Spike Report

RAS Frontstage

Introduction

Five day time-boxed spike into the front-end state of the RAS Frontstage user interface.

As with the Spike report on the Response Ops UI, I'll be considering the frontend with particular reference to:

- Separation of concerns; keeping logic separate from templates.
- Opportunities, viability, and value of migrating components to the SDC Pattern Library.
- Adherence to correct usage of the Pattern Library guidelines.
- Opportunities for better application of the external dependencies of the Pattern Library
- Correction of incorrect frontend code, or of errors.

Aims of report

- Review current state of the RAS Frontstage UI
- Find problems, and recommend solutions
- Find opportunities for improvement, and suggest methods through which to achieve this
- Write up recommendations from which to generate tickets.

Code Structure

Templates

On the whole, the templates are of a better structure and condition than those in the Response Ops UI report I wrote. There is better use of blocks, and more recent refactoring has taken place.

The code still has some issues:

- Convoluted conditionals:

```
{# Sign in page #}
{% if errorType %}
<div class="panel panel--error">
    <div class="panel__header">
        {% if errorType == "failed" %}
        <h1 class="panel__title venus">Incorrect email or
password</h1>
        {% elif errorType|length > 1 %}
        <h1 class="panel__title venus">There are {{ errorType|length
}} errors on this page</h1>
        {% else %}
        {% for error in errorType %}
        <h1 class="panel__title venus">{{ errorType[error][0] }}</h1>
        {% endfor %}
        {% endif %}
    </div>
    <div class="panel__body" data-qa="error-body">
        <p class="mars"><a href="#sign-in-details" id="try-again-link"
class="js-inpagelink">Please try again</a></p>
    </div>
</div>
<br />
{% endif %}
```

Pages like this could benefit from:

- Extracting the error panel into partial blocks, or a number of partial blocks
- Providing the template with more values, that allow the conditionals to be simpler:
    - `showErrorCount` - could toggle the display of the `<h1 class="panel__title venus">There are {{ errors|length }} errors on this page</h1>`
    - `errors` - a list of all the errors to show. The view would need refactoring to only return `Incorrect email or password` when that is an error in the page, and to return all the errors in other circumstances.

- Convoluted variable use: `{{ errorType[error][0] }}`, which can be fixed by providing simpler data structures from the view
- Splitting pages with multiple output possibilities to include blocks for each possibility:

```
{% if conditional %}
{% block conditional-is-true %}
{% else }
{% block conditional-isnt-true %}
{% endif %}
```

NB: This deliberately avoids indentation that would be based on meta-elements, but a merit-based assessment should occur when considering whether the code readability or outputted HTML is priortised. If the clauses are more complicated than this, readability is more important than outputted HTML.

- Common patterns (e.g. table rows, and pattern library components) are repeated throughout the templates, and could benefit from a DRY approach to code reuse. Various approaches can be taken to this, and Jinja2 provides several possibilities for discussion (e.g. macros, filters, Python modules)
- The Jinja2 whitespace settings could be similarly improved as the whitespace in Jinja2 on Response ops

On the whole, the RAS Frontstage has the same problems as the Response Ops UI, but to a lesser extent by a fair way.

Stylesheets (CSS)

There is ***very little*** separate CSS in the RAS Frontstage, with only 220 lines of CSS that isn't in the Pattern Library (37 rules, 4 components, 1 override). Looking at the code, we could extract virtually all of this into the pattern library, either as new components, or as new altered states of existing components. I think we could empty this file with relatively little effort.

CSS is not minified, but equally is a very small file. Whilst we could introduce a minifier if a chance presents itself, I don't think this is a priority at all. CSS loaded from the pattern library ***is*** minified and sourcemapped.

Javascript

The RAS Frontstage contains almost no Javascript, including only:

- The HTML5 Shiv for browsers under IE9
- The pattern library
- A simple inline script that removes the `no-js` class, to indicate that Javascript is available for use.

So, the page has a very uncomplicated set of scripts, and loads a single file of ~80k, which is a very good load size.

