44-563: Unit 13

Developing Web Applications and Services

Includes

- Each class
- Schedule (no class W or F)
- Adding dates.
- Adding tests
- Generated test reports
- Repeating groups
- Adding client-side calculations
- Client visits Fri Dec 1 & 8

Each Class

Each Class

- ❑ Sit with your team.
- ❑ Pull fresh code
- ❑ Open **cge0N** folder with VS Code.
- ❑ Run **nodemon app**
- ❑ Browser:
  - ❑ Open app at localhost:808N (different ports by section)
  - ❑ Open BitBucket **repo**
  - ❑ Review **Issues Tracking** - update issues, see what needs to be done.

Schedule

Schedule

- No class Wednesday or Friday (Happy Thanksgiving!)
- Next Friday, Dec 1 - client in
- Following Friday, Dec 8 - final presentations
- See course site for finals schedule

Adding Dates

Good practice

h/t to **Raja Srikar Karthik Chinta** from Section 6 for reminding us!

Typically, in every data record, we add:

- Date Created
- Date Modified

Mongoose makes it easy (as do most ORMs). Just add the fields to your schema and default the value when the item is created (or modified).

```
dateCreated: { type: Date, default: Date.now },
```

Adding Authorization

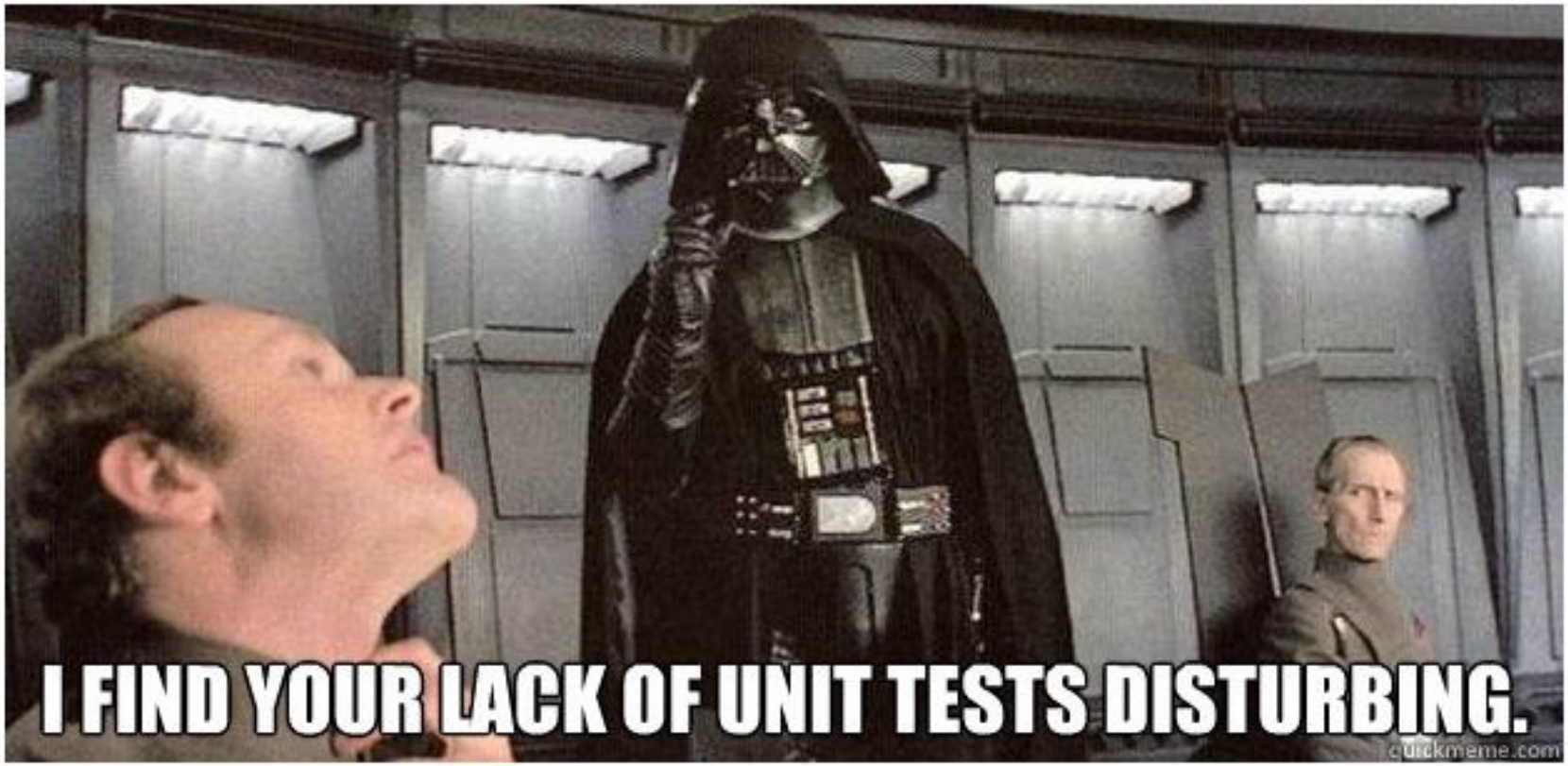
Good practice

Some features are limited to authorized users

Passport is used to provide authentication and authorization.
Estimates access must be limited to logged in users.
Passport makes this easy (once the framework has been implemented).
Add "passport.isAuthenticated" to each method in the estimates controller.

```
// GET all JSON
api.get('/findall', passport.isAuthenticated, (req, res) => {
  res.setHeader('Content-Type', 'application/json')
  const data = req.app.locals.puppies.query
  res.send(JSON.stringify(data))
})
```

Adding Tests



If Darth Vader says it, it must be true

https://medium.com/javascript-and-the-server/test-all-the-units-770fd3f08437

Good practice: Tests

- Always start with **tests**
- For Node/Express apps, good options include:
  - Mocha - good framework
  - Chai - assertion library
  - Supertest - makes API testing easy
