Cross-platform development with JavaScript

20 February 2017 | **Insight Zuhlke, Software Engineering** | **Katharina Bähr**
Reading time: 12 minutes

In the last blog post I give an overview about the technologies used for my cross-platform development experiment. Now I will show you the results of the demo applications:

Let the magic happen

In the following examples, I will show you how you can develop a cross-platform application with the chosen stack. The example application will show a list of persons. You can select one by clicking on a list item. After selecting you will see that person´s profile. This list and detail view is conceived like a master-detail interface. On the list will be a kind of infinite scroll mechanism, so you can scroll down the list and it will dynamically load more data from the web service. Sometimes I will only show the interesting parts and no CSS at all.

These are a lot of technologies but they are very alike and we will share a lot of code through modularization and reuse. Apart from this, most code will be for bootstrapping and wiring everything together.

Node.js web service

We will start with the Node.js web service so the applications are able to consume some data. We will use `express` because it provides a nice way of handling request handlers.

For installing required npm packages, we need to execute on the console:

```
npm install --save express body-parser express-serve-static-core
serve-static
```

and for installing typings with typings (`npm install typings`):

```
typings install --global express body-parser express-serve-static-core
serve-static
```

For the web service, we are creating a `Server` class and a method for bootstrapping. In the `constructor`, we define some API methods which will handle incoming requests.

```
class Server
    app: express.Application;
    static bootstrap(): Server {
        return new Server();
    }
    constructor() {
        this.app = express();
        this.app.all('*', (req: express.Request, res:
express.Response, next: express.NextFunction) => {
            res.header('Access-Control-Allow-Origin', '*');
            res.header('Access-Control-Allow-Methods', 'GET, POST,
OPTIONS, PUT, PATCH, DELETE');
            res.header('Access-Control-Allow-Headers', 'X-Requested-
With, content-type, Origin, Accept');
            next();
        });

        this.app.get("/persons", (req: express.Request, res:
express.Response) => {
            let start: number = Number(req.query.start);
            let limit: number = Number(req.query.limit);
            let response: Array<Person> = (start && limit) ?
personlist.slice(start, start + limit) : personlist;
            res.send(response);
        });

        this.app.get("/persons/:name", (req: express.Request, res:
express.Response) => {
            let name = req.params.name;
            personlist.forEach((person: Person) => { //personlist is
an array with some properties (not shown here)
                if (person.name === name) {
                    res.send(person);
                    return;
                }
            });
        });
        this.app.listen(3000, () => {
            console.log("Listening on port 3000...");
        });
    }
}
```

The only thing left now is to start the web service by calling `Server.bootstrap();` This web service will now listen on port 3000 and return data when requested.

Reusable code as npm module

Next, I continue with the `person-service` which I will publish on npm so we can share it across the platforms. Yes, I could also just create some module and reuse it. Using npm is just because we can and because it´s pretty easy. This could make sense in a larger project when different teams are working on different pieces of code or platforms, so you could easy modularize and manage it through small packages (probably using private npm packages or self-hosted).

The web service uses the new fetch standard which is not integrated into TypeScript, so you need to install a polyfill with

```
npm install whatwg-fetch --save
```

First I create the class `PersonService` which will expose methods for fetching data from the published web service.

```
export class PersonService {
  getPersons(start: number, limit: number): Promise<Array<Person>>{
    return
fetch('http://js-all-persons.azurewebsites.net/persons?start=' + start
+ '&limit=' + limit, {
      method: 'GET',
      headers: { 'Content-Type': 'application/json' }
    }).then((response: Response) => {
      return this.handleResponse(response);
    }).then((result: Array<Person>) => {
      return result;
    }, (e: Error) => {
      console.log("error", e);
      return null;
    });
  }

  getPersonDetails(name: string): Promise<Person> {
    return fetch('http://js-all-persons.azurewebsites.net/persons/' +
name, {
      method: 'GET',
      headers: { 'Content-Type': 'application/json' }
    }).then((response: Response) => {
      return this.handleResponse(response);
    }).then((result: Person) => {
      return result;
    }, (e: Error) => {
      console.log("error", e);
      return null;
    });
  }

  private handleResponse(response: Response): Promise<any> {
    if (String(response.type) === 'opaque') {
      return null;
    }
    if (response.status !== 200) {
      console.log('Problem occured. Status Code: ' , response.status);
      return null;
    }
    return response.json();
  }
}
```

We now have a method for a range of data and one for person details. You can easily imagine how we could also create methods or classes for business logic, data transitions or maybe even view models. I put the TypeScript definition file into the module itself in order to access the data typed.

