

44-563: Unit 10

Developing Web Applications and Services

Includes

- Project
- Design Patterns
- MVC
- Models
- Controllers
- Views
- Exam 2 next Wed
- Client next Friday

Project

Project Objectives

- Apply the concepts learned in the course
- Apply software engineering principles:
 - Separation of concerns, loosely-coupled design, following conventions (very important!)
- Practice collaborative coding
- Provide a useful, working application
- Gain experience with some widely-used patterns in software engineering
- Practice deployment with Heroku.

Design Patterns

Design patterns

- Offer standard solutions to common **software engineering** problems. They are not specified at the code-level, but rather provide a general prescription/template for how to solve a problem.
- Some are very well-known and widely-used
- Patterns help create maintainable & extensible code
 - **Loosely-coupled.** Components are separate from each other. Not intertwined. Easy to update one without breaking many.
 - **Cohesive.** Each component does one thing, and everything it does is *about that one thing*. Two components don't get all up in each other's business.

Design pattern examples

- Chain of responsibility (e.g., in nested views in a GUI)
- Publish/subscribe (e.g., IoT)
- Singleton (one instance of a class, ever)
- Object pool (recycle objects that are no longer needed)
- Lock (to prevent multiple threads from accessing 1 resource)

MVC Pattern

MVC Popularity

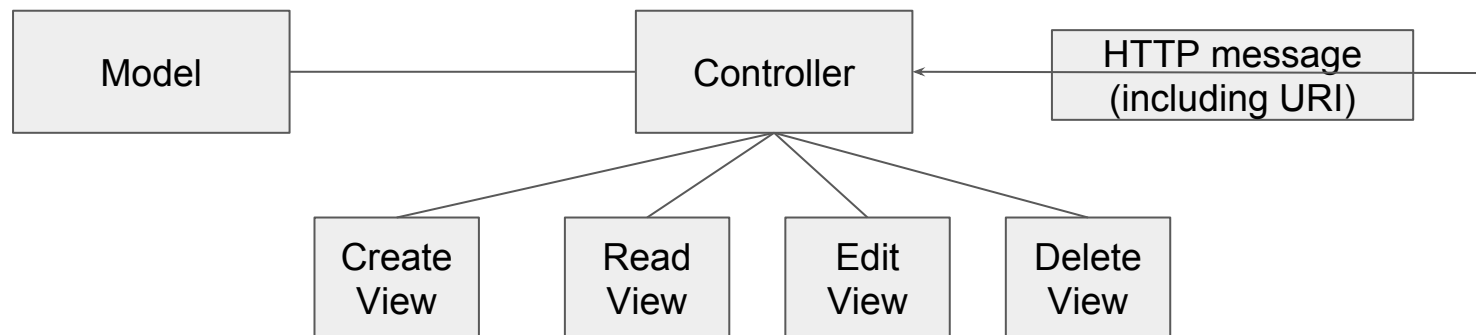
- Nearly all major server-side platforms offer MVC approaches (Node, Java, C#, Ruby, PHP)
- Some variations exist, so instead of MVC, you may see **MV***.
- Client-side has MVC too. Angular 2 is a popular client-side framework that uses a component-based MVC approach.