  - Mochawesome - html reporting
- Use Dependency Injection.
  - Add module exports to your app.
  - Pass in the entire **app** for testing.

Good practice: Coverage

- Percent of code covered by tests:
  - % of **statements** // e.g. a switch statement
  - % of **branches** // e.g. all options on a switch
  - % of **functions** // has it been called?
  - % of **lines** // similar to statements

  1 line; 2 statements: `a=1;` `LOG.info(a);`
- For a Node/Express app, a good option is:
  - nyc https://istanbul.js.org/
- Reporting options:
  - text - default report in the console
  - lcov - generates html output

Project configuration

In package.json

- Add **dev-dependencies** needed for testing, coding standards, reporting
- Configure choices as desired
- Add **scripts** to create short names for complex commands
- To run a script, type **`npm` *`scriptname`***

Add testing, coverage, and reporting options.

Add to package.json

```
"devDependencies": {
    "chai": "^3.5.0",
    "mocha": "^3.4.2",
    "mochawesome": "^2.3.1",
    "nyc": "^11.3.0",
    "should": "latest",
    "supertest": "^2.0.1"
  },
  "scripts": {
    "start": "nodemon app.js",
    "test": "nyc mocha --timeout 10000 --reporter mochawesome --recursive"
  },
  "nyc": {
   "reporter": [  "lcov",  "text"  ]
  },
```

To run a script, use npm *scriptname* (e.g. **`npm test`**)

The **nyc** means "run a coverage report on our mocha command"

```
"devDependencies": {
   "chai": "^3.5.0",
   "eslint-plugin-import": "^2.7.0",
   "eslint-plugin-node": "^5.1.1",
   "eslint-plugin-promise": "latest",
   "eslint-plugin-react": "latest",
   "eslint-plugin-standard": "^3.0.1",
   "mocha": "^3.4.2",
   "mochawesome": "^2.3.1",
   "nyc": "^11.3.0",
   "should": "latest",
   "standard": "^10.0.3",
   "supertest": "^2.0.1"
 },
 "nyc": {
   "reporter": [  "lcov",  "text"  ]
 },
 "scripts": {
   "start": "nodemon app.js",
   "test": "standard && nyc mocha --timeout 10000 --reporter mochawesome --recursive"
 },
 "standard": {
   "ignore": [
"/mochawesome-report/"
   ]
 },
```

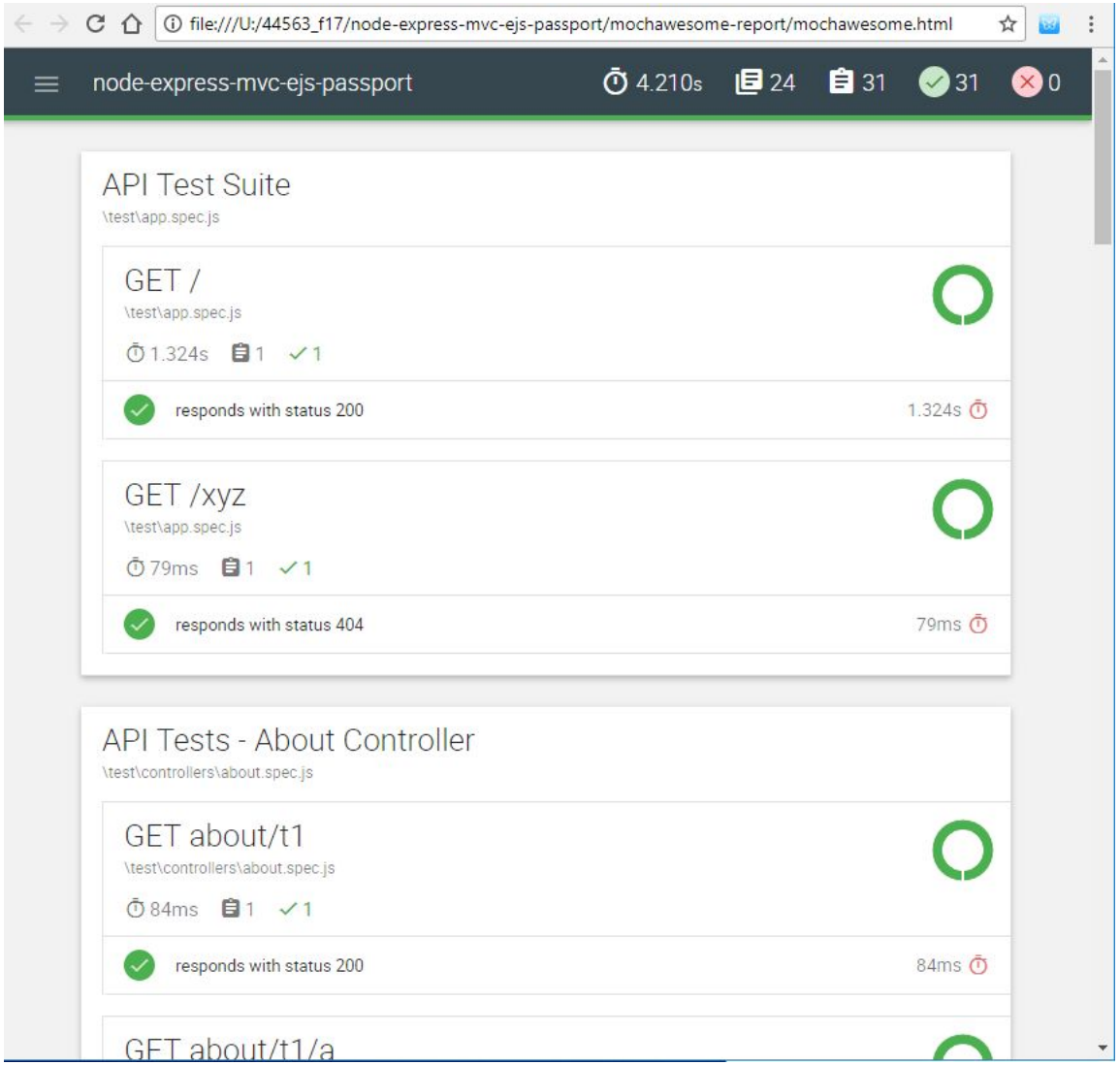
Serious standards.

REQUIRES JS STANDARD
In all code before running any tests.

Option: Enforce JavaScript standard before testing.

