TCS — Blockchain Tutorial 1

# Build a MVC Framework To Explore the Bitcoin Blockchain

The following tutorial relies on snapshot images of the code in the forthcoming tutorial descriptions. Unfortunately, it takes a lot more time to format code in the tutorials, hence the snapshots. However, this makes it more difficult for you to enter the code, as you would have to type it in using the image snapshots as a guide, which is prone to errors, etc.

So to alleviate this issue, I will make the tutorial code available so that you can cut and paste from it. As such, it will be useful to have the tutorial code open alongside the tutorial. So as not to mix up the downloaded tutorial code with the code you develop as yu read the tutorial, it might be a good idea to open both in separate code editors. For example, use Brackets to develop and Sublime Text or Notepad++ to view the downloaded tutorial code. Then you can cut and paste form Sublime Text into Brackets, etc.

You can download the tutorial code from <https://github.com/martinwh/bitcoin_blockchain_explorer>, but you will need to set up a GitHub account.

These tutorials are focused on exploring blockchain technologies largely in the context of the Bitcoin blockchain. Initially we have 6 tutorials:

1. **Blockchain Tutorial 1** — This tutorial, we will start by building a [model, view, controller](https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html) (MVC) design pattern. We will use this to implement our blockchain exploration code as the tutorials progress. We will test this MVC framework by connecting it to a SQLite database with pre-stored bitcoin data.
2. **Blockchain Tutorial 2** — Next, we will examine the blockchain concept further by building a simple JavaScript based blockchain. Although this will not be a real, fully functioning blockchain, it will serve to illustrate the blockchain structure.
3. **Blockchain Tutorial 3** — We will follow this by investigating bitcoin APIs to explore some Bitcoin currency exchanges. A natural consequence of this would be to build a Bitcoin to fiat money, e.g. USD, £, etc. converter application. We won’t have time for this at the moment, however you are welcome to look at last year’s task where we actually build a blockchain currency convertor, see the references in Study Direct.
4. **Blockchain Tutorial 4** — Next, we will explore the Bitcoin blockchain through a set of API endpoints probvided by blockchain.info.
5. **Blockchain Tutorial 5** — We follow on by exploring the bitcoin blockchain with the blockchain.info WebSockets to retrieve bitcoin data in real-time.
6. **Blockchain Tutorial 6** — Finally, we will build a bitcoin blockchain explorer using the Webchain APIs

There are many more tutorials we could develop, but what we have above is probably enough for the time available. If you want to take this further, a good initial read is a book called ‘[Building Bitcoin Websites — A beginners guide to Bitcoin focused on web development](https://www.amazon.co.uk/Building-Bitcoin-Websites-Beginners-Development/dp/153494544X)’. Indeed, the starting point for this tutorial series partially comes form the author of this book.

We could, of course, omit the MVC framework in the Blockchain Tutorial 1 and get straight to the point of using the APIs and WebSockets to look at bitcoin data — we won’t because it is useful to present our bitcoin data in different views (e.g. build an application), and also we need to select from different backend bitcoin blockchain data sources. However, no harm in taking a quick look now. For example, we can look at bitcoin digital currency exchanges using various third-party APIs. Usually a bitcoin exchange will publish a set of API endpoints that allow you to access information such as last price, low price, high price, timestamp, etc.

The following simple piece of PHP code, Figure 1, will allow you to access Bitstamp’s latest price along with the time stamp:

|  |
| --- |
| // Get the latest Bitcoin price from Bitstamp  $Bitstamp\_Url = "https://www.bitstamp.net/api/ticker/";  $json = json\_decode(file\_get\_contents($Bitstamp\_Url), true);  $priceBitstamp = $json["last"];  $timeStamp = $json["timestamp"];  echo "<b>Bitstamp Last Price was:</b> $";  echo $priceBitstamp;  echo "<b> on the </b> ";  echo date ('m/d/Y', $timeStamp);  echo "<b> at </b> ";  echo date ('H:i:s', $timeStamp);  echo "</p><p>"; |

Figure 1: A piece of PHP code to access Bitstamp’s latest bitcoin price

You can create a PHP file and insert this code, run it from a browser and it should return something like:

**Bitstamp Last Price was:** $8106.93**on the**11/20/2017**at**13:23:57

If you simply take the API endpoint (or resource): <https://www.bitstamp.net/api/ticker> and stick it in a browser, e.g. Chrome, you should see it return in JSON format the Bitstamp bitcoin exchange ‘name: value’ pairs as follows, Figure 2:

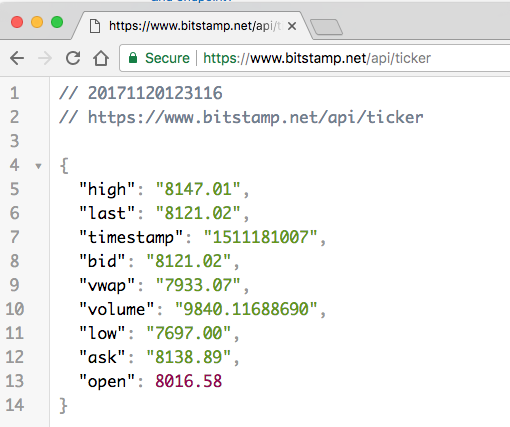


Figure 2: JSON response to the Bitstamp ticker API endpoint (JSON Viewer extension in Chorme)

Clearly, we can see that we can access information related to the current bitcoin exchange value. For the third tutorial (Blockchain Tutorial 3), we will take a look at several bitcoin exchanges, particularly Bitfinex, Bitstamp and Coinbase. There are others, and some of them seem a little bit flaky in terms making responses to their API endpoints. However, let’s first build our MVC framework.

# A Simple MVC Framework with PHP

Ok then, we don’t want to simply knock up a few bits of PHP code, as indicated above, to access bitcoin data. Instead we are going to move steadily forward towards building a simple blockchain explorer using the bitcoin blockchain as our source for the bitcoin data. So, to start, we will develop a simple MVC (model view controller) framework to hold our code.

Because this is currently only a 4-hour section of the overall TCS module I will be quite brief in parts. This means that you may have to take some of the framework code on trust rather than worry about the detailed working. You will also need to do further research offline! For example, if you don’t know anything about MVC, PHP, JavaScript, … then go do the research.

So, this tutorial (Blockchain Tutorial 1) implements a simple MVC framework; through this tutorial and some background research you should become familiar with the Model, View, Controller (MVC) design pattern. You are about to develop this simple MVC framework using technologies such as HTML5, CSS3, JavaScript, JQuery, PHP and SQLIte, along with libraries such as Bootstrap. I did consider using an existing framework, such as SLIM (a micro framework for PHP — [look it up](https://www.slimframework.com/)), but it is more instructive to develop our own simple version so that you can better understand the concepts.

In this tutorial, we will use the MVC design pattern whereby a view makes a request a **controller**, which then invokes a **model** to retrieve data from a data store (or model, which could be a JSON file, SQLite database, or a third-party API endpoint) and passes a respose back to the controller, which then updates the **view** to publish the response (e.g. bitcoin data) on a web browser.

This tutorial will demonstrate how easy it is to replace a view from a website without any modifications to the model and virtually no modifications to the controller, which is possible with MVC design patterns as it effectively separates out the data (or model) from the business logic (the control) and the web visualisation (or view).

We mentioned above that we will not simply create bits of PHP code to access the bitcoin information, neither shall we use procedural PHP. Instead, we will use classes and objects in PHP (later versions of PHP are object oriented), and although these are just simply introduced here some background reading may be needed if you are not familiar with [object oriented PHP](https://www.killerphp.com/tutorials/php-objects-page-1/). Here is a very short, but nice example of [procedural PHP versus object oriented PHP](https://gist.github.com/seqs/7382545) on GitHub.

Further, we will also introduce the notion of *data-access abstraction* using PDO (PHP Data Objects). The [PHP Data Objects](http://php.net/manual/en/intro.pdo.php) (PDO) extension defines a lightweight, consistent interface for accessing databases in PHP.

## Building an MVC Framework using PHP

Let’s now build our MVC framework using object oriented PHP using a step by step approach:

1. Start by creating a folder called bitcoin\_blockchain\_explorer. We will continue to use this folder and organise your following tutorials 2, 3, 4, 5 and 6 code in this folder integrated into the MVC framework, sort of.
   * You may already be using a localhost environment on your own PC or laptop, e.g. WAMP, MAMP, XAMPP or even the Mac built-in apache web server.
   * You may instead be using your public\_html environment as your Local Folder (if you haven’t got this set up simply do a search on Google for something like: ‘Sussex ITS web space’ and follow the instructions to set up your ITS web space.
2. Create a file named as ‘index.php’ any text editor, e.g. Notepad++, Brackets, Sublime Text (I am currently using Visual Studio Code as I have just discovered it is easy to integrate with [GitHub — a version control repository and hosting system](https://github.com/).
   * Now type the following lines in ‘index.php’ file, Figure 3.
     + Note, when I say ‘type’, you can of course type, but if you have downloaded the tutorial code, as indicated at the start, from GitHub you can simply ‘cut and paste’.

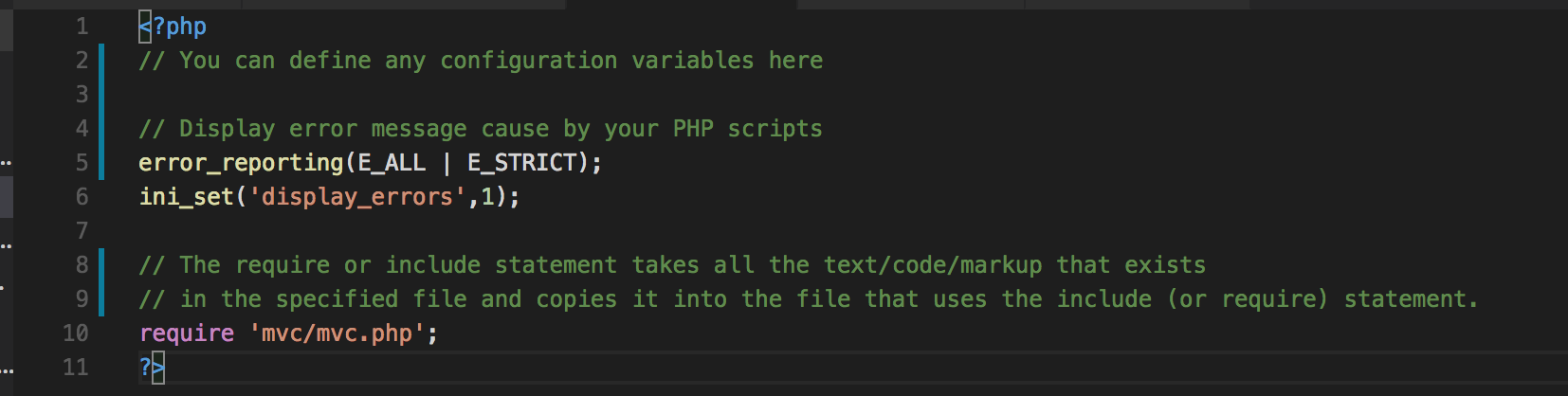


Figure 3: The index.php file

* + Code explanation
    - Line 1 and 11: ‘<?php’ denotes the start of the PHP code, and ‘?>’ denotes the end of the PHP code.
    - Anything after ‘//’ is treated as comments, and these are only used for code readability. The PHP compiler (embedded in the browser) will simply ignore the whole line, starting from the ‘//’ characters.
      * We have added these some comment lines to show how to use comments in your PHP code.
    - Line 5 and 6: The PHP library function ‘error\_reporting()’ and ‘ini\_set()’ enables debugging of your code, i.e. any errors generated will display when your PHP script has any bugs. If you remove this, you will not be able to see any errors in your code and your browser will block the site in case of any error — you will simply be left scratching your head wondering where the bug is! So, it is very important that you keep these functions in your code to see any error reporting during PHP code development. As the PHP functions we will develop are relatively simple we will manage with this error reporting, but for anything more sophisticated you might want to introduce other [PHP debugging options](http://php.net/manual/en/debugger.php), which range from printing out variables, building a console\_log function, to including a debugging library.
    - Line 10: The require 'mvc/mvc.php'; tells the browser to load or include a file named ‘mvc.php’ from the ‘mvc’ folder into the current file.

1. Effectively, this index.php file is acting as a configuration file to define global variables and URLs, since this file is the starting point of our site.
2. Create a folder called ‘mvc’ inside your ‘’ folder
3. Then create three subfolders inside the ‘mvc’ folder, namely:
   * controller
   * model
   * view

Your folder structure should now look like that shown in Figure 4.

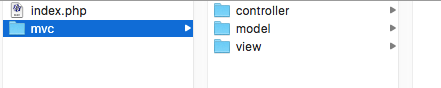


Figure 4: Set up for the MVC folder structure on a Mac, should be a similar structure on a PC

* + As you can see, we now have three different folders to hold the ‘model’, ‘view’ and ‘controller’; we will make sure that we store:
    - The data access code inside the ‘model’ folder,
    - Front end presentation code in the ‘view’ folder,
    - And business logic or control code in the ‘controller’ folder.

## Adding SQLIte with PHP to the MVC Framework

In this tutorial, we will use SQLite with PHP in your MVC framework to store locally any data we wish to present in a view, e.g. bitcoin data. This makes the view more dynamic. SQLite is a very convenient executable database, more precisely:

“SQLite is an in-process library that implements a [self-contained](https://www.sqlite.org/selfcontained.html), [serverless](https://www.sqlite.org/serverless.html), [zero-configuration](https://www.sqlite.org/zeroconf.html), [transactional](https://www.sqlite.org/transactional.html) SQL database engine. The code for SQLite is in the [public domain](https://www.sqlite.org/copyright.html) and is thus free for use for any purpose, commercial or private. SQLite is the [most widely deployed](https://www.sqlite.org/mostdeployed.html) database in the world with more applications than we can count, including several [high-profile projects”.](https://www.sqlite.org/famous.html)

As mentioned, to become more familiar with MVC concepts we are creating our own MVC framework, and we envisage the need to make it dynamic and capable of storing local data. To meet this need we will need to connect with the aforementioned SQLite database using PHP. Our PHP code to do this will largely cover several functionalities:

* + Connection to the database
  + Creating tables in the database
  + Inserting data into the database tables
  + Retrieving data from the database tables
  + Deleting data from the database tables
  + Closing the connection to the database

We will also introduce [PHP Data Objects](http://php.net/manual/en/intro.pdo.php) (PDO) as a data-access abstraction, “which means that, regardless of which database you're using, you use the same functions to issue queries and fetch data”[[1]](#footnote-1).

### Why are we using SQLite?

Putting aside the description of SQLite above as a good enough reason, we could use the University web server’s mySQL database, but that is impractical for teaching purposes. However, I do believe that the lab machines now have a local host installed, so we could use an installation of a localhost with its MySQL (or SQLite) installation. There are however, many other solutions for creating a data storage mechanism. But, we will use SQLite, which is a relational database management system developed in the C language. This is an extremely small size database management system — roughly about 350 Kbyte in size. Every time we create a new database using SQLite, it generates a small portable executable file that can be copied over to a different location.

This means we can locate our SQLite database, effectively a C executable, in your public\_html space (or on your local host).

### Building an MVC Framework using PHP and SQLite

We already created our MVC framework folders above, let’s continue using the same framework to progress further in terms of database operations. But first, let’s confirm that we have SQLite installed with our PHP installation. To do this, create a PHP file inside your public\_html folder and name it ‘phpinfo.php’

1. Now copy following three lines of in it, Figure 5:

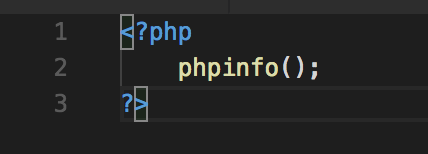


Figure 5: The phpinfo,php file for running the phpinfo() function

Let’s see its output by typing its URL into a web browser, e.g. http://localhost:8888/bitcoin\_blockchain\_explorer/phpinfo.php

If you are using your University web space, the host path will be different (users.sussex.ac.uk/) and need to incorporate your your username (~username/) and path (bitcoin\_blockchain\_explorer/phpinfo.php )to wherever you have your files located, of course, Figure 6.

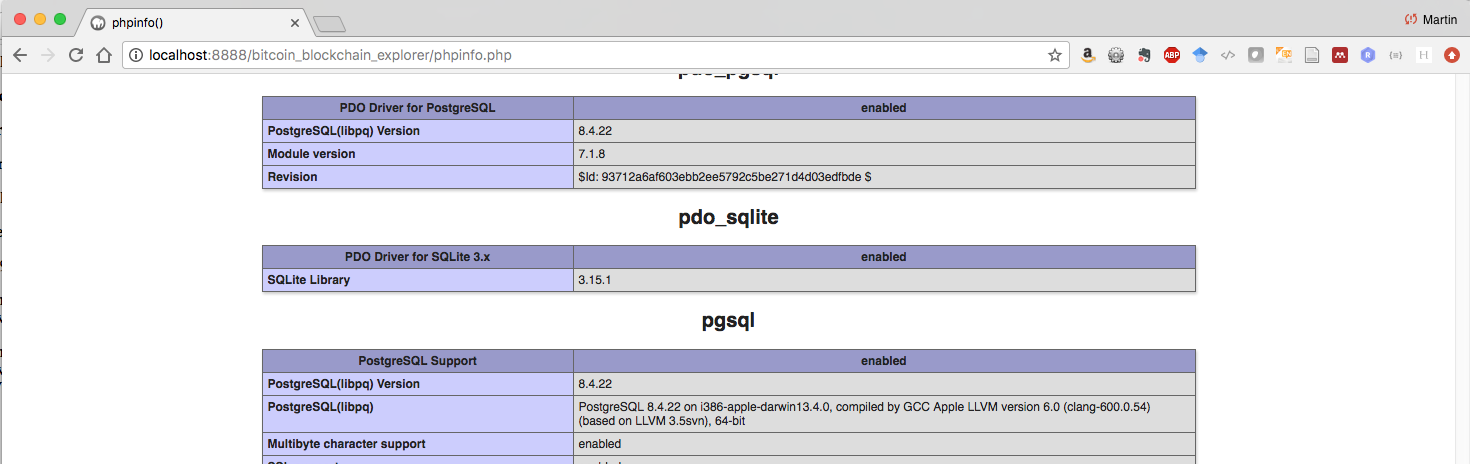


Figure 6: Check the PDO driver for SQLite3 is enabled

On the ITS web server your URL might be:

users.sussex.ac.uk/~your-user-name/bitcoin\_blockchain\_explorer

1. Create a new folder inside your model folder and name it ‘db’. Your MVC folder structure should show the db folder under the model folder, Figure 7.
   * Don’t worry about the rest of the MVC structure indicated in Figure 7 yet, you haven’t got there yet — Figure 7 is a leap forward from Figure 2; this is because I have taken a snapshot of my developing structure as I write this tutorial. I am developing the code, then going back in time to write the tutorial …

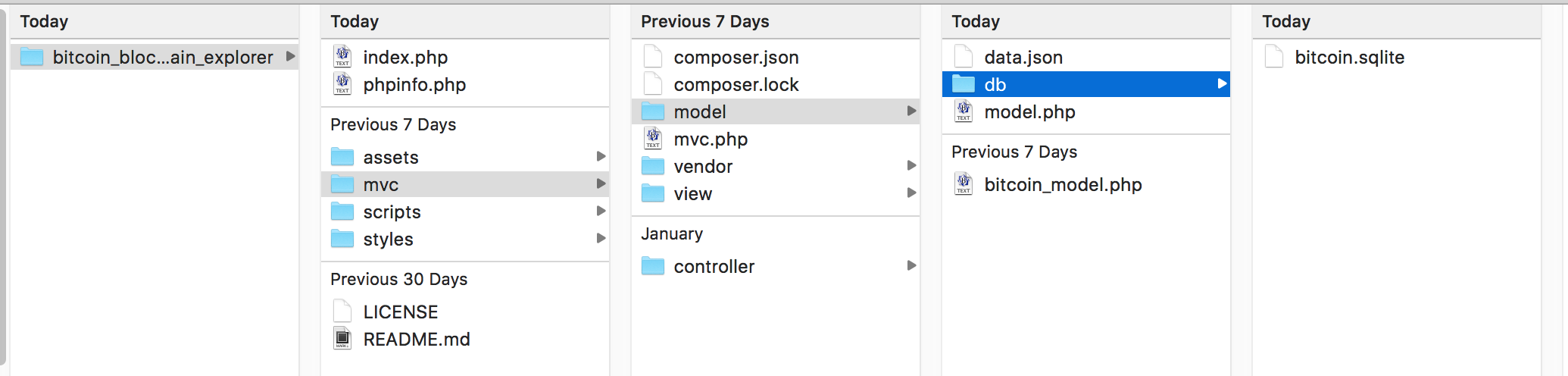


Figure 7: Adding the db sub-folder to your MVC structure

1. You have created a ‘db’ sub-folder in your Model folder, but this folder may not be script write-enabled on the ITS web server at the moment, i.e. you can read any data from this folder, but you can’t write anything through a PHP script. If you are using your localhost you don’t need to worry about this. But, if you are using your ITS web space to do this tutorial you will need to write enable this folder. To do this, download putty, login to the unix.sussex.ac.uk server and assign read write permissions to it. Follow the steps below to assign read and write permission to your ‘db’ folder.
   * Download putty.exe tool from the URL: ‘http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html’
   * Type as shown in the image below, see Figure 16.
     + Host Name: unix.uscs.susx.ac.uk
     + Port: 22
     + Click open

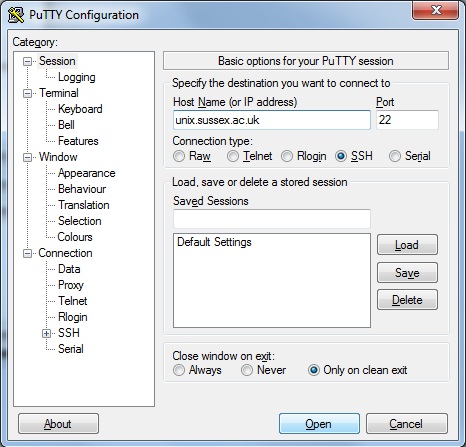


Figure 8: Putty interface

* + On the shell screen, Figure 9, you will be asked to enter your login user name, enter your login user name.
  + Type following commands:
    - cd public\_html/ mobile3dapp/lab7/part\_b//
      * Or wherever you have your ‘db’ folder located
    - touch db/
    - chmod o+rw db/
    - exit

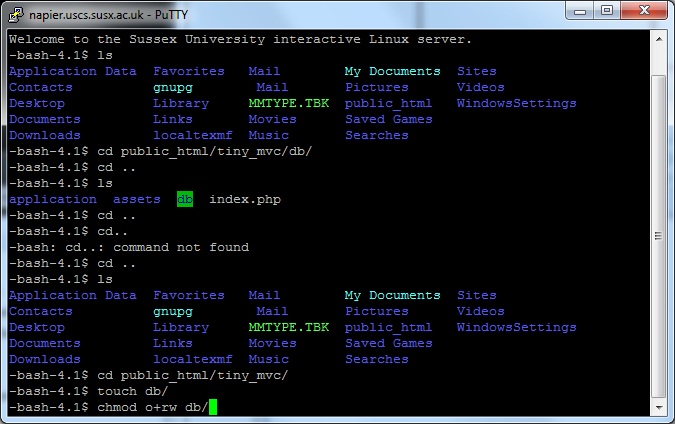


Figure 9: Changing db permissions to allow read and write

* + If you are working on a localhost, you should be ok. However, you’ll need to do this if you port to public\_html.

## Coding the Model, View and Controllers

Now let’s get down to developing the code for the model, view and controller.

1. Before we create code inside those model, view, controller folders, Figure 4, let’s create an entry point for our application. This PHP file will refer to all three components, model, view and controller and it will initialize (create an instantiation) the controller of our application.
2. Create a PHP file inside ‘mvc’ folder and name it as ‘mvc.php’
   * Now type the following code in it and save the file, Figure 10.

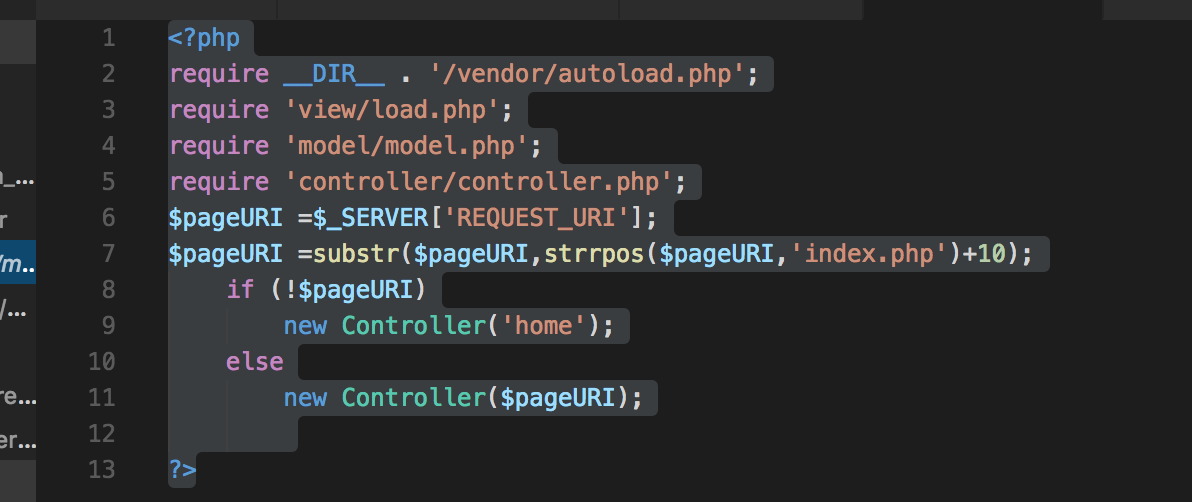


Figure 10: The PHP entry point file — mvc.php

* + Code explanation
    - Ignore line 2, i.e. don’t input this. As mentioned above, I am taking screen shots of currently developing code, which is ahead of this stage. However, here, I have just installed composer to manage libraries and we aren’t using any libraries at the moment — I am looking ahead, so to speak. The autoload.php file does what it says on the tin, manages and auto loads any required libraries that you specify. As I say, I haven’t started to use it properly yet.
    - Lines 3, 4 and 5 are code to include the three different files from the three folders (model, view and controller folders), these statements will include three PHP files (load.php, model.php and controller.php). This means from this point we can use any Class function (methods) or variables from within these three files (load.php, model.php and controller.php) in our application.
      * Just as we included with the ‘require’ in the index.php the mvc.php file so at this level inside the mvc.php we include other php files.
    - Line 6: ‘pageURI’ is assigned the value of the URI (Uniform Resource Identifier)
      * For example: $SERVER[‘REQUEST\_URI’] will return the filename, file, and in this case the method: bitcoin\_blockchain\_explorer/index.php/bcexplorerbs from the URL <http://localhost:8888/bitcoin_blockchain_explorer/index.php/bcexplorerbs>
    - Line 7: ‘pageURI’ is now assigned the value (i.e. the method or function) of the controller (in this example: bcexplorerbs) from URI using the substring function.
      * To be clear: The substring function (substr()) above will return ‘bcexplorerbs’ from the URI: ‘/index.php/ bcexplorerbs’. It does this by using the strrpos() function to find the position of the last occurrence of index.php in the URI stored in the variable $pageURI and then adding 10 (the number of characters in index.php/ to find the starting character of the controller function (method) being called, which in this example was bcexplorerbs.
    - Line 8 to 11: Here, we make a decision, if the ‘pageURI’ variable gets nothing from line 7 (for example, when you only enter ‘…/index.php’ so that you have not entered a name of a controller method to call), then the default controller is called with parameter ‘home’. Clearly, if the variable ‘pageURI’ is not empty then a controller method is called by the name set as a value of the ‘pageURI’ variable. It stands to reason, if you enter a name for a controller method after index.php, then that controller method must exist. At the moment, we have no error trapping, so if you type something like …/index.php/foobar then a PHP fatal error will occur, Figure 11. Here, we can see that the method foobar does not exist.

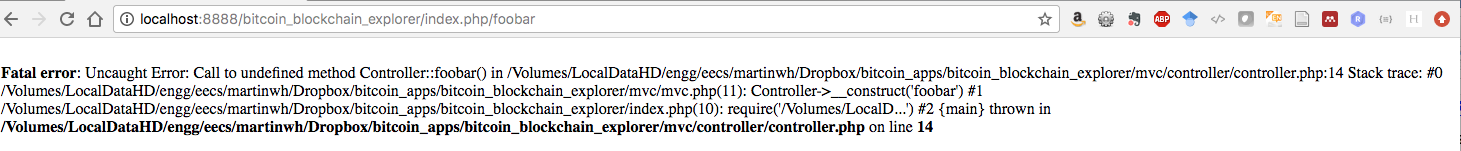


Figure 11: Oops, what is this foobar method? Should really do some error trapping!

* + - So, in this example the value of the ‘pageURI’ variable is the name of controller method. The new Controller($pageURI); php code will call the controller constructor from the Controller class and pass the name of the controller method as the parameter.
    - If the user types no controller name, e.g. if a user types: http://localhost:8888/bitcoin\_blockchain\_explorer/index.php, the home controller method will be called as a default. The home controller method would, of course, need to present a view to the user.

### Let’s build the controller class

Now let’s create our core components of the MVC framework starting with the controller class. Since, we are going to have many controllers in our example, we need to identify the names of these controllers from the browsers URL.