MVC Introduction

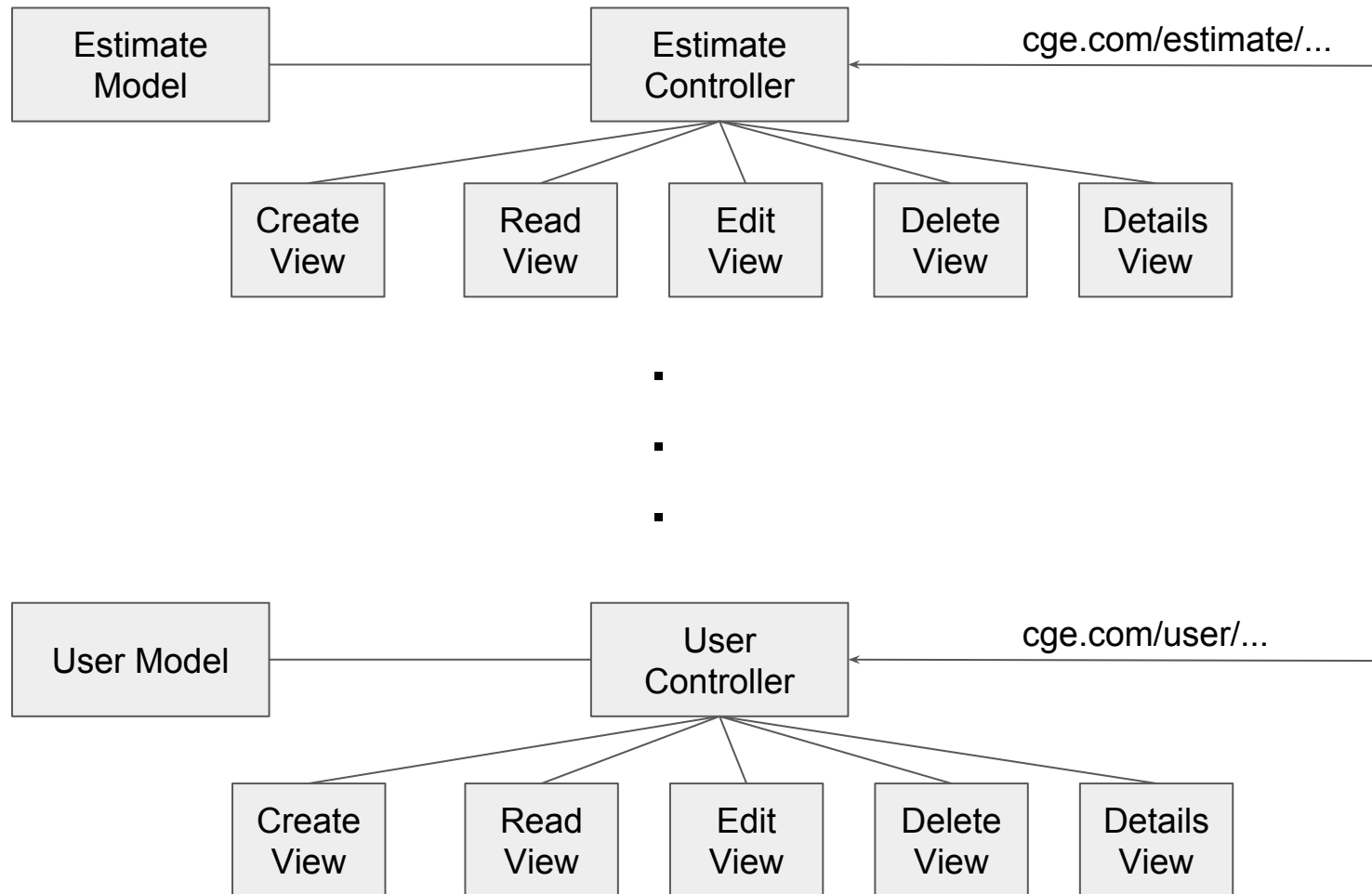
- The short version of our MVC:
 - **Models** are dumb objects representing a resource.
 - **Views** are dumb code-enhanced, dynamically-generated HTML for interacting with the user.
 - **Controllers** do the work. They listen for requests and provide appropriate (we hope!) responses. This doesn't always happen until we get everything working right. :)
- Next, we'll look at each of these individually.

MVC Example

- We use a very popular pattern on the server-side.
- We specify each resource with a **model**.
- Requests (HTTP method + URI) are handled by a **controller**. Typically, one controller for each model.
- We have a set of CRUD **views** for each model.
- This pattern is called **Model-View-Controller** or **MVC**.



MVC Project, Illustrated



Models

Models

- Most web apps **begin with defining models**
- Models describe domain-specific **concepts**
- One model per concept (e.g. product, estimate).
- So standard (and time-consuming) that **scaffolding** apps have been built to help (Stongloop/Loopback-type features are available in nearly every major server platform)
 - Loopback, Sails, Trails provide scaffolding for Node apps
 - Visual Studio provides scaffolding for .NET apps (C#)
 - Grails and Spring provide scaffolding for Java/Groovy apps

Model schemas

- Like DTDs for XML we may want to specify conditions for our models.
 - **Mongoose*** schemas allow us to define models in detail.
 - One schema per model.
 - Specify for each model property:
 - data type, required or optional, default value, list of possible values, min value, max value, etc.
- ❖ Each Mongoose schema serves as our **detailed specification** (use this information to configure web components and field properties in the views).

*a NodeJS module that acts as an Object Document mapper: it translates MongoDB documents into JavaScript objects.
See [this document](#) for an overview.

Model schema example

```
var person = new mongoose.Schema(  
  {  
    name: String,  
    living: Boolean,  
    updated: { type: Date, default: Date.now },  
    age: { type: Number, min: 18, max: 65, required: true },  
  })
```


Project seed data

- During development, it is often helpful to provide sample data, often called "seed data".
- One sample data file per model.
- By convention, these files are kept in the **data** folder.
- Our sample data is in JSON.
- Each time our app starts, it will load this seed data into the application.
- You may add, edit, delete as you like while developing the application, because the app will be re-seeded with the same sample data with each new run.

Persistence

- During development, we're using a temporary, in-memory database. It lasts only as long as our application is running.
- When our app is complete and working correctly, we will switch from our in-memory database to persisting our data in JSON or MongoDB.
- MongoDB is a popular no-SQL (not only SQL) data management system that stores data as Binary JSON documents (aka BSON).

Controllers

Controllers

- Controllers provide the control flow logic and business logic.
- They can be the more interesting (i.e. challenging) parts of the application (along with the views).
- There is one controller for each model.
- The controller does what we did with Loopback (and more!)
- One controller provides a RESTful API to one resource model.

Controllers & routing

- First, set the default URI path for the controller resource in the main app, e.g. `/estimate`
- The controller handles the *remaining* URI.
- There are different request handlers for each **HTTP method + URI combination**.
- Controllers can respond in many ways depending on the request.

Respond with JSON

What does the user want?

1. GET <http://127.0.0.1:8081/estimate/findall>
2. GET <http://127.0.0.1:8081/estimate/findone/5>

What are they interested in?

