JSON PHP

A common use of JSON is to read data from a web server, and display the data in a web page.

This chapter will teach you how to exchange JSON data between the client and a PHP server.

The PHP File

PHP has some built-in functions to handle JSON.

Objects in PHP can be converted into JSON by using the PHP function json_encode():

PHP file

```
<?php
$myObj->name = "John";
$myObj->age = 30;
$myObj->city = "New York";

$myJSON = json_encode($myObj);

echo $myJSON;
?>
```

The Client JavaScript

Here is a JavaScript on the client, using an AJAX call to request the PHP file from the example above:

Example

Use JSON.parse() to convert the result into a JavaScript object:

```
const xmlhttp = new XMLHttpRequest();
xmlhttp.onload = function() {
  const myObj = JSON.parse(this.responseText);
  document.getElementById("demo").innerHTML = myObj.name;
}
xmlhttp.open("GET", "demo_file.php");
xmlhttp.send();
```

PHP Array

Arrays in PHP will also be converted into JSON when using the PHP function json_encode():

PHP file

```
<?php
$myArr = array("John", "Mary", "Peter", "Sally");

$myJSON = json_encode($myArr);

echo $myJSON;
?>
```

The Client JavaScript

Here is a JavaScript on the client, using an AJAX call to request the PHP file from the array example above:

Example

Use JSON.parse() to convert the result into a JavaScript array:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onload = function() {
  const myObj = JSON.parse(this.responseText);
  document.getElementById("demo").innerHTML = myObj[2];
}
xmlhttp.open("GET", "demo_file_array.php", true);
xmlhttp.send();
```

PHP Database

PHP is a server side programming language, and can be used to access a database.

Imagine you have a database on your server, and you want to send a request to it from the client where you ask for the 10 first rows in a table called "customers".

On the client, make a JSON object that describes the numbers of rows you want to return.

Before you send the request to the server, convert the JSON object into a string and send it as a parameter to the url of the PHP page:

Example

Use JSON.stringify() to convert the JavaScript object into JSON:

```
const limit = {"limit":10};
const dbParam = JSON.stringify(limit);
xmlhttp = new XMLHttpRequest();
```

```
xmlhttp.onload = function() {
  document.getElementById("demo").innerHTML = this.responseText;
}
xmlhttp.open("GET","json_demo_db.php?x=" + dbParam);
xmlhttp.send();
```

Example explained:

- Define an object containing a "limit" property and value.
- Convert the object into a JSON string.
- Send a request to the PHP file, with the JSON string as a parameter.
- Wait until the request returns with the result (as JSON)
- Display the result received from the PHP file.

Take a look at the PHP file:

PHP file

```
<?php
header("Content-Type: application/json; charset=UTF-8");
$obj = json_decode($_GET["x"], false);

$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");
$stmt = $conn->prepare("SELECT name FROM customers LIMIT ?");
$stmt->bind_param("s", $obj->limit);
$stmt->execute();
$result = $stmt->get_result();
$outp = $result->fetch_all(MYSQLI_ASSOC);

echo json_encode($outp);
?>
```

PHP File explained:

- Convert the request into an object, using the PHP function json_decode().
- Access the database, and fill an array with the requested data.
- Add the array to an object, and return the object as JSON using the json_encode() function.

Use the Data

Example

```
xmlhttp.onload = function() {
  const myObj = JSON.parse(this.responseText);
  let text = "";
  for (let x in myObj) {
    text += myObj[x].name + "<br>";
  }
  document.getElementById("demo").innerHTML = text;
}
```

PHP Method = POST

When sending data to the server, it is often best to use the HTTP `POST` method.

To send AJAX requests using the `POST` method, specify the method, and the correct header.

The data sent to the server must now be an argument to the `send()` method:

Example

```
const dbParam = JSON.stringify({"limit":10});
const xmlhttp = new XMLHttpRequest();
xmlhttp.onload = function() {
  const myObj = JSON.parse(this.responseText);
  let text ="";
  for (let x in myObj) {
    text += myObj[x].name + "<br>";
  }
  document.getElementById("demo").innerHTML = text;
}
xmlhttp.open("POST", "json_demo_db_post.php");
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xmlhttp.send("x=" + dbParam);
```

The only difference in the PHP file is the method for getting the transferred data.

PHP file

Use $_POST instead of $_GET:

```
<?php
header("Content-Type: application/json; charset=UTF-8");
$obj = json_decode($_POST["x"], false);

$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");
$stmt = $conn->prepare("SELECT name FROM customers LIMIT ?");
$stmt->bind_param("s", $obj->limit);
$stmt->execute();
$result = $stmt->get_result();
$outp = $result->fetch_all(MYSQLI_ASSOC);

echo json_encode($outp);
?>
```

https://www.w3schools.com/js/js_json_php.asp

json_decode

(PHP 5 >= 5.2.0, PHP 7, PHP 8, PECL json >= 1.2.0)

json_decode — Decodifica uma string JSON

json_decode(string `$json`, bool `$assoc` = ?): mixed

```php
<?php
$json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

var_dump(json_decode($json));
var_dump(json_decode($json, true));

?>
```

```php
$json = '{"foo-bar": 12345}';

$obj = json_decode($json);
print $obj->{'foo-bar'}; // 12345
```

PHP JSON COMPLETE TUTORIAL

THE DEFINITIVE JSON GUIDE FOR PHP DEVELOPERS

This is the ultimate guide to use JSON objects with PHP.

In this tutorial (updated in 2020) I'll show you:

- How to create and send JSON objects
- How to decode JSON objects
- All the encoding options explained
- How to set the JSON Content-Type
- JSON validation… and more

Plus: **working examples** you can copy and use right away.

CONTENTS

CHAPTER 1

What is JSON?

CHAPTER 2

JSON encoding

CHAPTER 3

Encoding options

CHAPTER 4

Sending a JSON object

CHAPTER 5

JSON decoding

CHAPTER 6

Validation and errors

EXAMPLE

Create a JSON object from database data

EXAMPLE

Send a JSON file as an email attachment

Chapter 1
What is JSON?

You have heard of JSON before.

But what is it, exactly?

And what is it used for?

In this first chapter, I'm going to explain how JSON works and what are its uses in web development.

What is JSON?

JSON is a **data container** used to send, receive and store variables.

As Wikipedia defines it, JSON is a "data interchange format".

Many web applications use this data format to exchange data over the Internet.

This is how a JSON object looks like:

```
{
  "Name": "Alex",
  "Age": 37,
  "Admin": true,
  "Contact": {
    "Site": "alexwebdevelop.com",
    "Phone": 123456789,
    "Address": null
  },
  "Tags": [
    "php",
    "web",
    "dev"
  ]
}
```

As you can see, a JSON object is a *container* for other variables.

More precisely, a JSON object contains a list of *key* => *value* pairs, separated by a colon.

The *keys* are the names of the variables.

In the above example, the keys are “Name”, “Age”, “Admin”, “Contact” and “Tags”.

The keys are always strings and are always enclosed in double quotes.

The *values* are the actual **values of the variables** identified by the keys.

While the keys are always strings, the values can be any of the following types:

- **Strings**, like “Alex” (the *Name* variable).
- **Numbers**, like 37 (the *Age* variable). Numbers can be integers or floats.
- **Boolean** values (“true” or “false”), like the *Admin* variable.
- **Null** values, like the *Address* variable.

Strings are always enclosed in double quotes (“”). Numbers, Booleans and null values are not.

A value can also be a **JSON object itself**, containing more nested *key => values*.

In other words, a JSON object can contain one or more JSON objects.

For example, the “Contact” variable is a JSON object with the following *key => value* pairs:

- “Site” (*key*) => “alexwebdevelop.com” (*value*)
- “Phone” (*key*) => 123456789 (*value*)
- “Address” (*key*) => null (*value*)

Objects are enclosed by curly brackets: “{ }”.

Note that the whole JSON is an object itself, so it is enclosed by curly brackets too.

There is one more possible type: **JSON arrays**.

A JSON array is a list of ordered, unnamed elements. In other words, a list of *values* without *keys*.

The “Tags” variable in the previous example is a JSON array.

Arrays are enclosed by square brackets: “[ ]”.

JSON objects and arrays can contain any number of elements, including other objects and arrays.

The above example is simple, but JSON structures can also be very complex with tens or hundreds of nested elements.

Note:

A JSON array is a perfectly valid JSON by itself, even if it is not inside an object.

For example, this is a valid JSON:

```
[
  "Apple",
  "Orange",
  "Kiwi"
]
```

How can you use JSON?

JSON is very popular among web developers. Indeed, most of today’s web applications use JSON to send and receive data.

For example, libraries like Angular.JS and Node.JS use JSON to exchange data with the back-end.

One of the reasons why JSON is widely used is because it is very lightweight and easy to read.

Indeed, when you look at the previous JSON example, you can easily understand how the data is structured.

As a comparison, this is how the same data is represented in the more complex XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<Element>
  <Name>Alex</Name>
  <Age>37</Age>
  <Admin>true</Admin>
  <Contact>
    <Site>alexwebdevelop.com</Site>
    <Phone>123456789</Phone>
    <Address></Address>
  </Contact>
  <Tags>
    <Tag>php</Tag>
    <Tag>web</Tag>
    <Tag>dev</Tag>
  </Tags>
</Element>
```

JSON is much more readable, isn’t it?

