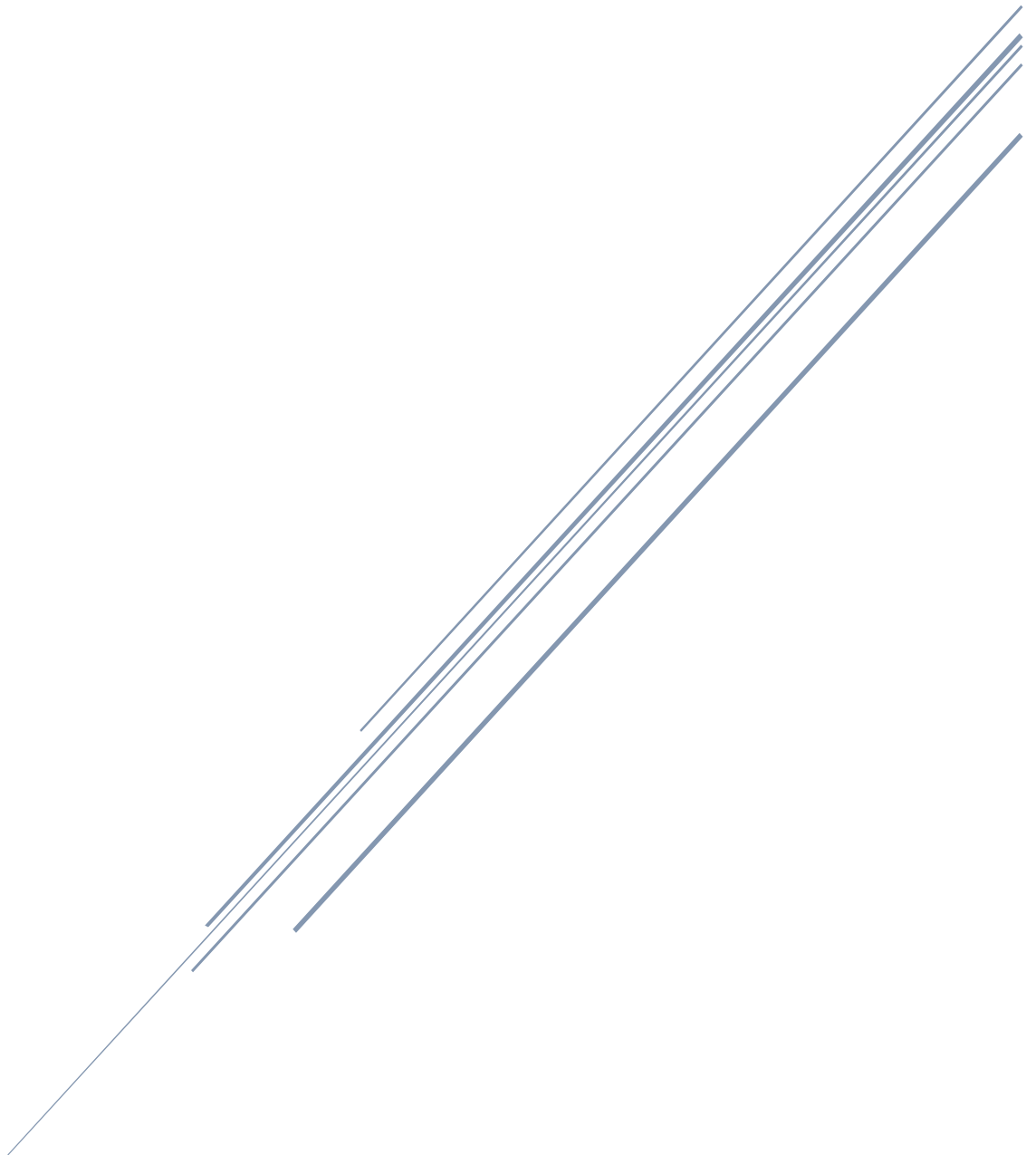


# MAKERS ACADEMY, PORTFOLIO SUBMISSION

No. 2



Placement: Compare the Market  
Apprentice: Krzysztof Kasprzak

## Table of Contents

<b><i>Fix loss of information on RS refresh task</i></b> .....	<b>2</b>
<b>The task</b> .....	<b>2</b>
Project and the framework I worked on .....	2
The ticket, it's requirements and how I got involved.....	4
Functionality of Run Scenarios and Compare with Live .....	7
<b>Steps to complete the task</b> .....	<b>7</b>
Allspark environments and recognising if bug is reproducible locally .....	7
Finding the relevant html view and its controller .....	8
Getting visibility and assessing the presentational component .....	10
Exploring Run Scenarios controller and repairing page title .....	13
Loss of provider details and the solution .....	15
Navigating away from RS and CWL views after refresh .....	18
Testing the impact on existing test suites .....	19
Unit testing the changes made .....	21
<b>Integration and feature testing, raising and resolving another ticket</b> .....	<b>24</b>
Integration and feature testing .....	24
Raising a ticket to fix test pipeline .....	25
Pipeline fixing .....	25
<b>In conclusion</b> .....	<b>28</b>
Things I learned and improved upon during the work.....	28
Things I would do differently next time .....	28
Value I brought and the impact I made .....	29

## Fix loss of information on RS refresh task

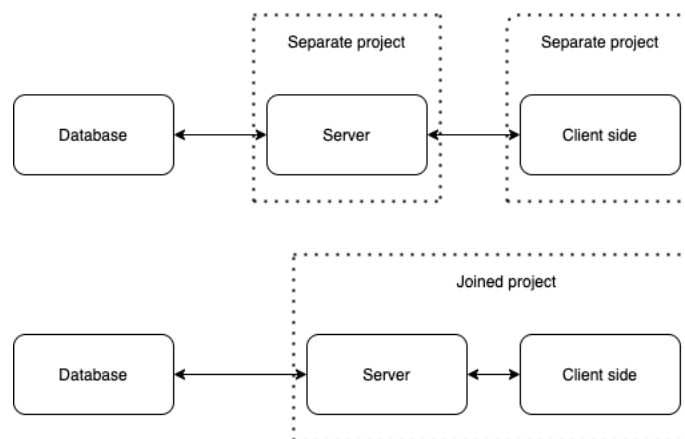
### The task

Project and the framework I worked on

As I mentioned in my first portfolio submission, I am a member of Allspark team. With that, I am not only responsible for rewriting Allspark UI (user interface) from Angular JS to React JS; I am also involved in maintaining, patching and bug fixing the existing Allspark until it will be discontinued.

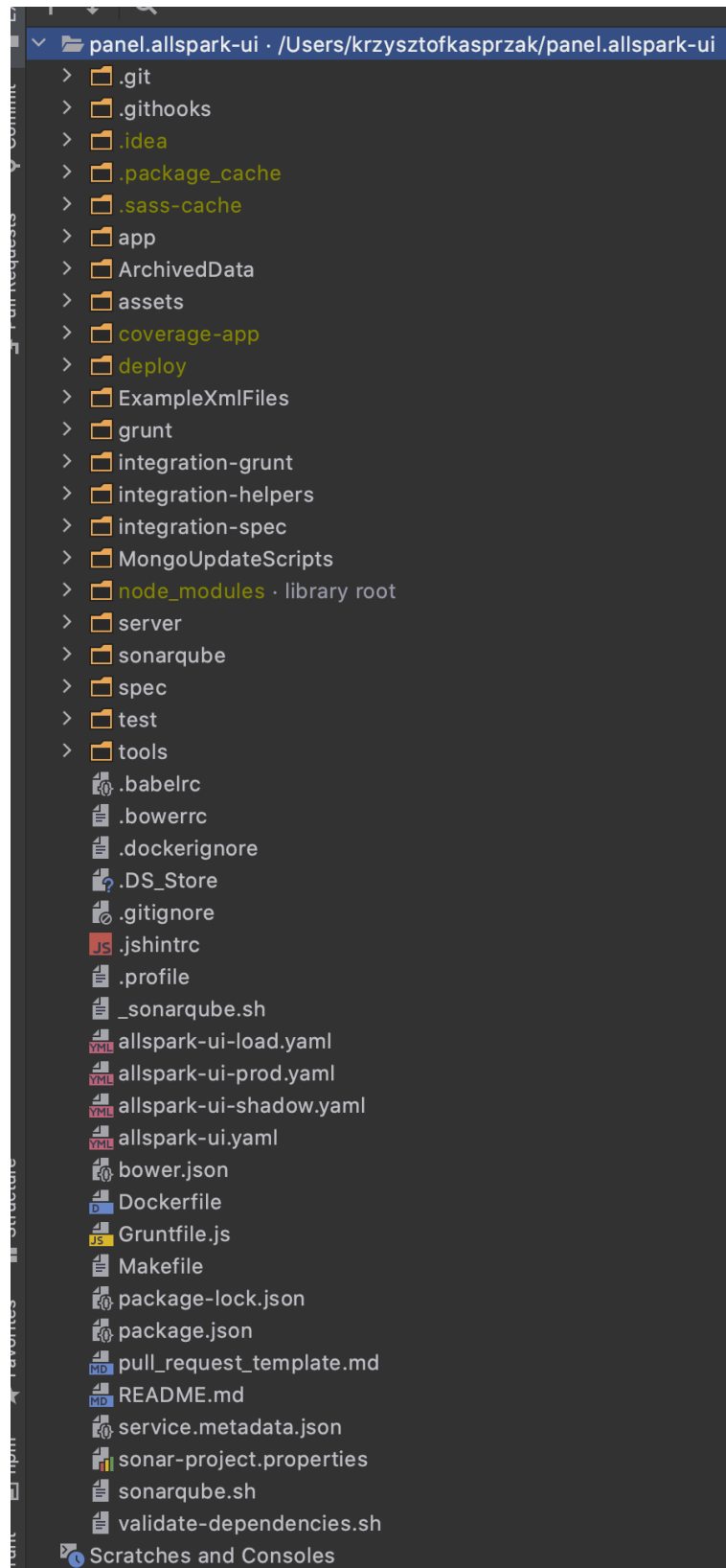
The task I am about to describe here, is set around the work on the front-end part of existing Allspark UI project, using Angular JS framework.

To elaborate, unlike the upcoming react version of Allspark, current Allspark UI project, encapsulates `app` part, which is meant to be run and executed on the client side (in the user's browser), and the `server` part, which is responsible for consequently calling and manipulating data in database (see fig. 1.1 below for graphical representation of that difference).



**Fig. 1.1 diagrams the difference in how the react Allspark UI is going to be kept (the upper diagram); i.e., server side will be kept in single project and deployed as such and will be receiving queries from client side (front-end) to execute changes on a standalone database. Whereas current Allspark UI with its frontend written in Angular JS, has the server capabilities kept in a one project and deployed together with client side.**

The figure on the next page is showcasing the root repository (fig. 1.2) in which `app` and `server` directories can be found.



