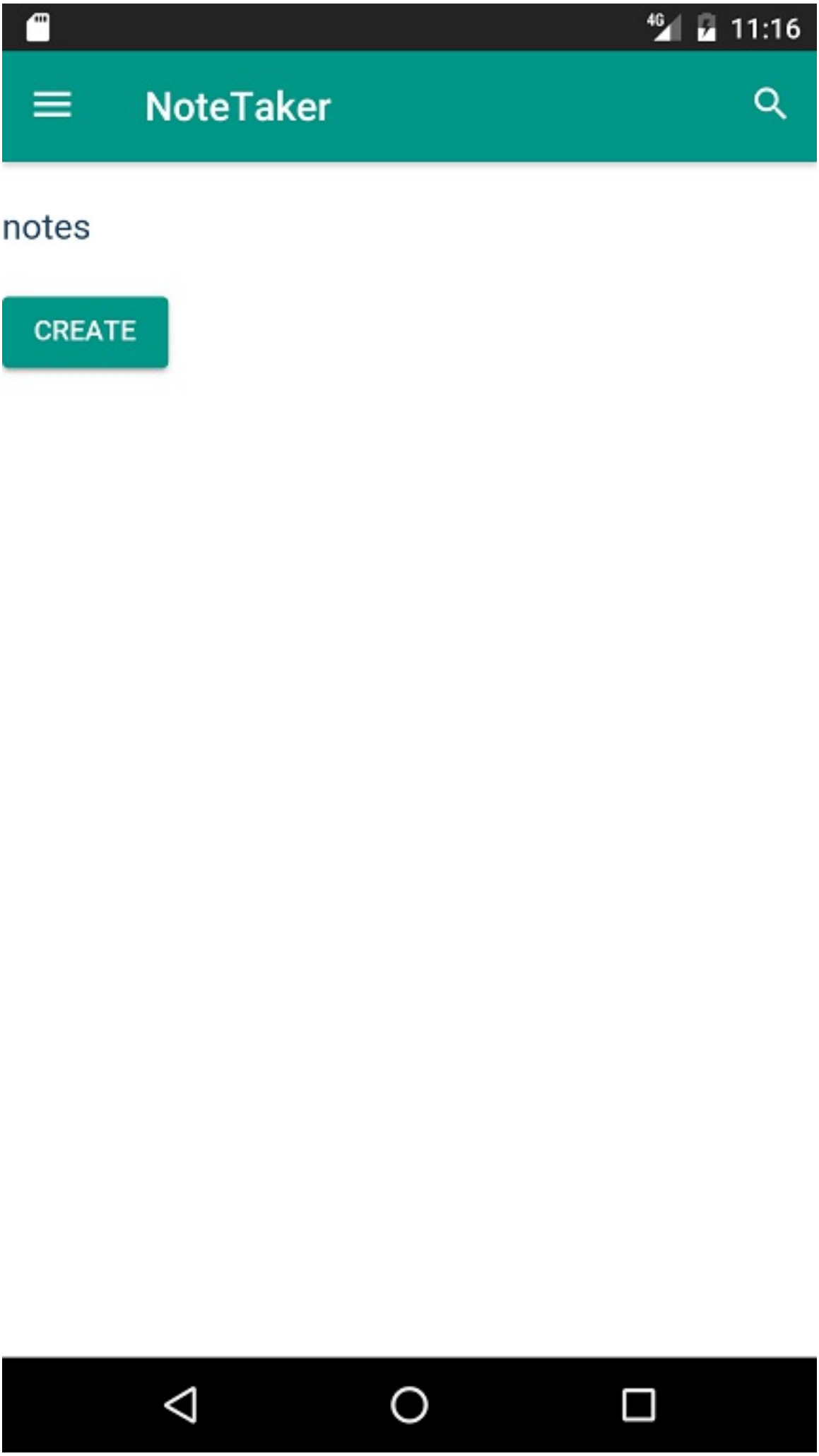

11:16
NoteTaker
notes
CREATE

Image - NoteTaker - add navigation - OnsenUI


11:16
Create Note
create note

OnsenUI - update UI

We can now start to modify our initial UI for our OnsenUI based app.

We'll add a standard Material Design icon for creating a note, which we'll use with our existing navigation stack. This will use the standard floating action button pattern, as defined in the Material Design specification,

```
<ons-fab position="bottom right">
  <ons-icon id="create-note" icon="md-plus"></ons-icon>
</ons-fab>
```

- Material Design specification - Floating Action Button

OnsenUI - update UI - grid layout

As we saw with the jQuery Mobile example, we need to use a grid layout for our initial notes. As we're considering our app for Android, initially, we can again consult the Material Design guidelines for a **grid list**.

Usage is defined as follows,

> A grid list is best suited to presenting homogenous data, typically images, and is optimized for visual comprehension and differentiating between similar data types.

So, we're OK with our notes pattern and layout. OnsenUI provides components for rows and columns, e.g.

```
<ons-row>
  <ons-col>
    <div class="note-card">
      <h5>note 1</h5>
    </div>
  </ons-col>
  <ons-col>
    <div class="note-card">
      <h5>note 2</h5>
    </div>
  </ons-col>
</ons-row>
```

As we are styling our own app, we can also add a custom class to recreate some cards for our notes. We can create our CSS ruleset for this new class, and then add some custom styling,

```
.note-card {
box-shadow: 0 1px 4px #dddddd;
background-color: #ffffff;
}
```