1. Create a PHP file inside the ‘controller’ folder and name it as ‘controller.php’, then type the following code in it, Figure 12.
   * For example: In the example URL below the name ‘dbReadeBitcoinData’ at the end of the URL after index.php is a name of a controller method, whose function is to get bitcoin data from the database (it is useful to call your method something obvious to do with its function):
     + users.sussex.ac.uk/~your-user-name/bitcoin\_blockchain\_explorer/index.php/dbreadbitcoindata,
   * And, the corresponding version on a local host
     + http://localhost:8888/bitcoin\_blockchain\_explorer/index.php/dbreadbitcoindata
   * Now type following code in it.

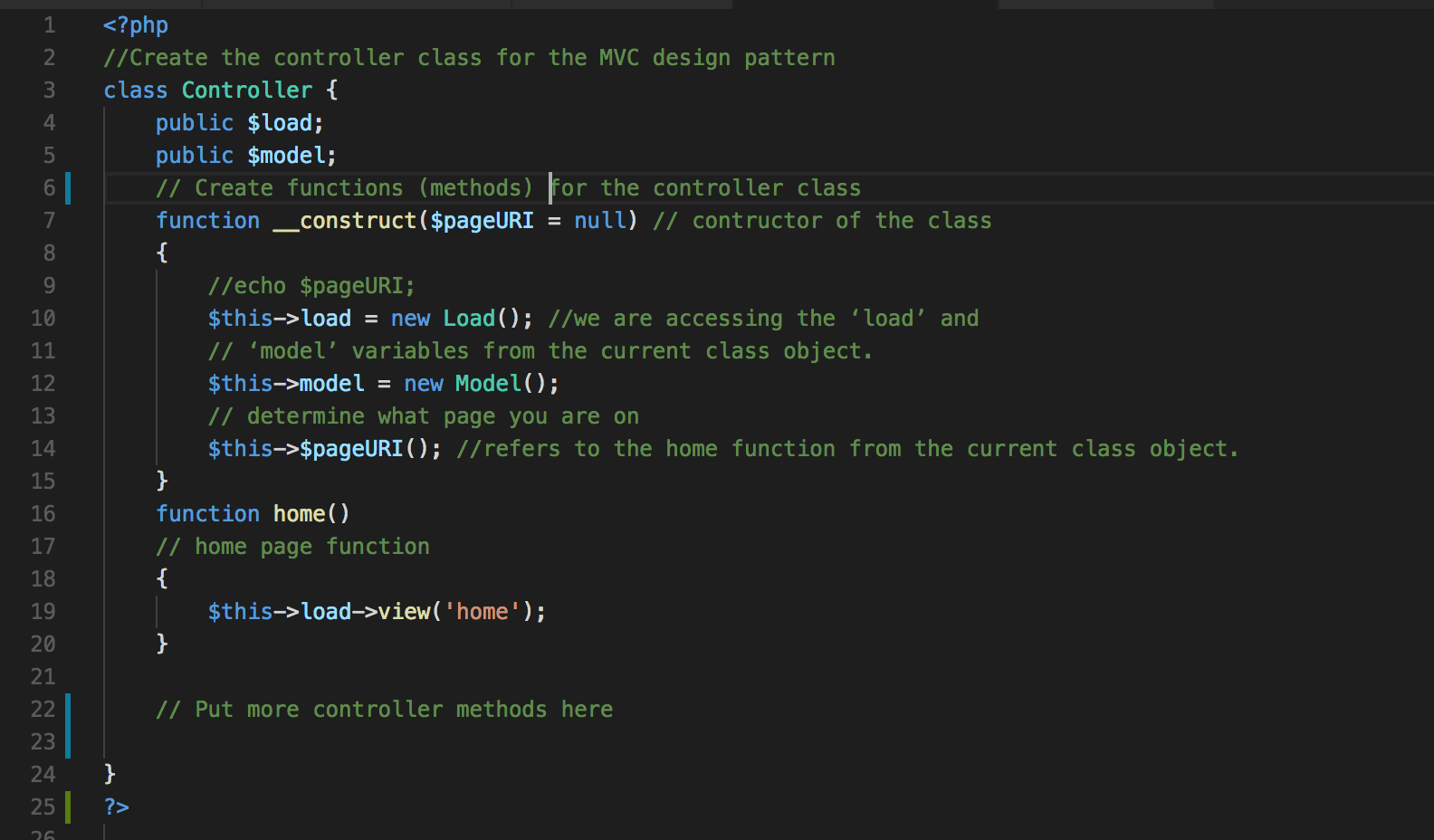


Figure 12: The initial PHP controller file

* + - Lines 3 to 24: define a class called ‘Controller’. Everything between parentheses ‘{‘ and ‘}’ is the body of the class.
      * **Class definition**: A class is a construct, which contains variables and functions. A user can create multiple instances/objects of one class, each object has separate memory space and has a separate copy of all members defined in that class.
    - Lines 4 and 5: Define ‘load’ and ‘model’ variables, and since they are declared as ‘public’, they can be accessed from anywhere using the instance of the class.
    - Lines 7 to 15: We define a constructor of the class, again everything between parentheses ‘{‘ and ‘}’ is the body of constructor.
      * **Constructor definition**: A constructor is the first function that is executed when a class is initialized. This means anything in constructor will execute immediately as soon as we create an instance of the class.
      * Why the two underscores?
        + The constructor can be by passed by specifically calling the Home function in a URL?
    - The pseudo-variable ‘$this’ refers to the current object of the class
      * ‘$this->load’ and ‘$this->model’ means we are accessing the ‘load’ and ‘model’ variables from the currently instantiated class objects for .
    - ‘new Load()’ defines a definition of a new object of the class named ‘Load’. Effectively an instantiation of the Load Class that will be defined in the load.php file, when we write it.
    - Similarly, ‘new Model()’ defines a definition of a new object of the class named ‘Model. Effectively an instantiation of the Model Class that will be defined in the model.php file, when we write it.
    - Line 7: We have now added a ‘pageURI’ parameter to our \_\_constructor.
      * This is will receive the name of the controller from ‘mvc.php’ file
    - Line 14: $this->$pageURI
      * This means, we are now calling a function name identified by ‘pageURI’ variable. We plan to have many functions (or methods) inside the controller class, each one picked off by the mvc.php file from the URL.
    - Each function is a separate controller method and it can now call different models and views.
    - Lines 16 to 20: Contain a definition of the ‘home’ function.
    - Line 19: $this->load->view('home');’ refers to the home function from the current class object.
    - Lines 22 onwards: Here we place definitions of new functions, i.e. controller methods in this case.
      * As mentioned above somewhere, we are going to create several controller methods for accessing a SQLIte database, which we will use initially for storing some test bitcoin data.
      * function dbCreateBitcoinData() — this will be used to write some test bitcoin data to the SQLite Database, may be used eventually to record bitcoin data gather via APIs. But, initially used for testing.
      * function dbReadBitcoinData() — This will be used initially to read test bitcoin data from the database, can be faster when testing new views, for example, rather than waiting for AJAX based request/response round trip calls.
      * function dbDelete() — simply deletes the database tables, requiring them to be created again. Likely to be needed if you develop further and find you are running out of resource.
  + Add, the following code to the controller.php file, Figure 13.

### Screen%20Shot%202017-11-20%20at%2015.59.35.png

Figure 13: Some initial controller methods.

### If you want to check that these controller methods, and your MVC framework, is actually working ok at this stage, stick some PHP echo statements in as indicated in Figure 13, obviously uncomment them and comment out the other code in each method. You can then run the methods as indicated above, Figure 14.

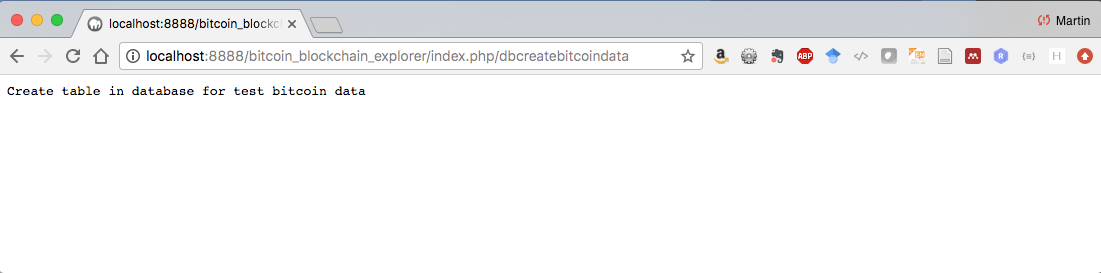


Figure 14: Test that your create, read and delete methods work ok.

By doing this and getting the right result we know that your MVC framework is working so far.

Let’s explain what the actual methods are going to do, obviously we have to write the models to accompany them

* + **function dbCreateBitcoinData()** in the Controller Class — this is a method that calls the currently loaded model class, which has a function or method called (you guessed it) dbCreateBitcoindata() in a Model Class (that we haven’t written yet). Don’t be confused because these methods have the same name, we could have called it something else. I only did this to emphasis the functionality, just note that at some point you will create a view that calls the Controller method (dbCreateBitcoindata())that calls the model method (dbCreateBitcoindata()), hence a separation of code.
    - Line 27: here the data variable receives the response returned from the model, as in $this->model, the one that was instantiated in the Controller constructor method. So, $this->model->dbCreateBitcoinData() is referring to an instantiation of the dbCreateBitcoinData() method (or function) that we have yet to create in the model.php (that we required in the mvc.php file) file’s Model Class.
  + **function dbReadBitcoinData()** in the controller class — this code is similar to that described above, except here:
    - Line 35: we read the table bitcoin data, stuff it into the data variable, and
    - Line 36: echo it out after converting it to JSON format. Doing it this way will allow the calling function, i.e. a JavaScript call, perhaps using AJAX and JQuery to use the Controller dbReadBitcoinData() method as a method in a URL through, for example, the jQuery.getJSON() function.
  + **function dbDelete()** in the controller class — and so on. Ok, here we anticipate a response that simply triggers a message sent to a view via the controller; same in the dbCreateBitcoindata() method.
  + We will create some more methods later, first let’s create some models.