Generated Test Reports

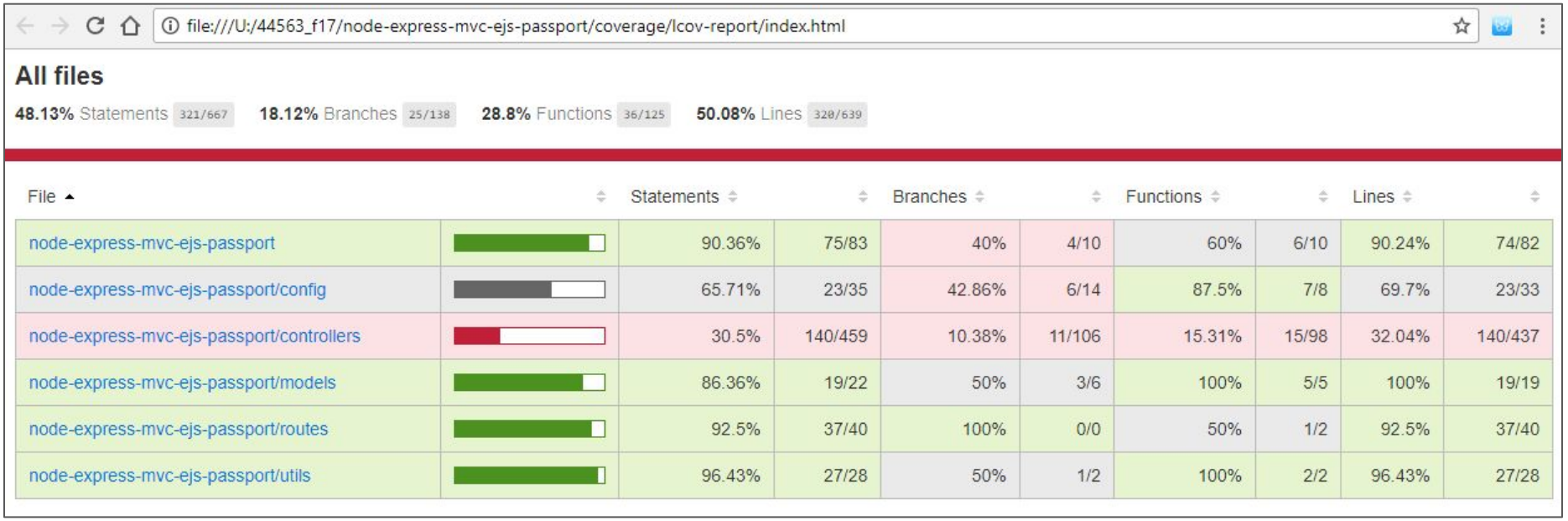
mochawesome.html



Running **npm test** will generate a mochawesome summary of testing results.

Adding --recursive to our script means it will test **all .js** files nested under the **test** folder.

coverage/lcov-report/index.html



File	Statements		Branches		Functions		Lines	
node-express-mvc-ejs-passport	90.36%	75/83	40%	4/10	60%	6/10	90.24%	74/82
node-express-mvc-ejs-passport/config	65.71%	23/35	42.86%	6/14	87.5%	7/8	69.7%	23/33
node-express-mvc-ejs-passport/controllers	30.5%	140/459	10.38%	11/106	15.31%	15/98	32.04%	140/437
node-express-mvc-ejs-passport/models	86.36%	19/22	50%	3/6	100%	5/5	100%	19/19
node-express-mvc-ejs-passport/routes	92.5%	37/40	100%	0/0	50%	1/2	92.5%	37/40
node-express-mvc-ejs-passport/utils	96.43%	27/28	50%	1/2	100%	2/2	96.43%	27/28

Coverage of utils is pretty good.
Coverage of controllers needs work.

Repeating Groups

Repeating groups

Properties may be multiples (e.g. puppy parents, miscellaneous costs, job materials)

This affects the associated:

- Model
- Seed Data
- Controllers
- Views:
  - Display
  - Inputs
  - Index

We will look at each of these individually

Repeating in model

Example: puppy parents

The type is array. Default uses square brackets.

```
parents: {
   type: Array,
   required: false,
   default: [
         {
              parentName: 'Mom',
              parentBreed: 'Bichon Frise',
              parentAge: 5
         },
         {
              parentName: 'Dad',
              parentBreed: 'Cavalier King Charles',
              parentAge: 6
         }
   ]
 }
```

Repeating in data

Example: puppy parents

The type is array. Parents uses square brackets.

```
{
   "_id": 3,
   "name": "Max",
   "breed": "Beagle",
   "age": 7,
   "parents": [
         {
              "parentName": "Lassie",
              "parentBreed": "Beagle",
              "parentAge": 5
         },
         {
              "parentName": "Sport",
              "parentBreed": "Beagle",
              "parentAge": 6
         }
   ]
 }
```

