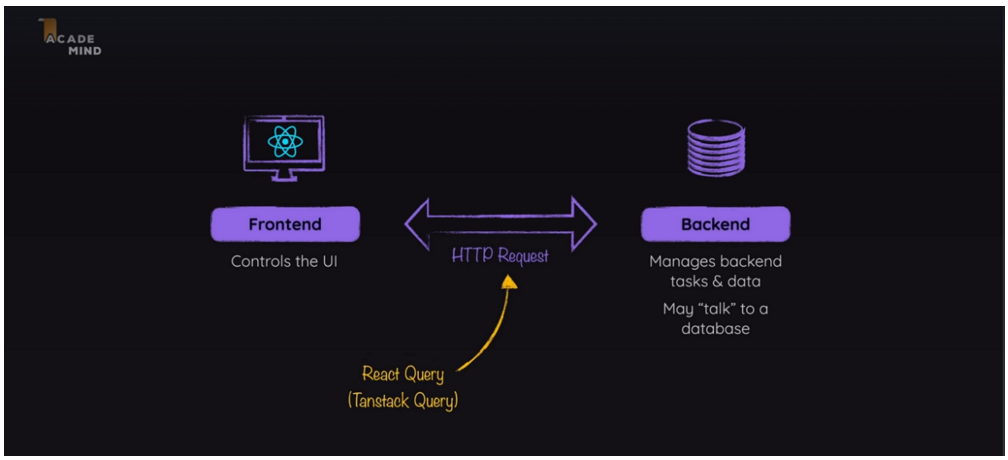
Section 24: React Query / Tanstack Query: Handling HTTP Requests With Ease

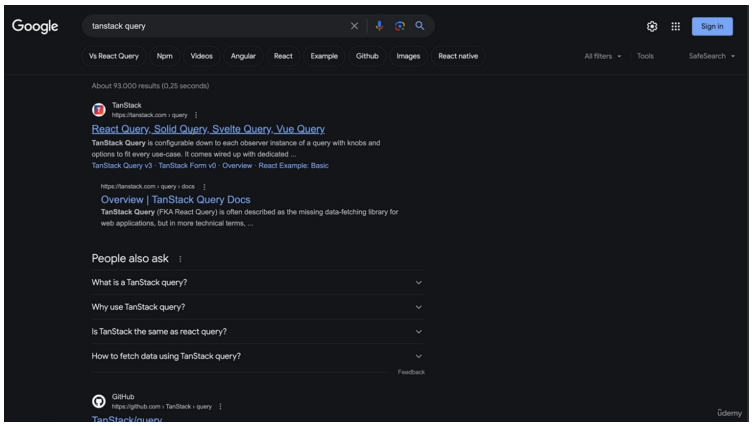
Monday, October 14, 2024 12:21 PM

It's a library that helps you with sending HTTP requests from inside your React app. It helps you with connecting your React frontend to a backend.

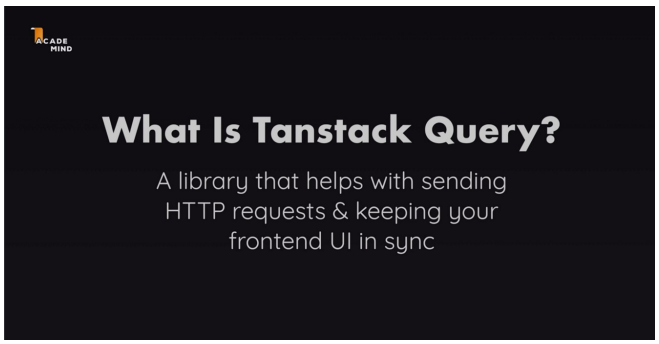


In this section, we will learn why we would use it instead of **useEffect** and **fetch**

Project Setup and Overview

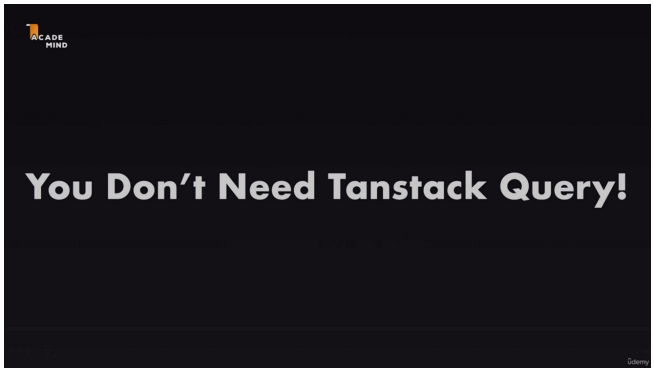


React Query: What and Why

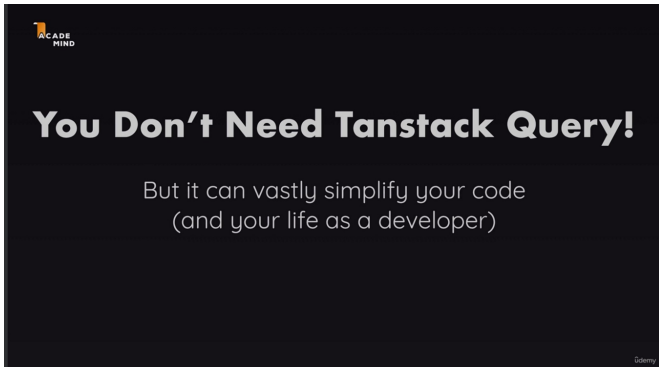


Tanstack query is a library that helps you with sending HTTP requests and keeping your frontend UI in sync with your backend data.

But you don't need Tanstack Query to do that.



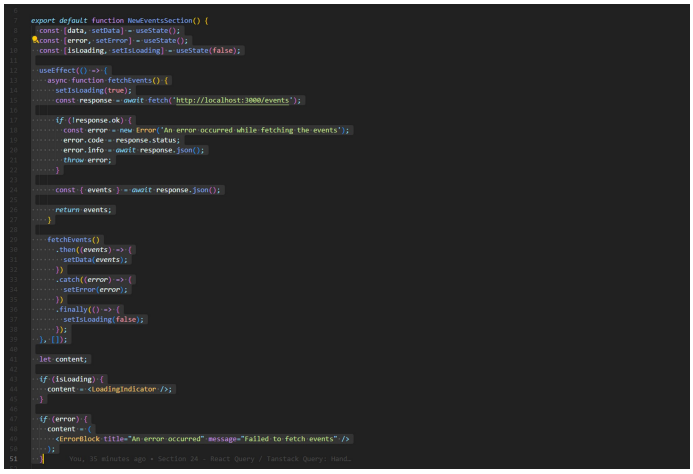
You could do that with **useEffect** and **fetch** function. But Tanstack Query can vastly simplify your code. Because it comes with many built in more advanced features which might be needed by more complex React apps and which are relatively difficult and challenging to build on your own.

