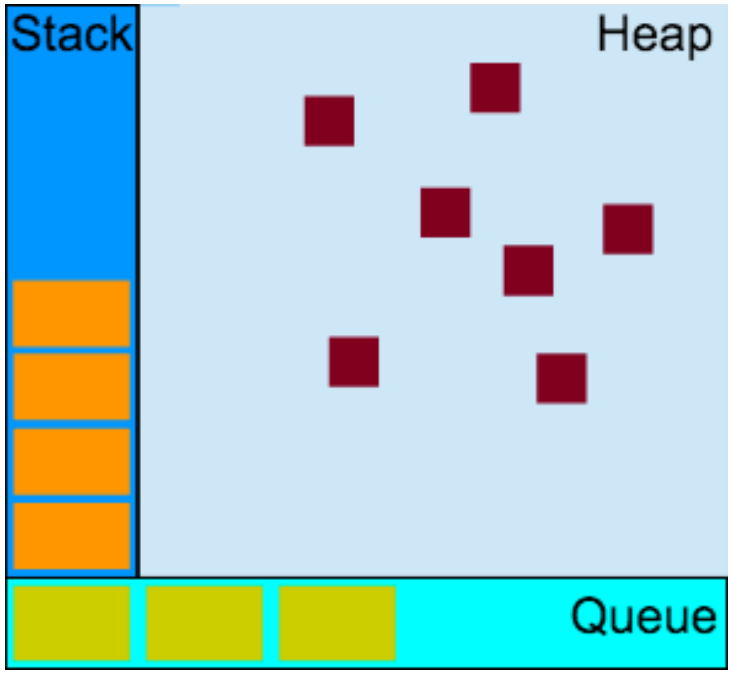
AJAX and REST APIs

Review: the event queue

- In C/C++, Java, etc., function calls live on the stack, and dynamic objects live on the heap
- The function on the top of the stack executes

Review: the event queue

- In JavaScript, function calls live on the stack, objects live on the heap, and *messages live on the queue*
- The function on the top of the stack executes.
- *When the stack is empty, a message is taken out of the queue and processed.*
- Each message is a function



Review: event-driven programming

- In event-driven programming, the flow of the program is determined by events
  - Example: page load
  - Example: user clicks a button
- Event-driven programming is useful for GUIs like web applications

Review: events and HTML

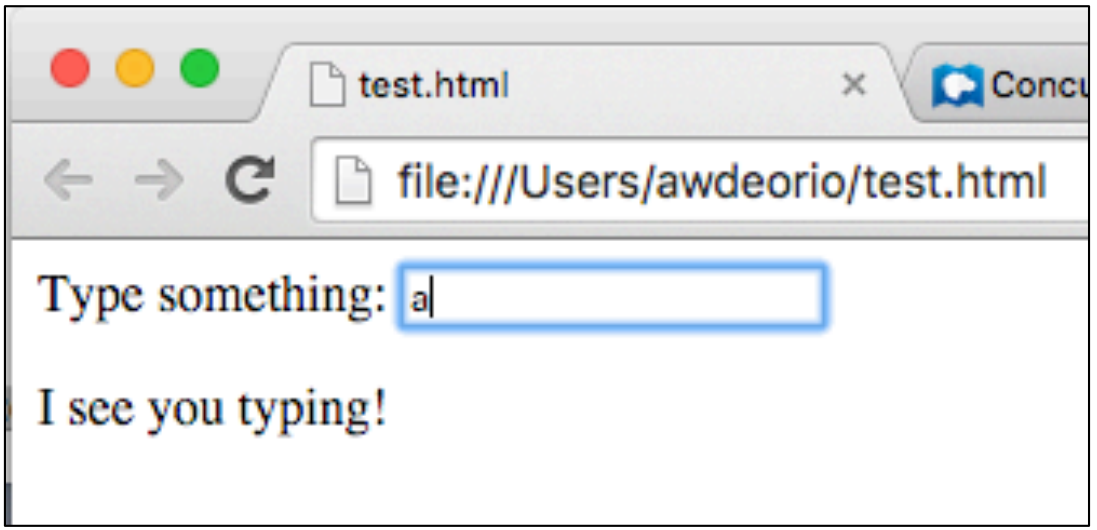
- You can couple JavaScript functions with events in HTML
- test.html

```
<html>
<head><script src="test.js"></script></head>
<body>Type something
<input type="text" onkeyup="hello()">
<p id="hello"></p>
</body>
</html>
```

