IONIC 6

QUICKSTART GUIDE

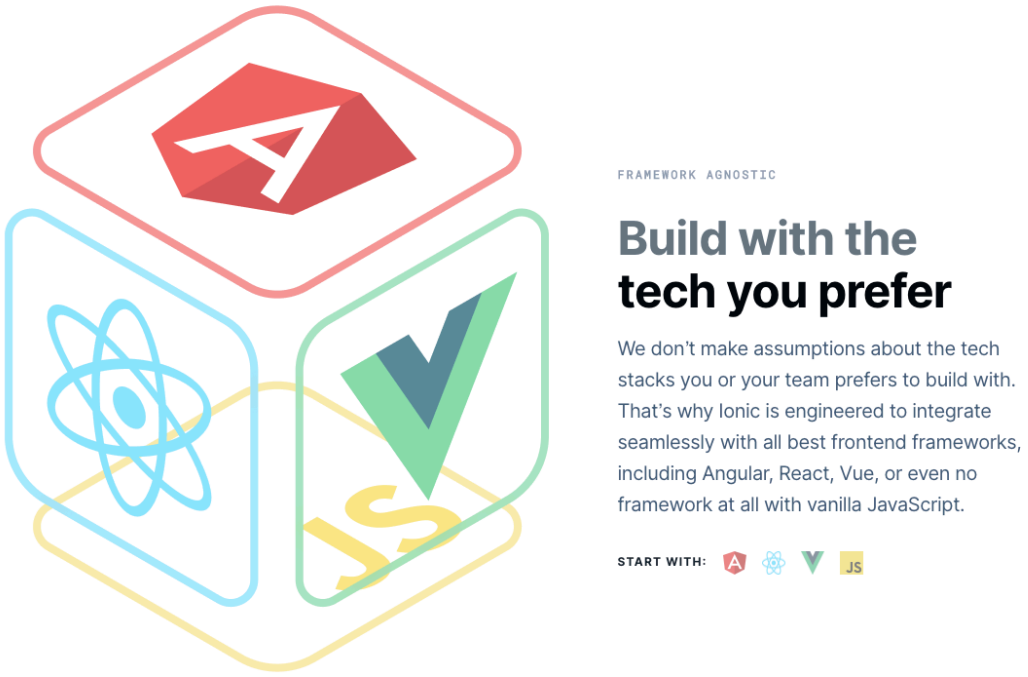
Learn the basics of Ionic 6
using Angular & Capacitor

Contents

1 Getting Started with Ionic

1.1 Introduction

Ionic is an open source fUI toolkitramework that allows us to develop mobile and desktop apps using web technologies – HTML, CSS and Javascript. It integrates wiht popular frameworks like Angular, React and Vue, but we will focus **only on Angular in this guide**.



Source:https://ionicframework.com/

Ionic is built to perform and run fast on all of the latest mobile devices, and your app is packaged for native platforms using a tool like Capacitor which was created by Ionic as well.

The Ionic app is running inside a webview on mobile devices, but of course you don't see a navigation bar inside that app later! In fact, you usually can't see a difference between a native app built with Swift/Java or Ionic, Especially not the end users, who don't care about which technology you used to build your app.

You might also see a lot of references to Cordova, but for all new Ionic apps I recommend Capacitor instead - think of it as Cordova 2.0. It's not ceompletely like that, and you can learn more about the actual differences here.

The focus of Ionic itself is on the frontend UX and UI, meaning controls, components or gestures to create a platform specific look for both iOS and Android.

With Ionic your app is built from one single code base, and the apps are often referred to as cross platform apps or also hybrid apps (*although the latter has a bitter taste for some people*).

More accurate these days is actually the term web native apps as our apps are first class web apps and powerful native apps at the same time.

We will get more into all the details about Ionic and Capacitor in this guide - for now let's set up our environment so we can build apps!

1.2 Installation & Setup

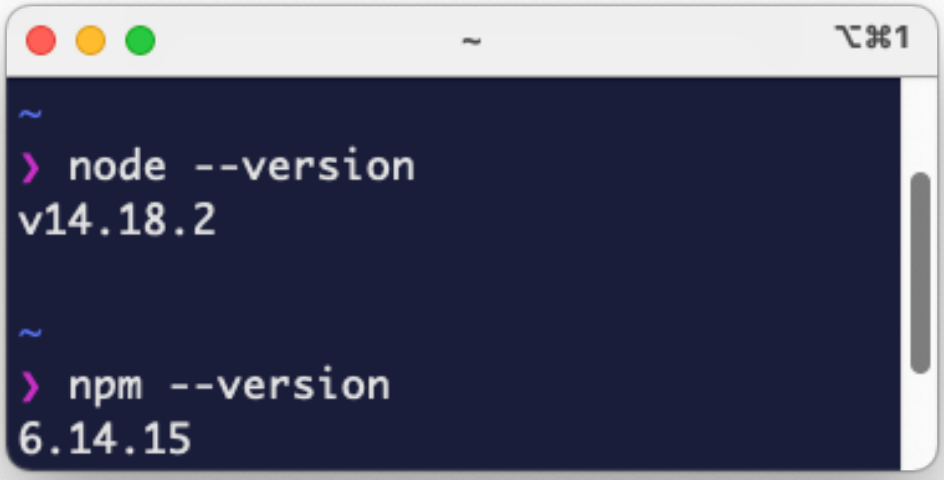
Before you install Ionic, you need to make sure you have Node.js installed correctly.

You can either just download it from there or use a version manager like nvm or n. I actually use n because it's super simple and does the job very well!

You can confirm that Node is installed correctly by running the following on your command line:

```
node --version
npm --version
```

You can see my output below, although your versions will likely be more up to date.



Be careful of using the latest versions, since I sometimes had problems with those in combination with Ionic, Capacitor or other build related tools.

Nonetheless, if you get an output you can continue and finally install the Ionic CLI:

```
npm install -g @ionic/cli
```

This will install the package **globally** on your machine. If you don't get any error, you should be able to check the installed version by running:

```
ionic -v
```

Note: This is the **Ionic CLI version** and you shouldn't confuse this with the **framework version**!

Usually you don't really need to care about the version of the CLI, only major new versions of the framework will be interesting for you later down the road.

1.3 Code Editor

While we are here, let me recommend my editor of choice since years for all Angular related projects: Visual Studio Code!

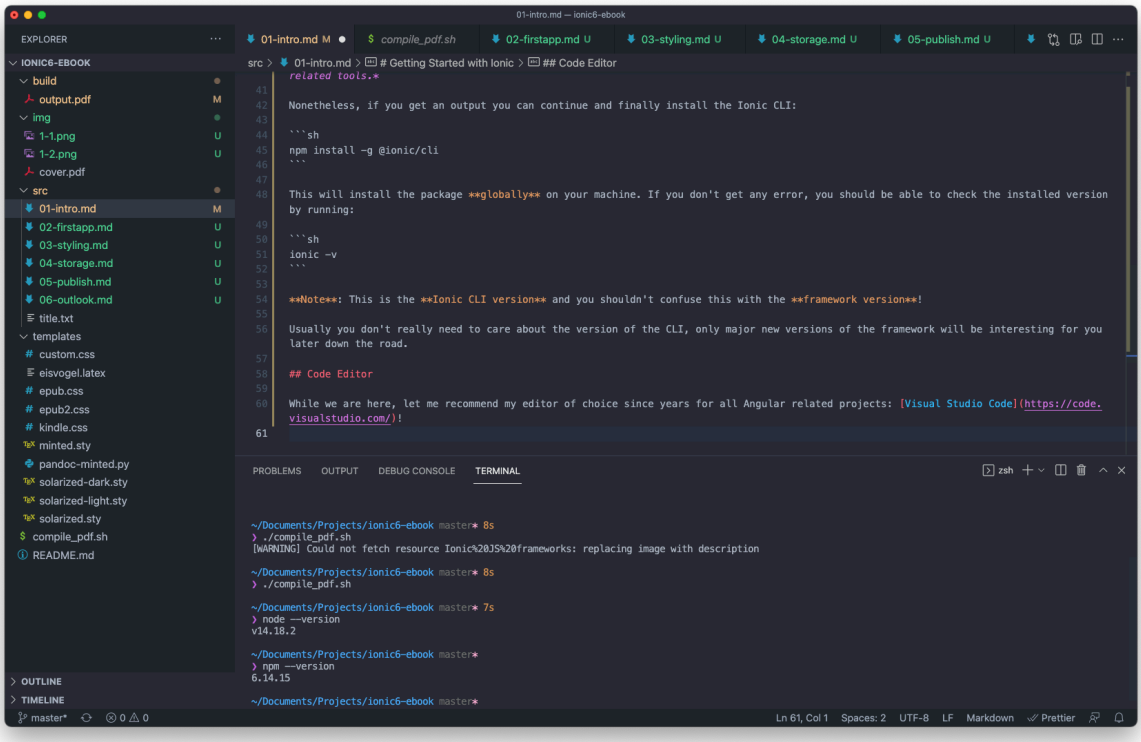


Figure 1.1: VSC

It’s exactly what we need to work with HTML, CSS and JavaScript and many extensions make this really an awesome code editor.

