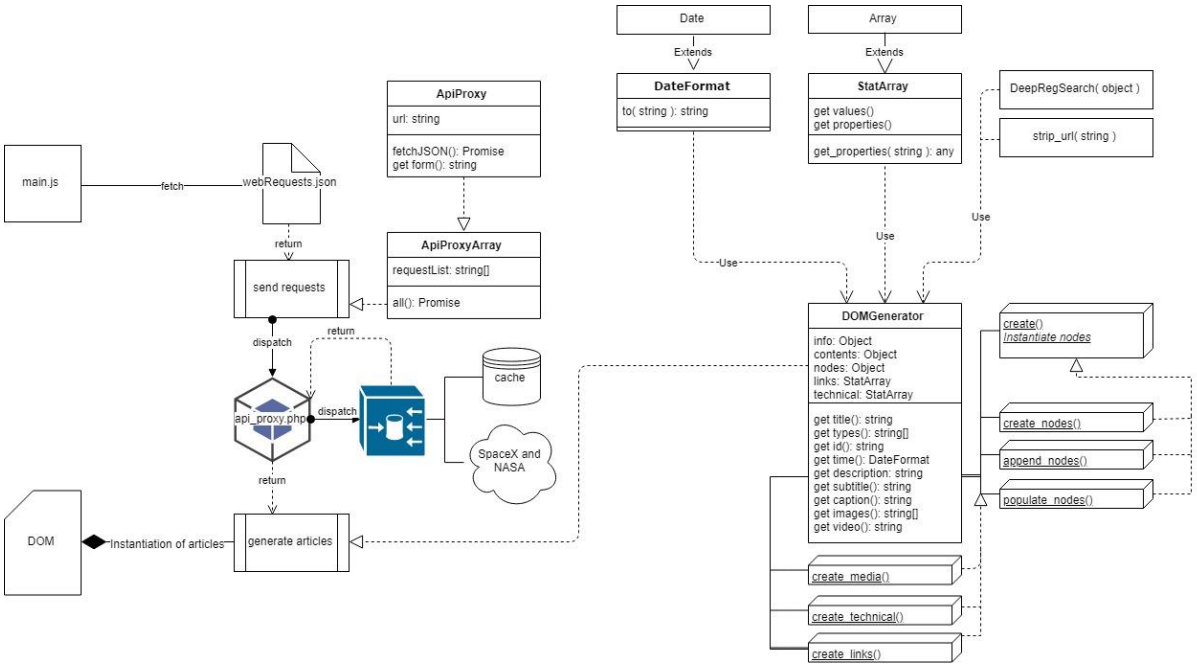
Oblig X

Ask Hetland Berentsen

13/11/2018

Workflow and program structure



Across the whole project, I have done two major refactorings. I intended for the project to be object oriented, where requests were processed within an object. The requests were sent using asynchronous XMLHttp, and for every completed request some action would occur, and one final other action to take place upon all resolved requests. This was done using classes, but the data processing was done using functions taking various segments of the response and handling them in different ways. Initially the API was reached through the browser, upon learning that this is an illegal action, the whole project was refactored to allow php handle the application programming interface, and the data processing.

Allowing the php program to process most of the data was efficient, albeit lacked feedback. The second refactoring was done in order to make almost every aspect of the program object oriented. The php program responsibilities were demoted to only cache and interface with NASA and SpaceX. The two most vital parts of the front-end of the project is the two classes: ApiProxy and DOMGenerator.

The ApiProxy object(s) is responsible for the interfacing with the api_proxy php program on the server. The class ApiProxyArray which implements the ApiProxy object sends an arbitrary amount of requests to the php program, handling the responses asynchronously, using the .all(a, b) method. This method takes two arguments, the first being a callback function which is called on every resolved request, and the second argument being a callback function to be called once all the requests have been resolved.

The DOMGenerator is a class that generates an article based on the input. Once the DOMGenerator has been instantiated with a JSON object of a valid format (including the two properties: info and contents), the object is then ready to use the .create() method. This method will generate an article based on a number of criterias given by the JSON object. The solution was intended to be able to make

several different sections per article, and being able to navigate between them using buttons, but this functionality was revised in the refactored versions. As of the latest build, the DOMGenerator only supports the default section type, but implementation of other section types has been made easily expandable, as the structure is built with this in mind. In order to implement this functionality, the methods:

```
create_table_section   (){}
create_gallery_section (){}
create_stats_section   (){}
```

can be populated.

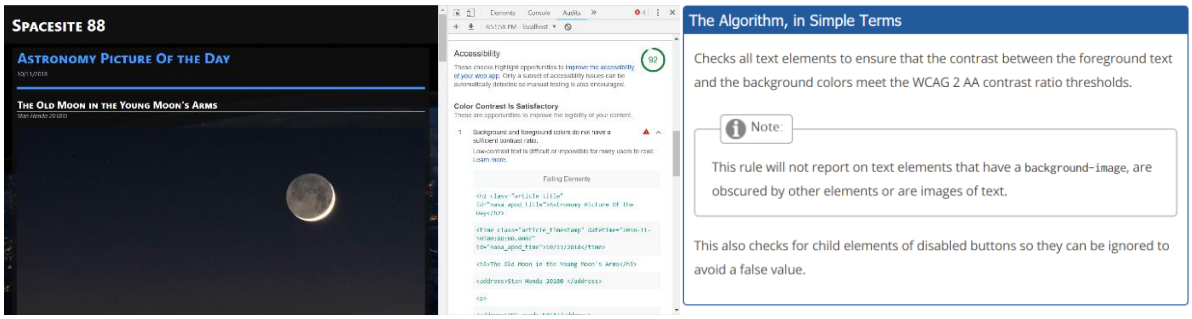
The webRequests.json file that is stored on the server contains all the requests that the program will base its generated articles on. The structure of this object is an array of objects meant for specifying the requests to be sent, each request object contains two first level properties: info and url, which is to be given to the php program. The php program will return a new object with the same info structure, but the url property replaced with the content property, containing the json formatted response from the web-api. Once the javascript has received all the responses, it will start to generate articles.

