

# Web API webinar

---

## ► Introduction

### What is an API ?

Let's say that for an installation, you need to retrieve pictures from Instagram. Could be all pictures matching a specific hashtag, or all pictures posted by a specific user. Obviously, you won't be able to directly browse Instagram's servers to retrieve the pictures you need. This would be a massive security issue on their side, and you would have to find your way through all the pictures that all users post to find the ones you like.

Instead, what if Instagram set up some simple service you could send queries to, and that service would then reply to you with what you asked for? For instance, you could ask for all pictures matching the hashtag [#vvvvisdope](#), posted after the 7th of June 2021 in Berlin's area.

Well, this is what an API (which stands for Application Programming Interface) is about. It acts as an intermediary (interface) between you and the resources you want to access, and provides a documented and easy way to retrieve those resources.

In the end, we don't care about where or how Instagram stores data, we just know that if we send the correct query, we will get the data we want.

API serve all kind of purposes. You need to know the weather forecast for the next five days? There's an API for that. Want to get a random Kanye West quote? There's also an API for that.

### REST ?

REST is one of the many "types" of web APIs you might encounter. It stands for *Representational State Transfer* and defines a whole bunch of things (you can read [this article from IBM](#) if you're curious), but for now we'll just try to keep in mind that REST APIs are based on HTTP calls. What does that mean you ask? Let's interact with an API!

### How do we access an API ?

As we just said, the cool thing about REST APIs is that they're based on simple HTTP calls. And you already have an application that's super good at making HTTP calls at hand : your web browser. With just a web browser, we can already have some basic interactions with REST APIs. Let's get our hands dirty in the next part!

## ► Implementing a REST API in vvvv

### The Chicago Art Institute API

There are tons of APIs you can play with. Have a look at [this repo](#) for a list of public APIs. At the end of this workshop, you should be able to play with many of them.

Let's pick the [Art Institute of Chicago API](#). It allows us to browse the institute's public collection. The first thing you should do when working with an API is having a look at the documentation. This is where you'll find all its

capabilities, get some examples and in-depth explanations on how to use it. So let's browse the documentation here.

How can we have access to the list of artworks the Institute offers? Can you spot that in the documentation?

This url might do what we're looking for :

```
https://api.artic.edu/api/v1/artworks
```

And what happens if we type this in an address bar? We get a huge text back : this is a JSON string that presents the data in a structured way! Let's see if we can spot some interesting items there...

Another small exercise, can you find a way to search artworks by keywords?

And that's it, that's the basis of REST APIs : making calls to URLs that return some text, and retrieve interesting data from it.

Let's now see how we would do such a thing in vvvv.

We'll start by installing a little nuget that will make our lives easier when working with HTTP calls : it's called VL.SimpleHTTP. Let's install it with the following command :

```
nuget install VL.SimpleHTTP
```

It just contains two nodes : **Request (Blocking)** and **Request (Async)**. For now, let's use the **async** version. And just to see what it does : type <http://www.vvvv.org> in the **URL** pin and bang the **Refresh** pin, you see it just returns some HTML. So what we are doing here is exactly the same as we did before when we pasted the URL in a web browser.

Let's try to build a very simple application that allows us to search for a keyword using the Institute's API and display the results in a sort of slideshow. The solution for this problem is actually bundled with the library! Have a look at the help patch called *Art Institute of Chicago API*.

## Making a collaborative message board

We have now seen how to read data from an API. That's cool but REST API allow us to write, update and delete data as well! Let's talk about HTTP a bit : so far, we've only done **GET** queries : if you have a look at the **Method** input pin of the **Request** node, you'll see it's set to **GET**.

You can think of a method as "flag" that we put on our request to indicate what we want to do with our query. For instance, if we want to retrieve some information, we use **GET**. If we need to send some information, we would use **POST**. To modify something, we'd use **PATCH**. And to delete something, well .. **DELETE**.

Now keep in mind that this method thing is only a flag, and if someone designs an API where you can delete stuff with a **GET** well .. that would work. But it's considered very bad practice to do so. Anyway, let's patch.

We want to do this : create a little patch that displays text messages that users can submit from all over the world. Each text message can define :

- A message title
- A message body
- A background color
- A time to be displayed in seconds

We'll need three things :

- An API endpoint that allows us to create new messages
- One that allows us to retrieve all messages users have created so we can display them
- A database to store these messages

There is a very cool service that does all these things at once : it's called [Directus](#). Now, learning how to use it is not the scope of this workshop. Just remember this : it's a *Content Management System* (CMS) where you can design your own data structure.