- Material Design Guidelines - grid list

Image - NoteTaker - grid layout portrait - OnsenUI

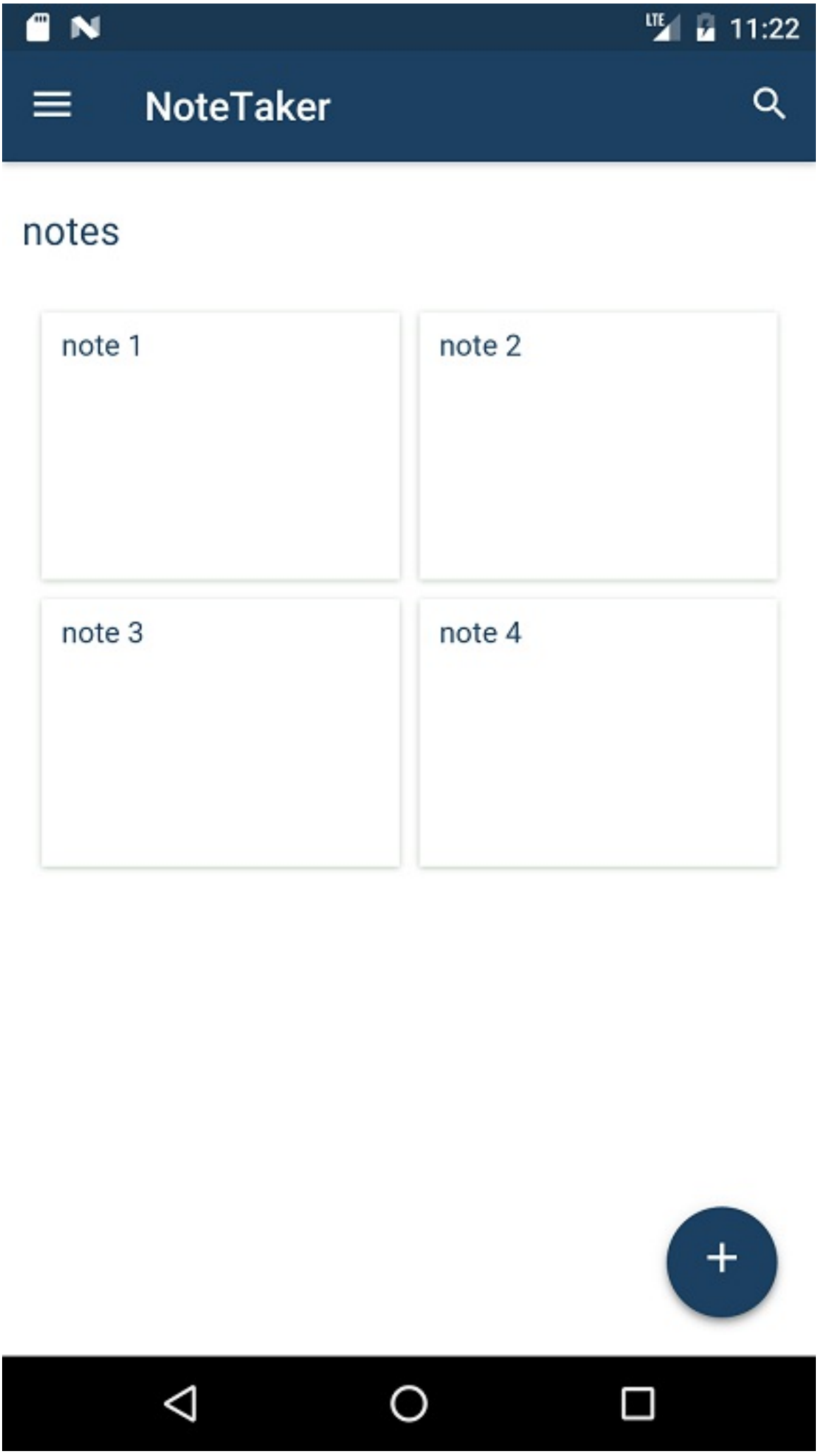
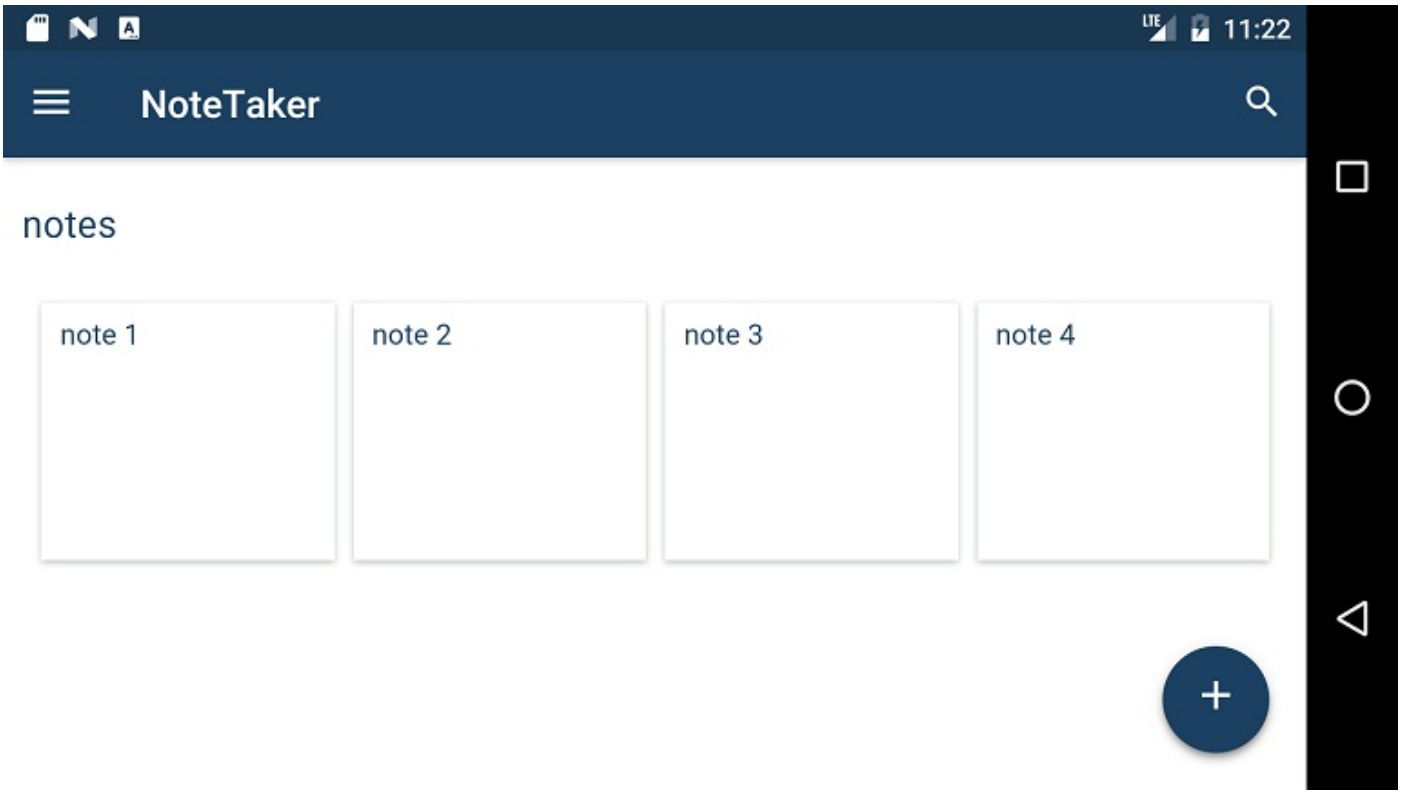

11:22
NoteTaker
notes
note 1
note 2
note 3
note 4

Image - NoteTaker - grid layout landscape - OnsenUI



OnsenUI - create note page

Our next initial update is to add options for a user to create their notes on the **Create Note** page.

Naturally, we now need to add a form with various fields for a note. With OnsenUI, one of the most common components we use is `<ons-input>`. This component supports many different common form elements, including

- checkbox, radio button, password field...

For our **Create Note** page, we need the following minimum elements and options,

- title, content, tags

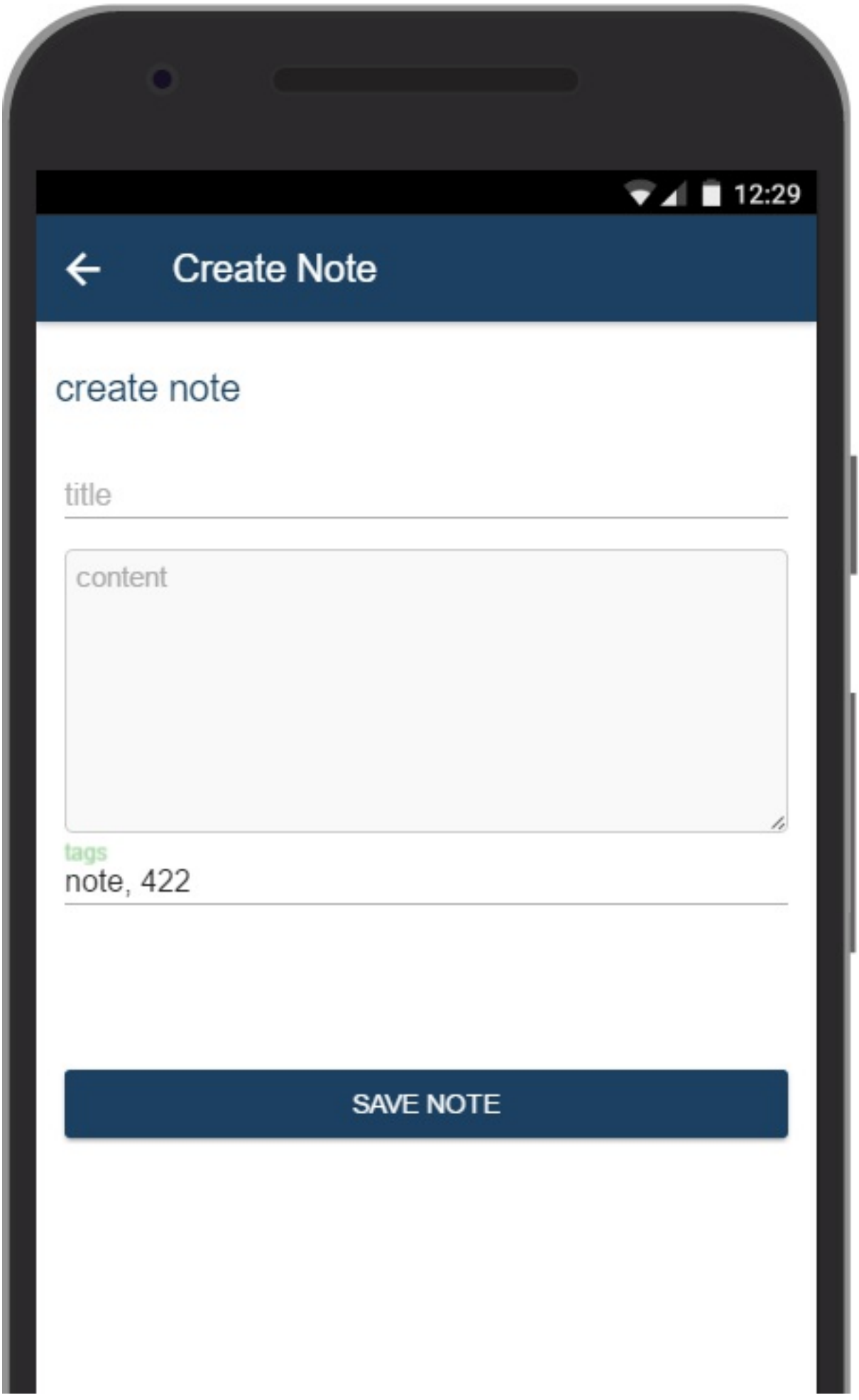
So, our form's structure will be as follows for this page,