Review: events and HTML

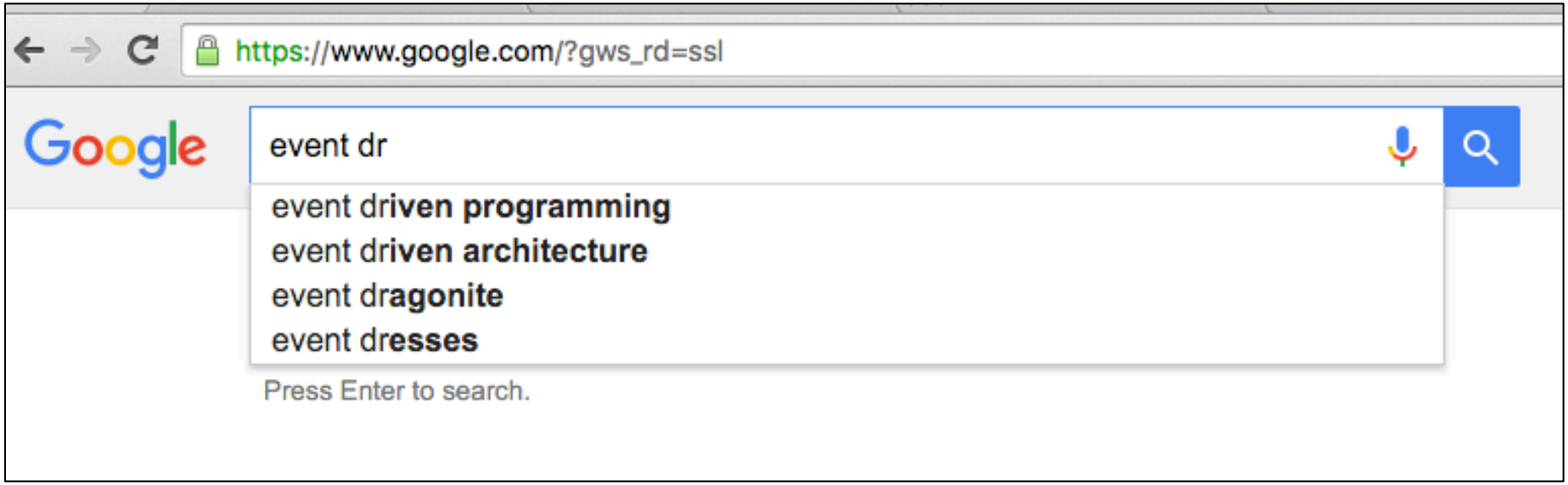
- You can couple JavaScript functions with events in HTML
- test.js

```
function hello() {
    document.getElementById("hello").innerHTML =
"I see you typing!";
}
```



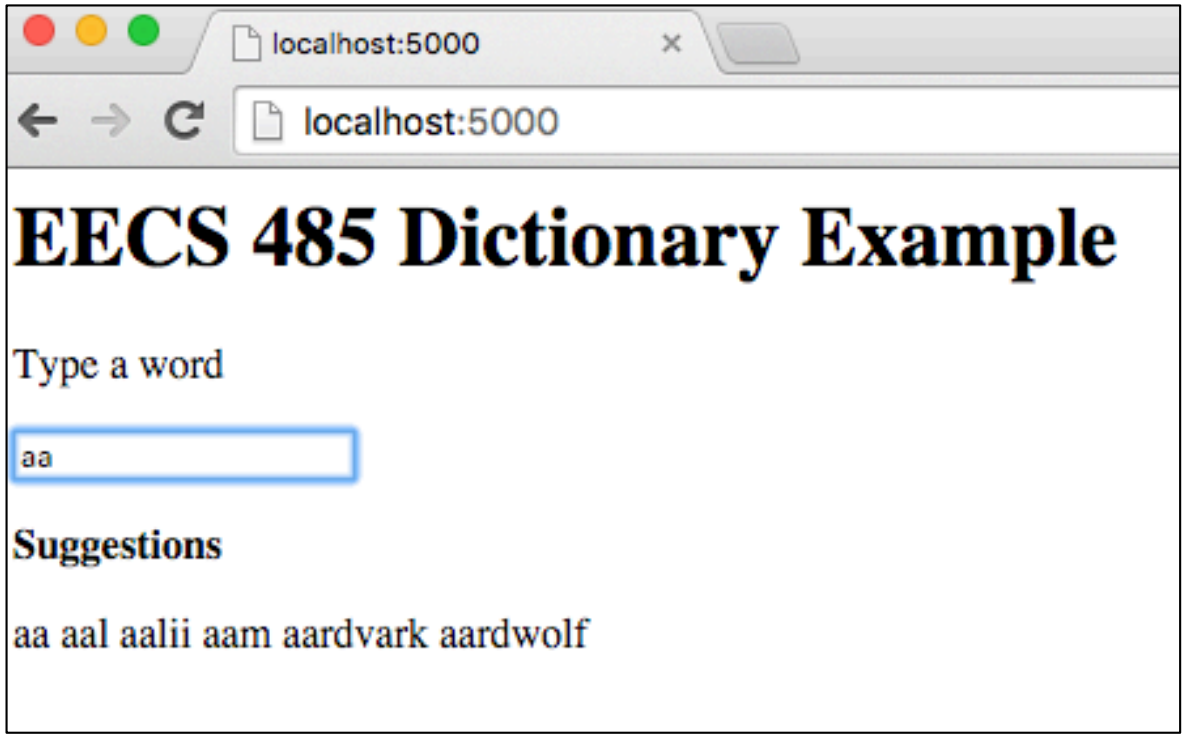
Another way to use events

- Another way to use events:
  - 1. Request data from a server
  - … do other stuff (like accept user input)
  - 2. Process data when it arrives
- Example: Google search suggestions



