Synchronous vs. Asynchronous Code

Objectives:

- Understand how our current code is synchronous
- Learn about asynchronous code

AJAX stands for **asynchronous JavaScript And XML**. This probably doesn't help if we don't cover what asynchronous means. And in order to do that, we need to discuss what synchronous means first!

Synchronous Code

This is what we've been doing up to this point, when code runs in the order in which it is written. For example, here is some synchronous code that we have seen before. This is taken from a project where a Flask server is querying a database:

```
@app.route('/friends')
def index():
    mysql = connectToMySQL("friendsdb")                             # Step 1
    print("Connected to our database!")                             # Step 2
    all_friends = mysql.query_db("SELECT * FROM friends")           # Step 3
    print("Fetched all friends", all_friends)                       # Step 4
    return render_template('index.html', friends = all_friends)     # Step 5
```

The code is executed exactly as we see it written.

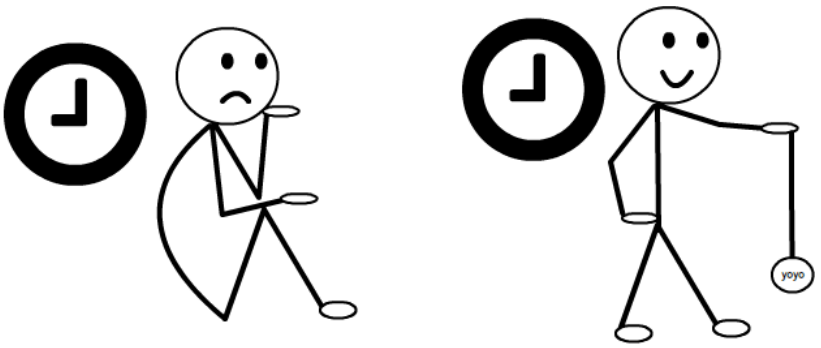
. Connect to the database
. Print to celebrate our connection to the database
. Make a query and store the result in a variable
. Print the result
. Render a template with the data from the result

This has been serving our purposes very well, but consider what would happen if the query is super complex with lots of joins. This would cause the query to take a long time to complete. How would that affect the output we see? We would experience a lag in between the two prints. In other words, we would see Step 2 printed, then wait for Step 3 to complete, and then finally, after a lag, see the print in Step 4. **Our code would not be able to move on to step 4 until step 3 is completed.**

Most of the time we love synchronous code. After all, in the example above, we are relying on the fact that we are going to wait for each line of code to complete before moving on to the next. It would be awful if we proceed to Steps 4 and 5 before we get a response from Step 3 - we would be trying to render a template without any data!

Asynchronous Code

While our synchronous code above is doing its job just fine, let's think beyond the server. What caused this function to run to begin with? To invoke this function, a client must have made a request to the /friends route. The server does not send a response until it has the necessary data, so what does that mean for the user experience on the client side?



A client with nothing to do but wait vs a client kept occupied while waiting

Of course the client must wait for the response from the server, but in the meantime, we can keep the client busy with other tasks! With **asynchronous programming,** our client's code does not actually have to run in the order it is written. Below is an example demonstrating how we may give the user a chance to play a game until the response from the server comes back.

```
$('#sendRequest').click(function(){          // Step 1 - a click happens
    $.ajax({                                 // Step 2 - make an AJAX get request to /friends
        url: "/friends",
        method: "GET"
    })
    .done(function(res){                     // Step 4 - receive the response
        $('#game').hide();                   // Step 5 - hide the game
        $('#result').html(res);              // Step 6 - manipulate the html to display the d
ata from the server
    })
    $('#game').show();                       // Step 3 - show the game to occupy the user whi
le waiting
})
```

Notice that the code is executed in an order **different from how it is written.**

. A click on the #sendRequest element triggers the code to run
. A get request is made to /friends
. Jump to the last line of code and the #game element is shown
. The response comes back
. The #game element is hidden
. The #result element is altered to show the response from the server

A Roommate Analogy

If you're having trouble visualizing synchronous versus asynchronous programming, consider this analogy. Let's say you have a roommate (let's call her Python), and one day, you leave her a list of chores that need to be done:

```
Chores for Python:
1. Wash dishes
2. Sweep the floors
3. Sign for package delivery
4. Feed the cat
```

When you come home many hours later, you find Python sitting patiently on the porch, gazing down the street. Puzzled, you walk into the house and see the dishes have been done, the floors have been swept, but the cat is desperately trying to get your attention to feed her. For some reason, Python hasn't fed the cat yet! That's when you are hit with the realization: Python is outside waiting for the package to be delivered! It's not her fault, she's just being synchronous. She cannot move on to the fourth chore until the third chore is completed.

Fed up with your synchronous roommate, you decide to get a new one (let's call him JavaScript). JavaScript reads the same list of chores and washes the dishes, sweeps the floors, and then, being an absolute genius, very smartly moves on to feeding the cat. In the meantime, JavaScript listens for the doorbell so he can sign for the package at the appropriate time.

So is JavaScript asynchronous?

No, JavaScript is synchronous. However, we can use JavaScript to **write asynchronous code**. To do this, we need to use **callbacks,** which are functions *passed as arguments to other functions that will be invoked later*. Look again at this code snippet and identify the callbacks:

```
$('#sendRequest').click(function(){          // the .click() method accepts a callback
    $.ajax({
        url: "/friends",
        method: "GET"
    })
    .done(function(res){                     // the .done() method accepts a callback
        $('#game').hide();
        $('#result').html(res);
    })
    $('#game').show();
})
```

Notice that the callbacks are used to define the code that we want to be invoked *later.* The `.click()` method accepts a callback that defines what we want to do *after* the click has occurred. The `.done()` method accepts a callback that defines what we want to do *after* the response returns from the server.

Here's a simple example demonstrating this concept of asynchronicity: