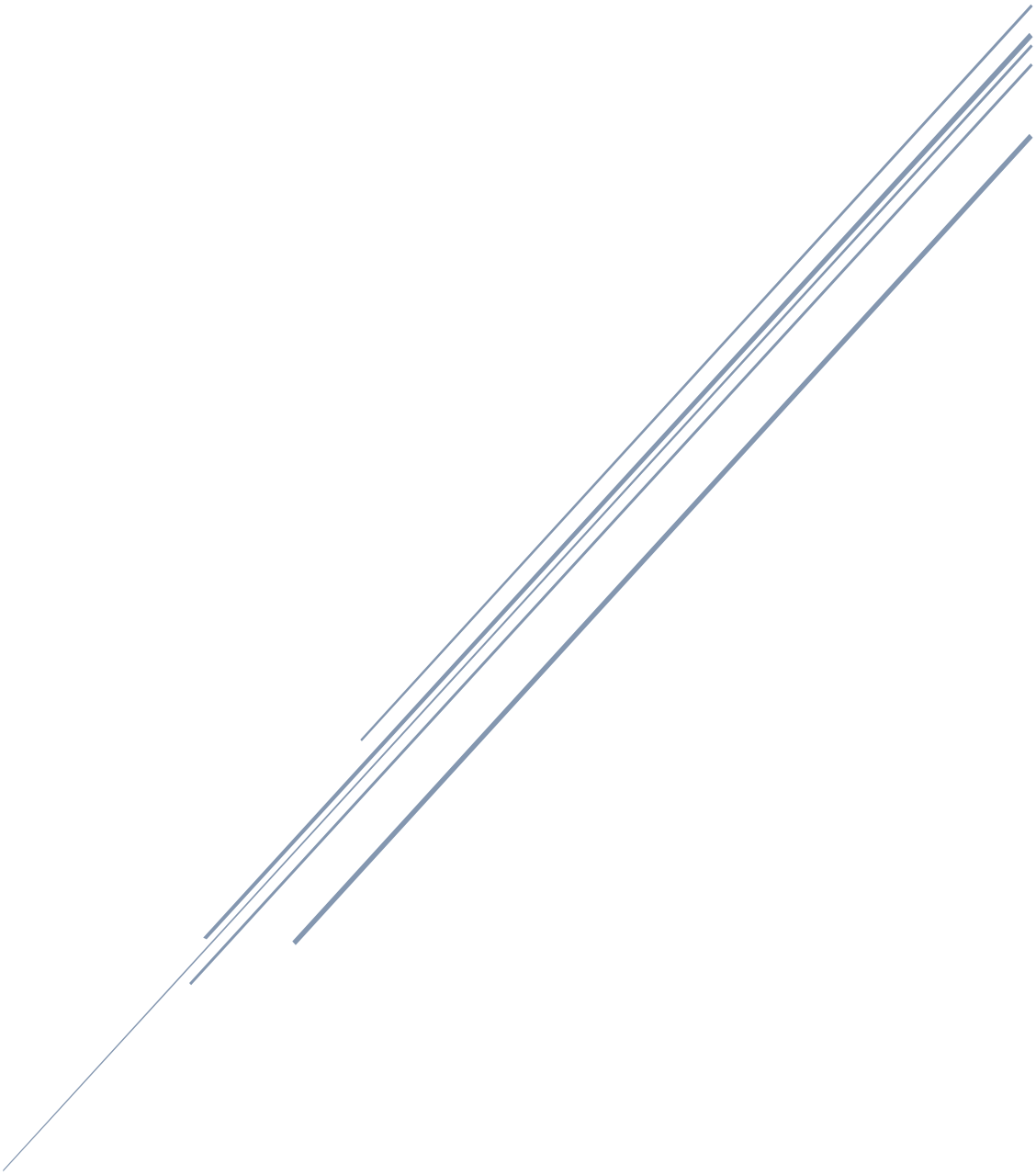
MAKERS ACADEMY, PORTFOLIO SUBMISSION

No. 4



Placement: Compare the Market

Apprentice: Krzysztof Kasprzak

Table of Contents

Design investigation task

The task

Project and the frameworks I worked with

Continuing my involvement in the project of rewriting the Allspark UI (user interface) from Angular JS to React JS, I yet again happened to participate in a task of investigating what can be achieved with the new frontend library (React) to be applied. This task required me to work with: Angular JS framework, React JS library and spiking (experimenting) with changes written in Node JS for the controller-side of service. The last one is a runtime environment that allows for the use of JavaScript in the backend, acting as a headless browser to do so (by default, java script is a language that is understood by modern browsers; hance, the need to emulate the environment).

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

The task serial number was AL-3292 and it was labelled as a task under the Epic of `React-BCE`. That is, it was a task that contributed to the development of the new UI (see fig. 1.1 below).

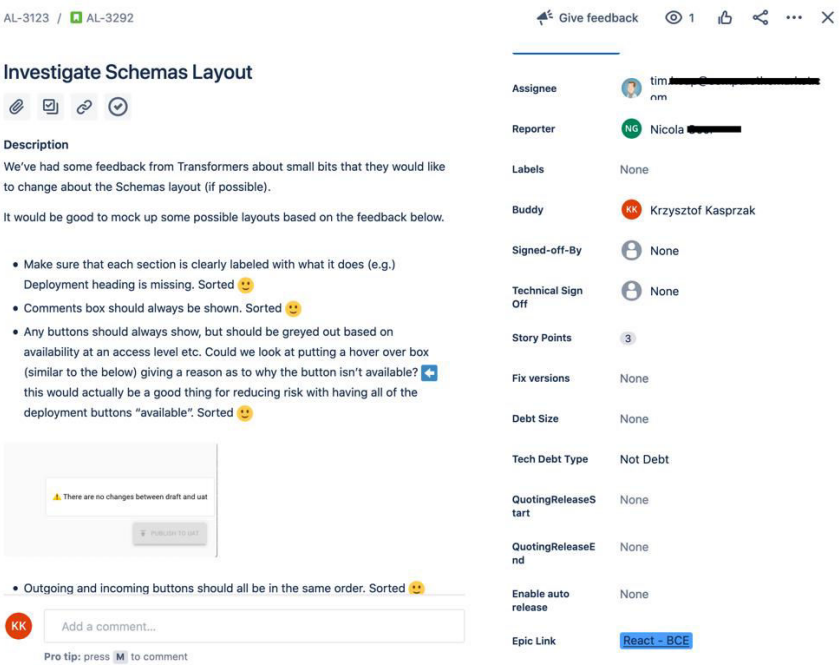


Fig. 1.1, depicts the ticket outline as presented in Jira (tool in use for sprint planning and documentation). Bottom right corner includes the link to the Epic of new UI creation. Higher up on the right-hand side it can be seen that the ticket was evaluated to be of 3 point value.

