CAB230 Web Computing Assignment

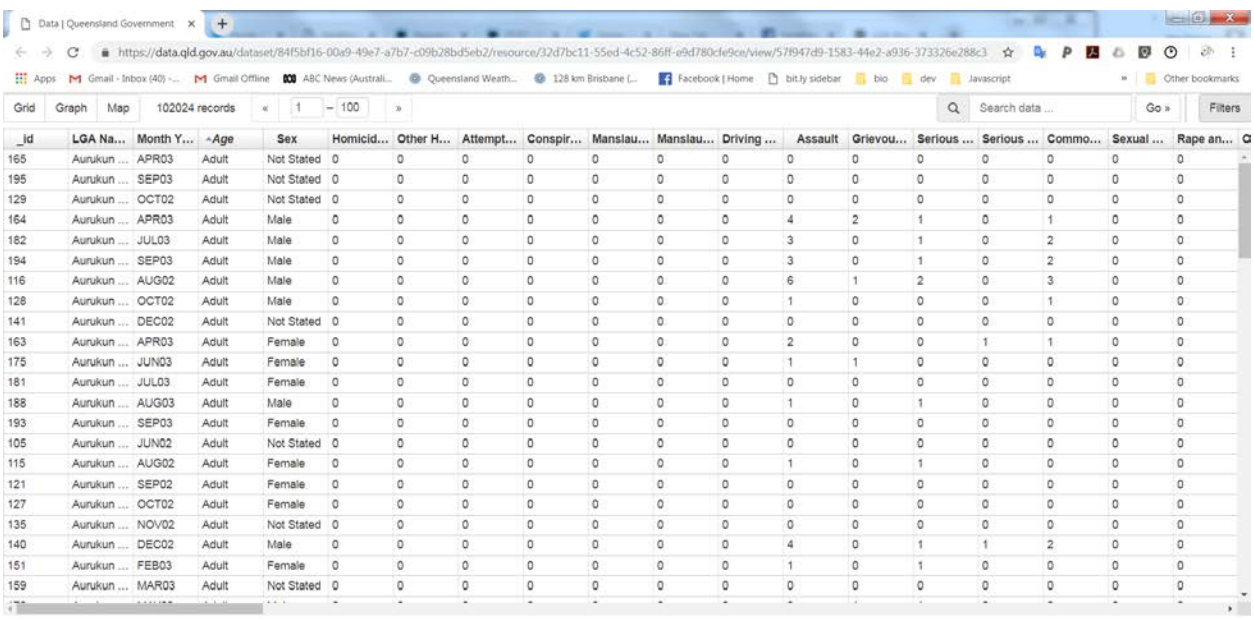
Getting Started with the API

This document presents a short guide to help you get started with the API you will be using over the course of the assignment. Initially, you will work with this site purely as a data source. In what follows we will work through the process of hitting this API using `fetch` based JavaScript calls as we did in the lecture and the prac for week 5.

As discussed in the lectures, we will be working with a dataset based on the following crime statistics made available by the state government:

https://data.qld.gov.au/dataset/lga_reported_offender_numbers/resource/32d7bc11-55ed-4c52-86ff-e9d780cfe9ce

As you can see if you select the visualisation tab, there are a massive number of columns listed, each reflecting an offence or range of offences covered by the criminal justice system.