Another way to use events

- Let's build an application that suggests words from the dictionary

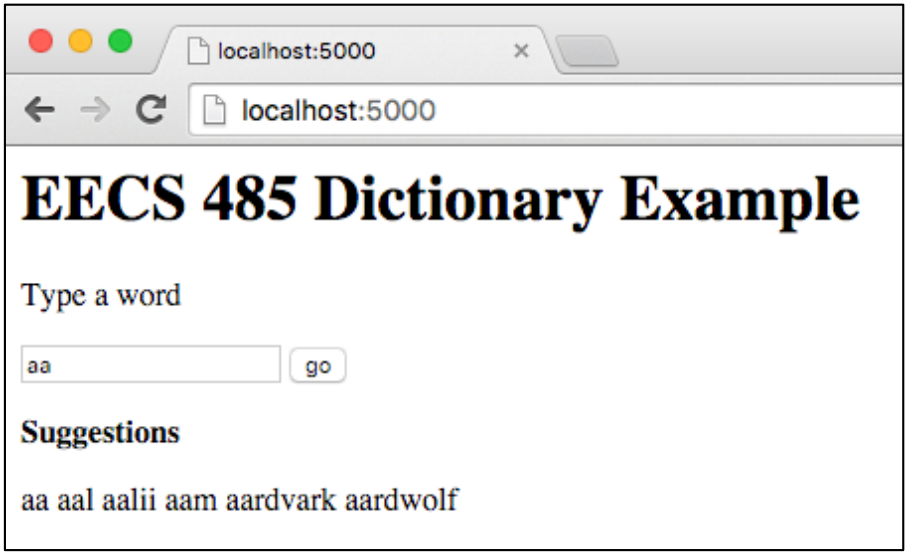


Another way to use events

- Let's build an application that suggests words from the dictionary
- We will write our application three ways
  - Entirely on the server with page loaded with dictionary lookup already there
  - Client requests dictionary lookups from server using *synchronous* requests
  - Client requests dictionary lookups from server using *asynchronous* requests
    - This is called *AJAX*

Server side application

- Entirely on the server: page loads with dictionary lookup contents
- Load web page
- Type a word
- Click "go"
- Page reloads

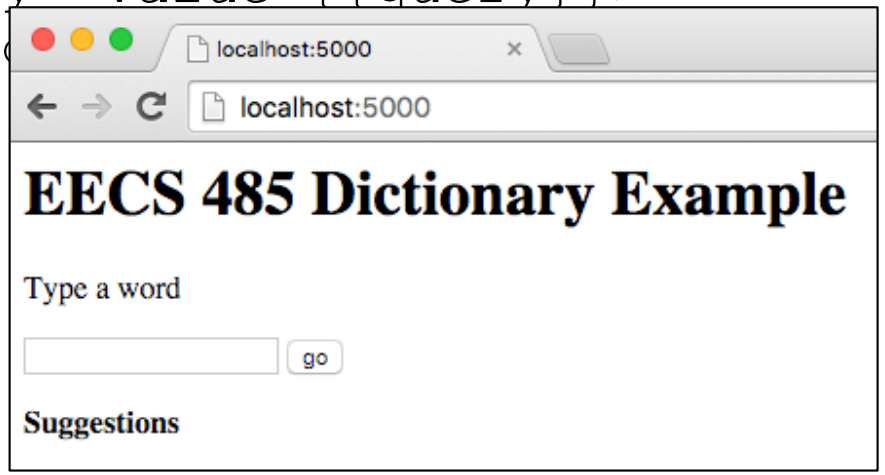


- For this example, we'll use Python with the Flask library

Server side application

- templates/index.html, using jinja2

```
<html>
<head></head>
<body>
<h1>EECS 485 Dictionary Example</h1>
<p>Type a word</p>
<form action="" method="post"
enctype="multipart/form-data">
<input type="text" name="query" value={{query}}>
<input type="submit" name="ad
</form>
<p><b>Suggestions</b></p>
<p>{{suggestions}}</p>
</body></html>
```



Server side application

- Now we need to lookup the suggestions
- First, import libraries and load dictionary
- app.py

```
import time
from flask import Flask, request, render_template
app = Flask(__name__)

# load dictionary from hard drive

dictionary = [ line.rstrip('\n')
  for line in open('/usr/share/dict/words') ]
```

Server side application

- Next, add a route for / that renders the page from the index.html template

```
@app.route('/', methods=['GET', 'POST'])
def query():
    query = ''
    suggestions=[]
    # lookup code will go here
    return render_template(
        'index.html',
        query=query,
        suggestions=' '.join(suggestions)
        )
```

