Express Jobly

Download exercise <../code.zip>

This is a multi-day exercise to practice Node, Express, and PostgreSQL with relationships. In it, you'll build "Jobly", a job searching API.

> NOTE **Goals & Requirements**
>
> - This is a pure API app, taking values from the query string (GET requests) or from a JSON body (other requests). It returns JSON.
> - This gets authentication/authorization with JWT tokens. Make sure your additions only allow access as specified in our requirements.
> - Be thoughtful about function and variable names, and write developer-friendly documentation *for every function and route* you write.
> - The starter code is well-tested, with excellent coverage. We expect your new contributions to maintain good coverage.
> - Model tests check the underlying database actions. Route tests check the underlying model methods and do not rely directly on the database changes. This is a useful testing design consideration and you should continue it.
> - We *strongly encourage you* to practice some test-driven development. Write a test before writing a model method and a route. You will find that this can make the work of adding to an app like this easier, and much less bug-prone.

Take your time, be organized and clear, and test carefully. Have fun!

Part One: Setup / Starter Code

Download the starter code. Do a quick skim of the code to get a sense of the main components and the organization.

We've provided `jobly.sql`, which will create a database (with a small amount of starter data) and a test database. Set those up. (Some of the tables included are not currently used by the application; you'll add the parts of the app that will use those tables in the exercise).

Read the tests and get an understanding of what the *beforeEach* and *afterEach* methods are specifically doing for our tests. Be prepared to discuss this with us during code reviews.

Run our tests, with coverage. Any time you run our tests here, you will need to use the `-i` flag for Jest, so that the tests run "in band" (in order, not at the same time).

Start up the server *(note that, unlike most exercises, we start this server on port 3001)*.

Test the API in Insomnia.

First Task: sqlForPartialUpdate

A starting piece to document and test:

We've provided a useful method in `helpers/sql.js` called *sqlForPartialUpdate*. This code works, and we use it, but the code is undocumented and not directly tested. Write unit tests for this, and thoroughly document the function.

Part Two: Companies

We've provided a model and routes for companies.

Adding Filtering

The route for listing all companies (*GET /companies*) works, but it currently shows all companies. Add a new feature to this, allowing API users to filter the results based on optional filtering criteria, any or all of which can be passed in the query string:

- *nameLike*: filter by company name: if the string "net" is passed in, this should find any company who name contains the word "net", case-insensitive (so "Study Networks" should be included).
- *minEmployees*: filter to companies that have at least that number of employees.
- *maxEmployees*: filter to companies that have no more than that number of employees.
- If the *minEmployees* parameter is greater than the *maxEmployees* parameter, respond with a 400 error with an appropriate message.

Some requirements:

- Do not solve this by issuing a more complex WHERE clause than is needed (for example, if the user isn't filtering by *minEmployees* or *maxEmployees*, the WHERE clause should not include anything about the *num_employees*.
- Validate that the request does not contain inappropriate other filtering fields in the route. Do the actual filtering in the model.
- Write unit tests for the model that exercise this in different ways, so you can be assured different combinations of filtering will work.

  Write tests for the route that will ensure that it correctly validates the incoming request and uses the model method properly.
- Document all new code here clearly; this is functionality that future team members should be able to understand how to use from your docstrings.

> ATTENTION **Stop and get a code review**
>
> Before you continue, make sure you have completed the exercises in the previous section and **make sure your code is well-documented, organized, and tested.**

#

Part Three: Change Authorization

Many routes for this site do not have appropriate authorization checks.

Companies

- Retrieving the list of companies or information about a company should remain open to everyone, including anonymous users.
- Creating, updating, and deleting companies should only be possible for users who logged in with an account that has the *is_admin* flag in the database.

Find a way to do this where you don't need to change the code inside the view function, and where you don't need to SELECT information about the user on every request, but that the authentication credentials provided by the user can contain information suitable for this requirement.

Update tests to demonstrate that these security changes are working.

Users

- Creating users should only permitted by admins (registration, however, should remain open to everyone).
- Getting the list of all users should only be permitted by admins.
- Getting information on a user, updating, or deleting a user should only be permitted either by an admin, or by that user.

As before, write tests for this carefully.

> ATTENTION **Stop and get a code review**
>
> Before you continue, make sure you have completed the exercises in the previous section and **make sure your code is well-documented, organized, and tested.**

Part Four: Jobs

Add a feature for jobs to the application.

We've already provided a table for this. Study it.

> NOTE **Research!**
>
> Our database uses the *NUMERIC* field type. Do some research on why we chose this, rather than a *FLOAT* type. Discover what the *pg* library returns when that field type is queried, and form a theory on why. Be prepared to discuss this during code reviews.

Adding Job Model, Routes, and Tests

Add a model for jobs — you can pattern-match this from the companies model.

Updating a job should never change the ID of a job, nor the company associated with a job.

Write tests for the model.

Add routes for jobs. The same routes should be handled as we did for companies (for now, omit the special filtering on the *GET /* route), with the same security requirements (anyone can get the jobs, but only admins can add, update, or delete them). Make sure you suitably validate incoming data.

Write tests for the routes.

Adding Filtering

Similar to the companies filtering for the *GET /* route, add filtering for jobs for the following possible filters:

- *title*: filter by job title. Like before, this should be a case-insensitive, matches-any-part-of-string search.
- *minSalary*: filter to jobs with at least that salary.
- *hasEquity*: if *true*, filter to jobs that provide a non-zero amount of equity. If *false* or not included in the filtering, list all jobs regardless of equity.

Write comprehensive tests for this, and document this feature well.

Show Jobs for a Company

Now that the app includes jobs, change the *GET /companies/:handle* feature so that it includes all of the information about the jobs associated with that company:

```
{ ... other data ... , jobs: [ { id, title, salary, equity}, ... ] }
```

> ATTENTION **Stop and get a code review**
>
> Before you continue, make sure you have completed the exercises in the previous section and **make sure your code is well-documented, organized, and tested.**

Step Five: Job Applications

We've provided a table for applications. Incorporate this into the app by adding a method onto the *User* model, allowing users to apply for a job.

Add a route at *POST /users/:username/jobs/:id* that allows that user to apply for a job (or an admin to do it for them). That route should return JSON like:

```
{ applied: jobId }
```

Change the output of the get-all-info methods and routes for users so those include the a field with a simple list of job IDs the user has applied for:

```
{ ..., jobs: [ jobId, jobId, ... ] }
```

Document this carefully and write tests.

> ATTENTION **Stop and get a code review**
>
> This exercise builds off of previous sections and we really want to see your code at this point, before you continue onto the other steps.

Further Study

Before you continue, make sure you have completed all the parts above, including strong documentation, tests, and developer documentation artifacts.

Here are some broad ideas for further study. We do not provide solutions on any of these particular tasks.

Choosing Random Password

When admins add a user via the *POST /users* route (not the self-registration route), they should not provide a password. Instead, the system will make a random password for the user (you can find third-party libraries that will generate excellent random passwords). This route should continue to return the same information, so an admin can send the user that token to authenticate to the site, and the user can then change their password to something only known to them.

This is a very real-world feature.

Use *enum* Type

Research PostgreSQL's *enum* types and change the *state* column in the applications table to be an enum that consists of 'interested', 'applied', 'accepted', 'rejected'.

Add Technologies for Jobs

Add a table for technologies which is a many to many with jobs (a job can require "Python" and "JavaScript", and these technologies could be linked to many jobs).

Add Technologies for Users

Make the technologies table a many to many with users as well and create an endpoint that matches users with jobs where the technologies are the same.

Solution

› Express Jobly