So, JSON is the format used by front-end apps and by back-end systems (including PHP) to talk to each other.

But JSON has other uses, too.

For example:

- exchange data over the Internet between HTTP services
- use online platforms like cloud services, SMS gateways and more
- **create modern APIs for your clients**

Therefore, it’s important for a PHP developer like you to learn how to handle it.

The good news is: it’s really easy.

There are three basic operations you need to learn:

1. **Create (or *encode*)** a JSON object.
2. **Send** a JSON object to a front-end app or to a remote service.
3. **Decode** a JSON object received by your PHP script.

Let’s start with the encoding step.

Chapter 2
JSON encoding

Creating a JSON object with PHP is simple:

You just need to use the *json_encode()* function.

Let’s see how to do it in practice with a few examples.

In PHP, JSON objects are string variables.

So, *in theory*, you could create a JSON string like this:

```
/* A JSON object as a PHP string. */
$json =
'
{
  "Name": "Alex",
  "Age": 37,
  "Admin": true
}
';
```

However, creating a JSON string like that is not very handy.

Especially if the JSON structure is complex.

Fortunately, you don’t have to do that.

A much better solution is to **encode a PHP variable into a JSON object** using the *json_encode()* function.

json_encode() takes care of building the JSON structure for you and returns the JSON object as a string.

The best way to create a JSON object is to start from a **PHP array**.

The reason is that PHP arrays are a perfect match for the JSON structure: each PHP array *key* => *value* pair becomes a *key* => *value* pair inside the JSON object.

For example:

```
/* The PHP array. */
$array = array("Product" => "Coffee", "Price" => 1.5);

/* The JSON string created from the array. */
$json = json_encode($array);

echo $json;
```

The *$json* variable looks like this:

```
{"Product":"Coffee","Price":1.5}
```

json_encode() takes three arguments:

1. The **variable to be encoded** as a JSON object. (*$array*, in the previous example).
2. A list of **encoding options**, which we will cover in the next chapter.
3. The maximum nesting depth of the JSON. You can ignore this, unless you need to work with huge JSON objects.

You will learn about the encoding options in the next chapter.

But there is one option that I want you to use now: **JSON_PRETTY_PRINT**.

This option makes the output JSON more readable by adding some spaces. This way, you can print it nicely within *<pre>* tags and make it easier to read.

This is how it works:

```
/* The PHP array. */
$array = array("Product" => "Coffee", "Price" => 1.5);

/* The JSON string created from the array, using the JSON_PRETTY_PRINT option.
*/
$json = json_encode($array, JSON_PRETTY_PRINT);

echo '<pre>';
echo $json;
echo '</pre>';
```

Now, the output is more human-friendly:

```
{
  "Product": "Coffee",
  "Price": 1.5
}
```

Associative and numeric arrays

PHP **associative** arrays are encoded into JSON objects, like in the above example.

The elements of the PHP array become the elements of the JSON object.

If you want to create **JSON arrays** instead, you need to use PHP **numeric** arrays.

For example:

```
/* A PHP numeric array. */
$array = array("Coffee", "Chocolate", "Tea");

/* The JSON string created from the array. */
$json = json_encode($array, JSON_PRETTY_PRINT);

echo '<pre>';
echo $json;
echo '</pre>';
```

This time, the output is a JSON array (note the square brackets and the fact that there are no *keys*):

```
[
  "Coffee",
  "Chocolate",
  "Tea"
]
```

You can create nested JSON objects and arrays using PHP multi-dimensional arrays.

For example, this is how you can create the first example:

```
$array = array();

$array['Name'] = 'Alex';
$array['Age'] = 37;
$array['Admin'] = TRUE;

$array['Contact'] = array
(
  'Site' => "alexwebdevelop.com",
  'Phone' => 123456789,
  'Address' => NULL
);

$array['Tags'] = array('php', 'web', 'dev');

$json = json_encode($array, JSON_PRETTY_PRINT);

echo '<pre>';
echo $json;
echo '</pre>';
```

To recap:

- PHP associative arrays become JSON objects.
  (The key => values of the PHP array become the key => values of the JSON object.)
- PHP numeric arrays becomes JSON arrays.
- PHP multi-dimensional arrays become nested JSON objects or arrays.

Chapter 3
Encoding options

json_encode() supports 15 different encoding options.

Don’t worry… you don’t need to know them all.

But some of them can be useful.

In this chapter, I’m going to show you the ones you need to know and explain how they work (with examples).

The *json_encode()* function takes the variable to encode as first argument and a list of **encoding options** as second argument.

There are 15 different options you can use. Let’s look at the most useful ones.

You already used an encoding option in the last chapter: **JSON_PRETTY_PRINT**.

This option adds some white spaces in the JSON string, so that it becomes more readable when printed.

White spaces, as well as other “blank” characters like tabs and newlines, **have no special meaning inside a JSON object**.

In other words, this:

```
{
   "Product":   "Coffee",
   "Price":    1.5
}
```

has exactly the same value as this:

```
{"Product":"Coffee","Price":1.5}
```

Of course, spaces **do matter if they are inside variables**.

For example, the “Name 1” and “Name 2” variables in the following JSON are different:

```
{
  "Name 1": "My Name",
  "Name 2": "MyName"
}
```

If you want to use more options together, you need to separate them with a “|“.

(The technical reason is that the option argument is actually a bitmask).

For example, this is how you can use the JSON_PRETTY_PRINT, JSON_FORCE_OBJECT and JSON_THROW_ON_ERROR options together:

```
$array = array('key 1' => 10, 'key 2' => 20);

$json = json_encode($array, JSON_PRETTY_PRINT | JSON_FORCE_OBJECT |
JSON_THROW_ON_ERROR);
```

All right.

Now let’s look at the other *json_encode()* options.

- **JSON_FORCE_OBJECT**

Remember how PHP associative arrays are encoded into JSON objects, while numeric arrays are encoded into JSON arrays?

With this option, PHP arrays are always encoded into JSON objects regardless of their type.

By default, without this option, if you encode a numeric array you get a JSON array:

```
/* A PHP numeric array. */
$fruits = array('Apple', 'Banana', 'Coconut');

$json = json_encode($fruits , JSON_PRETTY_PRINT);

echo '</pre>';
echo $json;
echo '</pre>':
```

This is the output:

```
[
  "Apple",
  "Banana",
  "Coconut"
]
```

But if you use the JSON_FORCE_OBJECT option, the numeric array is encoded as a JSON object like this:

```
/* A PHP numeric array. */
$fruits = array('Apple', 'Banana', 'Coconut');

$json = json_encode($fruits , JSON_PRETTY_PRINT | JSON_FORCE_OBJECT);

echo '<pre>';
echo $json;
echo '</pre>';
```

```
{
  "0": "Apple",
  "1": "Banana",
  "2": "Coconut"
}
```

This option comes in handy when working with front-end apps or web services that accept JSON objects only.

The PHP array numeric keys (in this case: 0, 1 and 2) become the keys of JSON object.

But remember: JSON objects keys are always strings, even when they are created from a numeric array like in this case.

You can see that the keys are strings because they are enclosed by double quotes.

- **JSON_INVALID_UTF8_SUBSTITUTE**
- **JSON_INVALID_UTF8_IGNORE**
- **JSON_PARTIAL_OUTPUT_ON_ERROR**

JSON expects the strings to be encoded in UTF-8.

If you try encoding a string with invalid UTF-8 characters, *json_encode()* will fail and will return FALSE instead of the JSON string.

