Redux

State Management(Durum yönetimi) nedir?

State, uygulamalarımızdaki bileşenlerin her birinin o anda sahip olduğu özellikler ve bilgilerdir. Bu özellik ve bilgilerin değişkenlik göstermesi state denen kavramı ortaya çıkarıyor. Örneğin bir checkbox'ın işaretli olup olmaması bir durumdur, bir bilgidir. Bu bilgiyi uygulamamızda nasıl ilerleyeceğimizi belirlemek için kullanırız. Öte yandan bu checkbox'ın konumu, boyutu, şekli v.b. bilgiler sabit olduğu için bunu bir durum olarak belirtmemek daha mantıklı olur.(Tabii ki, bunları da birer durum olarak ele alabiliriz. Ben sadece yaygın olarak kullanım şeklini söyledim.).State management ise bu durumların her birinin yönetimini ve organizasyonunu sağlar.

Neden Redux?

Bir uygulama, Redux kullanmadan bile (prop drilling veya context api) durum yönetimini yönetebilir, ancak bazı problemlerle karşılaşabilir. Bunlar şunları içerebilir:

1. State yönetimi karmaşık hale gelebilir: Redux, bir uygulamanın durumunu merkezi bir yerde yönetmek için tasarlanmıştır. Bu nedenle, Redux kullanılmadan state management, birden fazla bileşen arasında durum değişikliklerini yönetmek için karmaşık ve zorlu olabilir.
2. Bileşenler arasında durum geçişi zor olabilir: Redux, bileşenler arasında durum geçişlerini yönetmek için tasarlanmıştır. Redux kullanılmadan bile, bileşenler arasındaki durum geçişleri daha az tutarlı olabilir ve yönetmesi daha zor olabilir.

3. Uygulamanın performansı etkilenebilir: Redux, state management için optimize edilmiştir ve uygulamanın performansını artırmak için tasarlanmıştır. Redux kullanılmadan bile, uygulamanın performansı, durum yönetimi nedeniyle etkilenebilir ve daha yavaş çalışabilir.
4. Kod tekrarı olabilir: Redux, uygulamalarda durum yönetimini kolaylaştırmak için tasarlanmış bir kütüphanedir. Redux kullanılmadan bile, durum yönetimi için kod tekrarı olabilir ve bu da uygulamanın bakımını zorlaştırabilir.

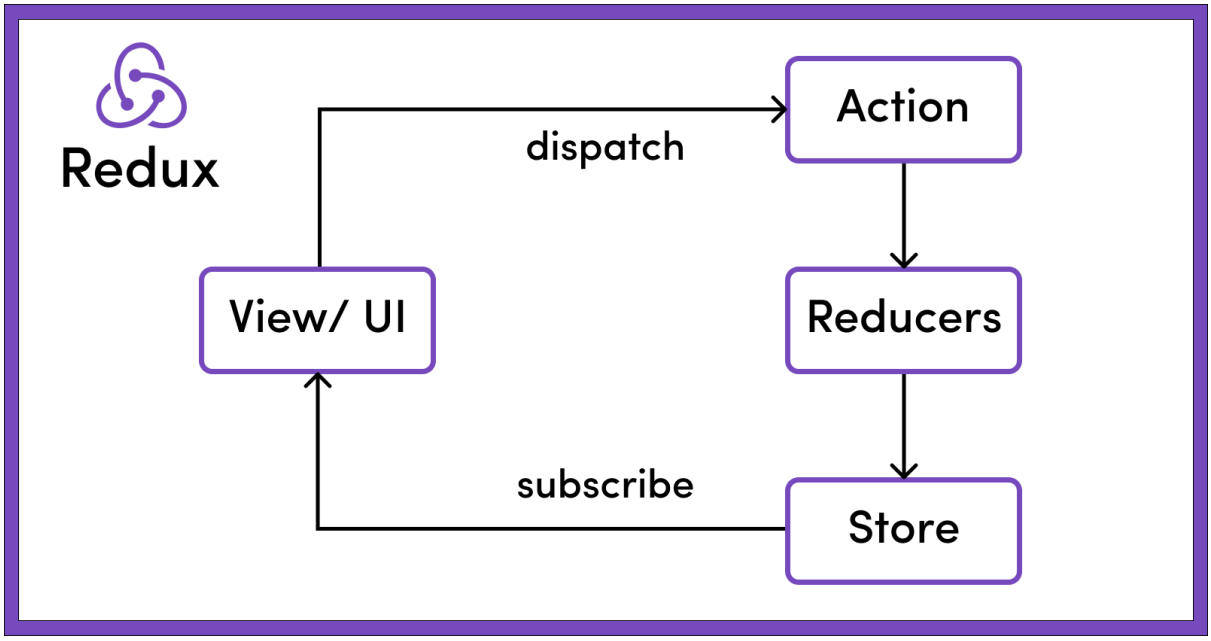
Bu nedenlerden dolayı, bir uygulama geliştiricisi Redux gibi bir durum yönetimi kütüphanesi kullanarak uygulamanın durumunu merkezi bir yerde yönetebilir ve uygulamanın daha öngörülebilir, daha kolay test edilebilir ve daha iyi performans göstermesini sağlayabilir.

Redux Nedir?

Redux, bir JavaScript uygulama durum yöneticisi (state management) kütüphanesidir. Redux, uygulama durumunun yönetimini kolaylaştırarak, uygulamanın daha öngörülebilir ve daha kolay test edilebilir hale gelmesini sağlar. Redux, özellikle büyük ve karmaşık uygulamalarda, birçok bileşenin durumunu koordine etmek için kullanılır.

Redux, tek bir "store" adı verilen merkezi bir veri deposu kullanır. Bu depo, uygulamanın durumunu tek bir yerde toplar ve uygulama boyunca kullanılabilir. Uygulama bileşenleri, Redux aracılığıyla bu depoya erişebilir ve durumu değiştirmek için "actions" adı verilen özel nesneler gönderebilirler. Bu eylemler, "reducers" adı verilen fonksiyonlar tarafından işlenir ve depodaki durumu günceller. Böylece, uygulama bileşenleri, Redux depodaki durum değişikliklerini takip edebilir ve güncellemeleri algılayarak yeniden render edebilirler.

Redux, özellikle React gibi bileşen tabanlı kütüphanelerle birlikte kullanıldığında etkilidir, ancak diğer JavaScript uygulama çerçeveleriyle de kullanılabilir. Redux, birçok farklı özelleştirme seçeneği ve eklentiyle birlikte gelir ve geniş bir topluluk tarafından desteklenmektedir.



Günlük Hayattan Örnek Vericek Olursak

Redux'un kullanılan yapıyı günlük hayattan bir örnekle açıklamak gerekirse, bir müşteri hizmetleri merkezini düşünebiliriz. Müşteri hizmetleri merkezi, müşterilerin sorunlarını çözmek için birçok farklı departmana sahiptir. Her departmanın kendi uzmanlık alanı ve sorunları çözmek için kullanılan araçları vardır.

Ancak, bir müşterinin sorunu birkaç departmanı ilgilendirdiğinde, sorunu çözmek daha karmaşık hale gelebilir. Bu noktada, merkezi bir sistem olan müşteri hizmetleri yönetim yazılımı devreye girer. Bu yazılım, müşterilerin sorunlarını takip etmek için bir merkezi veri deposu kullanır. Her departman, bu veri deposuna erişebilir ve müşterilerin sorunlarını güncelleyebilir. Ayrıca, bir müşterinin sorunu çözüldüğünde, tüm departmanlar bu değişikliği görebilir ve gerektiğinde sorunun takibini devralabilir.

Redux, birçok bileşenin durumunu koordine etmek için kullanılan benzer bir merkezi veri deposu yapısı kullanır. Her bileşen, bu depoya erişebilir ve durumu güncelleyebilir. Bu şekilde, bileşenler arasındaki iletişim daha kolay ve öngörülebilir hale gelir.

