

CS 338: Graphical User Interfaces

Lecture 4-2: Web Development III

Web Dev Continued

- We've already covered (briefly)...
 - HTML, CSS, JavaScript, jQuery, Bootstrap
- This week, at a higher level, we'll covered (briefly) how to pass information from the server (Flask) to the interface (JavaScript → HTML)

Flask

- A Python-based "microframework" for web development
 - "Micro" in the sense that it tries to provide only the core functionality for a web server, while allowing you to add other components as needed (e.g., a database)
- We will assume Python 3.X in our work (though Flask is compatible with 2.X)
- You've (hopefully!) gotten this installed already as part of Assignment 0

Flask

- Server application
 - A minimal server application might look like this:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Flask

- Server application
 - To run the server...
 - Flask docs mention using the `FLASK_APP` shell variable in conjunction with the `flask` shell command
 - We will use an alternative approach:
 - Add code to start the server in the application...

```
if __name__ == "__main__":
    app.run(host='127.0.0.1', port=8080, debug=True)
```

- ... and execute your Python file in the usual way
- ```
> python3 run.py
```
- Now point your browser to <http://127.0.0.1:8080> and voila!

# Flask

- Server application
  - Note that we set "debug=True" in our server
  - This conveniently gives us trace information for errors...

A screenshot of a web browser window displaying an error page. The title bar says "TypeError: must be str, not NoneType". The address bar shows "localhost:5000". The main content area has a red background with white text. It says "TypeError" and "TypeError: cannot concatenate 'str' and 'NoneType' objects". Below that is a blue header bar with the text "Traceback (most recent call last)". The stack trace is listed in grey boxes:

```
File "/Users/mitsuhiko/Development/flask/flask.py", line 650, in __call__
 return self.wsgi_app(environ, start_response)
File "/Users/mitsuhiko/Development/werkzeug-main/werkzeug/wsgi.py", line 406, in
__call__
 return self.app(environ, start_response)
File "/Users/mitsuhiko/Development/flask/flask.py", line 616, in wsgi_app
 rv = self.dispatch_request()
File "/Users/mitsuhiko/Development/flask/flask.py", line 535, in dispatch_request
 return self.view_functions[endpoint](**values)
File "/Users/mitsuhiko/Development/flask/test.py", line 8, in index
 return 'Hello ' + name
```

... however (!!)

Attention  
The interactive debugger ...  
allows the execution of  
arbitrary code. This makes it  
a major security risk and  
therefore it **must never be**  
**used on production**  
**machines.**

# Flask

---

- Routing
  - Typically you would use different code to render different URLs for your site
  - Flask allows for **routing** decorators that specify the URL

```
@app.route('/')
def index():
 return 'Index Page'

@app.route('/hello')
def hello():
 return 'Hello, World'
```

# Flask

---

## ■ Static files

- Web sites normally have simple static files that you just want to serve to the front end (e.g., CSS, JS, images)
- Rather than do any fancy URL routing, Flask allows a simple addition to the app specification:

```
app = Flask(__name__, static_folder='public', static_url_path='')
```

- Put your files in **/public/<path>** ,  
Use the URL **/<path>**
  - E.g., a file at /public/img/photo.jpg  
can be loaded at http://127.0.0.1:8080/img/photo.jpg

# Flask

---

- Templates
  - Basically HTML, but include both static HTML *and* special template syntax for dynamic content
  - Provide a way to re-use HTML/code consistently across a web site (or parts of one)
  - Provide a way to use back-end data (e.g., a database) to build web pages dynamically
  - There are various options out there for template syntax...  
Flask uses a framework called Jinja for its syntax
    - Special markers "% ... %" and "{{ ... }}" have special meaning
    - Other systems like Django, Express.js modules use the same idea

# Flask

---

- Templates
  - One common use case: base + page HTML/template
  - Base template:

```
<html>
<head>
 <title>My Site - {% block title %}{% endblock %}</title>
 ...
</head>
<body>
 <nav>
 ...
 </nav>
 <section class="content">
 {% block content %}{% endblock %}
 </section>
</body>
</html>
```

# Flask

---

## ■ Templates

- One common use case: base + page HTML/template
- Page template that uses the base template:

```
{% extends 'base.html' %}

{% block title %}
 About Us
{% endblock %}

{% block content %}
 <h1>About Us</h1>
 <p>...</p>
{% endblock %}
```

- And then to render the template in your routing function:

```
@app.route('/')
def index():
 return render_template('index.html')
```

# Flask

---

- Templates

- Another common use case: passing data to a template
- Routing function passes a variable as a keyword arg...

```
@app.route('/')
def index():
 return render_template('index.html', message="Hello World!")
```

- ... and the template uses the variable with "{{ ... }}"

```
<h1>{{ message }}</h1>
```

- There are lots of extra features, like auto-escaping, filtering, formatting, etc.

# Flask

---

- Templates
  - Loops can iterate over lists of data elements:

```
@app.route('/about')
def about():
 data = {
 'related': [
 {'name': 'Rent A Goat', 'url': 'http://rentagoat.com'},
 {'name': 'We Rent Goats', 'url': 'http://werentgoats.com'},
 {'name': 'Goat Yoga', 'url': 'https://goatyoga.net/'}
]
 }
 return render_template('about.html', data=data)
```

```

 {% for site in data.related %}
 {{ site.name }}
 {% endfor %}

```

# Flask

---

- This covers communication in one direction
  - When the user does something, we usually need to...
    - Store this action somehow in a back-end database
    - Reflect the change in the
  - How does this info get to the server?

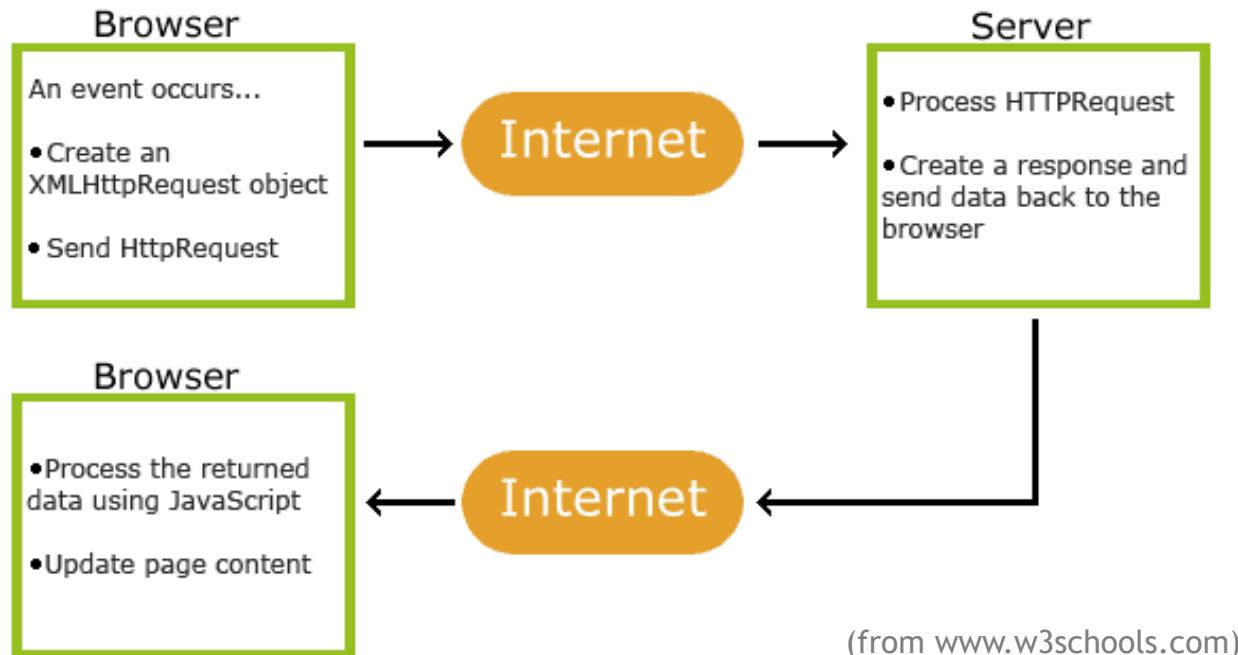
# AJAX

---

- AJAX = Asynchronous JavaScript And XML
- The acronym is, unfortunately, tied to two specific languages, but that's not the important part
  - In fact, many applications don't even use XML (they use JSON instead)
- What's important is the technique of passing information dynamically, front <-> back end
- The user stays on the same page the whole time; AJAX + JavaScript handle all the necessary changes to the page

# AJAX

- One view of the AJAX process:



# Browser: jQuery AJAX

---

- The "pure" JavaScript involves creating and passing HTTP objects directly
- You could do this — but more common today is to use a convenient higher-level API
- jQuery actually provides a number of methods to send data to the server and then receive a response
  - Not at page load! (e.g., via templates)
  - But rather, after the page has loaded, get more data
  - This is typical for many web pages today
    - Like when you see that "Loading..." message for a table/etc.

# Browser: jQuery AJAX

---

- (One way to see how much JavaScript/AJAX action in a web site: turn off JavaScript and load it...)
  - <http://www.drexel.edu>
  - <http://www.philly.com>

# Browser: jQuery AJAX

---

- **jQuery get method**

```
$.get(URL, data, function(data), dataType)
```

- URL (required): where to ask for information
- data (optional): any data to pass to the server, if needed
- function (optional): fn to run if/when server call succeeds
- dataType (optional): “json”, “xml”, etc. (auto by default)

- Typical pattern:

```
$.get("/api/test", {clicked: "yes"}, function(data) {
 alert("Data from server: " + data);
});
```

# Browser <-> Server Data

---

- The data sent between browser and server can be "packaged" using different formats
- Arguably the most common format today is JSON
- JSON = JavaScript Object Notation, which includes:
  - String, number, boolean, null
  - Arrays
  - JSON Object (recursively) with keys and values

```
{ "employees": [
 { "firstName": "John", "lastName": "Doe" },
 { "firstName": "Anna", "lastName": "Smith" },
 { "firstName": "Peter", "lastName": "Jones" }
]}
```

# Server: AJAX Handler

---

- When the browser makes a request to a server URL (e.g., "/api/test"), the server responds with data
- Recall that Flask provides our server, and has routing specifications for different URLs
- Let's assume that:
  - All AJAX calls are served by URLs that start with "/api/"
  - The server responds to all AJAX calls with JSON

# Server: AJAX Handler

---

- Flask handles an AJAX call like any other, except that it should return JSON (instead of HTML)
- We can simply add to our list of routing functions:

```
@app.route('/')
def index():
 return 'Index Page'

@app.route('/hello')
def hello():
 return 'Hello, World'

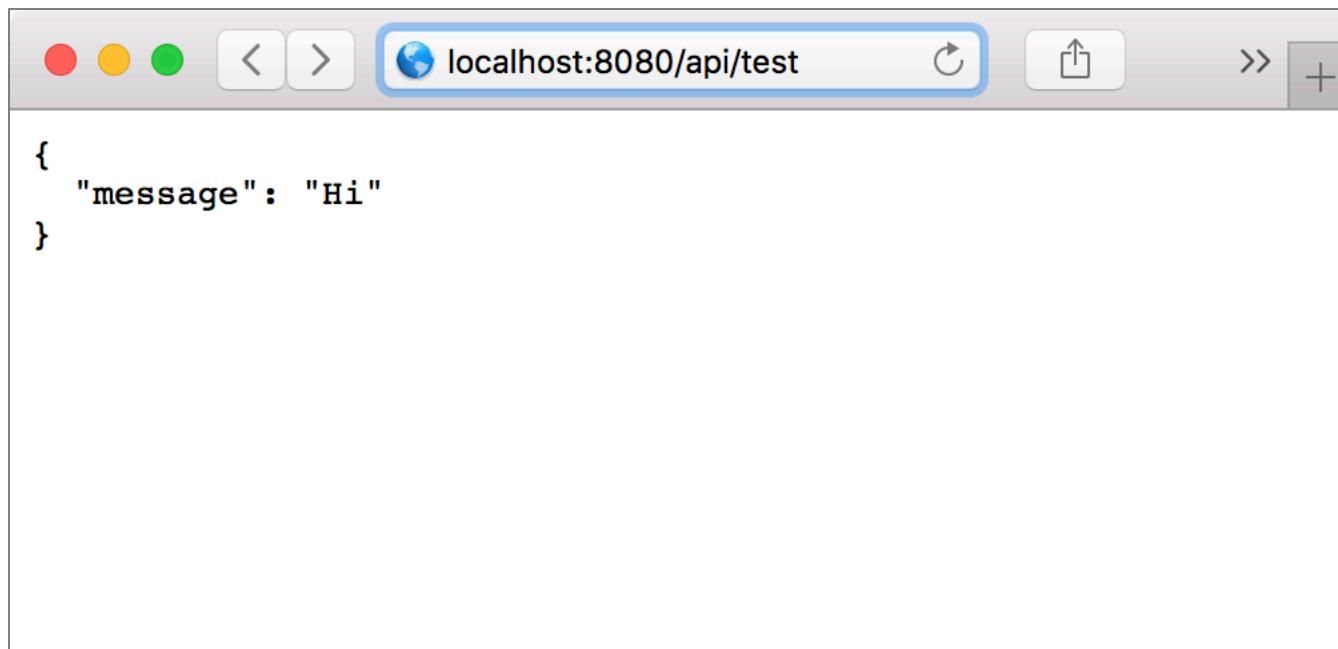
@app.route('/api/test')
def api_test():
 return jsonify({'message': 'Hi'})
```

jsonify() packages the data into JSON

# Server: AJAX Handler

---

- To test the handler, you can point your browser to the AJAX URL and see what it returns!



# Server: AJAX Handler

- When data is passed to the server, you'll need to extract the information from the **request** object
  - jQuery call on the browser:

```
$.get("/api/test", {clicked: "yes"}, function(data) {
 alert("Data from server: " + data);
});
```

- Flask handler on the server:

```
@app.route('/api/test')
def api_test():
 clicked = request.args.get('clicked', default='no')
 return jsonify({'message': 'User clicked ' + clicked})
```

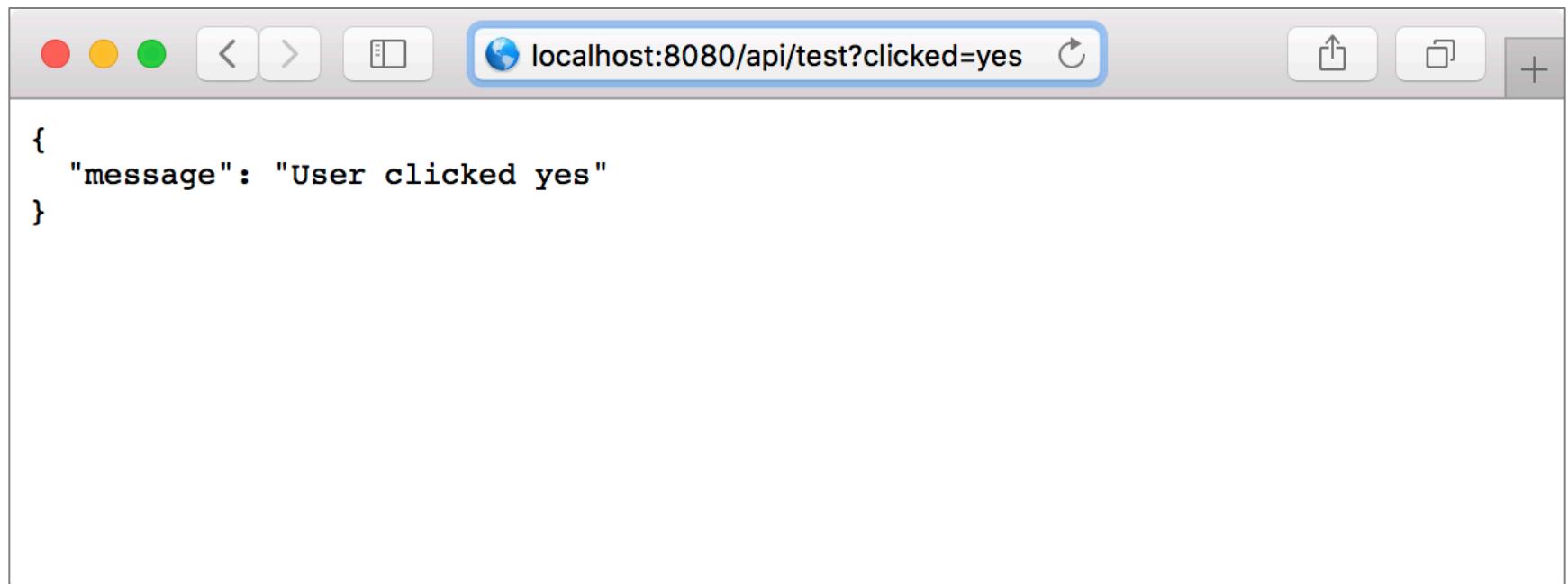
Gets the named argument from the request object

Default applies if argument is not supplied

# Server: AJAX Handler

---

- Again, you can check the handler in your browser
- Arguments must be specified in URL format



# Browser: jQuery AJAX

---

- Ok, now back to the browser...
- The data from the server is returned to the success function supplied to the **get** method:

```
$.get("/api/test", {clicked: "yes"}, function(data) {
 alert("Data from server: " + data);
});
```

- Here, we're just showing the data in an alert
- But normally, we would use the data to build relevant HTML within the web page
  - That is, we use jQuery to construct DOM elements and populate the page with the new elements

# Example #1

---

- Let's return to our Goat Pasture web site and build a dynamic, AJAX-y "Adopt a Goat" page
- We'll provide a list of goats for adoption, and the user can click one to adopt it
- Importantly, the user's actions need to be reflected in a central database on the back-end server
  - We'd like a simple database that interfaces easily with Python but has plenty of powerful features
  - SQLite gives us more than enough features for this course
  - Of course, you could substitute a more powerful DB

# Example #1

---

- SQLite table

```
sqlite3 goats.db

CREATE TABLE goats (
 uid INTEGER PRIMARY KEY,
 name TEXT NOT NULL,
 age INTEGER,
 img TEXT
);

.mode csv
.import dev/data.csv goats
```

- Data

```
1,Billy,3,goat1.jpg
2,Doc,5,goat2.jpg
3,Sarah,4,goat3.jpg
4,Kid,2,goat4.jpg
5,Buffy,3,goat5.jpg
6,Smiles,4,goat6.jpg
7,Einstein,7,goat7.jpg
8,Frank Beard,6,goat8.jpg
9,LuLu,1,goat9.jpg
10,Pegasus,4,goat10.jpg
11,Zeus,12,goat11.jpg
12,Willow,2,goat12.jpg
13,Big Al,7,goat13.jpg
```

# Example #1

---

- Python Database class

```
class Database:

 def __init__(self):
 self.conn = sqlite3.connect(SQLITE_PATH)

 def execute(self, sql, parameters=[]):
 c = self.conn.cursor()
 c.execute(sql, parameters)
 return c.fetchall()

 def get_goats(self, n, offset):
 data = self.execute(
 'SELECT * FROM goats ORDER BY uid ASC LIMIT ? OFFSET ?',
 [n, offset])
 return [
 {
 'uid': d[0],
 'name': d[1],
 'age': d[2],
 'img': d[3]
 } for d in data]

 def close(self):
 self.conn.close()
```

# Example #1

---

- Python Flask database + routing code

```
def get_db():
 db = getattr(g, '_database', None)
 if db is None:
 db = Database()
 return db

@app.teardown_appcontext
def close_connection(exception):
 db = getattr(g, '_database', None)
 if db is not None:
 db.close()

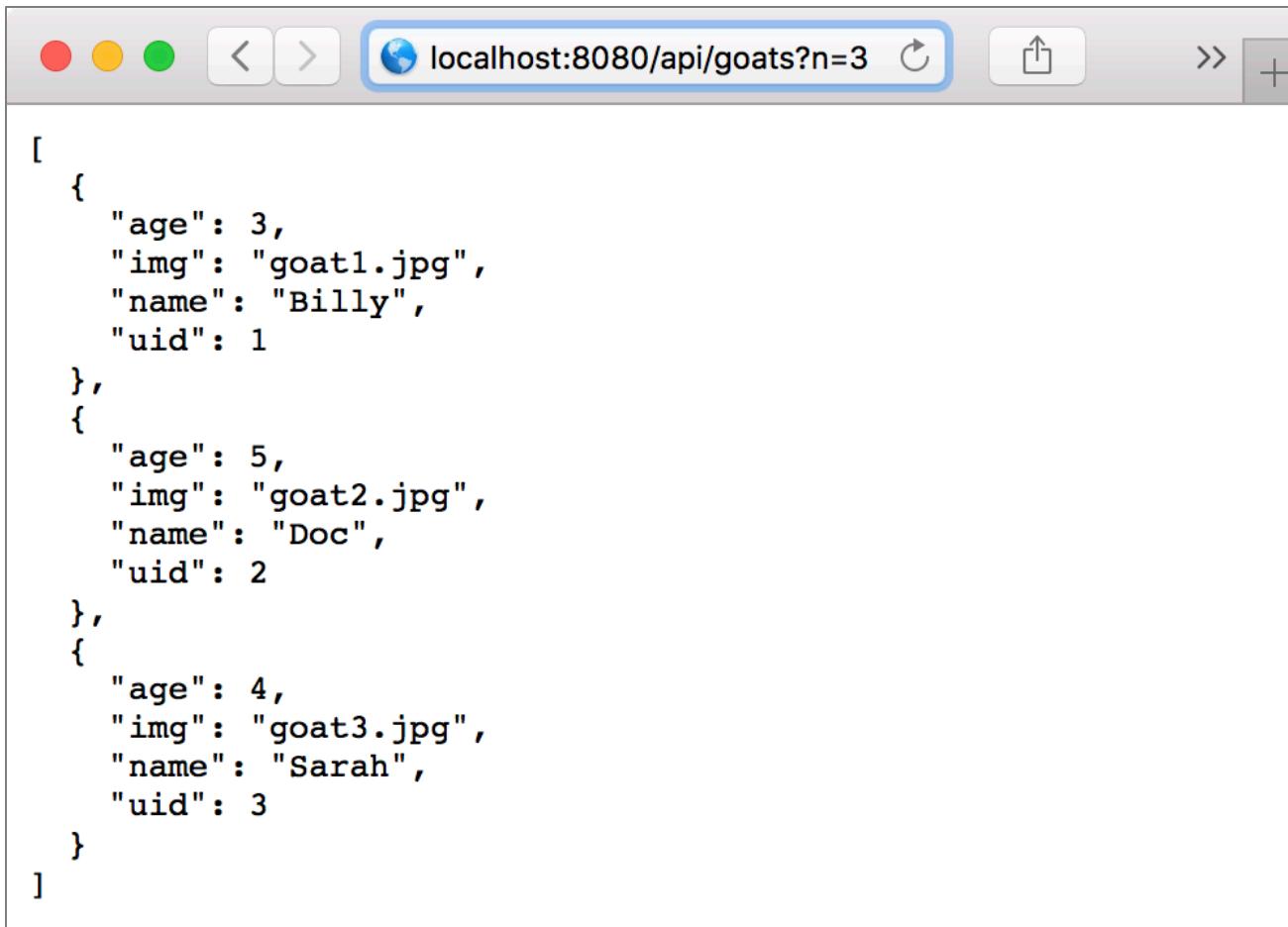
@app.route('/api/goats')
def api_goats():
 n = request.args.get('n', default=6)
 offset = request.args.get('offset', default=0)
 goats = get_db().get_goats(n, offset)
 return jsonify(goats)
```

- Note that database must be stored in global **g** variable

# Example #1

---

- Testing the API handler



A screenshot of a web browser window. The address bar shows the URL `localhost:8080/api/goats?n=3`. The page content displays a JSON array of three goat objects:

```
[
 {
 "age": 3,
 "img": "goat1.jpg",
 "name": "Billy",
 "uid": 1
 },
 {
 "age": 5,
 "img": "goat2.jpg",
 "name": "Doc",
 "uid": 2
 },
 {
 "age": 4,
 "img": "goat3.jpg",
 "name": "Sarah",
 "uid": 3
 }
]
```

# Example #1

---

- HTML

```
<h1>Adopt a Goat</h1>

<p>
 Please see our awesome goats below and
 adopt one today!
</p>

<div id="cards" class="container">

</div>
```

# Example #1

---

## ■ JavaScript

```
let goatsPerColumn = 3;
var goatsPerPage = 3;

// updateCards() on next slide

$(function () {
 $.get("api/goats", { n: goatsPerPage, offset: 0 }, function (goats) {
 updateCards(goats);
 });
});
```

# Example #1

---

## ■ JavaScript

```
function updateCards(goats) {
 for (var i = 0; i < goats.length; i++) {
 let goat = goats[i];
 let card = $('

').addClass('card')
 .append(
 $('').addClass('card-img-top').attr('src', '/img/goats/' + goat.img)
).append(
 $('

').addClass('card-body')
 .append(
 $('

').addClass('card-title').text(goat.name)).append($(' ').addClass('card-text').text(goat.age + ' years old')).append($('').addClass('btn btn-primary').text('Adopt'))); if (i % goatsPerColumn == 0) $('#cards').append($(' ').addClass('card-deck')); $('#cards .card-deck').last().append(card); } }


```

# Example #1

Home    About Us    Goat School ▾    Adopt a Goat    Account ▾

## Adopt a Goat

Please see our awesome goats below and adopt one today!



Billy  
3 years old  
[Adopt](#)



Doc  
5 years old  
[Adopt](#)



Sarah  
4 years old  
[Adopt](#)

# Example #1

---

- Ok, not bad.
- We're dynamically receiving the first 3 goats from the server and building the card deck
- However...
  - When we click "Adopt", nothing happens!  
How can we alter the page to reflect the change?  
More importantly, how can we tell the server?!?
  - We can only see 3 goats!  
What about the next 3? and the next 3? and ...

# Example #2

---

- Let's handle the "Adopt" button and send a notification of adoption to the server
- JavaScript: outline

```
const goatsPerColumn = 3;
const goatsPerPage = 6;

function buildCards(goats) {
 ...
}

function loadGoats() {
 ...
}

function adoptGoat(uid, value) {
 ...
}

$(function () {
 loadGoats();
});
```

```
function buildCards(goats) {
 ...
 $('<a>').addClass('btn btn-primary').text('Adopt')
 .click(function () {
 goat.adopted = goat.adopted ? 0 : 1;
 adoptGoat(goat.uid, goat.adopted);
 })
 ...
 if (goat.adopted)
 card.addClass('adopted');
 ...
}

function loadGoats() {
 $.get("api/goats", {
 n: goatsPerPage,
 offset: 0
 }, function (goats) {
 buildCards(goats);
 });
}

function adoptGoat(uid, value) {
 $.get("api/adopt", {
 uid: uid,
 value: value,
 n: goatsPerPage,
 offset: 0
 }, function (goats) {
 buildCards(goats);
 });
}
```

# Example #2

---

- CSS

```
.card.adopted {
 opacity: .25;
}
```

# Example #2

---

- Python Database class

```
class Database:
 ...

 def select(self, sql, parameters=[]):
 c = self.conn.cursor()
 c.execute(sql, parameters)
 return c.fetchall()

 def execute(self, sql, parameters=[]):
 c = self.conn.cursor()
 c.execute(sql, parameters)
 self.conn.commit()

 def get_goats(self, n, offset):
 data = self.select(
 'SELECT * FROM goats ORDER BY uid ASC LIMIT ? OFFSET ?', [n, offset])
 return [{
 'uid': d[0], 'name': d[1], 'age': d[2], 'adopted': d[3], 'image': d[4]
 } for d in data]

 def update_goat_adopted(self, uid, value):
 self.execute('UPDATE goats SET adopted=? WHERE uid=?', [value, uid])

 ...
```

# Example #2

---

- Python Flask routing code

```
@app.route('/api/goats')
def api_goats():
 n = request.args.get('n', default=6)
 offset = request.args.get('offset', default=0)
 goats = get_db().get_goats(n, offset)
 return jsonify(goats)

@app.route('/api/adopt')
def api_adopt():
 uid = request.args.get('uid')
 value = request.args.get('value')
 get_db().update_goat_adopted(uid, value)
 return api_goats()
```

- Note that "/api/adopt" also returns the list of goats, which avoids making a second AJAX call

# Example #2

Home    About Us    Goat School ▾    Adopt a Goat    Account ▾

## Adopt a Goat

Please see our awesome goats below and adopt one today!



