CAB432 Cloud Computing
Some Server Side Processing

In the last prac, we introduced some exercises involving Node, notably a script to find some golden retrievers at Flickr using the Flickr REST web services. We then parsed the resulting JSON and generated a web page to display the thumb nails, each wrapped in a link to the original full size image. In this prac we will shift to the server side. The exercises will mimic those on the client side, but we will instead do all our work within the server, ultimately serving a page similar to the one we saw in Prac 3. However, our approach will show you how to work with a query parameter in the URL. Initially we will work from first principles, using the bare http and https request and response objects provided by node. Later, we will replace this code with a simpler version based on axios (https://github.com/axios/axios). Later we will take an express app and Dockerise it, but we will defer that to another sheet.

Flickr on the Server Side

The Flickr API Exercises. Part II (server side):

This exercise combines Node server exercises and 'client-side' REST calls in the one task. We begin with a hand-constructed app like `app.js`, the one in the lecture last week. We will use a separate `routes` directory and host the Flicker related routing in a separate file. And we will later add in some of the usual middleware that we expect to find. For Assignment 1, you may decide to use Express generator as the route management will become more sophisticated, or you may just decide to extend this app directly.

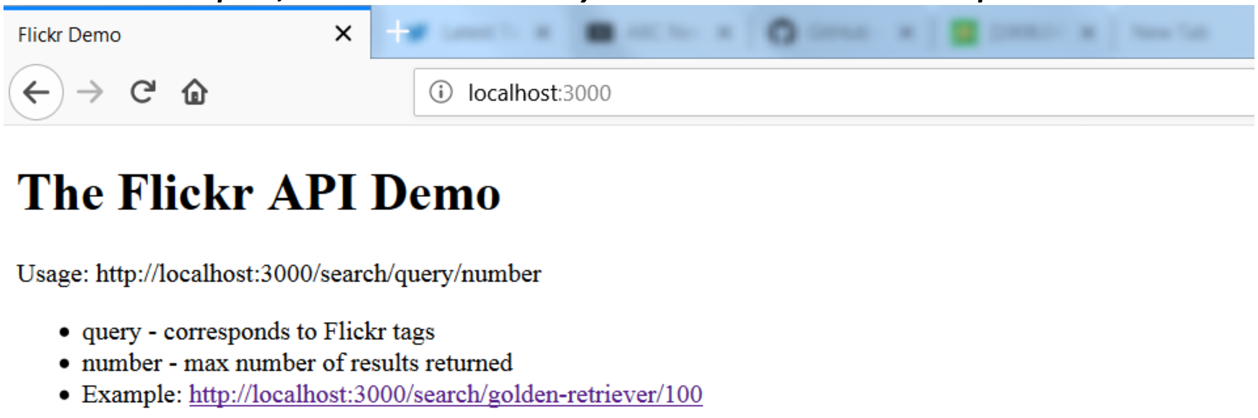
Our starting point is the archive `flickr-starter-files2.zip`. This will create the basic directory structure, with the app in the top directory and the router in the `routes` subdirectory. The files `app.js` and `routes/flickr.js` both contain skeleton code for you to work with. The original simple express server we discussed in lectures is given below.

```
const express = require('express');
const app = express();

const hostname = '127.0.0.1';
const port = 3000;

app.get('/', function (req, res) {
    res.send('Hello World!');
});

app.listen(port, function () {
    console.log(`Express app listening at http://${hostname}:${port}/`);
});
```

Here we support a GET request to the Node server at the root path, corresponding to the request-response pairing in the call back function shown. Other calls will yield a 404 error.

The application we are building will support two request paths:

1. ***The root path, which will show a very basic introduction and example***:

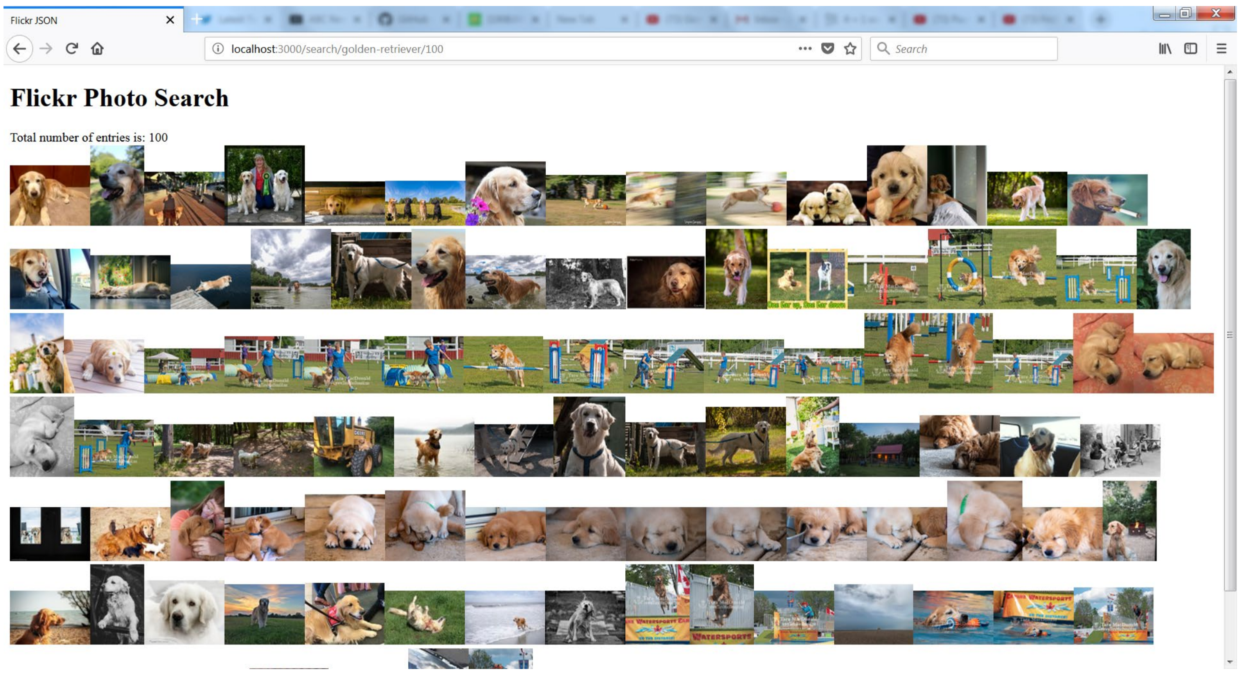


The Flickr API Demo

Usage: http://localhost:3000/search/query/number

- query - corresponds to Flickr tags
- number - max number of results returned
- Example: http://localhost:3000/search/golden-retriever/100

