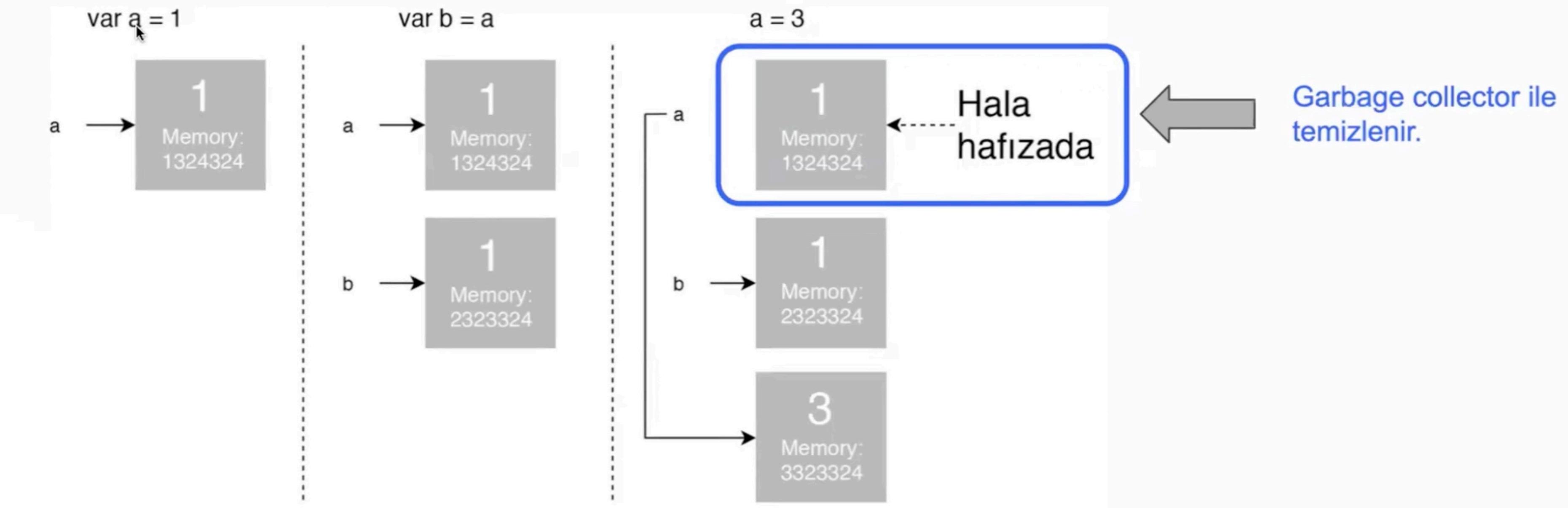


# JS'de Primitive Tipler **Immutable**'dır



# JS'de Immutability

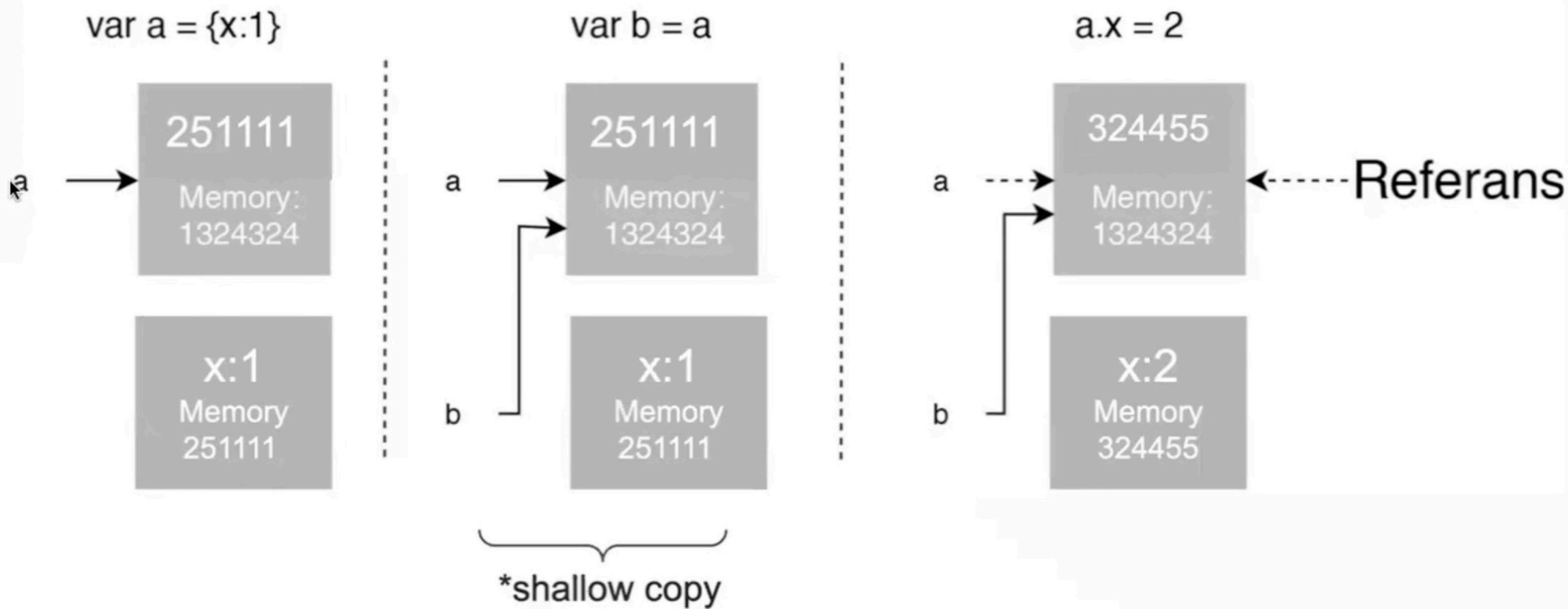
**Immutable:** Değişemez

- JS'de primitive tipler (String, Number, Boolean)

**Mutable:** Değişebilir

- JS'de complex tipler (Object, Array, Date)

# JS'de Complex Tipler **Mutable**'dır



# Yazılımda Immutability

Mutasyon değişimi gizler, bu da “beklenmedik” yan etkiler yaratabilir.

- Kodumuzda bazı buglar ortaya çıkabilir.
- Değişmezliği zorunlu kıلدığımızda, application'ı ve modelimizi basit tutabilir ve uygulama hakkında mantık yürütmemeyi kolaylaştırabiliriz.

değer değişimi **Immutable** bir **fonksiyon** ile yapılırsa:

- Değişimler takip edilebilir (Öngörülebilirlik).
- Hatalı değer değişimleri engellenebilir (Data geçerliliği).
- Tüm değişimler zaman çizgisine göre kaydedilip loglanabilir.
- Değişimle birlikte gerekli iş akışı ve fonksiyonlar tetiklenebilir.

```
const [counter, setCounter] = useState(0);

counter = 10; // HATA
setCounter(10); // DOĞRU

function setCounter(newCounter) {
    // Değer değişimi
    // + Değişim takip mekanizması
    // + Değişimi yönetme şansı
}
```

# Reducer Function

State datasını değiştiren özel bir fonksiyon çeşididir.

```
const reducerFunction = (state, action) => {
  //action'a göre yeni bir state oluşturma
  return newState;
}
```

- **state**: Önceki değer
- **action**: Gerçekleştirilecek olan ön tanımlı işlem objesi

```
const action = {
  type: "ARTTIR",
  payload: 5
}
