**AJAX**

**Useful URLs**

Any URL on the Internet can be used to make our own websites better. Remember that URLs are just the human-friendly way of identifying the location of the computer that holds the files that we are requesting.

**Pages**

Some URLs lead you to a page. For example, if you navigate to  [http://www.codingdojo.com](http://www.codingdojo.com/) your browser will render the landing page of Coding Dojo. You navigated to a URL and that URL rendered HTML/CSS and JavaScript as its response. Let's fetch this response in a different way. Go ahead and open up your terminal and type in the following commands:

curl http://www.codingdojo.com

This is the response that your browser gets when navigating to the website. The browser interprets the HTML/CSS and JavaScript to display a complete page.

**Images**

Some URLs only hold an image. For example, if you navigate to  <http://pokeapi.co/media/img/1.png> your browser will render a picture of Bulbasaur. Let's try fetching this response using curl:

curl http://pokeapi.co/media/img/1.png

Image files sure look funky! Now that we know that link contains a picture of a Bulbasaur, we can use that URL to make our website better. We can include the image of Bulbasaur on our website by including an *img* tag to our website:

<img src="http://pokeapi.co/media/img/1.png">

**Videos**

Some URLs only hold a video. For example, if you navigate to  <http://a1068.v.phobos.apple.com/us/r1000/019/Video4/v4/e5/e7/a9/e5e7a98f-ae35-7399-867b-aad0915cc445/mzvf_8081360670963169027.640x352.h264lc.U.p.m4v> your browser will render a music video of Kanye West. Let's try fetching this response using curl:

curl http://a1068.v.phobos.apple.com/us/r1000/019/Video4/v4/e5/e7/a9/e5e7a98f-ae35-7399-867b-aad0915cc445/mzvf\_8081360670963169027.640x352.h264lc.U.p.m4v

Hopefully the video file didn't scare you. We can include this video on our website by including a video tag to our website:

<video src="http://a1068.v.phobos.apple.com/us/r1000/019/Video4/v4/e5/e7/a9/e5e7a98f-ae35-7399-867b-aad0915cc445/mzvf\_8081360670963169027.640x352.h264lc.U.p.m4v" controls></video>

**CDN**

Some URLs hold useful code. CDNs (content distribution network) is a fancy way of saying that. For example, if you navigate to  <https://code.jquery.com/jquery-2.1.3.js> you will see the code for jquery. Let's try fetching this response using curl:

curl https://code.jquery.com/jquery-2.1.3.js

We can include this code on our website by using a script tag:

<script src="https://code.jquery.com/jquery-2.1.3.js"></script>

**Request Response**

How many requests do you think this web page will send? Load the code into a browser, open up the developer console, click on the network tab, and refresh the page.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>HTPP Request Response</title>

<script src="https://code.jquery.com/jquery-2.1.3.js"></script>

</head>

<body>

<img src="http://pokeapi.co/media/img/1.png" alt="bulbasaur">

<video

src="http://a1068.v.phobos.apple.com/us/r1000/019/Video4/v4/e5/e7/a9/e5e7a98f-ae35-7399-867b-aad0915cc445/mzvf\_8081360670963169027.640x352.h264lc.U.p.m4v"

controls></video>

</body>

</html>

To load this page, the browser had to make 4 GET requests. First to GET the page. Then as it was reading the page it saw that we are including jQuery's CDN. So the browser makes another GET request to get the code located in the URL specified in the src attribute and includes it on the page. Then the browser has to send another GET request when it realizes the *img* tag's source is located in the PokeAPI URL. Finally, the browser makes another GET request to load the video located in the URL provided in the video element's *src* attribute.

# API (Application Programming Interface)

APIs define a set of rules in which you can interact with a particular software. For web developers, URLs, files, or code that are specifically designed to be used by other developers are called APIs. URLs are just particular directions for your request to make to find a particular file somewhere out there on the Internet. We can interact with URLs by passing it GET variables or POST variables.

For example, let's say I have a huge database of Pokemon and I want to allow other developers to access this database to create cool Pokemon applications like the Pokedex. I can create a series of URLs that map to important information in a predictable manner. The developers will need to read the documentation of your API to learn how to properly interact with your URLs and get predictable results. This is exactly what PokeAPI did. Their documentation can be found here:  <http://pokeapi.co/>.

## Every API is Different

Every API you interact with will be different. It is up to the developer who wrote the program to specify the protocols needed to interact with the API. Some APIs will let you access information as long as you ask for it in the URL. Some APIs will give you better information if you submit more data through a form. Some APIs will only allow you to request a certain URL a limited number of times per day. Some APIs will require you to sign up on their website to be provided an authenticity token to pass with every API request you make.

APIs are usually web URLs that accept information from you.  You send that information either by appending it to the URL, or by posting a form with input fields.  The server at that web URL receives the request and returns an answer in JSON (or other similar data format).  You will create your own API in later chapters but in essence that is all you need to have to build an API: if a URL takes some data and outputs JSON (or other similar formats), you have created an API.  Take a quick look at some of the APIs listed below.

1. Open Weather Map: <http://openweathermap.org/api>
2. GitHub API: <https://developer.github.com/v3/>
3. Google Maps Directions API: <https://developers.google.com/maps/documentation/directions/>
4. Twitter API: <https://dev.twitter.com/rest/public>
5. Flickr API: <https://www.flickr.com/services/api/>

Although you have to interact with each API differently, notice how they are all collections of URLs that you can interact with as long as you follow the protocols that the developer has specified in the documentation.

## What's the Big Deal?

* **Speed (Developers):** Instead of creating everything from scratch, you can utilize APIs of other web services to build some cool apps more easily. There are lots of useful APIs out there.
* **Cross development (Managers/Developers):**APIs allow experts/developers of different languages (PHP, Ruby, Python, etc.) to work together efficiently. For example, let's say you had a team of Java developers who built an app in Java and you had another team of PHP developers who built an app in PHP. Say you had another team of JavaScript developers who built their app in Node. What could you do if you wanted these three apps to work together (send data back and forth)? You could either have everyone migrate to just one language (not recommended) or just have three teams build APIs (URLs where you can submit/retrieve information) so that these three apps can work together to communicate information back and forth!
* **Wider reach of audience (Marketers):**APIs allows marketers to market their product/services to a wider audience of developers. Then, these developers can build applications that use the API to reach an even wider audience of consumers and developers. For companies like Twitter, Facebook, Flickr, and Google, they wanted to reach out to a wider audience so they made it easy for developers to use their services to build their brand even more.

**What's Happening When We Make an AJAX Request to PokeAPI?**

**Traversing JSON**

Some URLs have JSON (JavaScript Object Notation) that we can use. JSON is just one of many ways of representing data. Navigate to  <http://pokeapi.co/api/v1/pokemon/1/> to see JSON data about Bulbasaur. This means that instead of having our own database about Bulbasaur, we can just refer to this link to get more data about this Pokemon. How can we use this JSON data to display Bulbasaur's types (grass and poison), its height, and weight onto our website? Thanks to jQuery, we have a convenient function $.get to do AJAX (which is a way of getting information from another URL without having to reload your own page). If we were to do it using pure JavaScript (which is how the *$.get* function in jQuery is implemented), we would have to write a lot of code to make AJAX work for all different types and versions of browsers. Thanks, jQuery!

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Gotta Catch'em All</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

$.get("http://pokeapi.co/api/v1/pokemon/1/", function(res) {

console.log(res);

}, "json");

})

</script>

</head>

<body>

<div id="bulbasaur">

</div>

</body>

</html>

We are using JQuery to load the JSON data that lives in <http://pokeapi.co/api/v1/pokemon/1/> and logging it to the console. We are passing in three arguments to the *$.get* function.

1. The URL that we want to load information from (i.e. "[*http://pokeapi.co/api/v1/pokemon/1/*](http://pokeapi.co/api/v1/pokemon/1/)")
2. Function you want to run after information was successfully loaded (i.e.*function(res) { console.log(res) }*)
   * *$.get* will automatically pass an argument to this function that holds the information from the URL we requested
   * We can access this argument by having our function take an argument (i.e. *res*)
   * You can name this argument anything you want, you can name it *'data'*for example
3. The type of information we are expecting back which in our case is *json*.

You can read more about $.get [here](http://api.jquery.com/jquery.get/).

**Traversing JSON**

Go ahead and play with the JavaScript Object that is logged onto your console. Chrome has done a great job allowing us to traverse through different parts of the Object by clicking inside. Let's say we want to display Bulbasaur's types (poison, grass), onto our console. How can we do this?

*Getting Bulbasaur's Types*

Once we click on the Object that is logged in our console, we will see a lot of attributes. One of those attributes is named '*types.'* Chrome has provided us with a nice GUI arrow that specifies that there's more inside. Go ahead and click the arrow and notice that the*'types*' attribute is storing an array of two objects. If we open up each of these objects, we see that they both have a 'name' attribute. This looks like the data that we want. How can we traverse the JSON to get here?

