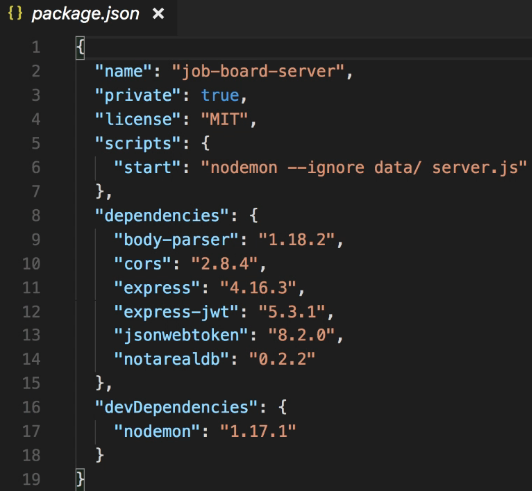
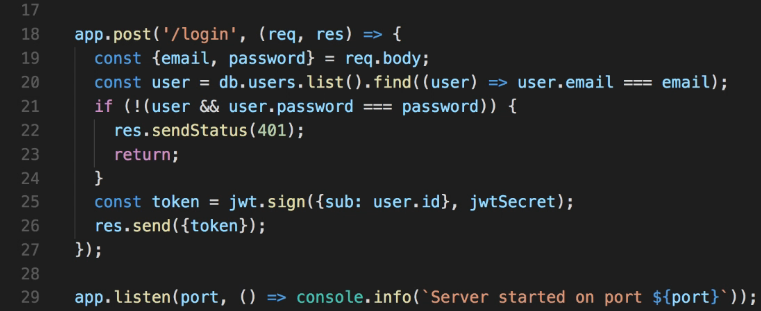
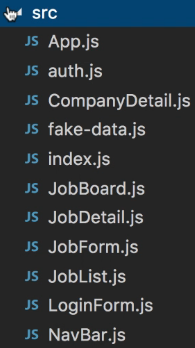
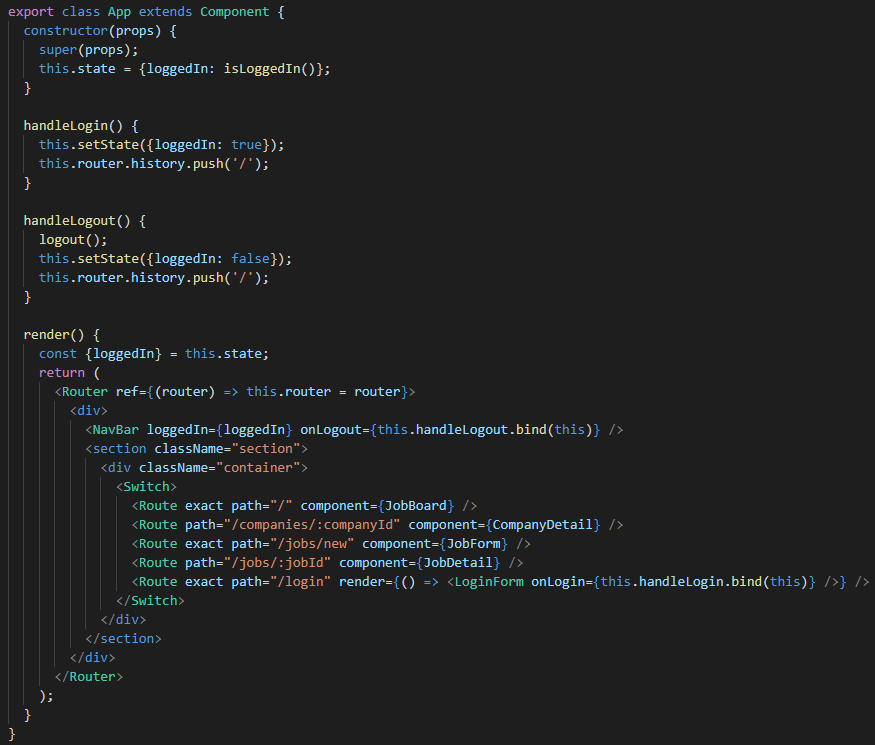
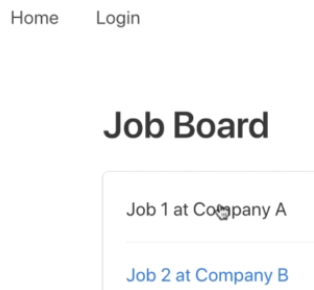
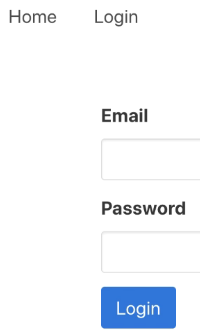
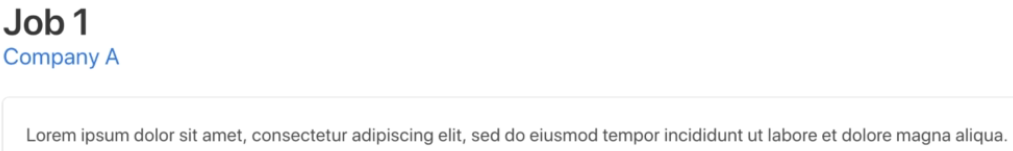
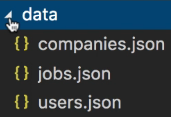
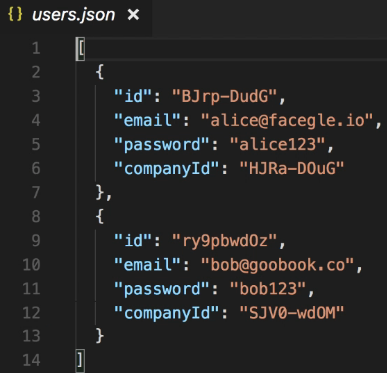
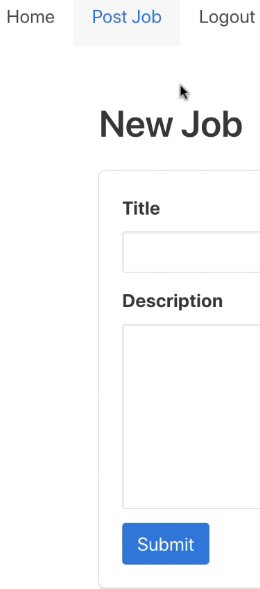
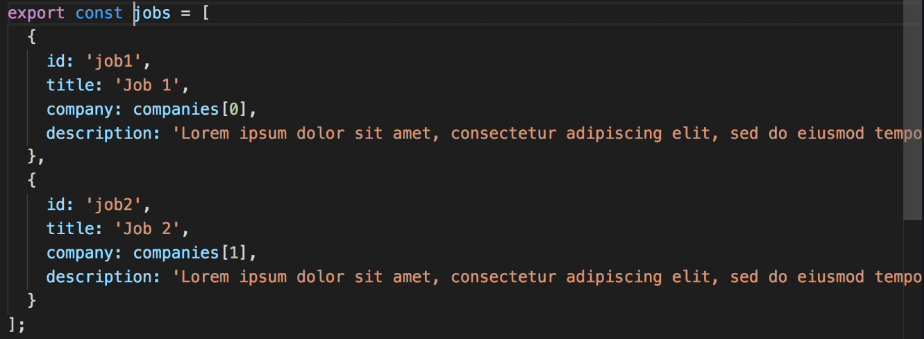
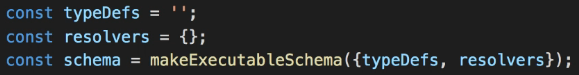
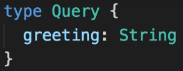
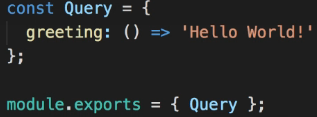
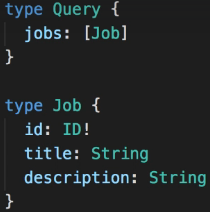
**The Job Board Application**  
\* Go to the github page of the application, click “Clone or Download” and copy the url.  
\* Open terminal in the folder where you want to have it.  
**git clone thatUrl folderName**  
git clone https://github.com/up2skill/graphql-job-board.git job-board  
\* The “client” and “server” folders are like different projects even though they’re in 1 repository.  
\* **“server”**  
 **npm install**  
\* “server.js” contains most of the server code.  
  
