Lunch.ly Reservation System

Download exercise <../express-lunchly.zip>

Lunchly is an Express app that is *not* an API server, nor is it RESTful.

Instead, it's a server-side templated application with custom URLs.

Part One: Setting Up

1. Create a project folder, Git repository, and ***package.json***.
2. Create a database, ***lunchly***
3. Read in the sample data in ***data.sql***

   *(Later, if you're curious and want to see the script that generated all this sample data, it's in the solution directory as **seed.py**)*

3. Install the required node modules:

   ```
   $ npm install
   ```

4. Start the server with ***nodemon***
5. Take a tour of the site at ***http://localhost:3000*** — you should be able to see the list of customers, view a customer detail page, and add a customer. Adding a reservation is not yet complete, so expect errors if you try to create a new reservation.

Part Two: Look at the Code!

There's lots of existing sample code. **Give yourself a quick tour!**

app.js
   Our application object; can be imported from other files/tests

models/
   Model objects as as classes, to "abstract away" database handling

routes.js
   Routes for the web interface

server.js
   Functionality to start the server (this is the file that is run to start it)

templates/
   Jinja templates, rendered with the JS "Nunjucks" library

seed.py
   File to create sample data (you don't need to use this, but may find it interesting as an example of how to create realistic fake data.)

Part Three: Class (Static) Methods

Many methods on objects are called on an *instance* of a class (called on an individual cat, to use our example above). In some cases, though, you may want a method that isn't called on a particular cat, but is called on the ***Cat*** class *itself*.

This is most often useful for methods that do things like create a new cat, or look up a cat in a database — you'd want to call this function *before* we had an individual cat yet, since the job of this function is to find/create one.

While many languages call these "class methods," in Javascript they're often called "static methods," so JS uses the ***static*** keyword to create these methods.

Take a look at the `get(id)` method of the ***Customer*** classes. These methods are meant to be called on the class. They look up the corresponding customer in the database, and return a JS instance of the correct class. Find where this code is being used in the handlers and make sure you understand it.

Read about these here:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static>

Part Four: Nunjucks

So far, we've used Express to render JSON or simple strings. In many cases, this might be all our backend does, and you'd have a dedicated front-end to parse JSON responses and create a UI.

However, it is possible for Express to template complete HTML pages, the way we did in Flask. To do this, we use the "Nunjucks" library, which is an implementation of the Jinja2 language in Javascript.

Take a quick look at the templates in ***/templates/***. There's nothing particular you need to do here yet, but you may find it useful to see how easy it is to template in JS.

Note: Configuring nodemon for templates

By default, ***nodemon*** will not listen for changes to .html files. This means that when you start editing your templates, you'll need to manually stop and start your server in order to see your changes take effect.

To fix this, you can tell nodemon what extensions to watch with the **-e** flag. For this assignment, it's worth telling it to listen for changes to HTML, CSS, and Javascript, so you should start your server with the following command:

```
$ nodemon -e js,html,css
```

Part Five: Full Names

In several templates, we show customer names as `{{ firstName }} {{ lastName }}`. This is slightly tedious, that we have to write out both fields, but also might be inflexible for future data changes: what if we added a middle name field later? What if we added a prefix field for labels like “Ms.” or “Dr.”?

Add a function, `fullName`, to the Customer class. This should (for now) return first and last names joined by a space. Change the templates to refer directly to this.

Part Six: Saving Reservations

We’ve already written a ***.save()*** method for customers. This either adds a new customer if they’re new, or updates the existing record if there are changes.

We don’t yet have a similar method for reservations, but we need one in order to save reservations. Write this.

Further Study

Part Seven: Add Search Functionality

It would be nice to search for a customer by name, rather than having to find them in a list. Add a quick search form to the bootstrap navigation bar to search for a customer by name.

Do this by continuing the pattern of abstracting database operations to the model classes — any route(s) you write shouldn’t directly use the database. Think of a good name for any new methods on the class.

You can either make a new route and template to show results, or you could probably make it work with the existing listing index route and template (as you’d prefer).

Part Eight: Best Customers

We like to show our best customers — those that have made the most reservations. Add a new route that finds our top 10 customers ordered by most reservations.

Like before, do this by adding functionality to the model class, so that there isn’t SQL directly in your route handlers.

Make sure you do this counting work in the database, rather than trying to do all the counting in Javascript.

Getters and Setters

Read about getters/setters in JS:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get>
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set>

Getters and setters allow you to use an object like this:

```
cat = new Cat();
cat.color = "orange";

console.log("This cat is ", cat.color);
```

while having the *color* property incorporate business logic. For example, our class could be:

```
class Cat {
  get color() {
    return this._color;
  }

  set color(val) {
    if (val === "green")
      throw new Error("No green cats!");
    this._color = val;
  }
}
```

Here, we can have validation when we set a property, while "abstracting away" this operation. The user of this object doesn't need to know/understand that this happens — just say things like `cat.color = "orange"`.

(Of course, we could have done this by writing a function named something like *setColor*, and then users of the class would set the color with `cat.setColor("orange")`, but these kind of explicit setter functions, or corresponding `getColor()` getter functions can be tedious to use, with all of those parentheses).

There are plenty of opportunities to use getters and setters in this codebase. Here are some examples of ways you could use them:

- For notes on a customer or reservation, use a hidden ***_notes*** property to ensure that if someone tries to assign a falsey value to a customer's notes, the value instead gets assigned to an empty string.
- Turn ***fullName*** into a getter.
- Use the getter/setter pattern with ***numGuests*** on a reservation, such that the setter throws an error if you try to make a reservation for fewer than 1 person.
- Use the getter/setter pattern with ***startAt*** on a reservation, so that you must set the start date to a value that is a ***Date*** object.

- Use the getter/setter pattern with ***customerId*** on a reservation, such that once a reservation is assigned a ***customerId***, that key can never be assigned to a new value (attempts should throw an error).

Talk with your partner about other ways you could exploit the getter/setter pattern.

Other fun things!

Should you wish to continue with these ideas, there is plenty of other functionality you could add to this system:

- Add a new field for middle name, which can be optional, but should appear in the full name displays.
- Add a feature to edit existing reservations. Make sure to keep SQL out of the routes themselves.
- On the customer listing page, show the most recent reservation for each. Make sure when you do this that you continue to list all customers, even those without any reservations!
- This uses the “Moment.js” library to format dates prettily. This is a powerful library for handling all sorts of time/date features. This is often used by websites to show “pretty relative dates”, like “5 minutes ago”, “1 week ago”, “more than 2 years ago”, and so on. Learn about this feature and change the customer detail page to show the most recent reservation for every customer, but with one of these pretty versions.
- Research how to write tests with supertest when you have a template-based Express app. Then write some tests for your routes! (You can unit test your model methods too.)
- Add proper full-text search (that can handle things more faster and more flexibly than ***ILIKE*** queries). PostgreSQL has a very comprehensive FTS (full text search) system. You can read about this <https://www.compose.com/articles/mastering-postgresql-tools-full-text-search-and-phrase-search/>.

  This is a pretty big feature to learn about and add, but this might be a neat weekend day project if you’re interested in learning more about real-world backend search.

Solution

View our Solution <solution/>