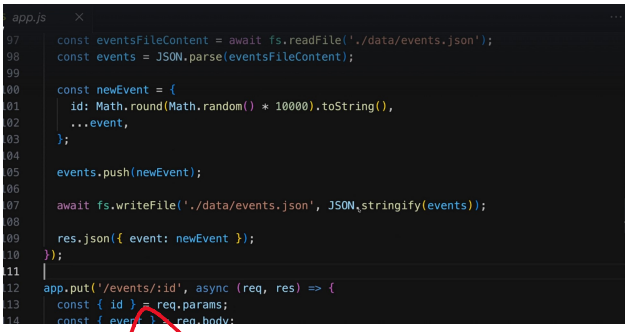


For example look at this code,

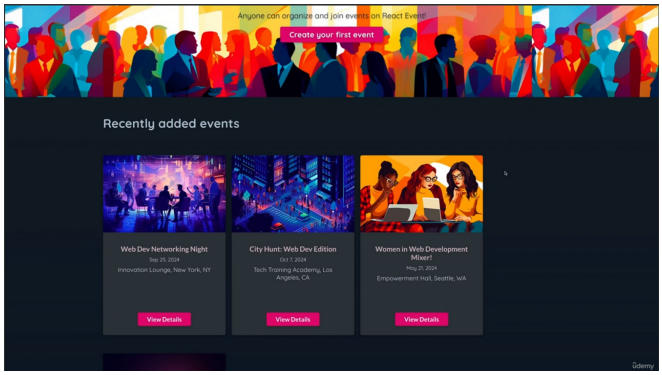
We have to write lot of code to fetch the data manage the state. We have to do it in every component that wants to send HTTP requests.

We could simplify the code and reuse by building a **custom hook** though. But even if we would build such a custom hook, this code would still have some problems or missing features to be precise.



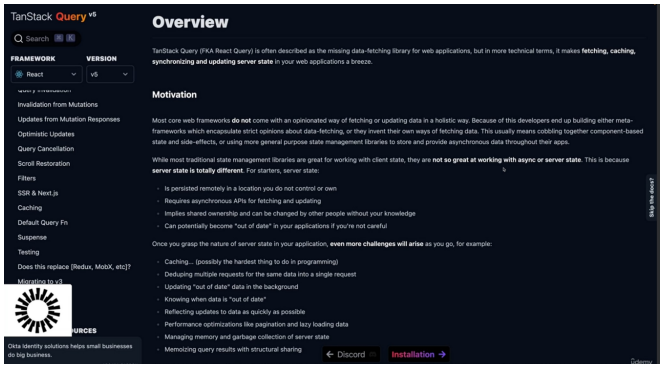


For example, I am on this website.

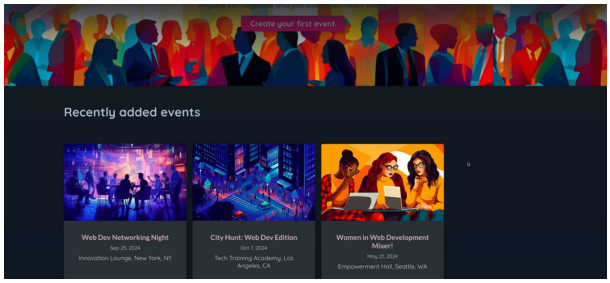


Backend application

I then switch to a different tab to do something there



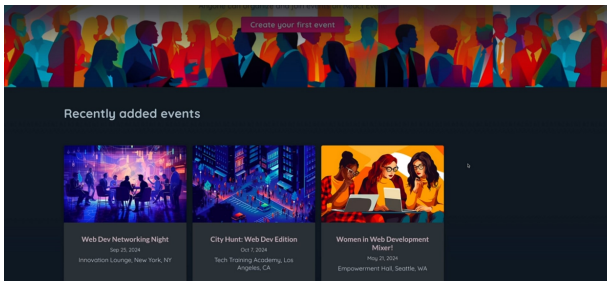
I then comeback to this website. I might want to trigger a refetch. So that behind the scenes, we're fetching new data and we're updating the UI if we found out that the data we're currently displaying is outdated.



Currently in our code, we don't have that behavior and it would take us quite a bit of code to write to implement this behavior in a good way.

```jsx
export default function NewEventsSection() {
  const [data, setData] = useState();
  const [error, setError] = useState();
  const [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    async function fetchEvents() {
      setIsLoading(true);
      const response = await fetch('http://localhost:3000/events');

      if (!response.ok) {
        const error = new Error('An error occurred while fetching the events');
        error.code = response.status;
        error.info = await response.json();
        throw error;
      }

      const { events } = await response.json();

      return events;
    }

    fetchEvents()
      .then((events) => {
        setData(events);
      })
      .catch((error) => {
        setError(error);
      })
      .finally(() => {
        setIsLoading(false);
      });
  }, []);

  let content;

  if (isLoading) {
    content = <LoadingIndicator />;
  }

  if (error) {
    content = (
      <ErrorBlock title="An error occurred" message="Failed to fetch events" />
    );
  }
```

Another feature we might want in this app here could be caching. So that data once fetched is cached. Stored in memory and we can reuse that data if we need it again.

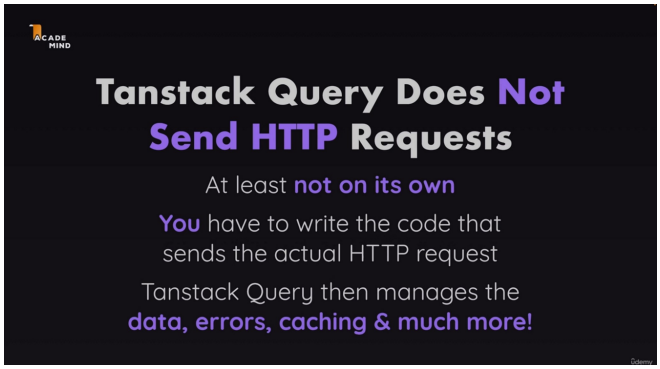


That's where Tanstack Query comes into play. With this library we will not just be able get rid of bunch of code,
But we'll also get some advanced features like caching behind the scenes, data fetching.

Installing & Using Tanstack Query - And Seeing Why it's Great!

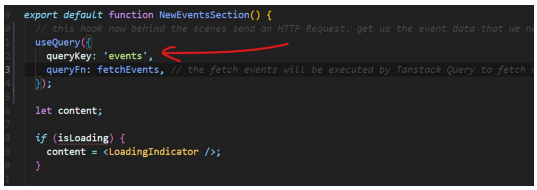
i. npm install @tanstack/react-query

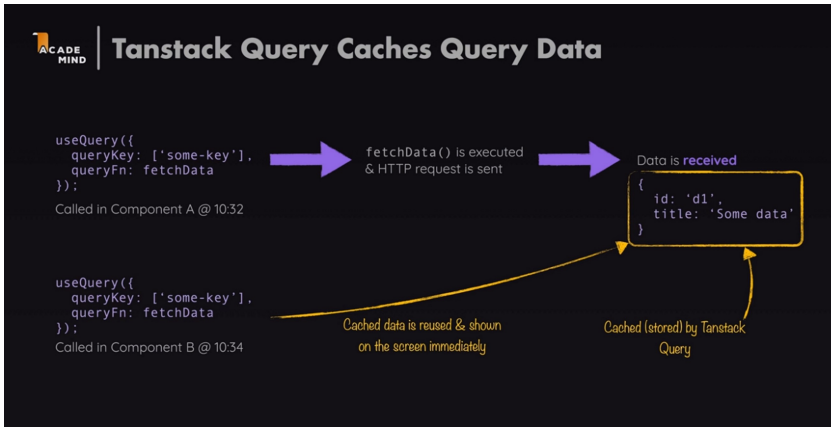
Tanstack Query does not come with built-in logic to send HTTP requests. Instead it comes with logic for managing those request, for keeping track of the data and the possible errors



The code for sending the requests must come from your side.

Query key will be internally be used by React Query/Tanstack Query to cache the data .





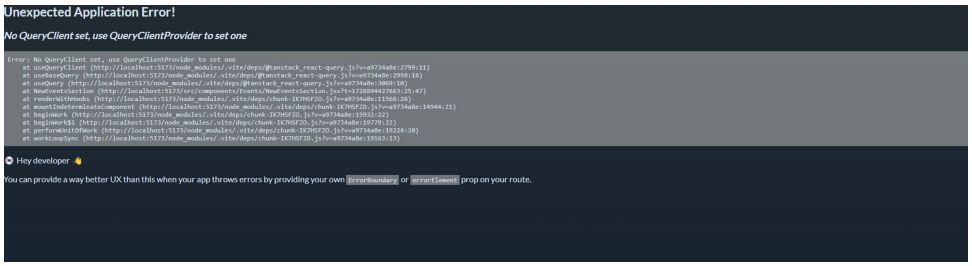
So, that the response from that request could be reused in the future if you are trying to send the same request again.

```jsx
export default function NewEventsSection() {
  // this hook now behind the scenes sends a request. get us the event data that we need in this section and also give us loading state and potential
  const {data, isPending, isError, error} = useQuery({
    queryKey: ['events'], // key name is totally up to you. you could have object or anything else
    queryFn: fetchEvents, // the fetch events will be executed by Tanstack Query to fetch my data.
  });

  let content;

  if (isPending) {
    content = <LoadingIndicator />;
  }

  if (isError) {
    content = (
      <ErrorBlock
        title="An error occurred"
        message={error.info?.message || 'Failed to fetch events.'}
      />
    );
  }
```

```jsx
export async function fetchEvents() {
  const response = await fetch('http://localhost:3000/events');

  if (!response.ok) {
    const error = new Error('An error occurred while fetching the events');
    error.code = response.status;
    error.info = await response.json();
    throw error;
  }

  const { events } = await response.json();

  return events;
}
```

You code must make sure to return an error response.

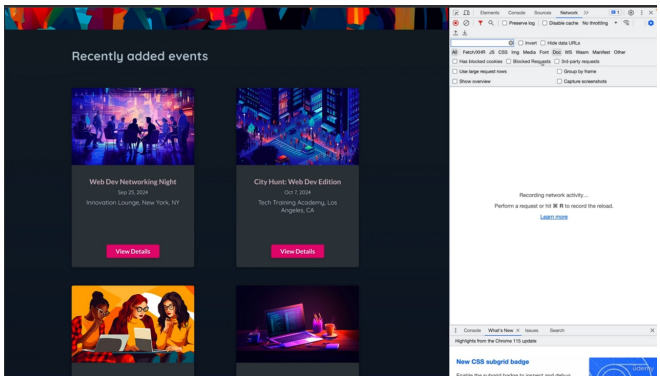
But after writing this code, we get this error



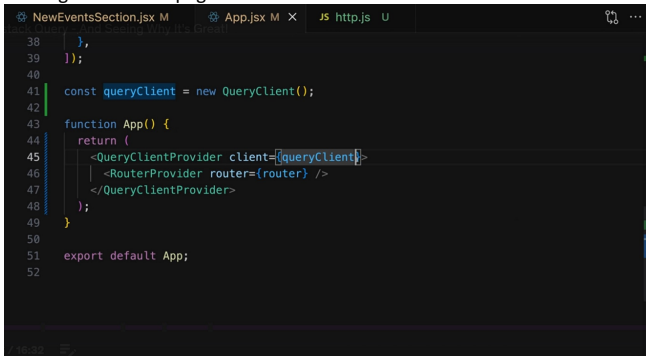
Because in order to use React Query and the useQuery hook, you must wrap the components that do want to use these features with a special provider component. (Provided by Tanstack Query). I will do that in the <App/> component here.

```jsx
// This is a general configuration object, that will be required by T
const queryClient = new QueryClient();


function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <RouterProvider router={router} />
    </QueryClientProvider>
);
}

export default App;
```

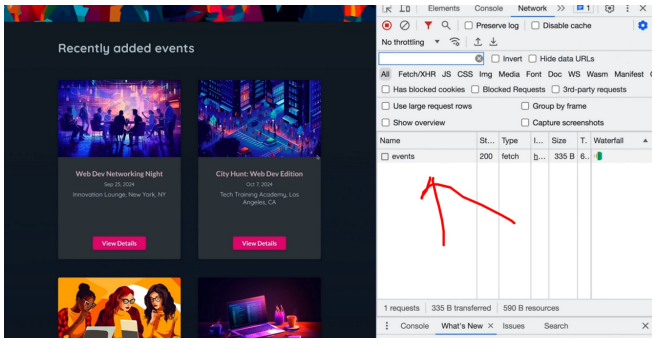
Open the developer tools and open the network tab.



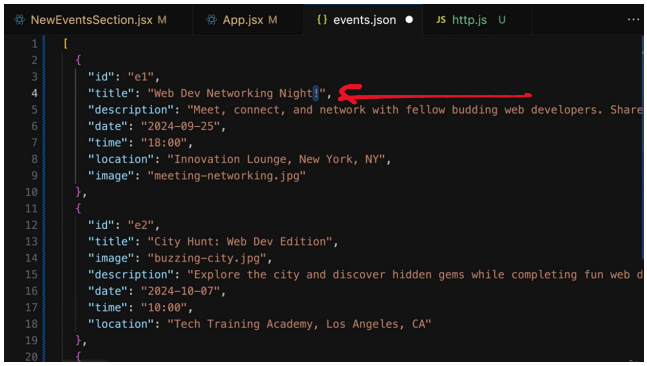
Now I go out of this page



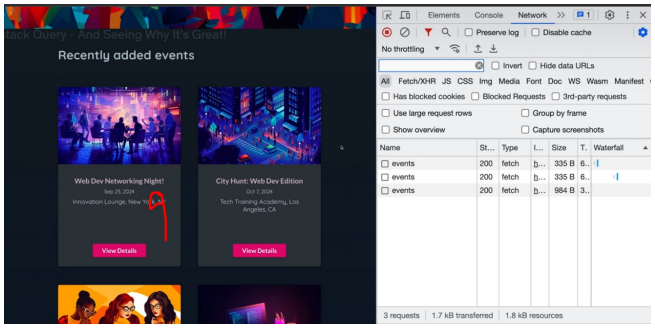
Then come back to it, then you will see this HTTP request being sent here and that's being sent because of Tanstack Query which reacts to us going away from this screen and coming back to it.



The advantage is, if some data change to my backend, let's say add a ! here

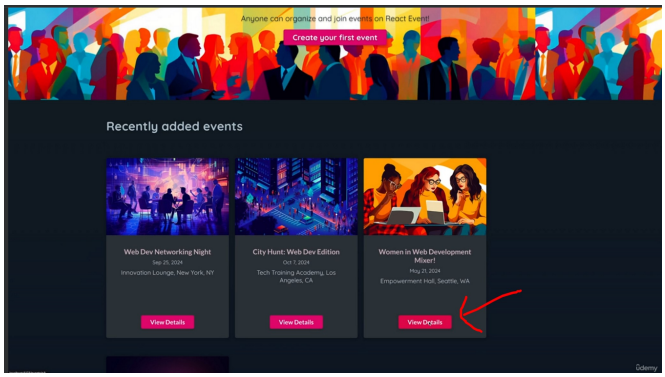


Now if I come back that updated data is fetched.

