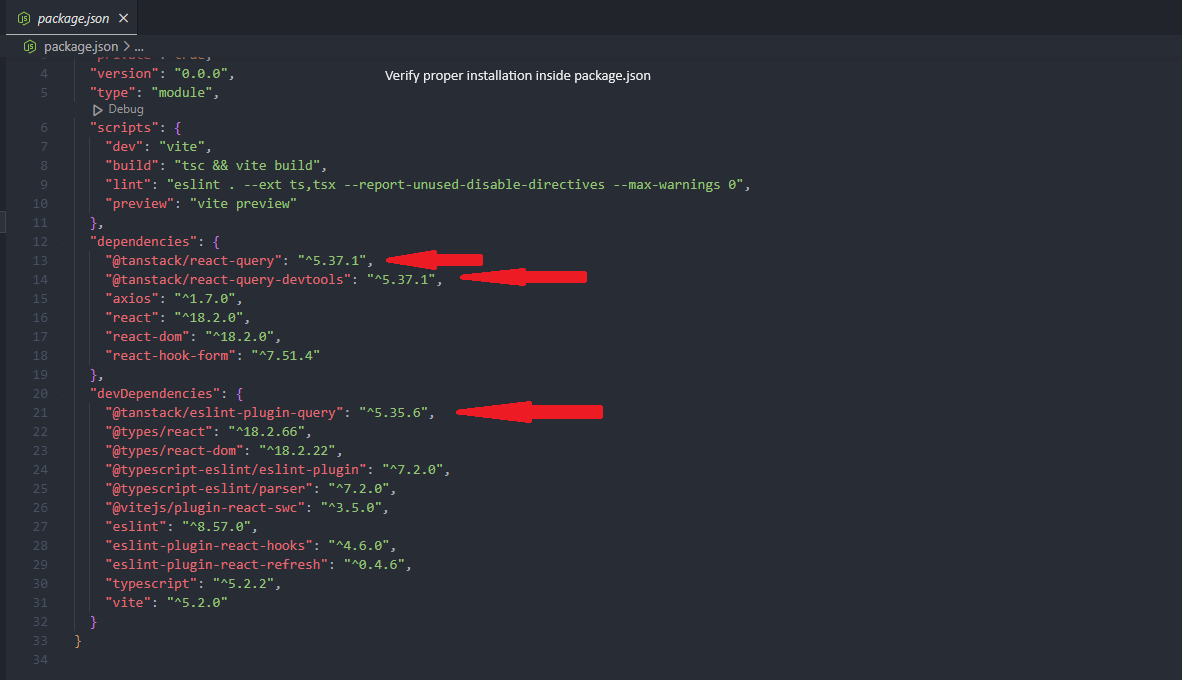
**React query (Tanstack query)**

Benefits:

1. Caches our backend data inside a custom hook that returns us required variables that are useful for client notifications out of the box (such as isFetching, isPending, isError and many more).
2. Saves us backend requests by letting us give its queries a ‘key’ parameter, this key parameter is an indicator that keeps the first time-fetched data in our caching system, as long as it hasn’t been invalidated (Will be explained in much more detail later).

Installation-

1. To install react query we need to npm install / yarn add @tanstack/react-query @tanstack/react-query-devtools.
2. After that we need to npm install /yarn add @tanstack/eslint-plugin-query -D as a dev dependency for a better auto-completion and error detection.

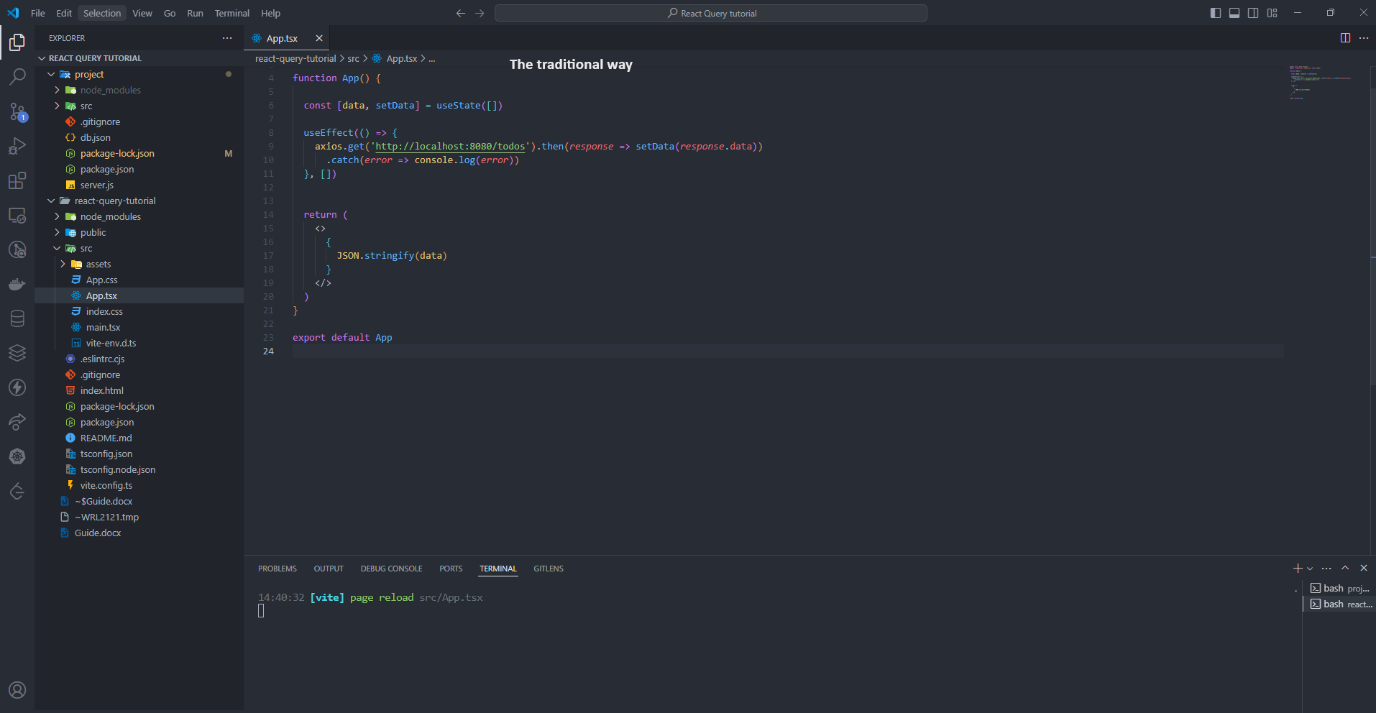


1. After we installed the es plugin we need to add it to the extends array inside our .eslintrc.cjs and also add “@tanstack/query” inside the plugins array

A screen shot of a computer program

Description automatically generated

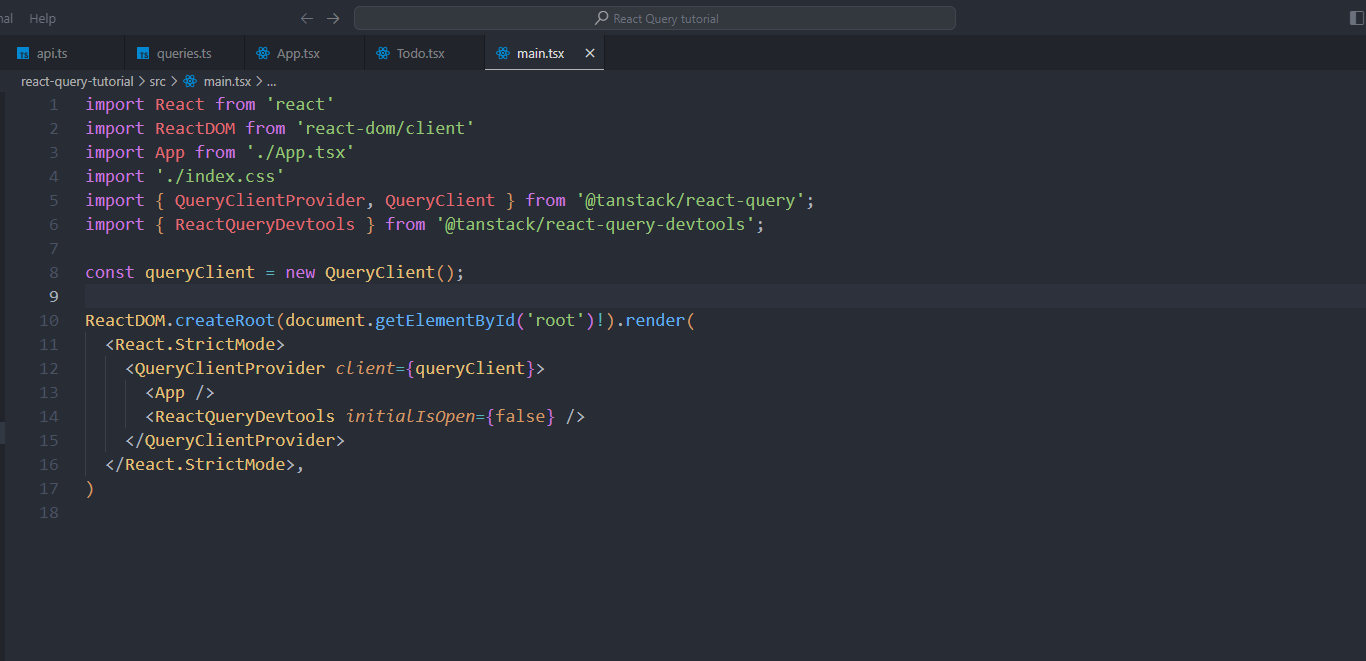
To see what react-query is about we shall start with the traditional way of fetching data from our server to our client app, so I am going to fetch a list of dummy data in the form of a to-do list from a fake backend that’s listening on port 8080.



(I know this is not how we are actually fetching data in our app, we are currently using async thunk in redux, but we are about to see that react query is even more convenient than using async thunk.)

One more step before we start writing queries:  
We need to make sure that QueryClient gets passed to our application and our application is wrapped in a special context so let’s go to main.tsx and wrap our App component with the required context:

1. Import QueryClientProvider and QueryClient from ‘@tanstack/react-query’
2. Optional: import ReactQueryDevTools from ‘@tanstack/react-query-devtools’
3. Instantiate QueryClient class (it can receive [configuration arguments](https://tanstack.com/query/latest/docs/reference/QueryClient) for default options)
4. Pass the newly created queryClient instance as “client” property to the QueryClientProvider
5. Optional and depended on step 2, if we decided to use ReactQueryDevTools, we should include it inside the QueryClientProvider as well



So how does a react-query api query look like?

Another nice thing about react-query is that it is just a wrapping tool for your fetching system, which means we can keep using Axios (or fetch if someone wanted to).

In the next example I am going to fetch an array of the todos ids. A Todo interface is comprised of

export interface Todo {

    checked: boolean;

    title: string;

    description: string;

    id: number;

}

First step: Will be creating the API itself using Axios:

A screen shot of a computer program

Description automatically generated

Second step: will be wrapping the api query we just wrote in an application query function.

A screenshot of a computer

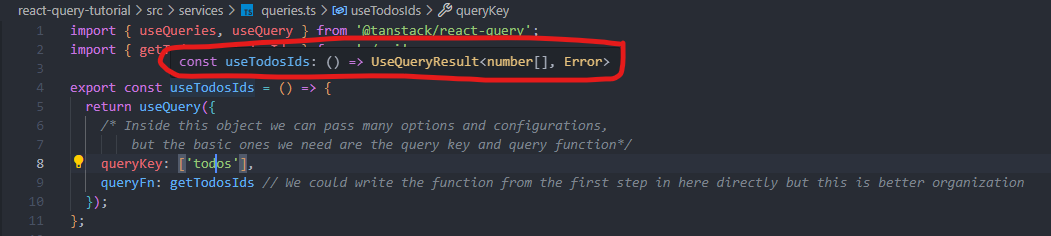
Description automatically generated

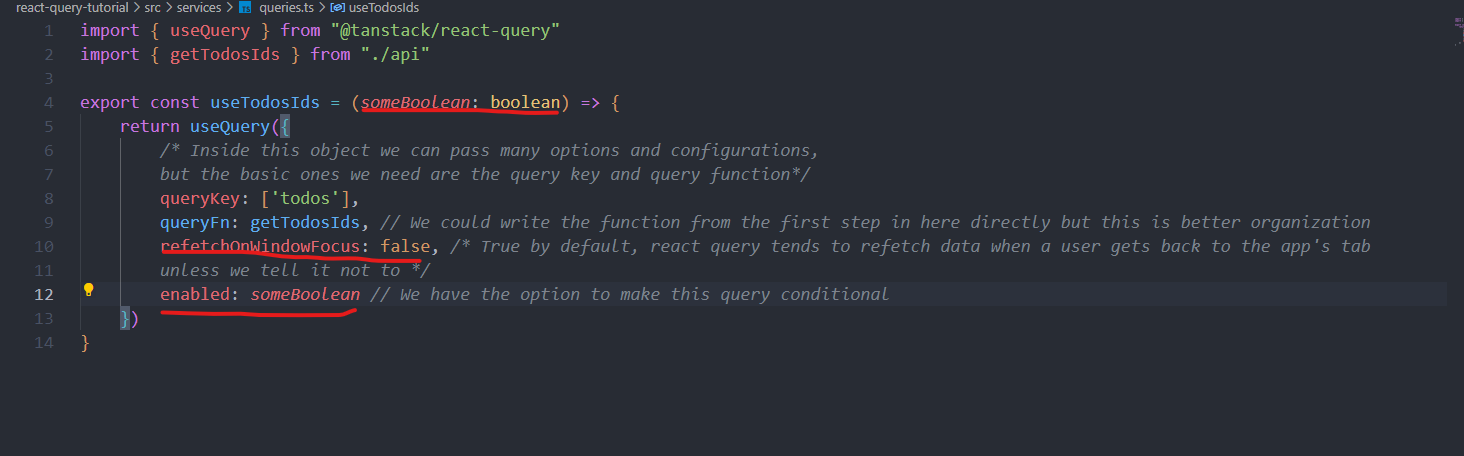
The query key is an array of arbitrary strings (or numbers, or objects, but mostly strings) that we choose, we can choose any string we want as long as it’s not in use in another query function (if it is in use by another query function a [mutation](https://tanstack.com/query/latest/docs/framework/react/guides/mutations) will invalidate both queries), later when we use mutations and invalidate queries, we’ll see the true importance of this key string (Invalidation in short means telling react-query to re-fetch the data once again from the backend and update its cache because our data base has changed [And hence the word “invalidation”- it means our cache is no longer valid]).

QueryFn is pretty self-explanatory.

More points:

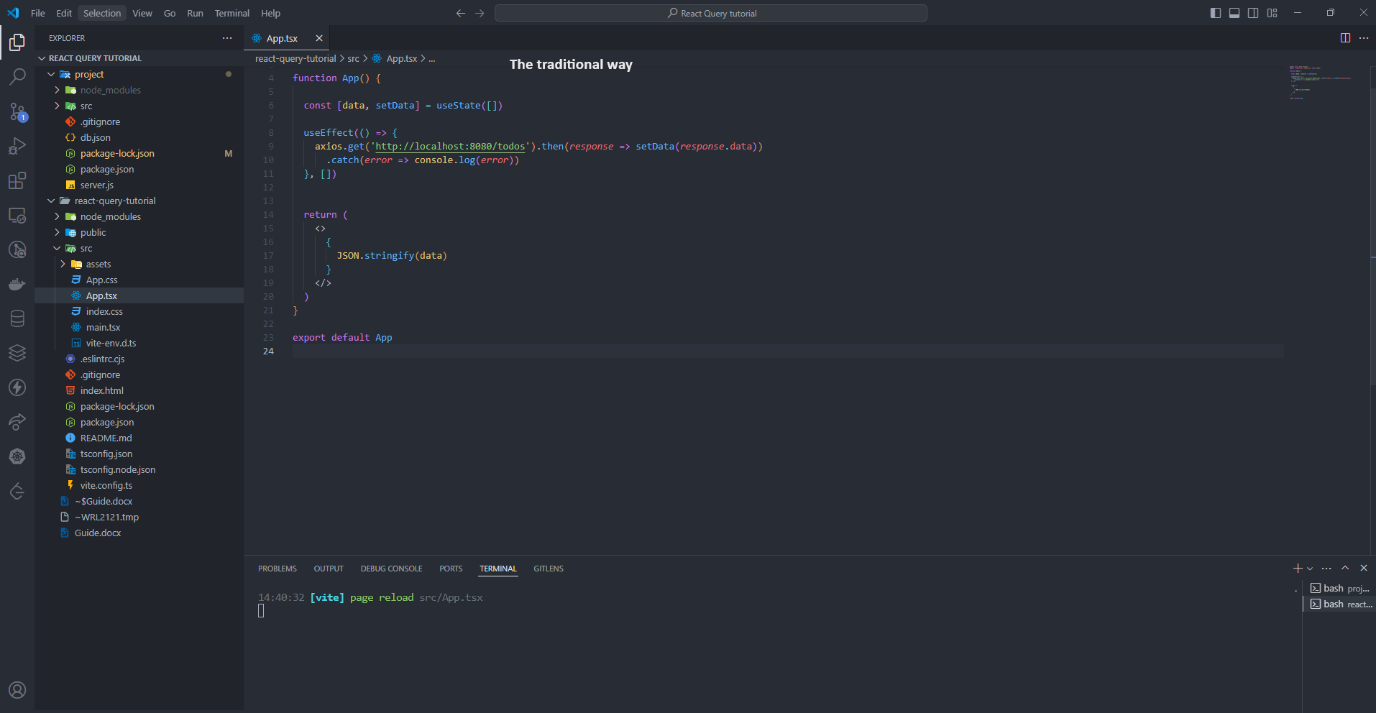
1. You probably saw that I wrapped the inner useQuery in a custom hook called useTodosIds, that’s for the simple reason that hooks can only be called inside jsx/tsx functions or inside other hooks. We can also use useQuery and useMutation inside the components themselves, but it is hard to track and can make us repeat our code.
2. React query is intelligent enough to tell what is the return type of the query function without us having to explicitly tell it what it is



Some examples of other highly used query properties are these (I am going to delete them, adding them was just for demonstration purposes):  


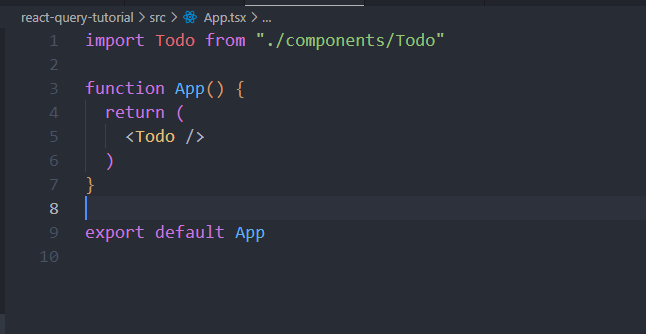
Using our newly created query function inside our app:

Coming back to our app.tsx file, it currently looks like this:

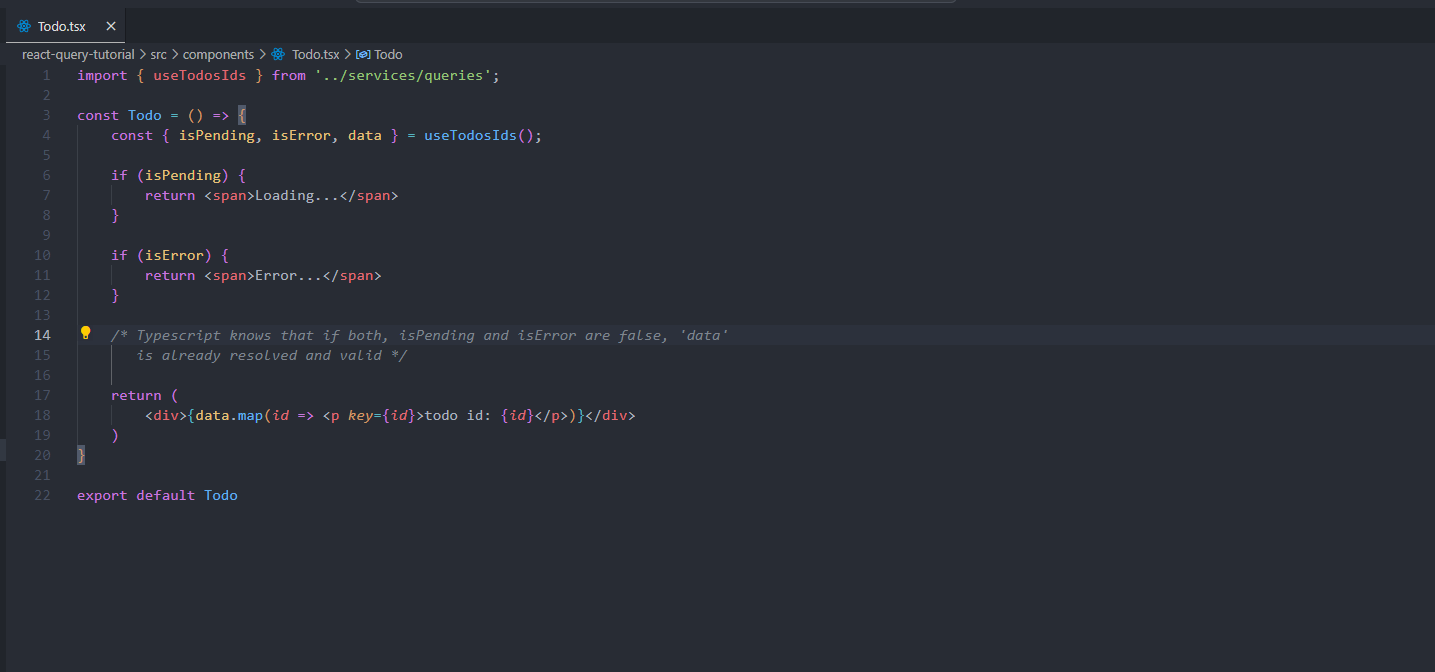


Notice that we don’t even include a loading state, a failed state or caching system here.

For better organization, I’ll move the logic of the query fetching inside a component named <Todo />



Inside our Todo component we can see now the properties that react-query gives us (Benefit number one at the beginning of this document):



Now if we are opening our app in the browser we can see that the query works.

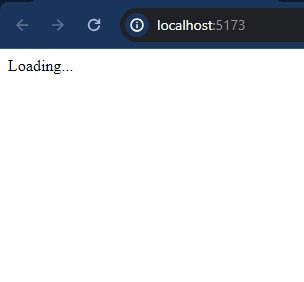


Figure 1First, we get the status on pending because the request is not resolved yet

A screenshot of a computer

Description automatically generated

Figure 2Then we get the list of the todo ids!

And just for demonstration purposes, I changed the Base\_URL to 8081 (a port that no one’s listening to) and got isError = true, so we got this:

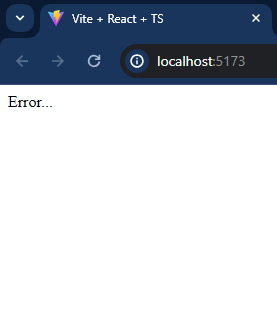
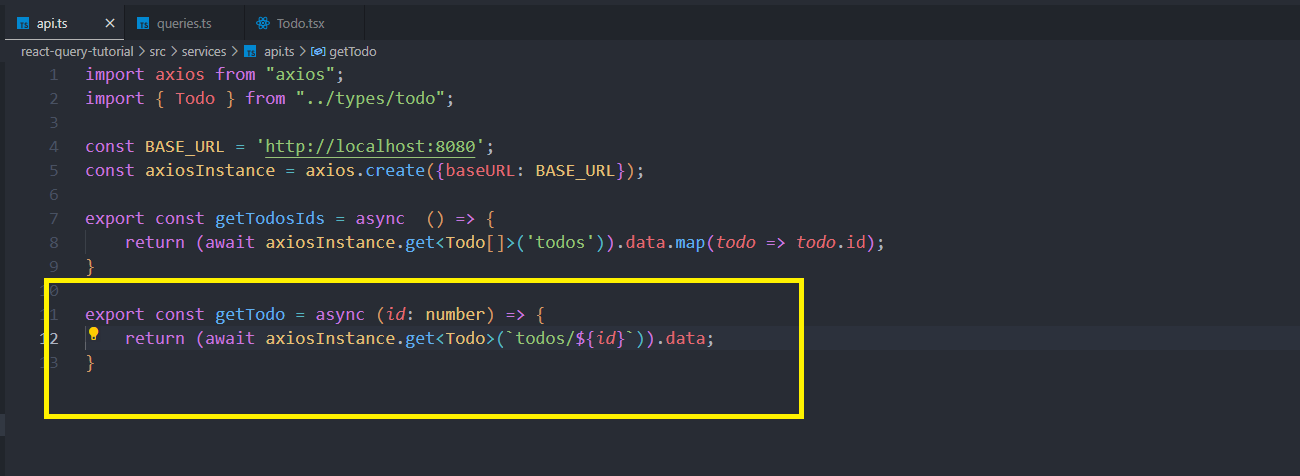


Figure 3 Intentional error caused us to get isError = true

Another side note: React query also provides us with a custom hook called useIsFetching, this hook returns us the amount of query functions that are fetching at the moment (We will definitely have more than one query function in our app).

Moving on to our next challengs- Getting a todo by using a todo.id:

* First, Let’s go to services/api.ts (The file where we define axios requests) and create a new function.



* Now let’s go to queries.ts file (where we wrap our axios functions inside react-query functions and wrap our getTodo.



T=This time I need you to notice something new, I am using a hook called useQueries (plural), while in our first query function, I used a hook called useQuery (singular). The difference between them is that useQueries is used for fetching data for multiple queries simultaneously (also known as run parallel queries). We give this function an input of array of any kind, and get as a result an array of queries. Instead of having one query that its result is an array of data, we get every array element as a query (with the element itself as data). This process happens in parallel and is good for scenarios where we need to call a query ‘inside a loop’ which is something we are not allowed to do in react.

So why not take an array of input, map it and call a query inside the map?

users.map(user => {

useUserProfiles(user.id);

}) // That won’t work because hooks can’t be called inside a map

But we could also define a useQuery that fetches all of the data and then filter this data on the client side inside the component itself.

That’s true, but this approach has three major downsides:

1. **Performance Issues**: If the amount of data is large, fetching all the data at once can lead to performance issues, such as increased load times and higher memory usage.
2. **Scalability**: This approach doesn't scale well for large datasets. Fetching all data can become impractical and inefficient.
3. **Data Freshness**: Keeping the data updated can be more challenging. You might need additional logic to handle data updates and invalidation.

A nice rule of thumb for fetching all the data and filtering it in the component vs using useQueries would be:

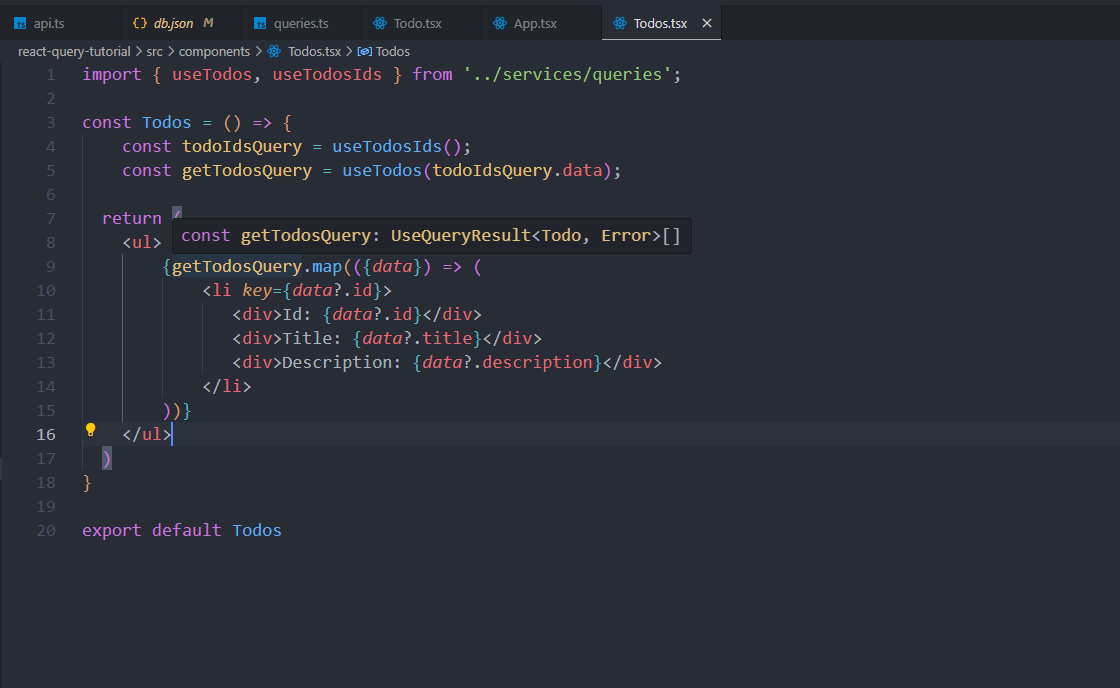
* **Fetch All Users**: Use this approach when the dataset is small, and you need to filter and display a significant portion of the data frequently. It is also useful when you want to reduce the number of network requests.
* **Use useQueries**: Use this approach when dealing with large datasets, when you only need specific pieces of data, and when you want to ensure data freshness and efficient data management.

Okay, now back to our code:

This marked function returns an array of query results, not just one query result UseQueryResult<Todo, Error>[]

* Then we can use this query function on our different components:

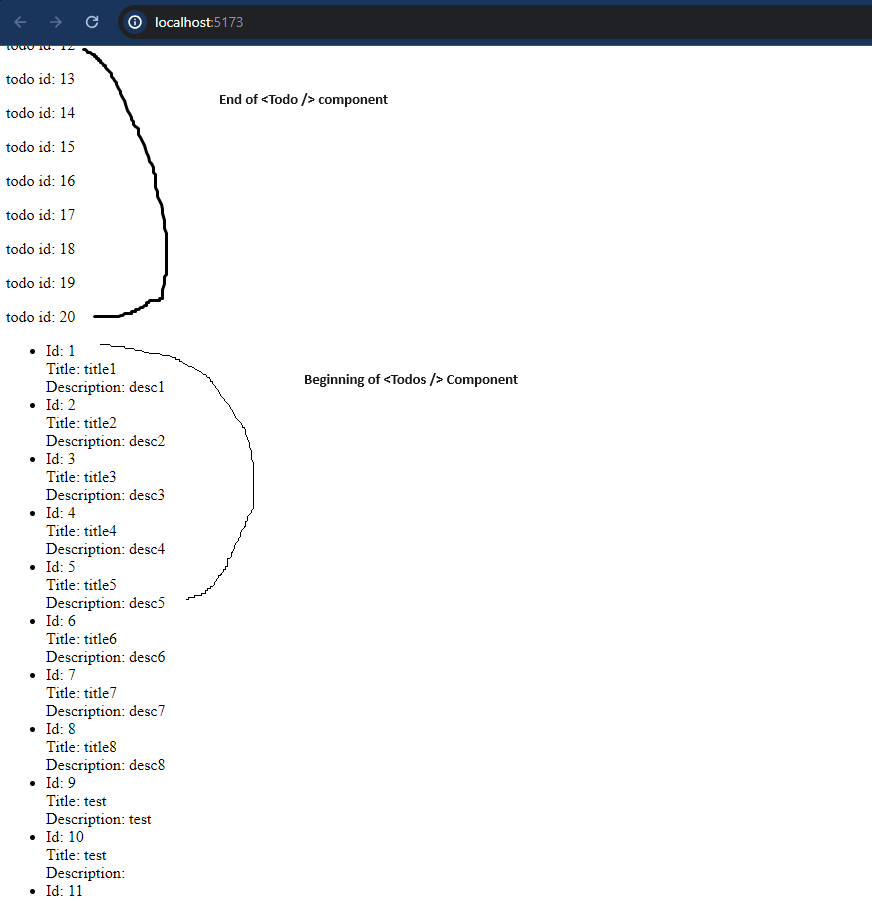
Let’s create a new component that’s called <Todos>, there we will use the first query we wrote to get an array of ids and pass this array of ids to turn them into an array of todo objects.



* Now we just add <Todos /> to our app component:

A screenshot of a computer program

Description automatically generated



The image above can’t show it but the list items were fetched one by one and in parallel, which is better for performance (than waiting for the whole list and in the meantime looking at an empty page or a loading spinner). Another benefit is that we have more granular control over each of the queries that were back from useTodos, we can invalidate them individually if we need to. Let’s say we want to edit just one of the todo objects we just fetched and change its title, if we do it now, the only query that will need to refetch is the query that belongs to this specific todo (notice the queryKey we used inside useTodos

queryKey: ['todo', *id*],

)

Instead of invalidating a big query that fetches all of the todos and now needs to refetch them just because one todo has changed.

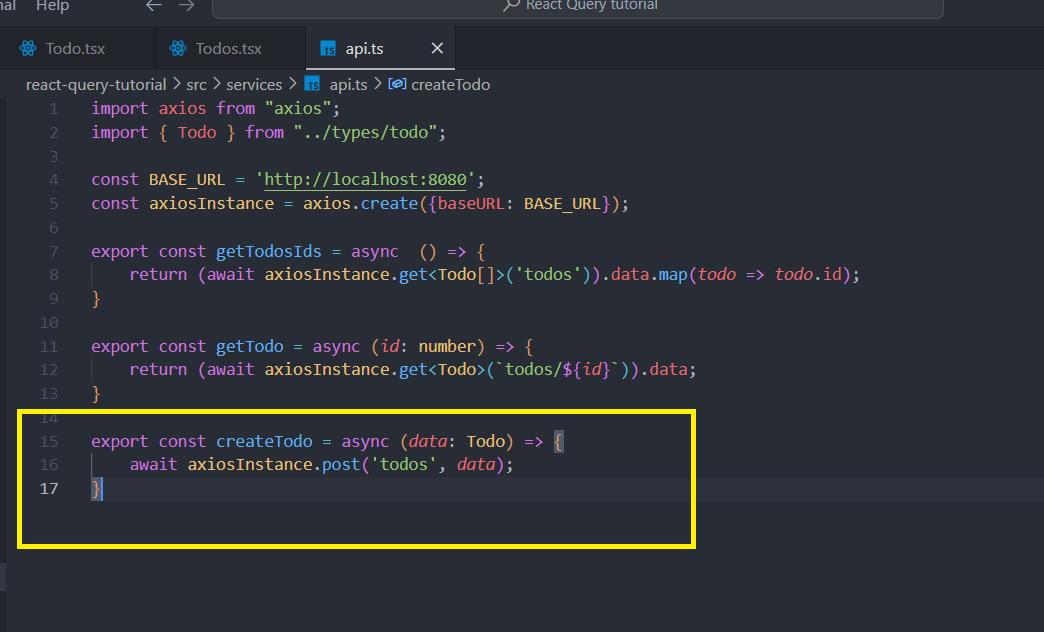
**Open react-query devtools and show them all the newly created queries**

Moving on to using mutations!

Mutations are, in simple terms, actions that effect our database entities (fetching does not effect or mutate our database entities, just brings them to our client side), actions such as deletion, addition or edition do it, so for this type of actions we should use mutations.

Let’s demonstrate this with adding a new to-do to our list

* 1. Go to our services/api.ts file for defining a new axios function

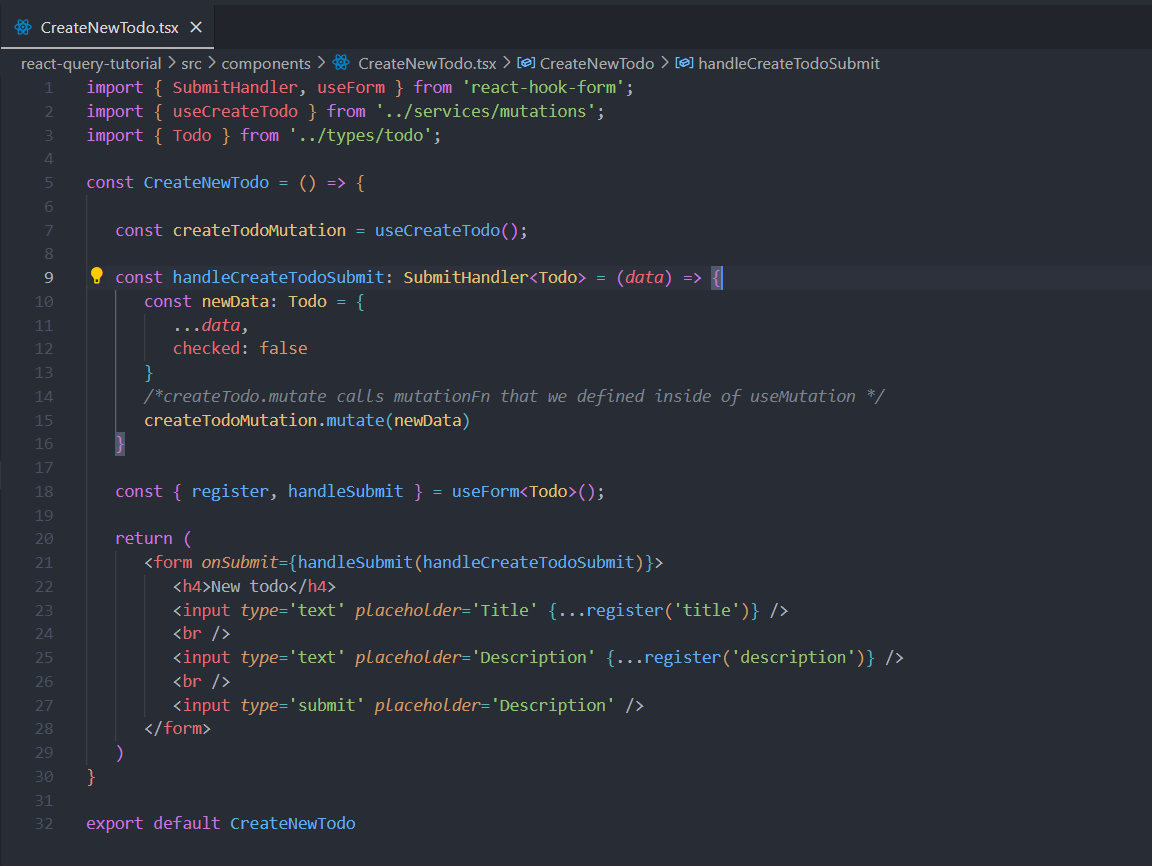


* 1. In the services directory let’s create a new file called mutations.ts
  2. Let’s write our hook that returns useMutation and import useMutation from ‘@tanstack/react-query’. useMutation, as wll as useQuery and useQueries, is configurable, we can intercept its data and make extra steps in different states of the mutation:

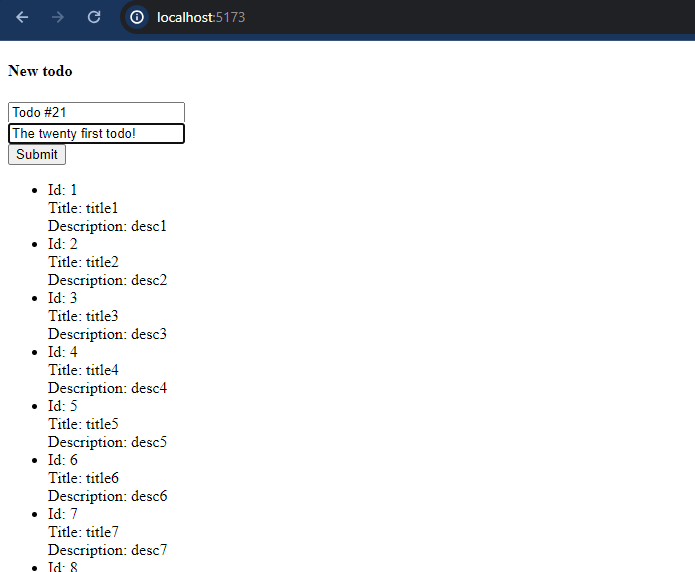
A screenshot of a computer program

Description automatically generated

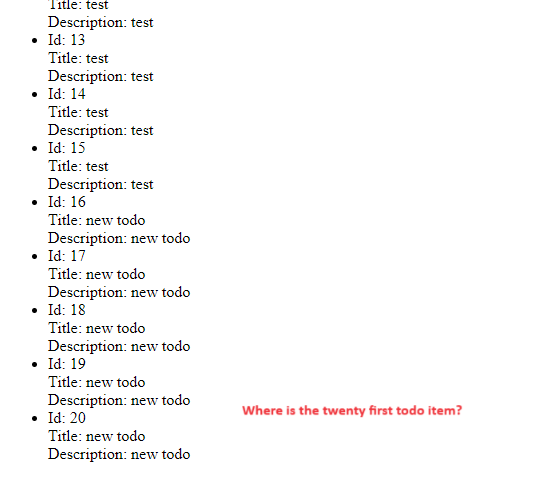
* 1. Time to go to our <Todo /> component, we also need to change its name from <Todo /> to <CreateNewTodo /> because we are going to use the Todo interface there, and using an interface with the same name of the component function causes typescript error. After that we are going to delete its current content and make it a form that creates a new todo and then calls our just created useCreateTodo hook.



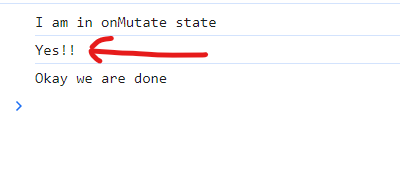
Now let’s get back to our app and insert a new todo



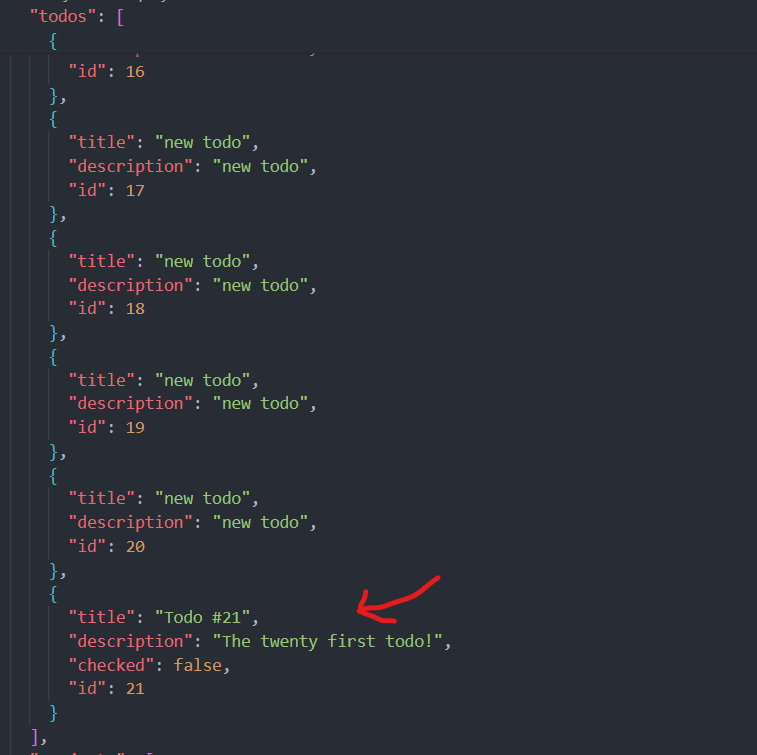
Okay, after inserting the twenty first todo we can see that nothing new was rendered on the list



Okay let’s check our console to see if an error was thrown or if we hit onSuccess:



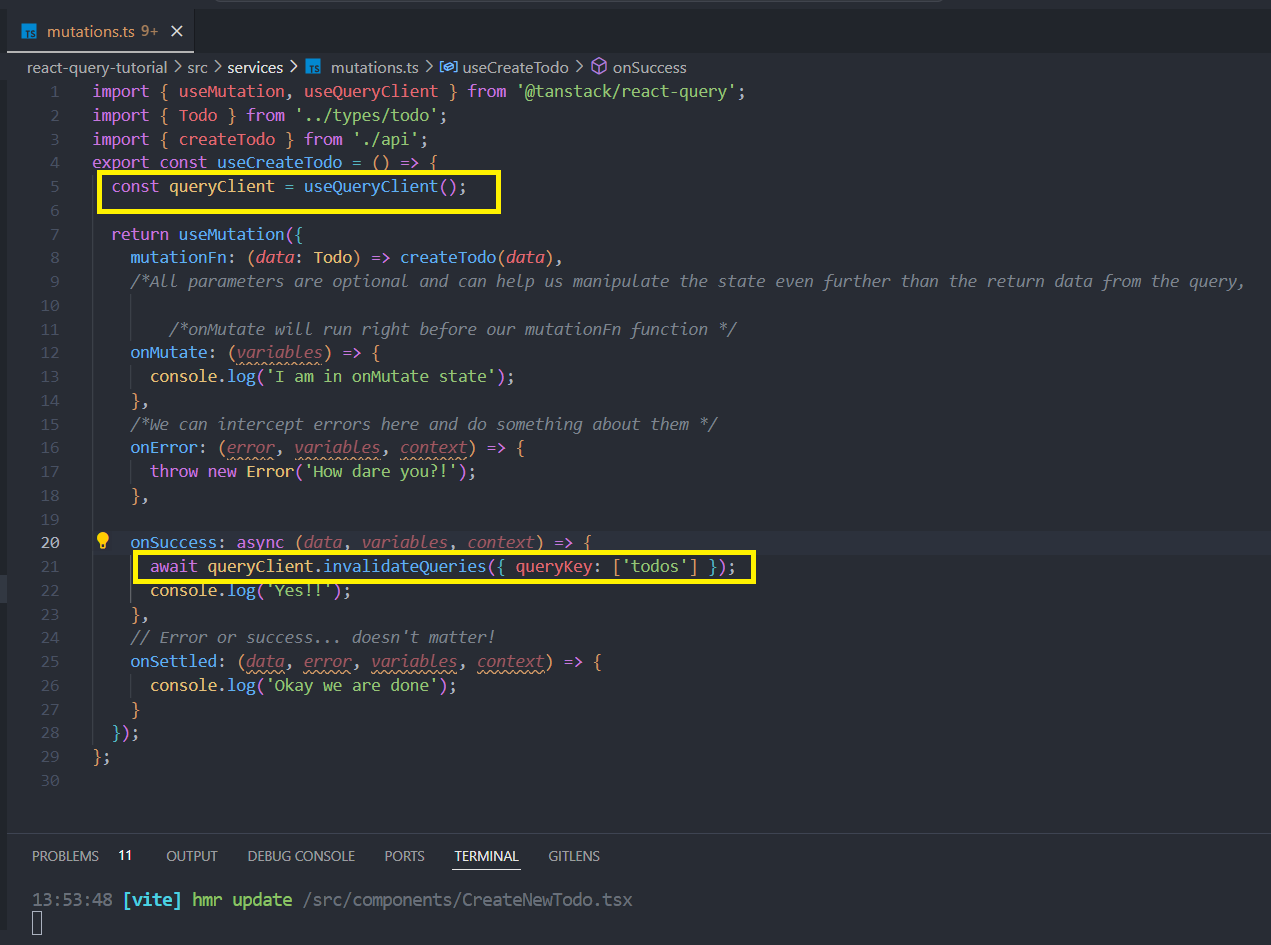
Okay, we hit onSuccess, so why don’t we check out our fake database to really make sure the new todo was saved?

Okay it’s in our database, so how come we don’t see it on our app (unless we refresh the page)?

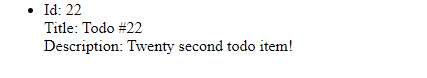
Let them answer

The answer is- Invalidation, our query function that fetches all the todo ids (which has the query key of “todos”) was mutated, there is a new entity record in our database as we just saw. But we never told react-query to invalidate this query, and as smart as react-query is, it is still not smart enough to guess which of the queries should be invalidated, it is our responsibility as programmers to explicit that:

We need to import useQueryClient hook to manipulate our caching system, and tell it to invalidate the query with the “todos” query key if the mutation was successful (There is not point of refetching the data from the database if there was an error and it wasn’t updated, right?) By the way, invalidateQueries is an asynchronous action so we need to turn our onSuccess method into an async method.



Getting back to our app, let’s try to add to-do number 22 and see if now it renders on the screen (almost) seamlessly.



It does! By the way, pay attention to how the new list item gets added to the list, because we used useQueris to fetch the queries item, they stay stale and only the new list item is added, that’s a great user experience. In other words, the user will see the previous cache state snapshot, until the new data is fetched in the background.

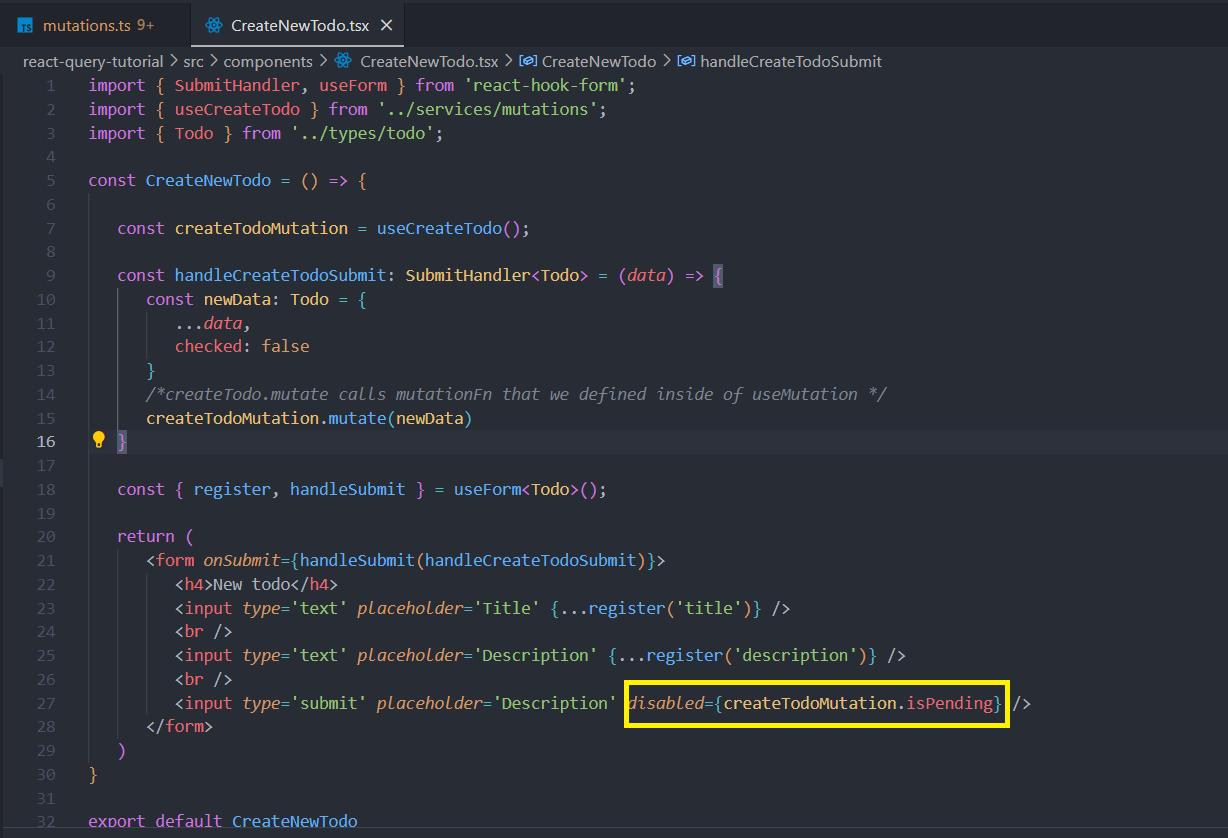
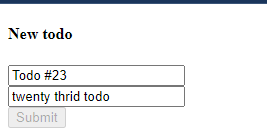
Say we have a scenario where one of our users is an idiot and likes to hit the submit button as much as possible, by doing that they are sending a mutation while a previous mutation is still running:  
the solution to this problem is as simple as could be.  


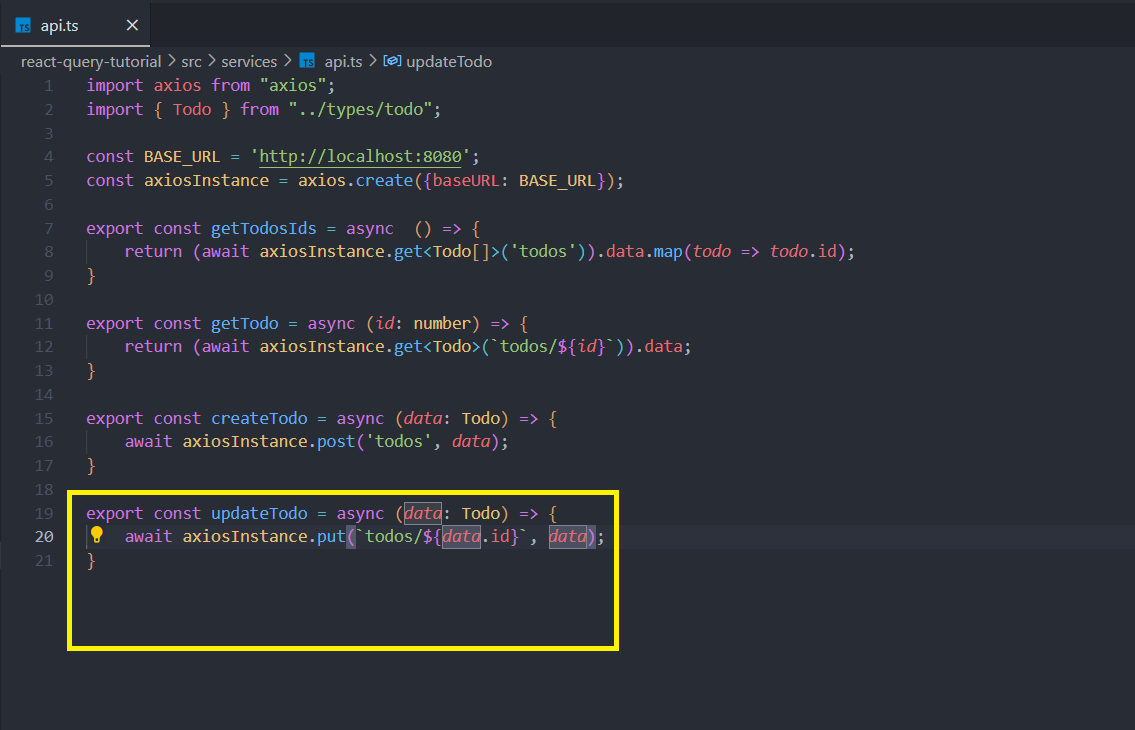
Figure 4 A better solution would be using isSubmitting by useForm but this is just for showing the capabilities of react query



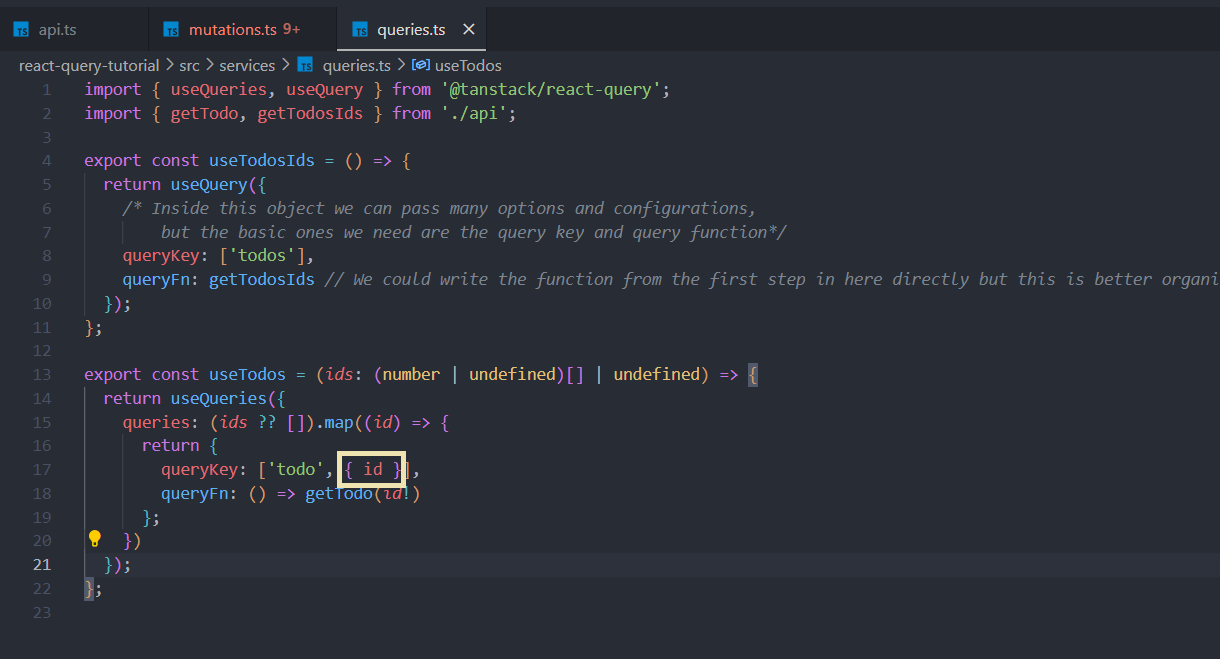
Let’s move on to updating a todo:

When we create a new todo, we add a new item to our previous fetched todos’ list, but when it comes to updating a single todo, because we used useQuery and split every todo item into its own query, we can invalidate just this specific todo query instead of the whole list (I already mentioned it in this document, I know). So, let’s get to it:

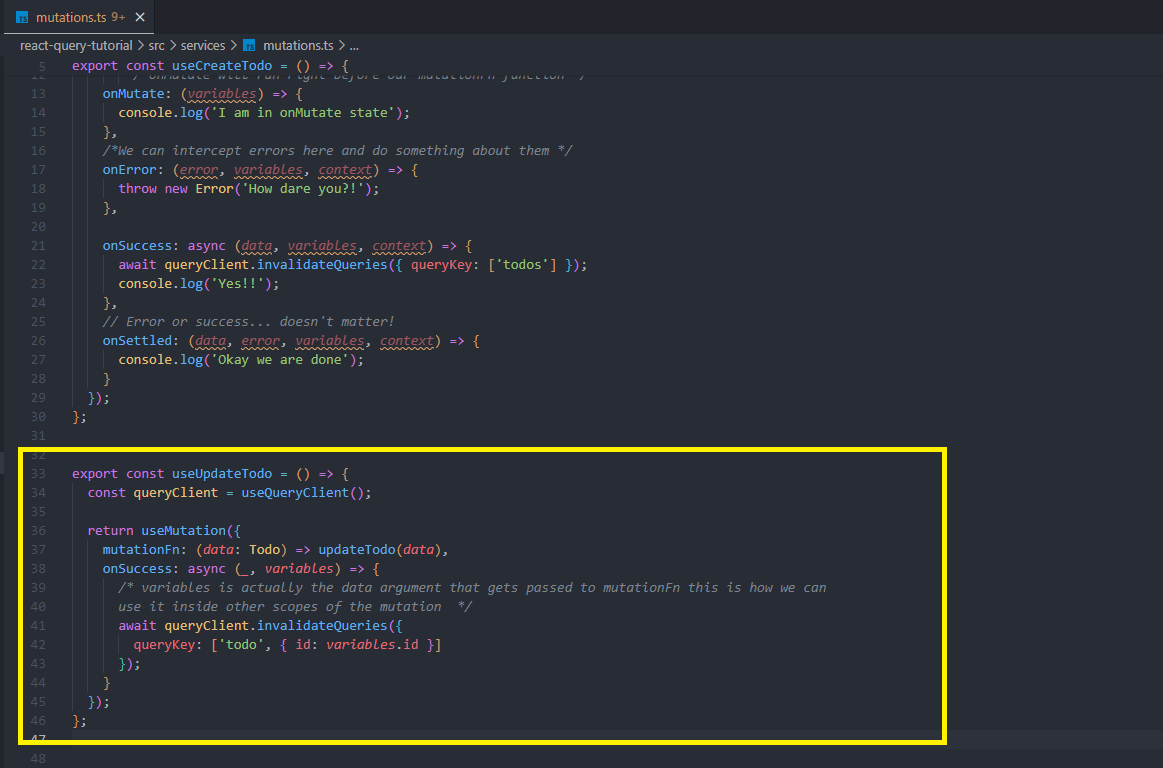
1. As usual, let’s start with defining a new axios function inside api.ts



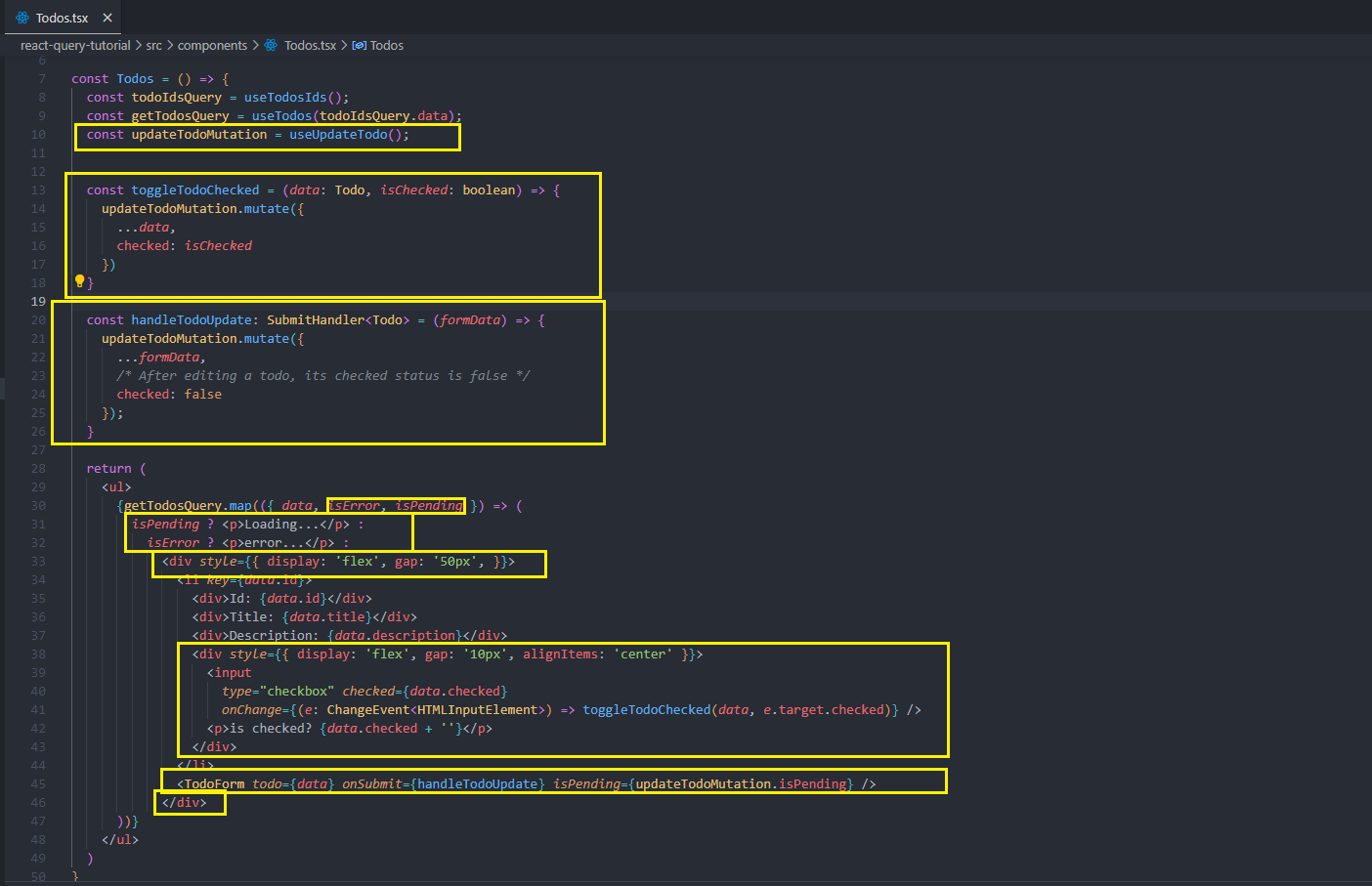
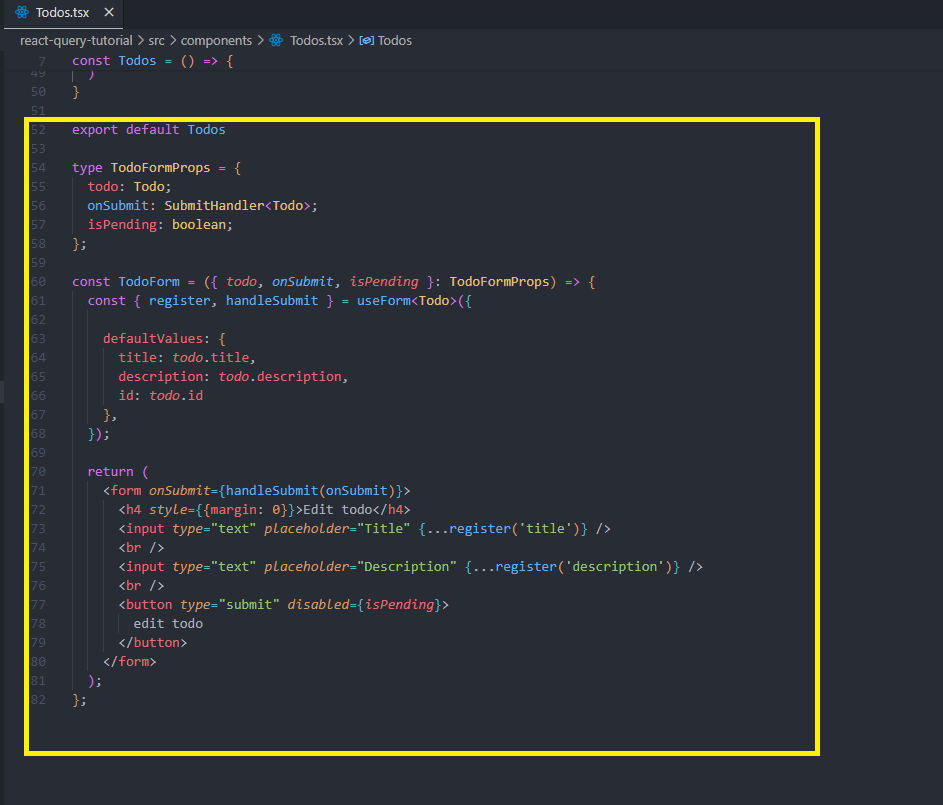
1. We should make a small modification in our useTodos useQueries function. For clearer code readability we should put the second queryKey parameter (id) inside curly braces. **['todo', { id }]** clearly shows that the second part of the key is an object with an **id** property, making it easier to understand what the key represents at a glance (It is not a must though).



1. Time to use this api function inside a useMutation, so let’s move to mutations.ts and create a new mutation, this mutation only effects useTodos, because the id doesn’t change so the query that belongs to useTodosIds does not change after the mutation runs (it stays the same array of todo ids),



4: Use our mutation hook in our component. For that we will go to our <Todos /> component and implement two new functions, one of which is responsible for changing the title or the description of our todo, and the other one is an implementation of a checkbox, which when clicked on changes the status of a todo.checked (from true to false or otherwise). For that we are going to put together a form component for every todo list item and a checkbox for every todo list item. By the way, the way the code looks right now with all these question marks and later the implementation of a conditional form (for editing a todo, we can’t do that if a todo is not successful) makes our code look amateur and just bad, remember I said that typescript is smart enough to know that if a certain query’s isPending= false and isError= false then it realizes that the data is settled and valid? Let’s use it inside our map!

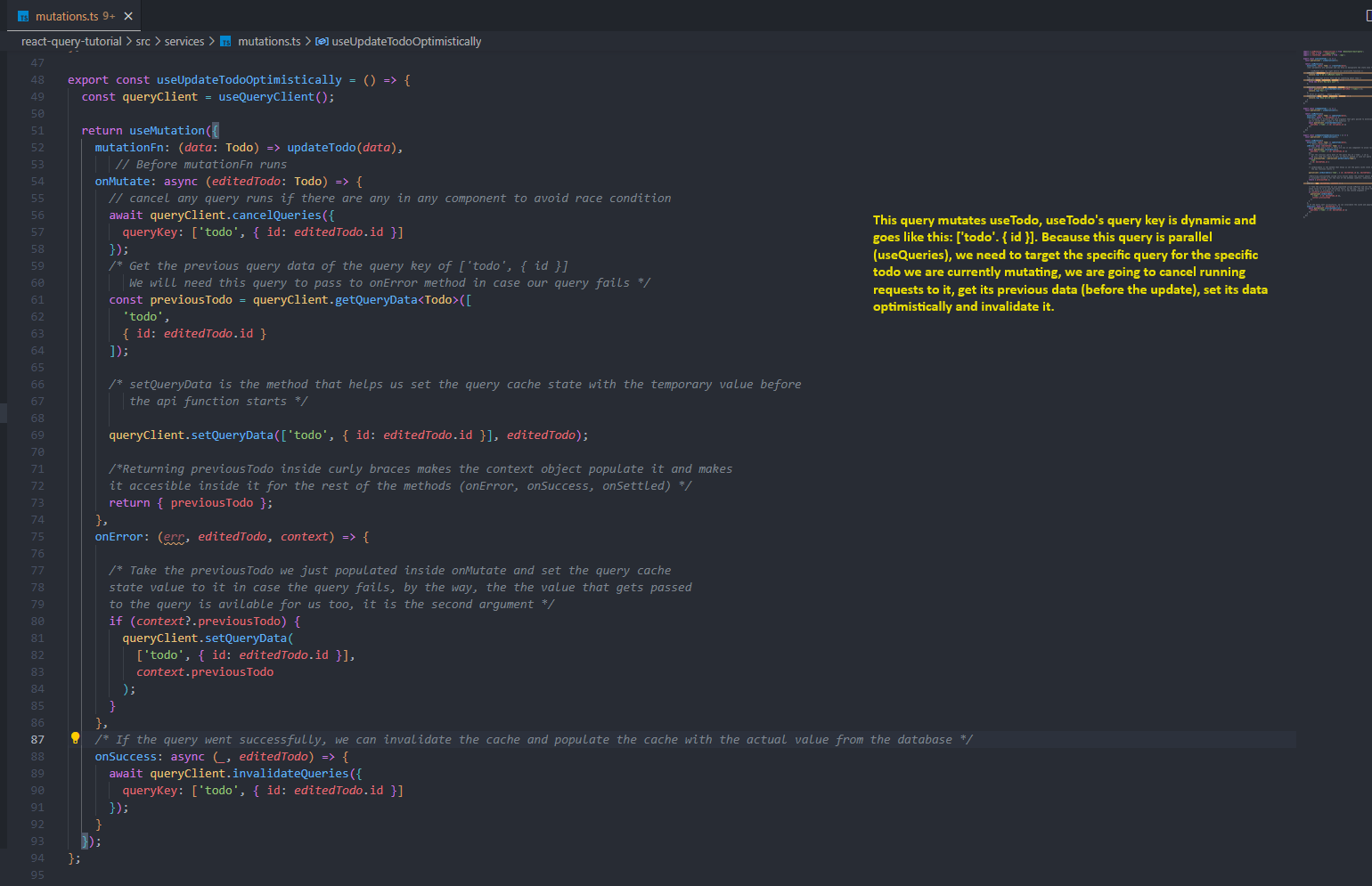
Now let’s get back to our app and thest both functionalities, starting with editing the todos’ title and description. Well it works, but it takes really long for the text to update after we hit the “edit todo” button (Don’t worry, in reality it’ll be shorter). Anything we can do to fix this? Yes, it is called Optimistic updates!

Optimistic updates:

Regular updates (not optimistic) occur after the query function settles and has the status of success, a process that usually takes a while because it is asynchronous. This thing ofter causes a delay from the moment an action was taken (a button has been clicked for example) until the moment the users see the new value on the screen. In most cases we will take care of this delay by showing a loading spinner or a skeleton, but sometimes we can use a different concept that thanks to react-query’s caching system is possible, it goes like that:

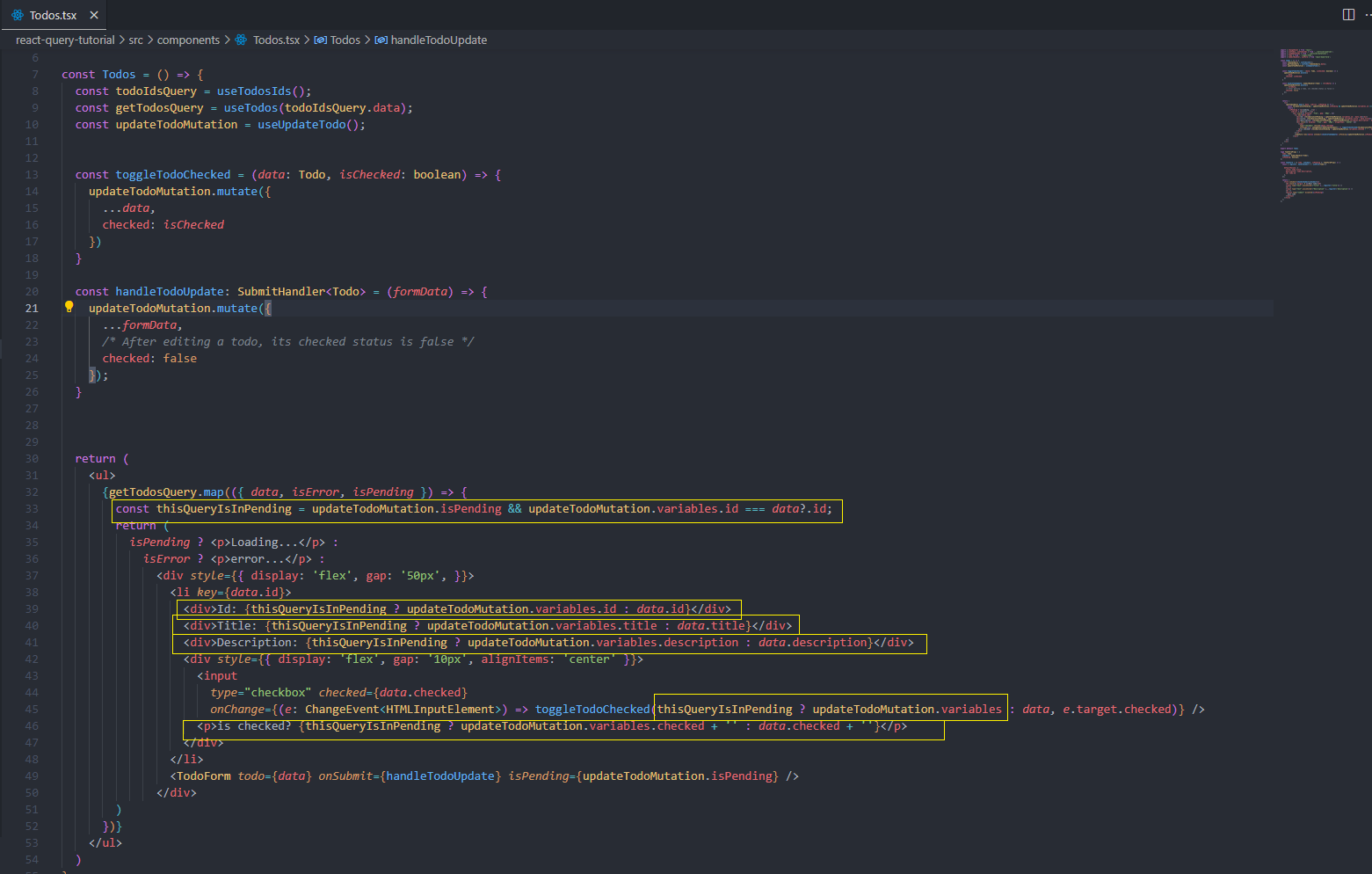
1. Before the asynchronous action even start (inside onMutate), take the input that you received for the query function and put it inside the query cache. Technically we can store the query cache with every temporary value that we want but there is no reason for us to use any different value than the value the function was received.
   1. If the query went successfully, revalidate the cache and then, because the temporary value that is stored inside the cache is identical to the newly received value from our database, the user won’t see any difference or delay.
   2. If the query returned an error, replace the temporary value with the previous value. Yeah, that looks awkward but having an error is awkward anyway…

So let’s write a new version of updateTodo mutation but this time make it update optimistically:

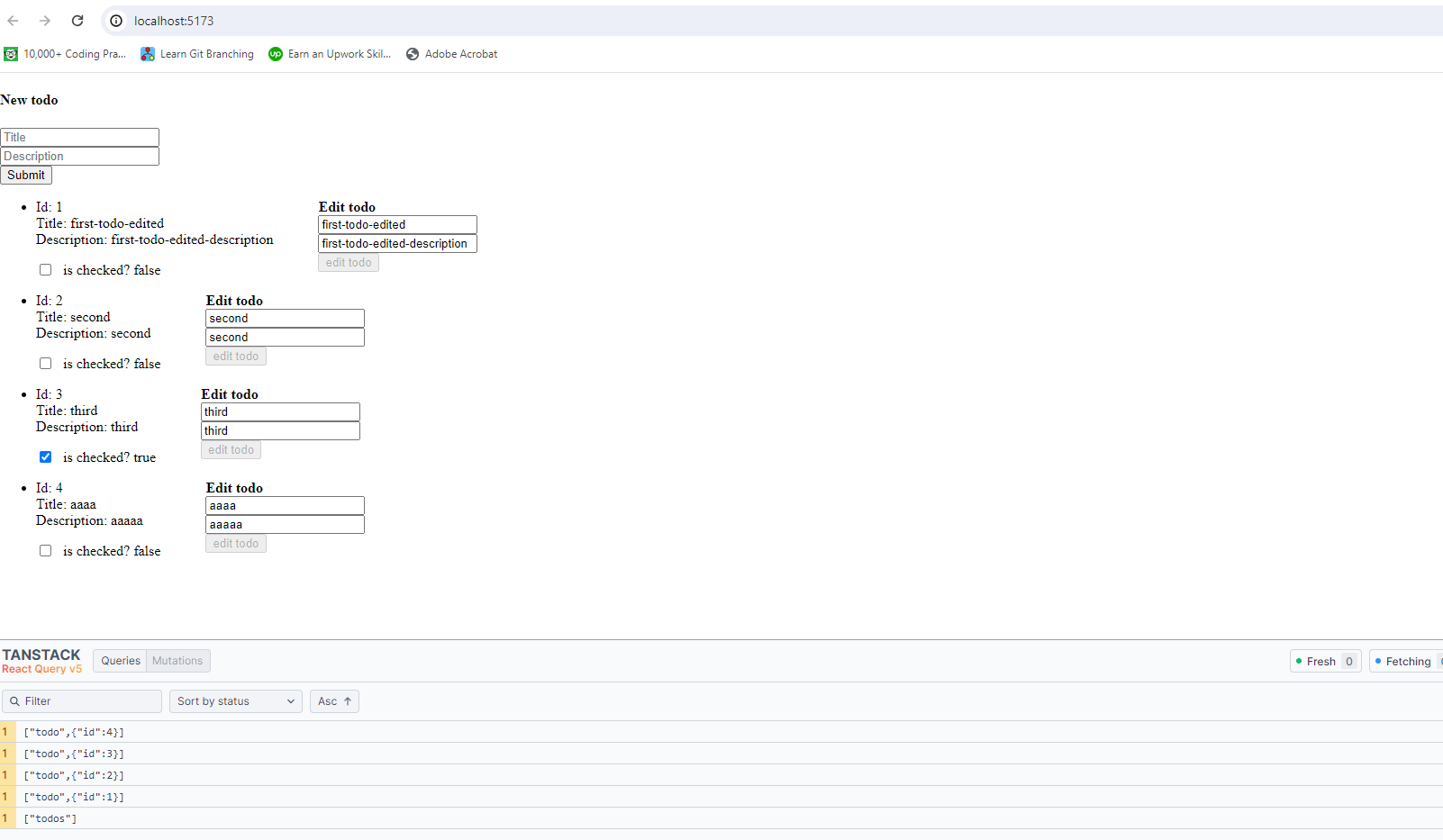


So, what we did here can be divided into three parts, one for onMutate phase, which is before the function runs: In here we need to cancel any running requests in case there are any in some components (queries can be run simultaneously in different areas of the app). The second part is onError, as explained before, the previous snapshot of the cache state (before the edition of the chosen todo) is now available for us inside the context object thanks to us saving it inside onMutate, in case there is an error (and hence we entered onError block) we take that previous snapshot and reupdate the cache state to what it was before the function was invoked (because the value did not actually get to the database, so keeping it anyway will give the user false information, it’s better to let it back to the previous value than mislead the user). Third part, onSuccess, we invalidate the cache and tell useTodos to refetch the data from the database because there is fresh data to collect.

Okay, let’s put some optimism to our <Todos /> component:



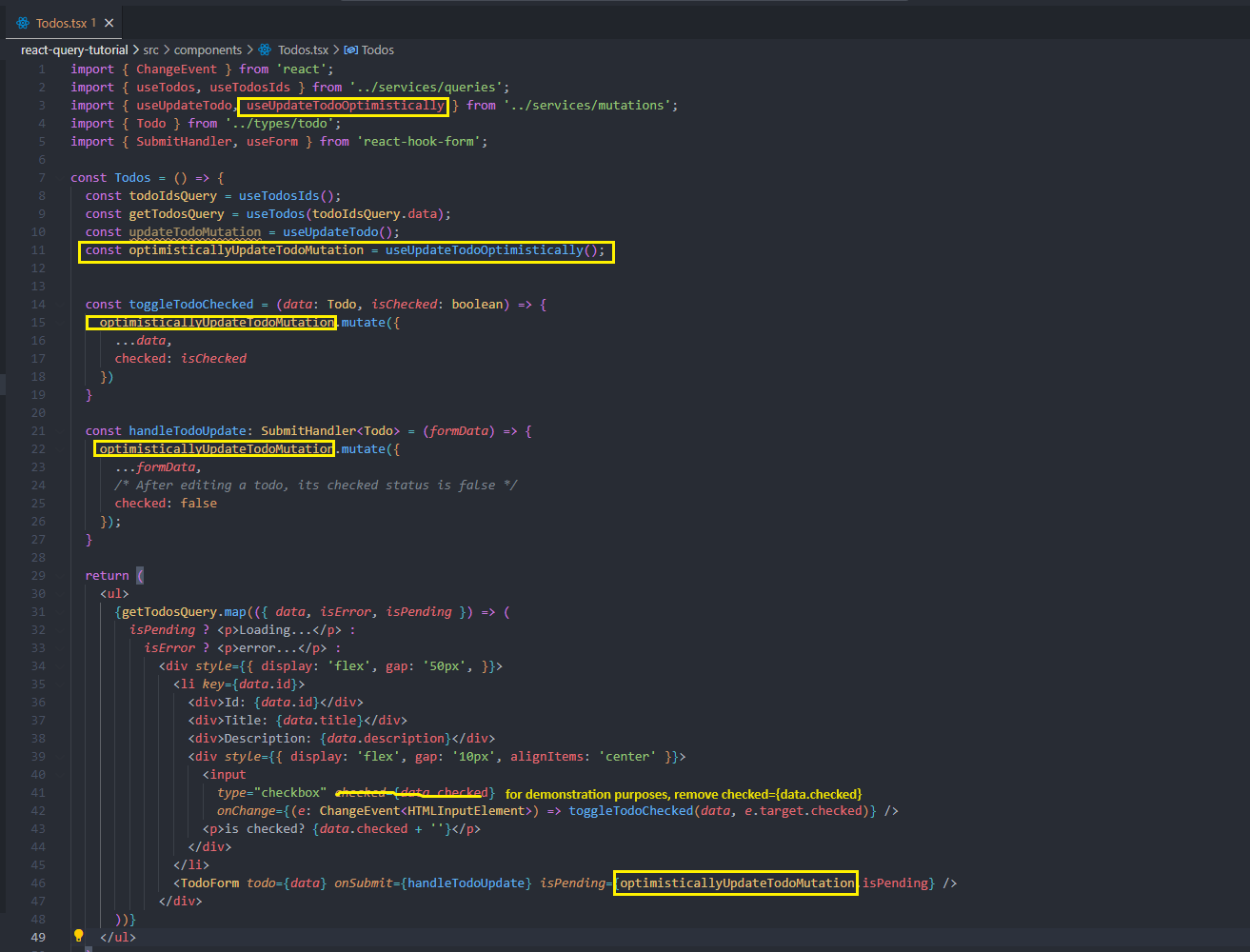
Okay, let’s get back to our app and see if it is updating immediately now.



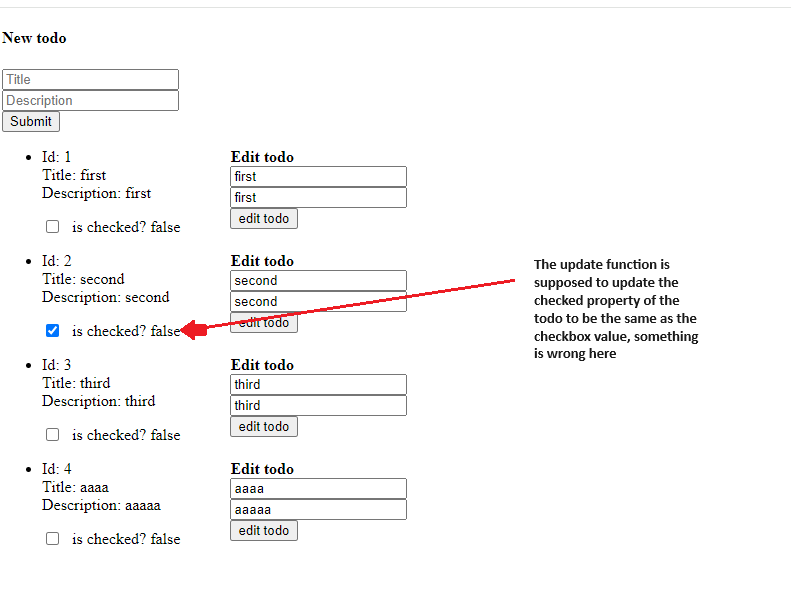
And it is updated immediately! (you can tell by the fact that the button is disabled because the status of the request is “pending” but the new content already appears inside the list item.

