MVC Design Pattern with PHP and SQlite

**Table of Contents**

[Part C — MVC Design Pattern with JavaScript, JQuery, AJAX, JSON, PHP and SQLite 2](#_Toc36130615)

[Introduction 2](#_Toc36130616)

[Technology Review 2](#_Toc36130617)

[SQLite 2](#_Toc36130618)

[JavaScript Object Notation (JSON) 3](#_Toc36130619)

[Asynchronous JavaScript and XML — AJAX 3](#_Toc36130620)

[JQuery 4](#_Toc36130621)

[Pre-requirements for Lab 8 4](#_Toc36130622)

[MVC Framework with JSON, AJAX, JQuery and PHP and SQLite 5](#_Toc36130623)

[Example 1 — Accessing JSON based Web Services using JQuery, AJAX and JavaScript 5](#_Toc36130624)

[Example 2 - Develop your own JSON based service and parse it through JQuery and AJAX 11](#_Toc36130625)

[Example 3 – Data retrieval by selection from SQLite using AJAX, JQuery and PHP 15](#_Toc36130626)

[Some Additional Exercises 26](#_Toc36130627)

[Appendix 28](#_Toc36130628)

[Appendix — Review results from Lab 7 28](#_Toc36130629)

# Part C — MVC Design Pattern with JavaScript, JQuery, AJAX, JSON, PHP and SQLite

## Introduction

The purpose of the previous Lab 7 Part A and Part B — MVC Design Pattern with PHP and SQLite — was to become familiar with MVC concepts by creating our own MVC framework. We continued our development in Part B by adding SQLite database connectivity. We created a database table, inserted new data and then retrieved and published already entered data from SQLite using the MVC framework.

For a review of Lab 7, see Appendix — Review results from Lab 7.

This Lab 8 tutorial continues to explore various uses of JavaScript, in particular the JQuery API with AJAX methods and PHP in your MVC framework.

There are many concepts covered in this Part C tutorial such as:

* Use of JQuery, AJAX and JSON to pass information via a service request or a service response without refreshing or leaving the webpage.
* Dealing with third party web services, such as Flickr as an example, to include third party data in our Web3D App site.
  + While Flickr API, per se, may not be that useful, you are doing a Virtual Museum for your 3D App, and some of the larger museums do have their own API. So, it is entirely possible that you could choose 4 objects from a museum with their own API and use that to access data. Evidence of such in your assignment would count towards a deeper understanding.
* Returning a JSON object from the back end and parsing it in a handler to get meaningful data for the front end.
* Use JSON for data transfer between your server and 3D App pages.

And, there are many other functionalities you could develop.

## Technology Review

Next, we will review, very briefly some of the technologies we are going to use in this Lab 8 Part C tutorial.

### SQLite

We have already covered SQLite in Lab 7, but just to remind you that SQLite is a ‘self-contained, embedded, full-featured, public-domain, SQL database engine — the most used database engine in the world’[[1]](#footnote-1). Because SQLite is self-contained, i.e. an executable file, we can create instances of it in your public\_html, which is easier to manage — further, ITS don’t offer us access to a MySQL instance.

### JavaScript Object Notation (JSON)

We have seen JSON files already, because we developed one in Lab 6, however in this tutorial we will read out data from the database using appropriate SQL methods and convert the resulting array of data into a JSON object using the PHP json\_encode() method and then echo that JSON object to a waiting JavaScript handler.

Have a read through this small description of JSON <http://en.wikipedia.org/wiki/JSON> to make sure you are more familiar with JSON, and I have also written a short description in the Canvas 3D Apps — Technology Stack — MVC, AJAX, JSON and Data Stores section.

A JSON file structure looks something like that shown in Figure 1. Recall, we did some work on such JSON files in Lab 6.



Figure 1: Example JSON file for illustration purposes only

JSON contains a readable and easy to understand text, that can be compressed for faster data transfer over the network. You can create such a JSON file structure by hand, but often such a structure is created programmatically to avoid mistakes — largely because a JSON syntax error fails silently! However, there are many JSON validators online with which to check any JSON files you do create by hand. You can see from Figure 1 that a JSON file is largely composed of attribute, value pairs, e.g. “firstName”: “John”, can handle nesting “address”: {*the address is a set of attribute:value pairs*}, and arrays, e.g. “phoneNumber”: [ *an array of phone numbers in {“attribute”: “value “}, { }].*

### Asynchronous JavaScript and XML — AJAX

Remember, AJAX (Asynchronous JavaScript and XML) which you touched on in when you created the gallery in Lab 5 and also in Lab 6 and 7, is method for receiving and transferring the data to and from a server in the background without refreshing a web page. As above, check out the Study Direct resources 3D Apps — Self-Learning section for more information.

### JQuery

JQuery is a client side-scripting library based on JavaScript that utilises AJAX methodology. This allows us to use many of JQuery’s built-in functionalities that might take a long time for us to develop. In short JQuery simplifies JavaScript for coders. As above, check out the Canvas 3D Apps — Technology Stack section for more information.

## Pre-requirements for Lab 8

This Part C tutorial requires a very basic knowledge of PHP programming and the MVC design pattern along with JSON. If you have completed the MVC Part A and B of the Lab 7 tutorial then go ahead and complete this Lab 8 Part C, but if you haven’t finished Part A and B, then we strongly recommend you complete them before you start this Part C tutorial. Part C will be using the source code we developed in Part A and B.

# MVC Framework with JSON, AJAX, JQuery and PHP and SQLite

Recall from Part A and B tutorials, where we created our MVC framework, let’s continue using the same MVC framework to progress further in terms of JSON and AJAX operations. But, before you start this tutorial, make a copy (backup) of your previous work so that you can use your last framework for another project you might to do in future. That is, create a Lab 8 part\_c folder with a copy of your Lab 7 part\_b work in it.

During this Lab Part C tutorial we will be developing three examples:

1. Accessing third party JSON based Web Services using JQuery, AJAX and JavaScript
2. Develop our own JSON based web service and parse it through JQuery and AJAX
3. Data retrieval by selection from SQLite using AJAX, JQuery and PHP

We will finish off with some suggested exercises to increase your skills.

Follow the step-by-step guide to continue building your MVC framework.

## Example 1 — Accessing JSON based Web Services using JQuery, AJAX and JavaScript

These days, many web sites provide various functionalities through web services for other users to use in their own web sites. It is good practice to access these web services to enhance capabilities of our own sites. The goals of this exercise are to understand:

* How to get data from third party web services.
* How JQuery works.
* Reinforce your current practical knowledge of how AJAX works from Lab 6.

Let’s start by retrieving a JSON packet via AJAX and JQuery from a **Flickr web service**, parsing it through JavaScript and publishing it using HTML. Note, in these examples, we are not concentrating on presenting the HTML with particularly nice CSS, that’s down to you.

1. Modify your controller.php file inside your controller folder and add one new method named ‘apiGetFlickrService’ as shown in Figure 2:

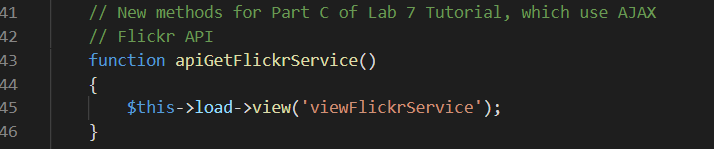


Figure 2: Adding a apiGetFlickrService controller function

* + Code explanation
* We have added another controller method named ‘apGetFlickrService’ to call only a view function named ‘viewFlickrService’.
* Note here that we are not accessing our Model class, instead we are going to access a third-party data using their API. That is, as you will see from Figure 4 we are bypassing the Model class. Is this a good idea? Does it conform to MVC design patterns?

1. Now you need to create 3 files: one named ‘viewFlickrService.php’ inside your view folder, and another flickrService.js for your js folder, see Figure 3. The 3rd file is, of course the CSS, which I am sure you can create yourself, or download from the **Live Feedback Site**. The operation of this code is shown in the sequence diagram in Figure 4
   * We can see that the HTML invokes the JavaScript loadImages() method (or function), via the onclick() function.
   * Further, we can also see from the sequence diagram that the Model class is bypassed completely. Effectively, the controller loads the view and the view makes a direct AJAX request to the Flickr web service, which returns a response direct to the view. So, is this strictly an MVC design pattern? Maybe, maybe not, but you could have a go, as an exercise, at converting the code to a more recognizable MVC design pattern.
     + We leave this as an exercise for you to polish the code
   * But, basically, you would need to use the jQuery .getJSON() method to call the Controller class method apiGetFlickrService() as before, but then as shown previously, invoke a PHP method in the Model class, e.g. $data = $this->model->apiGetFlickrService(); where you would need to create the apiGetFlickrService method in the Model class.
   * This method in the Model class would simply use the Flickr API call:  
       
     http://api.flickr.com/services/feeds/photos\_public.gne?jsoncallback=?   
       
     to return the image array in JSON format. You would decode the returned data using the PHP json\_decode() function and PHP get\_file\_contents() function, and assign the result to a variable, e.g. $data. You can then pick off the data items you want, implying you need to know what the Flickr web service is returning (study the Flickr API) and assign this to an array variable, which you can return to the controller as you did in previous examples. Or, you could use the console.log() method to inspect the returned data first, then develop an appropriate handler.
     + An important part of this process is to rough out your code to see what is returned from the web service because this allows you to inspect the returned JSON object using the console.log() function, you can then build the handler and set out your code MVC style.
   * You can then convert the image array returned to the controller to a JSON object in the Controller class method, and load this into the view by echo’ing it out after loading the viewFlickrService.php view.