Of course you can use something else like Sublime, Atom or whatever you prefer, but I still recommend you give VSC a try!

Once you got your environment finished, let’s create our first app.

2 Your First Ionic App

2.1 The Ionic CLI

We are now ready to start the first project – *already a bit excited*?

The cool thing is that we can create, modify and build our project using the Ionic command-line interface (CLI). It's all we need to develop Ionic apps!

Now navigate to a folder where you want to start a new Ionic app and run:

```
ionic start myApp blank --type=angular
```

This might take a bit depending on your connection, 1-2 minutes are completely normal as it will install quite a few packages!

There are different elements in this command, so let me explain the structure:

```
ionic <command> <project-name> <template> <framework>
```

In our case we want to **start** a new app with the name **myApp** using the **blank** template and use **Angular** as our JS frameworks!

There are actually three different templates available to begin with:

- tabs: A tabs based layout
- sidemenu: A sidemenu based layout
- blank: An empty project with a single page

Especially for beginners I recommend the blank template to understand how everything works.

> I still remember how motivated I started the tabs template with Ionic v1 and was completely lost in the code trying to figure out how everything played together (not knowing about Angular.js wasn't a help at that point).

At this point your Ionic apps is finished. *Sounds funny?*

Well I can prove!

Navigate into the folder of your app and then use the serve command to see a preview of your app:

```
cd ./myApp
ionic serve
```

This will compile the project and run it on a local server. You don't have to take care of anything here, just sit and wait..

After a quick build time, your browser should open and you should see a preview of your app running on http://localhost:8100/.

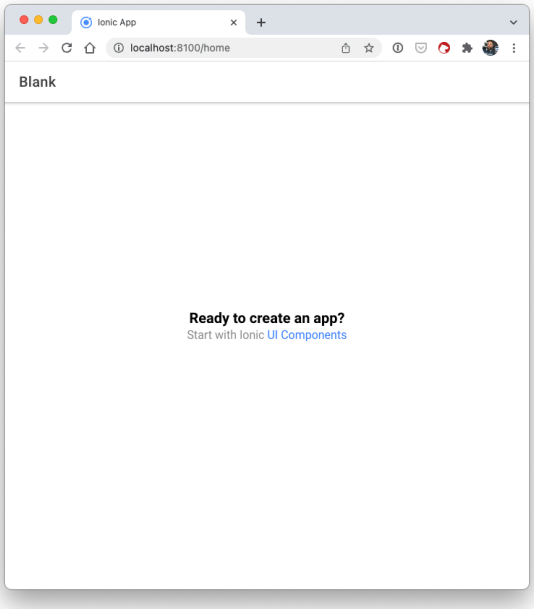


Figure 2.1: Ionic browser preview

That is a very good sign, because you have just created and started your first Ionic app!

This is the preview function which you will use 95% of the time to develop your apps – and it's unbelievable fast.

If you are a former native developer, this will feel so much faster, and if you are coming from web development you feel right at home!

You can also install another package for a cool preview that might help in the beginning, so run:

```
npm install @ionic/lab
ionic lab
```

This command runs a special lab view that shows how your app looks on iOS or Android side by side!

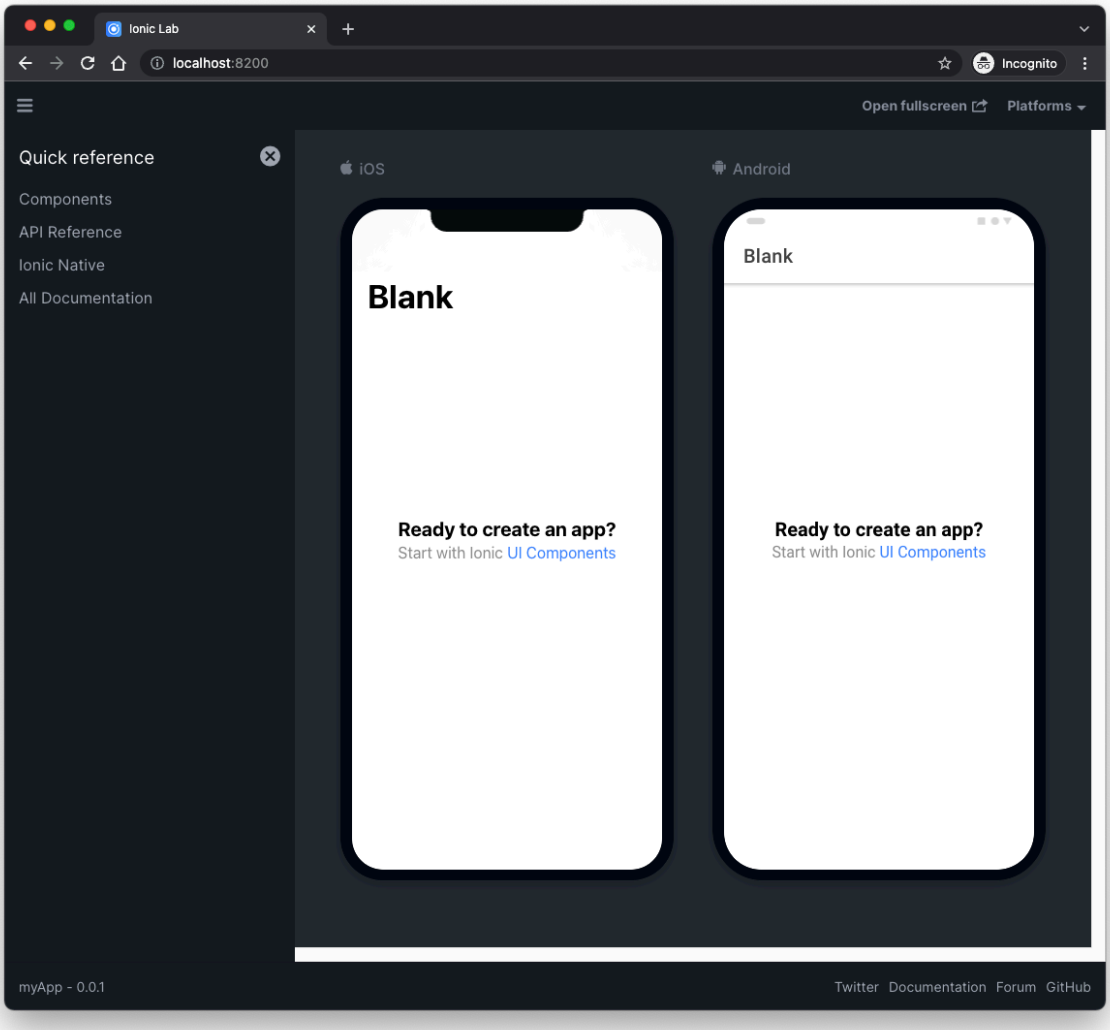


Figure 2.2: Ionic lab

Fascinating, isn't it?

At the top right corner you can pick which **platform** you would like to see the preview for – different platforms will automatically have a **different styling** (but more on this later).

For now we leave that view and take a look at what we have actually downloaded and bootstrapped before.

2.2 Ionic Project Structure

The files inside of your folder might look quite **scary and overwhelming** if this is your first encounter with Ionic or an Angular project – but most of the time you will be working only in the **src** folder and can forget about the rest!

Your app folder will look very likely look like the one in the image below.

So what is all of that?

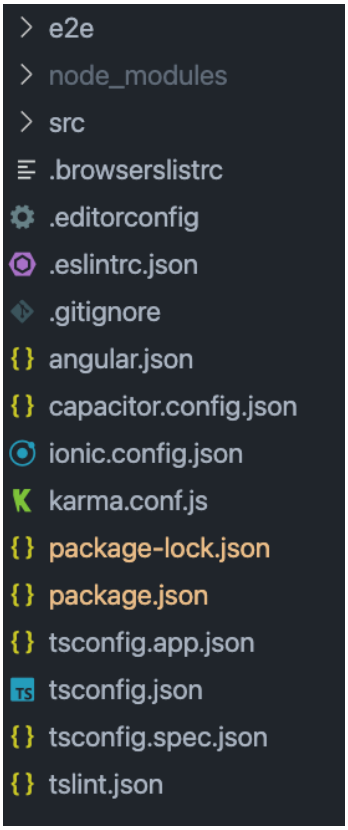


Figure 2.3: Ionic project structure

In general we can ignore all files starting with a dot, since those are only relevant for your general environment and we won't inspect them closer in this guide.

Now let's talk about the folders:

- **e2e**: These are end to end testing files, but if you don't plan to use testcases (which you maybe should at a later point) you can ignore the folder for now
- **node_modules**: This folder is automatically generated once you install the npm dependencies with `npm install` (Ionic already did this for you in the beginning). This command will scan the **package.json** for all the packages that need to be installed and is a classic Node.js file
- **src**: This folder is the most important folder, and 99% of your work will happen in that folder. It's the folder that contains your actual Angular code

Next to the folders we got some files, so let's talk about the most relevant as well:

- **angular.json**: This file holds configuration values for our Ionic/Angular project and you don't really need to touch or change it in the beginning
- **capacitor.config.json** / **ionic.config.json**: More configuration for your project, especially when you build the native apps later
- **package.json**: Already mentioned before - all your package dependencies are specified in this file
- **ts***: Configuration files for Typescript, nothing we need to touch today

With the knowledge about all of these folders we can dive into what is actually our app and change a few things!