Could we return this in JSON?

How many values will be returned?

Ex: Get all JSON

- GET <http://127.0.0.1:8081/estimate/findall>
- Request has access to app.locals & app.locals has estimates.
- estimates has data and meta information.
- To access just the collection (not the complex object), use .query.
- Set the response type to JSON and JSON.stringify our collection array

```
1  const express = require('express')
2  const api = express.Router()
3  const Model = require('../models/estimate.js')
4
5  // Handle request for all estimate JSON.....
6  api.get('/findall', (req, res) => {
7    res.setHeader('Content-Type', 'application/json')
8    const data = req.app.locals.estimates.query
9    res.send(JSON.stringify(data))
10  })
```

Ex: Get JSON by id

- GET <http://127.0.0.1:8081/estimate/findone/1>
- The request has the id as a string
- **parseInt** to get integer, 2nd arg wants to know the base
- Use [lodash](#) **find** to get the desired item

```
api.get('/findone/:id', (req, res) => {  
  res.setHeader('Content-Type', 'application/json')  
  const id = parseInt(req.params.id, 10) // base 10  
  const data = req.app.locals.estimate.query  
  const item = find(data, { _id: id })  
  if (!item) { return res.end(notfoundstring) }  
  res.send(JSON.stringify(item))  
})
```


Respond with a view

What does the user want?

1. GET <http://127.0.0.1:8081/estimates/>
2. GET <http://127.0.0.1:8081/estimates/create>
3. GET <http://127.0.0.1:8081/estimates/delete/id>
4. GET <http://127.0.0.1:8081/estimates/details/id>
5. GET <http://127.0.0.1:8081/estimates/edit/id>

The controller can send back a view to be rendered in the browser.

Every model will have 5 standard views.

Respond with a view

By handling the path to *estimates* separately, our approach is reusable across products.

Given the following 5 typical view requests handled by each controller:

1. `api.get('/')`
2. `api.get('/create')`
3. `api.get('/delete/id')`
4. `api.get('/details/id')`
5. `api.get('/edit/id')`

What would you name the associated view (see options on right)?

`edit.ejs`

`create.ejs`

`details.ejs`

`index.ejs`

`delete.ejs`

Respond with a view

By handling the path to *estimates* separately, our approach is reusable across products.

Given the following 5 typical view requests handled by each controller:

- | | |
|----------------------------------------|--------------------------|
| 1. <code>api.get('/')</code> | <code>index.ejs</code> |
| 2. <code>api.get('/create')</code> | <code>create.ejs</code> |
| 3. <code>api.get('/delete/id')</code> | <code>delete.ejs</code> |
| 4. <code>api.get('/details/id')</code> | <code>details.ejs</code> |
| 5. <code>api.get('/edit/id')</code> | <code>edit.ejs</code> |

What would you name the associated view (see options on right)?

`edit.ejs`

`create.ejs`

`details.ejs`

`index.ejs`

`delete.ejs`

Ex: Get index

- GET <http://127.0.0.1:8081/estimate>
- Simplest: use the response render() method to reply with a view file.
- More about views coming up...

```
10 // GET to this controller root URI
11 | api.get('/', (req, res) => {
12 |   res.render('estimate/index.ejs')
13 | })
```

Ex: Get create

- GET <http://127.0.0.1:8081/estimate/create>
- Build a new default Model, include logging
- Add more information to the response render() method: customize the title, define the layout, & name the data item variable returned

```
31 // GET create
32 | api.get('/create', (req, res) => {
33 |   LOG.info(`Handling GET /create${req}`)
34 |   const item = new Model()
35 |   LOG.debug(JSON.stringify(item))
36 |   res.render('estimate/create',
37 |     {
38 |       title: 'Create Estimate',
39 |       layout: 'layout.ejs',
40 |       estimate: item
41 |     })
42 | })
```

Respond by executing

- Controllers can respond by getting data - or views.
- Controllers can respond by executing **data modification commands** against our data store.
- We've covered the GET (or *read*) requests from CRUD.
- Can also **execute** Create, Update, or Delete.
- After executing the data modification, what should the user see?
- For example, after deleting an estimate, we typically want to re-route the user back to the index page (that displays all items in the collection).

Ex: Execute a delete

```
204 // DELETE id (uses HTML5 form method POST)
205 | api.post('/delete/:id', (req, res) => {
206 |   LOG.info(`Handling DELETE request ${req}`)
207 |   const id = parseInt(req.params.id, 10) // base 10
208 |   LOG.info(`Handling REMOVING ID=${id}`)
209 |   const data = req.app.locals.estimates.query
210 |   let item = find(data, { _id: id })
211 |   if (!item) {
212 |     return res.end(notfoundstring)
213 |   }
214 |   item = remove(data, { _id: id })
215 |   console.log(`Permanently deleted item ${JSON.stringify(item)}`)
216 |   return res.redirect('/estimate')
217 | })
```

10 controller request handlers

- Get JSON data requests (e.g. to populate comboboxes)

- findall `api.get('/findall')`
- findone `api.get('/findone/:id')`

- Get views to display for CRUD (show fields)