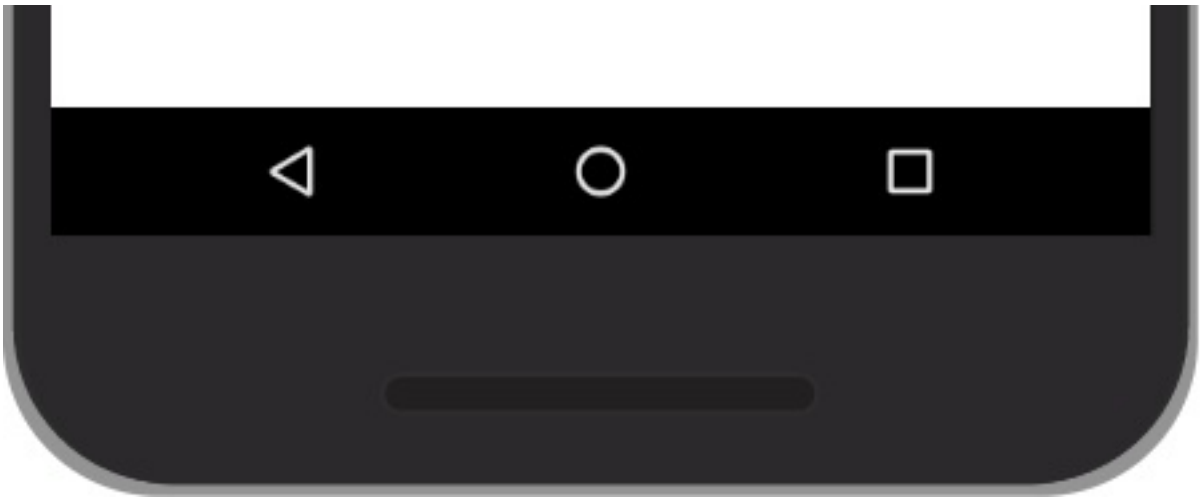
```
<section style="padding: 10px 0 10px 0">
  <ons-input type="text" placeholder="title" modifier="material"
  style="display: block; width: 100%"></ons-input>
</section>
<section style="padding: 5px 0 5px 0">
  <textarea class="textarea" placeholder="content" modifier="material"
  style="width: 100%; height: 150px; resize: vertical;"></textarea>
</section>
<section style="padding: 10px 0 10px 0">
  <ons-input type="text" placeholder="tags" modifier="material"
  style="width: 100%; height: 100px; resize: vertical;"></ons-input>
</section>
<section>
  <ons-button modifier="large">save note</ons-button>
</section>
```

We're using a mixture of standard HTML5 elements and recent OnsenUI components to get the desired layout and rendering for our create note page. Our design is now as follows,

Image - NoteTaker - create note page - OnsenUI

With the addition of material design modifiers, we get the placeholders showing as tips for each input field. As a user starts to type in a given input field, the specified value for the *placeholder* attribute is shown as a gentle reminder to the user. The user can also expand or contract the *textarea* in a restricted vertical direction.





OnsenUI - update UI - navigation and splitter structure

Our app's navigation is currently using the `<ons-navigator>` component to push the **create note** page to the navigation stack. We then use the back button to pop this page from the corresponding stack. However, this slightly restricts our navigation options as we need to provide a main menu for the app.

So, we need to update the navigation options to ensure that the navigator component works correctly with the main menu.

We can set this main menu using the `<ons-splitter>` component. This splitter component offers different frames that allow us to render varied content.

For example, we can add our menu with a left splitter, which contains the menu links. Each link will then load the requested link page url into the splitter content. These frames normally contain a `<ons-page>` component, but we can also nest multiple navigation components, if necessary.

Basic usage is as follows,

```
<ons-splitter>
  <ons-splitter-side id="menu" side="left" width="220px" collapse swipeable>
    <ons-page>
      <ons-list>
        ...
      </ons-list>
    </ons-page>
  </ons-splitter-side>
  <ons-splitter-content id="content" page="home.html"></ons-splitter-content>
</ons-splitter>
```

To combine the navigator and splitter component, we need to consider how they will complement each other to ensure that the navigation stack is respected correctly. i.e. that one navigation option does not override the other. In effect, we need to ensure that one navigation option does not block another, in particular with regard to the way OnsenUI uses promises for their navigation and animation objects.

So, the structure of our HTML needs to ensure that it respects the options for navigation relative to a single app. We can maintain our navigator component as the initial load option for the app, and then add our splitter component.

```
<ons-navigator id="navigator" page="splitter.html"></ons-navigator>
<ons-template id="splitter.html">
  <ons-splitter>
    <ons-splitter-side id="menu" side="left" width="220px" collapse swipeable>
      <ons-page id="menu.html">
        <ons-list>
          <ons-list-item url="home.html" class="menu-link" tappable>home</ons-list-
item>
          <ons-list-item url="about.html" class="menu-link" tappable>about</ons-list-
```

```
item>
        </ons-list>
      </ons-page>
    </ons-splitter-side>
    <ons-splitter-content id="content" page="home.html"></ons-splitter-content>
  </ons-splitter>
</ons-template>
```

As the app loads, the initial navigator component has been set to load the splitter template. As this template then loads, it starts with the menu page, and then renders the defined content, our home.html page.

The `<ons-splitter-content>` component now becomes the container for the link's pages. The navigation stack will be preserved correctly, and we can now build out our page structure using this general pattern.

OnsenUI - update UI - navigation and splitter promises

It's also interesting to note that this is slightly different from the prescribed pattern in the OnsenUI docs. After testing a pattern with the navigator component as a parent container to the splitter component, I found that a series of errors were continually being reported relative to blocked promises. In effect, the splitter and its associated animation was in conflict with subsequent calls to the menu itself, and the navigator component as well.

