Spike Report

Response Operations Frontend

Introduction

A five-day time-boxed spike report into the current condition of the frontend of the response operation User Interface.

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 frontend on response ops UI
- Identify areas for improvement
- Make recommendations for future tasks

Code structure

On the whole, the use of Flask has provided for a natural leaning to a separation of concerns, so the use of Models, Controllers, Views, and Themes is already in a good state. The following are of note:

Templates (Jinja2 system)

Many templates are overpopulated with logic:

- If clauses are valid, but overuse of them should be replaced with booleans passed from the view.

  - Many convoluted if clauses are in place, which should also be migrated to be booleans passed from the view.
  - Nested if clauses should be avoided where possible, by separating templates into partials.

Good use of if clauses in templates

```
{% if errors %}
<ul>
    {% for error in errors %}<li>{{ error }}</li>{% endfor %}
</ul>
{% endif %}
```

Bad use of if clauses in templates

```
  {% if ce.statuses|length > 0 and (ce.responseStatus == 'Not started'
or ce.responseStatus == 'In progress' %}
```

- Table rows and other repeated DOM elements should be broken out into partials wherever the amount of repeated markup is large – for loops containing significant amounts of mark up are common.

Good use of partials

index.html

```
    ...various markup...
    {% if rows %}
        {% block table %}
    {% else %}
        {% block table_no_rows %}
    {% endif %}
```

partials/table.html

```
    {% extends 'index.html' %}
    {% block table %}
    <table>
        <thead>
            ...header markup...
        </thead>
        <tbody>
            {% block table-rows %}
        </tbody>
    </table>
    {% endblock %}
```

partials/table-no-rows.html

```
        {% extends 'index.html' %}
        {% block table_no_rows}
        <p>No data found.</p>
```

partials/table-rows.html

```
        {% extends 'index.html' %}
        {% block table-rows %}
            {% for row in rows %}
            <tr>
                <td>{{ row.cell1_content }}</td>
                <td>{{ row.cell2_content }}</td>
            </tr>
            {% endfor %}
        {% endblock %}
```

- Frequently used patterns could be extracted out into blocks that generate them. Jinja has macros that can help with this:

```
{% macro input(name, value='', type='text', size=20) -%}
    <input type="{{ type }}" name="{{ name }}" value="{{
        value|e }}" size="{{ size }}">
{%- endmacro %}
```

Usage

```
<p>{{ input('username') }}</p>
```

although we shouldn't necessarily use these - it's a point for discussion. Shane Edwards, another front-end developer, suggests using Pypi modules for including standard components - this would be a longer term task though, and could be a further improvement, after using another method for separation of concerns.

Stylesheets (CSS)

The strategy of the Pattern Library is to extract all components available into the library, rather than only extract components that we expect to be used in more than one product UI. As such, we should extract all the components we currently have in the Response Ops UI into the library, reducing our own use of CSS to only tweaks to existing systems.

This has the benefit that the Pattern Library can be used to shared any component across SDC, but also that all the CSS and JS code will be subject to the linting rules that are part of the build process in the Pattern Library system.

After that, our own CSS will be incredibly minimal. We can add a linter to our build process to match the style linter standards of the Pattern Library linter, meaning the quality of our CSS will be ensured.

If, at any point, we end up with large amounts of CSS, we should consider adding override classes to the Pattern Library, as it's relatively likely that other people are having to override the same type of thing on other products.

In a few places, we use css 'hacks' to help with the graceful degradation of our code, see below:

```
display: inline-block;
*display: inline;
*zoom: 1
```

The above is a hack for very old versions of Internet Explorer being unable to support the css `inline-block` display type. Given our remit to provide very usable and supportive code, we ***probably do*** need it, however, it may be of use to incorporate something like autoprefixer in the process we use to minify our css code.

Javascript

Because of our requirement for systems to work without Javascript, we actually use a very small amount of Javascript. What we do use though, has a few issues that don't meet the standards of our Javascript linter in the Pattern Library:

- All functions and operations occur in global scope, making them have potential side-effects for other included scripts, and for each other. We should consider use of namespacing and closures more often.
- Some functionality could reasonably be extracted into the Pattern Library.
- jQuery is loaded from a remote server that isn't under our control. This raises some issues:
    - Users who block external JS will not benefit from the functionality it adds.
    - A compromised remote server could lead to us using malicious code.
    - An extra request will take place, when we could bundle together our own JS and 3rd party code, helping us to avoid security issues. Whilst the load time isn't much, the extra TTFR will be needless.
    - We'll lose the benefit of the CDN that jQuery loads from (the idea that jQuery is pre-cached in many cases) but we can achieve the benefits of a CDN by using `cdn.ons.gov.uk`, or any CDN system that we can control.
- None of our frontend JS is unit-tested, we should consider the benefits of adding a test suite.
- Our minifier currently only removes whitespace from our code, similar to a GZip pass, when it could perform variable name replacement. We can achieve this by either tweaking the settings of our minifier, or using a different one.

HTML Markup

Our HTML markup also has a few issues:

- In places, we have invalid markup that actually makes the document invalid. This isn't so bad with modern browsers, but older browsers and screen readers may struggle.

- We have a few places where editors and IDEs have clearly introduced errors such as this: `<br></br>`.

> In HTML 1,2,3 this would be `<br>`, and in XHTML4 it would be `<br />`, in HTML5, it's just `<br>` again. This will occur when an IDE blindly closes tags despite their `OMITTAG NO` status in the DOM.