- Index `api.get('/')`
- Create `api.get('/create')`
- Delete `api.get('/delete/:id')`
- Details `api.get('/details/:id')`
- Edit `api.get('/edit/:id')`

- Handle execute data modification requests

- insert new `api.post('/save')`
- update `api.post('/save/:id')`
- delete `api.post('/delete/:id')`

2 provide
JSON

5 provide
views

3 execute data
modifications

Controller request handlers

- Get JSON data requests (e.g. to populate comboboxes)
 - findall `api.get('/findall')`
 - findone `api.get('/findone/:id')`
- Get views to display for CRUD (show fields)
 - Index `api.get('/')`
 - Create `api.get('/create')`
 - Delete `api.get('/delete/:id')`
 - Details `api.get('/details/:id')`
 - Edit `api.get('/edit/:id')`
- Handle execute data modification requests
 - insert new `api.post('/save')`
 - update `api.post('/save/:id')`
 - delete `api.post('/delete/:id')`

Complete set
of URI + HTTP
verb
combinations

HTML5 forms
only accept two
actions **GET**
and **POST**

M10

M10 - Project Repos

- Each section will work together to build the project.
- Today, we will pull (the nearly empty) project repository.
- See the links on the Canvas course overview page.
- Each person must have a working version of the project repo for our in-class activity Wednesday.

Views

Views

When starting a new express project, you can specify the view engine using the express generator (e.g., `express --ejs`) - or you can specify it in your existing app.

- Views are used to dynamically generate pages (or parts of pages) to display to the user.
- Views include **HTML** and **code** that operates on the HTML (to construct the page dynamically).
- Every major platform offers views and view engines (aka [template engines](#)); .NET offers view engines, Spring and Java offer view engines, many options to choose from
- What view engine we are using?
- By convention, views are kept in the **views** folder.

5 Views

- 5 views per model.
- Create views **server-side** using **ejs**.

5 typical view requests handled by each controller:

1. `api.get('/')`
2. `api.get('/create')`
3. `api.get('/delete/id')`
4. `api.get('/details/id')`
5. `api.get('/edit/id')`

`index.ejs`

`delete.ejs`

`details.ejs`

`edit.ejs`

`create.ejs`

App views

- We use views to create the overall project application as well.
- When designing, **decompose** each view.
- Find components.
- Identify **reusable components**, e.g. header.ejs, footer.ejs, navbar, layout.ejs.
- Identify **repeating components**, e.g. entries.
- Use folders to organize.

Specify application
layout
in
app.js
& the
ejs files
in root
views folder.

Next: 5 Examples

5 typical view requests handled by each controller:

1. `api.get('/')`
2. `api.get('/create')`
3. `api.get('/delete/id')`
4. `api.get('/details/id')`
5. `api.get('/edit/id')`

In each of the 5 examples, what is the URI? How is the view used?

Will this view be easier? Or harder? Why?

