# HTTP and HTML

Problems for Exercise for the [“PHP Web Development Basics” course @ SoftUni](https://softuni.bg/trainings/2163/php-web-development-basics-september2018#lesson-9661).

**Install** “[Postman](https://www.getpostman.com/)” REST Client to **ease** your task.

You can check your solutions here[: https://judge.softuni.bg/Contests/1263/HTTP-and-HTML-Basic-Lab](:%20https:/judge.softuni.bg/Contests/1263/HTTP-and-HTML-Basic-Lab).

# Part I: HTTP

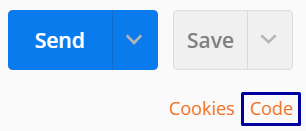
## GitHub Repos for User "SoftUni-Tech-Module"

Use POSTMAN to send a **"GET"** request to receive all the repositories, after that all you have to do is **copy the URL of the request** and paste it as a solution in judge.

Don’t forget that you should use the API provided from GitHub: **https://api.github.com**

## GitHub Issue by id

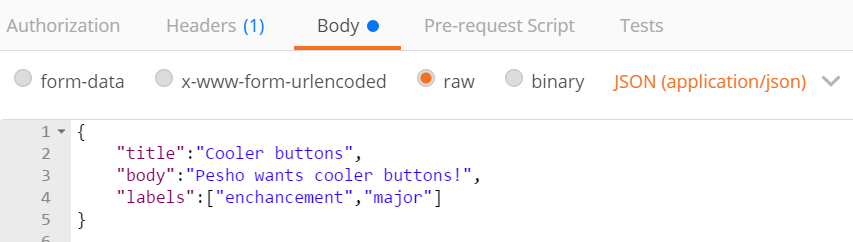
Use POSTMAN to send a **"GET"** request to receive an Issue by id. Try getting issue with id 1 in repo "TestRepository" in "SoftUni-Tech-Module". Don't forget that you should use the API provided from GitHub: [**https://api.github.com**](https://api.github.com)**.**



**Post the HTTP code as a solution in judge.**

## GitHub Issue

Use POSTMAN to send a "POST" request to the server with the following JSON as body (send it as **application/JSON)**:



You need to use your GitHub **account credentials** to submit issues. Under the Authorization tab, select Basic and enter your username and password. Send the request to the URI [**https://api.github.com/repos/SoftUni-Tech-Module/TestRepository/issues**](https://api.github.com/repos/SoftUni-Tech-Module/TestRepository/issues)

**Post the RESPONSE BODY as a solution in judge.**

## Patch Issue

After creating an Issue, mark down the id of your created Issue and send "PATCH" request, changing the title to "Ugly buttons".

**Paste the RESPONSE BODY as a solution in judge.**

# Part II. HTML5

## Welcome to HTML

Create a Web page, which looks like the screenshot:



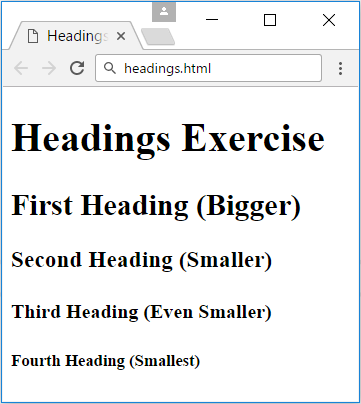
* *File name:* welcome.html
* *Title:* Welcome
* *Paragraph of text:* I am learning **HTML** and **CSS**!

### Hints

* Modify the code from the examples in the slides.

## Headings

Create a Web page like the screenshot:



* *File name*: headings.html
* *Title:* Headings
* *Large heading:* “**Headings Exercise**”
* *Smaller headings:* [**First**…**Fourth**] **Heading**

### Hints

* Use <h1> through <h5> tags.

## Paragraphs

Create a Web page like the screenshot:



* *File name*: paragraphs.html
* *Title:* Paragraphs
* *Large heading:* “**Paragraphs**”
* *Emphasized text:* **First**, **Second** and **Third paragraph**
* **Empty line** between the **second** and **third** paragraph

### Hints

Use **<p>**, **<em>** and **<br/>** tags.

