# Adding Spelling Suggestions to Search Tool

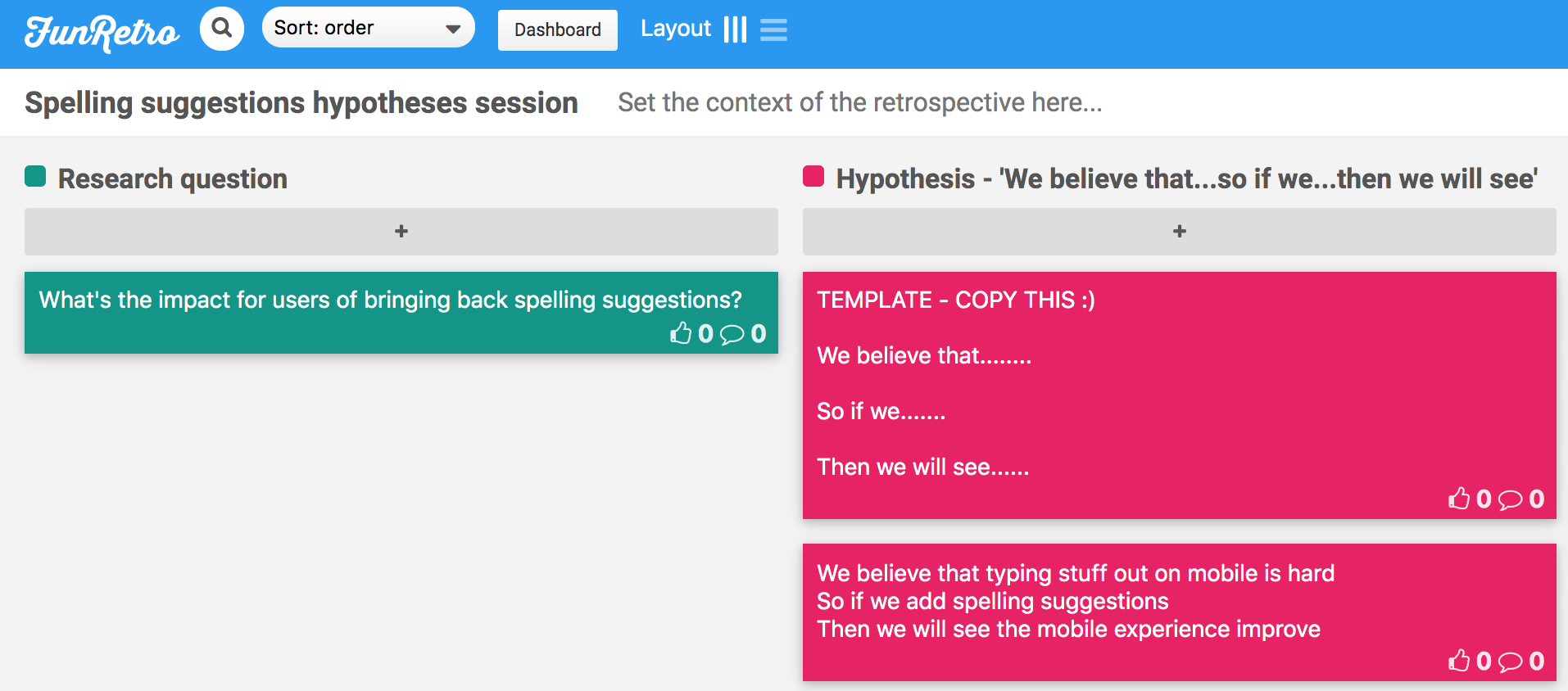
The languages that I used during this ticket were Ruby, Javascript and HTML/CSS, and there were also some command line functions.

The project that I was working on for this piece of work was around a search tool. Users would be able to use this tool to locate specific content that’s on the website. There is a lot of content, which means the search tool needs to be able to handle a wide variety of terms. There are also a lot of potential users (anyone in the UK), which means there would be a wide variety of computer literacy when using this search tool. As a result, it needs to be highly accessible, and one of the ways that this was to be achieved was to use a spelling suggestions feature.

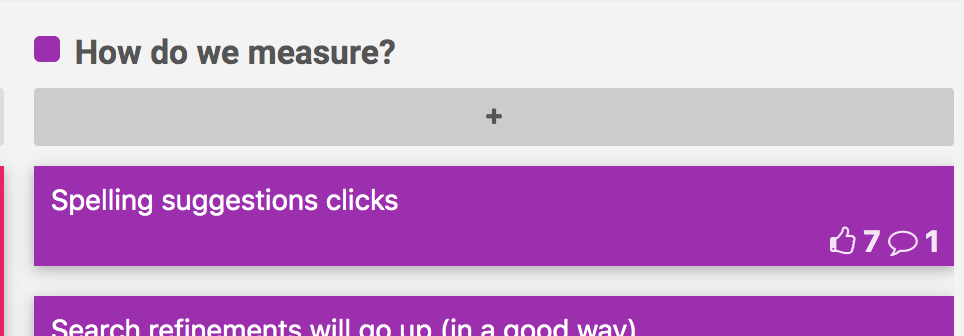
The spelling suggestions feature would provide users with a best-guess suggestion for words or phrases that it identified as misspelled. For example, if someone typed ‘*Incmoe tax*’, it would ideally suggest ‘*Income tax*’ as a suggested search term. Users could then opt to click on that suggestion, and have the new search roll with the suggested term.

Initially my team had a hypothesis session to work out how we would design this feature, and what metrics we use to track its success. I was part of this session, and I made several suggestions on what to measure. In particular, I made the point that the mobile and desktop usage may differ, and it would be important to track these separately. It may be the case that more mistakes are made in one format than another, and that designing it differently for mobile and desktop could lead to more effective usage on both.

To do this session I used a tool called FunRetro, which allowed my team and I to put our ideas into a shared space, and then discuss and organise them. I paired up with the product manager to come up with a few ideas that would then feed back to the team. I used a format for the hypothesis, which is shown in the screenshot below.

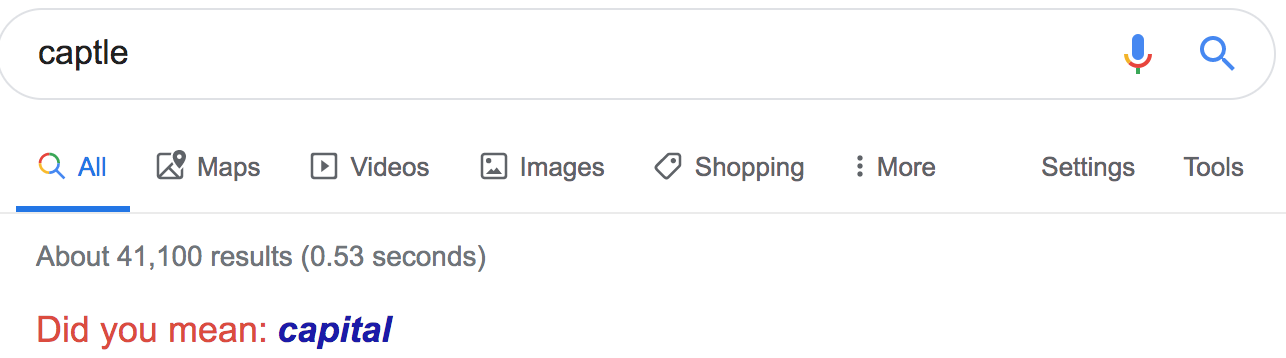


Myself and the team then started looking at how we would measure these hypotheses, so that when the spelling suggestions were launched, we could see what impact it was having. I suggested several metrics that could be used to test the hypotheses, and in particular I suggested tracking how many times people were clicking on the suggestion vs. just seeing the suggestion.