Redux kullanırken billinmesi gerekenler

Redux kullanırken bilinmesi gereken bazı konular şunlardır:

1. Store: Redux'ta durum yönetimi, bir "store" olarak adlandırılan merkezi bir veri deposunda yapılır. Store, uygulamanın tüm bileşenleri tarafından paylaşılabilir ve değiştirilebilir bir durum yönetimini sağlar.
2. Action: Redux'ta, bileşenlerin durumunu değiştirmek için "actions" adı verilen objeler kullanılır. Actions, tipi ve verisi olan bir objedir ve uygulamanın herhangi bir yerinde kullanılabilir.
3. Reducer: Reducer, Redux'ta durum değişikliklerini işleyen fonksiyonlardır. Bir action, reducer'a gönderilir ve durum değişikliğine neden olur. Reducer'lar, her bir durum parçası için ayrı ayrı yazılabilir ve birleştirilebilir.
4. Dispatch: Bileşenlerin durum değişikliklerini tetiklemek için Redux'ta "dispatch" fonksiyonu kullanılır. Dispatch, bir action'ı reducer'a gönderir ve durum değişikliğine neden olur.
5. Middleware: Redux'ta middleware, dispatch işlemini geçici olarak durdurabilen ve işleyebilen bir fonksiyondur. Middleware, logger, async request ve authentication işlemleri için kullanılabilir.
6. React-Redux: Redux ve React birlikte kullanıldığında, "react-redux" adı verilen bir paket kullanılır. React-Redux, React bileşenleri ile Redux arasındaki bağlantıyı kolaylaştırır ve bileşenlerin durumlarını Redux store'dan almasını sağlar.
7. Immutable state: Redux, durum yönetimi için immutable data (değiştirilemez veri) kullanılmasını önerir. Bu, durum değişikliklerinin bir kopyası oluşturulmasını ve orijinal verinin değiştirilmemesini sağlar.

Redux Store

Redux store, Redux kütüphanesinin merkezi bileşenidir ve uygulama durumunu (state) yönetmek için kullanılır. Redux, JavaScript uygulamalarında durumun tutulması ve yönetilmesi için popüler bir kütüphanedir.

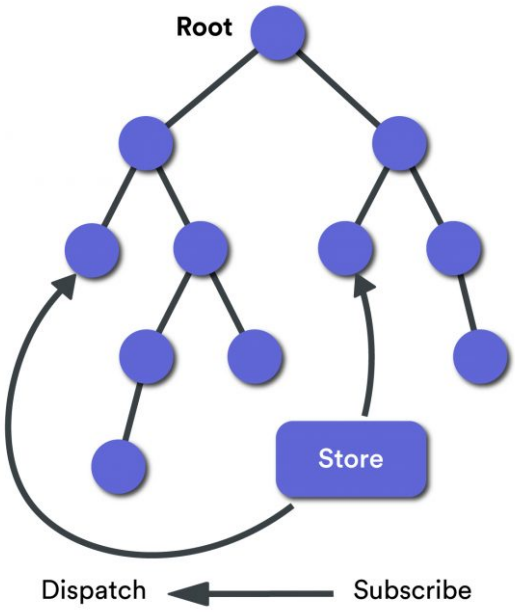
Redux store, uygulama durumunu tek bir yerde saklar. Uygulama durumu, uygulamanın mevcut durumunu temsil eden verilerin bir koleksiyonudur. Örneğin, bir web uygulamasında kullanıcı oturum açmış mı, sepete eklenen ürünler neler, açık olan modallar hangileri gibi bilgiler uygulama durumu olarak saklanabilir.

Store, uygulama durumunu değiştiren eylemleri alır ve bir sonraki durumu hesaplayarak günceller. Eylemler, uygulama durumunu değiştirmek için store'a

gönderilen birer nesnelerdir. Store, bu eylemleri birleştirir ve reducer (azaltıcı) fonksiyonunu kullanarak yeni bir durum (state) oluşturur. Sonuç olarak, store'da tutulan durum güncellenir ve bu durum, uygulama bileşenlerine yayılır.

Redux store, tek yönlü veri akışı prensibine dayanır. Bu prensibe göre, bileşenler sadece store'dan durumu alabilir ve durumu güncellemek için eylemler gönderebilir. Bu, uygulama durumunun tutarlı ve öngörülebilir olmasını sağlar.

Redux store, React ile kullanılmak üzere tasarlanmıştır, ancak Redux'in JavaScript ile herhangi bir uygulama geliştirmek için kullanılabilir.



Redux’ta Action

Redux'ta "action" (eylem), uygulama durumunu değiştirmek için Redux store'a gönderilen bir nesnedir. Bir action, uygulama durumunda gerçekleşen bir olayı temsil eder ve bu olayın gerçekleştiği bilgisini ve gerektiğinde ilgili verileri içerir.

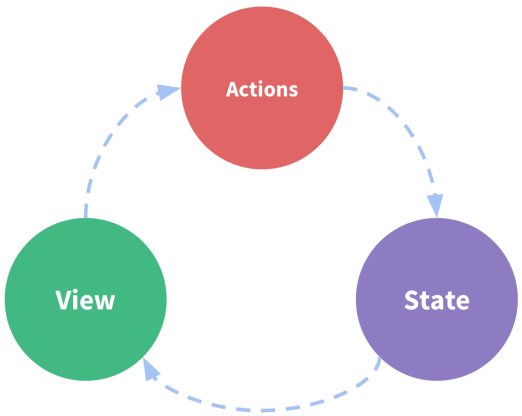
Bir action, genellikle bir JavaScript nesnesi olarak temsil edilir ve en azından bir "type" (tip) alanını içerir. Type alanı, eylemin ne tür bir olayı temsil ettiğini belirtir ve genellikle bir sabit string ifade olarak kullanılır. Örneğin:

```
const addToCart = {
  type: 'ADD_TO_CART',
  payload: {
    productId: 123,
    quantity: 2
  }
};
```

Yukarıdaki örnekte, `addToCart` isimli bir action tanımlanmıştır. `type` alanı, bu action'ın "ADD_TO_CART" olayını temsil ettiğini belirtir. `payload` alanı ise ilgili verileri içerir ve bu örnekte eklenen ürünün `productId` ve `quantity` bilgilerini içerir.

Action'lar, Redux store'a `dispatch` fonksiyonu aracılığıyla gönderilir. Bileşenler, kullanıcının etkileşimleri, veri alışverişi veya zamanlayıcılar gibi çeşitli olaylara tepki olarak action'ları dispatch edebilir. Store, gönderilen action'ı reducer (azaltıcı) fonksiyonuna ileterek, uygulama durumunun güncellenmesini sağlar.

Action'lar, uygulama durumunun nasıl değişeceğini açık bir şekilde tanımlamaya yardımcı olur ve Redux'in durum yönetimini tahmin edilebilir ve tekrarlanabilir hale getirir. Ayrıca, action'lar uygulama durumunun geçmişini izleme, test etme ve zamanında geri alma gibi geliştiriciye birçok avantaj sağlar.



Reduxta action nasıl dispatch edilir?

`useDispatch` kancası, React Hooks API'sinin bir parçasıdır ve Redux eylemlerini dispatch etmek için kullanılır. Bu kancayı kullanarak, bir React bileşeni içerisinde Redux store'a erişebilir ve eylemleri gönderebilirsiniz.

Öncelikle, `react-redux` paketini yüklemeli ve `useDispatch` kancasını import etmelisiniz:

Ardından, Redux store'unu `Provider` bileşeniyle sarmalamanız gerekir. Bu, uygulama genelinde Redux store'a erişimi sağlar. Örneğine daha sonra bakacağız.

```
import { useDispatch } from 'react-redux';

function MyComponent() {
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    // Eylemi dispatch etme
    dispatch({ type: 'INCREMENT' });
  };

  return (
    <div>
      <button onClick={handleButtonClick}>Artır</button>
    </div>
  );
}
```

Yukarıdaki örnekte, `useDispatch` kancası kullanılarak `dispatch` fonksiyonu elde edilir. Ardından, bir düğmeye tıklama gibi bir olay tetiklendiğinde `dispatch` fonksiyonu kullanılarak `INCREMENT_COUNTER` tipinde bir eylem gönderilir.

`useDispatch` kancası, bileşenin içindeki herhangi bir yerde eylemleri dispatch etmek için kullanılabilir. Bu sayede Redux store'a erişim sağlanır ve durumu güncellemek için eylemler gönderilebilir.

Redux’ta Reducer

Redux'ta bir reducer, uygulama durumunu güncellemek için kullanılan bir JavaScript fonksiyonudur. Bir reducer, bir önceki durum ve bir eylem alır ve yeni bir durum döndürür.