## My TODO List

Create a Web page like the screenshot:



* *File name:* todo.html
* *Title:* TODO List
* *Large heading:* My TODO List
* *Unordered list:* <ul>…</ul>
* *Three list elements:* **<li>…</li>**

### Hints

* Create an HTML file named todo.html like in the previous exercises.
* Assign the <title>…<title>.
* Use <ul>…</ul> to create an unordered list.
* Put each of the list items in a <li>…</li> tag.
* Use **&amp;** for the **ampersand** (**&**)
* Use &ndash; for the **long hyphen** (**–**).

## Hello HTML

Create a Web page like the screenshot:



* *File name:* hello.html
* *Title:* “**Hello HTML**”
* *Large heading:* “**Hello HTML!**”
* *Paragraph of text:* I am <**your name** (**bold**)>. I am from <**your town as link to your town's Web site**>.
* *Paragraph of text:* I study **<*specialty* (*italic*)>** at <**link to SoftUni**>.

### Hints

* Modify the code from the example in the slides.

## Website

Create three web pages connected to each other by local links like the screenshot below:



* *File names:* index.html, todo.html, hello.html
* *Titles:* **Home**, **TODO List**, **Hello**
* *Large headings:* **Home**, **My TODO List**, **Hello HTML!**
* *Unordered lists:* home.htmlandtodo.html
* *Hyperlinks:* index.html - **links** to todo.html and hello.html
* *Index hyperlinks:* todo.html and hello.html

### Hints

You can use the todo.html and hello.html pages that you have created for the previous exercises.

* Create index.html and use <ul>…</ul> to create an unordered list with the links
* Use <a href="">…</a> to create a hyperlink inside a <li>…</li> element
* Use href="todo.html" to connect index.html to todo.html and href="hello.html" to connect it with hello.html.
* Create “**back to home**”links following the same logic

## Forms

Create a Web page like the screenshot:



* *File name:* contact.html
* *Title:* “**Contact Us**”
* *Large heading:* “**Contact Us**”
* *Form: holding three elements*
* **Two** text boxes
* *Button:* with text “**Submit**”

### Hints

* For the form use **<form>…</form>**
* Inside the form place two textboxes using **<input type="text" />**
* Use **<input type="submit" value="Submit" />** for the button

## Dump Variable

Write a PHP script **DumpVariable.php** that declares a variable. If the variable is **numeric**, **print it** by the built-in function **var\_dump()**. If the variable is **not numeric**, **print its type** at the input. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| "hello" | string |
| 15 | int(15) |
| 1.234 | float(1.234) |
| array(1,2,3) | array |
| (object)[2,34] | object |

# Part IV: Try the Web

So far you might have understood how PHP’s I/O works. Once a WEB Server receives a request and needs to dispatch it to the PHP parser it carefully prepares the environment eligible for WEB development. One of the core problems is how to pass a user input to a PHP file, when it’s part of the WEB Environment. The HTTP Standard defines some Request ways. One of them is the **query string** and the another is the **FORM data.**

We will discuss the form data later, and stick to the query string right now.

The query string is all that part you might have already seen **after the question mark in the address bar**. For instance, <http://site.com/script.php?name=john&last_name=smith> has a query string, namely: “name=john&last\_name=smith”.

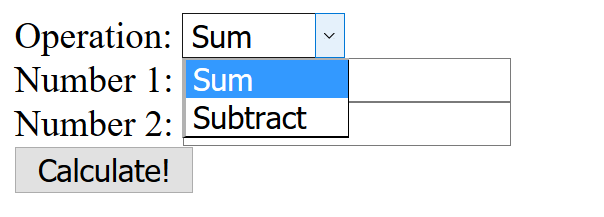
A part of the environment preparation is converting the query string to key/value pairs. You can find in the [$\_GET superglobal variable](http://php.net/manual/en/language.variables.superglobals.php) an associative array containing **“name” as key and “john” as its value**. Same applies for “last\_name” and “smith”.

The query string can be manually filled as in the above example or generated from FORM submission with method GET.

A **Form is an HTML element that contains a fieldset with name/value pairs** e.g. text fields, dropdowns, checkboxes and so forth. After the form is submitted the values of the fields are sent (along with their name attribute as keys) either as form data or query string depending on the “method” attribute of the form tag. The default method is “GET” which produces a query string.