  
**npm start**  
\* This will run it with **nodemon**, that is a tool that automatically restarts the script whenever we make changes to the code.

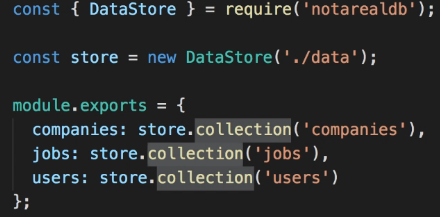
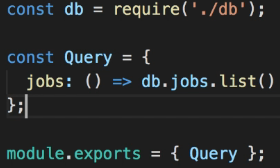
\* **“client”**  
  
**npm install  
npm start**  
\* **App.js**

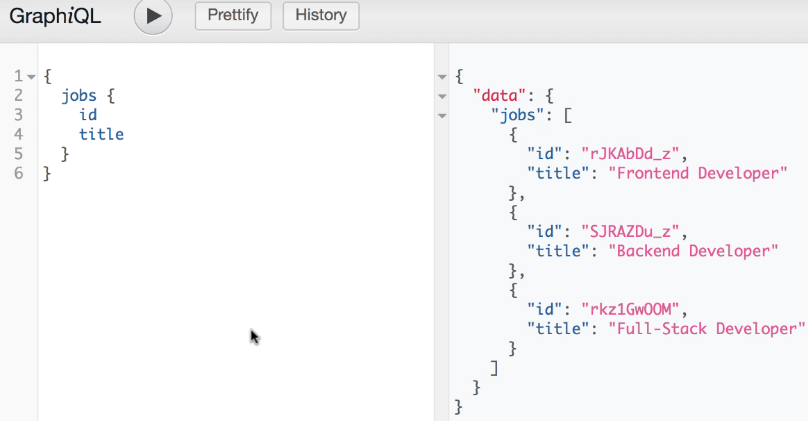
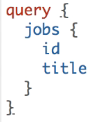
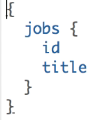
   
  
=> **Right now it’s hard-coded**.  
=> **Our task will be to replace it with data coming from the server by making GraphQL Queries**.  
\* **The login already calls the server because it doesn’t use GraphQL**.  
\* In “server”.  
  
=> This is effectively the database for our server.  
\* In a real application, you would probably use something like MySQL or PostreSQL or MongoDB as the database.  
\* The way we have it makes it easy to look at the data and we don’t need to configure anything.  
   
\* **What we need to do is expose a GraphQL API from the server and then call it from the Frontend**.

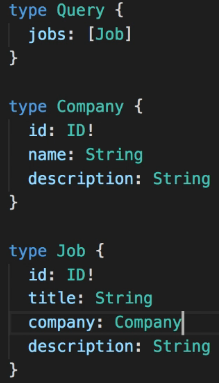
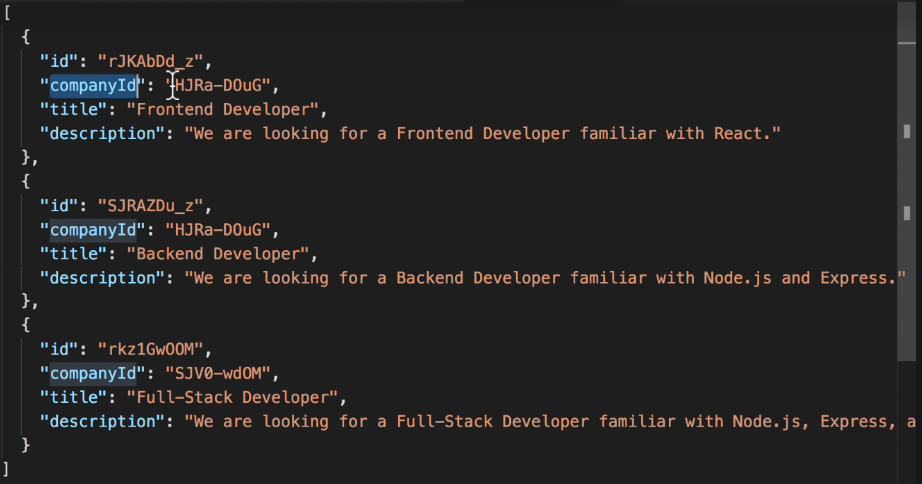
**GraphQL Server Endpoint**  
\* Our fake data in JobBoard:  


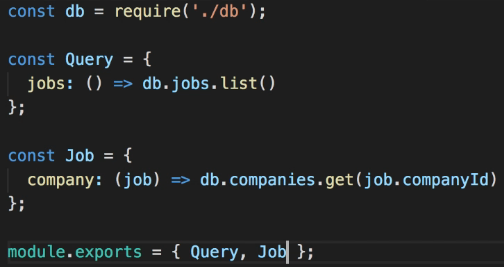
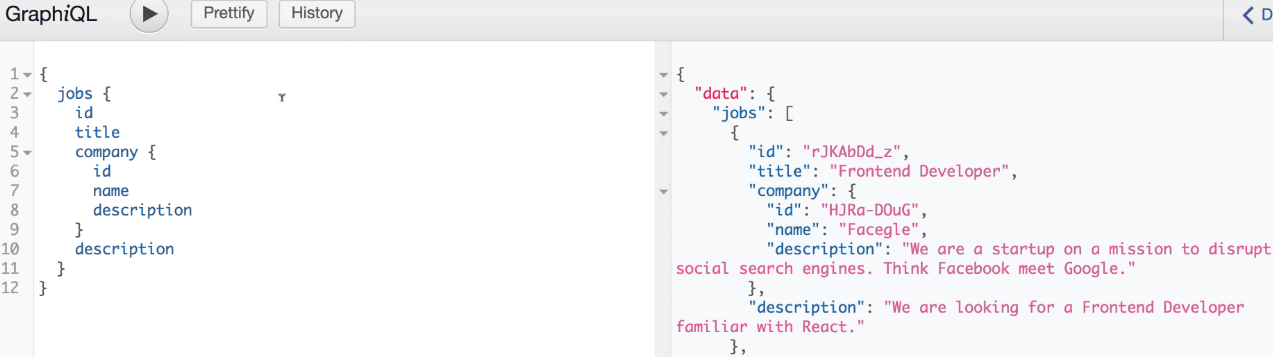
\* Let’s implement the server part.  
**npm install graphql graphql-tools apollo-server-express**  
\* Server.js  
  
  
\* For testing purposes, we can also enable the GraphiQL tool.  
  
  
\* **Since this project will be more complex, let’s split the typeDefs and resolvers into multiple files**.  
\* **typDefs => schema.graphql**  
\* **We always need a root type that is called Query**.  
\* We should return a list of jobs but right now I just want to get to a point where we can start the server and check that everything is wired up correctly before we introduce new concepts.  
  
\* How we we use it?  
  
  
**fs.readFileSync()**  
=> **We need to specify the encoding if we want the file to be read as a string, otherwise this function returns a buffer instead**.  
\* **resolvers => resolvers.js**  
**module.exports** => **because this is a node module so it follows the** **commonjs format**.  
  
=> This way in the other file we’ll be able to import this object and use it.  
  