The pointing system in the team I belong to is quite simple. Whenever the team (including me) attempts to evaluate the size of the story (which takes place on the weekly refinement session), each point represents an expectation of taking up roughly 1 working day to complete.

My involvement in the task stemmed from the fact that at the time being, Tim (my pair partner) and I were the only developers available in the team to work on the new Allspark UI, Sharon, our senior developer was busy working on migrating data in our databases at the time. We took on and completed the ticket.

As the task was to investigate technical aspects, it didn't require from me uploading code updates to live environment via pipelines. The aspects to investigate and/or spike were following (completion of those was equivalent of completing the ticket):

- *Make sure that each section is clearly labelled with what it does (e.g.) Deployment heading is missing.*
- *Comments' box should always be shown.*
- *Any buttons should always show, but should be greyed out based on availability at an access level etc. Could we look at putting a hover over box (similar to the below) giving a reason as to why the button isn't available? □ this would actually be a good thing for reducing risk with having all of the deployment buttons "available".*
- *Outgoing and incoming buttons should all be in the same order.*
- *All buttons should display either horizontally or vertically.*
- *Can Jira stories be Hyperlinked*
- *Can we display more than two stories?*
- *Can we have areas which are drag-able to make bigger (particularly stories section) or if not, some kind of scroll bar or pop-up box to enable more information to be seen on the screen*
- *Tabs for incoming/outgoing details.*

Fig. 1.2, transcript of the areas that needed to be investigated in order to complete the task.

On top of the areas to investigate outlined above, it was also agreed that Tim and I shall create some wireframes of how the layout of the schema's page would look like. Wireframe is a graphical representation of how intended UI will present itself, and what possible functionality might it include.
Creation of such wireframe mock-up was to visualise some of the users' requirements of the design (who in this case acted as UX designers; they were the source of ideas how the sketches were meant to look like and what functionality should the surface design obey in accordance with the logic behind it).
Next, the wireframe was to be used not only as a documentation piece, but also to further crystalise the UI's final form by using it as a tool to collect feedback from the same user base, upon which the ultimate look of the page was to be decided.

Schema's page, what is it for

This page view is a hub that either holds or leads to all of the functionality of Allspark interface regarding individual provider (see fig. 1.4 on page 5 to see the view of the schema's page from the old UI). To navigate to the page, user has to first navigate to the page that holds list of existing providers sorted by product categories (products include home, car and pet insurances to name a few) and select specific provider of a specific product (see fig. 1.3 on the next page for UML activity diagram that depicts the set of actions required from the user to land on the schema's page). Arguably, it is a part of the UI that is most used; hence, great care needed to be taken to reproduce and streamline it for the benefit of the users' ease of access, increased work efficiency and usage comfort.

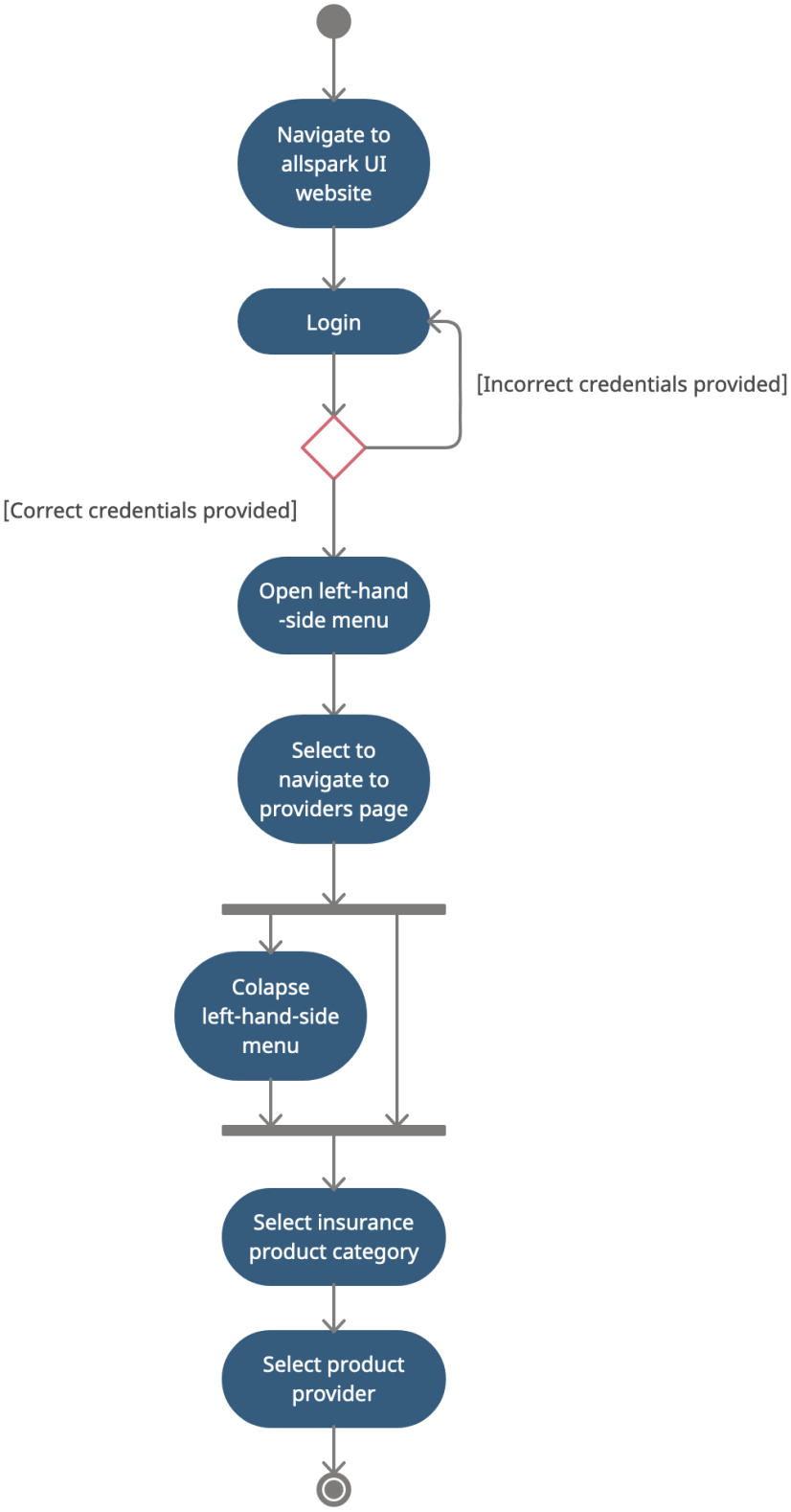


Fig. 1.3, activity diagram I have created that depicts a set of actions user needs to carry out in order to land on a schema's page. It highlights the need of a user to be able to provide correct user details. It also underlines that the left-hand-side menu does not have to be collapsed in order to access the page functionality; collapsing it is fully optional.