`index.ejs`

`create.ejs`

`delete.ejs`

`details.ejs`

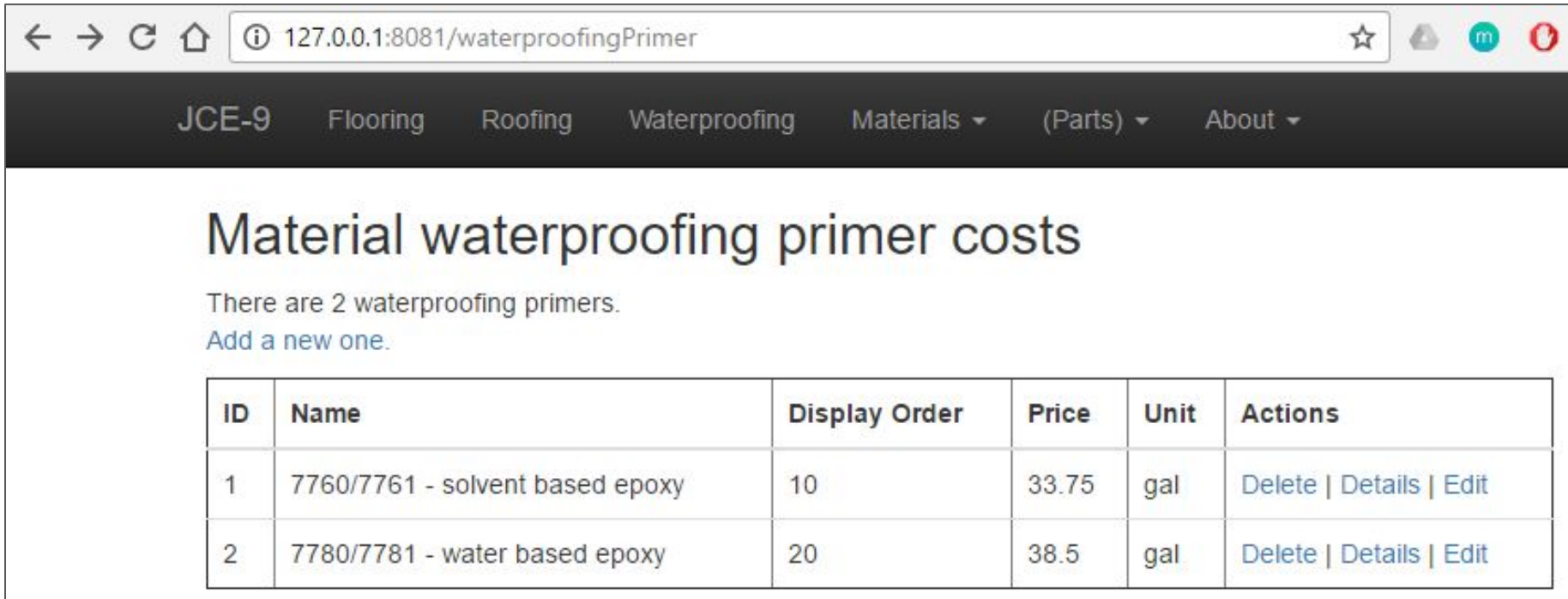
`edit.ejs`

Index.ejs (list all)

What is the URI? What is the **model**?

What is the **controller** that provides this view?

How is the view used?



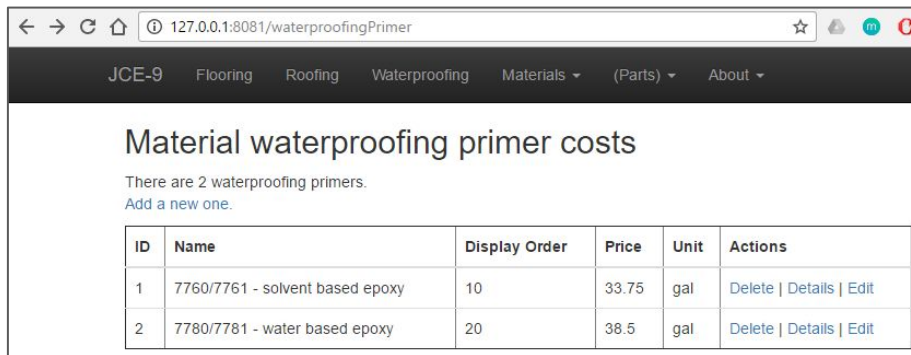
The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8081/waterproofingPrimer'. The page has a dark navigation bar with links: JCE-9, Flooring, Roofing, Waterproofing, Materials (dropdown), (Parts) (dropdown), and About (dropdown). The main content area is titled 'Material waterproofing primer costs'. Below the title, it states 'There are 2 waterproofing primers.' and includes a link 'Add a new one.' in blue. A table follows, listing two primers with their IDs, names, display orders, prices, units, and actions (Delete, Details, Edit).

ID	Name	Display Order	Price	Unit	Actions
1	7760/7761 - solvent based epoxy	10	33.75	gal	Delete Details Edit
2	7780/7781 - water based epoxy	20	38.5	gal	Delete Details Edit

Index.ejs: Behind the Scenes

app.js

```
178 app.locals.waterproofingBasecoats = db.find(waterproofingBasecoats);
179 app.locals.waterproofingEstimates = db.find(waterproofingEstimates);
180 app.locals.waterproofingPrimers = db.find(waterproofingPrimers);
181 app.locals.waterproofingTopcoats = db.find(waterproofingTopcoats);
```



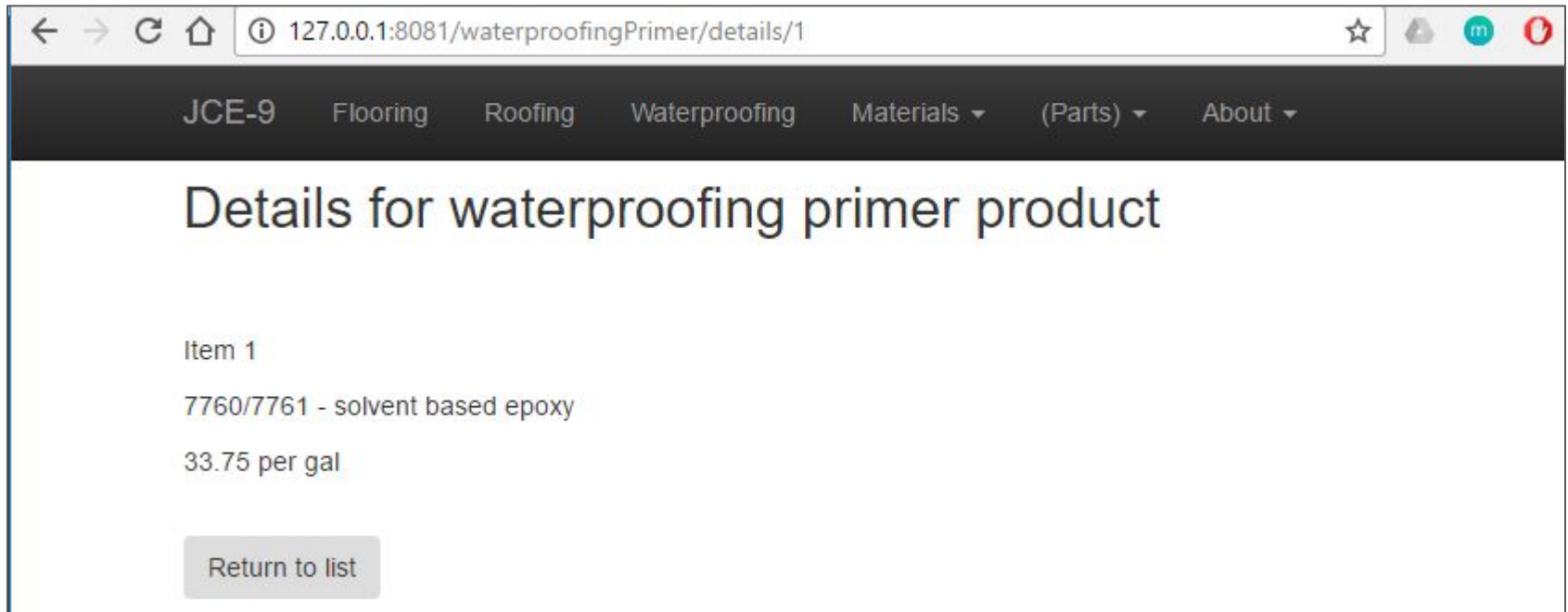
ID	Name	Display Order	Price	Unit	Actions
1	7760/7761 - solvent based epoxy	10	33.75	gal	Delete Details Edit
2	7780/7781 - water based epoxy	20	38.5	gal	Delete Details Edit

