

chatserver.epmikida.repl.co

CSE 503

Introduction to Computer Science for Non-Majors

Dr. Eric Mikida

epmikida@buffalo.edu

208 Capen Hall

Day 24
AJAX (Part 1)

Recap

- We learned how to run our own Python web server with bottle

Recap

- We learned how to run our own Python web server with bottle
 - `@bottle.route()` → Tell bottle how to respond to requests

Recap

- We learned how to run our own Python web server with bottle
 - `@bottle.route()` → Tell bottle how to respond to requests
 - `bottle.run()` → Run our server

Recap

- We learned how to run our own Python web server with bottle
 - `@bottle.route()` → Tell bottle how to respond to requests
 - `bottle.run()` → Run our server
 - `bottle.static_file()` → Respond to a request with an html file

Recap

- We learned how to run our own Python web server with bottle
 - `@bottle.route()` → Tell bottle how to respond to requests
 - `bottle.run()` → Run our server
 - `bottle.static_file()` → Respond to a request with an html file
 - `bottle.template()` → Respond with a templated html file

Recap

- We learned how to run our own Python web server with bottle
 - `@bottle.route()` → Tell bottle how to respond to requests
 - `bottle.run()` → Run our server
 - `bottle.static_file()` → Respond to a request with an html file
 - `bottle.template()` → Respond with a templated html file
 - `bottle.request.query` → Dictionary for the requests query string

Bottle Web Servers so far...

We are now able to write a web server which responds to HTTP requests

On the client side...how can we get user input?

How can we get information from the server without reloading the entire web page each time?

Chat Server

Over the next two lectures we will build up a working chat server

End Goal: chatserver.epmikida.repl.co

Much of the code will be using concepts we've already learned

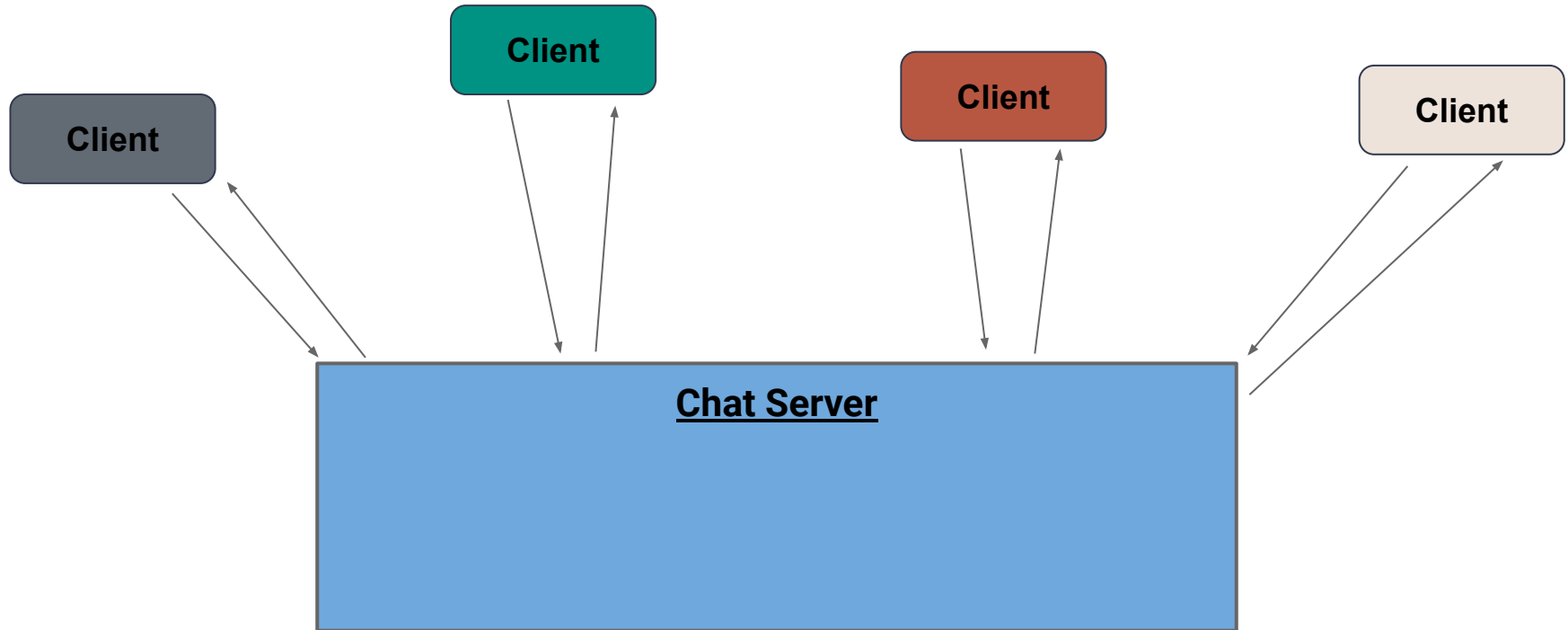
We'll be adding something called AJAX (probably next lecture)