Server side application

- Simulate a slow server

```
@app.route('/', methods=['GET', 'POST'])
def query():
    query = ''
    suggestions=[]
    # lookup code will go here
    time.sleep(1)
    return render_template(
        'index.html',
        query=query,
        suggestions=' '.join(suggestions)
        )
```

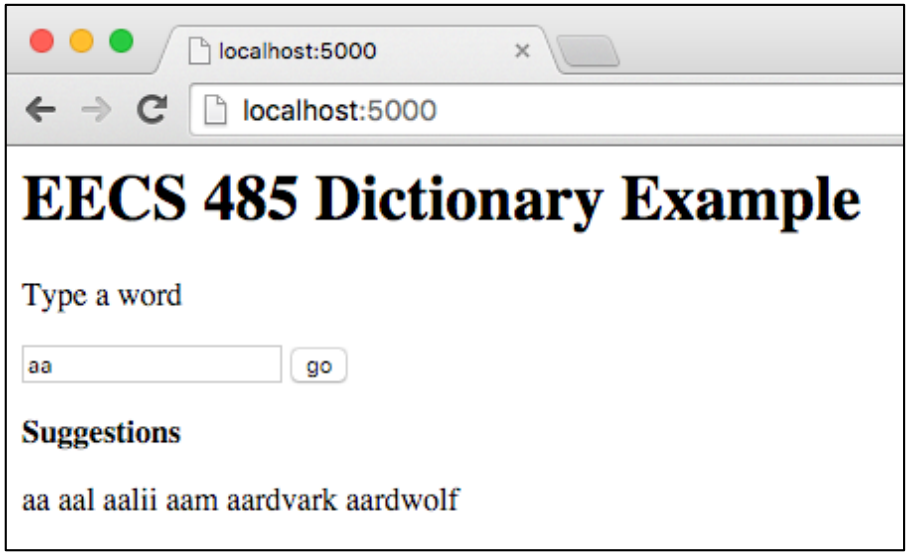
Server side application

- Now, we need to accept a post request from the form and look up the word

```
@app.route('/', methods=['GET', 'POST'])
def query():
    query = ''
    suggestions=[]
    if request.method == 'POST':
        query = request.form['query']
        if query != '':
            suggestions = [ word for word in dictionary
                            if word.startswith(query) ]
    time.sleep(1)
    return render_template(
        'index.html',
        query=query,
        suggestions=' '.join(suggestions)
        )
```

Server side application

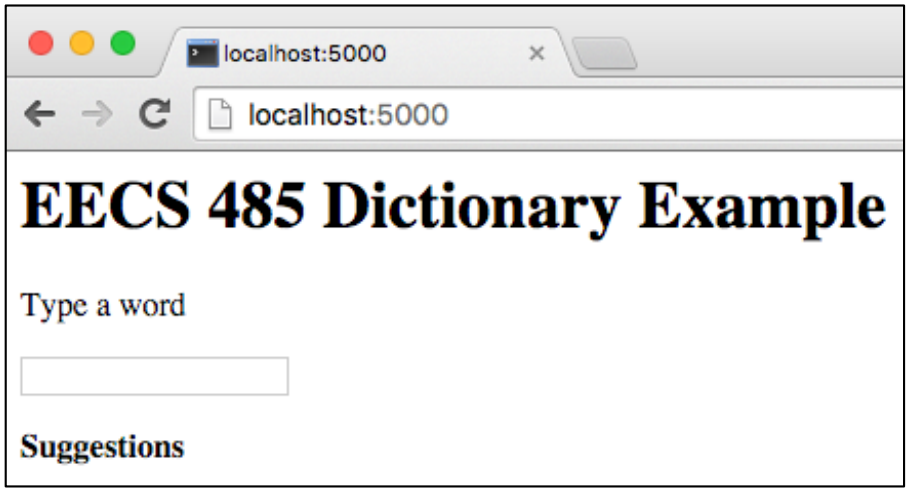
- It works!



- But ...
  - You have to click "go" each time
  - The entire page reloads!

Client side application

- Let's get rid of that "go" button
- Use JavaScript events to trigger a query to the server



- Then we won't have to reload the page!
- We'll make a client-side application using JavaScript

Client side application

- We'll need 3 files:
  - The server application: `app.py`
  - The web page: `/static/index.html`
  - The client application: `/static/javascript/completion.js`

Client side application

- We'll start with the HTML
- No need for jinja2 templates, since we'll use JavaScript to fill in the suggestions
- First, tell the browser where to find the JavaScript source code

```
<html>
<head>
<script src="/static/javascript/completion.js">
</script>
</head>
```

Client side application

- Next, add an input box with an event handler for a key press
- The function will be called on the text inside the text box

```
<html>
<head><script src="/static/javascript/completion.js">
</script></head>
<body>
<h1>EECS 485 Dictionary Example</h1>
<p>Type a word</p>
<input type="text"
onkeyup="completion(this.value)">
```

Client side application

- Finally, add a named HTML object where we can fill in the suggestions

```
<html>
<head><script src="completion.js"></script></head>
<body>
<h1>EECS 485 Dictionary Example</h1>
<p>Type a word</p>
<input type="text"
onkeyup="completion(this.value)">
<p><b>Suggestions</b></p>
<p id="words"></p>
</body>
</html>
```