## Calculate Two Numbers

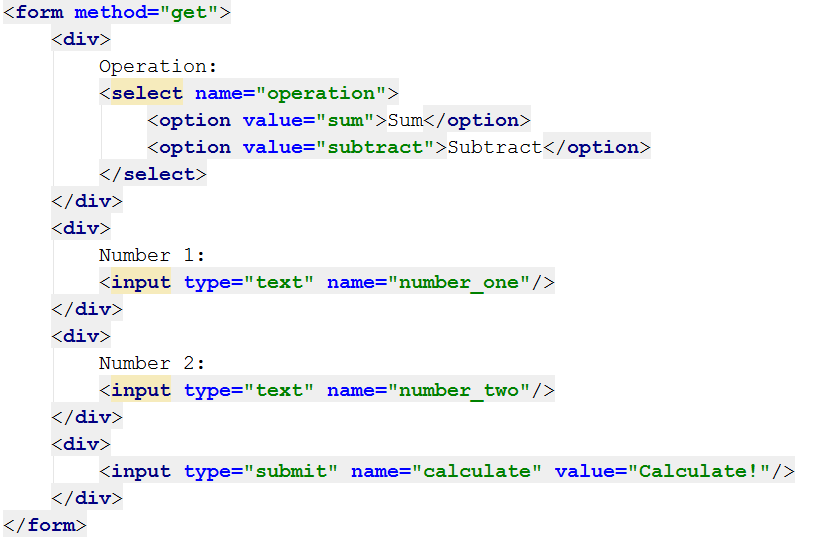
CLI is gone. No console anymore. We need a web interface. So let’s define one:



From the above web page, a user can choose an operation between Sum and Subtract from a dropdown, then enter 2 numbers in text fields and could click “Calculate!” button.

What we need to achieve is to take that input and perform the logical calculation.

Let’s see how the above web interface could be as HTML code:

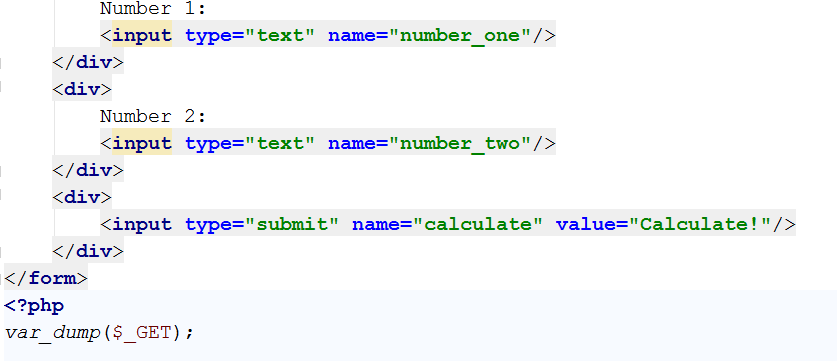


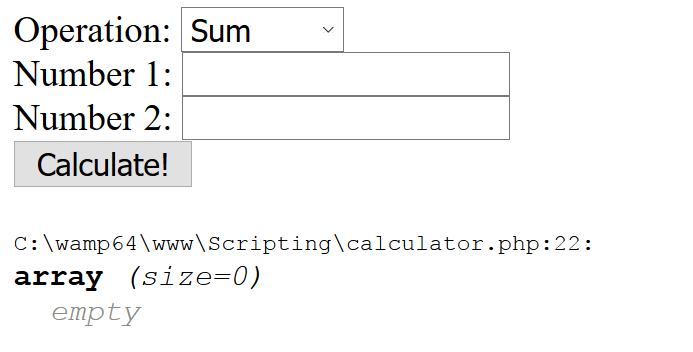
It defines a FORM with method GET (the method can be omitted, as GET is the default). The fieldset in the form is as follows:

* Dropdown named “operation” with values either “sum” or “subtract”
* Text field “number\_one” with user value
* Text field “number\_two” with user value
* Submit button

