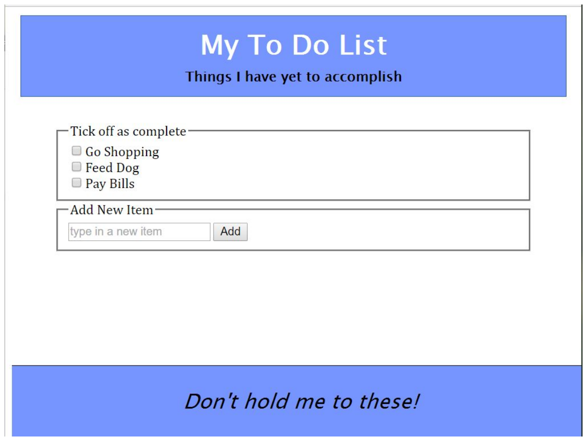
Python Flask Server - To-Do List

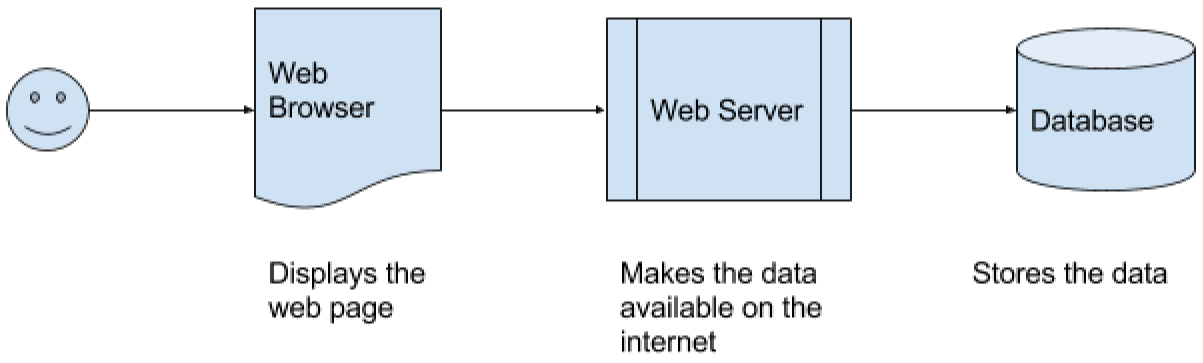
This project assumes that you have completed the JavaScript to-do list.

The JavaScript stage of this project gave us a dynamic web page, we could add and delete items to our to-do list. The user interface looked like this:



In this project we will not be changing the user interface at all, we are going to build a web service for storing our to-do list. This will mean that our to-do list will still exist even when we close the web browser.

This is what we are aiming for.



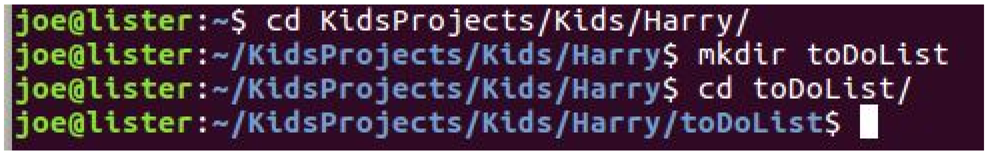
This multi layered structure is how nearly ALL web based systems are built. Facebook, Twitter, RunKeeper, Strava, SoundCloud, Uber, Google Maps. They all provide web services to back up their pages.

We are going to use a Python library called Flask to build a web server. Our first task is to write a 'Hello World' web server in Flask.

Hello World in Flask

This project will require a number of files, so we will create a new directory for our To-Do list application. Open a terminal and navigate to your personal directory within the KidsProjects Git repository.

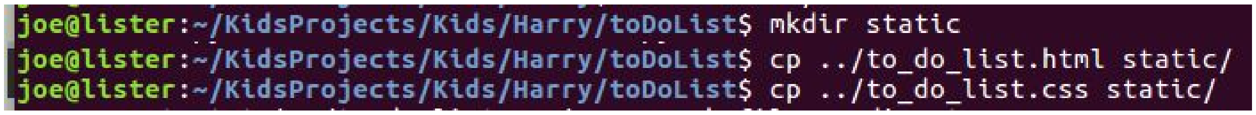
Using Harry as an example:



```
joe@lister:~$ cd KidsProjects/Kids/Harry/
joe@lister:~/KidsProjects/Kids/Harry$ mkdir toDoList
joe@lister:~/KidsProjects/Kids/Harry$ cd toDoList/
joe@lister:~/KidsProjects/Kids/Harry/toDoList$
```

The first line navigates to Harry's directory.
The second line creates a new directory for our application called **toDoList**.
The third line then navigates into the directory.

