React and TypeScript: `useReducer` Hook

This lesson is a detailed analysis of typing the `useReducer` hook in TypeScript. It also provides a great use case for discriminated unions and state machines.

WE'LL COVER THE FOLLOWING

Overview

In this lesson, we will implement a common data fetching scenario with the `useReducer` hook. We will see how to take advantage of TypeScript's discriminated unions to correctly type reducer's actions. Finally, we will introduce a useful pattern for representing the state of data fetching operations.

Do we need a reducer?

We will base our code on an example from the official React documentation. The demo linked from this article is a simple implementation of a very common pattern, fetching a list of items from some backend service. In this case, we're fetching a list of Hacker News article headers.

What functionality is missing in this little demo? When fetching data from the backend it's useful to indicate to the user that an operation is in progress, and if the operation fails, to show the error to the user. Neither is included in the demo as coded.

The attached code uses `useState` hook to store the list of items after it is retrieved. We will need two additional pieces of state to implement our enhancements, a Boolean indicating whether an action is in progress and an optional string containing the error message.

We could use more `useState` hooks to store this information. There's a better way to do this, though. Notice that we're modifying multiple pieces of state at the same time as a result of certain actions. For example, when data is retrieved from the backend, we update both the data piece and the loading indicator piece. What's more, we're modifying the state in multiple places. Wouldn't it be cleaner and easier to follow if there was only a single place in the component where the state is updated?

We can achieve it by using the `useReducer` hook. It will allow us to centralize all state modification, making them easier to track and reason about.

The type of `useReducer`

Let's take a look at `useReducer`'s type signature. I simplified the code slightly by taking the initializer out of the picture.

```
function useReducer<R extends Reducer<any, any>>(
   reducer: R,
   initialState: ReducerState<R>
): [ReducerState<R>, Dispatch<ReducerAction<R>>];
```

Our hook is a function; yes, React hooks are just functions. It accepts two parameters: `reducer` and `initialState`.

The first parameter's type must extend `Reducer<any, any>`. `Reducer` is just an alias for a function type that takes a state object and an action and returns an updated version of the state object. In other words, the reducer describes how to update the state based on some action.

```
type Reducer<S, A> = (prevState: S, action: A) => S;
```

The second parameter allows us to provide the initial version of the state. `ReducerState` is a conditional type that extracts the type of the state from the

`Reducer` type.

Finally, `useReducer` returns a tuple. The first element in the tuple is the recent version of the state object. We will render our component based on the values contained in this object. The second item is a dispatch function. It is a function that will let us dispatch actions that will trigger state changes. Similar to `ReducerState`, `ReducerAction` extracts an action type from `Reducer`.

Behold the power of static typing, reading a well-typed function's signature is often enough to understand its purpose.

Typing state

Now is the time to fill in the gaps and define types representing state and actions.

It was already mentioned that apart from the data received from the server, we're also going to store a flag indicating if we're loading the data and an optional error message.

Therefore, the shape of state can be described with the following types:

```
type State = {
 data?: HNResponse;
 isLoading: boolean;
 error?: string;
}


type HNResponse = {
 hits: {
   title: string;
   objectID: string;
   url: string;
 }[]
};
```

`HNResponse` interface is based on the response received from https://hn.algolia.com/api/v1/search endpoint which we're going to use in this example. It is a free service that returns headers of Hacker News articles.

Typing actions with discriminated unions

An action is an object that represents some event in our application and results in the modification of the state. What kind of actions are there in our app?

- The first action describes the fact that the user typed some text into the search text field. This action will initiate a backend request.
- The second action will be triggered when the data from the backend is fetched. Action objects should contain this data.
- The third action will represent an error that occurred during data fetching. It should encompass the error message.