2.3 Working with Angular

Until now we haven't seen much of the code – so let's change that!

As said before, we are working inside the **src** folder of your project, therefore we inspect what we currently got.

The **app** folder is more or less the **entry point** of your app. Everything inside that folder is used when your app is bootstrapped.

This folder works in combination with the **index.html**, which also imports some stuff and has this part somewhere inside the body of your HTML:

```
<body>
  <app-root></app-root>
</body>
```

This is the place where your app will be loaded into!

You rarely have to touch that file, but it's good to know how everything works.

Our App is made up of different **components** or also called **pages**, which are organized in **modules** when using Angular. After starting our app we already have on page inside the folder called **home**.

If you generate a page in an Ionic project, the page always comes with the same files:

- **routing.module.ts**: Defining a route to this page - more about navigation later
- **module.ts**: The module of that page where you add imports and declare the actual component
- **page.html**: The template of your page
- **page.scss**: The styling for this page
- **page.spec.ts**: A file for testcases
- **page.ts**: The "controller" of the page template

We can see that each page consists of different parts – and they are of course connected.

Within the SCSS file you can define styling for your view, the HTML page represents your view and the Page TypeScript file contains the class that is associated with your view.

The connection between view and class comes from Angular and we don't have to add anything to get it working. Your **home.page.t**s will most likely look like this:

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

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

  constructor() {}

}
```

For Javascript or web developers this looks very object oriented and for native developers it looks also kinda familiar – and it is!

The `@Component` decorator is used internally by Angular and we don't have to change anything here, the imports at the top import different components from other NPM packages and you'll see some more of them during the next chapters.

Inside our class we can define variables like in all other languages, so go ahead and change your class to this:

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

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

  myVariable = 'I open at the close';

}
```

Now how to show this value in our view?

We can access this variable through **Angulars data binding** by using double brackets {{ and }} inside our HTML view. Therefore open your **home.page.html** and change it to:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Ionic Blank
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">

  {{ myVariable }}

</ion-content>
```

We don't mind all the other HTML tags right now and only focus on the one line inside of our `ion-content` which loads the value of our class right into the view!

If you have the preview still open and save your file, you will see that a reload is triggered. That's because Ionic uses **live reload**, and whenever you make changes those are immediately reflected inside the preview!

Ok until now it's pretty static to just see the string, so we add a button and call a function to change the string to something else!

We start with our class where we implement a **function** that changes our variable. We can access all the variables of a page by using `this`, so change your **home.page.ts** to look like this:

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

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {
  myVariable = 'The force is with me!';

  updateMyValue() {
    this.myVariable = 'Now the force is even stronger!';
  }
}
```

I think how a function works should be quite clear to you.

But how to call it from the view?

We need to add a button using Ionics components (which already have some basic styling!) and tell it to call the function once someone clicks/taps that button.

Therefore, open the **home.page.html** again and change it to:

```
<ion-header>
  <ion-toolbar>
    <ion-title>
      Ionic Blank
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">

  <ion-button expand="full" (click)="updateMyValue()">Click me!</ion-button>

  {{ myVariable }}

</ion-content>
```

Besides creating an `ion-button` which should expand to the full size of the view, we also specify what should happen on a click event – in this case the function of our class will be called.

You might notice now that when you click the button, the value of **the view is directly updated**. This is again automatically working thanks to the **data binding of Angular** between view and class!

Enough of all of this for now – you can of course play around a bit more with the code, HTML and Angular but we haven't talked about all those strange tags in the HTML yet.

2.4 Ionic Components

From that previous example you learned about some of the most basic Ionic components. There is a huge variety of Ionic components available that you can use to build your app, and they are awesome for some many reasons, let's just pick 2:

1. They are standard **web components** - more on what that means in our styling chapter but the good thing is, you can customise each of them to your needs and you don't need to use the standard UI (*although the out of the box styling of Ionic components is pretty great*)
2. The **look different on iOS and Android automatically** - we'll see how this works later but you can already see this by using the lab command or selecting a device preview in your browser debugging tools

Understanding different Ionic components and how to use them is part of the learning curve, and something you will soon feel comfortable with!

To get a better feeling about your current code, let's inspect the tags that we got:

- `ion-header`: The header is a parent component for the toolbar area of your app, and is one of the 3 main elements next to the conent and footer element
- `ion-toolbar`: The toolbar can be positioned inside a header or footer and is fixed in its container
- `ion-content`: The main content area of your page that automatically scrolls whne it's longer than the view
- `ion-button`: A simple button component with a bunch of possible customization options

In the beginning I recommend you take a rough look at all available Ionic components. When you work on your app later you will come back to this view and find the right elements to create your page, and you will get better over time knowing when which component makes sense!

We could go on and use them one by one, but from going through 40 components you will maybe remember 5, so let's instead focus on another essential area that brings a lot of problems for new Ionic or Angular developers.

2.5 Navigation and Routing

Ionic has no own routing concept and relies on the underlying JS framework - in our case, we are using the **Angular Router** for navigation in our app.

Right now on our browser preview we see the HomePage - but how is it actually loaded?

To understand this we need to open our **app/app-routing.module.ts** in which we will find:

```
import { NgModule } from '@angular/core';
import { PreloadAllModules, RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  {
    path: 'home',
    loadChildren: () => import('./home/home.module').then( m =>
 ↪  m.HomePageModule)
  },
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  },
];

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

This is the first place for routing information in our app and the place where we can add more information about how our app works. Right now, we have two routes defined inside the array.

The second is actually a simple redirect that will change the empty path '' to the 'home' path, so it's like going to google.com/ and being redirected to google.com/home.

Inside the definition for the 'home' path we can now spot the `loadChildren` key in which we supply a path to the **module file of our home page**. This module file holds some information and imports for the page, but you can think of it as **the page that gets displayed**.

Ok, cool, we now have a router and are loading a page through a path, so how is this connected with actual HTML or the *index page of the app*?

If you happen to inspect your **index.html** the only thing you'll find inside the body is:

```
<body>
 <app-root></app-root>
</body>
```

The only thing we display is an **app-root** component, which is still not very clear. This app root is replaced by the first real HTML of our app, which is always inside the **app/app.component.html**:

```
<ion-app>
  <ion-router-outlet></ion-router-outlet>
</ion-app>
```

This is the key to understanding how the routing works: The Angular Router will **replace router outlets with the resolved information for a path**.

This means inside the body, at the top level, we have this special Ionic router outlet (*which is the standard Angular outlet plus some animation extras*) wrapped inside a tag for the Ionic app itself.

Once we navigate to a certain path, the router will **look for a match inside the routes we defined**, and display the page inside the right outlet.

We needed this short detour to get a solid understanding about why the things we've done work as they do.

2.6 Basic Routing Concepts

So **router outlets** inside our app will be replaced by the Angular Router whenever we find a path match. For all pages that you somehow want to access inside your app, you need to **define a path**.

The good thing (*sometimes, not always*) is that when you use the Ionic CLI to generate new pages, a new routing entry will be added automatically. You can run inside your project the command to generate new pages like this:

```
ionic g page list
ionic g page details
```

Your routing inside the **app-routing.module.ts** now holds the information for the new paths:

```
const routes: Routes = [
  {
    path: 'home',
    loadChildren: () => import('./home/home.module').then( m =>
↪   m.HomePageModule)
  },
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full'
  },
  {
    path: 'list',
    loadChildren: () => import('./list/list.module').then( m =>
↪   m.ListPageModule)
  },
  {
    path: 'details',
    loadChildren: () => import('./details/details.module').then( m =>
↪   m.DetailsPageModule)
  },
];
```

This means, your app can now show content at these routes:

- **/home**: The content of the home.page.html
- **/list**: The content of the list.page.html
- **/details**: The content of the details.page.html

You could in fact already reach those apps by using the URL directly in your address bar. But that's not how native apps work, right?

In order to access one of these pages you need to use the Angular router, and you can navigate to a page both from the HTML code and also from the TS file.

Let's start with our **home.page.html** and change it to:

```
<ion-header>
  <ion-toolbar>
    <ion-title> Ionic Blank </ion-title>
  </ion-toolbar>
```

```
</ion-header>

<ion-content class="ion-padding">
  <ion-button expand="full" routerLink="/list">Open list!</ion-button>
  <ion-button expand="full" (click)="openDetails()">Open details from
   ↪  code!</ion-button>
</ion-content>
```

Now we can already move to the list page with the first button - give it a try!

Let's quickly see how to do the same from code by implementing a new function inside the **home.page.ts**:

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {
  constructor(private router: Router) {}

  openDetails() {
    this.router.navigateByUrl('/details');
  }
}
```

We have now also **injected** the Angular router as a service inside the constructor of this class. Angular handles the dependency injection for us, and everything we add to the constructor like this will be available in our class!

If you inspect you browser behaviour closely you can now see: **There is a transition animation between the pages!**

We get this little detail because our app is wrapped inside the `ion-router-outlet` which automatically adds those page transitions, either forward or backward. And of course they are different for iOS and Android!