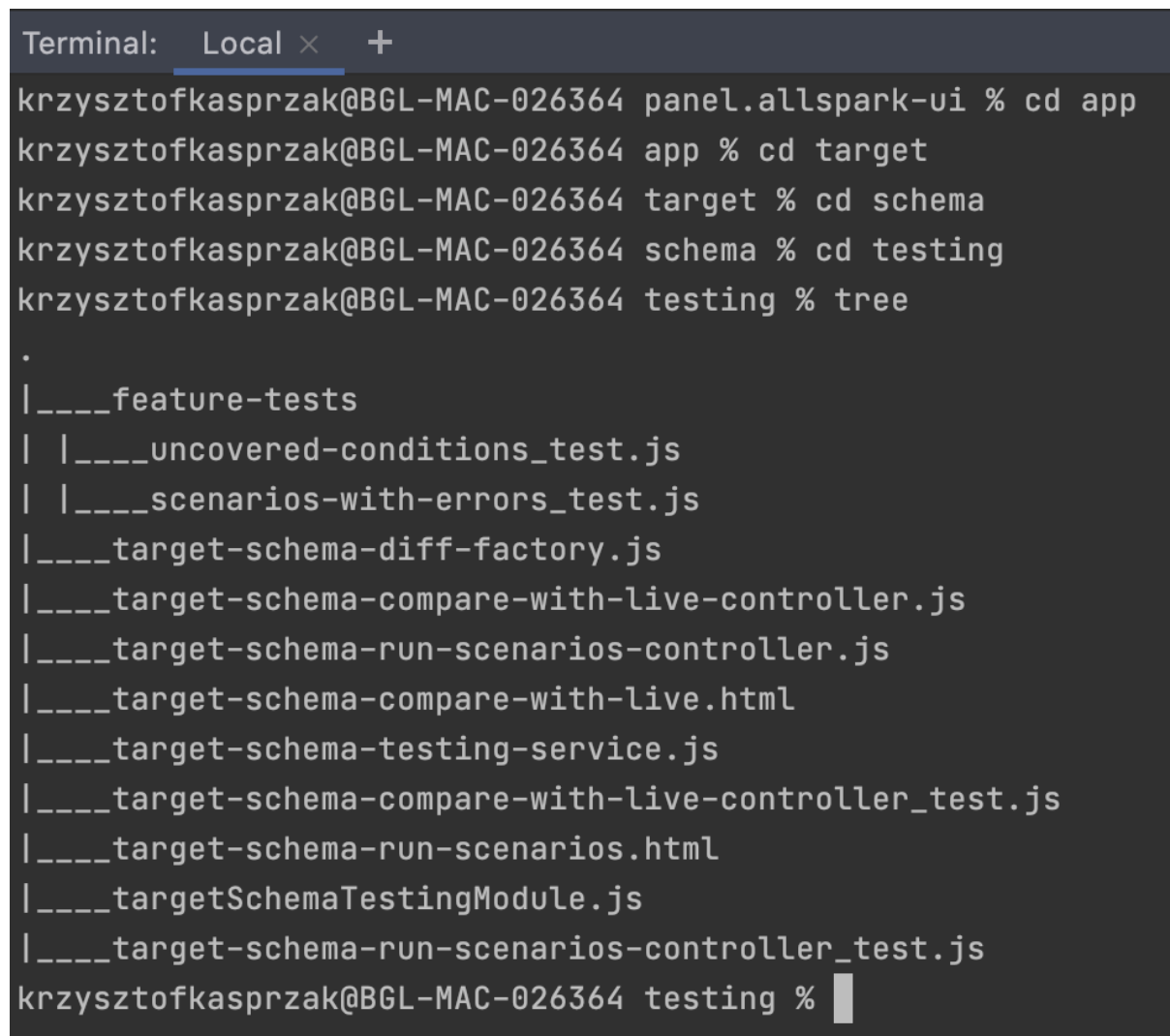
**Fig. 1.2, depicts the first layer of root directory of the project I worked with. For the purpose of the ticket, I specifically worked within `app` directory. The screen shot was taken from the explore bar in JetBrains Rider; an IDE I use at my placement.**

The next figure, (fig. 1.3 below) depicts the structure of `testing` sub-folder, and how it is nested in the `app` folder.

I have created the outcome seen on the figure by setting up and using an alias command in terminal window. Alias commands are user specified (i.e., developer) commands that are equivalents to specific bash (in terminal) commands. Advantage of using such commands is streamlined efficiency. As with a `tree` command I created, it was done by declaring:

```
alias tree="find . -print | sed -e 's;[^\/*];|____;g;s;____|; |;g'"
```

Which allows me to print file tree diagrams to terminal by simply typing `tree` not a whole command of finding and printing its contents.



```
Terminal: Local x +
krzysztofkasprzak@BGL-MAC-026364 panel.allspark-ui % cd app
krzysztofkasprzak@BGL-MAC-026364 app % cd target
krzysztofkasprzak@BGL-MAC-026364 target % cd schema
krzysztofkasprzak@BGL-MAC-026364 schema % cd testing
krzysztofkasprzak@BGL-MAC-026364 testing % tree
.
|____feature-tests
| |____uncovered-conditions_test.js
| |____scenarios-with-errors_test.js
|____target-schema-diff-factory.js
|____target-schema-compare-with-live-controller.js
|____target-schema-run-scenarios-controller.js
|____target-schema-compare-with-live.html
|____target-schema-testing-service.js
|____target-schema-compare-with-live-controller_test.js
|____target-schema-run-scenarios.html
|____targetSchemaTestingModule.js
|____target-schema-run-scenarios-controller_test.js
krzysztofkasprzak@BGL-MAC-026364 testing %
```

**Fig. 1.3, shows the steps I had to go through to get to the relevant directory in which I worked the most, I needed to traverse: panel.allspark.ui (the root repository) -> app -> target -> schema -> testing . Then after applying my custom `tree` command, it printed a full tree diagram of what's in testing directory. The unusual look of the terminal is due to my use of IDE's tool, which is simply a terminal feature. It allows the user (developer) to open up a terminal which automatically navigates into the project's root folder. Hence, the snippet doesn't include steps of navigating (using `cd` command) into panel.allspark.ui.**

The ticket, it's requirements and how I got involved

The task serial number was AL-2721 and it was labelled as a bug to be fixed. AL part of the number states it is a fix to be done for Allspark team, and the number is just an incrementally generated number of an issue/ticket.

I got involved in working on it in a plain way. It just so happened to be at the top of priority list for tickets in the sprint, once Tim (my pair partner at the time, the same developer as I worked with during the creation of Proof-of-Concept from previous submission) and I completed previous tasks. However, we picked it up just a day before he went off for his annual leave, leaving me in charge of working my way through the bug. As can be seen on the figure below (fig. 1.4), I was the main assignee.

The screenshot shows a Jira ticket interface. At the top, the ticket ID is AL-2465 / AL-2721. The title is "Fix loss of information on RS refresh". Below the title are icons for attachments, comments, and a checkmark. The "Description" section contains the following text:

**ORIGINATOR:** Sarah [redacted]

**REQUIREMENT:**

As a user I want the function and header information to remain once a refresh of RS or CWL is run so that I do not need to reselect the provider in order to continue with further change.

Once it is run, do not go back to root menu - stay in the position it was when RS or CWL was run

**USER BENEFIT:**

To minimise the time and effort required to make changes

**EXAMPLE:**

Policy Expert screenshots attached.

Currently, if RS or CWL is run and then the screen refreshed later in order to re-run, the PA title disappears from the heading and all but the 'Provider' option disappear so the PA must be

At the bottom of the description is a comment box with the user "KK" and the text "Add a comment...". Below the comment box is a tip: "Pro tip: press **M** to comment".

On the right side of the ticket, there is a metadata panel with the following fields:

- Done:** Done
- Assignee:** Krzysztof Kasprzak
- Reporter:** Sarah [redacted]
- Labels:** Efficiency
- Buddy:** None
- Signed-off-By:** None
- Technical Sign Off:** Sharon [redacted]
- Story Points:** 0
- Fix versions:** Allspark 11/03/21
- Debt Size:** None
- Tech Debt Type:** Not Debt
- Epic Link:** Usability
- Sprint:** None +2
- Priority:** Medium

**Fig. 1.4, snapshot of a ticket documentation that I completed for this project.**

The screenshot shows the details of the ticket, specifically the "Requirement", "User Benefit", "Example", and "Acceptance Criteria" sections.

**REQUIREMENT:**

As a user I want the function and header information to remain once a refresh of RS or CWL is run so that I do not need to reselect the provider in order to continue with further change.

Once it is run, do not go back to root menu - stay in the position it was when RS or CWL was run

**USER BENEFIT:**

To minimise the time and effort required to make changes

**EXAMPLE:**

Policy Expert screenshots attached.

Currently, if RS or CWL is run and then the screen refreshed later in order to re-run, the PA title disappears from the heading and all but the 'Provider' option disappear so the PA must be reselected in order to proceed with further change.

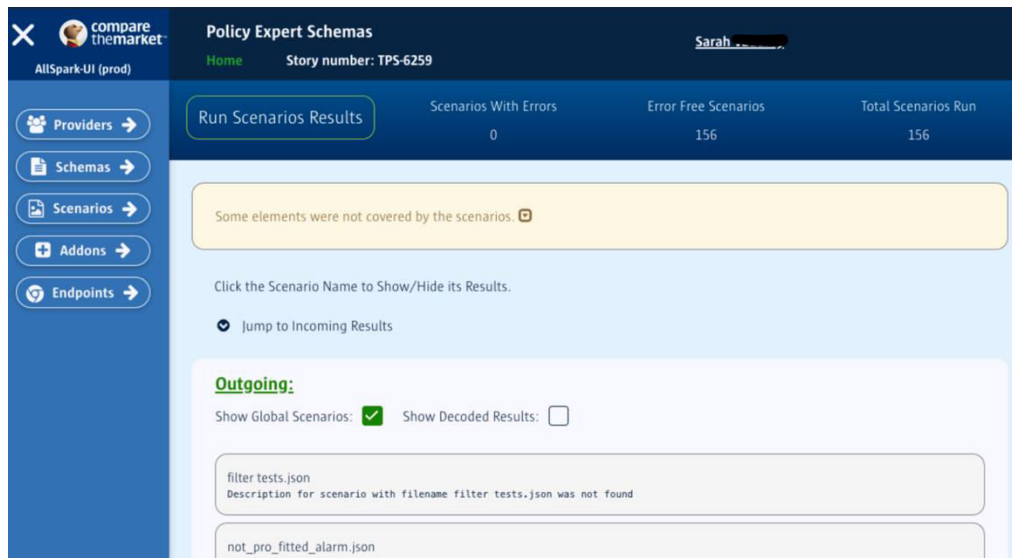
**ACCEPTANCE CRITERIA:**

1. Refresh of RS or CWL is performed and Provider name remains in heading
2. Refresh of RS or CWL is performed and all options remain in left side menu

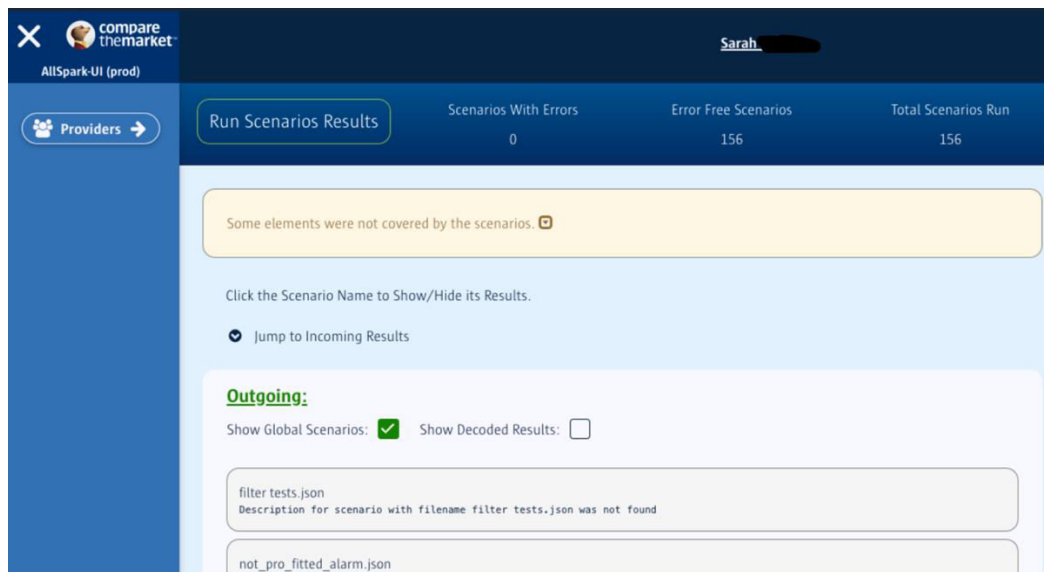
Or provide a refresh button

**Fig. 1.5, details of the requirements, benefits, description of a problem and acceptance criteria.**

As seen on the fig. 1.5 (on the previous page), The problem was happening whenever a user would refresh page while running scenarios (RS) or comparing with live (CWL) chosen provider product.



**Fig. 1.6, depicts how the Run Scenarios view should and looks like whenever user would run it for a chosen product provider, in this case Policy Expert. Apart from presenting scenarios outcomes (centre of the web page), it should include aforementioned provider name, product type (green 'Home' notion) and the story number. It also should allow the user to switch between, schemas, scenarios, addons and endpoints in the left-hand side, dropdown menu.**



**Fig 1.7, show results of page refresh applied to Run Scenarios (with exact same outcome on Compare With Live side) display. As can be appreciated, no information about the provider and the product persists and the user is no longer able to browse provider specific options in the left-hand side menu. He/she is forced to go and re-browse provider from providers.**

With the problem occurring (see fig. 1.6 and 1.7 on the previous page for the depiction of how the bug affects the user experience), advised solution was either to make sure that:

- This data and functionality loss is not happening while refreshing, or
- Custom Refresh button is added to the app, that allows users safely refresh scenario runs and live comparisons.

Out of these two proposed UI (made by users, who in this case acted as UI designers, since they were the ones to advice for best UI experience) changes, Tim and I were leaning towards attempting to fix it by simply persisting the data upon refreshes. This way it would spare us problem of introducing new presentational component (refresh button), its logic, and its testing.

Having understood the ticket briefing, I was ready to make a start on investigating the problem and where it might be occurring.

#### Functionality of Run Scenarios and Compare with Live

The bug affected views give the user following functionalities:

- Run Scenarios:

It allows the user to assess if existing product provider details can be used to build a provider adapter (json data file) which can successfully be mapped into an xml file and send back to product provider for purposes of their analysis. In order to test it, Run Scenarios feature conducts a series of randomised tests on data to check if it all fits into an acceptable pattern.