For example:

```
/* This generates an invalid character. */
$invalidChar = chr(193);

$array = array("Key 1" => 'A', "Key 2" => 'B', "Key 3" => $invalidChar);
$json = json_encode($array, JSON_PRETTY_PRINT);

if ($json === FALSE)
{
  echo 'Warning: json_encode() returned FALSE.';
}
else
{
  echo '<pre>';
  echo $json;
  echo '</pre>';
}
```

```
Warning: json_encode() returned FALSE.
```

If you set the JSON_INVALID_UTF8_SUBSTITUTE option, all invalid characters are replaced by a special “replacement” UTF8 character: “ufffd”.

This way, you can get a valid JSON object even if there are invalid characters somewhere:

```
$invalidChar = chr(193);

$array = array("Key 1" => 'A', "Key 2" => 'B', "Key 3" => $invalidChar);
$json = json_encode($array, JSON_PRETTY_PRINT | JSON_INVALID_UTF8_SUBSTITUTE);

if ($json === FALSE)
{
  echo 'Warning: json_encode() returned FALSE.';
}
else
{
  echo '<pre>';
  echo $json;
  echo '</pre>';
}
```

```
{
  "Key 1": "A",
  "Key 2": "B",
  "Key 3": "ufffd"
}
```

The JSON_INVALID_UTF8_IGNORE option has a similar effect.

The only difference is that the invalid characters are completely removed instead of being replaced:

```
$invalidChar = chr(193);

$array = array("Key 1" => 'A', "Key 2" => 'B', "Key 3" => $invalidChar);
$json = json_encode($array, JSON_PRETTY_PRINT | JSON_INVALID_UTF8_IGNORE);

if ($json === FALSE)
{
  echo 'Warning: json_encode() returned FALSE.';
}
else
{
  echo '<pre>';
  echo $json;
  echo '</pre>';
}
```

```
{
    "Key 1": "A",
    "Key 2": "B",
    "Key 3": ""
}
```

JSON_PARTIAL_OUTPUT_ON_ERROR is similar, too.

This option replaces invalid characters with NULL:

```
$invalidChar = chr(193);

$array = array("Key 1" => 'A', "Key 2" => 'B', "Key 3" => $invalidChar);
$json = json_encode($array, JSON_PRETTY_PRINT | JSON_PARTIAL_OUTPUT_ON_ERROR);

if ($json === FALSE)
{
  echo 'Warning: json_encode() returned FALSE.';
}
else
{
  echo '<pre>';
  echo $json;
  echo '</pre>';
 }
```

```
{
  "Key 1": "A",
  "Key 2": "B",
  "Key 3": null
}
```

- **JSON_NUMERIC_CHECK**

By default, all PHP strings are encoded as strings in the JSON object.

When the JSON_NUMERIC_CHECK option is set, *json_encode()* automatically encodes PHP numeric strings into JSON numbers instead of strings.

The following example shows the difference.

This is the default behavior:

```
$array = array(
  'String' => 'a string',
  'Numeric string 1' => '0',
  'Numeric string 2' => '1234',
  'Numeric string 3' => '1.103',
  'Numeric string 4' => '-0.3',
  'Numeric string 5' => '5e12'
);

$json = json_encode($array , JSON_PRETTY_PRINT);

echo '<pre>';
echo $json;
echo '</pre>';
```

```
{
  "String": "a string",
  "Numeric string 1": "0",
  "Numeric string 2": "1234",
  "Numeric string 3": "1.103",
  "Numeric string 4": "-0.3",
  "Numeric string 5": "5e12"
}
```

As you can see, all the values are strings (in double quotes).

If you set the JSON_NUMERIC_CHECK option, integer and float numeric strings become JSON numbers:

```
$array = array(
  'String' => 'a string',
  'Numeric string 1' => '0',
  'Numeric string 2' => '1234',
  'Numeric string 3' => '1.103',
  'Numeric string 4' => '-0.3',
  'Numeric string 5' => '5e12'
);

$json = json_encode($array , JSON_PRETTY_PRINT | JSON_NUMERIC_CHECK);

echo '<pre>';
echo $json;
echo '</pre>';
```

```
{
  "String": "a string",
  "Numeric string 1": 0,
  "Numeric string 2": 1234,
  "Numeric string 3": 1.103,
  "Numeric string 4": -0.3,
  "Numeric string 5": 5000000000000
}
```

- **JSON_THROW_ON_ERROR**

This option is available as of PHP 7.3.0.

So, if you have an older PHP version it will not work for you.

This option makes *json_encode()* throw a JsonException if an error occurs.

You will see how it works in practice in the “Validation and errors” chapter.

There are a few more *json_encode()* options.

However, they are more specific, and you will probably never use them.

Feel free to ask me about them in the comments if you want more details.

Chapter 4
Sending a JSON object

Now you know how to create a JSON object from a PHP array.

The next step is to send your JSON object to a front-end application or to a remote service.

In this chapter I’m going to show you exactly how to do that.

If you are creating a JSON object, it’s because you need to send it to a front-end application or to a remote service.

You can do that either as a **reply to a remote request**, or as a **direct HTTP request**.

Sending a JSON object as a reply to a remote request

This is the case when your PHP script receives a remote request and must reply with a JSON object.

For example, when a front-end app (running on the remote user's browser) sends a request to your PHP back-end, or when a remote HTTP service connects to your API script to retrieve some data.

When your PHP back-end receives the request, it prepares the response data and encodes it into a JSON object (as you have learned in the previous chapters).

To send the JSON object as a reply to the remote caller, you need to:

1. Set the JSON HTTP content-type: **application/json**.
2. Return the JSON as a string.

To set the content-type, you need to use the PHP header() function. Like this:

```
header('Content-Type: application/json');
```

Important:

You must call *header()* before sending any output to the browser.

That means that you cannot execute any *echo* statement before *header(), and* there must be no HTML code before the *<?php* tag. Empty lines are not allowed either.

After setting the content-type, you can return the JSON string:

```
/* Set the content-type. */
header('Content-Type: application/json');

/* The array with the data to return. */
$array = array("Coffee", "Chocolate", "Tea");

/* The JSON string created from the array. */
$json = json_encode($array);

/* Return the JSON string. */
echo $json;
```

You must not send anything else other than the content-type header and the *$json* string.

Sending a JSON object as a direct HTTP request

In the previous scenario, a front-end app or a remote service connects to your PHP back-end. Then, your back-end sends the JSON object as a reply.

In other contexts, your PHP script must be **the first** to send the JSON object.

In such cases, you need to **open an HTTP connection** and send the JSON data along with it.

You need to open a direct HTTP connection when you want to use a remote service, for example:

- when sending data to a cloud service such as an online storage space
- when using a service provider like a SMS gateway
- when using APIs provided by social networks or SAAS applications

and so on.

You can handle outbound HTTP connections using the PHP cURL library.

First, you need to initialize a cURL session with curl_init(), using the service URL as parameter:

```
/* The remote service URL. */
$url = 'https://remote-service.com';

/* The cURL session. */
$curl = curl_init($url);
```

Next, you need to set some cURL parameters with the curl_setopt() function:

- CURLOPT_POST, to tell cURL to send a POST HTTP request;
- CURLOPT_POSTFIELDS, to set the JSON object as the POST request content;
- CURLOPT_HTTPHEADER, to set the JSON content-type.

Like this:

```
/* Tell cURL to send a POST request. */
curl_setopt($curl, CURLOPT_POST, TRUE);

/* Set the JSON object as the POST content. */
curl_setopt($curl, CURLOPT_POSTFIELDS, $json);

/* Set the JSON content-type: application/json. */
curl_setopt($curl, CURLOPT_HTTPHEADER, array('Content-Type: application/json'));
```

Finally, you can send the request with curl_exec():

```
/* Send the request. */
curl_exec($curl);
```

For example, YouTube provides a data API to perform operations through HTTP calls.

One of such operations is to post a comment reply.

To do that, you need to send the following JSON object:

```
{
  "snippet": {
    "parentId": "YOUR_COMMENT_THREAD_ID",
    "textOriginal": "This is the original comment."
  }
}
```

Here is an example of how to do that.

(The YouTube API requires some authentication steps that are not reported here.)

```
/* Create the array with the comment data. */
$comment = array();
```

```
$comment['snippet'] = array(

  "parentId" => "YOUR_COMMENT_THREAD_ID",
  "textOriginal" => "This is the original comment."
);

/* Encode it into a JSON string. */
$json = json_encode($comment);


/* The YouTube API URL. */
$url = "https://www.googleapis.com/youtube/v3/comments?part=snippet&key=12345";

/* The cURL session. */
$curl = curl_init($url);

/* Tell cURL to send a POST request. */
curl_setopt($curl, CURLOPT_POST, TRUE);

/* Set the JSON object as the POST content. */
curl_setopt($curl, CURLOPT_POSTFIELDS, $json);

/* Set the JSON content-type: application/json. */
curl_setopt($curl, CURLOPT_HTTPHEADER, array('Content-Type: application/json'));

/* Send the request. */
$return = curl_exec($curl);

/* Print the API response. */
echo $return;
```

Chapter 5
JSON decoding

You know how to create and send JSON objects from your PHP script.

In this chapter, you are going to learn how to decode the JSON objects that your PHP application receives.

JSON is a data interchange format.

Just like you can send JSON objects to a front-end app or to a remote service, you can **receive JSON objects** from them as well.

In fact, most of the time you will receive a JSON object first and then send a JSON object as a reply.

After you receive a JSON object, you need to **decode it** to access the variables contained inside.

To do that, you need to use the *json_decode()* function.

json_decode(), as its name suggests, decodes a JSON string into a **PHP object or array**. All the variables contained in the JSON object will be available in the PHP object or array.

Here is how it works.

Let's take our first JSON object example:

```
$json =
'
{
  "Name": "Alex",
  "Age": 37,
  "Admin": true,
  "Contact": {
    "Site": "alexwebdevelop.com",
    "Phone": 123456789,
    "Address": null
  },
  "Tags": [
    "php",
    "web",
    "dev"
  ]
}
';
```

As long as the JSON is a string, there is no easy way to access all the variables contained in it.

This is where *json_decode()* comes into play.

By using *json_decode()*, you will be able to access all the variables as **object properties** or **array elements**.

By default, *json_decode()* returns a generic PHP object.

Each JSON variable is decoded according to these rules:

- JSON objects become PHP objects
- JSON arrays become PHP numeric arrays
- JSON strings become PHP strings
- JSON numbers become PHP integers or floats
- JSON null values become PHP null values

- JSON Boolean values become PHP Boolean values (*true* or *false*)

For example (using the above JSON):

```
$jsonData = json_decode($json);

echo '<pre>';
var_dump($jsonData);
echo '</pre>';
```

The output from the above code shows how the PHP object is created:

```
object(stdClass)#1 (5) {
  ["Name"]=>
  string(4) "Alex"
  ["Age"]=>
  int(37)
  ["Admin"]=>
  bool(true)
  ["Contact"]=>
  object(stdClass)#2 (3) {
    ["Site"]=>
    string(18) "alexwebdevelop.com"
    ["Phone"]=>
    int(123456789)
    ["Address"]=>
    NULL
  }
  ["Tags"]=>
  array(3) {
    [0]=>
    string(3) "php"
    [1]=>
    string(3) "web"
    [2]=>
    string(3) "dev"
  }
}
```

So, if you want to access the "Age" element of the JSON object, you can do it like this:

```
$jsonData = json_decode($json);

echo $jsonData->Age;
```

```
37
```

Note:

JSON objects are decoded into PHP objects. However, **JSON arrays are decoded into PHP numeric arrays**.

In the output, you can see how the *"Tags"* JSON array becomes a PHP numeric array.

Since *"Tags"* is a numeric array, you can iterate through its element using a foreach loop, like this:

```
$jsonData = json_decode($json);

foreach ($jsonData->Tags as $tag)
{
  echo $tag . "<br>";
}
```

In some cases, the names of the JSON variables cannot be used as names for PHP variables.

The reason is that JSON keys can contain any valid UTF8 characters, unlike PHP variable names. For example, PHP variables cannot contain the dash "-" character.

In these cases, you can access the variable inside the decoded object using this syntax:

```
$json =
'
{
  "Invalid-php-name": "Variable content"
}
';

$jsonData = json_decode($json);

echo $jsonData->{'Invalid-php-name'};
```

json_decode() options

The first *json_decode()* argument is the JSON string.

The second argument is a Boolean option called *$assoc*.

If this parameter is set to true, *json_decode()* decodes JSON objects into **PHP associative arrays** instead of PHP objects.

Let's see again the first JSON example.

This time, we set the the *$assoc* option to true:

```
/* The second argument is set to true. */
$jsonData = json_decode($json, TRUE);

echo '<pre>';
var_dump($jsonData);
echo '</pre>';
```

The output from the above code shows how the PHP associative array is created:

```
array(5) {
  ["Name"]=>
  string(4) "Alex"
  ["Age"]=>
  int(37)
  ["Admin"]=>
  bool(true)
  ["Contact"]=>
  array(3) {
    ["Site"]=>
    string(18) "alexwebdevelop.com"
    ["Phone"]=>
    int(123456789)
    ["Address"]=>
    NULL
  }
  ["Tags"]=>
  array(3) {
    [0]=>
    string(3) "php"
    [1]=>
    string(3) "web"
    [2]=>
    string(3) "dev"
  }
}
```

You can access the elements just like any array element:

```
$jsonData = json_decode($json, TRUE);

echo 'Name is: ' . $jsonData['Name'];
```

Note that JSON arrays are still decoded into PHP numeric arrays, just like in the previous case.

Again, you can see from the output that the *"Tags"* element is a PHP numeric array.

More decoding options

The third *json_decode()* argument is the recursion depth. Its default value is 512 and you can safely ignore it.

The fourth and last argument is a list of options, much like the second *json_encode()* argument. In fact, some of the options are the same.

Let's take a quick look.

- **JSON_OBJECT_AS_ARRAY**

This option has the same effect as setting the *$assoc* argument to true. It makes *json_decode()* return PHP associative arrays instead of PHP objects.

- **JSON_THROW_ON_ERROR**

This option makes *json_decode()* throw a JsonException if an error occurs.

You will see how it works in practice in the "Validation and errors" chapter.

- **JSON_INVALID_UTF8_IGNORE**

This option works as for *json_encode()*.

Normally, if the source JSON string contains an invalid character, *json_decode()* returns NULL.

For example, if you put an invalid UTF-8 character in the JSON string and you try decoding it, you get NULL in return:

```
$invalidChar = chr(193);

$json =
'
{
  "Valid char": "a",
  "Invalid char": "' . $invalidChar . '"
}
';

$jsonData = json_decode($json, TRUE);

echo '<pre>';
var_dump($jsonData);
echo '</pre>';
```

```
NULL
```

Enabling the JSON_INVALID_UTF8_IGNORE option makes *json_decode()* ignore invalid characters:

```
$invalidChar = chr(193);

$json =
'
{
  "Valid char": "a",
  "Invalid char": "' . $invalidChar . '"
}
';

$jsonData = json_decode($json, TRUE, 512, JSON_INVALID_UTF8_IGNORE);

echo '<pre>';
var_dump($jsonData);
```

```
echo '</pre>';
```

```
array(2) {
  ["Valid char"]=>
  string(1) "a"
  ["Invalid char"]=>
  string(0) ""
}
```

- **JSON_BIGINT_AS_STRING**

This option is useful when the JSON object contains very large integers.

When an integer exceeds the maximum PHP size, it is converted into a *float* and some precision is lost.

For example:

```
$json =
'
{
  "Small number": 10,
  "Big number": 1234567890123456789
}
';

$jsonData = json_decode($json, TRUE);

echo '<pre>';
var_dump($jsonData);
echo '</pre>';
```

You can see how, in the output array, the big integer is decoded into a float and some precision is lost:

```
array(2) {
  ["Small number"]=>
  int(10)
  ["Big number"]=>
  float(1.2345678901235E+18)
}
```

The JSON_BIGINT_AS_STRING makes *json_decode()* turn big integers into PHP strings, so you can handle them properly (for example, with the BCMath extension) without losing precision:

```
$json =
'
{
  "Small number": 10,
  "Big number": 1234567890123456789
}
```

```
';

$jsonData = json_decode($json, TRUE, 512, JSON_BIGINT_AS_STRING);

echo '<pre>';
var_dump($jsonData);
echo '</pre>';
```

```
array(2) {
  ["Small number"]=>
  int(10)
  ["Big number"]=>
  string(19) "1234567890123456789"
}
```

Chapter 6
Validation and errors

In this chapter you will learn:

- How to properly validate JSON objects and variables
- How to catch encoding and decoding errors

So, if you want your code to be secure and solid, be sure to read this chapter.

Variable validation is crucial for web security.

In your PHP applications, you must validate any *untrusted variable* before you can use it.

(This is one of the first things I teach in my PHP Security course).