Reducerler, Redux'ta uygulama durumunu yönetmek için temel bir yapı taşıdır. Redux uygulamasında, tüm durum, uygulama genelinde tek bir JavaScript nesnesinde saklanır. Bu nesnenin güncellenmesi, uygulama içindeki tüm bileşenlerin güncellenmesine neden olur.

Reducerler, bir Redux store'da kullanılan state yönetimine yardımcı olur ve Redux store'da bulunan durumu güncellemek için kullanılır. Reducer'lar, bir eylemin nasıl ele alınacağını belirler. Her eylem, bir tip özelliği taşır ve bir reducer, belirli bir tip için durumu güncelleyen kodu içerir.

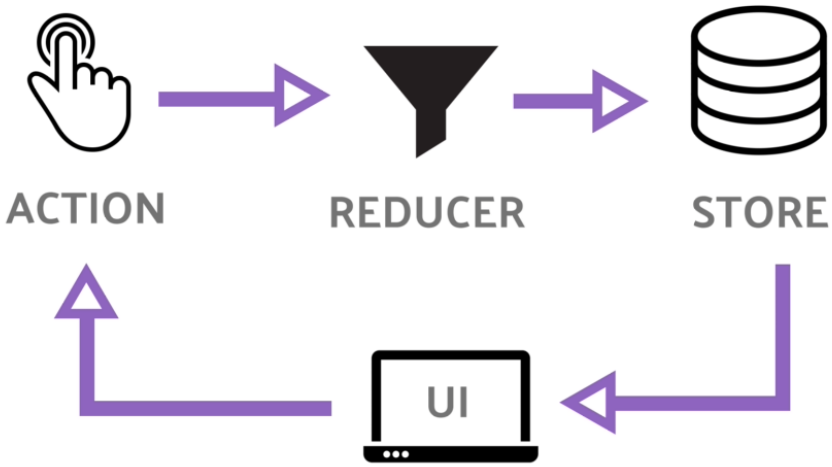
Reducer fonksiyonu, uygulama durumunu güncellemek için bir önceki durum ve bir eylem alır ve yeni bir durum nesnesi döndürür. Reducer fonksiyonu, Redux'ta her zaman saf bir fonksiyon olarak yazılmalıdır. Yani, bir önceki durum ve eylem üzerinde değişiklik yapmaz, bunları alır ve yeni bir durum nesnesi döndürür.

İşte bir örnek reducer fonksiyonu:

```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

Yukarıdaki örnekte, bir `counterReducer` adlı bir fonksiyon tanımlanmıştır. Bu fonksiyon, bir önceki durum ve bir eylem alır. Eylemin türüne bağlı olarak, durumu güncelleyen kod çalıştırılır ve yeni bir durum nesnesi döndürülür. `switch` ifadesi, eylem türüne göre belirli bir kod bloğunu çalıştırmak için kullanılır. Eylem türü `INCREMENT` ise, durum 1 artırılır, `DECREMENT` ise durum 1 azaltılır. Varsayılan durumda, mevcut durum nesnesi döndürülür.

Redux'ta, uygulama durumunu yönetmek için birden fazla reducer kullanabilirsiniz. Her reducer, yalnızca bir bölümü yönetir ve tüm reducer'ların bir araya gelmesiyle uygulama durumu oluşur.



Redux Kurulumu Nasıl Gerçekleşir

1- Paketleri kurun: Redux'u projenize eklemek için öncelikle Redux paketini kurmanız gerekmektedir. Bu işlem için, npm veya yarn paket yöneticisini kullanabilirsiniz.

```
npm install redux react-redux
```

veya

```
yarn add redux react-redux
```

2- Redux Store'u oluşturun: Redux'un merkezi bileşeni olan store'u oluşturmanız gerekmektedir. Store, uygulama durumunu yönetmek için kullanılır.

```
import { createStore } from 'redux';
import rootReducer from './reducers'; // Projede kullanılacak olan root reducer

const store = createStore(rootReducer);
```

Yukarıdaki kodda, `createStore` fonksiyonunu kullanarak store'u oluşturuyoruz. `rootReducer` adlı bir root reducer fonksiyonu, tüm reducer'ların birleştirildiği bir yapıdır. Bu, projenizdeki reducer'ları içeren bir dosyaya bağlı olarak değişiklik gösterebilir.

3- Store'u projenizde kullanın: Oluşturduğunuz store'u projenizde kullanmanız gerekmektedir. Genellikle `Provider` bileşeni ile projenin en üst düzeyinde

sarmalayarak, alt bileşenlerin store'a erişmesini sağlarsınız.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store'; // Redux store

import App from './App'; // Projedeki ana bileşen

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Yukarıdaki kodda, `Provider` bileşenini kullanarak store'u projenin en üst düzeyine yerleştiriyoruz. Böylece, içerideki bileşenler store'a erişebilir hale gelir ve storeda tutlan bütün verileri kontrol edebilirler.

4- Reducer'ları oluşturun: Projede kullanmak istediğiniz reducer'ları oluşturmanız gerekmektedir. Reducer'lar, durumun nasıl güncelleneceğini belirler. Reducerlar dispatch edilen aksiyonları izler ve aksiyonların type özelliğine göre storedaki veriyi günceller.

```
// Örnek reducer
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

export default counterReducer;
```

5- Eylemleri oluşturun: Eylemler(Action), durumu güncellemek için Redux store'a gönderilen nesnelerdir. Eylemler, genellikle bir tür (type) ve gerektiğinde verilerinizi içeren bir payload alanı içerir.

```
// Örnek eylemler
export const increment = () => {
  return {
    type: 'INCREMENT'
  };
};

export const decrement = () => {
  return {
    type: 'DECREMENT'
  };
};
```

Yukarıdaki örnekte, `increment` ve `decrement` adında iki eylem tanımladık. Her bir eylem, sadece bir tip (type) alanına sahiptir. Eylem fonksiyonları, genellikle bu tipi döndüren bir nesne döndürür.

6- Bileşenlerde Redux'u kullanın: Redux'u kullanmak istediğiniz bileşenlerde, `react-redux` paketinden sağlanan bazı kancaları kullanabilirsiniz. Örneğin, `useSelector` kancası durumu seçmek için, `useDispatch` kancası eylemleri göndermek için kullanılabilir.

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './actions';

function Counter() {
  const counter = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <p>Counter: {counter}</p>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
}
```

Yukarıdaki örnekte, `useSelector` kancasını kullanarak durumu seçtik ve `useDispatch` kancasını kullanarak eylemleri gönderdik. Böylece, bileşenin içindeki butonlara tıkladığınızda ilgili eylemler gönderilecek ve durum güncellenecektir.

Redux'u projenize entegre etmek için bu adımları izleyebilirsiniz. Bu adımları takip ederek, Redux'u kullanarak uygulama durumunu etkili bir şekilde yönetebilirsiniz.

useSelector() nedir?

`useSelector` kancası, Redux ile React uygulamaları arasında bir köprü görevi görerek Redux store'daki durumu seçmenizi sağlayan bir React Hooks fonksiyonudur.

Redux, uygulama durumunu merkezi bir depoda (store) saklar. Bileşenlerin bu duruma erişmesi gerektiğinde `useSelector` kancası kullanılır. Bu kancayı kullanarak, Redux store'da saklanan durumu seçebilir ve bileşeninizin bu durumu kullanmasını sağlayabilirsiniz.

`useSelector` kancası, bir seçici işlev ve bir bağımlılık dizisi alır. Seçici işlev, Redux store'daki durumu seçerken kullanılır ve bağımlılık dizisi, bileşenin yeniden render edilmesi gereken durum değişikliklerini belirtir.

İşte `useSelector` kancasının temel kullanımı:

```
import { useSelector } from 'react-redux';

function MyComponent() {
  const userData = useSelector((state) => state.users);

  // userData kullanarak işlemler yapabilirsiniz

  return (
    <div>
      {/* JSX */}
    </div>
  );
}
```

Yukarıdaki örnekte, `useSelector` kancasını kullanarak Redux store'daki `users` durumunu seçiyoruz. Bu durumu `userData` değişkenine atıyoruz ve bileşende kullanabiliriz.

Herhangi bir durum güncellemesi olduğunda, `useSelector` kancası otomatik olarak bileşeni yeniden render eder ve sadece bağımlılık dizisinde belirtilen durum değişiklikleri olduğunda bileşen güncellenir. Bu şekilde, Redux store'daki durum değiştiğinde bileşeniniz otomatik olarak güncellenir ve doğru verileri yansıtır.