```

# Reducer Function

1. State parametresine initial değer tanımla.
2. action type'lar için -if yerine- switch case'i tercih et.
  - o action, genelde içinde type ve payload property'leri olan bir objedir.

```
const action = {  
  type: "ARTTIR",  
  payload: 5  
}
```

3. Yeni bir state dönmeyi unutma: (immutability)
  - o obje veya array tipindeki stateler için **spread** operatörünü kullanabiliriz.

```
const reducerFunction = (state = initialState, action) => {  
  switch (action.type) {  
    case MINUS:  
      const newState = {  
        ...state,  
        total: state.total - action.payload,  
      };  
      return newState;  
  
    case ADD:  
      return {  
        ...state,  
        total: state.total + action.payload,  
      };  
  
    default:  
      return state;  
  }  
};
```

## useReducer Hook'u

**useState**'e alternatif olarak **state** oluşturmamıza yarar.

Reducer fonksiyon ile:

- Tüm değişimler bir merkezden yönetilir
- Yapılacak state değişimleri kısıtlanabilir
- Hatalı değer değişimleri engellenebilir

# useReducer Kullanımı

1. useReducer hook'unu ve reducer'ı import et.

```
import { useReducer } from "react";
import reducerFunction, { initialState } from "./reducers";
```

2. tanımla.

```
const [state, dispatch] = useReducer(reducerFunction, initialState);
```

- **state**
- **dispatch(actionObject)** : Action tetiklemek için kullanırız
- **reducerFunction**: Yazacağımız reducer fonksiyonu
- **initialState**: state'in başlangıç değeri