The connection between app.js and *.ejs is accomplished by assigning a property to app.locals. It can then be accessed in .ejs (omit app.locals in the .ejs file: just write the property name)

waterProofingPrimers/index.ejs

```
1 <% console.log(waterproofingPrimers.query); %>
2 <div class="container">
3   <h2>Waterproofing primers</h2>
4   <p>There are
5     <%= waterproofingPrimers.query.length %> waterproofing primers. <br/>
```

Details.ejs (view only)




URI? How is the view used?

Does this look relatively easy? Or hard?

Details.ejs: Behind the Scenes

Waterproofing primers

There are 2 waterproofing primers.

 Add a new one.

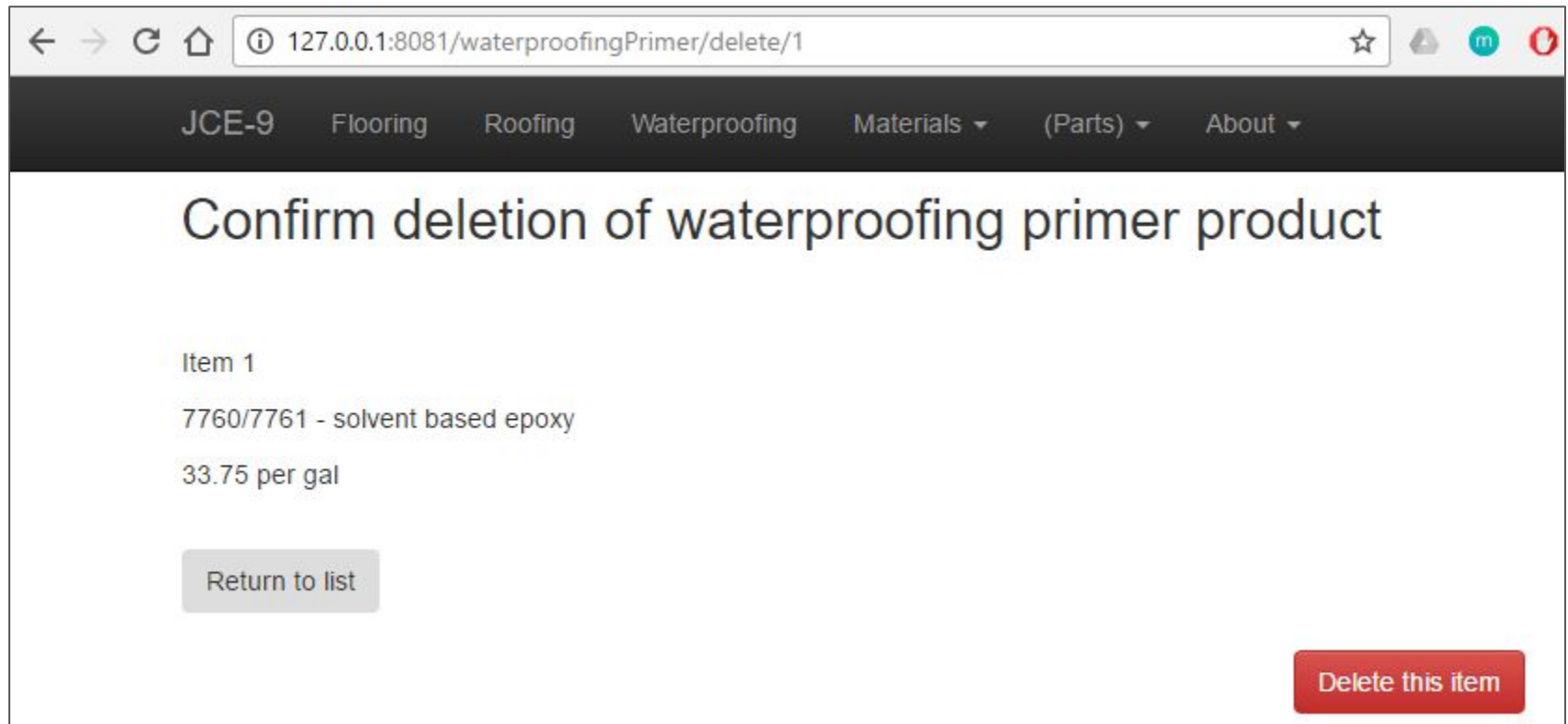
ID	Name	Display Order	Price	Unit	Actions
1	7760/7761 - solvent based epoxy	10	33.75	gal	<button>Delete</button> <button>Details</button> <button>Edit</button>
2	7780/7781 - water based epoxy	20	38.5	gal	<button>Delete</button> <button>Details</button> <button>Edit</button>