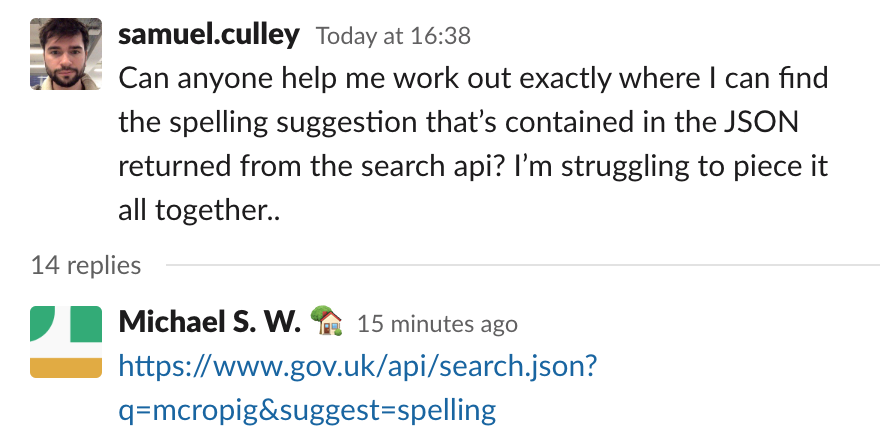


Once the measuring requirements had been established, I was able to help with the design for the spelling suggestions. This involved speaking with a designer and our front-end developer, and working out what the best design would be for the suggestions. There were a few options across the web that we were able to draw inspiration from, but we mainly looked at Google. Google search already uses spelling suggestions, and if we were to stick closely to the format they used, it was more likely that our users would be familiar with the feature.

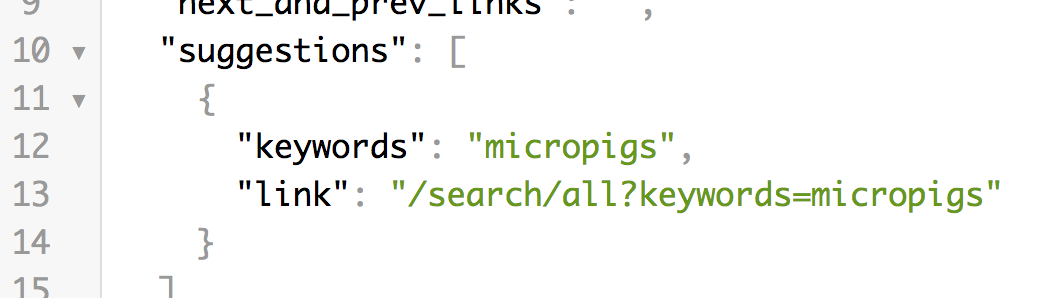
I then did some research on how Google had designed their system. I noted where the positioning for their spelling suggester was, and how it phrased the suggestion. I also wanted to see how prominently it was displayed, and looked at a few test cases for spelling mistakes, to get an idea of how it would work with different search phrase lengths.



In order to start building the feature out, I needed to understand a bit more about how the backend for it would work. I spoke to the senior developer on my team who was setting up the backend for it, and found out more about how the suggestions were generated.

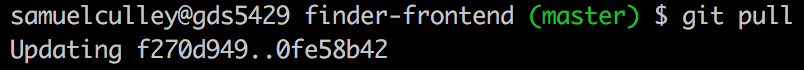


Whenever a user made a search, the results would be returned as part of a JSON object from the database. Those results would then be used to populate the page. There was also a check to see if there was a spelling suggestion was appropriate for the search term, and if so, this would also be included in the result set. The screenshot below shows the spelling suggestion for the term ‘*mcropigs’*. There was also a link provided that would go to a new search with the suggested term.



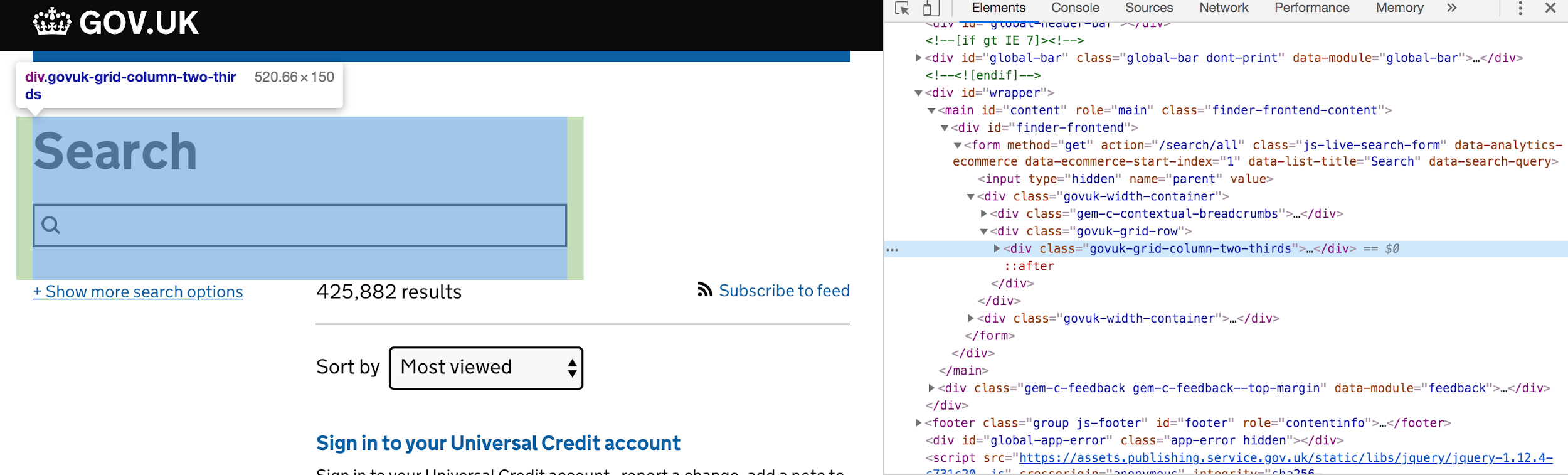
I had all the knowledge in place, and now it was time to implement the change. I knew that the spelling suggestion was available as part of a set of search results, and that I would be able to present it on the page using the design that had been agreed on. This would involve me modifying the layout of the page, and how the backend passed information to the frontend.

