Trying Application Programming Interface (API) without coding experience:

Kick start with Europeana APIs with PHP

Go Sugimoto
Austrian Centre for Digital Humanities
Austrian Academy of Sciences
go.sugimoto@oeaw.ac.at

The Original

This is the original version of an online tutorial in the Programming Historian: Introduction to Populating a Website with API Data. Although the core content is the same, there have been some big changes, including the title. This version is slightly longer, more informally written, and has more explanations and related information. It would be useful for the readers who prefer more casual version of the tutorial. [2020-02-13]

Scope

Application Programming Interfaces (APIs) are frequently used as a means of Open Data. Indeed, over the last years, many humanities researchers have started to share their data on the web with APIs. As a result, there are a large amount of valuable datasets available. However, APIs are often tailored for developers, and it is still not easy for the researchers who have little IT experience to work on them.

This tutorial offers the participants the possibility to **quickly learn the technology without prior knowledge of programming to start using a vast amount of data (often freely) available on the web**. In particular, it uses **Europeana APIs** to examine millions of cultural heritage objects from museums, libraries, and archives across Europe. Once you learn the principles, it is just a matter of time to try other APIs and take advantage of data useful for your (research) purposes. The world of Big Data is waiting for you!

Contents and expected outcomes

The tutorial consists of two parts. The first part provides the basic theory of APIs including:

- What are APIs?
- Why are they important?
- List of useful APIs

A practical hands-on starts in the second part with:

- Europeana API registration to get an API key
- Viewing Europeana API data on a web browser
- Installing XAMPP
- Using a local web server and creating a web page with PHP and HTML
- Developing a web page for Europeana API with PHP and HTML
- Using an API template to access Harvard Art Museum API

After the tutorial, the participants will be able to understand the basics of APIs and use them with PHP (and HTML) on a local server on their local machines. PHP is a programming language especially suited for web development, while HTML is a markup language to create webpages and applications. The participants will learn to **build their own web page which displays API data**.

Basic technology to learn:

- HTML
- PHP
- JSON

Basic concepts to learn:

- APIs
- Metadata
- Web server

Software requirements

- Web browser (Firefox, Internet Explorer, Chrome, Safari etc)
- XAMPP
- Text editor (Atom etc)

XAMPP creates a local web environment. It is free and includes two important packages for this tutorial: **Apache** web server and **PHP**. In this way, you can create a test website and simulate an access to APIs on your PC.

We need a text editor for a simple programming. You can use pre-installed editors such as Notepad (Windows), but I suggest to use a free software, Atom (Mac, Windows, Linux) which has more useful features. If you get into programming in the near future, it is a good option.

What is Application Programming Interfaces (APIs)?

Internet story so far

To explain APIs, let's briefly go back to the time when the World Wide Web (WWW) was born in the 1990's. It is the core service of the Internet. The WWW was initially designed for **human-to-machine(computer) communication**. We, humans, created websites on our computers, and other humans see them on their computers (with web browsers). At that time, they were mostly **static web pages**, because they are documents and the contents are fixed by webmasters. We were just happy to passively view somebody's web pages with texts (interconnected with hyperlinks) and photos with a **Graphical User Interface (GUI)**. The interaction was mostly between humans and machines, or we could say that machines "coordinated" the human communications.

Then, we got excited and ambitious about the Internet technology. We started to create **dynamic web pages** where human users can interact with each other. Web pages are dynamic, because the contents are not fixed, and dynamically changed by the actions of the users. For example, when we search web contents, send emails, and submit and share documents and photos, the contents of the web pages change. Social media are typical examples. We not only consume web contents, but also generate them. To manage such **web resources**, we needed **database** systems behind websites, so that we can store and manage data properly and efficiently. Data include web resources as well as user information. If a dynamic website works like a software application (which we use often on local machine), we may call it **web application**. Due to the possibility to submit resources and the creation of large databases, a huge amount of data have been created. Millions and billions of websites and datasets you have seen these days.

At this stage, we realise we need more **machine-to-machine communication**. As we have **big data** sometimes far more than humans can actually browse and work on, we need a method for machines to smoothly communicate each other, which is called **web service**. APIs are typical web services[1].

Let's think about a real situation. You maintain a ski website and want to update weather forecast for your ski fields every half an hour. You receive the forecast from a meteorological website which contains weather data. Instead of checking such a website on your own and

[1] In this tutorial, we focus on web APIs, especially data service APIs.

update data manually, it is easier to build a website which can automatically fetch the weather data and display them in a regular interval. In other words, a website communicates with another website. This **machine-to-machine data exchange and automation** is possible, when we use APIs. You can regard the APIs of the meteorological website as a kind of (database) service.

But, can't that be done by embedding a snippet of a website, like we insert a YouTube video on Facebook? That's right, but APIs normally offer standardised raw data and access methods, meaning it is easier to manipulate and customise the data. For instance, what if you need to convert Fahrenheit to Celsius, or show a line chart instead of a bar chart? With embedding, you cannot change how the data are presented. In addition, the separation of data and design is also a great benefit for web developers.

A little funny thing is that a website can offer **both dynamic web pages and APIs**. For instance, while a human user visits a Wikipedia website to read a nice pre-designed article, Wikipedia also offers APIs for another user to let her/him develop a mobile quiz app, which uses Wikipedia's machine-readable raw data. Of course, delivering **APIs are voluntary**, so there are many websites without APIs. But, the number of APIs is growing considerably.

Why are APIs useful for you?

Apart from technical advantages of APIs described above, what are the benefit for an ordinary research user like you?

Good question. So I try to summarise some of the reasons. If you conduct a research, you may prefer to have:

- More data
- Related data
- Interlinking data
- Remote data
- Up-to-date data
- Different types of data (interdisciplinary study)
- Data with better quality
- Data not obtainable (by yourself)
- Data with low price, or gratis

This list is not exhaustive. You can think of more reasons. In general, taking advantage of the power of the Internet, sharing data and reusing data is trendy. "Open Data" and "data-driven research" have been under the spotlight of academia and industries for the last several years. APIs are one of the growing areas of their movement. By accessing a wide range of a large amount of datasets, we can do much more than we could do before, which, as a consequence,

leads to enormous change in our research practice, and hopefully more new discoveries and innovations.

API data for everybody

Most of the time, normal (dynamic) websites are sufficient for us, but there are some cases you may need more and some other reasons why I promote APIs.

To be honest, sometimes I am not very happy with the current practice of Open Data. There is a strong tendency that normal websites are prepared for ordinary users and APIs are for developers. I think this is not very FAIR.

It is understandable that machine-to-machine communication should be developed by developers who have skills. Surely, you need some technical skills to use APIs and it is a big hurdle for normal users. However, **the vast majority of the consumers of the data APIs provide is the normal users** who often do not have any programming skills. In addition, APIs and websites often do not offer the same service and/or data. For example, we can not easily mix the data from different websites in a customised/personalised way. This is proven in my James Cook Dynamic Journal (JCDJ) project.

In author's opinion, APIs should be for everybody. **Anybody should be able to use them as they use word processing or spreadsheet software**. He supports universal design for APIs and Open Data, and this is exactly why he would like to provide this tutorial. Corresponding to the movement of the FAIR (Findable, Accessible, Interoperable, Reusable) data principles, the tutorial also attempts to promote **data reuse by non-techie researchers**. If you are interested in this subject of APIs and universal design, you can learn more about it in my academic article.

That was the quick theory on APIs. Now, let's get our hands dirty with real exercises!

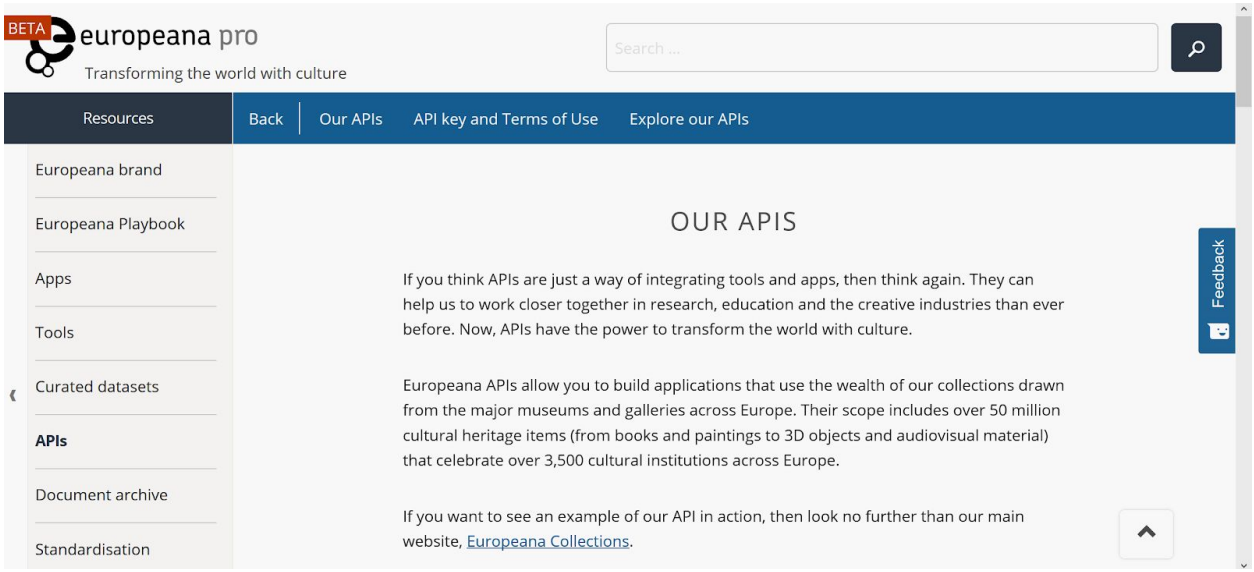
Europeana API

The first API we will try is Europeana. It is one of the biggest sources of information for cultural heritage in Europe. It collects data from museums, archives, libraries etc from all over Europe. Currently, it contains over 50 million objects. There are photos, paintings, books, newspapers, letters, sculptures, coins, specimens, 3D visualisations, and more. If you are humanity specialists, you can imagine what kind of interesting resources there can be. Can't wait?

The goal of this section is to create a website which displays Europeana API data.

To complete the task step by step, we learn how to register yourself with Europeana APIs, to access API data with a web browser, to install XAMPP, to make a simple web page, and to develop another web page to show the API data.

You could read through the documentation of Europeana APIs here, but please do it later on your own and we now make a shortcut.



API registration

1. Fill your personal information at the Europeana API website
2. Click Request Key button
3. In your email inbox, you will find an API key

The first Go with API

Your first view of the API data should be as easy as possible. You can do so with your API key and web browser. So, let's forget about technological aspect for the time being, and just copy and paste the following URL to the address bar of your web browser. Note that you have to replace YOUR_API_KEY with the actual key you get in your email.

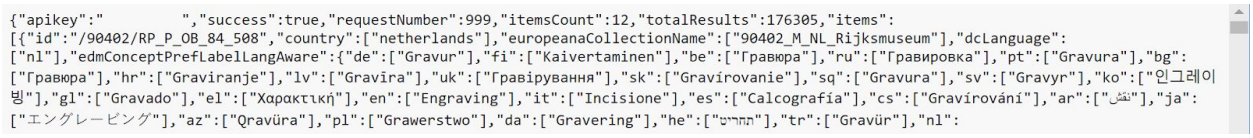
Sample 1
https://www.europeana.eu/api/v2/search.json?wskey=YOUR_API_KEY&query=London

What do you see?

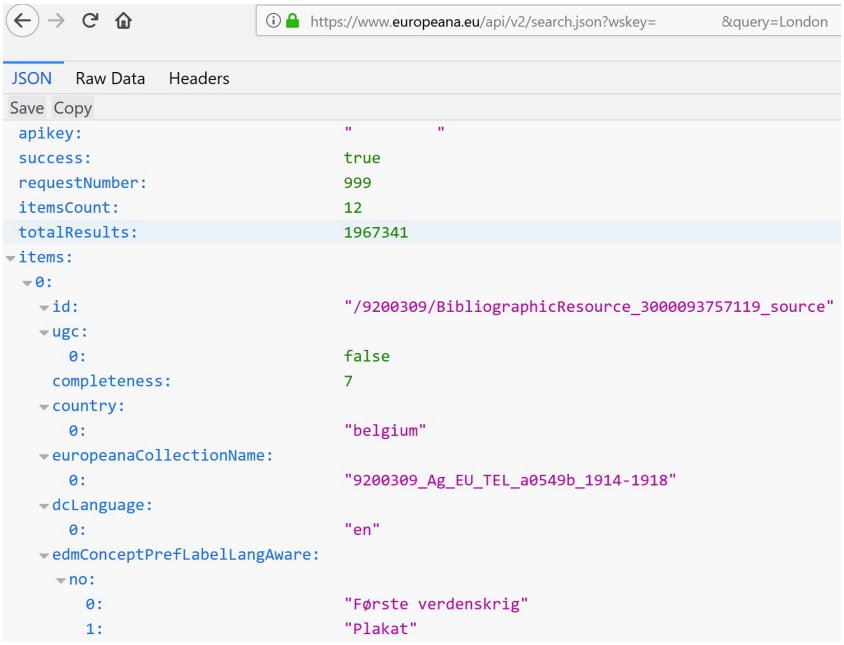
You should see a lot of texts. Congratulations! This is your first data view. You are using Europeana API already.

If you use the latest Firefox, you may see more organised structured data. If you use Internet Explorer or others, you may get a message (below). In this case, save the file and open it in a text editor (such as Notepad or Atom).

API is so easy!
So, let's have a look at what you type (sample 1). It is just a URL. Exactly the same as what you do when viewing a website. For example, to see Europeana website, you type a URL (https://www.europeana.eu). There are some difference, though. You use your API key after `wskey=`, which means your personalised access to this web address. It is followed by `query=London`. You are right. We are querying Europeana database and your search keyword is "London". Europeana offers different types of APIs, but we use the search API.



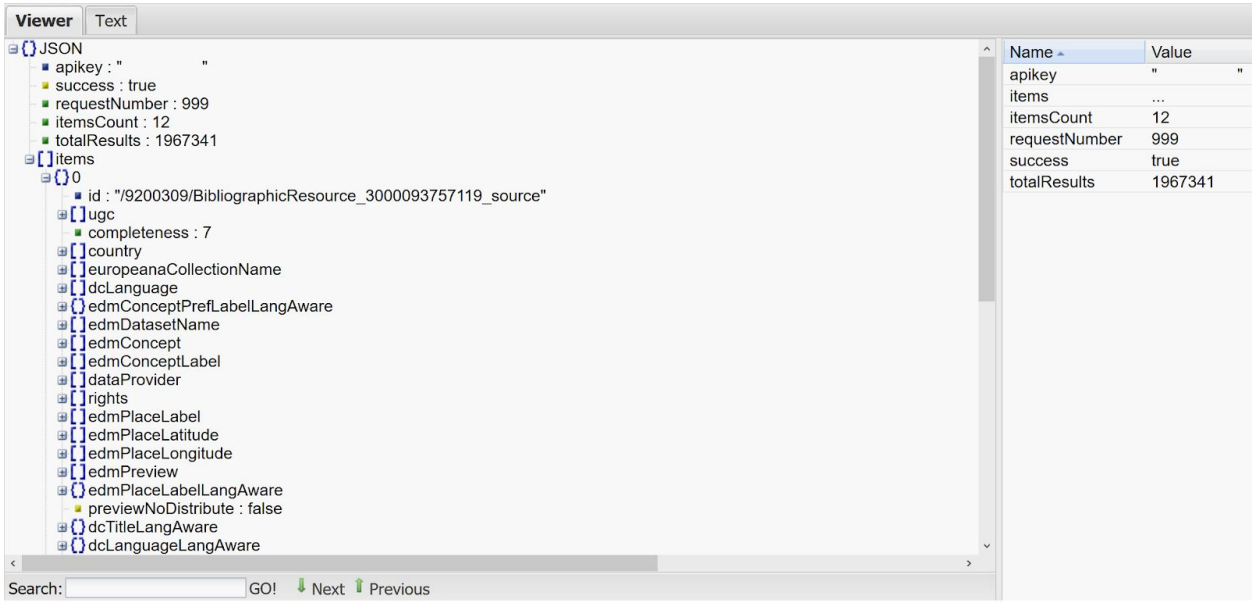
Untidy JSON data structure (raw data) in Chrome



Tidy JSON data structure in Firefox

Understanding API data (JSON)

If your browser does not support a tidy JSON view (the latest Firefox should have a pre-installed JSON viewer), please copy and paste the entire data to an online JSON viewer. It allows us to view the data more easily by expanding (+ button) and collapsing (- button) data hierarchy.



Online JSON viewer

Now, if you look carefully the first lines of the data, you may notice something understandable:

```
{"apikey":"YOUR_API_KEY","success":true,"requestNumber":999,"it
emsCount":12,"totalResults":1967341,
```

You read literally: “apikey” is your API key. Your API access is successful, and you get 1967431 results, but only the first 12 items (records) are returned (to avoid flood of data). After that, you have actual data (i.e. 12 items).

In order to organise data, Europeana uses a particular format/structure, called **JSON (JavaScript Object Notation)**[2]. The data are wrapped with curly brackets (which is called **Object**). It always starts with `{` and ends with `}`. Inside, the data are represented with pairs of strings. Each pair has two components separated by a colon (`:`). For instance, `"totalResults":1967341`. We call this format **name-value pair**. Name is `"totalResults"` and `1967341` is data value. The former is a kind of intex to be used to retrieve the latter (data value). If there are more than one pair, name-value pairs are separated by comma (`,`). To sum up, the simplest JSON data look like:

[2] http://json.org/

```
{
"name 1": "value 1",
"name 2": "value 2"
}
```

By specifying a name, we are able to make a query to retrieve a corresponding value, which is of our interest. As you can see, JSON is a very simple data format, therefore, it is not only very easy for humans to understand, but also for machines (computers) to process data. For this reason it is used in many APIs. Name-value (aka **key-value**) pairs are often used in many programming to store data, so it is good to remember this structure.