**Billy**  
3 years old  
[Adopt](#)



**Doc**  
5 years old  
[Adopt](#)



**Sarah**  
4 years old  
[Adopt](#)

## Example #2

---

- Summary: When the user clicks "Adopt"...
  - The browser sends a request to the server with the goat's **uid** and **adoption** value
  - The server updates the database with the adoption value
  - The server re-pulls the updated data from the database and returns this as a response
  - The browser takes this response and re-populates the cards with the new data
- All the user sees is a grayed-out card!
  - Of course, we could have done this easily with JavaScript
  - But the above method ensures that all changes are reflected in the database, which is essential for a real site

## Example #3

---

- Unfortunately, we can still only see 3 goats at once
- Let's add **pagination** to our example
  - Pagination is ubiquitous in real web sites
  - For our example, it wouldn't be a big deal to send all the data and have JavaScript deal with it (just 13 goats!)
  - But on a real site, you might literally have millions/billions of records (e.g., Google)...  
so you need to ask the server to give you a "page" of data

# Example #3

---

- Bootstrap provides nice HTML/CSS for pagination



```
<nav aria-label="Page navigation example">
 <ul class="pagination">
 <li class="page-item">Previous
 <li class="page-item">1
 <li class="page-item">2
 <li class="page-item">3
 <li class="page-item">Next

</nav>
```

- But handling the actual pagination via JavaScript is left to the programmer (since it's data-specific)

# Example #3

---

- HTML

```
<div id="cards" class="container">

 </div>

 <nav>
 <ul id="paginator" class="pagination justify-content-center">

 </nav>
```

# Example #3

---

## ■ JavaScript

```
function createPageLink(text, toPage, active, disabled) {
 let link = $('').addClass('page-item').append(
 $('<a>').addClass('page-link').text(text).click(function () {
 currentPage = toPage;
 loadGoats();
 })
);
 if (active)
 link.addClass('active');
 if (disabled)
 link.addClass('disabled');
 return link;
}

function updatePagination(total) {
 let pages = Math.ceil(total / goatsPerPage);
 $('#paginator').empty().append(
 createPageLink('Previous', currentPage - 1, false, currentPage == 1)
);
 for (var page = 1; page <= pages; page++)
 $('#paginator').append(
 createPageLink(page, page, page == currentPage, false)
);
 $('#paginator').append(
 createPageLink('Next', currentPage + 1, false, currentPage == pages)
);
}
```

# Example #3

---

- Python Database class additions

```
def get_total_goat_count(self):
 data = self.select('SELECT COUNT(*) FROM goats')
 return data[0][0]
```

- Python Flask routing code changes

```
@app.route('/api/goats')
def api_goats():
 n = request.args.get('n', default=3)
 offset = request.args.get('offset', default=0)
 response = {
 'goats': get_db().get_goats(n, offset),
 'total': get_db().get_total_goat_count()
 }
 return jsonify(response)
```

# Example #3

Home   About Us   Goat School ▾   Adopt a Goat   Account ▾

## Adopt a Goat

Please see our awesome goats below and adopt one today!

Billy  
3 years old

[Adopt](#)

Doc  
5 years old

[Adopt](#)

Sarah  
4 years old

[Adopt](#)

Previous 1 2 3 4 5 Next

Home   About Us   Goat School ▾   Adopt a Goat   Account ▾

## Adopt a Goat

Please see our awesome goats below and adopt one today!

Kid  
2 years old

[Adopt](#)

Buffy  
3 years old

[Adopt](#)

Smiles  
4 years old

[Adopt](#)

Previous 1 2 3 4 5 Next

# Final Note

---

- We've assumed here that we can use a GET request for all AJAX communication
- Normally, though, a GET should only be used for a "safe" request that doesn't make changes
  - E.g., we really shouldn't use it for the "adopt" function
- For any request that changes things, use POST
  - Data doesn't appear in the URL
  - Doesn't have size restriction that GET does
  - Avoids a web crawler (like google) crawling to your web site and submitting who-knows-what!

# That's It

---

- Our intro to the various technologies...
  - HTML, CSS, JavaScript, jQuery, Bootstrap, Flask
- ... gave you the basics of the front-end interface, but only a taste of the back-end server — only what was need to serve pages
- Now, we've seen a much more realistic scenario, with data being passed dynamically between the browser and the server
  - The basic ideas here can take you a long way!