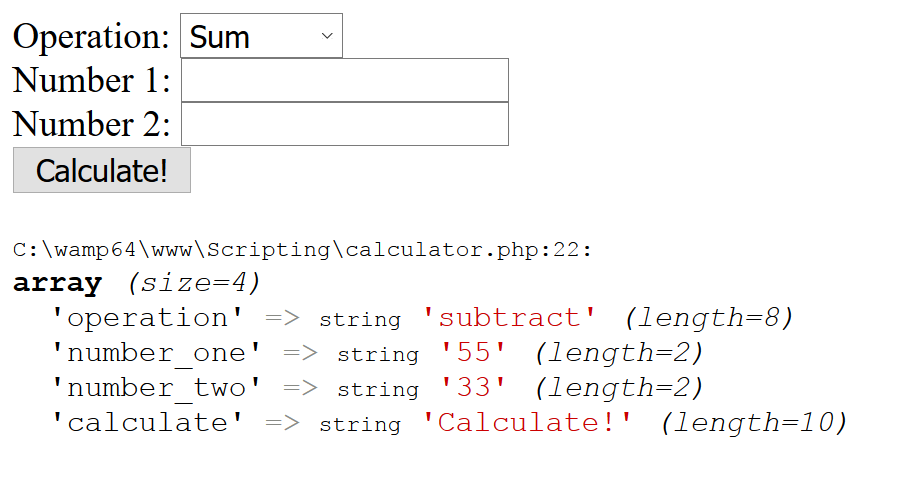
This means that after the form submission the key “operation” will be in the $\_GET superglobal with value either “sum” or “subtract” (to be honest, people can modify it, but we will talk about it later), as well as for “number\_one” and “number\_two”.

Let debug the $\_GET superglobal and open our PHP file in the web browser



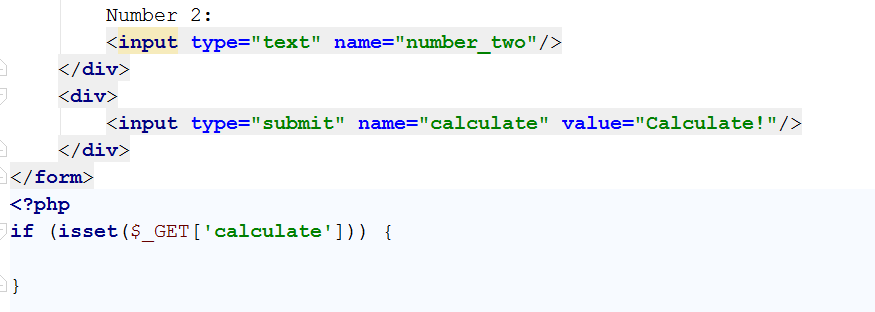


Let’s populate the form and click Calculate

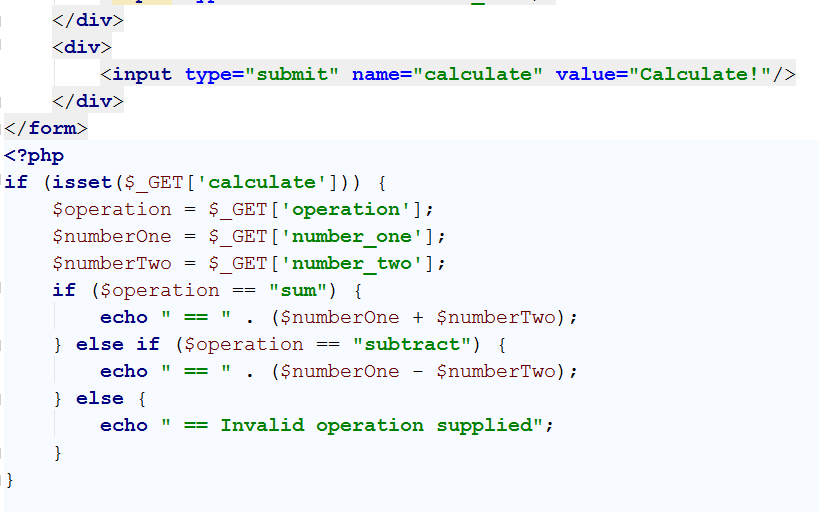


Everything seems OK, so we can now take the input from the $\_GET superglobal and create our logic