HTML Markup

Markup within RAS Frontstage is also of relatively good quality, with no obvious invalid markup, in contrast to Response Ops UI. Some points could be made though:

- Some use of `div`s could be changed to more semantically appropriate tags.
- Use of ARIA markup attributes could be improved in places, but is largely good already.
- There are some `OMITTAG NO` tags that self close, which is unnecessary, but unlikely to cause issues.
- There are occasional indentation errors.
- There are some instances of extra blank lines (more than one in a row).
- Indentation is in places mixed between 2 and 4 spaces.
- Very occasionally, invalid tags are used, for example `<emphasis>` (rather than `<em>`), which is used in `cookies-privacy.html`
- There is a use of `confirm` on one page, which should be replaced by a more appropriate way of confirming

Python views

Worthy of note to themselves, the views in RAS Frontstage are clearer and more concise, making them easier to understand for the casual editor or for developers with less experience of Python. This also indicates that the

extraction of some logic that is currently in the Jinja2 templates would be an easier refactor.

Design standards and adherence

Because the RAS Frontstage uses the pattern library, with very little frontend code external to that, it is largely very adherent to design standards set out. Because the pattern library is itself externally verified by DAC for accessibility, it's also very adherent to widely accepted standards on accessibility, including WCAG, and the Service Manual accessibility guidelines.

Notes on accessibility:

- In a small number of cases, we cross context elements as other element types without explaining this:

  ```
  <a href="[LINK TO SOMEWHERE]" class="btn">Button text</a>
  ```

  The above is functionally a link, and only forwards the user to another URL, but for design reasons is presented as a button. In this context, the role should be clarified using a `role` attribute. *NB: The `role` attribute should in this case be set to **whatever the element is doing**, not whatever it is styled as. The above code is correct if the button styled element is being used for navigation, and wrong if it's being used for a functional reason - this could be subjective.*

  ```
  <a href="[LINK TO SOMEWHERE]" class="btn" role="button">Button
  text</a>
  ```

  or

  ```
  <a href="[LINK TO SOMEWHERE]" class="btn" role="link">Button text</a>
  ```

  clarifies the ambiguity of the use of button styling, but link tags. Largely speaking, it's possible to avoid this just by not using anchor tags for buttons, but that is not always possible, as designers may want elements to appear out of context for style and convention reasons.

Build Processes

Like the Response Ops UI, the RAS Frontstage lacks a frontend bundler, or task runner. In difference to the Response Ops UI, though, it had one that was removed, and could be reinstated relatively easily.

Benefits of including a task runner or bundler:

- Allows for a 'watch' mode, for recompiling frontend code on the fly as it is changed in development.
- Allows for compilation of code to a minified state for production
- Allows integration of extra tools, such as:
  - Static analysis tools and linters, to help smoke test code, and highlight issues before they are deployed to an environment
  - Bundlers and transpilers to allow:

        - Writing of code in modules, to allow for clearer separation of code
        - Writing of modern code, that is transpiled to more cross browser supportive versions
        - Minification and other size compressing services.
- Allows for integration of test frameworks for the frontend should we need them
- *All of the above* allows us to highlight potential issues, or even cause builds to fail at an earlier stage than we currently would, preventing unnecessary use of build servers for invalid builds

Opportunities for improvement

Because the RAS Frontstage is already in a relatively good state, we have the opportunity to improve, rather than correct and fix it.

Some of these include:

- Introduction of an i18n library, and migration of template strings into it:
    - to separate strings into a separate area, to allow easier change of them without risking template validity
    - to prepare for the possibility of translation needs
- Use an Isomorphic SPA approach, or React SSR to create the frontend, to allow frontend developers to create a more manageable source code, with which to create more easily scaled UI.

Sum-up

Overall, the RAS Frontstage is in good state, and performs its task well. Whilst I have been able to make some suggestions, there are not many recommendation that require any large changes, and not many that I would consider vital.

The opportunities for improvement may offer good follow-on tasks, but none is a vital step, and should be considered Nice-To-Have rather than essential.

The RAS Frontstage would also benefit from several of the recommendations made for Response Ops UI, such as creation of a set of frontend coding style guides.

Recommendations

- Refactor and split templates into more separated forms
- Encourage code reuse in templates by extracting common patterns into reusable code
- Extract as much CSS as possible into the pattern library
- Add minifier and sourcemapper for CSS if opportunity presents itself
- Correct any invalid HTML
- Consider use of more semantically valid tags where applicable
- Correct indentation of HTML, where needed
- Clarify context of out-of-context elements
- Add in (or possibly re-add) task runner/bundler setup for build process and ongoing development