**npm start**  
\* So we added GraphQL to our Express application.

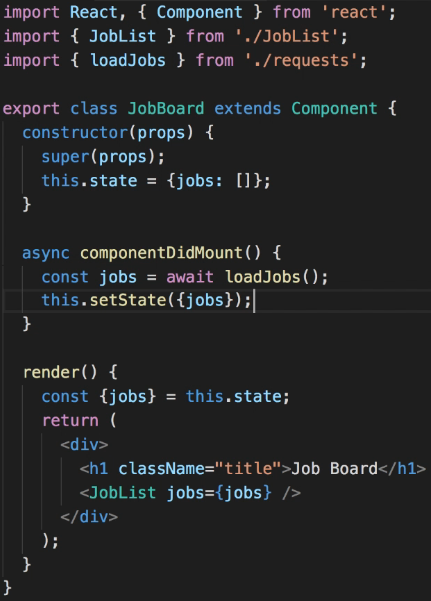
**Returning an Array of Jobs**  
\* **In GraphQL, we can create our own custom types, we’re not limited to the built-in ones**.  
\* **You’ll want an ID for almost all your custom types**.  
=> **It’s such a common requirement that GraphQL provides a built-in type for IDs**.  
\* The ID type is similar to a string, in fact it is sent as a string in JSON but using ID gives some extra information, it tells the client that it’s a unique identifier, not intended as a human-readable string.  
**!** => **should never be null**.  


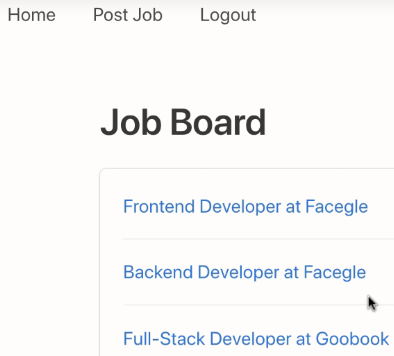
\* In db.js, we have:  
  
\* **This is essentially our persistance layer, for this example we’re not using a “true” database like MySQL or MongDB, instead, we save our data in local JSON files**.  
\* The basic operations you can do with any database are pretty similar: you can read items, create items, update items and delete items => CRUD. So you could easily replace this simple db with some code that connects to a real database.  
\* To store data in JSON files we use a package called **notarealdb - that’s a project that I’ve written precisely for this sort of sample applications, it lets you create a collection for each object type you want to store like “jobs” and each colleciton is stored in its own JSON file.**  


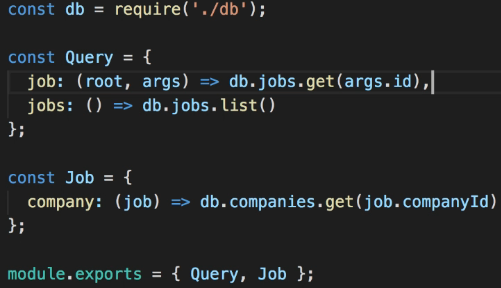
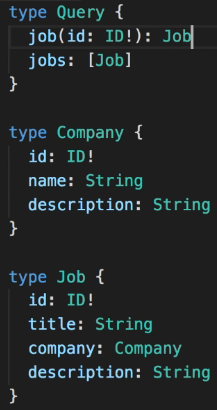
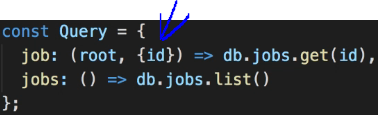
**Nested Objects in GraphQL Queries**  
\* **We need to explicitly list all the fields we want to get back from the server**.  
\* If we don’t specify the description for example, it won’t return it:  
  
=> **This is to prevent overfetching**.  
=> **That way the response from the server is smaller and that means less traffic over the network and therefore our application should also be a bit faster**.  
\* **It also means that it’s up to each client to decide what data he needs. The GraphQL engine takes care of this for us. It will return only the requested fields. We don’t need to do anything special in our server code**.  
\* We can write the root object explicitly.  
 