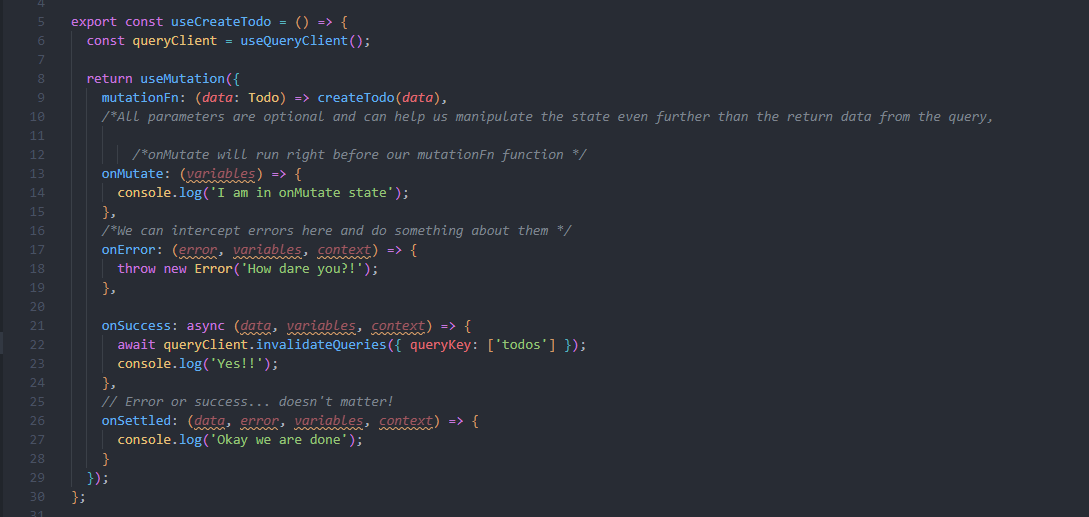
**But wait a minute! Why the hell did we create updateTodoOptimistically if we are not even using it?!**

The answer is because there are two ways to optimistically update data and I wanted to start with the easier, yet more primitive and dirtier one. Let’s undo this and use the more elegant way:



Great, updating the title and the description works smoothly, and it appears that checking and unchecking the checkbox works just fine, it only seems so. It might take us many attempts until we see it (and this is why I wanted for this demonstration to delete the checked prop from the checkbox input), but if we click the checkbox many time and as fast as we can, we will eventually run into a race condition problem, and then after a short while of looking fine, they value will change again to something weird:

****

“Okay, so let’s disable the checkbox while the mutation is pending”. Sure, we can do that, but watching the text value change right away and then see the checkbox freeze for 500ms is something contradictive, they kind of cancel each other, so our optimistic update loses its seamless effect. My point is, optimistic updates are not suitable for all cases. Another scenario where we shouldn’t use optimistic updates is the useCreateTodo mutation:  


Because useCreateTodo mutation mutates useTodoIds query, and useTodoIds data get passed to useTodos, the todos’ list will reload anyway unless we optimistically update useTodos as well, the problem with updating useTodos is that it requires us to setQueryData for query key that not yet exists (because the query hasn’t been invoked yet). it would have worked if we were to choose the second option of fetching the whole todos from the database, but it was important to demonstrate useQueries.

onMutate: async (*editedTodo*: Todo) => {

  queryClient.setQueryData(

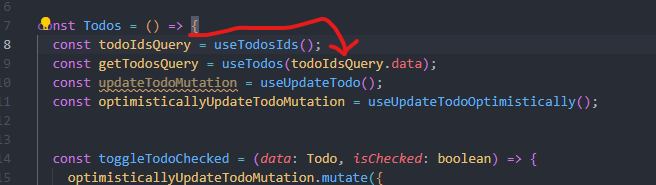
          ['todo', { id: *newTodo.*id }],

*context*.newTodo

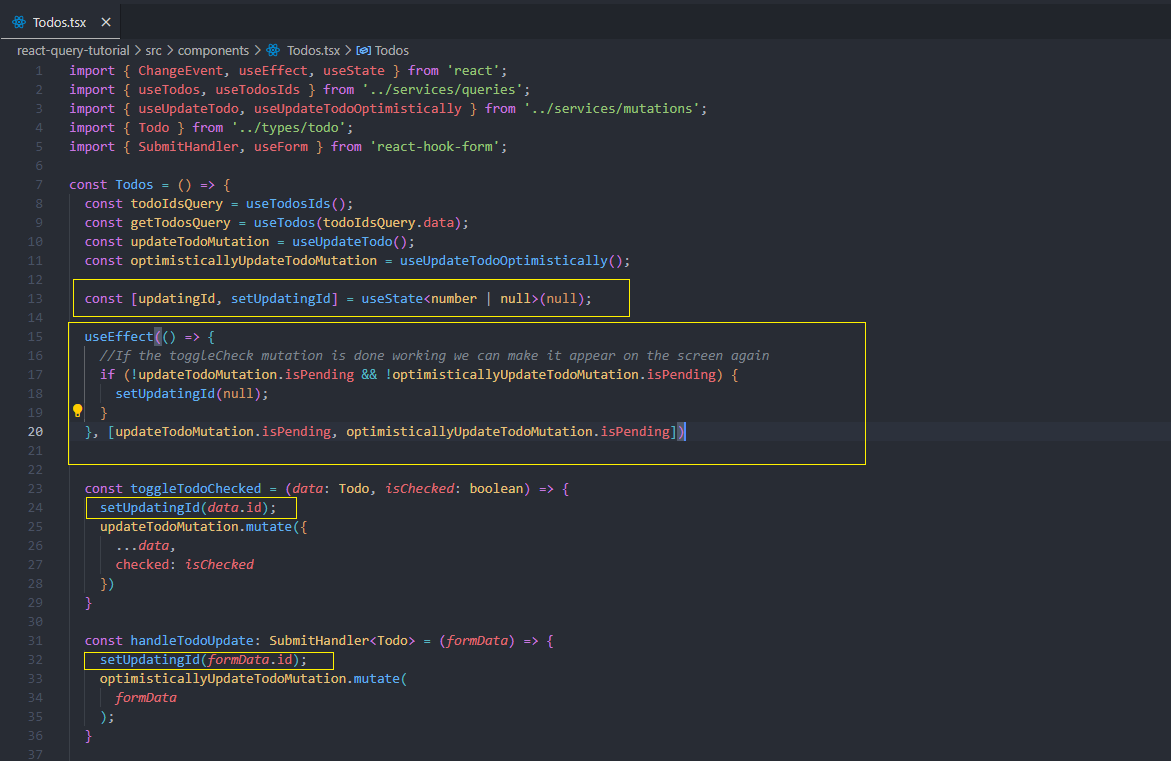
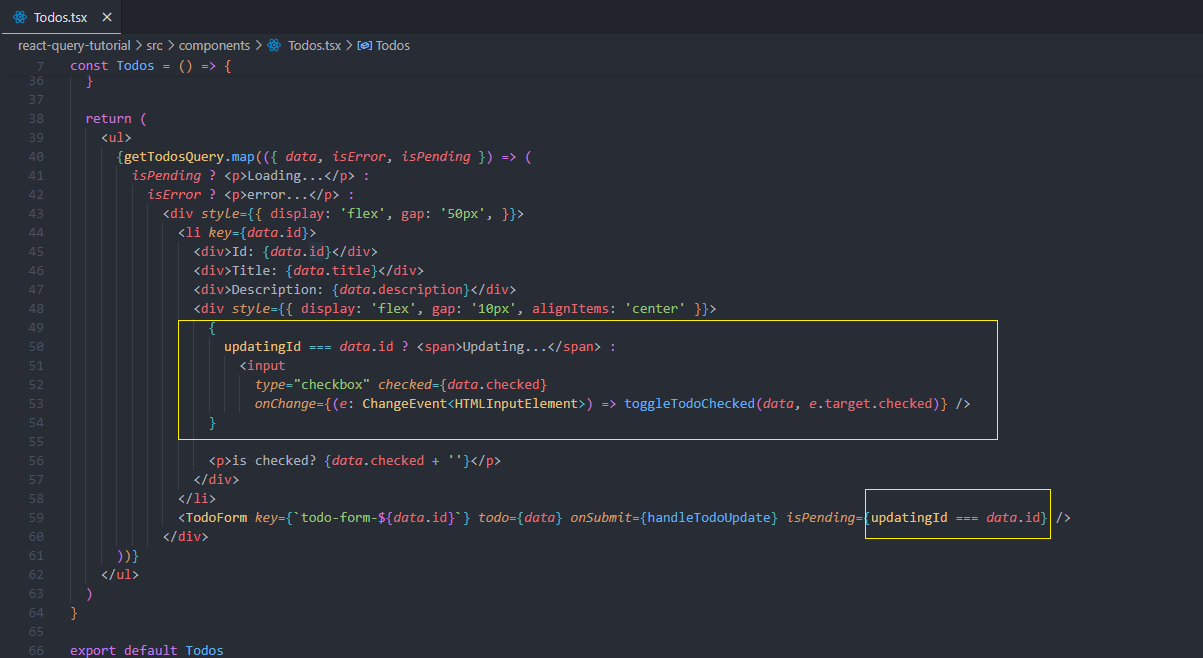
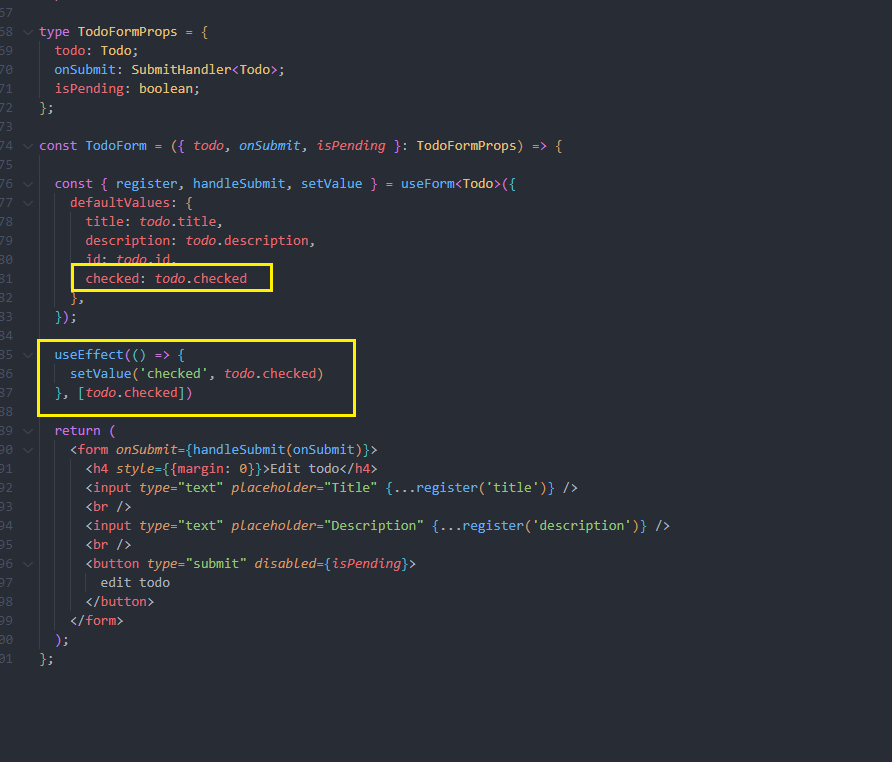
        );

// they key of [‘todo, {id: newTodo.id}] will find nothing, because the query doesn’t exist at the time of the optimistic update (onMutate phase), it will exist only if the query is successfully done resolving the api function (onSuccess phase)

}



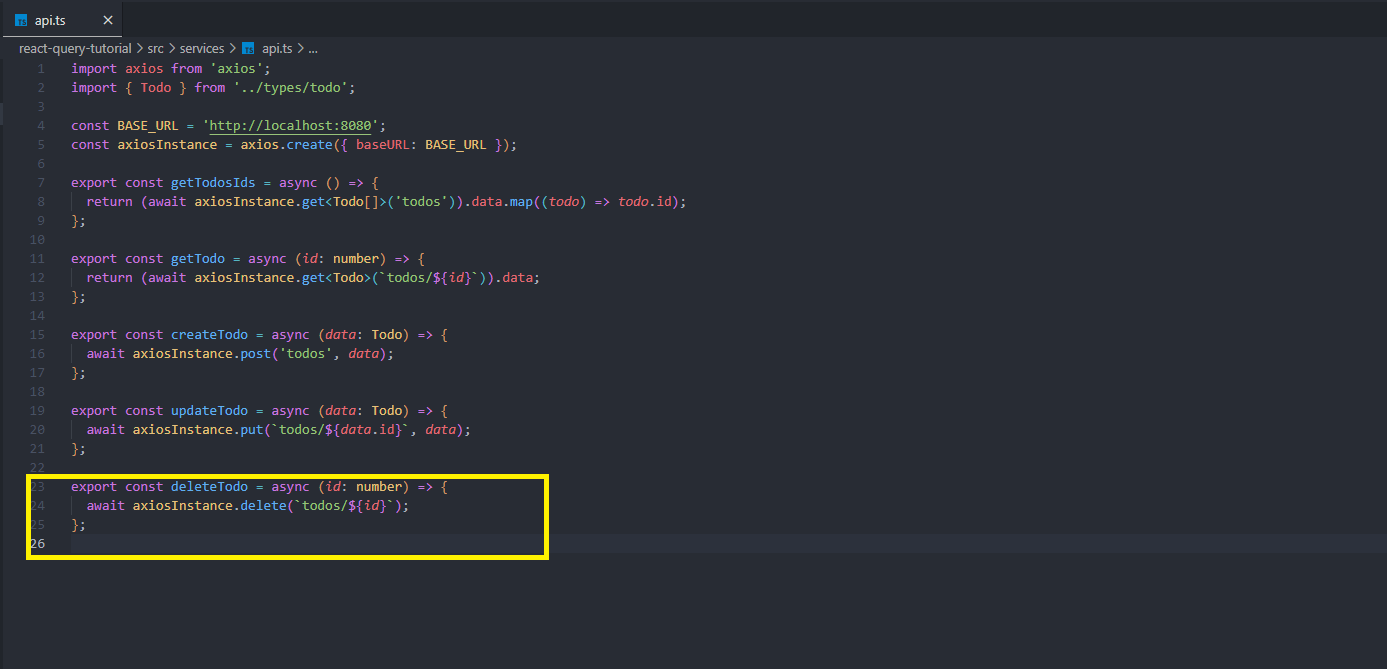
With that being said, let’s get back to developing our app. We are going to keep the optimistic updates for the function that changes the title and the description and get back to the traditional way (updateTodoMutation) for the checkbox, We are keeping the optimistic attitude for the text changing function because invoking this function takes way longer than clicking a checkbox, so by the time a troll updates the text again right after sending it, the original data will be fetched already and race condition will be avoided, and even if the troll is fast enough to re-enter a new text and submit in less than 500ms, don’t worry, we are disabling the edit button while the mutation is pending. Other things we should change in our <Todos /> component: Currently, when an update mutation is in pending, all the edit buttons get disabled, this is not the behavior we want, we just need to disable the button responsible for the specific query that’s being edited. We also want to show an “updating…” text instead of the checkbox when a specific todo’s checked status gets changed (i.e., after a checkbox is clicked on). One more little thing, after a user updates the title or the description, the checked status of the updating todo shouldn’t automatically become false, so let’s take care of that as well:

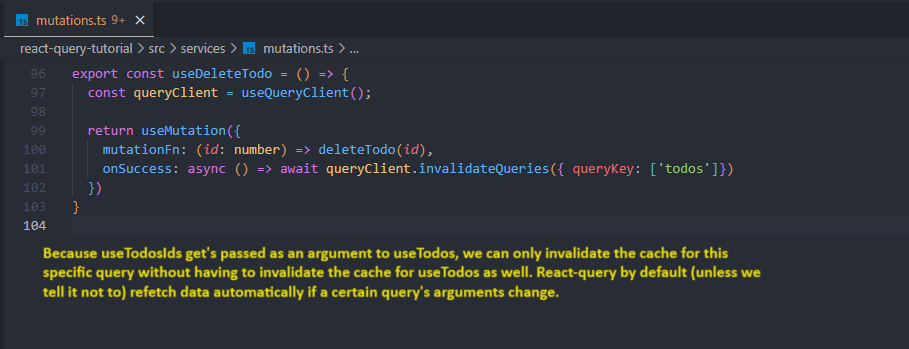
**Challenge- Delete a todo, regular and optimistically**

At this point I am going to leave this as a challenge for you. If you want to watch the solution you can move on to the next pages, but if you don’t want to, then stop right here and get your hands dirty.