Aurelia web app

Aurelia is a single-page application framework which comes with a lot of bootstrapping, tooling code and dependencies like all the other MV* Frameworks nowadays. Because I am using one of their skeletons, all we have to do to get started is to install all required packages with

```
npm install
```

```
jspm install
```

```
typings install
```

and for installing the published `person-service`:

```
npm install person-service
```

After everything is properly installed, we can create a `people.html` and `people.ts`. Because we have only one route I start straight with the only view that will contain a list of people on the left and the details for a person, on the right.

```
<template>
  <require from='../components/list-item/list-item.html'></require>
  <require from='../components/scroll-end/scroll-end'></require>
  <main class="content">
    <div class="col-sm-5 col-xs-12">
      <div class="person-list" scroll-end="load.call:loadPersons()">
        <div repeat.for="person of persons"
click.delegate="showDetails(person)" class.bind="person.name ===
selectedPerson.name ? 'person selected' : 'person'">
          <list-item person.bind="person"></list-item>
        </div>
      </div>
    </div>
    <div class="col-sm-7 col-xs-12 details" if.bind="selectedPerson">
      <div>
        <h2>${selectedPerson.Title}</h2>
        <img src.bind="selectedPerson.picture" alt="person image"
if.bind="selectedPerson"/>
        <div class="descr-header">Profile</div>
          <div class="detail-list">
            <div class="detail-item col-xs-6">
              <div>
                <span class="title">Balance</span>
                <span class="value">${selectedPerson.balance}</span>
              </div>
            <div>
              <span class="title">Eye Color</span>
              <span class="value">${selectedPerson.eyeColor}</span>
            </div>
            ...
          </div>
        </div>
      </div>
    </div>
  </main>
</template>
```

This HTML contains a `list-item` custom element and one custom attribute the `scroll-`

`end` element. The `scroll-end` element checks if you already reached the end of the list and then dynamically loads more data. The `list-item` is an HTML-only component and just encapsulated some recurring HTML code.