2. ***And the search path, which will query the Flickr account and serve the results as shown:***



Flickr Photo Search

Total number of entries is: 100

Here we will serve the root path from the app and handle all the search facilities in the Flickr router. Let's begin with `app.js`, building on the simple version above, but adding in some key elements to manage the routing. At the top we have the bare bones of an express server. The `app.get` method will consist of a string containing the HTML for the welcome page, which we will deal with later.

```
const express = require('express');
const flickrRouter = require('./routes/flickr');
const app = express();
const hostname = '127.0.0.1';
const port = 3000;
app.get('/', (req, res) => {
    const str =  'XXXXX';
    res.writeHead(200,{'content-type': 'text/html'});
    res.write(str);
    res.end();
});
app.listen(port, function () {
    console.log(`Express app listening at http://${hostname}:${port}/`);
});
```

You should now include the `flickrRouter` import (we don't need the `.js` file extension, but we do need the path) just below the other require statement at the top of the file:

```
const flickrRouter = require('./routes/flickr');
```

We redirect search requests to the external router via this statement, which you should include below the `app.get`:

```
app.use('/search',flickrRouter);
```

We now take a look at the `flickr.js` file and establish a very basic response:

```
const express = require('express');
const router = express.Router();
router.get('/:query/:number', (req, res) => {
            res.write(req.params.query + ":" + req.params.number);
            res.end();
});
module.exports = router;
```

The initial version is incredibly basic, but allows us to test that the route is defined correctly and that the parameters are coming through. Note the use of the `Router` constructor and the export of the `Router` object at the end of the file. Use of `req.params` allows us to capture the query parameters from the URL (the colon in `/:query` tells us that it is a parameter.

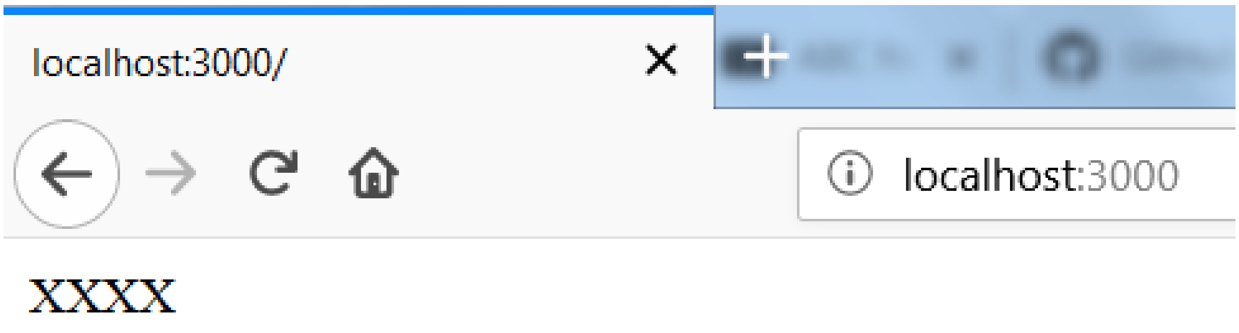
We now return to the top directory of the app and install express locally as usual:

```
$ npm install express
```

We then run the basic app:

```
$ node app
```

When we open a browser at http://localhost:3000 we see:

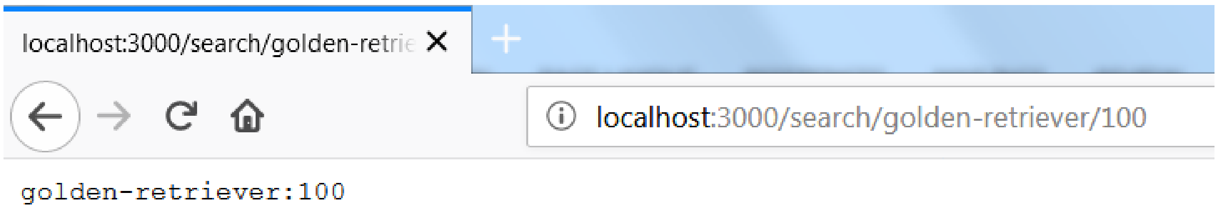


```
XXXX
```

We then try the search query

http://localhost:3000/search/golden-retriever/100

and we see:


localhost:3000/search/golden-retrie
localhost:3000/search/golden-retriever/100

```
golden-retriever:100
```

Always follow a process like this at the start so that you know that your routing is working correctly. The actual responses are more complex, and we will look at these now.

The Root Path

Here the response is a very simple HTML page as shown below:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Flickr Demo</title>
    </head>
    <body>
        <h1>The Flickr API Demo</h1>
        Usage: http://localhost:3000/search/query/number <br>
        <ul><li>query - corresponds to Flickr tags</li>
            <li>number - max number of results returned</li>
            <li>Example: <a href="http://localhost:3000/search/golden-
retriever/100">http://localhost:3000/search/golden-retriever/100</a></li>
        </ul>
    </body>
</html>
```

We will manage this response in two ways. Initially we will create a string which contains all the elements of the file and we will send this back by writing the string. You will need to create a string object based on these elements and use it to replace the XXXX string here:

```
const str =  'XXXX';
```

This fragment will get you started:

```
const str =  '<!DOCTYPE html>' +
        '<html><head><title>Flickr Demo</title></head>' +
```

and we will provide a full answer in Appendix A at the end of the document. We will also provide you with an alternative - loading the page as an html file from the top directory – as seen in Appendix C.

The Flickr Search Path

Our goal here is to provide a similar experience to the one we had last week on the client side, but we will extend the approach to allow specific tag queries and a bound on the number of images to be returned. We will begin by breaking up the URL from last week:

https://api.flickr.com/services/rest/?&method=flickr.photos.search&api_key=XXX&tags=golden-retriever&per-page=50&format=json&media=photos&nojsoncallback=1

We can break this into the host:

api.flickr.com

and the associated path, which includes the method, the api key, the query tags, the per-page limit, the JSON return format and options limiting the results to photos and ensuring that we get bare JSON rather than JSON wrapped in a callback function:

`/services/rest/?&method=flickr.photos.search&api_key=XXX&tags=golden-retriever&per-page=50&format=json&media=photos&nojsoncallback=1`

Each option forms part of an object passed as part of the request to the Flickr API. We begin with a skeleton structure in the `flickr.js` file, where we have included the `https` module at the top, as we will need to create a request to the Flickr API. I have called the relevant objects `flickReq` and `flickRes` to distinguish them from the `req` and `res` that are available from our server. The options object helps specify the request to Flickr. We put all of this together in `createFlickrOptions` further down the file. We explain the callback below:

```
const express = require('express');
const https = require('https');
const router = express.Router();
router.get('/:query/:number', (req, res) => {
    const options = createFlickrOptions(req.params.query,req.params.number);
    const flickReq = https.request(options, (flickRes) => {
      //This is the body of the callback for dealing with the results
      //from the Flickr API. We will assemble the response as it comes in,
      //parse it like we did last week and create the page.
    });
    flickReq.on('error', (e) => {
        console.error(e);
    });
    flickReq.end();
});
```

The `flickr` object just grabs the parameters from the URL we saw above, making it possible for us to make changes conveniently. The `options` object is more standard, supplying the fields needed for a valid request. Note the standard HTTPS port (443) and the method (GET). We will specify the host and an initial base path and add the Flickr options as shown. You will need to enter these in the file and complete the string.

```
const flickr = {
    method: 'flickr.photos.search',
    api_key: "XXXXX",
    format: "json",
    media: "photos",
    nojsoncallback: 1
};
function createFlickrOptions(query,number) {
    const options = {
        hostname: 'api.flickr.com',
        port: 443,
        path: '/services/rest/?',
        method: 'GET'
    }
    const str = 'method=' + flickr.method +
            '&api_key=' + flickr.api_key +
            '&tags=' + query +
            '&per_page=' + number +
            '&format=' + flickr.format +
            '&media=' + flickr.media +
            '&nojsoncallback=' + flickr.nojsoncallback;
    options.path += str;
    return options;
}
function parsePhotoRsp(rsp) {
    // Re-used from last week’s exercise
}
function createPage(title,rsp) {
    // Re-used from last week’s exercise
}
module.exports = router;
```

We will parse the response from Flickr and create the page using the same functions we wrote last week. Once all of these are in place, we must turn our attention to the request to Flickr, and managing the response in the callback we saw above:

```
const flickReq = https.request(options, (flickRes) => {
      //This is the body of the callback for dealing with the results
      //from the Flickr API. We will assemble the response as it comes in,
      //parse it like we did last week and create the page.
});
```

Probably the most important lesson lies in assembling the body of the response that comes from Flickr. We don’t know that the entire request will be satisfied in one chunk coming from the API servers, and so we will capture a chunk with each `'data'` event, and then process the result when we receive the `'end'`. The relevant code is on the next page:

```
const flickReq = https.request(options, (flickRes) => {
      let body = [];
      flickRes.on('data',function(chunk) {
          body.push(chunk);
      });
```

In the code above we create an array buffer to hold the chunks as they come back, and an event handler for the `'data'` events that facilitate this assembly. The relevant JS references are here:

- http://www.w3schools.com/js/js_arrays.asp
- http://www.w3schools.com/jsref/jsref_push.asp

Now when we receive the `'end'` message from the Flickr servers, we can create the response. First, we write the headers – at present I haven't bothered catching anything unusual, we just pass the `statusCode` and indicate that the result will be `text/html`. Initially it will come as stringified JSON and we will use `JSON.parse()` to get the object. We then parse the JSON response `rsp` and create the page as we did last week. Make sure that you understand what is going on – we are processing the response from the Flickr server (`flickRes`) but the `res.write` and `res.end` calls involve the response from *our* server.

```
      flickRes.on('end', function() {
          res.writeHead(flickRes.statusCode,{'content-type': 'text/html'});
          const bodyString = body.join('');
          const rsp = JSON.parse(bodyString);
          const s = createPage('Flickr Photo Search',rsp);
          res.write(s);
          res.end();
      });
});
```

Note the use of the `join` method to concatenate the elements of the array. You can use a `toString` method here if you wish, but later on, it will cause some difficulties as we try to parse the body using the JSON parser as `toString` inserts a comma between each entry. The join method allows us to specify the separator. We do this by entering an empty string so that we avoid syntactic clashes.

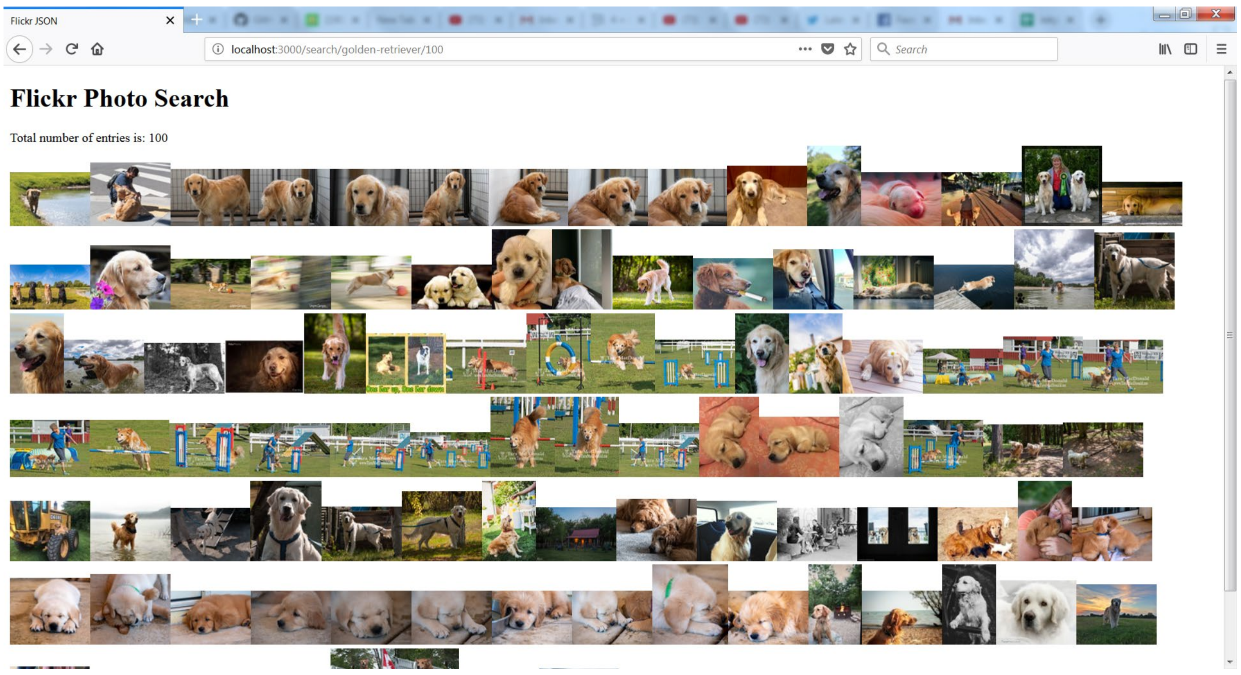
The methods discussed are here:

- http://www.w3schools.com/jsref/jsref_tostring_array.asp
- http://www.w3schools.com/jsref/jsref_join.asp

The `createPage` function calls the `parsePhotoRsp` function and between them they do the same work as last week. Our final step is to include a handler for the error event – which we keep simple at this stage – and to send the actual request to the Flickr server.

```
router.get('/:query/:number', (req, res) => {
    //Request code not shown
    flickReq.on('error', (e) => {
        console.error(e);
    });
    flickReq.end();
});
```

A full listing is provided in the appendix. When we run the completed version of this code, the result is similar to the client side:

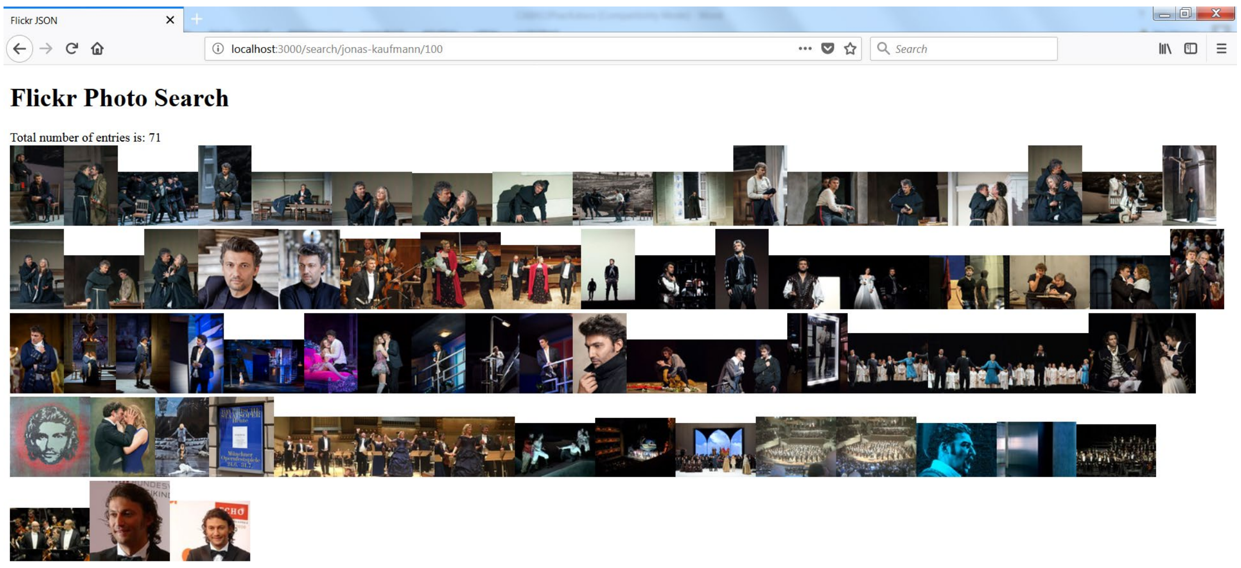


Flickr Photo Search

Total number of entries is: 100

We might even have interests that extend beyond golden retrievers. The approach is exactly the same. We will use two other tags and use the same limit of 100 for each call. The first call looks for images of the great operatic tenor Jonas Kaufmann (https://en.wikipedia.org/wiki/Jonas_Kaufmann). The other looks for images of London.

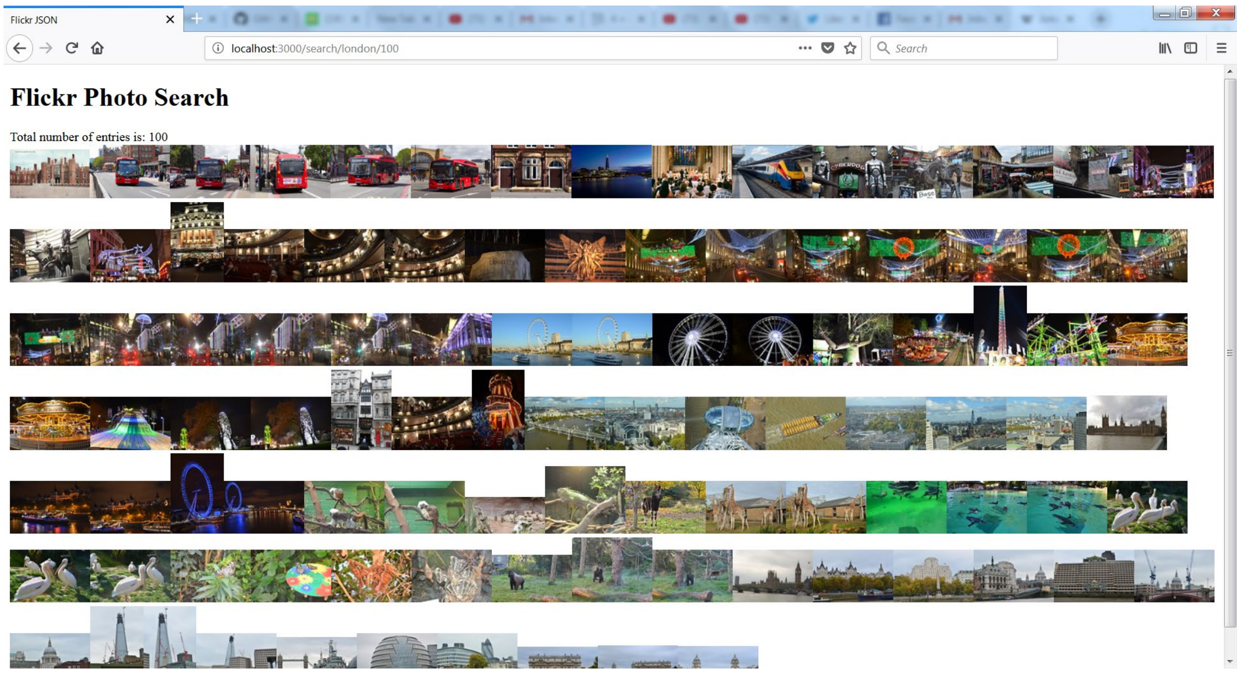
http://localhost:3000/search/jonas-kaufmann/100



Flickr Photo Search

Total number of entries is: 71

http://localhost:3000/search/london/100



Flickr Photo Search

Total number of entries is: 100

Some Middleware

More technically, we will now add some simple middleware to our application. In an earlier version of this worksheet we used the security middleware `helmet` on all requests as a kind of hygiene measure. However, a recent release of this software prevents the Flickr requests from operating without more careful configuration, so we will leave this until later in the unit. For now, we will limit ourselves to logging using `morgan` on the router `flickr.js`.

General middleware – applied to all requests – would normally be inserted in the main application, with an `app.use(mymiddleware())` statement appearing just after the declaration of the app via `const app = express();` as shown below in the dummy example:

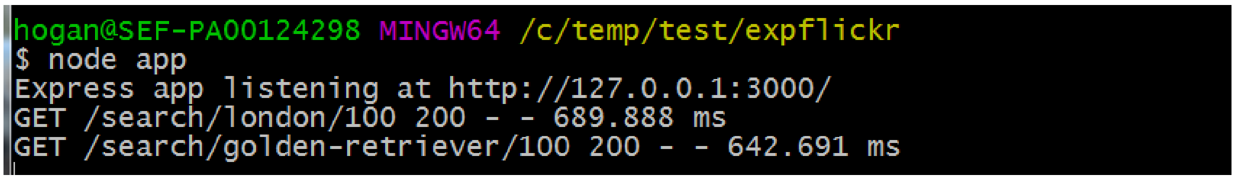
```
const express = require('express');
const flickrRouter = require('./routes/flickr');
const mymiddleware = require('mymiddleware');
const app = express();
app.use(mymiddleware());
const hostname = '127.0.0.1';
const port = 3000;
app.get('/', (req, res) => {
```

Here `mymiddleware` would be bound to the app and used for all requests. In contrast, we bind the `morgan` middleware only to the router, so that there is no logging from the root – just the Flickr search path. Here we use the predefined format 'tiny' which gives the minimal log output (see https://www.npmjs.com/package/morgan#tiny for details).