Repeating in controller

```
api.post('/save/:id', passport.isAuthenticated, (req, res) => {
 LOG.info(`Handling SAVE request ${req}`)
 const id = parseInt(req.params.id, 10) // base 10
 LOG.info(`Handling SAVING ID=${id}`)
 const data = req.app.locals.puppies.query
 const item = find(data, { _id: id })
 if (!item) { return res.end(notfoundstring) }
 LOG.info(`ORIGINAL VALUES ${JSON.stringify(item)}`)
 LOG.info(`UPDATED VALUES: ${JSON.stringify(req.body)}`)
 item.name = req.body.name
 item.breed = req.body.breed
 item.age = parseInt(req.body.age, 10)
 item.parents = []
 item.parents.length = 0
 if (req.body.parentName.length > 0) {
   for (let count = 0; count < req.body.parentName.length; count++) {
     item.parents.push(
          {
            parentName: req.body.parentName[count],
            parentBreed: req.body.parentBreed[count],
            parentAge: parseInt(req.body.parentAge[count], 10)
          }
     )
   }
   LOG.info(`SAVING UPDATED puppy ${JSON.stringify(item)}`)
   return res.redirect('/puppy')
 }
})
```

1. Set property to empty array (and set length to zero - should be optional)
2. Inspect how response body brings data from the client. There are many parentNames.
3. For each **parentName**, create and push a new item.

Inspecting Response Data

To capture this, set a breakpoint in the app.post() callback function and then just write req.body in the Debug Console

```
req.body
◢ Object {name: "Max", breed: "Beagle", age: "7" …} ⓘ
    age: "7"
    breed: "Beagle"
    name: "Max"
  ▷ parentAge: Array[2] ["5", "6"]
  ▷ parentBreed: Array[2] ["Beagle", "Beagle"]
  ▷ parentName: Array[2] ["Lassie100", "Sport100"]
```

Repeating in views

There are three types of views.

1- Views that **display** only:

details.ejs

delete.ejs

2- Views that allow user **input**:

create.ejs

Edit.ejs

3- Views that list all:

index.ejs

Recommendation: Don't duplicate effort in the highly similar views. Pull the shared content into a **partial view** and use it in both.

1- Display only

```
<br/>
<br/>
<h2>Puppy details</h2>
<br/>
<br/> ID:
<%=  puppy._id %>
 <br> Name:
 <%=puppy.name%>
   <br/> breed:
   <%=puppy.breed%>
<br/> Age (years):
<%=puppy.age%>
<br/>
<br/>
<% for (let j = 0; j < puppy.parents.length; j++) { parent = puppy.parents[j]; %>
  <div> Parent Name:<%=parent.parentName%>
<br/> Parent Breed: <%=parent.parentBreed%>
<br/> Parent Age (years):  <%=parent.age%> years
<br/>
<br/>
  </div>
  <% } %>
    <br/>
    <br/>
    <form method="get" action="/puppy">
<input type="submit" value="Return to list" class="btn btn-caution" />
    </form>
```

Use a JavaScript for loop

Insert HTML content once for each item.

2- User inputs

```
<!-- parents  -->
<p></p><div id="parentList"><% const origParentCount = puppy.parents.length;  %>
<span id="parentCount">   <%=origParentCount%>  </span> Parent Entries

<% for (var iEntry=1; iEntry <= origParentCount; iEntry++) {parent=puppy.parents[iEntry-1];
%>
<div id="oneEntry"><p></p>

<div class=row>  <div class="col col-xs-4">    <div class="form-group">
<label for="parentName" name="parentNameLabel">Parent Name </label>
<input type="text" class="form-control" id="parentName<%=iEntry%>" name="parentName"
value="<%= parent.parentName %>">    </div>  </div>

<div class="col col-xs-4">    <div class="form-group">
<label for="parentBreed" name="parentBreedLabel">Parent Breed</label>
<input type="text" class="form-control" id="parentBreed<%=iEntry%>" name="parentBreed"
value="<%= parent.parentBreed %>">    </div>  </div>

<div class="col col-xs-4">    <div class="form-group">
<label for="parentAge" name="ageLabel">Parent Age (yrs)</label>
<input type="number" class="form-control" id="parentAge<%=iEntry%>" value="<%=
parent.parentAge %>" name="parentAge" min="1" max=30>    </div></div>

</div><p></p></div>
<%  }  %>
</div>
```

Create a unique client-side **id** for each input in the loop (append the index from the loop).