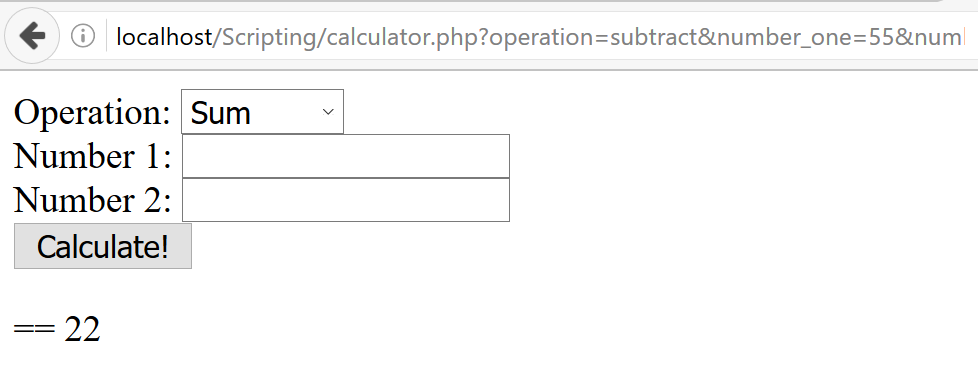
First of all, we need to make the calculation only when the submit button is clicked. We can determine that by the presence of its key in the $\_GET superglobal.



In the conditional statement body, we can now perform the core logic



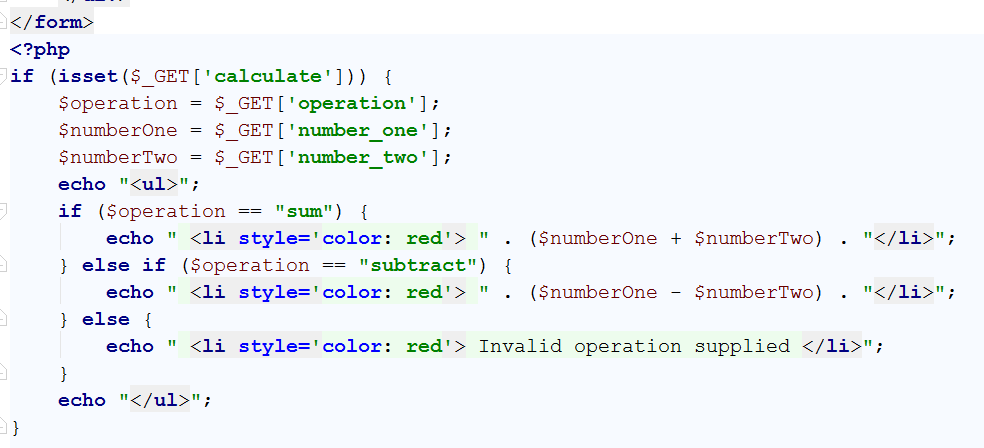
And the output of the previous input is:



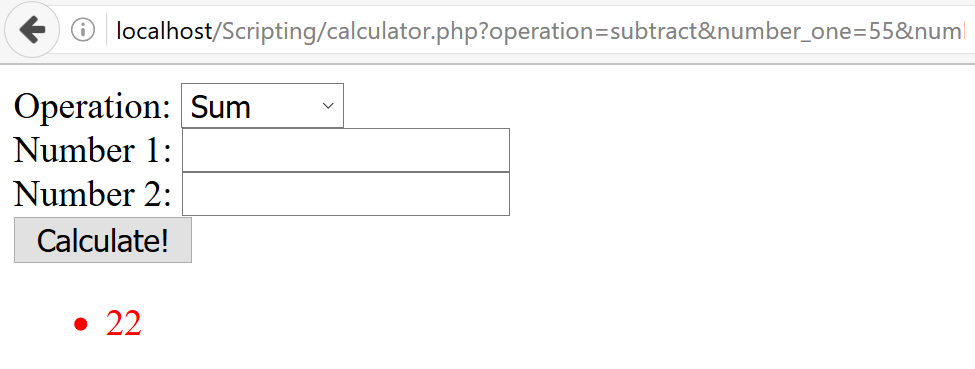
Whooila!