 (a) viewFlickrService.php

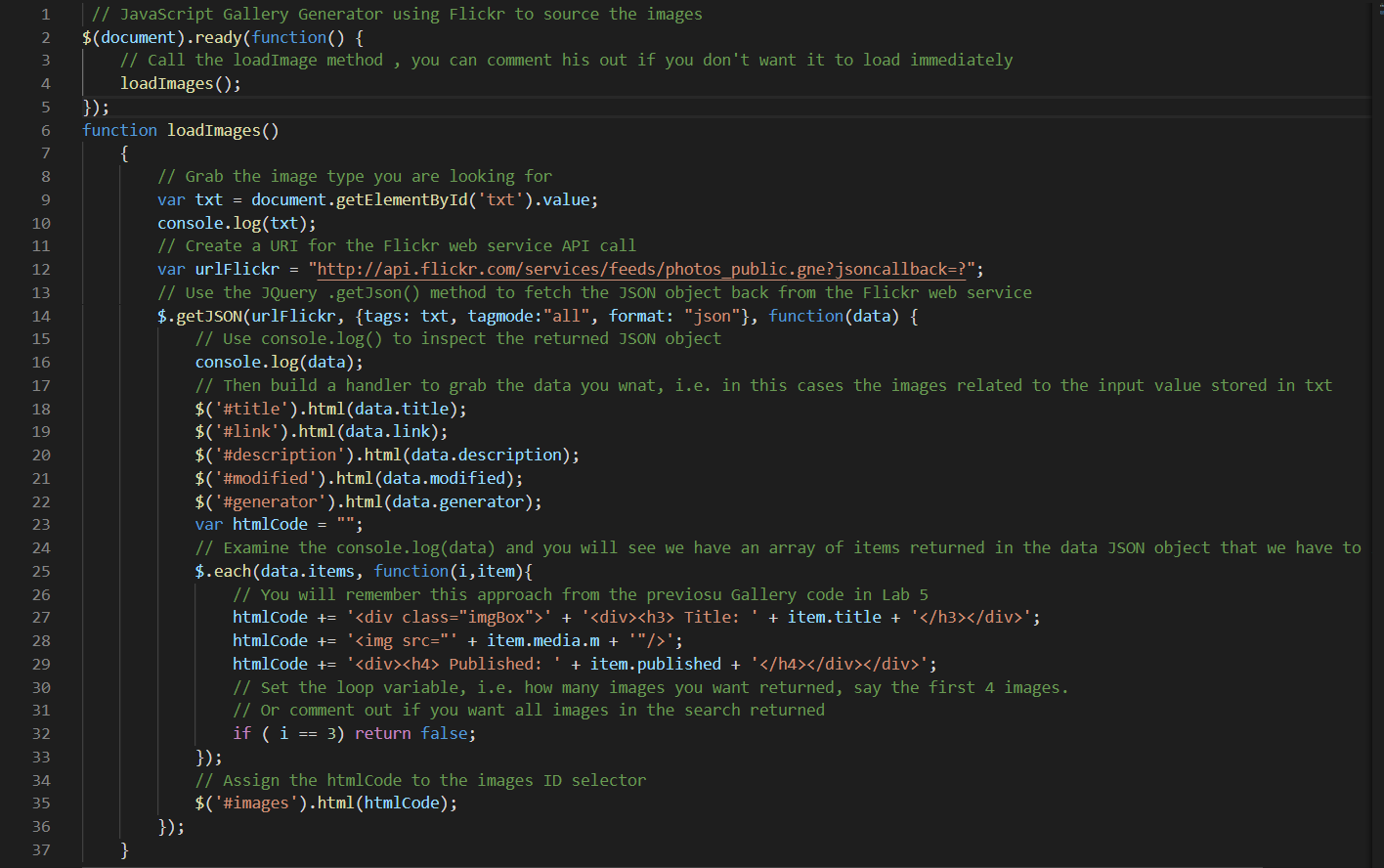
(b) flickrService.js

Figure 3: (a) viewFlickrService.php and flickrService.js codes to implement the Flickr web service

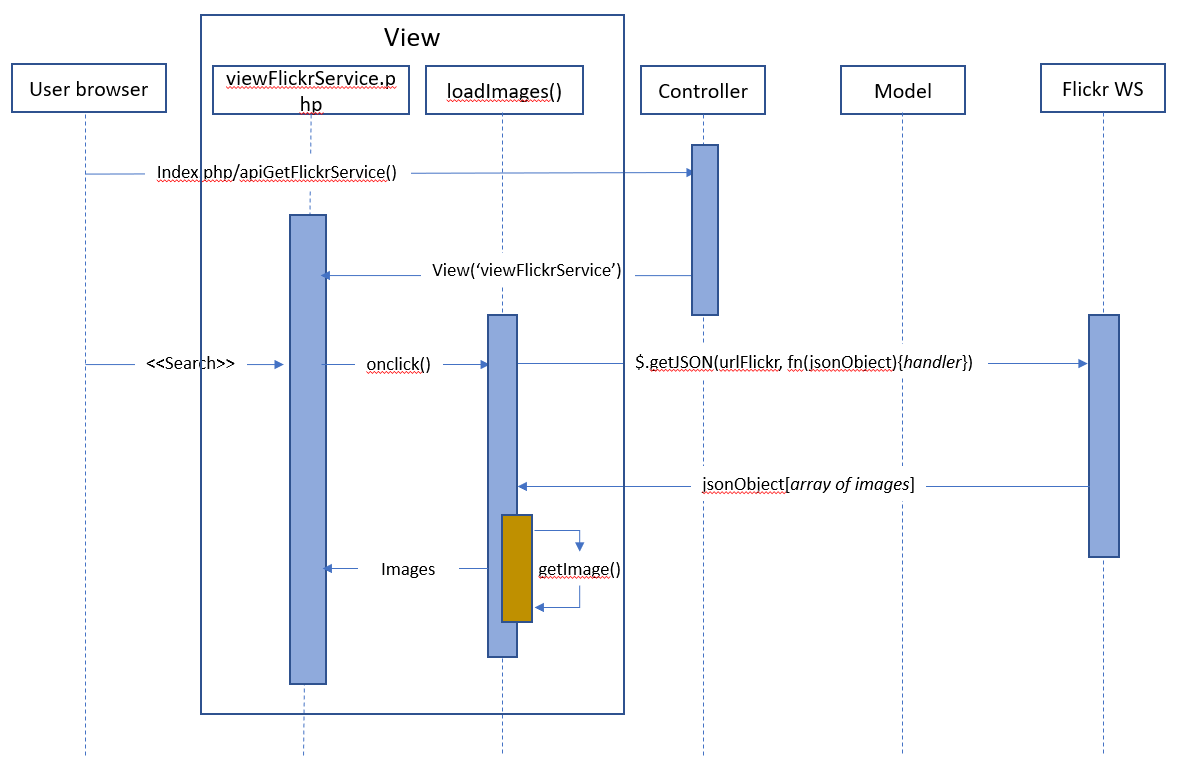


Figure 4: Sequence diagram showing the operation of the getFlickrService process file

* + Consider the sequence diagram in Figure 4 a bit a little more carefully. We could build a set of Model classes that invoke different third-party APIs, e.g. a Model class for the Flickr APIs, a Model class for the Victoria and Albert Museum API, a Model class for your SQLIte database.
  + Similarly, In the same way we could have many different models, that had the same data, e.g. your coke, sprite and pepper data stored in an SQLite database, or a set of JSON files, or in a Firebase cloud, etc. But, your Controller class methods would be the same, as far as a front-end view was concerned. In your controller class you would simply declare your different model classes and write a controller API class with lots of methods to access the desired models. That is, your Controller class methods (i.e. your own API) would abstract out the different models (SQLite, JSON, Firebase).
  + **This is a great opportunity to get into the deeper understanding marks. I will leave you to consider this further, as an exercise, particularly for when you consider how you are going to organize the code based as an MVC design pattern for your final assignment.**
  + Code explanation (Figure 3) (a)
* Line 9: we define some CSS rules to style the image returned from the Flickr Web Service, you could use the CSS developed in previous labs, of course, e.g. img-thumbnail from Bootstrap 4! Or, download this file from the **Live Feedback Site.**
* Line 33: declares the jQuery library, we need this to access the $.getJSON() function. Remember, we used this in Lab 6.
* Line 34: declares your JavaScript code that defines the loadImages function that uses the Flickr web service.
* Line 16 to 18: we have created an input field with id txt and a default value for the search text.
* Line 26 to 28: we have created a button with title as ‘Search’, we have also added an event ‘onClick’ which invokes a function called loadImages().
  + Code explanation (Figure 3) (b)
* Line 2 to 5: Normally, you would be developing all sorts of JS code here for your application including calling any functions you have developed, e.g. the loadImages() function In this particular case, if you call the load images function it triggers the Flickr service endpoint with random images, so we need a way to define an initial call. For example, set an initial search term in the HTML input tag for your Gallery theme; I set ‘Sprite’ line 17 in the viewFlickrService.php file. You may develop more sophisticated solution. Note JS functions have to be declared before they are called in JS, hence they are left outside the $(document).ready(function() method
* Line 6 to 37: is JavaScript code containing a loadImages() function
* Line 9: a variable named ‘txt’ has been declared and assigned value from the text field we created in our html code — notice we are using the document.getElementById function to get the value of the text input in the text box. We could have used a jQuery ID selector. That is, these are the same functionally:

        var txt = document.getElementById('txt').value;

        var txt = $(‘#txt’).val();

* Line 14: The $.getJSON() function is a member of jQuery library, it uses AJAX to retrieve the Flickr image feed with parameters: tags, tagmode, and format
* Here we are passing a value typed in the input box to search the Flickr feed
* Flickr returns a JSON packet, since we defined format = json in our code
* Line 14: the function receives the resulting JSON object in data object.
* Line 16: Now, this is important, we use the console.log() function to examine the JSON object returned to see if there is any other data we might like to use. Look at the Object I the Chrome Inspect Tool.
* Line 18 to 22: in this case we have seen that the parameters in the JSON object might be useful in relation to the image array and each image returned. Actually, they are not that interesting, but it serves to illustrate.
* Line 25: we loop through all elements in data object
* Line 26 to 29: We build the HTML code as we did before in Lab 5 to include the media (images) received from the Flickr web service, a title and publications date.
* Line 32: we restrict the number of images to 4 (i.e. 0 to 3 elements from array) only — this is just an arbitrary number.