How do you represent the type of these actions in TypeScript? We can take advantage of a useful concept called discriminated union type.

```
type Action =
 | { kind: 'request' }
 | { kind: 'success', results: HNResponse }
 | { kind: 'failure', error: string };
```

`Action` is a union of three object types. What makes it special is the fact that all of those types have a common property called type. The type of this property in each interface is a different literal type. This lets us distinguish between those types.

Why is this useful? TypeScript creates automatic type guards for discriminated unions. This means that if we write an `if` statement in which we compare the type property of a given `Action` object with a specific `kind` (e.g. `success`), the type of the object inside the statement's body will be narrowed to the matching component of the union type.

For example, in the following code, accessing `action.results` will not cause a compile error because the type of action inside the body of the if statement will be appropriately narrowed!

```
function display(action: Action) {
  if (action.kind === 'success') {
    console.log(action.results);
  }
}
```

Implementing the reducer

We're all good to implement the reducer. As already mentioned, it takes a state and an action and returns an updated state.

For request actions, we're going to set the `isLoading` flag to true.

The success action will disable the `isLoading` flag and also set data to the results received from the server.

Finally, failure action will also disable the `isLoading` flag and set the error message.

```
function reducer(state: State, action: Action): State {
 switch (action.kind) {
   case 'request':
     return { isLoading: true };
   case 'success':
     return { isLoading: false, data: action.results };
   case 'failure':
     return { isLoading: false, error: action.error };
 }
}
```

Thanks to discriminated unions, we can access the action's properties inside case blocks in a type-safe way.

Using the hook

All that is left is to pass our reducer to `useReducer` hook.

```
const [{
  data,
  isLoading,
  error
}, dispatch] = useReducer(reducer, { isLoading: false });
```

I'm passing the reducer function to the hook along with the initial state which has `isLoading` set to `false` and the remaining properties `undefined`. The result is a pair with the current state object as the first element (which I'm instantly destructuring) and the dispatch function as the second element instantly destructuring) and the dispatch function as the second element.

Next, I need to update the usage of the `useEffect` hook so that it dispatches relevant actions.

```
useEffect(() => {
    let ignore = false;
    dispatch({ kind: 'request' });
    fetch(`https://hn.algolia.com/api/v1/search?query=${query}`)
        .then(
            (response: Response) => response.json().then(
                (results: HNResponse) => {
                    if (!ignore) dispatch({ kind: 'success', results });
                }
            ),
            (error: Error) => dispatch({ kind: 'failure', error: error.mes
sage })
        );

    return () => { ignore = true; }
}, [query]);
```

Finally, we should update the JSX to take the new pieces of the state into account and show the loading indicator and the error message when available.

```
return (
    <div>
        <input value={query} onChange={e => setQuery(e.target.value)} />
        {isLoading && <span>Loading...</span>}
        {error && <span>Error: {error}</span>}
        <ul>
        {data && data.hits && data.hits.map(item => (
            <li key={item.objectID}>
            <a href={item.url}>{item.title}</a>
            </li>
        ))}
        </ul>
    </div>
);
```

Below you can find the full implementation of the component. Note that we're still using the `useState` hook to store the query. This information is completely unrelated to the rest of the state, therefore there would be no completely unrelated to the rest of the state, therefore there would be no advantage in including it in the state managed by `useReducer`.

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import { Component, useState, useEffect, useReducer } from 'react';

type HNResponse = {
  hits: {
    title: string;
    objectID: string;
    url: string;
  }[]
};

type State = {
  data?: HNResponse;
  isLoading: boolean;
  error?: string;
}

type Action =
  | { kind: 'request' }
  | { kind: 'success', results: HNResponse }
  | { kind: 'failure', error: string };

function display(action: Action) {
  if (action.kind === 'success') {
    console.log(action.results);
  }
}

function reducer(state: State, action: Action): State {
  switch (action.kind) {
    case 'request':
      return { isLoading: true };
    case 'success':
      return { isLoading: false, data: action.results };
    case 'failure':
      return { isLoading: false, error: action.error };
  }
}

function App() {
  const [query, setQuery] = useState<string>();
  const [{
    data,
    isLoading,
    error
  }, dispatch] = useReducer(reducer, { isLoading: false });

  useEffect(() => {
    let ignore = false;

    dispatch({ kind: 'request' });
    fetch(`https://hn.algolia.com/api/v1/search?query=${query}`)
        .then(
            (response: Response) => response.json().then(
                (results: HNResponse) => {
                    if (!ignore) dispatch({ kind: 'success', results });
                }
```