$.get("http://pokeapi.co/api/v1/pokemon/1/", function(res) {

console.log(res.types[0].name);

console.log(res.types[1].name);

}, "json");

Let's look at the first *console.log* above.  The object we received back from the pokeapi has been put into a parameter named *'res'*.  Within this object, we want to examine an attribute named *'types'*.  Since the '*types'* attribute holds an array of two objects, we index to [0] to grab the first object.  Within *that* object, we access that object's *'name'* attribute to have 'poison' logged to our console.  In the second console.log, we are doing the same thing except we grab the second object from the *res.types array*.  Since we know that *res.types* is an array, we can rewrite our code using a *FOR* loop.  What if the pokeapi someday updates Bulbasaur and adds another type? Using a *FOR loop* makes our application more flexible.

$.get("http://pokeapi.co/api/v1/pokemon/1/", function(res) {

for(var i = 0; i < res.types.length; i++) {

console.log(res.types[i].name);

}

}, "json");

**Building HTML**

What if we wanted to display the JSON data onto our website instead of just logging it to our console? We can include the data from JSON object into our HTML so that the users of our website can see that Bulbasaur's types are*poison* and *grass*.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Gotta Catch'em All</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

$.get("http://pokeapi.co/api/v1/pokemon/1/", function(res) {

var html\_str = "";

html\_str += "<h4>Types</h4>";

html\_str += "<ul>";

for(var i = 0; i < res.types.length; i++) {

html\_str += "<li>" + res.types[i].name + "</li>";

}

html\_str += "</ul>";

$("#bulbasaur").html(html\_str);

}, "json");

})

</script>

</head>

<body>

<div id="bulbasaur">

</div>

</body>

</html>

We are passing in three arguments to the $.get function.

1. The URL that we want to load information from (i.e. "[*http://pokeapi.co/api/v1/pokemon/1*](http://pokeapi.co/api/v1/pokemon/1)/")
2. Function we want to run after information was successfully loaded
   * We want to build a string that looks like "*<h4>Types</h4><ul><li>poison</li><li>grass</li></ul>*" and inserting its content inside the div with the id of Bulbasaur. The browser will read the string and render it as HTML when we call *$("#bulbasaur).html(html\_str)*
3. The type of information we are expecting back which in our case is*json*.

## Assignment: Pokemons

Create a website that has all the pictures of the original Pokemon (pokemon #1 ~ pokemon #151). Use the PokeAPI to get the images. You can find the image of the first pokemon on  <http://pokeapi.co/media/img/1.png>, second pokemon on <http://pokeapi.co/media/img/2.png>, third pokemon on <http://pokeapi.co/media/img/3.png>, and so on.

You can list out 151 images or you can use a for loop (use a for loop).

START WORKING ON THIS

## Assignment: Pokedex

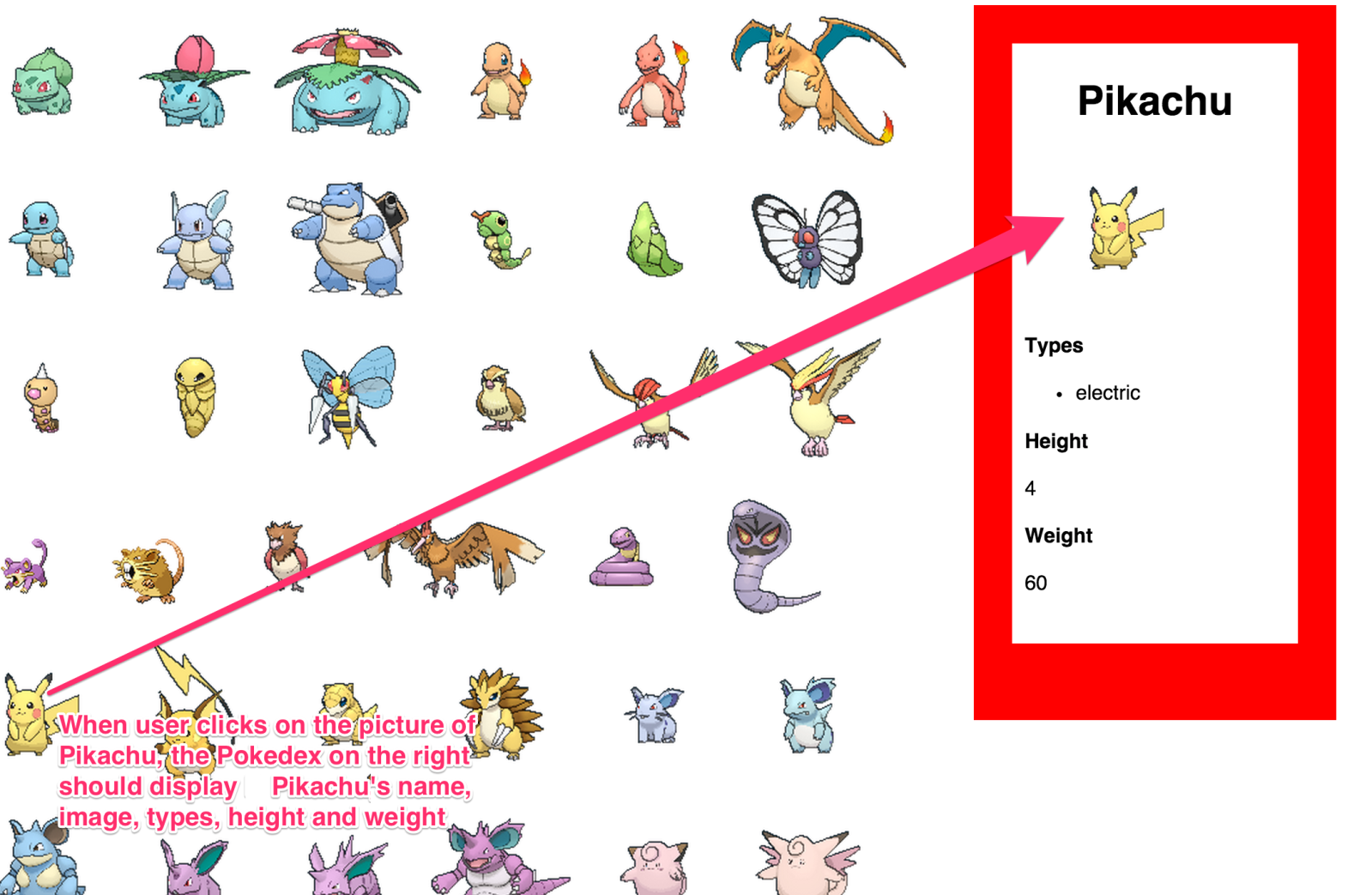
#### **What's that Pokemon? Build Your Own Pokedex.**

## Default Page



## Clicking on one of the image

## alt text



How do you know which Pokemon was clicked? Maybe we can give each Pokemon a unique id that corresponds with their number in the URL?

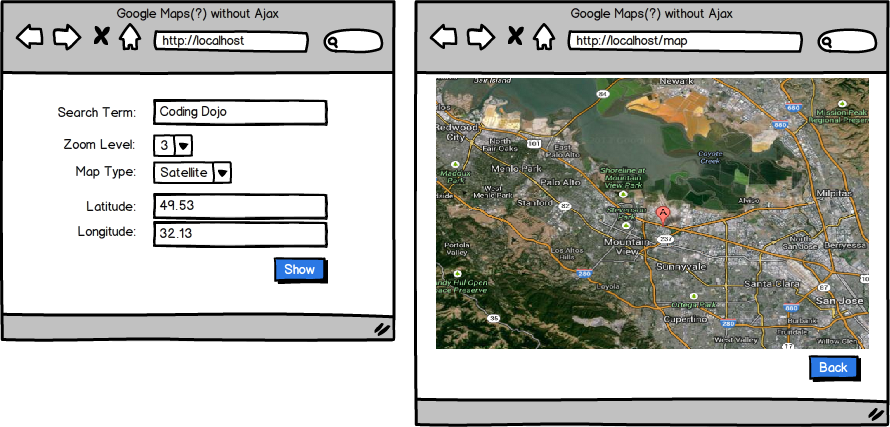
<img id="2" src="">

So that when the image is clicked, we get its id and add it to the end of this URL: "[*http://pokeapi.co/api/v1/pokemon/*](http://pokeapi.co/api/v1/pokemon/)" then we make the AJAX request with that URL that we constructed.