You should explore the Flickr API feed in more detail: <http://www.flickr.com/services/feeds/docs/photos_public>

1. Execute your newly created program to see the output
   * Type any keyword such as: ‘cold coke’ click search, see Figure 5.
   * Note: I have used a localhost in the URL, your URL could be different depending on hosting.

Note: Your Page was never refreshed, and the contents instantly appeared below the text field and the button, this is because we are using jQuery and AJAX features on our webpage above. Further, according to the simple CSS we created, the images are simply floating left, we could create more sophisticated CSS, of course, to better style the returned images. You can easily imagine a couple of improvements:

* Use Bootstrap 4 images components, particularly the img-thumbnail class to create a gallery of thumbnails, then use an open source lightbox, e.g. fancybox to present each thumbnail in a larger format.
* You could create a gallery of your own 3D model images created in 3ds Max, expanding or adapting your old gallery PHP code to create a gallery in your MVC framework. You might replace the XMLHttpRequest with a JQuery AJAX request, etc.

Finally, You can see this is a similar approach to the Image Gallery code you did in Lab 5, but instead of using XMLHttpRequest we have used the JQuery .getJSON method to make an AJAX request to Flicker's web service. Now we have the JSON object back, we can strip out the date we want. You should be able to see that we can adapt this code to build a gallery as well, and we could change the HTML to use Bootstraps cards! Each card to hold a single image and associated data, set up Bootstrap rows and columns to hold the cards, build JS function to do the handler and 'pass by value' to set up row and column parameters

Alternatively, you could set images to Bootstrap thumbnails and build in lightbox functionality (fancybox).

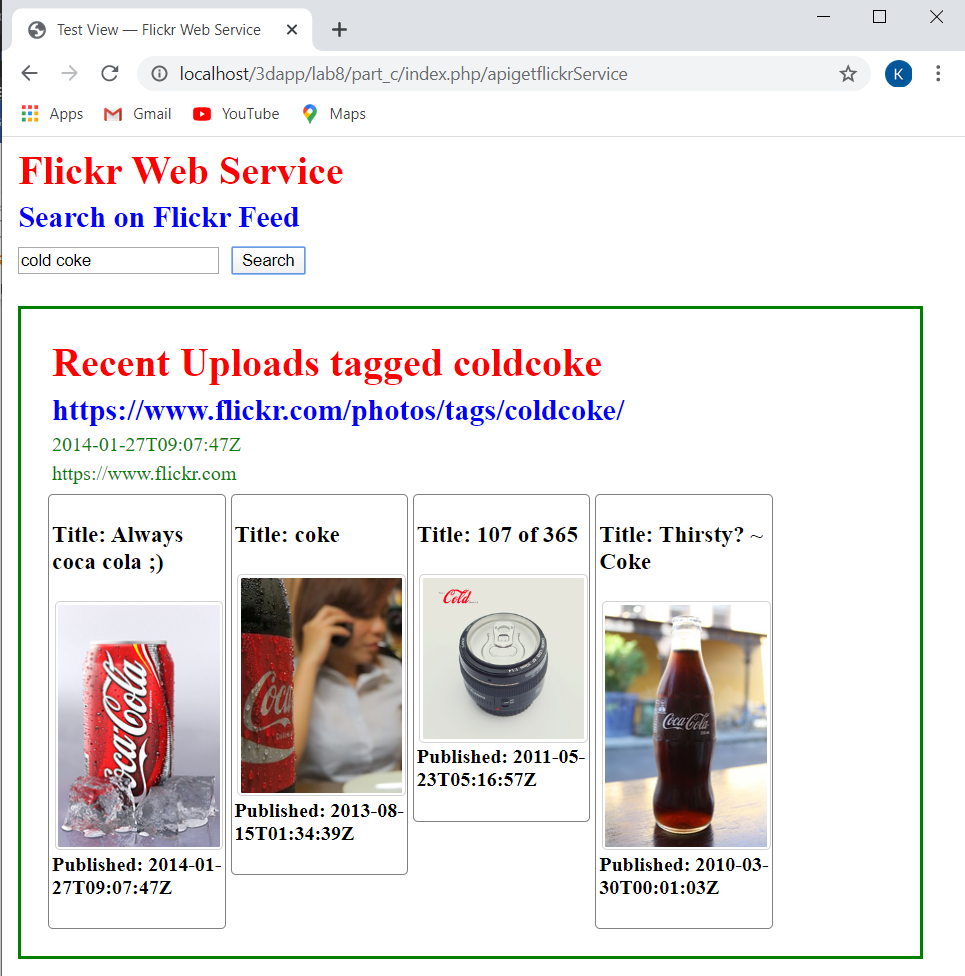


Figure 5: The result from the apiGetFlickrService web service

## Example 2 - Develop your own JSON based service and parse it through JQuery and AJAX

1. First, we need to create a JSON structure (a very simple one for this demo) and parse it through JavaScript. So, create a file named ‘createJson.php’ inside your ‘part\_c/application/model’ folder and simply type the following JSON output shown in Figure 6. You can see that the JSON file is easily readable.



Figure 6: A simple JSON structure echo’d to the browser, this is really acting as a model

* + Code explanation
* Nothing tricky, just one PHP echo statement to display a JSON packet on a browser. In reality, you would read such data from a database and write it to a JSON structure — see Example 3.

1. Now modify controller.php file and add another controller method see Figure 7.

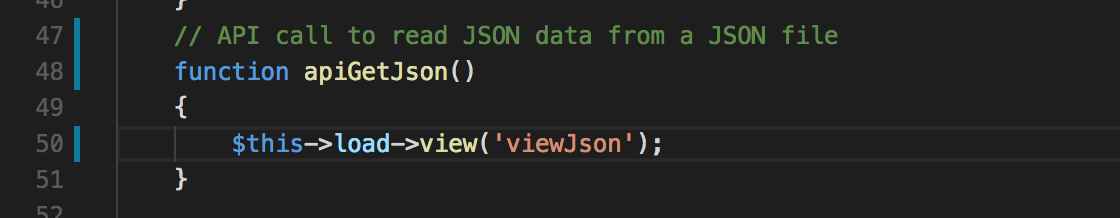


Figure 7: A controller function to call a view that will use PHP to read the JSON

* + Code explanation
* We have added another controller apiGetJson loads a view function named ‘viewJson’.

1. Create a new PHP file named viewJson.php and type the following code in it, see Figure 9, and save it in your view folder.
   * Here, this time we are being particularly deviant, see Figure 8, because we are now bypassing the controller. The view, in effect, via the JavaScript code, is making an AJAX call direct to the createJson.php in the Model. Is this wrong, who’s to say?

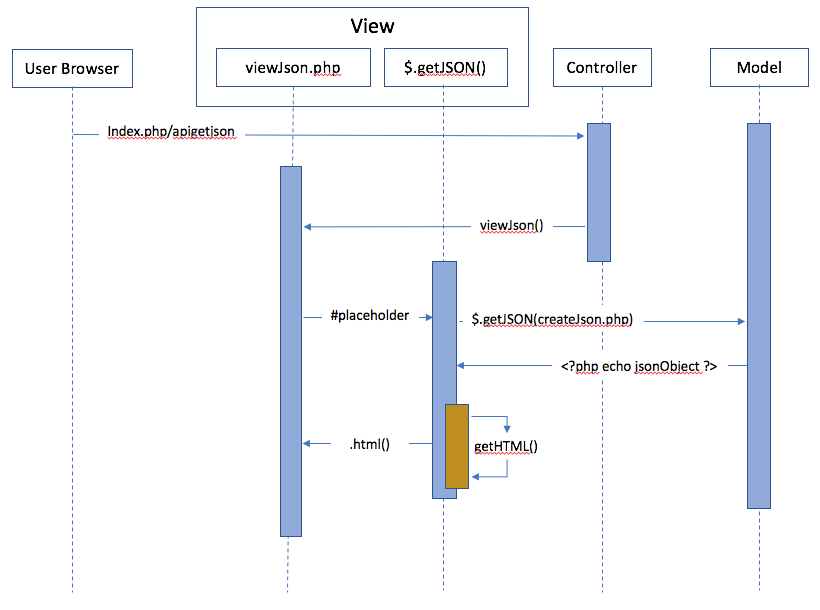


Figure 8: Sequence diagram for the apiGetJson

* + However, as above, and as an exercise, you could reorganize this code to better separate the model from the view via the controller. You should consider organizing your code so that the view invokes a method in the controller, which in turn invokes a method in the Model class (this is the request sequence), and then the model responds by sending the result back to the controller, whereby the Controller class method sends the result (or loads the view with the result) back to the view.
  + This is the way you should do it for your assignment, and if you look at the code for Lab 9 on the **Live Feedback Site**, that is how we do it.