_id	LGA Na...	Month Y...	Age	Sex	Homicid...	Other H...	Attempt...	Conspir...	Manslau...	Manslau...	Driving ...	Assault	Grievou...	Serious ...	Serious ...	Commo...	Sexual ...	Rape an...
165	Aurukun ...	APR03	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0
195	Aurukun ...	SEP03	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0
129	Aurukun ...	OCT02	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0
164	Aurukun ...	APR03	Adult	Male	0	0	0	0	0	0	0	4	2	1	0	1	0	0
182	Aurukun ...	JUL03	Adult	Male	0	0	0	0	0	0	0	3	0	1	0	2	0	0
194	Aurukun ...	SEP03	Adult	Male	0	0	0	0	0	0	0	3	0	1	0	2	0	0
116	Aurukun ...	AUG02	Adult	Male	0	0	0	0	0	0	0	6	1	2	0	3	0	0
128	Aurukun ...	OCT02	Adult	Male	0	0	0	0	0	0	0	1	0	0	0	1	0	0
141	Aurukun ...	DEC02	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0
163	Aurukun ...	APR03	Adult	Female	0	0	0	0	0	0	0	2	0	0	1	1	0	0
175	Aurukun ...	JUN03	Adult	Female	0	0	0	0	0	0	0	1	1	0	0	0	0	0
181	Aurukun ...	JUL03	Adult	Female	0	0	0	0	0	0	0	0	0	0	0	0	0	0
188	Aurukun ...	AUG03	Adult	Male	0	0	0	0	0	0	0	1	0	1	0	0	0	0
193	Aurukun ...	SEP03	Adult	Female	0	0	0	0	0	0	0	0	0	0	0	0	0	0
105	Aurukun ...	JUN02	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0
115	Aurukun ...	AUG02	Adult	Female	0	0	0	0	0	0	0	1	0	1	0	0	0	0
121	Aurukun ...	SEP02	Adult	Female	0	0	0	0	0	0	0	0	0	0	0	0	0	0
127	Aurukun ...	OCT02	Adult	Female	0	0	0	0	0	0	0	0	0	0	0	0	0	0
135	Aurukun ...	NOV02	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0
140	Aurukun ...	DEC02	Adult	Male	0	0	0	0	0	0	0	4	0	1	1	2	0	0
151	Aurukun ...	FEB03	Adult	Female	0	0	0	0	0	0	0	1	0	1	0	0	0	0
159	Aurukun ...	MAR03	Adult	Not Stated	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The site is quite sophisticated, and allows the user to sort and filter based on a range of data source selections and criteria. There is also the option to graph particular fields or to show the incidence of offences by geographical region on a map. If you want a grade of 6 or 7 standard mark for the client side of this assignment, then you will have to do the same. Start using this site to explore your ideas.

Over the past couple of weeks Tylor has taken this CSV file and ‘wrangled’ the data into a form which is more suitable for a small, well-defined API. There are a range of endpoints available to you, and these are listed briefly, and without a lot of detail, in the bullet points below:

- `/register (POST) – Register using email and password`
- `/login (POST) – Logging into your account using the details provided. This allows access to authenticated routes via a JSON Web Token.`
- `/offences (GET) – returns an array of offences recorded. (Open endpoint).`
- `/search?offence=xxx (GET) – The primary search route. Note that query params need to be url encoded. (Requires authentication).`

These main endpoints will not change over the course of the assignment, but in the coming days we will add some additional filtering options for the search on `offences`.

The goal of this document is not to provide you with a comprehensive API reference, but to get you started in using the API to the point that you can register, and login and do some very basic queries. The material will make very little sense if you have not seen the week 5 lecture or completed most of the week 5 prac. We are here to hit a public API with a series of POST and GET calls. We will be using the modern `fetch()` approach introduced in the lecture, and there will not be a single `XMLHttpRequest` in sight.

Our approach will be to build on the earlier prac work, defining buttons for each of the tasks we want to manage. The account registration should be executed only once for each email address. Please just use your QUT email and register once and once only.

In what follows we are going to use the `innerHTML` on the page only to show the good bits. For the error messages, you will need to open up the console in the page as you did in the debugging exercises in the prac.

The code is available in the `starter-files.zip` archive as usual. These are very basic calls and are intended solely to get you started. In particular, I am not doing any of the form processing for you at the moment. But you should be able to use this code as a basis for the first stages of the assignment.