The outlined answer, https://community.onsen.io/topic/709/onsen-react-uncaught-in-promise-splitter-side-is-locked/2, was not satisfactory, so it led me to design and test this solution. Thankfully, it no longer blocks the required promises for the UI components.

OnsenUI - update UI - navigation and splitter logic

Whilst the structure of our navigation and splitter components is now set, we also need to update our app's JS logic to ensure that we can use both navigation options correctly with the navigation stack.

For the splitter component, we have a number of issues we need to consider to ensure that the menu and requested content is rendered correctly. So, we need to check when each page is loaded within the app, which then allows us to correctly load the menu, and provide the option to handle the **create note** button and navigation.

As the OnsenUi `init` event is fired for the first page, we can check that the initial page has indeed loaded correctly. Relative to our menu option, we add this check to force the logic to check the target page before loading the menu itself. If not, the execution of the JS logic will return 'null' for the menu open selector for a given page. e.g.

```
if (event.target.id === 'home') {
  //get menu icon - query selector OK due to one per ons page
        var menuOpen = document.querySelector('.menu-open');
        //check menu open is stored...
        if (menuOpen) {
          console.log("menu open stored...");
        }
}
```

We're simply checking that we can actually now use the menu open selector to toggle the state of the menu. We can then add an event listener for the main menu, which allows us to open the menu on any applicable ons page,

```
//add event listener for main menu
menuOpen.addEventListener('click', function(event) {
  event.preventDefault();
        //open main menu for current page
        menu.open();
}, false);
```

Then, we need to handle multiple possible links in the menu itself, and ensure that the requested page is loaded in the splitter content. e.g.

```
if (event.target.id === 'menu.html') {
  console.log("menu target...");
  //es6 Array.prototype.forEach iteration...
  Array.from(menuLink).forEach(link => {
    link.addEventListener('click', function(event) {
      event.preventDefault();
      var url = this.getAttribute('url');
      console.log("menu link = "+ url);
      content.load(url)
      .then(menu.close.bind(menu));
    }, false);
  });
}
```

So, this is an example of where our updated navigator and splitter component structure helps with the overall logic. We only check and add a listener for each menu item as and when the menu is actually loaded in the app. Otherwise, we can get the situation of locked promises for the splitter, which are then not resolved correctly.

This then allows us to correctly select our menu, and menu items, and also select the **create note** option as well on a given page. We can set this navigation stack for the **create note** option by checking it against a given page event, e.g.

```
if (event.target.id === 'home') {
        //set navigation
  onsNav(event.target);
}
```

For now, the `onsNav()` function remains the same as we've already seen.

OnsenUI - update UI - splitter, navigation, and backbutton

As we're using the `<ons-splitter>` and `<ons-navigator>` components to create the correct structure for our app, we also need to consider how each will interact with the standard hardware backbutton on Android.

For Android devices in general, it's useful to note that the default prescribed behaviour for this hardware backbutton is simply to close the app. A user can then reopen the app if necessary from the recent items using the overview button.

This behaviour pattern is replicated by Cordova, which fires an event to handle this hardware button within an app. It's part of the default `cordova.js` file.

OnsenUI also sets handlers for this hardware button for given components within the UI. e.g.

- `Dialogs` - this will close a cancelable dialog
- `Navigator` - as long as page stack is not empty, this will pop a page from navigation stack
- `Splitter` - this will close the menu if it is currently open

So, as we're working with the menu system based upon the splitter component, we have to be careful how we handle this hardware back button. If nothing else, whilst the default action is to simply exit an app we might find it useful to offer some sort of feedback to users before an **exit** is executed. e.g.

```
// initially disable hardware backbutton on Android
ons.disableDeviceBackButtonHandler();

// set custom backbutton handler
ons.setDefaultDeviceBackButtonListener(function(event) {
  ons.notification.confirm('Exit app?') // check with user
  .then(function(index) {
    if (index === 1) { // 'ok' button
      navigator.app.exitApp(); // default behaviour - exit app
    }
  });
});
```

Another option might simply be to maintain an in-app tracker for the pages pushed and popped relative to the splitter component. However, we'd still need to be mindful of the default, expected behaviour for Android. We shouldn't be changing this too much from user expectations and habits.

OnsenUI - update UI - splitter options & structure

We now have a working menu and navigation stack within our initial app.

There are many different ways to use this splitter option, which will often be informed by the page and navigation requirements of an application. For example, initially we can identify the following page and link requirements for our NoteTaker app,

- Main Menu
  - home
  - media
  - notes
  - tags
- navigation stack
  - create note
  - edit note
  - tag note
  - delete note
  - ...

So, we have a clear distinction between those links that require a standard menu on the page, and those that will use the navigation stack.

OnsenUI - load initial notes

We now need to add our initial notes for the app, load them as the app starts, and render them on the home screen.

For this, we'll be using IndexedDB for app based storage, including offline support. We can then save to a cloud based data store as and when a user requests saving a specific note.

So, as before, we'll add our initial check for IndexedDB support as part of the `deviceready` event.

```
//set variable for IndexedDB support
var indexedDBSupport = false;
if("indexedDB" in window) {
  indexedDBSupport = true;
  console.log("IndexedDB supported...");
} else {
  console.log("No support...");
}
```

We can create a simple variable to store the boolean result. We can then check this variable after the `deviceready` event has fired and returned successfully.

We're OK for Android, so we can continue to add IndexedDB data storage.

As you should recall from earlier, an IndexedDB database is local to the browser, and effectively only available to users of the local, native app. In effect, IndexedDB databases follow the same pattern of read and write privileges that we find for other browser-based storage options, including LocalStorage.

So, we could easily create databases with the same name, and then deploy them to different apps. Thankfully, they

will remain domain specific as well.

As we start to build and test our database, obviously the first thing we need to do is create an opening to our database.

```
var openDB = indexedDB.open("notetaker", 1);
```

We are creating a variable for our database connection, specifying the name of the DB and a version.

As we add IndexedDB support to our app, we can create our required DB, and then check it has persisted during subsequent application loading and usage.

Then, we can open a connection to the DB. We'll also need to add checks for three events, including upgrade, onsuccess, and any returned errors.

After connecting to the DB, upgrading our database, we are then ready to use the success event. So, we have our initial connection and upgrade for the DB, and then any output for the success event as part of the loading of the application itself.

From this point, we can start to build out our database for our NoteTaker app. We'll add the required initial **object stores**, or loose database tables as they're often known.

For this app, we'll also add our required **keypath**, and a useful index.

Our **keypaths** will help set unique identifiers for the data, and the index will naturally be particularly useful for the internal search option.

We'll update our upgrade event to include the creation of our required object stores.

```
...
openDB.onupgradeneeded = function(e) {
    console.log("DB upgrade...");
    //local var for db upgrade
    var upgradeDB = e.target.result;
    if (!upgradeDB.objectStoreNames.contains("ntos")) {
      upgradeDB.createObjectStore("ntos");
    }
}
...
```

We'll check a list of existing object stores looking for an existing example of the object store we need to create. As it's not there initially, we can simply create the required object store for our new notes.

So, our app is using the following pattern. As a user opens the app for the first time, the `upgradeneeded` event is run. This code checks for an existing object store, in this case our `ntos` object store. If it's not there, we create a new object store, and run the `success` handler. Subsequent app opening and loading will again check for the version number and, assuming it has not been upgraded, simply continue to load and run the application.