## Code Quality

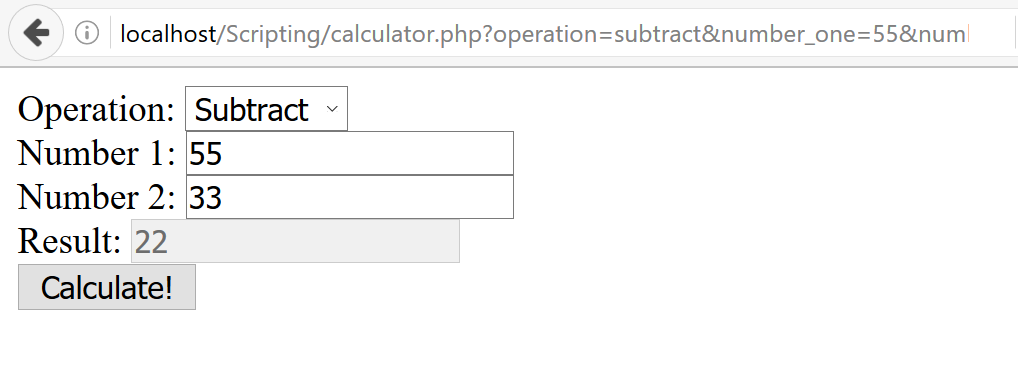
What if we want to output the result not prefixed by “==” but by a single bullet and in red color? Well, it’s not that hard. We just need to output the relevant HTML and the Web Browser will render it



And its respective output:



Good, eh? But **some of our HTML is wrapped in quotes, mixed in the PHP logic, making it harder to maintain**. It’s not a single <li>, but **what if it’s the whole of your Facebook wall**, with all the images, styles and videos? What if we want to output it to another text field (readonly) after the Number 2 field?



Well, we can write it that way:



But **now it’s totally unmaintainable**, with string concatenations and embedding big PHP logic between the HTML presentation.

The solution is called [Separation of Concerns](https://en.wikipedia.org/wiki/Separation_of_concerns).

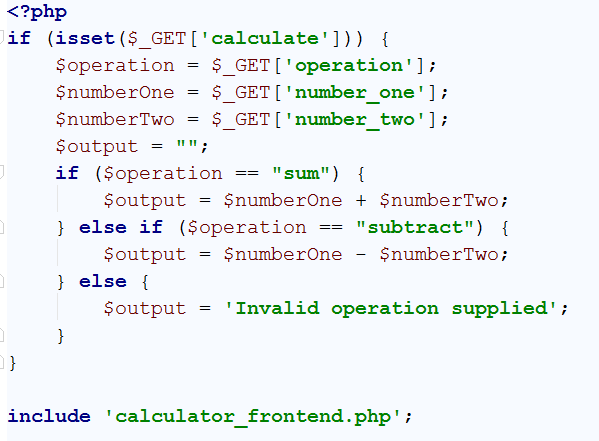
**Never mix HTML and PHP**! Separate the logic and the HTML and only add PHP to the HTML when you need to output value, not whole dynamic HTML.

Let’s create a PHP file that handles the $\_GET request and renders the HTML. Using the “[include](http://php.net/manual/en/function.include.php)” construction, one can embed one file’s content in another file.

So our files might be called:

* calculator.php
* calculator\_frontend.php

calculator.php:



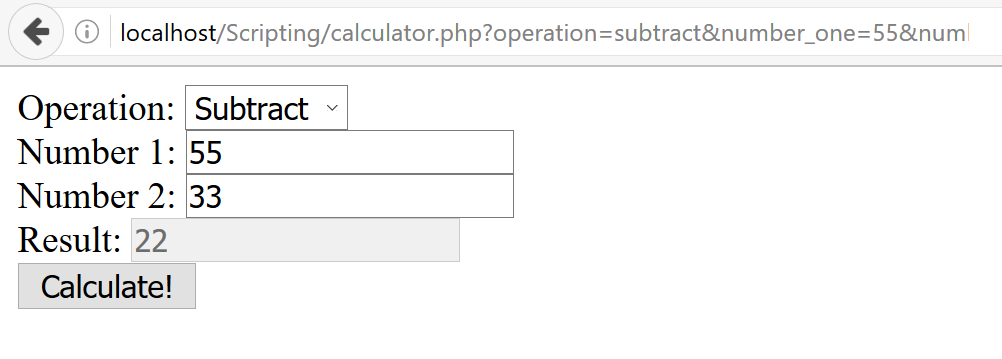
Assigns output to the $output variable and includes the content of the calculator\_fronend file, which should contain the HTML.

