

# AMIS Talent Launch

## API, REST, JSON, Node

---

September 2017

(v0.99)

In this workshop, you will get to know APIs – especially RESTful APIs – and learn how to invoke these APIs using the HTTP protocol. You will learn what JSON – the data exchange format – is and how it can be used. You will design a RESTful API and after an introduction to JavaScript (the programming language) and Node (the platform for running JavaScript programs), you will continue to implement your API – making it communicate with various backend systems. You will make your API publicly available – both by exposing your local API through the *ngrok* PaaS service and by deploying your API implementation on the *now* cloud service.

You will work with tools such as Chrome browser, Visual Code Studio editor, Postman (testing REST APIs) and Apiary.io (for API design and mock implementation).

You can get access to the sources for the practices from the GitHub repository:

<https://github.com/GJPJansen/amis-course-2017> .

## 1. Programmatic access to Public Data

For this practice, you first need to install two tools:

- Chrome Browser - <https://www.google.com/chrome/browser/desktop/index.html>
- Postman – add on for Chrome - <https://www.getpostman.com/apps> - installation instructions: [https://www.getpostman.com/docs/postman/launching\\_postman/installation\\_and\\_updates](https://www.getpostman.com/docs/postman/launching_postman/installation_and_updates)

## Web applications powered by Data

Many of the world's best known web sites and almost all web applications are powered by dynamic data. Data that is retrieved from a backend system and combined in the browser with HTML and CSS to present the data in an attractive, meaningful context and with controls for scrolling and navigating, filtering, drilling down etc.

Open the site <http://www.bbc.com/news/world> that presents world news stories – to human readers.



Open the URL: <http://news.bbc.co.uk/1/hi/help/rss/default.stm>. This page introduces the newsfeeds of the BBC. A newsfeed is the programmatic counterpart of a website aimed at human users. The newsfeed can be accessed by computer applications to get hold of data in a structured format.

Check out the newsfeed for the world news: <http://feeds.bbc.co.uk/news/world/rss.xml?edition=uk> in your browser.

← → ↺ feeds.bbci.co.uk/news/world/rss.xml?edition=uk

## BBC NEWS

### What is this page?

This is an RSS feed from the BBC News - World website. RSS feeds allow you to stay up to date with the latest news and features you want from BBC. To subscribe to it, you will need a News Reader or other similar device. If you would like to use this feed to display BBC News - World content on your website, [Help, I don't know what a news reader is and still don't know what this is about.](#)

### RSS Feed For: BBC News - World

Below is the latest content available from this feed. This isn't the feed I want.

#### Myanmar's Aung San Suu Kyi to miss UN General Assembly debate

It comes as the nation's de facto leader faces criticism over her handling of the Rohingya crisis.

#### Hurricane Irma: Quarter of Florida Keys homes 'destroyed'

Hurricane evacuees return to scenes of devastation as President Trump prepares to visit Florida.

#### Edie Windsor: Gay rights trailblazer dies aged 88

"The world lost a tiny but tough-as-nails fighter for freedom, justice and equality," her wife said.

#### North Korea threatens US with 'greatest pain' after UN sanctions

Pyongyang's envoy to the UN rejects "illegal resolution" that imposed new sanctions on the country.

#### Rebel Wilson wins large defamation payout

The "unprecedented" payout follows the actress's claim her career was stifled by untrue articles.

#### Brazil corruption scandal: President Temer slams judiciary

His statement came hours before a Supreme Court justice authorised a new probe into the president.

#### Seattle Mayor Ed Murray resigns amid sexual abuse allegations

Ed Murray denies the historical abuse allegations but says stepping down is "best for the city".

#### Five sailors missing after ship collision off Singapore

Authorities are conducting rescue operations and have deployed search vessels and a helicopter.

#### Spain Catalonia: Ballot papers for banned referendum to be seized

The independence vote has been blocked by the constitutional court but local leaders vow to hold it.

#### French protests target Macron labour reforms

In the first big test of the new French president, protesters challenge changes to labour laws.

Now check the source for the page (using CTRL U or Context Menu | View Page Source):