```
          ),
          (error: Error) => dispatch({ kind: 'failure', error: error.message })
        );

    return () => { ignore = true; }
  }, [query]);

  return (
    <div>
      <input
        value={query}
        onChange={e => setQuery(e.target.value)}
        placeholder="Type to search" />
      {isLoading && <span>Loading...</span>}
      {error && <span>Error: {error}</span>}
      <ul>
        {data && data.hits && data.hits.map(item => (
          <li key={item.objectID}>
            <a href={item.url}>{item.title}</a>
          </li>
        ))}
      </ul>
    </div>
  );
}

ReactDOM.render(
    <App />,
    document.getElementById("root")
);
```

Even better state representation

If we take a look at the interface representing the state of this component, we will notice that some combinations of properties are not valid.

For example, it is not possible that `isLoading === true` while `data` is not empty.

Similarly, `error` and `data` cannot be both defined at the same time.

How can we improve this? Let's represent the state as a state machine!

```
type State =
  | { status: 'empty' }
  | { status: 'loading' }
  | { status: 'error', error: string }
  | { status: 'success', data: HNResponse }
```

Why is this approach better? Because it makes illegal states unrepresentable. The previous interface definition allowed certain combinations of property values even though we were sure that they would never occur in reality.

In a more complex component, this could force us to add some typecasts or handle impossible situations. Thanks to discriminated unions we can eliminate this issue.

It is generally a good idea to make your types match reality as closely as possible.

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import { Component, useState, useEffect, useReducer } from 'react';

type HNResponse = {
  hits: {
    title: string;
    objectID: string;
    url: string;
  }[]
};

type State =
  | { status: 'empty' }
  | { status: 'loading' }
  | { status: 'error', error: string }
  | { status: 'success', data: HNResponse };

type Action =
  | { kind: 'request' }
  | { kind: 'success', results: HNResponse }
  | { kind: 'failure', error: string };

function display(action: Action) {
  if (action.kind === 'success') {
    console.log(action.results);
  }
}

function reducer(state: State, action: Action): State {
  switch (action.kind) {
    case 'request':
      return { status: 'loading' };
    case 'success':
      return { status: 'success', data: action.results };
    case 'failure':
      return { status: 'error', error: action.error };
  }
}

function App() {
  const [query, setQuery] = useState<string>();
  const [state, dispatch] = useReducer(reducer, { status: 'empty' });

  useEffect(() => {
    let ignore = false;
    dispatch({ kind: 'request' });
    fetch(`https://hn.algolia.com/api/v1/search?query=${query}`)
        .then(

            (response: Response) => response.json().then(
                (results: HNResponse) => {
                    if (!ignore) dispatch({ kind: 'success', results });
                }
            ),
            (error: Error) => dispatch({ kind: 'failure', error: error.message })
        );

    return () => { ignore = true; }
  }, [query]);

  return (
    <div>
      <input value={query} onChange={e => setQuery(e.target.value)} />
      {state.status === 'loading' && <span>Loading...</span>}
      {state.status === 'success' && <ul>
        {state.data && state.data.hits && state.data.hits.map(item => (
          <li key={item.objectID}>
            <a href={item.url}>{item.title}</a>
          </li>
        ))}
      </ul>}
      {state.status === 'error' && <span>Error: {state.error}</span>}
    </div>
  );
}

ReactDOM.render(
    <App />,
    document.getElementById("root")
);
```

The final lesson looks into another advanced typing scenario.