Edinburgh Napier University

SET08101 Web Tech

Lab 9 - API Design

Dr Simon Wells

1 Aims

The aim of this lab is to attempt to bring together the work we have done so far to identify opportunities for applying some RESTful principles and developing APIs. Instead of just providing websites, that require a human or a browser to connect to them, an API can let other software act as a web client. For example, a mobile app can connect to an API to retrieve data, probably in JSON format, and a web browser can connect to the exact same web address and recieve HTML. This embodies one of the principles of RESTful design that data can have many forms, even if that data is accessed at the same address.

At the end of the practical portion of this topic you will:

- have some understanding of the process of developing an API using HTTP verbs and URL routes
- have some experience of the JSON language for describing data
- be able to parse JSON strings into JavaScript
- be able to serialise JavaScript objects into JSON strings

2 Activities

This lab will require you to do a significant amount of background reading and exploration, as well as critical thinking. Whilst developing APIs is technically straightforward, JSON and HTTP are well understood technologies for representing and transporting data, the naming of collections, their arrangement within an URL hierarchy, the structure of the data on the server, and in responses to user requests, can all have many forms. This makes API writing both a technical process, but more importantly, a design process[1].

Like all design and development tasks, this is going to be an *iterative* process. We will try some things out, see if we are happy with the results, do some research, look to other examples for inspiration, and eventually an acceptable solution should coalesce. At each sub-activity, we may read or discover something that makes us return to an earlier stage to redesign somethings we already thought was complete. This is fine. It is part of the natural process, both of learning but also of design.

The next set of activities are basically designed to help us explore the process of taking existing web sites, such as our cipher websites, and turning them into an API. You might start off working on one activity and discover you need to return to an earlier one to adjust something. This means that the process is working properly and that you are being critical of the work that you've already done, in light of new ideas and knowledge that you have subsequently.

2.1 Data & Functionality

Start by considering the list of features that your cipher website has. What functionality does the interface expose to the user? What data can they supply to the website? and what do they get back in return? It might be that you haven't considered typing text into a textbox widget as supplying data, but that is exactly what it is. Similarly the process of updaing the DOM or writing the output of a cipher function into a text widget so that the user can read it is sending data back to the user. An API can do exactly the same, offering a way to send data to a given URL, which causes a function to run and which should generate a response. The difference is that the data can be in other forms than HTML. We will use JSON for representing data, but you should be aware of other technologies, such as XML (extensible markup language), RDF (resource description format), CSV (comma separated values). There are actually many languages and formats designed to store data so that it can be reused, some of them are quite specialist, for example the DOT language for representing graphs whilst others, like RDF, seek to be a general purpose data representation tool.

[1]It's a bit like naming variables and functions in JS, then organising these into packages and working out what data one function must send to another. There are many ways to write a program, and some are easier to understand, maintain, and extend, than others.

For the cipher websites, we just named an HTML file according to the cipher it contained, or even just numbered the pages. However there are a number of ways that we might structure the functionality of our cipher websites more generally as an API. For example, we might have an /encypt/ and a /decrypt/ route to which a user can send the message to encrypted or decrypted. Althernatively we might have a route for each cipher, e.g. /caesar/ then a subroute for encrypted and decryped versions of a supplied message, e.g. /caesar/encrypted/ and /caesar/decrypted/. When you call /caesar/encrypted/ and supply a message in the request, then an encrypted message is returned and similarly when the decrypted route is called. Note that rather than using encrypt and decrypt, which are verbs, we've chosen to use the adjective form. This might seem a bit clunky right now but it fits with the RESTful approach of treaing URLs as objects and collections of objects using nouns and adjectives to indicate named objects, or types of object, and their properties. We could equally well have taken yet another approach and had a /message/<cipher>/encrypted route or even /message/encrypted/ route but the caller has to tell the server in the request which cipher to use. Not that this last way is quite nice as it means that we have a simple URL route, e.g. /message/encrypted/ but it does mean that we need to make the range of ciphers available to be used *discoverable* to the user. That means we perhaps need a second route, perhaps named /ciphers/ and calling this route with the GET verb will return a list of available ciphers.

