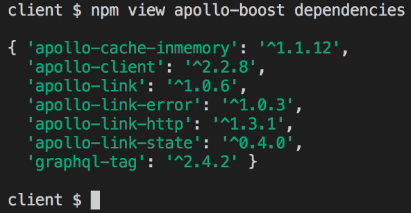
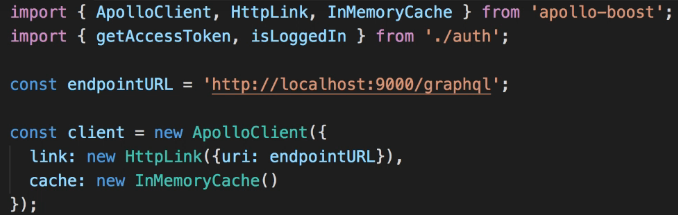
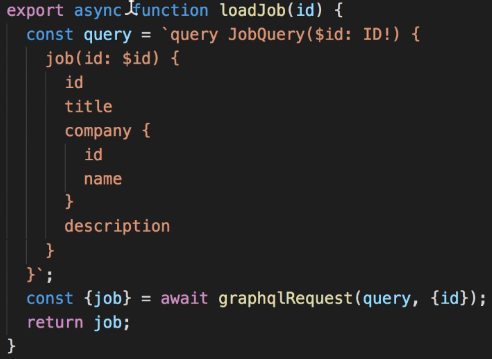
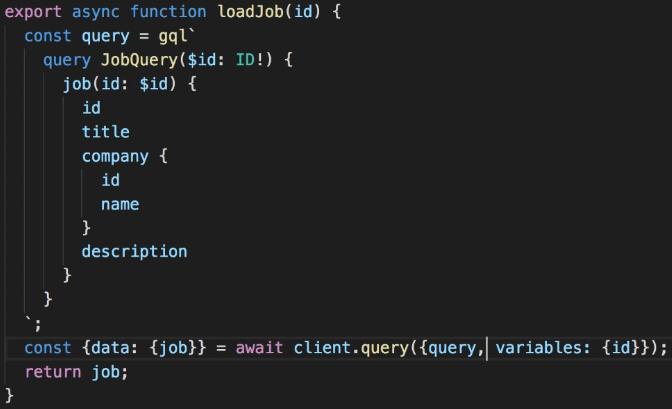
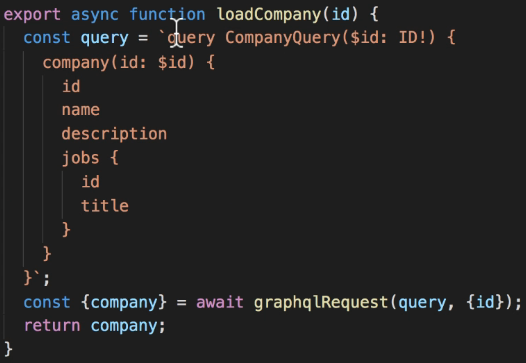
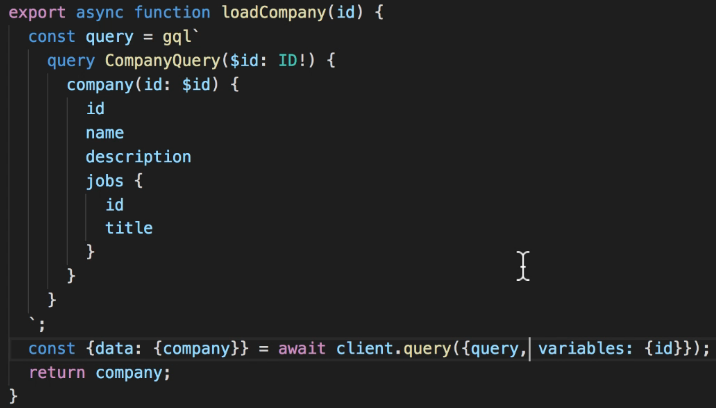
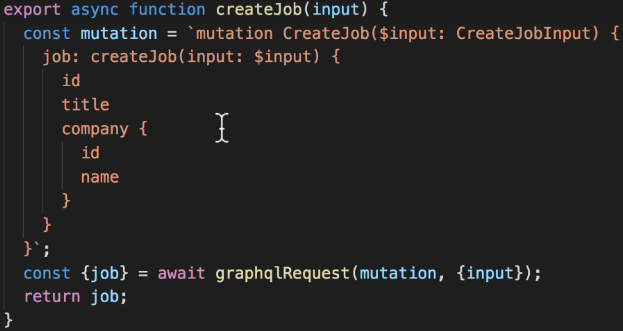
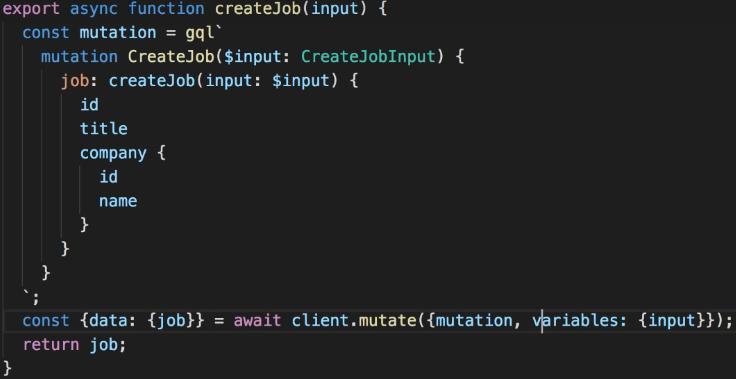
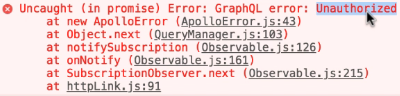
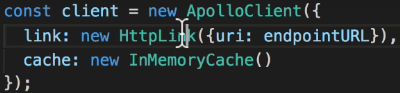
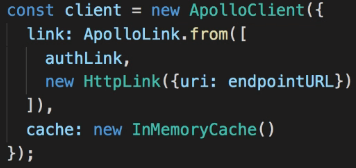
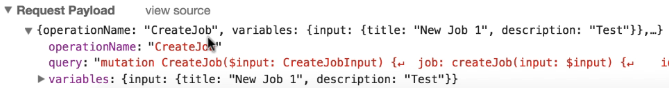
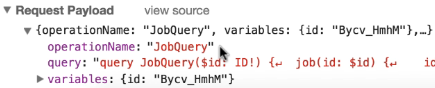
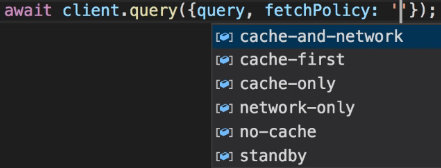
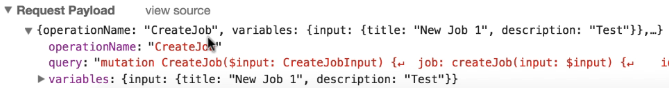
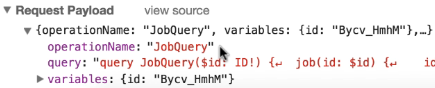
**Apollo Client Setup**  
**Apollo** => **a popular library ot make GraphQL requests and provides some useful features like caching** .  
\* So far in our Frontend application we used our graphqlRequest() function that we wrote ourselves and makes HTTP Requests using the FETCH API.  
\* This works absolutely fine and has the advantage of being simple.  
\* However, if we look at our application and inspect the Network Requests, every time we display a new component like the JobDetail, it will make a new GraphQL Request.  
\* In the Network Tab filter to see only the /graphql requests and not the OPTIONS ones.  
  