Image - check and load IndexedDB

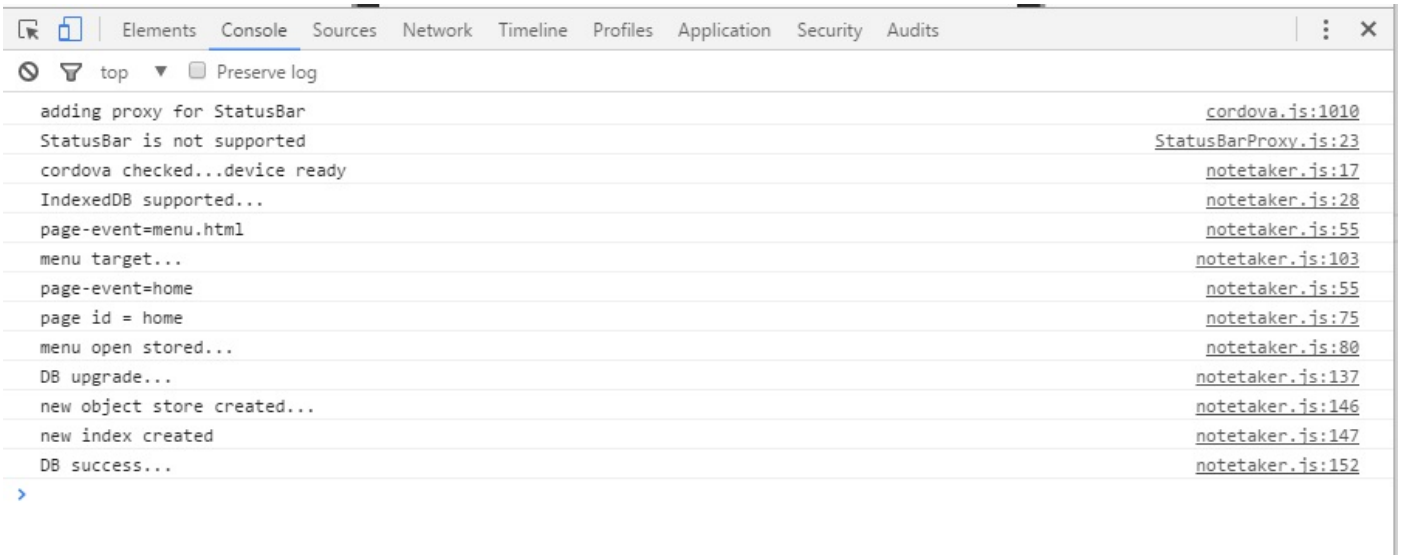
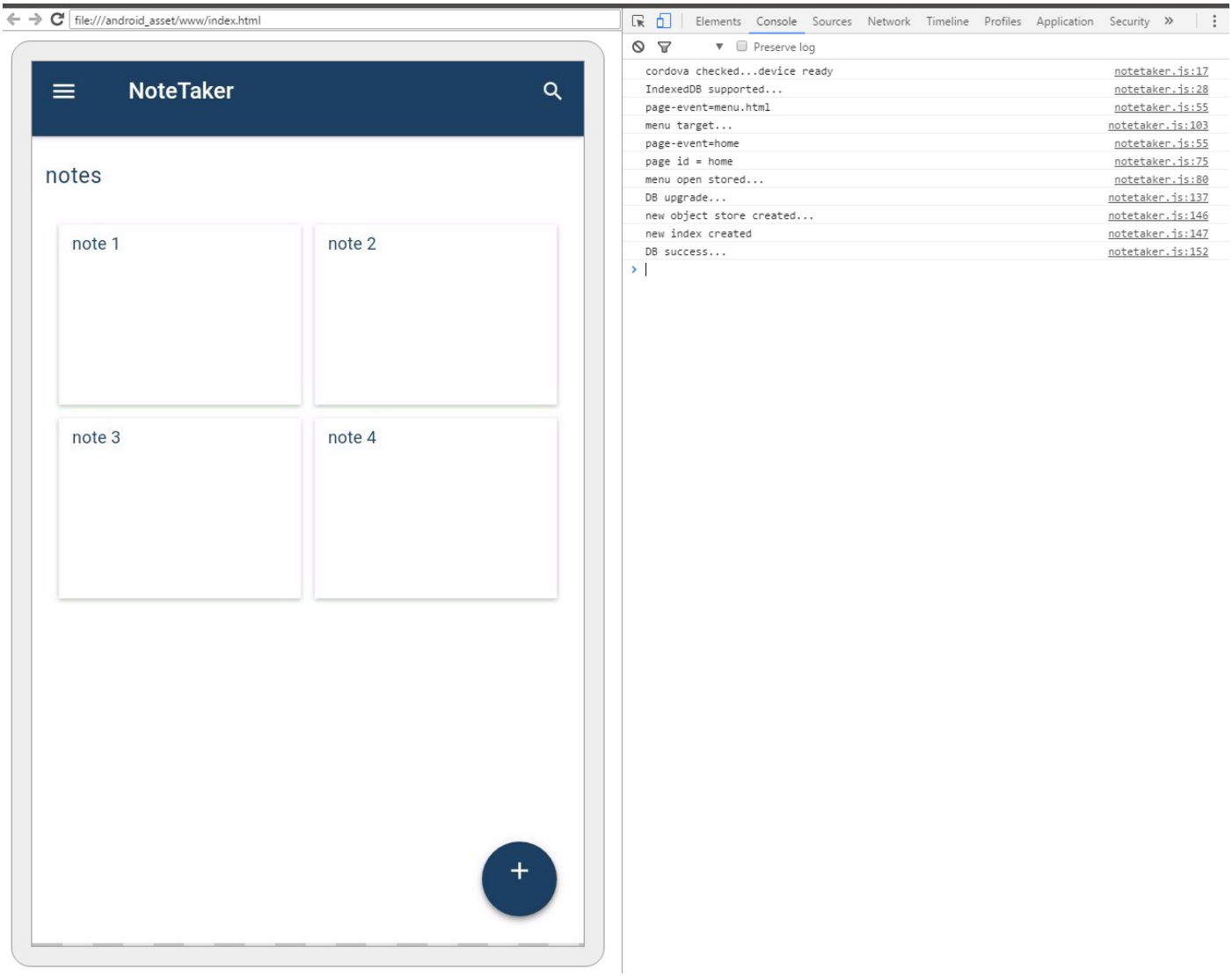


Image - check and load IndexedDB



With our database and object store now setup within our app, we can start to add some data for the initial notes.

IndexedDB allows us to simply store our objects in their default structure. So, in essence, we can simply store our JavaScript objects without modification in our IndexedDB database.

We'll now create the required **transactions**, which will act as a bridge between our app and the current database, thereby allowing us to add our data. For adding our data we'll use the `readwrite` operation on our previous object store, `ntos`.

```
var dbTransaction = openDB.transaction(["ntos"],"readwrite");
```

Once we have this transaction, we can use it to retrieve the object store we're going to use for our data.

```
var dataStore = dbTransaction.objectStore("ntos");
```

We'll set the schema for the note objects,

```
// note
var note = {
  title:title,
  note:note,
  tags:tags
}
// add note
var addRequest = dataStore.add(note,key);
```

We'll simply set the schema to match the input fields we defined for the **create note** form.