3- Index (list all)

```
<br/><br/><div class="container-fluid"> <div class=row> <div class="col col-md-12">
<a href="puppy/create" class="btn btn-primary new"> Create a new puppy  </a>
<h2>Recent puppy entries</h2>
<p>There are   <%= puppies.query.length %> puppies.  <br/>  </p>
<% for (var i = puppies.query.length-1; i >= 0; i--) {   puppy = puppies.query[i];   %>
  <div class="panel panel-default"> <div class="panel-heading"></div>
  <div class="text-muted pull-right"></div><div class="panel-body">
   ID:           <%=puppy._id %>   <br/>
   Name:         <%=puppy.name%>   <br/>
   Breed:        <%=puppy.breed%>  <br/>
   Age (years):  <%=puppy.age%>    <br/><br/>

   <%=puppy.parents.length%> parents
   <% for (let j = 0; j < puppy.parents.length; j++) { parent = puppy.parents[j]; %>
   <div class="arrayItem">
        Name:  <%= parent.parentName %><br/>
        Breed:  <%= parent.parentBreed %> <br/>
        Age:  <%= parent.parentAge %>
   </div>
   <br/>
   <% } %>
<br/> <br/>
<a href="/puppy/delete/<%= puppy._id  %>" class="btn btn-danger"> Delete </a>
<a href="/puppy/details/<%= puppy._id %>" class="btn btn-info"> Details </a>
<a href="/puppy/edit/<%= puppy._id %>" class="btn btn-warning"> Edit </a>
    </div> </div>
  <% } %>
   </div>
 </div>
</div>
```

Use an inner JavaScript for loop

Insert HTML content once for each item.

Adding Client-Side Calculations

Use case: Immediate Results

As client enters or edits inputs, calculated values should be immediately updated

Immediate Results

Question: Client-side or Server-side?

Should immediate responses to input changes happen on the **client** (within the browser) - or must it require inputs to be sent back to the **server** for calculation after each change?

Immediate Results

Answer: **Client-side** where possible

Generally, immediate changes and calculated values can be implemented **client-side**.

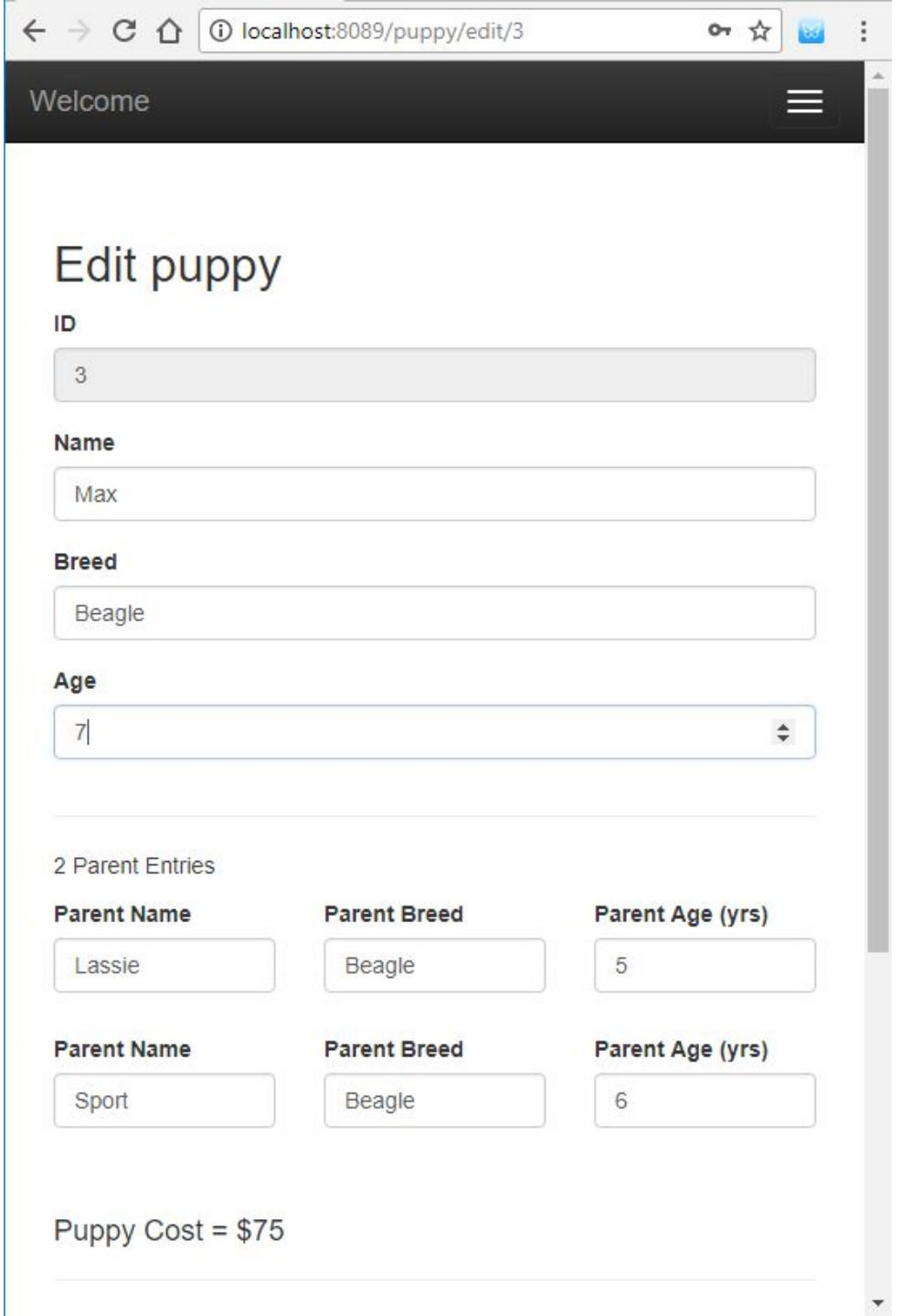
- Only the final user inputs need to be sent back to the server for storage.
- All calculated values will be regenerated as needed.