2.7 Passing Parameters to a Details Page

At this point you might wonder how you could get data to another page?

Let's also do a quick example on that right here!

The most recommended and secure way for navigation is using **route parameters**. These are part of the URL like:

```
To make this navigation work we need to tell the Angular router exactly
 ↪  which routes exist in our application, and include **dynamic data as
 ↪  wildcards** inside the routing information.

For example this let's change our routing info inside the
 ↪  **app-routing.module.ts** to this:

```ts
  {
    path: 'list',
    loadChildren: () => import('./list/list.module').then( m =>
 ↪  m.ListPageModule)
  },
  {
    path: 'list/:id',
    loadChildren: () => import('./details/details.module').then( m =>
 ↪  m.DetailsPageModule)
  },
```

Now our second path to the details page has a parameter that we can fill!

We can simply change our function to have another path component in the URL now:

```
  openDetails() {
    this.router.navigateByUrl('/list/1337');
  }
```

Next step is reading that data on the according page, which in our case is now the `DetailsPage`.

To achieve this, we inject the `ActivatedRoute` from which we can read the information like this inside the **details.page.ts**:
```

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-details',
  templateUrl: './details.page.html',
  styleUrls: ['./details.page.scss'],
})
export class DetailsPage implements OnInit {
  myId = null;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.myId = this.route.snapshot.paramMap.get('id');
  }
}
```

How to confirm that it works?

Well let's simply print out that value inside the **details.page.html** liek we learned before:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button></ion-back-button>
    </ion-buttons>
    <ion-title>details</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-card>
    <ion-card-content>
      {{ myId }}
    </ion-card-content>
  </ion-card>
</ion-content>
```

You are now able to move **forward** to a page and include information in the path!

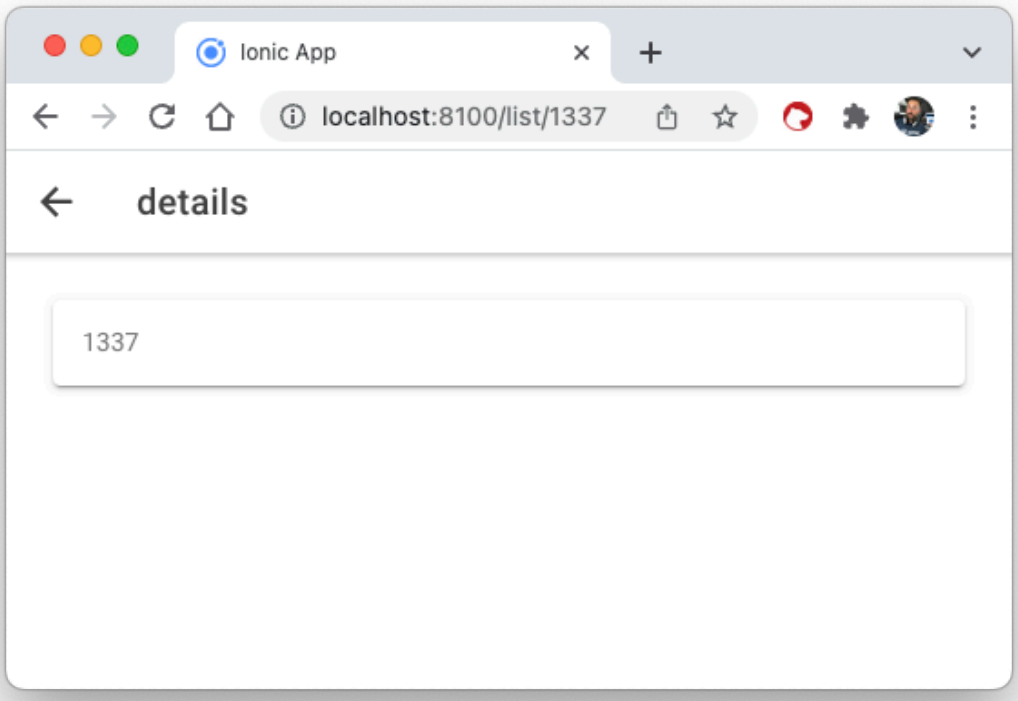


Figure 2.4: Ionic navigation

For the fun I integrated another component - the `ion-card` - because it's a component used in many places that you should be familiar with.

Do you spot anything else on this page? Yes? Good.

I also included the `ion-back-button` inside our header. This component automatically displays a back arrow (or different icon or text) when we navigated to a page like this and got a page in our history that we can jump back to.

There is nothing else you need to do - Ionic automatically handles this for your.

> Note: Even if we didn't add a back button, we could have used a drag gesture from the side to go back - just like in any other native app!

After going through this chapter you should have a basic understanding about Ionic projects, some Angular basics and which files to touch to add some logic and interaction to your app.

3 Styling Ionic Apps

First of all, in general we can style and structure our app through all kinds of **CSS**.

You might have noticed that each new generated page comes with its own *.scss* file, so those files are used to directly style one specific page.

Additional we got the **theme** folder right next to our app folder. This is a place to override **Ionic variables** (more on this soon) and to import e.g. custom fonts or more individual styling files.

Finally we also got another styling file at **app/global.scss** where we can apply some **global styling** to our app.

These 3 are the areas where we can add styling, and of course we need to work with our HTML to apply CSS classes to different objects.

3.1 Global Styling

By default, Ionic components come with custom styling for different platforms:

- `ios`: iOS styling, visible on iPhone and iPad
- `md`: Android styling following Material Design
- `core`: Any other platform will also use the `md` styling, meaning as a website your app looks like the Android version

You will see the according styling when you run the app on a device running a specific OS, or you can also test this with your browser debugging tools.

Ionic automatically handles this by adding the according CSS class to the `html` tag:

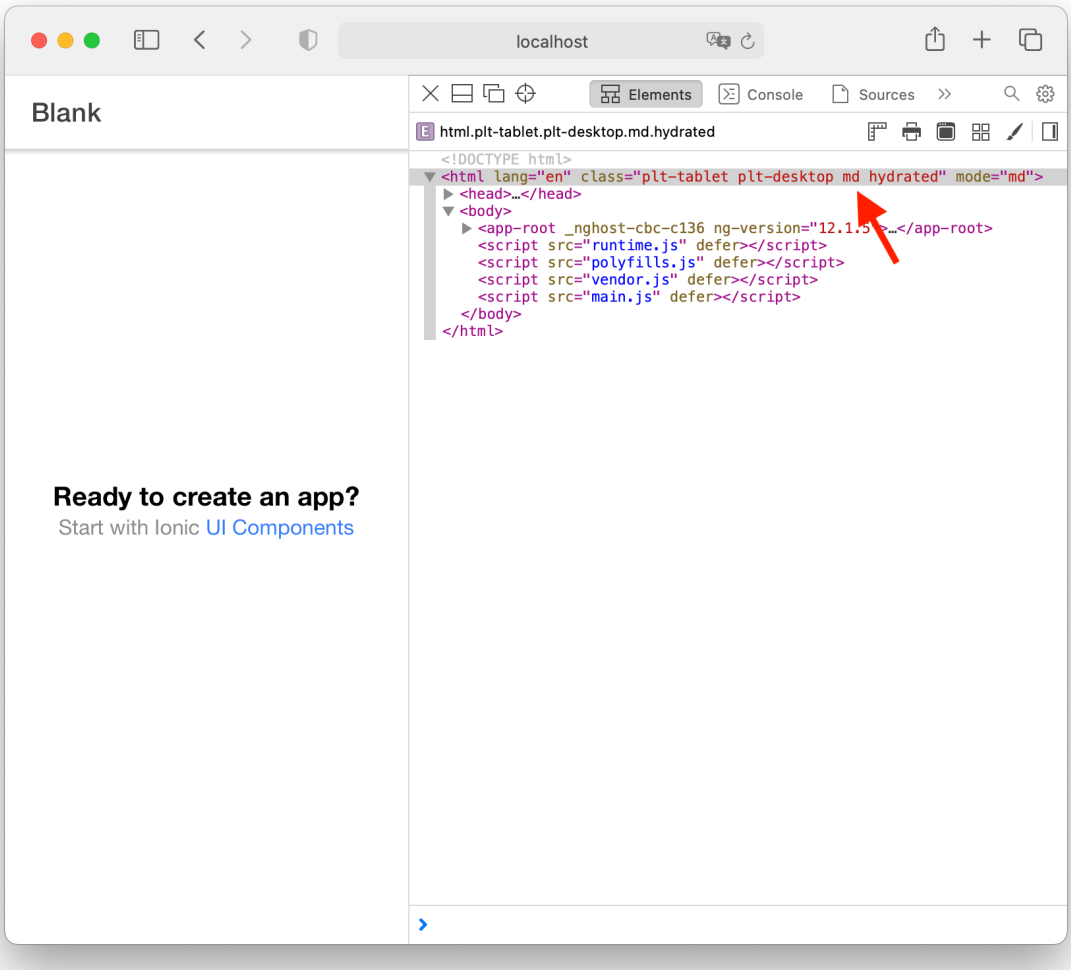


Figure 3.1: Ionic platform styling

Usually this platform specific styling is great, as it gives the users a native look and feel, but like all things Ionic, you could change this as well.