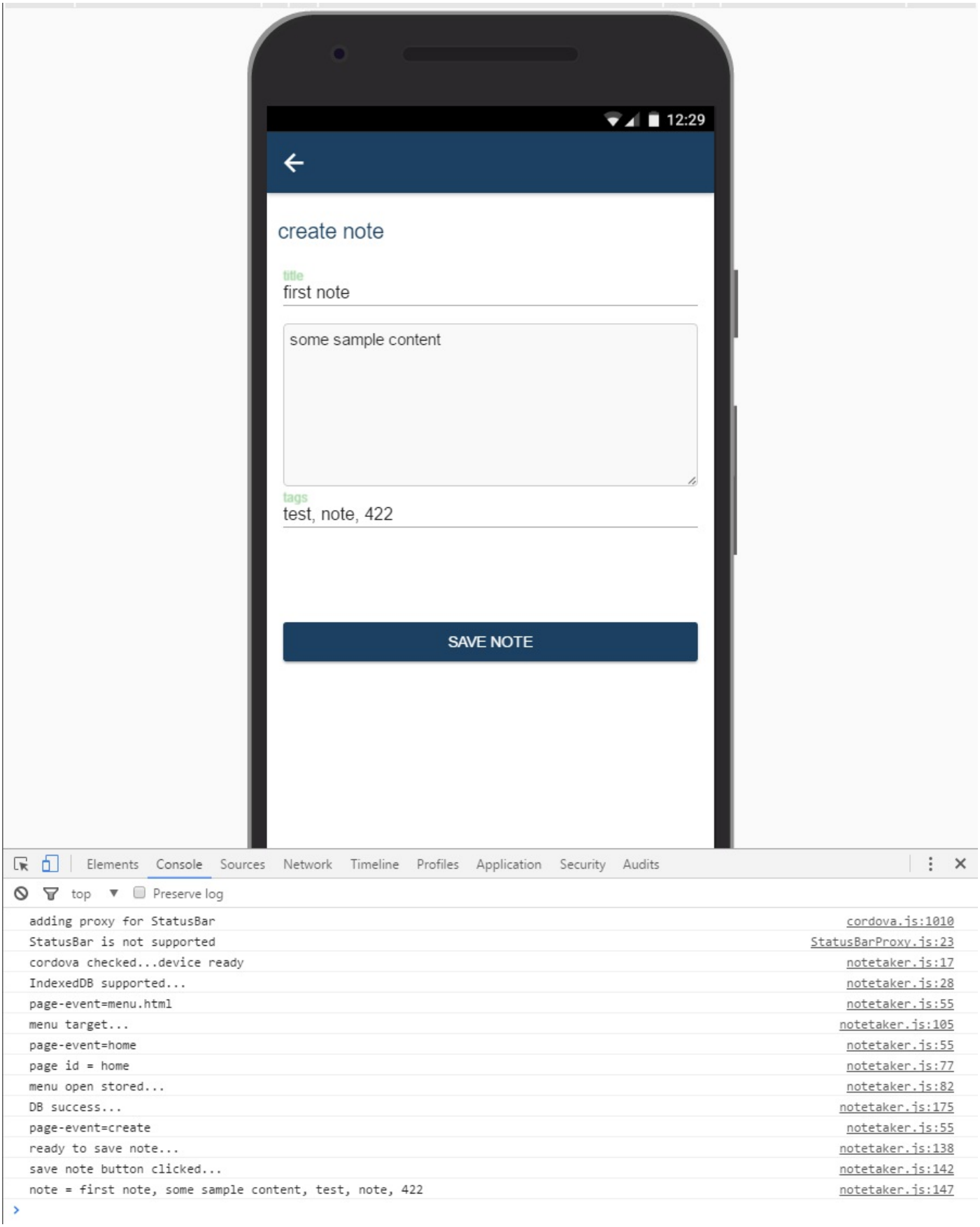
We'll also add a handler for the **create note** form. We'll check for the input values, which are then passed to a function to be saved within our database.

```
function createNote(page) {
        //note save handler - check create note page is active...
        if (page.id === "create") {
                console.log("ready to save note...");
                document.getElementById('noteSave').addEventListener('click',
function(event) {
                        //prevent any bound defaults
                        event.preventDefault();
                        console.log("save note button clicked...");
                        //get values for note - title, content, tags
                        var noteTitle = document.getElementById('noteTitle').value;
                        var noteContent =
document.getElementById('noteContent').value;
                        var noteTags = document.getElementById('noteTags').value;
                        console.log("note = "+noteTitle+", "+noteContent+",
"+noteTags);
                });
        }
}
```

As we're using page events within the app, we need to check that the **create note** page is active, available in the DOM, before we can start to add listeners for events. If not, then we'll simply get an error thrown for the **notesave** element. Obviously, it needs to be in the DOM before we can attach a listener.

We can then get the values for the input fields as a user clicks the save button. In a production app, we would also be expected to check the validity of each input before submission of the form.

Image - check and load IndexedDB



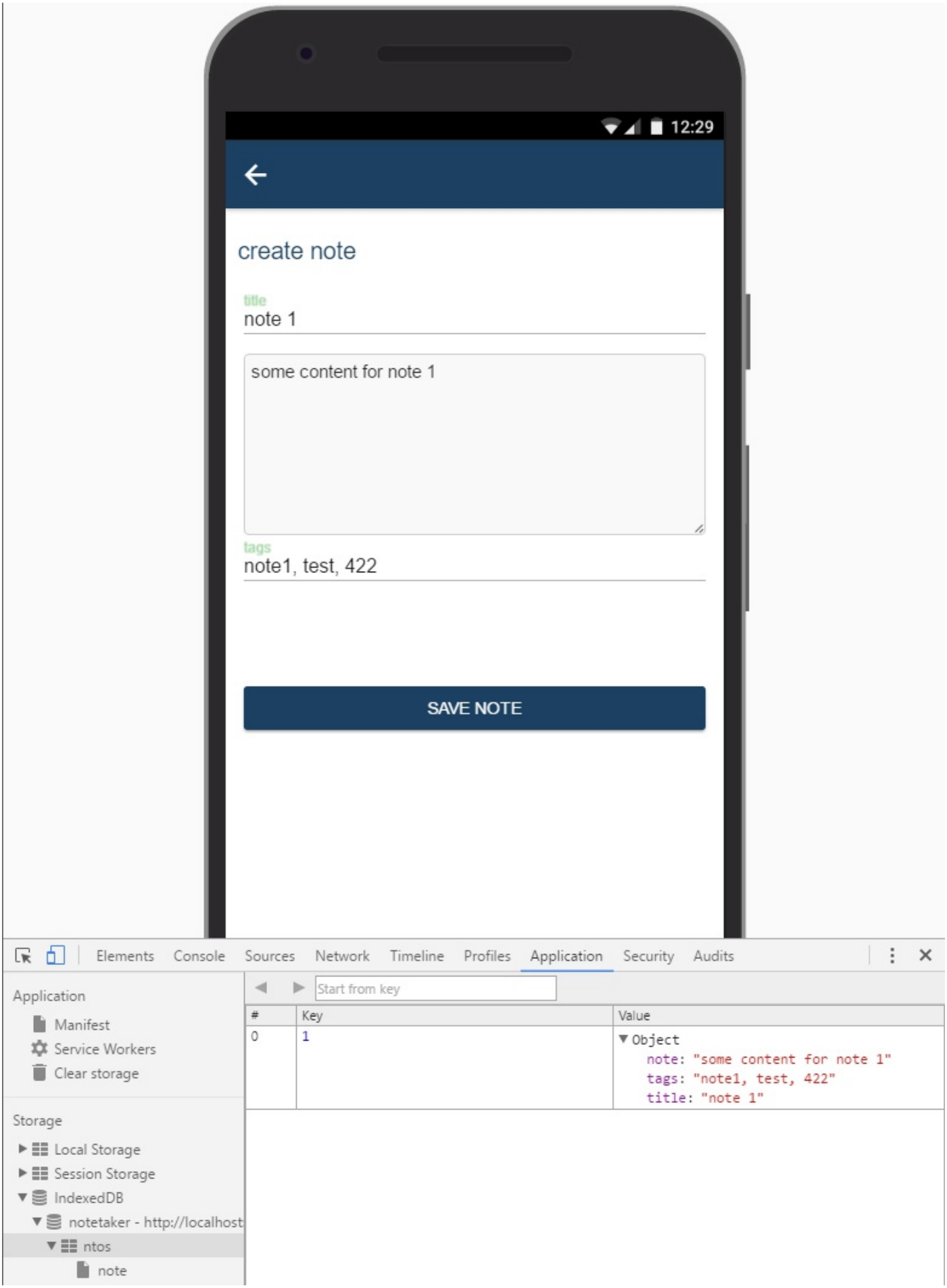
We can now add our `saveNote()` function, which will store the values for our note in the specified object store in the database,

```
//save note data to indexeddb
function saveNote(title, content, tags){
        //define a note
        var note = {
                title:title,
          note:content,
```

```
                tags:tags
        }
   // create transaction
        var dbTransaction = db.transaction(["ntos"],"readwrite");
        // define data object store
        var dataStore = dbTransaction.objectStore("ntos");
        // add data to store
        var addRequest = dataStore.add(note);
        // success handler
        addRequest.onsuccess = function(e) {
          console.log("data stored...");
          // do something...
        }
        // error handler
        addRequest.onerror = function(e) {
    console.log(e.target.error.name);
          // handle error...
        }
}
```

For our initial save, we get the following result,

Image - check and load IndexedDB



Then, we need to decide what to do with the tags, the success handler, and rendering of the new note.

For example, as part of the validation for the **tags** input field, we can control the structure of the input text for the tags. So, we might then extract the individual tags from a comma separated list or simply by spaces. There are inherent benefits and issues with each option. To split each tag from our string we can use the JavaScript function `split()`,

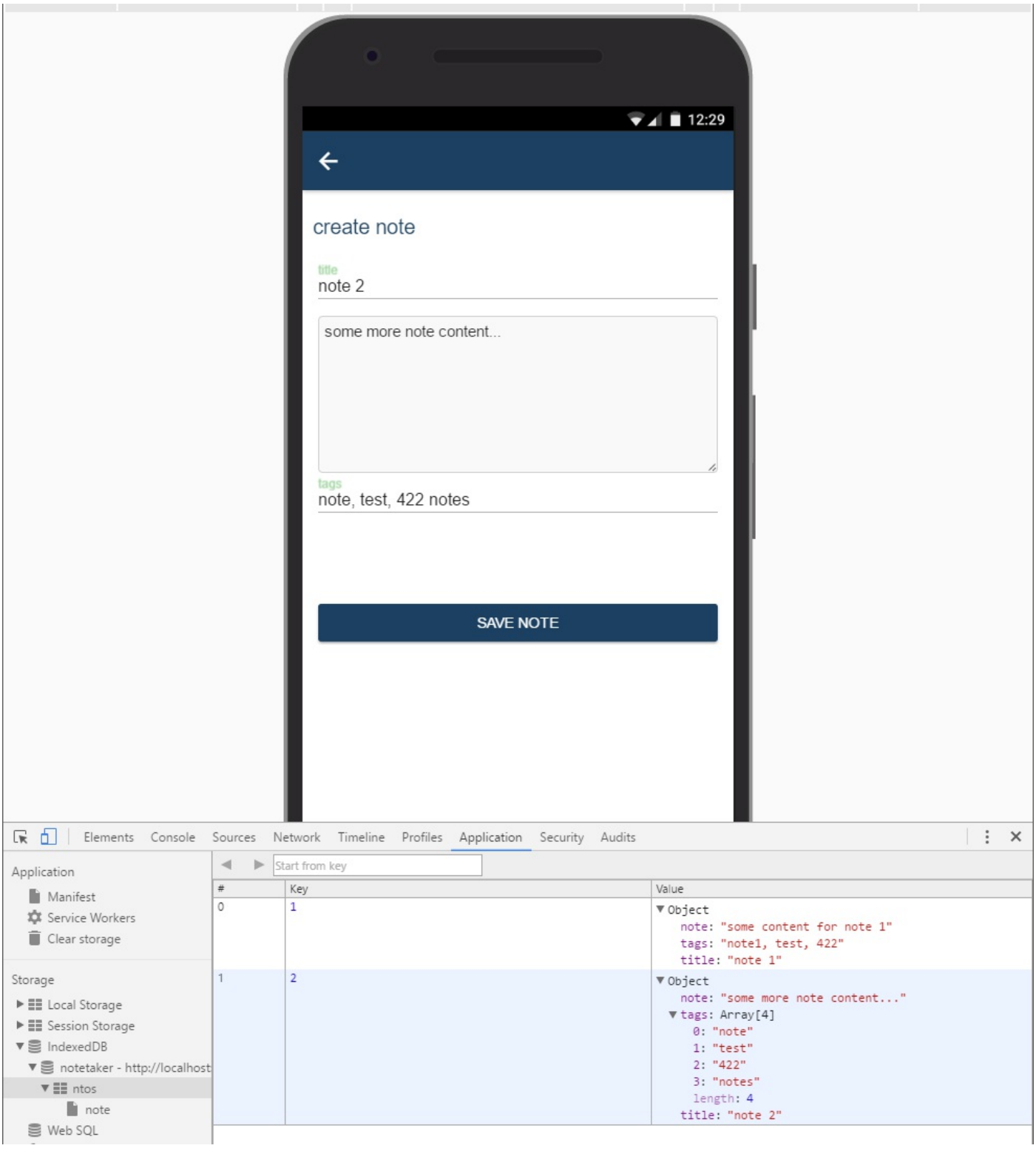
and then combine it with a standard regular expression. e.g.

```
...
tags.split(/[ ,]+/);
...
```

So, we can now split our tags based on a standard sequence of one or more commas or spaces. If will also avoid outputting multiple consecutive spaces or a potential comma and space sequence. In effect, we're trying to avoid producing empty results for our tags.

We can now return the following, and then store in the DB.

Image - check and load IndexedDB



Next update for our app, we need to consider how to handle the success event for saving our note data in our database's object store.

We might simply return a user to the **create note** form, after showing a notification &c. to provide feedback for saving the note. Or, we might return the user to the home page with the new note rendered with a feedback message. There are many different options, but the common factors include the following,

- feedback to a user to inform them whether the note was successfully saved or not
- consistent rendering of the notification, buttons, location...

Likewise, we also need to consider how to handle the return for an error message if we're unable to save a note &c.

For our current app, we'll add the following pattern once a user has successfully saved a note in the database.

- show notification with two options
  - **return to notes** and **create new note**

As such, we'll then need to handle the following for each option.

For the first option, **return to notes**, we'll listen for the button's click event, then dismiss the notification, pop the **create note** page from the navigation stack, and return to the home page.

For the second option,**create new note**, we would again listen for the button's click event, dismiss the notification, and reset the form fields.