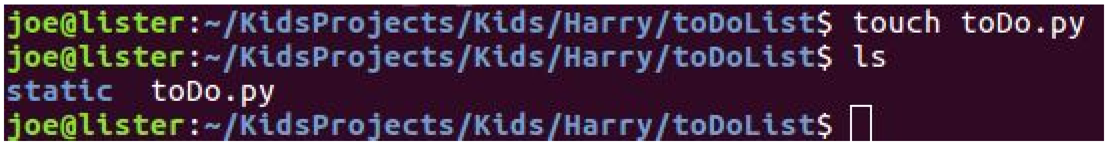
We should now copy in our existing ToDoList HTML and CSS code. Flask applications like to put static resources into a directory called **static**. So create that directory now, then copy your existing HTML and CSS files into it (Ask a teacher for help if you need it).



```
joe@lister:~/KidsProjects/Kids/Harry/toDoList$ mkdir static
joe@lister:~/KidsProjects/Kids/Harry/toDoList$ cp ../to_do_list.html static/
joe@lister:~/KidsProjects/Kids/Harry/toDoList$ cp ../to_do_list.css static/
```

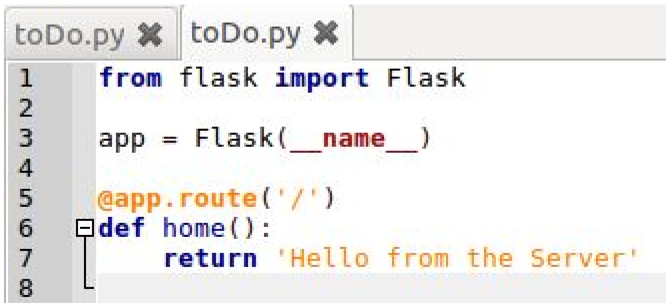
Open the HTML file up in Geany ready for us to edit them. The CSS file should not need changing.

We will now create a new python file to contain our server code. Using your terminal, if you are in the directory for the Flask application, use the following command to create the file.



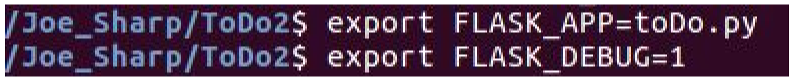
```
joe@lister:~/KidsProjects/Kids/Harry/toDoList$ touch toDo.py
joe@lister:~/KidsProjects/Kids/Harry/toDoList$ ls
static  toDo.py
joe@lister:~/KidsProjects/Kids/Harry/toDoList$
```

Open your python code up in Geany and type the following code.



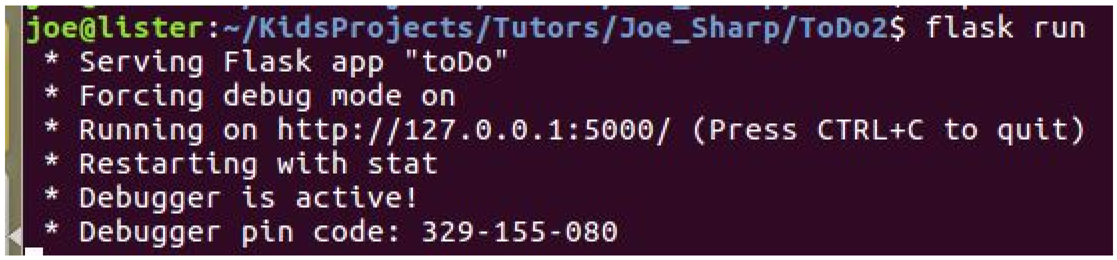
```
toDo.py ✖  toDo.py ✖
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Hello from the Server'

```

We are now ready to run this example. In the terminal (the one that is open at the correct directory) type the following commands, this will point Flask at our application, and also monitor for changes.



```
/Joe_Sharp/ToDo2$ export FLASK_APP=toDo.py
/Joe_Sharp/ToDo2$ export FLASK_DEBUG=1
```

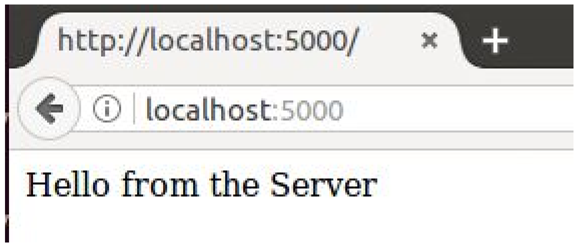
We can then run the flask application as shown. Enter the **flask run** command and press **Enter** and the application should start running!



```
joe@lister:~/KidsProjects/Tutors/Joe_Sharp/ToDo2$ flask run
 * Serving Flask app "toDo"
 * Forcing debug mode on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 329-155-080
```

We will now use a web browser to visit our new server. Open Chromium and go to the following URL.
http://localhost:5000/

It should open up with something like this.



http://localhost:5000/

localhost:5000

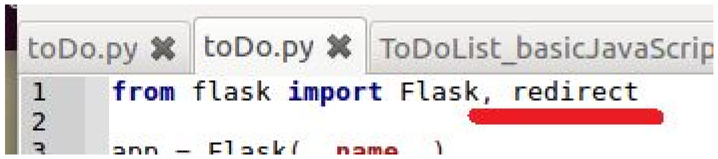
Hello from the Server

We are now ready to start building our to-do list application!

Point our Server at the Static Resources

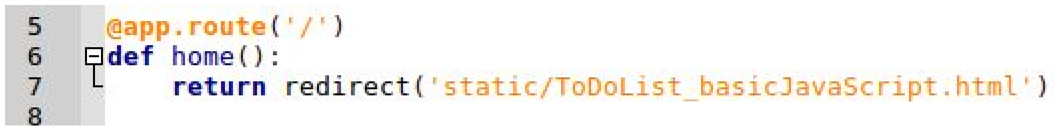
Now that we have a working server, let us expose the static resources we copied in earlier. These static resources our the HTML and CSS files used as the interface for our application.

