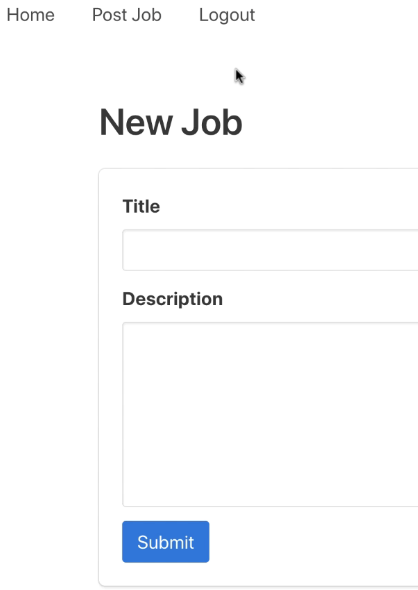
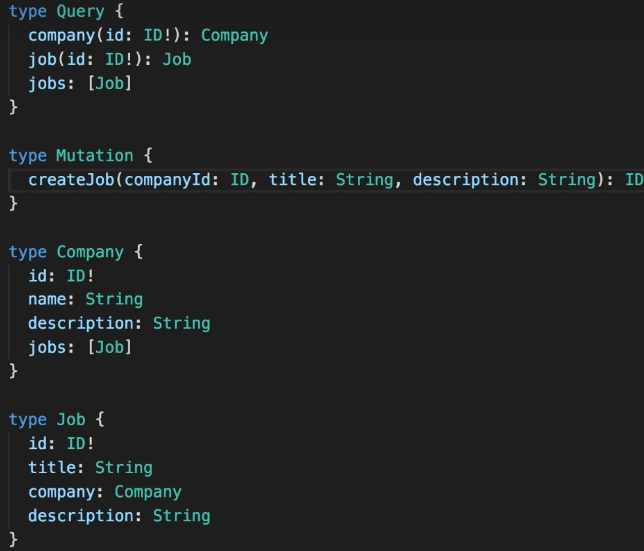
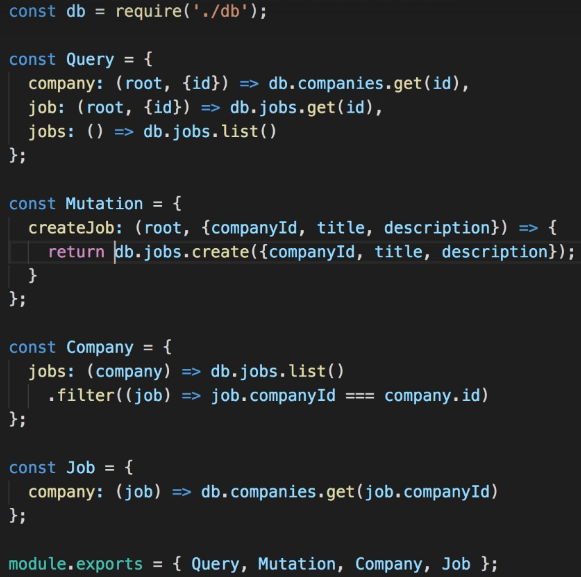
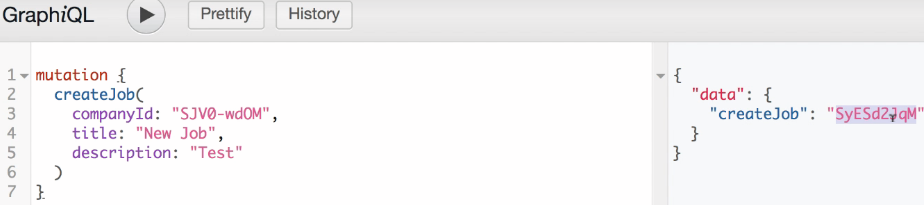
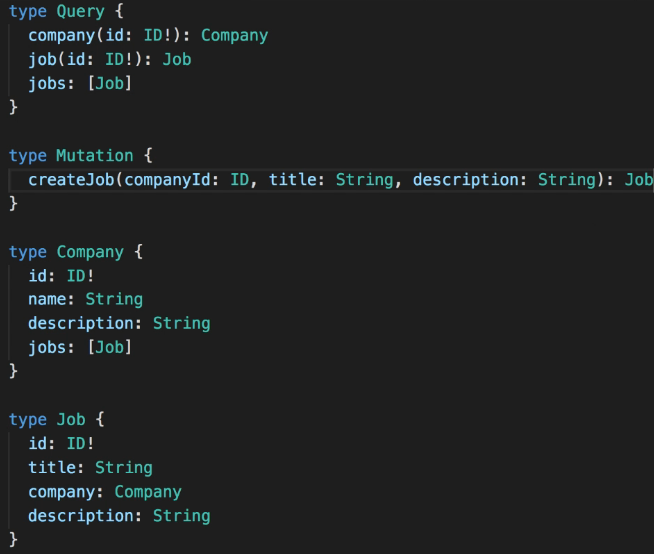
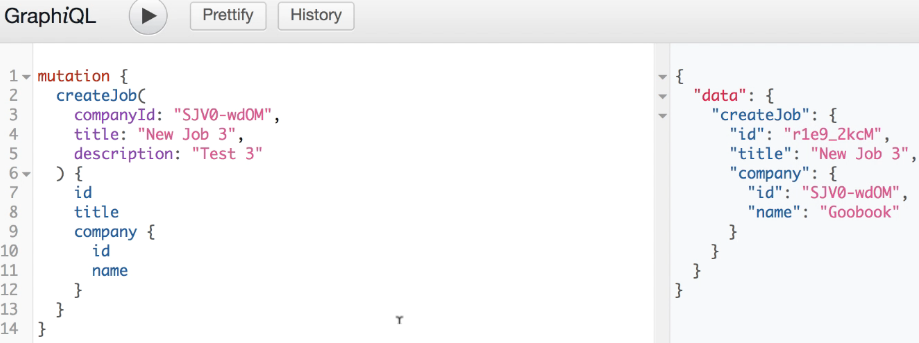
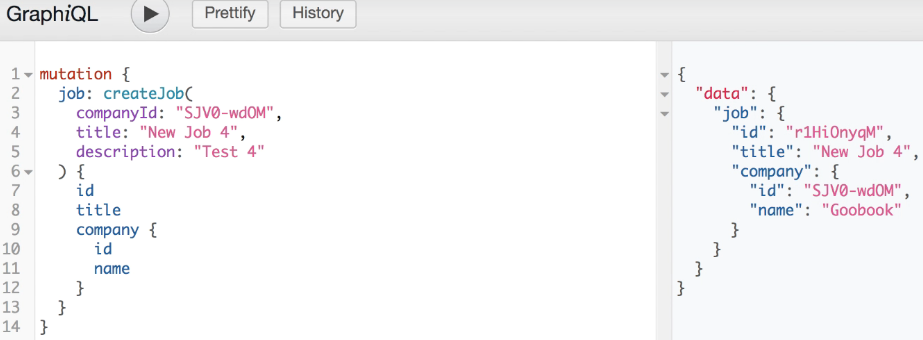