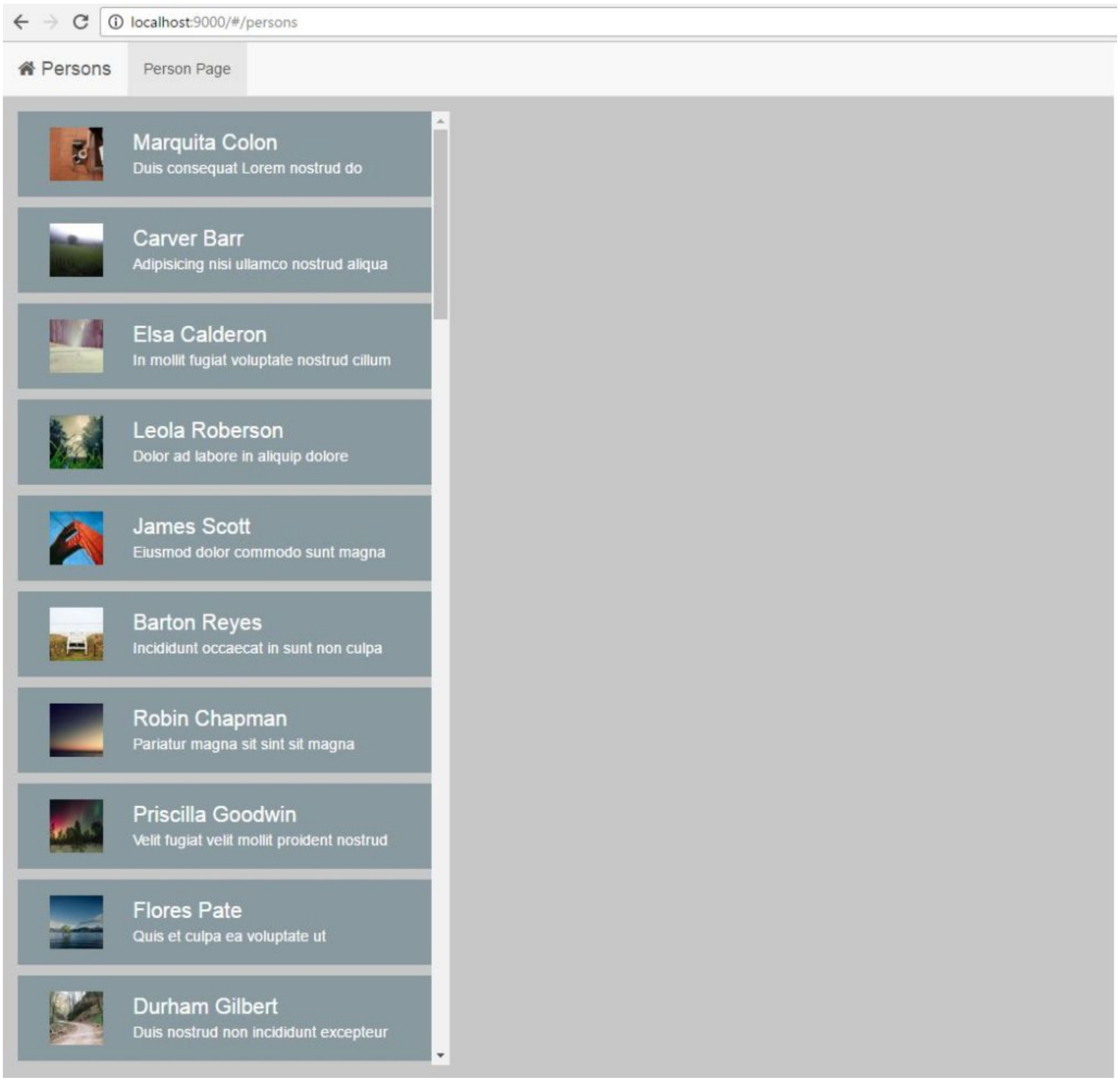
To get some life into the view we require some data from the `person-service` in the related `Persons` class:

```
import { autoinject } from 'aurelia-framework';
import { HttpResponseMessage } from 'aurelia-http-client';
import { PersonService, Person } from 'person-service'; //importing
the PersonService (our npm module) and the typings for Person

@autoinject()
export class Persons {
  persons: Array<Person> = new Array<Person>();
  selectedPerson: Person;
  itemRange: number = 20; //start with 20 items
  itemStart: number = 0;

  constructor(private personService: PersonService) {
  }

  activate() {
    this.loadPersons();
  }

  loadPersons() {
    this.personService.getPersons(this.itemStart,
this.itemRange).then((data: Array<Person>) => {
      this.persons = this.persons.concat(data);
      this.itemStart += 10;
    }, error => {
      console.log("Error loading persons.", error)
    });
  }

  showDetails(choosePerson: Person) {
this.personService.getPersonDetails(choosePerson["name"]).then((data:
Person) => {
      this.selectedPerson = data;
    }, error => {
      console.log("Error loading persons.", error)
    });
  }
}
```

The used Aurelia skeleton comes with gulp so we only have to execute:

```
gulp serve
```

and the application is running transpiled in a local web server and reachable over `localhost:9000`.



localhost:9000/#/persons

Marquita Colon
Duis consequat Lorem nostrud do

Carver Barr
Adipisicing nisi ullamco nostrud aliqua

Elsa Calderon
In mollit fugiat voluptate nostrud cillum

Leola Roberson
Dolor ad labore in aliquip dolore

James Scott
Eiusmod dolor commodo sunt magna

Barton Reyes
Incididunt occaecat in sunt non culpa

Robin Chapman
Pariatur magna sit sint sit magna

Priscilla Goodwin
Velit fugiat velit mollit proident nostrud

Flores Pate
Quis et culpa ea voluptate ut

Durham Gilbert
Duis nostrud non incididunt excepteur

Electron desktop app

With Electron you can wrap web applications into desktop applications. For this, Electron uses the chromium web driver and Node.js and offers you a lot of packages to communicate with the underlying system.

To getting started you need to download these packages

```
npm install electron -g
```

for prebuilding the desktop app and

```
npm install electron-packager -g
```

for packaging the app into a specific platform executable.