# Traditional Web Applications

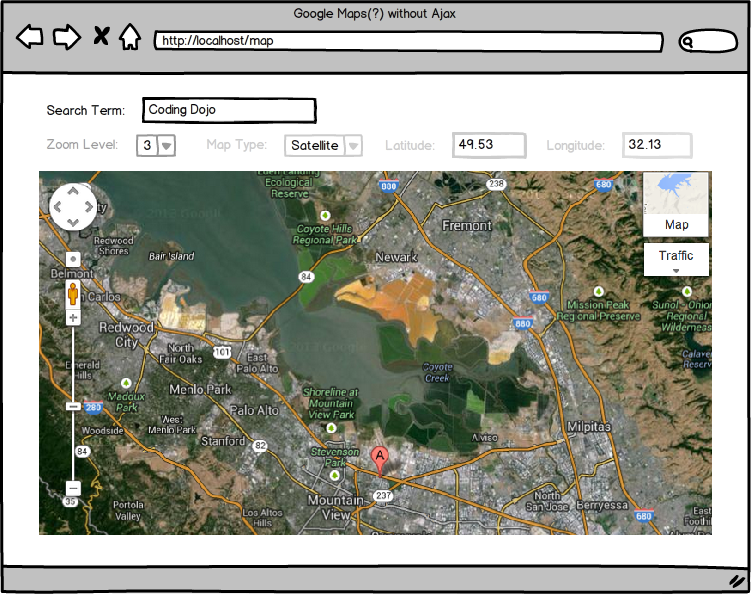
With traditional web apps (web apps built prior to Ajax or apps built in the early 2000s), once the browser sent an HTTP request and rendered the HTTP response returned from the web server, you did not expect what you see in the browser to change. It was impossible back then to imagine (let alone create) an app where the contents would refresh as you change some of the values on the web app. When Google first used Ajax to give "keyword suggestions" right as you were typing the search term, it shocked the whole industry! In other words, traditional web apps were designed to serve static contents and for the developers to use a form to pass data from one page to the other. It was never really designed to build web apps that dynamically change contents without the browser going to a whole new page. If you look back on some of the recent web 2.0 sites you've used, you'll know that new web apps don't function like the old apps. It doesn't have so many forms for you to fill out, but instead, as you click and do things within the page, it seems to automatically update part of the web page, without having to go to a whole new URL. Ajax has now become an integral part of building modern web apps.

If you had to build something like Google Maps without Ajax, you would have done something like below.



## After AJAX

With Ajax, our Google Map could work like something below.



Now, notice that the zoom level, map\_type, latitude and longitude (these are coordinates of where the center of the map is) are still there in the form but are grayed out. This is because the information is still within the form, but just hidden! In other words, instead of <input type="text" name="zoom\_level" />, you would now have <input type="hidden" name="zoom\_level" />. As the user clicks on different items on the HTML page (e.g. the + or the -) or as the user changes the search term or the map type, you would change the value of the form input, send the form information to the server, retrieve some information from the server (maybe a new map?) and then update part of the original HTML page. Also, notice that with Ajax, we don't even need a submit button as we can tell the browser to submit this form automatically behind the scene when certain events occur (when some value gets changed, etc).

So let's review what's happening with the new Google Maps example (with Ajax):

1. When the user clicks on +/-, it updates the value of <input type="hidden" name="zoom\_level" />.
2. When the user changes the map\_type, it updates the value of <input type="hidden" name="map\_type" />.
3. When the user clicks on the mouse and moves left/right/top/bottom, it updates the value of both longitude/latitude (as these are coordinates for where the center of the map should be).
4. Whenever any of these changes occur, the client sends this form information to the server (e.g. localhost/process or whatever URL you choose) and tells the browser not to actually go to that page nor reload the page.
5. The client waits for the server to send some information back, and once this data is received (say the new map info), it displays the new map somewhere in the HTML (using JavaScript, jQuery, or any other JavaScript library that allows HTML manipulations).

With this new approach of submitting the form information WITHOUT reloading the page, we can create cooler apps! Let's look at more examples to see how Ajax has been changing how web apps are built.

## AJAX In Facebook

Back in the days when Ajax was not yet available, every single HTML form (once submitted) would go a new page, specifically wherever the form's action attribute specified.  For example, consider the following form:

<form action="/post/message" method="post">

<textarea name="message"><span class="support tag"></textarea>

<input type="submit" value="Post a message" />

</form>

Whenever someone clicks on the submit button, the browser would go to "/post/message". If you wanted to come back to the original page, you needed to press the back button or type the previous URL manually in the browser.

Imagine how posting a message in Facebook could have been without Ajax. Imagine that you wanted to post a comment in Facebook in a page that looked like below:

For example, you went to the textarea where you could post a message/comment and pressed enter. If this resulted in loading a new page every single time you posted a comment on any picture/post/event, it would drive you crazy! (Imagine you were posting a comment in one of the timelines way back and once you posted the comment, it reloaded the page where you had to scroll all the way down to find the things you were reading about.)  Back before the days of Ajax, this was the norm; there was no way of having your browser send another HTTP request behind the scene and just update part of the HTML on the page. With Ajax, though, now what's happening is something like below:

1. You fill out a message/comment.
2. You press enter.
3. It submits the form but does NOT have the browser redirect to that page.
4. The server that received the form information does whatever it wants to do (e.g. save the information in the database, return some type of message indicating whether it was successfully posted, etc), and sends some type of response back to the client/browser.
5. The client waits for this information; once it is received, the client updates part of the HTML with whatever it wants (e.g. maybe the new comment that just got submitted).
6. Maybe while the client is waiting for some of this information to be returned from the server, it displays an icon that looks like below.



## Auto-complete Ajax Example

When you go to sites like Google or Amazon, as you type in the search term, it displays a drop-down of possible search terms, like below.



How is this done?  Using Ajax!  Whenever the browser fetches new information from the server AFTER the page has been loaded, it does this via Ajax. Let's say it again: any time the browser retrieves additional information from the server AFTER it renders the page, it is all Ajax (assuming the sites were built in PHP, Python, Ruby/Rails, .NET, Java, or basically everything except Node+socket.io).  Now, let's see how this auto-complete drop-down menu in Amazon could have worked.

1. The user types something in the Search box.
2. Whenever the user changes or types something in the search box, the browser sends a HTTP request to say /auto\_complete\_suggestions and sends data (maybe it sends data that was stored in a form like <form ...><input type="text" name="search\_term" /></form>. Again the browser does NOT go to this new page nor reload the page, it just waits. In other words, this data submission from the client/browser to the server is happening in the background (or without going to another page).
3. The browser waits for the server to send some information back.
4. Once the server sends some information back to the client/browser (say top 10 search suggestions), the browser displays these 10 search suggestions in a drop-down format using JavaScript.

### Summary

Let's say you want to create a page that can make a ***remote HTTP request***(when a button is clicked, when a form is submitted, or when someone pressed a key) and want to have your app **use the HTTP response from that request**to update ***part of your page***.  You can do this using Ajax.  You do NOT always have to submit a FORM to do an Ajax call but as we often have to send information to a specific URL to retrieve certain information, we're introducing the FORM concept as it would make it easier for you to group your data in preparation for making a new HTTP request.

## API Can Give You JSON

We will be creating a quotes application using the quotes API that we will create. We will also be creating a series of links of which any developer may use to contribute to our quotes database. For example, Twitter API has a series of links that developers can use to access the Twitter database of tweets. It's imperative that you follow each part of the application as it's being presented, because we will continue to modify our application to introduce different strategies in creating your API. First we will create a set of links where if developers request a particular URL, they will get back a response that would be JSON data holding our quotes. After we do this, we will modify our API so that it returns a partial (HTML) instead so that we don't have to dynamically create our HTML using the JSON data.

APIs are usually a web URL where you can pass information (either in the URL by the GET method or by posting a form to the URL) where the server returns JSON (or other similar data format). Let's say we have a great database of quotes, and we want to create an API where other developers, including ourselves, can access this great resource to make applications. Let's create our own API and use it to build an application that might look something like this. Go ahead and build the HTML/CSS using this picture. Yes, type out all of the quotes as well. Building our HTML/CSS first will help us visualize what we need to do with our data.



Now, let's build our API. Download  [quote\_model.sql](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_2653/handouts/chapter2653_3301_quote-model.sql" \t "_blank) and run it in your workbench. This will create a schema called 'myownapi' and inside it, we will have a table 'quotes' with columns 'id', 'quote', and 'author.' The table should already be pre-populated with great quotes. Now that we have a great database of quotes, how can we create an API that other developers (including us) can use in our future projects?