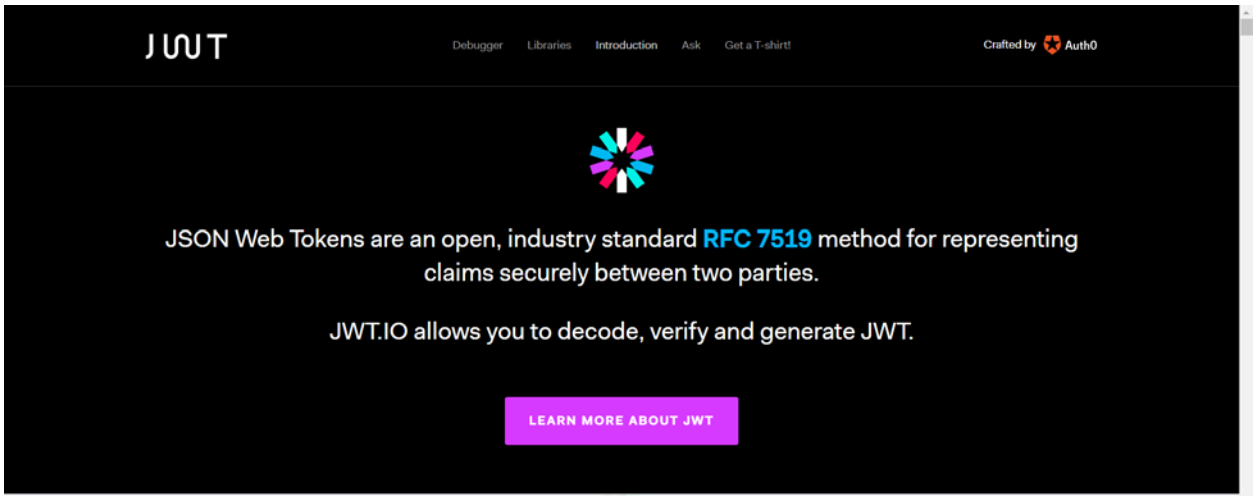
The HTML page is very simple, as shown on the next page. We begin with a cute heading at the top, we include a small set of buttons, and then follow with a single div which we call `app`. Each button affects that div. We do not distinguish between them and we do not maintain the result of any previous calls. We will, however, need to keep track of a token, and that is something we will consider in a moment.

```html
<!DOCTYPE html>
<html>

<head>
   <title>API Starter Code</title>
   <meta charset="UTF-8" />
</head>

<body>
   <h1>Getting Started with Crime</h1>
   <button id="regBtn">Register</button>
   <button id="logBtn">Login</button>
   <button id="offBtn">Offences</button>
   <button id="serBtn">Search</button>

   <div id="app"></div>

   <script src="src/index.js">
   </script>
</body>

</html>
```

Registration and Login:

Registration is the first step in working with the API. If the database doesn't know you, then you can't perform a login to the system. If you can't login, then you can't acquire an access token. If you can't acquire an access token, then you can't really do all that much.

The API site here doesn't formally maintain a session. The way this works is that registered users may login to the site using their previously supplied credentials. On login, the system will return a JSON Web Token (see https://jwt.io/ for details):



This JWT provides you with authorisation to use the API. You do not need the JWT to access the `/offences` route as it is an open endpoint. However, if you want to access the interesting data using `/search`, then you need to provide this token in the request header. This is both dead simple and surprisingly tricky, in the sense that it is easy to make dumb errors that will cost you a lot of time. We will walk you through this below.

But first, we need to register. In practice, and in your assignment, we will expect that registration and login will be handled through a form. People will enter their details – we require an email and a password – and you will need to grab these data from the text boxes and then send them to the server using a POST request. The code below is essentially that POST request, only here we have provided the examples in their urelencoded form:

```
const regButton = document.getElementById("regBtn");
regButton.addEventListener("click", () => {

  fetch("http://hackhouse.sh:3000/register", {
    method: "POST",
    body: 'email=woof%40dog.com&password=WalksAndFoodForever',
    headers: {
      "Content-type": "application/x-www-form-urlencoded"
    }
  })
    .then(function(response) {
      if (response.ok) {
        return response.json();
      }
      throw new Error("Network response was not ok");
    })
    .then(function(result) {
      let appDiv = document.getElementById("app");
      appDiv.innerHTML = JSON.stringify(result);
      regButton.disabled = true;
    })
    .catch(function(error) {
      console.log("Problem with fetch ",error.message);
    });
});
```

Much of this code is familiar to you, but some is not. The fetch request is a POST rather than a GET and so we must include a second argument to the function containing the parameters of the request. So the brace after the URL is the opening brace of a JSON object which contains the `method`, the `body`, and the `headers` for the request. The

headers object here contains a single content type header which indicates that the body contains urlencoded data – this is typical of text grabbed from a form.