### Let’s create some model classes

Ok, now we have a feel for the Controller Class and its methods, let’s develop some corresponding Model Classes.

1. Now create another file called ‘model.php’ inside ‘model’ folder
   * Type following code in it, Figure 15.

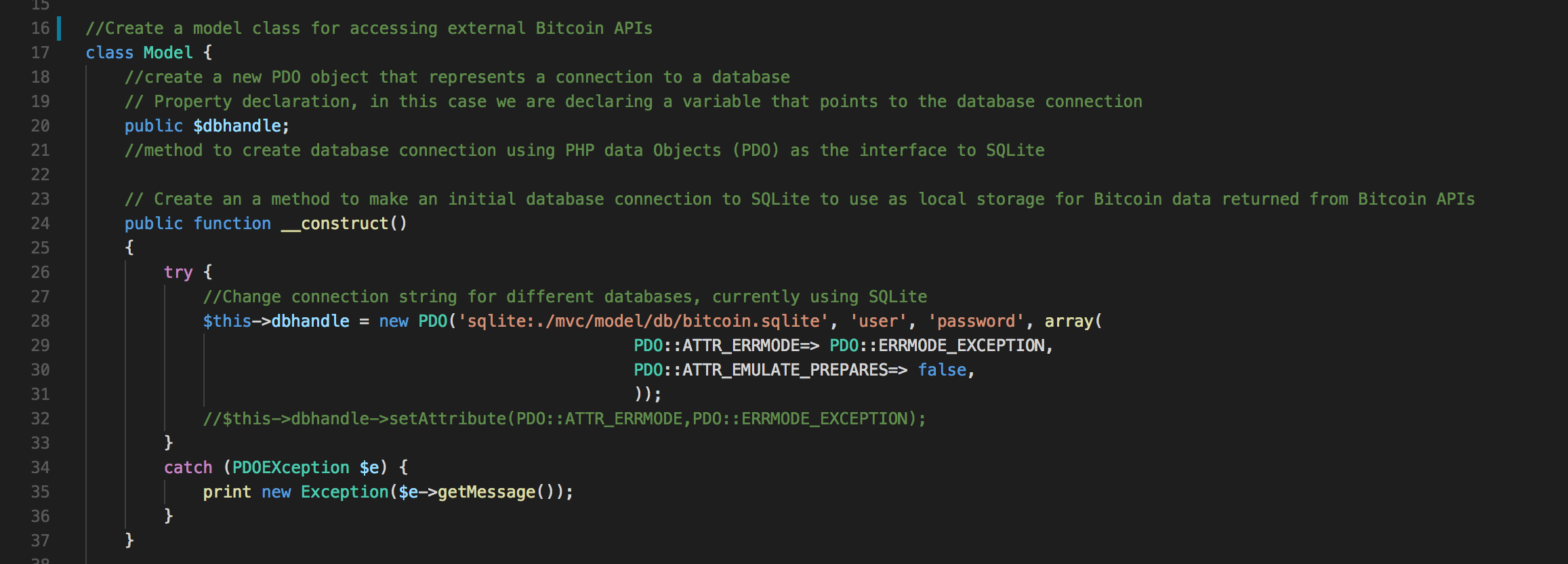


Figure 15: The PHP Model file — Connect to the SQLite database using PDO

* + Code explanation
    - Lines 17 to the last } bracket: Defines a class called ‘Model’. Everything between parentheses ‘{‘ and ‘}’ is the body of the class.
      * Recall: a class is construct, which contains variables and functions. A user can create multiple instances/objects of one class, and each object has separate memory space and also has a separate copy of all members defined in that class.
    - Line 20: Contains declaration of a public variable named ‘dbhandle’
    - Lines 24 to 40: We now have a constructor for the model class that creates a PDO object
    - Line 31 to 36: Assigns the new PDO (PHP Database Object) to the $dbhandle variable.
  + The PDO object first specifies a database connection string ($DSN) starting with the type of database, in this case SQLite, followed by a path to the database and database name, which in this case the path is the db folder and the database name is bitcoin.sqlite — this is referred to as a DSN or database source name.
    - If you now wanted to change the database to another database, e.g. mySQL, you would only need to change the DSN.
  + Next you pass in the user name ($user) and password ($password) in this case.
  + For development learning, it is a good idea to set the PDO error mode attribute PDO::ERRMODE\_EXCEPTION, which throws an exception if there is an error.
    - Also, to prevent [SQL injection attacks](https://www.youtube.com/watch?v=GBDbclDfc84) you must set the PDO::EMULATE\_PREPARES to false. Both attributes can be set by passing them into the new PDO function as an array for the fourth parameter. Alternatively, you can set these attributes using the setAttributes function, see the commented out code Line 32.
    - However, we won’t necessarily worry too much in this tutorial about SQL injection attacks and how to prevent them using [prepared statements](http://us3.php.net/manual/en/pdo.prepared-statements.php), because later we will be largely using API calls rather than the database. But, you might like to read up some more on this topic. You could also use a [PHP PDO Wrapper Class](http://www.imavex.com/php-pdo-wrapper-class/), such as that provided by IMAVEX to make life easier. Also, w3schools.com have a good example on how to use [PHP Prepared Statements in PDO](https://www.w3schools.com/php/php_mysql_prepared_statements.asp) if you want to look at this further.

1. Line 34 to 37: If the connection to the database fails PDO will throw a PDOException, which can be trapped in the catch part of the try/catch block
2. This should provide a successful connection to the database!

Now let’s create the three functions we need to create the table, read the table and delete the table for the bitcoin test data.