In the Europeana search API , the actual data of users' interest are stored within `items`. Here, you see slightly different structure. It contains numbers with square brackets with numbers (`[0]`, `[1]`, `[2]`...). Each bracket is an item/record and we have 12 records. The square bracket represents an ordered collection of values, called an **array**. The number starts with 0. It is a bit strange at first, but this is a rule, so take it as it is. The array is one of the data types of JSON (see also PHP data types in the following section). Similar to name-value pairs, we can simply specify a number to retrieve data in the list. Inside each array, we have name-value pairs. Sometimes the name may have a nesting structure, so that arrays can be repeated. In the Europeana API, this part depends on each record. Some records have more data than others, so the data structure and values may not be consistent.

As there can be a long list of names in a record, let me explain some of the names:

Names	Explanation	Example value
id	Identifier of this item	/9200309/BibliographicResource_3000093757119_source
country:	Country of the data provider	Belgium
dataProvider	Data provider of this item	Royal Library of Belgium
rights	Predefined rights statement (Creative Commons etc)	http://rightsstatements.org/vocab/InC/1.0/
title	Title of this item	Stand Not Upon The Order Of Your Going, But Go At Once Shakespeare Macbeth 3-4 Enlist Now
edmPreview	URL of the preview of this item in Europeana	https://www.europeana.eu/api/v2/thumbnail-by-url.json?uri=http%3A%2F%2Fuurl.kbr.be%2F1017835%2Fthumbs%2Fs&size=LARGE&type=IMAGE

edmIsShownAt	URL (web page) of this item at the website of the data provider	http://uurl.kbr.be/1017835
edmIsShownBy	URL (media file) of this item at the website of the data provider	https://www.rijksmuseum.nl/nl/collectie/RP-P-OB-84.508
type	The type of the item	IMAGE
guid	URL of the item page in Europeana	http://www.europeana.eu/portal/record/90402/RP_P_OB_84_508.html

It is outside of the scope of this tutorial to explain the data model of Europeana (**Europeana Data Model: EDM**), but short explanation would be handy, because all records are based on it. It consists of different descriptions (i.e. **metadata**) about cultural heritage items, including:

1. **Dublin Core** metadata to describe a cultural heritage object (stored in museums, libraries and archives). It includes the description of mostly physical aspects of the object such as title (Mona Lisa), creator (Leonardo da Vinci), size (77 cm × 53 cm), date (1503-1517?) , place (France), owner (Louvre museum), and type (painting). In the Europeana API, it is often specified with prefix `dc`.
2. **Metadata about digital version** of the physical object. It may include URLs where user can view the object (both at the Europeana website and external website), digital formats (jpg), and licensing information (Creative Commons).

To know more about EDM, you can consult their documentation. I used to be one of the main contributors of the documentation. :)

Metadata is power
Metadata is data about data. We use it very often, even without noticing it. The most typical example is a library catalogue. When we look for a book, we use the author, title, date of publication, ISBN etc to find it in a bookshelf. Metadata is those descriptions of the book. In the same way, we use metadata to search something (a flight ticket, a website, news, a video clip) on the internet. As our data become bigger and bigger (billions), metadata is extremely important not only to discover and identify data, but also to process and preserve them. In humanities, many metadata models and formats have been proposed and developed in libraries (MARC, FRBR), archives (EAD), and museums (LIDO, CIDOC-CRM). It is a continuous effort of various communities to develop metadata to represent the data/knowledge of their domains. EDM too is a model to capture the essence of data aggregated from those domains.

Just to view data via APIs, you actually don't need XAMPP we will see in the next section. You can either do it like above, or use Europeana Rest API Console where you can set parameters (e.g. "London" as search keyword) and check the data without any software installation.

Searching and viewing Europeana datasets are good, but it is not very convenient, because we can only view raw data and/or the default data view. So, let's move away from web browser and try to customise the data view by ourselves. That's a developer's job! But don't worry, we make it as easy as possible.

Note that it is a good idea to keep API data view open on a web browser, when developing a web page (from now on in this tutorial too), because you often need to examine the data in this way.

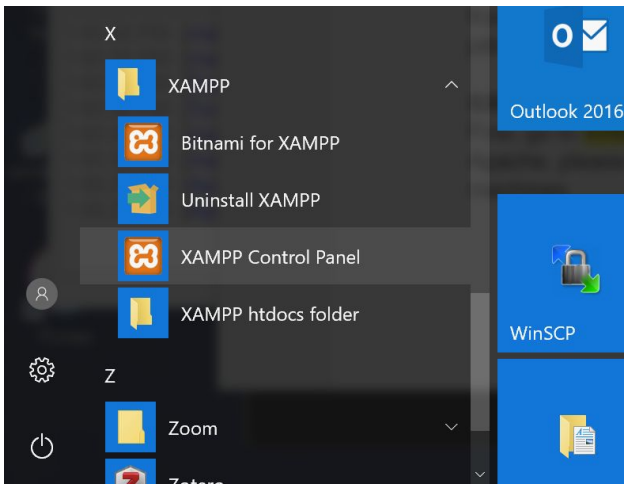
XAMPP installation

Now, we have to set up a new environment. Please go to XAMPP website, download the software for your OS, and install it. XAMPP package has everything you need, so it should be pretty straightforward to install. The current version is 7.2.7 (July 2018).

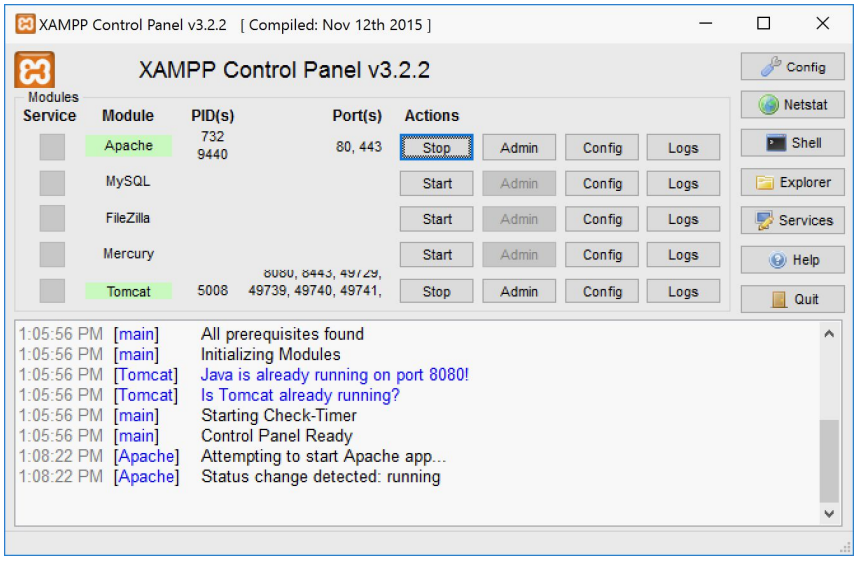
Download XAMPP from the website

XAMPP and the first Go with PHP

When your installation is complete, let's get started. First, go to start menu and click XAMPP Control Panel. If you do not see a green highlight for Apache Module, please click the buttons to start Apache. Then, we can use them on our local machines.

