***Giáo trình***

**Lập trình Ứng dụng Di động**

**GS. Phan Viết Hoàng**

**Bài học 11 (Buổi 11)**

**Sửa lỗi, kiểm thử**

# References

# ‘Building Mobile App with Ionic 2’*, Josh Morony, 2016*

**Thử và debug**

**(Lesson 8: Testing and Debugging)**

When creating your application you are, without a doubt, going to make some mistakes and run into some errors. It’s not always obvious where you’ve gone wrong either, so it’s important to know how you can go about tracking down the problem.

This lesson is going to cover how best to debug Ionic 2 applications, but it won’t cover what debugging is in general or how to use debugging tools (e.g. viewing the source code, setting breakpoints, looking at network requests and so on). If you are not familiar with browser based Javascript debugging, then I highly recommend having a read through of [this](https://developer.chrome.com/devtools/docs/javascript-debugging) first.

**Browser Debugging**

Your desktop browser will usually be your first point of call when developing your application. It’s great to be able to debug directly through the browser because, especially with live reload, you can quickly see the impact code changes have and iterate really quickly.

But it’s important to keep in mind that **it is not representative of how the application will run on an** **actual device**. For the most part, how it behaves in the browser will be the same as the way it works ona real device, but there can be diﬀerences and in the case of any Cordova plugins, they won’t work when testing through a desktop browser.

A good approach is to do the majority of your testing in the browser, and once you’re getting close to being happy with how everything is working switch to testing on a real device to make sure everything still works correctly.

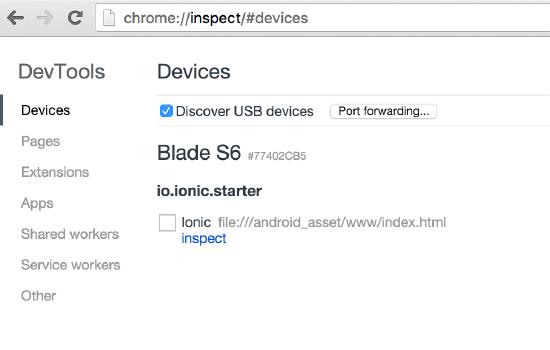
**Android Debugging**

Debugging on Android is as simple as attaching the device you want to debug on and then running:

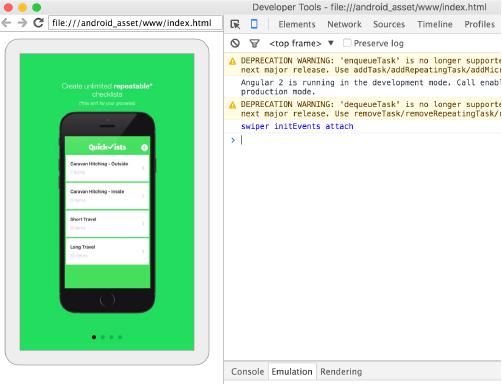
ionic run android

on your desktop browser (in Chrome) you will now be able to go to the following address:

chrome:*//inspect/#devices*



and then click ‘Inspect’ on the device you want to debug on, and you will see the debugging tools:

****

**IMPORTANT:** You will need to have USB debugging enabled on your device for this to work. EnablingUSB debugging is diﬀerent for diﬀerent devices, so just Google “[YOUR DEVICE] enable usb debugging”.

# References

# ‘Mobile App Development with Ionic 2’, *Christopher Griffith, 2017*

**Debug ứng dụng Ionic**

**(Chapter 10. Debugging and Testing Your Ionic Application)**

There will come a time when you need to debug your Ionic application. In this chapter, we will look at various solutions and tools you can use to help you resolve possible issues that may arise. Since Ionic is built atop a collection of technologies, this means that debugging your application may not always be straightforward. This can be a challenge for some developers who are used to having a complete ecosystem to work within. Let’s learn how to use some of the debugging tools.

Debugging your Ionic application can be broken down into three distinct phases: browser level, emulator debugging, and on-device debugging. Each one of the phases offers different levels of insight into your application. Since Ionic is built atop web technologies, often the best place to start debugging is with our browser. But even before turning to those solutions, there are some other first steps we can take.

With Ionic 2 being developed in TypeScript, this is actually our first line of defense in writing a working application. If your IDE is complaining about a variable or a function, that is usually a good sign that something is amiss. There are times when a reference is incorrectly updated when refactoring or a path to a module is off by a sub folder. Spotting these simple mistakes can save a lot of questioning on why your application is having issues.

Now, your IDE won’t be perfect in spotting everything while you are editing your code. The next element you can use in resolving issues is the compiler output itself. As you might have noticed, when compiling your Ionic 2 application, a lot of messages are written to your console’s window. If you are like most, this is probably just a small window, tucked in the corner of your screen. However, more than once have we seen error messages get written to this window, only to have them quickly scroll out of view.