- Compare with Live:

Similarly, as Run Scenarios, it also tests if the existing product provider details are suitable to be mapped into an xml file. However, it does it by comparing the provider details to the latest instalment of provider details that were recorded in a data cluster in Amazon Web Services (AWS) cloud storage. Once compared, differences are presented to the user.

#### Steps to complete the task

##### Allspark environments and recognising if bug is reproducible locally

While approaching this task, I made sure to apply all the problem solving and investigation steps/techniques I have either developed naturally, learned at Makers Academy or during my ongoing placement.

The project is vast, and has three different production environments:

1. Locally, can be run by developer on `localhost` (http port that can be accessed on a local machine when the program is being active in the development mode) ports that are being contained in containers ran by docker. Docker is a platform as a service (PaaS) software that allows users to contain programs (e.g., web apps) in containers; virtual machines that run their programs in isolation from any other processes that are taking place on the user machine, making sure that no outside process would interrupt the app. Running Allspark UI as a dockerised app locally also better emulates its behaviour higher up, in shadow and production environments, as it is CtM's (Compare the Market) technological standard to deploy and host contained services on Amazon ECR (Electric Container Registry) as Kubernetes clusters (set of containerized apps) which are being pushed there by use of GoCD Pipelines.
2. Shadow, environment which is live and kept on Amazon ECR. It is technically a mirrored service of Production environment. The only difference is that Shadow is for developers to interact with it without needing to interfere with Production build, making sure that any maintenance or development advances can be safely checked in live settings without obstructing user experience.
3. Production, as the name suggests environment in which most update, user available product is running. Same as Shadow, it also is kept in Kubernetes cluster on Amazon ECR and is being updated thorough GoCD Pipelines.

With that, first, rather obvious way to tighten the bug search effort I applied, was to check in which environments the bug actually happens. This simple step of checking where the bug can be reproduced allows to potentially eliminate any external causes, like users creating providers with particular schemas, or some infrastructure problems.



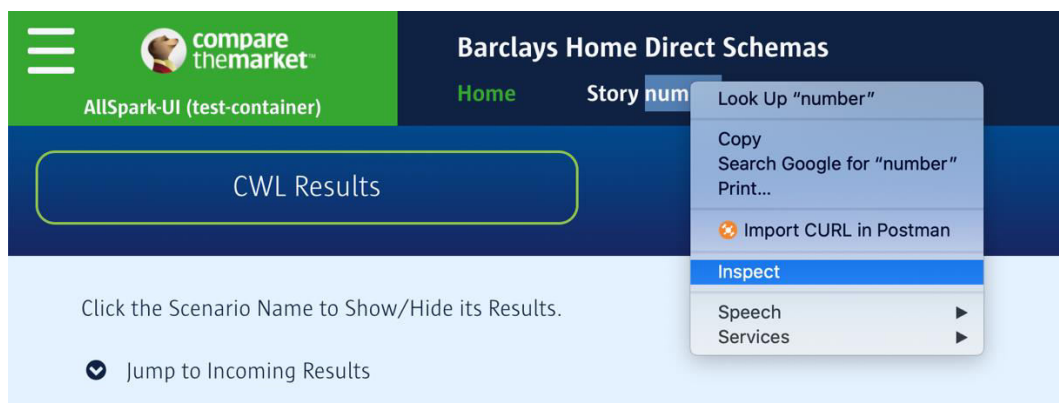
It did result in a such conclusion this time. I was able to reproduce the bug in all three environments. This was good for two reasons:

- The problem was clearly within the codebase, and not caused by any factors outside my control.
- As it also existed while ran locally, it allowed for easier debugging and testing; there was no need for me to be pushing and checking changes in Shadow environment or simulating new situation locally via schema stubs (as the local version doesn't connect anywhere, its providers are being artificially replaced to imitate real life cases). Which could be the case if that would be caused by some specific provider schema whose conditions would not be previously stubbed in a local version.

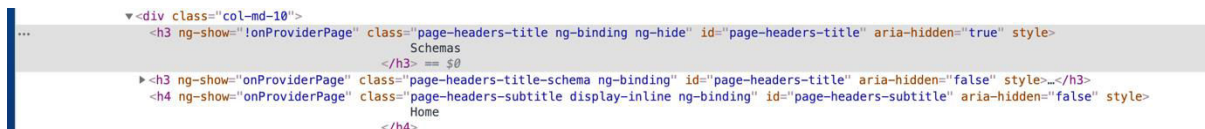
Additionally, combining this finding with knowledge that the bug only affects the left-hand side menu and the header info, not obstructing RS and CWL process on page refresh (as seen on [figs 1.6 and 1.7](#)), it was apparent that I shall look into the front-end implementation of these elements upon choosing scenario runs or comparison with live. That was my next step of the process.

#### Finding the relevant html view and its controller

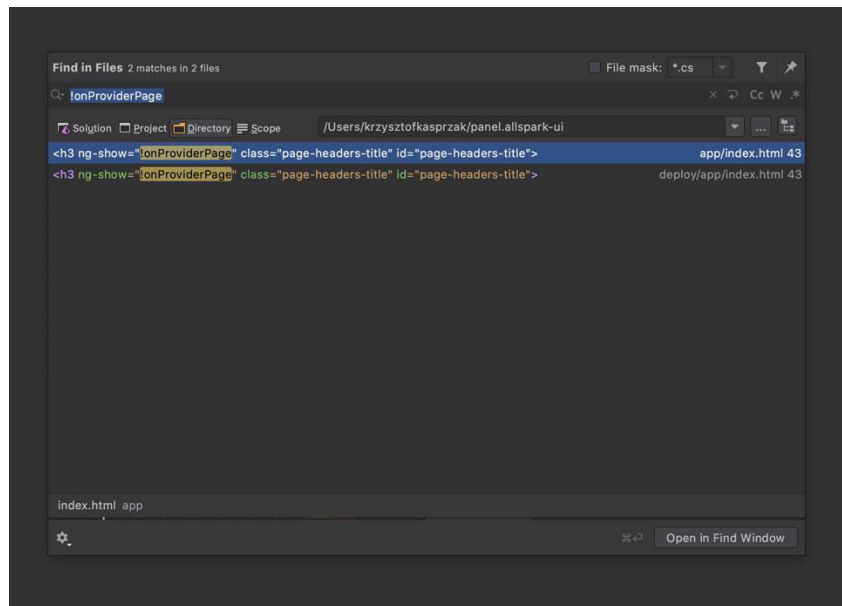
With knowledge gained from just an initial glance at the code, I then went on and used Chrome developer tools (browser build in functionality that helps analysing the source code of web pages and web apps) to inspect the header html source code (source code is a code that is being read by the browser to display and control the web page). Figures below (1.8, 1.9 below and 2.1, 2.2 on the next page) evidence how I used Chrome developer tools and search functionality in IDE to find html file and its corresponding controllers (files that hold the logic for displayed html file); all thanks to being able to follow the flow control of code.



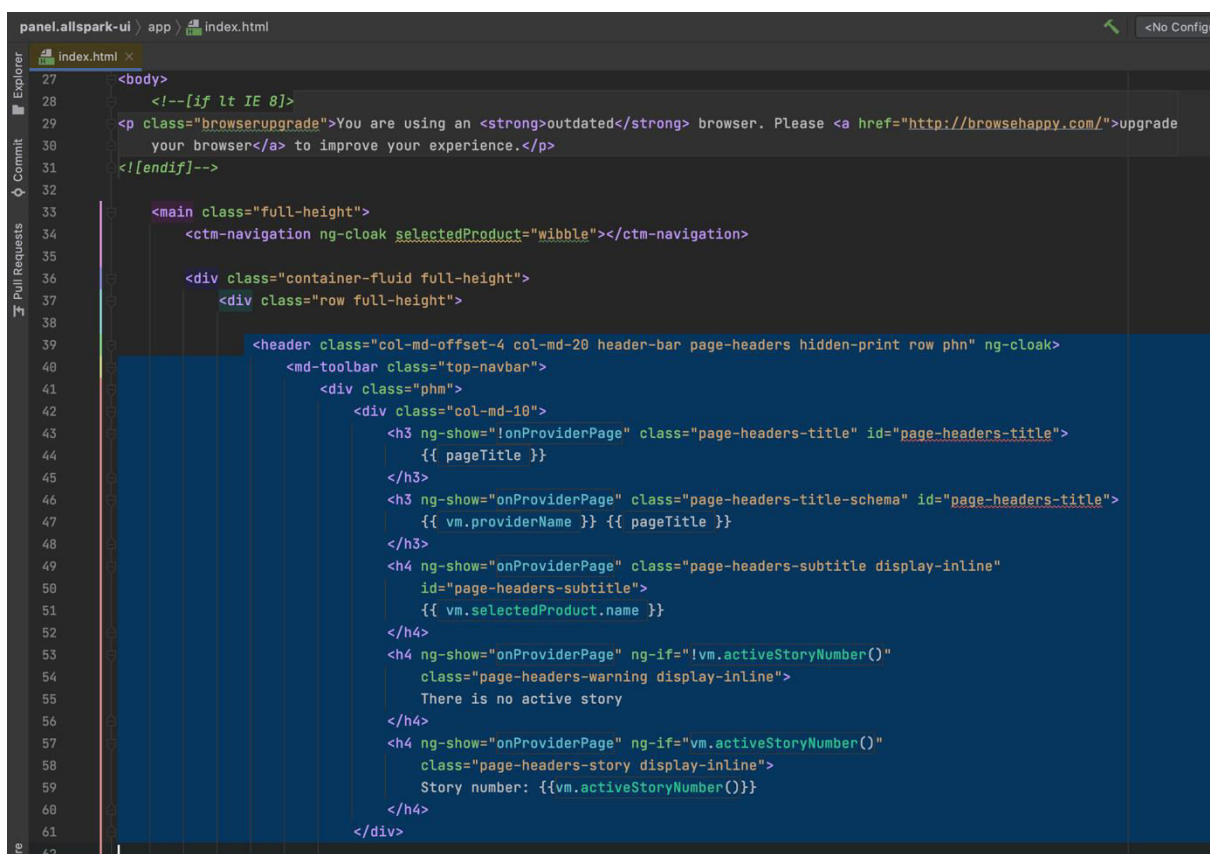
**Fig. 1.8 shows how I can use built in developer tools in Chrome browser to right click and inspect displayed html elements. I did so to try and expose the html source code for the text displayed in the header.**



**Fig. 1.9, displays the code snippet I tracked down by inspecting the header text elements. Highlighted piece (in grey) is responsible for displaying `Schemas` text, upon user being in a Schemas menu site. I used the snippet to try and search its presence in the project, to find relevant html file.**



**Fig. 2.1** is a snippet of how I can use my IDE to search through the project repository in search of code pieces. To use this functionality, I use certain key shortcut which is: **command+F+Shift**. I paste the element found in the source code `!onProviderPage` and it found that it only is being used in `index.html`, this was my presentational component for the header.



**Fig. 2.2**, snippet of a code from `index.html` file that is responsible for displaying the header information that was disappearing after refreshing the page in RS and CWL windows.

The index.html file also revealed its controller, it simply was the mainController.js (see snippet of code below).

```
<!doctype html>
<html class="no-js" lang="en" ng-app="allsparkApp" ng-
controller="mainController as vm">
```

**Fig. 2.3, the very beginning of index.html file. It states that the controller is mainController class, held in main-controller.js, in the same `app` folder as index.html.**

Having found these two components in the code base, I was ready to inspect them closer to find where the information on provider details potentially leaks.

Getting visibility and assessing the presentational component

Looking at the code in index.html (fig. 2.2 on the previous page), I made sure to follow a lead down the line. As I've seen how my colleagues are doing debugging before; I have made sure to check if it is the html's fault or the controller's.

That's a natural way of structuring your debugging technique; make sure to work from presentation down to logic, step by step. This way should the presentational component be faulty, it saves time (as these components are simpler in nature), and generally that would be an easier fix.

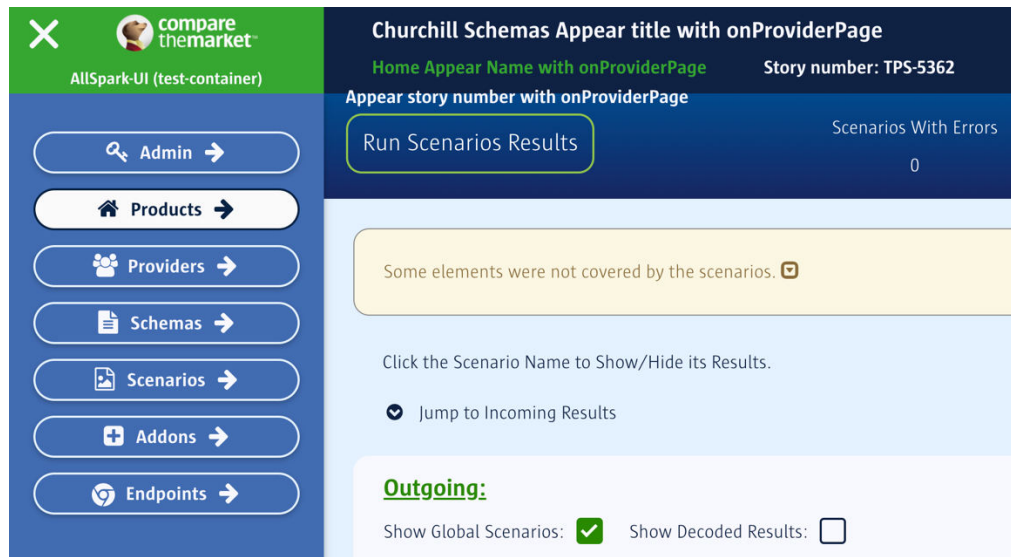
To do so, I have temporarily modified the html header text, to directly reflect what information it is giving me. This gives more visibility on how data from controller is being passed to the html file (see fig. 2.4 below and its description).

```
<div class="container-fluid full-height">
  <div class="row full-height">

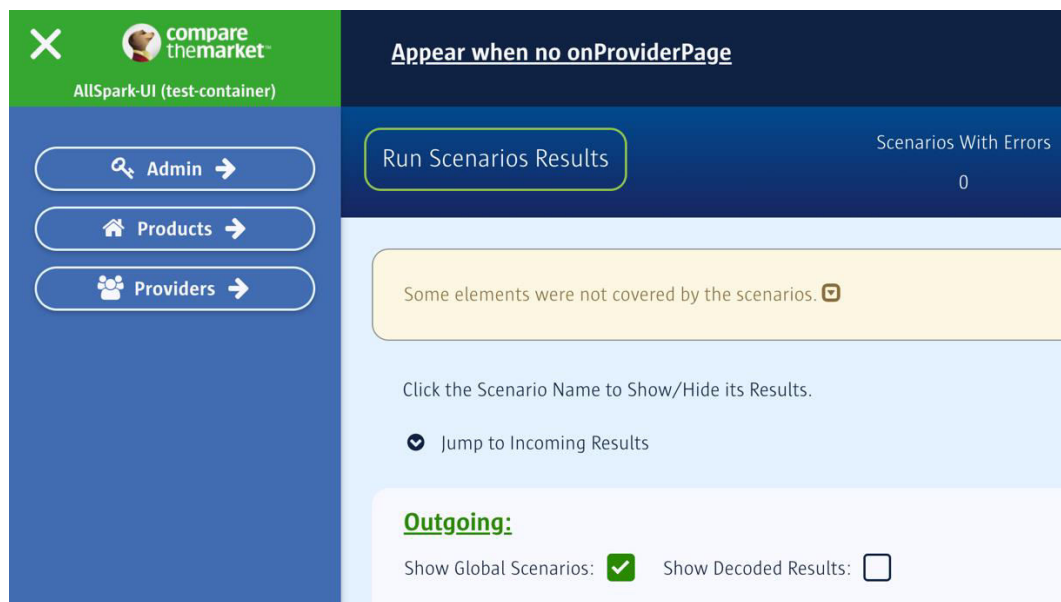
    <header class="col-md-offset-4 col-md-20 header-bar page-headers hidden-print row phn" ng-cloak>
      <md-toolbar class="top-navbar">
        <div class="phm">
          <div class="col-md-10">
            <h3 ng-show="!onProviderPage" class="page-headers-title" id="page-headers-title">
              {{ pageTitle }} Appear when no onProviderPage
            </h3>
            <h3 ng-show="onProviderPage" class="page-headers-title-schema" id="page-headers-title">
              {{ vm.providerName }} {{ pageTitle }} Appear title with onProviderPage
            </h3>
            <h4 ng-show="onProviderPage" class="page-headers-subtitle display-inline"
              id="page-headers-subtitle">
              {{ vm.selectedProduct.name }} Appear Name with onProviderPage
            </h4>
            <h4 ng-show="onProviderPage" ng-if="!vm.activeStoryNumber()"
              class="page-headers-warning display-inline">
              There is no active story Appear with onProviderPage
            </h4>
            <h4 ng-show="onProviderPage" ng-if="vm.activeStoryNumber()"
              class="page-headers-story display-inline">
              Story number: {{vm.activeStoryNumber()}} Appear story number with onProviderPage
            </h4>
          </div>
        </div>
      </header>
    </div>
  </div>
```

**Fig. 2.4, I change index.html temporarily and added some comments to text components to check if `onProviderPage` element is being returned from controller on page refresh and see if it is only a problem of that or is there any issue with how the header loads (if possibly `onProviderPage` would load correctly). For example, with changes made I expected that if there is a problem with controller and lack of loading `onProviderPage` element, then only "Appear when no onProviderPage" would appear in the header. Otherwise, if that would be correct, I expected to see further "Appear" messages, but potentially without provider details.**

With the changes described on the previous page (fig. 2.4), I went on and checked the outcomes. From them, it was clear that it is not the fault of index.html; it was `onProviderPage` element that was not being reloaded on the page refresh (see figures 2.5 and 2.6 below for the screen captures of results).



**Fig. 2.5 is Run Scenarios view (same results were obtained for CWL view) of some provider selected, which displays everything in the header, along with my temporary additional comments, which as they should, show that header text behaves as it should, with `onProviderPage` element loaded correctly.**



**Fig. 2.6, shows Run Scenarios view after page refresh. As it can be seen, the changes I made clearly show that `onProviderPage` is not being reloaded via controller on the page refresh.**

With that, I had a look into the controller directly associated to the index.html in hopes of finding `onProviderPage` element which determines whether to display header info. To my surprise, it was not being used in main-controller.js at all. Hence, I ran a search in the project repo for "onProviderPage" again (same as I did on fig. 2.1 in [Finding the relevant html view and its controller](#)).

This has led me to navigation-controller.js file, which as the name suggests, is mainly used by navigation components, not only the header. There I found that it does have a class method to change the

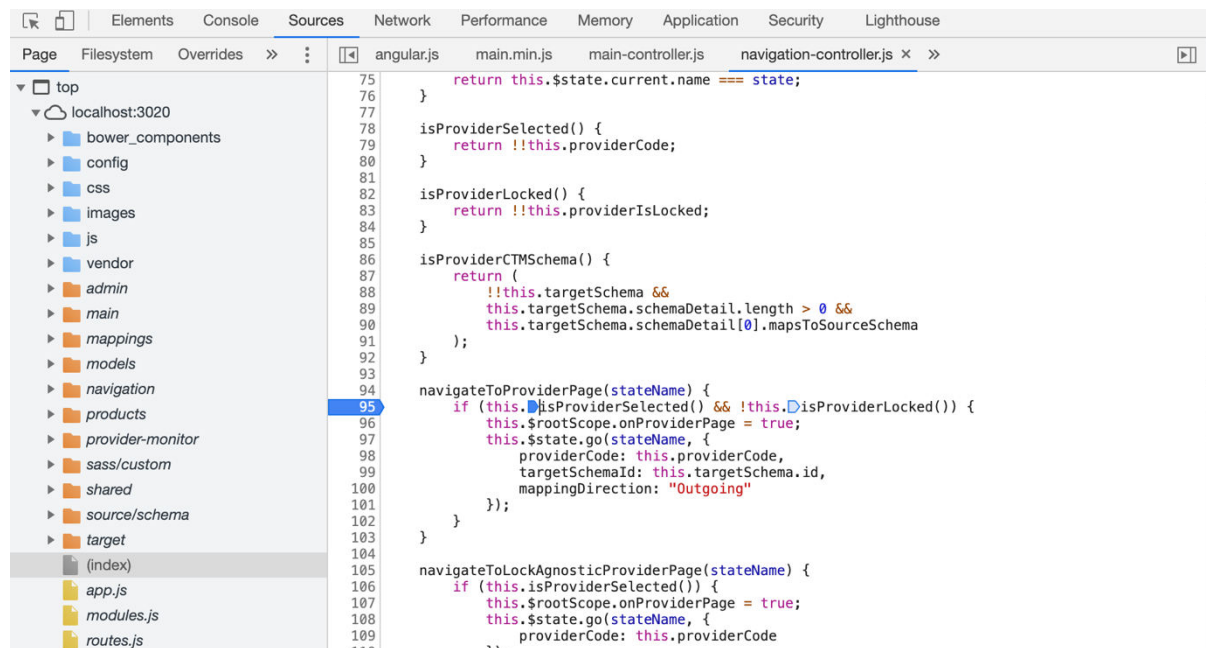
`onProviderPage` element, which is a Boolean that defines whether any of the provider associated pages are being currently viewed (for example, RS and CWL). As seen on the figure below (fig. 2.7):

```
navigateToProviderPage(stateName) {  
  if (this.isProviderSelected() && !this.isProviderLocked()) {  
    this.$rootScope.onProviderPage = true;  
    this.$state.go(stateName, {  
      providerCode: this.providerCode,  
      targetSchemaId: this.targetSchema.id,  
      mappingDirection: "Outgoing"  
    });  
  }  
}
```

**Fig. 2.7, a code snippet of a class method found in the navigation-controller.js that is responsible for making sure to update `onProviderPage` which is being passed around the controllers and html views in `\$rootScope` element. An element that is Angular's JS way to pass around objects for the use of html views. As the name suggests, it is a root object, meaning that every html view and controller can access it.**

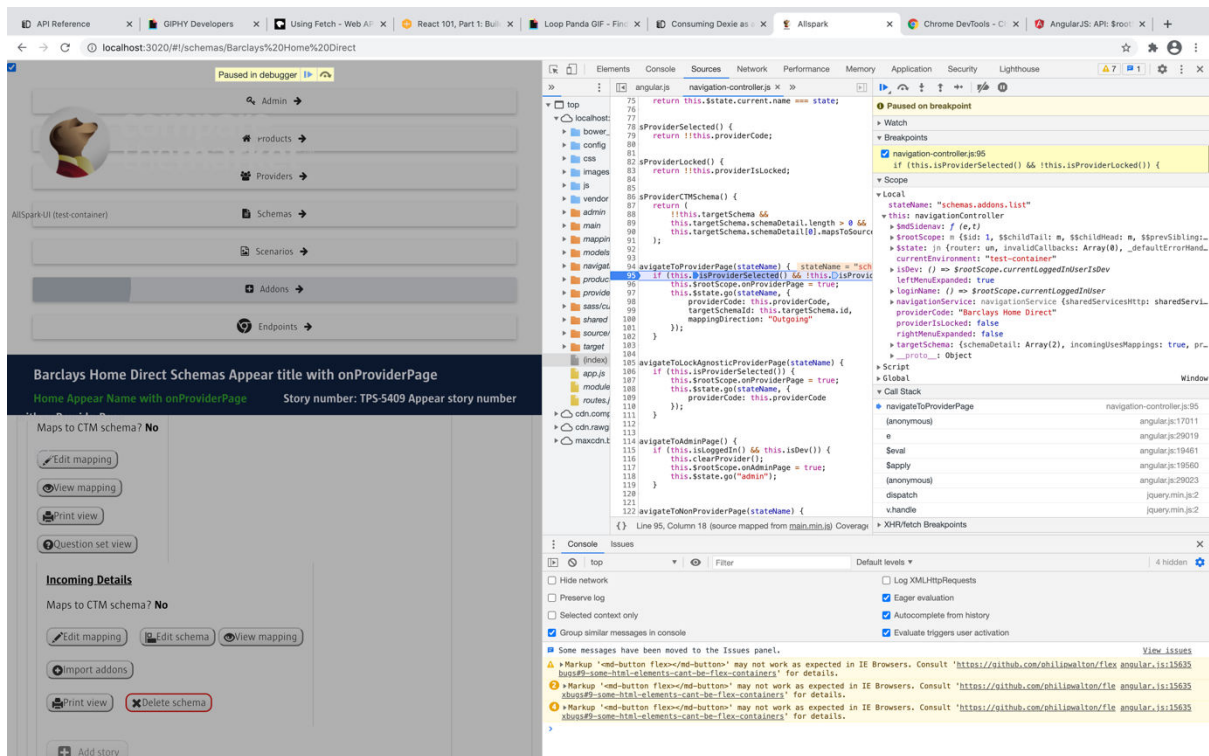
With that I followed and used Chrome developer tools debugger tool. It is a tool that allows to break and stop a code execution in on selected by the user pieces of code. I put a break point on the `navigateToProviderPage` and investigated the behaviour (see fig. 2.8 below to see how I set up the tool and fig. 2.9 on the next page on how it behaves when working). Once the break point was set up, it became apparent that the `navigateToProviderPage` method is not being triggered when: in Schemas, it also is not being used when choosing to do RS or CWL.

