Using the National Rail Transport API

National Rail provides a set of Transport web services, described in their portal site at https://developer.transportapi.com/documentation/train-information. Although the UK has several train operators, National Rail operates to integrate schedule and information services for all of them. Given our geographical situation, I'll limit this document to a description of the services as they apply to Scotrail.

The portal site (URL given above) provides train information as follows:

- **Nearby Stations**: a service to provide a list of stations, listed in order of increasing distance from a specific geo-location coordinate pair. This service can be used to get information based on the GPS location provided by a smartphone. JSON is a supported format, and by adding &callback=… to the URL, the service will return JSONP. For example, the following URL will return a JSONP list of stations for a specific latitude and longitude:
  `http://transportapi.com/v3/uk/train/stations/near.json?lon=-0.102323&lat=51.527789&callback=func`
- **Stations within a bounding box**: (i.e. within an area described by maximum and minimum latitude and longitude values):
  `http://transportapi.com/v3/uk/train/stations/bbox[.format]?{minlon=&}{minlat=&}{maxlon=&}{maxlat=}[&page=][&rpp=]`
- **Stations served by a specific operator**:
  `http://transportapi.com/v3/uk/train/operator/{atoc_code}/stations[.format]?[rpp=]`
  atoc_code is the National Rail Operators ID code – the code for ScotRail is "SR". rpp is the number of results per page
- **Scheduled station departures**: i.e. a list of departures from a given station for a specific data and starting time:
  http://transportapi.com/v3/uk/train/station/{station_code}/[date]/[time]timetable[.format]?[limit=][&origin=][&destination=][&calling_at=][&called_at=]
  The inputs to this function are station_code (see later), date in yyyy-mm-dd format time in hh:mm format. Optional inputs (query variables) are limit (max number of results), origin (the code for the station the train started from), destination (the code of the station at the end of the line), calling_at (a code for a station to call at and called_at (code of a station the train has previously stopped at). The {station_code} in the URL is the station you would be boarding the train at. Generally with this query, you would want to specify a calling_at station code or a destination station code (where you intend to get off). The result is a list of departures from the stated station, commencing at the date and time specified.
- **Live station departures**: as above, but containing corrected times of past departures and arrival estimates for ones in the future
- **Scheduled service**: i.e. the next scheduled train service past a given date and time

Of these, the nearby stations and scheduled station departures are likely to be the most useful for an app that would provide good train information for someone looking for train information for a journey. To use the services, you will need to apply for a developer key,

which is added as a query parameter to every request. Developer keys for small apps (less than 1000 requests per day) are free. To get a developer key, fill in the form at https://developer.transportapi.com/signup. You will of course have to adhere by the terms and conditions to get access to the services, but unless you're either attempting to do something criminal (such as hack the services) or stupid (such as posting silly numbers of requests for no reason) this should not be a problem.

Station Codes

Stations throughout the UK are identified by three letter alphabetical codes – for example Paisley Gilmour Street is PYG and Glasgow Central Station is GLC. To send a request for a train departure time, you need to know the code of the station you intend to depart from. A file, station_codes.js is available on Moodle at the same location as this document. This lists the codes in the format…

```
var stations = {
      "Aberdeen": "ABD",
      "Aberdour": "AUR",
      etc..
}
```

This format is optimal for looking up stations by name, since:

```
var code = stations["Aberdeen"];
```

will put the appropriate station code into the code variable. You can create a <select> list of the stations names that will allow a user to select a station to get the code using the following code snippet:

```
function fillSelectList(select_id) {
    var list = "";
    for (station in stations) {
        list += "<option value='" + stations[station] +
                              "'>" + station + "</option>";
    }
    $("#" + select_id).html(list);
}
```

Similar code can be used to create a listview:

```
function getListViewItems() {
    var index, list = "";
    for (station in stations) {
        list += "<li id='" + stations[station] + "'>" +
                                        station + "</div></li>";
    }
    return list;
}
```

In this case, the function returns a list in a format that could easily be stashed in a string variable, which will have an advantage for displaying filtered content.