Add the **redirect** import to the Python code. We will use this function to redirect the browser to our HTML page when the user visits the server.



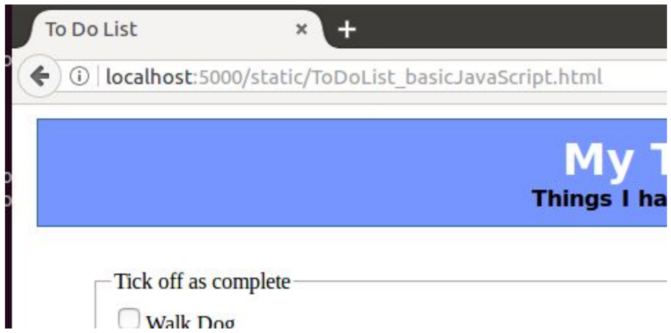
```
from flask import Flask, redirect

app = Flask(__name__)
```

Now we are going to change the **home()** function we wrote earlier. Instead of returning a fixed **'Hello from the Server'** it will redirect the visitor to the HTML page. Note that the filename you use must match the filename of the HTML File you copied in earlier.



```
@app.route('/')
def home():
    return redirect('static/ToDoList_basicJavaScript.html')

```

Now refresh your page and you should be redirected to your HTML page.

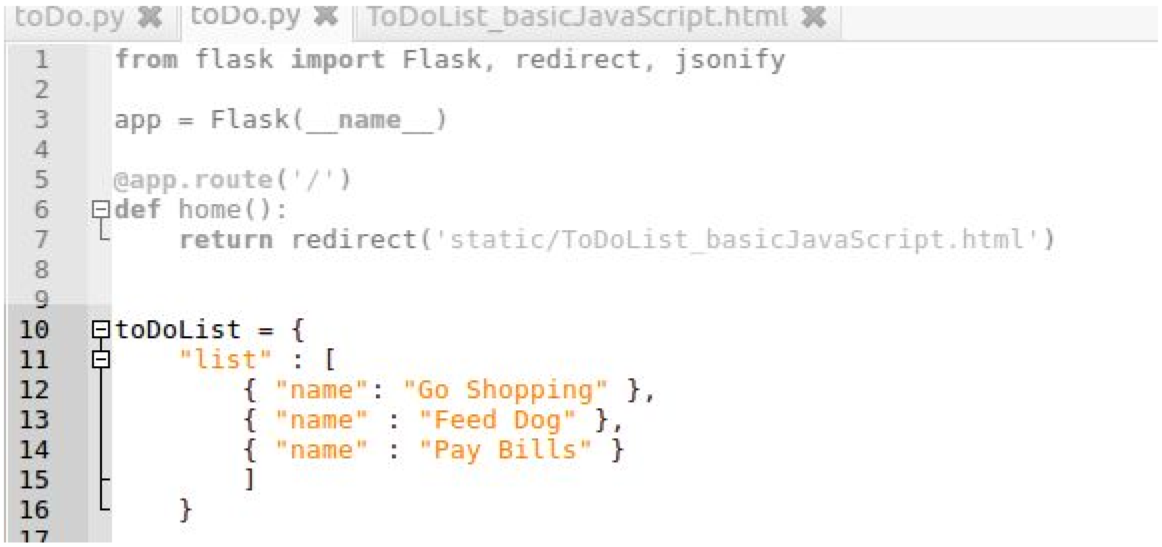


If any of this does not work, check you have the filenames correct, is the styling working?

Once everything is working, you are ready to proceed.

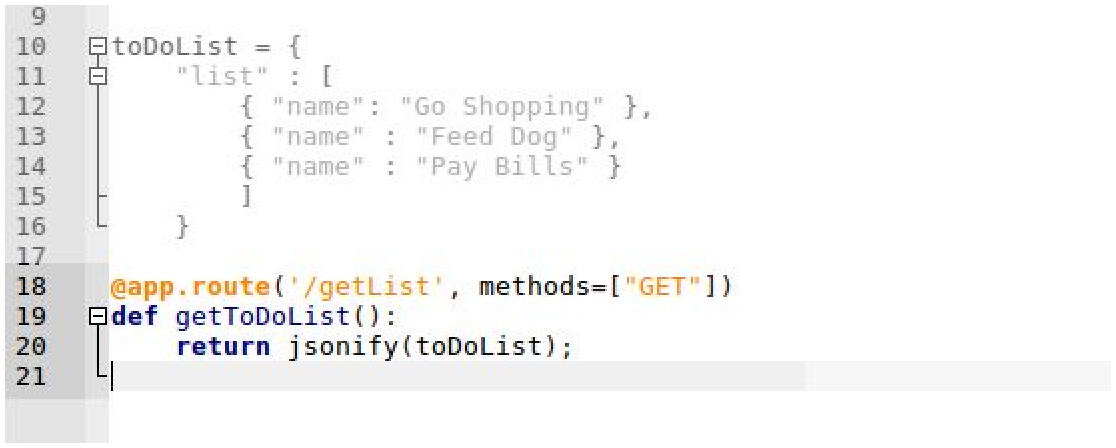
Create a Server Side List

We are now going to create a list of to-do items that will live on our server. We will just use an in-memory list (rather than a database) at this stage. To create our new list, add the following to our Python code:



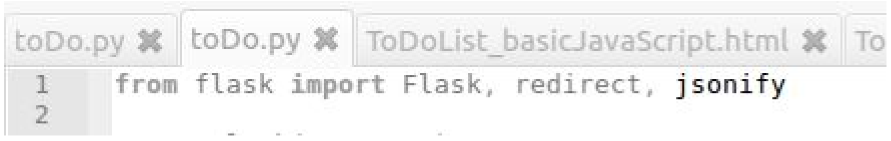
```
from flask import Flask, redirect, jsonify

app = Flask(__name__)

@app.route('/')
def home():
    return redirect('static/ToDoList_basicJavaScript.html')


toDoList = {
    "list" : [
        { "name": "Go Shopping" },
        { "name" : "Feed Dog" },
        { "name" : "Pay Bills" }
        ]
    }
```

This creates an object that contains a property **list**. This **list** is an array of objects. Each object has a property called **name** that contains the text of our to-do item. We now need to expose this as a REST interface (Web Service) so add the following code.



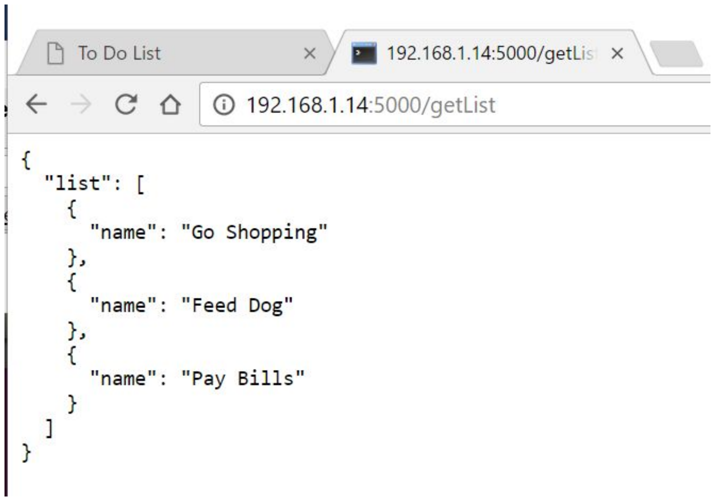
```

toDoList = {
    "list" : [
        { "name": "Go Shopping" },
        { "name" : "Feed Dog" },
        { "name" : "Pay Bills" }
        ]
    }

@app.route('/getList', methods=["GET"])
def getToDoList():
    return jsonify(toDoList);

```

Here we are calling a function called **jsonify** so we must **import** that function at the top. Change the top import line by adding the **jsonify** import as shown.



```
from flask import Flask, redirect, jsonify

```

Save your code. If we are running DEBUG correctly, then the Flask framework should detect the change and redeploy the application. Visit the URL http://localhost:5000/getList and it should look like the following:



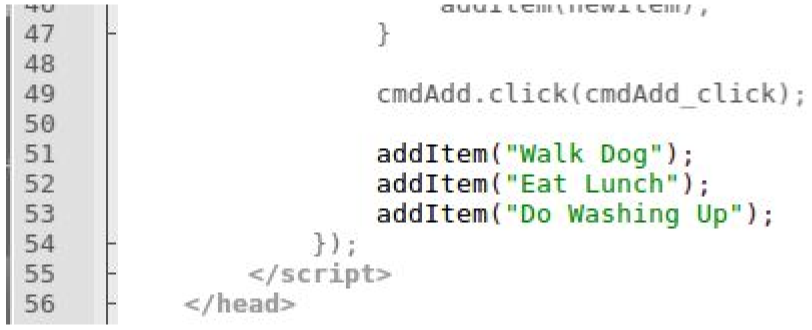
```
{
  "list": [
    {
      "name": "Go Shopping"
    },
    {
      "name": "Feed Dog"
    },
    {
      "name": "Pay Bills"
    }
  ]
}
```

If you have gotten this far, then congratulations; you have written your first web service.

Next we need to use this web service in our HTML page.