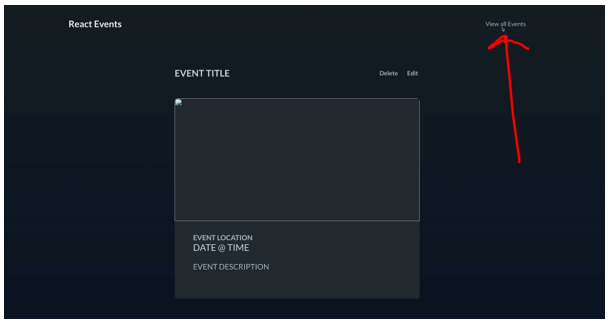


Understanding & Configuring Query Behaviors - Cache & Stale Data

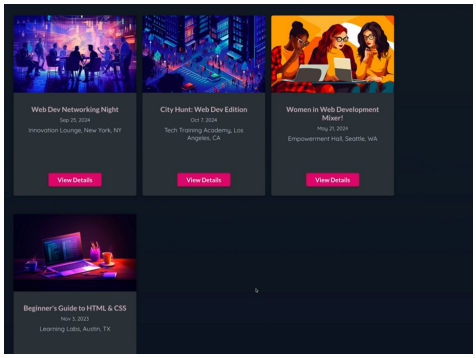
React Query caches response data. Click on this button.



Now go back to 'View all Event's button.

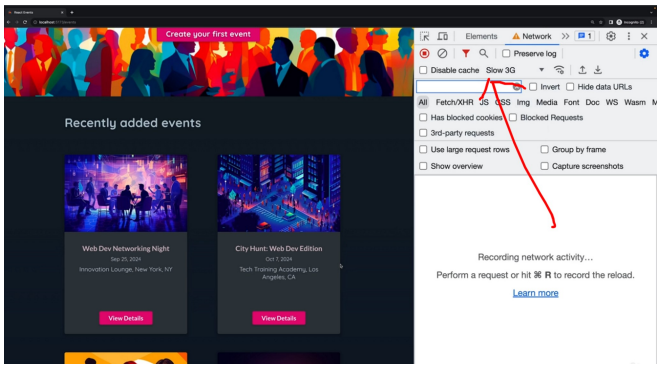


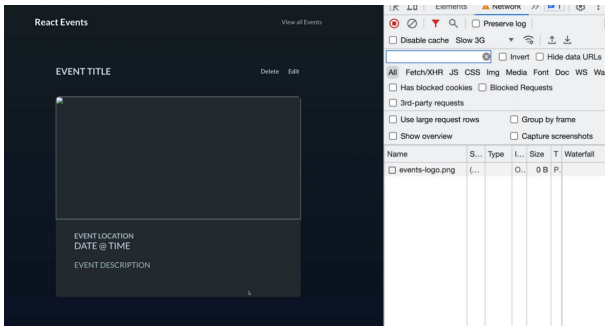
Events here are available instantly.



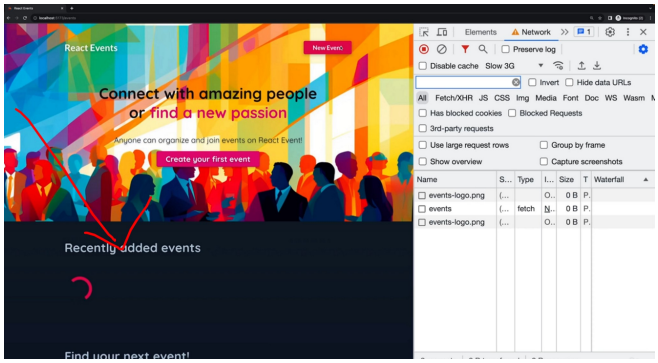
Before we we use 'useEffect', this was not the case.

There if we went to different page,

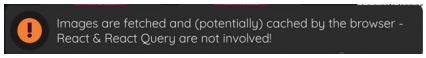




and came back. and a brand new request was sent and all the data was fetched again.



But here with react query, the data is available pretty instantly.



By default, React Query caches the data that's why react query shows all the data instantly but behind the scene it also Sends a new request to check for the updated data silently.

This controls after which time React Query will send such a behind the scene request to get updated data
If it found data in your cache.

```
export default function NewEventsSection() {
  const { data, isPending, isError, error } = useQuery({
    queryKey: ['events'],
    queryFn: fetchEvents,
    staleTime:
  });

  let content;
```

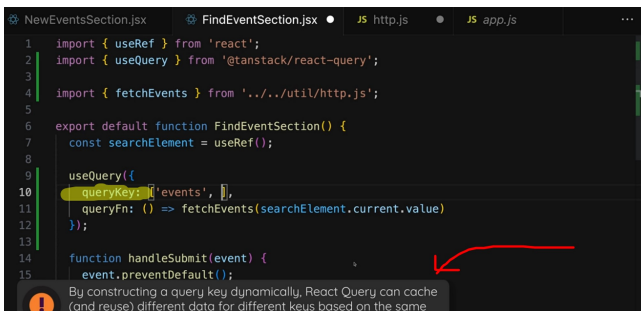
The default is 0, which means it will use data from the cache,
But it will then always also send such a behind the scene request to get updated data.

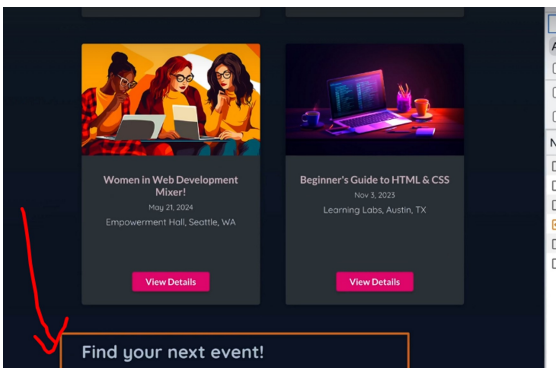
```
export default function NewEventsSection() {
  const { data, isPending, isError, error } = useQuery({
    queryKey: ['events'],
    queryFn: fetchEvents,
    staleTime: 0
  });
```

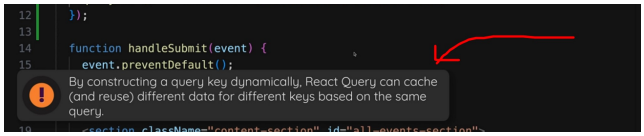
It will wait 5000 milliseconds before sending another request. (So If this component was rendered and therefore this request was sent and within 5 seconds it is rendered again and the same request would need to be sent, React Query would not send it if the staleTime is 5000). With this property, we can control so that no unnecessary request is sent)