First, let's connect our database with our CI application:

if(ENVIRONMENT == 'production')

{

$db['default']['hostname'] = 'localhost';

$db['default']['username'] = 'root';

$db['default']['password'] = '';

$db['default']['database'] = '';

}

else

{

$db['default']['hostname'] = 'localhost';

$db['default']['username'] = 'root';

$db['default']['password'] = 'root';

$db['default']['database'] = 'myownapi';

}

Then let's set our default controller to quotes:

$route['default\_controller'] = "quotes";

$route['404\_override'] = '';

//end of routes.php

At this point, we will replace the main controller with a quotes controller.  This controller will have an instance method index() that loads the index view file, as well as an instance method index\_json() that echoes the JSON encoding of all quotes in our database. Note: we have not created our model or implemented the all() instance method yet. After visualizing the behavior we want, now we can construct our model.

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Quotes *extends CI\_Controller* {

public *function* index\_json()

{

$this->load->model("Quote");

$data = array();

$data["quotes"] = $this->Quote->all();

echo json\_encode($data);

}

public *function* index()

{

$this->load->view('index');

}

}

Let's create the Quote model we are going to use with one method called all() that will return all the quotes in our quotes table:

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Quote *extends CI\_Model* {

public *function* all()

{

return $this->db->query("SELECT \* FROM quotes")->result\_array();

}

}

You just created your own API that either you or other developers can use! You can now go to http://localhost:8888/quotes/index\_json to access the quotes database through a URL. If you decide to deploy your API with the domain name www.coolapi.com, developers will be able to access it with the URL http://www.coolapi.com/quotes/index\_json.

## Using Our API

Let's have our index method in our Quotes controller render the view named 'index' which will just have a button that says 'Get All Quotes.' Once this button is clicked, we will make an AJAX request to /quotes/index\_json to get a response back in JSON. We will be using AJAX because we don't want to refresh the whole page; we want to stay on the sample page while updating only the sections that have changed. Then we are going to build our HTML string with the JSON data and replace whatever is inside the <div id="quotes"></div> with our HTML string which will be interpreted as HTML. We want to wrap author's names in h1 tags and wrap the quotes in p tags. We can do the following with this code:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Quotes</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

$('#get\_all\_button').click(function(){

$.get('/quotes/index\_json', function(res) {

var htmlStr = "";

for(var i = 0; i < res['quotes'].length; i++) {

htmlStr += "<div class='quote'>";

htmlStr += "<h1>" + res.quotes[i].author + "</h1>";

htmlStr += "<p>" + res.quotes[i].quote + "</p>";

htmlStr += "</div>";

}

$('#quotes').html(htmlStr);

}, 'json');

});

});

</script>

</head>

<body>

<h1>Quotsy</h1>

<button id="get\_all\_button">Get All Quotes</button>

<div id="quotes">

</div>

</body>

</html>

# API Can Give You HTML

An API does not have to return JSON. Let's create a URL that will return a partial (HTML) as a response instead of JSON. This will allow us to not go through the process of dynamically generating our HTML with JavaScript. This process can be very error-prone sometimes and we recommend for the next two assignments to use URLs that return HTML instead of JSON. First let's modify our Quotes controller to have another instance method called index\_html that will return a partial located inside of partials/quotes. Now if other developers want the JSON format of our data they can request /quotes/index\_json and if other developers want the HTML format instead, they can request /quotes/index\_html.

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Quotes *extends CI\_Controller* {

public *function* index\_json()

{

$this->load->model("Quote");

$data["quotes"] = $this->Quote->all();

echo json\_encode($data);

}

public *function* index\_html()

{

$this->load->model("Quote");

$data["quotes"] = $this->Quote->all();

$this->load->view("partials/quotes", $data);

}

public *function* index()

{

$this->load->view('index');

}

}

The partials/quotes view will look something like this:

<?php

foreach($quotes as $quote)

{ ?>

<div class="quote">

<h1><?= $quote['author'] ?></h1>

<p><?= $quote['quote'] ?></p>

</div>

<?php

} ?>

And we will be able to use this API in our index.php in a similar way we used our JSON API.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Quotes</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

$('#get\_all\_button').click(function(){

$.get('/quotes/index\_html', function(res) {

$('#quotes').html(res);

});

});

});

</script>

</head>

<body>

<button id="get\_all\_button">Get All Quotes</button>

<div id="quotes">

</div>

</body>

</html>

When we do $this->load->view, we are printing out whatever is contained in that file. Since what we echo back will have HTML elements such as <body>, the browser can then use that information to render the page appropriately.

## Don't Turn On Ajax Until You Know Your API Works

Instead of having our users click on a button to get all quotes, let's have them display as soon as the page loads. So instead of waiting for the clicking of a button event, we are just going to make an AJAX call with $.get as soon as the document is ready and replace our HTML inside of div with the id of quotes.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Quotes</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

$.get('/quotes/index\_html', function(res) {

$('#quotes').html(res);

});

});

</script>

</head>

<body>

<div id="quotes">

</div>

</body>

</html>

Let's make our API more versatile. We can let developers post information to the URL /quotes/create by using a form if other people want to add on to our great quotes selection. The users of our API will have to post a form with information about the author and his or her quote to successfully save their entry into our database. For example, if we include a form on our page that posts to the URL "quotes/create" with inputs author and quote, the form submission should update the quotes database. We can modify our main page like this:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Quotes</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

$.get('/quotes/index\_html', function(res) {

$('#quotes').html(res);

});

});

</script>

<style>

form {

margin-bottom: 30px;

}

</style>

</head>

<body>

<form action="/quotes/create" method="post">

<p>

<label>Author: </label>

<input type="text" name="author">

</p>

<p>

<label>Quote: </label>

<textarea name="quote"></textarea>

</p>

<input type="submit" value="Add Quote">

</form>

<div id="quotes">

</div>

</body>

</html>

We will have to modify our controller and model respectively.

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Quotes *extends CI\_Controller* {

public *function* \_\_construct()

{

parent::\_\_construct();

$this->load->model("Quote");

}

public *function* index\_json()

{

$data["quotes"] = $this->Quote->all();

echo json\_encode($data);

}

public *function* index\_html()

{

$data["quotes"] = $this->Quote->all();

$this->load->view("partials/quotes", $data);

}

public *function* create()

{

$new\_quote = $this->input->post();

$this->Quote->create($new\_quote);

redirect("/");

}

public *function* index()

{

$this->load->view('index');

}

}

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Quote *extends CI\_Model* {

public *function* all()

{

return $this->db->query("SELECT \* FROM quotes")->result\_array();

}

public *function* create($new\_quote)

{

$query = "INSERT INTO quotes (author, quote) VALUES (?, ?)";

$values = array($new\_quote['author'], $new\_quote['quote']);

return $this->db->query($query, $values);

}

}

Now that we have verified that this URL will work without Ajax, let's build it with Ajax. Whenever we are working with Ajax, it is very helpful to build without it first and then turn it on. Once we know that our API is acting as expected, then can we turn on Ajax. One problem at a time. Once we enter in a new quote, we just need to have the part of the page with the quotes to refresh instead of the whole page refreshing. Separate your page into different chunks and update entire chunks every time it gets updated. This will help you write code that is easy to read and maintain.

# Now Let's Turn On Ajax

Now that we know our API works as expected, we can turn on Ajax. If we want to post data to a URL through Ajax we can use jQuery's function $.post. **We use $.get if we want to send GET variables to the API and we use $.post if we want to send POST variables to the API.**Once again, to do this in pure JavaScript, we would have to write different code for different browsers and versions. Thanks, jQuery!

Let's hook up our jQuery by modifying our main page to this:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Quotes</title>

<script src="https://code.jquery.com/jquery-2.1.3.min.js"></script>

<script>

$(document).ready(function(){

// we are getting all of the quotes so that when the user first requests the page the page

// will already be rendering the quotes

$.get('/quotes/index\_html', function(res) {

// this url returns html string so we can dump that straight into div#quotes

$('#quotes').html(res);

});

$('form').submit(function(){

// there are three arguments that we are passing into $.post function

// 1. the url we want to go to: '/quotes/create'

// 2. what we want to send to this url: $(this).serialize()

// $(this) is the form and by calling .serialize() function on the form it will

// send post data to the url with the names in the inputs as keys

// 3. the function we want to run when we get a response:

// function(res) { $('#quotes').html(res) }

$.post($(this).attr('action'), $(this).serialize(), function(res) {

$('#quotes').html(res);

});

// We have to return false for it to be a single page application. Without it,

// jQuery's submit function will refresh the page, which defeats the point of AJAX.

// The form below still contains 'action' and 'method' attributes, but they are ignored.

return false;

});

});

</script>

<style>

form {

margin-bottom: 30px;

}

</style>

</head>

<body>

<form action="/quotes/create" method="post">

<p>

<label>Author: </label>

<input type="text" name="author">

</p>

<p>

<label>Quote: </label>

<textarea name="quote">

</textarea>

</p>

<input type="submit" value="Add Quote">

</form>

<div id="quotes">

</div>

</body>

</html>

We would also have to modify our controller as well to render a partial of all the quotes instead of causing a redirect to '/' as we did previously.