calculator\_frontend.php:



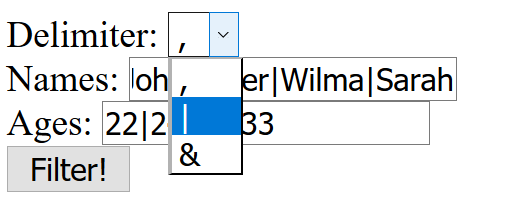
As you can see, **we have opened PHP only for conditional statement and for echo statement**. Once to check if **$output** is set and once to print the value of $output variable inside the HTML’s value attribute.

The output is just the same:



## Render Students Info

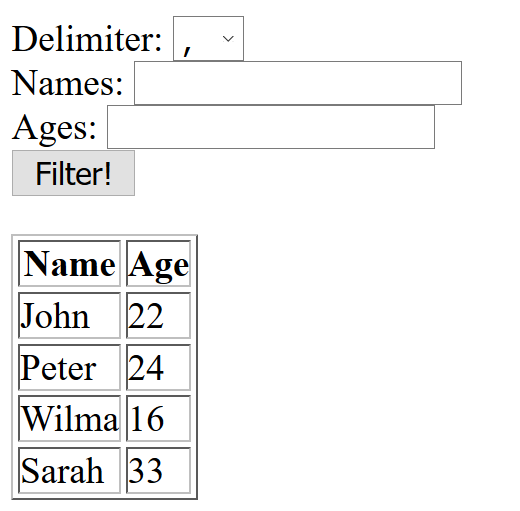
You are given an HTML form that contains three input fields. The first one is a dropdown containing delimiters (“,”, “|”, “&”). The second one containing student names separated by the selected delimiter. The third one – student ages separated by the selected delimiter.



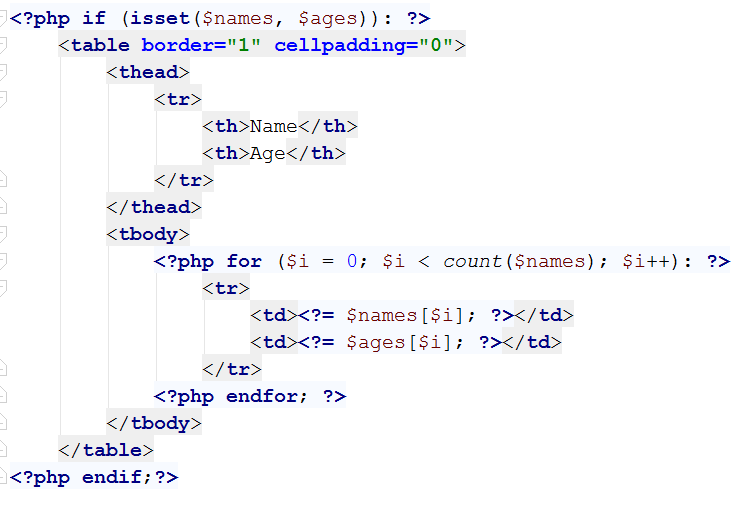
With the following HTML



You need to output an HTML table with headers “Name” and “Age” and each student as a row in this table



1. Separate the HTML and the PHP
2. Split the input from Names and Ages by the selected delimiter
3. Render a table only if names and ages arrays are initialized
4. Use foreach/endforeach or for/endfor to introduce new table rows



## Filter Legal Students

From the above task, render the HTML table only with these students are at least 18 years old.

## Paginate Students

Render max 5 students per page and two hyperlinks under the table – “Previous” and “Next”. The “Previous” hyperlink should not exist when there are no previous pages (e.g. we are on the first page) and the “Next” hyperlink should not exist when there are no next pages (e.g. when we are on the last page).

After clicking on “Next” hyperlink, the next chunk of 5 students is rendered. After clicking on “Previous” the previous chunk of 5 students is rendered.

HINT: Consider persisting the students’ collection between requests, you may send it back with the GET request not-preferable, but is the only one taught so far) or persist it in a session/cookie (or even a file).