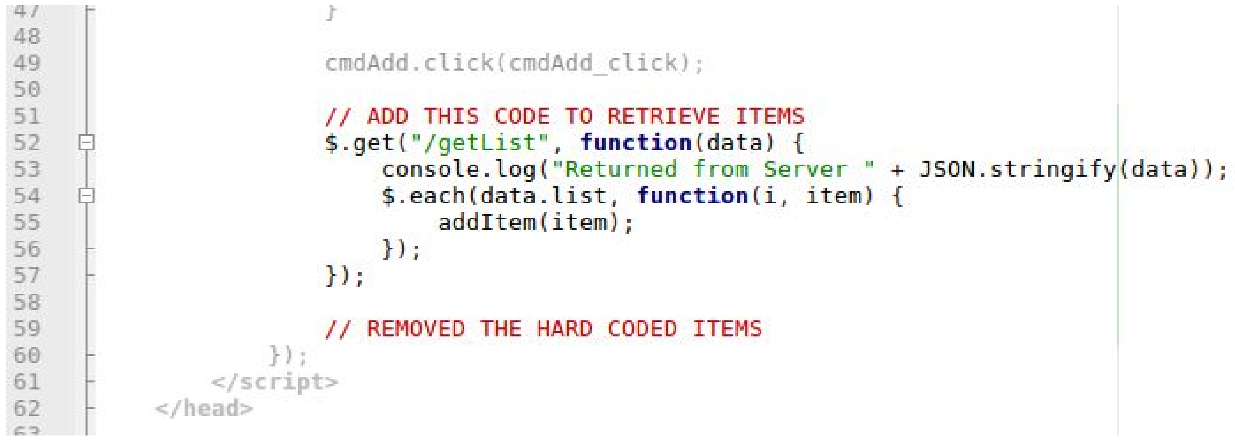
Change our HTML Page to use the server

If you followed the last project exactly then your HTML/JavaScript code should contain a set of lines that add some hard coded items into the list.



```
                }

                cmdAdd.click(cmdAdd_click);

                addItem("Walk Dog");
                addItem("Eat Lunch");
                addItem("Do Washing Up");
            });
        </script>
    </head>
```

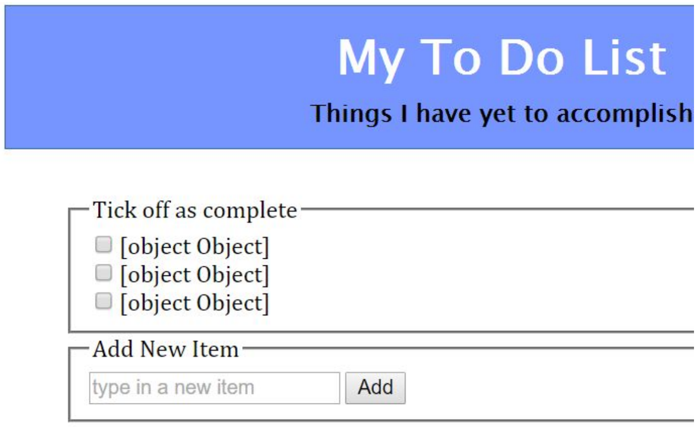
We shall delete those and replace it with the following (you don't need to type the comments).



```
                }

                cmdAdd.click(cmdAdd_click);

                // ADD THIS CODE TO RETRIEVE ITEMS
                $.get("/getList", function(data) {
                    console.log("Returned from Server " + JSON.stringify(data));
                    $.each(data.list, function(i, item) {
                        addItem(item);
                    });
                });

                // REMOVED THE HARD CODED ITEMS
            });
        </script>
    </head>
```

What is this code doing? It is using something called AJAX (Asynchronous JavaScript and Xml) to retrieve data from our web service using HTTP GET.

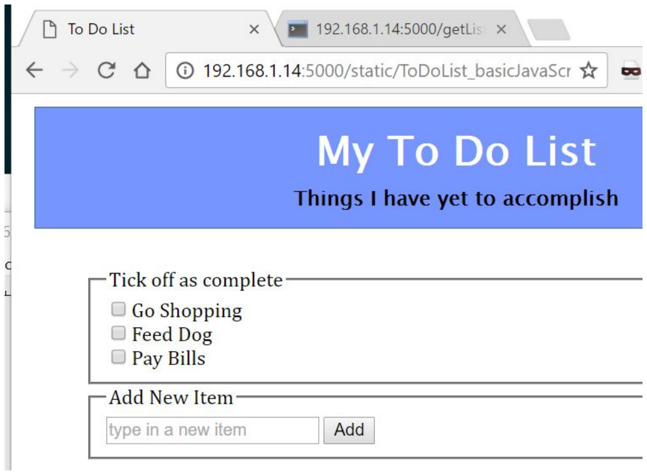
When the web service returns with data, the function we pass in gets called and the data is given to use for use. Here I am calling the **addItem** function that we wrote earlier. Now reload your page, and it will probably not look quite right any more. It should look like this:



So what is with those [object Object] lines? We have changed our to-do items from simple strings to full objects. We now need to update the **addItem** function to handle this correctly. Find where we are building the label, we need to add a reference to the **name** property on the **item** we are given. As shown here:

```
// Build a label
var newLabel = $("<label>")
    .text(item.name)
    .addClass("strikethrough");
```

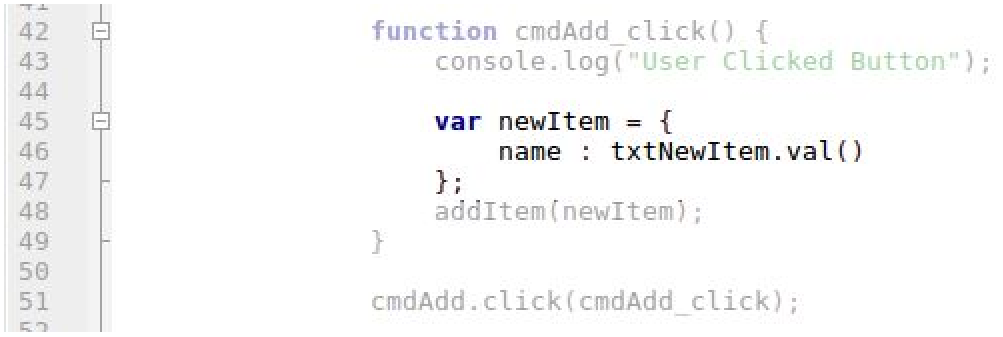
Add this **.name** and then save your code. Re-open the page and should look like this:



Now we are talking. Try adding some items to the Python code to check that the list is truly being based from the Python server.