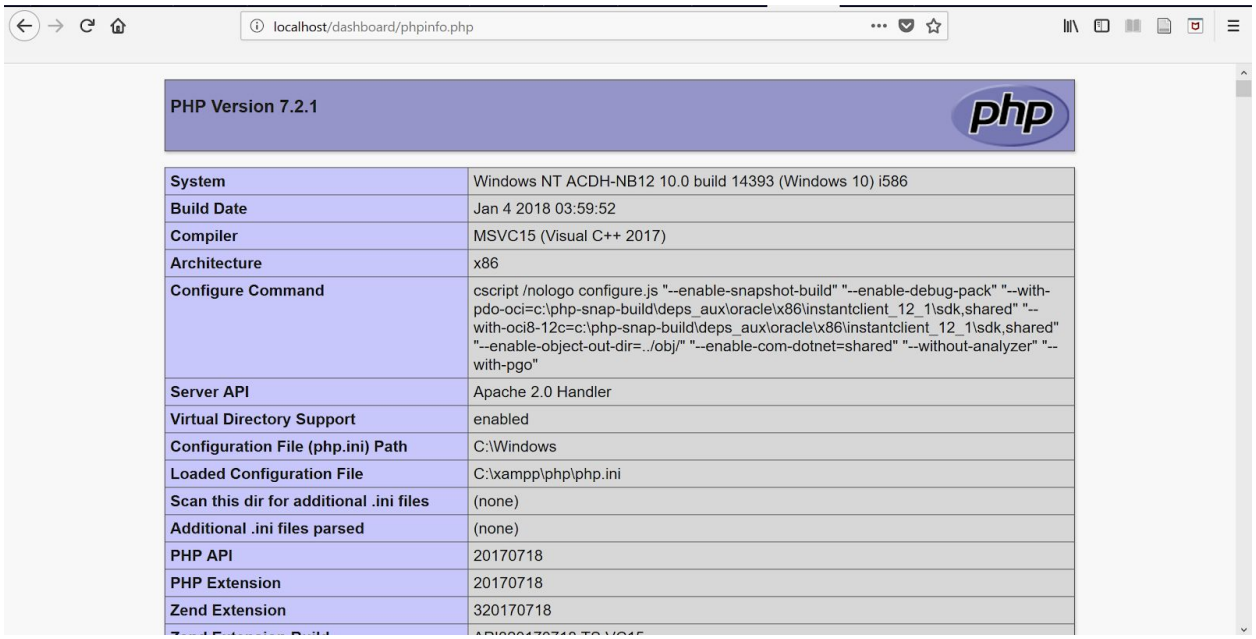


In Windows 10, start menu has XAMPP Control Panel



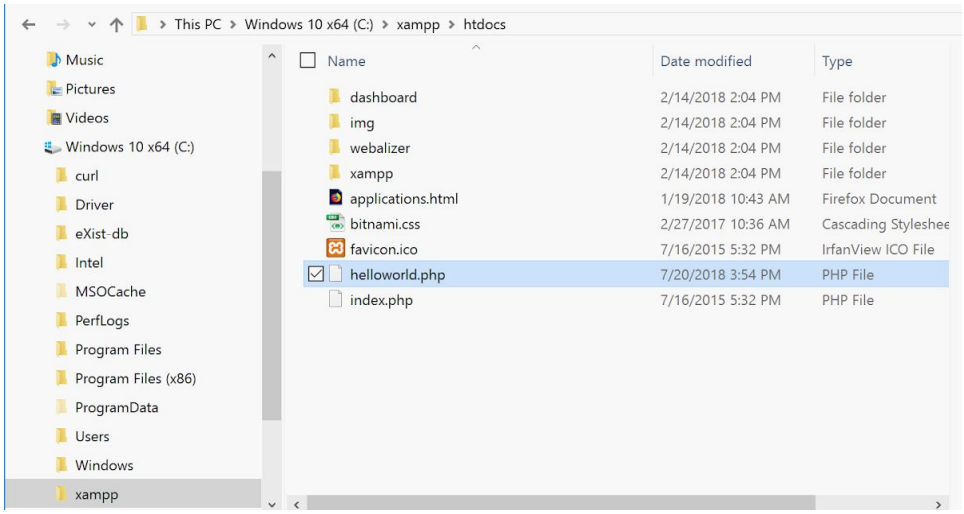
Click Start button for Apache Module, and it is started

Go to http://localhost/dashboard in your browser to see if Apache is working



Optionally go to http://localhost/dashboard/phpinfo.php in your browser to see if PHP is working (i.e. if you see this page)

If you see the screens like above, everything should be OK. Go to **htdocs** folder we created (shortcut is recommended to be created at the desktop). In this folder, we put all the files necessary to create a website. Right now it's almost empty, so let's create a brand new PHP file. Inside the htdocs folder, right click and select create a new text file.
After creating, rename the file to helloworld.php. Alternatively, you can always start your editor and save an empty file with the name, helloworld.php.

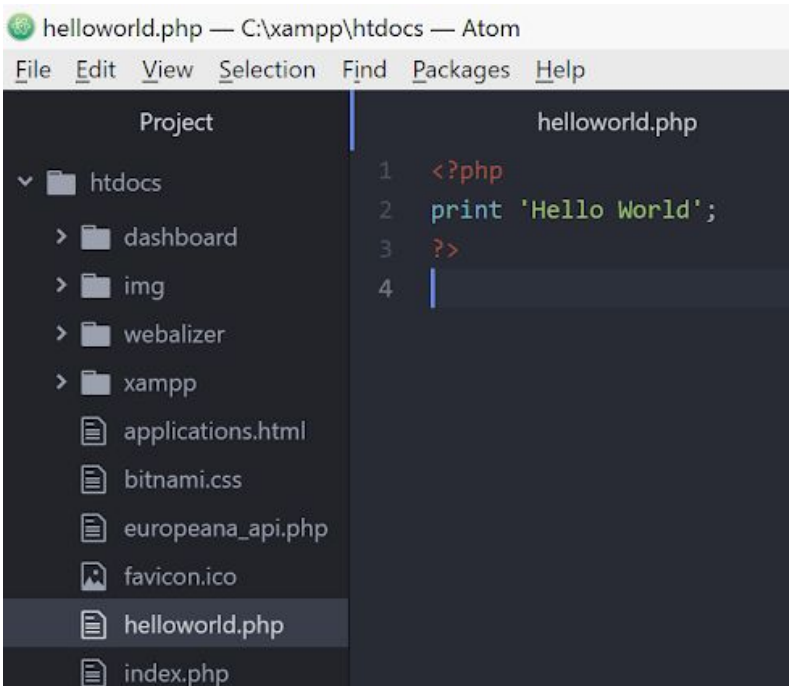


Put all the files in htdocs folder (e.g. c:xampp->htdocs)

As you may have heard, it is developer's tradition to display "Hello World" for the first code. Open the helloworld.php in your text editor, and please write (or copy and paste) the following and save it.

Sample 2

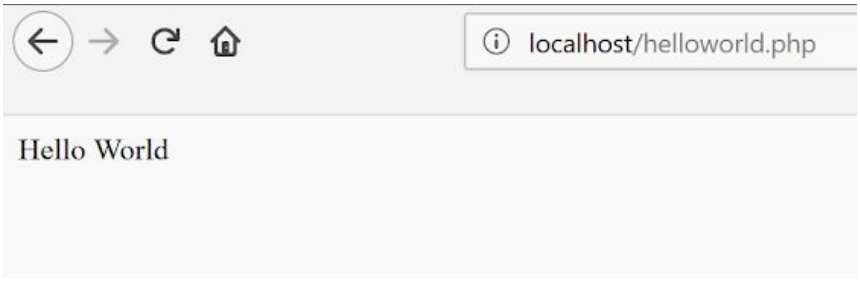
```
<?php
print 'Hello World';
?>
```



Editing helloworld.php with Atom text editor

Open your web browser and type http://localhost/helloworld.php in the address bar. When working on PHP code, it is recommended to keep the browser open with the web page you are editing, so, as soon as you save the file, you can see the outcome immediately. In addition, you could create a bookmark for your convenience.

The outcome? You see “Hello World” in a white background. Congratulations! You have just made the first PHP code. For the sake of curiosity, PHP code should start with `<?php` and ends with `?>`. Just like JSON, those lines declare that the file is PHP. `print` means display the following code “Hello World” as a text.
In PHP, you can use either `''` or `""` (single or double quotes) to indicate that **data type is string** (i.e. text) (There are other data types such as integer, Boolean, or array, but let’s focus on string for now). Very simple.



Your first PHP web page in a browser

The first go with HTML

In sample 2, PHP uses `'Hello World'` as a normal text. But, PHP can work with HTML very well. They can be embedded each other. HTML is a **markup language** which creates web pages and applications[3]. So, let’s make a little change to try with HTML. Change `'Hello World'` into `'<b>Hello World</b>'`, and save the file. Now, refresh the same page on the browser (click “reload current page” button)[4]. What’s happened?

“Hello World” should become in bold. This is because HTML code `<b></b>` makes the texts, sandwiched by the enclosing tags, bold. **HTML basically adds an annotation to the content in-between** (`<b>` means bold).

[3] To view the HTML of a website with your web browser, right click in a text area on the home page (or go to the top menu bar of your browser), and select “view page source”.
[4] It is a good idea to get used to frequently reload/refresh a web page, when developing a website. The change you made with HTML may not be visible, until the page is refreshed.

Be careful that most HTML tags need to have an **opening tag** () and a **closing tag** (). If you mistype or omit them, web browser may not be able to display information properly[5]. HTML takes care of many things of a website, but in this case, it defines the display format on the browser.

HTML images and links

Before moving on to APIs, we would like to do an exercise to create a simple website with some basic HTML and PHP coding. The first exciting stuff is to display an image on a PHP web page. Modify sample 2 as follows and save it. Be aware that the first line has changed slightly, adding
 at the end. Do you see a nice image? Fantastic!

Sample 3

```
<?php
print '<b>Hello World</b><br>';
print                                                    '<img
src="https://upload.wikimedia.org/wikipedia/commons/thumb/7/79/Rame
sses_Vi_closeup.jpg/242px-Ramesses_Vi_closeup.jpg">';
?>
```


 is a line break, so with sample 3, you should be able to see “Hello World” in bold and an image from the next line. In the next line, we are print text strings for the upcoming code, using single quotes. The content of the texts is <img src=””>, which specifies the URL of an image.

[5] In programming in general, you may need to be patient and be precise. Computer may not understand your requests, if you mistype or forget something. So, look at your code very carefully!

Hello World in bold and an image from Wikipedia

In HTML, (enclosing) tag names are called **elements.** Within an element, there are also **attributes** which are additional values that configure the elements or adjust their behavior. Normally attributes appear as name-value pairs, and separated by `=`. For example, `<img>` is an element, and `src=""` is an attribute.