1. Create the function dbCreateBitcoinData() by typing in the model.php file the following code, Figure 16.

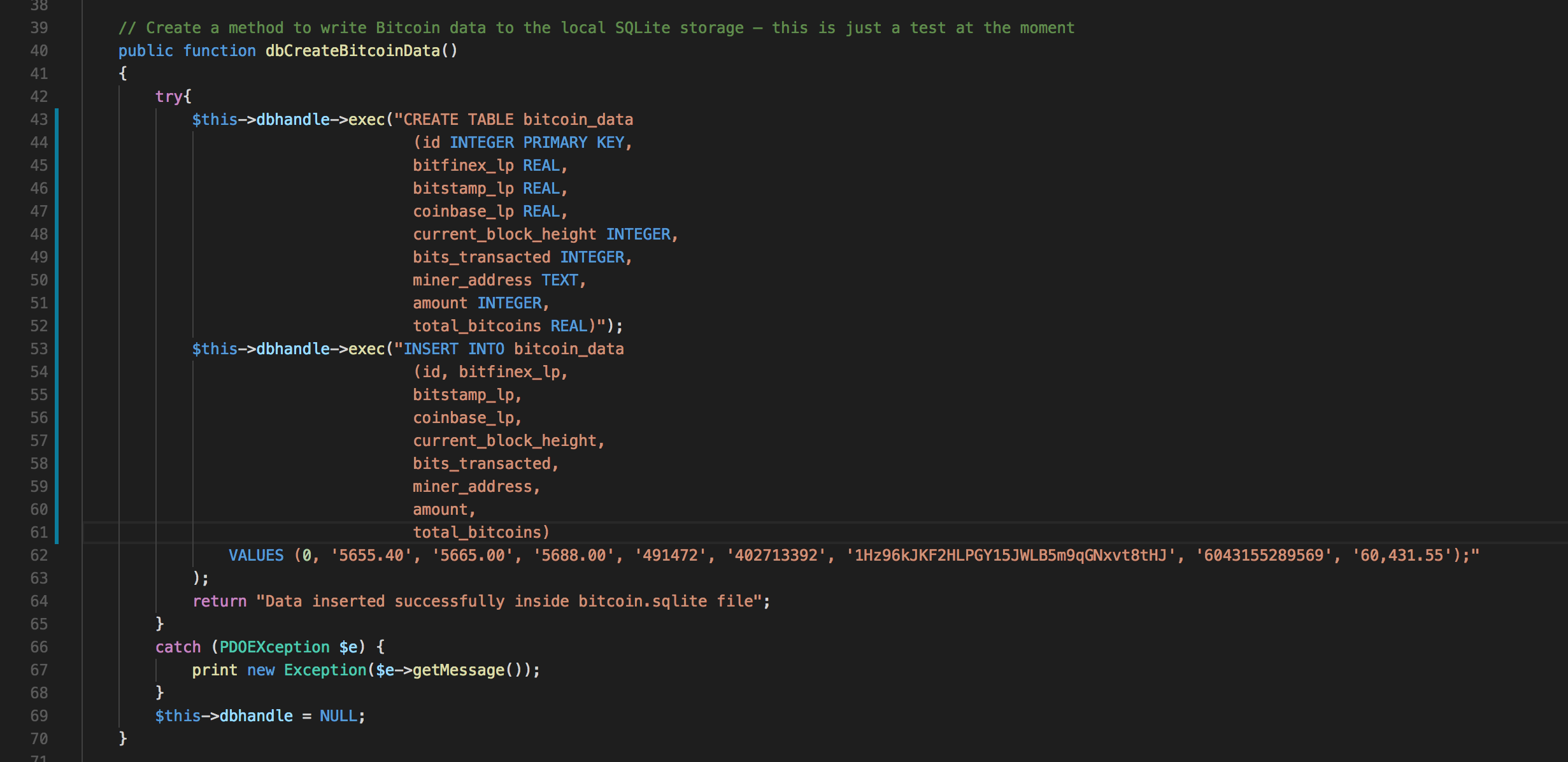


Figure 16: PHP method to create the bitcoin test data

**Code explanation:**

* Line 43: The PDO::exec() function executes an SQL statement in a single function call, where in this case the SQL statement is designed to create a table in the database. The SQL query string:
  + Creates a table named ‘bitcoin\_data’ with nine field names:
    - ‘Id’ field: an integer primary key field, which is automatically incremented, when we enter any data in all other fields. So, its default values would be 1,2, 3, 4, and so on.
    - ‘bitfinex\_lp’ field: stores a REAL number, which in this case is the last price of bitcoin on the bitfinex exchange.
    - ‘bitstamp\_lp’ field: stores a REAL number, which in this case is the last price of bitcoin on the bitstamp exchange.
    - ‘coinbase\_lp’ field: stores a REAL number, which in this case is the last price of bitcoin on the bitfinex exchange.
    - ‘current\_block\_height”: stores INTEGER, which is the actual number of the last block appended to the bitcoin blockchain.
    - ‘bits\_transacted’: stores the number of bits transacted in this block on the blockchain, note this is not the bitcoin value.
    - ‘miner\_address’: This is an address STRING of the actual miner that validated this transaction on this block, who then appended the block to the bitcoin blockchain,a nd was subsequently rewarded with , at today’s arte, 12.5 BTC.
    - ‘amount’: This is the amount of SATOSHI, an INTEGER, that this miner holds at this miner address, you need to divide by 100,000 to get BTC.
    - ‘total\_bitcoin’: This is a REAL number of BTC that this miner holds.
* Line 53 to 63: Clearly insert data into the table just created. This test data was simply gather by executing various API calls in the browser and manually picking off the appropriate JSON response values. It is interesting to note that this was done around the start of November 2017, and by the 20th November 2017 the value of bitcoin has increased from around $5,500 to around $8,250!
* Line 66 to 69: If the table has not been not created the PDO exec function would throw an error, which is trapped in the catch block.
* Line 69: Closes the database connection, which was opened in the \_\_constructor of the model class.
* Lastly line 64: Returns a success message to the function calling program, i.e. the controller.
* Note, we created both a table create functionality in this method and then inserted data into the table. In reality, you would need better refined methods, e.g. connect to the database, crate required tables according to some DB schema, and separate methods to insert, update, read, delete data from the table, etc. Also, note that we didn’t use prepared statements!

1. Create the function dbReadBitcoinData() by typing in the model.php file the following code, Figure 17**.**

* Lines 78 to 79: sets up an SQL query to retrieve all data from the bitcoin\_data table, an object of the data is created and stored into the ‘query’ array. PDO::query() executes an SQL statement in a single function call, returning the result set (if any) returned by the statement as a PDOStatement object.
* Line 88 to 102: contains a while loop, that repeats its body for the number rows found in database through the query.
  + The PDO::Statement::Fetch fetchs a row from a result set associated with a PDOStatement object.
  + The values for each row are stored in an array named ‘result’, so that we can easily pass it to the ‘view’ for displaying on the screen.
* Line 110: closes the database connection, which was opened in \_\_constructor of the model class.
* Line 113: returns an array containing values of all data from the database bitcoin\_data table.

1. Create the function dbDelete() by typing in the model.php file the following code, Figure 18**.**

* This code is fairly self-explanatory, and probably a wee bit simplistic, but hey, we haven’t got all year!

****

Figure 17: PHP method to read the bitcoin test data

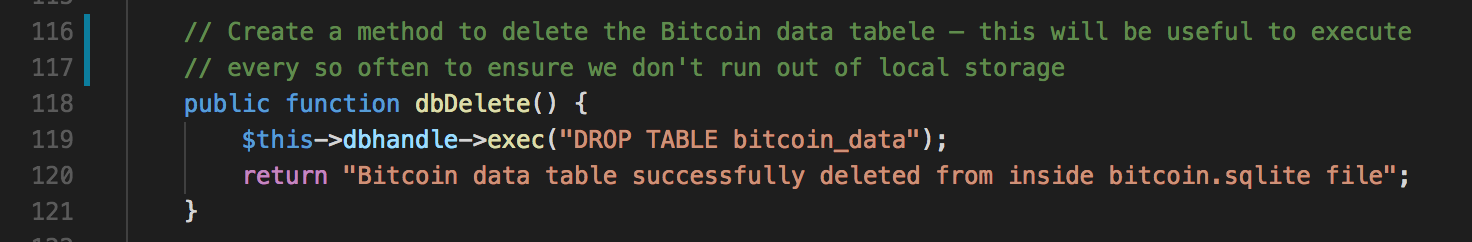
****

Figure 18: The PHP delete method to drop the bitcoin\_data table

Now, baring in mind that we are going to create some more model functions (methods) later to access bitcoin data through third party APIs, we have enough models and controller functionality to get going. Go back to the controller.php file and make sure you understand the connection between the controller methods and the model methods. It’s a good idea to have both files open together using a code editor with a split screen, e.g. Brackets or VS Code.

Assuming we have good Controller and model code, we can now start to build some views to present the bitcoin data (in the SQLIte database model) to the front end (view). Here we will use HTML5 and CSS3 along with JavaScript and JQuery to make AJAX calls to the controller methods.

### Let’s knock up some views

Ok then, I have some old Dreamweaver grid based HTML code lying around that I have adapted to create a simple view — just to get a quick look at the bitcoin table data. But, first we need to create the load Class for the views.

1. Now let’s create our view. Since in this tutorial we are going to create a number of different views for the same class, we will need different HTML files (one for each view) and one intermediary .php file to select (or load) those different views according to the instructions provided by the controller.
2. So, create a file named ‘load.php’ inside the ‘view’ folder. You can begin to see now why we called this file load.php and not view.php. The role of load.php will be to load a view, where a view will be an html (or PHP if the file contains a PHP script) file.
   * Now type the following code in the load.php file, Figure 19.

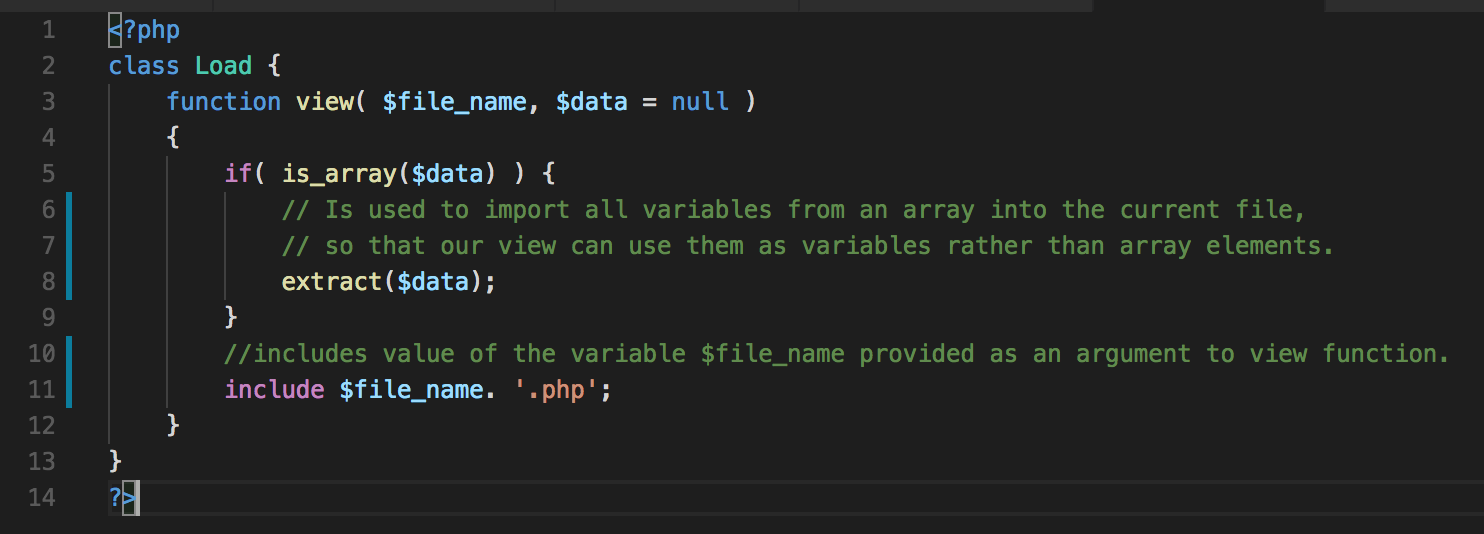


Figure 19**:** The PHP load file

* + Code explanation
    - Lines 2 to 10: defines a class called ‘Load’.
      * Everything between parentheses ‘{‘ and ‘}’ is the body of the class.
    - Lines 3 to 9: defines a function called ‘view’.
      * Everything between parentheses ‘{‘ and ‘}’ is the body of the function.
    - Line 5: the ‘if’ statement checks the ‘$data’ variable to see if it is a pointer to an array? If it is a pointer to an array, the program control goes to line 8, if it is not pointer to an array the program control will skip line 8.
    - Line 6: a predefined php function named ‘extract’ is used to import all variables from an array into the current file, so that our view can use them as variables rather than array elements.
    - Line 8: includes value of the variable $file\_name provided as an argument to the view function.
      * **I**n the controller class method dbCreateBitcoinData() we passing ‘view\_simple\_message’ string to the ‘view’ function, so in above case the value of the variable ‘$file\_name’ is ‘view\_simple\_message’.
      * Succeeding the variable $file\_name there is a ‘.’ character, which is used for concatenation of two strings. In above case the value of $file\_name (which should be ‘view\_simple\_message’) is concatenated with ‘.php’ string, and the resulting string becomes ‘view\_simple\_message.php’.
      * You will write a view\_simple\_message.php file later that simply echo’s out the return response message from the model that we mentioned above somehwhere. For example, you can see from the Controller class method dbCreateBitcoinData(), that the $data variable is also passed into the view\_simple\_message view and this $data variable contains the message ‘Data insert successfully iside bitcoin.sqlite file’, which you can see in the Model Class method dbCreateBitcoinData().