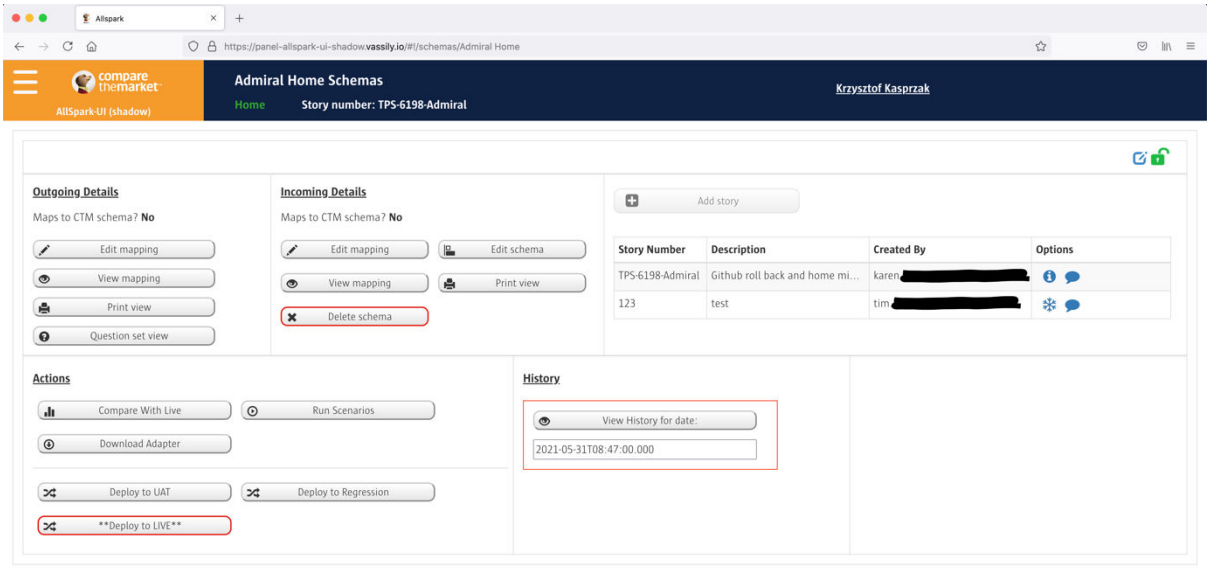


Fig. 1.4, depicts the view of schema's page as it is in the old UI. As can be seen, it allows users to access tools to edit data schema's (hence then page name), compare them with ones that are live and many more. It also provides information about the provider, like the history of which Jira stories are associated to any changes of the provider details, who created those etc.

Creation of the ticket

My contribution in creating the ticket

As presented on the fig. 1.1, it was reported by Nicola, our BA. However, I did co-operate with her to elicit the details. I am involved in updating the Transformers team (our user base) about the progress of creating the new UI on a monthly basis. During those update sessions, I don't only showcase the progress, but also actively collected feedback and suggestions from users. These suggestions and feedback then helped in creating tickets and their descriptions to better specify requirements.

For example, relating to the ticket described here, I have collected a verbal feedback from users during the update session preceding the creation of the ticket about hyperlinking Jira story numbers to actual Jira tickets and potential change on displaying more than just two stories in the top-right story table (see fig. 1.4 above).

Steps to complete the task

Creating the page wireframes, acting as a UI designer

As outlined in `The ticket, it's requirements and how I got involved`, to complete the ticket it was required to create some wireframes of how the schema's page will look like. I have worked on it together with Tim in a pair, mostly driving throughout the process. I have created the wireframes using a free trial of a popular wireframing tool: Balsamiq Wireframes. It is a graphical user interface dedicated for building website wireframe mock-ups. I chose to use the tool since I had a brief exposure to it previously during my Makers Academy bootcamp (I used it to mock-up website interfaces for one of my bootcamp projects). The tool is easy and intuitive to use; additionally, created mock-ups can be saved as pdf documents and used as production artefacts (see fig. 1.5 on the next page to see the Balsamiq interface).

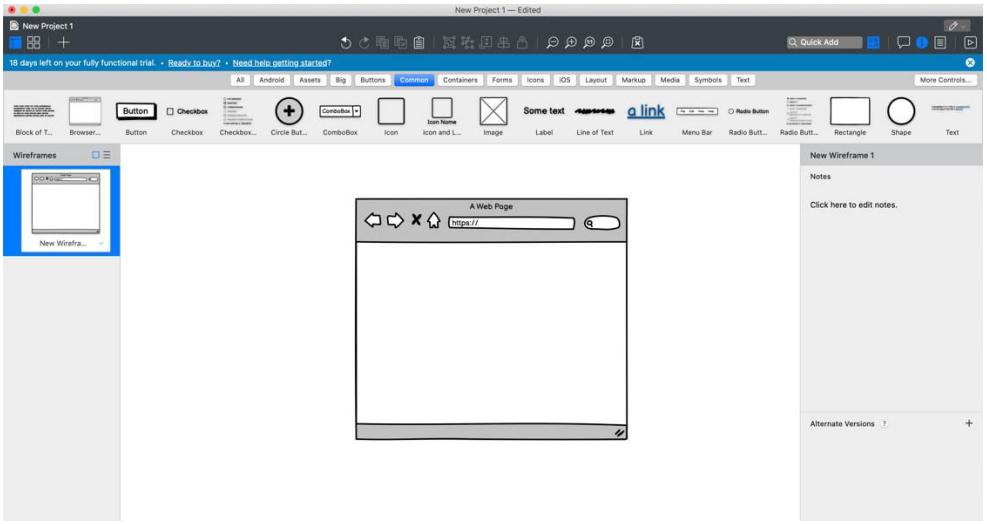


Fig. 1.5 depicts a GUI of Balsamiq Wireframes app I used to create mock-ups. As can be seen, this tool has a selection of editable elements that can be used to represent design of intended interface (top bar of the app).

Since our team doesn't have access to UI designers, the commitment of creating it all was entirely up to me and Tim. The only other design input came from the initial feedback I got from the user base that helped to elicit details of the ticket. Creating the wireframes, I have stuck to all the tickets described recommendations (as outlined on fig. 1.2 in `The ticket, it's requirements and how I got involved`). The initial, two potential visions for the schema's details I created can be found below:

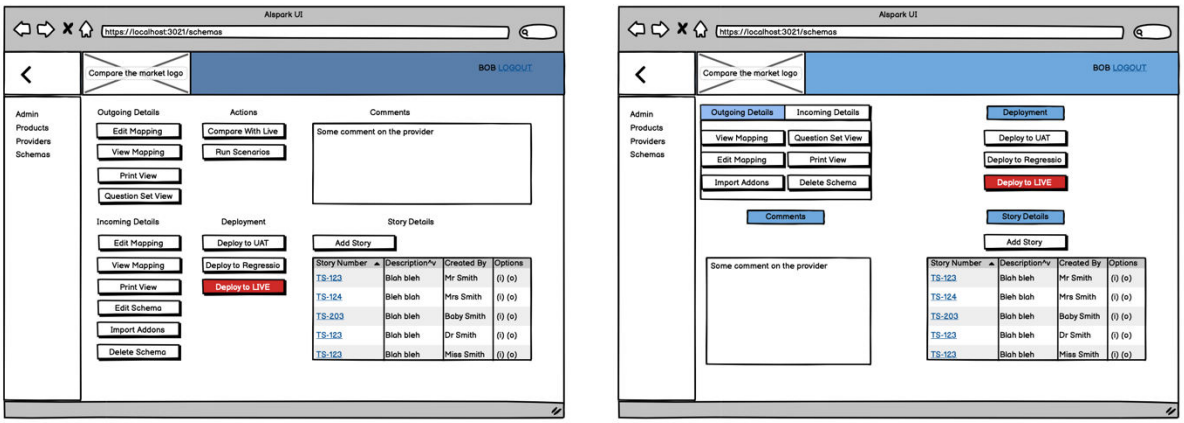


Fig. 1.6: initial mock-ups of the schema's page for the new UI I have created while paring with Tim who was navigating during the process.

As can be appreciated, during this task I took the usability requirements and made an artifact out of them (the wireframes) which envisions and details of what the technical implementation should potentially lead to. Having adhered to the user comments, the potential layout mock-ups have functionalities aligned vertically, comments area is permanent (as opposite to the old GUI, where it becomes visible only once any comments on a provider exist) and story details table is bigger, including hyperlinks to specific stories. The right-hand side representation of Outgoing/Incoming details functionality shows a potential tabs section to choose between actions for each. It was an idea given by Tim (while navigating) that I have incorporated into the wireframes.

Consulting it with Nicola, and adjusting the wireframes in accordance with her advice

Having created the first draft of wireframes, I took on the responsibility of consulting those with Nicola (team's BA). As a user of the UI herself and a person with whom I co-operate the most in order to set up my monthly updates and get any further feedback on behalf of the entire Transformers team, she is the go-to person to obtain any user sign off stamps (approval of the developed changes/outcomes of investigations).

As it turned out, between the time of the ticket creation and commencing the work on it, she already has been given one more idea on how the new UI could be tweaked. In the context of the project, her advice and my collaboration with her is the closest experience I have to working with any UI designers. Her ideas make the right impact to the usability, and she is the one to approve them. Her latest requirement was concerned about the naming concept of the page and how to access it.

Namely, the page shouldn't really be called schema's page, but rather provider details page. Reason being that accessing, updating, and deleting of provider schema's is only one of many functionalities this page offers. Furthermore, it doesn't make sense to be keeping the access to it in a left-hand side menu (as it is the case in the old UI, and how I reproduced it in the initial wireframes on fig. 1.6 on the page above). Each provider is accessed by being chosen on the providers page, which then redirects the user to its details' page (old schema's page); it is the reason why keeping a schema's option in the left-hand side menu is not coherent.

Additionally, she suggested that instead of keeping it in the LHS menu (which originally was made so that a user can navigate back from a provider's functionalities such as compare with live to the provider details' page), a name of a currently chosen provider can be displayed in a header and act as a hyperlink to navigate back to its details' page.

As shown on the fig 1.7 below, I made sure to update the wireframes accordingly to what Nicola has recommended; more importantly, prior to it I have also consulted it with my team manager Marc, to discuss potential changes, to make sure he approves of it as well. He was content with it.

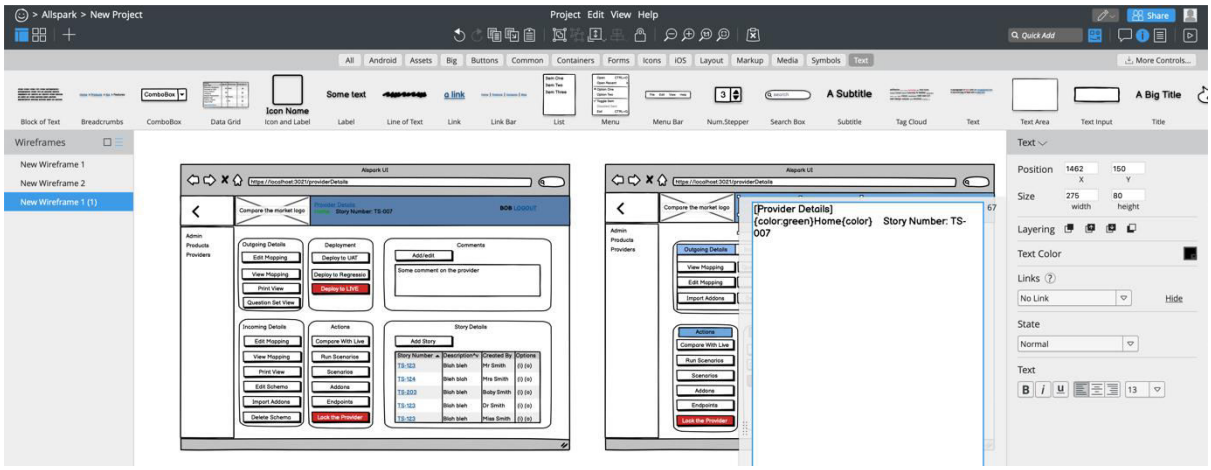


Fig. 1.7, a snippet of a Balsamiq Wireframe tool which includes changes I made based on Nicola's recommendations; LHS menu doesn't include provider details option, the URL is renamed form schemas to providerDetailsPage and header includes provider information + the link to get back to the page.

As per fig. 1.7, it can be seen how I was able to customise wireframe elements to better reflect the intended UI. Wrapping a text in `[]` makes it appear to be a link and applying `color` option right after the text and specifying it right before the text in `{}` makes it appear in chosen font-colour.

Figure above also shows how in the improved version of wireframes I wrapped UI elements in `boxes` with rounded edges. This was to better envision the final look which Tim and I were striving for; providers' page was already created to include each provider in such `box`.

Presenting the wireframes to the user base and collecting further feedback

After successfully completing the wireframes, the time came to present those to the user base and gather some final feedback on the ultimate look and functionality of the page view to be created after the investigation. As this have coincided with the monthly update on the progress, I took this opportunity to use the meeting to present the Transformers team with what is the vision for the page.

Overall, the feedback was positive, and the users were happy with what Tim, and I created. During the update meeting I have faced the team with the need to choose which version should be created, one with an interactive tab for outgoing and incoming details, or more static version which has separate functionality panels for each (as on fig. 1.7 on the previous page).
It was also needed to make the users decide if they prefer to keep the provider details' page tab in the LHS menu or substitute it with the clickable link in the header on the provider's name.

Following the meeting, Nicola has helped me to prompt the users to decide on the changes to be incorporated. Thanks to that I did receive more concrete feedback (see fig 1.8 below).