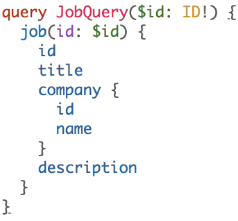
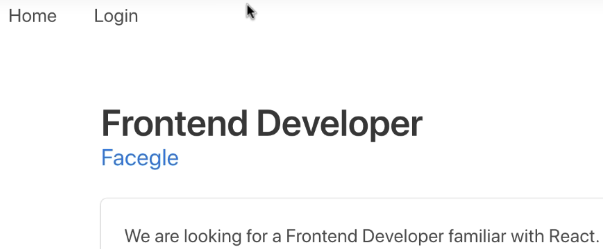
**Object Associations: Job/Company**  
\* **If the data returned by our resolver is missing a certain field, that will result in a null value being returned to the client**.  
\* **So keep in mind that just because the server starts up successfully and returns some data, it doesn’t necessarily mean that the data is what you expect**.  
\* It’s still up to you to write the correct code and test it of course.  
\* **jobs**  
  
\* **companies**  
  
=> The companyId matches the ID of a company, so the companyId in the jobs is what you would call a **foreign key** in relational databases, we can use it as a reference to fetch the company object with the same id.

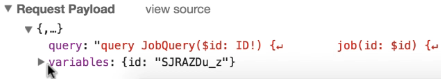
\* Each resolver function receives some arguments:  
**1st argument** => the parent object - in this case we’re resolving the company for a job so it’s the job.  
  
=> **Return the company whose ID is the asme as the companyId of this job. That’s why these functions are called resolvers. This function resolves a company for a given job**.  
=> When the client sends a query that wants the jobs, the GraphQL engine first invokes the jobs resolver for the Query type and this returns an array of job objects from the database, at this point the GraphQL engine goes through each job in the array and invokes the company resolver for the Job type.  


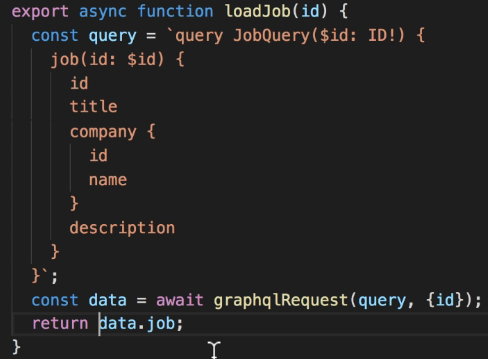
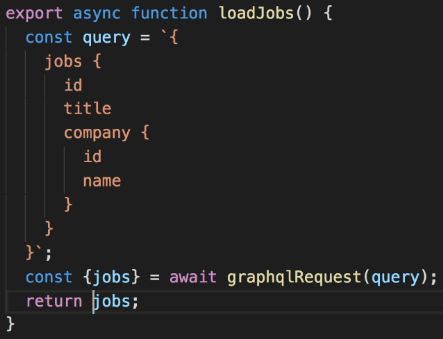
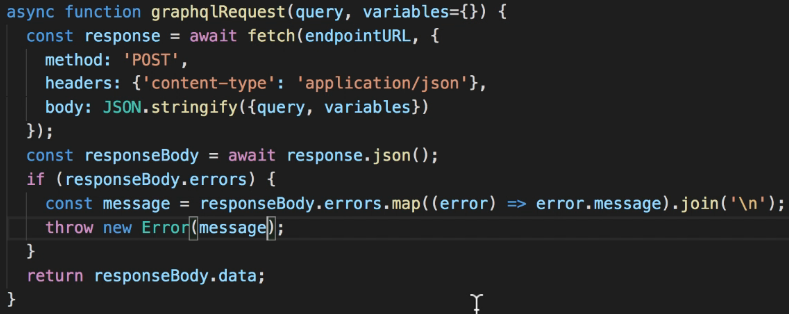
**Fetching Jobs in the Client**  
\* It’s time to call the GraphQL server from the frontend.  
\* Let’s create **requests.js** for all the GraphQL requests.  
 

