9. Refactoring and Optimisation

9.1 Introduction

This chapter discusses steps taken to refactor the source code and how the app was optimised.

9.1.1 Background

After integrating the app with the API, the author proceeded to test the app on his Android device (HTC One M8s, Android version 5.0.2). In order to install the app on his device, the author had to download Android’s SDK and install the necessary components; and set the “allow USB debugging” option to true in “Developer Options” settings. Running “ionic run android” terminal command installs the app on your device. Debugging the app while testing on device was achieved by using Chrome. Visiting ‘chrome://inspect/#devices’ URL enables you to debug the app as if you were testing on a browser i.e. you can inspect the app’s HTML tags and view the console.

The results of testing on the author’s devices are as follows:

1. The app feels like a website and not a mobile app: the pre-loader primarily caused this. A pre-loader was used when making Ajax calls and while useful in some cases e.g. booking a session, it wasn’t useful when navigating between pages e.g. navigating from the “home tab” to the “coaches tab” to the “settings tab”.
2. The app froze, sometimes, when scrolling up and down a list e.g. list of coaching sessions
3. The settings page took a considerable time to open

These were the reasons behind the need to optimise the app. But optimising the app required the reuse of some codes and hence the need to refactor.

9.2 Refactoring

Refactoring mainly involved removing Ajax requests from the controller and placing them in a service or factory depending on the function of the Ajax requests. Ajax requests fetching a list of objects e.g. pathways, coaching sessions, coaches etc. were placed in factories whilst those performing a function e.g. booking a session, editing a profile etc. were placed in services.

With these criteria in mind, lets take a look at how the “pathwaysFactory” factory is used.

9.2.1 “pathwaysFactory” factory

In fig 9.1 on line 376, a deferred object is created; this deferred object is used to return a promise if the Ajax request fails or succeeds. The ‘pathwaysFactory’ factory is used in a similar way: if the request is successful then a set of actions is taken, but if it fails another set of actions is taken.

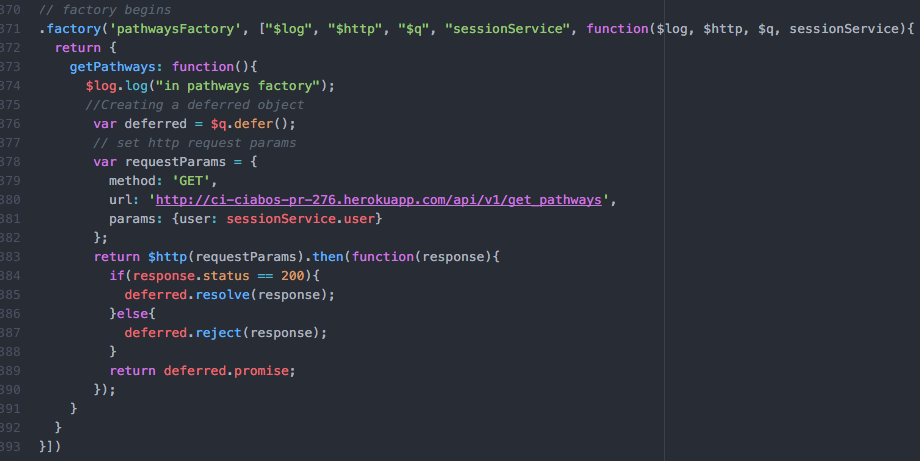


Fig 9.1: ‘pathwaysFactory’ factory

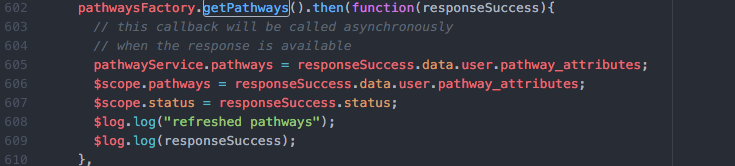


Fig 9.2: ‘pathwaysFactory’ used in the ‘HomeTabCtrl’ controller

9.3 Optimisation

This section will discuss how issues raised in the ‘Background’ section were resolved.