Client side application

- On to the JavaScript!
- We're writing the callback function that will be used when a `keyup` event occurs

```
function completion(query) {

}
```

Client side application

- `XMLHttpRequest` is an API supported by the browser
- Use it to send a request via HTTP
- Despite the name, the response can be XML, JSON, HTML or plain text
- We'll use plain text today

```
function completion(query) {
    var req = new XMLHttpRequest();
    //...
}
```

Client side application

- Just like before, we'll use POST
- We will need a new route on our server to handle these requests: `/api/v1/dictionary`
- `false` means synchronous request
- Function will wait for response

```
function completion(query) {
    var req = new XMLHttpRequest();
    req.open("POST", '/api/v1/dictionary', false);
    req.setRequestHeader(
        "Content-type",
        "application/x-www-form-urlencoded"
    );
    //...
}
```

Client side application

- Send the request
- Wait for response

```
function completion(query) {
    var req = new XMLHttpRequest();
    req.open("POST", '/api/v1/dictionary', false);
    req.setRequestHeader(
        "Content-type",
        "application/x-www-form-urlencoded"
    );
    req.send("query=" + query);
    document.getElementById("words").innerHTML =
        req.responseText;
}
```

Client side application

- Modify the DOM with suggestions text

```
function completion(query) {
    var req = new XMLHttpRequest();
    req.open("POST", '/api/v1/dictionary', false);
    req.setRequestHeader(
        "Content-type",
        "application/x-www-form-urlencoded"
    );
    req.send("query=" + query);
    document.getElementById("words").innerHTML =
        req.responseText;
}
```

Client side application

- On to the server in `app.py`
- Again, initialize dictionary

```
import time
from flask import Flask, request
app = Flask(__name__)
dictionary = [ line.rstrip('\n') for line in
    open('/usr/share/dict/words') ]
```

Client side application

- / URL now returns a static file

```
# ...
@app.route('/')
def root():
    return app.send_static_file('index.html')
```

Client side application

- Add URL route for dictionary lookup

```
# ...
@app.route('/api/v1/dictionary', methods=['POST'])
def query():
    suggestions=[]
    if request.method == 'POST':
        query = request.form['query']
        if query == '': return ''
        suggestions = [ word for word in
           dictionary if word.startswith(query) ]
    time.sleep(1) # simulate slow server
    return ' '.join(suggestions)
```

RESTful APIs

- The `/api/v1/dictionary` URL is an example of a RESTful API
- REST = **RE**presentational **S**tate **T**ransfer
- Style of programming on the web
  - Not a protocol!!
- No formal specification

RESTful APIs

- Basic idea is to use HTTP as the “language” for invoking remote services
- Each request is self-contained – no memory
- In this example, the dictionary lookup is a remote service
- Data is often returned as XML or JSON
  - We used plain text in this example for simplicity

Testing our RESTful API

- Test our RESTful API without a browser
- Start server

```
$ python app.py
```

- Send POST request from command line

```
$ curl --data "query=aa"
http://localhost:5000/api/v1/dictionary
aa aal aalii aam aardvark aardwolf
```

- It's working!

Testing our RESTful API

- Now, let's test it together with our HTML and JavaScript

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

- Browse to `http://localhost:5000/`

```
127.0.0.1 - - [06/Oct/2015 13:46:15] "GET / HTTP/1.1" 304 -
```