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml:stylesheet title="XSL Formatting" type="text/xsl" href="/shared/bhp/xsl/rss/nolsol.xsl"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:content="http://purl.org/rss/1.0/modules/content/" xmlns:atom="http://www.w3.org/2005/atom" version="2.0" xmlns:me="http://www.bbc.co.uk/news/rss/1.0/modules/content/">
  <channel>
    <title>[[[CDATA[BBC News - World]]]]</title>
    <description>[[[CDATA[BBC News - World]]]]</description>
    <link>http://www.bbc.co.uk/news/1/1</link>
    <image>
      <url>http://www.bbcimg.co.uk/news/shared/img/bbc_news_120x60.gif</url>
      <title>BBC News - World</title>
      <link>http://www.bbc.co.uk/news/1/1</link>
    </image>
    <generator>RSS for Node</generator>
    <lastBuildDate>Wed, 13 Sep 2017 06:24:45 GMT</lastBuildDate>
    <copyright>[[[CDATA[Copyright: (C) British Broadcasting Corporation, see http://news.bbc.co.uk/2/1/help/rss/4498287.stm for terms and conditions of reuse.]]]]</copyright>
    <language>[[[CDATA[en-gb]]]]</language>
    <ttl>15</ttl>
  </channel>
  <item>
    <title>[[[CDATA[Myanmar's Aung San Suu Kyi to miss UN General Assembly debate]]]]</title>
    <description>[[[CDATA[It comes as the nation's de facto leader faces criticism over her handling of the Rohingya crisis.]]]]</description>
    <link>http://www.bbc.co.uk/news/world-asia-41250057</link>
    <guid isPermaLink="true">http://www.bbc.co.uk/news/world-asia-41250057</guid>
    <pubDate>Wed, 13 Sep 2017 06:23:05 GMT</pubDate>
    <media:thumbnail width="976" height="549" url="http://c.files.bbci.co.uk/3532/production/_97781631_1d9bc72a-48f4-4084-b1a9-475baf0814d7.jpg"/>
  </item>
  <item>
    <title>[[[CDATA[Hurricane Irma: Quarter of Florida Keys homes 'destroyed']]]</title>
    <description>[[[CDATA[Hurricane evacuees return to scenes of devastation as President Trump prepares to visit Florida.]]]]</description>
    <link>http://www.bbc.co.uk/news/world-us-canada-41247063</link>
    <guid isPermaLink="true">http://www.bbc.co.uk/news/world-us-canada-41247063</guid>
    <pubDate>Wed, 13 Sep 2017 06:40:34 GMT</pubDate>
    <media:thumbnail width="976" height="549" url="http://c.files.bbci.co.uk/0134/production/_97780565_vilano.jpg"/>
  </item>
  <item>
    <title>[[[CDATA[Edie Windsor: Gay rights trailblazer dies aged 88]]]]</title>
    <description>[[[CDATA["The world lost a tiny but tough-as-nails fighter for freedom, justice and equality," her wife said.]]]]</description>
    <link>http://www.bbc.co.uk/news/world-us-canada-41248327</link>
    <guid isPermaLink="true">http://www.bbc.co.uk/news/world-us-canada-41248327</guid>
    <pubDate>Wed, 13 Sep 2017 02:45:19 GMT</pubDate>
    <media:thumbnail width="976" height="549" url="http://c.files.bbci.co.uk/0134/production/_97780380_mediatem97780299.jpg"/>
  </item>
  <item>
    <title>[[[CDATA[North Korea threatens US with 'greatest pain' after UN sanctions]]]]</title>
  </item>
```

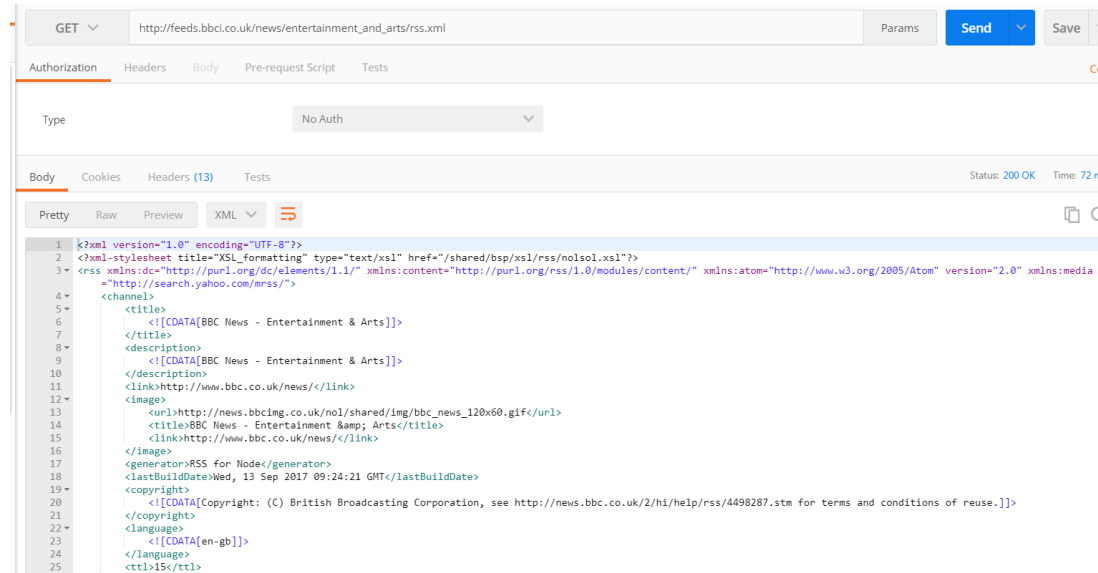
This is not regular HTML – as your browser would normally receive and show.

Open the Developer Tools – Ctrl Shift I or Menu | More Tools | Developer Tools. Inspect the response that your browser received. What is the format of the data returned from the newsfeed? Which response header specifies this format?

Try the Entertainment newfeed: [http://feeds.bbc.co.uk/news/entertainment\\_and\\_arts/rss.xml](http://feeds.bbc.co.uk/news/entertainment_and_arts/rss.xml) . Verify if the format and structure of the data are similar to that of the World News feed.

Then try a different RSS feed; pick one of the Dutch RSS feeds at <https://nos.nl/feeds/>. Compare the responses from the BBC and NOS RSS feeds: are they the same? Can we use the same computer program to deal with data from these feeds?

Use Postman to make a request to one of the newfeeds. Instructions on using Postman can be found here: [https://www.getpostman.com/docs/postman/launching\\_postman/sending\\_the\\_first\\_request](https://www.getpostman.com/docs/postman/launching_postman/sending_the_first_request)

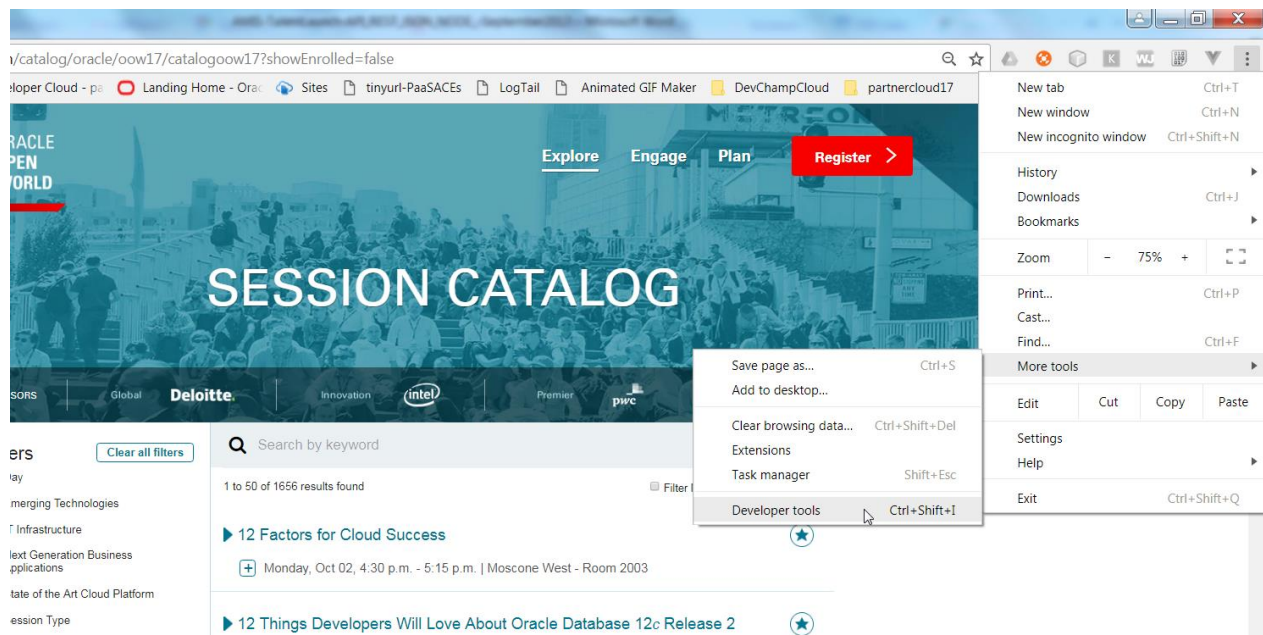


## Analyzing and leveraging API calls from web applications

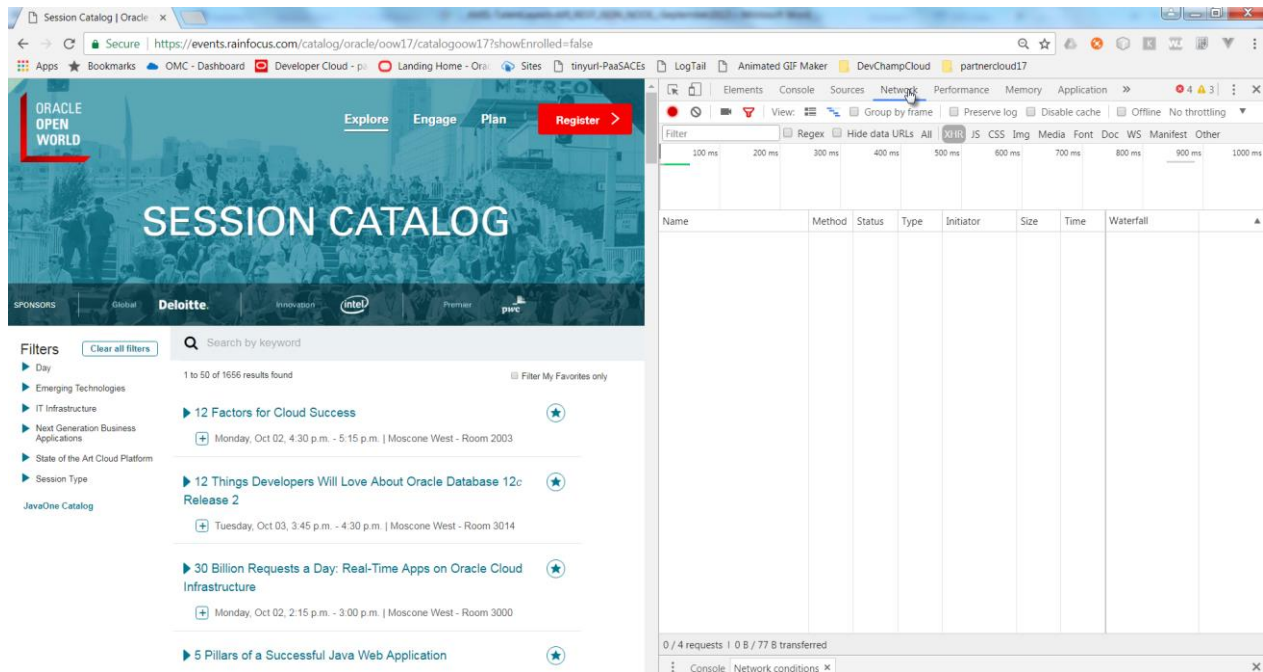
Open this URL: <https://events.rainfocus.com/catalog/oracle/oow17/catalogoow17?showEnrolled=false> in the Chrome browser. It will take you to the Session Catalog for Oracle OpenWorld – an overview of all sessions (over 1600 of them) that take place during the yearly Oracle OpenWorld conference in San Francisco. As a human user, we can search, browse and inspect details for the sessions, by interacting with this website through our browser.

If data is available in a browser to human users, it can be retrieved programmatically and persisted locally in for example a JSON document. A typical approach for this is web scraping: having a server side program act like a browser, retrieve the HTML from the web site and query the data from the response. It is very well possible that the rich client web application is using a REST API that provides the data as a JSON document. An API that our server side program can also easily leverage. That turns out the case for the OOW 2017 website – so instead of complex HTML parsing and server side or even client side scraping, the challenge at hand resolves to nothing more than a little bit of REST calling.

Open the Developer Tools – Ctrl Shift I or Menu | More Tools | Developer Tools.

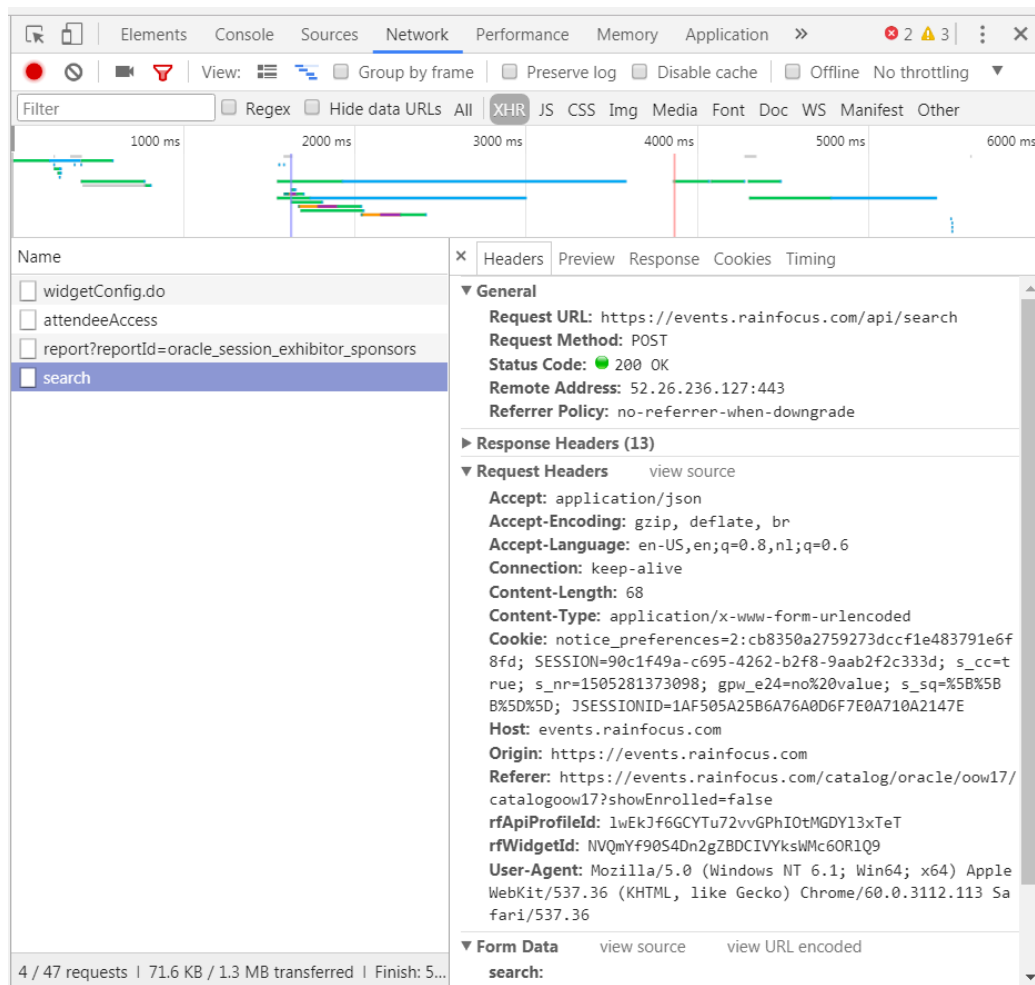


Open the Network Tab:



and press F5 to reload the page. Then click on the XHR category (XHR == XML Http Request aka AJAX call).

Select the call with name equal to search:



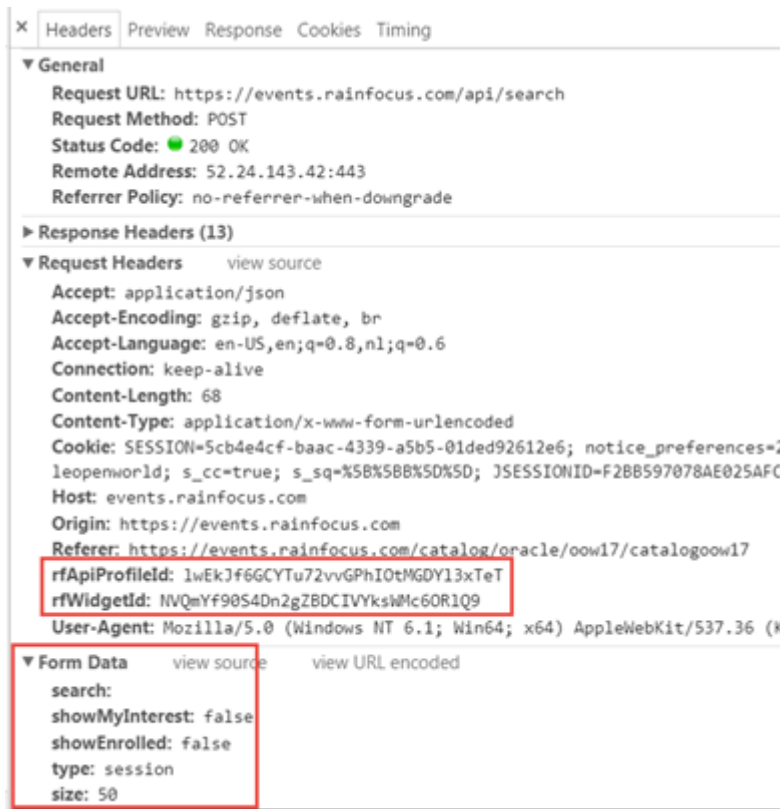
Try to understand what kind of request was made. You should identify the URL and the HTTP Verb (Request Method). Now turn to the request headers. In addition to URL and Method, this call also includes a number of headers.

Check the Preview tab to see a pretty formatted overview of the response that was received to this specific call. The session data is formatted into a JSON document – a structured format that can easily be processed by computer programs – especially those written in JavaScript.

Let's use Postman to construct the HTTP request we should send in order to receive the OOW session data. If we know how to make the request in Postman, it is only a small step to make that call from our own computer program and thereby get programmatic access to the data. We can then move forward to analyze the data, create reports or dashboards with it, store the data in our own database etc.

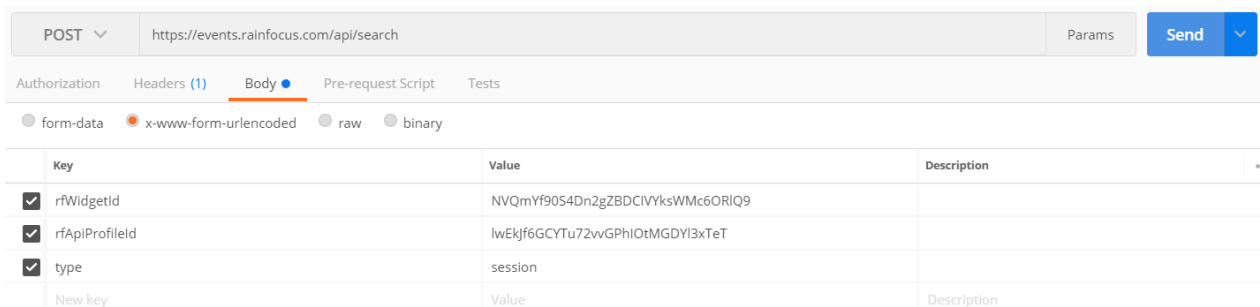
The information in the browser tools reveals two headers that turn out to be required when calling for session data:





A little experimenting with custom calls to the API in Postman made clear that `rfWidgetId` and `rfApiProfileId` are required form data.

Now you turn to Postman and construct a new request, as shown in the screenshot below.



You have to set the URL (`https://events.rainfocus.com/api/search`) and the method (POST). On the Body tab, specify `x-www-form-urlencoded`. Specify the keys `rfWidgetId` and `rfApiProfileId` – and use the values you have found on the Headers tab in Chrome. Also specify a key called `type` and specify `session` as value.

Press Send – and inspect the Response.



POST <https://events.rainfocus.com/api/search> Params Send Save

Authorization Headers (1) **Body** Pre-request Script Tests Code

form-data x-www-form-urlencoded raw binary

Key	Value	Description
<input checked="" type="checkbox"/> rfWidgetId	NVQmYf90S4Dn2gZBDCIVYksWMc6ORIQ9	
<input checked="" type="checkbox"/> rfApiProfileId	lwEkjf6GCYtu72wGPhiOtMGDI3xTeT	
<input checked="" type="checkbox"/> type	session	
New key	Value	Description

Body Cookies (7) Headers (13) Tests Status: 200 OK Time: 1297 ms

Pretty Raw Preview JSON