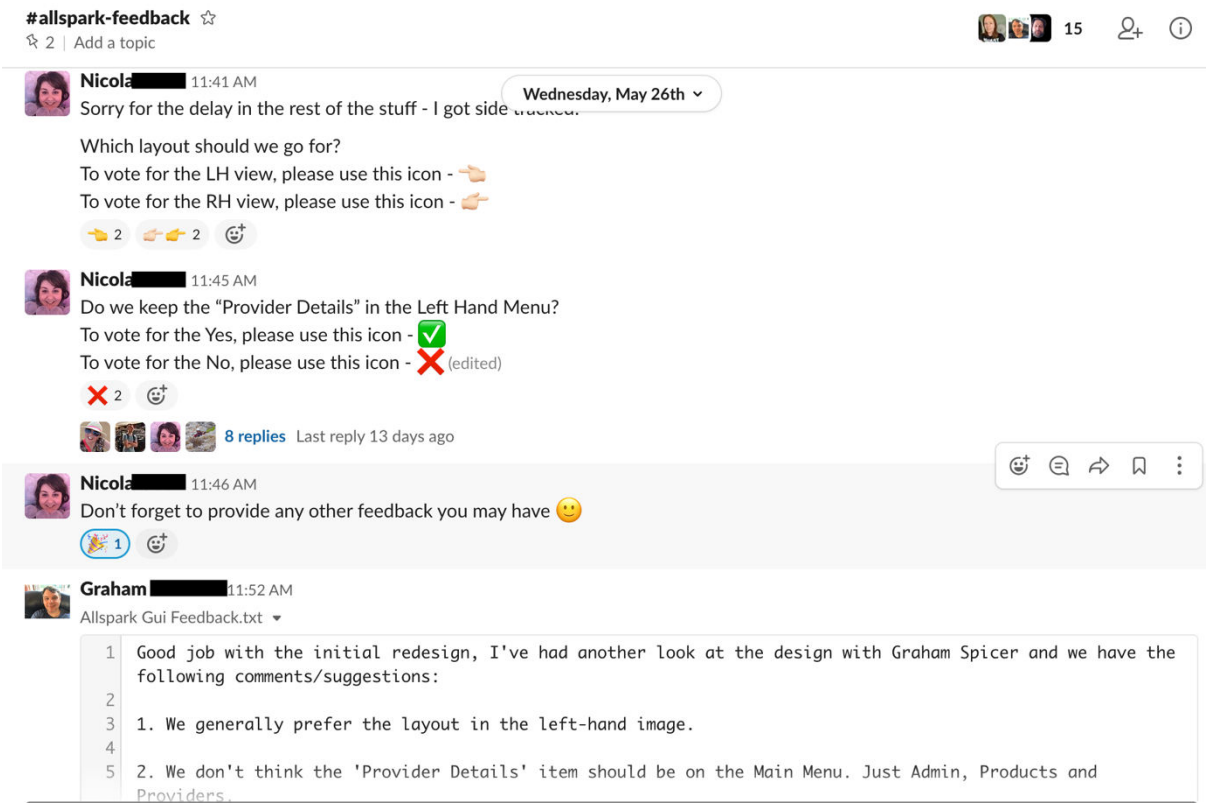


Fig. 1.8, a snippet of a slack channel in which the feedback on the UI designing I did was collected. As can be seen, users were in favour of removing the provider details' page tab from LHS menu but were indecisive of which direction to take on with the incoming/outgoing details panels.

Following the meeting, Graham, Allspark user who is very passionate about the interface, provided me with a very detailed feedback which overall praised the outcomes I achieved, and pointed towards more specific direction (see the transcript of his feedback notes on the next page under fig. 1.9). This was a great response and concluded the wireframing exercise. Together with his feedback and our mock-ups, it was enough of artefacts to determine the level of design and functionality requirements for the upcoming creation of the page.

'Good job with the initial redesign, I've had another look at the design with Graham Spicer and we have the following comments/suggestions:
1. We generally prefer the layout in the left-hand image.
2. We don't think the 'Provider Details' item should be on the Main Menu. Just Admin, Products and Providers.
3. On the Providers screen the Products and Providers should be sorted alphabetically. In fact, this should be the standard throughout Allspark where it can be applied.
4. When you select a provider and the Provider Details screen loads - the Main Menu should be collapsed by default.
5. The four left-hand regions should be re-ordered and contain the following buttons:

Outgoing Details (Top Left)
- *Edit Mapping*
- *View Mapping*
- *Print View*
- *Question Set View*

Incoming Details (Top Middle)
- *Edit Mapping*
- *View Mapping*
- *Print View*
- *Edit/Delete Schema*

Comments (Top Right)
- *No change from image*

Actions (Bottom Left)
- *Run Scenarios/Coverage*
- *Compare With Live*
- *Edit Scenarios*
- *Endpoint Config*
- *Download Adapter*
- *Lock Provider*

Deployment (Bottom Middle)
- *Deploy to UAT*
- *Deploy to Regression*
- *Deploy to Live*

Story Details (Bottom Right)
- *No change from image*

6. The Edit Schema and Delete Schema buttons on the Incoming Details region should be combined to give the same number of buttons as the Outgoing Details which should make layout easier.
7. The AddOns related buttons should be removed.
8. We need the Download Adapter button and the Actions region seems the correct place to locate it.
9. The 'Provider Details' item at the top of the screen should include the provider name and product (i.e. 'Quote Detective Van Details') and should be made into a link that returns you to the Provider Details screen.'

Fig. 1.9, a transcript of Graham's feedback and further suggestions on the direction Tim and I should take on in developing the page.

Investigating the length of story table, overview

Another part of the ticket I made a significant contribution to, was checking if it is possible to return more than two stories associate to a given provider under the story table section (top right functionality shown on fig 1.4).

I did so on my own, as soon as Tim and I were finished with creating the wireframes we knew that we could investigate the rest of criteria separately, distributing work evenly between us.

This sub-part of the task has required me to track the flow control of retrieved data, understanding the algorithm method that was limiting the data returned, changing the algorithm, and adjusting how the sifted through data can be displayed.

Tracking and understanding where the data comes from

To find which parts of the code in the old Allspark UI project (written in Angular JS framework and Node JS) are responsible for fetching the data from database about current and past stories for a given provider, I have used my IDE's (JetBrains Rider) functionality of looking for a text snippet globally within the scope of the project (see fig 2.1 below for the snippet of the functionality used). I found that, on the view level, the table with details of stories is being displayed by `target-schema-display.html` file.