If you want one specific component to always look like the iOS version, you could directly set the mode on the element:

```
<ion-button mode="ios">I will always be iOS</ion-button>
```

You can do this on **every Ionic component**, as all of them come with two styling files.

If you (for whatever reason) like the UI concept of one platform simply more, you could even set the

mode for your whole application at the top level inside the `forRoot()` function of your module:

```
@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [BrowserModule, IonicModule.forRoot({
    mode: 'ios'
  }), AppRoutingModule],
  providers: [{ provide: RouteReuseStrategy, useClass: IonicRouteStrategy
 ↪  }],
  bootstrap: [AppComponent],
})
export class AppModule { }
```

Another way to target platform specific styling would be using the class name inside your CSS. Let's say you only want to change the appearance of the MD button, you could prefix your styling with the `md` class:

```
.md ion-button {
  font-weight: 600;
}
```

Now all this was just general information about Ionic apps, but we need to understand one some mor concepts in order to effectively style our apps.

3.2 Web Components

All Ionic components are Web Components, built with Ionics own Web Component compiler called Stencil.

What exactly does this mean?

From the outside, what you usually see looks like this:

```
<ion-item>My content</ion-item>
```

The `ion-item` is a Web Component, and when you inspect these components with a debugging tool inside your browser you see that there's actually a lot going on under the hood!

Figure 3.2: Web Component content

A Web Component is basically a container for some elements, with additional functionality. Everyone can create a Web Component and define the elements within, plus any logic or styling tied to that element.

We as the user can simply use a Web Component like the item without knowing what's actually going on inside of it - it's like importing a module in your HTML. We only need to know the tag name, the browser will handle the rest.

While this is a great concept, there's another thing about Ionic components we need to understand.

3.3 Shadow DOM

Shadow DOM is an API that is part of the Web Component model, and most Ionic components make use of it.

One of the main features of the Shadow DOM is **isolation**, which basically means the whole content of the Web Component is not part of your actual DOM but lives isolated from the global scope in a **shadow tree** instead.

Why?

Imagine Ionic would supply a lot of styling with their components (which they do), and you use the components in your page. Now with bad luck you might have the same CSS rules and names like Ionic has used, which introduces a bunch of problems between your styling and the Ionic styling.

Simple example: Ionic defines a `.button` styling, but your company also has a `.button` rule - using the Ionic component would have leaked their CSS rules into your own page causing a mess.

This was a common problem in the past as projects got bigger and different components just didn't play well together on one page.

With Shadow DOM, the styling of a Web Component is **isolated** from the rest of your page, which means it won't mess up your overall styling - but it also becomes harder to actually **inject** styling into these components.

3.4 Adding Styling to Ionic components

Alright enough theory for now, let's get back into our app and try to style some elements with this new knowledge.

First, let's try to use one of the predefined colors - you can find all of them inside the **variables.scss** of your project!

Bring up your **home.page.html** again and change it to:

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title> Ionic Blank </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-button color="secondary" expand="full" routerLink="/list">Open
   ↪  list!</ion-button>
  <ion-button color="warning" expand="full" (click)="openDetails()">Open
   ↪  details from code!</ion-button>
</ion-content>
```

These colors are shipped with every Ionic app, and they way to use them is by placing the `color` property on an Ionic component that allows this property.

If you quickly want to generate new values, you can also use the Ionic color generator!

Now let's put in a bit of standard CSS for our elements inside the **home.page.scss**:

```
ion-button {
  margin: 30px;
  height: 50px;
}
```

All of this works pretty flawless, so *what's the big deal with these web components Simon*?

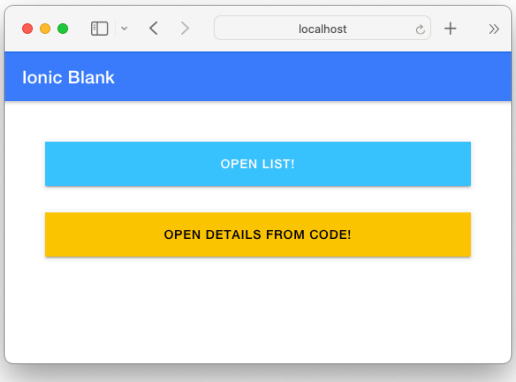


Figure 3.3: Ionic variables and styling

Let's try to change something else:

```
ion-button {
  margin: 30px;
  height: 50px;
  background: red;
  color: white;
  border-radius: 8px;
}
```

Do you see any effect of these changes? No?

Good. Me neither.

This is the problem that most developers experience with Ionic components, and I can understand the confusion.

The CSS properties we try to set are set **inside the Shadow componet** and we can't simply overwrite them from the outside!

For every Ionic component, you can find a list of CSS Custom Properties usually at the end of the documentation of a component.

In the case of the `ion-button` we could now change the code and use **CSS variables** instead, which will be respected and used inside the component:

```
ion-button {
  margin: 30px;
  height: 50px;
  --background: red;
  --color: white;
  --border-radius: 8px;
}
```

At this point it's still not working because we actually target those values directly on the component, so let's remove the values we added from the elements again:

```
  <ion-button routerLink="/list">Open list!</ion-button>
  <ion-button (click)="openDetails()">Open details from code!</ion-button>
```

Ok I can see your head is spinning, and I totally agree that this stuff is challenging.

The only way to get better? **Use the components**, look at their properties and over time you'll get better at writing the correct CSS!

In the end, styling Ionic apps comes down to writing CSS. So any course on CSS basics can be helpful.

3.5 More Material?

In fact this topic is so complicated that I've written a whole book about it and how to style amazing Ionic apps like world class apps.

If you're interested in learning more about styling, check out my book Built with Ionic!

4 Storing Data with Ionic Storage

Although this might me beyond a beginners quickstart guide, a very basic question developers have is usually: *Where to store the data of my app?*

Of course you might have a real backend with API in the future, but usually you still need to store information inside your app from time to time.

Lucky us, there are already great solutions out there which embrace either the localstorage of a browser or even the native device memory.

4.1 Setting up Ionic Storage

Ionic Storage is our go-to package for easily managing data. With Ionic Storage we can save **JSON objects** and **key/value pairs** to different storage engines, unified through one interface.

Ok in easy words this means, Storage will internally select which **storage engine** available is and select the best possible solution for us.

If you run the preview, it will try: **IndexedDB**, **WebSQL** and finally **localstorage**.

The problem with localstorage in general is that this can get cleaned from the OS of a mobile device and you **lose all data**. Not a very good idea.

To get started, simply install the following packages in your app:

```
npm install @ionic/storage-angular
npm install cordova-sqlite-storage
npm install localforage-cordovasqlitedriver
```

The first is the actual package, the rest is necessary to embrace the underlying **SQLite database** of a mobile app when our app runs on a real device one day.

To use storage we now also need to import it into our module, therefore open the **src/app/app.module.ts** and add it like this:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouteReuseStrategy } from '@angular/router';

import { IonicModule, IonicRouteStrategy } from '@ionic/angular';

import { AppComponent } from './app.component';
import { AppRoutingModule } from './app-routing.module';
import { IonicStorageModule } from '@ionic/storage-angular';
import { Drivers } from '@ionic/storage';
import * as CordovaSQLiteDriver from 'localforage-cordovasqlitedriver';

@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [
    BrowserModule,
    IonicModule.forRoot(),
    AppRoutingModule,
    IonicStorageModule.forRoot({
      driverOrder: [
        CordovaSQLiteDriver._driver,
        Drivers.IndexedDB,
        Drivers.LocalStorage,
      ],
    }),
  ],
  providers: [{ provide: RouteReuseStrategy, useClass: IonicRouteStrategy
 ↪  }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

After the setup we can put Storage to use, but we need to make sure that we initialize it correctly before doing any operation.

For this, let's open the **src/app/app.component.ts** and change it to:

```
import { Component } from '@angular/core';
import { Storage } from '@ionic/storage-angular';
import * as CordovaSQLiteDriver from 'localforage-cordovasqlitedriver';
```

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.scss'],
})
export class AppComponent {
  constructor(private storage: Storage) {
    this.init();
  }

  async init() {
    await this.storage.defineDriver(CordovaSQLiteDriver);
    await this.storage.create();
  }
}
```

Alright, now we can use Ionic Storage to write and read some data!

4.2 Read and Write to Storage

We've learned how to use Angular data binding in the first part of this guide, so let's combine that knowledge with saving data to storage and loading it again later.

To begin with, we can inject the `Storage` service in our page and create two new functions to write and read data.