If your application isn’t launching for some reason, the first place to start is to look at the build output to your console. You may find you missed a comma in an object or some equally hard to spot typo in your code. One recommendation we have is to have your console window actually show more than a few lines of output. This gives you the ability to see the various build events, and if an issue occurred.

Coming back to the fact that we are not working with complete IDE (like Xcode or Android Studio), some of the build debugging tools are not there to assist us. The most common one that we have found is that it does not recognize that there may be unsaved files in your project when you attempt to build your application. Often when working on a particular screen, you may have its HTML file open to adjust some markup, the SCSS open to modify the visual look, and the TypeScript file open as you edit the component’s code. It is easy to forget to save one or more of the files before you attempt your next build. Then as you test your application, you are scratching your head wondering why something is not working correctly. One of the simplest steps you can take as you start to debug your application is to ensure that all the files are saved.

A lot of your development will be in your local browser and using $ ionic serve to run your application. Now, one of the features of ionic serve is that it will watch your source files and trigger a rebuild when it sees that one of the files has been updated. This is very useful most of the time, but the Ionic build process takes a moment or two to run. There are times when the build scripts are running, and something causes them to run again before they finish their first build of cycle. This can result in a “white screen of death,” as it commonly known. In reality, the *main.js* file was incorrectly built or is missing altogether. Our recommendation when you encounter this issue is to make sure all your files are saved and run the build sequence again. In fact, we will stop our instance of $ ionic serve, and start fresh.

Once you have your application up in Chrome, Chrome’s standard Developer Tools (DevTools) will become your primary method of debugging your application. You might recall a bit of advice that we had about the order of development: try to focus on everything that you can do *not* to have to test or run on your mobile until you have to. The ability to quickly leverage Chrome DevTools to work with your application can save you hours. If you are coming from a traditional web development background, then these tools should need no introduction. But for those who are not familiar with them, we recommend reading up on them on Google’s developer site.

To bring up the DevTools, you can use the Chrome menu, found on the top right of the browser window, then select Tools→Developer Tools. Or you can simply right-click or Ctrl-click on a Mac, and select Inspect Element.

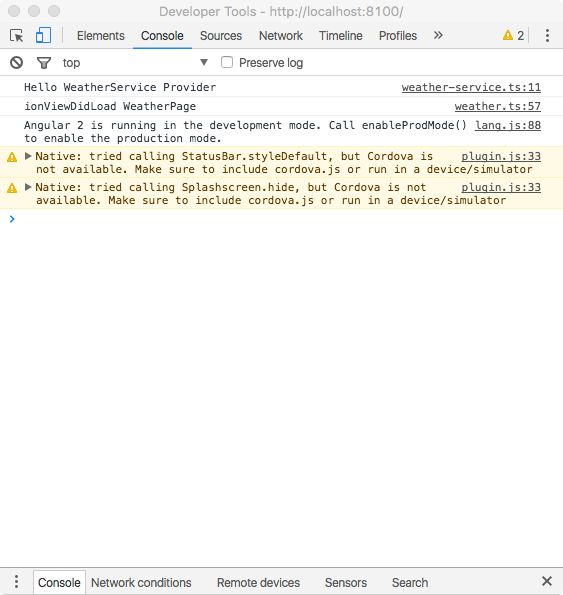
There are several useful shortcuts for opening the DevTools:

* Use Ctrl+Shift+I (or Cmd+Opt+I on Mac) to open the DevTools.
* Use Ctrl+Shift+J (or Cmd+Opt+J on Mac) to open the DevTools and bring focus to the Console.
* Use Ctrl+Shift+C (or Cmd+Shift+C on Mac) to open the DevTools in Inspect Element mode, or toggle Inspect Element mode if the DevTools are already open.

With these tools, we have a variety of options we can use to debug our Ionic application. If it is an issue related to the visual display of an element, we can navigate the rendered DOM to locate the element, then use the CSS inspector to uncover the cascade of styles that are being applied to that element. We often use this technique in understanding the base styling of an element, so we can then apply the smallest amount of changes to our Sass file to achieve the desired look.

But, more often than not, we are actually leveraging the JavaScript console and the accompanying debugger. Now the Ionic build system will combine all our source code into one file; but by leveraging the source map, we are able to reference the original sources for our code.

There are a few features in DevTools that are extremely useful for mobile developers (see Figure 10-1).



###### **Figure 10-1. Chrome’s device preview toggle**

Located in the upper left is the Device Toolbar, which will change your browser’s window to allow you to see how your Ionic application would appear on a mobile screen (Figure 10-2).