\* So to display the JobDetail, it makes a GraphQL request.  
\* That contains the query to fetch a single job in the Request Body.  
\* If we got back to the “Home” view, it makes another request - this time it’s the query to retrieve all the jobs. If we click a specific job again, it makes another request to the server. This is actually the same data we requested before.  
=> **So if we had some sorst of caching in our app, we could avoid making this request and just reuse the data we retrieved earlier**.  
\* That’s where you might consider using **Apollo client** because one of the main features it provides is precisely **Caching**.  
**npm install apollo-boost graphql**  
\* This **graphql** package is the same one we already use in the server, it provides functionality for parsing queries and stuff like that.  
\* Right now when we make a GraphQL request over HTTP, we just send the query as a string to the server but as we’ll see, with Apollo Client, there’s an additional tep that involves parsing the query. That’s why we need the graphql library as a dependency.  
\* **apollo-boost** => a convenience package that brings in a bunch of other dependencies.  
\* We can run this command to see which ones:  
**npm view apollo-boost dependencies**  
  
\* You can see that it includes the apollo-client library and a few related modules.  
\* They’re all required to use Apollo Client so to make things easier, there’s this apollo-boost package.  
**ApolloClient**  
**link** => **how to connect to the server**.  
**HttpLink**  
**uri** => **the server address to connect to**.  
**InMemoryCache**  
**cache** => **cached objects will be kept in memory**.  
\* **There are other implementations available - you can persist the cache to Local Storage for example**.  
\* **Or AsyncStorage if you use React Native**.

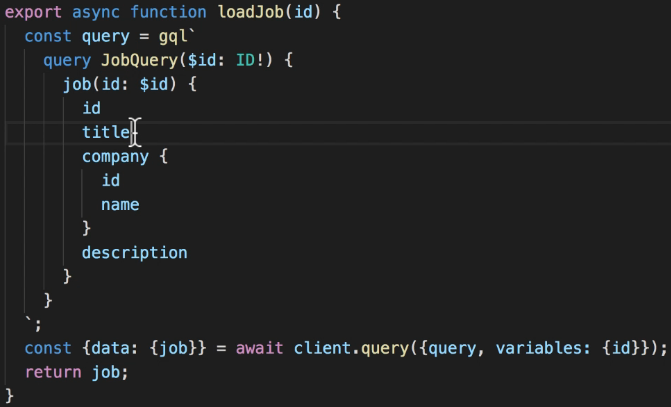
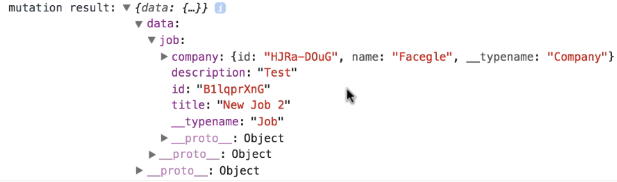
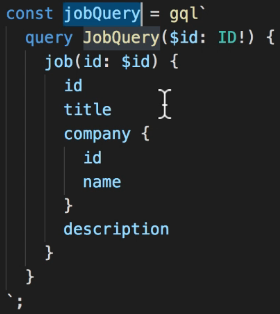
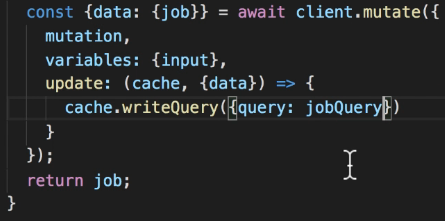
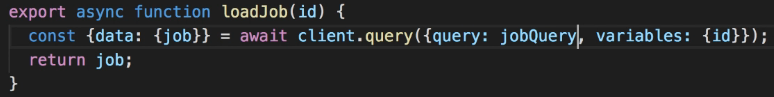
  
\* That’s the minimal amount of set up required to use Apollo Client in our application.

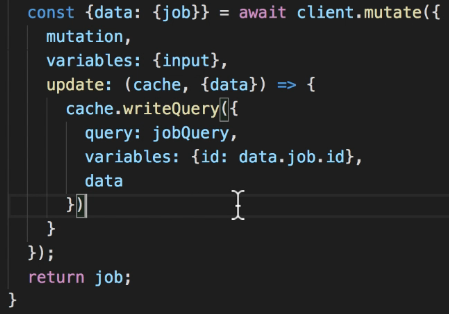
**Queries with Apollo Client**  
\* Let’s start with the loadJobs request.   
**query()**  
\* With Apollo Client we need to send a structured object that is the result of parsing this query using the GraphQL library.  
  
\* We can use this as a **Tag Function**.  
=> **Tag Functions are one of the features in ES6 together with template strings**.  
=> It basicaly means that this template string will be processed by the gql function.  
=> The gql function is effectively parsing this string into an object that represents the GraphQL query.  
\* The query() method returns a Promise that resolves to an object that has a few properties like “data”, “errors”, etc. it’s basically an object representing the GraphQL response.  
\* **We can use Destructuring to change**:  
  
=> **You can use nested objects when destructuring**.  
\* Now let’s update loadJobs.  
   
\* **I’m using an extension on Visual Studio Code that provides GraphQL support**.  
   