I first needed to find the appropriate application to make changes to. The tool I was working on had several key components, but I only wanted to focus on the frontend. I knew from previous work that this meant the app I needed to look at was ‘finder-frontend’. This had a Github repo, which I had already cloned onto my computer, but I pulled to make sure I had the latest updates.



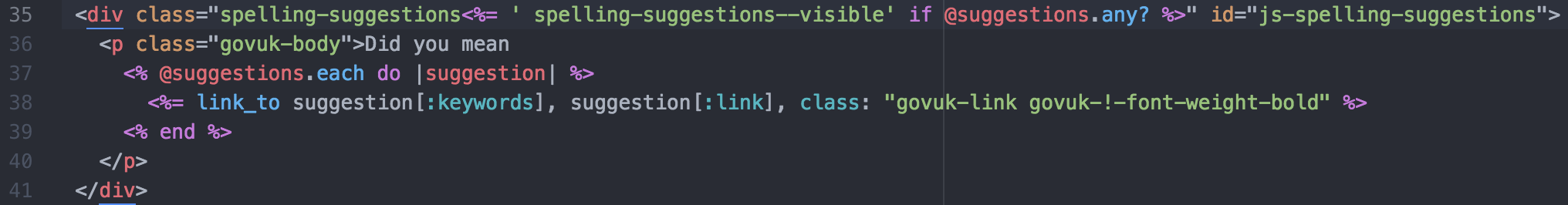
I then created a new branch, and gave it a title that would be useful for the rest of the team. I settled on ‘bring-back-spelling-suggestions’ as this was appropriate for the context. Once I was on the new branch, I was able to make changes without worrying about affecting the master branch.

I then had to work out which bits of the code I would need to change. I already knew how the design for the suggestions bit would look, so I decided to implement this first to make testing easier. Once I had locked in the display side of it, I could then make changes to the control layer and see how those affected the view. I therefore wanted to start with the HTML/CSS aspects of the page. The best way to find the relevant parts of the codebase was to look at the class names of elements on the page that I would want to edit, and search for those across all the files. I loaded up the actual page, and used chrome’s dev tools to inspect the relevant element.



From the screenshot above, you can see that the selected element on the page had the class ‘govuk-grid-column-two-thirds’. There were several hits with the search, but the key ones that stuck out were the ‘finder-frontend.scss’ and the ‘\_show\_header.html.erb’, as these would both be affecting the area of the page that I was concerned with. I decided to open both of those up to try and find what I was looking for.

I went to the .erb file first, as this would be where I could put some HTML in to occupy the right spot on the page. I found the area that was above where I wanted to put the suggestions, and started to put in some code beneath. The code below shows the finished version, but I’ll walk through it line by line.



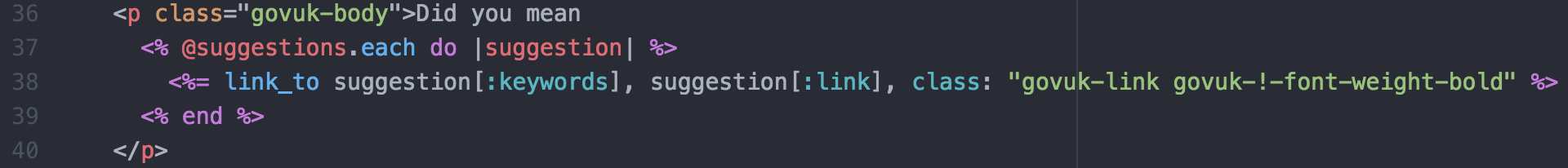
I initially created a div which would contain the whole body of work. This would separate it from the div above, making sure that they didn’t interfere with each other. To name the div, I used a few guidelines. On my team, we use particular gems to provide corporate styling, and these can be imprinted onto HTML elements by calling them with specific class names. However, my div was the child of a parent div that also used this, which meant mine would inherit any required styling at this point. I therefore needed to create a class that could specifically identify this segment, so that I could add my own styling as needed.

I learned about the Block Element Modifier methodology, which was the type used where I work. This involves creating classes and styling that follows the order of BEM, meaning that at first I would be creating a block, and within that creating elements. Finally, if I need to modify anything about these, a modifier could be added to create special styling for this.

I already knew that the spelling suggestions block would only show up in certain circumstances, so I needed to give it a base Block class name, and then add a modifier to it which would determine whether it was visible or not.

The code on line 35 shows this in action - it uses the erb method of inserting ruby code into the HTML file, and I added a conditional that checked to see if there were any spelling suggestions that had been returned in the search result set. If there weren’t any, then I wouldn’t need the block to be visible, so it wouldn’t get the class name ‘spelling-suggestions--visible’. If there were suggestions in the results, then it would add this class name, which I could then use later to either hide or show the block accordingly.

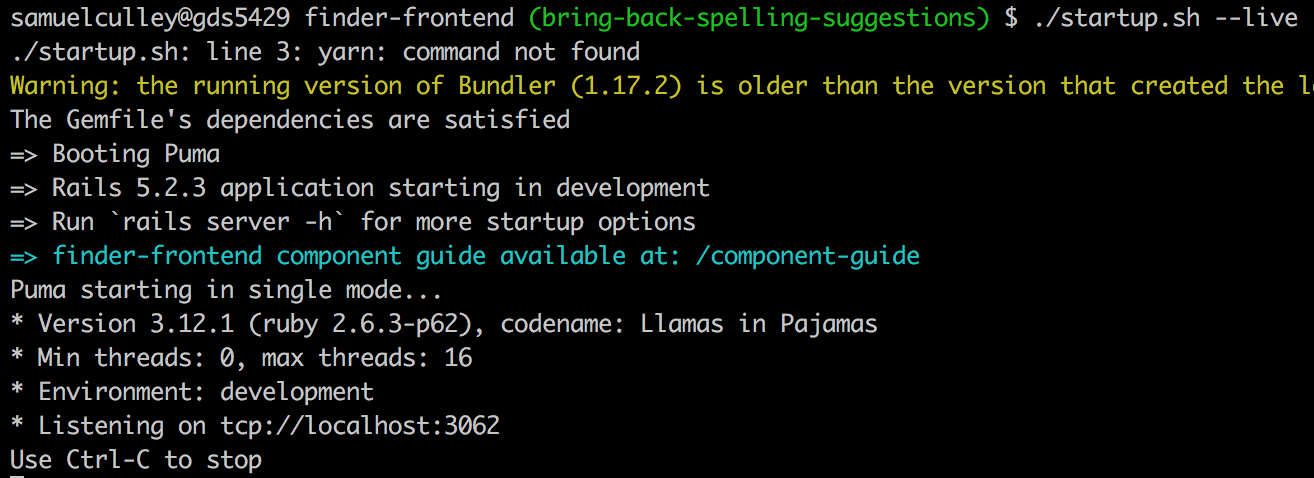
I then moved on to creating the text part of the feature. I used a paragraph tag, as it would contain text, and gave it a class name that conformed with my organisation’s gem, ‘govuk-body’. I wanted the text to always say “Did you mean”, and then substitute whatever the suggestion was at the end. As it would always have the same starting phrase I put this in first.



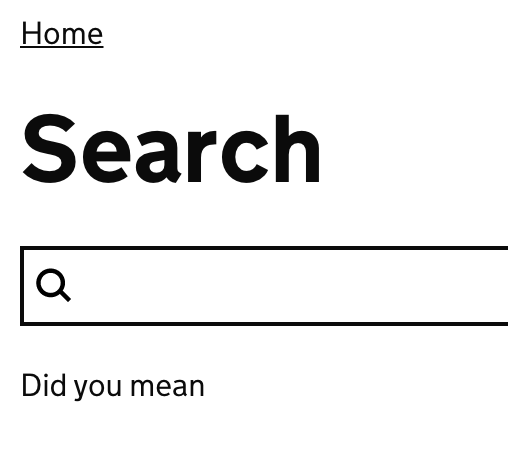
I then wanted the suggestion to be put in, but I also needed the suggestion to link to a new search with itself as the search term. I wrote a loop that would iterate over the suggestions that were returned in the JSON object, and create a link to that suggestion. This used erb Ruby code which would take the ‘keyword’ value from the suggestion object and use that as its text, and then take the ‘link’ value as its hyperlink.

The class name for the link was based on my organisation’s styling gem, and it would style it as a link, as well as ensure that it was bold text.

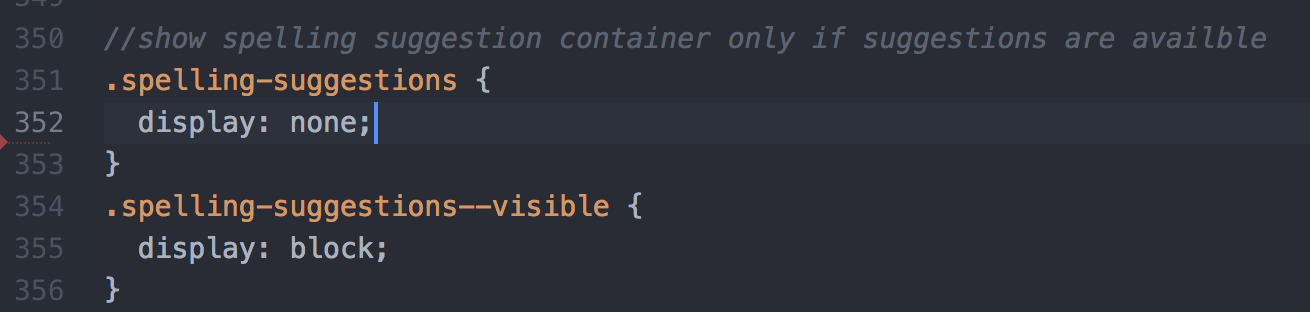
Once the basic HTML frame was ready, I needed to manipulate it from the controller. I tested it in browser by spinning up a local version of the application. I did this using a script that was available in the repo, which would run through all the app’s contingencies and create a locally hosted version.



This set a rails application up, which I then accessed in the browser by going to the localhost port 3062. I initially saw an issue whereby the spelling suggestion was visible even when no mistake had been made. This was because I hadn’t added any styling rules to the class for the suggestion block, which meant it would always be visible.



I then went to edit the ‘finder\_frontend.scss’ file. Here I would be able to specify custom styling for the block I had created. As I had created new classes for my block, namely the ‘spelling-suggestions’ and ‘spelling-suggestions--visible’ classes, I wanted to add these to the end of the style sheet.

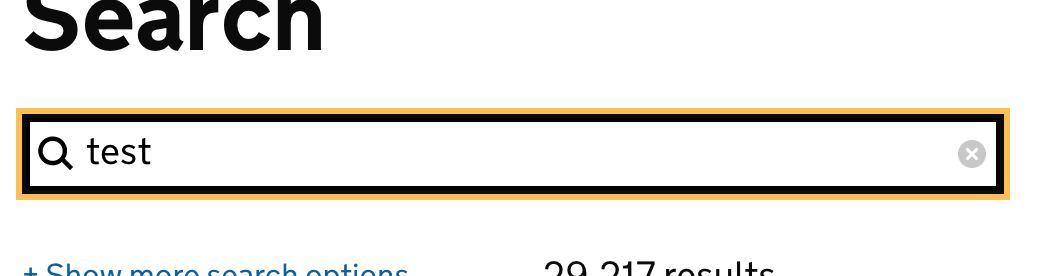


I wanted to make sure that the block wasn’t displayed whenever there weren’t any suggestions. My code in the template would only use the --visible modifier if there was a suggestion, so in any other case, I wanted to make sure the display was set to none.

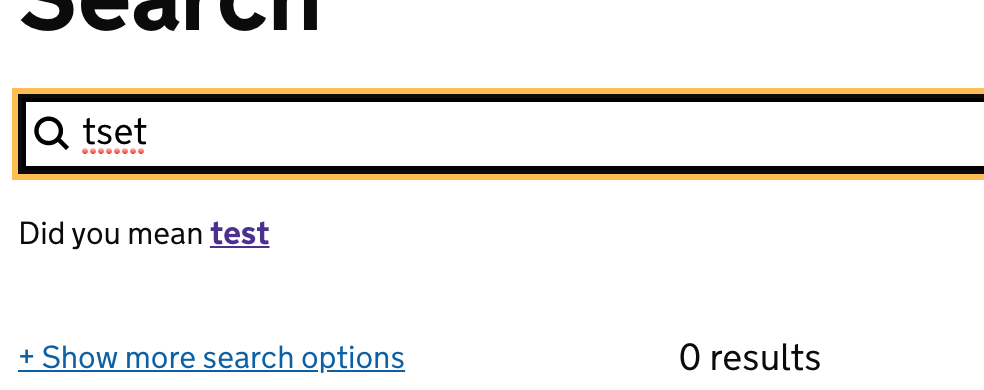
If the class for the block had the --visible modifier active, then the display would be set to block. This would override the previous ‘none’ display, as CSS works in a cascading priority. This means that the lower styles would override ones above it if they conflicted.

Once I’d saved this, I went back to my localhost version of the app to test it out.