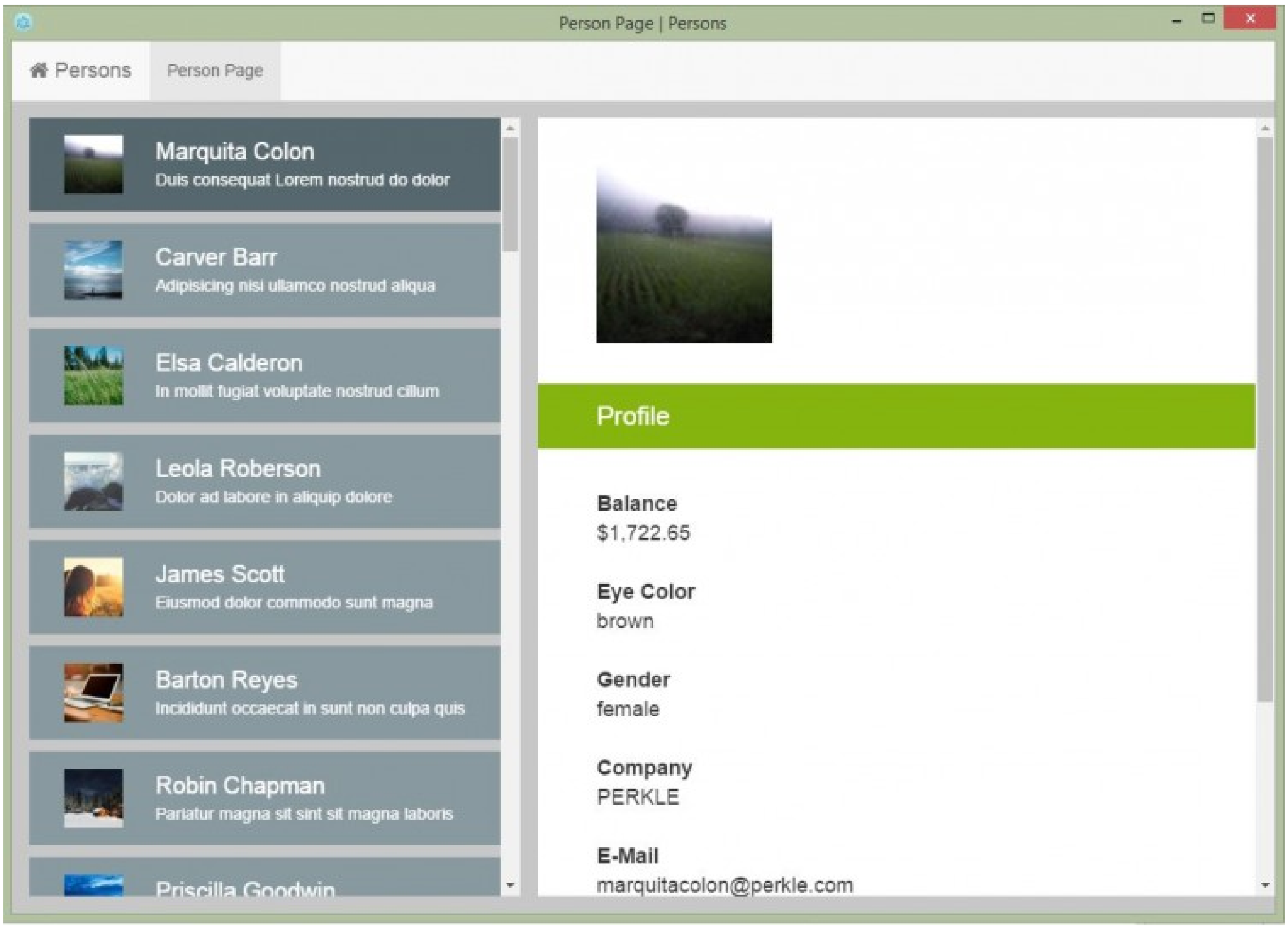
Next, you have to create a commonly named `main.js` startup file that looks like this:

```
'use strict';
const electron = require('electron');
const {app, BrowserWindow} = electron;
let mainWindow;

mainWindow.setMenu(null);
mainWindow.maximize();

app.on('window-all-closed', () => {
    if (process.platform !== 'darwin') {
        app.quit();
    }
});

app.on('ready', () => {
    mainWindow = new BrowserWindow({
    width: 800,
    height: 600
});

mainWindow.loadURL(`file://${__dirname}/index.html`); //define app
entry point
    mainWindow.webContents.on('did-finish-load', () => {
    mainWindow.setTitle(app.getName());
});

mainWindow.on('closed', () => {
    mainWindow = null;
});
```

You need to add a reference to this `main.js` file into your `package.json` so the electron command knows where to start.

```
{
    ...
    "main": "main.js"
    ...
}
```

That´s all configuration needed.