1. Create the view\_simple\_message.php file, Figure 20, and put this is in the view folder.

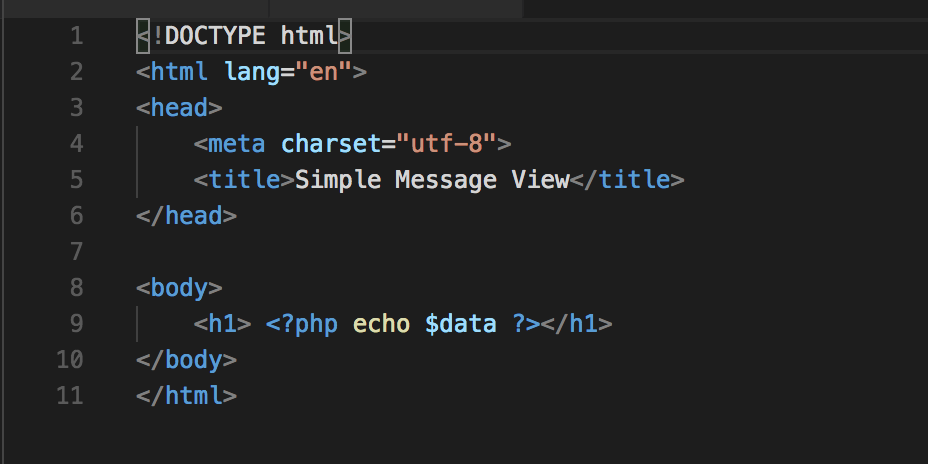


Figure 20: The view\_simple\_message view

* + - I don’t think you need me to explain this HTML file to you! You can see it is a simple HTML file designed to echo out in PHP the message.
    - You should test it by creating the table in the model by calling the Controller method in the URL in a browser, Figure 21.

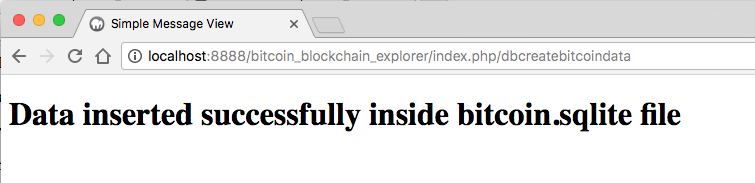


Figure 21: Execute the dbcreatebitcoindata method in the URL

* + - Do recall, we have been a tad lazy here; we really should have created a db connection method, a db table creation method, insert methods, and so on. But, for convenience we wrapped up create table and insert data into the database table into the one function (method).
    - Test your dbreadbitcoindata() method, Figure 22. Note, I happen to have a JSON Viewer installed as an extension in my browser. You can see we are successfully reading the data you inserted into the SQLite database. We could also have added a return message to the response, but echo’ing it out is need for the call back jQuery function later.

****

Figure 22: Execute the dbreadbitcoindata method in the URL

* + - Go ahead and test the dbdelete method, then create the bitcoin\_data table again (because you just deleted it.

Assuming everything works so far, lets move on. And create another view.

So far we have the view\_simple\_message view, Figure 20. We are now going to create a view called bcexplorerdw, which is short for ‘blockchain explorer using an old bit of Dreamweaver HTML code I found lying around and was too lazy to create a new view from scratch’. The good news is that you can copy that view from the source code if you created a GitHub account and downloaded the tutorial code, Figure 23.

* + - So, get hold of this code and stick it in the view folder. I won’t explain in any depth what the HTML5 code for this does.
    - Suffice to say that we have some libraries up the top there, of which the main one is the JavaScript getBitcoinData.js file. This does exactly what is says, uses JavaScript (notably JQuery) to call a controller method to get the bitcoin data we stuffed into the SQLite database. Which controller method would that be?
    - We’ll look at this getBitCoindata.js file a little bit later.
    - The body of the HTML5 file simply has a header with a title and a simple navigation menu, I am just using this to navigate back and forth between a home page, which is currently empty and the Dreamweaver page BITCOIN\_DW (showing the main view for Tutorial 1) and the Bootstrap page BITCOIN\_BS (Showing the man view for Tutorials 2 to 6, Tutorial 1 just gives a link back to the BITCOIUN\_DW page.
    - The body of the HTML5 file simply does some CSS formatting on the data returned from via the getBitcoinData.js file where the data is bound to the associated div tags via id selectors.
    - At this point, take a good look at the downloaded code for these tutorials. The MVC frame needs some other folder, etc. It will need a styles folder to hold the CSS files, just copy the one over from the down loaded code.
    - Similarly, you will need a scripts folder and associated JavaScript files. However, we will look at the getBitCoinData.js and getViewdata.js files below more carefully. You can copy these over though; just make sure you understand how they work.
    - Next, you will need to create the associated Controller Class method to load it, Figure 24.
    - I asked you above what method would that be. Clearly, we can see that we are simply loading the view bcexlorerdw. Remember, the discussion about creating the loader above.

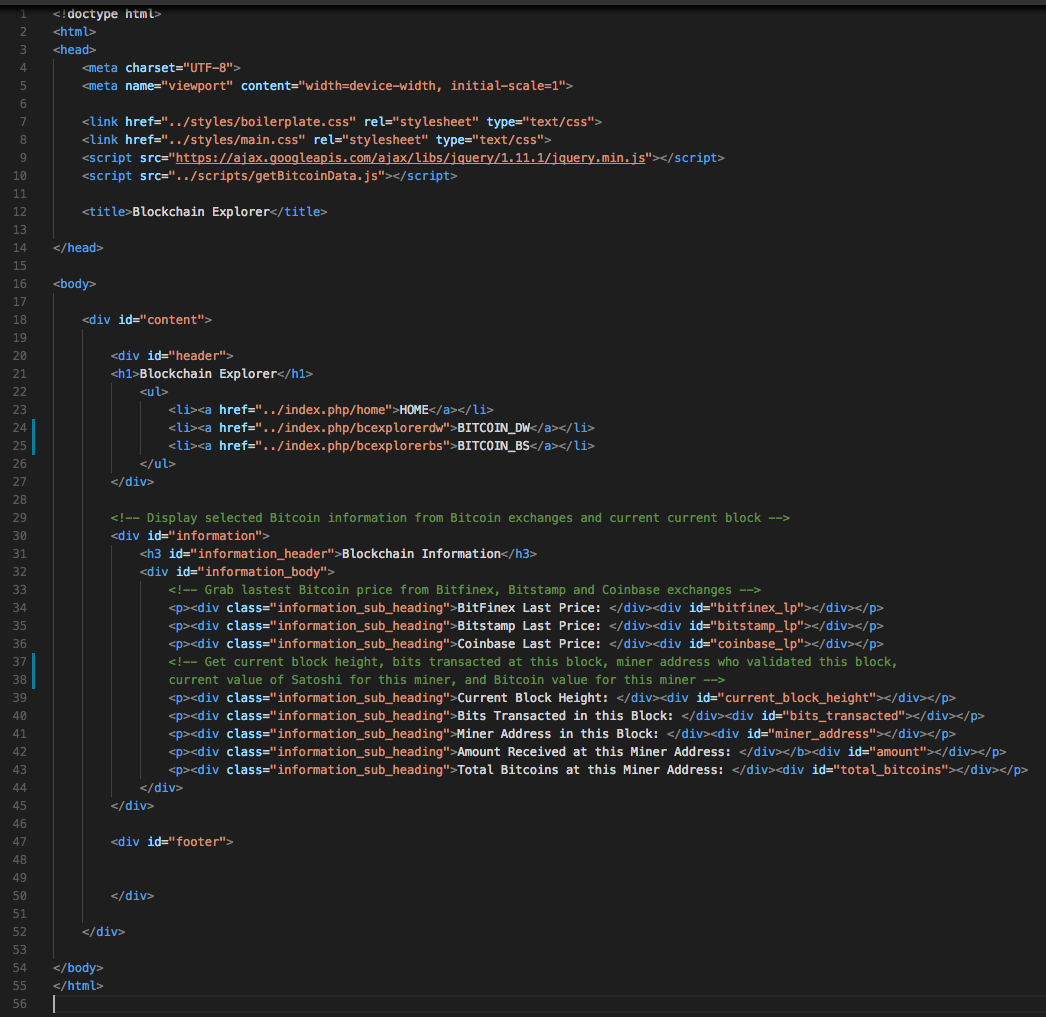


Figure 23: The bcexplorerdw.php view

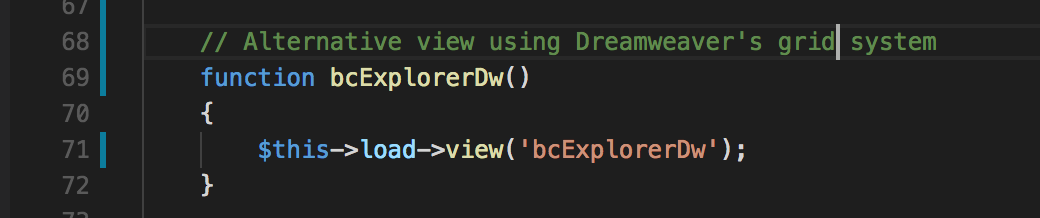


Figure 24: The controller class method for loading the bcexplorerdw view.

### Let’s connect everything together with our JavaScript

Ok, now we need to have a play around with AJAX, JavaScript and JQuery to asynchronously grab data from the back end and dump it in a front-end view, i.e. connect everything together!

We will do this by building a JavaScript handler that requests the controller method via a URL and processes a returned JSON packet. Conveniently, JQuery provides some nice functions for this, i.e. .ajax() and .getJSON().

JQuery is very convenient for building an abstraction around JavaScrript to:

* + Exploit AJAX and JSON to pass the information via a service request or a service response without refreshing or leaving the webpage.
  + Dealing with third party web services such as blockchain.info APIs to include the data in our blockchain explorer.
  + Creating a JSON object and parsing it to get meaningful data
  + Use JSON for data transfer between webpages (or between server and web page, etc)

Let’s just take a step back and consider the technology we will now use to connect the front-end views to the controller.

**What is JSON?**

Have a read through this small description of JSON <http://en.wikipedia.org/wiki/JSON> or check out <https://www.json.org/>. A simple JSON file structure looks something like that shown in Figure 25. We can see that it is simply a format consisting of name:value pairs and array data types; basically a human-readable way of storing or transferring data that can be easily accessed programmatically.

# Macintosh HD:Users:mzp20:Desktop:Screen Shot 2013-03-05 at 13.26.46.png

Figure 25: 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.

