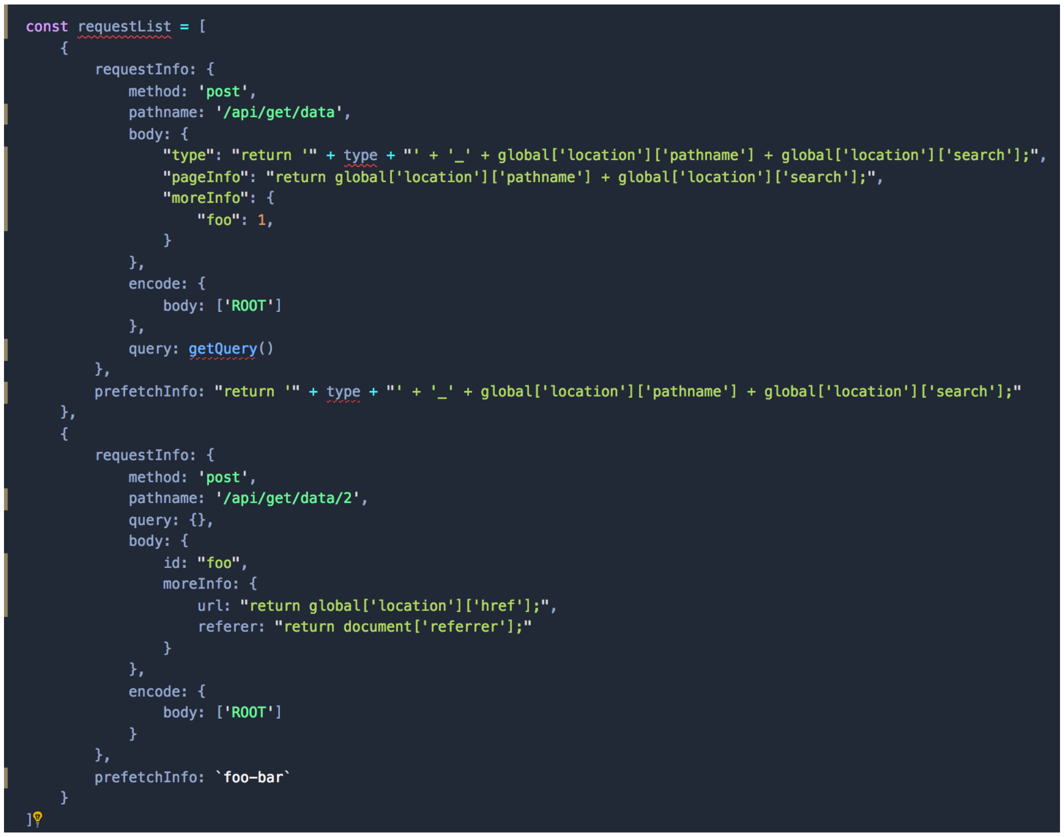
Prefetching is the idea is to write a generic flow that allows any request(s) to be prefetched provided certain criteria is met:

1. During our build process we define the set of request(s) to be prefetched which in turn get into the corresponding “shell” file as a config object.
2. This “config” is an object containing information around what all needs to be prefetched for this page.
3. The “requestList” key in the config object provides an array of <RequestMetaData> request metadata objects.
4. The critical part around generalising the flow is to allow any such metadata to work both on server and more importantly on client. If the metadata is being written on the server (which is mostly the case) but it’s value is dependent on the client e.g. ***window.location.pathname***, then we need to devise a way to allow these values to rehydrate on the client once the server written metadata loads on the client.
    a. The structure of “RequestMetaData” is as follows:



```
const requestList = [
    {
        requestInfo: {
            method: 'post',
            pathname: '/api/get/data',
            body: {
                "type": "return '" + type + "' + '_' + global['location']['pathname'] + global['location']['search'];",
                "pageInfo": "return global['location']['pathname'] + global['location']['search'];",
                "moreInfo": {
                    "foo": 1,
                }
            },
            encode: {
                body: ['ROOT']
            },
            query: getQuery()
        },
        prefetchInfo: "return '" + type + "' + '_' + global['location']['pathname'] + global['location']['search'];"
    },
    {
        requestInfo: {
            method: 'post',
            pathname: '/api/get/data/2',
            query: {},
            body: {
                id: "foo",
                moreInfo: {
                    url: "return global['location']['href'];",
                    referer: "return document['referrer'];"
                }
            },
            encode: {
                body: ['ROOT']
            }
        },
        prefetchInfo: `foo-bar`
    }
]
```