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Quotes *extends CI\_Controller* {

public *function* \_\_construct()

{

parent::\_\_construct();

$this->load->model("Quote");

}

public *function* index\_json()

{

$data["quotes"] = $this->Quote->all();

echo json\_encode($data);

}

public *function* index\_html()

{

$data["quotes"] = $this->Quote->all();

$this->load->view("partials/quotes", $data);

}

public *function* create()

{

// this is an associative array with 'author' and 'name' with values user entered in the form

// this is what $(this).serialize() sent over to this URL

$new\_quote = $this->input->post();

$this->Quote->create($new\_quote);

// after we create the new quote then we can query the database again and it will include the new

// one we just included

copy

$data["quotes"] = $this->Quote->all();

// then we respond to the AJAX request with a partial that will use the $data variable to generate

// the appropriate html

$this->load->view("partials/quotes", $data);

}

public *function* index()

{

$this->load->view('index');

}

}

## JavaScript / jQuery Debugging Steps (Use Google Chrome, Please)

#### 1. INSPECT ELEMENT

The very first thing that you should do when debugging javascript is to inspect the element and look at the c**onsole** tab to see if there are any errors in your JavaScript code. The JavaScript console may tell you where in your JavaScript code there are syntax errors, saving you lots and lots of time in debugging.

#### 2. Use alert() or console.log()

If you don't see any error messages in the console use the **alert()** or **console.log()** functions at each line of your code to find out where the error exists. We recommend using them in the following locations in your code:

* Right after the **document.ready()** function just to make sure your jQuery is loading properly.
* Right after the**.submit()** handler for your form to make sure your**event listener** is working
* Within the **success function** of your **$.post()** or **$.get()** function.

If you are creating **variables** within your jQuery or JavaScript, you can use **console.log()**to display the properties of them; perhaps there is an error with a particular property of an object, or you are using a string instead of a float. Stuff like that is perfect for logging to the console.

#### 3. Check Network tab

There is a tab called **'network'** when you open '**inspect element'** in your browser. If you recall, this tab allows you to monitor the HTTP requests and responses sent by your web page. Check to see if your page is connecting to the file you are submitting your form to and make sure there aren't any errors on that page. Click on the tab**response/preview** and check whether the URL is returning data for you. If your Javascript/jQuery is working correctly, but your PHP code is awry, using the **network tab** will allow you to see your PHP errors in the **response/preview** tab without turning Ajax off, which is nice because that lessens the scope of potential errors!

#### 4. Check if response is a valid **json** format

Force your form to submit to the page you are posting data to and make sure that the output is in valid json format. If it is not, then the problem is not with your javascript code. This can be checked by using the **network tab** mentioned above. Make sure on your PHP side you are not using any **headers, echoes**(exceptfor the json\_encode line)**,**or **var\_dumps**. The only thing that should print to the screen is the **json-encoded** data. This means**any** PHP errors will make your response an invalid json format.

**Important Concepts**

When we make an Ajax call, we're basically making a ***new HTTP request***.  Consider the code below:

$.post('http://localhost/hello/world', function(data){

...

});

Here, this code makes a new HTTP request (e.g.*localhost/hello/world*).  When the HTTP response is received, **it calls the function that follows, and it passes the *HTTP response in*to that function using an input parameter named '*data*'**.  As always, we could name this 'data' input parameter anything that makes sense to us, just like any other local variable.

Some important concepts to ponder and understand before moving on:

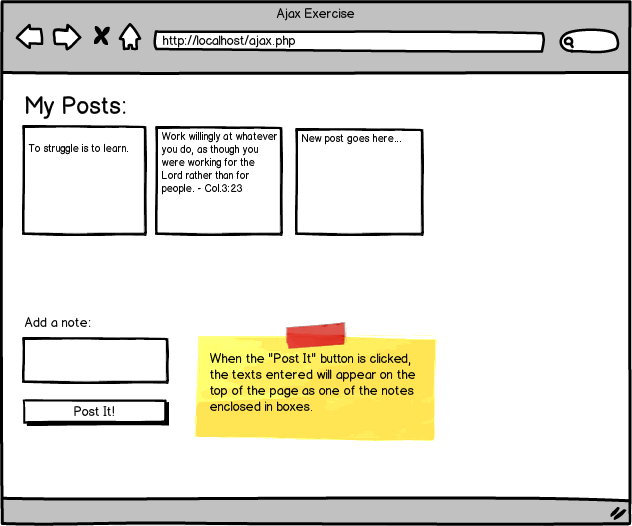
* Since an Ajax call is initiating a new HTTP request, you do not always have to send the code to the same domain. In fact, you could technically send an Ajax call to any URL (e.g. *http://www.cnn.com,*[*http://www.disney.com*](http://www.disney.com/) or whatever) although most recent browsers block this for security. You'll learn how to get around this in one of the advanced assignments in the next chapter.
* It's the HTTP Response from the Ajax call that gets stored to the variable 'data'. You'll later learn how this is done but for now just know that the HTTP response gets stored in that variable *'data'*. This particular *.post*also assumes that the HTTP response is in 'HTML' or text format.  If you were expecting a 'json' output, you could have passed 'json' as the additional argument to the *.post* method (if you are expecting 'json' data, just make sure that the HTTP response starts and ends with a curly bracket).
* Nowhere in the code do we specify whether the HTTP request/response is handled by a particular interpreter. In fact, Ajax on the JavaScript end (step 1 and step 3) is done exactly the same REGARDLESS of whether you're doing Ajax with Rails, Java, C#, PHP, Python, etc as long as the HTTP RESPONSE given back to JavaScript is something your JavaScript can understand (HTML, json or even XML).  In other words, you shouldn't ask how Ajax is done with a particular language as this is all handled by Javascript and it works the exact same way.
* If you want to POST information to a URL, you could use jQuery's .post method.  If you do not want to POST any information but want to merely retrieve information from a URL, you could also use jQuery's*.get* method. Either one would work.

Now, as your Ajax gets more complicated, you may be tempted to ask how could I have my JavaScript client pass something to my database (SQL, MongoDB, etc)? This is the wrong question to ask! :) Your JavaScript NEVER talks to the database nor your controller, config file or routing file. Your JavaScript only knows how to send a new HTTP request and again doesn't care who or what database is being used to process that request (as long as the HTTP response back to the JavaScript client is given back in HTML or in jSON). This is an important distinction to make.

## Assignment: Ajax Posts

Create a simple ''posts page'' where users can post notes and have those notes immediately appear on the page.

Build this in CodeIgniter (ignore the URL shown in the wireframe below).



### Instructions

1. Make sure that you build this form without Ajax first. In other words, make sure the form is posted to say '/notes/create' and make sure '/notes/create' adds a new record in the database and return in json format or you can return a partial (up to you) the results.
2. Make sure that you're storing the information in the database (create a table called 'posts' with 'id', 'description', 'created\_at', and 'updated\_at').
3. Once '/notes/create' is all working perfectly (adding new records to the database and outputting the right json data, or the right partial), then turn Ajax on and make sure it's working now with Ajax.

### Helpful things to remember

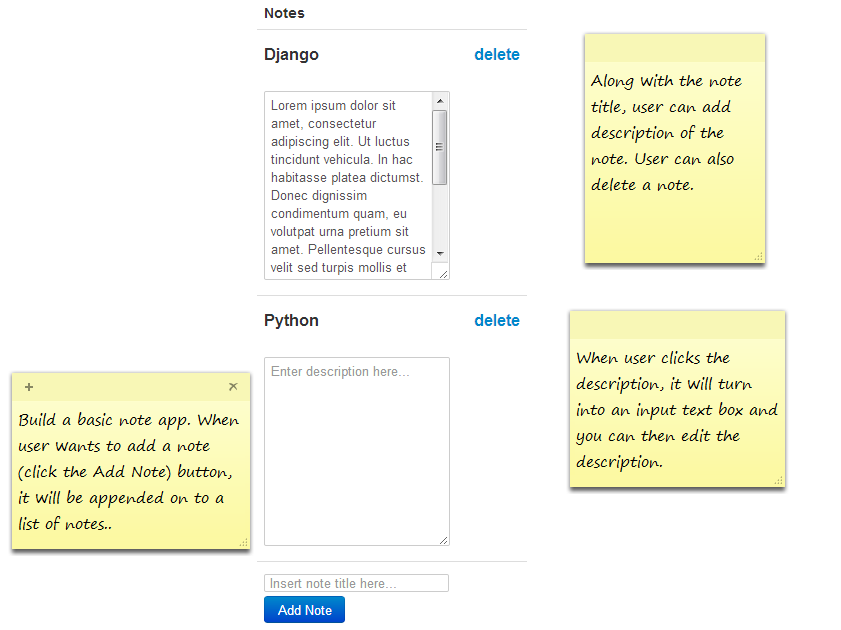
* Remember that .post sends a new HTTP request and can use information received back from that request to update part of the page. The .post method again does not send that information directly to your controller or model. It simply initiates a new HTTP request.
* It's your back-end code's job to provide relevant HTTP response (HTML or json) back for your JavaScript to use. When your back-end codes echo or prints something, all of these are sent back as the HTTP response (in other words, if your codes store information to the proper database but do not print or echo anything, your HTTP response would basically be blank and your JavaScript code may not know what to do).