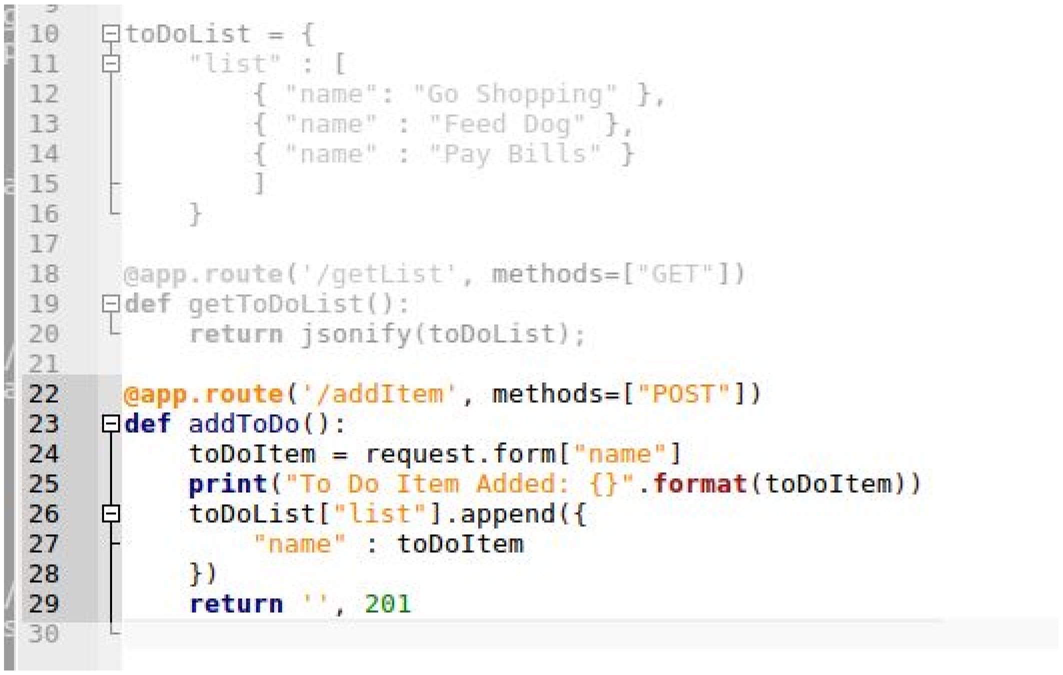
Add Items to List on Server

You may have noticed that adding items is now broken. What happened? We changed the **addItem** function to expect **Objects** rather than **Strings**. We now need to update the **cmdAdd_click** function to create an object as shown:



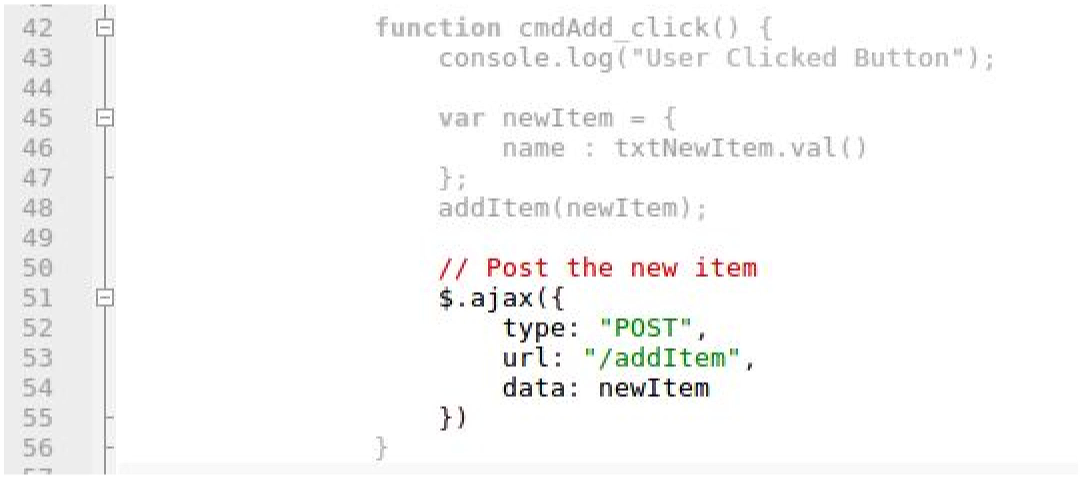
```
        function cmdAdd_click() {
            console.log("User Clicked Button");

            var newItem = {
                name : txtNewItem.val()
            };
            addItem(newItem);
        }

        cmdAdd.click(cmdAdd_click);
```