**mutate()**  
   
  
  
\* In our graphqlRequest() function if the user is logged in, we set the “authorization” header with the access token proving the user’s credentials.  
\* We currently have nothing like that configured in our ApolloClient.  
\* So that’s something we’ll need to sort out.

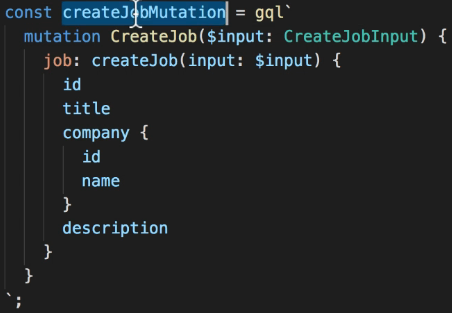
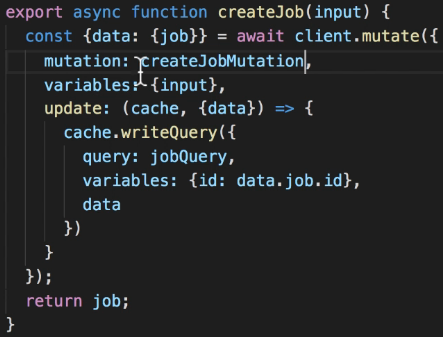
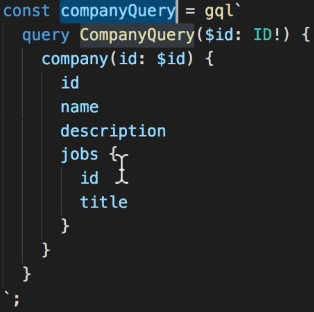
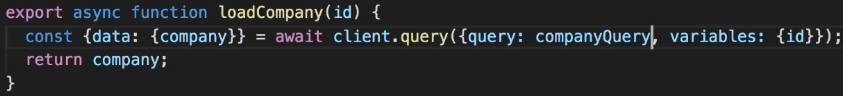
**Authentication with ApolloLink**  
\* The client is configured to use an HttpLink to communicate with the server.  
  
=> We need a way to customize the behavior of this “link”.  
  
**ApolloLink.from([])**  
  
=> This takes as argument a function with 2 parameters:  
**1st operation** => **the GraphQL query or mutation to be executed**.  
**2nd forward** => **a function that allows us to chain multiple steps together**.  
=> **Our authLink code will be executed first**.  
\* By adding our authLink before HttpLink, we can prepare the request before it gets sent.  
**operation.setContext()**  
\* That’s it.  
\* We’ve seen how to customize the Apollo Client behavior to work with our authentication mechanism. We wrote this authLink which is a custom ApolloLink instance and we use it in the ApolloClient configuration with ApolloLink.from([]) that takes an array of link instances and combines them together.  
\* First it executes our authLink that will set the authorization header and then the standard HttpLink that will send the request over HTTP.