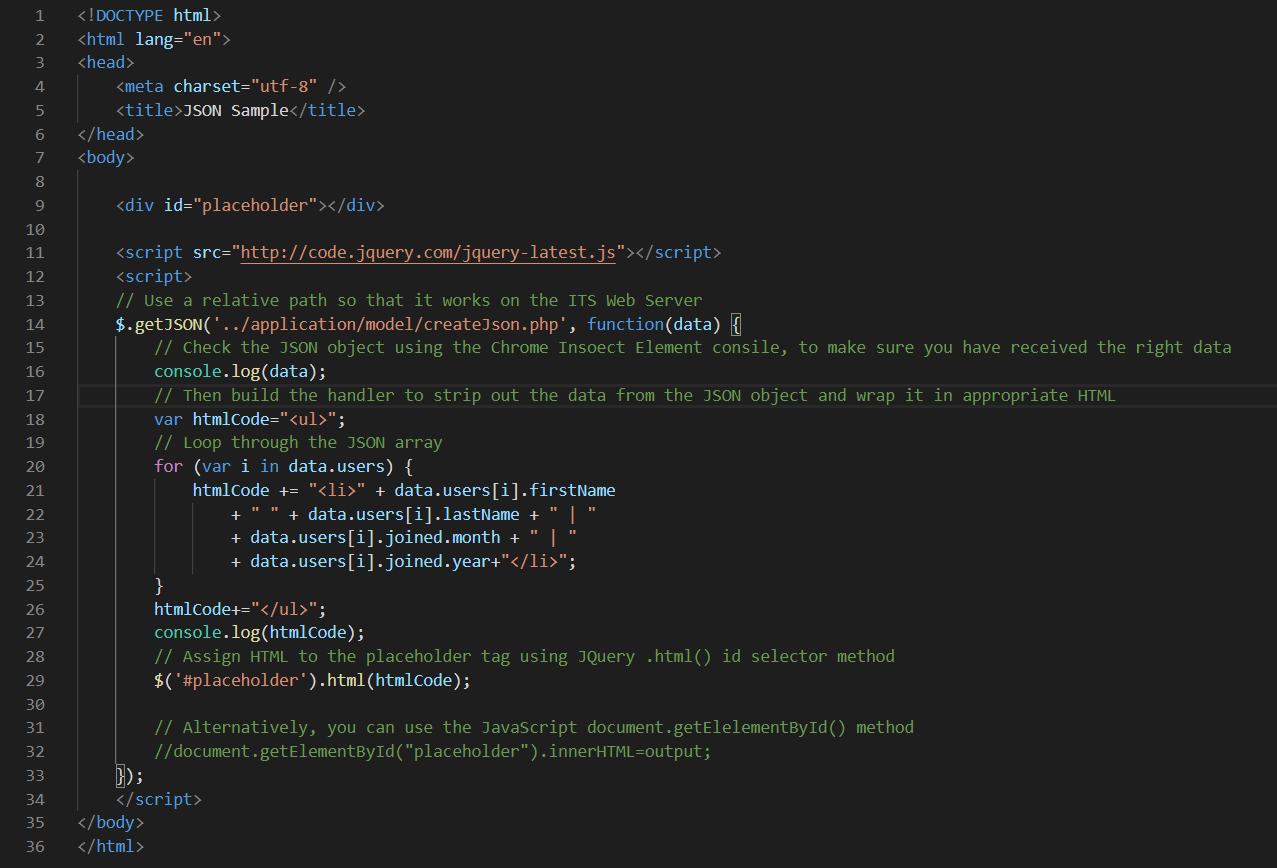


Figure 9: JavaScript code using JQuery .getJSON() to read the createJson.php

* Code explanation
* Line 11: contains reference to the latest JQuery library
* Line 15 to 33: makes an AJAX call with the jQuery $.getJSON function, with an ‘approachable’ path to the createJson.php file as a parameter. This should be a relative path.
* Line 20 to 26: we parse through the JSON packet and store HTML tags in a variable named ‘htmlCode’
* Line 29: we display the value of htmlCode on the ‘placeholder’ div located on line 8.
* Let’s see the output of our code, see Figure 10.

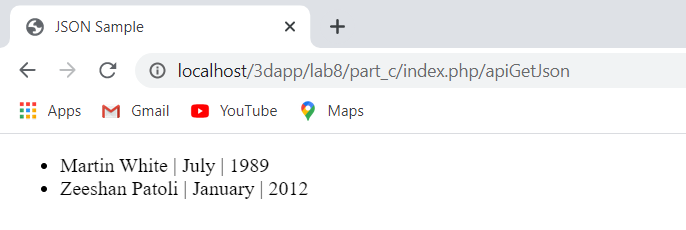


Figure 10: The resultant data displayed on the web page

Note: You can see from the output we have retrieved only the information we need from the JSON object, avoiding any unnecessary data, which we don’t need.

At this point it is worth reflecting a little on how to invoke calls to the Data Abstraction Layer, e.g. the PDO Objects, such as dbCreateTable, dbInsertData, dbGetData, and so on. In reality we are likely to use something like a JavaScript onclick=”myFunction()” (or JQuery .click()), where myFunction may exploit the JQuery .getJSON AJAX method with a URL to one of the Controller class methods, which then invokes the PDO method in the Model class. This ‘example 2’ effectively bypasses this approach and uses the JQuery .getJSON() method to get a JSON packet direct from the model.

## Example 3 – Data retrieval by selection from SQLite using AJAX, JQuery and PHP

During this example, we will:

* Develop a new controller function called ‘apiLoadImage’ in the controller class,
* Develop a new function dbGetBrand() in the Model class to retrieve brand names of drinks and display them in a drop-down list for selection by the user.
* Although we will also add brand names to the database table, this dbGetBrand() function will initially access a temporary array of brand names.
* However, you should consider adapting the existing dbGetData() function to get only the brand names from the table. At the end of this tutorial I set some exercises to get data from the database table in different ways.
* You will also need to update your dbCreateTable and dbInsertdata methods in your Model Class.
* Develop a new Model ‘modelDrinkDetails.php’ to retrieve data from database based on the parameter, drinks brand name.
* In this case, rather that create a method in the existing Model class, we are going to develop a separate model, simply to illustrate the notion of having many possible different Models. However, we won’t configure this Model as a method in a class. This Model will open and close a connection to the existing drinks database.
* Again, I will leave it to you as an exercise to create a method dbDrinkDetails() in the existing Model class.
* Develop a new view named ‘viewDrinks.php’
* We will create all the HTML here and inject it into the view. First, we will allow the user to select the drink brand name from a drop-down list. The we will retrieve the data from a selected drink from the PHP file using jQuery and AJAX

Further, in case you get a bug in your PHP code, you will need to do some research on how to debug PHP code, which can be tricky. Use of echo statements at various points in your code flow may well do the trick, I have done this when developing these Lab 7 and 8 tutorials in the past. Interestingly, this time with the 2020 version, I completely forgot to update the dbCreateTable and dbInsertdata methods in the Model Class, which also requires you to create the table again and insert data. As a result, I spent a couple of hours scratching my head and then decide to install a useful [Chrome Logger](https://craig.is/writing/chrome-logger) debug extension for Google Chrome, which is really cool! I will briefly describe it below and how I used it to track down my PHP errors.

1. Let’s start by adding a controller function in controller.php file, see Figure 11.



…

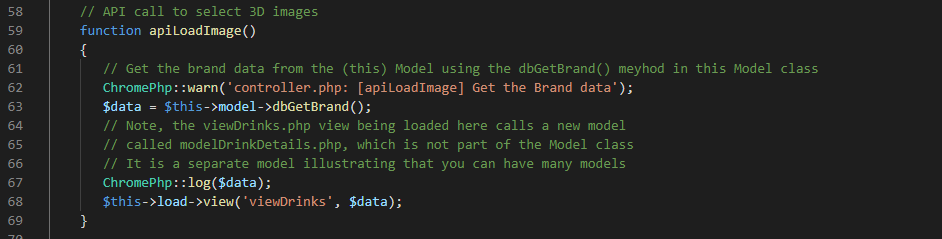


Figure 11: New controller function added to the controller.php

* Code explanation
* Line 2: You can see that I have included the ChromePhp.php file downloaded from the Chrome Logger site indicated above, don’t forget to student this tool. Also, you can see that I have installed the ChromePhp.php file in a debug folder at the same level as your application folder.
* Line 2 and 3 act in a similar to the JavaScript console.log() function allowing you to print out messages and data in the Chrome console, provided you have installed and switched on the extension, see [Chrome Logger](https://craig.is/writing/chrome-logger) installation details. In this case we can see that we are simply outputting a message 'controller.php: Hello World' and the server details
* Line 62 and 67: You can see that we also print out another message and also the $data values returned from the Model method dbGetBrand(). We can see in Figure 12 that the Chrome Logger console debugger has output the initial message, only really added to show it was working, and the server information (unique ID: …), not that interesting, another debug message from apiLoadImage method in the controller and the actual data retrieved for the brands from the model. So, everything looks good so far.

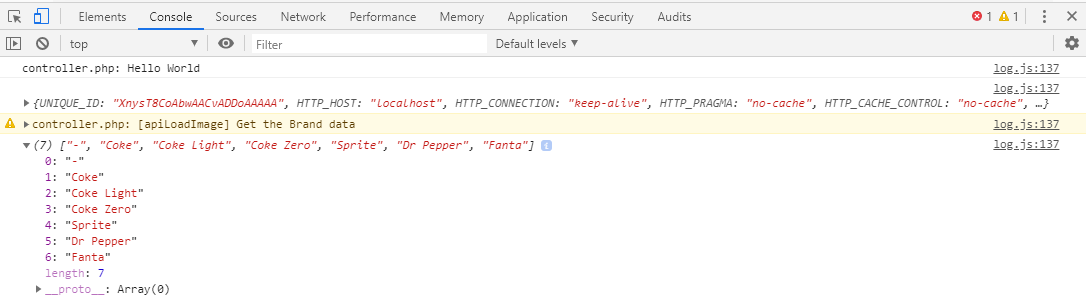


Figure 12: Chrome console showing the Chrome Logger debug messages and data..

* Line 63 we call the function dbGetBrand. We can see this function in Figure 13, put this in your Model class in the model.php file. We could have simply utilized the dbGetData function instead, but if you have more than one brand name the drop-down menu would then contain duplicated brand names, unless you process the returned data to dump duplications. One way to handle duplications is to design the data base schema more efficiently in the first place. That is, create a table in the database of brand names and link this via a foreign key in the Model\_3D table. However, as mentioned, for now the function dbGetBrand() is simulating the brandName table, whose brand names are returned as a data array.
* Another interesting thing to raise here is that the point of this Example 3 is to illustrate the use of AJAX with the jQuery .getJSON() function. One could argue that your 3D App might already know what brand names are available and thus not need access to a dbGetBrand() function to populate the menu. So, another way would be to just fire off an AJAX service request for the brand name required.
* Line 68 (Figure 11): we call a view ‘viewDrinks’, which we will create in this section that consumes the brand data stared in $data.

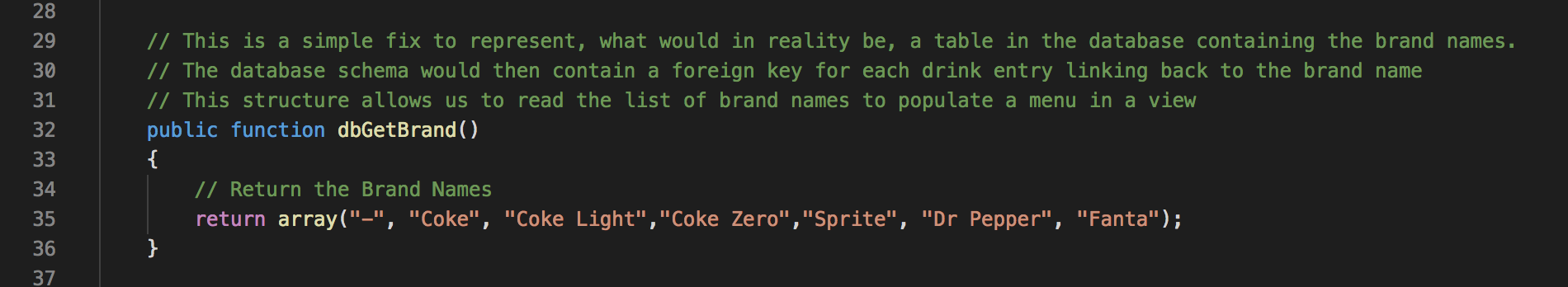


Figure 13: The new function dbGetBrand().

1. Now create the new view file (that is being called from the Controller class apiLoadImage() method (or function) in Figure 11 above named viewDrinks.php inside your view folder and type the following code in it, see Figure 14. For convenience, I have heavily commented the code. You can ignore the debug statements if you wish. I have had to chop up the code screen shots into three parts to get them into this document, see Figure 14, 15 and 16 Also, you can download the code from GitHub to save some typing.

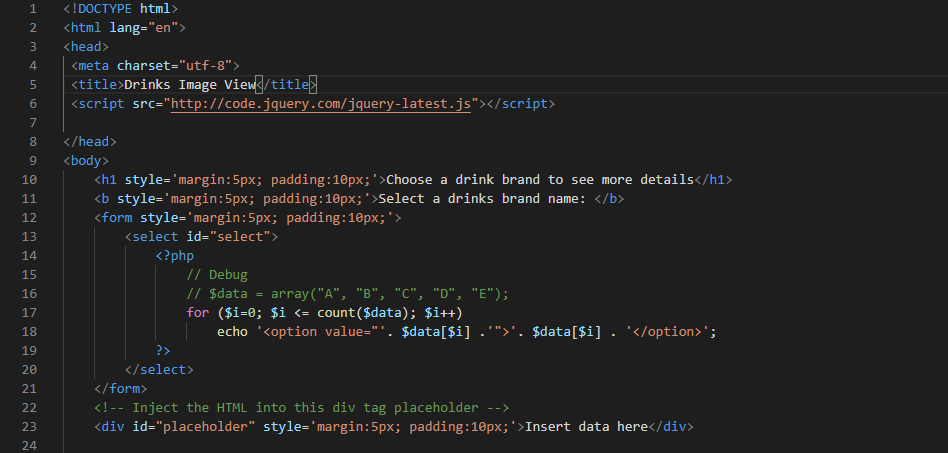


Figure 14: The new view viewDrinks.php

* Code explanation, also see the comments in the code
* Line 6: We have added a jQuery library to use jQuery library functions in our 3D App page.
* Line 12 to 21: We define a drop-down list containing names of drinks that we received from the dbGetBrand() function from the Model class through the controller. As a design choice, if we think that as part of a 3D App we would already know this information, we could have hard coded it here:
* $data = array("A "B", "C", "D", "E");
* Line 24: we set a place holder with id ‘placeholder’, we will use this to inject our returned data and HTML through AJAX

Now let’s consider the JavaScript needed to get the data, see

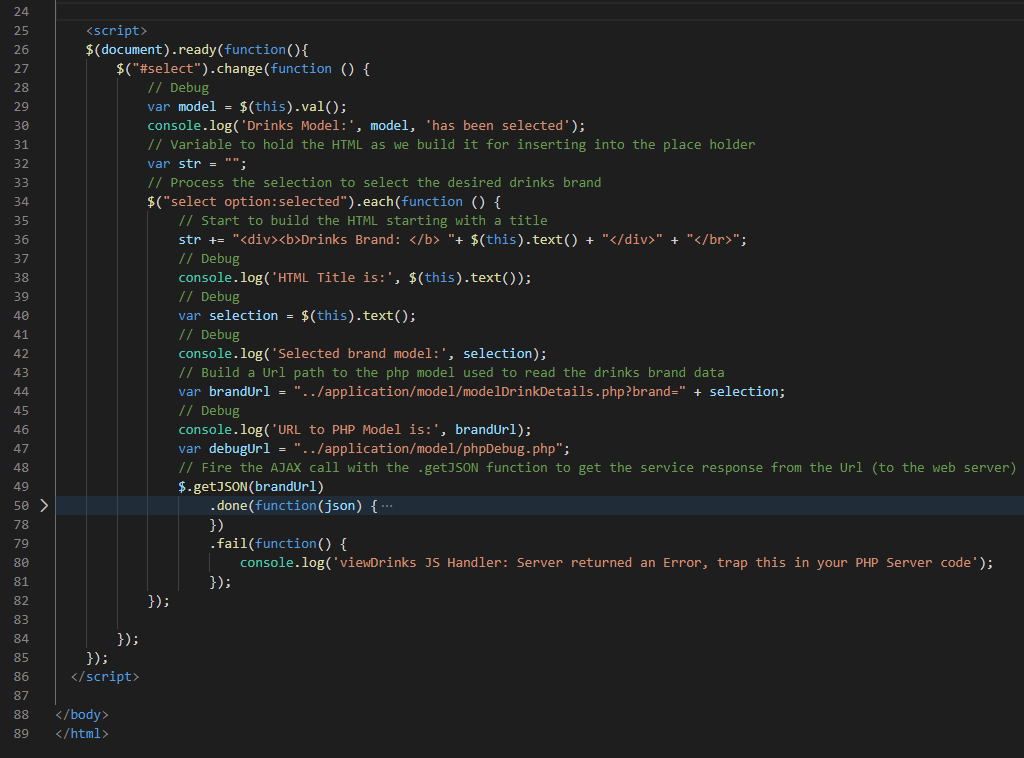


Figure 15: The JavaScript code needed to make an AJAX call to the database via the model

* Line 26 to 81: contains a jQuery $(document).ready(function() that contains a handler for an AJAX function that will be implicitly called every time we select different drinks items from the dropdown list in .
* In this function, we call a new PHP model, modelDrinkDetails.php using the jQuery .getJSON() function to get more details about the selected drink brand and publish the results in the ‘placeholder’.
* Line 28-30: we get a selected value from the drop-down list and store in the select variable called ‘model’ (line 29) so that we can use the use console.log() to test and debug, which in this case is just checking which drink brand has been selected. You can eliminate this code, leave it in, or comment it out later. There is some basic debug/test code threaded throughout this viewDrinks.php file, which you can eliminate later. It is a good habit to insert some debug code as you go along.
* Line 34: with the selected option, we start to process the selected brand name by starting by initially building the HTML contents, which is stored in a string variable ‘str’ on line 36, followed by some debugging code.
* Line 44: we set up the URL to the new model: modelDrinkDetails.php, for the AJAX request.
* Line 47: You can ignore, I set this up to a test phpDebug.php file in the model folder, because I was getting an internal server error 500 and I was playing around to make sure the internal server error 500 was being generated by the modelDrinkDetails.php file on the server side. It was, and was likely to be a coding error.
* Line 49: We use the .getJSON function with the URL to a new model: modelDrinkDetails.php with a GET parameter set to as the selected drink brand name.
* Make sure your URL path is relative
* Line 50: we have uses jQuery differed .done() method to wrap the handler function(JSON), which is currently hidden in Figure 15 for ease of reading. Figure 16 details the handler code. Make sure you line up the code snippets using the line numbers so that you get all the brackets correct. If you get stuck, download form GitHub for example working code. If you get an error coming back from the server, e.g. the JSON object does not contain the data, or an internal server error 500 of some sort, the jQuery .fail() method will catch it. You will see I have placed a console.log() error message her, but I am only surmising what the error might be. It is up to you to trap it in the server code (modelDrinkDetails.php). As I alluded to above, I actually did have an error because I forgot to update the dbCreateTable and dbInsertData methods in the model modelDrinkDetails.php; more on that later.