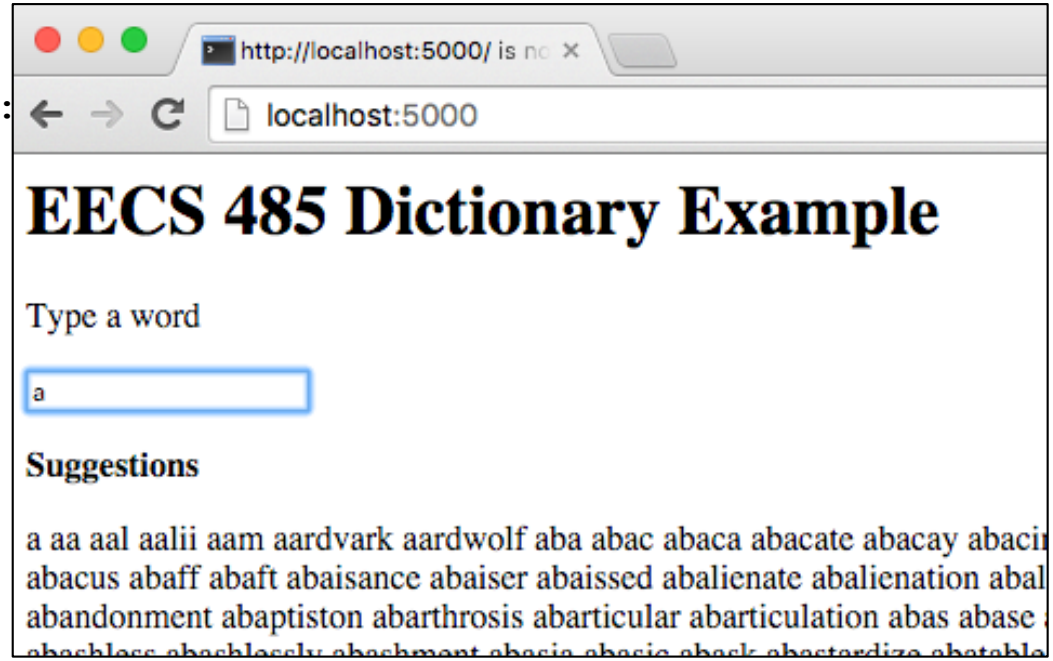
Testing our RESTful API

- Now, let's test it together with our HTML and JavaScript

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [06/Oct/2015 13:46:15] "GET / HTTP/1.1" 304 -
```

- Type "a"

```
127.0.0.1 - - [06/Oct/2015 13:46:
```



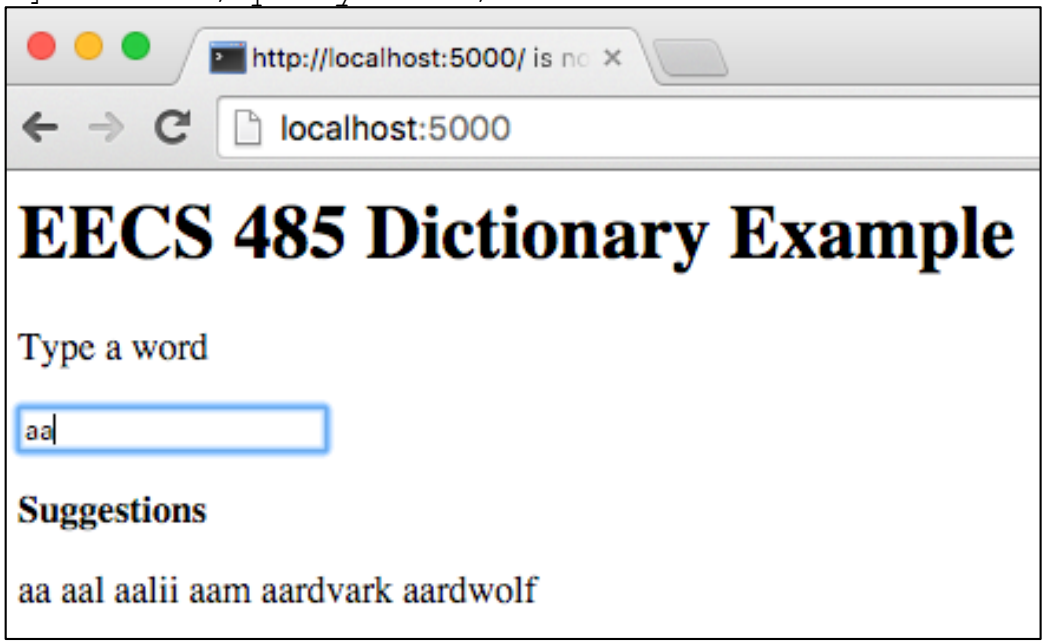
Testing our RESTful API

- Now, let's test it together with our HTML and JavaScript

```
$ python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [06/Oct/2015 13:46:15] "GET / HTTP/1.1" 304 -
127.0.0.1 - - [06/Oct/2015 13:46:17] "POST /query HTTP/1.1" 200 -
```

- Type "a" again

```
127.0.0.1 - - [06/Oct/2015 13:50:3
```



Problems

- We now have a client-side app that makes calls to a server-side RESTful API
- Problem:
  - It takes time to load the data
  - The UI hangs while waiting for data!
    - Typing "aar" results in:
      - Press "aar" quickly
      - See "a"
      - Wait
      - See "aa"
      - Wait
      - See "aar"
  - FRUSTRATING!!!

Problem

- The root of the problem is *synchronous* code that *blocks* until it returns
- Blocking means "wait until this function returns"

```
function completion(query) {
    var req = new XMLHttpRequest();
    req.open("POST", '/api/v1/dictionary', false);
    req.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
    req.send("query=" + query);
    document.getElementById("words").innerHTML =
        req.responseText;
}
```

Solution

- The root of the problem is *synchronous* code that *blocks* until it returns
- The solution is *asynchronous* code that *does not block*
- Here's the idea
  - Request data
  - ... do something else
  - Process/display data

AJAX

- AJAX: Asynchronous JavaScript and XML
- It doesn't have to be XML. JSON, plain text, etc. are OK

- Key idea: event handler submitted to the *queue* which will display the data

AJAX

- Old synchronous code:

```
function completion(query) {
    var req = new XMLHttpRequest();
    req.open("POST", '/api/v1/dictionary', false);
    req.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
    req.send("query=" + query);
    document.getElementById("words").innerHTML =
        req.responseText;
}
```

- All we need to change is this function

AJAX