`useSelector` kancası, Redux'un durumu seçmek için güçlü bir araçtır ve React bileşenlerinde Redux store'una erişimi kolaylaştırır.

Redux kullanırken dikkat edilmesi gerekenler

1. Redux'i gereksiz yere karmaşıklaştırmayın: Redux, büyük ölçekli ve karmaşık uygulamalarda kullanılmak için idealdir. Ancak küçük ve basit uygulamalar için Redux kullanmak gereksiz bir karmaşıklık olabilir. İhtiyacınıza uygun bir çözüm olduğundan emin olun ve gereksiz yere Redux kullanmaktan kaçının.
2. Durumu mümkün olduğunca küçük tutun: Redux, merkezi bir durum yönetimi sağlar. Ancak tüm uygulama durumunu Redux'ta saklamak yerine, sadece gerçekten paylaşılması gereken verileri ve uygulama durumunu tutun. Diğer bileşen özel durumlarını lokal state olarak kullanmak daha uygun olabilir.
3. Durum değişikliklerini takip etmek için Redux geliştirici araçlarını kullanın: Redux, durum değişikliklerini izlemek ve hata ayıklamak için geliştirici araçlar sağlar. Bu araçları kullanarak durumun nasıl değiştiğini ve eylemlerin nasıl etkilediğini takip edebilirsiniz. Bu, uygulama geliştirme sürecini kolaylaştırır.
4. Action tiplerini ve reducer'ları ayrı dosyalarda tutun: Action tipleri ve reducer'lar, Redux uygulamasının temel yapı taşlarından biridir. Bu nedenle, bu dosyaları ayrı dosyalarda tutmak, kodunuzun daha düzenli ve bakımı kolay hale gelmesini sağlar. Ayrıca, yeniden kullanılabilirliği artırır.

Redux Toolkit

Redux Toolkit, Redux tabanlı uygulama geliştirme için bir dizi kullanışlı araç ve önerilen bir yaklaşım sunan bir pakettir. Redux Toolkit, Redux'in temel prensiplerini korurken, geliştirme sürecini daha kolay hale getirmek ve kodun daha verimli olmasını sağlamak için tasarlanmıştır.

Redux Toolkit'in sağladığı bazı özellikler şunlardır:

1. Store Yapısının Basitleştirilmesi: Redux Toolkit, daha basit ve anlaşılır bir şekilde Redux store yapısını oluşturmanızı sağlar. Tek bir yerde yapılandırma yaparak,

çok sayıda tekrarlayan kod yazmak zorunda kalmazsınız.

2. Immutability ve İmmutable Update Yaklaşımının Kolaylaştırılması: Redux Toolkit, durumun değiştirilemezliği ve güncellenmesi için kolay bir yol sağlar. Özellikle, `createSlice` fonksiyonuyla oluşturulan reducer'lar, otomatik olarak durumu değiştirmek için gerekli olan doğru immutability yöntemlerini kullanır.
3. Redux Middleware'nin Kolaylaştırılması: Redux Toolkit, yaygın olarak kullanılan Redux middleware'leri (örneğin, Redux Thunk veya Redux Saga) kolayca entegre etmenizi sağlar. İhtiyaç duyulan middleware'leri tek bir yerde yapılandırmanızı ve kullanmanızı sağlar.
4. Performans İyileştirmeleri: Redux Toolkit, Redux'in performansını iyileştirmek için bazı optimizasyonlar sunar. Örneğin, `createSlice` ile oluşturulan reducer'lar, Redux'in performansını artıran bir dizi optimize edici işlem gerçekleştirir.
5. Modern JavaScript Özelliklerinin Kullanılması: Redux Toolkit, modern JavaScript dil özelliklerini (örneğin, destructuring, arrow fonksiyonları, kısa fonksiyon tanımlamaları) kullanarak kod yazmayı kolaylaştırır.

Redux Toolkit, Redux'i daha kolay ve verimli hale getiren bir dizi kullanışlı araç sağlar. Özellikle, büyük ve karmaşık Redux uygulamaları geliştirirken Redux Toolkit kullanmak, geliştirme sürecini daha hızlı ve daha düzenli hale getirebilir.

Redux Toolkit Kavramları:

configureStore() Nedir?

`configureStore`, Redux Toolkit'in sağladığı bir API'dir ve Redux store'unu oluşturmak için kullanılır. Bu API, Redux Toolkit'in sunduğu özellikleri etkinleştirmenizi ve middleware'leri yapılandırmanızı sağlar.

`configureStore` fonksiyonu, aşağıdaki gibi kullanılır:

```
import { configureStore } from '@reduxjs/toolkit';

const store = configureStore({
  reducer: rootReducer,
  middleware: [/* middleware'leri burada belirtin */],
  devTools: process.env.NODE_ENV !== 'production', // Geliştirici araçlarını etkinleştirme
});
```

Yukarıdaki örnekte, `configureStore` fonksiyonunu kullanarak Redux store'u oluşturuyoruz. İşlev, bir yapılandırma nesnesi alır ve bu nesne aracılığıyla store yapılandırması yapılır.

`reducer` özelliği, tüm reducer'ları birleştiren ve store'a atanacak olan bir kök reducer'ı belirtir. Örneğin, `rootReducer` şeklinde bir kök reducer'ınız varsa, onu burada belirtirsiniz.

`middleware` özelliği, Redux middleware'lerini belirtmek için kullanılır. Middleware'ler, eylemler ve reducer'lar arasında işlemler gerçekleştirmek için kullanılır. Örneğin, Redux Thunk veya Redux Saga gibi middleware'leri burada belirtebilirsiniz.

`devTools` özelliği, Redux geliştirici araçlarını etkinleştirme veya devre dışı bırakma seçeneğidir. Genellikle, geliştirme ortamında Redux geliştirici araçlarını etkinleştirirken, üretim ortamında devre dışı bırakılır.

`configureStore` fonksiyonu, daha önce Redux'da yapılandırma yapmak için ayrı ayrı yapılan adımları basitleştirir. Redux Toolkit tarafından önerilen en iyi uygulama yaklaşımlarını kullanarak bir Redux store'u oluşturmanıza olanak sağlar.

Bu şekilde, `configureStore` kullanarak Redux store'unu kolayca yapılandırabilir ve projenizde Redux'u kullanmaya başlayabilirsiniz.

combineReducers Nedir?

`combineReducers`, Redux'ta kullanılan bir yardımcı işlevdir. Bu işlev, birden fazla reducer'ı birleştirerek tek bir kök reducer oluşturmanıza olanak tanır. Kök reducer, Redux store'da kullanılacak olan tek bir reducer'dır.

`combineReducers` işlevi, aşağıdaki gibi kullanılır:

```
import { combineReducers } from 'redux';

const rootReducer = combineReducers({
  reducer1,
```

```
  reducer2,
  // Diğer reducer'ları buraya ekleyin
});

export default rootReducer;
```

Yukarıdaki örnekte, `combineReducers` fonksiyonu kullanılarak iki reducer (`reducer1` ve `reducer2`) birleştirilir ve `rootReducer` adlı bir kök reducer oluşturulur. Kök reducer, Redux store'unun durumunu yönetmek için kullanılır.

Her bir reducer, kendi bağımsız durumunu yönetir. `combineReducers` işlevi, her bir reducer'ın ilgili durum parçasını belirleyerek ve eşleştirerek kök reducer'ın durumunu oluşturur. Bu sayede, farklı reducer'lar farklı durum parçalarını yönetebilir ve birleştirilebilir.

Oluşturulan kök reducer, Redux store'a atanabilir ve diğer yapılandırma adımlarından geçirilerek bir Redux store oluşturulabilir.

`combineReducers` işlevi, Redux uygulamalarında reducer'ları modüler hale getirmek için kullanılır. Her bir reducer, belirli bir bölgedeki durumu yönetir ve birleştirilerek kök reducer oluşturulur. Böylece, büyük uygulamalarda reducer'lar daha yönetilebilir hale gelir ve bağımsız bir şekilde geliştirilebilir.

Bu şekilde, `combineReducers` kullanarak Redux'ta birden çok reducer'ı birleştirerek bir kök reducer oluşturabilir ve Redux store'unuzun durumunu etkili bir şekilde yönetebilirsiniz.

createSlice() Nedir?