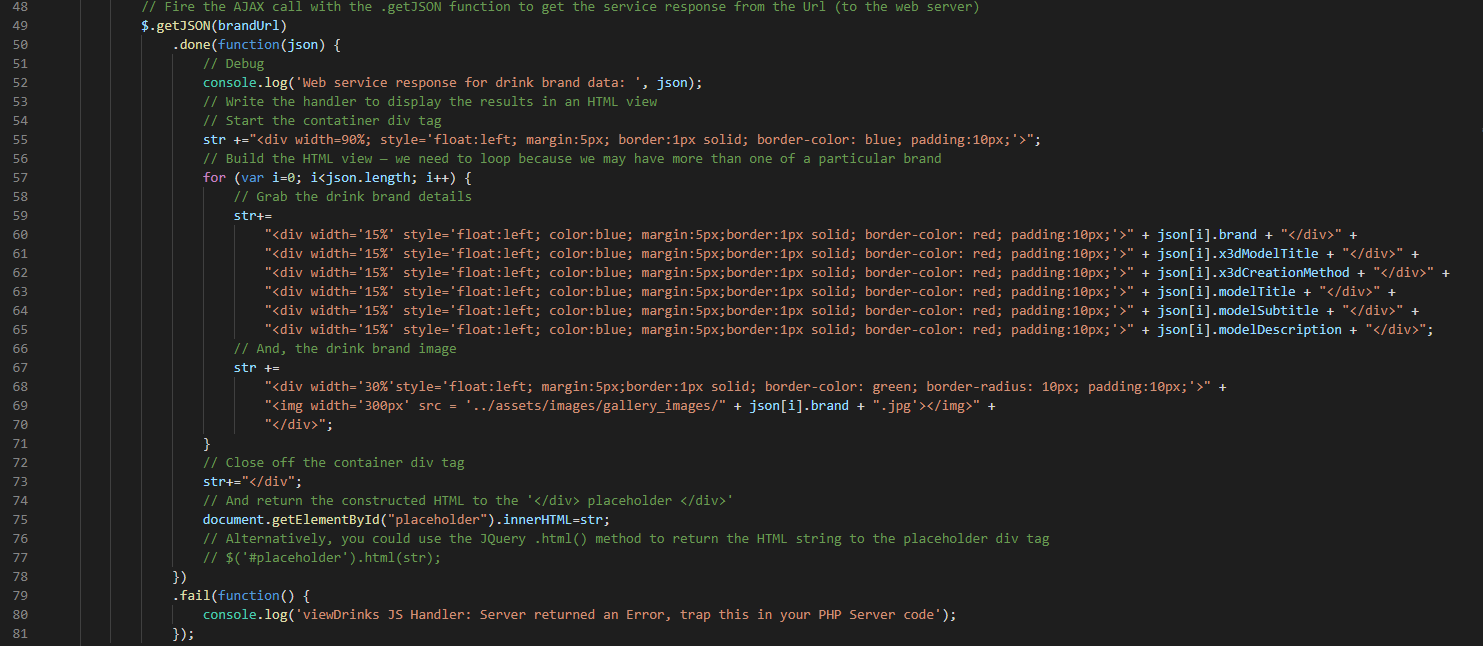
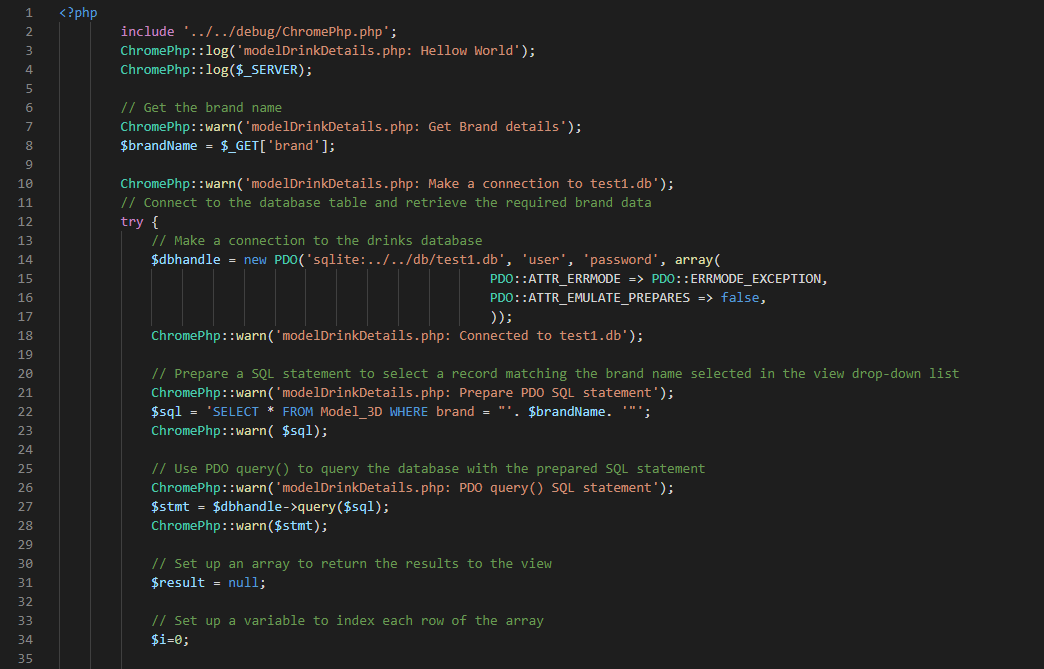


Figure 16: The JavaScript handler for parsing the data returned form the serve as a JSON object.

* Line 52 is debugging again, this is important so that you can see the returned JSON object in the Chrome inspect tool. If, for example, all your debugging code above is showing everything is ok, but your JSON object is not returned properly you may see this here, e.g. wrong data. If so, It will, most likely, mean you have a bug in your new model, which we will write next.
* Line 55 to 78: We define some handler code to deal with the JSON ‘data’ received from your new model PHP file (modelDrinkDetails.php ) that we will create in the next step. Note, we keep appending the HTML we are building the string variable — it might have been a better idea to name this variable more clearly, e.g. strHTML, but we didn’t. Note, we are using a for loop, because we need to iterate around the JSON object array, and there might be more than one entry for the drinks brand name selected.
* Line 75: we use the JavaScript document.getElementById() function to pass the HTML back to the placeholder div tag. Alternatively, line 77, we could use the jQuery .html() method to return the HTML code with database results to the place. You can see I have commented this method out, but if you uncomment it, you will see it overwrites the document.getElementById() method, unless you comment that out too. Think about this, we used the jQuery .HTML() method quite extensively in Lab 6, check out the getJsonData.js file.
  + You will also see here, that we have been a bit lazy, we should have put all this JavaScript code inside a separate .js file and referenced it, etc. Make sure you do this for your final assignment, otherwise you are likely to lose marks. Also, as this was really just a test to illustrate the MVC framework and getting data from a backend database via the model, we have been a bit lazy building the HTML too. It is quite awkward, not very flexible to build the complete HTML as a string, we would probably only pass minimal data through to the view so that we could better manage the CSS across media screen breaks.

1. Now we need to define a new model requested in above view file. Let’s create a file inside your model folder and name it as ‘modelDrinkDetails.php’ and write the following code in it, see Figure 17. Again, you can ignore the simple PHP debug and comment statements, and we have already discussed how to use the Chrome Logger debug approach above, so we won’t mention that again. However, you can see in Figure 17 that we have extensively used the Chrome Logger extension for debugging.

* If you look closely at this code you will see that we have merely copied the code from the original model.php, and we have been a bit lazy, we haven’t created a class and method. We should really make this a method in the Model class, but I will leave you to do that. However, it does illustrate that in an MVC design pattern we can have many models, many views, etc.



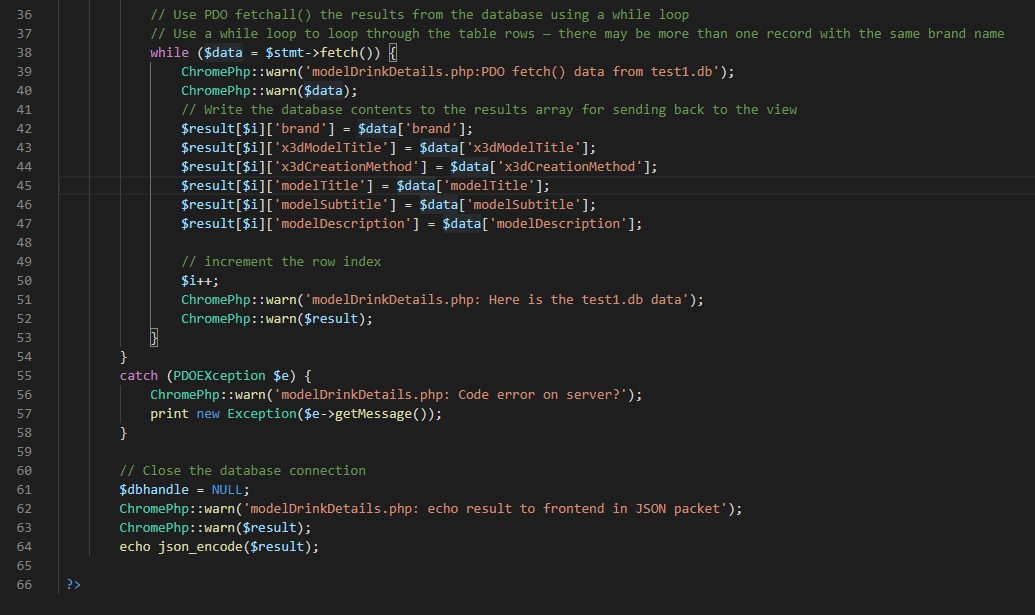


Figure 17: The modelDrinkDetails.php using PDO objects, note the Chrome Logger debugging

* Code explanation
* This modelDrinkDetails.php file retrieves the data from database based on the $brandNamevalue received as a GET parameter to this file. One thing should should be aware of when debugging, if you use simple PHP echo statements, (e.g. echo $brandName;) and then invoke the modelDrinkDetails.php file directly from a URL in a browser, then the echo statements would output to the browser our debug data. If we insert echo statements before the final line, i.e. line 47, our $.getJSON(brandUrl, function(json) {) method in the previous viewDrinks.php file will not receive the intended JSON object, and the view will fail. However, you can see that I now recommend using the Chrome Logger extension instead of simple PHP echo statements
* Line 8: brandName variable is set with the PHP $\_GET() function.
* Line 10 – 18: We create a connection to the database, with some debugging code as already described. This assumes previously that you have created the database and inserted data in the Model class in the file model.php. If you recall in Tutorial 7 Part B we created three basic functions; 1) dbCreateTable, 2) dbInsertData and 3) dbGetData. This exercise 3 is just an example of creating a slightly more sophisticated method for getting data from the database.
* It’s worth mentioning that we could simply use the dbGetData to extract all the data, as it is so small, and then select the particular drinks brand in the JavaScript handler.
* Line 22: Prepares a SQL statement to query the DB, again note debugging.
* Line 27: uses the PDO prepared statement to query the DB, again note debugging.
* Line 31 – 34: sets up some variables.
* Line 38 – 53: Uses the PDO fetch() to fetch the result from the DB, and stores the result in the result array in a while loop, again note debugging.
* Line 55 to 58: Creates a catch block to trap any PDO exception, and as an example discussed earlier, you may recall when I was completing this tutorial myself again this year, I got rather blasé and forgot to update the dbCreateTable and dbInsertData methods in the Model class in the model.php file. The result was that the PDO query of the database was querying the old database tables and not returning the correct results. I had inadequate debugging code and scratch my head for quite a while before inserting the Chrome Logger debugging code as discussed. The result then was that I immediately saw there was no result being returned form the database. That is the code failed at this point:

// Use PDO query() to query the database with the prepared SQL statement

ChromePhp::warn('modelDrinkDetails.php: PDO query() SQL statement');

$stmt = $dbhandle->query($sql);

ChromePhp::warn($stmt);

It was then obvious there was a problem with the database, so check the Model class, and viola, I hadn’t updated said methods (dbCreateTable, dbInsertData) to match the new modelDrinkDetails PDO query.

* Line 61: Closes the database connection.
* Line 64: Encodes the database results into JSON format and echo’s the JSON packet out to the waiting $.getJSON(brandUrl, function(json) {) method in the previous viewDrinks.php file, and some debugging code

1. Let’s See the output of Example 3, type the URL in a browser, see Figure 18. Take a note of the Chrome Dev Tools Inspect window where you can see the debug console.log() and Chrome Logging data. I will leave you to check the Chrome console and verify all the debug statements, etc. But, we can see that we have a correct result, i.e. the Fanta image (in lieu of a 3D model) and associated data form the SQLite database.

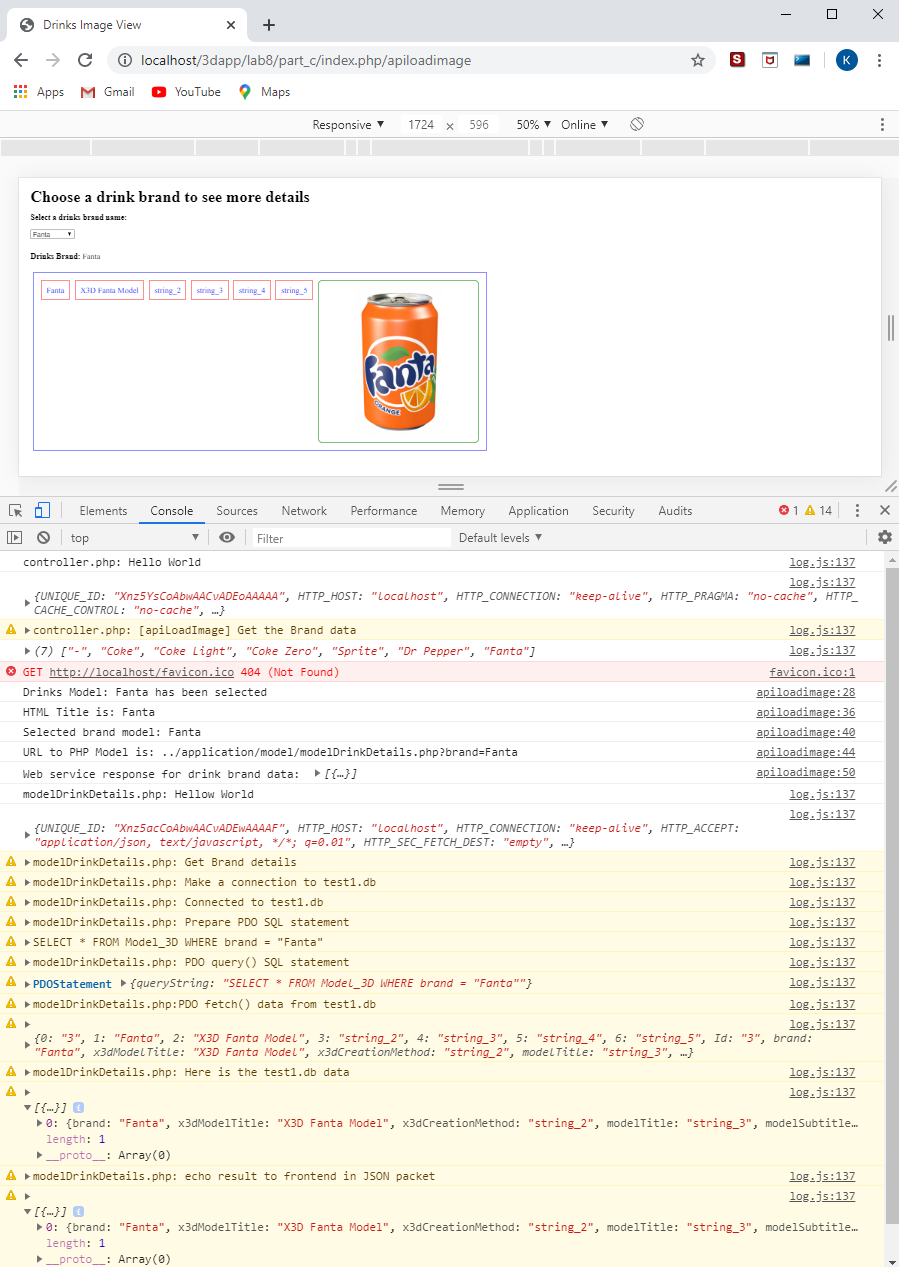


Figure 18: Result of retrieving data using the apiLoadImage() controller method.

**Note: I have used Localhost in URL, your URL could be different depending on your hosting.**

1. Select a different drinks brand from the drop-down list to see details of different drink brand, see Figure 19.

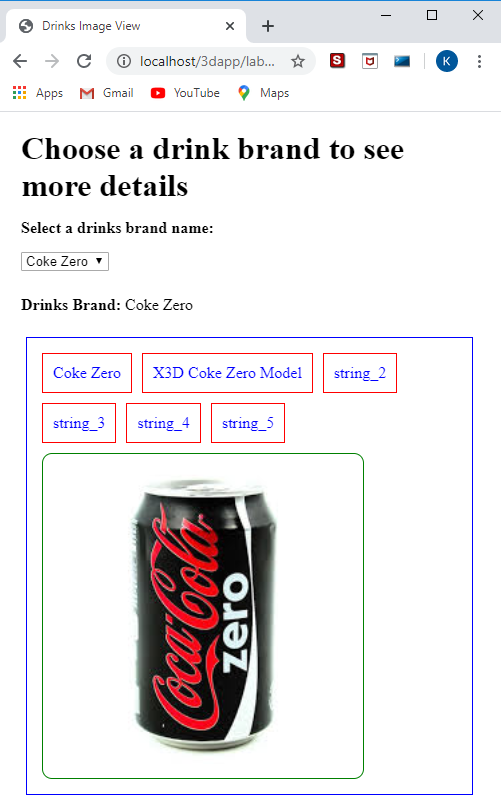


Figure 19: Result of retrieving data using the apiLoadImage() controller method.

Note: I have used Localhost in URL, your URL could be different depending on your hosting. If you are developing on a PC or Mac using a Localhost, when you port your code to your public\_html/ ITS Web Space you will need to make your URL path in any .getJSON() function either relative or static w.r.t. to your host.

Some things to note with this example 3:

* As before, I have been a bit deviant with the flow of control between the model, view and controller. You can trace through the example code and create a sequence diagram to show what is happening. Based on this, you can then modify the code to operate such that the controller always handles requests and responses. In this case, you would first pass your result array back to the controller method, and then in that method echo out the result using the PHP json\_encode() function.
* Further, I have not inserted loads of data in the database table, simply reverting to dummy data. Suffice to say your version, for your assignment will be populated with good data.
* You should note, as can be seen from Figure 18 and 19 that I have ‘borrowed’ a set of test images from the Internet. I have stored these in my assets folder, from which you can see the path in the associated code above.
* You could, just as easily, extend the JavaScript handler to access your 3D models and render these along with the other back end data.

## Some Additional Exercises

1. Given these three examples, you should be able to build a good backend using SQLite that will have a Data Abstraction Layer (DAL) composed of a set of appropriate PHP methods in a Model class that can execute common CRUD (Create, Read, Update and Delete) functions on the SQLite database.
2. You should also be able to develop a set of PHP controller methods (I labeled mine apiThisAndthat() to distinguish between the PHP Model class methods dbThisAndThat() in the DAL. We discussed this in the first example.
3. You should now be able to take your Lab 6 results and reconfigure this to exploit your MVC framework (Lab 7 and 8). When you have done this, and it is working ok, you have will have a solid foundation, i.e. MVC framework, to develop your assignment 3D App.
4. Finally, you should, in particular go back over your Lab 6, lab 7 and Lab 8 code and spend some time refining it. For example:
   * Go back to example 3 and build the brand names into the SQLite database schema. You have already added brand names, but there are no duplicates. You could quite easily have more than one ‘coke’ brand with different model data, image data and other data. So a search on the database to select the coke brand and using the brand name as defined in this simple table would end up populating the drop-down list multiple times. So, you will have to either:

* Dump duplicate brand names in the returned brand names in either the Model, perhaps with some fancy SQL statement, or write some PHP code to remove duplicates before inserting returned values into the results array.
* Or, in the controller you can process the returned array to delete duplicates before loading into the view via an ‘echo json\_encode($data)’, where data now contains the unique brand name list
* Or, you could just write some JavaScript in the JavaScript handler to ignore duplicated brand names.
* Alternatively, create a table of brand names and link these via a foreign key (FK) to the brand data table, etc. You will need to take account of the FK in the SQL statements if you want to also return the brand name along with the drink brand data.
  + The gallery code may or may not be useful, if you do use it, how can you improve it?
  + I mentioned, in lab 6, that you have instanced 3 time the X3D renderer. For your assignment this will be four times and depending on how complicated your 3D model (and animation) is this will eat up browser resource and may malfunction, but you may get away with it. You should consider using the X3D switch statement so that you only instance X3D once, but switch in different scenes (models), which implies a set up that may use the same cameras, lights, etc.
  + Consider carefully how to organize your SQLite database, you may want to spend some time designing the database schema to give you more optimal results, but you can keep it simple.
  + Make sure that you organize your MVC design pattern carefully so that the controller manages requests from the view and responses from the model, etc.
  + Think carefully about how you name your classes, this is obvious Controller and Model, but also how you name the associated methods (or functions), e.g. I put ‘api’ in front of each of the methods in my Controller class to imply this layer is a bit like an API layer that is abstracting out the Model class from the view. Similarly, I put ‘db’ in front of each of the Model methods, I could have used ‘dal’ to indicate this is a data abstraction layer, abstracting out the database. Further, then give the actual methods a meaningful name, and take the opportunity to change the name if after further development the method is really doing something slightly different. For example, in example 3 above, I created the apiLoadImages, well yes, it is loading the images, but it is also loading the associated data from the database. Perhaps, I should change this method name to something like apiLoadImageData? The point is to make your code readable! This also means adding comments.