This was the very first point at which I knew that I had to look into the controllers for CWL and RS views (their position in the project has been illustrated on the fig. 1.3 in [Project and the framework I worked on](#)), as these were responsible for managing the state of objects passed around on the faulty view, and there were no signs of broken code anywhere so far.



**Fig. 2.8, in order to use debugging tool, I have gone into the developer tools console, looked into the "Sources" tab, and browsed in its left-hand side source code for the project folder dropdown. In the folder navigator, I chose navigation-controller.js file, and on its line 95 (first line on which `navigateToProviderPage` method will start running when invoked). This way, whenever the function of interest was hit, I was able to play and inspect its processes step by step.**



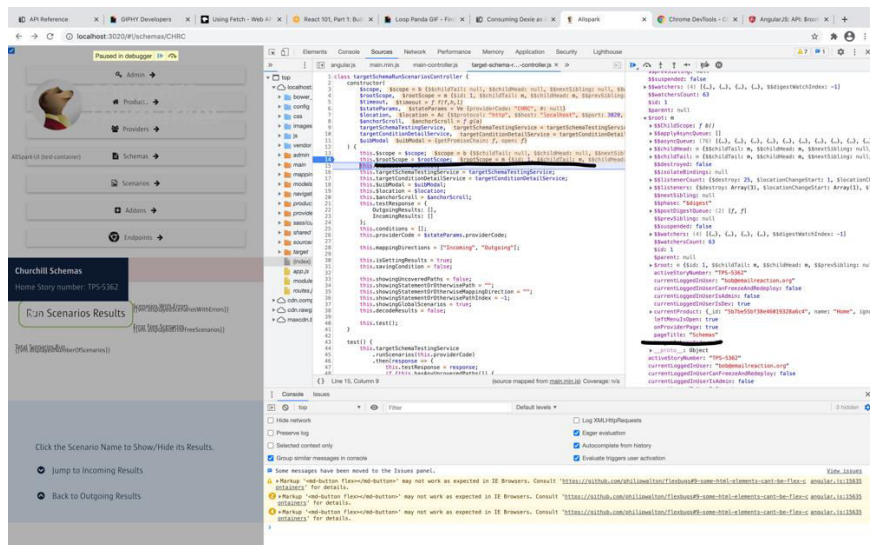


**Fig. 2.9, shows what happens when a break point is hit. It allows the user to jump the process step by step, or completely skip it (yellow pop up on the top of the website). It also displays details of each data piece being handled at the moment of operation step (in the source file next to the blue line of code, or in the window to the right of source file view). Those capabilities prove to be extremely useful in debugging and adding the understanding of code flow.**

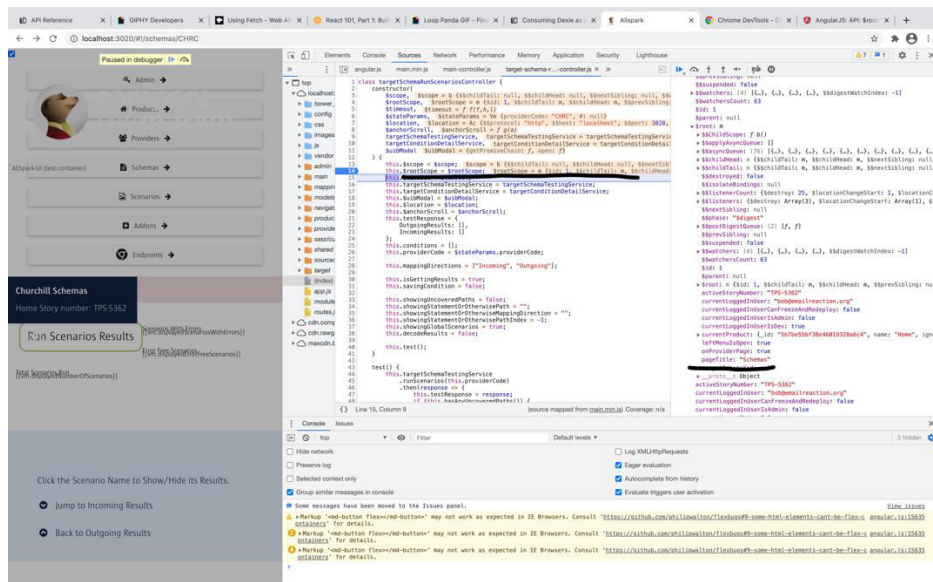
## Exploring Run Scenarios controller and repairing page title

As I have decided after looking at the navigation-controller.js, I looked into controllers for Run Scenarios and Compare With Live. These were pretty much identical and required same changes. Hence, for the purpose of this submission, from now on I will only focus mainly on Run Scenarios and refer to Compare with Live, only where there were slight changes I had to make.

Looking at target-schema-run-scenarios-controller.js, file that contained RS controller, and applying a break point in the debugger in Chrome to the class constructor declaration of `rootScope` I have found out that, upon page refresh, the RS view controller does not update one of the pieces of data displayed in the header: `pageTitle` (see figures 3.1 and 3.2 on the next page respectively to see that `pageTitle` is not being retrieved upon page refresh).



**Fig. 3.1** as can be seen on a bottom right-hand side of the screenshot, once user runs scenarios, it retrieves `pageTitle` and even aforementioned `onProviderPage` all included in a `\$rootScope`.



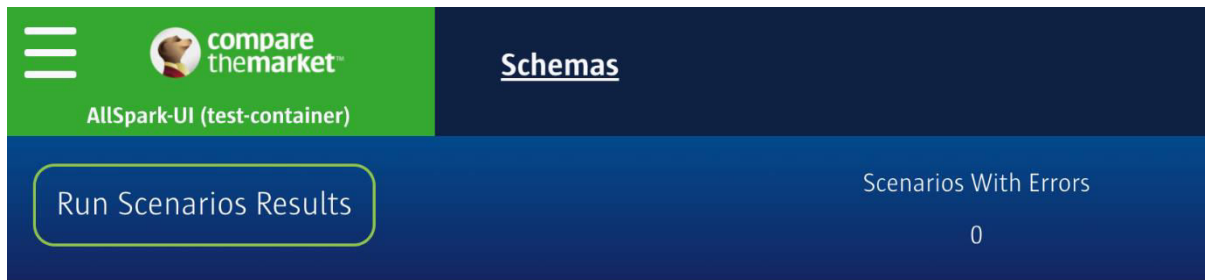
**Fig. 3.2**, shows that after carrying out the same debugging process, after page refresh, a bunch of objects are missing, including `pageTitle`.

The solution to this problem was rather simple, I just had to make sure that upon constructor initialisation, I would explicitly set `pageTitle` to a string value of "Schemas". I was able to fix it in such simplistic way, as RS and CWL views are always only a part of Schemas window. (see fig. 3.3 below for a code change I made and fig. 3.4 on the next page for a snapshot of how it fixed missing page type in a header).

```

15 app/target/schema/testing/target-schema-run-scenarios-controller.js
@@ -10,6 +10,9 @@ class targetSchemaRunScenariosController {
10     targetConditionDetailService,
11     $uibModal
12   ) {
13     $rootScope.pageTitle = "Schemas";
  
```

**Fig. 3.3**, piece of code changed I did to target-schema-run-scenarios-controller.js to keep the page title updated.



**Fig. 3.4, result of the code change. Page title stays correct and visible after refresh.**

#### Loss of provider details and the solution

Fixing the rest of the missing parameters did give me a hard time. After having some quick catch ups with my line manager, who knows the old Allspark UI well, I was told that pieces of information I was looking for are all included in an object that's named `targetSchema`; this would have in itself an entire object of `provider` which simply holds information about provider. With that, I started looking into what parameters are being passed down to RS and CWL.

It all clicked with I realised that there is no `targetSchema` nor `provider` passed into a constructor of the controller class for RS (see fig 3.5 on the next page). I also have used debugging function in Chrome developer tools (same way as on fig 3.2 in the previous sub-section) to meticulously look over all the objects that were being passed around in all the controller's defined parameters and the `$rootScope` itself. There was no sign of them.

I had to make sure to initialise them in the constructor, otherwise, these two would never re-rendered on the page refresh event.

I added `targetSchema` and `provider` parameters in the controller constructor. However, it still wouldn't make any difference without an attempt to read them from server API of the project as without it, these were not passed on to the controller. I studied the controller more closely for clues, there were instances of getting for example scenarios via API service function callers located in `target-schema-testing-service.js` (named in the constructor as `targetSchemaTestingService`) file in the same folder (see fig 3.6 on the next page).



```

1 1 class targetSchemaRunScenariosController {
2 2   constructor(
3 3     $scope,
4 4     $rootScope,
5 5     $timeout,
6 6     $stateParams,
7 7     $location,
8 8     $anchorScroll,
9 9     targetSchemaTestingService,
10 10    targetConditionDetailService,
11 11    $uibModal
12 12  ) {
13 13    + $rootScope.pageTitle = "Schemas";
14 14    + this.provider = null;
15 15    + this.targetSchema = null;
16 16    this.$scope = $scope;
17 17    this.$rootScope = $rootScope;
18 18    this.$timeout = $timeout;
19 19  }
20 20  @@ -38,6 +41,18 @@ class targetSchemaRunScenariosController {
21 21    this.decodeResults = false;
22 22    this.test();
23 23    this.updateProvider().then();
24 24    this.updateTargetSchema().then();
25 25  }
26 26  + async updateProvider() {
27 27  +   this.provider = await this.targetSchemaTestingService.getTargetProvider(this.providerCode);
28 28  +   this.$rootScope.$emit('providerHasChanged', this.provider);
29 29  + }
30 30  + async updateTargetSchema() {
31 31  +   this.targetSchema = await this.targetSchemaTestingService.getTargetSchema(this.providerCode);
32 32  +   this.$rootScope.targetSchema = this.targetSchema;
33 33  + }
34 34  }

```

**Fig. 3.5,** shows a snippet of constructor for class that was in target-schema-run-scenarios-contorller.js and addition of two of asynchronous methods (lines 44-55), that I wrote to update `provider` and `targetSchema` objects. These I also have added to the constructor to make sure they are explicitly passed and retrieved on each page refresh (identical situation and changes were made for CWL).

```

1 1 class targetSchemaTestingServiceFactory {
2 2   constructor(sharedServicesHttp) {
3 3     this.sharedServicesHttp = sharedServicesHttp;
4 4   }
5 5
6 6   compareWithLive(providerCode) {
7 7     return this.sharedServicesHttp.post('targetSchemas/test/compareWithLive/${providerCode}');
8 8   }
9 9
10 10  runScenarios(providerCode) {
11 11    return this.sharedServicesHttp.post('targetSchemas/test/runScenarios/${providerCode}');
12 12  }
13 13
14 14  getOutgoingConditions(providerCode) {
15 15    return this.sharedServicesHttp.get(
16 16      'conditions/byProviderCode/${providerCode}/withMappingDirection/Outgoing/true'
17 17    );
18 18  }
19 19
20 20  getIncomingConditions(providerCode) {
21 21    return this.sharedServicesHttp.get(
22 22      'conditions/byProviderCode/${providerCode}/withMappingDirection/Incoming/true'
23 23    );
24 24  }
25 25  + getTargetSchema(providerCode) {
26 26  +   return this.sharedServicesHttp.get(
27 27  +     'targetSchemas/byProviderCode/${providerCode}'
28 28  +   );
29 29  + }
30 30  + getTargetProvider(providerCode) {
31 31  +   return this.sharedServicesHttp.get(
32 32  +     'providers/byCode/${providerCode}'
33 33  +   );
34 34  + }
35 35  }
36 36  }
37 37  }

```

**Fig. 3.6,** shows the file that includes service functions which call apps `server` API endpoints to receive provider data information and conduct different data operations (for example fetching outgoing conditions on line 14 or running scenarios on line 10). I have added similar methods that call server endpoints to get target schema and provider itself by calling those with `providerCode` object; an object that I found persists after page refresh on RS and CWL views and is a simple string representing a given provider.

With those findings, I have used a project search function of my IDE (same as on fig. 2.1 in [Finding the relevant html view and its controller](#)) to search for functions or methods with names starting on `getProvider` in hope to find an already existing service call to grab provider. I did find it in some unrelated to RS and CWL controller located in provider-monitor-controller.js (see fig. 3.7 below).

```
getProviderFromService(providerCode) {  
    const url = "providers/byCode/" + providerCode;  
    this.sharedServicesHttp.get(url)  
        .then(  
            provider => this.updateProviderServiceWithProvider(provider,  
providerCode),  
            () => this.setErrorMessage(`Error loading provider:  
${providerCode}`)  
        );  
}
```

**Fig. 3.7, depicts an example of a service call method to get provider with provider code. I used this example to know what URL I had to implement to my `getTargetProvider` method in target-schema-testing-service.js file.**

Thanks to that, I got to know the URL address for the endpoint and was ready to implement it in target-schema-testing-service.js on lines 26-20 (see. fig. 3.6 on the previous page). This method is a simple call to return a result of calling an endpoint service-related function of separately defined `sharedServicesHttp` Java Script class of API methods. I understand that `sharedServicesHttp` is kept as a central, reusable class of methods (function associated with objects, which is the case for all of the functionalities written and discussed in this ticket) that controllers and related service files can be drawn on and reused. This is a good practice as reusability helps with maintainability, reduces number of tests needed to be written and adds to code readability.