`createSlice`, Redux Toolkit'in sağladığı bir API'dir ve Redux reducer'ını oluşturmayı kolaylaştırır. Bu API, Redux reducer'ının yapısını ve ilgili eylemleri otomatik olarak oluşturmanıza olanak tanır.

`createSlice` fonksiyonu, aşağıdaki gibi kullanılır:

```
import { createSlice } from '@reduxjs/toolkit';

const slice = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: (state) => state + 1,
```

```
    decrement: (state) => state - 1,
  },
});

export const { increment, decrement } = slice.actions;
export default slice.reducer;
```

Yukarıdaki örnekte, `createSlice` fonksiyonunu kullanarak bir Redux reducer'ı oluşturuyoruz. İşlev, bir yapılandırma nesnesi alır ve bu nesne aracılığıyla reducer yapısını tanımlar.

`name` özelliği, reducer'ın adını belirtir. Bu ad, Redux DevTools gibi araçlarda kullanılabilir ve reducer'ı tanımlayan bir etiket olarak hizmet eder.

`initialState` özelliği, reducer'ın başlangıç durumunu belirtir. Bu, reducer'ın ilk çalıştığında hangi durumla başlayacağını belirler.

`reducers` özelliği, reducer'ın işleyeceği eylemleri ve bunlara bağlı olarak durumun nasıl güncelleneceğini belirtir. Örnekteki `increment` ve `decrement` eylemleri, durumu artırmak ve azaltmak için kullanılır.

`createSlice` fonksiyonu, verilen yapılandırma nesnesi temelinde bir reducer ve ilgili eylemleri otomatik olarak oluşturur. Oluşturulan reducer, Redux store'a eklenirken kullanılabilir.

`slice.actions` ifadesi, oluşturulan eylemleri bir nesne olarak döndürür. Bu nesne, oluşturulan eylemleri diğer bileşenlerde kullanmanıza olanak tanır.

`slice.reducer` ifadesi, oluşturulan reducer'ı döndürür. Bu reducer, Redux store'da kullanılmak üzere birleştirilebilir.

`createSlice` API'si, Redux reducer'ının yapısını basitleştirir ve tekrarlayan kod yazımını azaltır. Ayrıca, durum güncellemelerini kolaylaştırır ve Redux Toolkit tarafından önerilen en iyi uygulama yaklaşımlarını kullanır.

Bu şekilde, `createSlice` kullanarak Redux reducer'larını kolayca oluşturabilir ve projenizde Redux'u kullanmaya başlayabilirsiniz.

Sliceda bulunan actionlar nasıl kullanılır?

Slice'daki eylemleri kullanmak için, oluşturulan slice nesnesinin `actions` özelliğine erişmeniz gerekmektedir. `createSlice` fonksiyonu tarafından döndürülen slice nesnesinin `actions` özelliği, oluşturulan eylemleri içeren bir nesneyi temsil eder.

```
import { createSlice } from '@reduxjs/toolkit';

const slice = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: (state) => state + 1,
    decrement: (state) => state - 1,
  },
});

export const { increment, decrement } = slice.actions;
export default slice.reducer;
```

Bir üstte yazptığımız slice örneğini ele alalım, `increment` ve `decrement` eylemleri oluşturulmuştur ve `slice.actions` nesnesi içinde yer almaktadır. Bu eylemleri kullanabilmek için `slice.actions` özelliğini import edebilir ve ardından ilgili eylemi kullanabilirsiniz.

Örneğin, bir bileşen içinde bu eylemleri kullanmak için:

```
import { increment, decrement } from './path/to/slice';

// Bileşen içerisinde...
const dispatch = useDispatch()

dispatch(increment()); // increment eylemini dispatch eder
dispatch(decrement()); // decrement eylemini dispatch eder
```

Yukarıdaki örnekte, `increment` ve `decrement` eylemleri dispatch edilirken `dispatch` fonksiyonu kullanılır. Bu eylemler, reducer tarafından yakalanarak Redux store'daki durumu güncellemektedir.

Ayrıca, eylemlerden önce import ettiğiniz `slice.reducer`ı da Redux store yapısına dahil etmeniz gerekmektedir. Böylece, eylemler dispatch edildiğinde, ilgili reducer çalışacak ve durum güncellenecektir.

Bu şekilde, slice'daki eylemleri kullanarak durumu güncelleyebilir ve Redux store'unda değişiklikleri tetikleyebilirsiniz.

Redux toolkit nasıl kurulur?

Redux Toolkit'ı kullanarak basit bir todo uygulaması oluşturmak için aşağıdaki adımları izleyebilirsiniz:

Adım 1: Proje Kurulumu

İlk olarak, boş bir proje oluşturun ve projenin klasörüne gidin. Ardından, aşağıdaki komutu kullanarak Redux Toolkit'i projenize ekleyin:

```
npm install @reduxjs/toolkit react-redux
```

Adım 2: Store Oluşturma

Projenizin kök dizininde, `src` klasörü içinde `store.js` adında bir dosya oluşturun. Bu dosyada, Redux store'unu oluşturmak için Redux Toolkit'i kullanacağız. İşte temel bir örnek:

```
import { configureStore } from '@reduxjs/toolkit';
import todoReducer from './reducers/todoReducer';

const store = configureStore({
  reducer: {
    todos: todoReducer,
  },
});

export default store;
```

Yukarıdaki kodda, `configureStore` fonksiyonunu kullanarak Redux store'unu oluşturuyoruz. `reducer` özelliğine, `todos` adında bir anahtarla `todoReducer`'ı ekliyoruz. Bu, todoReducer'ın todo durumunu yöneteceği anlamına gelir.

Adım 3: Reducer Oluşturma

`reducers` klasörü içinde `todoReducer.js` adında bir dosya oluşturun. Bu dosyada, todo durumunu yönetmek için bir reducer oluşturacağız. İşte basit bir örnek:

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = [];

const todoSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    addTodo: (state, action) => {
      state.push(action.payload);
    },
    deleteTodo: (state, action) => {
      return state.filter(todo => todo.id !== action.payload);
    },
    // Daha fazla eylemi buraya ekleyebilirsiniz
  },
});

export const { addTodo, deleteTodo } = todoSlice.actions;
export default todoSlice.reducer;
```

Yukarıdaki kodda, `createSlice` fonksiyonunu kullanarak bir slice oluşturuyoruz. `name` özelliği, slice'ın adını belirtir ve `initialState` özelliği başlangıç durumunu temsil eder. `reducers` özelliği, todo eylemlerini ve bu eylemlere bağlı olarak durumu güncelleyen işlevleri tanımlar.

Adım 4: Bileşen Oluşturma

`components` klasörü içinde `TodoApp.js` adında bir dosya oluşturun ve aşağıdaki gibi bir bileşen oluşturun:

```
const handleAddTodo = () => {
    if (text.trim() !== '') {
      dispatch(addTodo({
        id: Math.random().toString(),
        text: text.trim(),
      }));
      setText('');
    }
  };

  const handleDeleteTodo = (id) => {
```

```
    dispatch(deleteTodo(id));
  };

  return (
    <div>
      <input
        type="text"
        value={text}
        onChange={e => setText(e.target.value)}
      />
      <button onClick={handleAddTodo}>Add Todo</button>
      <ul>
        {todos.map(todo => (
          <li key={todo.id}>
            {todo.text}
            <button onClick={() => handleDeleteTodo(todo.id)}>Sil</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default TodoApp;
```

Yukarıdaki kodda, `TodoApp` bileşenindeki state ve dispatch işlevini almak için `useSelector` ve `useDispatch` kancalarını kullanıyoruz. `handleAddTodo` işlevi, todo eklemek için kullanılır ve `addTodo` eylemini dispatch eder. `handleDeleteTodo` işlevi, todo silmek için kullanılır ve `deleteTodo` eylemini dispatch eder.

Bileşen içinde bir `<input>` alanı ve bir "Add Todo" düğmesi bulunur. Kullanıcı bir todo girer ve "Add Todo" düğmesine tıkladığında `handleAddTodo` işlevi çağrılır ve yeni todo eklemek için `addTodo` eylemi dispatch edilir. `<ul>` içinde, todo listesi map fonksiyonu ile döngülenir ve her bir todo için bir `<li>` oluşturulur. Her todo için bir "Delete" düğmesi bulunur ve tıklanıldığında `handleDeleteTodo` işlevi çağrılır ve ilgili todo'nun `deleteTodo` eylemi dispatch edilir.

Adım 5: Uygulama Başlatma

`index.js` dosyasına gidin ve Redux store'unu oluşturduğumuz `store.js` dosyasını içe aktarın. Ardından, `<Provider>` bileşenini kullanarak uygulamanızı Redux store ile sarmalayın. İşte bir örnek:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import TodoApp from './components/TodoApp';

ReactDOM.render(
  <Provider store={store}>
    <TodoApp />
  </Provider>,
  document.getElementById('root')
);
```

Yukarıdaki kodda, `<Provider>` bileşeni Redux store'uyla sarmalanır ve `<TodoApp>` bileşeni içinde kullanılır. Bu sayede, Redux store, `<TodoApp>` bileşenine durumu ve dispatch işlevini sağlar.