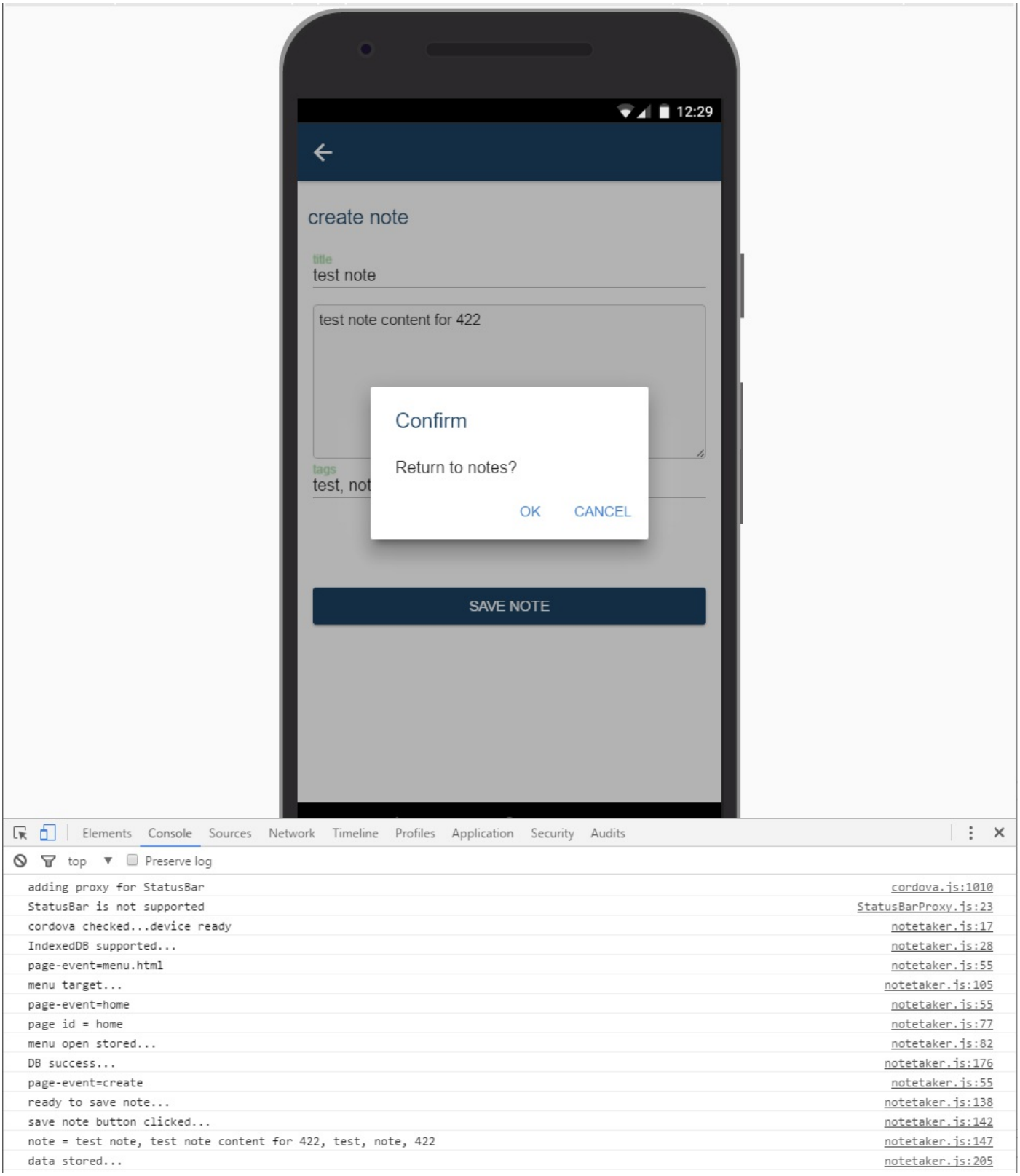
We'll now add the first option, **return to notes**, after a user has successfully saved a new note. We'll need to update the `saveNote()` function, in particular for the success event.

```
...
// success
addRequest.onsuccess = function(e) {
  console.log("data stored...");
  //update user on note stored
  ons.notification.confirm('Return to notes?') // check with user
  .then(function(index) {
    if (index === 1) { // 'ok' button
        document.querySelector('#navigator').popPage(); // return to previous page
    }
  });
}
...
```

We can add a notification to Onsen's `ons` object, and define it as a confirm notification. We'll simply check with the user whether they wish to return to their notes, the home page in this app, or cancel, which will make the **create note** page active again.

So, a user can now create a new note, save it to the app's database, and then either continue with additional notes or simply return to the home page.

Image - save note success



However, we also need to consider how to handle the current **create note** page, assuming a user selects the **cancel** option in the confirmation window.

The user will be returned to the current page in the navigator stack, our **create note** page. However, when they return to this page, the input fields will still show the previous entry data for our last note.

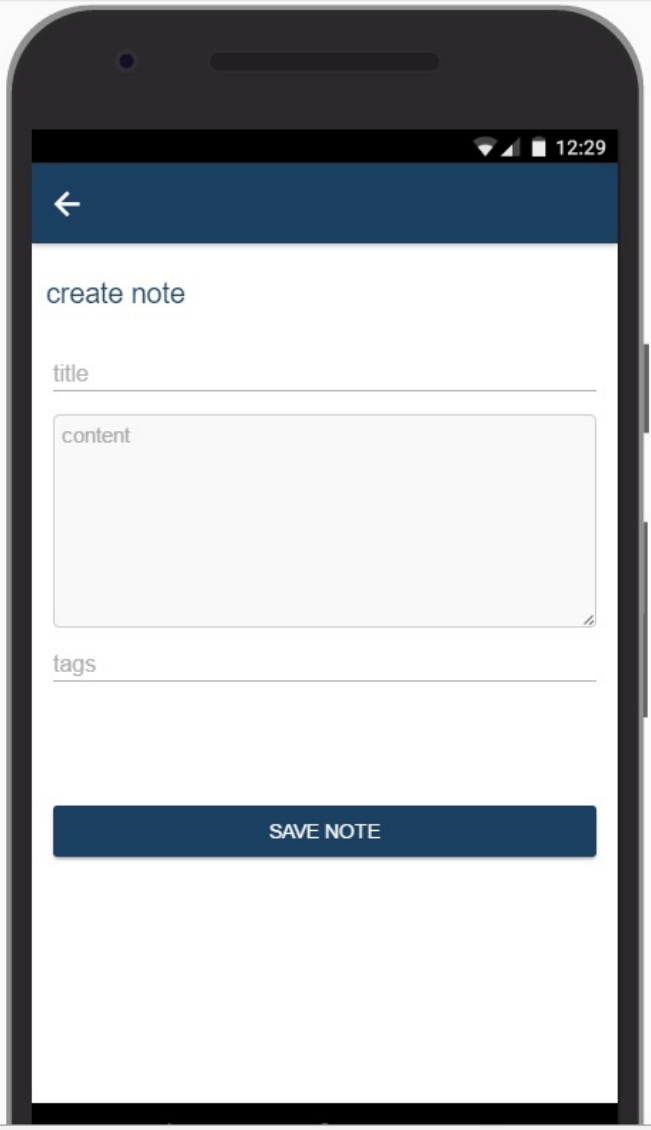
Programmatically, we need to ensure that these input fields are cleared, which will then allow a user to create another new note. There are a number of ways we could achieve this result, for example emptying each field using plain JavaScript, but we'll work with the existing OnsenUI navigator object instead.

We can use this option to ensure that the top page on the navigator stack is still the **create note** page, and effectively refresh this page for our user.

```
//update user on note stored
ons.notification.confirm('Return to notes?') // check with user
.then(function(index) {
        if (index === 1) { // 'ok' button
                document.querySelector('#navigator').popPage(); // return to previous
page
  } else if (index === 0) { // check 'cancel' button
                document.querySelector('#navigator').replacePage('create.html',
{'animation': 'none'});
        }
});
```

We can replace the current top page, and set the animation for this event to none to ensure that the refresh does not jar the UI rendering for the app.

Image - save note success



Elements Console Sources Network Timeline Profiles Application Security Audits

top Preserve log

```
adding proxy for StatusBar                                              cordova.js:1010
StatusBar is not supported                                              StatusBarProxy.js:23
cordova checked...device ready                                          notetaker.js:17
IndexedDB supported...                                                  notetaker.js:28
page-event=menu.html                                                    notetaker.js:55
menu target...                                                          notetaker.js:105
page-event=home                                                         notetaker.js:55
page id = home                                                          notetaker.js:77
menu open stored...                                                     notetaker.js:82
DB success...                                                           notetaker.js:176
page-event=create                                                       notetaker.js:55
ready to save note...                                                   notetaker.js:138
save note button clicked...                                             notetaker.js:142
note = test note, some content for our test note, test, note, 422, refresh  notetaker.js:147
data stored...                                                          notetaker.js:205
page-event=create                                                       notetaker.js:55
ready to save note...                                                   notetaker.js:138
>
```