Here we see another way to get information into a .ejs file. `res.render()` allows us to pass an object, and its properties can be referenced in .ejs. Easy-peasey!

```
// GET /details/:id
api.get('/details/:id', function(req, res) {
  console.log("Handling GET /details/:id " + req);
  var id = parseInt(req.params.id);
  var data = req.app.locals.waterproofingPrimers.query;
  var item = find(data, { '_id': id });
  if (!item) { return res.end(notfoundstring); }
  console.log("RETURNING VIEW FOR" + JSON.stringify(item));
  return res.render('waterproofing_primers/details.ejs',
    {
      title: "WP Primers",
      layout: "layout.ejs",
      waterproofingPrimer: item
    });
});
```

```
<br/>
<br/>
<h2>Water proofing primer details</h2>
<br/>
<br/>
<p>Item
  <%= waterproofingPrimer._id %>
</p>
<p>
  <%= waterproofingPrimer.name %>
</p>
<p>
  <%= waterproofingPrimer.price %> per
  <%= waterproofingPrimer.unit %>
</p>
<br/>
<form method="get" action="/waterproofingPrimer">
  <input type="submit" value="Return to list" class="btn btn-caution" />
</form>
```

Delete.ejs (view & delete)



URI? How is the view used?

Easier or harder? Which view is this similar to?

create.ejs

127.0.0.1:8081/waterproofingPrimer/create

JCE-9 Flooring Roofing Waterproofing Materials (Parts) About

Create a new waterproofing primer product

There are currently 2 waterproofing primers.

The largest ID is 2.
The largest display order is 20.

ID

Name

Unit

Price

Display Order

required>

[Post new entry](#)

[Return to list](#)

Display useful
information:
largest ID &
largest display order
(can be removed
when the form is
working)

What is special about
the ID field?

URI? How is the view used?

Easier or harder?

Edit.ejs (modify)

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8081/waterproofingPrimer/edit/1`. The browser's navigation bar includes back, forward, and refresh buttons. The page has a dark navigation menu with links for JCE-9, Flooring, Roofing, Waterproofing, Materials, (Parts), and About. The main content area is titled 'Edit waterproofing primer product' and contains the following form fields:

- ID**: A text input field containing the value '1'.
- Name**: A text input field containing the value '7760/7761 - solvent based epoxy'.
- Unit**: A text input field containing the value 'gal'.
- Price**: A text input field containing the value '33.75'.
- Display Order**: A text input field containing the value '10'.

At the bottom of the form, there are two buttons: a blue 'Save' button and a grey 'Return to list' button.

What is special about the ID field?

URI? How is the view used?

Easier or harder? Similar to another view?

Implementing views

- Use HTML5 to display content.
- Use Bootstrap classes for styling.
- Delete is a dangerous operation (use red).
- Provide the ability to cancel (navigate back to list).
- Use **forms** for user inputs.
- Use one **form-group** for each input
 - Include a **label** for the input
 - **Validate** all inputs (min, max, required, read-only,...)

Suggestions

Debugging

1. First, ensure [all 10 request handlers](#) are in the controller.
2. Test each GET + **URI combination**. Is the correct view (even if not perfect) or the right type of JSON displayed?
3. When testing an action (e.g. click on "Delete" button), verify the URI is getting built correctly.
4. Second, ensure each **action** routes to the desired URI.
5. Finally, verify **behavior** associated with the action.