```

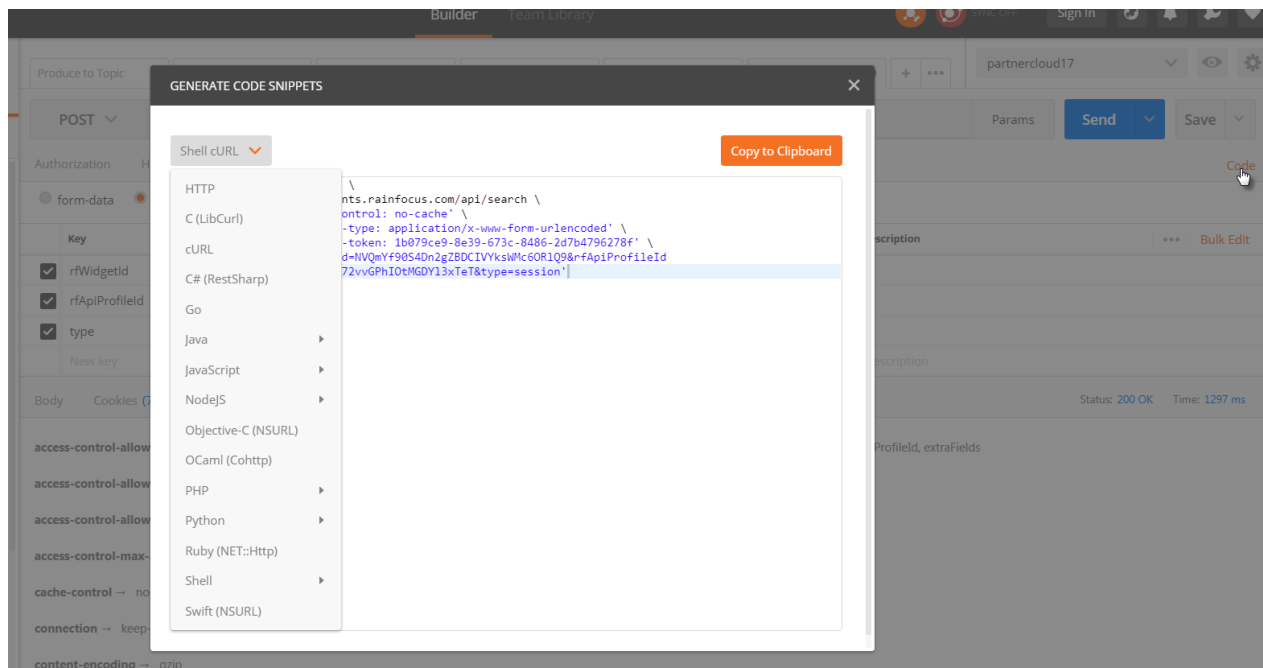
1 {
2   "responseCode": "0",
3   "responseMessage": "Success",
4   "sections": true,
5   "sectionList": [
6     {
7       "sectionId": "1",
8       "sectionTitle": "",
9       "total": 1656,
10      "numItems": 50,
11      "from": 0,
12      "size": 50,
13      "items": [
14        {
15          "sessionId": "149369513967100187K",
16          "code": "CON5598",
17          "abbreviation": "CON5598",
18          "title": "12 Factors for Cloud Success",
19          "abstract": "Think that developing apps for the cloud is changing everything? Almost! Luckily, we already have something to help us. Ever heard of '12-factor apps'? This set of best practices is an amazing guide to how you should create/modify your apps for this new cloudy world. Want to know

```

Scroll through the Response body. Toggle to Raw and Preview and back to pretty. Postman recognizes the format of the data (JSON) – and that allows it to show the pretty print with the colors and the indentation. Postman knows about this format from the Response Headers. Check them out in the Headers tab in Postman.

Extend the Postman request with additional filters: only request sessions on Tuesday October 3<sup>rd</sup>, somehow related to API. Hint: first try this search filter in the user interface, then take a look at the request sent from the browser to the backend.

Click on the Code button. This gives access to code snippets in many different technologies for making the same call we made Postman make from a computer program in a specific technology. Later in this training, we will use the NodeJS code snippet to make our call from JavaScript instead of Postman.



Open the site <http://www.tvguides.nl/lijsten> . It provides an overview – to human users – of the Dutch television guide. It allows users to navigate to specific channels, search on genre, date and part of day. Now your challenge: prepare a Postman request that obtains program details for Net 5 for tomorrow evening. Hint: use Developer Tools | Network tab in the browser to learn how the browsers gets program data from the backend.

Check out this site: <https://www.rijkswaterstaat.nl/apps/geoservices/rwsnl/awd.php?mode=html> . It provides the actual water heights (and gives access to historic water heights). Can you find the call to the backend API for this site? If not – how is the data passed to the browser in this case? In order to get programmatic access to the *waterstanden*: what can we do? What about programmatic access to the results in the Tour de France 2017 (see: <http://www.letour.com/le-tour/2017/us/stage-21/classifications.html> )

## 2. Public APIs

APIs are a gateway to data and operations. APIs expose – programmatically – resources that the API publisher is willing to share with API consumers. Note that this sharing can be free and for all – but can also be limited through access restrictions and payment requirements.

A long list of Public APIs is published here: <https://github.com/toddmotto/public-apis> .

### Genderize.io

A simple API is the one at: <https://genderize.io/> . Construct a Postman request that uses this API to indicate what are the genders of the first names Robin, Joe and William.

### Live Airtraffic Feed

One of the APIs listed in this GitHub Repo is at: <https://www.adsbexchange.com/data/#> ; this API provides access to information about airplanes currently (or historically) in flight.

Using Postman, construct a call to this API that returns information about airplanes currently flying over our heads. Make sure that you can tell the airline, mode, year of manufacturing, point of departure and destination for each airplane.

Feel free to try out one of the other APIs listed.

## APIs with Authentication Requirements - Spotify

So far we have seen fairly simple APIs. Most could be invoked using GET requests, just from the location bar in the browser. And most did not require any form of authentication. Some APIs are a little bit more involved to access. Sometimes because the request is more complex or requires several steps to go through and sometimes because we need to get access to the API – through credentials (username and password) or a special token (entrance ticket) with limited validity.

Let's take a look at the Spotify API: <https://developer.spotify.com/web-api/> ; this API provides information about among others artists, albums, songs and playlists. The details about the endpoints, resources and search parameters are in this API overview: <https://developer.spotify.com/web-api/endpoint-reference/>.

To be able to use the Web API, the first thing you will need is a Spotify user account (Premium or Free). To get one, simply sign up at [www.spotify.com](https://www.spotify.com).

Next, register your application at <https://developer.spotify.com/my-applications/#/applications/create> . This can be as simple as: My First Spotify Client App. After pressing Create, your app gets assigned a client id and a client secret.

You now have credentials: username and password for your account as well as client id and client secret for the application we are creating. At this point, the application will be no more than the Postman Collection.

Before we can invoke one the Spotify API endpoints, we first have to invoke the `/api/token` endpoint of the Accounts service in order to receive a token that we can then include in our API calls.

Follow the steps outlined here: <https://developer.spotify.com/web-api/authorization-guide/#client-credentials-flow> that instruct you to perform a call – from Postman – to the `/api/token` endpoint, providing your application’s client id and secret key; this request will result in a response that contains a token.

This looks like this:

The screenshot shows the Postman interface for a POST request to `https://accounts.spotify.com/api/token`. The 'Authorization' tab is selected. The 'Type' is set to 'Basic Auth'. The 'Username' field contains the client ID `00381c238d4048c99a0b1a105d96e0bd`. The 'Password' field contains the client secret, represented by dots. A 'Show Password' checkbox is visible below the password field.

Specify URL and Method (POST). Then set username to the client id and password to the client secret of your Spotify App.

On the headers tab:

The screenshot shows the Postman interface for the same POST request, but with the 'Headers' tab selected. It displays a table of headers:

Key	Value
<input checked="" type="checkbox"/> Content-Type	application/x-www-form-urlencoded
<input checked="" type="checkbox"/> Authorization	Basic MDAzODFjMjM4ZDQwNDhjOTlhMGlxYTEwNWQ5NmUwYm...
<input type="text" value="New key"/>	Value

On the Body tab, set the form parameter called `grant_type` to the value `client_credentials`:

The screenshot shows the Postman interface for a POST request to `https://accounts.spotify.com/api/token`. The 'Body' tab is selected, and the 'x-www-form-urlencoded' radio button is chosen. A table with two columns, 'Key' and 'Value', contains one entry: 'grant\_type' with the value 'client\_credentials'. A 'Send' button is visible in the top right corner.

Key	Value
grant_type	client_credentials

Press Send. The response should contain an access token:

The screenshot shows the Postman interface after the request has been sent. The 'Send' button has been clicked, and the response is displayed in the 'Preview' tab. The response is a JSON object with the following fields: 'access\_token', 'token\_type', and 'expires\_in'. A red dashed arrow points from the 'Send' button to the 'access\_token' field in the response.

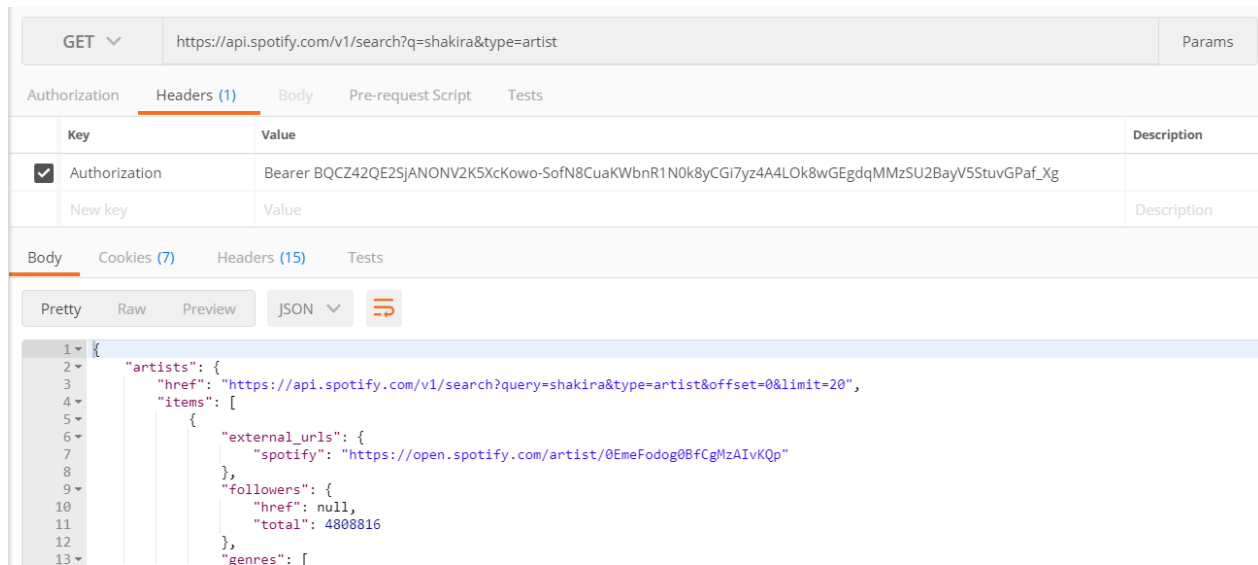
```
1 {
2   "access_token": "BQ0DVe814MC1pH4RGpP4WewMHNvSb9t-1kNGFY-p2LJYixhFRJtLaQUE1PIzS2zaFPj4QHhD1sutAgyR-TR8Pg",
3   "token_type": "Bearer",
4   "expires_in": 3600
5 }
```

Using this token – we can now start to query real music data. Before we start doing that – save the request to a new Postman Collection called *Spotify API Project*. A Postman collection can be saved, copied, shared etc.

Construct a call to find details about an artist called Shakira using the API endpoint:

<https://developer.spotify.com/web-api/search-item/> .

This looks like this:



Using the Spotify API, perform a search for one of your favorite artists. With the artist identifier, retrieve details about the artist. Finally, get the list of tracks from the latest album of the artist.

Save all requests to the same Postman Collection.

Using the Runner option, you can automatically make Postman run through all requests in a collection.

Try this now for the collection *Spotify API Project*. See for details on running collections:

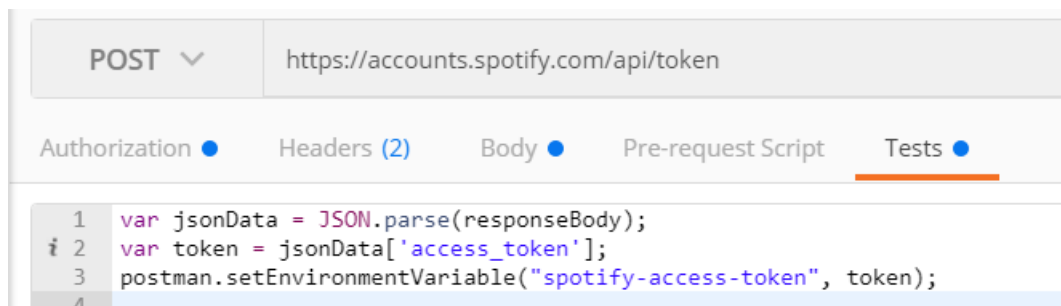
[https://www.getpostman.com/docs/postman/collection\\_runs/starting\\_a\\_collection\\_run](https://www.getpostman.com/docs/postman/collection_runs/starting_a_collection_run).

Additionally, you can add tests to a request – to verify whether the request returned the expected response. This is most meaningful for your own APIs that you want to continuously verify the health and correctness of. See [https://www.getpostman.com/docs/postman/scripts/test\\_examples](https://www.getpostman.com/docs/postman/scripts/test_examples).

It is fairly easy to take a value from a response (such as the `access_token`) and store it in a variable and to use the value in that variable in subsequent requests. The crucial call is:

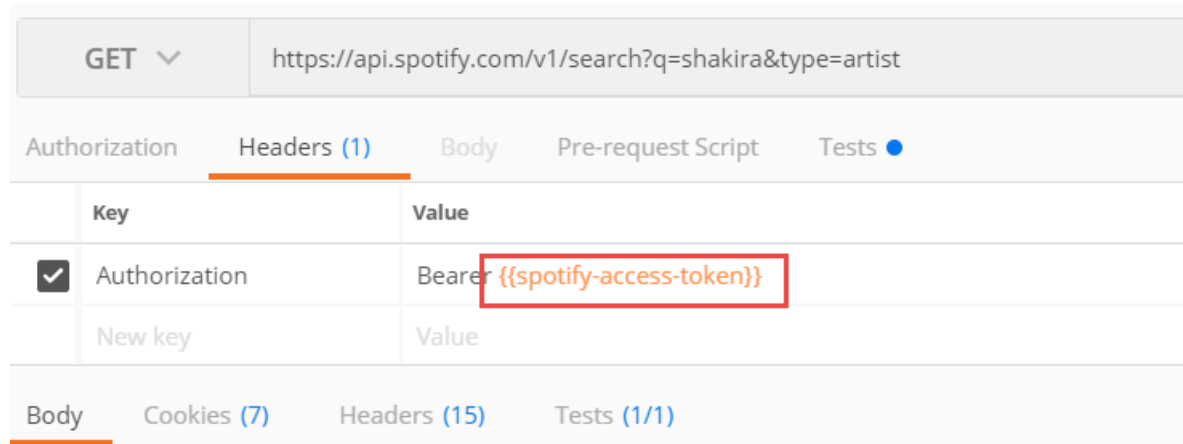
```
postman.setEnvironmentvariable('name of variable', value)
```

For example:



this code snippets stores the access\_token element from the Response in the environment variable spotify-access-token.

In the request header for subsequent API calls, we can use the {{spotify-access-token}} expression to refer to that variable and have its value injected when the request is made.



Clearly this makes the collection easier to run over and over again – because the hard coded token would expire after one hour and would have to be manually replaced.

Use this same approach of a test script to read a value from a response body and store that value in an environment variable to bring the ID of the artist from the search response to the request for artist details.

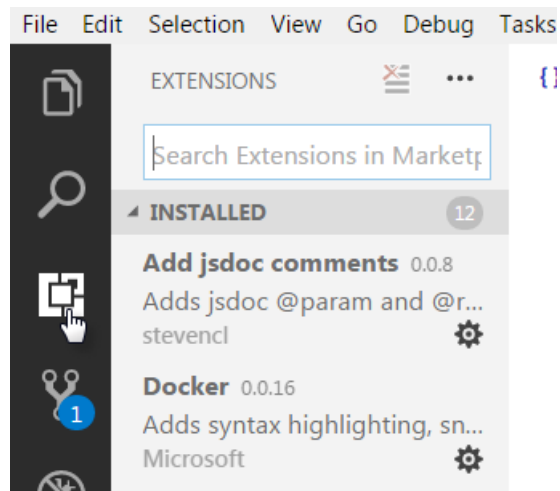


### 3. Getting around JSON

For this practice, you should first install a code editor: Visual Studio Code:

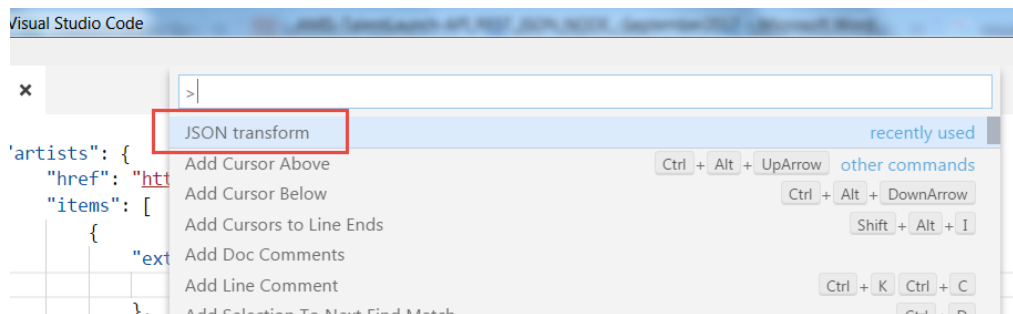
<https://code.visualstudio.com/>.

After installation, run Visual Studio Code. Open the tab with extensions:



Type the name of a useful extension to explore JSON: *JSON Transform*. Install this extension following the instructions.

Note: with Ctrl+Shift+P you can open the command palette at any time; JSON Transform will be listed on this palette and can be opened from here:



### Create your own JSON document with HRM data

Create a JSON document in Visual Studio Code. Call it: `hrm.json`.

In this document, define a department called Marketing, located in Utrecht and identified by an identifier 621. This department contains two employees – John Doe, 12 December 1974, Assistant Manager, salary of 2800 and an identifier: 8192 and John's manager Anna Finch, 24 June 1965, Marketing Manager, salary of 4100 and an identifier of 1302. Anna has three hobbies: opera, Thai cuisine and Norwegian literature. John's hobbies are Gaming and... nope, nothing else.

Use the right mouse button (context menu ) option Format Document to keep your document nice and tidy. See how the editor will point out syntax errors in your document when you violate the rules of a valid JSON document.

Now extend the document: the department has two locations, Amsterdam in addition to Utrecht. And there is a second department – called Finance, located in Nieuwegein and with 718 as its identifier. It currently has just a single employee – but more are on the way – called Harvey Hunk, 16 June 1992 who is the department's Secretary with a salary of 2100.

Open the JSON Transform window.

JSON data can be navigated using simple instructions:

- name of property (this can refer to a scalar property, to an object that itself has properties and to an array that can contain scalar values or objects with properties)
- index of element in array
- [] or [\*] to include all elements in an array
- [x:y] – slice of array: returns all elements from position x to position y (x and y are optional; position is zero-based)
- \* wildcard – to traverse all properties on a certain level

The first two of these can also be used in JavaScript – as you will see later in this training.

To get all employees in the first department, type:

```
departments[0].employees
```

```
{} hrm.json x departments[0].employees | Press 'Enter' to confirm your input or 'Escape' to cancel {} json-transform-preview.json x
1 {
2   "departments": [
3     {
4       "name": "Marketing",
5       "locations": [
6         "Utrecht",
7         "Amsterdam"
8       ],
9       "id": 621,
10      "employees": [
11        {
12          "name": "John Doe",
13          "birthdate": "12021974",
14          "job": "Assistant Manager",
15          "salary": 2800,
16          "id": 8192,
17          "manager": 1302,
18          "hobbies": [
19            "gaming"
20          ]
21        },
22        {
23          "name": "Anna Finch",
24          "birthdate": "24061965",
25          "job": "Marketing Manager",
26          "salary": 4100,
27          "id": 1302,
28          "hobbies": [
29            "opera",
30            "Thai cuisine",
31            "Norwegian Literature"
32          ]
33        }
34      ]
35    }
36  ]
37 }
```

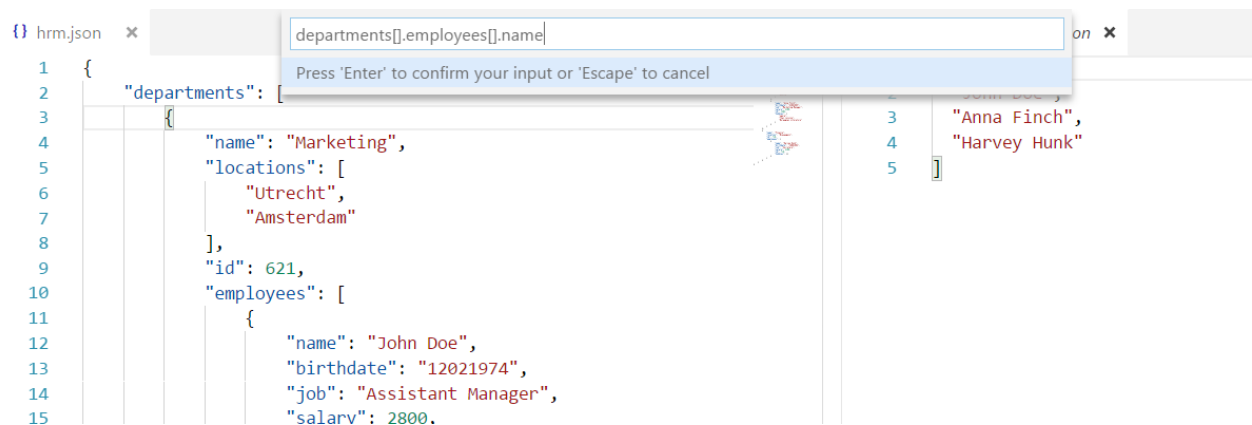
```
1 [
2   {
3     "name": "John Doe",
4     "birthdate": "12021974",
5     "job": "Assistant Manager",
6     "salary": 2800,
7     "id": 8192,
8     "manager": 1302,
9     "hobbies": [
10      "gaming"
11    ]
12  },
13  {
14    "name": "Anna Finch",
15    "birthdate": "24061965",
16    "job": "Marketing Manager",
17    "salary": 4100,
18    "id": 1302,
19    "hobbies": [
20      "opera",
21      "Thai cuisine",
22      "Norwegian Literature"
23    ]
24  }
25 ]
```

To get the 2<sup>nd</sup> hobby of the 2<sup>nd</sup> employee in the first department:

```
{} hrm.json x departments[0].employees[1].hobbies[1] | Press 'Enter' to confirm your input or 'Escape' to cancel {} json-transform-preview.json x
1 {
2   "departments": [
3     {
4       "name": "Marketing",
5       "locations": [
6         "Utrecht",
7         "Amsterdam"
8       ],
9       "id": 621,
10      "employees": [
11        {
12          "name": "John Doe",
13          "birthdate": "12021974",
14          "job": "Assistant Manager",
15          "salary": 2800,
16          "id": 8192,
17          "manager": 1302,
18          "hobbies": [
19            "gaming"
20          ]
21        },
22        {
23          "name": "Anna Finch",
24          "birthdate": "24061965",
25          "job": "Marketing Manager",
26          "salary": 4100,
27          "id": 1302,
28          "hobbies": [
29            "opera",
30            "Thai cuisine",
31            "Norwegian Literature"
32          ]
33        }
34      ]
35    }
36  ]
37 }
```

```
1 "Thai cuisine"
```

Type an expression that returns all names of all employees.



What do you think this expression will return: `*[*].*[*].name`

Note: at this URL <http://jmespath.org/tutorial.html> you will find instructions on the JSON search & transform syntax.

## Exploring a more substantial JSON document

Download the file <https://github.com/mledoze/countries/blob/master/countries.json> from the GitHub Repo <https://mledoze.github.io/countries/> on Country data.

Open this file in Visual Studio Code.

Using the JSON Transform tool, extract a list of all names of all countries:

```
[*].name.common
```

And a list the common names of all countries in Asia:

```
[?region=='Asia'].name.common
```

The official languages of Spain:

```
[?name.common=='Spain'].languages
```

To find the largest country (by area):

```
max_by([], &area).name
```

And now find the largest land locked country:

```
max_by([?landlocked], &area).name
```

Many more public data sets can be found in many places on the internet, for example:

- <https://github.com/caesar0301/awesome-public-datasets>
- [Kaggle](#),
- <https://github.com/opendatajson>.
- Nederlandse overheid: [https://data.overheid.nl/data/dataset?res\\_format=JSON](https://data.overheid.nl/data/dataset?res_format=JSON) .

## 4. Design an API

Sign up for Apiary.io at <https://apiary.io/>.

Watch first 20 minutes of introduction video at: [https://help.apiary.io/api\\_101/understanding-apiary](https://help.apiary.io/api_101/understanding-apiary). Then, go to [https://help.apiary.io/api\\_101/api\\_blueprint\\_tutorial/](https://help.apiary.io/api_101/api_blueprint_tutorial/) to read the tutorial for using Apiary for designing an API using the API Blueprint format. An alternative format is Swagger.

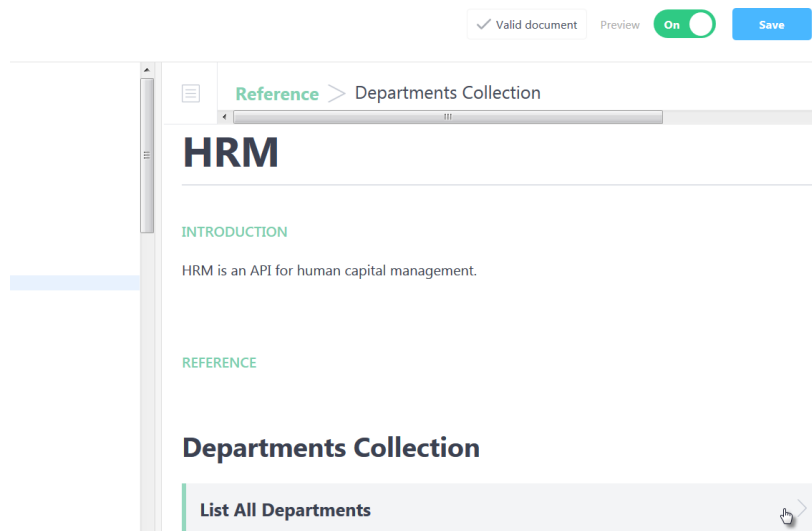
Start a new API project for an HRM API that covers the Human Capital domain – including departments and employees.

### Specify the first Resource and Operation

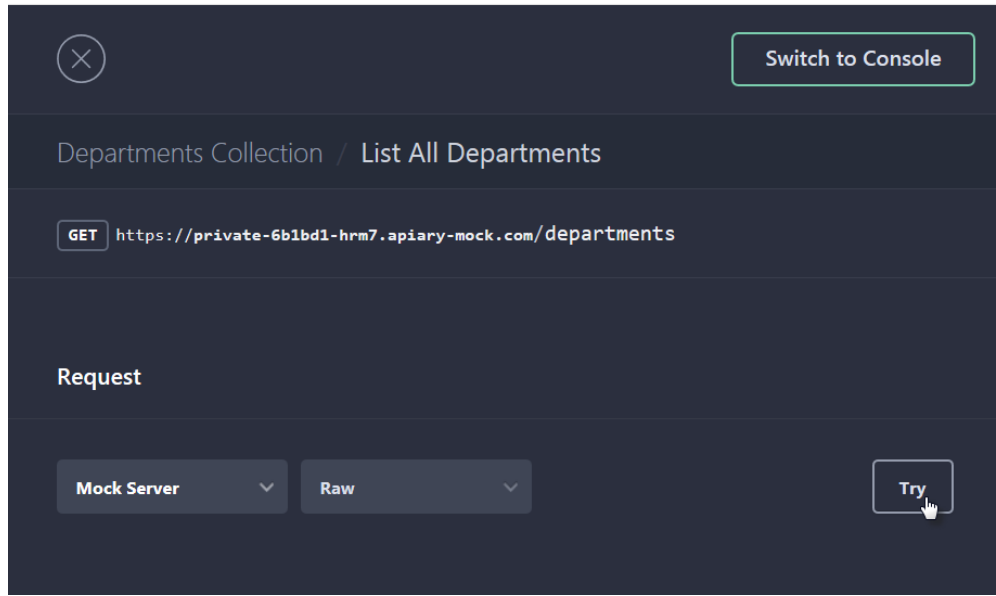
Modify the prepared Polls API – specify the Departments resource and a GET operation to retrieve all departments. Provide an example of the contents of the response – probably a JSON string with details about several departments.

Save the definition.

Browse through the documentation on the right hand side of the page. Locate the Departments resource, click on it to test the data retrieval through the GET operation.



Perform a test call – to the Mock Server using the Raw option for technology.



Inspect the response.

Go to the Inspector and verify that the test call was observed. Inspect the details for the test call.

Copy the URL for the Resource endpoint to the clipboard. Open Postman. Paste the URL into a new request in Postman and verify that the call can be made successfully and returns the expected response.

Save the request to a new Postman Collection called *HRM API Project*.

## Design the JSON data structure for Employees

Create a sample JSON document with a full structure with all properties for an employee. This document will be used for the API request or response body contents for POST, PUT and GET request for Employee resources.

## Complete the API Design

Continue to design the HRM API. Add resources and operations one at a time. Describe the operation, the parameters (URI Template and Query), provide samples for request message and/or response message, headers, content types etc. Save the API design regularly and test the operations on the resources – both from within Apiary as well as from Postman. Build up a Postman collection with test requests for all operations.

The complete API should support these resources and operations:

- GET
  - List of departments
  - Department Details (incl count employees, salary sum)
  - Employees in department (sorted)

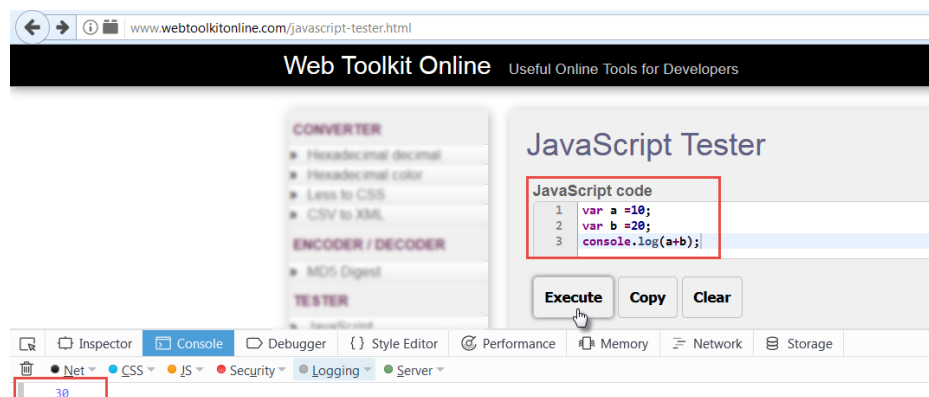


- Employee details (incl name manager)
- Employees (sorted, paging, filtered by *salary*, *job*, *name*)
- PUT
  - Update Employee
- POST
  - Create Department
  - Create Employee
- DELETE
  - Delete Employee

## 5. Introduction to JavaScript

JavaScript is a programming language like many others. It uses variables to hold values, loops to perform iterative operations (such as processing all elements in a collection), if-else constructs to perform operations under specific conditions. It can perform String manipulation and math operations on numeric values. JavaScript is used in for programs that run in a web browser – typically to manipulate the user interface and make programmatic calls to backend services – or that run on the server side and can do anything you want them to.

Open the site <http://www.webtoolkitonline.com/javascript-tester.html> - here you can enter small snippets of JavaScript and run them in the browser. Open the developer tools (Ctrl Shift I) and the console – to inspect the output of your programs.



Using the statement `console.log("some string")` you can write strings to the console output.

- a) Write and execute a snippet of JavaScript that prints five strings to the output – with an increasing number of asterisks (\*, \*\*, \*\*\* etc.)
- b) Write a snippet that flips a coin and returns *heads* or *tails* . Hint: the function Math.random() returns a random number between 0 and 1. To conditionally execute a section of code use if (condition) {...} else {...}
- c) Create an array with four string elements (from, me, to, you).

Print the second element of this array to the output.

Print the length of the array to the output.

Loop over all elements in the array and write their value in uppercase to the output.

Add the string “!” to the end of the array. Write all strings in the array to the output.

Add the strings “with” and “love” to the beginning of the array. Write all strings in the array to the output.

Change the second element in the array – “love” – to “greetings”. Again, write all strings to the output.

Sort the array alphabetically. Write the strings to the output.

- d) Create a function p that takes a string parameter s and writes the value of to the console. Write calls to function p to write the values 10 to 1 to the output.
- e) In addition to variables that hold scalar values, we can also use complex objects in Javascript. An object can be created in simple, straightforward manner. For example:

```
var person = {"name": "John Doe", "birthYear": 1969, "hobbies" : ["opera", "sailing"]};
```

Object properties can be accessed, modified and added:

```
console.log( person.name);  
person.hobbies.push("reading");  
person.salary = 5000;
```

JSON stands for JavaScript Object Notation. JSON is the string representation of JavaScript objects. You can write the string representation of an object to the console using:

```
JSON.stringify(person);
```

In addition to turning an object to a string, we can also parse a string to create an object from it:

```
var person = JSON.parse('{ "name": "Jane Doe", "hobbies": ["parasailing", "kick boxing"] }');  
console.log(person.name);
```

Construct an Employee object with properties name (Roy Rogers), job (CFO) and an array of responsibilities. Assign the object to a variable.

Add a property to the Employee for salary (6700). Change the job to Controller. Add 'Birthday presents' to the list of responsibilities.

Print the object's JSON representation to the output.

Assign the object's JSON representation to a string variable called empJson. Replace in this string Birthday with Christmas. Parse the contents of empJson into a new variable called emp2. Write the responsibilities of emp2 to the output and verify which presents this emp2 takes care of.

## 6. Introduction to Node

The Node platform supports running Server Side JavaScript Node applications. These applications can handle HTTP requests (as we expect from an API implementation), interact with the local file system and various backend systems including remote REST services and databases. Documentation for Node can be found here: <https://nodejs.org/api/>.

You will first install Node on your local environment. Subsequently, you will work your way from simple programs to somewhat complex applications.




Native versions of Node.js are available for Mac OS X, Windows and Linux. Go to <https://nodejs.org/en/download/> and download the installer for your platform.



### Downloads

Latest LTS Version: v6.11.3 (includes npm 3.10.10)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
 <b>Windows Installer</b> <small>node-v6.11.3-x64.msi</small>	 <b>Macintosh Installer</b> <small>node-v6.11.3.pkg</small>	 <b>Source Code</b> <small>node-v6.11.3.tar.gz</small>

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binaries (.tar.gz)

Linux Binaries (x86/x64)

Linux Binaries (ARM)

Source Code

32-bit		64-bit	
32-bit		64-bit	
64-bit			
64-bit			
32-bit		64-bit	
ARMv6	ARMv7		ARMv8
node-v6.11.3.tar.gz			

Next, run the installer to perform the installation. You can check on the success of the installation by opening a command line window or terminal and typing `node -v`. This should result in an indication of your current version of Node.js – which should be 6.x.y or higher (8.x is also available):

```
C:\Users\lucas_j>node -v
v6.9.4
```

Editing Node programs can be done in any text editor. Some editors are 'Node aware' – that means they can format Node code, check the syntax of the code and even run and debug the code. An example is Visual Studio Code from Microsoft – the tool that you installed and used earlier in this training.

A Node.js application is a JavaScript application (more formally: ECMA Script v6 or ES6) that can be run - outside the browser – on the Node.js platform [which contains the V8 engine for JavaScript]. JavaScript is one of the most popular programming languages around – used in virtually any web site, web application and in many apps and APIs. The number of resources on JavaScript is huge and growing. Popular sites for introductions and references to JavaScript include <https://www.w3schools.com/js/default.asp> and <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference> - it may be useful to at least create bookmarks for these sites in your browser.

A node application – the informal way of referring to Node.js applications – is simply run on the command line using the command:

### **node app.js**

where app.js is the presumed name of the (main) application script. The name is yours to decide. Other common names include main.js and of course hello-world.js.

In this section, you will edit, create and run a few simple Node applications.

- a) Open a command line (terminal) in the part1-hello-world directory of the workshop resources.  
Run

```
node hello-world.js
```

and see what happens. (well, the expected of course!). Open the file hello-world.js. No big surprises?

- b) Most Node programs will be more structured than just a bunch of lines of code. Creating functions is a level of structuring found in almost all Node programs. The hello-world-2.js application does the same thing as hello-world.js, in a more structured way.

```
node hello-world2.js
```

- c) Run hello-world-3.js and pass a command line parameter. For example:

```
node hello-world-3.js Johan
```

See what the result is. Inspect the code in hello-word-3.js. You will see how node applications

can get access to the command line parameters and how functions can be used to organize program logic. Again, no big surprises.

- d) Now also check `hello-world-4.js`. This application has the same functionality as the previous one. The interesting thing is that a [reference to a] function is assigned to a variable. And that this reference is leveraged to invoke the function. Passing around references to functions is not necessarily an everyday affair in all program languages you may have encountered.

Open `hello-world-5.js`. The functionality is still the same as the previous two programs. New is the use of function *reception*. This function is the unit where greeting of visitors is performed. The reception is strictly instructed as to how to perform the greeting. In fact, the function it should use for greeting is injected. Another example of passing around a reference to a function.

`hello-world-6.js` shows to we can use anonymous functions instead of named functions. Just so you know.

- e) Scheduling functions for delayed and possibly periodic execution is easily done in JavaScript and therefore in Node applications. Using `setTimeout()` and `setInterval()`, a reference to function is associated with a time period expressed in milliseconds to schedule the execution of the function. Check out `hello-world-7a.js` for examples.