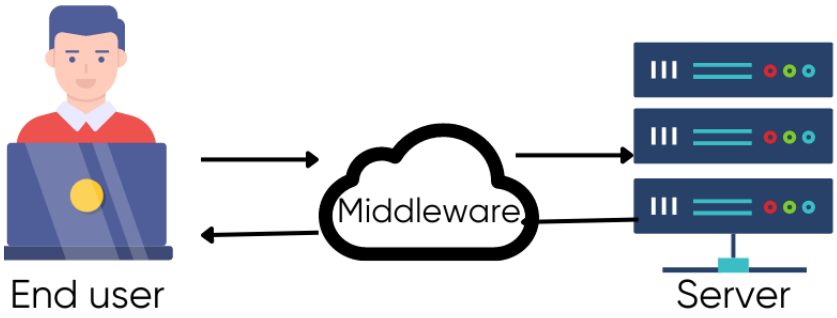
Bu şekilde, Redux Toolkit'i kullanarak basit bir todo uygulaması oluşturabilirsiniz. Redux Toolkit, Redux'u daha kolay ve hızlı bir şekilde kullanmanıza yardımcı olan bir dizi kullanışlı araç ve API'lar sağlar.

Middleware Nedir?

Middleware, bir yazılım sistemine gelen istekleri işlemek ve yanıtları oluşturmak için kullanılan bir yazılım bileşenidir. Genellikle web uygulamalarında ve API'lerde kullanılır. Middleware, isteklerin işlenme sürecinde araya girer ve farklı işlevler gerçekleştirerek isteği değiştirebilir, yanıtı işleyebilir veya ek işlemler yapabilir.

Bir middleware, bir isteği alır, gerekli işlemleri gerçekleştirir ve ardından isteği bir sonraki middleware veya son noktaya iletir. Middleware'ler genellikle bir zincir şeklinde bağlanır, böylece istek işleme süreci üzerinde farklı katmanlarda işlemler gerçekleştirilebilir.

Middleware'lerin kullanımı, bir uygulamanın farklı aşamalarında genel iş mantığını uygulamak, isteklerin doğrulanması, güvenlik önlemlerinin uygulanması, hata işleme, oturum yönetimi, günlükleme gibi işlevleri gerçekleştirmek için uygundur. Ayrıca, birden çok uygulama arasında kod paylaşımı yapmak veya özelleştirilmiş işlevselliği eklemek için de kullanılabilir.



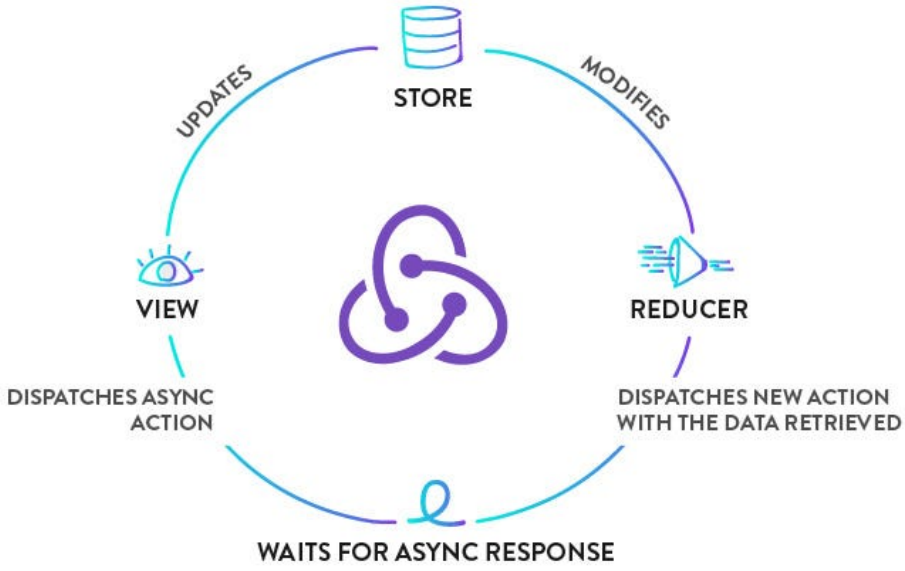
Redux Thunk Nedir?

Redux Thunk, Redux tabanlı uygulamalarda asenkron işlemleri yönetmek için kullanılan bir middleware'dir. Redux, uygulama durumunu yönetmek için kullanılan bir JavaScript kütüphanesidir. Redux Thunk, Redux'un eylem yaratıcıları (action creators) tarafından döndürülen işlevlerin kullanılmasını sağlayarak asenkron işlemlerin gerçekleştirilmesini kolaylaştırır.

Normalde, Redux eylemleri senkron işlemleri temsil eder. Ancak, bazı durumlarda verileri almak, sunucu çağrıları yapmak veya diğer asenkron işlemleri gerçekleştirmek gerekebilir. Redux Thunk bu durumlarda devreye girer. Eylem yaratıcıları, bir işlev döndürerek asenkron işlemleri gerçekleştirebilir. Bu işlev, Redux Thunk tarafından yakalanır ve eylemlerin asenkron olarak tetiklenmesini sağlar.

Redux Thunk kullanımı için genellikle ek bir kütüphane yüklemek gerekmez, çünkü Redux'un bir parçası olarak dağıtılır. Uygulama başlatılırken Redux Thunk middleware'i, Redux Store'a uygulanır. Böylece, Redux Thunk kullanılarak tanımlanan asenkron eylemler, Redux tarafından anlaşılır ve işlenir.

Örneğin, Redux Thunk kullanarak bir kullanıcının verilerini almak için aşağıdaki gibi bir eylem yaratıcısı tanımlayabiliriz:



Redux Thunk Kullanımı

Redux Thunk kullanmak için aşağıdaki adımları izleyebilirsiniz:

1. Redux Thunk'u yükleyin: Redux Thunk, redux-thunk paketi olarak yayınlanır. Projenizde kullanmak için projenizin bağımlılıklarına eklemek için aşağıdaki komutu kullanabilirsiniz:

```
npm install redux-thunk
```

2. Redux Store'u yapılandırın: Redux Thunk'u Redux Store'a eklemek için `applyMiddleware` fonksiyonunu kullanmanız gerekmektedir. Aşağıdaki gibi yapılandırabilirsiniz:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

Yukarıdaki kodda, `thunk` middleware'i `applyMiddleware` fonksiyonuna eklenir. `rootReducer` ise Redux Store'un ana reducer'ıdır.

3. Asenkron eylem yaratıcılarını tanımlayın: Asenkron işlemleri gerçekleştirmek için eylem yaratıcıları (action creators) işlevler olarak tanımlanır ve Redux Thunk tarafından yakalanır. Bu işlevler, genellikle asenkron API çağrıları, veri alışverişi veya diğer asenkron işlemleri içerir.

Örneğin, bir kullanıcının verilerini almak için aşağıdaki gibi bir eylem yaratıcısı oluşturabilirsiniz:

```
import axios from 'axios';

const fetchUser = () => {
  return (dispatch) => {
    dispatch({ type: 'FETCH_USER_REQUEST' });

    axios.get('/api/user')
      .then((response) => {
        dispatch({ type: 'FETCH_USER_SUCCESS', payload: response.data });
      })
      .catch((error) => {
        dispatch({ type: 'FETCH_USER_FAILURE', payload: error.message });
      });
  };
};
```

Yukarıdaki örnekte, `fetchUser` adlı eylem yaratıcısı bir işlev döndürür. İçindeki işlev, Redux Thunk tarafından yakalanır ve Redux tarafından çağrılır. İşlev, isteği başlatmadan önce bir yükleme isteği eylemi tetikler. Ardından asenkron bir işlem gerçekleştirmek için `axios` gibi bir kütüphaneyi kullanır. İşlem başarılı olduğunda `FETCH_USER_SUCCESS` türünde bir eylem tetiklenir ve alınan veriler payload olarak iletilir. İşlem başarısız olduğunda ise `FETCH_USER_FAILURE` türünde bir hata eylemi tetiklenir.

