UI Behaviours and Optimistic Rendering

Prior Knowledge

- Be able to perform http requests in React.
- Be able to catch errors from http requests.
- Be able to update React state.

Learning Objectives

- Understand how to align users expectations with UI updates.
- Understand optimistic rendering entails updating the UI before the back end processing has finished.
- Be able to perform state updates optimistically.
- Understand when it is appropriate to perform optimistic rendering.
- Understand how to give appropriate feedback if a request fails.

Updating UI

When requesting data from an API for a client side application the behaviour of GET requests is often intuitive. Making a fresh request gets the most up to date data which is static until a new request is made. Most users are familiar with this concept and it is expected behaviour however we need to be mindful of these expectations when making updates to the data via other methods, such as POST, PATCH etc. In particular API's being used by multiple users will cause multiple updates at the same time.

Refreshing data vs local updates

Consider the situation where you are displaying a list of data from an api. In this case, a list of cakes whenever the component is first loaded. If we have a component for POSTing new cakes to our API (the CakeAdder component, a form for example) we have to consider what happens after we have made that POST request and how to update the UI.

```
const CakeList = () => {
  const [cakes, setCakes] = useState([]);

  useEffect(() => {
    getCakesFromApi().then((cakesData) => {
      setCakes(cakesData);
    });
  }, []);

  addNewCake = (cakeFormData) => {
    postCakeToApi(cakeFormData).then((newCakeFromApi) => {
      // update the UI
    });
  };

  return (
    <section>
      <h2>Tasty Cakes</h2>
      <CakeAdder addNewCake={addNewCake} />
      <ul>
        {cakes.map((cake) => {
          return <CakeCard cake={cake} />;
        })}
      </ul>
    </section>
  );
};
```

Given that the `CakeAdder` component will invoke the `addNewCake` function after the user has filled out and submitted the form we need to give our users some visual feedback to reflect that change. What feedback the user receives varies on the UI we want but we'll focus on the `cakes` list for now. We have two choices here, either fetch new cake data from the API that will include our new cake or perform the UI update locally with the data we already have.

The UI a user would generally expect is to see the effect of their change, in this case the cake being added to the list is a good visual representation of the change so we can perform that UI update locally, no need to re-fetch the cakes data.

```
addNewCake = (cakeFormData) => {
  postCakeToApi(cakeFormData).then((newCakeFromApi) => {
    setCakes((currentCakes) => [newCakeFromApi, ...currentCakes]);
  });
};
```

A similar approach should be taken to other types of request that affect the state of the API. So a DELETE request would result in an item being removed from the page, a PATCH request would result in it being change etc. Exactly what these look like will depend on the UI but a good rule of thumb is to show the user the results of their action. Other actions can also be taken like confirmation popups, styling changes etc.

Here is an example of a component showing how many messages a user has sent

```
const MessageStats = () => {
  const [sentMessageCount, setSentMessageCount] = useState(0);

  useEffect(() => {
    api.getSentMessageCount().then((msgCount) => {
      setSentMessageCount(msgCount);
    });
  }, []);

  const handleSendMessageClick = () => {
    api.sendMessage().then((msgCount) => {
      setSentMessageCount(msgCount);
    });
  };

  return (
    // component html/css, including...
    <button onClick={handleSendMessageClick}>Send message</button>
  );
};
```

In this example, the `sentMessageCount` value is updated on two occasions:

- When the component is first rendered via `useEffect`
- When the user sends a message, via the `sendMessage` api function.

When we set the state after the user sends a message, we will be using the latest data from the database. However, if this were a 'real world' application, with many simultaneous users, the latest `sentMessageCount` may have jumped considerably in the time between the component mounting and the messages being sent.

At this point, as developers, we need to decide what we are trying to present here: a factual report of how many messages have been sent at that moment, or the impression that a user has had an impact by pressing the button. If it is the latter, then seeing the number increment by one might be more effective. This is demonstrated below:

```
const handleSendMessageClick = () => {
  api.sendMessage().then((msgCount) => {
    setSentMessageCount((currentCount) => currentCount + 1);
  });
};
```

Optimistic rendering

Optimistic rendering is the technique of rendering UI updates without confirmation of success from the back end based on the assumption that the request will succeed. There are several requests to a server that won't require any kind of server side validation and will be successful the vast majority of the time. Think of features such as clicking a like button or up-voting a users post. Facebook's servers will process a huge number of requests to like posts with almost all of them being successful.

No matter how fast the servers are however it will take some amount of time to process the request. In order to improve our UX we can assume that the request will succeed and give our user immediate feedback that their request has succeeded whilst it is processed in the background. When the request is eventually successful, we have no need to perform further updates, so can stay quiet.

It should be clear that this approach should be avoided in some circumstances - financial transactions, for examples, should never give the indication that something has happened when it hasn't, or vice versa. But there may be advantages in some cases to adopting the optimistic rendering approach.

Non-optimistic Approach

The example component above takes a non-optimistic approach. It's waiting for the response to the PATCH request before updating the UI. Notice that it doesn't use the msgCount from the API and is instead increasing the `sentMessageCount` by one to align with the users expectations.

```
const handleSendMessageClick = () => {
  // msgCount not used
  api.sendMessage().then((msgCount) => {
    // non-optimistic UI update - waiting for the response
    setSentMessageCount((currentCount) => currentCount + 1);
  });
};
```

Optimistic approach

By adopting an optimistic approach we assume that the request will be successful. In this case, we don't need the api response to perform our UI update and the server will correctly process this request in the vast majority of cases. This allows us to perform the UI update immediately whilst the api call is processed in the background. This gives us a much more responsive UX and gives the user feedback that their actions are happening immediately.

```
const handleSendMessageClick = () => {
  // update the UI optimistically
  setSentMessageCount((currentCount) => currentCount + 1);
  // perform the API request in the background
  api.sendMessage();
};
```

Error handling

If we are going to assume success then we can succeed silently. However there is always a chance of things going wrong, even if it's something like the network connection cutting out. We will need to handle these errors and give our users appropriate feedback. It may be enough to simply undo the change you made so that the user can try again or you may need to keep some err state to give your user more detailed feedback.

```
const MessageStats = () => {
  const [sentMessageCount, setSentMessageCount] = useState(0);
  const [err, setErr] = useState(null);

  useEffect(() => {
    api.getSentMessageCount().then((msgCount) => {
      setSentMessageCount(msgCount);
    });
  }, []);

  const handleSendMessageClick = () => {
    setSentMessageCount((currentCount) => currentCount + 1);
    setErr(null);
    api.sendMessage().catch((err) => {
      setSentMessageCount((currentCount) => currentCount - 1);
      setErr('Something went wrong, please try again.');
    });
  };

  return (
    // component html/css, including...
    {err ? <p>{err}</p> : null}
    <button onClick={handleSendMessageClick}>Send message</button>
  );
};
```

To summarise the differences between these approaches:

Optimistic rendering	Rendering on response
Both `setState` and the API request trigger simultaneously	`setState` only happens after the API response
Cannot use data from the API response	Data from the API response is available
Both actions will always trigger	`setState` will only trigger if the API response is not an error
Cannot accurately represent the database	Can accurately represent the database at the time of response

It is of course possible to use elements of both approaches. You may wish to have some effect immediately on triggering the handler - for example, disabling a button to prevent multiple requests from being made. You can also undo state changes should a request fail. If so, you will need to think carefully how to achieve this gracefully and without disorienting a user.