JSON objects are no exception.

A JSON string received from a remote source **is not safe** until you validate it.

This is true for JSONs received from the request string, like front-end apps requests, as well as for those received from remote services.

When you receive a JSON object, you need to:

1. Make sure it is a **valid JSON string**, by checking **decoding errors**.
2. **Validate each variable** contained inside the JSON object.

JSON decoding errors

By default, *json_decode()* returns NULL if it cannot decode the provided JSON string.

So, to check that the *json_decode()* argument is a valid JSON, you can simply check that its return value is not NULL.

Like this:

```
/* An invalid JSON string. */
$json =
'
{
  "Invalid element (no value)"
}
';

$jsonData = json_decode($json);

if (is_null($jsonData))
{
  echo 'Error decoding JSON.';
}
```

Note:

Do not use the *"if (!$jsonData)"* syntax.

Why? Because if you decode an empty JSON string into an empty PHP array, this syntax will consider the empty JSON string as an invalid JSON.

For example, the following code will print the error message:

```
$json = '{ }';

$jsonData = json_decode($json, TRUE);

if (!$jsonData)
{
  echo 'Error decoding JSON.';
}
```

If the decode fails, you can get the error code using the *json_last_error()* function, and the error message using the *json_last_error_msg()* function:

```
if (is_null($jsonData))
{
  echo 'Error decoding JSON.<br>';
  echo 'Error number: ' . json_last_error() . '<br>';
  echo 'Error message: ' . json_last_error_msg();
}
```

JSON Exceptions

From PHP version 7.3.0, you can set the *json_decode()* JSON_THROW_ON_ERROR option.

This option makes *json_decode()* throw a JsonException on errors, instead of returning NULL.

In this case, you need to use the **try/catch** syntax.

You can get the error code and message directly from the JsonException object.

Here is an example:

```
/* An invalid JSON string. */
$json =
'
{
 "Invalid element (no value)"
}
';

try
{
 $jsonData = json_decode($json, FALSE, 512, JSON_THROW_ON_ERROR);
}
catch (JsonException $je)
{
 echo 'Error decoding JSON.<br>';
 echo 'Error number: ' . $je->getCode() . '<br>';
 echo 'Error message: ' . $je->getMessage();
}
```

JSON variables validation

After you have successfully decoded the JSON object, you need to validate each variable contained in it.

As an example, suppose that you expect a JSON object with two variables: a "Name" string variable and a "Price" float variable. Like this:

```
{
  "Name": "Irish coffee",
  "Price": 2.5
}
```

After you have decoded the JSON string into a PHP object or array, you need to check that:

- Both the "Name" and "Price" variables are set.

- The “Name” variable is a valid string. It must not contain invalid characters and its length must be valid.
- The “Price” variable is a valid float number. It must be a positive number lower than a maximum value.

Let’s see how it’s done in practice.

Let’s start from the “Name” variable:

```
/* Decode the JSON string into a PHP array. */
$jsonArray = json_decode($json, TRUE);

/* Check for decoding errors. */
if (is_null($jsonArray))
{
  echo 'Error decoding JSON.<br>';
  echo 'Error number: ' . json_last_error() . '<br>';
  echo 'Error message: ' . json_last_error_msg();
  die();
}


/* Check that the "Name" variable is set. */
if (!isset($jsonArray['Name']))
{
  echo 'Error: "Name" not set.';
  die();
}

/* Check that Name contains only printable characters. */
if (!ctype_print($jsonArray['Name']))
{
  echo 'Error: "Name" contains invalid characters.';
  die();
}

/* Check the Name length. */
$minLength = 2;
$maxLength = 16;
$nameLength = mb_strlen($jsonArray['Name']);

if (($nameLength < $minLength) || ($nameLength > $maxLength))
{
  echo 'Error: "Name" is too short or too long.';
  die();
}
```

(Of course, the exact validation steps depend on how your application is going to use the variable).

And this is how to validate the “Price” variable:

```
/* Check that the "Price" variable is set. */
if (!isset($jsonArray['Price']))
{
  echo 'Error: "Price" not set.';
  die();
}
```

```
/* Check that Price is a float. */
if (!is_numeric($jsonArray['Price']))
{
  echo 'Error: "Price" is not a number.';
  die();
}

/* Check that Price is positive and less that a maximum value. */
$maxPrice = 1000;

if (($jsonArray['Price'] <= 0) || ($jsonArray['Price'] > $maxPrice))
{
  echo 'Error: Price value is not valid.';
  die();
}
```

Note:

If you want to know more about float numbers validation, I explain how to properly validate float variables in this free lesson from my PHP Security course.

Example
Create a JSON object from database data

Web applications keep their data on the database.

You will often need to use that data to create your JSON objects.

In this example I'll show how to do just that.

In this example, you are going to write a simple PHP script that returns information about a music album.

The information is retrieved **from the database** and then returned as a JSON object.

This is how the final JSON looks like:

```
{
  "Title": "Aqualung",
  "Artist": "Jethro Tull",
  "Year": 1971,
  "Duration": 2599,
  "Tracks": [
    "Aqualung",
    "Cross-Eyed Mary",
    "Cheap Day Return",
    "Mother Goose",
    "Wond'ring Aloud",
    "Up to Me",
    "My God",
    "Hymn 43",
    "Slipstream",
    "Locomotive Breath",
    "Wind-Up"
  ]
}
```

The information is stored in two database tables.

The first table, named “albums”, contain the album name, artist, and year.

The second table, named “tracks”, contain the album track names and duration.

Here is the SQL code to create and populate the tables (click to expand):

albums table

tracks table

This script uses the PDO extension to connect to the database.

If you want to know more about PHP and MySQL, you can refer to this complete tutorial:

- How to use PHP with MySQL

Here is the PDO connection snippet (remember to change the connection parameters to suit your development environment):

```
/* The PDO object */
$pdo = NULL;

/* The connection string. */
$dsn = 'mysql:host=localhost;dbname=myschema';

/* Connection step. */
try
{
  $pdo = new PDO($dsn, 'root',  '');
  $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch (PDOException $e)
{
  die();
}
```

Now, you need to create the PHP associative array that will be encoded into the JSON object:

```
/* The PHP array with the data for the JSON object. */
$data = array();
```

Then, you need to select the music album from the database.

For this example, suppose that you need to retrieve the album with ID 1 (the one already present in the table, if you used the above SQL code).

Once you have the query result, you can get the album title, artist, and year.

Then, you can add them to the *$data* array.

Here is how to do it:

```
/* The album ID to get from the database. */
$albumId = 1;

/* Run the search query. */
$albumQuery = 'SELECT * FROM albums WHERE album_id = :album_id';
$albumParams = array(':album_id' => $albumId);

try
{
  $albumRes = $pdo->prepare($albumQuery);
  $albumRes->execute($albumParams);
}
catch (PDOException $e)
{
  die();
}

$albumRow = $albumRes->fetch(PDO::FETCH_ASSOC);

/* Save the information to the $data array. */
if (is_array($albumRow))
{
  $data['Title'] = $albumRow['album_name'];
  $data['Artist'] = $albumRow['album_artist'];
  $data['Year'] = intval($albumRow['album_year'], 10);
}
```

Next, you need to select the album tracks.

You will need to add all the track names to the *"Tracks"* JSON array.

Note: since *"Tracks"* is a JSON array, you need to use a PHP **numeric array**.

You also need to calculate the album duration to set the *"Duration"* variable. To do that, you can sum all the track lengths.

Here is the code:

```
/* Initialize the "Duration" element at 0. */
```

```
$data['Duration'] = 0;

/* Create the "Tracks" numeric array. */
$data['Tracks'] = array();

/*  Run the search query.
    Note: the result is ordered by track number.
*/
$tracksQuery = 'SELECT * FROM tracks WHERE track_album = :album_id ORDER BY
track_n ASC';
$tracksParams = array(':album_id' => $albumId);

try
{
  $tracksRes = $pdo->prepare($tracksQuery);
  $tracksRes->execute($tracksParams);
}
catch (PDOException $e)
{
  die();
}

while (is_array($tracksRow = $tracksRes->fetch(PDO::FETCH_ASSOC)))
{
  /* Add each track name to the "Tracks" numeric array. */
  $data['Tracks'][] = $tracksRow['track_name'];

  /* Add this track's length to the total album length. */
  $data['Duration'] += intval($tracksRow['track_length'], 10);
}
```

Finally, set the JSON content-type, create the JSON object and return it:

```
/* Create the JSON string. */
$json = json_encode($data, JSON_PRETTY_PRINT);

/* Set the JSON content-type. */
header('Content-Type: application/json');

/* Return the JSON string. */
echo $json;
```

Example
Send a JSON file as an email attachment

In this last example, you will:

- Save a JSON file on the local file system
- Send the JSON file as an email attachment

Let’s start with the JSON string from the previous example:

```
$json =
'
{
  "Title": "Aqualung",
  "Artist": "Jethro Tull",
  "Year": 1971,
  "Duration": 2599,
  "Tracks": [
    "Aqualung",
    "Cross-Eyed Mary",
    "Cheap Day Return",
    "Mother Goose",
    "Wond'ring Aloud",
    "Up to Me",
    "My God",
    "Hymn 43",
    "Slipstream",
    "Locomotive Breath",
    "Wind-Up"
  ]
}
';
```

The first thing you need to do is to save the JSON string as a *.json* file.