***Figure 10-2. The Ionic weather app running with Chrome’s device mode enabled***

This toolbar allows you to select a variety of popular mobile devices: Nexus 5x, Galaxy 5, iPhone 6, as well as the ability to define a custom device. For some devices, Chrome will automatically overlay the screen chrome (like a status bar or soft keys). You can also rotate the screen from portrait to landscape to see how the elements will respond.

# iOS Mode

Even though iOS screen sizes are listed, you will need to append ?platform-mode=ios to the end of the URL in order for your application to trigger its iOS display rules. Alternatively, you can use ionic serve --platform=ios to append that flag automatically.

Another useful capability in DevTools is the ability to simulate a variety of network conditions and speeds. To do this, select the Network tab. There you can adjust the network’s apparent speed. This is quite useful to make sure your application properly handles both the offline and online cases.

We have found that we are able to resolve 70%-80% of our issues just by working within DevTools and our browser.

# Dealing with CORS Issues

As you start expanding out your application, you will likely begin to work with a range of services and their APIs calls. These remote requests are usually a source of frustration, as they often require some level of authorization. One of the first levels of security with remote APIs is Cross-Origin Resource Sharing (CORS). This is a mechanism that gives web servers cross-domain access controls, which enable secure cross-domain data transfers. Modern browsers (like our Ionic App) use CORS in an API container, such as XMLHttpRequest or Fetch, to mitigate risks of cross-origin HTTP requests. Make sure you review the security and access information of the APIs that your application might use.

Because we are testing our application locally in a browser, these external requests will now be blocked. This is not an issue when testing in an emulator or on an actual device, but having to run in both an emulator and on-device can lengthen your testing cycle. There are two options to enable local development. You can set up a proxy like we did for the IonicWeather application. Or a faster option is to simply launch Chrome with its CORS disabled. Now this is something you should only be doing while you are testing your application. We are going to launch an instance of Chrome with CORS disabled via the command line.

For macOS users, the command will be:

open -n -a /Applications/Google\ Chrome.app --args --user-data-dir= "/tmp/chrome\_dev\_session" --disable-web-security

The -a flag opens the specified application. The -n flag will open a new instance of the application, even if one is already running. The two arguments that occur after the --args flag disable CORS and define a directory for user data.

For Windows users, the command is:

chrome.exe --user-data-dir="C:/Chrome dev session" --disable-web-security

Now when you use $ ionic serve, this version of Chrome that is running will allow your HTTP requests through.

# 

# Debugging through an iOS or Android Simulator

How do we tackle debugging issues that exclude using DevTools and our local browser? That is when we can turn to our device emulator. Although not a full substitute for a real device, it will allow a close approximation of our mobile environment. In order to run our Ionic application in this mode we need to use the $ ionic emulate <platform> command:

ionic emulate ios --consolelogs --serverlogs ionic emulate android --consolelogs –serverlogs

or the shorthand version:

# ionic run ios -c -s ionic run android -c -s livereload Flag

Ionic CLI used to support live reload through the --livereload or --l flag. However, some low-level changes in the security protocol caused this feature to stop working. The Ionic team is actively working on finding a solution.

Although we are now running within an emulator, we still have some other powerful debugging tools available to us. For macOS users, Safari will now become our tool of choice for finding and eliminating bugs. The first step is to enable the Develop menu in Safari. Go to Safari→Preferences→Advanced, then enable Show Develop menu in menu bar.

From this new Develop menu, we can now select our application running in the iOS simulator and debug it. Safari’s developer tools are very similar to Chrome Dev Tools — they enable inspection of the page’s DOM, the source files, and the JavaScript console.

## Android

Chrome also offers the same type of functionality. In the address bar, type **chrome://inspect**. This will display a list of devices that Chrome will be able to remotely debug. Once you have your application successfully running in the emulator it will be listed here. Now you can use the same Chrome DevTools to work with your application running within the Android emulator.

# Debugging On-Device

Just as the emulators are a step up from browser-based testing, the final level of testing should take place on an actual device. To do this, we need to build our Ionic application for installation on actual mobile devices. In order to run our Ionic application in this mode we need to use the $ ionic run <*platform*> command:

ionic run ios

ionic run android

This will produce either an IPA file (for iOS) or an APK file (for Android) that can be loaded and run on your device. Let’s look at each platform independently.

## Android

Before we load our application on the device, let’s enable on-device debugging for Android. This is done within the Developer Options setting, which is not enabled by default. To reveal this setting, open the Settings, then the About Phone sections. Next, tap seven times on the Build Number section. If you encounter an issue in locating this option, the specific steps for your device should be found via your favorite search engine. Once this option is unlocked, you can then enable Allow USB debugging.