# Appendix

## Appendix — Review results from Lab 7

You may recall from Lab 7 by the end of Part B you will have created a simple MVC framework upon which you can expand to include more functionality. New functionality will largely be added by created more PHP based API methods in the Controller class and their associated PHP database methods in the Model class.

Figure 20 illustrates your Lab 7 Part B result if you simply invoke the index.php page, here the default ‘home’ controller method loads the view3DAppTest\_2.php view.

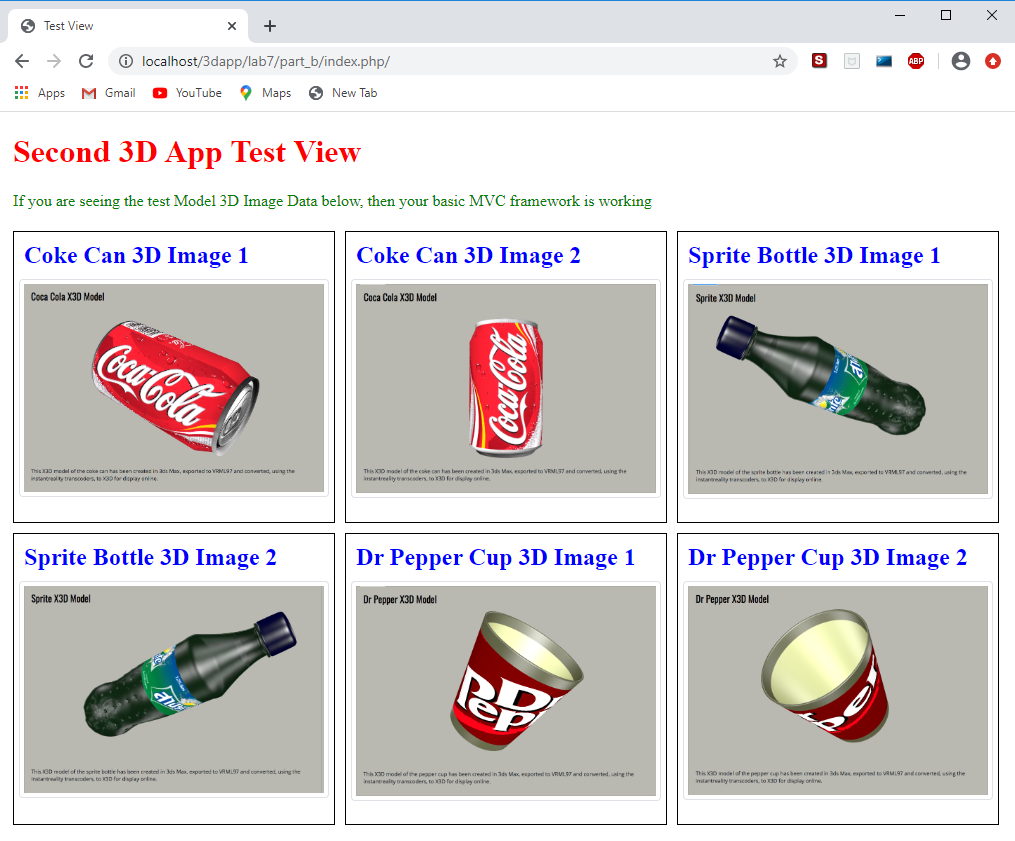
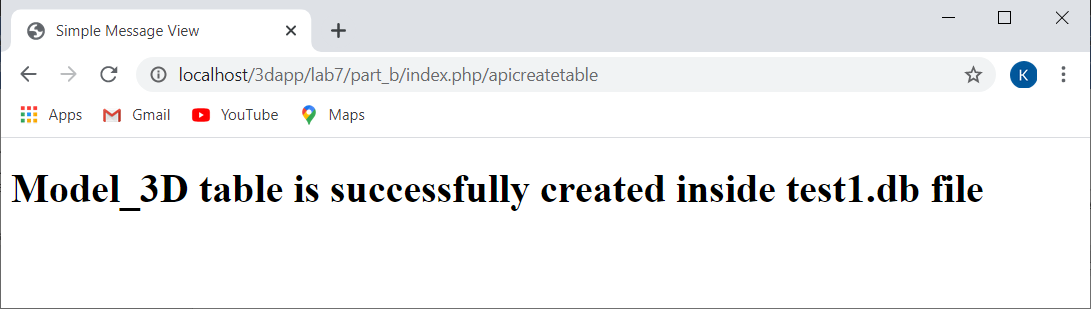


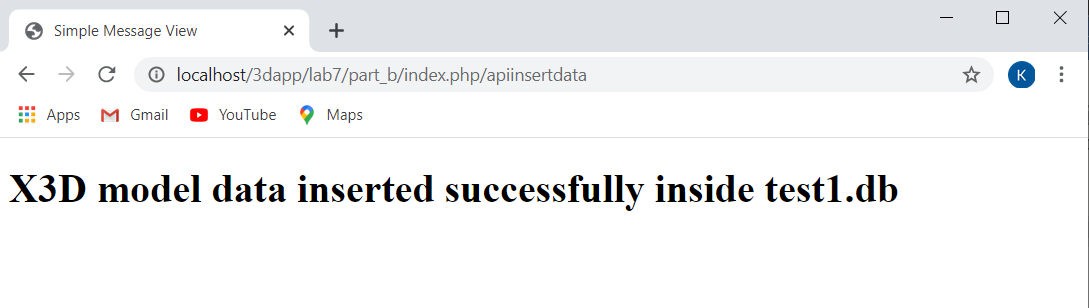
Figure 20: Accessing the controller methods via the URI

As soon as you invoke the index.php page, the back-end Model class makes a connection to the SQlite database as part of the Model class construct method. So, the database is now ready to use. Why is this? It is because Controller class construct method instantiates a new model class object.

Figure 21 shows how we first invoke the Model class dbCreateTable() method via the apiCreateTable() method in the Controller class, and a success message is returned from the dbCreateTable() method. We could have, and perhaps should have, also returned a message from the Model class contruct() method to indicate a successful database connection, but we relied on the absence of an error message instead. Figure 21 also show a successful insertion of data into the SQLite database table, and the data returned from the SQLite database table when the apiGetData() Controller class method is called in the URL.

Clearly, it is not a great stretch of the imagination to start developing new Model class methods to access data in different ways, create other tables, and so on. – we can think of this Model class as a data abstraction layer (DAL). Similarly, we can create new views that combine the data, e.g., combine the data in Figure 20 and Figure 21 — note that we haven’t entered the actual data from your Lab 6 data.json file yet, I left that for you to do.





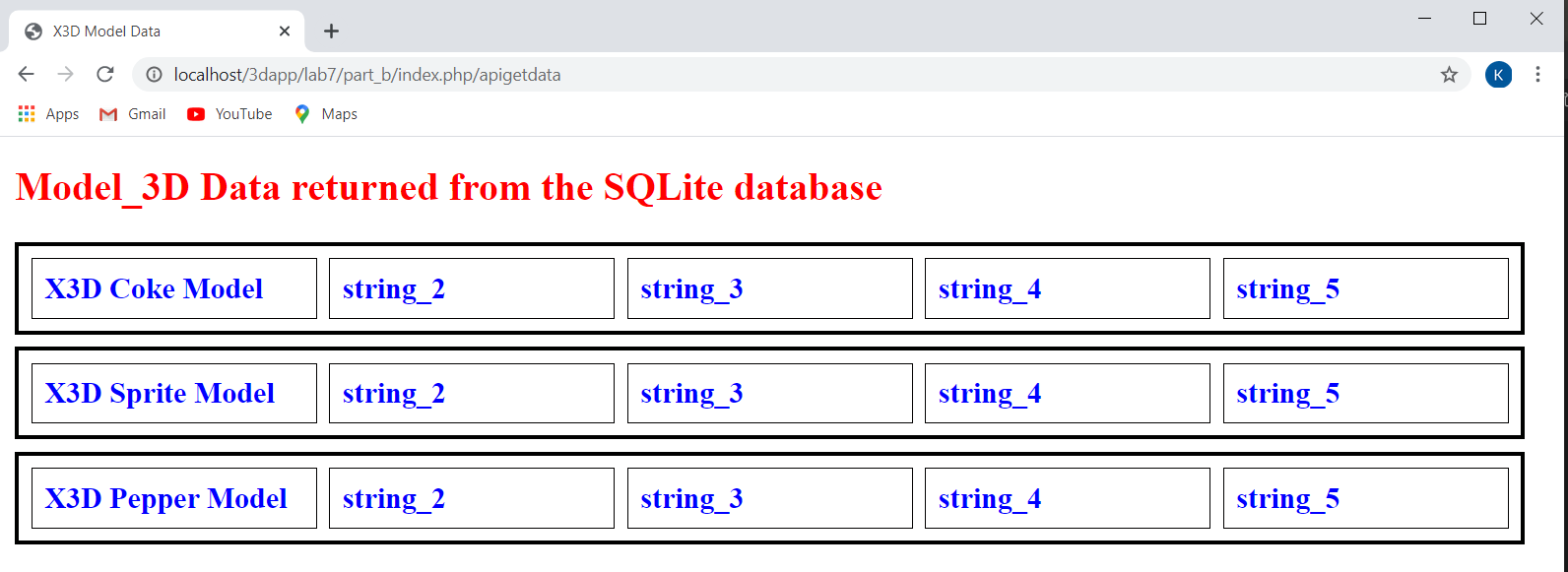


Figure 21: apiGetdata is used to get the database contents and display in an HTML table after the database table is created and data inserted

1. https://www.sqlite.org/ [↑](#footnote-ref-1)