You need to focus on the body and directly edit it to show the email address and password you want to use. The `%40` is the '@' character from the email address. In the file I have supplied for your use, the email address and password are edited down to placeholder characters. You will need to fix them up for yourselves.

```
fetch("http://hackhouse.sh:3000/register", {
  method: "POST",
  body: 'email=woof%40dog.com&password=WalksAndFoodForever',
  headers: {
    "Content-type": "application/x-www-form-urlencoded"
  }
})
```

The remaining code is boilerplate and very similar to that which you saw in the lectures. Open the index.js file in your editor and update the email and password to the values you want to use. Scroll down in the file to the next call, which hits the `/login` route. Again update the body, making sure that the email and password here are exactly the same as those you use in the registration request. Save the file and reload the html page in the browser. You should see something like the following:

When you hit the registration button, you will get a simple stringified JSON response telling you that the registration has been successful. The Register button is then disabled, at least until the next time you reload the page.

If you do not get a success message, there will be something wrong with the email address or password that you have used. You can find out more by inspecting the page, opening up the console, and interrogating the response object.

In the normal course of events, we now login. This doesn't happen on registration and it is a separate task. As you would expect, you must use the credentials you supplied during registration, and we did this on the previous page. The call is again a POST, and it is very similar in shape to the registration request. The details are on the following page. Note initially the declaration at the top of the file. `JWT` will hold the token when we get it back from the login.

```
let JWT = null;

//regButton code goes here – removed for space

const logButton = document.getElementById("logBtn");
logButton.addEventListener("click", () => {
    fetch("http://hackhouse.sh:3000/login", {
      method: "POST",
      body: 'email=woof%40dog.com&password=WalksAndFoodForever',
      headers: {
            "Content-type": "application/x-www-form-urlencoded"
      }
    })
      .then(function(response) {
          if (response.ok) {
              return response.json();
          }
          throw new Error("Network response was not ok.");
      })
      .then(function(result) {
          let appDiv = document.getElementById("app");
          appDiv.innerHTML = JSON.stringify(result);
          JWT = result.token;
      })
      .catch(function(error) {
          console.log("Problem with fetch ",error.message);
      });
});
```

The parameters to the request are almost identical. The only difference is that we are hitting a different endpoint.

In the second `then` clause you will see the usual AJAX update, but here we also assign the token to the `JWT` variable. Click on the login button and you should see the image on the following page. The assignment statement accesses the token field from the JSON

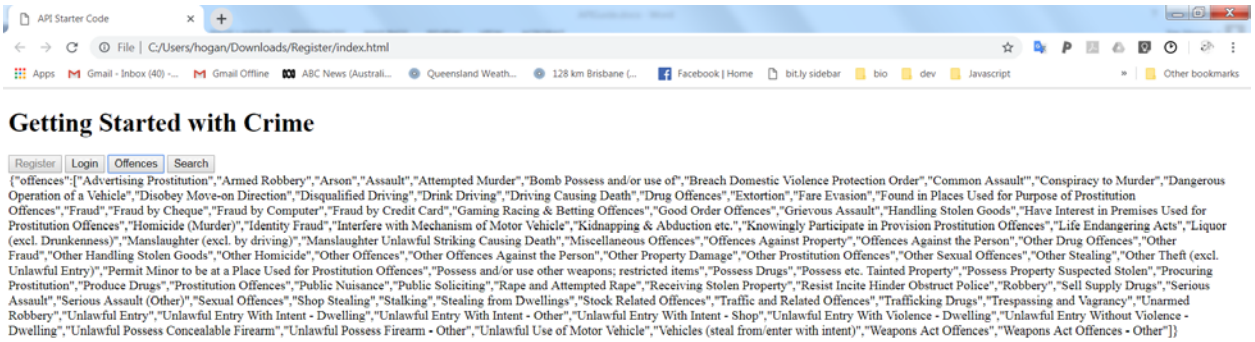
and stores the token for future use. The token is valid for 24 hours, after which you need to login again.