To do that, you need to:

1. Define the file system path where to save the file.
2. Define the file name.
3. Save the JSON string into the file.

Define the file path

If the JSON file should *not* be accessible to remote users, like in this case, you must save it **outside of the webserver root**.

“Outside the webserver root” means that you cannot access it with a remote HTTP request.

However, local PHP scripts will still be able to access it.

For example, if the webserver root is *“/var/www/public/”*, you can save the file inside “*/var/www/private/*“.

Let’s define a *$path* variable with the file path:

```
/* The path where to save the JSON file. */
$path = '/var/www/private/';
```

Define the file name and save the file

Next, you need to choose a file name. For example: *music.json*.

So, save the file name in the *$fileName* variable:

```
/* The JSON file name .*/
$fileName = 'music.json';
```

Now it’s time to save the JSON string to the file.

The simples way to do that is by using the *file_put_contents()* function.

This is how it’s done:

```
/* Save the file. */
if (file_put_contents($path . $fileName, $json) === FALSE)
{
  /* Error saving the file. */
  echo 'Error saving JSON file.';
  die();
}

/* Save OK. */
echo 'JSON file successfully saved.';
```

Send the email

To send emails with PHP, I highly suggest you use PHPMailer.

PHPMailer supports a lot of functionalities like attachments, HTML emails, SMTP settings and more. And it’s easy to use.

You can find all you need to get started in my PHPMailer complete tutorial.

So, let’s create an email:

```
use PHPMailerPHPMailerPHPMailer;
use PHPMailerPHPMailerException;
```

```
/* Create the PHPMailer object. */
$email = new PHPMailer(TRUE);

/* Set the mail sender. */
$mail->setFrom('me@mydomain.com');

/* Add the recipient. */
$mail->addAddress('you@yourdomain.com');

/* Set the subject. */
$mail->Subject = 'Hey, here is the music JSON file.';

/* Set the mail message body. */
$mail->Body = 'Hi there. Please find attached the JSON file with the music album
data.';
```

Now, attach the JSON file:

```
/* Add the JSON file as attachment. */
$mail->addAttachment($path . $fileName);
```

And finally, send the email:

```
/* Open the try/catch block. */
try
{
  /* Send the mail. */
  $mail->send();
}
catch (Exception $e)
{
  /* PHPMailer exception. */
  echo $e->errorMessage();
  die();
}
```

Conclusion

N

In this tutorial, you learned everything you need to use JSON objects with PHP.

What is your experience with JSON?

Are you going to use what you learned today, or do you need to use JSON objects in some other way?

Let me know by leaving a comment below.

Copyright notice

The images used in this post have been downloaded from Freepik.

https://alexwebdevelop.com/php-json-backend/

PHP JSON

last modified July 26, 2020

PHP JSON tutorial shows how to work with JSON in PHP.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easily read and written by humans and parsed and generated by machines. The `application/json` is the official Internet media type for JSON. The JSON filename extension is `.json`.

The `json_encode` function returns the JSON representation of the given value. The `json_decode` takes a JSON encoded string and converts it into a PHP variable.

PHP frameworks such as Symfony and Laravel have built-in methods that work with JSON.

PHP JSON encode

In the following example, we use the `json_encode` function.

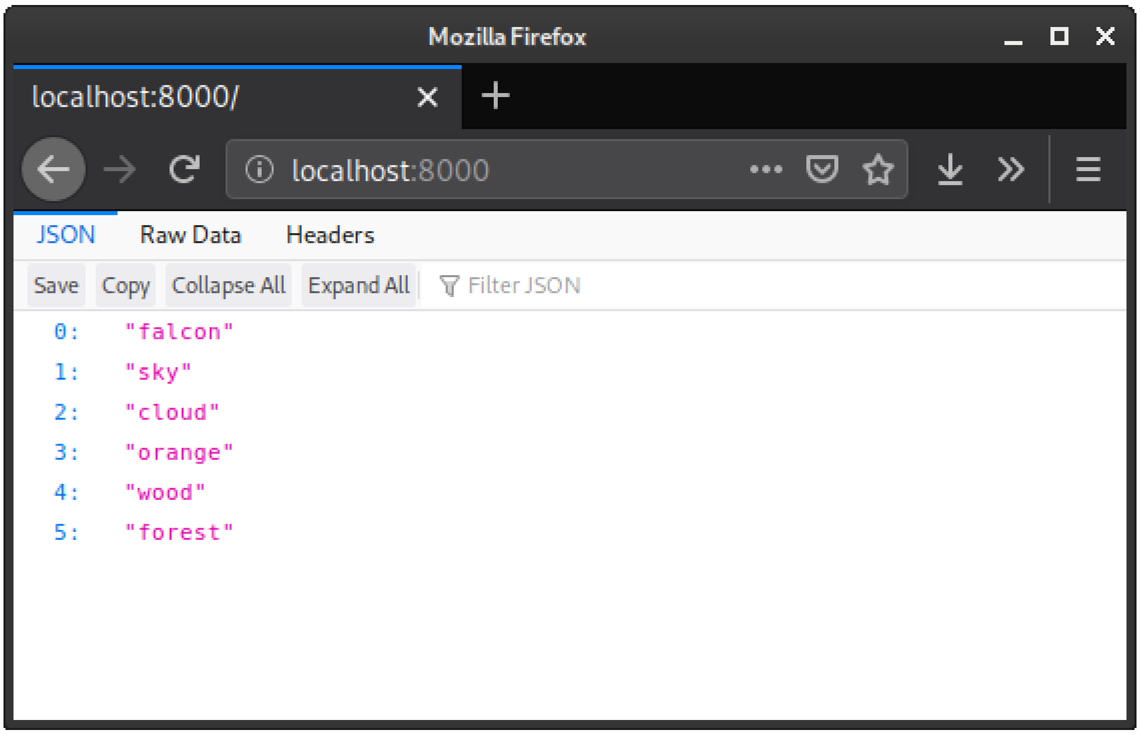
encode.php

```
<?php

$data = ["falcon", "sky", "cloud", "orange", "wood", "forest"];

header('Content-type:application/json;charset=utf-8');
echo json_encode($data);
```

The example transforms a PHP array into a JSON string.

```
$ php -S localhost:8000 encode.php
```

We start the server and locate to the `localhost:8000`.



```
0:    "falcon"
1:    "sky"
2:    "cloud"
3:    "orange"
4:    "wood"
5:    "forest"
```

Modern web browsers show JSON view for JSON data when they receive an appropriate content type in the header of the response.

PHP JSON decode

In the following example, we use the `json_decode` function.

decode.php

```
<?php

$data = '{
    "name": "John Doe",
    "occupation": "gardener"
}';

$a = json_decode($data, true);

header('Content-type:text/html;charset=utf-8');
echo "{$a["name"]} is a {$a["occupation"]}";
```

The example transforms a JSON string into a PHP variable.

```
$ php -S localhost:8000 decode.php
```

We start the server.

```
$ curl localhost:8000
John Doe is a gardener
```

We send a GET request with curl.