No suggestion with no spelling mistake

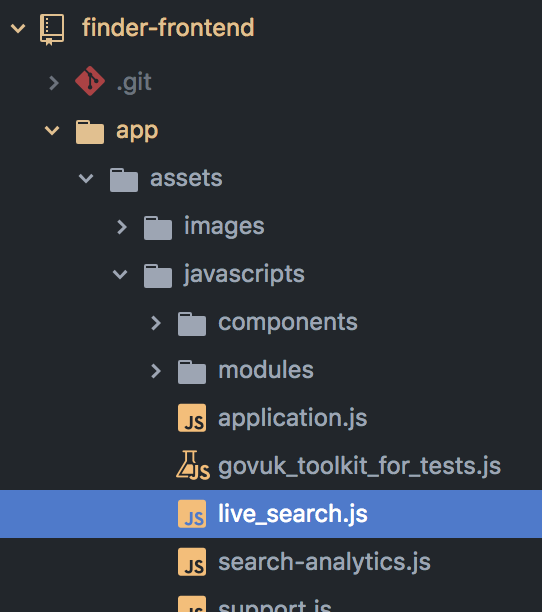


Suggestion with spelling mistake

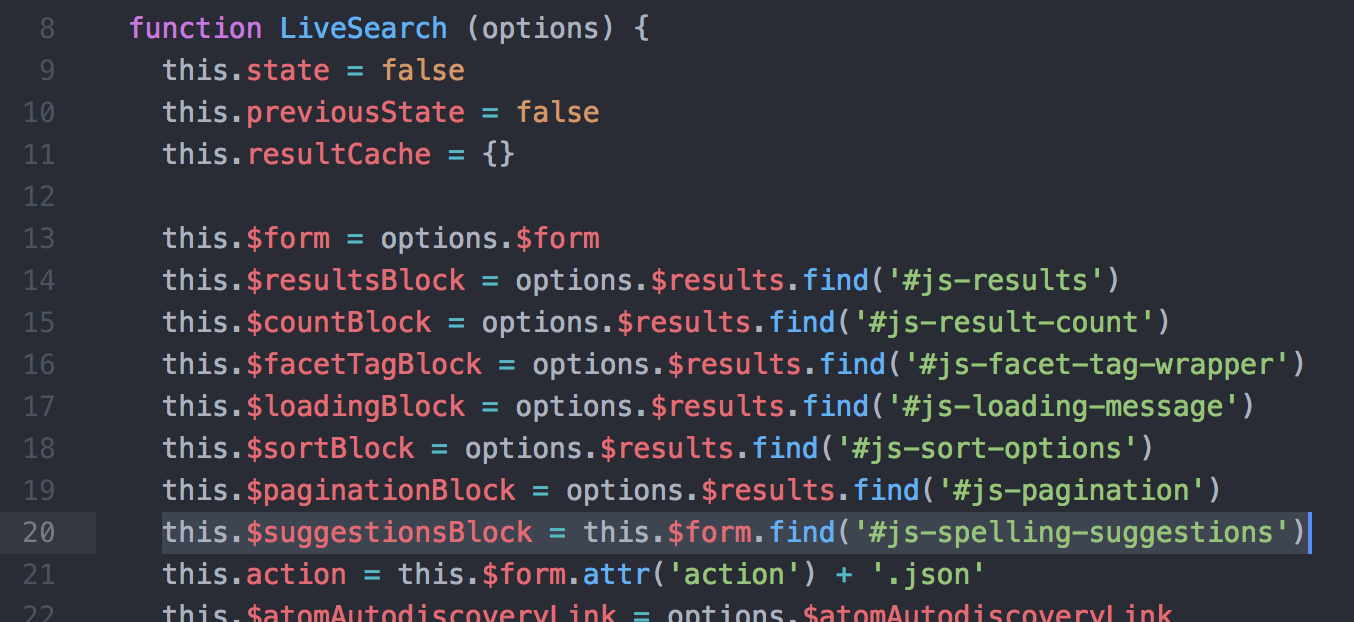


This looked like it was working perfectly, and displaying the suggestion with the appropriate link. However, this was only half the battle - another piece of functionality built into the page meant that users could refine their search, which loads a new set of results - but without reloading the entire page. Instead, only the results section of the page would change. This is achieved by using an asynchronous Javascript setup.

In the current state, if a user were to change their search term by simply typing in a new one, the results would update, but the suggestion would remain the same. I needed to piggyback on the current Javascript functions that updated specific parts of the page, and ensure that the spelling suggestion block updated alongside it. I first needed to find out where the results reload took place in the codebase. I looked through the apps folders to find out where the Javascript was stored, and found it in the assets folder. From here, I worked out that the ‘live-search.js’ file was the one that contained the relevant functions that handled reloading results with a new search.



The first thing I needed to do was make sure I could refer to the suggestions block within the LiveSearch function. This required me to set up an instance variable which would specifically refer to the suggestion block. As shown on line 20, this ties to the block by referring to the id ‘js-spelling-suggestions’ which I put in the HTML earlier. This also used the jQuery ‘$’ operator, which let me find the specific element.

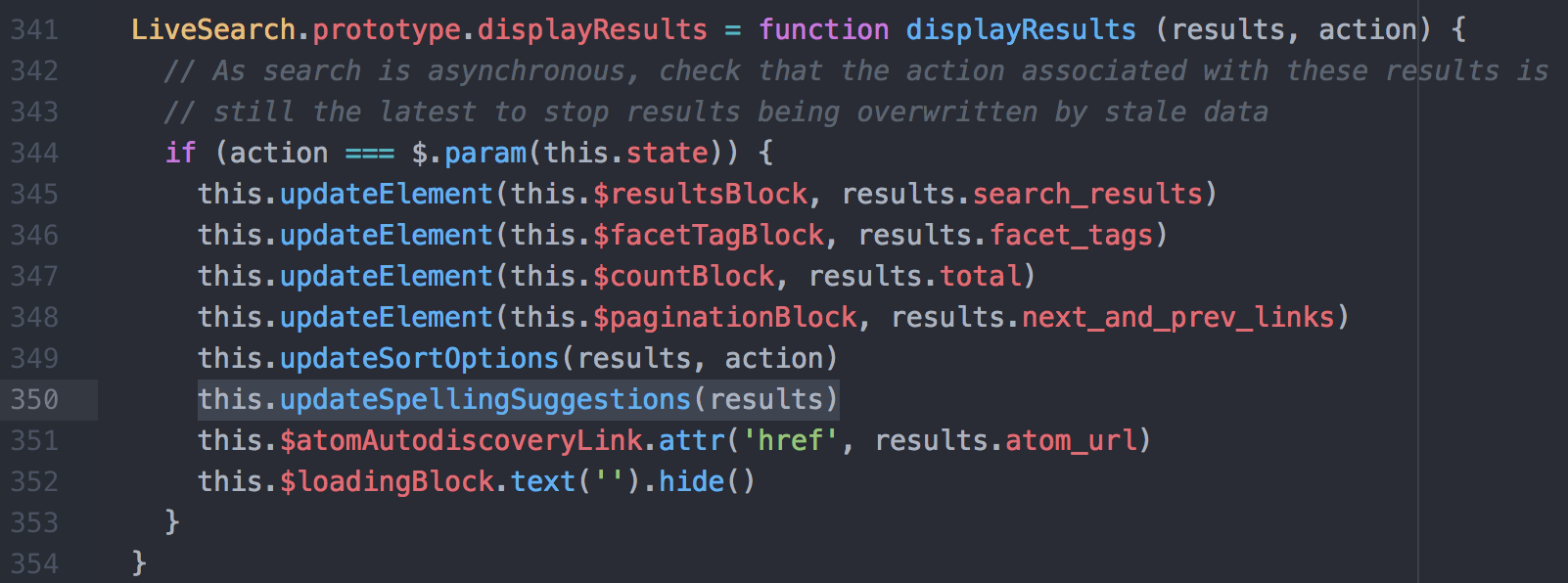


I needed to do two things at this point - create a function that would handle the re-rendering of my suggestions block, and insert a call for that function at the appropriate point. I decided that writing a token function first would be ideal, so that I could more easily test when it was getting called. Once I was comfortable with my function call, I would finish designing the function to act appropriately.