```
function completion(query) {
    var req = new XMLHttpRequest();
    var show_suggestions = function() {
        if (req.readyState == 4 &&
            req.status == 200) {
          document.getElementById("words").innerHTML =
              req.responseText;
        }
    }
    req.onreadystatechange = show_suggestions;
    req.open("POST", '/api/v1/dictionary', true);
    req.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
    req.send("query=" + query);
}
```

- These lines of code have not changed

AJAX

```
function completion(query) {
    var req = new XMLHttpRequest();
    var show_suggestions = function() {
        if (req.readyState == 4 &&
            req.status == 200) {
          document.getElementById("words").innerHTML =
              req.responseText;
        }
    }
    req.onreadystatechange = show_suggestions;
    req.open("POST", '/api/v1/dictionary', true);
    req.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
    req.send("query=" + query);
}
```

- Big difference: write a callback function that will process the data

AJAX

```
function completion(query) {
    var req = new XMLHttpRequest();
    var show_suggestions = function() {
        if (req.readyState == 4 &&
            req.status == 200) {
          document.getElementById("words").innerHTML =
              req.responseText;
        }
    }
    req.onreadystatechange = show_suggestions;
    req.open("POST", '/api/v1/dictionary', true);
    req.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
    req.send("query=" + query);
}
```

- Register the callback function on the event queue

AJAX

```
function completion(query) {
    var req = new XMLHttpRequest();
    var show_suggestions = function() {
        if (req.readyState == 4 &&
            req.status == 200) {
          document.getElementById("words").innerHTML =
              req.responseText;
        }
    }
    req.onreadystatechange = show_suggestions;
    req.open("POST", '/api/v1/dictionary', true);
    req.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
    req.send("query=" + query);
}
```

- Tell `open` to use an asynchronous request

Continuation-passing style

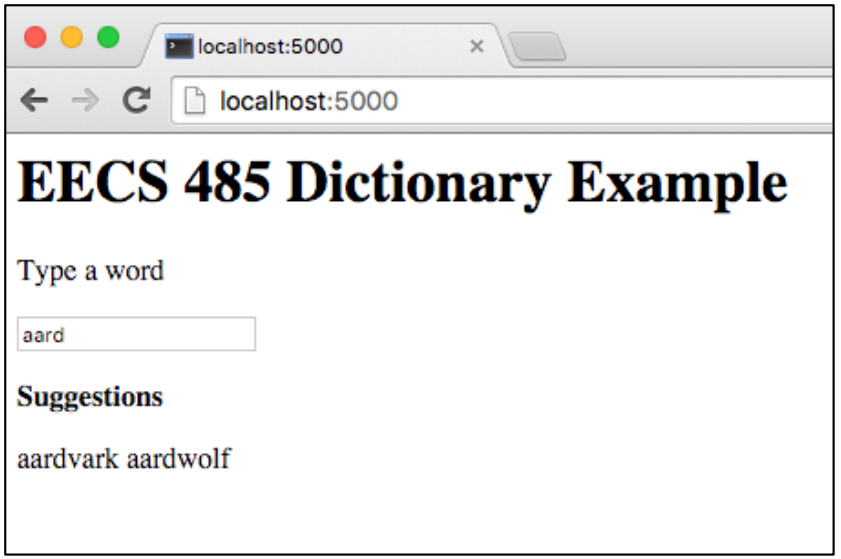
- Continuation-passing style (CPS) originated as an intermediate representation for compilers
- It's great for programming non-blocking systems!
  - Remember, non-blocking just means "Do other stuff while waiting for the server"
- More on CPS http://matt.might.net/articles/by-example-continuation-passing-style

Continuation-passing style

- Not CPS:
  - Send request
    ```
    req.send("query=" + query);
    ```
  - Wait, then respond:
    ```
    document.getElementById("words").innerHTML =
            req.responseText;
    ```
- CPS:
  - Turn second action into a callback function
    ```
    var show_suggestions = function() {
        document.getElementById("words").innerHTML =
                    req.responseText; }
    ```
  - "Do this callback when you're finished"
    ```
    req.onreadystatechange = show_suggestions;
    ```
  - Send request
    ```
    req.send("query=" + query);
    ```

Testing

- We now have an AJAX application
  - The server application: `app.py`
  - The web page: `static/index.html`
  - The client application: `static/javascript/completion.js`
    - Makes an asynchronous call to server
- The GUI no longer hangs!
- ... but, it still waits for each response:
- "a", then "aa", then "aar", "aard" ...



Server Latency with AJAX

- The application still waits for each response:
  - "a", then "aa", then, "aar", "aard" ...
- Solution: multiple server threads