```
const express = require('express');
const https = require('https');
const logger = require('morgan');
const router = express.Router();
router.use(logger('tiny'));
router.get('/:query/:number', (req, res) => {
```

Having made these edits, we save the files and install the new packages. The screenshot below shows the logs from three calls:

- http://localhost:3000/search/london/100
- http://localhost:3000/
- http://localhost:3000/search/golden-retriever/100



```
hogan@SEF-PA00124298 MINGW64 /c/temp/test/expflickr
$ node app
Express app listening at http://127.0.0.1:3000/
GET /search/london/100 200 - - 689.888 ms
GET /search/golden-retriever/100 200 - - 642.691 ms
```

As is clear from this output, logging does not include paths handled by the app.

In the appendices which follow we will generalize the URL path query to the more common and extensible `key=value` style query strings. This requires that we parse the URL directly in order to store the parameters and their associated values. In the final section before the appendices, we will use the axios library to handle requests more conveniently.

Using Axios

The axios library (https://github.com/axios/axios) greatly simplifies the task of performing HTTP requests within node and on the client side. The effect here is to get rid of most of the complexity in the `https.createRequest` call you saw above, and in the end the code doesn't look very different from the `fetch()` calls we used last week. And when we say that the code is simplified, we really mean it. The approach is once more based on promises, and the style once more relies on the chaining of `.then` clauses.

In the earlier code we created an options object to handle many of the parameters and headers associated with the call. Here we will use this existing object because it is there, but in most cases the axios defaults will be sufficient.

Many of the usual fields are available on the axios response object, but here we will be concerned only with:

- `status` – which contains the status code relating to the response.
- `data` – which contains the 'payload', usually in JSON form.

The 27 lines of code below replace the first 36 lines of the listing in Appendix B, and are a great deal simpler. Note the use of `response.status` to get the code in the first clause as `rsp` is passed `response.data`, the JSON object:

```
const express = require('express');
const axios = require('axios');
const logger = require('morgan');
const router = express.Router();
router.use(logger('tiny'));
router.get('/:query/:number', (req, res) => {
    const options = createFlickrOptions(req.params.query,req.params.number);
    const url = `https://${options.hostname}${options.path}`;
    axios.get(url)
        .then( (response) =>  {
            res.writeHead(response.status,{'content-type': 'text/html'});
            return response.data;
        })
        .then( (rsp) => {
            const s = createPage('Flickr Photo Search',rsp);
            res.write(s);
            res.end();
        })
        .catch((error) => {
            console.error(error);
        })
});
```

Note the template literal used in the construction of the URL. See Appendix B for details.

Appendix A: Full Listing `app.js`