Go ahead by changing the **home.page.ts** to this:

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { Storage } from '@ionic/storage-angular';

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {

  person = {
```

```
    name: '',
    age: ''
  };

  constructor(private router: Router, private storage: Storage) {}

  openDetails() {
    this.router.navigateByUrl('/list/1337');
  }

  savePerson() {
    this.storage.set('my-person', this.person);
  }

  async loadPerson() {
    const result = await this.storage.get('my-person');
    if (result) {
      this.person = result;
    }
  }
}
```

To set data, we just need to use a specific **key** and then the data, and to read it again we just need to key.

The calls to Ionic Storage are **asynchronous**, which means they don't immediately return a value.

If you are not familiar with this concept yet you can check out my video about async Javascript or find a full course on this topic inside the Ionic Academy!

Anyway, we need more buttons and inpuit fields for our actions, so let's also change the **home.page.html** to:

```
<ion-header>
  <ion-toolbar color="primary">
    <ion-title> Ionic Blank </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-item>
    <ion-label position="floating">Name</ion-label>
```

```
    <ion-input type="text" [(ngModel)]="person.name"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label position="floating">Age</ion-label>
    <ion-input type="number" [(ngModel)]="person.age"></ion-input>
  </ion-item>

  <ion-button expand="full" (click)="loadPerson()">Load person</ion-button>
  <ion-button expand="full" (click)="savePerson()">Save person</ion-button>
</ion-content>
```

Now you can enter some values, hit save and then refresh the page - by clicking load now you get back the values from before!

But where is the data stored?

Insiden the browser preview you can see all stored data by toggling the developer tools, then going to the **Application** tab and from there drilling down into your **IndexedDB**, which is the first choice of Ionic storage when running inside a browser.

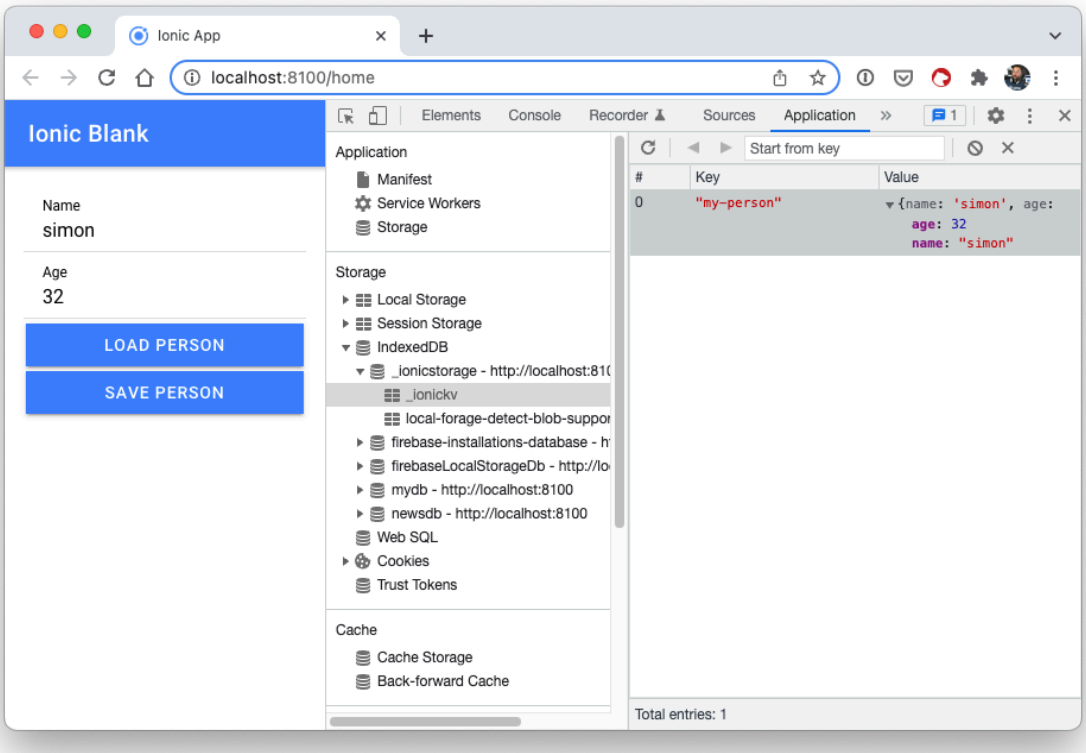


Figure 4.1: Ionic storage browser

We've now learned some of the most basic concepts for building Ionic apps, but there's one thing we haven't seen: Our Ionic app on a real device as a native iOS or Android app!

And that's what we will achieve next.

5 Publishing Ionic Apps for iOS, Android & Web with Capacitor

In previous courses or tutorials you might have seen Cordova instead of Capacitor, which both act as the layer between your Javascript code and native iOS/Android code.

The Ionic company came up with Capacitor to improve tooling and *how* we interact with this native layer. The focus of Capacitor is to run everywhere - **even inside a browser**, which was mostly not possible with Cordova before.

In a nutshell, Capacitor (as well as Cordova) helps to wrap your current web project into a native application by displaying it inside a web view and creating the necessary native iOS and Android projects. On top of that you can access the underlying native SDK to e.g. capture an image or use the gyrosensor.

Capacitor generates a native project inside your folder once, and you simply check in the added project to your source control just like your other code and treat it like any source asset - "**Code once, configure everywhere**"!

At the same time, you can still integrate Cordova plugins (though not all) inside your Capacitor project and use them when deployed to a device.

Ionic now recommends Capacitor for all new projects, and therefore it's the logical choice if you are just starting out!

Once you have installed all required tools, we can continue to build our app.

Note: Most likely your final goal is to ship a mobile application to the Play Store and the App Store, and in order to do so you nee to have accounts at:

- Android: Register inside the Google Play Developer Console for one time fee of $25
- Apple: Subscribe to the Apple Developer Program, pricing varies based on account type

5.1 Setting up your Environment

Since you will now touch *native land*, you might have to install a few things.

For iOS you don't really need a lot – the only thing you should have installed is Xcode, the IDE that native iOS developers use to build their apps with Swift or Objective-C. You can easily download Xcode directly from the App Store on your Mac as well!

Note: If you want to build iOS apps you also need a Mac to build your app or use a cloud service like AppFlow to build your binary in the cloud.

In order to build Android Apps you need the **Java Development Kit** (JDK) and the **Android SDK**. Most likely Java is already installed on your machine, so the next thing would be to download Android Studio.

It's like Xcode for iOS and it will help you to download the Android SDK and keep things up to date if you need to install a later version at some point.

If you encounter any problems during the setup fo your antive tools you can also check out the official guide for iOS or guide for Android.

5.2 Adding Capacitor platforms

To get started we need to add the native platform that we want to build for to our project, and we can do this right with the Ionic CLI.

Because Capacitor won't magically update your native projects and just generate them once in the beginning (unless you alter use the Capacitor configure package), we need to set our **app id** before we add the platforms.

If you have created any mobile app in the past you know that each app needs a **unique bundle id**, which usually looks something like **com.devdactic.myapp**, basically a reverse domain and your app name.

We should set this up inside our **capacitor.config.json** right now:

```
{
    "appId": "com.devdactic.mycoolapp",
    "appName": "myApp",
    "webDir": "www",
    "bundledWebRuntime": false
}
```

Now we can add the according platforms by simply running:

```
ionic build
ionic cap add ios
ionic cap add android
```

Once you added the platforms you got a new folder at **ios** and **android** inside your project which contains a native project that you can open.

Give it a try by executing a local Capacitor script inside your project with **npx**:

```
// Open Xcode
npx cap open ios

// Open Android Studio
npx cap open android
```

Right now the apps won't work because we haven't created a build of our app - you can see this by a warning on your CLI when you added the platfrms:

[capacitor] [warn] sync could not run–missing www directory.

Capacitor basically takes the output (the www folder) and syncs it into the right place of your iOS or Android project. And from now on, whenever we make changes and want to build a new native app, we can simply create a new build and sync the changes.

Let's do this right now:

```
ionic build
npx cap sync
```

Ok we are very close to seeing our Ionic app on a device - just a few more steps!

5.3 Creating a native iOS App

You should have Xcode open right now, if not simply run again:

```
npx cap open ios
```

You are inside Xcode but might feel a bit lost, but don't worry. There are only a handful of things you need to do right now.

First, you need to be enrolled in the Apple Developer Program to build your app.

After that you need to add your Apple ID inside Xcode like described in the official Ionic docs.

When you've done this, you should be able to select your development team inside the **Signing & Capabilities** tab within Xcode:

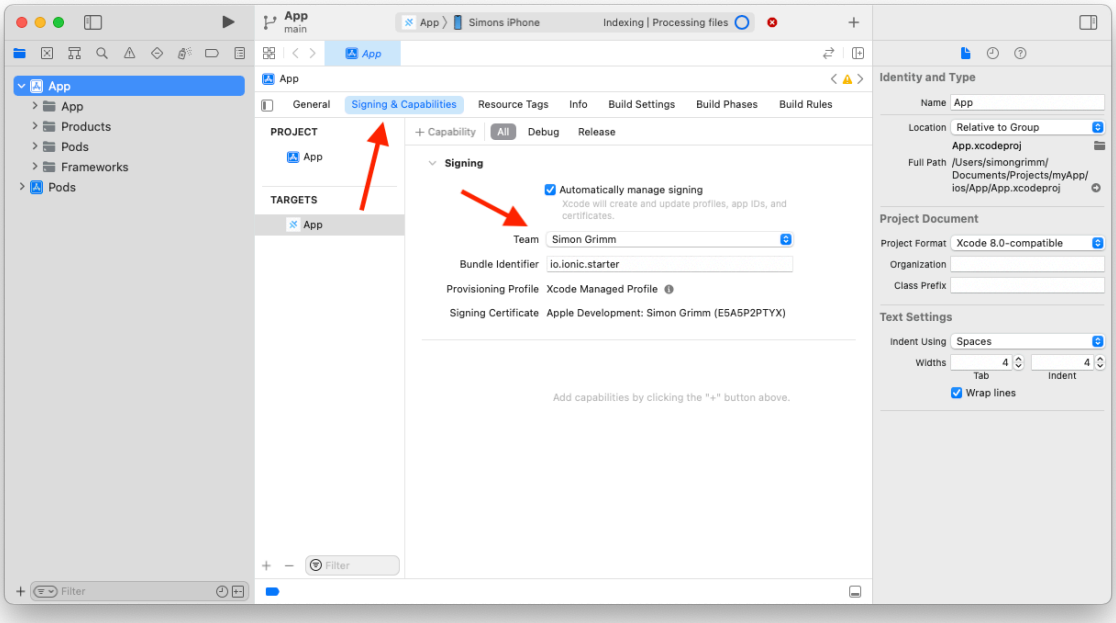


Figure 5.1: Xcode select team

At this point, you only need to plug in your iOS device, select it within the top area and hit the play button!

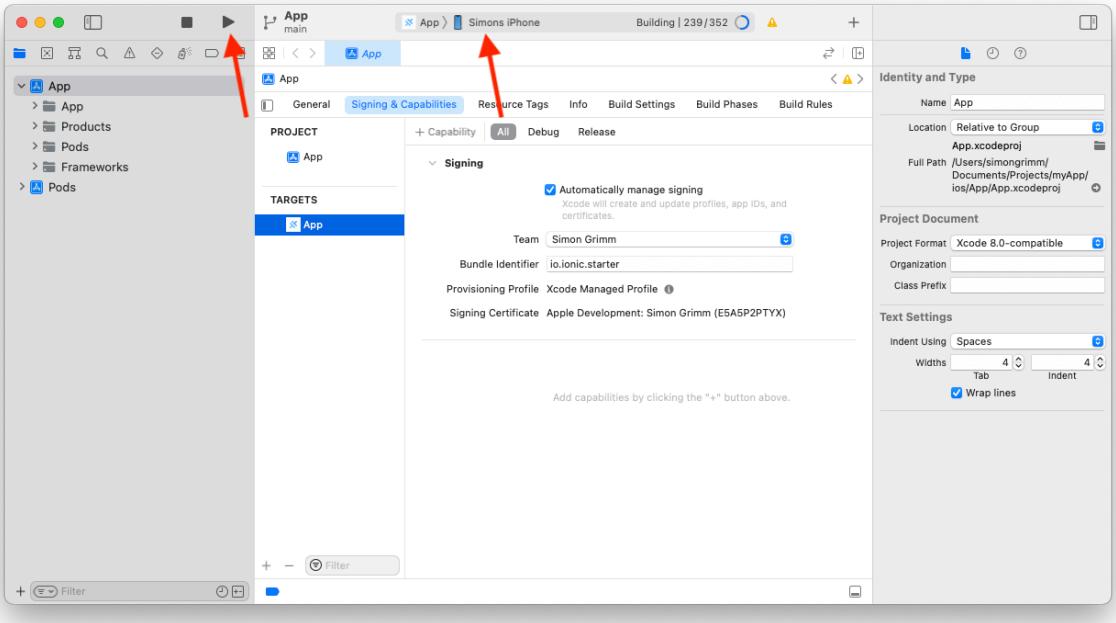


Figure 5.2: Xcode run app

This is a pretty big milestone: **You just created your first native iOS app!**

While you think this is pretty cool, let me shock you with *one more thing…*

You can actually have the same live-reload that you are used from your browser on a device!

After configuring everything inside Xcode you only need to run the following command:

```
ionic capacitor run ios -l --external
```

You now need to select your device, and after the build and deploy you will again see the app on your device.

Note: If the app closes directly after start, simply open it again. It just happens sometimes.

Apply a change to your Ionic code somewhere, hit save and see the app update in real time on your device.

This is one of the most powerful commands, and any native developer would kill for that functionality!

5.4 Creating a native Android App

For Android, we should open Android Studio for now as well:

```
npx cap open android
```

Here we actually don't need any team configuration, and we just need to make sure that our device is plugged in, then select it from the list at the top and hit the run button!

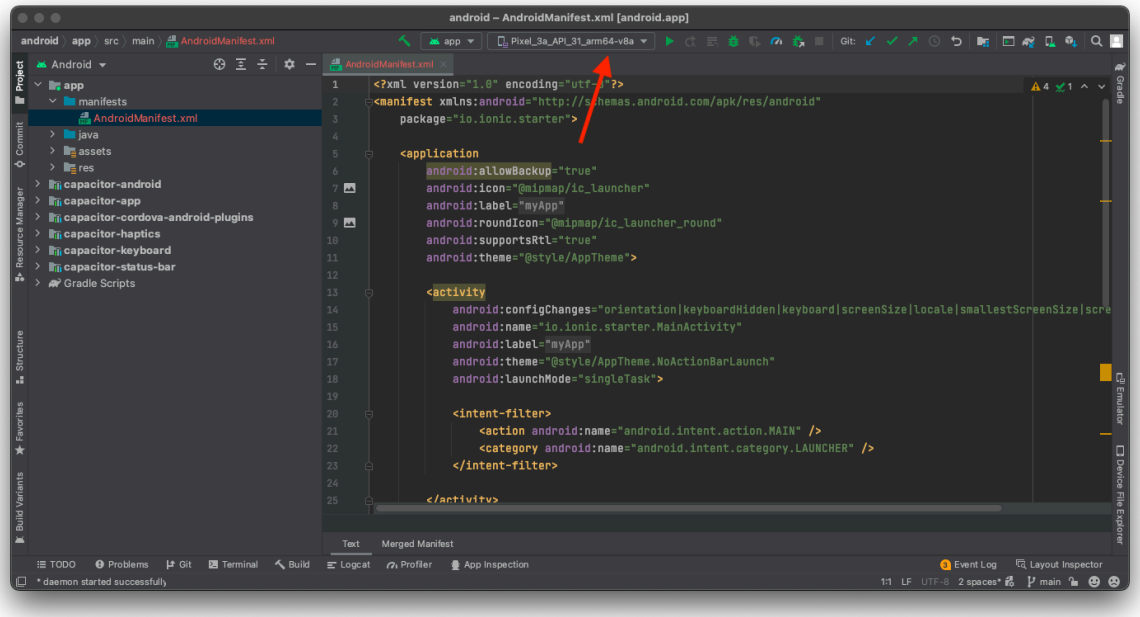


Figure 5.3: Android run app

And boom: **You just created your first native Android app!**

But of course, you can also directly run the app with the cool live-reload functionality we used on iOS:

```
ionic capacitor run android -l --external
```

In fact you can first run the command and open your app on an Android device and afterwards run it again for iOS. By doing this, you can have **live-reload on iOS and Android** on two real devices at the same time!

This is possible because the app is now loading the source assets from a local server, and both apps basically pull their source from the same server that the Ionic CLI starts.

5.5 Build Ionic for the Web

You've seen the preview of your Ionic app on the browser all the time, but let me quickly give you the necessary information to really build and deploy a website from your project as well.

Because after all, **Ionic runs everywhere the web is**!

To do this, we want to generate another build, optimised for production:

```
ionic build --configuration production
```

This flag will optimise your code, and you want to use it as well when you create the final build for your native projects, as this can result in a reduced bundle size and faster app loading times.

After running the command you should see a **www** folder in your project (*which we already had from the build for Capacitor anyway*) and this folder is what you can simply throw into your web hosting!

You can test this by installing the http-server package on your machine and then running this inside your Ionic folder:

```
http-server www
```

Go ahead and visit http://127.0.0.1:8080/ and you will see your *Ionic website*.

If you want to host this app somewhere, I recommend you check out my tutorial on building and deploying your Ionic app as a PWA!

5.6 Adding Capacitor Plugins

Ok cool we got all those projects and even native apps, but we haven't seen the real power of Capacitor yet!

After all, we might want to use native functionality like accessing the camera in our app, and for this we need to use plugins.

Now you got three options:

1. There are official Capacitor plugins
2. There are community plugins for Capacitor
3. You can use most of the existing Cordova plugins

We won't get into the last two points, but rest assured that the procedure is mostly the same.

For this guide, let's add a new plugin by installing the Capacitor Camera package:

```
npm install @capacitor/camera
npx cap sync
```

Whenever we add or remove plugins we also need to sync them to the native platforms, and you will have to restart any ongoing live-reload to get those changes.

Now let's use the plugin by first adding a new button and image inside our **src/app/home/home.page.html**:

```
<ion-content class="ion-padding">
  <ion-button expand="full" (click)="captureImage()">Capture
  ↪ image</ion-button>
  <img [src]="image">
</ion-content>
```

Now we can use the Camera plugin and simply call the `getPhoto()` function and assign the base64 string to a local variable inside the **src/app/home/home.page.ts**:

```
import { Component } from '@angular/core';
import { Camera, CameraResultType } from '@capacitor/camera';

@Component({
  selector: 'app-home',
  templateUrl: 'home.page.html',
  styleUrls: ['home.page.scss'],
})
export class HomePage {
  image = null;

  constructor() {}

  async captureImage() {
    const image = await Camera.getPhoto({
      quality: 90,
      allowEditing: true,
      resultType: CameraResultType.Base64,
    });
```

```
    this.image = '' + image.base64String;
  }
}
```

Do you see any specific iOS or Android code? No?

Me neither - and that's the magic of Capacitor!

Under the hood Capacitor will route our camera call to the according function in native Java or Swift code, which then invokes the camera when the app runs on a device.

But Capacitor plugins sometimes even come with a web implementation fallback, so you have **one function that works across all platforms**!

Now let's see this in action, but if you tried to run your app at this point you will crash the app when clicking the button.

And there's even a hint in the logs why:

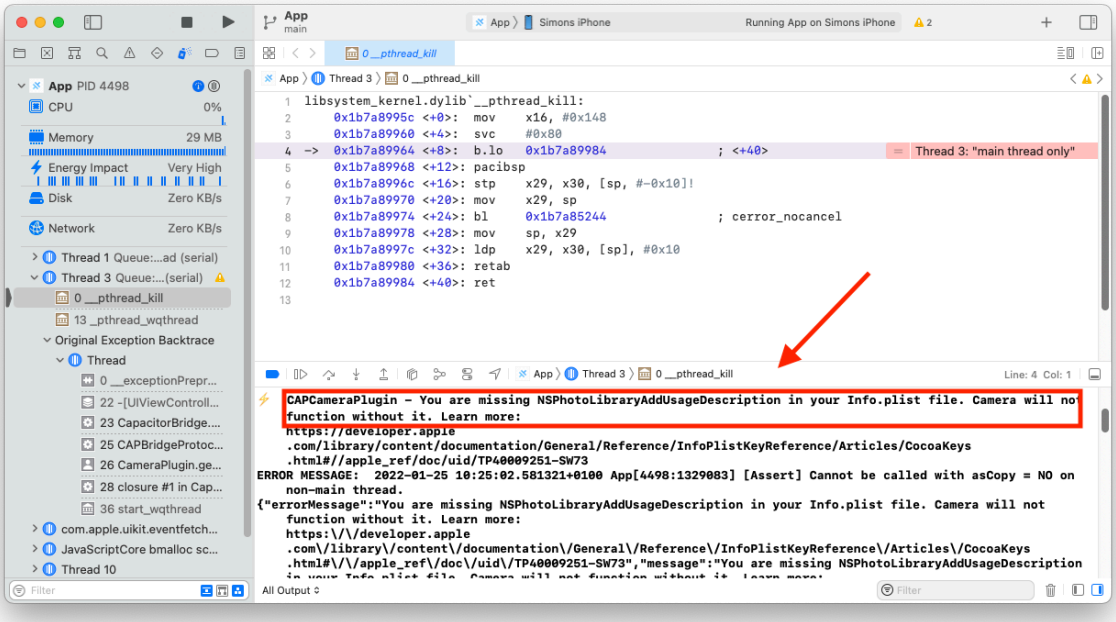


Figure 5.4: Permission error

For some native functionality we need to ask the user for consent, and to do this we sometimes need to setup some keys ofr both iOs and Android.

Let's do this for the camera by adding the following entries inside the `dict` of the **ios/App/App/Info.plist**:

```
<key>NSCameraUsageDescription</key>
<string>To capture images</string>
<key>NSPhotoLibraryAddUsageDescription</key>
<string>To add images to the library</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>To select images from the library</string>
```

For Android, we need to open the **android/app/src/main/AndroidManifest.xml** and in the section for permissions at the bottom change it to:

```
    <!-- Permissions -->
    <uses-permission
↪   android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission
↪   android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.INTERNET" />
```

Now you can run your app again and you should be able to capture an image on both Android and iOS!

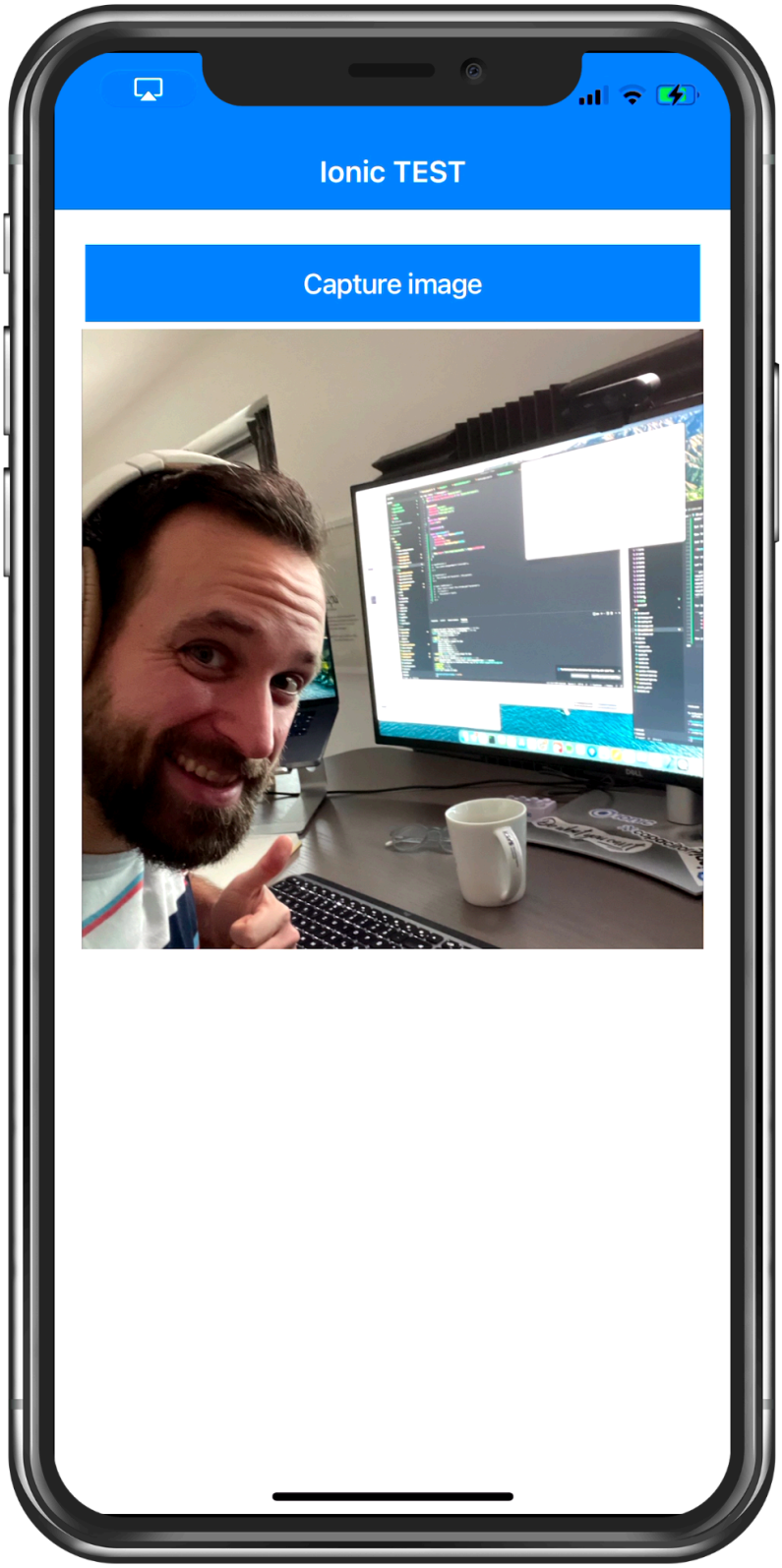


Figure 5.5: Image capturing

Great success - that's me while writing this chapter in my castle of loneliness.

And if you see yourself inside the app after capturing an image or selecting one from the library, you achieved everything from this chapter!

6 What's next?

First of all congratulations on finishing the Ionic Quickstart guide - and thank you for spending your time with me!

I hope you enjoyed the process of creating your first Ionic app with Capacitor for iOS, Android and the web and got a decent understanding for the basics, or at least a *feeling* about Ionic.

It's not for everyone, there are people who enjoy Flutter or React Native more, or maybe you even want to focus solely on native iOS or Android.

Whatever you do, just keep one thing in mind:

Nobody cares about the technology you use.

Except your technical friends and developer bubble on social media.

Average Joe just wants a good app that solves a problem - and you can create exactly that app with a framework like Ionic or any of the other mentioned.

Now if you enjoyed this Quickstart guide, I would love to welcome you inside the Ionic Academy soon - it's my place to help you with everything Ionic and the **fastest way to learn Ionic**!

With over 50 video courses, app templates and quick wins dedicated only to Ionic it's the best place to make progress faster and join a community of helpful Ionic developers.

To get a preview you can simply check out all the free videos on my YouTube channel - and the Ionic Academy courses go into even more detail.

Finally, you can always reach me on Twitter if you got any questions.

May you build something amazing with Ionic.

Happy Coding, Simon