I created a new function called updateSpellingSuggestions. This would take in a results parameter, which is where I would get the wording for the suggestion and the appropriate hyperlink.



I then set it up to console.log() a test string, to determine when it was being called. I then needed to work out where I wanted this function to be called - I searched through the other JS functions, and found one whose name seemed to be relevant; ‘displayResults’. This function is called whenever a new set of results were obtained, and it ran through a series of functions designed to update aspects of the page. This was perfect for what I wanted, so I added my function to that sequence, on line 350. This took in the results parameter from the parent function, which I would be able to use in my new function.



I then ran the local page, just to see whether the console.log() within my function was executing. I loaded up the page, and then put in a search term. On pressing enter, I saw the console.log() message show up. This meant that my function was being run.

At this point, I needed to add code to this function that would update the spelling suggestion block of the page with any updated information. The code below is what I wrote, and I’ll explain it line by line.



On line 225, the function will stop executing if there isn’t even a suggestions block on the page. This is because this feature is limited to only one of the search tools, and as it’s the only one with the suggestions block, I didn’t want any changes to happen in any other tools that didn’t have that block.

On line 226, I set up a boolean variable which will return true if there any suggestions in the new result set, and false if there aren’t any.

On line 232, I set up a conditional, which uses the previous boolean. If there is a suggestion present, this code block will run. First, it finds the <a> tagged element in the suggestions block, and removes it. This will take out any old suggestions from the previous search.

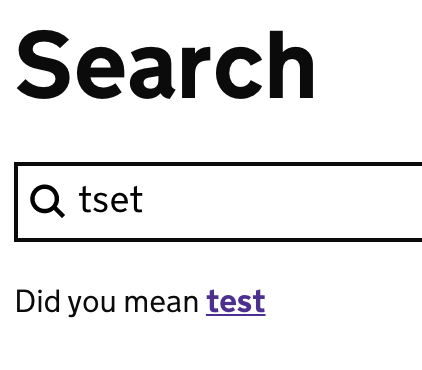
Next, it iterates over the suggestions in the result set. Although there is only ever one suggestions, I still wanted to iterate over it, as there may be later cases where more than one suggestion is provided.

On line 235, it sets up a new variable, which is an <a> element. This contains the new suggestion’s information, where the text is the suggestion, and this links to the new search term. It also sets the class to this new element to match the original one.

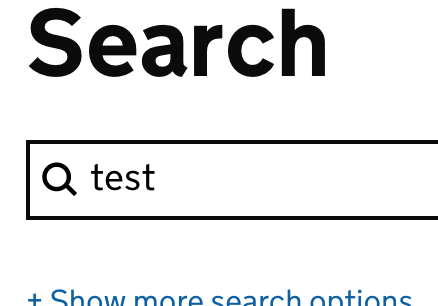
Lastly, this loop appends the new suggestion link to the <p> element, which completes the suggestion on the page.

The last step for the updateSpellingSuggestions function was to hide the element if no suggestions were present. On line 245, the class attribute for the spelling suggestions block is modified based on the suggestionsPresent variable - if a suggestion is present, then the block’s class is set to visible. Otherwise, the element is set to the ‘spelling-suggestions’ class, which I had set earlier to not be visible.

I ran the page to see what happened. When I first entered a search term that was misspelled, the suggestion showed up.



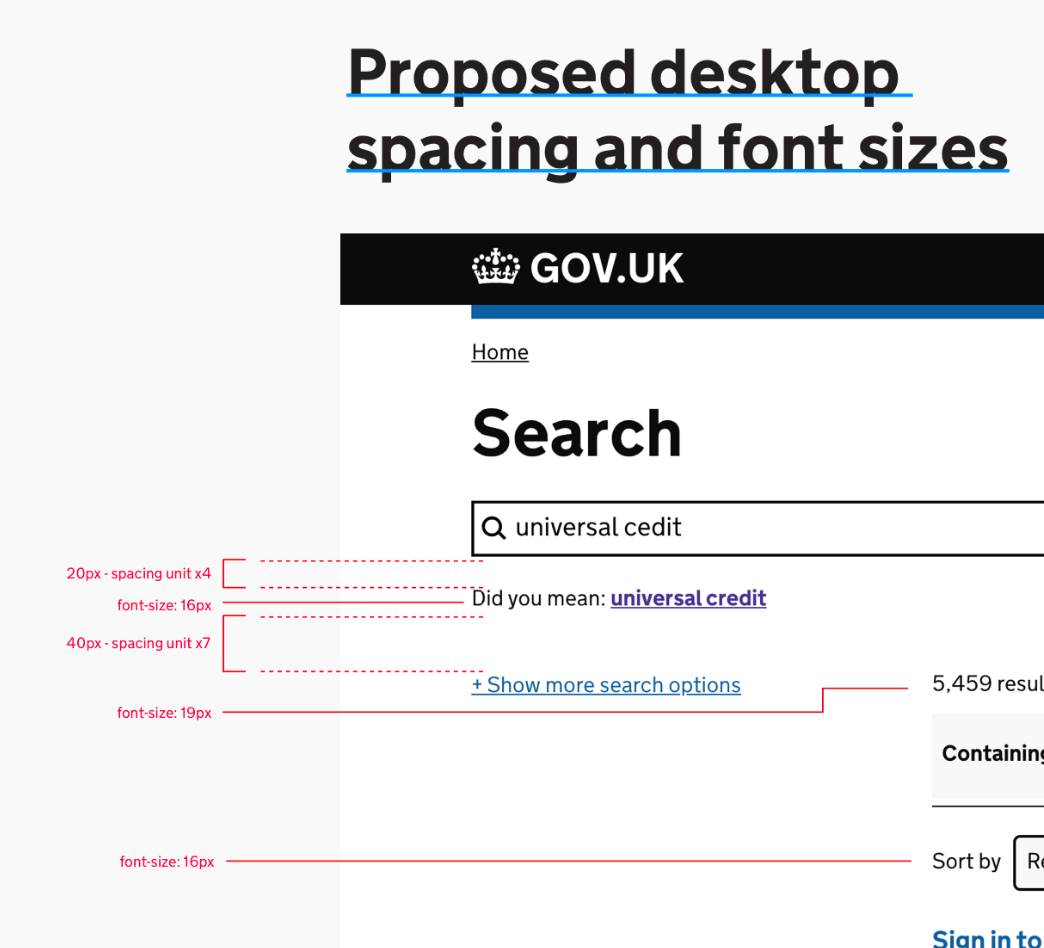
When I manually changed the search term to be correct, the suggestion then disappeared