Marquita Colon
Duis consequat Lorem nostrud do dolor

Carver Barr
Adipisicing nisi ullamco nostrud aliqua

Elsa Calderon
In mollit fugiat voluptate nostrud cillum

Leola Roberson
Dolor ad labore in aliquip dolore

James Scott
Eiusmod dolor commodo sunt magna

Barton Reyes
Incididunt occaecat in sunt non culpa quis

Robin Chapman
Pariatur magna sit sint sit magna laboris

Priscilla Goodwin

Balance
$1,722.65

Eye Color
brown

Gender
female

Company
PERKLE

E-Mail
marquitacolon@perkle.com

The Aurelia TypeScript skeleton that I used, already includes an `index.js` and a `README.md` section about Electron. So you just need to add the provided `index.js` into your `package.json` and execute everything with

```
electron .
```

or build it with :

```
electron-packager . 'person-electron-app' --platform win32 --version
1.3.8
```

for needed platform and project name/folder. Electron is not only good for wrapping a web app into an executable. Because it´s not a sand-boxed web browser you are able to access the filesystem, user shell and more, so you can write real desktop applications with it.

NativeScript app

NativeScript uses reflection to access the native device API´s and additionally offers a lot of wrappers around them. NativeScript comes with a mighty CLI integration similar to the Cordova one. To work with this CLI you firstly need to install it with the command:

```
npm install -g nativescript
```

dependent on your platform you have to install the environments and some tools for building Android and iOS apps. Happily NativeScript helps you with this, you can read more about it in the official install guide. After everything is properly installed and your platforms are added, you can create your project with the command

```
tns create PersonApp --template tns-template-master-detail-ts
```

There are a lot of templates for NativeScript, so you don´t need to waste time on the project setup and can just start coding. Here I used a template for a master-detail view with TypeScript. The `app/app.ts` inside the project is the first executed file, here we have to reference the starting view. I renamed the files and put them into folders so the `app.ts` looks like this now

```
import * as application from "application";
application.cssFile = "app.css";
application.mainModule = "person-list/person-list";
application.start();
```

I installed the person-service from npm and added methods for page load, item tapping and routing into the `person.ts`.

```
...
import { PersonService, Person } from "person-service";
```

```
let personService: PersonService = new PersonService();
let persons: ObservableArray =  new ObservableArray([]);
let itemRange: number = 20;
let itemStart: number = 0;

let twoPaneLayout = Math.min(platform.screen.mainScreen.widthDIPs,
platform.screen.mainScreen.heightDIPs) > 600;

export function loaded(args: EventData) {
    let pageData = new Observable({
        persons: persons
    });

    loadPersons();
    (args.object).bindingContext = pageData;
}

export function loadPersons(args?: EventData) {
    personService.getPersons(itemStart, itemRange).then((data:Array)
=> {
        persons.push(data);
            itemStart += 10;
        }, error => {
            console.log("Error loading Persons." , error)
        }
    );
}

export function showDetails(args: ItemEventData) {
    let selectedPerson: Observable = new Observable();
    let navigationEntry = {
        moduleName: "details/details-page",
        context: selectedPerson
    };

personService.getPersonDetails(args.view.bindingContext.name).then((da
ta: Person) => {
        selectedPerson.set('selectedPerson', data);
        if (!twoPaneLayout) {
            frames.topmost().navigate(navigationEntry);
        }
    }, error => {
        console.log("Error loading persons.", error)
    });
};
```

Everything you set on the `bindingContext` will be available in the corresponding view. In the code above I created the object `pageData`. We can push items to the `persons` `Array` at any time and because the `pageData` object is an `Observable` the view will update itself with the new data. The view is written in an XML syntax similar to how you would define the view in native iOS or Android.

```
<Page loaded="loaded" class="content">
    <ActionBar title="Persons" class="page-title">
        <StackLayout orientation="horizontal"
ios:horizontalAlignment="center" android:horizontalAlignment="left">
            <Label text="Persons"/>
        </StackLayout>
    </ActionBar>

    <StackLayout class="article-list">
        <!-- here we register the tab handler -->
        <ListView items="{{ articles }}" itemTap="listViewItemTap">
            <ListView.itemTemplate>
                <StackLayout orientation="horizontal" class="article">
                <StackLayout orientation="vertical">
                     <Image src="{{ picture }}" class="img"/>
                </StackLayout>
                <StackLayout orientation="vertical" class="descr">
                    <Label text="{{ name }}" textWrap="true"
class="title-name" />
                    <Label text="{{ about }}" textWrap="true"/>
                </StackLayout>
                </StackLayout>
            </ListView.itemTemplate>
        </ListView>
    </StackLayout>
</Page>
```