Let’s say, we have to prefetch Product page requests. Now the main requests we need to prefetch in this case are:

- /api/get/data
- /api/get/data/2

So let's take Product page request and analyse as to how do we define the metadata so that it successfully runs both on server as well as hydrates on the client. Each <RequestMetaData> has certain fields:

1. **requestInfo** - metadata for the request
    a. pathname - pathname of the request to be made
    b. method - HTTP verb/method
    c. query - query parameter for the request
    d. body - body in case it's a POST request
    e. encode - an object containing keys, for query and body, which need to be encoded while the request is being made. (More on this later)
2. **prefetchInfo** - prefetch request key. The way prefetching is implemented currently is that, we somehow get the request parameters, make the request and put it in a runtime object against a key. Later when we need this prefetch response (say for a request made by a flux store), we just get the value against this key. Each such flux store needs to send out some metadata in the request to identify if this particular request needs to consult a prefetched request and if so, which key should it use. Once we have both information, we just get the the value against the key (which is a promise) and return to the caller.

    Sequence of steps are:
    a. HTML makes the prefetch request (cache first)
    b. SW returns the response immediately (if present)
    c. SW makes the network request for updated data, caches and notify the main thread through "postMessage"
    d. Main thread reads the updated data

Populating dynamic client values

So now that we've defined the basic parameters of our request object like pathname, http method, let's take a look at how do we pre-define the runtime arguments like the query/body which are client dependent. Continuing on our example, for the product page, we need the <pid, lid, itemId> tuple to be sent in the request. Since these parameters are at runtime and available only in the ***window.location*** object, we somehow need to tell the index.hbs to compute these values and make the request. There are 2 ways of doing this:

1. Define an expression and use "eval" for getting the value e.g. ***eval(window.location.hostname)*** . We already know the damage due to eval, so not an option.
2. We define something called function string literals (technically they are strings which are the ***function implementation***). E.g. if we have a function:

```
function foo(a, b) { return a + b;}
```

Our function string would just become **“*return* a + b;”**. Now we send this function string to the client. But why not send the function itself ?? Note that, we don’t/can’t JSON.stringify functions. And our build does a “JSON.stringify” for this RequestMetaData. Once index.hbs reads this function string, it would somehow have to get a function instance out of it and execute to get the runtime values.
Okay, now all we need is to do is somehow, convert this string back to a function on the client and execute. We do it via:

- Check if the string is a function string (Defined by a ***return*** statement)
- If yes, then we use the “Function” constructor to generate a function instance whose definition is the given function string.
- Execute the function instance

This algorithm recursively runs over all key,value pairs in any metadata object like query/body:

E.g., if we wanted to generate *pid* we would need the URL pathname and query parameters. So, our function string, **s**, could look like:

“return (new URL(global['location']['href']).searchParams.get('pid'));”

And then we would use the “Function” constructor to generate the function instance and execute it. Every function instance by default is passed down 2 variables:

1. *window* - window variable is used as “global” in the string definition
2. *document* - document variable is used as “document” in string definition

foo = Function("global", "document", s);

And executing it as:

value = foo(window, document);

This way we can compute values at runtime while defining them at build time.

Now the final part is the “encode” variable in the “requestInfo” key. Since our query could be objects instead of strings, we would need to stringify and encode them, then send it over. But if we stringify them at build time, then we would need to then JSON.parse everything in “index.hbs” to possibly get back the required object. JSON.parse is expensive, hence we define our objects as they are at build time and let “index.hbs” know which of them should be encoded, stringified. This “encode” key contains an array of such keys.

Function String validation

Since the function definition for needed fields are written as strings, there’s potential for typographical issues or incorrect JS being written. To support this, we now run all such function definitions through *JSDOM* which validates such definitions during build time. If at any moment in time there’s invalid JS being written, build will fail and an error will be thrown. The error has to be fixed for build to finally succeed. For e.g. if there’s a typo in the definitions.