## \*\* Paginate Students (2)

Decorate more the last task. Between the hyperlinks “Previous” and “Next” render hyperlinks with pages. Display exactly 3 hyperlinks with pages. But be careful! In some ways there might not be 3 pages until the end of the collection, so you need to render the previous pages. The only exception of the rule is when there aren’t 3 pages at all.

For instance, if we have 16 students, we will need 4 pages. This means on the first request we will render:

[1] [2] [3] [Next]

The red color means the page is the current page and cannot be clicked

The green color means the page is available for clicking

When clicking on “Next” or “2” the pager will be like:

[Previous] [2] [3] [4] [Next]

But when clicking on “3” there is no fifth page, so in order the slider to show exactly 3 pages, we need to make it remain the same way, just make “2” clickable in favor of “3”:

[Previous] [2] [3] [4] [Next]

The next step is to go to the last page. Clicking on “4” will make the slider look like:

[Previous] [2] [3] [4]

The exception of the rule is when we have for example 9 students. The only thing we can do here is to render 2 pages (which is normal, no results for 3 pages ☺):

[1] [2] [Next]

[Previous] [1] [2]

# Part III. Optional

## Firebase: All Books

Firebase is a cloud-based DB, **storage** and **app** platform (BaaS).

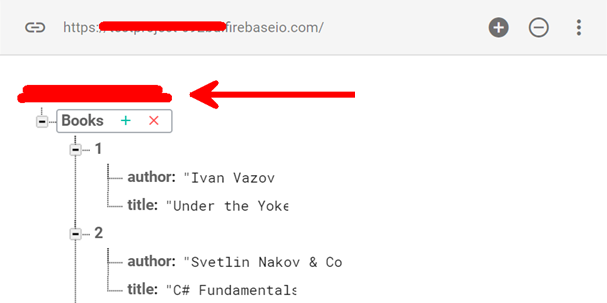
Register at: [**https://console.firebase.google.com**](https://console.firebase.google.com)**.**

Create a “**TestApp**” and in the create the **following** structure:

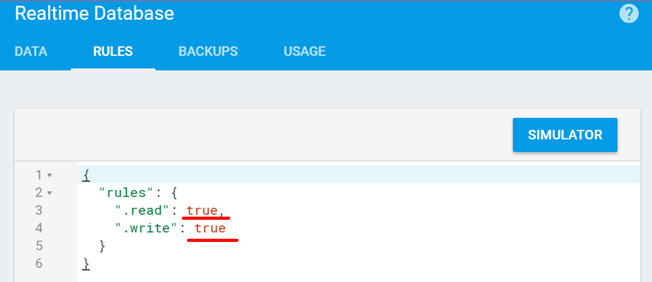


First task is to “**GET**” all books. To consume the request with **POSTMAN** your **url** should be the **following**: **https://{databaseId}.firebaseio.com/.json.**

**DatabaseId** is unique for every application. You can **find** yours from here:



We **should** also do one more configuration. Go to Database/Rules and set **.read** & **.write** actions to “**true**”. This will allow us to **send** request with **POSTMAN**. Beware that now everyone can **manipulate** our database and even **delete** it. (this is for **testing** purposes only).



## \*Firebase: Get Book #1

“**GET**” the Book with **id**: 1. Don’t forget the **.json** extension at the end (otherwise you will receive the whole **html**).

## \*Firebase: Create Book

To **create** a book, we will have to send a “**POST**” request and the JSON body should be in the **following** format:

{

"title":"New Title",

"author":"New Author"

}

## \*Firebase: Patch Book #7

The HTTP command “**PATCH**” **modifies** an existing HTTP **resource** (it can also create the resource if it does **not** exist). The JSON body should be in the **following** format:

{

"year": 1981,

"author": "Author Changed"

}

## \*Firebase: Change Book #7 Author

This time we have to execute a “**PUT**” command (the difference is that with “**PUT”** we can update a resource **partially**). In our case we have to **change** the author’s name to "**New author was assigned**".

**REQUEST**: **https://{databaseId}.firebaseio.com/Books/7/author.json**

The JSON body should be in the **following** format:

"**New author was assigned**".