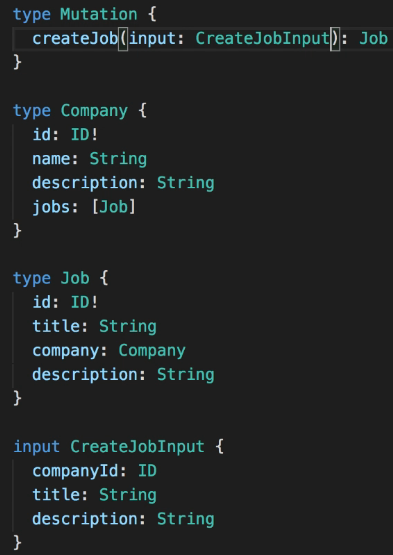
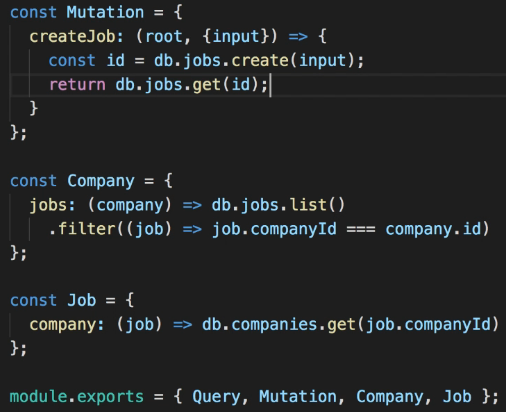
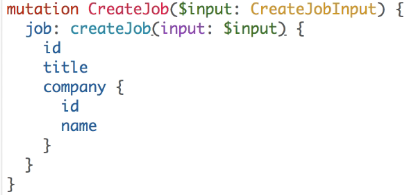
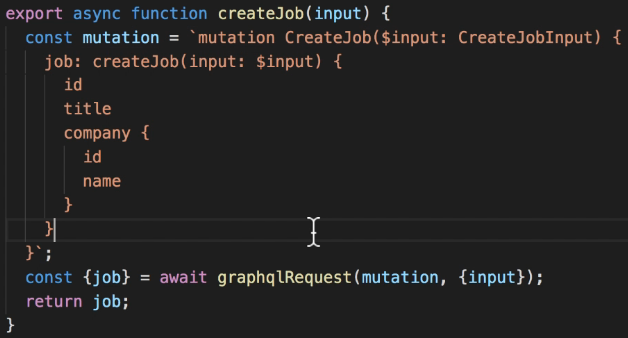
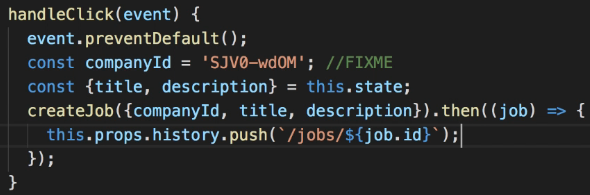
**Creating a Job**  
\* We’ll start looking at how to modify data using GraphQL.  
  