You just need to know that the endpoint for deleting a todo needs to get a DELETE request to the endpoint of 'http://localhost:8080/todos/:id'

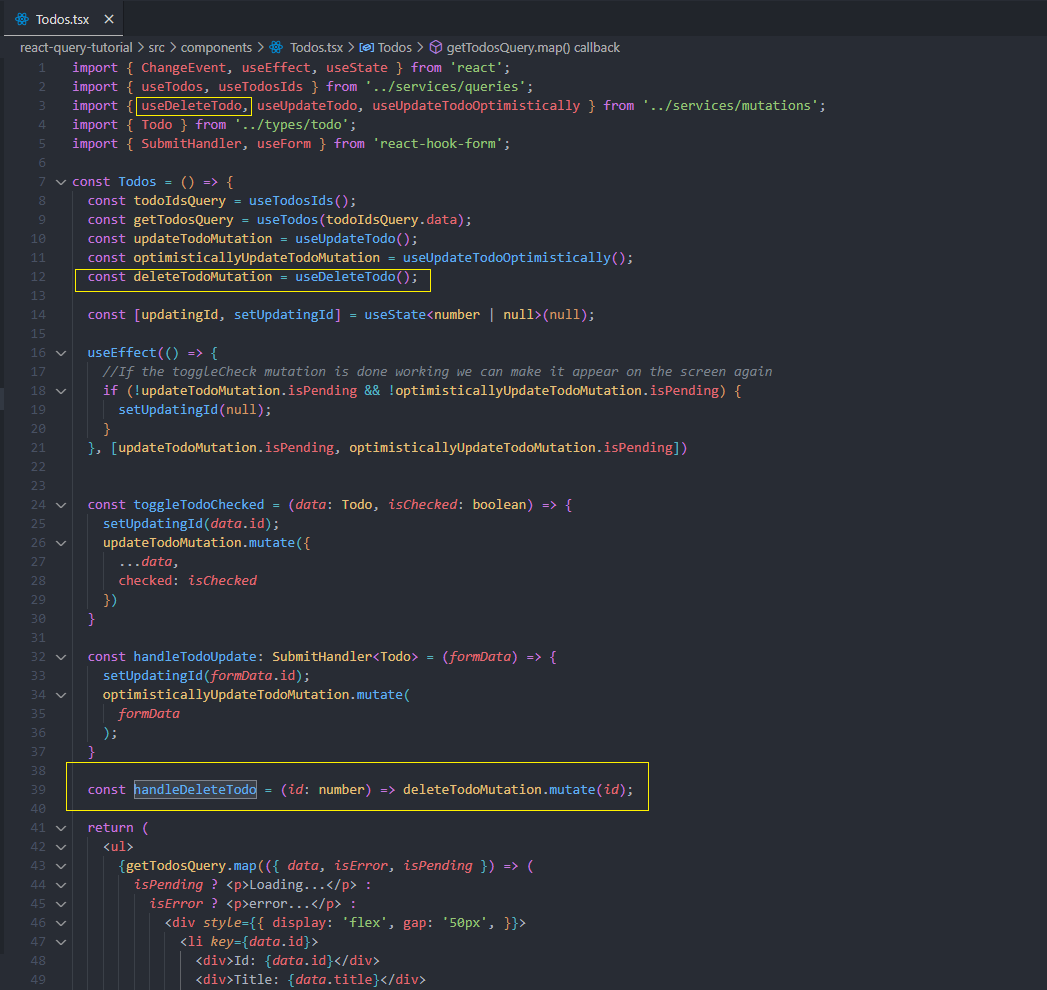
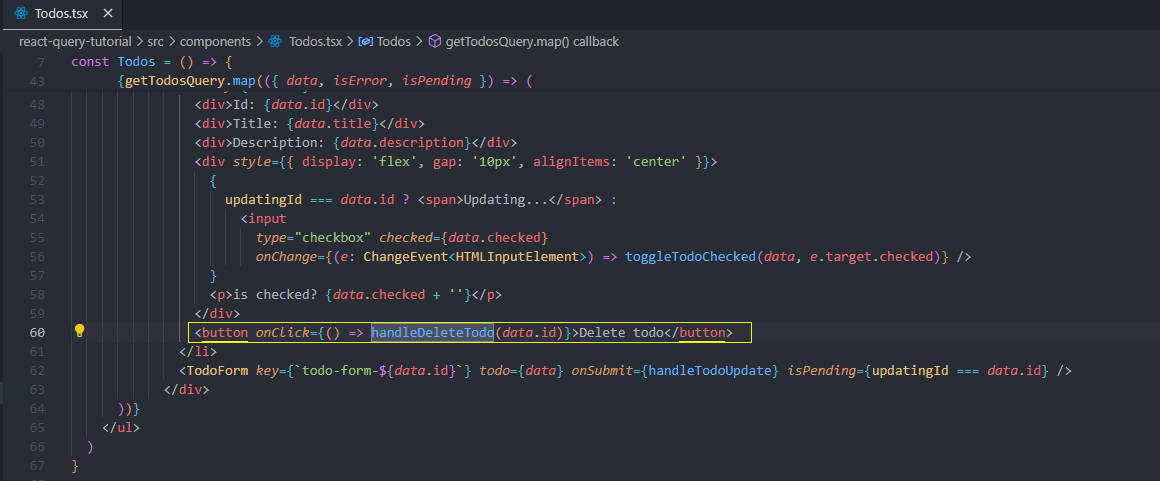
First step- going to our api.ts and define an axios api function:  


Second- let’s go to mutations.ts and create a new mutation to invoke this delete api function:



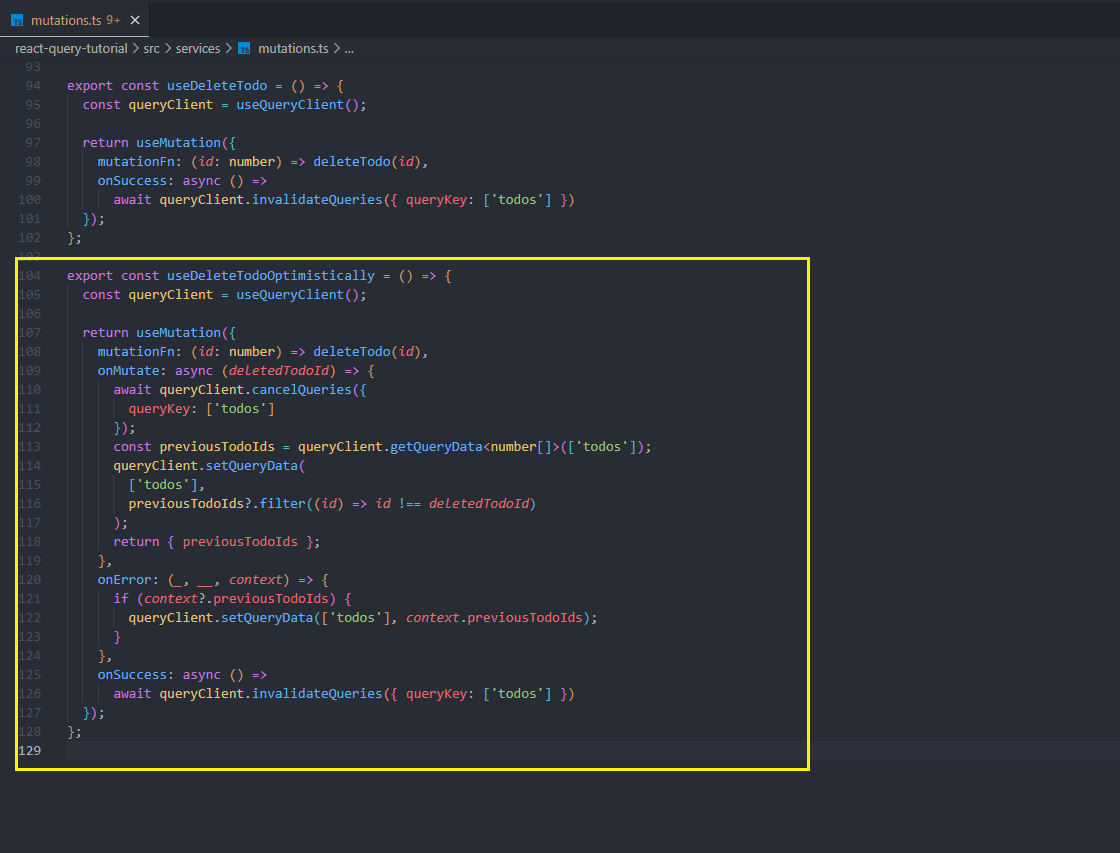
In case you are interested on how to make a query not fetch in case its arguments change, here is the way:  


Third step- use useDelete inside our <Todos /> component:

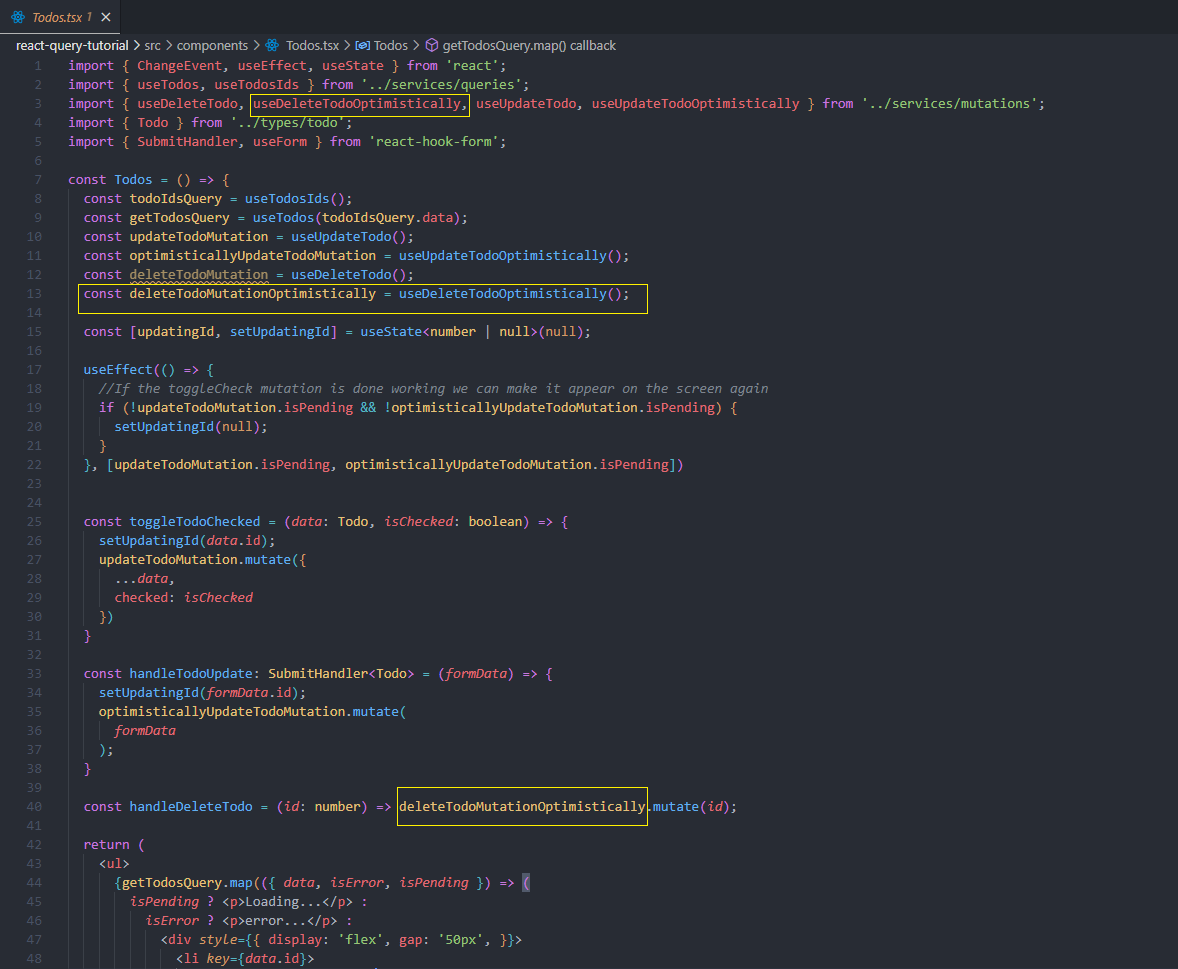
  


Test you app now and delete one of the todos, it is supposed to hold for 500ms and then delete the todo of the page.

Let’s create the optimistic version of deleteTodo

First step- get back to mutations.ts and create the mutation function, the ritual is to stop any current incoming request, save the previous cache state snapshot (the state value prior to the deletion process) into a variable, update the cache state manually and return the previous cache state snapshot in curly braces so it gets passed to the context object.  
Second step- create the onError phase, it gets three arguments, the error object, the variable that was passed to mutationFn and the context object, the only argument we need is the context object. So inside onError let’s set the query cache state data back to our previous cache state, since the update was failed.  
Last step- Let’s create onSuccess phase where we instruct react-query to invalidate our useTodoIds query.  
Reminder query invalidation is an asynchronous function  
Here is the code:  


Last step- update our <Todos /> component:



That’s it, now any todo that we delete gets removed from the page right after we click the Delete todo button!

Thank you very much for partaking in this tutorial!