If you put an image on the htdoc folder, you can also specify it like `myimage.jpg` instead of `http://...`

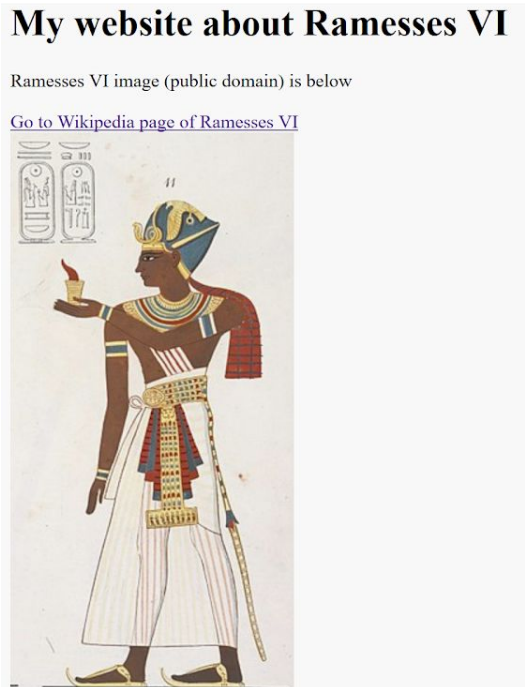
Great! Now let's make a real website. The real power of Internet lies in hyperlinks, so we add a link to Wikipedia. Copy and paste the following code and replace with the entire sample 3.

Sample 4 (Download helloworld.php)

```
<?php
print '<h1>My website about Ramesses VI </h1>';
print '<p>Ramesses VI image (public domain) is below</p>';
print  '<a  href="https://en.wikipedia.org/wiki/Ramesses_VI">Go  to
Wikipedia page of Ramesses VI</a><br>';
print                                                    '<img
src="https://upload.wikimedia.org/wikipedia/commons/thumb/7/79/Rame
sses_Vi_closeup.jpg/242px-Ramesses_Vi_closeup.jpg">';
?>
```

Now, it really looks like a website, doesn't it? `<h1>` is a heading (the biggest size), while `<p>` is a paragraph (i.e. normal text). `<a>` is the hyperlink and `href=` specifies the URL, in this case, a Wikipedia article. As attributes often contain extra information about the element that you don't want to appear in the actual content, the texts "`Go to Wikipedia page of Ramesses VI`" is displayed in place of the lengthy string of URL (i.e. "`https://en.wikipedia.org/wiki/Ramesses_VI`").

Was it easy, or difficult? Anyway, what you have just created is the essence of all the websites on the Internet, containing texts, hyperlinks, and media (image). The only difference is a better layout, more decorations, and perhaps interactive functions. But, well done! You have just become a webmaster!



Complete web page with an image and a hyperlink

Web server and XAMPP (Apache and PHP)
When creating a website, you need a **web server**. It is a powerful computer in which web contents are stored. They should be switched on 24/7, so they are always available. There are millions of servers worldwide which constitute the Internet.

From a user's perspective, s/he types a URL in a browser to view a web page. Then, the browser (i.e. client computer) identifies the server of the web page (which exists somewhere in the world) and asks it to access the page. The server responses to the browser and "serves" the web page. This **client-server model** is the founding stone of the Internet.

To run a web server on your own, you need to pay and register your web address (often via a web hosting service), and it is a bit complicated. XAMPP, instead, **simulates the environment of a web server in your local machine** (Apache), without creating a real website. In fact, you can only view the web page you create in this tutorial via `localhost`, because it does not exist on the web. It is a good way to test your website.

PHP is a code to be executed on a web server, generating HTML for the client. It is included in XAMPP, so we don't need to install separately.

PHP and the first go with API

Finally, we would like to work on API, using what we have learned. We would not like to go into details of programming, but you have to know one important thing.

In programming, we use a lot of **variables**. They are basically references. and, in a way, are similar to the names in name-value pairs of JSON. They name a location where values are stored, so that an unpredictable or changeable values can be accessed through a predetermined name. When you need to use a value, you can simply refer to the variable of a value.

In PHP, variables are represented by prefix `$`. Let's assume I often say "floccinaucinihilipilification", one of the longest English words, meaning the act of something unimportant. For example:

```
$it = 'floccinaucinihilipilification';
print '<p>I think he is doing '.$it.' again</p>';
```

Don't worry about the detail of the syntax. It is enough you understand the core concept here. First, I put this long word in a variable called "it" (`$it`). Then, I can use `$it` to refer to the long word in the second line. The result is HTML paragraph of "I think he is doing floccinaucinihilipilification again". It is like a mathematical formula, right? I guess you understand the following without explanation. x and y are variables containing numbers.

x = 10
y = 5
x + y = 15

Variables are also useful when numbers are assigned, so the calculation of formulas would become simpler. In the following example, you don't have to understand everything, but you can sense how the variables work. Imagine that you can change the `$speed_limit` anytime, and other part will be automatically updated. The code is very flexible and reusable.

```
$speed_limit = 60;
print '<p>When the speed limit is '.($speed_limit).'
km/h,</p>';
print '<p>'.($speed_limit-10).'km/h is safe</p>';
Print '<p>'.($speed_limit+50).'km/h is dangerous</p>';
Print '<p>'.($speed_limit*3).'km/h is super dangerous</p>';
```

I think he is doing floccinaucinihilipilification again

When the speed limit is 60 km/h,

50km/h is safe

110km/h is dangerous

180km/h is super dangerous

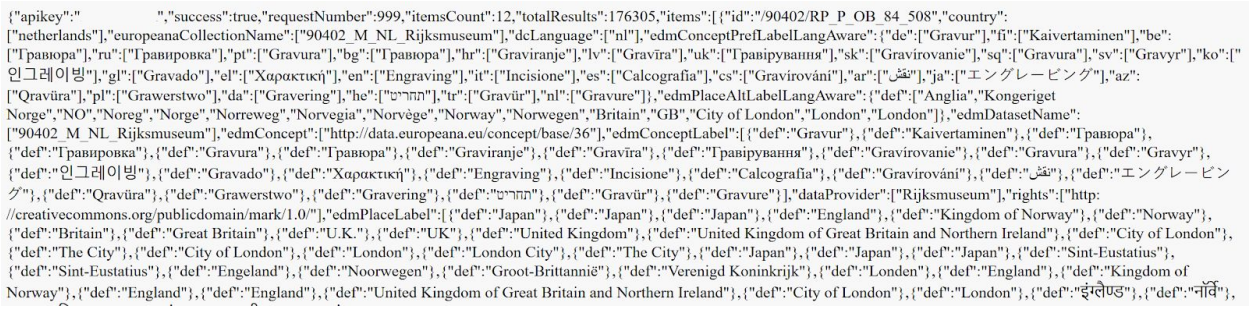
Outcomes of the variable examples (download the complete file)

Now, if you understand variables, let's create a new PHP file called europeana_api.php. Copy and paste Sample 5 (Again, replace `YOUR_API_KEY!`) and save it. Please open your browser pointing to localhost/europeana_api.php What do you see?

Sample 5

```
<?php
$apikey = 'YOUR_API_KEY';

$contents_europeana =
fopen("http://www.europeana.eu/api/v2/search.json?wskey=$apikey&que
ry=London&reusability=open&media=true", "r");
$json_europeana = stream_get_contents($contents_europeana);
fclose($contents_europeana);

print $json_europeana;
?>
```

The first Europeana API code in PHP



JSON data from Europeana API on your local web server

Looks familiar? Yes, that's the JSON we saw in a web browser. But, now it's a big difference. We see it on our web server (`localhost`), not on the Europeana server.

Have a look at the code. The first line defines the variable `$apikey` and store the value of `YOUR_API_KEY`. By doing so, we will not need to type YOUR_API_KEY every time we need it later. Even if we will not use it (this case!), it is a good practice, as you never know what will happen in future. The next line also defines another variable `$contents_europeana` to put some data in it, actually data from Europeana API.
`fopen()` means please open data from `"http://www.europeana.eu/api/v2/search.json?wskey=".$apikey."&query=London&reusability=open&media=true"`, and it is reading only ("`r`"), not writing. The URL includes `$apikey`, thus, it is equivalent to [`"http://www.europeana.eu/api/v2/search.json?wskey=YOUR_API_KEY&query=London&reusability=open&media=true"`](http://www.europeana.eu/api/v2/search.json?wskey=YOUR_API_KEY&query=London&reusability=open&media=true).

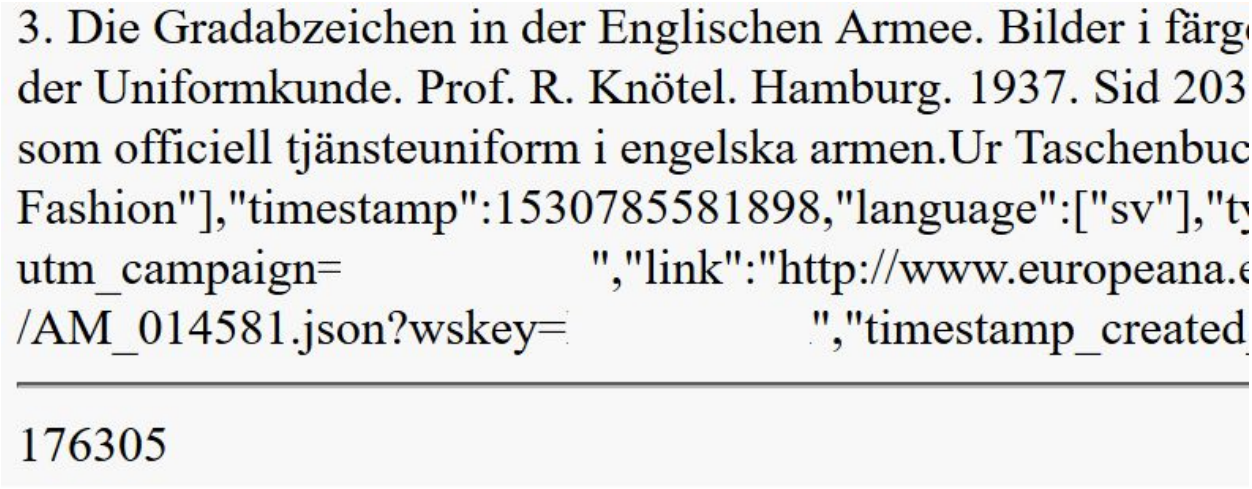
The next line declares another variable `$json_europeana` and assign `stream_get_contents($contents_europeana)` as a value. It obtains the data you opened. As we open the data, we close it with `fclose`. Here, `$contents_europeana` is starting to do a good job not to repeat things. Up to this point, everything happens behind the scenes and you don't see anything on your browser. Only `print $json_europeana;` enables us to display the value of the variable `$json_europeana`, namely the JSON data you already know.