```
node hello-world-7a.js
```

Notice how you may think that the program is complete – when in fact it is still running.

When the scheduled function has associated data – input parameters that are used in its execution – it is not trivial to pass that data context into the function. A naïve approach is shown in `hello-world-7b.js` – and hopelessly fails.

- f) Program `hello-world-8.js` deserves special attention. It shows how we not only can pass around references to functions but to the combination of a function and its [data] context. This package – function plus context – is called a *closure*. And that is one of the hot concepts in programming in recent times.

Try:

```
node hello-world-8.js kwik kwek kwak
```

and see the outcome.

Now take (another) look at the code. In the `args.forEach()` call, for each program argument there is a function registered through `setTimeout` to be executed at a later point in time. The function

to be executed – when the timeout happens – is returned from `getGreeter()`. For `forEach()` invokes `getGreeter()` to return a function and registers that function with `setTimeout()`. Note however that `getGreeter()` does not just return a function: it returns a function with a reference to the local `toGreet` variable. And here we have the closure: the combination of the function – a reference to the function defined in `var g` – and its context: the variable `toGreet` at the time of returning the capture.

`getGreeter()` is invoked for each command line argument. This results in a distinct capture for each of these arguments, with distinct values for the `toGreet` variable, part of the closure.

- g) The program `hello-world-8a.js` does the same thing as `hello-world-8.js`, except that the code is organized differently. The program leverages a module, referred to as *greeter* and imported from the file `greeter-module.js`.

Check how the functionality in the file `greeter-module.js` is exposed – and what is not exposed. Verify how in `hello-world-8a.js` we get access to what is exposed by the module.

Remove the call to the `localPrivateFunction()`. Add a function `welcomeAll()` to the `greeter-module.js` file. Have this function write a single line to the console, to welcome everyone: 'Welcome y'all!'. Expose this function from the module.

Invoke this new function `welcomeAll()` to `hello_world-8a.js`.

- h) A small additional step: `hello-world-9.js` adds our first core Node module. By adding the first line in the program:

```
var util = require('util');
```

we can now make use of the functionality in the core module *util* in the program. Examples are: `util.log` and `util.format`.

See documentation about this core module: <https://nodejs.org/dist/latest-v6.x/docs/api/util.html> .

- i) File manipulation can easily be done in Node.js applications. The program `file-writer.js` is an example: it writes a file with all command line arguments on separate lines in the file. And it does so in very few lines of code – just check out: `file-writer.js`.

Also run this program and inspect the file that gets created:



```
node file-writer.js kwik kwek kwak
```

- j) Debugging – inspecting the execution of a Node program including call stack, values of variables and flow through the program – can be done with the debugger in Visual Studio Code (see <https://code.visualstudio.com/docs/editor/debugging> ) or with the built in debugger (see <https://nodejs.org/api/debugger.html>) .

Using the debug option in Visual Studio Code, debug the file-writer.js program. Inspect the value of variable val for each iteration, by placing a breakpoint on line 8 (where fs.appendFileSync takes place).

## 7. Handling HTTP Requests

One of the key strengths of Node.js is its ability to help out with all kinds of HTTP(S) interactions. In the next section we will make use of a package (Express) that adds very convenient capabilities on top of core Node.js. For now, we will stick to the core functionality available out of the box.

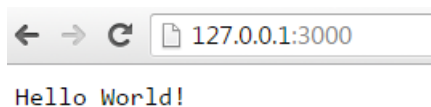
The sources discussed in this section are in directory part2-http.

- a) The first program creates an HTTP Web Server that listens for HTTP requests on port 3000 and returns a static response to any request. You can test this server by calling <http://127.0.0.1:3000> from your browser, CURL, wget, SoapUI, Postman or any other HTTP speaking tool.

Run:

```
node http.js
```

and access the URL as described above.



This may sound daunting: creating an HTTP server. Wow. That must be quite a lot of work. And with Node.js of course it is not. Inspect the code in http.js. Note how the core module http is used – require is similar to import.

Change the response to: Bonjour Monde!

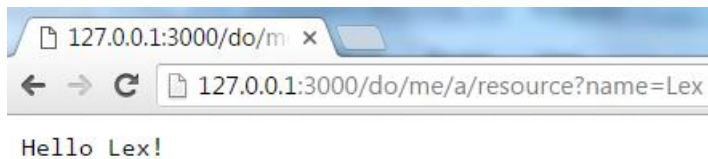
- b) With this first major step to easily made, we may get greedy. What about processing query parameters in the URL – the HTTP equivalent to command line parameters? Easy. And we will tackle URL paths at the same time.

Run http-2.js:

```
node http-2.js
```

and access the url: <http://127.0.0.1:3000/do/me/a/resource?name=Lex>

Check the HTTP response:



And check in the command line (terminal) window:

```
C:\data\sigs-node-js-31march2016\sigs-nodejs-amis-2016\part2-http>node http-2.js
server running on port 3000
URL /do/me/a/resource?name=Lex
path: /do/me/a/resource
queryObject: {"name":"Lex"}
URL /favicon.ico
path: /favicon.ico
queryObject: {}
```

We can see how this node application could easily learn about both query parameters and URL path.

Now open http-2.js in a text editor and analyze the code – not there is a lot of it.

- c) The program http-3.js will return the file index.html in the public folder for any HTTP request to port 3000. Try it out.

Note how core module fs is leveraged to read the file, alongside core module http to handle the http request. Also note that any HTTP request sent to port 3000 will get this same response – regardless of URL path, query parameters and HTTP method.

- d) Program http-4.js also returns the same response to any HTTP request. However, instead of returning the contents from a local file as the response, it goes out to fetch a resource from the internet. Just run http-4.js:

```
node http-4.js
```

and access port 3000 on the local host.

Note: it takes quite a while to load this document. If you add a log statement to the program to log each request – and better yet: the URL for each request – you would understand why this takes so much longer than expected – and why the page looks so much poorer than expected. Hint: `console.log(req.url);`

## 8. Express Web Application Framework

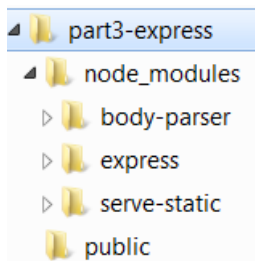
Many Node.js applications solve a similar challenge: handling HTTP requests. This happens for example for static file serving, rich client web applications and for the implementation of REST APIs. Although the core capabilities in Node.js for dealing with HTTP interactions, with the Express framework developers get even better facilities for constructing Node.js applications that handle HTTP requests.

In this section, we will get introduced to Express. We will use the resources in folder part3-express.

- a) As a first step, you need to install a few packages that we will use on top of core Node.js. On the command line (in the terminal window), in the part3-express directory, please use npm to install three packages, like this – or skip to the section just below the next figure:

```
npm install express
npm install body-parser
npm install serve-static
```

After you have executed these three installation steps, verify that a subfolder node\_modules was created. Inspect the contents of this folder.



Alternatively – and now I tell you – you could have simply used:

```
npm install
```

The npm tool would have inspected the package.json file in your current directory, would have found three dependencies and installed those dependencies. Note that you can use npm install in combination with a package.json file at any time to refresh the dependencies and bring in the latest (allowed) versions of these packages.

- b) The simplest web application we can run with Express is defined in express.js. Run this program and open URL <http://127.0.0.1:3000>. A static response is returned.

express-2.js does exactly the same, in an even more compact manner. Check out the source.

- c) The most compactly code web server you have ever seen: open express-3.js and behold! Run express-3.js and access the same URL: <http://127.0.0.1:3000> from the browser.

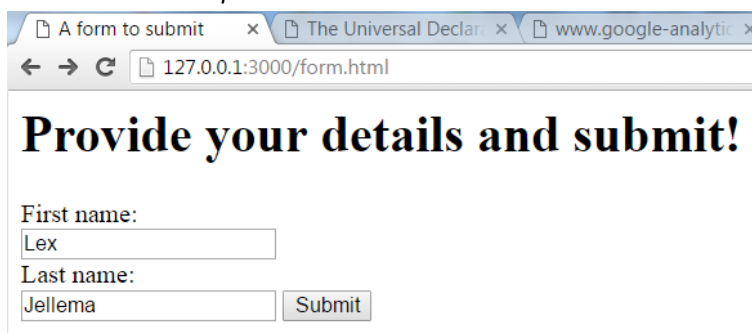
Note how the browser shows an image – also fetched from the Node.js application. Click on the download link for the PDF document. This document too is returned by the Node.js application.

Where do these resources come from?

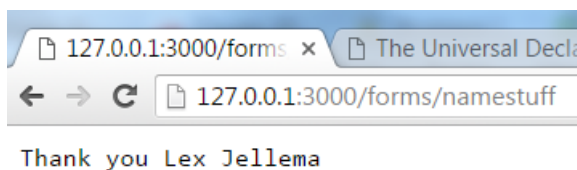
- d) Inspect the sources for express-4.js. Here we handle a form submission. This source more or less belongs together with the HTML source form.html in directory public. Check out that file. try to understand how the form and the node application fit together.

Now run express-4.js and access <http://127.0.0.1:3000> from the browser.

Click on the link *Open Form*. Fill in the form fields and click on Submit.



You will get thanked for your submission; the thank you messages uses the values you had submitted from the form.



- e) REST APIs are an important area for Node.js. The application express-5.js is an example of a simple REST API. Run this application.

Then access the URL <http://127.0.0.1:3000/departments> . You will get a JSON response with a list of departments.

Pick one of the department identifier values from the and access URL: [http://127.0.0.1:3000/departments/the\\_value\\_you\\_picked](http://127.0.0.1:3000/departments/the_value_you_picked) (for example <http://127.0.0.1:3000/departments/10>). You will get a detail response: a single department record.

The application behind this REST API is about 20 lines long. That is it. 20 lines. That is a good time to say: wow!

Open express-5.js and analyze the code.

The data exposed by this API is loaded from a static file on the local file system into a variable *departments*, when the application is started. Handler functions are registered with Express for GET requests to the URL paths /departments and /departments/:departmentId. The functions perform simple yet effective actions in response to requests.

- f) Try to extend express-5.js in the following way: GET requests for the URL path /time should return a response with the current time. Hint: a string representation of the current time can be created using `new Date().toUTCString()`.
- g) Program express-6.js has maybe six lines more than express-5.js. Just so you know. Now run express-6.js.

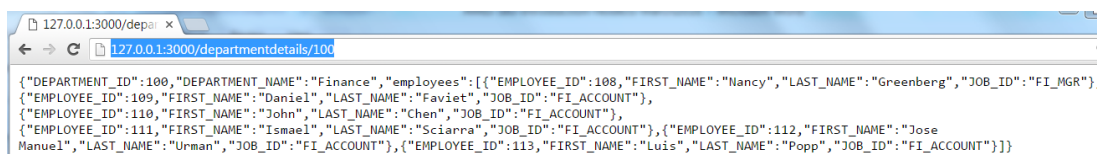
List all departments using: <http://127.0.0.1:3000/departments>.

Now open the department form: <http://127.0.0.1:3000/departmentForm.html> . Enter details for a new department and press submit.

Next, list all departments again, using <http://127.0.0.1:3000/departments> . You should see the new department added to the collection.

This is a very simple example of a REST API that not only supports GET but POST as well. Inspect the source of express-6.js to see how this is done.

- h) Express-7.js adds the departmentdetails resource – accessible at [http://127.0.0.1:3000/departmentdetails/DEPARTMENT\\_ID](http://127.0.0.1:3000/departmentdetails/DEPARTMENT_ID), for example <http://127.0.0.1:3000/departmentdetails/100>. Run express-7.js and try out this department details resource.



Then inspect the source for express-7.js. Where do the data for *departmentdetails* come from?

## 9. Calling out to REST APIs from Node Applications

The last practice in the previous section had you look at a Node program that not only handled incoming HTTP requests but also called out to HTTP endpoints itself. The data in this practice is retrieved from an external API. So here we have an example of a Node.js application making an external HTTP(S) call. The most interesting bits:

- see how the request details (host, port, path and method) are configured
- see how the response to this request is handled in callback function that receives the response and how event listeners are registered for the data, end and error events
- see how the final response is created (`res.send(JavaScript record)`) in the *end* event handler

Calling out to backend systems – beyond just the local file system – is obviously an important thing to do in Node programs. Frequently, Node applications implement APIs that receive REST calls that are to be handled by interacting with one or more backend systems. Calling out to REST APIs, interacting with databases, invoking (SOAP/XML) Web Services, communicating over various protocols with message brokers, event busses etc. is all in day's work for many Node programs and therefore for Node programmers – such as yourself.

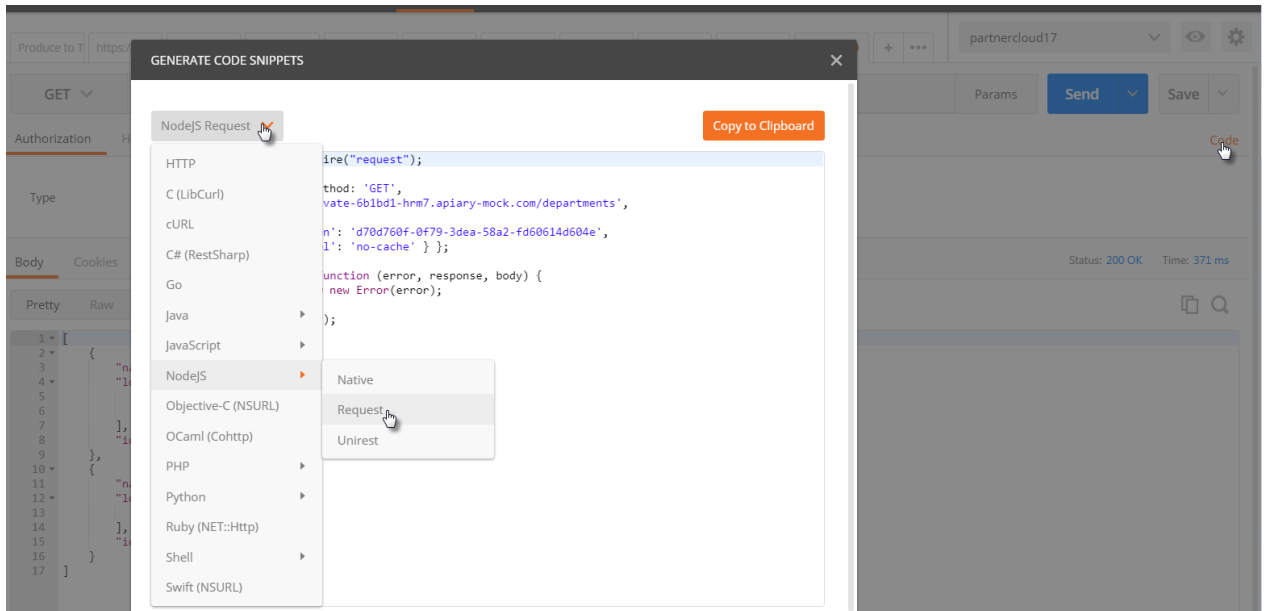
In this section, we will go over a number of examples of HTTP callouts from Node. In the next section we will implement a simple API that makes multiple callouts and combines the results in its own response.

Note: various ways for making HTTP requests from Node are available – using native Node mechanisms or through additional Node (npm) modules. Which approach is best for you is not always easy to determine. For now, let's work with the *request* module – but know that it is but one of many options.

- a) Open Postman and load the collection you have created in section 4 to test – the mock instance of – your HRM API. Locate the request for retrieving all departments.

Click on the Code link. Open the dropdown with technologies. Select NodeJS | Request (or Native).





A snippet Node code is shown by Postman. Copy the code to the clipboard, create a new file in Visual Studio Code and paste the code into that file.

Run the file. A call should be made to your API – the mock implementation provided by Apiary. The results should be written to the console output. If you make a change in the example response message in the API Design in Apiary and make the call again, a different response will be received and written to the output.

- b) Extend the simple Node program, to make it write to the console [only] the names of the departments retrieved from the API call.

In order to do so, you need to parse the response received from the API – turn text into a JavaScript object (hint: `JSON.parse()`) – and iterate over the array of department objects found inside the response.

Now also write the location of the department to the console.

- c) In section 2 – you took a brief look at a public API called Genderize.io (<https://genderize.io/>) . This API receives a request with one or more first names and responds with an indication of the likely gender for each of those names. In this practice, leverage that API in doing the following in a new Node application (hint: you can use the code in Postman for calling this API as a starting point):
  - a. read a file that contains first names, each on its own new line
  - b. determine the gender for each first name using the Genderize.io API (using as few API calls as possible)

- c. write a file with on each line a first name, the likely gender, the likelihood of that prediction and the number of occurrences used to make the prediction
- d) Bonus: This site offers a service to add text to voice functionality to a website: <https://responsivevoice.org/> . In the background, it uses HTTP Requests like this one to get audio translations for English sentences:  
<https://code.responsivevoice.org/getvoice.php?t=Some%20message%20from%20me%20to%20you%20on%20a%20beautiful%20morning&tl=en-GB&pitch=0.5&rate=0.5&vol=1>

If you click on this URL, you should be greeted by a friendly female voice.

Your challenge: Create a Node program that uses this API to create an MP3 file containing the audio for a piece of text. As a next step, you can read the English text to turn to audio from a file and or solicit user input. Note: multiple sound bites returned from the API can be merged together into a single MP3 fragment.

Other services – Google, IBM Watson, Azure (<https://docs.microsoft.com/en-us/azure/cognitive-services/speech/api-reference-rest/bingvoiceoutput>) ... - offer similar functionality. Additionally, several services are available to turn speech (audio snippets) into text. Another interesting area to investigate...

## 10. Complex REST API for retrieving rich Artist information

The REST API that we exposed in the previous section was quite straightforward. Information retrieved from a static local file, doing a little manipulation. It was a fine start. In this section, we make it a little bit more interesting. We will work on an API that returns a JSON document in return to an HTTP GET request that specifies the name of a particular artist. This JSON document contains details on the artist – such as a genre label, a biography, a list of the most recently released albums with their details and more. To gather this information, the Node application that exposes this API has to go out and fetch data from external services such as the Spotify API (<https://api.spotify.com/v1>).

The situation can be described like this:



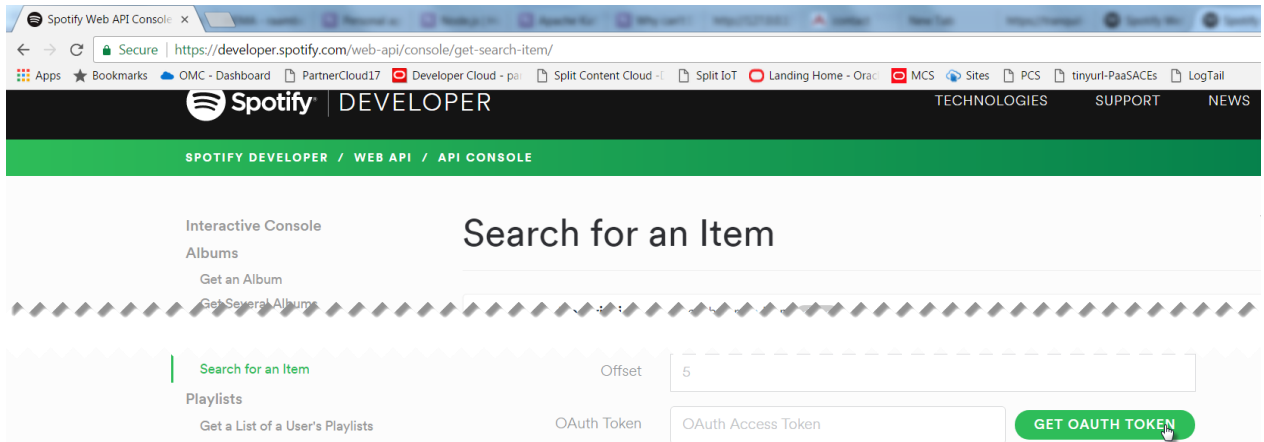
We will build up this API in a number of steps. You will find the resources in folder `part5-artist-api`.

- Open the command line window in folder `part5-artist-api`. Before the application can be started, you need to use `npm` to install a few packages: `express`, `request`, `body-parser` and `async`.

```
npm install express request body-parser async
```

- You need an authorization token to access the Spotify API. Go to <https://developer.spotify.com/web-api/console/get-search-item/>.

Click on Get OAuth Token.



Copy the value that appears in the field OAuth token to the clipboard. Open file artist-enricher-api-lib.js and paste the token into the variable *token*:

```

artist-enricher-api-lib.js x greeter-module.js artist-enricher-api-1.js module.js artist-
1 var artistEnricherApiLib = module.exports;
2
3 // please go to https://developer.spotify.com/web-api/console/get-search-item/
4 artistEnricherApiLib.token = 'BQBa7jyzpHoxrcBmhkKEdf5T5HUKQApPSnhdfRS0dEpVul';
5 artistEnricherApiLib.spotifyAPI = 'https://api.spotify.com/v1';

```

- c) Run the artist-enricher-api-1.js program. It listens to HTTP requests of the form <http://127.0.0.1:5100/artists/get?artist=artistName> (such as <http://127.0.0.1:5100/artists/get?artist=b52s> or <http://127.0.0.1:5100/artists/get?artist=u2>) and similar calls of the form [127.0.0.1:5100/artists/artistName](http://127.0.0.1:5100/artists/artistName) such as [127.0.0.1:5100/artists/u2](http://127.0.0.1:5100/artists/u2). Try out such a call from your browser or using CURL.
- d) Inspect the code in artist-enricher-api-1.js. The familiar Express style configuration of a web server to handle GET requests can be found, for three different URL paths. The configuration of a callback function for the URL path /artists/\* that hands off the work to function handleArtists() while passing the value of the *artist* query parameter and a similar one for /artists?artist=\* and the root path / .

Also check the function handleArtists() that gathers data from the external API and then calls the function composeArtisResponse () to return the response. Check how the Spotify API is invoked using the request() function and how the response is processed.

- e) Stop artist-enricher-api-1.js and run artist-enricher-api-2.js. Make the same HTTP GET call as before. You will now see – after a somewhat longer waiting time - additional information in the response: details for the albums released by our artist. Spotify offers an API that provides a list of albums – maximum of 50 per call. For each album, we get the name and an image URL for the cover image. We do not get a release date.

Inspect the code. A small section is added to invoke Spotify a second time, to fetch details about albums. One thing you could notice is that the code slightly becomes harder to read – additional indentation for each consecutive outbound call and callback function to handle the response. The call to fetch album information uses information from the first call, so in this case it is justified that the two calls take place sequentially instead of in parallel. However, sometimes multiple callouts can be done at the same time because they are mutually independent. Waiting for the two response at the same time makes the overall wait shorter.

The handling of asynchronous callbacks and the coordination of parallel calls can be quite complex. We will next look at package `async` that helps bringing some order to that chaos.

- f) Package `node-async` (<https://github.com/caolan/async> and installable with `npm install async`) is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. It can be used in `node.js` applications as well as in client side JavaScript running in a web browser. It makes it quite easy for example to perform multiple asynchronous activities in parallel and work with the combined result from all these activities.

We will make use of `async` to orchestrate the two outbound REST API calls we make to Spotify. Notice how `async.waterfall` is used to organize the sequential calls to the Spotify API. The functionality is not spectacular – control of the program flow and readability of the code increases notably. Note how `callback()` is used to indicate the completion of *unit* in `async` waterfall. Also note how `callback(null, artist.spotifyId)`; is used to specify that no error occurred (the first parameter is null) and also to pass a result from this unit to the next: `artist.spotifyId` is passed in as the first input argument to the next unit in the waterfall. (through the shared context variable `artist` this same value is accessible using `artist.spotifyId`).

Run `artist-enricher-api-3.js` and make one or more API calls from your browser, CURL, Postman, SoapUI or whatever your favorite tool is.

- g) Spotify offers yet another API that allows us to retrieve details for albums – primarily the release date. In `artist-enricher-api-4.js`, you will find the waterfall extended with a third unit. This third unit calls the albums details API on Spotify. This can be done for up to 15 albums at one time. Because the second call to Spotify in the waterfall results in a list of up to 50 albums, we need multiple calls to the album details API. Of course these calls should not be made sequentially. In the source code for `artist-enricher-api-4.js` you will see how a special `async` mechanism is used: `async.forEachOf`. The `forEachOf` function is handed an array – in this case an array of album arrays. For each element in the array, `forEachOf` will execute the function. When all function calls for the elements in the array have signaled their completion – through the `callback()` call – the completion function in `forEachOf` is executed to do any final processing of the joint results. In this example, all that needs to be done is another call to `callback()` to tell `async.waterfall` that this

unit is complete.

Run `artist-enricher-api-4.js`, make one or more calls and verify that the release date is now added for each album. You can try to vary the size of the chunk – the number of albums in one array – to see if that impacts the processing time. The smaller the chunk, the larger the number of [parallel] calls to Spotify.

- h) Bonus: for the truly adventurous, there are several ways to further explore this API:
  - a. use the APIs from iTunes  
(<https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/>), MusicBrainz  
([https://wiki.musicbrainz.org/Development/JSON\\_Web\\_Service](https://wiki.musicbrainz.org/Development/JSON_Web_Service)), MusicGraph  
(<https://developer.musicgraph.com/>) to retrieve additional data on artists and their work
  - b. implement some form of web scraping to retrieve Artist biographical data from Wikipedia
  - c. Embrace the still fairly new notion of Promises to replace the Callbacks (see for example <https://technology.amis.nl/2017/05/18/sequential-asynchronous-calls-in-node-js-using-callbacks-async-and-es6-promises/>); this also means dropping module `asynch` and relying on more native mechanisms in ES6.

## 11. Implementation of Your HRM API

You now have the tools and skills to start implementation of the HRM API you designed earlier on. Proceed in a step by step – resource and operation by resource and operation fashion.

- a) Use npm init to create a Node application – hrm-api - that can handle HTTP requests. Add Express, using `npm install express -save`
- b) Implement the logic to handle a GET request to the *departments* resource. Return a mock response – hard coded in your JavaScript code; use the same mock response that you defined the API design.
- c) Run the application. Invoke the HRM API's /departments resource from a browser. Make a call to your local API from Postman – using the Postman collection that you created for testing the API Design.
- d) Extend the application with code to handle a GET request for a specific department: probably something like /departments/{department identifier} . Start with returning the hard coded response messages that you specified in the API Design in Apiary.

Run the application. Try out the new endpoint & resource from your browser and from Postman.

- e) Implement the POST request to add a department in the hrm-api application.

Perform a validation on the body and when the message contains a valid department definition, then add the new department to the in memory collection and return a proper response message (possibly with the assigned identifier and/or the location to the newly created resource); verify that with a subsequent call to get departments, the new department can be retrieved and the same with get department details for the new department. In case the body sent with the POST request does not contain a valid department object, then return an appropriate response – status and error message.

Test the POST operation to create new departments from Postman.

- f) Instead of hard coding the response messages – quite silly – it is time for the next level: using local files as the store for departments and employees. Use files `departments.json` and `employees.json` to read departments and employees from and to write them to after processing a change. Modify the Node hrm-api application – make it read in these two files when the

application starts – and use them for constructing responses instead of hard coded data. Whenever a change is made to a department or employee resource – the relevant file(s) is (are) rewritten.

Note: we will ignore concurrent transactions for now and pretend each request is in perfect isolation.

Run the application. Test if the data returned from the HRM API is indeed the data from the files. Run the POST request from Postman to create one or more new departments. Stop the application. Start the application again. Verify if the newly added departments show up in the list of departments. If they are, you have implemented (simplistic) persistence.

- g) Implement the API operations for the Employees resource – GET, POST, PUT, DELETE – and make sure that changes are persisted to the file(s).

Test the API operations from Postman.



## 12. Publish your API

APIs are not much use if are only locally available on your laptop. Most APIs will need to be accessed from farther away – mobile apps running on devices anywhere in the world, servers running in any data center or even the laptop of the person sitting next to you.

There are several ways to share your (running) API with others. You can open up your laptop itself and allow connections to be made to it and you can deploy the API – make it run – on a server that is already accessible, for example in the cloud. In this practice you will do both.

- a) Use ngrok to make the API publicly available: see <https://technology.amis.nl/2016/12/07/publicly-exposing-a-local-service-to-nearby-and-far-away-consumer-on-the-internet-using-ngrok/>.

In summary: with ngrok, you install a local agent that communicates with a cloud service, asking that service whether requests have arrived for the local agent to handle. The agent will take such requests, execute them locally – call your API at localhost or 127.0.0.1 – and send the response to the cloud service. The cloud service will reply with that response to the original caller. The cloud service will assign a public URL to represent the local agent (who in turn represents your API) and anyone who wants to call the API can send the request to that URL.

Have one of your fellow students make calls to your API.

- b) Instead of exposing the locally running API to the world through a cloud intermediary, you can also run the API on a public cloud. Running Node applications on a cloud somewhere is very simple to do and is free – up to a certain level of activity. Publish your API on the *now* cloud service: see the tutorial in this article: <https://technology.amis.nl/2017/09/01/rapid-and-free-cloud-deployment-of-node-applications-and-docker-containers-with-now-by-zeit/>. Again, have one of your fellow students make calls to your API

### 13. API with external backend – 3<sup>rd</sup> party REST API

APIs usually do not have stand-alone implementations. The API handles incoming HTTP requests and interacts with external back end systems to retrieve data and persist data. APIs are themselves typically stateless – they may cache data but do not hold data for long periods and they are themselves not typically the final source of truth for data.

We have a third party HRM API at our disposal – that we should implement our API against. This means that the functionality offered by this API to retrieve employees and departments and to create, update and delete employees can be and should be leveraged. In other words: you have to change the implementation of your API – and add calls to this API.

One of the endpoints that this API offers:

[https://129.150.91.133/soa-infra/resources/default/HRM\\_REST\\_API/DeptEmpAPI/employees](https://129.150.91.133/soa-infra/resources/default/HRM_REST_API/DeptEmpAPI/employees)

A GET request to this endpoint will result in a JSON document with a list of employees.

- a) Change your API implementation; add a call to this end point. Ensure that a call to your API will return the Employees returned by this 3<sup>rd</sup> party API.
- b) The GitHub repo with lab materials contains a folder called 3rdparty-hrm-api. This folder contains a Postman Collection: HRM\_REST\_API.postman\_collection.json. Load this collection in Postman – and verify that you can execute all requests successfully.
- c) Implement support for the creation of new employees in your API using the matching operation in the third party API ([https://129.150.91.133/soa-infra/resources/default/HRM\\_REST\\_API/HRMRestAPI/employees](https://129.150.91.133/soa-infra/resources/default/HRM_REST_API/HRMRestAPI/employees) )
- d) Demonstrate your API as it currently stands to your fellow students and your trainer.

## Bonus: Miscellaneous and Advanced Topics

In this section, we will look at a number topics, some which advanced. These include:

- sending emails from node.js applications
- prompting command line input from users
- working with Server Sent Events to push data from the Node.js server application to a browser client application
- use of new language features: Promises and Generators

Check the scripts in folder part6-miscellaneous.

### Sending Emails

Sending emails is one of many operations that are easily performed from Node using one of many modules available for Node. File `emailer.js` creates a (custom) module – leveraging the `emailjs` module for sending emails. In `mailclient.js`, this custom module is used to send an email.

- a) Run `npm install` in folder `part6-miscellaneous\email`, to install the `emailjs` node module.
- b) Open file `emailer.js`. See how the module `emailerAPI` is exposed from this file. Modify the file with your settings for email account credentials and mail server.
- c) Open file `mailclient.js`. See how the (local, custom) module `emailerAPI` is required. Update the subject, body and addressees – to send your own personalized email message to your own addressee of choice. Then run the `mailclient`:

```
node mailclient
```

Check if the email was sent (and received) successfully.

### Prompting input from users

Node is frequently used as backend for web applications. Any input required is gathered through the web interface in the browser. However, Node applications will frequently run as server side programs without this browser based outlet. In this case too they can get user input, for example just on the command line. Node module `prompt` (<https://www.npmjs.com/package/prompt>) can be used for this, as is seen in the code sample in directory `part6-miscellaneous\prompting`.

- a) Open command line in directory `part6-miscellaneous\prompting`
- b) Run `npm install` to download node module `prompt`
- c) Run sample program:

```
node index.js
```

Provide the requested input and see how it is processed.

Type *exit* to end the dialog.

- d) Inspect the – very simple – code in index.js. If you want to experiment with various settings and options, such as validations or colors in the command line dialog, feel free to do so – using the documentation at <https://www.npmjs.com/package/prompt>.

## Server Sent Events to Push Messages to Web Clients

Node can push messages to browser clients, using WebSockets and the simpler Server Sent Events (SSE) mechanism. We will use the latter in this next lab.

- a) Open command line in directory part6-miscellaneous\server-push.
- b) Run `npm install` to get hold of required node modules (body-parser, serve-static, prompt and express).
- c) Inspect file app.js. This file serves a static file – index.html – and exposes a single endpoint - /updates – where SSE clients can register.

File index.html contains a JavaScript fragment, where the client registers with the SSE Server – at the /updates endpoint. A listener is associated with this SSE connection – to read the consumed message and update the UI subsequently, in the plainest way imaginable.

Back to app.js: it uses the prompt module to solicit input from the user on the command line. Every piece of input is published as server sent event to all SSE clients, to be published in the browser UI.

You can look at file sse.js for some more details on how the SSE clients and topics internals are implemented. You will notice that there is not too much to it.

- d) Run app.js on the command line:

```
node app.js
```

And open your browser at `http://127.0.0.1:3000/`.

Type some input on the command line. This input should appear in the browser.

Open a second browser window at the same URL. Type some more input on the command line.

This input should appear in all browsers.

## Callbacks and Promises

Node (and JavaScript in general) makes heavy use of asynchronous mechanisms. Many operations are performed asynchronously with regard to the invoker and will inform the invoker about the result of the action through a callback. Many calls to functions in Node programs will carry a reference to a callback function – the function that should be invoked to asynchronously return the result of the operation.

- a) Navigate to directory part6-miscellaneous\promises. Open the file app.js. Here is an example of an asynchronous execution. Function doStuff is called – it will do its thing (including a 1.5 second wait) and then callback to report its outcome. The call to doStuff has to provide a callback function in order to be notified of the result from doStuff.

Run app.js: `node app.js`

You will see that the last line in the program – is executed prior to the execution of the callback function. This is not uncommon in Node programs: what appears to be the last line is in fact not the last action taken. Note how the `setTimeout()` built in timer also takes a callback function: the function to be executed when the time is up.

- b) Open app2.js. Timing is added, to check on the total execution time. Run app2.js.
- c) In app3.js, the variable `numberOfLoops` dictates how many times and for which values the `doStuff()` function gets executed. In a sequential for loop, each call to `doStuff()` is made. And here we see one of the powers of asynchronous operations: the calls to `doStuff()` are made sequentially but since we do not wait for `doStuff()` to complete its work before making the next iteration in the loop, the calls are virtually parallel. Instead of waiting five subsequent times for a call to `doStuff()` to be completed – which would take 5 times 1.5 seconds or a little over 7.5 seconds – we have to wait less than half that time. Or do we?

Run app3.js.

Check if the results are reported on the console in the expected order – step 0..step5. If they are not, how can that be?

- d) In app4.js an array called `results` is introduced to try to capture the results from the calls to `doStuff()`. Check out the logic in app4 and consider whether this will do the job.

Next, run app4.js and verify whether our objective – end up with an array with all results

produced by `doStuff()` collected. We will get back to this.

- e) It is quite common to take the result from a call to a function and use that result in a subsequent call to another function – and use the result of that call to invoke yet another function. With asynchronous functions (or functions that return their result in a callback) that is not so trivial as in a purely synchronous programming model is the case. Take a look at `app5.js`. Here, the result from `doStuff()` is passed to `doAdditionalStuff()` and that outcome to `doMoreStuff()`. Each of these functions returns its result asynchronously. Therefore, we end up with a number of nested callback functions.

Run `app5.js` to see what it produces.

Having to work with nested callback functions is one of the consequences of the JavaScript programming model. And it can be the cause of programs of which the logic is hard to track and that are therefore difficult to create and certainly to maintain. Organizing the work properly – ensuring that everything happens in parallel when that is possible and is orchestrated in the proper sequence when that is required – is not a simple thing.

We will see two ways in which this “callback hell” as it is sometimes referred to can be addressed.

- f) On the command line, enter

```
npm install
```

to have node module `async` (for documentation check out <http://caolan.github.io/async/> ) installed. This module can help bring simplicity to the asynchronous programming model.

Open `app6.js`. Compare `app6.js` to `app4.js`. The two are very similar. Note that the for loop that drove the calls to `doStuff()` in `app5` and previous versions of the application is no longer there. Instead, the `async.forEachOf` operation from the `async` module organizes the parallel calls, driven by the `steps` array. This array is initialized with *numberOfLoops* elements that are set using the `fill` method and subsequently handed to the `map()` method (see [https://www.w3schools.com/jsref/jsref\\_map.asp](https://www.w3schools.com/jsref/jsref_map.asp) ). In the `map()` method, the value for each element in the array is mapped to the desired value which is in this case just the index of the array element.

When all calls have returned their asynchronous result, the function passed to `async.forEachOf` as second parameter is invoked to wrap up the whole affair.

Now run `app6.js`.

The results are gathered into the result array correctly this time. The evaluation of the value of `i` – the index variable into the results array – is done in the proper context and at the right time. Note that the total execution time is now written to the console only once.

- g) Take a look at `app11.js`. This program shows the usage of the waterfall operation in `async` – used to chain together asynchronous operations that should be executed subsequently and that may take input from their predecessor.

Run `app11.js`. Check the final outcome. Try to determine how that outcome was arrived at.

- h) In `app7.js`, the nested callbacks from `app5.js` have been reimplemented using the waterfall mechanism in `async`. Instead of nested callback, in the waterfall construct we specify a number of subsequent asynchronous calls to make. Each step concludes with a call to the waterfall callback function, passing the outcome of the step and in doing so making that outcome available for the next step in the waterfall.

Run `app7.js`.

Try to understand the flow through the program. Verify whether the result produced by three subsequent asynchronous function calls taking place in parallel for multiple values in the original `steps` array is gathered correctly in the `results` array. Hopefully through the use of `async`, the nested callbacks are no longer quite as bad as before.

- i) Add logic in `app7.js`: a fourth function should be created – along the lines of `doStuff()` and companions – to add the string *'For starters, then '* at the beginning of the parameter passed in. This function should be called last in the `async.waterfall` sequence. The result for Step 0 should then become: *"For starters, then STEP 0 Enriched! (The finishing touch)"*.
- j) One of the recent evolutions in JavaScript (ES6 and beyond) has been the introduction of the Promise. See <https://strongloop.com/strongblog/promises-in-node-js-an-alternative-to-callbacks/> for background. Using Promises, even more than with module `async`, asynchronous calls can be dealt with in a way that almost looks and feels synchronous or at least sequential.

In `app9.js`, Promises have been used to implement the nested series of asynchronous calls that are needed for Step 1, similar to `async.waterfall()` in `app7.js`. Run `app9.js`, verify the outcome, and try to understand the logic.

- k) Next, in `app10.js`, the logic from `app7.js` has been reimplemented using Promises (`Promise.all` instead of `async.forEachOf`). The code is compact. Try to understand its logic.

Run app10.js. Verify the results produced.

- I) Just as before, extend app10.js: a fourth function should be created – along the lines of doStuff() and companions – to add the string *'For starters, then '* at the beginning of the parameter passed in. This function should be called last – this time in the promise chain. The result for Step 0 should then become: *"For starters, then STEP 0 Enriched! (The finishing touch)"*.

Program app12.js is the Promise based counterpart to program app11.js. See how much more compact the programming is using Promises than with async – which is already way better than nested callbacks!

Run app12.js and verify that it produces the required result.