Chat Server Design

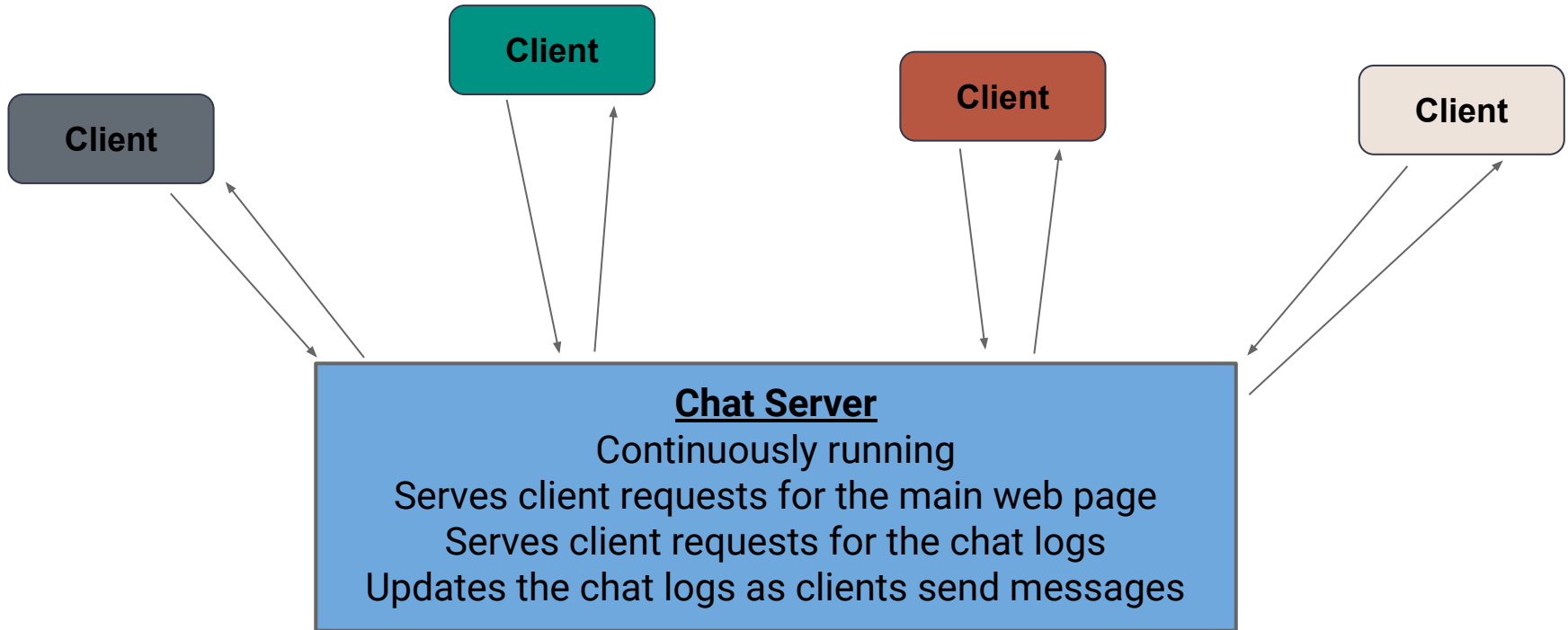
Before we code: Let's figure out the different parts of our web app and set up our file structure

(when working on larger projects, laying out a basic structure and understanding how pieces interact keeps you organized)

End Goal



End Goal



Chat Server Design

What do we need:

Chat Server Design

What do we need:

1. A front end web page (with interactive components)

Chat Server Design

What do we need:

1. A front end web page (with interactive components)
2. Front end JavaScript allowing users to interact with the page

Chat Server Design

What do we need:

1. A front end web page (with interactive components)
2. Front end JavaScript allowing users to interact with the page
3. Web server code to run the server and handle requests

Chat Server Design

What do we need:

1. A front end web page (with interactive components)
2. Front end JavaScript allowing users to interact with the page
3. Web server code to run the server and handle requests
4. A place to store messages that persists even when server stops

Chat Server Design

What do we need:

1. A front end web page (with interactive components)
2. Front end JavaScript allowing users to interact with the page
3. Web server code to run the server and handle requests
4. A place to store messages that persists even when server stops
5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Chat Server Design

What do we need:

1. A front end web page (with interactive components)
2. Front end JavaScript allowing users to interact with the page
3. Web server code to run the server and handle requests
4. A place to store messages that persists even when server stops
5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Note this is just one possible design!