For the case that we are on a device with a width larger than 600px and want to show the master and the detail view next to each other, we can define XML views with a specific name schema. So I create a second view with the name `person-list.minWH600.xml` which looks like this:

```
<Page loaded="loaded" class="content">
    <GridLayout rows="auto, *" columns="300, *">
        <ActionBar title="Persons" class="page-title">
        ...
        </ActionBar>

        <StackLayout class="article-list">
        ...
        </StackLayout>

        //if an item got selected, include the details view here
        <GridLayout row="1" col="1" bindingContext="{{ selectedItem
}}">
            <app:details-view/>
        </GridLayout>
    </GridLayout>
</Page>
```

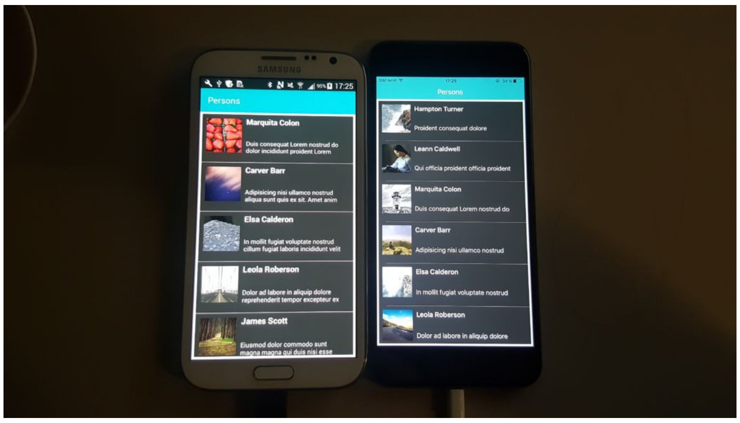
As you can see we used a lot of CSS classes. We are able to style the components with basic CSS (NativeScript only supports a subset) so we can reuse a lot of rules from the web app. For really writing the CSS just once, the best approach probably would be to start with the app CSS and think at first about how to slice the styles. For this example, I put all the style definitions into the `app.css`. The corresponding `details-page` looks pretty simple, because we are just listening to the navigate event and setting the page context.

```
import { EventData } from "data/observable";
import { Page } from "ui/page";
import * as scrollViewModule from "ui/scroll-view";

export function pageNavigatedTo(args: EventData) {
    var page = <Page>args.object;
    page.bindingContext = page.navigationContext;
}
```

For the view we can reuse some of the structure and just replace the `div`´s and `span`´s through `StackLayout`´s and `Labels.`

```
<Page navigatedTo="pageNavigatedTo">
    <ActionBar title="Details" class="page-title">
        <NavigationButton />
        <StackLayout orientation="horizontal"
ios:horizontalAlignment="center" android:horizontalAlignment="left">
            <Label text="Details"/>
        </StackLayout>
    </ActionBar>
    <ScrollView>
        <StackLayout class="detail-list">
            <Label text="{{ name }}" class="descr-name"/>
            <StackLayout orientation="vertical">
                <Image src="{{ picture }}" class="detail-img"/>
            </StackLayout>

            <Label text="Profile" class="descr-header"/>

            <StackLayout class="detail-item">
                <Label text="Balance" class="title"/>
                <Label text="{{ about }}" class="value"/>
            </StackLayout>

            <StackLayout class="detail-item">
                <Label text="Eye Color" class="title"/>
                <Label text="{{ eyeColor }}" class="value"/>
            </StackLayout>
            ...
        </StackLayout>
    </ScrollView>
</Page>
```

we can run the app with several commands, I used `tns livesync android` which is pretty cool because it reloads your app as soon as you save your changes in the editor.



The time will come

Okay, we are not there yet, this is not the bright future you may have imagined. Granted, transforming a web application to a desktop one with only some additional code is really awesome. The code we had to write for all different platforms was not much, but the code we were able to reuse, especially with Aurelia and NativeScript, was not that much too. I had to nearly rewrite the whole view and reusing CSS also works not just out of the box. These little examples demonstrate how much you can achieve with a bit of JavaScript and how easy you can support several platforms. I am already convinced that these frameworks, JavaScript itself and the possibilities regarding cross-platform development will improve from day to day until we are finally there and that JavaScript is a real player for cross-platform development.