**Mutations** => **In GraphQL, operations that modify the data**.  
\* **They must be kept separate from queries**.  
\* **Mutation is a root type just like Query**.  
\* Mutations usually have arguments.  
\* The ID will be assigned by the server, we don’t want to pass it from the client when creating a job.  
\* **Mutations have to return a result, just like Queries**.  
\* In this case at a minimum we may want to return the job ID that’s generated by the server.  
   
  
\* We’ll come back here and discuss some best practices for mutations later.  
\* **Mutation root type contains all operations that have side effects => that modify the data**.  
\* **Query root type contains all operations that only read data without modifying it**.  
\* **Writing resolvers for Mutations is really no different from the other resolver functions except of course that in this case the function can modify the data**.

**Best Practices for Mutations**  
\* Let’s now improve this mutation so that it’s easier to work with from a Client’s perspective.  
\* In a client application after we create a job, we may want to display the new job including all its fields.  
=> So maybe it would be better to return a Job object here instead of just this ID.  
   
  
\* We send the companyId as an argument in the request, so we already know its value but not the company name for example.  
\* So a client application could use this response data to display the job details without the need to make a separate query.  
=> **By making the mutation return a Job object, instead of just the ID, we can potentially make a single request to the server instead of 2**.  
**Aliases** => GraphQL supports aliases (both Queries and Mutations), since we return a job, it’s weird to have it named as createJob in the response, we can change it:  


\* The next thing we can improve about this mutation is the arguments.  
=> We currently have 3 arguments: companyId, title, description.  
=> **Let’s say we want to pass their values as variables. We can give a name to this mutation and then declare the variables**.  
  
=> Then we would need to replace each value with a variable.  
=> And when making the request, we would also need to pass 3 values as query variables.  
=> So having lots of variables is not very convenient from the client point of view.  
\* **Let’s see how we can make things easier for clients that call our API**.  
\* Even if we had the right fields in the Job and wanted to use it for the mutation, GraphQL doesn’t allow us to use a regular type for an input argument.  
\* The types we wrote so far like Job and Company are   
**Output Types** => we can return them in response to a query.  
\* As arguments we can only use   
**Input Types** => **input** instead of type.  
\* It’s common to name input types like the mutation where they’re used, adding “Input at the end”.  
   
   
  
\* You can see how the mutation is simpler now that we have a single argument instead of 3.  
\* **Our mutation was working fine even before these changes but now our API is easier to use from the client point of view**.  
\* **Things like accepting a single input argument for each mutation are some of the best practices that are emerging in the GraphQL community**.

**Calling a Mutation from the Client**  
\* **Sending a GraphQL request for a mutation is no different from sending a query**.  
  
\* Once the user clicks submit and the job has been created, we should probably display the job details.  
**this.props.history.push()** => **pushing a new path into the browser history**.  
  
\* We had to hard-core a companyId because we need to pass it to the mutation but the form only has title and description.  
\* We’ll see how to fix this after we talk about authentication.