PHP JSON read from file

In the following example, we read JSON data from a file.

data.json

```
[
    {"name": "John Doe", "occupation": "gardener", "country": "USA"},
    {"name": "Richard Roe", "occupation": "driver", "country": "UK"},
    {"name": "Sibel Schumacher", "occupation": "architect", "country":
"Germany"},
    {"name": "Manuella Navarro", "occupation": "teacher", "country": "Spain"},
    {"name": "Thomas Dawn", "occupation": "teacher", "country": "New Zealand"},
    {"name": "Morris Holmes", "occupation": "programmer", "country": "Ireland"}
]
```

This is the JSON data.

readjson.php

```
<?php

$filename = 'data.json';

$data = file_get_contents($filename);
$users = json_decode($data);
?>

<html>
<table>
    <tbody>
        <tr>
            <th>Name</th>
            <th>Occupation</th>
            <th>Country</th>
        </tr>
        <?php foreach ($users as $user) { ?>
        <tr>
            <td> <?= $user->name; ?> </td>
            <td> <?= $user->occupation; ?> </td>
            <td> <?= $user->country; ?> </td>
        </tr>
        <?php } ?>
    </tbody>
</table>
</html>
```

In the code example, we read the file with `file_get_contents` and decode it into an PHP array with `json_decode`. Later, we place the data into a table utilizing PHP foreach loop.

PHP JSON read from database

In the following example, we read data from an SQLite database and return it in JSON.

cities.sql

```
BEGIN TRANSACTION;
DROP TABLE IF EXISTS cities;

CREATE TABLE cities(id INTEGER PRIMARY KEY, name TEXT, population INTEGER);
INSERT INTO cities(name, population) VALUES('Bratislava', 432000);
INSERT INTO cities(name, population) VALUES('Budapest', 1759000);
INSERT INTO cities(name, population) VALUES('Prague', 1280000);
INSERT INTO cities(name, population) VALUES('Warsaw', 1748000);
INSERT INTO cities(name, population) VALUES('Los Angeles', 3971000);
INSERT INTO cities(name, population) VALUES('New York', 8550000);
INSERT INTO cities(name, population) VALUES('Edinburgh', 464000);
INSERT INTO cities(name, population) VALUES('Berlin', 3671000);
COMMIT;
```

This SQL code creates a `cities` table in SQLite.

```
$ sqlite3 test.db
sqlite> .read cities.sql
sqlite> SELECT * FROM cities;
1|Bratislava|432000
2|Budapest|1759000
3|Prague|1280000
4|Warsaw|1748000
5|Los Angeles|3971000
6|New York|8550000
7|Edinburgh|464000
8|Berlin|3671000
```

With the `sqlite3` command line tool, we generate an SQLite database and create the `cities` table.

fetch_all.php

```
<?php

$db = new SQLite3('test.db');
$res = $db->query('SELECT * FROM cities');
$cities = [];

while ($row = $res->fetchArray()) {
    $cities[] = $row;
}

header('Content-type:application/json;charset=utf-8');
echo json_encode(['cities' => $cities]);
```

In the example, we retrieve the data from the database and return it as JSON.

PHP JSON and JS fetch API

In the following example, we use JavaScript fetch API to get the JSON data from a PHP script.

data.json

```
[
    {"name": "John Doe", "occupation": "gardener", "country": "USA"},
    {"name": "Richard Roe", "occupation": "driver", "country": "UK"},
    {"name": "Sibel Schumacher", "occupation": "architect", "country":
"Germany"},
    {"name": "Manuella Navarro", "occupation": "teacher", "country": "Spain"},
    {"name": "Thomas Dawn", "occupation": "teacher", "country": "New Zealand"},
    {"name": "Morris Holmes", "occupation": "programmer", "country": "Ireland"}
]
```

The JSON data is stored in a file.

data.php

```
<?php

$filename = 'data.json';

$data = file_get_contents($filename);
header('Content-type:application/json;charset=utf-8');
echo $data;
```

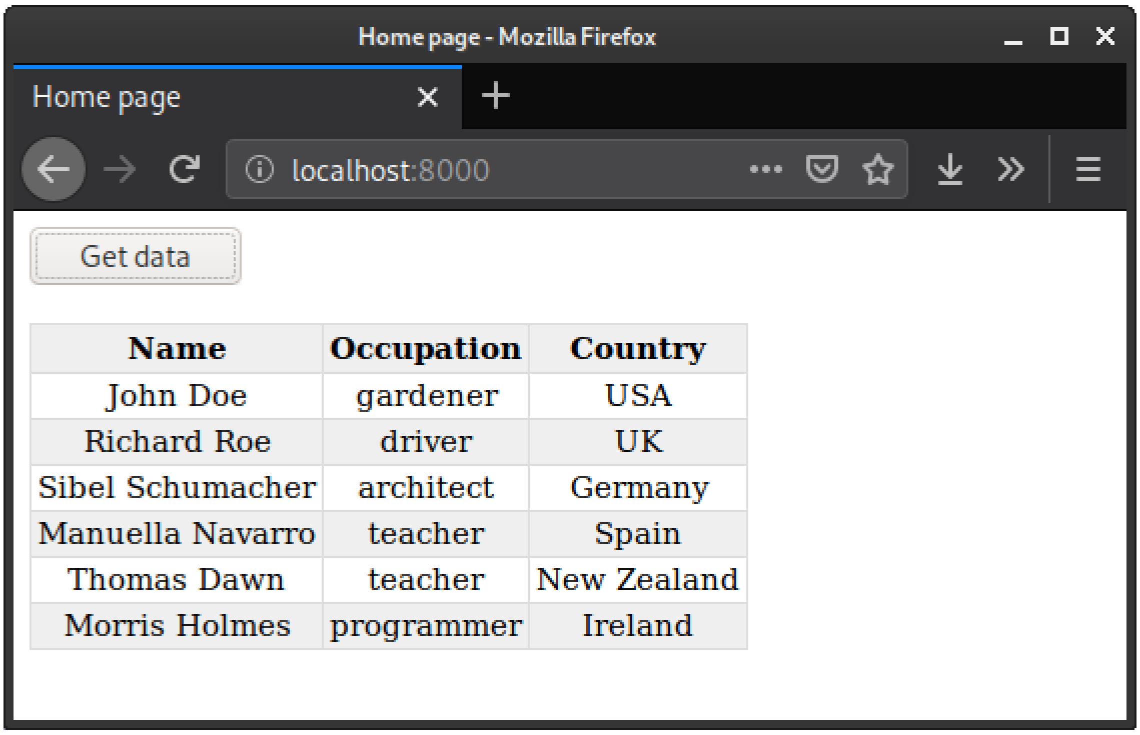
We read the data and return it to the client.

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home page</title>
    <style>
        th,
        td {
            font: 15px 'Segoe UI';
        }

        table,
        th,
        td {
            border: solid 1px #ddd;
            border-collapse: collapse;
            padding: 2px 3px;
            text-align: center;
        }
        tr:nth-child(odd) {background: #efefef}
        th {
            font-weight: bold;
        }
    </style>
</head>

<body>

    <button id="getData">Get data</button>
    <br>
    <br>
    <div id="output"></div>


    <script>

        const getBtn = document.getElementById('getData');
        const output = document.getElementById('output');
        const table = document.getElementById('table');

        getBtn.addEventListener('click', () => {

            fetch('/data.php')
                .then((res) => {
                    return res.json();
                })
                .then((data) => {
```

```
                output.innerHTML = '';
                let table = createTableAndHeader();

                output.appendChild(table);
                appendRows(table, data);
            })
            .catch((err) => {
                console.log("error fetching data");
                console.log(err);
            })

        });

        function createTableAndHeader() {

            let table = document.createElement("table");

            let tr = table.insertRow(-1);
            let headers = ["Name", "Occupation", "Country"];

            for (let i = 0; i < 3; i++) {

                let th = document.createElement("th");
                th.innerHTML = headers[i];
                tr.appendChild(th);
            }

            return table;
        }

        function appendRows(table, data) {

            for (let i = 0; i < data.length; i++) {

                let tr = table.insertRow(-1);

                for (const [_, value] of Object.entries(data[i])) {

                    let cell = tr.insertCell(-1);
                    cell.innerHTML = value;
                }
            }
        }
    </script>
</body>

</html>
```

We have a button in the document. When we click on the button, the `fetch` function retrieves JSON data from the `data.php` script. An HTML table is dynamically built and filled with the data.



Name	Occupation	Country
John Doe	gardener	USA
Richard Roe	driver	UK
Sibel Schumacher	architect	Germany
Manuella Navarro	teacher	Spain
Thomas Dawn	teacher	New Zealand
Morris Holmes	programmer	Ireland

PHP JSON in Slim

In the following example, we return JSON data from a Slim application.