In order to manipulate the JSON data, we need to use it in PHP format. Please add three lines at the end of europeana_api.php, but before closing of PHP (`?>`), and save it.

Sample 6

```
$data_europeana = json_decode($json_europeana);
print '<hr>';
print $data_europeana->totalResults;
```

You see the same JSON data on your browser, but if you look at the bottom, you see find a horizontal line and numbers you remember:



Retrieve a part of JSON data (total Results)

`Json_decode` converts the value of `$json_europeana` (JSON) into PHP code. HTML `<hr>` makes a horizontal line to distinguish JSON data above and below. This is not absolutely necessary, but it for a readability purpose. `$data_europeana->totalResults;` displays the `totalResult` data. With `->,` we can refer to a particular position of data hierarchy and obtain the data value, in this case, `totalResults`. In this way, we can specify a part of data we need to display.

I know it gets complicated, but don't bother too much with the code itself. You only need to understand the general concept and workflow, because we will make a code template later, so you only need to copy and paste with some adjustment.

The next step is the final approach to the Europeana search API in this tutorial. We will create a table view of the same data we have been using. Bear with me!

Table view of Europeana data

So far, data can be seen, but they are kind of raw data. It is not easy to read. So, we re-organise them to build a table view application. You may want to keep what you have done, so **we will add codes on top of the previous ones**. If you don't want to look back, you can simply delete `print $json_europeana;` and `print $data_europeana->totalResults;` and continue the tutorial.

Please add the following code at the end of europeana_api.php, but, again, before closing of PHP (`?>`). Have a look at the file in the browser. You should see a table with a header row. Getting better, right? We just need to put data in the table later. We are almost there!

Sample 7

```
print '<hr>';
// Table view of Europeana data
print          '<table          border=1><tr><th>Title</th><th>Data
Provider</th><th>External Link</th><th>Thumbnail</th></tr>';

// SAMPLE8_COMES_HERE

print '</table>';
```

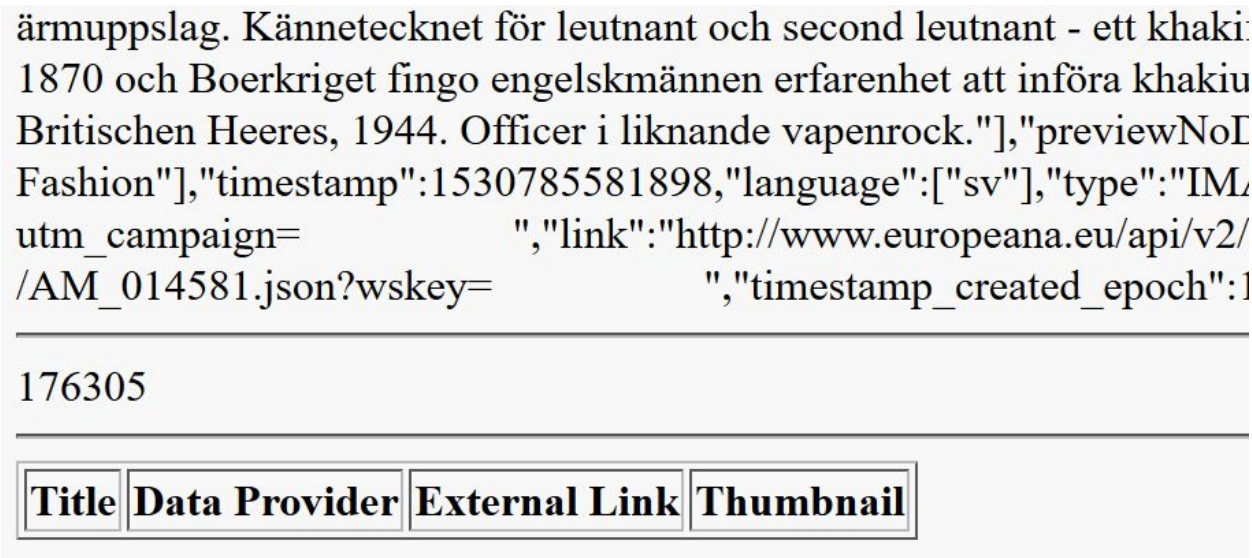


Table header is created

In sample 7, the first line is to separate the previous code from the former one.

On the second line, `//` means ignore this line, so `Table view of Europeana data` and `SAMPLE8_COMES_HERE` are skipped. This is a **commenting function**. Developers use it to document their code, so that they can understand what the code is for and what it does in the future, as well as other developers who need to work on the same code. People often forget what they have done, so it is the best practice to record the history of the code.

On the third line, we set up a table with HTML code. Full explanation can be found in a HTML tutorial, but the first line `<table border=1>` defines a table. Note that `print "</table>";` at the end of the sample 7 is a closing HTML tag. `border` specifies the

thickness of the table border lines. While `<tr></tr>` generates a header row in a table, `<th></th>` provides columns and heading names within.

Next, we will manipulate Europeana data which now should be stored in the variable `$data_europeana`. Replace `//SAMPLE8_COMES_HERE` with the following code.

Sample 8

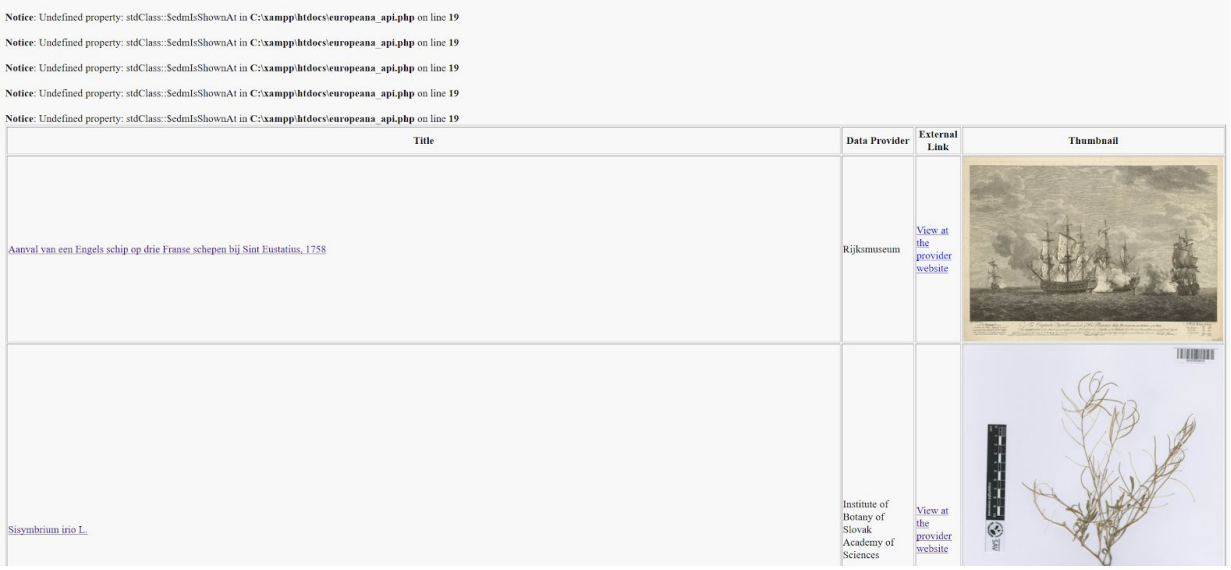
```
foreach($data_europeana->items as $item) {
     print '<td><a href="'.$item->guid.'">'
.$item->title[0].'</a></td>';
     print '<td>'.$item->dataProvider[0].'</td>';
     print '<td><a href="'.$item->edmIsShownAt[0].'">View at the
provider website</a></td>';
     print '<td><a href="'.$item->guid.'"><img
src="'.$item->edmPreview[0].'"></a></td></tr>';
}
```

```php
<?php
$apikey = "YOUR_API_KEY";

$contents_europeana = fopen("http://www.europeana.eu/api/v2/search.json?wskey=".$apikey."&query=London&reusability=open&media=true", "r");
$json_europeana = stream_get_contents($contents_europeana);
fclose($contents_europeana);

print $json_europeana;
$data_europeana = json_decode($json_europeana);
print "<hr>";
print $data_europeana->totalResults;

print "<hr>";
// Table view of Europeana data
print "<table border=1><tr><th>Title</th><th>Data Provider</th><th>External Link</th><th>Thumbnail</th></tr>";
foreach($data_europeana->items as $item) {
    print '<td><a href="'.$item->guid.'">' .$item->title[0].'</a></td>';
    print '<td>'.$item->dataProvider[0].'</td>';
    print '<td><a href="'.$item->edmIsShownAt[0].'">View at the provider website</a></td>';
    print '<td><a href="'.$item->guid.'"><img src="'.$item->edmPreview[0].'"></a></td></tr>';
}
print "</table>";
?>
```

Entire code, including `foreach` in a table

If you are lucky, you now have a cool stuff. What you see is what boring text data of JSON actually contains. As it has URLs of thumbnails, we can show the images in the table. Very nice!