START WORKING ON THIS

## Assignment: Ajax Notes

**Note: Upload your work in GitHub and allow your collaborators to add/suggest new features for your work.**

Create a Note Manager Application where you can add, edit and delete a note. For example, once you've added two notes, imagine that your application shows something like below:



Note that the user can edit the note description, delete the note, or add a new note.

Now looking at the image above, how many forms would you create?

If you said 5, we're on the same page. We see the following 5 forms that need to be in the web page for the above example.

1. A form for updating the note information (title, description) for the 'Django' note - note that there is no 'button' to submit this form, but that's okay. We'll initially have this 'Update' button in our app and update our information using this 'button' and once we get Ajax working just hide this 'Update' button. Now ask yourself what information would you need to include in this form? Title, Description, and maybe the ID of the note that was created???
2. A form for deleting the note information for the 'Django' note - now ask yourself what information would you need to include in this form? Maybe the id of the note that was created???
3. A form for updating the note information (title, description) for the Python note.
4. A form for deleting the note information for the 'Python' note.
5. A form for creating a new note - now what information should you have in this form?

Once you identify what forms you need, building the app becomes much easier.

### Instructions

1. You should have a database table called 'notes' with 'id', 'title', 'description', 'created\_at', and 'updated\_at' and adding notes adds a new record in the database.
2. Make sure that you make all of this work WITHOUT Ajax first (when you submit the form, it should add/delete/update the note in the database and also output a valid JSON data, or a partial).
3. There is a way to retrieve the primary ID of a new record that was just inserted. If you're using CodeIgniter, this can be done by returning $this->db->insert\_id. For pure PHP, you can do it using a mysql\_insert\_id function (<http://php.net/mysql_insert_id>)

GitHub is a social network hub for programmers. You can look at what other programmers are working on, check their code and make suggestions or changes. You may share projects you are working on and collaborate with other programmers for updates or additional features.

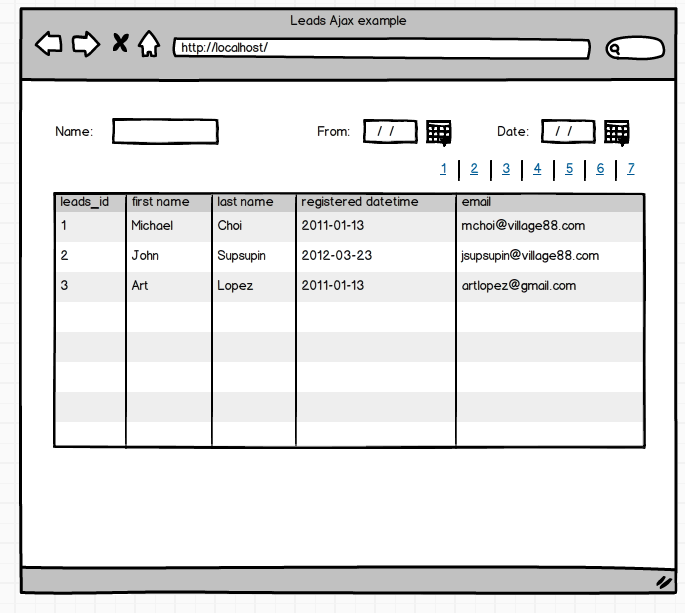
Upload your work for this exercise on GitHub and ask some of your cohort mates or other programmers to peruse your code and suggest changes/updates. Do the same thing for your cohort mates' work.

START WORKING ON THIS

**Assignment: Leads Search and Pagination**

Create using Ajax the application shown below. As the name gets changed, or as you update the from/to date, or as you update the page number, your application should update the table with the appropriate lead information

This is a GREAT assignment where you'll learn a lot and this assignment could be added to your portfolio (see *www.codingdojo.com/portfolio*on how you can add assignments like this to your portfolio page). Only do this if you have extra time. Some of our students have showed this assignment to potential employers and they were genuinely impressed by the fact that the students could build something like this all from scratch.



When you see this, do you see any form? You don't because there is no submit button? Well. In Ajax, you don't need to submit the form just with the submit button and in fact, a lot of Ajax forms do NOT have a submit button. So looking at the wireframe above, what information should be captured inside the form which you will be sending to another URL? Spend a few minutes and think about this.

**Answer**: basically, you want to include all information necessary for generating that table. This means you need one form to include:

1. name (*<input type='text' name="name" />*)
2. from date (*<input type='text' name="from\_date" />*)
3. to date (*<input type='text' name="to\_date" />*)
4. page number (*<input type='hidden' name="page\_number" />*)

Now you may be thinking but page numbers are links not drop-down or input type text. Well. Looks can be deceiving. You could still make the page numbers look like links but may be you can use jQuery to update the value of the page number in the form when those links are clicked, ultimately resulting in the form having a hidden page number that can be submitted via Ajax.

As you do this exercise, ask yourself how many form you would need and what information you would need to put inside the form(s). For the SQL data, you could get it from the MySQL course or click here to download: [Lead SQL](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_2154/handouts/chapter2154_1542_lead-gen-business.sql).

Make sure that as you filter the search, it updates the maximum page number as well. You can utilize mySQL's LIMIT syntax to get results for different pages. For example, "*SELECT \* FROM leads LIMIT 15, 5*" retrieves rows 16-20.

START WORKING ON THIS

# Access Control Origin

Sometimes we will run across this error called Access Control Origin. To bypass this, we have to make an Ajax request from our page to our own server and our server will make the request to the API and give us back the response. **We will be going over three different ways to do this in the next few assignments: using curl library, using file\_get\_contents, and revisiting our simple\_form\_dom library that we used to build web crawlers.**

After studying the iTunes API, we will run across this useful link:  <https://itunes.apple.com/search?term=beyonce&entity=musicVideo>. Go ahead and navigate to that link and you will notice that the URL is responsible for responding with JSON data regarding the name of the artist, which in this case is Beyonce. We are given back a JSON object with two keys: resultCount and result. There is an array of objects stored in the result key. Let's go to the first element in that array of objects and locate the previewUrl key. Inside the previewUrl key is a string that stores the URL or the location of Beyonce's music video: <http://a986.v.phobos.apple.com/us/r1000/035/Video/v4/7c/46/d7/7c46d730-ea58-a6a3-cb27-f467c04bb25c/mzvf_6214270163204897897.640x480.h264lc.U.p.m4v>.

This is exciting! We have a robust API that we can interact with, to make an application where we search for an artist and get previews of that artist's music videos. For example, a user wants to search for 'Kanye', we would get what they inputted into a form, make sure it is a valid name and we take out the spaces, and we insert it in between "https://itunes.apple.com/search?term=" and "&entity=musicVideo." So if we wanted a JSON object that contains the results for 'Kanye', we can go to this link:  <https://itunes.apple.com/search?term=Kanye&entity=musicVideo>. This is perfect. We can just grab what the user has inputted, construct the correct URL, and once we get the response back in JSON, I can dynamically create a video tag and insert the previewUrl on its 'src' attribute.

Let's first build our HTML/CSS. We just need a form with an input type of 'text' so that we can grab its value, generate the correct URL for that artist and make a get request. The URL should respond with JSON data with two keys, result Count and result. We are going to traverse into the result key, which holds an array of objects. We are going to grab the first object and get the value stored in its "previewUrl" key. We are going to grab this URL and insert a video tag with its src attribute set with this URL.

Our HTML/CSS might look something like this:

<h1>Enter Artist's Name:</h1>

<form action="#" method="post">

<label for="user\_input">Enter Artists's Name:</label>

<input id="user\_input" name="user\_input" type="search">

<input type="submit" value="search">

</form>

<div id="results"></div>

It doesn't matter what we specify for the attributes 'action' or 'method' in our form because we are going to use Ajax. We are going to dynamically generate the URL and we are going to make a get request to that URL. We need to remember to return false on our Ajax call to prevent the form from performing the default action of refreshing the page. Notice the for attribute on our label tag. This corresponds to the id attribute of its corresponding input. Also, rather than using a 'text' input, notice that we are using an input type of 'search'. This is a new HTML5 tag that allows different browsers to display their default search field so the user has a better experience. Being mindful of what type the input is becoming more important. For example, specifying an input of type 'number' will bring up a different keyboard for mobile users (the num pad).