1. Feels like a website issue: Using the ‘coaches tab’ as an example, the author wrapped the ‘coachesFactory.getCoaches’ function in a ‘getCoaches’ function (fig 9.3).

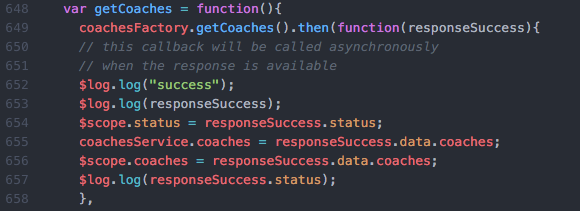


Fig 9.3: coachesFactory

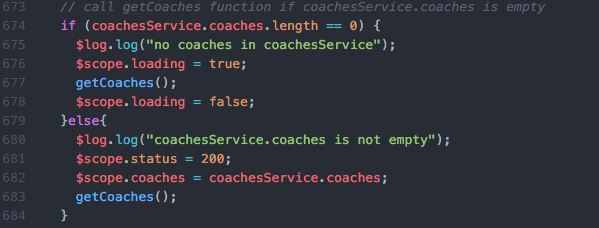


Fig 9.4: using ‘coachesService.coaches’ object

The effects of the code block in fig 9.3 and fig 9.4 are:

* when you visit the ‘coaches tab’ for the first time (fig 9.4, line 674), the app fetches an array of coaches object from the API; stores the array of coaches in a coachesService’s coaches object (fig 9.3, line 655) and displays the list of coaches (fig 9.3, line 656).
* when the ‘coaches tab’ is subsequently visited (fig 9.4, line 679), the array of coaches object is fetched from the coachesService’s coaches object and displayed (fig 9.4, line 682) whilst the app fetches an up-to-date list from the API.

The resultant effect is that rather than waiting for the up-to-date list of coaches, the app quickly displays the list of coaches the app has stored and asynchronously updates the list of coaches. This is consistent with how apps work and provides a good user experience!

2. App froze, sometimes, when scrolling up and down a list

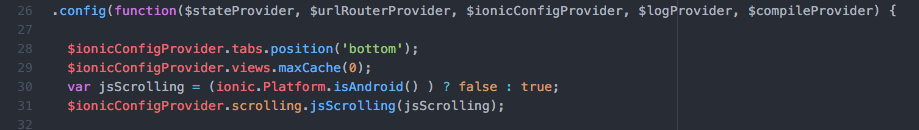


Fig 9.5: disable jsScrolling

The solution lies in enabling native scrolling. Because the freezing issue that occurs when scrolling is specific to the Android platform, native scrolling is only enabled when the platform is Android (fig 9.5, lines 30 to 31).

3. Considerable loading time of the settings page

The settings page took a while to load because of the number (582) of timezones in the timezones array returned by the API. Ionic’s “ng-repeat” is known to suffer performance hits when the number of objects is a lot. Ionic’s “collection-repeat” performs better than “ng-repeat”.

Using the ionic-modal-select plugin (<https://github.com/inmagik/ionic-modal-select>) resolved this issue. The plugin uses “collection-repeat” to iterate over collections. The author also had to re-design the “Edit Profile” page: instead of displaying a drop-down menu of timezones, a button that showed the list of timezones when selected was added.

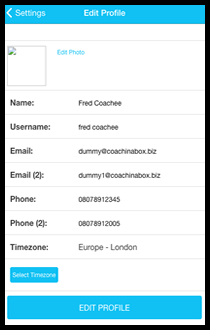
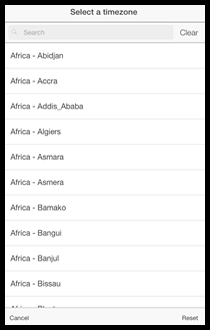
 

Fig 9.6: New Edit Profile Page Fig 9.7: Select a timezone

9.4 Conclusion

After refactoring the source code, the architecture felt more like a Model-View-Controller architecture. Optimising the app improved the performance of the app and greatly increased the user experience. Further optimisation is needed for page transitions and further tests in older versions of Android are required.