Working With Components

Client-side calculations are written in JavaScript.

JavaScript makes responding to events easy.

In the example app, when age is updated, the price changes.



Don't repeat yourself

In the example app, partial_edit.ejs is used by:

- views/puppy/create.ejs
- views/puppy/edit.ejs

```
<!-- COMMON TO EDIT AND CREATE -->
<%include partial_edit.ejs %>
```

Handle onchange event

in partial_edit.ejs we need to update each time the user changes the puppy age.

Assign an event handler to the onchange event.

In this case, we called it "onAgeChange()"

```
 <div class="form-group">
<label for="age">Age</label>
<input onchange="onAgeChange()" type="number" class="form-control" id="age" name="age"
value="<%= puppy.age %>" required>
</div>
```

OnAgeChange()

Inputs are strings, so call parseInt() or parseFloat() to get a number from the value of the input field with name = age.

```
<script>
function onAgeChange() {
   const newAge = parseInt($("input[name=age]").val());
   let price = 75
   if (newAge == 1) {
     price = 100
   }
   $("#puppyCost").html(price);
 }
 </script>
```

Helper function for $

Let's create a helper function to transform our integer into a nice currency integer. JavaScript makes it easy to add this new feature to all numbers.

```
function onAgeChange() {
  const newAge = parseInt($("input[name=age]").val());
  let price = 75
  if (newAge == 1) {
    price = 100
  }
  $("#puppyCost").html((price.toCurrencyInt());
}
```

Extending Number

Add our desired function(s) to the Number prototype. Let's create one for non-int currencies at the same time.

```
// Add  .toCurrency() method to all Numbers
Number.prototype.toCurrency = function () {
  return toCurrency(this); // we'll write toCurrency() that takes a Number arg

};

// Add  .toCurrencyInt() method to all Numbers
Number.prototype.toCurrencyInt = function () {
  return toCurrencyInt(this);  // we'll write toCurrencyInt() that takes a Number arg
};
```

Extending Number

Helper functions based on gist by Kyle Fox (https://gist.github.com/kylefox/780065)

```
// function toCurrency(amount)
function toCurrency(amount) {
  return "$" + amount.toFixed(2);  //returns string with $ fixed at 2 decimal places
};

// function toCurrencyInt(amount)
function toCurrencyInt(amount) {
  return "$" + amount.toFixed(0);  //returns string with $ fixed at 0 decimal places
};
```

Call calcs on open

When the window first loads, call all the update functions to calculate results based on starting inputs

```
// window.onload
// a function that will call all update functions to display calculated values
// when user first opens the window
window.onload = function () {
  onAgeChange();
}
```

Optional: Don't make hitting enter submit form

How?

1. For this document form, listen for all key press events.
2. Figure out what key was hit (either a charCode, a keyCode, or 0).
3. If key === 13 (enter), prevent the default behavior.

```
// prevent submission when hitting enter key - optional, but user-friendly
document.forms[0].onkeypress = function (e) {
  var key = e.charCode || e.keyCode || 0;
  if (key === 13) {     e.preventDefault();   }
}
```

M13

No Class Wed or Fri