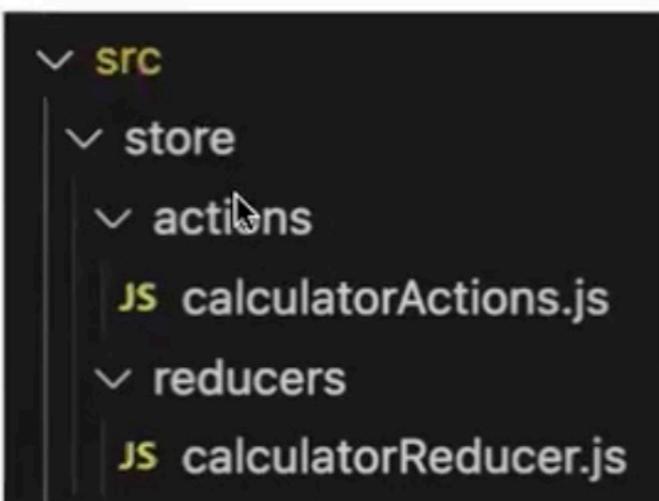
3. actionları dispatch et.

```
dispatch({ type: MEMORY_PLUS })
```

## Ekstra: Reducer Best Practices

### 1. dosya yapısını hazırla:

- o src/store içinde actions ve reducers klasörleri oluştur.
- o İçinde de actions.js ve reducers.js dosyalarını oluştur.



## Ekstra: Reducer Best Practices

### 2. actions dosyasında:

- **action type constant**'ları oluştur. (variable çünkü string değeri yanlış yazma ihtimali çok fazla)
- **action creator**'ları oluştur. (Action objesini oluşturan fonksiyonlar)
- bunları **export** etmeyi unutma.

```
export const CHANGE_OPERATION = "CHANGE_OPERATION";
export const CE = "CE_CLEAR_DISPLAY";

export const clearDisplay = () => {
  return { type: CE };
};

export const changeOperation = (operation) => {
  return { type: CHANGE_OPERATION, payload: operation };
};
```

# Ekstra: Reducer Best Practices

## 3. reducer dosyasında:

- o action type constant'ları import et.
- o initialState tanımla.
- o Reducer fonksiyonunda **Switch Case** kullan.
- o Yeni state return et.
- o reducer'ı export etmeyi unutma.

```
import { CE, CHANGE_OPERATION } from "../actions/calculatorActions.js"

export const initialState = {
  total: 0,
  operation: "-",
  memory: 0,
  temporary: 0,
};

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case CHANGE_OPERATION:
      return {
        ...state,
        operation: action.payload,
      };

    case CE:
      return {
        ...state,
        total: initialState.total,
        temporary: initialState.temporary,
      };

    default:
      return state;
  }
};

export default reducer;
```



# Ekstra: Reducer Best Practices

- useReducer Hook'unu kullan ve actionları action creator ile dispatch et.

- useReducer'ı react'tan, reducer'u ve action creator'ı oluşturduğum dosyalardan import et.
- useReducer'ı kullan.
- dispatch fonksiyonuna action creator ile oluşan action objesini gönder.

```
import { useReducer } from "react";
import reducer, { initialState } from "./reducers";

import { changeOperation, clearDisplay } from "./actions/calculatorActions.js";

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

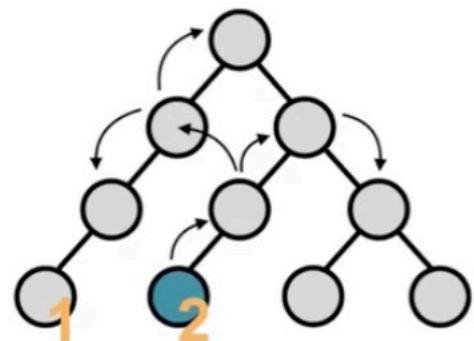
  return (
    <div className="App">
      <p>{state.total}</p>
      <CalcButton onClick={() => dispatch( clearDisplay() )} value={"CE"} />
      <CalcButton onClick={() => dispatch( changeOperation( "+" ) )} value={"+"} />
    </div>
  );
}

export default App;
```

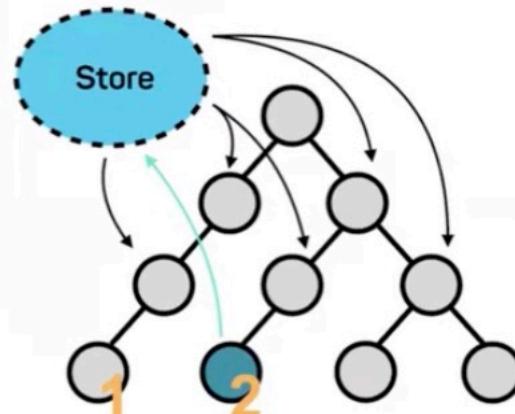
# Redux Hangi Problemi Çözer

- Componentler arası iletişim proplar aracılığı ile olur. Prop'lar da parent'dan child'a doğru iletilir.
- Çok componentli bir uygulamada component'ler arası prop taşımaya **prop drilling** denir. (kulaktan kulağa)
- Uygulamamız kompleksleştiğçe prop drilling performans kaybına neden olur, veri yönetimi zorlaşır ve gereksiz kod yazmaya neden olur.
- Redux'da tüm veriler merkezi bir store'da tutulur. Componentler de bu store'dan gerekli veriyi alırlar.