It would be nice if new items were added to the server. So let us head back to the Python (app.py) and add a new REST interface that uses HTTP POST. It will look like this:



```
toDoList = {
    "list" : [
        { "name": "Go Shopping" },
        { "name" : "Feed Dog" },
        { "name" : "Pay Bills" }
        ]
    }

@app.route('/getList', methods=["GET"])
def getToDoList():
    return jsonify(toDoList);

@app.route('/addItem', methods=["POST"])
def addToDo():
    toDoItem = request.form["name"]
    print("To Do Item Added: {}".format(toDoItem))
    toDoList["list"].append({
        "name" : toDoItem
    })
    return '', 201
```

Now that we have a new POST function, we should head back to the JavaScript and call it when a new item is added. Add the following code to the **cmdAdd_click()** event handler.



```
        function cmdAdd_click() {
            console.log("User Clicked Button");

            var newItem = {
                name : txtNewItem.val()
            };
            addItem(newItem);

            // Post the new item
            $.ajax({
                type: "POST",
                url: "/addItem",
                data: newItem
            })
        }
```

Now reload your page and try adding items.

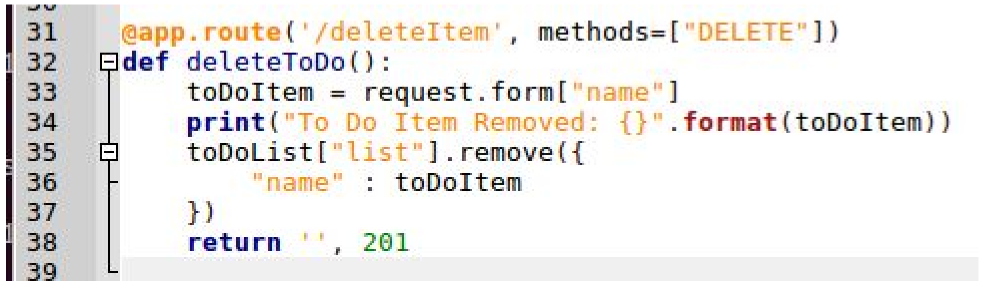
Here is the magic part. When we used a purely JavaScript to-do list, the list was reset whenever we refreshed the page. The data did not persist. Now that we have a server, this should be different. Try reloading the web page and you should find that any changes to the list have been remembered!

A note of caution, if you make changes to the Python and save it, then the list will be reset because the Python code will be reloaded. Still...once you leave the python code alone it will save the list as long as the server stays up!

Delete Items from Server

Last thing to implement, it would be nice if items that were ticked off were deleted. At the moment the app does not remember that items were ticked off. Instead of remembering their state and keeping the ‘completed items’ in our list, we shall simply delete them.

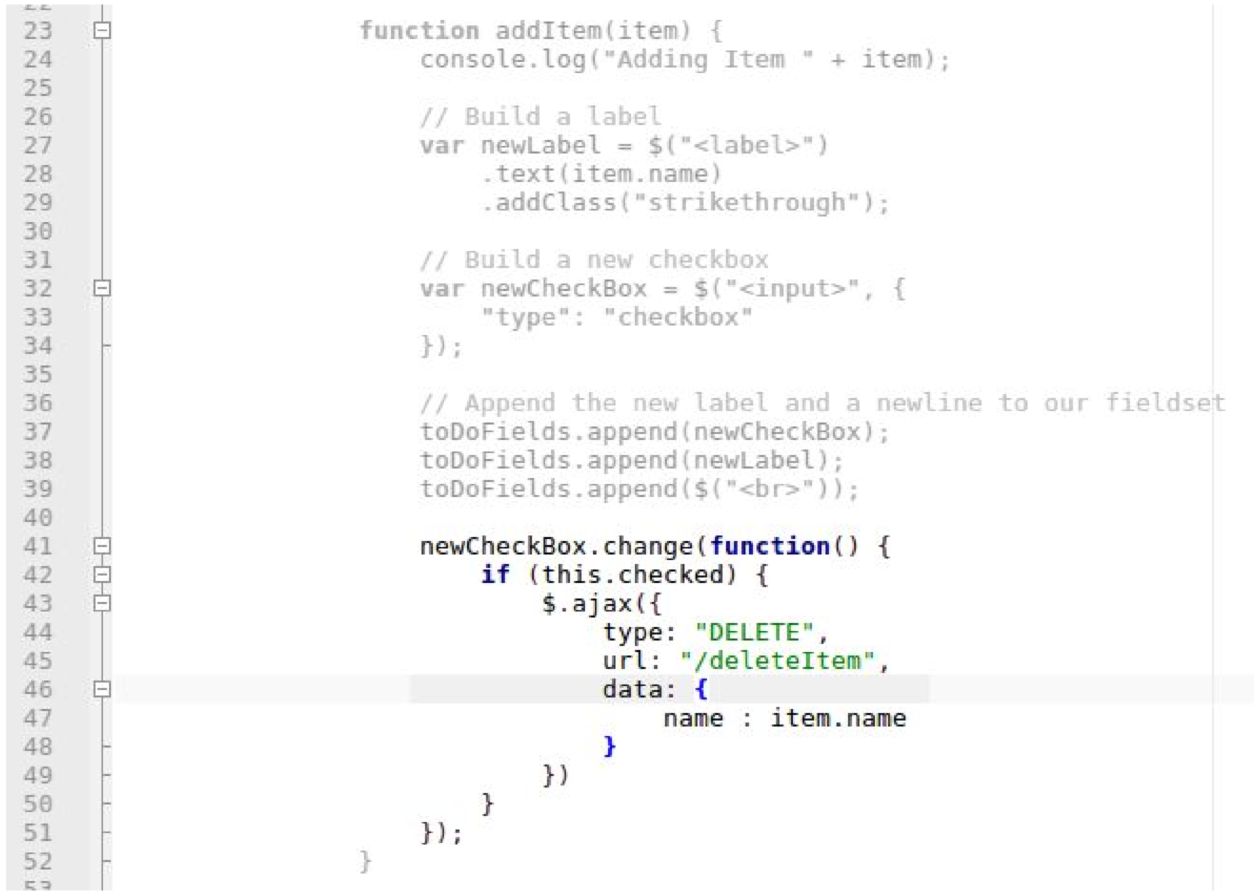
First we shall add a HTTP DELETE endpoint to our Python server code. Add the following code.



```
@app.route('/deleteItem', methods=["DELETE"])
def deleteToDo():
    toDoItem = request.form["name"]
    print("To Do Item Removed: {}".format(toDoItem))
    toDoList["list"].remove({
        "name" : toDoItem
    })
    return '', 201
```