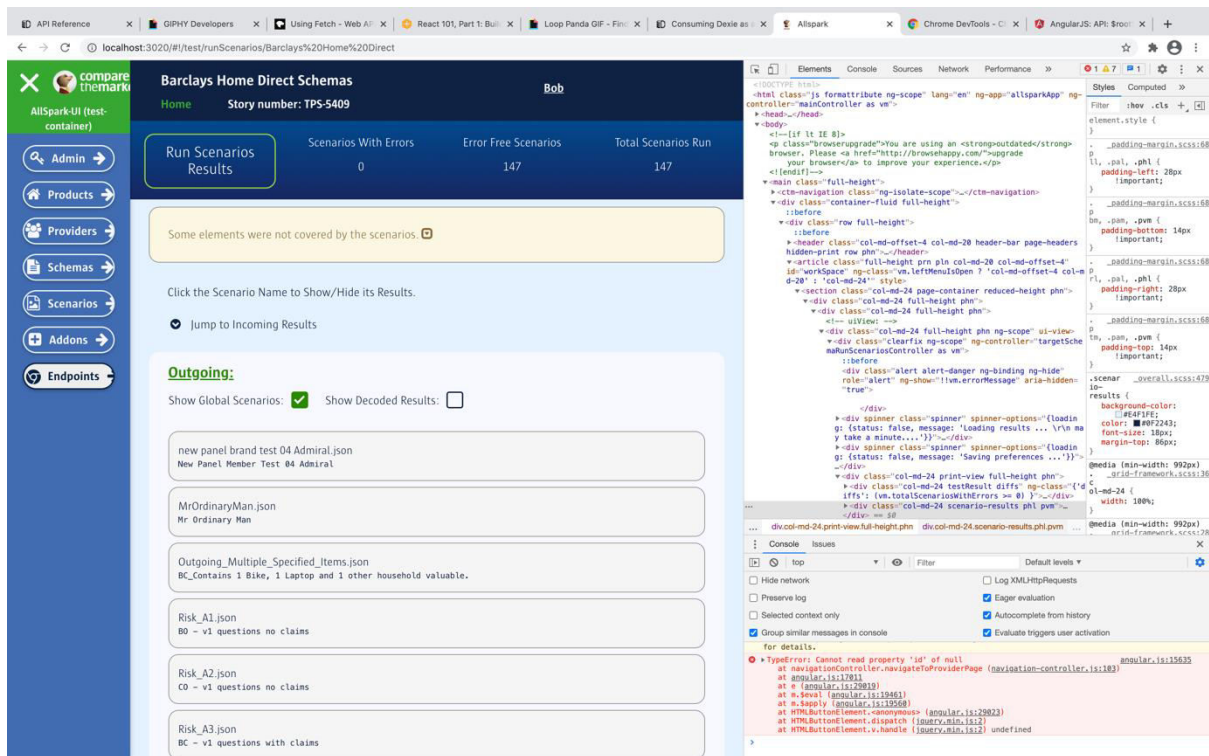
As for the idea that these sets of rules are being written in classes within JS, a prototype-based language: The use of classes simplifies the syntax; despite classes being only a syntactic sugar of prototype declarations introduced with an update of Java Script in ES6 update in 2015, it has a huge benefit of readability. Additionally, as the team members also have to work on back-end projects written in C#, the idea of defining classes is less alien and more approachable rather than prototypes; ultimately streamlining work processes for developers.

The process of finding and updating the target-schema-testin-service.js and the class contained in it with the new method to retrieve and return `targetSchema` object, was identical as the one I just described in this sub-heading so far.

With service calling methods to update target schema and provider, I was ready to incorporate those into two, new methods in the RS controller (see fig. 3.5 on the previous page): `updateProvider()` and `updateTargetSchema()`. Both of those were simple asynchronous methods to call the share API services via target-schema-testin-service.js `getTargetProvider` and `getTargetSchema` methods and update the classes `provider` and `targetSchema` objects, from initially being null, to becoming the actual provider and target schema.

As can be seen on fig. 3.5, these asynchronous methods are being triggered at the end, right after conducting relevant tests (line 43) to run scenarios. I included them at the end since previously, when I was manually testing the behaviour of bugged CWL and RS views (figures 1.6 and 1.7 respectively in [The ticket, it's requirements and how I got involved](#)), this data loss was not impacting the run of actual CWL or RS tasks, so provider and targetSchema fetching didn't have to go before the testing stage.

With those changes, I have managed to consecutively fix the header data loss. After manually testing the behaviour (see fig. 3.8 on the next page to see the UI outcomes), it was apparent that now, only the left-hand side menu was in need of being fixed; the changes to CWL controller were identical as the ones described for RS controller.



**Fig. 3.8, shows the state of the page view on the Run Scenarios option after refreshing. As can be appreciated, the relevant header info persists this time. Changes also affected the left-hand side menu to some extent. Now, the menu does show all of the product options, like Scenarios, Addons and Endpoints. However, after trying to navigate to one of those, developer tools are yielding an error happening in `navigationController` class about one of its methods not being able to read id property of passed in null object.**

As the figure above highlights, the only thing left to fix at that stage, was the ability to navigate off from Schemas to other provider options after page refresh. This was my next step.

Navigating away from RS and CWL views after refresh

The error described in the previous sub-heading and depicted on the figure 3.8 did not surprise me. I was expecting it after exploring the `navigateToProviderPage` (see figures 2.7 to 2.9) method and debugging it in sub-heading [Getting visibility and assessing the presentational component](#). The method, in its unchanged version, was not conditionally checking and updating if there is any `targetSchema` object being passed down via `\$rootScope`, which is needed to be able to navigate to provider's different options, as these are being applied to objects that `targetSchema` nests. With that in mind, I have made sure to conditionally update the target schema with already existing `updateProviderTargetSchema` method in this `navigationController` class (see fig. 3.9 below for the code snippet of `updateProviderTargetSchema`).

```
updateProviderTargetSchema(targetSchema) {
  this.targetSchema = targetSchema;
}
```

**Fig. 3.9, code snippet of simple method `updateProviderTargetSchema` that updates class object `targetSchema` in `navigationController` class.**

With such method, I have updated the navigation-controller.js (file in which `navigationController` class is defined) in a following way (see fig. 4.1 on the page below).

```

 10 app/navigation/navigation-controller.js
@@ -49,8 +49,9 @@ class navigationController {
 49 49         this.toggleLeftMenu(false);
 50 50         $mdSidenav("left").close();
 51 51     });
 52 +
 52 53     }
 53 -
 54 +
 54 55     toggleLeft() {
 55 56         this.$mdSidenav("left").toggle();
 56 57     }
@@ -96,8 +97,15 @@ class navigationController {
 96 97         this.targetSchema.schemaDetail[0].mapsToSourceSchema
 97 98     );
 98 99     }
 100 +
 101 +     isTargetSchemaPassedFromCWLorRS() {
 102 +         return this.$rootScope.targetSchema;
 103 +     }
 99 104
 100 105     navigateToProviderPage(stateName) {
 106 +         if (this.isTargetSchemaPassedFromCWLorRS()) {
 107 +             this.updateProviderTargetSchema(this.$rootScope.targetSchema);
 108 +         }
 101 109         if (this.isProviderSelected() && !this.isProviderLocked()) {
 102 110             this.$rootScope.onProviderPage = true;
 103 111             this.$state.go(stateName, {
 104 112                 providerCode: this.providerCode,
 105 113                 targetSchemaId: this.targetSchema.id,
 106 114                 mappingDirection: "Outgoing"
 107 115             });
 108 116         }
 109 117     }

```

**Fig. 4.1**, shows how I used `updateProviderTargetSchema` method to make sure that should user like to move away from RS or CWL views into some other provider specific pages after page refresh, to retrieve target schema upon refresh (line 106 and 107). As can be seen, on lines 101 to 103 I added a simple helper method to return target schema from `\$rootScope`. This, I later pass into the if statement as a condition on line 106. If there is no targetSchema returned from helper method, it returns null and with null being passed into if statement, it evaluates to false (feature of Java Script language), hence, the update of target schema takes place.

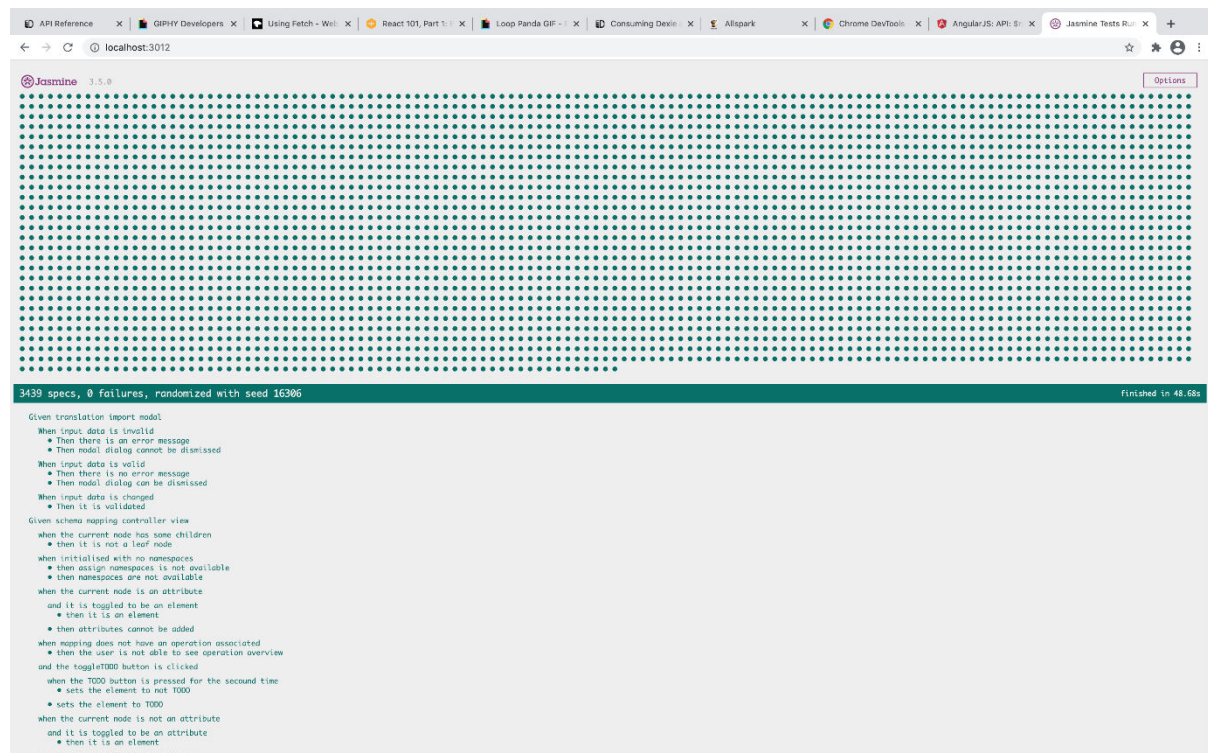
With that change made to navigation-controller.js, the bug has been fixed. User was now able to freely browse through left-hand side menu out form refreshed RS and CWL views.

#### Testing the impact on existing test suites

As I already mentioned throughout the above debugging description, I was using plenty of manual testing of my creation to assess the changes I was making along the way. However, what I didn't mention yet, was the fact that I was also successively checking if any of the changes were be breaking the base of already existing unit tests. The old Allspark UI has a rich base of about three and a half thousand unit tests. These are all



written with a help of Jasmine testing suite (open source Java Script testing framework that focuses on behaviour testing), and when project is ran locally, the tests are dockerised and ran in the separate localhost domain (localhost:3012) (see fig. 4.2 below for a screen capture of how the completed set of Jasmine tests looks like).



**Fig. 4.2, shows a web browser UI of Jasmin tests that were completed. Each dot on the screen corresponds to a single test passed (these are being printed in a real time, as the tests are progressing). Should some test fail, it would print `x` sign instead of green dot. Below the dotted space, there is a breakdown of each test description. Single sub-sets or even single tests can be run by choosing appropriate headings or dots in the UI.**

It was important to systematically check if any unit tests would break because, originally none of them have even detected that the bug existed; hence, any changes made should fall outside of the scope of existing assertions.