Chat Server Design - File Structure

For now let's setup the following (in a Python REPL with bottle):

- `main.py`: Our Python server code
- `index.html`: Our web page
- `chat.js`: JavaScript for interacting with the web page
- `chat.txt`: A file to store the chat logs
- `chat.py`: Python code for reading and writing our logs

Back End Web Server

To start, let's set up a basic web server to serve our HTML file...

Back End Web Server (main.py)

```
import bottle

@bottle.route('/')
def index():
    return bottle.static_file("index.html", root="")

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Back End Web Server (main.py)

```
import bottle ←  
  
@bottle.route('/')  
def index():  
    return bottle.static_file("index.html", root="")  
  
bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Import the bottle library
(don't forget to install it)

Back End Web Server (main.py)

```
import bottle  
  
@bottle.route('/')  
def index():  
    return bottle.static_file("index.html", root="")  
  
bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Import the bottle library
(don't forget to install it)

Setup a function that gets
called when someone sends a
request to root ("/")

This function will return index.html as a static file

Back End Web Server (main.py)

```
import bottle  
  
@bottle.route('/')  
def index():  
    return bottle.static_file("index.html", root="")  
  
bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Import the bottle library
(don't forget to install it)

Setup a function that gets
called when someone sends a
request to root ("/")

This function will return index.html as a static file

Lastly, run the server

Front End Web Page (index.html)

Now we can write the HTML for the web page...

It will include some new things we have not seen yet

Front End Web Page (index.html)

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

Front End Web Page (index.html)

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

Load our JavaScript file in head so it is available for us to use

Front End Web Page (index.html)

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

Use the `onload` attribute of our `body` element to call the `loadChat()` function from our JavaScript file.

Front End Web Page (index.html)

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

Create a **div** element and set its **id** so we have somewhere for our JavaScript file to put the chat messages

Front End Web Page (index.html)

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

We can create an **input** element with **type** set to **"text"** to create a text box.

By giving it an **id** we can access it from our JavaScript code.

Front End Web Page (index.html)

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

The **button** element is used to create a clickable button.

The **onClick** attribute allows us to set a function to be called when it is clicked (we'll define it in our JS)

The text between the open and close tag shows up on the button.

Front End JavaScript (chat.js)

Now that we have the HTML, we need to define the JavaScript code that it was expecting – chat.js

Front End JavaScript (chat.js)

```
function loadChat(){  
    // Load the chat...  
}  
  
function sendMessage(){  
    // Send a message  
}
```

Front End JavaScript (chat.js)

```
function loadChat(){  
    // Load the chat...  
}  
  
function sendMessage(){  
    // Send a message  
}
```

In `chat.js` we have to define the functions that our HTML was relying on: `loadChat` and `sendMessage`

Front End JavaScript (chat.js)

```
function loadChat(){  
  // Load the chat...  
}  
  
function sendMessage(){  
  // Send a message  
}
```

For now we can just have them do something simple, we'll create more complicated versions of them later.

This lets us set up the structure and put together the main pieces, without focusing on complex details until later...

A Minimal Test

With larger projects, don't expect to get everything working right away!

Let's do a sanity check now, just to see if we can request the web page from the server and display it...

What Went Wrong?

Why didn't our JavaScript update the web page?

What Went Wrong?

Why didn't our JavaScript update the web page?

Let's check the web server output...

What Went Wrong?

Why didn't our JavaScript update the web page?

Let's check the web server output...

Our server got a request for chat.js...did we handle that request?

HTML and HTTP Requests

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

HTML and HTTP Requests

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

On this line, we want to load chat.js...but where is chat.js located?

HTML and HTTP Requests

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

On this line, we want to load chat.js...but where is chat.js located?

On the server!

HTML and HTTP Requests

```
<html>
<head>
  <script src="chat.js"></script>
</head>

<body onload="loadChat();">

  <div id="chat"></div><br/>
  Message: <input type="text" id="message"><br/>
  <button onClick="sendMessage();">Send</button>

</body>
</html>
```

On this line, we want to load chat.js...but where is chat.js located?

On the server!

Loading a .js file sends a request to the server for that file...

Handling the chat.js Request

```
import bottle

@bottle.route('/')
def index():
    return bottle.static_file("index.html", root="")

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Our server currently only handles requests for "/"

Handling the chat.js Request

```
import bottle

@bottle.route('/')
def index():
    return bottle.static_file("index.html", root="")

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

Our server currently only handles requests for "/"

Let's add the option to handle requests for "/chat.js" as well...

Handling the chat.js Request

```
import bottle

@bottle.route('/')
def index():
    return bottle.static_file("index.html", root="")

@bottle.route('/chat.js')
def chatJS():
    return bottle.static_file("chat.js", root="")

bottle.run(host="0.0.0.0", port=8080, debug=True)
```


Handling the chat.js Request

```
import bottle

@bottle.route('/')
def index():
    return bottle.static_file("index.html", root="")

@bottle.route('/chat.js')
def chatJS():
    return bottle.static_file("chat.js", root="")

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

`bottle.static_file()` can be used to send `.js` files as well

Chat Server Design

What do we need:

1. A front end web page (with interactive components)
2. Front end JavaScript allowing users to interact with the page
3. Web server code to run the server and handle requests
4. A place to store messages that persists even when server stops
5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Chat Server Design

What do we need:

- ✓ A front end web page (with interactive components)
- ✓ Front end JavaScript allowing users to interact with the page
- ✓ Web server code to run the server and handle requests
- 4. A place to store messages that persists even when server stops
- 5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Storing Chat Logs (chat.txt)

Now we can create a place on the server to store the chat logs...

Storing Chat Logs (chat.txt)

Now we can create a place on the server to store the chat logs...

In this case we can just store them in a text file (let's call it chat.txt)

Reading and Writing Chat Logs (chat.py)

Now that we have a place to store our chat, we need to be able to read and write from the chat logs.

We'll write this in `chat.py` to keep it separate from server code.

Reading and Writing Chat Logs (chat.py)

```
filename = "chat.txt"

def get_chat():
    full_chat = []
    with open(filename) as file:
        for line in file:
            full_chat.append({"message": line.rstrip("\n")})
    return full_chat

def add_message(message):
    with open(filename, "a") as file:
        file.write(message + "\n")
```

Reading and Writing Chat Logs (chat.py)

```
filename = "chat.txt"
```

Create a variable with the filename so we can refer to it throughout the rest of the code

```
def get_chat():  
    full_chat = []  
    with open(filename) as file:  
        for line in file:  
            full_chat.append({"message": line.rstrip("\n")})  
    return full_chat
```

```
def add_message(message):  
    with open(filename, "a") as file:  
        file.write(message + "\n")
```


Reading and Writing Chat Logs (chat.py)

```
filename = "chat.txt"
```

```
def get_chat():  
    full_chat = []  
    with open(filename) as file:  
        for line in file:  
            full_chat.append({"message": line.rstrip("\n")})  
    return full_chat
```

```
def add_message(message):  
    with open(filename, "a") as file:  
        file.write(message + "\n")
```

Read from the chat file, and return a list of messages. We've put the messages in a dictionary...more on that later.

Reading and Writing Chat Logs (chat.py)

```
filename = "chat.txt"

def get_chat():
    full_chat = []
    with open(filename) as file:
        for line in file:
            full_chat.append({"message": line.rstrip("\n")})
    return full_chat
```

```
def add_message(message):
    with open(filename, "a") as file:
        file.write(message + "\n")
```

Write a function to add a new message to the chat file. Note the file mode: "a". This means append.

Reading and Writing Chat Logs (chat.py)

Note that chat.py does not have any server code...it just reads and writes files, and we can test it just like any other Python code.

Reading and Writing Chat Logs (chat.py)

Note that chat.py does not have any server code...it just reads and writes files, and we can test it just like any other Python code.

When building applications from smaller pieces, make sure to test the pieces individually, let's do that now with chat.py.

Chat Server Design

What do we need:

- ✓ A front end web page (with interactive components)
- ✓ Front end JavaScript allowing users to interact with the page
- ✓ Web server code to run the server and handle requests
- 4. A place to store messages that persists even when server stops
- 5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Chat Server Design

What do we need:

- ✓ A front end web page (with interactive components)
- ✓ Front end JavaScript allowing users to interact with the page
- ✓ Web server code to run the server and handle requests
- ✓ A place to store messages that persists even when server stops
- 5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Chat Server Design

What do we need:

- ✓ A front end web page (with interactive components)
- ✓ Front end JavaScript allowing users to interact with the page
- ✓ Web server code to run the server and handle requests
- ✓ A place to store messages that persists even when server stops
- 5. A way for the front end and back end to communicate ***even after the page is initially loaded***

Next lecture we'll tackle step 5 and bring it all together