We need to call this REST function from our JavaScript when the user clicks on the checkbox.

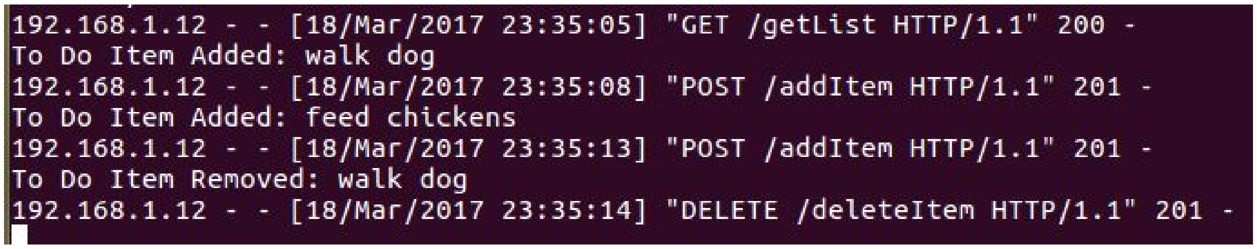
Checkboxes are added dynamically in the **addItem** function of our JavaScript. Add the following code to the end of that function.



```
function addItem(item) {
    console.log("Adding Item " + item);

    // Build a label
    var newLabel = $("<label>")
        .text(item.name)
        .addClass("strikethrough");

    // Build a new checkbox
    var newCheckBox = $("<input>", {
        "type": "checkbox"
    });

    // Append the new label and a newline to our fieldset
    toDoFields.append(newCheckBox);
    toDoFields.append(newLabel);
    toDoFields.append($("<br>"));

    newCheckBox.change(function() {
        if (this.checked) {
            $.ajax({
                type: "DELETE",
                url: "/deleteItem",
                data: {
                    name : item.name
                }
            })
        }
    });
}
```

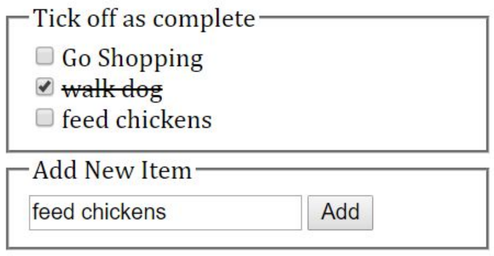
Final Testing

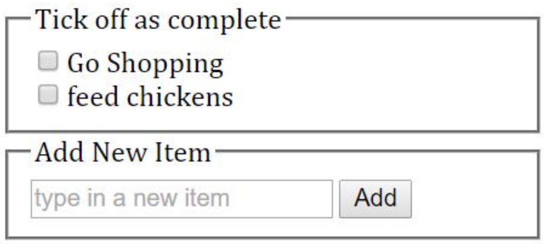
When you are running commands on the server, we have been printing things using the **print** function. The server log should be visible in the terminal you are running the server in. It will look something like this:



```
192.168.1.12 - - [18/Mar/2017 23:35:05] "GET /getList HTTP/1.1" 200 -
To Do Item Added: walk dog
192.168.1.12 - - [18/Mar/2017 23:35:08] "POST /addItem HTTP/1.1" 201 -
To Do Item Added: feed chickens
192.168.1.12 - - [18/Mar/2017 23:35:13] "POST /addItem HTTP/1.1" 201 -
To Do Item Removed: walk dog
192.168.1.12 - - [18/Mar/2017 23:35:14] "DELETE /deleteItem HTTP/1.1" 201 -
```

Play around with your application, reloading the page to check that ticked off items are deleted and new items are remembered. The following two screenshots show how a delete should be handled between page refreshes.





CONGRATULATIONS YOU HAVE COMPLETED THE SERVER BASED TO DO LIST