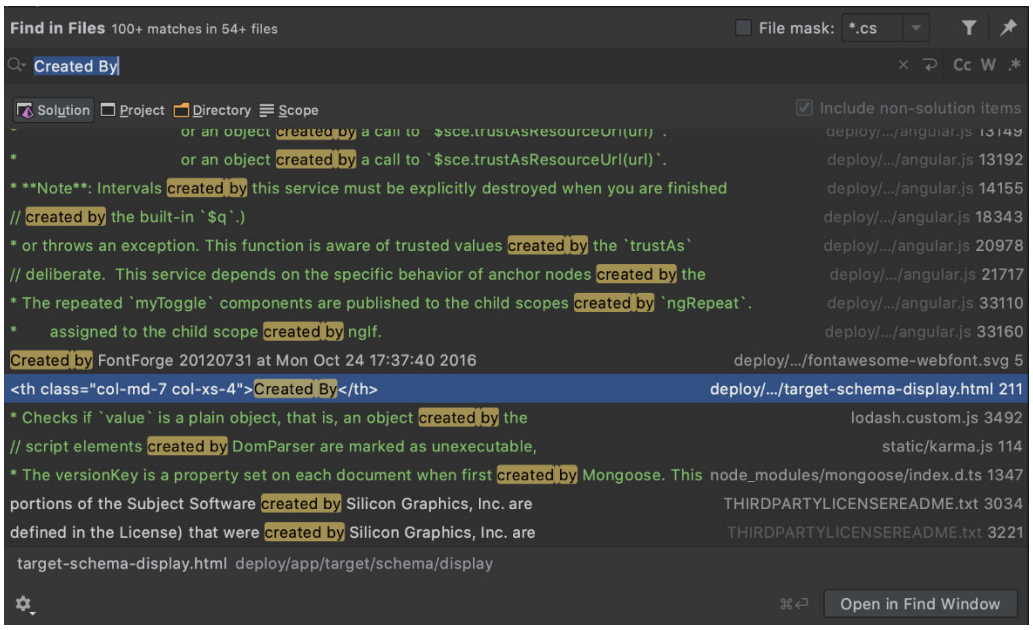


Fig. 2.1, screen of IDE's I use tool of finding code snippets within the scope of the project (triggered by pressing command + shift + f buttons). I decided to look for `Created By` phrase, as it is one of the columns in the table of interest. It indeed found the relevant html file.

From there I was able to find a controller file for the html view found. As the code in the file told me ( ), the controller must have been one in `target-schema-display-contorller.js`. It was indeed. There I found two methods that were responsible for getting the data from the `server` side of the project and sifting through the data to receive maximum of two stories (see fig. 2.2 on the next page for the code snippet that includes methods of interest).

```
getStoryList() {
    this.targetSchemaDisplayService
        .getStoryList(this.providerCode)
        .then(data => {
            this.showOnlyOneMergedStory(data);
        })
        .catch(message => (this.errorMessage = message));
}
showOnlyOneMergedStory(data) {
    this.storiesByProviderCode = [];
    if (data.length > 0) {
        this.storiesByProviderCode = [data[0]];
        if (!data[0].mergedSha && data.length > 1) {
            this.storiesByProviderCode.push(data[1]);
        }
    }
    this.$rootScope.$broadcast("providerStoryNumberChanged");
}
```

Fig. 2.2, code snippets that include `getStoryList` and `showOnlyMergedStory` methods used in the controller to display story numbers and their details.

From the `getStoryList` method I deduced that under `traget-schema-story-service.js` there exists another method called `getStoryList`, which is used by `getStoryList` method in the view controller (second and third line of code in the figure above). This other method (under ` traget-schema-story-service.js`) uses providerCode to make an API Call in order to receive the data. I have followed the used URL of the API call and found it being associated to `story-router.js` file of the server-side code of the project:

```
app.get('/api/stories/byProviderCode/:providerCode',
storyController.getStoriesByProvider);
```

Fig. 2.3, code snippet from `story-router.js` (which was stored under ‘ ./server/story/` directory of the project), which outlines to which function does the API request call.

Having done the similar process of chasing down to next method or function I found two more methods. `getStoriesByProvider` in `story-api-contorller.js` and `getStoriesProviderByProviderCode` in `apiControllerHelper.js` (see fig. 2.4 on the next page for the code snippet with aforementioned methods).

```
// In story-api-contorller.js
async function getStoriesByProvider(req, res) {
    const { activeStories, mergedStories } = await
getStoriesByProviderCode(req.params.providerCode, res);
    res.json(activeStories.concat(mergedStories));
}

// In apiContorllerHelper.js
async function getStoriesByProviderCode(providerCode, res) {
    const searchObject = { providerCode, mergedSha: null };
    const activeStories = await findAsync(Story, searchObject);
    if (!!res) {
        if (activeStories.length > 0) {
            storeJSON(res, "story", activeStories[0]);
        }
        else {
            clearJSON(res, "story");
        }
    }
    searchObject.mergedSha = { $ne: null };
    const sortObject = { updatedAt: -1 };
    const mergedStories = await findAndSortAsync(Story, searchObject,
sortObject);
    return { activeStories, mergedStories };
}
```

Fig. 2.4, snippet of the code taken from `story-api-contorller.js` and `apiControllerhelper.js` respectively which provide an entire logic of an algorithm that was responsible for loading only up to two stories in the table.

Understanding the algorithm and altering it, bypassing the need for dealing with parametric polymorphism

Having found all the pieces of the code that were working together, I needed to understand and alter it to obtain the desired effect.

I approached it from bottom to top. First looking at the ` getStoriesByProviderCode`, I realised that it looks into Mongo DB database and searches for stories by providerCode (unique code assigned to every insurance provider) and looks for both, story that is active (there is meant to be always only one active story), and past, merged stories. It returns them as an object containing both as separate collections. Search criteria to find merged stories is `mergedSha`. Hence, when looking for active tory it is set to be `null`, and when looking for merged ones, it is specifically set not to be `null`. The method also makes sure to sort merged stories from newest to oldest (using Mongo DB attribute `updatedAt: -1`). The method only looks for active story if response has an array of `activeStories` that is not empty, and then it chooses the first one in the array (most up to date, consequently, the only active story).

Secondly, the ` getStoriesByProvider` method in `story-api-ctonrller.js` takes the returned object with both of the collections (active story and merged ones), combines it and returns it as a single array of both of them (keeping the active story at the top of the array). It uses the `.concat()` JavaScript build in method which combines two separate arrays into one.

Lastly, in the target-schema-display-contorller.js` `showOnlyOneMergedStory`, used by `getStoriesList` (in the same file) makes sure to detect if the returned array of stories has active story, and it keeps it. It does it by checking that the returned data is not an empty array and checks if the first entry does not have `mergedSha` attribute (meaning it is still active, as it wasn't merged). Additionally, it also checks that if the array has more than one entry, and the first one was an active story, it also keeps a second entry; this corresponds to the latest, merged story. Those two are then being used to be displayed to the user in the GUI (graphical user interface).

Knowing it all, I understood that the only change I had to make in order to be able to return all of the previous stories associated to given provider, was to make sure that `showOnlyOneMergedStory` method would in fact be renamed to `recordStoryNubmers` and would just record the given data and keep the Angular broadcast about information that provider story number has changed.