Our Ajax call might look something like this:

$(document).ready(function() {

$('form').submit(function() {

var url = "https://itunes.apple.com/search?term=";

url += $('#user\_input').val();

url += "&entity=musicVideo";

$.get(url, function(res) {

if(res.results.length !== 0) {

html\_string = "<video controls src='" +

res.results[0].previewUrl +

"'><\/video>";

} else {

html\_string = "Not Found";

}

$('#results').html(html\_string);

}, 'json');

return false;

});

});

This should work! But we get an error that looks something like this:

No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:8888' is therefore not allowed access.

Apple didn't want us to interact with their API this way. If we directly copy and paste the URL into our browser, the API will give us the JSON object, but if we request the URL from a separate page, Apple will not send us a response. We can't give up yet. There is a clever trick around this restriction. We can make an Ajax call to one of our own URLs and post the value of the user input. The URL will be routed to one of our own controller methods, which will build the URL given the data that was posted by the AJAX call. We will then make a request from our SERVER, and since we are making this request from our SERVER (not a separate page), the iTunes API will happily respond with the JSON data. The controller will then respond to our original post from our view page with the JSON data. The client view page can then traverse through the JSON data we sent, to find 'previewUrl' and set it as the attribute of the video tag we are dynamically adding to our page.

Our new HTML/CSS might look something like this:

<form action="/main/get\_movie" method="post">

<label for="user\_input">Enter Artist's Name:</label>

<input id="user\_input" name="user\_input" type="search">

<input type="submit" value="search">

</form>

Notice that this time the form's action attribute points to one of our own URLs. This route will route to our Main Controller and its get\_movie method will get called. Our Ajax call might look something like this:

$(document).ready(function() {

$('form').submit(function() {

// load up any gif you want, this will be shown while user is waiting for response

$('#results').html("<img src='assets/loading.gif'>");

$.post($(this).attr('action'), $(this).serialize(), function(res) {

var html\_string = "";

if(res.results.length !== 0) {

html\_string = "<video controls src='" + res.results[0].previewUrl + "'><\/video>";

} else {

html\_string = "Not Found";

}

}, 'json');

// don't forget, without it the page will refresh

return false;

});

});

Our Main Controller file might look something like this:

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Main *extends CI\_Controller* {

public *function* index()

{

$this->load->view('index');

}

public *function* get\_movie()

{

$artist = str\_replace(' ', '', $this->input->post('user\_input'));

$url = "https://itunes.apple.com/search?term=".$artist."&entity=musicVideo";

$ch = curl\_init();

curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, 1);

curl\_setopt($ch, CURLOPT\_CONNECTTIMEOUT, 30);

curl\_setopt($ch, CURLOPT\_URL, $url);

curl\_setopt($ch, CURLOPT\_FOLLOWLOCATION, 1);

$data = curl\_exec($ch);

$info = curl\_getinfo($ch);

curl\_close($ch);

echo $data;

}

}

//end of main controller

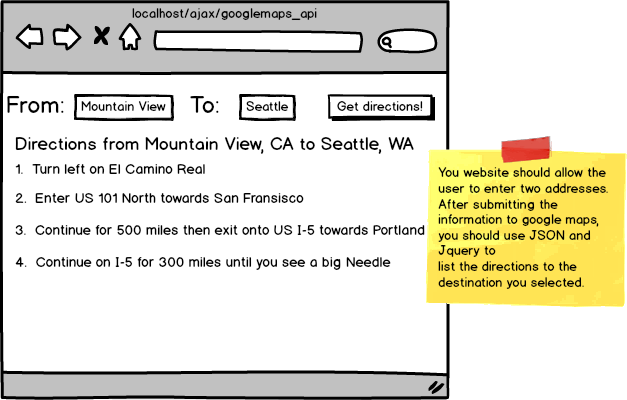
We are using the curl library to make a request to a URL that we specify, and to send the response back to our view file. You can learn more about the curl library here:  <http://php.net/manual/en/book.curl.php>.

**Assignment: Google Directions API**

Google has some wonderful APIs that provide some awesome data.  One of the API we want you to use is the Google's Direction Requests API. We are going to send an Ajax request from the client side to our server, and our server will make the request for us and give us back the request to bypass Access Control Origin again. **This time instead of using the curl library we will be using a built-in method in PHP for doing this.**

<https://developers.google.com/maps/documentation/directions/#DirectionsRequests>

Please take some time to read the documentation and figure out a way to build a simple web app where you can enter the from address and to address and where the app would display the driving directions from your from address to the "to" address.  You'll have to get an API key from them and also use https:// instead of http://.



Now, the browser will try to prevent your site from sending an ajax request to a different domain. This is how to get around this issue of only being able to send an Ajax request to the same domain.

1. Create a separate method in your Controller that will talk to the remote server.  Have your method output json.
2. Have your view file send an Ajax request that is handled by the method in #1.

For example, your form may be submitted to /maps/direction with the origin and destination input values.  Your controller at the end may look like below:

*class* Maps *extends CI\_Controller* {

public *function* index()

{

$this->load->view('index');

}

public *function* direction()

{

$html = file\_get\_contents("https://maps.googleapis.com/maps/api/directions/json?origin=".urlencode($this->input->post('origin'))."&destination=".urlencode($this->input->post('destination'))."&sensor=false");

$this->output

->set\_content\_type('application/json')

->set\_output($html);

}

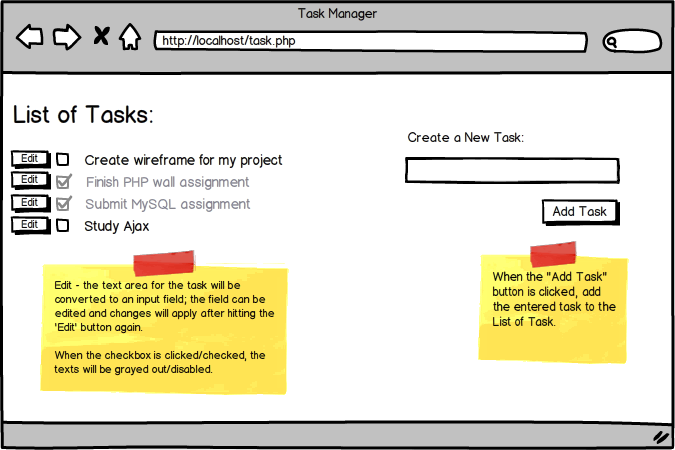
}

Again, make sure that the form is submitted normally and that it does generate proper JSON output before your turn Ajax on.

## Assignment: Task List

IMPORTANT: Only work on this if you have extra time.  We would recommend you do this after you do every required assignment in this chapter and the next chapter.

Create a Task Manager page where you can add, edit and mark completed tasks. Each task marked completed will be disabled. Build this in CodeIgniter (ignore the URL shown in the wireframe below).



Whenever a task is added, edited, or deleted, please update the database as well.

You should have a table called 'tasks' with the 'id', 'name', 'created\_at', and 'updated\_at'.

## Assignment: Weather App

#### Build your very own weather forecast app!

In order to use this API, you'll need to sign up for a free account at  <http://openweathermap.org/api>. Your profile will then be attached to a unique key, which you will add to the end of your URL request in the following format:

http://api.openweathermap.org/data/2.5/forecast/city?id=524901&APPID={INSERTAPIKEY}

// Make sure to put your unique API key in the URL (taking out the brackets).

// &APPID={INSERTAPIKEY} will need to be at the end of each URL you access below and in the assignment.

We will build a weather forecast application where we can input any city and find out the temperature in Fahrenheit. Thankfully there is an API or a set of URLs out there that will tell us the temperature of any city! We will be using the Open Weather API. Open Weather API gives us a set of very useful URLs that are fun to use. For example, navigate to the link [*http://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=<enter< a="" style="box-sizing: border-box; font-family: Gotham-Rounded-Book;"> app id>...</enter<>*](http://api.openweathermap.org/data/2.5/weather?q=London,uk&appid=%3Center) and you will find JSON data about London. In fact, even if you try to navigate to [*http://api.openweathermap.org/data/2.5/weather?q=L*](http://api.openweathermap.org/data/2.5/weather?q=L)os Angeles*&...*, the API will find you the same JSON data. This means that we have to validate different users' responses less on our application. Thanks, Open Weather API!