With a list of station codes available to our app, we can now build calls to the one of service points provided by the transport API. For example, to get a list of departures, we can use the Scheduled Station Departures service, which is arranged as follows:

> http://transportapi.com/v3/uk/train/station/{station_code}/[date]/[time]timetable[.format]?[limit=][&origin=][&destination=][&calling_at=][&called_at=][&api_key=][&app_id=]

This URL includes a number of required parameters that are better explained by example. Let's say we are walking towards Paisley Gilmour Street station and need to find out when the next train to Glasgow Central is due to leave. It is Wednesday 15th of April and we'll will get to the station around 2:20p.m. It would be useful to know which platform to head for (better than standing on platform 2 watching a train leave from platform 3). The info we will need to pack into the service call is:

Station code of Paisley Gilmour Street: PYG

Station we want to travel to – (calling_at): GLC

Earliest departure time: 14:20

Date of departure: 2015-04-15

We also want the timetable data to be returned in JSON format, and because this is a call to an external origin, we will want the JSON data returned in the JSONP format. Fortunately, jQUery will handle that for us, but we will also need to include our api_key and app_id, provided by registering with the TransportAPI site.

Putting this together into the URL, we get:

http://transportapi.com/v3/uk/train/station/PYG/2015-04-15/14:20/timetable.json?api_key=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx&app_id=xxxxxxxx&calling_at=GLC

Note that I have replaced the 32 character api_key and the 8 character app_id with "xxx" characters (use your own key – not mine). The rest of the URL is correct. Note the use of the "calling_at" parameter to indicate our destination. There is a "destination" parameter we could use, but that would only work for a journey where the user was specifically going to the final destination of the train (in this case, that would be ok). If we were making the reverse journey, the destination would be wherever was at the end of the line (Largs, Ayr, Gourock etc.).

Organising this as an AJAX call in jQuery is fairly simple:

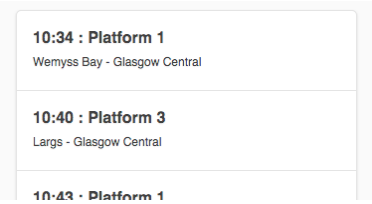
```
// These variables provide the core service address.  The KEYS value is used
// to encode whatever your api_key and app _id are – the values supplied by
// TransportAPI.
var CORE_URL = "http://transportapi.com/v3/uk/",
    KEYS = "api_key=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx&app_id=xxxxxxxx",
```

```
    URL = CORE_URL + "train/station/$$FROM$$/$$DATE$$/$$TIME$$/" +
         "timetable.json?" + KEYS + " &calling_at=$$TO$$";

// Now the AJAX calling code...

function getDepartureList(from, to, date, time) {
    var query_url = URL.replace("$$FROM$$", from)
                       .replace("$$TO$$", to)
                       .replace("$$DATE$$", date)
                       .replace("$$TIME$$", time);
    $.ajax({
        url: query_url,
        type: 'GET',
        async: true,
        contentType: "application/javascript",
        dataType: "jsonp",
        success: function(data) {
            showDepartures(data.departures.all);
            // this function displays the results in a listview
        },
        error: function(err) {
            console.dir(err);
        }
    });
}
```

Note the use of the String.replace() function to replace fixed tokens in the URL ($$FORM$$, $$DATE$$ etc.) with actual values. Normally these values would be picked up from a user-interface – input fields for time, date and listviews for 'from' and 'to' stations. The showDepartures() function is a straightforward one which displays various parts of the data returned from the service – in particular, the json data returned contains a departures list each item of which is info on a specific departure. In the above code, data.departures.all is an array of departure data items. Each element of this array is organised as follows:

```
item = {
    "mode":"train",
    "service":"13569815",
    "train_uid":"G90895",
    "platform":"1",
    "operator":"SR",
    "aimed_departure_time":"14:22",
    "aimed_arrival_time":"14:22",
    "aimed_pass_time":null,
    "origin_name":"Gourock",
    "source":"ATOC",
    "destination_name":"Glasgow Central"
}
```

In our app, we would probably want to display aimed_departure_time, platform, origin_name and destination_name as results – e.g. .