As can be seen on the fig. 2.5 below, I only had to remove unnecessary logic from ` showOnlyOneMergedStory` (I renamed it to `recordStoryNubmers` in another GitHub commit), and make sure that it just saves the `storiesByProviderCode` object to be equal to the passed in data, given passed array is not empty (changes on lines 147-160 under `target-schema-display-contorller.js`).

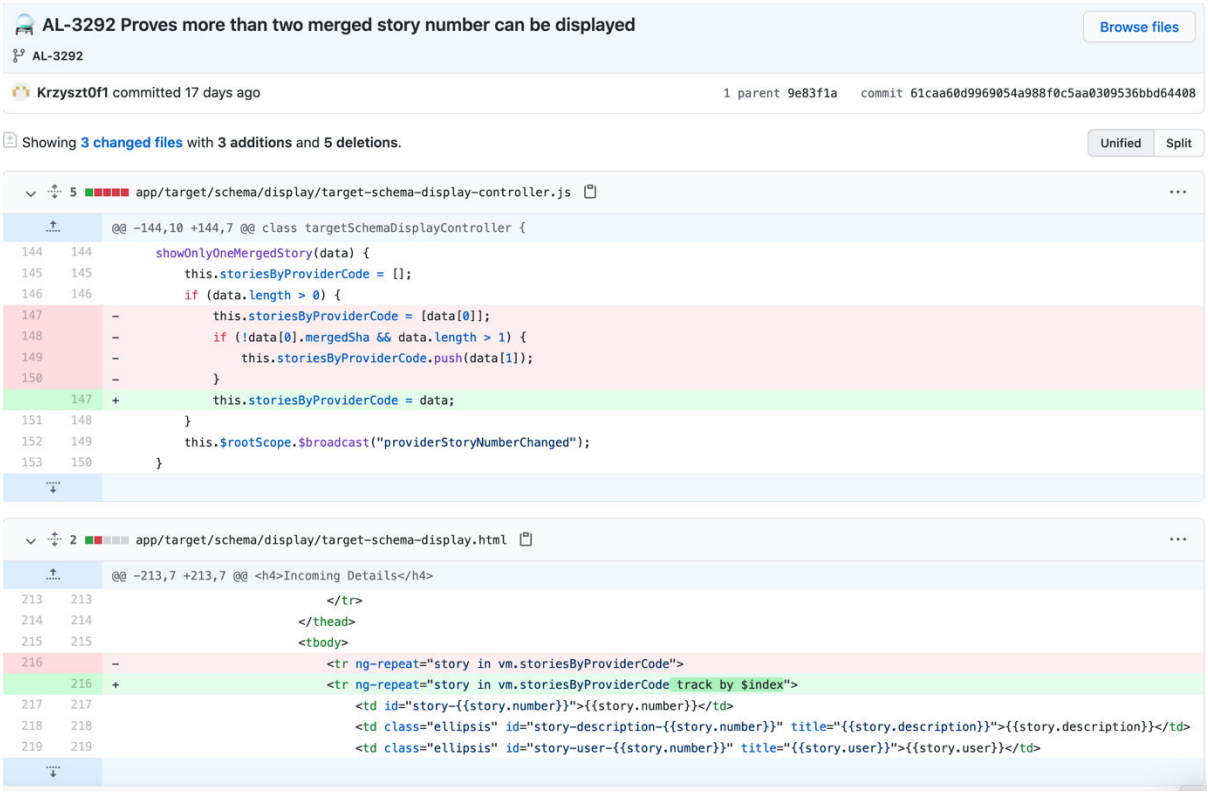


Fig. 2.5, documents the change I made to the code in order to experimentally allow the stories table to display more than two entries. This has removed a complexity in `showOnlyOneMergedStory` which existed due to original need for dealing with parametric polymorphism. Parametric polymorphism arises whenever a function or a data structure is free to return any type of object. In this case, passed in data was an array of objects. However, the objects in JavaScript can contain any type of data. Hence, the need for `showOnlyOneMergedStory` to previously check if its first element includes `mergedsha` property.

Interestingly, I also had to make a change to the html, presentational component (as seen on the figure above with change made on line 216 of `target-schema-display.html`). This change was required since, the Angular table element needed to have something to track the data passed in. It wasn't necessary previously, as with only two, distinct (active and merged) stories, they had different attributes, yet with multiple merged stories passed in, the table was unable to track which story was already passed in. I had to keep it tracked by `index` (position in the returned data array) as from the point of view of the Angular JS table element, it was the only object attribute that was unique for each story passed in. Attempts to track it by for example `mergedSha` attributes were resulting in some stories being read and displayed by the table multiple times.

Investigating hyperlinking story numbers to Jira tickets; adjusting the frontend

Lastly, to conclude the ticket, I also went on and investigated if it would be possible to include the story number entries in the table as hyperlinks that would redirect the user to relevant Jira tickets.

It is a feature that is lacking in the old UI, but the user base was fond of potentially introducing it. It did initially make me scratch my head. I wasn't sure of what would be the URL structure needed to access specific tickets. I was sceptical as the information that is being passed from the database about each story number does not include any URL's; this meant I needed to find a way of generating relevant Jira links.
In the end, after inspecting Jira ticket links, I realised that it would be rather easy. Each Jira ticket can be accessed via the following link: `https://comparethemarket.atlassian.net/browse/`. As such, the change I had to make, was to take the returned data (in a form of a string) and append it to the URL and decorate it as a html link component. This was not a difficult data formatting operation. The execution of code change can be seen on the figure below (fig. 2.6). The result of the spiked code on the possibility of extending number of entries to the table and including Jira hyperlinks can be seen on the fig. 2.7 below and the user's (Nicola) approval on fig. 2.8 on the next page.

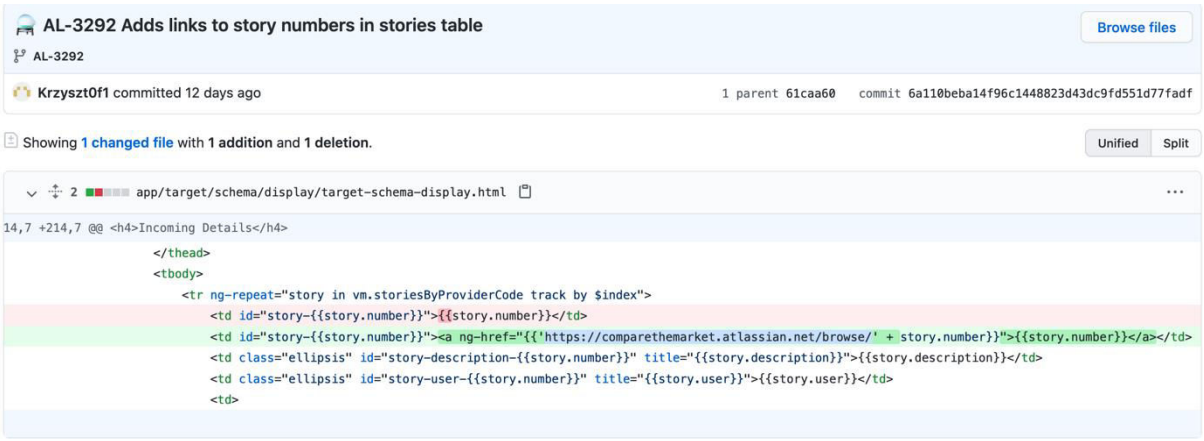


Fig. 2.6, screen of a change I did to the `target-schema-display.html` file committed to GitHub. This single change, which involves a simple data formatting of received string (story.number) into a URL address string by string interpolation, and passing it as a html link rather a simple text, has successfully made the story number display as clickable links, redirecting to relevant Jira tickets.