We are going to have a form where our user can indicate a city and have the city's temperature displayed on the screen. We will be listening for a form to be submitted so that we can gather what the user has filled out and add it to our URL   [*http://api.openweathermap.org/data/2.5/weather?q=*](http://api.openweathermap.org/data/2.5/weather?q=). For example, if the user has typed in 'Los Angeles', we would add this input to [*http://api.openweathermap.org/data/2.5/weather?q=*](http://api.openweathermap.org/data/2.5/weather?q=) so that the URL ends up being [*http://api.openweathermap.org/data/2.5/weather?q=Los*](http://api.openweathermap.org/data/2.5/weather?q=Los) Angeles&appid=. And do not forget to add the API key at the last part of the URL!

We can listen to a form being submitted with jQuery's submit() function. However, the submit() function automatically reloads the page so we are going to be returning false. Now that we know how to do AJAX we don't want our page to reload: we want the data to come to us and then we can dynamically update the parts that need to be updated in our current page. This should be a good starting point for setting up your jQuery to build this application.

$(document).ready(function() {

$('form').submit(function() {

// your code here (build up your url)

$.get(url, function(res) {

// your code here

}, 'json');

// don't forget to return false so the page doesn't refresh

return false;

});

});

START WORKING ON THIS

# The Internet Is Your API

We learned how to make AJAX calls to different URLs. We even made an AJAX call to a URL that specifically blocks AJAX calls from different domains. We were able to bypass this security feature by making an AJAX request to our own server and have our server use that data to make an AJAX request to the API. Since we used our server to make the AJAX call our API will happily respond to our server with the response. Then the server will get that response and send it back to our view file which we can then use to dynamically generate our HTML. This is a powerful concept. Yes, there are specific URLs out there that are meant to be easily used by developers to make their websites better. However, we shouldn't limit ourselves to just these official APIs. By making our server interact with the Internet, the sky is our limit with the wealth of resources on the Internet.

For example, let's say you want to build a web browser inside of your page. A user can type in any URL, such as http://www.google.com, you would send that URL as a string to your server. The server will then get that URL and make a request to that URL. We will get back the response, grab just what is in between <body></body> and display it on our page. It is as if we navigated to Google without ever leaving our page!

Our HTML/CSS might look something like this:

<form action="/main/get\_page" method="post">

<input name="url" type="text">

</form>

<div id="display"></div>

Our AJAX call might look something like this:

$(document).ready(function() {

$('form').submit(function() {

$.post($(this).attr('action'), $(this).serialize(), function(res) {

$('#display').html(res);

});

return false;

});

});

We will use the simple\_form\_dom.php we used to make our web crawler to make the process of sending the request and getting the response a breeze. Go ahead and download  *[simple\_form\_dom.php](http://s3.amazonaws.com/General_V88/boomyeah/company_209/chapter_2653/handouts/chapter2653_3315_simple-form-dom.php" \t "_blank)* and add it to application/libraries/ in your CI application.

Our controller might look something like this:

<?php if ( ! defined('BASEPATH')) exit('No direct script access allowed');

*class* Main *extends CI\_Controller* {

public *function* index()

{

$this->load->view('index');

}

public *function* get\_page()

{

require('application/libraries/simple\_form\_dom.php');

$html = file\_get\_html($this->input->post('url'));

foreach($html->find('body') as $element) {

echo $element;

}

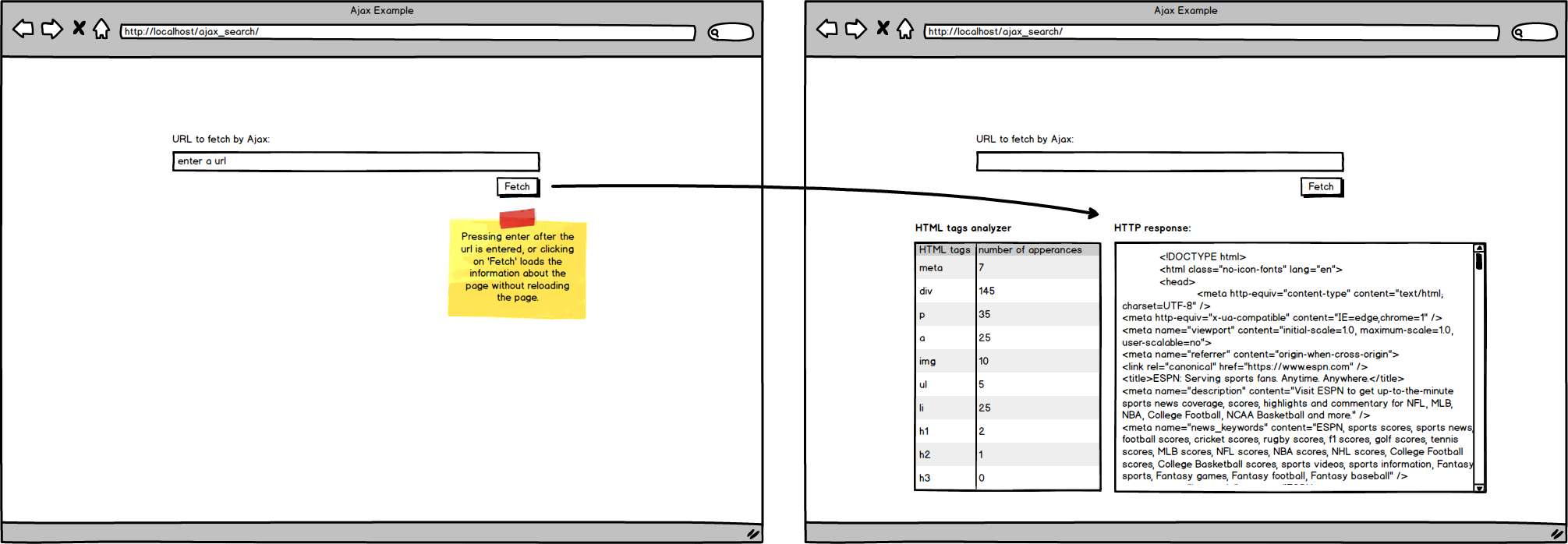
}

}

This is some powerful stuff.

**HTTP analyzer**

Create an application that accepts a url and sends information about that page as demonstrated below:



There are two ways to accomplish this.

**Doing this with front-end Javascript**

First is to have Javascript in your browser send a request directly and to have Javascript analyze the http response to generate the output.  If you were to do this, your code would look something like this:

$.get( {{destination url}} , function(res) {

# codes for analyzing the http response - probably anywhere from 10-20 lines of code

#

#

#

#

#

# codes for putting together the html - probably anywhere from 20-40 lines of code

#

#

#

#

#

#

#

#

# codes for updating the html - probably a few lines

#

}

However, ask yourself the following questions:

1. Is it really the best practice to put together HTML using front-end Javascript?
2. If the web application changes and the appearance changes dramatically, how hard would it be to change the website with this new look if Javascript was used heavily for putting together the HTML (like above)?

If you've thought about these questions above, you'll come to find that it's best to keep Javascript short and to have these analysis and html rendering be done by the back-end.

**Doing this with back-end**

Another way to handle tasks like this is to build your own API.  For example, say that you've built it so that sending a request to {{your\_website}}/api/http\_analyzer, you send a http response that looks like this:

{ tags: { "p": 3, "a": 15, ... }, response: "......" }

Your front-end javascript would now look like this:

$.get( "/api/http\_analyzer" , function(res) {

# codes for putting together the html - probably anywhere from 20-40 lines of code

#

#

#

#

#

#

#

#

# codes for updating the html - probably a few lines

#

}

Note that this is a lot simpler than the first approach.  Also, note that now you can make your API available to any one else in the world where they can use your API to get information about any URL!

Note however that you're still putting together the HTML using Javascript.  Can you think of how you could even make the code above simpler?  One way is to build another URL where it returns the html.  For example, what if sending a request to "/api/http\_analyzer/html" sends html response tat you can insert directly in Javascript?

If you were to do this route, your front-end code would look more like this:

$.get( "/api/http\_analyzer/html" , function(res) {

$(...).html(res);

}

**Assignment**

When you're working on real projects, you'll see all three variations.  Most amateur programmers rely heavily on Javascript to process/parse data and to put together HTML.  Most experienced developers rely on back-end to process/parse data and keep their front-end Javascript very very simple.

For you, try all three versions of doing this assignment and notice the differences.  Always remember that for your Javascript, you want to keep things simple and do NOT want Javascript to put together the HTML.  Javascript wasn't built for putting together HTML and you can use the back-end to much more efficiently put together HTML.

START WORKING ON THIS

**Your feedback is important for us**

Please take a few minutes to fill out two short surveys:

* PHP Track Survey - please visit this link: <https://forms.gle/cAVs8mLBaYdshMR7A> and when you write your feedback, make sure the course name is written as "**PHP Track**".
* General Training Survey - please visit <https://forms.gle/nyZQjMMhKFKyqa1v5>.  Your honest feedback here would be helpful as well as your general testimonial about your training so far.