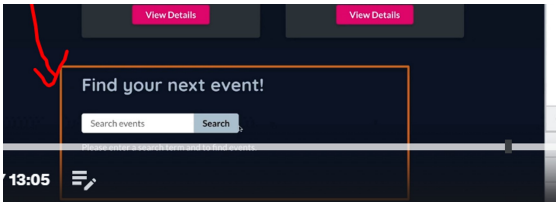
```
import LoadingIndicator from '../UI/LoadingIndicator.jsx';
import ErrorBlock from '../UI/ErrorBlock.jsx';
import EventItem from './EventItem.jsx';
import { fetchEvents } from '../../util/http.js';

export default function NewEventsSection() {
  const { data, isPending, isError, error } = useQuery({
    queryKey: ['events'],
    queryFn: fetchEvents,
    staleTime: 5000
  });

  let content;
```

Dynamic Query Functions & Query Keys



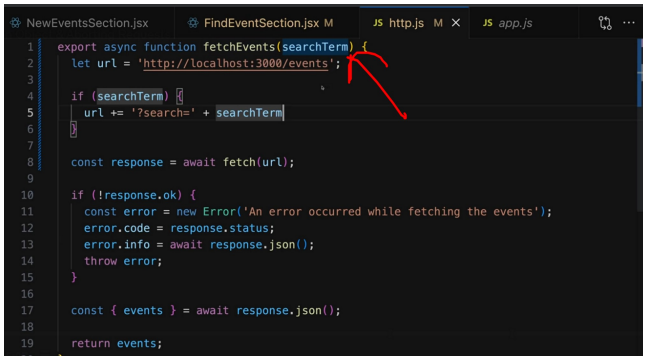




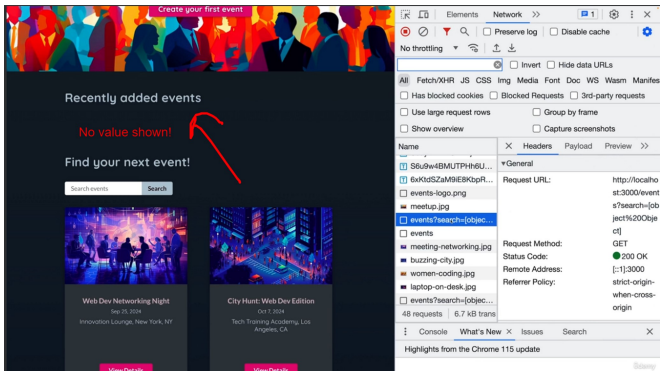


The Query Configuration Object & Aborting Requests

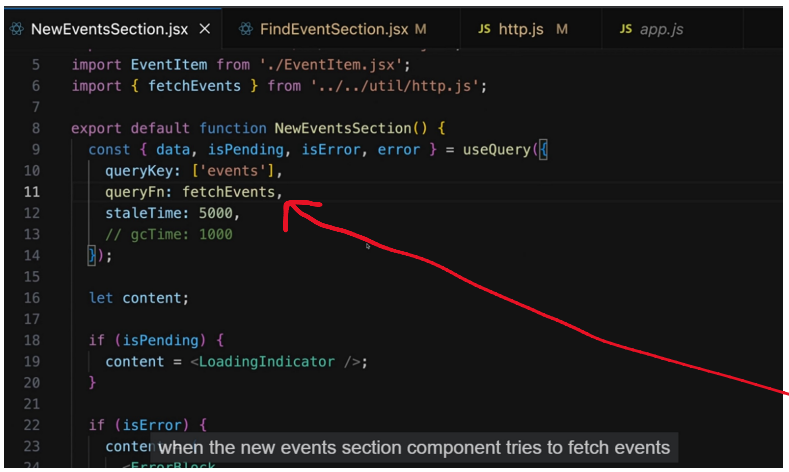
So why this search term set to some strange value



As we can see in this one request here

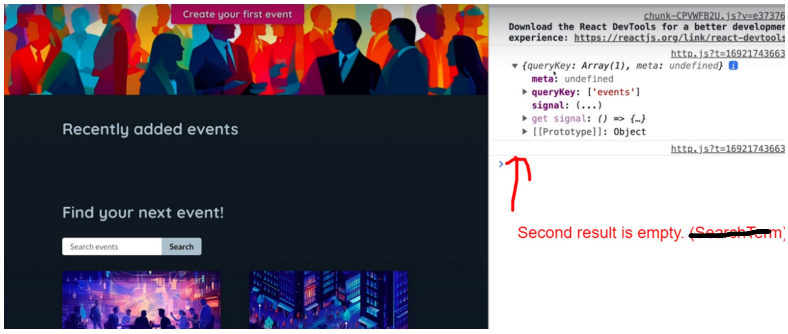


When this new event section component tries to fetch events with this useQuery usage?



Because the useQuery hook actually passes some default data to this query function you are defining here.

```jsx
export async function fetchEvents(searchTerm) {
  console.log(searchTerm);
  let url = 'http://localhost:3000/events';

  if (searchTerm) {
    url += '?search=' + searchTerm
  }

  const response = await fetch(url);

  if (!response.ok) {
    const error = new Error('An error occurred while fetching the events');
    error.code = response.status;
    error.info = await response.json();
    throw error;
  }

  const { events } = await response.json();
```

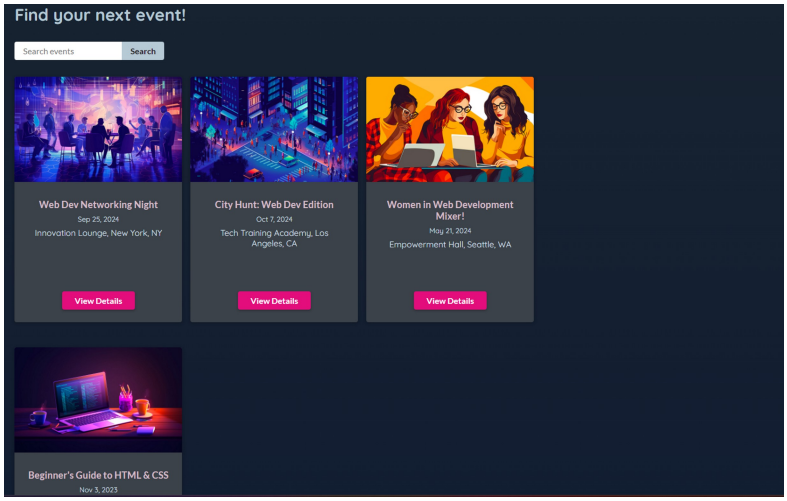


But the first result is an object. An object which I never passed to fetch events.
React Query by default passes some data to your query function. The data is passes in is an object that gives us information about the ***Query Key*** that was used for that query and a signal.

That **signal** is required for aborting that request, if you for example navigate away from this page before the request was finished.

Enabled & Disabled Queries

By default, we get all the list even we don't type anything to the input.