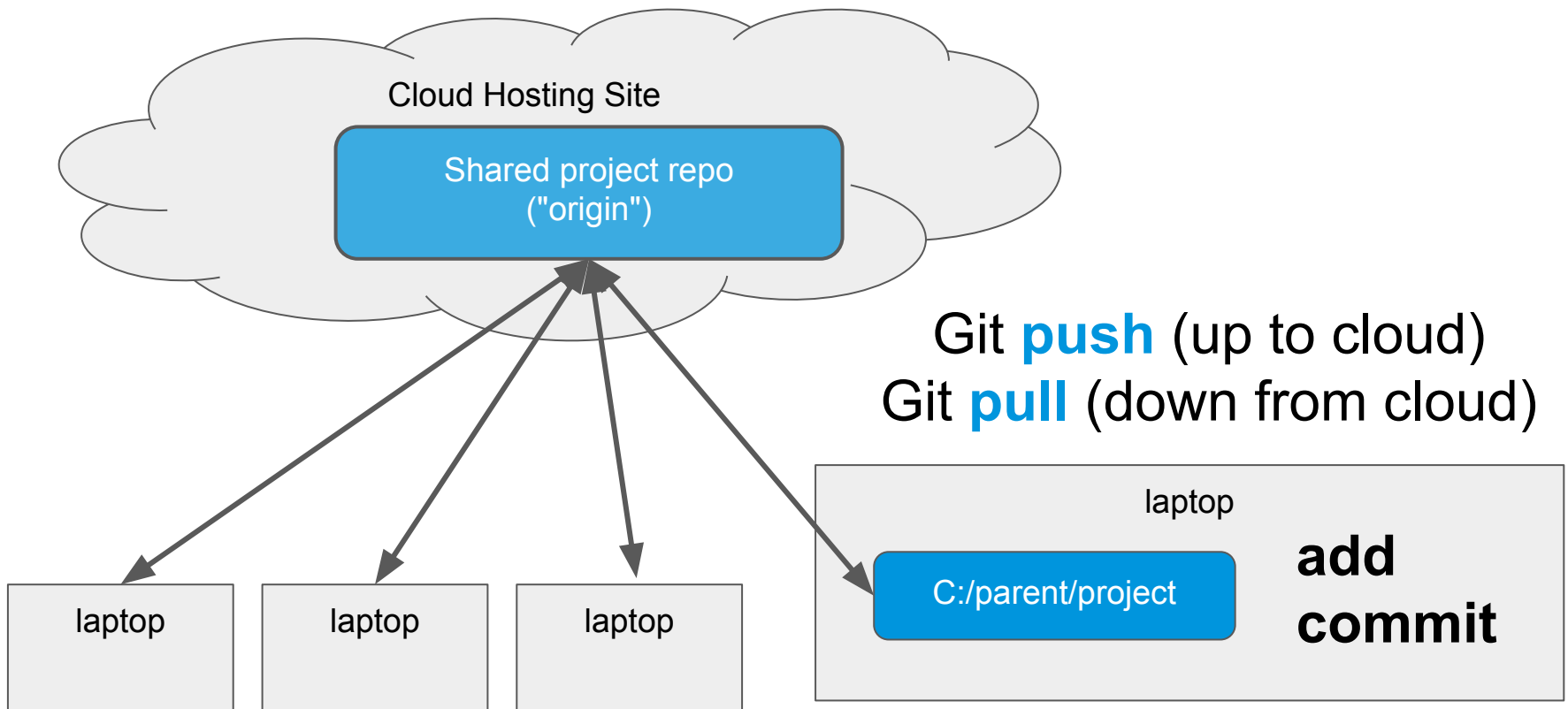
Recommendations

For happy, successful, collaborative coding:

- ❑ **Pull** code before you make any edits.
- ❑ When starting after a break, **pull** code.
- ❑ Do not work on stale code - pull new code!
- ❑ Make small, safe changes.
- ❑ Commit & push **small updates**.
- ❑ Always **pull** before you start to **edit**.
- ❑ Did we mention *pull*? Often? A lot? :)

git commands

1. Only **clone the code** once when you start (or start over).
2. Use **origin** as an alias for the common cloud repo.
3. **Add** and **commit** locally.
4. **Push** and **pull** from cloud.



Sharing the work

- Express web apps have many files.
- MVC organizes files by their role. There are folders for models, for views, for controllers.
- If we name our files correctly and put them in the correct folder, we can avoid explicitly having to configure all connections. Our code will be concise and just work. We have to follow conventions carefully - any spelling or capitalization error will cause it to not work.
- All well-organized server-side frameworks will include a "best practices" organization. We will follow some recommended best practices for a Node app (it looks similar in Java, C#, or Ruby).

W10

12 sets of initial files

1. See the project slides for the sets of initial files.
2. Each unit will be responsible for creating and tracking these files.
3. You may do the coding for them, and you may have help.
4. If no one else is assigned to help, assume they are your responsibility.
5. See project description link in Canvas.

In class

- Each unit is a group of 1-3 developers.
- Each unit has been assigned (or selected) to be responsible for creating and organizing the work in one of those 12 areas.
- Each section has a repository.
- In class, we will start at the beginning of the 12 areas and each team will create their empty folders and files.
- Work as a unit.
- Pull the repo code (it will be empty to start, but will change fast).
- Add & commit one or more items to your local repo. Push them to "origin" (the shared cloud location).
- Pull the repo and repeat until we all have a working skeleton.

In class

- If pair programming, allow another developer to type.
- In each of your files, add formal documentation to the top of all code files (not README or JSON).
- Document each JavaScript file as follows:

```
/**
 * @app.js
 * The main cost estimator express app file.
 *
 * File initially created and managed by
 *   Unit XX (two digit unit code)
 *   Name1 and email
 *   Name2 and email
 *
 * The extra line between the end of the @file docblock
 * and the file-closure is important.
 */

// file code goes here
```

**See
Exam 2
review guide**

Quiz