```
$ composer req slim/slim
$ composer req slim/psr7
$ composer req slim/http
```

We install `slim/slim`, `slim/psr7`, and `slim/http` packages.

public/index.php

```
<?php

use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;
use Slim\Factory\AppFactory;

require __DIR__ . '/../vendor/autoload.php';

$app = AppFactory::create();

$app->get('/city', function (Request $request, Response $response): Response {

    $cities = [
        ["id" => 1, "name" => "Bratislava", "population" => 432000],
        ["id" => 2, "name" => "Budapest", "population" => 1759000],
        ["id" => 3, "name" => "Prague", "population" => 1280000],
        ["id" => 4, "name" => "Warsaw", "population" => 1748000],
        ["id" => 5, "name" => "Los Angeles", "population" => 3971000],
        ["id" => 6, "name" => "New York", "population" => 8550000],
        ["id" => 7, "name" => "Edinburgh", "population" => 464000],
        ["id" => 8, "name" => "Berlin", "population" => 3671000],
    ];

    return $response->withJson(["cities" => $cities]);
});
```

```
$app->run();
```

We use the `withJson` method of the `Response` to return JSON data.

PHP JSON in Symfony

In the following example, we send a JSON response from a Symfony application.

```
$ symfony new symjson
$ cd symjson
```

A new application is created.

```
$ composer req annot
$ composer req symfony/orm-pack
$ composer req symfony/console
```

We install the `annot`, `symfony/orm-pack`, and `symfony/console` components.

.env

```
...
DATABASE_URL=sqlite:///%kernel.project_dir%/var/ydb.db
...
```

We set up a database URL for the SQLite database.

```
$ php bin/console doctrine:database:create
```

We create the database.

src/Command/CreateCityTable.php

```
<?php

namespace App\Command;

use Doctrine\DBAL\Driver\Connection;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;


class CreateCityTable extends Command
{
    private $conn;

    public function __construct(Connection $conn)
    {
        $this->conn = $conn;

        parent::__construct();
    }

    protected function configure()
    {
        $this->setName('app:create-table')
            ->setDescription('Creates City table')
            ->setHelp('This command creates a City table');
    }
    protected function execute(InputInterface $input, OutputInterface $output):
int
    {
        $q1 = "CREATE TABLE cities(id INTEGER PRIMARY KEY, name TEXT, population
INTEGER)";
        $this->conn->executeQuery($q1);

        $q2 = "INSERT INTO cities(name, population) VALUES('Bratislava',
432000)";
        $this->conn->executeQuery($q2);

        $q3 = "INSERT INTO cities(name, population) VALUES('Budapest',
1759000)";
        $this->conn->executeQuery($q3);

        $q4 = "INSERT INTO cities(name, population) VALUES('Prague', 1280000)";
        $this->conn->executeQuery($q4);

        $q5 = "INSERT INTO cities(name, population) VALUES('Warsaw', 1748000)";
        $this->conn->executeQuery($q5);

        $q6 = "INSERT INTO cities(name, population) VALUES('Los Angeles',
3971000)";
        $this->conn->executeQuery($q6);

        $q7 = "INSERT INTO cities(name, population) VALUES('New York',
8550000)";
        $this->conn->executeQuery($q7);

        $q8 = "INSERT INTO cities(name, population) VALUES('Edinburgh',
464000)";
        $this->conn->executeQuery($q8);

        $q9 = "INSERT INTO cities(name, population) VALUES('Berlin', 3671000)";
        $this->conn->executeQuery($q9);

        $output->writeln('table successfully created');

        return Command::SUCCESS;
    }
}
```

This is a command that creates the `cities` table in the database.

```
$ php bin/console app:create-table
```

The command is executed.

src/Repository/CityRepository.php

```
<?php

namespace App\Repository;

use Doctrine\DBAL\Driver\Connection;

class CityRepository
{
    protected $conn;

    public function __construct(Connection $conn)
    {
        $this->conn = $conn;
    }

    public function all(): array
    {
        $queryBuilder = $this->conn->createQueryBuilder();
        $queryBuilder->select('*')->from('cities');
        $stm = $queryBuilder->execute();

        return $stm->fetchAll();
    }
}
```

In the `CityRepository`, we have the `all` method which retrieves all rows. We use the Doctrine DBAL to execute the database query.

src/Controller/CityController.php

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Routing\Annotation\Route;
use App\Repository\CityRepository;

class CityController extends AbstractController
{
    /**
     * @Route("/city", name="city")
     * @param CityRepository $cityRepository
     * @return JsonResponse
     */
    public function index(CityRepository $cityRepository): JsonResponse
    {
        $cities = $cityRepository->all();

        return $this->json([
            'cities' => $cities
        ]);
    }
}
```

Inside the controller, we fetch all data using the `CityRepository`. The data transformed into JSON with the `json` helper.

```
$ symfony serve
```

We start the web server and locate to the `localhost:8000/city`.

PHP JSON in Laravel

In the following example, we send a JSON response from a Laravel application.

```
$ laravel new larajson
$ cd larajson
```

We create a new Laravel application.

.env

```
...
DB_CONNECTION=sqlite
DB_DATABASE=/tmp/ydb.db
...
```

We define the connection and the database file.

```
$ touch /tmp/ydb.db
```

We create the database file.

```
$ php artisan make:migration create_cities_table
Created Migration: 2020_07_25_101535_create_cities_table
```

A new migration for the `cities` table is created.

database/migrations/2020_07_25_101535_create_cities_table.php

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateCitiesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('cities', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->integer('population');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('cities');
    }
}
```

In the migration file, we create a schema for the `cities` table.

```
$ php artisan migrate
```

We run the migrations.

```
$ php artisan make:seeder CitySeeder
```

We create a data seeder for our database.

database/seeds/CitySeeder.php

```
<?php

use Illuminate\Database\Seeder;

class CitySeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('cities')->insert([
            ['name' => 'Bratislava', 'population' => 432000],
            ["name" => "Budapest", "population" => 1759000],
            ["name" => "Prague", "population" => 1280000],
            ["name" => "Warsaw", "population" => 1748000],
            ["name" => "Los Angeles", "population" => 3971000],
            ["name" => "New York", "population" => 8550000],
            ["name" => "Edinburgh", "population" => 464000],
            ["name" => "Berlin", "population" => 3671000]
        ]);
    }
}
```

The seeder inserts eight rows into the table.

```
$ php artisan db:seed --class=CitySeeder
```

We execute the seeder.

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;
use Illuminate\Http\JsonResponse;

Route::get('/city', function (): JsonResponse {

    $cities = DB::table('cities')->get();

    return response()->json([
        'cities' => $cities
    ]);
});
```

Inside the route, we select all data from the database and return it as JSON with the `json` helper.

```
$ php artisan serve
```

We run the web server.

```
$ curl localhost:8000/city
{"cities":[{"id":"1","name":"Bratislava","population":"432000"},
{"id":"2","name":"Budapest","population":"1759000"},
{"id":"3","name":"Prague","population":"1280000"},
{"id":"4","name":"Warsaw","population":"1748000"},
{"id":"5","name":"Los Angeles","population":"3971000"},
{"id":"6","name":"New York","population":"8550000"},
{"id":"7","name":"Edinburgh","population":"464000"},
{"id":"8","name":"Berlin","population":"3671000"}]}
```

We get the data with the `curl` tool.

PHP JSON from client

Symfony provides the `HttpClient` component which enables us to create HTTP requests in PHP. We can also send JSON data.

```
$ composer req symfony/http-client
```

We install the `symfony/http-client` component.

send_json_req.php

```
<?php

require('vendor/autoload.php');

use Symfony\Component\HttpClient\HttpClient;

$httpClient = HttpClient::create();
$response = $httpClient->request('GET', 'http://localhost:8000/greet',  [
    'json' => [
        'name' => 'Lucia',
        'message' => 'Cau'
    ]
]);

$content = $response->getContent();
echo $content . "\n";
```

The example sends a GET request with a JSON payload.

src/Controller/GreetController.php

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

class GreetController extends AbstractController
{
    /**
     * @Route("/greet", name="greet")
     */
    public function index(Request $request): Response
    {
        $params = [];
        if ($data = $request->getContent()) {
            $params = json_decode($data, true);
        }

        $msg = "{$params['name']} says {$params['message']}";

        $textResponse = new Response($msg , 200);
        $textResponse->headers->set('Content-Type', 'text/plain');

        return $textResponse;
    }
}
```

In the `GreetController`, we process the JSON data and generate a text message, which is returned to the client.

```
$ php send_json_req.php
Lucia says Cau
```

This is the output.

In this tutorial, we have worked with JSON data in plain PHP, Symfony, Slim, and Laravel.

List all PHP tutorials.

https://zetcode.com/php/json/