In the beginning, I suspect this was new to many of you... But we have now implemented this kind of interactivity and much more. In fact...

```html
<div class="blue-slider">
  <div class="color-label">blue: </div>
  <input type="range" id="slider-b" />
  <span id="value-b"></span>
</div>

Red: 110
Green: 54
Blue: 162

// Set oninput callback for each slider
sliders.forEach((slider) =>
  slider.addEventListener("input", update));

const update = function() {
  colorBox.style.backgroundColor = `rgb(${sliders[0].value}, ${sliders[1].value}, ${sliders[2].value})`;
  sliders.forEach((slider, index) =>
    labels[index].innerHTML = slider.value);
};
```
Over the semester, we have learned about, used and often implemented components in every one of these boxes, from the JavaScript running on the front-end (i.e. the client) to the routes and models in the server.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Along the way we have gained familiarity with JavaScript (perhaps a new language) and is emphasis on closures and callbacks. The idea of functions as values (functions as 1st class objects), and asynchronous execution hopefully now feels familiar.

```javascript
const wrapValue = (n) => { // function(n) {
    let local = n;
    return () => local; // function () { return local; }
}

let wrap1 = wrapValue(1); // () => 1
let wrap2 = wrapValue(2); // () => 2
console.log(wrap1()); // What will print here?
console.log(wrap2()); // What will print here?
```
We have made extensive use of React, and some its key principles:
- Maintain a single source of truth
- Props "flow" down
- Callbacks "flow" up

React implements a design pattern for highly interactive UIs. Your responsibility as the programmer is to define what you want rendered on the screen for a given state of the application and how you want to update that state in response to user actions and React fills in the "last piece" of the cycle by efficiently re-rendering the UI as the state changes.
We learned about writing backend servers using Node to create RESTful APIs. RESTful APIs are built around the idea of actions on resources, i.e. the film resource in the film explorer example. Are routes are typically queries to our persistence layer, in this example to a RDBMS via the Objection.js ORM.

<table>
<thead>
<tr>
<th>Route</th>
<th>Controller Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /api/films</td>
<td>Create new movie from request data</td>
</tr>
<tr>
<td>GET /api/films/:id</td>
<td>Read data of movie with id == :id</td>
</tr>
<tr>
<td>PUT /api/films/:id</td>
<td>Update movie with id == :id from request data</td>
</tr>
<tr>
<td>DELETE /api/films/:id</td>
<td>Delete movie with id == :id</td>
</tr>
<tr>
<td>GET /api/films</td>
<td>List (read) all movies</td>
</tr>
</tbody>
</table>

```javascript
app.get('/api/films', (request, response, next) => {
  return Film.query()
    .withGraphFetched('genres')
    .then((movies) => {
      response.send(movies);
    }, next);
});
```
We learned about two different approaches to data persistence on the backend in the form of relational databases and so-called NoSQL, but more precisely non-relational, databases.

<table>
<thead>
<tr>
<th></th>
<th>Relational (RDBMS)</th>
<th>Non-Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td>Table-oriented</td>
<td>Document-oriented, key-value, graph-based, column-oriented, ...</td>
</tr>
<tr>
<td><strong>Schema</strong></td>
<td>Fixed schema</td>
<td>Dynamic schema</td>
</tr>
<tr>
<td><strong>Joins</strong></td>
<td>Used extensively</td>
<td>Used infrequently</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td>SQL</td>
<td>Custom query language</td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td>ACID</td>
<td>CAP</td>
</tr>
</tbody>
</table>

```
SELECT * FROM people
WHERE age > 25;
```

```
db.people.find(
   { age: { $gt: 25 } }
)
```
We learned about test driven development (TDD) and concepts like unit testing and integration testing. We also talked about the use of behavior driven development (BDD) to move us from user story to scenario to test to implementation.
We learned about using git to manage development, and the value of continuous integration (CI). CI rigorously tests every integration in production-like environment, the motivation is to:
• Prevent development-production mismatch
• Test multiple browsers, etc.
• “Stress test” code for performance, fault-tolerance, etc.
CI is part of a larger DevOps approach.

Recall the DevOps principles:
• Involve operations in each phase of a system’s design and development,
• Heavy reliance on automation versus human effort,
• The application of engineering practices and tools to operations tasks

This manifested for us in our use of Travis-CI to test our builds, and automated preparation (e.g. running migrations/seeding) and ultimately deployment with Heroku.

https://www.atlassian.com/git/tutorials/using-branches
You also learned about ways to approach design from user stories to CRC cards to lo-fi prototypes. Our goal is to be able to quickly iterate on our design quickly and cheaply. These tools are intended to facilitate conversations with our stakeholders, e.g. customers.
Agile development processes, Scrum in particular, played an important role for us. In Scrum the short sprints provide frequent opportunities to update our approach in response to what we have learned about the problem or the application. Recall the key idea of lower-case "a" agility:

1. Find out where you are,
2. Take a small step towards your goal,
3. Adjust your understanding based on what you learned, and
4. Repeat

And when faced with two or more alternatives that deliver roughly the same value, choose the path that makes future change easier

Adapted from Mountain Goat Software

Adapted from Dave Thomas (https://www.youtube.com/watch?v=a-BOSpxYJ9M)
And we thought about how to approach building our own "Mona Lisa"s. Recall that an incremental approach calls for building a fully formed idea a bit at a time, and thus requires having a fully formed idea. In contrast, iterating allows you to move from vague idea to realization. The catch is we have to address the entire scope at one time, i.e. we are working on the entire image (a risky approach). Thus we sought the best of both where in each sprint we both add new features (incremental) and refine existing functionality (iterative) with strong focus on the highest priority features (the "face").
Over the course of the semester, you have done four assignments and 10 practicals yielding a complex Wikipedia-like platform with database persistence and user authentication.

Between these and the project, we have created 394 repositories on GitHub. On the just the master branches of your projects have made 594 commits (I suspect there are many more across all the branches) and done 860 Travis builds (when I looked last night)!
And the 40 different acronyms.... What I hope these (and especially those on the left) will help you with is remember the keys ideas when writing a test, or a user story or debugging.

**SMALL (So Many Acronyms Littering the Lectures)**

- F.I.R.S.T.
- S.M.A.R.T.
- R.A.S.P.
- DRY
- SoC
- SOFA
- SaaS
- TDD, BDD
- MVC
- WISBNWIIW
- ACID
- BASE
- CRUD
- HTML / DOM / CSS / JSX
- CRA
- CI / CD
- UI
- AJAX
- REST API
- URI / URL
- TCP/IP
- JSON
- UML
- CRC
- ORM
- POJO
- SQL / RDMs
- VCS
- SLO / SLA
Take-aways

• Behind every design decision there should be a user story
• Testing, not just a class requirement, it’s a good idea
• Develop iteratively and incrementally
• There should be one source of truth
• Don’t repeat yourself
• Don’t mutate state
• Do read error messages
• Automate all of the things
• Don’t break the build
• Avoid code smells
Write beautiful code, do the Right Thing and bump the lamp (or give the Statue of Liberty hair!)