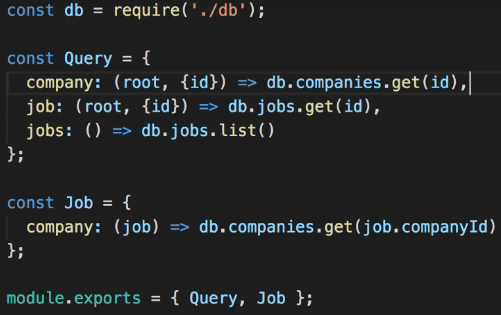
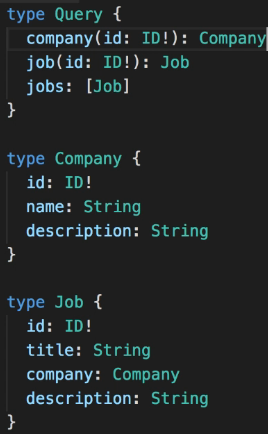


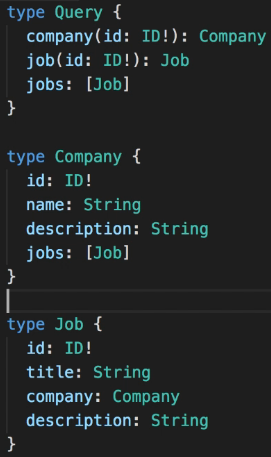
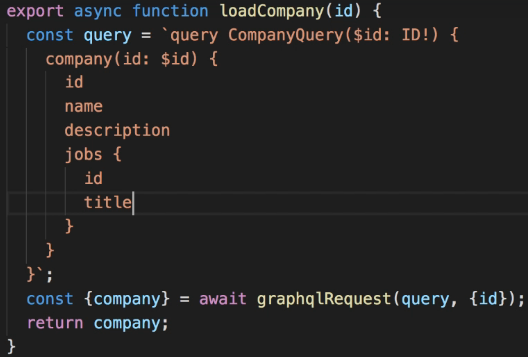
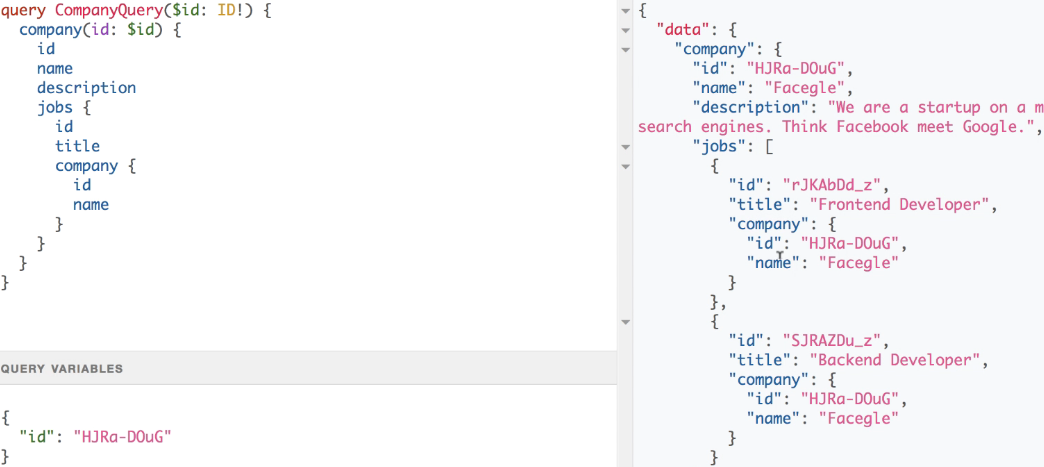
**Arguments: Returning a Job by ID**  
\* Right now on the server we can only get all the jobs. Let’s add the job by id.  
**job(id: ID!)**  
  
\* Resolvers => job: (root, **args**) => The args will contain the arguments passed in the GraphQL query.  
\* **In GraphQL strings have to be enclosed in Double Quotes “”**.  
\* **You can use Object Destructuring**:  
  
\* **I like it better this way because you can look at the resolver function and immediately see that it expects an ID in the arguments**.