There is one restriction: to use Chrome’s Remote Debugging, the device must be running Android 4.4 (KitKat) or later (aka API level 19). You will also need to be running Chrome 30 or later on your desktop.

Loading your application on your Android device can be done in multiple ways: you can upload the APK to a web server and download it, copy it to a service like Dropbox and install it from there, or even email the APK. Once the APK has been downloaded to your device, the Android installer should run, informing you of the permissions your application needs and then installing the app.

Just like debugging your Ionic application in the Android emulator, you follow the same steps for on-device debugging. Simply navigate to *chrome://inspect* in Chrome. With your Android device plugged into your computer and the Ionic application running, you should see your application listed. Selecting your application will then bring up the familiar Chrome Developer Tools.

If you can see your device in the inspect devices section but you can’t see the Cordova Web View, you may need to add android:debuggable="true" in the <application> node of your *AndroidManifest.xml*.

Now, we are able to fully interact with our application running in its native environment. For example, if your application used the device’s Camera, that functionality would not have been available in either the browser or in the emulator.

## iOS

Debugging your Ionic application on iOS follows a very similar path to Android debugging.

Just as a reminder, you will still need to sign this IPA with your development certificate and make sure your application ID, found within the *config.xml* file, matches what is listed in the development certificate. If they do not match, your IPA will not be properly signed and will not run on your iOS device.

If you encounter errors using the Ionic CLI to generate your IPA, you may need to open the actual Xcode project file and manually set the signing profile. Once this is set, the Ionic CLI should function normally.

With the IPA now built and installed on our iOS devices, connect your iOS device to your computer, then launch Safari on the desktop. Next, launch your Ionic application on your iOS device. From the Develop menu, you should see your iOS device listed. You may need to wait a moment while the WebView is located. Once it is selected, the same Developer Tools that you used with the simulator are available to you.

# Debugging Ionic Initialization

There are times when you are testing your Ionic application on either the emulator or live device and an issue occurs during the app’s initialization. Consequently, your app needs to be running before the remote debugger can locate and connect to it. Unfortunately, any console messages or actual errors will not be captured. With your application connected to a remote debugging session, switch to the console tab. Here we can force the WebView to reload itself, allowing us to view any issues that we missed. You can use the Reload button in the Dev Toolbar or simply enter this command into the console input field:

window.location.reload();

Either will force the WebView to reload itself, and since your remote debugging is already connected, you will be able to see what the issue might be.

# Additional Tools

Beyond these debugging solutions, there are several other options that you should be aware of that can assist your Ionic development:

*Cordova Tools Extension for Visual Studio Code*

Available for Visual Studio Code, the Cordova Tools Extension allows for code-hinting, debugging and integrated commands for Apache Cordova (PhoneGap). As an added bonus, it also has support for the Ionic Framework (albeit version 1). We have found this extension invaluable during our Ionic development workflow.

*Tools for Apache Cordova (TACO)*

If you use Visual Studio as your IDE for Ionic development, then you should install Tools for Apache Cordova, or “TACO” for short. This extension provides IntelliSense, debugging, and build support for Apache Cordova and Ionic projects; and unlike the Cordova Tools Extension, TACO does have beta support for Ionic 2. Visual Studio can also debug your code running on an iOS, Android, or Windows device, an emulator or simulator, or a browser-based debug target like Apache Ripple. An additional capability of TACO is its ability to build for Mac OS remotely via services like MacInCloud. *GapDebug*

Genuitec’s GapDebug is another debugging solution that has proven useful in our workflow.

Available for both Windows and Mac platforms, this tool offers easy installation application files onto your device using drag-and-drop. It has integrated versions of the Safari Web Inspector for iOS debugging and Chrome Dev Tools for Android debugging. You are also able to debug both iOS and Android apps whether you are running Windows or macOS. Unfortunately, Genuitec recently annouced that they will be ending development of this product.

*Ripple*

Ripple is a desktop-based emulator for Cordova projects. Essentially, it lets you run a Cordova application in your desktop application and fake various Cordova features. For example, it lets you simulate the accelerometer to test shake events. It fakes the camera API by letting you select a picture from your hard drive. Ripple lets you focus on your custom code so you can spend less time worrying about Cordova plugins.

*Augury*

# Augury, formerly known as Batarangle, is a Chrome Developer Tools extension that allows developers to visualize their Angular 2 application’s component tree and the data associated with it. Support for this solution is still emerging as Ionic 2 matures.

# Summary

As you can see, there is a wide range of solutions to debug your Ionic application across a wide range of testing environments. Depending on the stage of development, you should find the right set of tools to enable you to resolve any issues that might arise.