We can enable or disable a query by this, which will solve our problem.

```jsx
export default function FindEventSection() {
  const searchElement = useRef();
  const [searchTerm, setSearchTerm] = useState();
// the difference between isLoading and isPending is that Loading will not be true if this query
 const {data, isLoading, isError, error}  = useQuery({
    queryKey: ['events', {search: searchTerm}],
    queryFn: ({signal}) => fetchEvents({signal,searchTerm}),
    enabled: searchTerm !== undefined
  });
      You, 7 hours ago • Section 24 - React Query / Tanstack Query: Hand…
  function handleSubmit(event) {
    event.preventDefault();
    setSearchTerm(searchElement.current.value);
  }

  let content = <p>Please enter a search term and to find events.</p>;
```

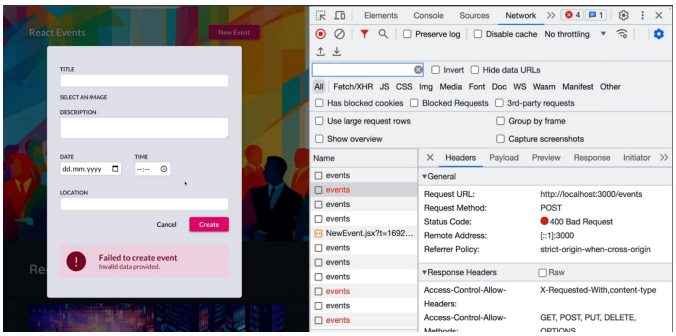
Changing Data with Mutations

React Query can not just be used to get data but also to send data, to create a new event for example.

Use 'useQuery' to get Data
Use 'useMutation' to post data/ send data.

You could also send post requests with **useQuery** **because after all you're writing the logic for sending the requests on your own anyways.**

```jsx
export async function fetchEvents({ signal, searchTerm }) {
  console.log(searchTerm);
  let url = 'http://localhost:3000/events';

  if (searchTerm) {
    url += '?search=' + searchTerm;
  }

  const response = await fetch(url, { signal: signal });

  if (!response.ok) {
    const error = new Error('An error occurred while fetching the events');
    error.code = response.status;
    error.info = await response.json();
    throw error;
  }
```



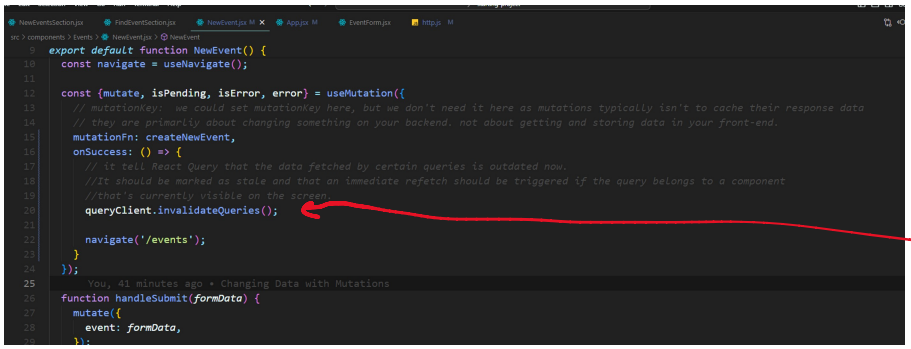
But this useMutation hook is optimized for such data changing queries. For example simply by making sure that those requests are not sent instantly when this component renders. (which was the default case with 'useQuery').

But that request are only sent when you want to send them.

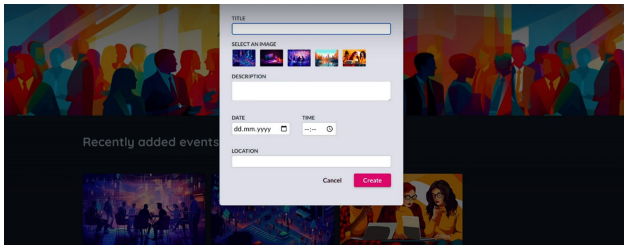
```jsx
vEventsSection.jsx   FindEventSection.jsx   NewEvent.jsx 2   http.js
import EventForm from './EventForm.jsx';

export default function NewEvent() {
  const navigate = useNavigate();

  function handleSubmit(formData) {}

  return (
    <Modal onClose={() => navigate('../')}>
      <EventForm onSubmit={handleSubmit}>
        <>
          <Link to="../" className="button-text">
            Cancel
          </Link>
          <button type="submit" className="button">
            Create
          </button>
        </>
```

Acting on Mutation Success & Invalidating Queries

It tell React Query that the data fetched by certain queries is outdated now. It should be marked as stale and that an immediate refetch should be triggered if the query belongs to a component that's currently visible on the screen.



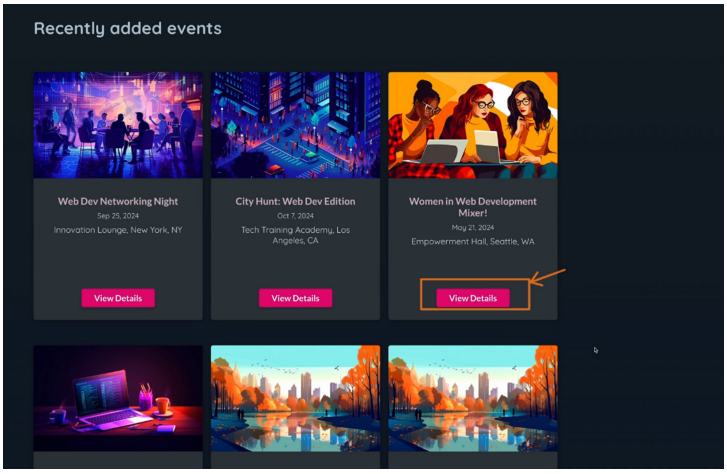
For example, I am creating a new event, of course the page that shows all the events is still visible on the screen. It's below this modal after all.



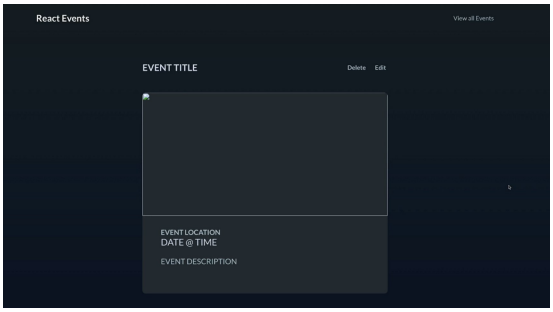
So, the query that was responsible for this section here after all should be re-executed.

A Challenge ! The Problem

If you click this button here

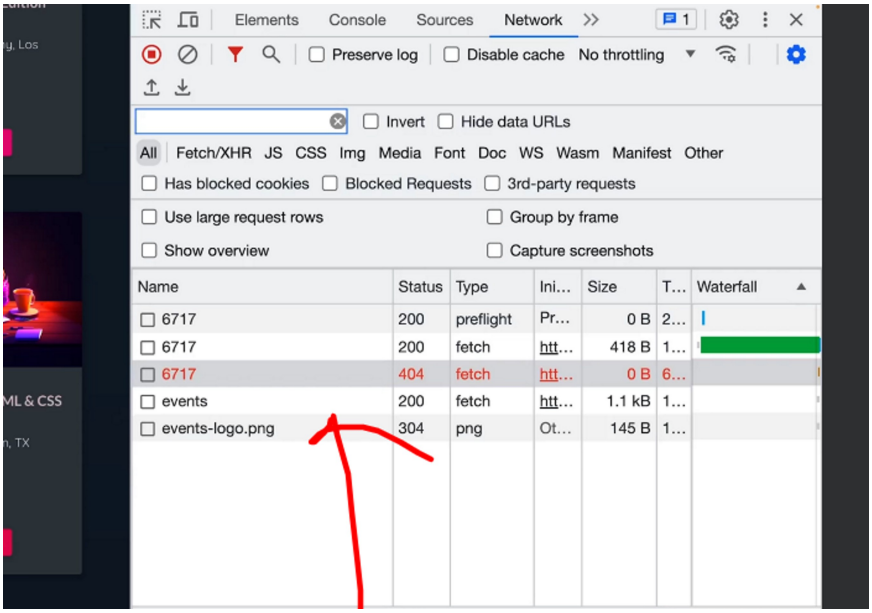


It will take you into details page, you job is now to make sure that as this page is loaded, the details data is loaded