The table now includes the title, data provider, external link, and thumbnail. Don't worry about errors in front of the table (we will fix them later). Just explore the web page you created to check what you can do.



Notice: Undefined property: stdClass::$edmIsShownAt in C:\xampp\htdocs\europeana_api.php on line 19

Notice: Undefined property: stdClass::$edmIsShownAt in C:\xampp\htdocs\europeana_api.php on line 19

Notice: Undefined property: stdClass::$edmIsShownAt in C:\xampp\htdocs\europeana_api.php on line 19

Notice: Undefined property: stdClass::$edmIsShownAt in C:\xampp\htdocs\europeana_api.php on line 19

Notice: Undefined property: stdClass::$edmIsShownAt in C:\xampp\htdocs\europeana_api.php on line 19

Title	Data Provider	External Link	Thumbnail
Aanval van een Engels schip op drie Franse schepen bij Sint Eustatius, 1758	Rijksmuseum	View at the provider website	
Sisymbrium irio L.	Institute of Botany of Slovak Academy of Sciences	View at the provider website	

Table shows Europeana data

Do you remember that Europeana records are stored in an array called items? Within it, each item is ordered with a number `[0]` to `[11]` and contains an record, right?
In order to manipulate data within an array, we need to use `foreach(){}`. It's a function to repeat your task. In our case, within the round brackets, we assign a new variable called `$item` for each record of the array (`$data_europeana->items as $item`). In other words, we can access each record from `[0]` to `[11]` with the variable `$item`. The loop will be specified within curly brackets `{}`. It is easier to understand when looking at the following lines, so leave it for now. As you know, we have to print the datasets, so `print` is used many times in the table. `<td></td>` represents a row in the table.

Now, we would like to scrutinise Europeana's JSON data, because we have to specify what data should be displayed. Apparently we don't need all of them. Please open a new window/tab and visit http://www.europeana.eu/api/v2/search.json?wskey=YOUR_API_KEY&query=London&reusability=open&media=true. You should use JSON viewer to identify the data we would like to fetch.

Let's take a look line by line inside the `foreach`. Basically each `print` line corresponds to a column in the table. The first line makes the first column, namely "Title". The row has `<a></a>`, implying it is a link. Do you remember the HTML syntax for the link? `href` specifies the target URL to which the user jumps, and between `<a></a>` will be the text appears on display. Don't worry about the detail of the syntax, but, in our case, `$item->guid` is the URL and `$item->title[0]` is the text.

Can you find `guid` and `title[0]` in your JSON viewer? While `guid` is the URL of the item page in Europeana, `title` is the title of the item. We use `[0]` after `title`, because it is stored

in the first name/key in an array[6]. As a result, the link and texts works as we intended in the first column.

For the second column, there is nothing more than `<td></td>`, implying only a textual data will be inserted. The content is `dataProvider[0]`, which is understandable, as we have already created the header “Data Provider” in the previous section. The third column has again link elements. `dataProvider[0]` is specified for the link, and simple sentence: “View at the provider website” is used for display. The last column also has a link for `guid`, but additionally, image is created in-between (If you forget the HTML syntax for image, please go back to the previous sections). This part is slightly new. When an image is sandwiched by a link, the former get a link when it is clicked. Thus, the image specified at `edmPreview[0]`, will have a link to the web page specified at `guid`, when it is clicked. You can double-check this on your browser, if it is working, or not. Super!

To sum up, `foreach` loops a task over an array. The repeating element is defined in `()` and the task is defined in `{}`. In our case, we display the data values of each item (from `[0]` to `[11]`) repeatedly, without writing `[0]` to `[11]` one by one. Very handy.

Error handling

Finally, let’s try to make it tidy. It is not 100% satisfactory, because we have error messages. What are they? They basically tell you that PHP cannot process Europeana data, because the data values we requested do not exist in the Europeana records.

So, if you click the image of the first record, you are directed to the correct website, but if you do the same for the second one and others, you cannot reach the website, simply because there is no link. Actually our web site still works, but, due to the absence of data, the error messages are displayed, which is not nice.

How can that happen? Do you remember when we first looked at JSON data in a browser, that data structure and values may not always be consistent. This is it. So, we have to do something when data values are not available.

By the way, developers always have to work a trial and error like this. In other words, they write a code, test it, and fix bugs, if it’s not working. We do the same. Let’s fix the bugs!

In order to fix the bugs, we should make a command like “please show us data only when they are available and ignore if they are not”. To this end, we can replace Sample 8 with the following code:

[6] Sometimes there is only one name/key in an array

Sample 9 (Download europeana_api.php)

```
foreach($data_europeana->items as $item) {
     print '<td><a href="'.(isset($item->guid)?$item->guid:'').'">'
.(isset($item->title[0])?$item->title[0]:'').'</a></td>';
                                                                print
'<td>'.(isset($item->dataProvider[0])?$item->dataProvider[0]:'').'<
/td>';
                                                print           '<td><a
href="'.(isset($item->edmIsShownAt[0])?$item->edmIsShownAt[0]:'').'
">View at the provider website</a></td>';
                                                print           '<td><a
href="'.(isset($item->guid)?$item->guid:'').'"><img
src="'.(isset($item->edmPreview[0])?$item->edmPreview[0]:'').'"></a
></td></tr>';
}
```

Sample 9 is almost identical with Sample 8, but `isset` is added. It is a PHP code to check if data is set or not within bracket `()`. In addition, the following syntax makes a conditional task:

```
isset($data)?'$data is set':'$data is not set';
```

In this case, if `$data` is available, the text `'$data is set'` will be used, and if `$data` is not available, the text `'$data is not set'` will be used. We can execute a task based on this code for a condition. For example, in the first column part of our code, if `$item->guid` is set, we use it. Otherwise empty text (`''`) is used. So, the links willl work properly. We use this pattern of code every time data is called from API (i.e. each time `$item->` appears). As a result, hopefully error messages should disappear.

Unfortunately, there is no good way to know in advance, if the data we need is available or not. This configuration is an advantage that, if there is a little mistake, something works at least, without breaking everything else, but the disadvantage is that we need to do something to avoid errors.

Please note that **we don't handle all the error scenarios in this tutorial**. For example, we didn't think of the potential situation where the data (`$item->edmIsShownAt[0]`) isn't an URL, or another array is nested. This is because this tutorial is NOT a programming lesson. To develop a proper application, you need to delve into PHP programming.

Anyway, big congratulations! You have just made a wonderful web page within a short space of time. I hope you are getting used to APIs by now.

API template

Generalising API call in PHP

As we have seen, the API code may be a bit complicated, but don't worry. What is important now is not to fully understand the meaning and memorise the syntax, but to **use the code as an API template**. Sample 10 is a core part of Sample 9. What you have to change is only some parameters in the template. Here, you can change the name of VARIABLE1, VARIABLE2, and VARIABLE3, as you wish. Depending on the URL of an API, HTTP and YOUR_API_KEY should be changed too. But, other parts should remain the same.

Sample 10

```
$apikey = 'YOUR_API_KEY';

$VARIABLE1 = fopen('HTTP', 'r');
$VARIABLE2 = stream_get_contents($VARIABLE1);
fclose($VARIABLE1);
$VARIABLE3 = json_decode($VARIABLE2);
```

In addition, you would need to adjust what you would like to do with the actual data. For example, Sample 11 generalises the data retrieval part, only consisting of `foreach` to cope with arrays and print the data values in a loop.

Sample 11 (Download template_api.php)

```
foreach($data as $item) {
    print 'WHATEVER YOU WANT TO DISPLAY';
}
```

By combining Sample 10 and 11, you would be able to manipulate a various type of JSON data.

Other APIs

We have used JSON so far, but there are other formats and protocols for APIs.