With none of the tests broken, it was also important to future-proof the subtle changes and reflect those in the unit tests. It was challenging to find my way around those. The way new tests are being written in Allspark project follows strict, team agreed structure.

First of all, tests should be written with a `describe` syntax and should follow a “When. Then” description styling; to keep the coherent testing theme throughout. Additionally, any arrangements of test conditions should take place in `beforeEach` block of code in every `describe` block, so that assertions (`it` elements) are easily readable and only is concerned about expecting specific outcomes. This is made to simplify reading, understanding and updating existing tests, as to understand a full arrangement, you only have to look into single block of code in `beforeEach`, not having to jump between this and any potential arrangements in `it` statements.

I also had to make sure to use a specific, Angular JS related way of asynchronising test expectations, which was widely used in already existing test suites. I had to ask my line manager for a quick catch-up call to understand the process better. However, once I consulted it with Marc, I was ready to add my test cases.

## Unit testing the changes made

The unit tests I had to implement for the changes were not extensive. All I had to do was to test if CWL and RS controllers would make sure to update provider and target schema after rendering by calling `getTargetSchema` and `getTargetProvider`. That's all. As this would be using sourced out service API methods to do so, the error handling wasn't in need of testing, this already has been tested while creating sharedServicesHttp`. Figures 4.3 and 4.4 below show and describe how I wrote my tests for getTargetSchema` and getTargetProvider`.`

```
... 20 app/target/schema/testing/target-schema-run-scenarios-controller_test.js ... Viewed ...
... 1 1 describe("Given run scenarios view", () => {
2 2   let scontroller;
3 3   let controller;
4 4   let rootScope;
5 5   let $scope;
6 6   let $stateParams;
7 7   + let deferredGetTargetSchema;
8 8   + let deferredGetTargetProvider;
9 9   let deferredPost;
10 10  let deferredGetOutgoingConditions;
11 11  let deferredGetIncomingConditions;
12 12  let targetSchemaTestingService;
13 13
14 14  beforeEach(module("allsparkApp"));
15 15
16 16  beforeEach(inject((_scontroller_, _$q_, _rootScope_) => {
17 17    $rootScope = _rootScope_;
18 18    $scope = _rootScope_.new();
19 19    deferredPost = _$q_.defer();
20 20    deferredGetOutgoingConditions = _$q_.defer();
21 21    deferredGetIncomingConditions = _$q_.defer();
22 22    + deferredGetTargetSchema = _$q_.defer();
23 23    + deferredGetTargetProvider = _$q_.defer();
24 24    targetSchemaTestingService = jasmine.createSpyObj("targetSchemaTestingService", [
25 25      "runScenarios",
26 26      "getOutgoingConditions",
27 27      "getIncomingConditions",
28 28      "getTargetSchema",
29 29      "getTargetProvider"
30 30    ]);
31 31    targetSchemaTestingService.runScenarios.and.returnValue(deferredPost.promise);
32 32    targetSchemaTestingService.getOutgoingConditions.and.returnValue(deferredGetOutgoingConditions.promise);
33 33    targetSchemaTestingService.getIncomingConditions.and.returnValue(deferredGetIncomingConditions.promise);
34 34    + targetSchemaTestingService.getTargetSchema.and.returnValue(deferredGetTargetSchema.promise);
35 35    + targetSchemaTestingService.getTargetProvider.and.returnValue(deferredGetTargetProvider.promise);
36 36    $controller = _scontroller_;
```

**Fig. 4.3, shows how I had to declare two new variables on lines 7 and 8. These were to be later set and read as results of promises returned by running methods in test: `getTargetSchema` and getTargetProvider`. To make the promises return asynchronously I used $_q_.defer() syntax, which is Angular's JS way of running and returning functions asynchronously and as promises. I applied it on lines 22 and 23. Next, to test reaction of methods in test, I have stubbed and set up watch on them using Jasmine's createSpyObj` property. It allows to substitute behaviour and result of method or function being tested (lines 28 and 29). Lastly, to set up what should the methods in test return if called, I made them to return promises of deferredTargetSchema` and deferredTargetProvider` (lines 34 and 35).`**

```
41 49 describe("when the page loads", () => {
42 50   it("then selected schema is tested", () => {
43 51     expect(controller.isGettingResults).toEqual(true);
44 52     expect(targetSchemaTestingService.runScenarios).toHaveBeenCalledWith($stateParams.providerCode);
45 53   });
46 54
47 55   + it("then it has targetSchema", () => {
48 56     expect(controller.isGettingResults).toEqual(true);
49 57     expect(targetSchemaTestingService.getTargetSchema).toHaveBeenCalledWith($stateParams.providerCode);
50 58   });
51 59   + it("then it has targetProvider", () => {
52 60     expect(controller.isGettingResults).toEqual(true);
53 61     expect(targetSchemaTestingService.getTargetProvider).toHaveBeenCalledWith($stateParams.providerCode);
54 62   });
55 63
56 64 }
```

**Fig. 4.4, shows the assertions I wrote to test the behaviour of `getTargetSchema` and getTargetProvider` when page loads (which includes a reload of the page). As can be seen, I haven't had to add a whole new describe` block as it already was set up to test other functionalities of RS controller. The arrangement for all these tests was done on fig. 4.3 on the previous page in the beforeEach` block starting on line 16. The describe` block on this figure, is nested under the beforeEach` statement in a common parent describe` block (line 1 on fig. 4.3); hence, it inherits arrangements (describe` block in this figure). The isGettingResults` object, is just a Boolean that in the controller, is responsible for deciding whether a user will see a loading animation. As in the case of these assertions, page loading still is happening, it should be defaulting to true. Then on lines 57 and 62, I am testing the methods in test, making sure that they are being called with appropriate parameter: providerCode`.`**

As expected, the tests have passed (see fig. 4.5 below) and have not interrupted any other Jasmine unit tests.

```
Ran 32 of 3422 specs - run all
32 specs, 0 failures, randomized with seed 33343

Given run scenarios view
  when the page loads
    when successful
      • then the testResults contains ScenarioDescription for incoming scenarios
      • then the testResults contains ScenarioDescription for outgoing scenarios
      • then all outgoing scenarios are displayed
      • then all incoming scenarios are displayed
      and global outgoing scenarios are filtered out
        • then all incoming scenarios are displayed
        • then only the non-global outgoing scenarios are displayed
      • then the spinner disappears
    when there are no uncovered paths
      • then the spinner is no longer shown
      • then the outgoing conditions are not retrieved
      • then the uncovered paths warning is not shown
      • then the incoming conditions are not retrieved
      • then it has targetProvider
    when there are uncovered paths
      and the call to get outgoing conditions fails
        • then the spinner is no longer shown
        • then the error message is displayed
      and the call to get outgoing conditions is successful
      and the call to get incoming conditions fails
        • then the error message is displayed
        • then the spinner is no longer shown
        • then the conditions are retrieved for incoming
      and the call to get incoming conditions is successful
      and the user expands the warning
        and there are not uncovered outgoing nodes
          • then they are not present in the output
        and there are uncovered incoming nodes
          • then they are present in the output
          • then the incoming nodes are displayed as-is
        and the user collapses the warning
          • then the paths are no longer shown
        and there are not uncovered incoming nodes
          • then they are not present in the output
        • then the paths are shown
        and there are uncovered outgoing nodes
          • then they are present in the output
          • then the outgoing nodes are displayed as-is
        • then the spinner is no longer shown
        • then the uncovered paths warning is shown
      • then the conditions are retrieved for outgoing
      • then the paths are not initially shown
      • then the spinner is still shown
      • then selected schema is tested
      • then it has targetSchema
```

**Fig. 4.5 is an outcome of running a snippet of test code written for RS controller. As can be seen, it ran in a randomised order, which is a default feature of Jasmine testing tool, to help avoid false positive/negative tests where they might just carry over some arrangements from one assertion to another. The tests I wrote so happened to be validated at the end this time; these have also passed.**

I also had an appreciation of developing my test from red to green. That is before I asserted to test the methods to pass, I also tested if the tests could fail, to avoid broken assertions that would pass on every occasion. Figures 4.6 and 4.7 on the next page show how I made sure to develop them in such manner.

```

49 > describe("when the page loads", () => {
50 >   it("then selected schema is tested", () => {
51     expect(controller.isGettingResults).toEqual(true);
52     expect(targetSchemaTestingService.runScenarios).toHaveBeenCalled($stateParams.providerCode);
53   });
54
55 >   it("then it has targetSchema", () => {
56     expect(controller.isGettingResults).toEqual(false);
57     expect(targetSchemaTestingService.getTargetSchema).toHaveBeenCalled("not the right object");
58   });
59
60 >   it("then it has targetProvider", () => {
61     expect(controller.isGettingResults).toEqual(false);
62     expect(targetSchemaTestingService.getTargetProvider).toHaveBeenCalled("not the right object");
63   });

```

**Fig. 4.6**, shows how I made false assertions against `getTargetSchema` and `getTargetProvider`. I checked to make sure that tests would fail because of incorrect objects being called with methods (lines 57 and 62). Additionally, I checked that `isGettingResults` assertions on lines 56 and 61 would fail if expected to equal `false`.

```

Ran 32 of 3422 specs - run all
32 specs, 2 failures, randomized with seed 27282
Spec List | Failures

Given run scenarios view > when the page loads > then it has targetProvider
Expected true to equal false.
Error: Expected true to equal false.
    at <Jasmine>
    at UserContext.<anonymous> (http://localhost:3012/js/tests/tests.min.js:57655:49)
    at <Jasmine>

Expected spy targetSchemaTestingService.getTargetProvider to have been called with:
[ 'not the right object' ]
but actual calls were:
[ 'A04876088302675925' ].

Call 0:
Expected $[0] = 'A04876088302675925' to equal 'not the right object'.
    at <Jasmine>
    at UserContext.<anonymous> (http://localhost:3012/js/tests/tests.min.js:57656:66)
    at <Jasmine>

Given run scenarios view > when the page loads > then it has targetSchema
Expected true to equal false.
Error: Expected true to equal false.
    at <Jasmine>
    at UserContext.<anonymous> (http://localhost:3012/js/tests/tests.min.js:57650:49)
    at <Jasmine>

Expected spy targetSchemaTestingService.getTargetSchema to have been called with:
[ 'not the right object' ]
but actual calls were:
[ 'A09318302439403312' ].

Call 0:
Expected $[0] = 'A09318302439403312' to equal 'not the right object'.
    at <Jasmine>
    at UserContext.<anonymous> (http://localhost:3012/js/tests/tests.min.js:57651:64)
    at <Jasmine>

```

**Fig. 4.7**, shows that when RS controller test suite ran, it failed only exactly on the points that it was meant to. It was a great news, it ultimately proved that tests I wrote were correct, and also that these were not interrupting any other tests (even when failing).



## Integration and feature testing, raising and resolving another ticket

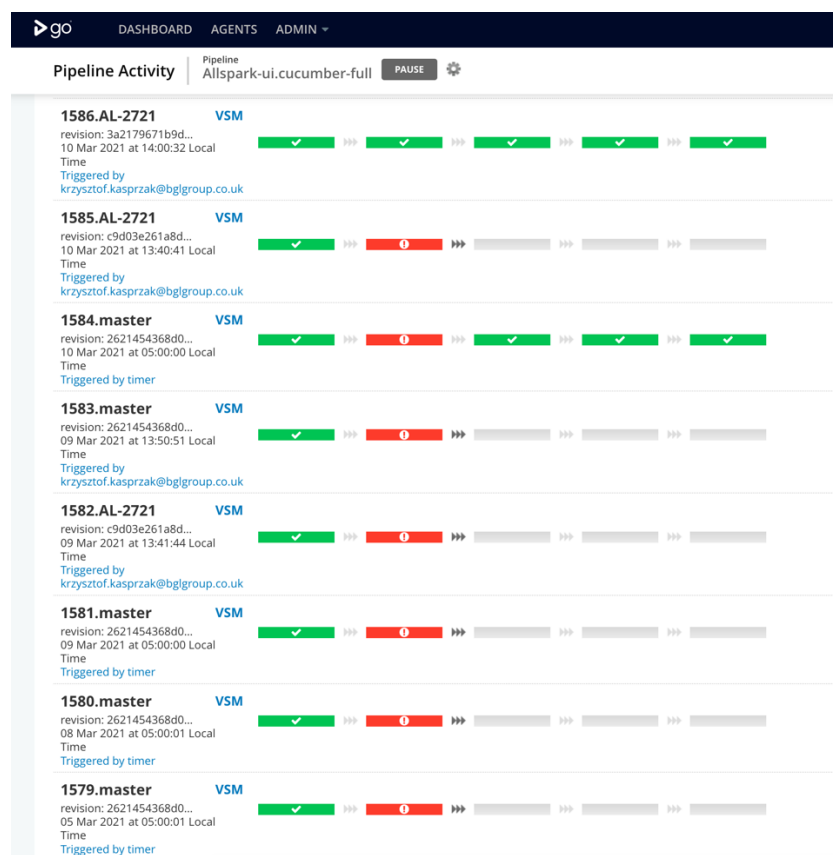
### Integration and feature testing

Upon finishing and conducting all the unit tests, next step before raising a pull request to merge changes to the master branch (branch on GitHub which gets deployed to live environments), was completing the integration and feature tests, to make sure that nothing was broken. These tests are written in cucumber testing framework; a testing framework written in Ruby programming language (object orientated). As it is a rather old framework, and only adds to a complexity of the project, as it requires a knowledge of another programming language; there is an ongoing effort to change from the use of cucumber to cypress framework; much more modern and powerful tool for feature and integration testing.