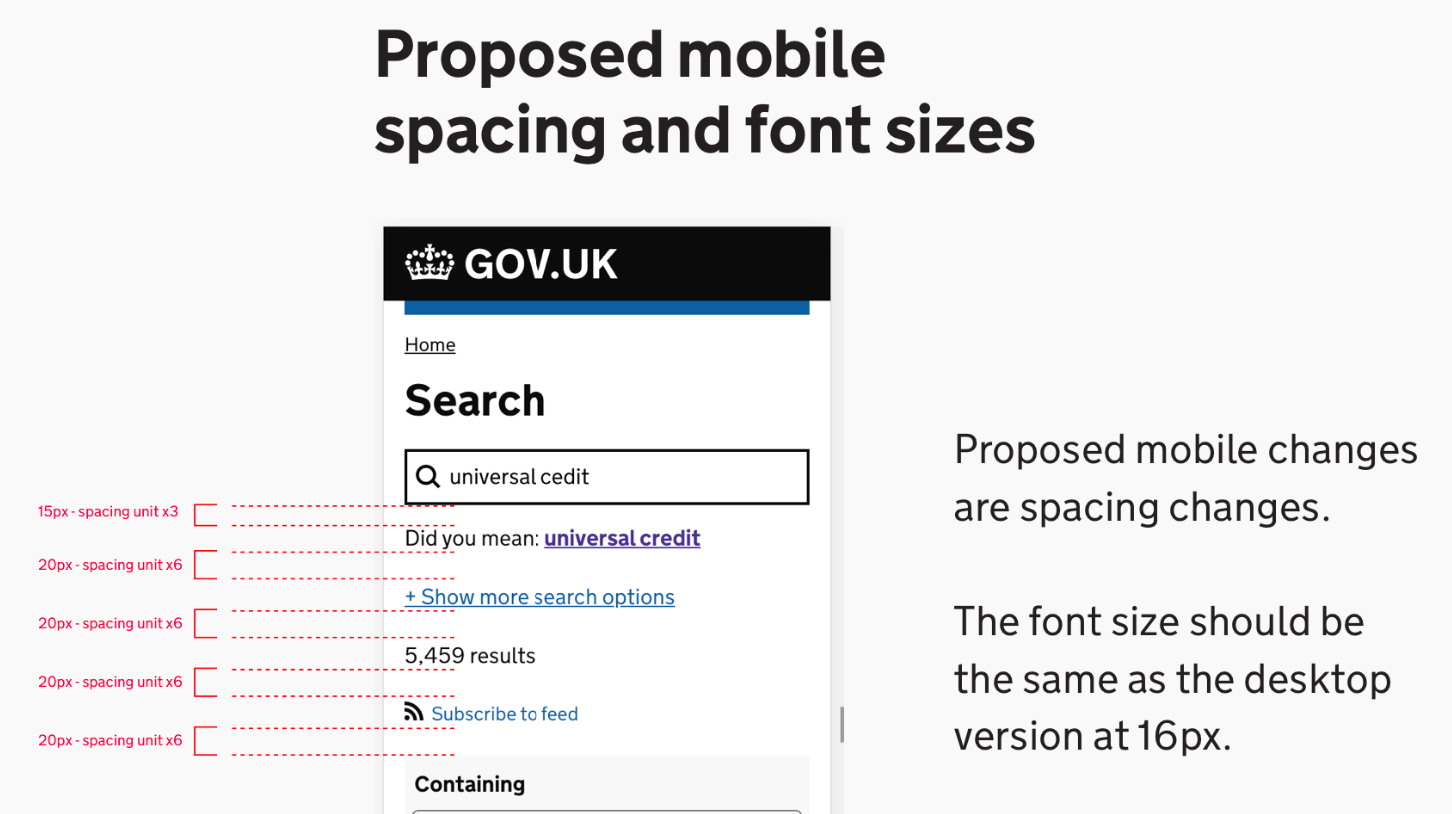


This meant that the function was executing appropriately, and removing the suggestion when the search was changed to a correctly spelled term.

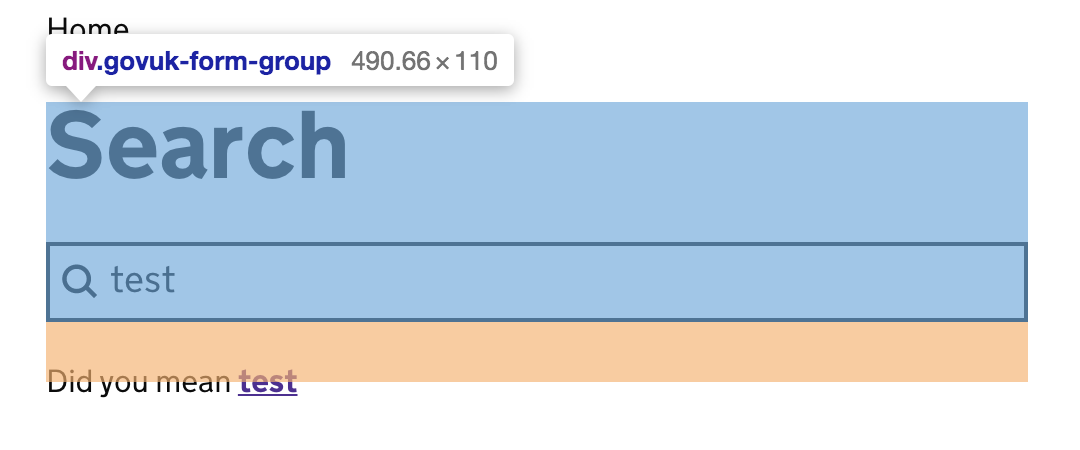
At this point, the feature was functional, but there were still some further changes in the ticket that needed to happen. I had asked the interaction designer on my team to take a look and see if the layout was appropriate, and he had provided me with some feedback. He wanted to make sure that the layout was responsive depending on the user’s device, and that the spacing between elements was consistent on the page, as the new suggestions block had highlighted the inconsistency.