For example, the National Diet Library of Japan uses XML format for their catalogue search API through SRU protocol. Use your web browser to view query results:
http://iss.ndl.go.jp/api/sru?operation=searchRetrieve&query=creator%20exact%20%22Takeshi%20Kitano%22

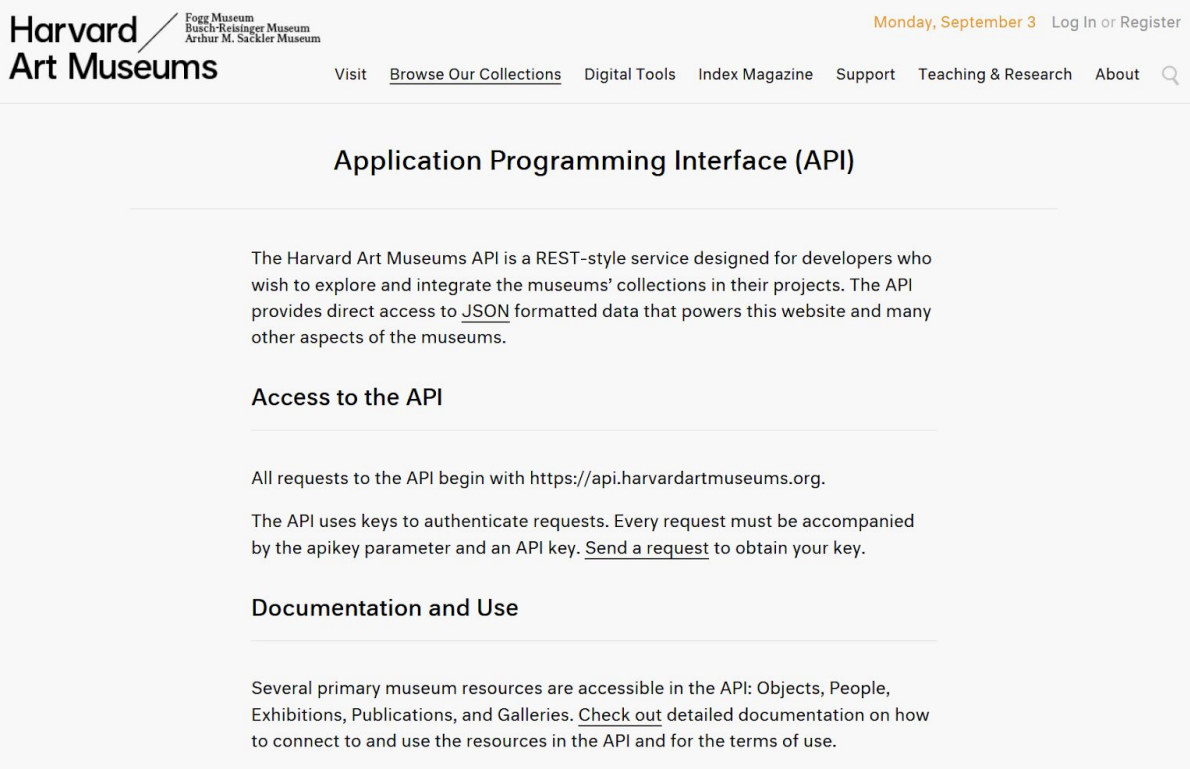
Another API is also XML, but offered by OAI-PMH protocol, which is very popular among libraries:
http://iss.ndl.go.jp/api/oaipmh?verb=ListRecords&metadataPrefix=oai_dc&from=2015-12-01

DBpedia (database version of Wikipedia) is equipped with dozens of formats, including JSON, JSON-LD, XML, RDF. Select a format from the top menu:
http://dbpedia.org/page/Tomoyasu_Hotei

All of them are offered without registration (no API key), which is handy.

Try the template with Harvard Art Museums

So, let's see if the API template actually works with other APIs. For this, we use the Harvard Art Museum APIs. Please have a quick look at their API documentation. As usual, you need to get an API key first.



Once you get it, let's quickly check their object search API on a web browser to understand the data structure:
https://api.harvardartmuseums.org/object?apikey=YOUR_API_KEY&keyword=andromeda

As you can see, records are present in the form of an array within `records` element. This gives you an idea what data you would like to fetch. Now, you know what to do with the template, right? Guess what Sample 12 shows in your browser.

Sample 12

```
<?php
$apikey = 'YOUR_API_KEY';

$contents_harvard                                                    =
fopen("https://api.harvardartmuseums.org/object?apikey=$apikey&keyw
ord=andromeda", 'r');
$json_harvard = stream_get_contents($contents_harvard);
fclose($contents_harvard);
print($json_harvard);
 // For display purposes, <hr> are added several times in this file
print '<hr>';
$data_harvard = json_decode($json_harvard);
print $data_harvard->info->totalrecords;
print '<hr>';

//SAMPLE 13 WILL COME HERE

?>
```

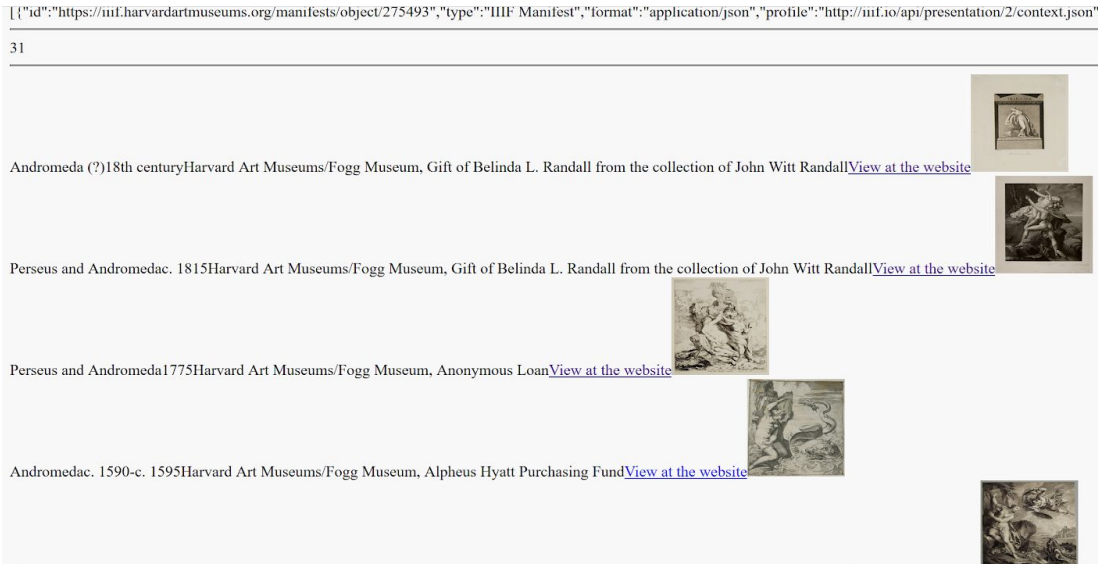
Apparently, new names were assigned as VARIABLEs. For instance, `$contents_harvard` and `$json_harvard` are used. But, all others should look the same.

If you are ready, you can add the following code below the comment in the sample 12:

Sample 13 (Download harvard_api.php)

```
foreach($data_harvard->records as $item) {
    print '<td>'.(isset($item->title)?$item->title:'').'</td>';
    print '<td>'.(isset($item->dated)?$item->dated:'').'</td>';
                                                                print
'<td>'.(isset($item->creditline)?$item->creditline:'').'</td>';
    print '<td><a href="'.(isset($item->url)?$item->url:'').'">View
at the website</a></td>';
                                            print           '<td><a
href="'.(isset($item->isShownAt)?$item->primaryimageurl:'').'"><img
src="'.(isset($item->primaryimageurl)?$item->primaryimageurl:'').'"
height="100" width="100"></a></td></tr>';
```

```
    print '<br>';
}
```



Outcome of harvard_api.php

Hopefully, you see something very similar to europeana_api.php. This time, we simply present each record separated by `<br>` (line break), and do not create a table on purpose. For this reason, our results look untidy, but that simply implies you can do whatever you want. One addition is `<img` element specifies the size of the image as `height="100" width="100"`, thus all images have the same size.

The point is the **API template can be reused**, therefore, the most difficult part would be **the examination of underlying data model of API and handling of data structures**. To manage that, you need to read an API documentation carefully.

APIs for everybody

As I said in the beginning, by now you may understand why **APIs are supposed to be for developers to build something new**. Normal users don't need to use APIs. But, once you understand the basic of APIs, it may not be a big deal, right? If you can learn a bit of programming, you are no longer restricted by what a website offers by default (i.e. Europeana's or Harvard Art Collection search engine interface). There are many things you can't do with the default website, but you are now free to build your own system, for example, to select, filter, compare, process analyse data from different APIs. So, what are you waiting for? Be brave and start your new project!

Useful APIs

- The New York Times (Famous newspaper)
- The Digital Public Library of America (Database of cultural heritage resources aggregated from US organisations)
- Archives Portal Europe (Finding Aids aggregation of European archives)
- VIAF (Database of persons and organisations aggregated from libraries around the world)
- Geonames (Geographical information)
- Wikipedia (World-famous encyclopedia)
- The Open Library (Freely available digital library in conjunction with Internet Archive)
- List of useful APIs for museums

The author's API projects

If you are interested, why not visiting two projects of the author, which experiments with APIs for Digital Humanities? Hopefully they can inspire you.

- James Cook Dynamic Journal (JCDJ)...Contextualisation of a book from The Open Library
- WiQiZi...Gamification of Wikipedia/DBpedia
- CAROL…Exploring the books from the Open Library in context

Thank you very much for your patience. I hope you learn the fundamentals of APIs.
Good luck with your API challenges. You can now go back to the list of APIs in this tutorial and start exploring the data paradise. I am happy if you can send me your feedback about the tutorial and/or inform me of wonderful applications you have created!