- In our case, we design a *Message* collection (this is how Directus calls it) like we would create a Class or a Record in vvvv (a message has a *Title*, a *Color*, and a *Body*, for instance)
- It then *automagically* generates an API that allows you to query this data structure you have created, and do all operations you can imagine on it : get or modify data, create, delete.. everything a REST API has to offer!

For the sake of this workshop, we have already created the structure that represents a message in Directus, and as a consequence we have a ready to use API to interact with it.

## Jumping in

Remember, documentation is our best friend. Let's have a look at the [Item section of the Directus API reference](#). Don't pay attention to [GraphQL](#) stuff for now, we're not covering it in this workshop.

This section covers how to interact with items. What's the endpoint to retrieve a whole collection of items? This query looks interesting :

```
GET /items/:collection
```

This is how queries are represented in most API documentations. It means that we must make a [GET](#) query to [/items/:collection](#), where [:collection](#) is the name of the collection we want to retrieve, in our case [messages](#). Of course, you should append before [/items](#) the domain name of the API you're targetting.

If you want to get some info about how urls are structured have a look at [this article from Mozilla's developer network]((https://developer.mozilla.org/en-US/docs/Learn/Common\_questions/What\_is\_a\_URL).

Let's try to repeat this process with a simple node that will allow us to publish a message from our patch!

The solution is hidden as Advanced nodes in the document that you used for the exercise!

The online Directus instance for this exercise is not online anymore! The important part is rather about making queries, so you can still study how these nodes are structured!

## ► Further reading

- During the workshop, we saw a tool called [Postman](#) that allows us to test and document queries. This is very useful when you're discovering an API and quickly want to understand how it behaves.
- If you have questions about how HTTP works, definitely have a look at Mozilla's [MSDN](#). It contains lots of articles on how it works :
  - They have a section covering [HTTP](#)
  - Another one explaining [HTTP request methods](#)

## Bonus : using the experimental serialization nodes

At the end of the workshop, we saw how to tackle the issues that come up when using the default Serialization and XML nodes. Here's the problem : when serializing an object with those and converting to JSON, all properties are considered to be strings. This can be a problem when sending queries to APIs that explicitly expect some properties to be integers, for instance.

Since a few versions, vvvv comes with new experimental nodes : [JsonToObject](#) and [ObjectToJson](#). They don't work out of the box though, there are a few steps we need to take before being able to use them.

The following procedure is shown at the end of the workshop, but here are the steps in case you need to quickly refer to that procedure later

### Ingredients

We'll need a bunch of things :

- A JSON representation of the object we want to serialize
- The [jtd-infer](#) command line tool
- The [jtd-codegen](#) command line tool
- [Visual Studio Community 2022](#)

Ok, let's go.

1. Start by saving the JSON representation of your object in a file named [object.json](#)
2. Run [jtd-infer](#) on that file using the following command : `jtd-infer.exe tree.json`. This outputs a JSON schema of your JSON object in the console window. Copy that in a new file named [schema.json](#)
3. Run [jtd-codegen](#) on the schema you just generated. This will produce a bunch of C# files that we will use in the next step. The command to run is `./jtd-codegen.exe schema.json --csharp-system-text-out ./ --csharp-system-text-namespace <YourNamespace>`. Replace `<YourNamespace>` with the namespace you want your nodes to live in.
4. Ok cool, there should now be a bunch of `.cs` files in your current folder. In vvvv, click the Quad Menu and click New/C# File. In the dialog, pick *Static Utils* and click *Create*.
5. You should now have Visual Studio open. In the solution explorer on the right, right-click and select Add/Existing item. Pick the `.cs` files that you created in step 3.
6. Inspect the files that you just imported, and make sure the types are what they are supposed to be. Sometimes, [jtd-infer](#) cannot infer the right type.

7. Once you're happy with the classes and properties names and types, save the solution and go back to `www`.
8. Search for the classes you have just imported in Visual Studio in the node browser : you should have `Create` nodes and getters/setters for those.
9. You can now use the experimental `ObjectToJson` and `JsonToObject` to easily serialize/deserialize proper JSON!

## Links we saw during the workshop

- [The forum post that introduces the experimental nodes](#)
- [The Directus documentation](#)
- [JsonMate](#)