Over the course of the assignment, a lot of effort was put into the programming part of the solution. Almost no coding was done within the HTML document, with the exception of the header, loading/error message and footer, which aren't dynamic. Almost all content is generated dynamically in javascript (even though php could possibly be better at dynamic content generation, this project was limited to using primarily javascript).

SASS

To style the project I chose to use SASS, this posed a reasonable challenge to get into, (but the fruit of labour proved savoury) and made the CSS workflow more oriented towards programming.

Upon testing the website for accessibility, google lighthouse found issues with the contrast between text and background. However, this test does not take background images into consideration. In addition to having background images, these images are also contained within pseudo-elements, and another background layered on top of the image to contain the text to heighten the contrast, which is also a pseudo-element. A thorough manual check validated that the contrast is sufficient.



Some of the articles support the addition of some details tags, wether they contain links or statistics. To style these tags I chose to make the layout one dimensional on mobile, so that the label is above the entry, but on larger devices the layout will switch to a two dimensional grid. This should not have posed a problem, but as it turns out, the contents of a details tag has some strange behaviour. Setting the display property of the details element to grid has no effect. Wrapping the contents in a div tag (and also hiding the wrapper from the accessibility tree) and set the display property of that to grid instead solves the issue.

The website caches its Web-API responses in a cache folder, and updates every 8th hour (or rarer, depending on the frequency of visits), as such a loadingbar is shown when the server updates its cache. This loadingscreen serves two purposes: If the visitor has Javascript disabled, a text message saying that the user has disabled javascript and therefore the website cannot serve a complete experience; and giving the visitor feedback on the progress of loading the resources. This loading screen will also change style of the cursor upon hovering over the loading-screen to match.

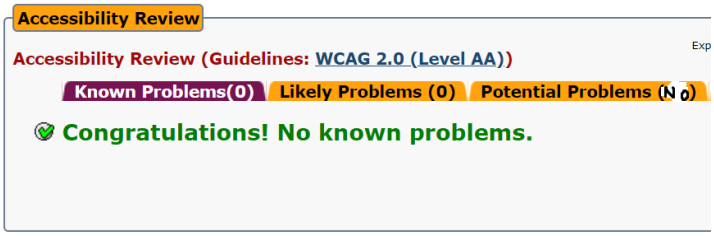
HTML

Since the DOMGenerator creates articles with media such as pictures or videos, the pictures needs an alt text to fulfil the requirements of accessibility. Not all Web-API responses have an applicable alt text that can be extracted to describe the contents of the image, so other another solution had to be implemented. The figure tag within the articles can contain an arbitrary amount of images, and as such the figcaption can describe the collection of images with at least a little precision. The text of the figcaption is generated based on a number of criteria, such as if the Web-API response is of a rocket launch, the figcaption will say something along the lines of “Images of the <rocket name> launch at <launch site>”. The figcaption won’t be entirely accurate all the time, but there isn’t a lot one can do to fix this, so I am content with this solution.

The details tag also presents an issue with accessibility. Since it has a grid layout I need to separate the entries from the labels. Initially this was done by specifying that each label and data-entry was a span tag, but this had no semantic role, so I tried changing the roles of these tags to Row-header and cell. This works well if all the entries are purely asemantic tags, but changing the roles of a tag with an implicit role will result in a bad role validation error. Since the entries can have different tags, some semantic and other asemantic, using the same method to solve both was a difficult problem to solve using roles. I chose instead to change all the span tags to p tags, so that all the entries have an implicit role.

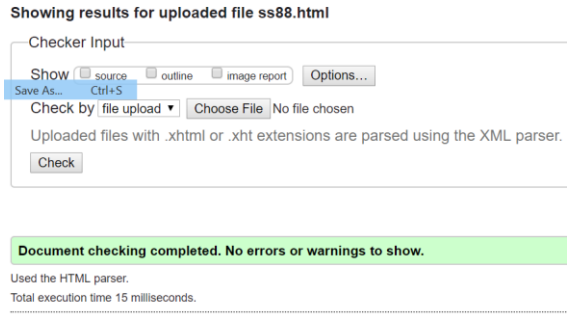
Validations

Achecker:

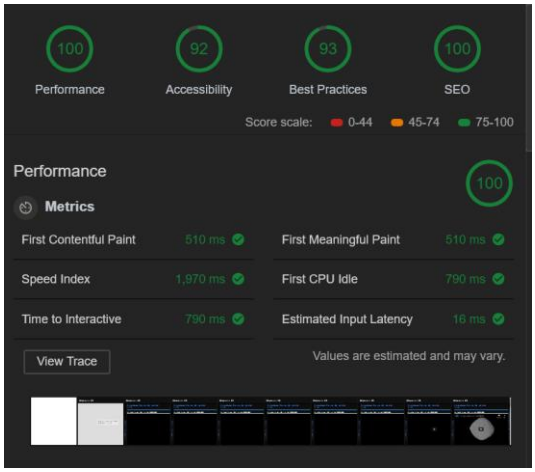


CSS:

HTML:



Google Lighthouse:



Git Repo

https://github.com/Krokkoguy/spaceSite