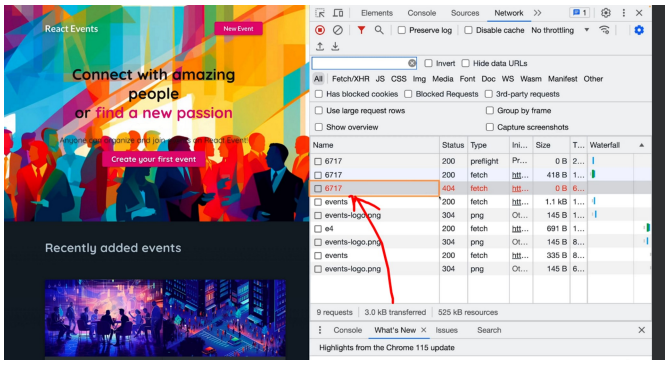
A Challenge! The Solution

We get this 404 error which we will solve next.

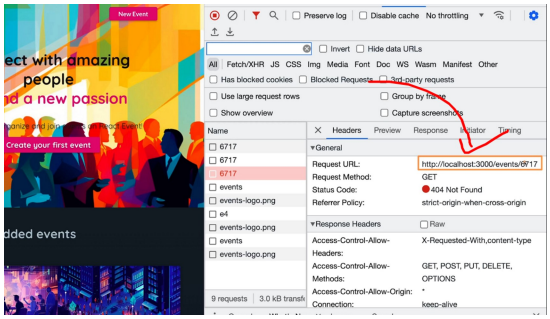


Disabling Automatic Refetching After Invalidations

If I delete an event and we are navigated back, I also got 404 request here.

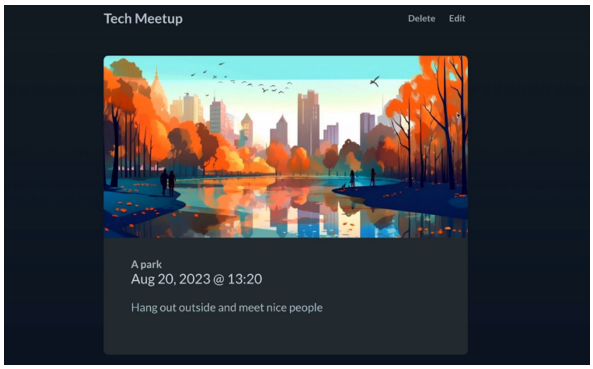


It was for one specific event with a specific id.



Why was this request event sent here, if we actually navigated back to the page that shows all events.

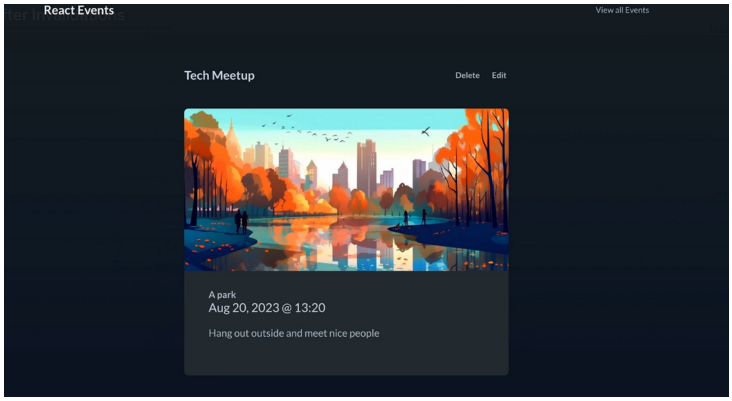
Because for deleting an event, we were on that details page and when then after deleting an event.



After deleting an event, we invalidated all the event related queries. (we still were on that page)

```
  queryFn: ({ signal }) => fetchEvent({ signal, id: params.id }),
});

const { mutate } = useMutation({
  mutationFn: deleteEvent,
  onSuccess: () => {
    queryClient.invalidateQueries({
      queryKey: ['events']
    });
    navigate('/events');
  }
});

function handleDelete() {
```

And therefore since we invalidated all queries, React Query went ahead and immediately trigger a refetch for this details query here.

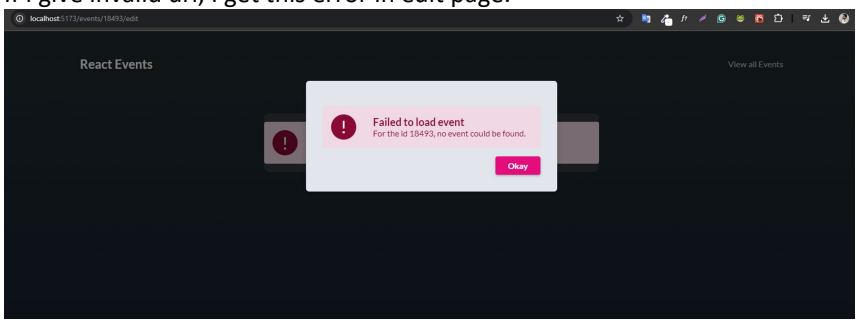


To avoid this behavior, add this property which makes sure when you call invalidate queries, these existing queries will not automatically be triggered again immediately. Instead they will just be invalidated and the next time when they are required, they will run again.

```
const { mutate } = useMutation({
  mutationFn: deleteEvent,
  onSuccess: () => {
    queryClient.invalidateQueries({
      queryKey: ['events'],
      refetchType: 'none'
    });
    navigate('/events');
  }
});
```

React Query Advantages in Action

If I give invalid url, I get this error in edit page.