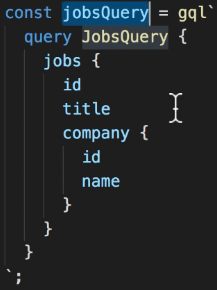
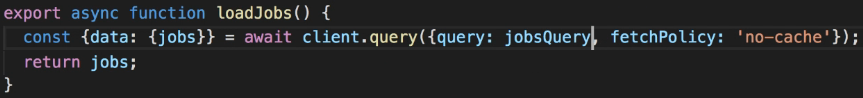
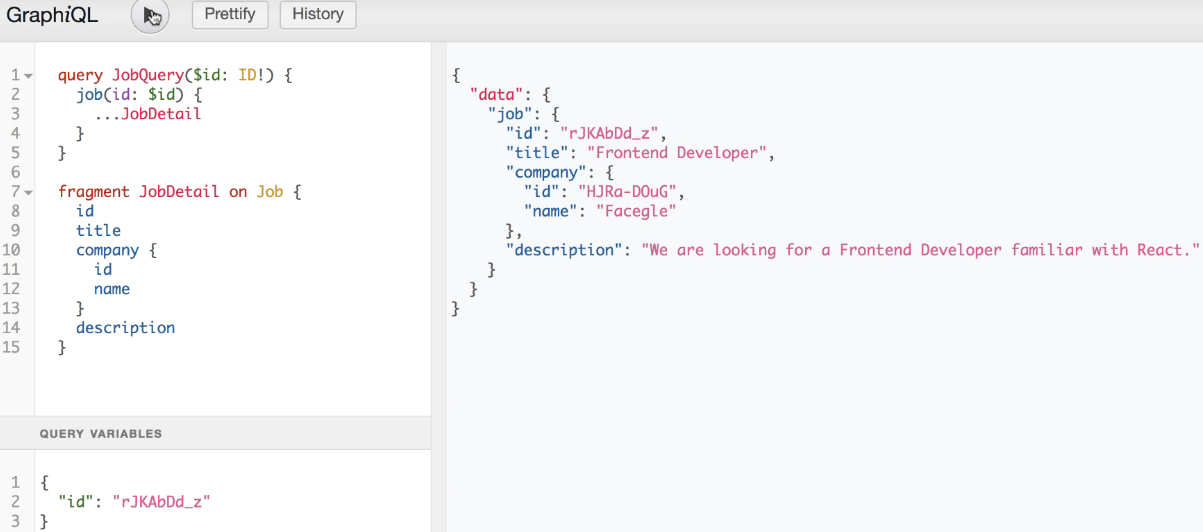
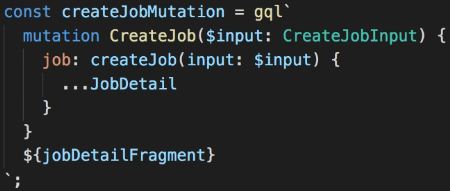
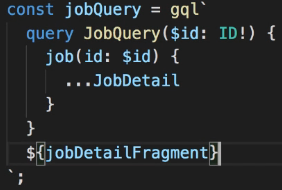
**Caching and Fetch Policy**  
\* **Now if we go back to “Home” page, it doesn’t make any new request**.  
\* Same if we open the same job details again.  
\* The first time we make a certain GraphQL query, it will fetch the data from the server and save it into the cache. If we make the same GraphQL query again, it will simply use the data from the cache.  
\* This is the default behavior with Apollo Client.  
\* **Caching is good because it avoids making unnecessary calls to the server**.  
\* However, caching also adds some complexity to our application. In some cases using a cache can result in some unexpected behaviors.  
=> For example creating a new job results in 2 new requests:  
1. => CreateJob operation  
  
\* Apollo always sets the “operationName” property which is useful for debugging.  
2. => The JobQuery  
  
=> So the 1st request is to create a job and the 2nd request is to fetch the new job details.  
=> This 2nd request is not really necessary because **when we create a job, in the response we already get the job details back so there’s no need to go and fetch the same data again** in a separate server call.  
\* **Apollo Client doesn’t cache this job data by default because this is the result of a Mutation**.   
\* We’ll need some special code to handle this case. We’ll see that in the next video.  
\* Let’s look at another strange thing:  
\* If we go back to the “Home” page, it doesn’t make any new request because it already has the jobs data in the cache.  
\* **But that also means that our new job is not in the list, it’s showing the cached data retrieved before we added the new job**.  
=> **You might be using old data**.  
\* If we reload the page, we see the new job because it makes a new call to the server.  
\* Sometimes it’s good to cache data because it makes your app respond faster by not having to call the server as often.  
\* But sometimes it’s actually better to always fetch fresh data even if that means waiting for a server response.  
\* **It’s possible to customize the caching behavior of Apollo Client**.  
**fetchPolicy => cache-first is the default, only if it doesn’t find it in the cache, it calls the server.**  
  
 **no-cache** => **This will always fetch the data from the server**.  
\* The list of jobs should now be reloaded every time.  
\* If we open the same job details again, it DOESN’T make a new request because it already has this job in the cache.  
\* But going “Home” will fetch the jobs data every time..