```
const express = require('express');
const flickrRouter = require('./routes/flickr');
const app = express();
const hostname = '127.0.0.1';
const port = 3000;
app.get('/', (req, res) => {
    const str =  '<!DOCTYPE html>' +
        '<html><head><title>Flickr Demo</title></head>' +
        '<body>' +
            '<h1>' + 'The Flickr API Demo' + '</h1>' +
            'Usage: http://localhost:3000/search/query/number <br>' +
            '<ul>' + '<li>query - corresponds to Flickr tags</li>'
                   + '<li>number - max number of results returned</li>'
                   + '<li>Example: <a href="http://localhost:3000/search/golden-
retriever/100">http://localhost:3000/search/golden-retriever/100</a></li>' +
            '</ul>' +
        '</body></html>';
    res.writeHead(200,{'content-type': 'text/html'});
    res.write(str);
    res.end();
});
app.use('/search',flickrRouter);
app.listen(port, function () {
    console.log(`Express app listening at http://${hostname}:${port}/`);
});
```

Appendix B: Full Listing `flickr.js`

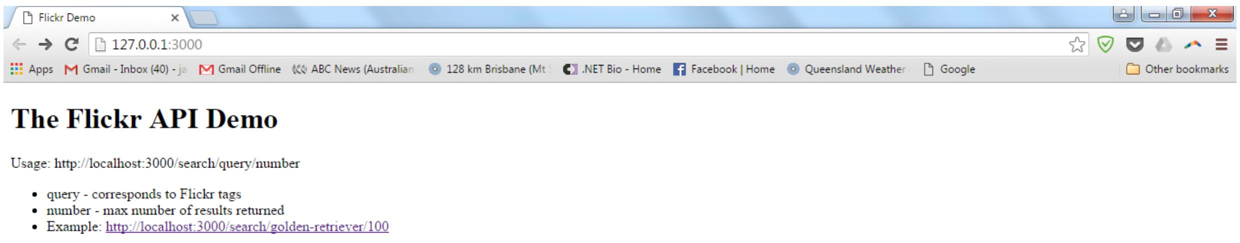
```
const express = require('express');
const https = require('https');
const router = express.Router();
router.get('/:query/:number', (req, res) => {
    const options = createFlickrOptions(req.params.query,req.params.number);
    const flickReq = https.request(options, (flickRes) => {
            let body = [];
            flickRes.on('data',function(chunk) {
                body.push(chunk);
            });
            flickRes.on('end', function() {
                res.writeHead(flickRes.statusCode,{'content-type':
'text/html'});
                const bodyString = body.join('');
                const rsp = JSON.parse(bodyString);
                const s = createPage('Flickr Photo Search',rsp);
                res.write(s);
                res.end();
            });
    });
    flickReq.on('error', (e) => {
        console.error(e);
    });
    flickReq.end();
});
const flickr = {
    method: 'flickr.photos.search',
    api_key: "XXXXX",
    format: "json",
    media: "photos",
    nojsoncallback: 1
};
function createFlickrOptions(query,number) {
    const options = {
        hostname: 'api.flickr.com',
        port: 443,
        path: '/services/rest/?',
        method: 'GET'
    }
    const str = 'method=' + flickr.method +
            '&api_key=' + flickr.api_key +
            '&tags=' + query +
            '&per_page=' + number +
            '&format=' + flickr.format +
            '&media=' + flickr.media +
            '&nojsoncallback=' + flickr.nojsoncallback;
    options.path += str;
    return options;
}
```

```
//Various font sizes used to fit URL on screen
function parsePhotoRsp(rsp) {
    let s = "";
    for (let i = 0; i < rsp.photos.photo.length; i++) {
        photo = rsp.photos.photo[i];
  t_url = `https://farm${photo.farm}.staticflickr.com/${photo.server}/${photo.id}_${photo.secret}_t.jpg`;
          p_url = `https://www.flickr.com/photos/${photo.owner}/${photo.id}`;
      s += `<a href="${p_url}"><img alt="${photo.title}" src="${t_url}"/></a>`;
    }
    return s;
}
function createPage(title,rsp) {
    const number = rsp.photos.photo.length;
    const imageString = parsePhotoRsp(rsp);
    //Headers and opening body, then main content and close
    const str = '<!DOCTYPE html>' +
        '<html><head><title>Flickr JSON</title></head>' +
        '<body>' +
        '<h1>' + title + '</h1>' +
        'Total number of entries is: ' + number + '</br>' +
        imageString +
        '</body></html>';
    return str;
}
module.exports = router;
```

Note that this version of `parsePhotoRsp` uses template literals. We may embed an expression in a string and have it evaluated – note the backward quotes - `  ` - and replaced by its value. See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Appendix C: Using an HTML home page.

Earlier we served the basic HTML page at the root of the site using a string which contained the necessary HTML elements:



The Flickr API Demo

Usage: http://localhost:3000/search/query/number

- query - corresponds to Flickr tags
- number - max number of results returned
- Example: http://localhost:3000/search/golden-retriever/100

A more professional response would be to serve a static web page of your choice from a file called `index.html`. We can use the earlier file server example from the lectures to make this happen. The adapted code is as follows. Note the use of the `fs` or filesystem module.

```
const express = require('express');
const fs = require('fs');
const flickrRouter = require('./routes/flickr');
const app = express();
const hostname = '127.0.0.1';
const port = 3000;
app.get('/', (req, res) => {
    res.writeHead(200,{'content-type': 'text/html'});
    fs.readFile('index.html', 'utf8', (err, data) => {
        if (err) {
            res.end('Could not find or open file for reading\n');
        } else {
            res.end(data);
        }
    });
});
app.use('/search',flickrRouter);
app.listen(port, function () {
    console.log(`Express app listening at http://${hostname}:${port}/`);
});
```

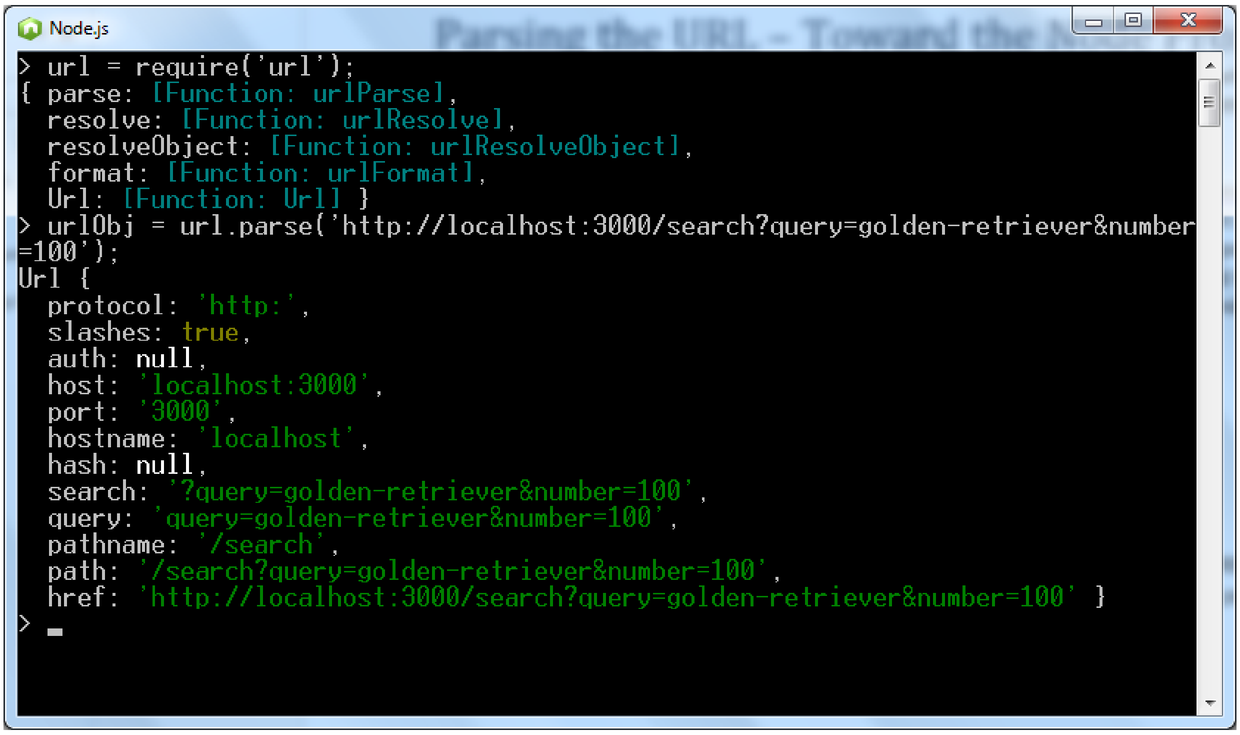
Appendix D: Parsing URL Query Strings.

This section examines URLs and how to parse them within Node, allowing us to work with more general queries of the form:

http://localhost:3000/search?query=golden-retriever&number=100

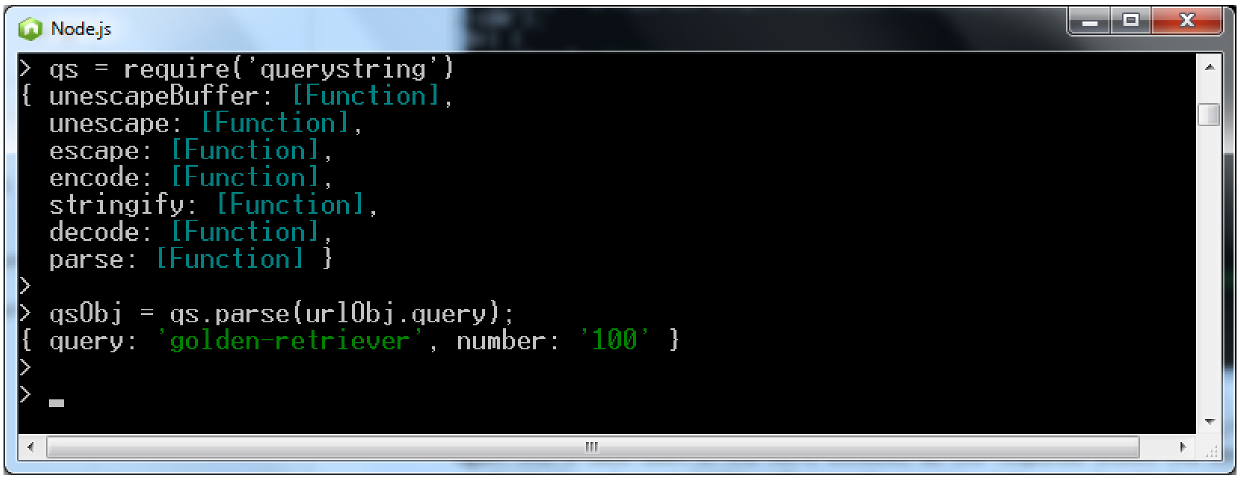
This form allows us to work with an arbitrary number of `key=value` parameters listed in any order. Working with the parsing examples may be easier in the Node interpreter. You can use `console.log()` if you want nicer output, but the important point is that you don't have to restart after each change and we will be working through a series of steps that can then be incorporated into our final system.

As before, the server code handles the creation of the Flickr REST call, but we need to extract the query string and parse the parameters. The methods we need are available in the `url` module. In the REPL session below, we grab the `url` module and then use the parse method on the URL above. The available fields are helpfully returned for us. We want to access the `query` field – yes, query gets used in multiple contexts here, but I have chosen not to hide this as that confusion is very common.



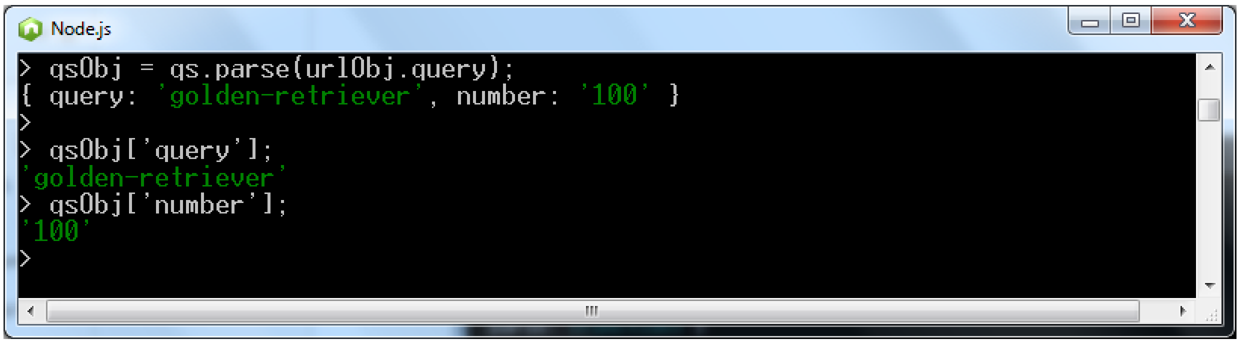
```
> url = require('url');
{ parse: [Function: urlParse],
  resolve: [Function: urlResolve],
  resolveObject: [Function: urlResolveObject],
  format: [Function: urlFormat],
  Url: [Function: Url] }
> urlObj = url.parse('http://localhost:3000/search?query=golden-retriever&number
=100');
Url {
  protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'localhost:3000',
  port: '3000',
  hostname: 'localhost',
  hash: null,
  search: '?query=golden-retriever&number=100',
  query: 'query=golden-retriever&number=100',
  pathname: '/search',
  path: '/search?query=golden-retriever&number=100',
  href: 'http://localhost:3000/search?query=golden-retriever&number=100' }
> 
```

Our next step is to parse the query field using the very helpful `querystring` module. Once again this is best seen in the REPL window, as the response shows the structure of the methods on each module. We will parse the query and then work with the resulting JSON to create a fresh query for the REST API:



```
> qs = require('querystring')
{ unescapeBuffer: [Function],
  unescape: [Function],
  escape: [Function],
  encode: [Function],
  stringify: [Function],
  decode: [Function],
  parse: [Function] }
>
> qsObj = qs.parse(urlObj.query);
{ query: 'golden-retriever', number: '100' }
>
> _
```

The properties of a JSON object are readily available through associative indexing:



```
> qsObj = qs.parse(urlObj.query);
{ query: 'golden-retriever', number: '100' }
>
> qsObj['query'];
'golden-retriever'
> qsObj['number'];
'100'
>
```

We now return to the example application. We will work with the original version and you can make the appropriate adjustments for axios. The path to the router remains the same:

```
app.use('/search',flickrRouter);
```

But we will add a separate path in the router itself, maintaining both approaches, the new:

```
router.get('/full', (req, res) => {
```

which will handle URLs of the form:

http://localhost:3000/search/full?query=golden-retriever&number=100

and the existing version:

```
router.get('/:query/:number', (req, res) => {
```

which handles path parameters as before:

http://localhost:3000/search/golden-retriever/100 .

Within the router, we handle all queries in the same way – at present we still only support the two parameters. Express uses some of the logic we saw above to give us the URL, the path and the query as separate objects. We can see these easily by logging:

```
console.log(req.url);
console.log(req.path);
console.log(req.query);
```

When we restart the server and point a browser at the endpoint with the URL:

http://localhost:3000/search/full?query=golden-retriever&number=100

we see the following logging information:

```
/full?query=golden-retriever&number=100
/full
{ query: 'golden-retriever', number: '100' }
```

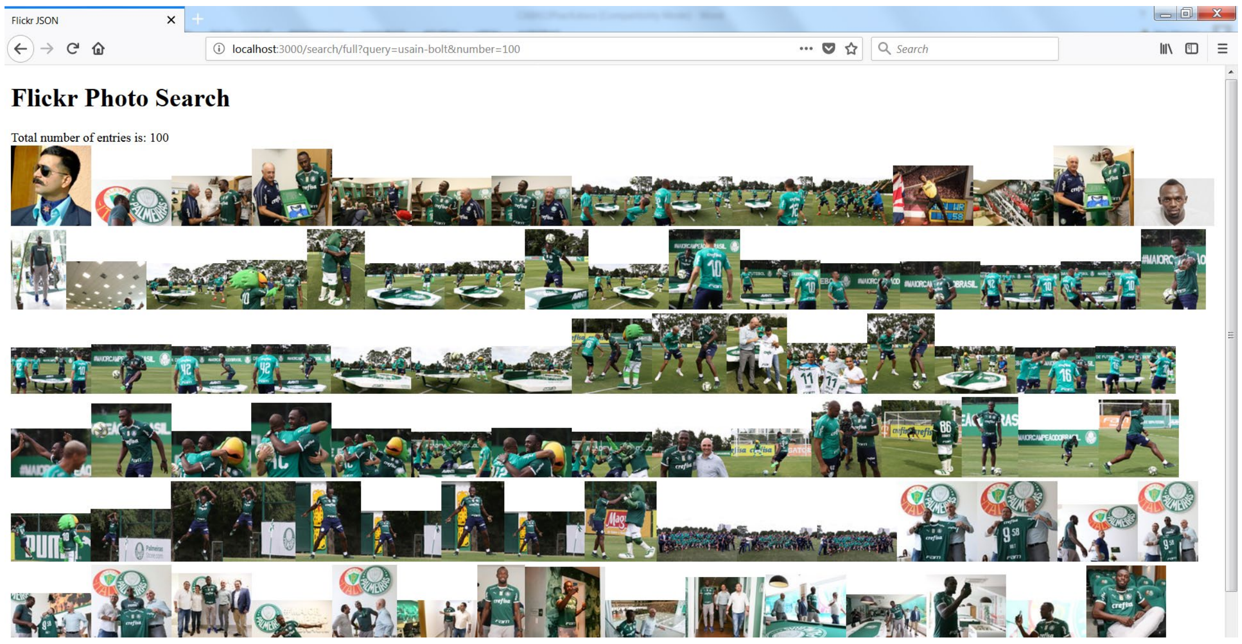
Thus, we can directly extract the information we need from this object. Given the way we have structured our earlier code, the only change necessary is to the call to `createFlickrOptions`:

```
const query = req.query;
const options = createFlickrOptions(query['query'],query['number']);
```

There is some duplicated code as the callback makes it awkward to refactor cleanly, but it works. Quickly confirm that it loads the golden retrievers correctly as before. Now, go one further, and find some guy called Usain Bolt with the following call:

http://localhost:3000/search/full?query=usain-bolt&number=100

Apparently in 2019 he was more of a footballer than the greatest sprinter in the world:



Flickr Photo Search

Total number of entries is: 100

Some of the tagging is evidently weird, but the system works. There are two final things before we finish. First, the query with `param_name=param_value` is essential when we have numerous options. When the call is more restricted, I have a preference for APIs that don't include the parameter name explicitly in the URL. Here we have managed to do both.