Making an Auto-completion list

For a device with a small screen, any technique you can use to make efficient use of screen space will benefit an app. Auto-complete is a familiar enough concept – for example, the Google search-box that shows a short list of matches to search terms as you type.

The automatic filter you can get by adding a data-filter="true" property to a listview is not ideally suitable for this because of the excessive length of the list (360 odd stations) – it would be best not to display the whole list at any time. Instead, the UI space could be minimised by using a listview and a separate filter box, with HTML code as below:

```
<form>
    <input type="text" data-type="search" class="filter-box"
           placeholder="From..." id="from-box">
    <div class='filterable_list'>
       <ul id="from-list" data-theme='b' data-role='listview'
           data-inset='true' data-filter='true' data-input="#from-box">
       </ul>
    </div>
</form>
```

The filter box

The list

Note that the <UL> element is currently empty and has a data-input setting that identified the text box that will be used as a filter, which has been characterised as a "search" box. To make this work nicely, it would be sensible to keep the <UL> item hidden until a few characters have been typed into the search box:

```
var lvList = getListViewItems();
$("#from-list").html(lvList);
$("#from-list").trigger('listview');
$("#from-list").hide();
$("#from-box").on( "keypress", function() {
    if($(this).val().length > 2) {
        $("#from-list").show();
    }
});
```

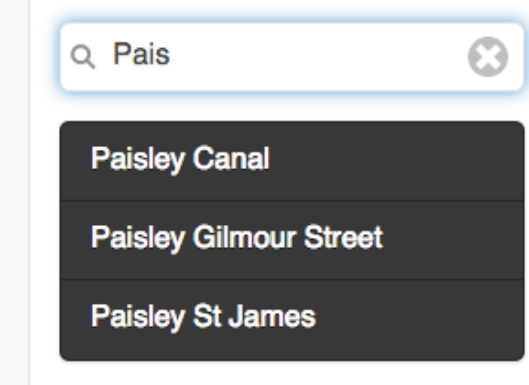
Note that above, `lvList` is used to store all of the HTML mark-up for a listview containing a complete list of stations. It is worth doing this because the list of stations could be used more than once in a page – the 'from' and 'to' station lists for example. I've shown the call to `getListViewItems()` here for clarity, but it would be preferable to make sure it is called only once in `$(document).ready()`. Stashing all of that mark-up in a single variable that is referenced more than once is efficient.

The process in the above code is to get the list items mark-up (from `getListViewItems()`), assign that to the <ul> of the list of "from" stations, trigger the jQuery mark-up of the list-view and then immediately hide the list-view. When the user types into the "from-box" (e.g. "Gla") this will filter the list of matching items and then show() it. Now instead of over 300 items, the list will show 3 or 4.

The only remaining issue is to deal with the user tapping on an item in the list to select it – we need to take the tapped item and transfer its displayed text to the search-box. Having picked up the text from the item tapped (which in turn will give us the code, since this text is a key in our stations[] array, we can then hide the list:

```
$("#from-list li").on("click", function() {
    var selectedItem = this;
    $("#from-box").val(selectedItem.textContent);
    // close all, in case still open
    $("#from-list").hide();
});
```

With this code, we can create an efficient way for users to pick from a long list of stations.



Demo App

There is a demo app at http://cis.uws.ac.uk/alistair.mcmonnies/mobile/TransportAPI/. Which uses the TransportAPI services along with the code above for auto-completion lists. You can access this on a desktop, tablet or phone. Please note the app is incomplete (no manifest and a few other features I'd like to add, such as identifying nearest station). The JS code in it is minified (so a copy of it will be of little use to you).