Kısaca asenkron aksiyonların oluşturulma şeması:

```
export const kisaYol = () => (dispatch) => {
     // asenkron işlemler
     // işlemler sonucu veri elde edilir
     // gelen veri store a aktarılır
  dispatch({ type: 'Akisyon Tipi', payload: 'Veri' });
};
```

4. Asenkron eylemleri tetikleyin: Asenkron eylem yaratıcılarını tetiklemek için `dispatch` fonksiyonunu kullanmanız gerekmektedir. Örneğin, bir bileşende bu eylem yaratıcısını tetiklemek için `mapDispatchToProps` fonksiyonunu kullanabilirsiniz:

```
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchUser } from '../actions';

const UserComponent = () => {
  const dispatch = useDispatch();
  const user = useSelector(state => state.user);

  useEffect(() => {
    dispatch(fetchUser());
  }, [dispatch]);

  return (
    <div>
      {user.loading ? (
        <p>Loading...</p>
      ) : user.error ? (
        <p>{user.error}</p>
      ) : (
        <div>
          <h2>User Details</h2>
          <p>Name: {user.name}</p>
          <p>Email: {user.email}</p>
        </div>
      )}
    </div>
  );
};

export default UserComponent;
```

Yukarıdaki örnekte, `useDispatch` ve `useSelector` kancalarını React Redux kütüphanesinden alıyoruz. `useDispatch` ile bir eylemi tetiklemek için dispatch fonksiyonunu alıyoruz. `useSelector` ile Redux Store'daki durumu seçiyoruz.

`useEffect` kancası, bileşen ekrana geldiği anda çalştırılan bir fonksiyondur. Bu örnekte, `fetchUser` eylem yaratıcısını (asenkron aksiyonu) tetiklemek için `dispatch` fonksiyonunu kullanıyoruz. `dispatch(fetchUser())` ifadesi, asenkron olarak kullanıcının verilerini getiren bir eylem tetikler.

Bileşen, Redux Store'daki `user` durumunu kullanarak kullanıcı bilgilerini görüntüler. Durum, `user.loading` (yükleniyor mu?), `user.error` (hata var mı?) ve `user.name` ve `user.email` gibi kullanıcı ayrıntılarını içeren bir nesne olarak kabul edilir. Duruma bağlı olarak uygun içeriği render eder.

Bu şekilde, Redux Thunk kullanarak asenkron işlemleri gerçekleştirebilir ve bileşenlerinizi bu işlemleri tetiklemek ve sonuçları kullanmak için kullanabilirsiniz.

Redux Thunk Yararları:

Redux Thunk kullanmanın bazı yararları şunlardır:

1. Asenkron İşlemleri Kolaylaştırır: Redux Thunk, asenkron işlemleri yönetmeyi kolaylaştırır. Eylem yaratıcıları, normalde tek bir eylem nesnesi döndürmek yerine işlev döndürebilir. Bu sayede asenkron işlemleri gerçekleştirmek için gerekli olan API çağrıları, veri alışverişi veya diğer asenkron işlemlerini daha kolay bir şekilde gerçekleştirebilirsiniz.
2. Daha İyi Durum Yönetimi: Redux Thunk, durumu daha iyi yönetmenizi sağlar. Asenkron işlemler sırasında, eylemler tetiklenir ve durum güncellenir. Bu sayede kullanıcı arayüzünde uygun geribildirim sağlayabilir veya yüklenme durumlarını yönetebilirsiniz.
3. Middleware Esnekliği: Redux Thunk, Redux'un middleware yapısını kullanır ve bu sayede uygulamanızın iş mantığına uygun olarak middleware zinciri oluşturmanızı sağlar. Farklı middleware'ler ekleyerek, güvenlik önlemleri, oturum yönetimi, hata işleme veya günlükleme gibi işlevleri kolayca entegre edebilirsiniz.
4. Test Edilebilirlik: Redux Thunk, test edilebilir kod yazmanıza yardımcı olur. Eylem yaratıcılarını ve middleware'leri ayrı ayrı test edebilirsiniz. Redux Thunk, eylemlerin asenkron olarak nasıl çalıştığını kontrol etmenizi ve test etmenizi sağlar.
5. Ekosistem Desteği: Redux Thunk, Redux ekosistemi içinde yaygın olarak kullanılan bir middleware'dir. Bu sayede Redux ile birlikte kullanılan diğer kütüphanelerle uyumluluğu sağlar ve geniş bir topluluk tarafından desteklenir.

Redux Thunk kullanmanın yararları, asenkron işlemleri kolaylaştırması, daha iyi durum yönetimi, middleware esnekliği, test edilebilirlik ve geniş ekosistem desteği gibi faktörlere dayanmaktadır. Bu nedenle, Redux tabanlı uygulamalarda asenkron işlemleri yönetmek için yaygın bir tercih haline gelmiştir.

Redux Toolkit Thunk

Redux Toolkit, Redux'u daha hızlı, daha kolay ve daha verimli hale getirmek için tasarlanmış bir Redux yardımcı kütüphanesidir. Redux Toolkit, Redux uygulamalarının geliştirilmesini kolaylaştırmak için bir dizi özellik, yardımcı fonksiyon ve standartlar sunar.

Redux Toolkit'un içinde bulunan bir özellik, Redux Thunk ile birleştirildiğinde Redux Toolkit Thunk olarak adlandırılır. Redux Toolkit Thunk, Redux Thunk'un Redux Toolkit ile birlikte kullanılmasını kolaylaştıran bir entegrasyondur.

Redux Toolkit Thunk, Redux Toolkit kullanırken asenkron işlemleri yönetmek için Redux Thunk'u kullanmanızı sağlar. Redux Toolkit Thunk, Redux Thunk'un getirdiği asenkron eylem yaratıcıları (action creators) ve middleware işlevselliğini Redux Toolkit ile birleştirir.

Redux Toolkit Thunk kullanmanın avantajları şunlardır:

1. Daha Basit ve Daha Az Kod: Redux Toolkit Thunk, Redux Toolkit ile birlikte kullanıldığında daha az boilerplate kodu gerektirir. Redux Toolkit'in sunduğu yardımcı fonksiyonlar sayesinde, daha kısa ve daha okunabilir eylem yaratıcıları ve azaltılmış yapılandırma kodu yazabilirsiniz.
2. Hızlı Entegrasyon: Redux Toolkit Thunk, Redux Toolkit ile sorunsuz bir şekilde entegre olur. İlk yapılandırmayı ve yapılandırma karmaşıklığını azaltır. Böylece, Redux Thunk'u Redux Toolkit projesine eklemek daha kolay hale gelir.
3. Yeni Başlayanlar İçin Kolaylık: Redux Toolkit Thunk, Redux'u öğrenen veya yeni başlayan geliştiriciler için daha kolay bir geçiş sağlar. Redux Toolkit'un sunduğu kolaylıklar ve Redux Thunk'un asenkron işlemleri yönetme yetenekleri, yeni başlayanların daha hızlı ve daha az zorlukla Redux tabanlı uygulamalar geliştirmelerini sağlar.
4. Redux Toolkit ile Uyumlu Ekosistem: Redux Toolkit Thunk, Redux Toolkit ile birlikte kullanıldığında Redux ekosistemi ile uyumlu olur. Redux Toolkit'in sunduğu diğer özellikler ve yardımcı fonksiyonlarla uyumlu çalışır ve diğer Redux kütüphaneleriyle sorunsuz bir şekilde entegre olabilir.

Özetlemek gerekirse, Redux Toolkit Thunk, Redux Toolkit ile Redux Thunk'un birleştirilmiş halidir. Redux Toolkit'in sunduğu kolaylıklarla birlikte Redux Thunk'un asenkron işlemleri yönetme yeteneklerini sağlar. Bu entegrasyon, Redux tabanlı uygulamaların geliştirilmesini daha kolay, daha verimli ve daha hızlı hale getirir.