Cucumber tests are also rather slow to run. With that, Allspark team has a separate GoCD pipeline to run those tests on the platform, as running it on developer's machine would unnecessarily take up time (full test suite takes up to an hour).

Hence, I followed team's general practice. I saved my changes using Git commands (for example, a command ``git add .`` would save an instance of all the file changes I made) . Git is a software package to track and manage file changes, which is a tool I use daily in my workplace. Then also using Git I pushed my changes on the working branch to the project repository on GitHub, a platform on which My team and I keep all the code bases for the projects.

Then, using cucumber testing dedicated Go pipeline, I have set it to read changes from my working branch (AL-2721) and run tests (see fig. 4.8 below to see a track of cucumber tests I found on in the history of runs for this pipeline).



**Fig. 4.8, shows a log of history of runs on Go pipeline named Allspark-ui.cucumber-full. As can be seen there, I have discovered there that it fails only on the second step of the pipeline stages. Additionally, as can be seen, it was failing before I checked in my changes. It also evidences that ultimately, I managed to fix the pipeline (the top run was successful).**

As the screenshot on the previous page hints, the pipeline is also set up to run every 24 hours on 5 am just to check if all the features are in working order. Unfortunately, it was failing for over a week for some reason. This had to be addressed.

#### Raising a ticket to fix test pipeline

I knew I had to raise the problem; I contacted my line manager and showed this to him. He directed me to try and run the same tests on my local machine, just to check if the problem exists within pipeline configuration or is it just some of the actual cucumber tests that are causing the failure.

Nevertheless, we agreed I should raise the ticket. Lack of a testing pipeline meant that nobody from the team could push changes to master branch as there was no means of testing if their changes would potentially impair any of the features.

I have raised the ticket (see fig. 4.9 shows the ticket I raised) and temporarily moved AL-2721 ticket to a blocked column under the agile board. I have also informed everyone the next day during the stand-up that the testing pipeline is broken (and on the same day in developers' slack channel).

The screenshot shows a Jira ticket interface. At the top left, the ticket ID 'AL-3127' is displayed. The title of the ticket is 'Fix cucumber tests'. Below the title, there are icons for attachments, comments, links, and a checkmark. The 'Description' section contains the text: 'The cucumber test has been failing for good two weeks now on one the first stage after building: <https://transformers.cicd.ctmers.io:8154/go/tab/pipeline/history/allspark-ui.cucumber-full> It needs to be fixed to allow for further deployment of new features.' Below the description, there are sections for 'OWASP vulnerabilities' (None), 'Important Details' (Reason for doing, Success criteria, Definition of Done, PRs/Tickets for other teams? Y/N, Can this affect performance? Y/N If Y load test and record), and 'Release Information' (None). On the right side, there is a 'Done' button and a 'Done' status. Below this, there is a table with fields: Assignee (Krzysztof Kasprzak), Reporter (Krzysztof Kasprzak), Labels (None), Buddy (None), Signed-off-By (Sharon), Technical Sign Off (Sharon), Story Points (0), Fix versions (NRC), Debt Size (None), and Tech Debt Type (Not Debt).

**Fig. 4.9, a ticket I created to address the need to fix cucumber testing pipeline (creation and understanding of simple analytics artefacts). I have used a Jira board to create an option to add a new issue and have marked it as a bug. I did not point it, as it is a common practice in my team not to mark bug fixes. I have briefly described the issue and shared a link which redirects to a log of history of runs of the pipeline.**

Ultimately, I decided to pick up the ticket and tried to fix the problem. By collaborating some with other developers, I managed to fix the pipeline.

#### Pipeline fixing

After running the cucumber tests locally to locate which bit of a code causes it, I finally found it. It was just a matter of waiting for those to run on my local machine. It was a scenario test to check the behaviour of lock feature on a provider's product. (see fig. 5.1 on the next page for the snippet for the fault code).

As can be seen on the figure, cucumber tests are meant to be written in a very specific way, to read test assertions in almost plain English. Behind the scenes, these scenario sentences are built of sets of code written in Ruby programming language (as can be seen on fig. 5.2 on the next page).



**Fig. 5.1, shows location, faulty scenario and the changes I had to make to fix the script of running the feature tests.**

Despite the pleasant to read appearance, these scenarios are extremely difficult to write; even more so to modify or adjust.

The initial idea was to try and do so; just debug the test, find the cause of failure and fix it. Together with Sharon (one of team's senior developers), I tried to debug those. We could not find anything that would seemingly break it.

To try and tighten the loop of looking for a bug, I tried to compare the history of commits and the history of the tests ran on the pipeline dedicated to cucumber testing. I did tighten the loop of potential changes made between 24<sup>th</sup> of January and 1<sup>st</sup> of March. Despite that, I couldn't find any change that would look to me as a change which would break the tests suite. Thankfully, after asking for help in the developer's slack channel, Marc, my line manager has managed to find the change; it indeed was a change made within the time period I suspected to be of relevance.



**Fig. 5.2, the breaking change to cucumber tests made to schema\_list.rb file. It is the kind of code that is behind creating plain English looking scenarios presented on the figure 5.1.**

Unfortunately, even with the faulty piece of code found, Sharon and I was unable to fix the code. I have described this problem in slack channel; further help came. Jose, our test engineer has suggested that there already exists a piece of code written in Jasmine unit test that checks functionality of locking feature (see fig. 5.3 on the next page).

```

545     [true, false].forEach(lockedByMe => {
546         describe("and the provider is locked by me", () => {
547             let emailAddress;
548             beforeEach(() => {
549                 emailAddress = generateRandomString();
550                 provider.lockedBy = lockedByMe ? emailAddress : generateRandomString();
551                 deferredGetProvider.resolve(provider);
552
553                 deferredGetLoggedInUser.resolve({
554                     emailAddress
555                 });
556                 $rootScope.$apply();
557             });
558
559             it("then the schema can be edited if locked by me", () => {
560                 expect(controller.canEditSchema("Outgoing")).toEqual(lockedByMe);
561             });
562
563             it("then schema can be viewed", () => {
564                 expect(controller.canViewSchema("Outgoing")).toEqual(true);
565             });
566
567             it("then schema can be printed", () => {
568                 expect(controller.canPrintSchema("Outgoing")).toEqual(true);
569             });
570
571             it("then schema can be deployed if locked by me", () => {
572                 expect(controller.canDeploy()).toEqual(lockedByMe);
573             });
574
575             it("then addons can be imported if locked by me", () => {
576                 expect(controller.canImportAddons()).toEqual(lockedByMe);
577             });
578         });

```

**Fig. 5.3, a piece of code Jose found in `app->target->schema->display->target-schema-display-controller_test.js` that already tests the functionality that faulty cucumber test was trying to verify.**

After he shared the piece of code and its position in the project, he suggested to potentially abandon the cucumber test and just modify the piece he found accordingly. Sharon and I were fond of this idea. Maintaining expensive and slow feature tests that are fragile is the opposite of best practices. If there is a way to substitute those with simpler and more accurate unit tests, it should be done. I began the task of rewriting the unit test. After closer inspection, I found however, that there is no need to modify it at all. The cucumber test that was failing was trying to test a case of where provider's product is locked by someone else than current user. I realised that, the code suggested by Jose, does this as well. The line 545 of the code, asserts both (by iterating through array of two instances, true and false), when it is locked by current user (true) and someone else (false). To make sure I was right, I did consult with Jose.

He indeed missed this detail. With this realisation of mine, the only change I had to still make was to delete the faulty cucumber test. As can be seen on the figure 5.2 on the page above, this was the case (only change in my pull request of fixing the pipeline). Before requesting to merge the changes to the master branch, I made sure to test the changes on a separate branch on the pipeline. It was passing. The problem was resolved (as seen on the figure 4.8 in [Integration and feature testing](#), where the last run triggered by me was passing).

## In conclusion

### Things I learned and improved upon during the work

By completing this ticket, I have managed to learn and practice how to follow the code logic and flow of information within Angular JS framework, by closely logging and debugging passed around parameters. I have enforced my ability to use various functionalities of my IDE (using build-in editor terminal, searching via code snippets and specific file names etc.) and implemented usage of bash command aliases. I have managed to create and log out diagrams of directory structures.

I have learned how to efficiently debug using chrome developer tools and enforced my problem-solving abilities by systematically chasing down the bug and finding its presence in specific files.

I have used a variety of manual and unit tests of my design (in accordance with team's best practices) to test changes I made in order to maintain existing code up to set acceptance criteria. I have made sure to follow design recommendations set by user base in order to create the UI more friendly towards them by fixing the bug in accordance with users' advice (who acted as UI designers in this case, as they knew best what features should be included in the product).

### Things I would do differently next time

In essence, there are two things I would reflect upon and note on.

Firstly, it was the cycle in which I tested UI changes I made. Namely, as I have described in [Unit testing the changes made](#), I did make sure to check if my tests were not false positive, but in my process, I did not write the tests prior to introducing changes. Something that is basically best common practice when developing new code. This way of approaching programming helps of creating new code that better adheres to requirements set by writing tests beforehand.

Normally, I develop new code components by defining tests first. However, in this situation, where I was not starting from a clean slate, I only tested my changes as I was writing them.

Here I had to first explore and learn on the go the way information and changes are being made within the angular JS framework; then check what (via manually testing and getting visibility) changes would fix the bug. The way I could implement tests first, changes second approach, would be to:

- Once the bug was fixed with initial code changes; revert to untouched codebase.
- With that, knowing what changes were done at a first attempt, try to find a place in existing test suite where new changes could be asserted and verified.
- Write tests first.
- Make the code changes to fix the bug again and try to pass new test assertions.

This way, I would not only further improve my testing practice but also the code changes as well. Writing those for a second time and up to strict requirements of tests, would improve the changes' reliability, and help me detect if I made any unnecessary/poor code corrections. Something I will definitely apply next time around when bug fixing.

Secondly, I unknowingly went against best practice of Angular JS framework of avoiding to use ``async await`` syntax while creating ``updateProvider()`` and ``updateTargetSchema()`` methods in RS and CWL controller (see figure 3.5. in [Loss of provider details and the solution](#) sub-heading). It did not affect the code base in any negative way in this case but could in some other (here no watcher or update function was not further dependant on these methods). It is generally undesired as in some cases, when using ``async await`` syntax as awaited code will run outside of Angular's JS digest loop (digest loop for running Angular written code), and therefore it won't trigger watchers or view updates.

In my mind, it was only natural to try and make these methods asynchronous with ``async await`` as calling for objects could take some time; additionally, it is a modern standard to use ``async await`` syntax.

A simple change to keep the old Angular JS framework happy and not prone to malfunctioning would be to use asynchronous `.then()` (as presented on the fig. 5.4 below) syntax.

```
48 +
49 +   updateProvider() {
50 +       this.targetSchemaTestingService.getTargetProvider(this.providerCode)
51 +       .then(provider => {
52 +           this.provider = provider;
53 +           this.$rootScope.$emit('providerHasChanged', this.provider);
54 +           if (this.provider === null ) {
55 +               throw new Error(`provider was not updated`);
56 +           }
57 +       })
58 +       .catch(err => {
59 +           this.errorMessage = err;
60 +       });
```

***Fig. 5.4, shows a change I actually came to make to `updateProvider()` method while working on a different ticket after completion of this one. It is a swap from use of `async await` syntax to `.then()` syntax; one that will not cause any undesired behaviour while using Angular JS. I needed to change it as in this future ticket, I had to reuse the method somewhere else; with it, since Angular's JS digital loop was not picking it up, controller was not acting as expected.***

Value I brought and the impact I made

By completing the task, I have not only supported the team independently (by not being assigned to pair up on the ticket once Tim left for his holiday, the team was able to maintain two independent streams of work at all times), but also made sure to be vigilant for any team impairing issues and informing others of those (raising an issue about broken testing pipeline). Through this I have secured my ability to deliver simple analysis and action artefacts (like creating the bug fix issue or my daily reporting of progress/blockers during stand ups). Finally, the testing I did complied with team's best practices (tests' layout).