```
$ gunicorn -w 4 app:app
[2015-10-07 13:01:19 -0400] [46935] [INFO] Starting gunicorn 19.3.0
[2015-10-07 13:01:19 -0400] [46935] [INFO] Listening at: http://127.0.0.1:8000
(46935)
[2015-10-07 13:01:19 -0400] [46935] [INFO] Using worker: sync
[2015-10-07 13:01:19 -0400] [46938] [INFO] Booting worker with pid: 46938
[2015-10-07 13:01:19 -0400] [46939] [INFO] Booting worker with pid: 46939
[2015-10-07 13:01:19 -0400] [46940] [INFO] Booting worker with pid: 46940
[2015-10-07 13:01:19 -0400] [46941] [INFO] Booting worker with pid: 46941
```

- Parallelism reduced the latency of subsequent requests

Server Latency with AJAX

- AJAX apps typically involve many tiny requests
  - Get search suggestions for new typed character
  - Fetch the 29-character email
  - Grab state name for a zipcode
  - Update back-end video game state
- If we use AJAX techniques, the HTTP server must be EXTREMELY low latency
  - Unlike standard Web “big push” POSTs

jQuery

- jQuery is a JavaScript library
- Like STL or Boost for C++
- Let's rewrite our app to use jQuery

- Again, all we need to change is our JavaScript code

jQuery

- OK, I lied a little bit. We need to change our HTML to include the jQuery library:

```
<html>
<head>
<script src="http://code.jquery.com/jquery-2.0.0.js"></script>
<script src="/static/javascript/completion.js"></script>
</head>


...
```

jQuery

- Back to completion.js

```
function completion(query) {
  $.post(
    "/api/v1/dictionary",
    { query: $("input").val() },
    function(data, status){
        $("#words").text(data);
    }
  );
}
```

jQuery

- Make a POST request
- To the `/api/v1/dictionary` URL
- With key=query and value=(text box contents)

```
function completion(query) {
  $.post(
    "/api/v1/dictionary",
    { query: $("input").val() },
    function(data, status){
        $("#words").text(data);
    }
  );
}
```

jQuery

- Callback function

```
function completion(query) {
  $.post(
    "/api/v1/dictionary",
    { query: $("input").val() },
    function(data, status){
        $("#words").text(data);
    }
  );
}
```

jQuery

- Callback function
- Selects "words" HTML object
  - `<p id="words"></p>`

```
function completion(query) {
  $.post(
    "/api/v1/dictionary",
    { query: $("input").val() },
    function(data, status){
        $("#words").text(data);
    }
  );
}
```

jQuery

- Callback function
- Modify the text within the HTML object

```
function completion(query) {
  $.post(
    "/api/v1/dictionary",
    { query: $("input").val() },
    function(data, status){
        $("#words").text(data);
    }
  );
}
```

REST API best practices

REST APIs frequently communicate using JSON

Using GET, and friends

- GET used to retrieve items
- PUT used to update items
- POST used to insert items
- DELETE used to delete items

Let's modify our app to use GET requests and JSON

JSON

- JSON is a data-interchange format (string) that looks like a JavaScript object
- Example:

```
{
  "suggestions": [
    "aa",
    "aal",
    "aalii",
    "aam",
    "aardvark"
  ]
}
```

Modifying the server

- Flask ships with a function for formatting JSON

```
from flask import Flask, request, jsonify
app = Flask(__name__)

# Load dictionary into memory
dictionary = [ line.rstrip("\n") for line in
  open("/usr/share/dict/words") ]

@app.route("/")def root():
  return app.send_static_file("index.html")

# ...
```

Modifying the server

- We're changing the API, so bump the version

```
# ...
@app.route("/api/v2/dictionary", methods=["GET"])
def query():
  suggestions=[]
  if request.method == "GET":
    query = request.args.get("query")
    if query != "":
      suggestions = [word for word in dictionary
                     if word.startswith(query)]
  return jsonify(suggestions=suggestions)

if __name__ == "__main__":
  app.run(debug=True)
```

Testing the server

- Send a GET request at the command line

```
$ curl -v "localhost:5000/api/v2/dictionary?query=aa"
...
< HTTP/1.0 200 OK
< Content-Type: application/json
{
  "suggestions": [
    "aa",
    "aal",
    "aalii",
    "aam",
    "aardvark",
    "aardwolf"
  ]
}
```

Modifying the client

- Raw JavaScript

```
function completion_rawJS(query) {
  var req = new XMLHttpRequest();
  var show_suggestions = function() {
    if (req.readyState == 4 && req.status == 200){
      response_obj = JSON.parse(req.responseText)
      response_words = response_obj.suggestions;
      response_text = response_words.join(" ")
      document.getElementById("words").innerHTML =
        response_text;
    }
  }
  req.onreadystatechange = show_suggestions;
  url =  '/api/v2/dictionary?query=' + query
  req.open("GET", url, true);
  req.send();
}
```

Modifying the client - jQuery

- With jQuery

```
function completion(query) {
  url = '/api/v2/dictionary?query=' + query
  handler = function(data, status) {
    response_words = data.suggestions;
    response_text = response_words.join(" ")
    $("#words").text(response_text);
  }
  $.getJSON(url, handler);
}
```

Conclusions

- AJAX is a pattern of programming that relies asynchronous calls
- Avoid blocking on IO
- Uses continuation passing style