Getting Data:

We now grab some data, which is after all why we are here. We will begin with a simple GET request with no authentication:

```
const offButton = document.getElementById("offBtn");
offButton.addEventListener("click", () => {
    fetch("http://hackhouse.sh:3000/offences")
        .then(function(response) {
            if (response.ok) {
                return response.json();
            }
            throw new Error("Network response was not ok.");
        })
        .then(function(result) {
            let appDiv = document.getElementById("app");
            appDiv.innerHTML = JSON.stringify(result);
        })
        .catch(function(error) {
            console.log("There has been a problem with your
fetch operation: ",error.message);
        });
});
```

This is exactly the same as so many of the examples you have seen and requires little explanation. It is a simple GET hitting the `/offences` endpoint, which provides a list of the offence categories used in the dataset. Click on the Offences button and you will see:



The final task is much more complicated. Here we are going to hit the `/search` endpoint.

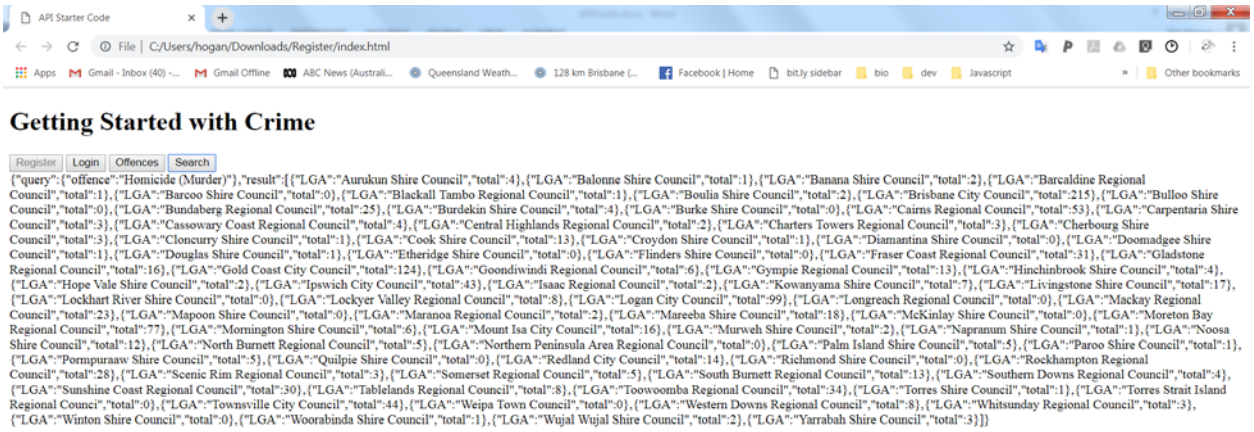
```
const searchButton = document.getElementById("serBtn");
searchButton.addEventListener("click", () => {

    //The parameters of the call
    let getParam = { method: "GET" };
    let head = { Authorization: `Bearer ${JWT}` };
    getParam.headers = head;

    //The URL
    const baseUrl = "http://hackhouse.sh:3000/search?";
    const query = 'offence=Homicide (Murder)';
    const url = baseUrl + query;

    fetch(encodeURI(url),getParam)
        .then(function(response) {
            if (response.ok) {
                return response.json();
            }
            throw new Error("Network response was not ok.");
        })
        .then(function(result) {
            let appDiv = document.getElementById("app");
            appDiv.innerHTML = JSON.stringify(result);
        })
        .catch(function(error) {
             console.log("Problem with fetch: ",error.message);
        });
});
```

The key lies in the construction of the request parameters and the URL for the query request. The request parameters show that the request is a GET, but most importantly here they provide the mechanism for authentication. We create an `Authorization` header, which follows the usual: `Authorization: <type> <credentials>` format. Here we use a `Bearer`, with the token providing the credentials. This is accomplished using the lines below, where the second assignment adds this JSON object as a field on the `getParam` object:

```
let head = { Authorization: `Bearer ${JWT}` };
getParam.headers = head;
```

Finally, we build the URL and include the query string. There are a number of ways of doing this. Here we use string concatenation and then call the `encodeURI` method to

ensure that the parameters are urlencoded. We then click on the search button and the results should appear as follows:



And so we are done… More to follow.