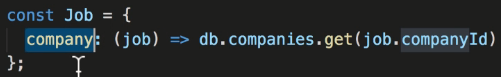
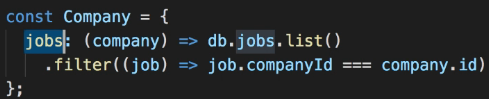
**Query Variables: Fetching a Job**  
\* Maybe we could use a template literal string in JavaScript to do the variable substitution, but actually **GraphQL provides a way to pass dynamic variables directly into the query**.  
\* We mentioned that you can optionally type Query in front of the request:  
   
=> **This is actually required if we want to use variables**.  
**Operation Name** => **We can also give an optional name to our query**.  
\* **It doesn’t make a difference it terms of what we get back as a result, but naming the Query can be useful for example for debugging**. **Operation Name will be logged as part of the error message**.  
\* What we’re interested in right now though is passing variables.  
**($id: ID!)**  
  
\* Variable values are sent as a separate JSON object.  
\* **The $ must be used only inside the query**.  
   
\* **We’re using the shorthand syntax so set both the property name and its value (id: id).**  




**Handling GraphQL Error Responses**  
\* So far we have loadJobs() and loadJob().  
\* There’s a fair amount of duplication between these 2 functions.  
\* And now we need to write another function loadCompany(), that will also be very similar.  
\* **Let’s refactor this code a little bit**.  
=> **We could extract a new function with all the common functionality**.  
=> **This function doesn’t need to be exported because it’ll only be inside this file**.  
=> **Most of the request configuration is the asme for any GraphQL request**.  
\* **What will vary is the Query string**.  
\* **We can make a parameter optional by giving it an empty value - empty object**.  
  
   
\* **We can use Object Destructuring**.  
\* **What if the GraphQL server returns an error response?**  
=> Let’s say we have a typo in the query - we misspell one of the fields like “titlez”. So we ask for a field that doesn’t exist.  
=> We get an error.  
  
=> The error message is not very helpful.  
\* If you open the network tab, you see:  
  
=> **The GraphQL server returns a much more useful error message**.  
\* **If it’s an error response, it’ll have an “errors” property in the returned JSON object**.  
  
  
\* This error overlay is only visible when you run the React application in development mode.  
\* If you build for production, your users won’t see it.  
\* Obviously in a production application you may want to display a user-friendly message saying “sorry, something went wrong.”

**Fetching a Company by ID**  
  


**Returning Jobs for a Company**  
  
\* In this case, we can have multiple jobs associated with the same company.  
\* **If you’re familiar with Data Modeling Concepts, we have a ONE-TO-MANY relationship between Company and Job**.  
   
\* This project uses a CSS framework called Bulma which is similar to Bootstrap or Materialize.  
\* There’s an interesting thing here - we can get the jobs or a single job and each job has a company. But then each Company also has some jobs.  
=> **So we could navigate these relationships in many ways**.  
  
=> **It proves that when we write a GraphQL query, we have full flexibility in what we request as long as it matches the schema**.  
\* We can keep going deeper if we want:  
  
\* In this case we’re just repeating the same data in multiple places but imagine you’re working with data like that of a social network for example. You could have a query where you ask for a user and the friends of that user and the friends of the friends, you could traverse the network of friends in any way you like.

\* GraphQL was invented at Facebook where they do have that sort of data and that’s probably no coincidence.  
\* What happens behind the scenes whenever a query is executed is that our resolver functions will be called as many times as required.  
\* So whenever there’s a Job object and we request its company field, this function will be invoked:  
  
\* And for each Company object if we requested the jobs field, this function will be called:  
  
\* So by writing these 2 simple functions, GraphQL allows us to navigate our data in many different ways and with arbitrary levels of nesting. That’s very powerful.