Attempt to design a URL hierarchy for a cipher API, considering how you would call it, which verbs could be used for each route, and what data would be supplied to and returned from each route. When you think you have a nice design, share it with a classmate and solicit their feedback. This is actually an important part of the API design process. The people who make the most use of an API are often not the designers of that API but people who need to use it to retrieve data of some sort.

2.2 JSON

Once you have some idea of your collections of objects in your API, then you will need to decide how to store the data associated both with the collections but also with each individual that makes up the collection. For example, for the /ciphers/ route called with a GET request, we should expect to recieve back the collection of ciphers. But what should this contain, perhaps it is just a list of the ciphers that your API supports? Perhaps there is more information than this, perhaps documenting each cipher available with parameteres that might need to be supplied. Or perhaps the list merely contains the names of the available ciphers and URLs for each individual cipher route. So that the client can choose to navigate to an individual cipher page to find out more about an individual cipher, and we tell our user how to do so. Already this sounds like we need to have a /ciphers/<cipher>/ route where <cipher> is replaced by the name or ID of a given cipher, e.g. caesar. Perhaps it might be time to return to the previous task and adjust our URL hierarchy, or perhaps we should plough on.

This section started by mentioning data associated with URLs and routes. We'll use JSON to represent our data. This is a simple language for describing data. You can get more information about the language itself from the JSON language webpage at `http://www.json.org`. JSON is easy to write using a text editor but there are also additional tools that help us to check with we have the syntax correct as, like all programming languages, it likes things to be written precisely and correctly. We can use JSON Lint `http://jsonlint.com/` to automatically check whether a given JSON document is corrent. It is also a useful web-based JSON editor for fairly short documents.

Here is a simple example of a JSON document:

```
{
  "firstName": "Jebediah",
  "lastName": "Springfield",
  "isAlive": true,
  "age": 125,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd
Street",
    "city": "Springfield",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
}, {
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Try typing that into jsonlint and playing around with it, letting the tool tell you when you've broken things. Use the railroad diagrams from json.org to help you understand how a JSON document is structured. JSON is quite straightforward. The basic *unit* of JSON is the document which can contain an object or a list. In turn an object or list can contain any number of encapsulated objects, lists, or key:value pairs.

Because JSON originally stems directly from JavaScript, it was developed as a way to serialise JS objects as text, hence JavaScript *Object Notation*, it is straightforward in JS to move between a string that contains JSON and an object that instantiates the variables that the JSON string described. The only thing to be aware of is making sure not to get string and object representations of JSON mixed up and attempting to treat one as the other.

This fragment of JS will give you an idea of how to turn a JSON string, such as the data that you might supply in an API call, or might be sent as the result of calling a third party API:

```
var j_str = '{"firstname":"simon"}';
var j_obj = JSON.parse(j_str);
console.log(j_obj);
console.log(j_str)
```

In our first line we have JSON encoded as a string and stored in j_str. We then use the JSON.parse() method to create a JS object hierarchy which we store in j_obj. The parse method simply converts each element that JSON describes in the string into objects, arrays, or basic datatypes so that they can be used for computation by JS. Note that we print out the value of j_obj and j_str so that you can compare what the string looks likes and the printed object.

Once JSON is parsed from a string into an object we can use it in our code, for example, we can add new or remove existing elements or perform computations with the data. For example, adding a new key to j_obj:

```
j_obj['lastname'] = "wells";
```

Finally we can convert our JS objects into JSON strings, for example:

```
var s = JSON.stringify(j_obj);
console.log(s);
```

This JSON could then be written to a file, persisted in a datastore, sent to a remote API, or returned to our user as the response to a HTTP request.

2.3 Putting It All Together

Try translating your url hierarchy into a node/express web-app. Implement the routes and return JSON or HTML documents depending upon how a given URL is called. You will probably have to explore the documentation for both Node and Express to achieve this, but that is part of the life of a developer.