**What is AJAX?**

AJAX (Asynchronous JavaScript and XML) is method for receiving and transferring the data to and from a server in the background without refreshing a web page. Although the XML bit is largely replaced by JSON format. An easy way to use AJAX is to use a library such as JQuery.

**What is 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 Study Direct resources Self-learning section for more information.

### Let’s get on with the JavaScript stuff

Ok, now we know what technology we are going to connect the views to the controller methods, lets build the JavaScript file getBitCoinData.js used to call the controller method dbReadBitcoinData(), Figure 26

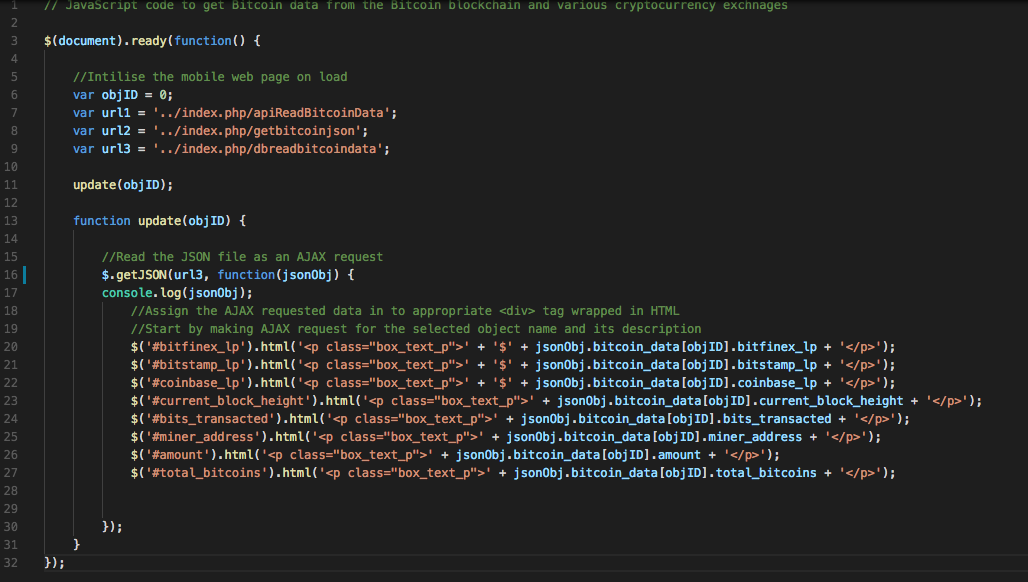


Figure 26: The getBitcoinData.js file

* + Code explanation
    - Line 3: This is a JQuery function $(document).ready(){*handler*} that detects the ready state of the DOM and executes the JavaScript *handler* code.
    - Line 6-9: Some variables are declared, the first allows, objID, us to select the first element of the JSON bitcoin data array in the function update(). Eventually, we may have other elements in the array. Url1, url2 and url3 just set up some URL paths. At the moment url3 is setting up the url path to the dbreadbitcoindata() method in the controller. Later on we will control the url variables to selct other paths such as a controller method to select bitcoin data from an api, e.g. apireadbitcoindata(). Finally, url2 is a path to another controller method that simply echos the bitcoin JSON data out to the browser. You haven’t written the code for the methods invoked by url1 and url2 yet.
    - Line 11: This obviously call the function update(), I am reusing code here so I can’t remember why I called this update. Should probably change its name.
    - Line 13 – 31: is the update() function. This function gets the bitcoin data as a JSON object from the back-end via the controller method.
    - Line 16: Here we use the JQuery .getJSON() method to make an AJAX request to the controller method dbreadbitcoindata() method, and we can see that the bitcoin data is returned in as a JSON object in function(jsonobj).
    - Lines 20-27: Simply uses id selectors to assign the returned bitcoin data, which is wrapped in HTML tags, to the associated div tag in the view. We can see that, for example, class=”box\_text\_p” expects to see some CSS for styling the return bitcoin data elements.
  + I think we have covered everything to test the new view. Check you have the appropriate PHP model methods connected to the controller methods (this should be easy as they have the same name) and that the correct controller method is called to invoke the new view.

1. If you run index.php in your localhost, for example, you should get the result shown in Figure 27.

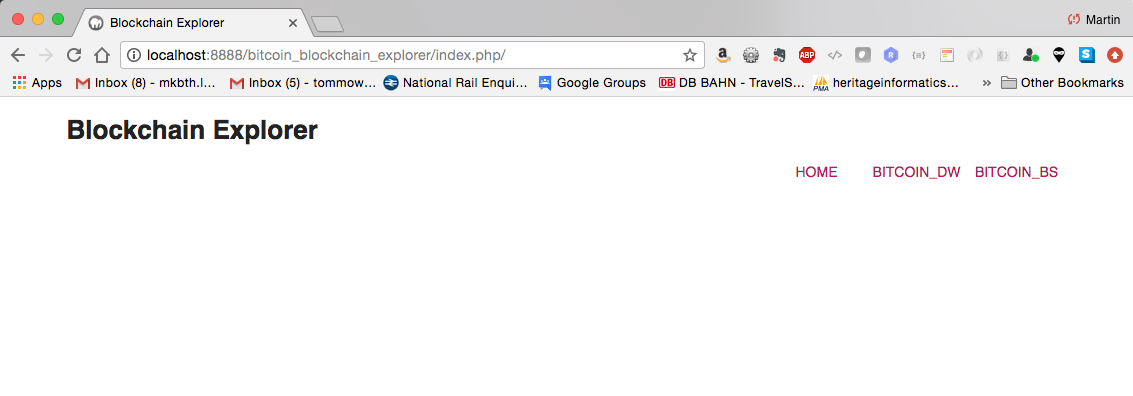


Figure 27: Run the bitcoin\_blockchain\_explorer/index.php/ file

1. Select BITCOIN\_DW on the navigation menu and you should get the result shown in Figure 28.
   * This should the bitcoin data retrieved from the SQLIte database
2. Test the other functionality implemented so far by entering the localhost path to the controller methods that you have completed so far:
   * Localhost:8888/bitcoin\_blockchain\_explorer/index.php/
     + As already mentioned, you should get the result shown in in Figure 27. You can click on the BITCOIN\_BS menu item, or enter
   * Localhost:8888/bitcoin\_blockchain\_explorer/index.php/bcexploererdw
     + You should get the result shown in Figure 28
   * Localhost:8888/bitcoin\_blockchain\_explorer/index.php/dbdelete
     + Will delete the SQLite database, actually drop the bitcoin data table, and you should get a message view back, which reminds me, don’t forget the view\_simple\_message view, Figure 20.
   * Localhost:8888/bitcoin\_blockchain\_exp lorer/index.php/dbcreatebitcoindata
     + Will create the table again, and insert bitcoin test data, Figure 21
   * Localhost:8888/bitcoin\_blockchain\_explorer/index.php/dbreadbiticoindata
     + Will echo the bitcoin database table out to the browser, and if you have a JSON Viwer extension installed you should see something like that shown in Figure 22.

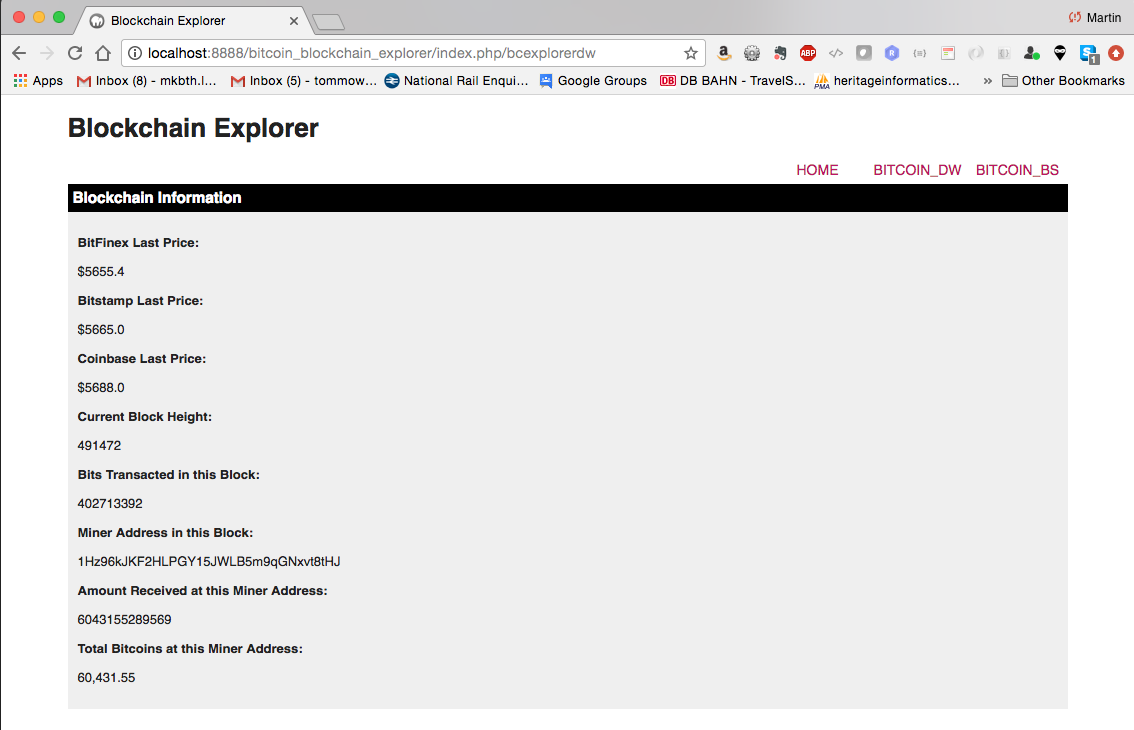


Figure 28: Selecting BITCOIN\_DW

That’s it for Tutorial 1. Next, we will start Tutorial 2, which involves knocking up a simple blockchain in JavaScript just to illustrate key blockchain structures and concepts.

1. http://php.net/manual/en/book.pdo.php [↑](#footnote-ref-1)