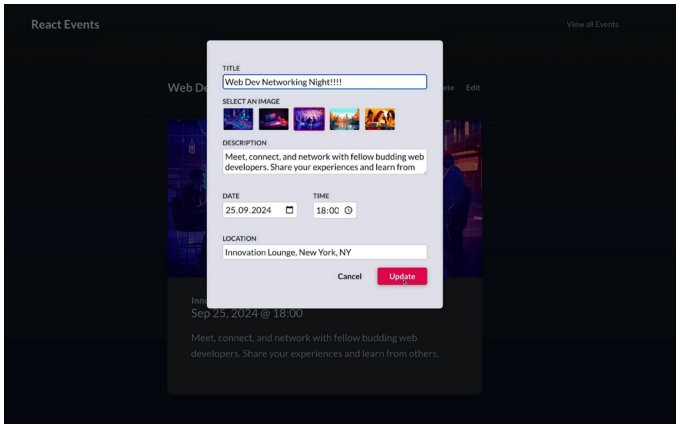


Updating Data with Mutations

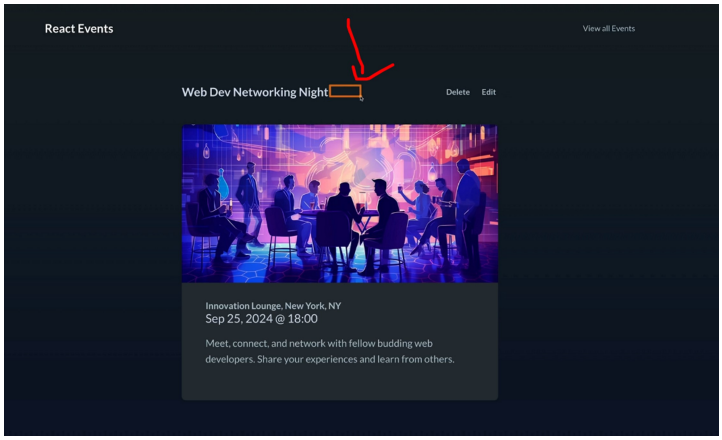
```jsx
export default function EditEvent() {

  const {data, isPending, isError, error} = useQuery({
    queryKey: ['events', params.id], //why id? because this query depends on the id of the event which
    queryFn: ({signal}) => fetchEvent({id: params.id, signal}),
  });

  const {mutate} = useMutation({
    mutationFn: updateEvent
  });

  function handleSubmit(formData) {       You, 31 minutes ago • Uncommitted changes
    mutate({
      id: params.id,
      event: formData
    });

    navigate('../');
  }

  function handleClose() {
    navigate('../');
  }
```

Optimistic Updating

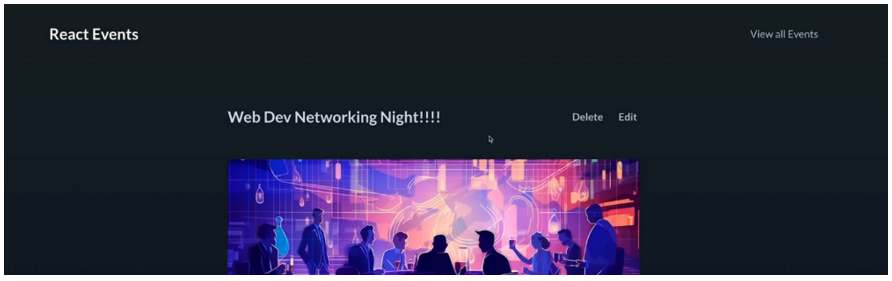
Our current updating has the problem whilst it works,

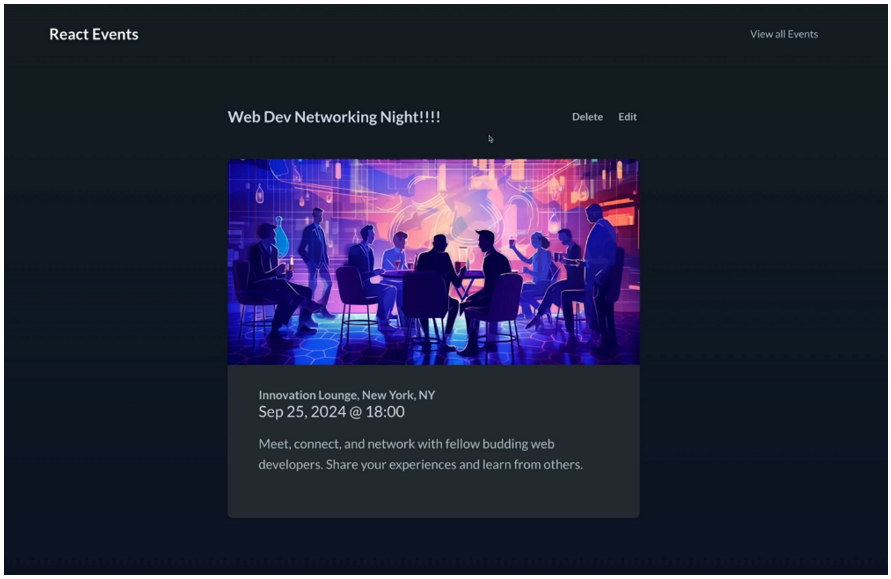


We're not reflecting the updated state on the page we're coming from



Unless we reload it.





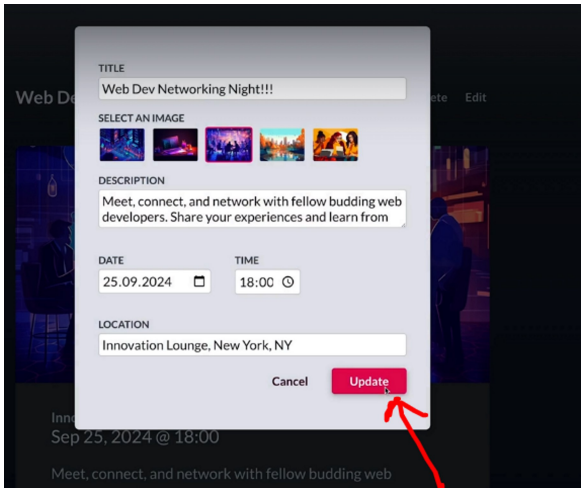
We could fix it by calling query client invalidate queries here and invalidate all the queries that need that data.

```
});

const { mutate } = useMutation({
  mutationFn: updateEvent,
  onSuccess: () => {
    queryClient.invalidateQueries()
  }
});

function handleSubmit(formData) {
  mutate({ id: params.id, event: formData });
  navigate('../');
}
```

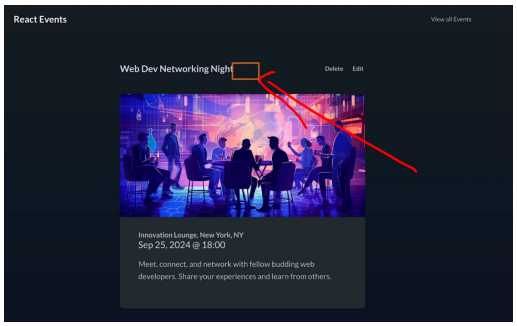
That would work here as well. But in that case you should wait with navigating away until this mutation is done.

Instead here I want to do something which is called ***optimistic updating.***

If I confirm my update here by pressing this update button.



I want to update this UI here instantly without waiting for response of my backend.



And if my backend then turns out to fail, if the update fails for whatever reason, I simply want to roll back the optimistic update I performed. This is an pattern or approach that's relatively easy to implement with React Query.

So to do that, we will ad onMutate:() > {}

This function will be executed when you call mutate();

```
const { mutate } = useMutation({
  mutationFn: updateEvent,
  onMutate: () => {

  }
});

function handleSubmit(formData) {
  mutate({ id: params.id, event: formData });
  navigate('../');
}
```

So, before this process is done, before you got back a response

Here I wanna update the data that's cached by React Query.

```
});

const { mutate } = useMutation({
  mutationFn: updateEvent,
  onMutate: async (data) => {
    const newEvent = data.event;

    await queryClient.cancelQueries({queryKey: ['events', params.id]});
    const previousEvent = queryClient.getQueryData(['events', params.id]

    queryClient.setQueryData(['events', params.id], newEvent);

    return {}
  },
  onError: (error, data, context) => {

  }
```

Using the Query Key As Query Function Input

Here under 'Recently added events', we are currently always showing all events which doesn't make lot of sense. It would be better if we would only see some events here.