- Our rendering settings within `Jinja2` don't trim out whitespace introduced by our use of Jinja tags tidily, so this:

```
<h1>Header</h1>
{# Jinja comment #}
{% if conditional %}
    <p>This is a paragraph</p>
{% endif %}
```

...will render this:

```
<h1>Header</h1>


    <p>This is a paragraph</p>

```

With all sorts of unpredictable whitespace. We can correct this with `Jinja2` setup changes.

Design standards and adherence

As this is an internal service, we have different requirements to our external services like `ras-frontstage`:

- Internal applications must function on a controlled ONS machine, and run JS in the latest version of Chrome.
- We ***don't*** have to adhere strictly to the GDS, WCAG, and other accessibility requirements, but with these caveats:
    - Components added to the Pattern Library, must be adherent to accessibility standards, ***if*** they are for both internal and external product use.
    - Whilst we don't have the same requirements, a general mindfulness of accessibility is still recommended, because we probably have internal staff who may have to use these systems with accessibility needs - and if we don't we may do in future.

Some notes:

- We have low adherence to WCAG guidelines, which we could usefully improve. Some recommendations:
    - Increase use of aria markup additions, such as roles.
    - Review the tags used, to find opportunities to switch tags for more semantically appropriate version - we have a lot of DIVs that could be a tag more appropriate to their role.
- Tools are available that can perform some static analysis as part of a site, and could incorporated into pipelines as smoke tests. The Government Accessibility Group has recommendations for these - these

may be more cost than benefit in terms of time investment, though, and may lead to either low priority tickets in the backlog, or a decision ***not*** to pursue these avenues.
- Moving more of our code into the Pattern Library is also useful for this, as the Pattern Library is externally verified for accessibility in the case of elements that are also use externally.

Build processes

Our frontend processes, both for build, and for ongoing development currently don't use any of the following:

- A task runner
- ES6+ bundling and transpilation
- ES6+ Modules
- Frontend unit tests (also mentioned in the ***Javascript*** section of this spike)
- SCSS to CSS transpilation *
- CSS minification *
- Real time re-compilation for to allow Front-end developers to view their changes real time.

> - These ***do happen*** but not in the build process - in the initialisation of flask, which means that the build won't catch fatal errors until the build runs the application. We could spot these issues earlier, by transpiling and bundling the assets at an early stage in the process.

Use of a task runner, like Gulp or Grunt, or a more complete bundling tool like Webpack, Rollup, or Parcel could add all the above with relative ease, and the task runners can also run other tasks, like spinning up to docker container, to launch the UI itself, with all development tools available and watching for changes - gives us a lot of flexibility.

The greatest advantage of this is smoother frontend development, with earlier warning of potential issues, and fail-fast builds.

There's an opportunity to discuss which bundler or task runner we use, ***if any***. It's my understanding that previously we used rollup, but we should discuss amongst ourselves what best suits the balance to be struck between frontend and backend developers. An important consideration is not making things easier for frontend development, whilst also introducing unreasonable levels of extra learning and complication for all developers, when some would prefer to have a simpler system. One important possibility is that rather than introducing a task runner or bundler, we **leave the system largely as it is**, but make simple changes to help the build fail fast, and move the majority of frontend functionality to the Pattern Library, separating the concerns, and giving developers the opportunity to either deal with the higher complexity frontend systems, or ***not***.

It should be a priority that improving the frontend doesn't place unreasonable, or business-inefficient demands on developers who need to prioritise working in other areas of the system. Frontend should not become a blocker in efforts to improve it.

Frontend refactoring work already done

Matt Simon, frontend dev on EQ, has already started a new branch in the Response UI repository, has begin work on some of the issues I've raised, meaning we have some scope to begin work, with at least some of this work already done.

Matt's branch in `response-ops` is `upgrade--to-pattern-library`, and he's already made these changes, at least to some extent.

- Some changes to the SCSS config
- Moved some components to Pattern Library already:
    - Breadcrumb
    - Button
    - Header
    - Pills
- Moved font files out to Pattern Library
- Incorporated some of our CSS to the Pattern Library
- Some refactoring in templates - mostly CSS class use

Notably, Matt's work is ***in progress*** and he hasn't been available to work on it for 2 months at the moment. We could usefully use his branch as a base for a frontend overhaul epic, though.

There are still many things we could work on after Matt's branch is completed an included, but it helps towards our goal.

Sum-up

The frontend of response ops is in a generally ok state. We have convoluted code, and some poor frontend practices, but now is a good time to start to address this, preventing build up of tech debt. We can generate and prioritise tickets, and begin work on a frontend epic relatively quickly, and start to see the effects of this soon.

There is some real scope for knowledge sharing, that we could apply in tech sessions or similar, and it would be useful to both improve the frontend and improve the familiarity with the best practices that the Pattern Library is attempting to introduce.

I've made some recommendations in a list below, and they haven't been particularly filtered. They are a mix of things we could/should/must do, and we should apply MoSCoW criteria, or similar in prioritising them - we can be mercenary in our decisions. The team should feel free to challenge any of them on the basis of the net benefit to the whole project.

Recommended tasks

*NB: This is intended as a list of all the things we **could** usefully do. Prioritsation will be necessary, and I expect at least some of the recommendations **not** to end up in changes to the system. An important part of this is making changes to the frontend that improve our code and efficiency - there may be issues I don't know about that prevent some of these happening.*

- Refactor templates to improve use of SOLID principles, particularly DRY
- Extract repository specific UI elements into the Pattern Library
- Adopt linters as smoke tests for our own frontend code
- Frontend JS testing
- Make JS minifier do more aggressive minification
- Tweak Jinja settings
- Create some frontend style guides
- Run some frontend knowledge sharing
- Consider potential tasks for improving our accessibility
- Consider the use or not of build tools, task runners, and bundlers
- Plan and run frontend knowledge sharing sessions