Toolkit Thunk Kullanımı

Redux Toolkit Thunk'u kullanmak için aşağıdaki adımları izleyebilirsiniz:

1. Redux Toolkit'u yükleyin: Redux Toolkit, redux-toolkit paketi olarak yayınlanır. Projenizde kullanmak için projenizin bağımlılıklarına eklemek için aşağıdaki komutu kullanabilirsiniz:

```
npm install @reduxjs/toolkit
```

2. Redux Store'u yapılandırın: Redux Toolkit kullanmak için, Redux Store'u yapılandırmak üzere `configureStore` fonksiyonunu kullanmanız gerekmektedir. Bu fonksiyon, Redux Toolkit tarafından sağlanır ve Redux Store'un oluşturulmasını kolaylaştırır. Aşağıdaki gibi yapılandırabilirsiniz:

```
import { configureStore } from '@reduxjs/toolkit';
import thunkMiddleware from 'redux-thunk';
import rootReducer from './reducers';

const store = configureStore({
  reducer: rootReducer,
  middleware: [thunkMiddleware],
});
```

Yukarıdaki kodda, `configureStore` fonksiyonu kullanılarak Redux Store yapılandırılır. `reducer` parametresine kök reducer'ı (root reducer) ve `middleware` parametresine Redux Thunk middleware'ini eklersiniz.

3. Asenkron eylem yaratıcılarını tanımlayın: Redux Toolkit ile eylem yaratıcıları (action creators) oluşturmak için `createAsyncThunk` fonksiyonunu kullanabilirsiniz.

Bu fonksiyon, asenkron işlemleri yönetmek için Redux Thunk tarafından kullanılan bir yardımcı fonksiyondur. Örneğin:

```
import { createAsyncThunk } from '@reduxjs/toolkit';
import axios from 'axios';

export const fetchUser = createAsyncThunk('user/fetchUser', async () => {
  const response = await axios.get('/api/user');
  return response.data;
});
```

Yukarıdaki örnekte, `createAsyncThunk` fonksiyonu kullanılarak `fetchUser` adlı bir asenkron eylem yaratıcısı oluşturulur. Bu eylem yaratıcısı, 'user/fetchUser' türünde bir eylem oluşturur ve asenkron işlemi gerçekleştirmek için bir işlev tanımlar.

4. Slice'ları oluşturun: Redux Toolkit ile slice'lar oluşturmak, reducer'ları ve eylem yaratıcılarını bir araya getirir. Slice, durumun bölünmüş (sliced) bir parçasını temsil eder. Aşağıdaki gibi bir örnek:

```
import { createSlice } from '@reduxjs/toolkit';
import { fetchUser } from './actions';

const userSlice = createSlice({
  name: 'user',
  initialState: { data: null, loading: false, error: null },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUser.pending, (state) => {
        state.loading = true;
      })
      .addCase(fetchUser.fulfilled, (state, action) => {
        state.loading = false;
        state.data = action.payload;
      })
      .addCase(fetchUser.rejected, (state, action) => {
        state.loading = false;
        state.error = action.error.message;
      });
    },
});

export default userSlice.reducer;
```

Yukarıdaki örnekte, `createSlice` fonksiyonu kullanılarak bir `userSlice` oluşturulur. Bu slice, "user" adında bir dilim adı (name), başlangıç durumu (initialState), azaltıcılar (reducers) ve ekstra azaltıcılar (extraReducers) içerir.

`extraReducers` kısmı, asenkron eylemleri dinlemek ve durumu güncellemek için kullanılır. `builder` nesnesi üzerinde `addCase` fonksiyonu kullanarak, `fetchUser` eylemlerinin durumu nasıl etkilediğini belirtiriz. `fetchUser.pending`, `fetchUser.fulfilled` ve `fetchUser.rejected` eylemleri için durumu güncellemek için uygun kodları ekleriz.

5. Root Reducer'ı oluşturun: Tüm slice'ları birleştirmek için bir kök reducer (root reducer) oluşturmanız gerekmektedir. Aşağıdaki gibi bir örnek:

```
import { combineReducers } from '@reduxjs/toolkit';
import userReducer from './userSlice';

const rootReducer = combineReducers({
  user: userReducer,
});

export default rootReducer;
```

Yukarıdaki örnekte, `combineReducers` fonksiyonunu kullanarak `userReducer`'ı kök reducer'a ekleriz.

6. Store'u bileşenlere bağlayın: Redux Toolkit ile yapılandırılmış Redux Store'u bileşenlere bağlamak için `react-redux` kütüphanesini kullanabilirsiniz. Aşağıdaki gibi bir örnek:

```
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchUser } from './actions';

const UserComponent = () => {
  const dispatch = useDispatch();
  const user = useSelector((state) => state.user);

  useEffect(() => {
    dispatch(fetchUser());
  }, [dispatch]);

  return (
    <div>
      {user.loading ? (
        <p>Loading...</p>
      ) : user.error ? (
        <p>{user.error}</p>
      ) : (
        <div>
          <h2>User Details</h2>
          <p>Name: {user.data.name}</p>
```

```
            <p>Email: {user.data.email}</p>
          </div>
        )}
      </div>
    );
};

export default UserComponent;
```

Yukarıdaki örnekte, `useDispatch` ve `useSelector` kancalarını kullanarak Redux Store'u ve `fetchUser` eylem yaratıcısını bileşene bağlıyoruz. `useEffect` kancası ile bileşenin ilk render edildiğinde `fetchUser` eylemini tetikliyoruz. Bileşen, Redux Store'daki `user` durumunu kullanarak kullanıcı bilgilerini görüntüler.

Redux Toolkit Thunk'u kullanarak Redux tabanlı bir uygulama geliştirebilirsiniz. Redux Toolkit'un sağladığı kolaylıklar ve Redux Thunk'un asenkron işlemleri yönetme yetenekleri sayesinde, daha hızlı, daha verimli ve daha sürdürülebilir bir Redux deneyimi elde edebilirsiniz. Redux Toolkit Thunk, asenkron eylem yaratıcılarını kolayca tanımlamanıza ve Redux Store'unuzda asenkron işlemleri yönetmenize olanak tanır. Ayrıca Redux Toolkit'un sağladığı diğer avantajlarla birleşerek, daha az boilerplate kod yazmanızı ve daha iyi bir kod düzeni elde etmenizi sağlar.

Sonuç Olarak

Redux, güçlü bir durum yönetim kütüphanesi olup uygulamalarınızın durumunu tutmak ve güncellemek için etkili bir çözüm sunar. Bu dökümanın sizin için yararlı olduğunu umuyoruz ve Redux'u kullanmayı öğrenirken size rehberlik ettiğini umuyoruz.

Redux'un temel kavramlarını öğrenmek, durumu merkezi bir şekilde yönetmek ve bileşenler arasında veri akışını sağlamak için önemlidir. Eylemler, azaltıcılar ve mağazalar gibi kavramlar, Redux'un işleyişini anlamak için temel taşları oluşturur. Bu kavramları öğrenmek, uygulamanızın büyümesi ve karmaşıklaşması durumunda daha iyi bir kod tabanı oluşturmanıza yardımcı olacaktır.

Redux, büyük ölçekli uygulamalar geliştirirken kullanıcı deneyimini iyileştirmeye yardımcı olur. Birleşik bir durum deposunda tutulan veri, uygulamanın farklı bileşenlerinde tutarlılık sağlar. Aynı zamanda Redux, test edilebilirliği artırır ve gelecekteki değişikliklere uyum sağlamayı kolaylaştırır.

Redux topluluğu da bu güçlü kütüphane etrafında büyümektedir. Redux ile ilgili birçok kaynak, örnek proje ve yardımcı araç bulunmaktadır. Ayrıca, Redux'un yanı sıra Redux Thunk, Redux Saga ve Redux Toolkit gibi yardımcı kütüphaneler de mevcuttur,

bu kütüphaneler Redux'u daha da geliştirmek ve geliştirme sürecini daha kolay hale getirmek için tasarlanmıştır.

Son olarak, bu dökümanı okuduğunuz için size teşekkür etmek istiyoruz. Redux'u kullanmayı öğrenmek, uygulamalarınızı daha etkili bir şekilde yönetmek ve geliştirmek için önemli bir adımdır. Umarız Redux'un gücünü keşfetmek, uygulamalarınızı daha da geliştirmenize yardımcı olur.

İyi şanslar ve Redux ile başarılı projeler geliştirmenizi dileriz!