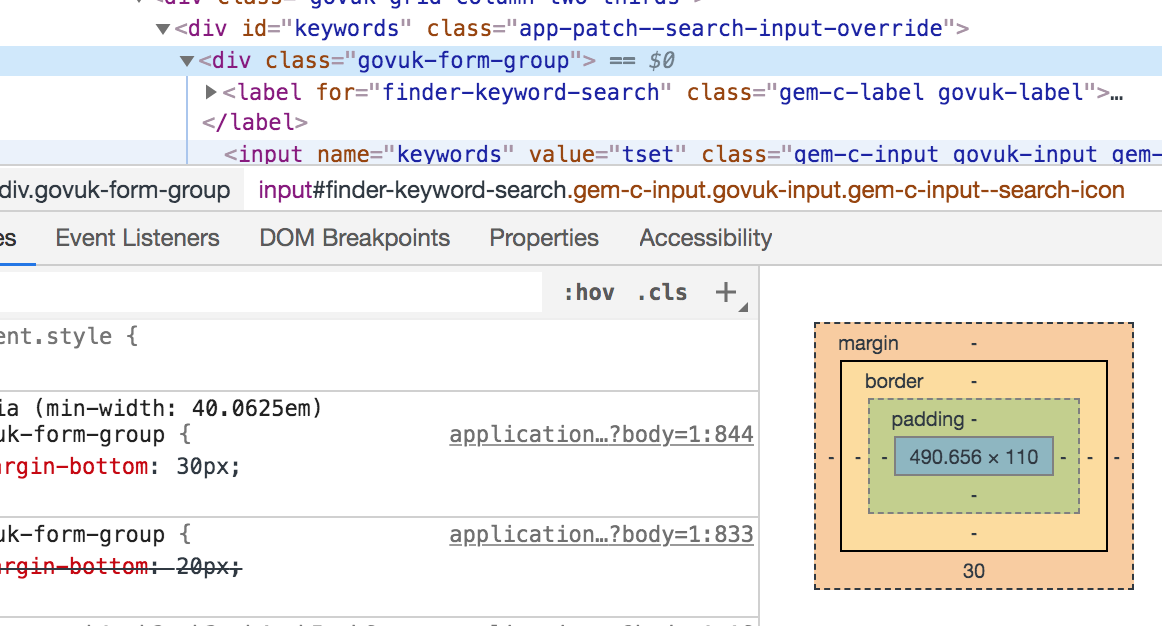
He added a few screenshots to the ticket, which set out his requirements for the styling on the page.





I then had to make changes so that these requirements were met. In order to work out which elements I would need to modify, I opened up the Chrome dev tools with the local page open. From here, I was able to select specific elements, and view the spacing in between them in numerical terms.





From here, I could put a number to each element’s distance between each other. I started out by looking at which elements would be appropriate to modify the margins for. Because the distances would be different depending on the screen size, I had to make two separate instructions in either case.



I decided that further modifying the additional block that I had added would be the easiest way to change the spacing. This would ensure that it would fit neatly with pre-existing elements, and as it sat in between most of the original elements, I could modify the margins above and below it so that I was making as few changes as possible.

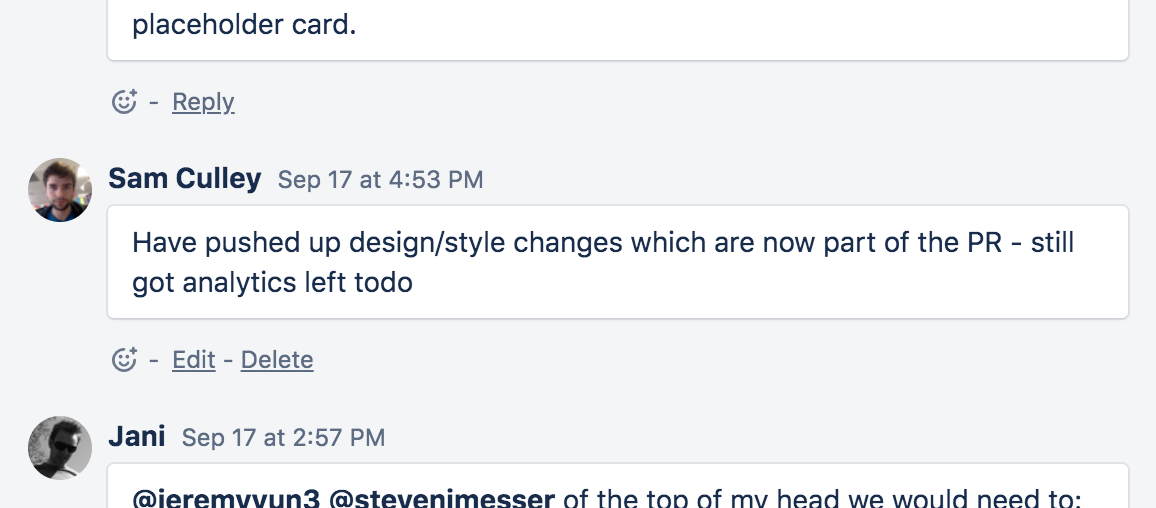
My organisation uses its own consistent spacing, which was available through a gem. Instead of using a pixel description, I would need to use the requirements from the designer in conjunction with the current spacing to work out the changes needed. For example, the gap between the search bar and the suggestions would need to be 3 spacing units. This meant that I would actually need to reduce the current distance between elements, so I added some CSS that set the margin-top to be -(govuk-spacing(1)). This meant that my suggestions block would be pulled closer to the element above it, thus making the spacing more consistent.

I also needed to ensure that these spacings were applied only on the right devices - in the above screenshot on line 350, I added a check for the device type, which is from my organisation’s gem. This actually works by checking the screen width, and if it’s above a certain width, it treats it as a tablet. I then set the margins for the element to match the requirements from the designer for the desktop view.

There were some further changes to the CSS that I made to ensure that the requirements were met, looking at the font size for certain elements. These also had the same logic that checked against the display’s size, and only affected the styling when it was a tablet or bigger.



At this point, I had completed what I was able to for the ticket. There was still a requirement left on it to add analytics tracking. This was something that I wouldn’t be able to do, so I needed to hand it over to the performance analyst on my team. I summarised the work I had done on the ticket, and linked to a pull request that I had made for this branch of work. This meant that others would be able to see the changes, and add additional changes and suggestions.



## Summary

Overall I thoroughly enjoyed this project - it had a lot of discussion beforehand which I was part of, and the design work that went into it was very visible. I felt like I understood a lot of what was going on, and the reasoning behind a lot of the designs. I also developed a further understanding about my organisation’s in-house styling rules, and some of the conventions around frontend design. I worked quite closely with the frontend developer and the interaction designer on my team, which let me observe the dynamic between them and the rest of the team.

I think my approach to this project was good, but I would like to have been able to go further with the ticket - implementing the analytics and some tests for it would have been great, and it’s a shame that I wasn’t able to get that far. Working to specific design specifications was really helpful, and my appreciation for clear requirements has only increased!

The end result for this project was a spelling suggestions feature being present on the main search tool that I’m working on. This will suggest spellings if it thinks there’s been a mistake, and lets the user refine their search easily and quietly. Additionally, the tracking that has been added will make it easier for us to identify what issues people are having when using our search tool, letting us make more informed decisions about improving it down the line.