**Updating the Cache After a Mutation**  
\* **fetchPolicy** => The 1st way to customize the behavior of the cache used by Apollo Client.  
\* **There’s also a more advanced and flexible approach that involves manipulating the cache directly**  
**----- in the last video -----**   
=> For example creating a new job results in 2 new requests:  
1. => CreateJob operation  
  
\* Apollo always sets the “operationName” property which is useful for debugging.  
2. => The JobQuery  
  
=> So the 1st request is to create a job and the 2nd request is to fetch the new job details.  
=> This 2nd request is not really necessary because **when we create a job, in the response we already get the job details back so there’s no need to go and fetch the same data again** in a separate server call.  
\* **Apollo Client doesn’t cache this job data by default because this is the result of a Mutation**.   
\* We’ll need some special code to handle this case. We’ll see that in the next video.  
**----- in the last video -----**   
\* **If we look at the code for createJob Mutation, you can see that we get back a job object with ID, title and company.** **\* While in the loadJob Query, we get pretty much the same data - a Job with ID, title, company and actually the description => so let’s add the description to the mutation so that we use exactly the same fields in both places, there’s a better way to reuse a group of fields like this without copy+paste.**

=> GraphQL supports **Fragments** - we’ll see that in the next video.  
  
\* What we want to do now is avoid making the 2 requests to the server every time we create a new job. Because when we send the mutation, we already get back the new job details.  
update => gives us pretty much full control over the cache - it’s a function that will be called after the mutation has been executed. It receives 2 parameters.  
**1st** => “**story / proxy / cache**” - **it’s an object that lets you manipulate what’s stored in the cache**.  
**2nd** => “**mutationResult**” - **it’s the response we get from the server when we send this mutation**.  
\* Let’s just console log it so that we can see what it looks like:  
  
\* After posting a new job:  
  
=> So the mutationResult basically contains the data in the GraphQL response.  
\* We can use destructuring and let’s save the newly created job into the cache so when we display the job details on the screen, Apollo can find the data in the cache.  
**writeQuery()** => **can be used to save the result of a query, it takes an object with some properties**.  
**query** => **this would normally be the query that generated the result** - **and by normally I mean whenever we run a query with Apollo Client unless we set special options like “fetchPolicy”**.  
=> after executing a query, Apollo Client normally calls this writeQuery() method, passing the query and the data it received as reponse.  
=> It this case, however, we’re doing something special - we want to update the cache with the data returned by the mutation, but we want that cached data to be used whenever we make a query to load the same job.  
=> So the query we want to cache is the one in the loadJob function. When we try to load a job that’s just been created, we want Apollo to use the job data in the cache. So this is the query that should be cached even though we store the data from the mutationResult.  
\* The small problem is that the loadJob queryt is a local variable of that function but we need to use it in the createJob mutation function. So let’s copy that query outside of the function.  
   
  
\* **This way we can be sure that the query written to the cache after the mutation is the same one used when loading the job because it’s the same variable**.  
\* We also need to specify the variables associated with the query.  
**variables** => **When we load a job, we pass the ID as a query variable so when writing to the cache, we need the same variables**.  
**data** => **the data to be saved, the value will be the value of the “data” parameter passed to this function, so data: data**.

  
\* To recap, what we’re doing with this “update” function: we tell Apollo Client: whenever you run this mutation, take the data returned in the response and save it to the cache as if it was the result of running the jobQuery for that specific job ID.  
=> **This way when we actually run a jobQuery with that job ID, it will find the data in the cache and avoid making a new call to the server**.  
=> Now when we post a new job, there’s only 1 request.  
=> If we go back to the “Home” page, it makes a request to show all the jobs.  
=> If we then go to the newly created job details, it doesn’t make any new calls to the server.

**GraphQL Fragments**  
\* **For consistency, we could move all the queries and mutations to top-level variables**.  
   
 

   
\* That’s it, that was just some refactoring to make our code a bit cleaner.  
**GraphQL Fragments** => **lets us reuse the same group of fields in different queries**.  
=> In createJob mutation and jobQuery query we use exactly the same fields.  
=> So we duplicate the same bit of GraphQL in 2 places.  
**fragment name on Type** => this is required because a fragment is a group of fields but each type has different fields, in this case we want to select some fields from a Job.  
**…fragmentName**   
  
\* I’ll put the fragment above all the queries and mutations.  
   
\* **We also need to include the fragment definition in the mutation**.  
=> This way the fragment definition will be included at this point.  
=> Note that this will not do a simple string substitution like with regular template literals. The jobDetailFragment variable is not actually a string, but a parsed GraphQL fragment. We can use it as an expression here because we’re using the gql Tag Function.  
  
\* Now if we ever need to select an additional field for example, we’ll just need to make the change in one place.