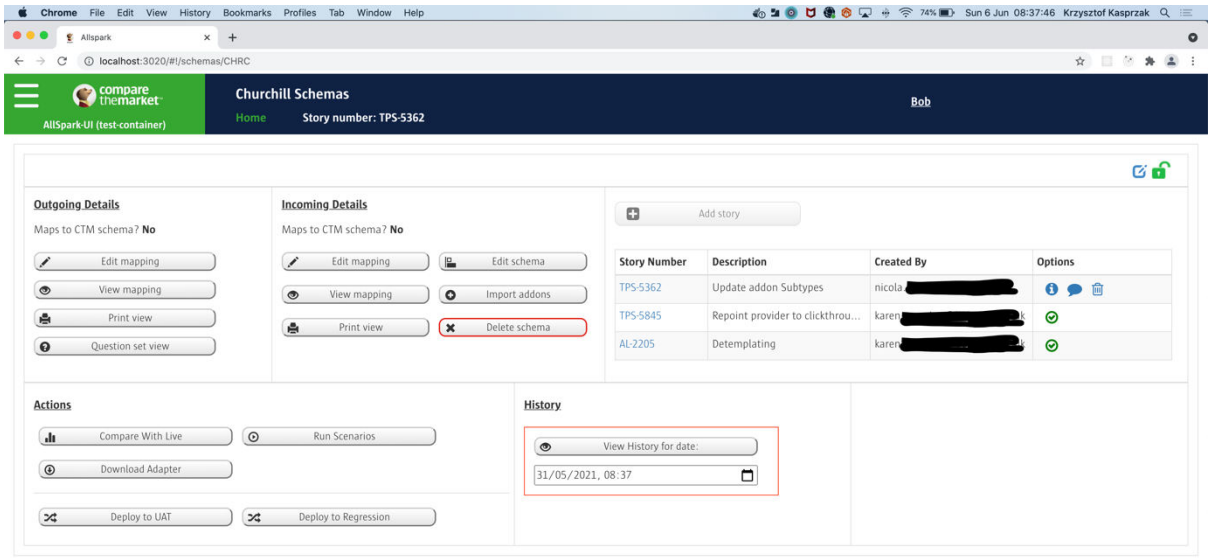


Fig. 2.7, screen capture of spiked changes reflected in updated look of the story number table in the top right corner of the schema's page. As can be seen, the changes I made to the html, view layer and business logic of getting all of the merged stories, allows to see more than just two merged/active stories. It also hyperlinks the story numbers to corresponding Jira tickets (`Story Number` column entries).

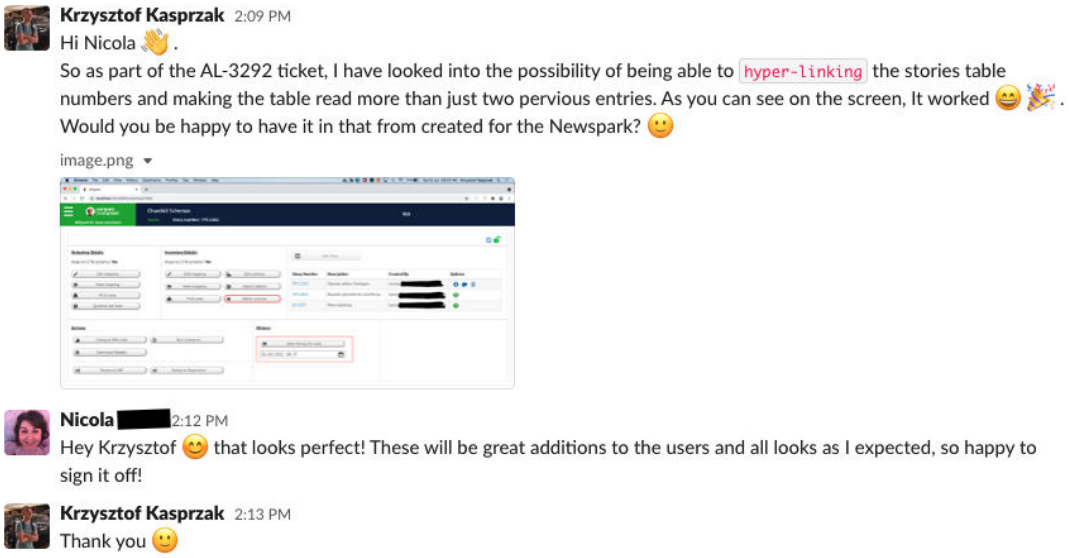


Fig. 2.8, a screen of how the changes I spiked were approved by Nicola to be introduced in the react rewrite; this includes both, story numbers hyperlinking and extending the table with more entries.

In conclusion

Things I learned and improved upon during the work

This varied ticket required from me to not only perform programming activates, but also pushed me to collaborate extensively with other stakeholders (users, business analysts, UI designers). It also has required from me to be capable of creating various development related artifacts (wireframes, activity diagrams and sprint tickets).

With that, I had a chance to learn how to create UML (unified modelling language) activity diagrams, improve my skills in creating website mock-ups and practice my communication skills.

Alongside this, I have also endured on uncovering, understanding, and modifying existing algorithms in the existing Allspark UI (retrieving more than two stories), manipulating and formatting requested data to desired form (displaying retrieved story numbers as hyperlinks) and further practiced how to track and trace flow control of projects written in Angular JS Framework.

Things I would do differently next time

Quite frankly, the only thing I will do differently next time has to do with the way I would prompt my user base to decide which GUI design options to choose. When it came to collecting further feedback based on final wireframes Tim and I created, they felt pressured to decide which options they would prefer.
To ease this feeling of being under pressure they experienced, next time in situation like this, I will inform them in advance about any input I might need from them.

Value I brought and impact I made

Summarising, by contributing to the ticket described in this piece, I have brought a good communication between project stakeholders, and made sure that user desired functionality was investigated to secure its future implementation. By creating UI mock-ups, I contributed towards creating business and requirements analysis artefacts, which will positively impact (improve) clarity of upcoming development tasks (creating the provider details' page).