Without Redux



With Redux



# Redux Nedir?

JS uygulamaları için öngörülebilir state deposudur.

- **Öngörülebilir**(predictable): tutarlı çalışan uygulamalar geliştirmeye yardımcı olur => immutable reducer functions sayesinde
- **Merkezileştirilmiş**(centralized): merkezi bir state ve logic kullanarak undo/redo v.b. gibi işlemleri içeren güçlü bir uygulama oluşturmaya izin verir.
- **Hata ayıklanabilir**(debuggable): Redux Devtools extension'ı ile hata ayıklamayı kolaylaştırır.
- **Esnek**(flexible): Var olan bir çok addon sayesinde özellikleri genişletilebilir.

# Redux Kullanımı

1. redux ve react-redux kütüphanelerini projeye yükle.
2. store oluştur: `(./store/store.js)`
  - o reducer function'ı reducer dosyasında oluştur.
  - o action creator fonksiyonları ve action type constant'ları actions.js dosyasında oluştur.
3. uygulamayı provider ile sarmala ve provider'a store'u prop olarak ilet.
4. useSelector hook'u ile store'dan veri al.
5. useDispatch hook'u ile action'ları tetikle.

# Redux Kullanımı

1. redux ve react-redux kütüphanelerini projeye yükle.

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

2. store oluştur:

- o reducer function'ı reducer dosyasında oluştur.
  - i. action creator fonksiyonları ve action type constant'ları actions.js dosyasında oluştur.

```
import { legacy_createStore as createStore } from "redux";
import { reducer } from "./reducers/index.js";

export const store = createStore(reducer);
```

# Redux Kullanımı

- Uygulamayı provider ile sarmala ve provider'a store'u prop olarak ilet.

```
import { Provider } from "react-redux";
import { store } from "./store/store";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <Provider store={store}>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </Provider>
);
```

# Redux Kullanımı

- useSelector() hook'u ile store'dan veri al.

```
const initialState = {  
  title: "Redux ile React App çok daha güçlü!"  
}
```

```
export const reducer = (state = initialState, action) => {  
  switch (action.type) {  
    case CHANGE_TITLE:  
      return { title: action.payload }  
  
    default:  
      return state;  
  }  
};
```

} state'imizin  
yapısı

} reducer  
fonksiyonumuz

```
import { useSelector } from "react-redux";  
  
const AnaSayfa = ({ color = "black" }) => {  
  const title2 = useSelector(store => store.title);  
  
  return (<h2>{title2}</h2>);  
};
```

} useSelector ile  
title'i state'den alalım.

# Redux Kullanımı

- useDispatch(...) hook'u ile action'ları tetikle.

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

const AnaSayfa = (props) => {

  const dispatch = useDispatch();

  const changeTitle = (newTitle) => {
    dispatch({
      type: "CHANGE_TITLE",
      payload: newTitle
    })
  };
};
```

# Çok Sayıda Reducer ile Çalışmak

Tek bir reducer içinde uygulamanın tüm actionları için logic tanımlamak çok verimli değildir.

## Best practice:

- Bu yüzden her özelliğe göre ayrı ayrı reducer'lar oluştururuz. (userReducer, productReducer, paymentReducer v.b.)
- store.js dosyamızda **combineReducers()** ile reducerlarımızı birleştiririz.

```
import {combineReducers, legacy_createStore as createStore} from "redux";

import { userReducer } from "./reducers/userReducer.js";
import { paymentReducer } from "./reducers/paymentReducer.js";
import { productReducer } from "./reducers/productReducer.js";

export const reducers = combineReducers({
  user: userReducer,
  payment: paymentReducer,
  product: productReducer,
});

export const store = createStore(reducers);
```

# Çok Sayıda Reducer ile Çalışmak

Tek bir reducer içinde uygulamanın tüm actionları için logic tanımlamak çok verimli değildir.

## Best practice:

- Bu yüzden her özelliğe göre ayrı ayrı reducer'lar oluştururuz. (userReducer, productReducer, paymentReducer v.b.)
- store.js dosyamızda **combineReducers()** ile reducerlarımızı birleştiririz.

```
import {combineReducers, legacy_createStore as createStore} from "redux";

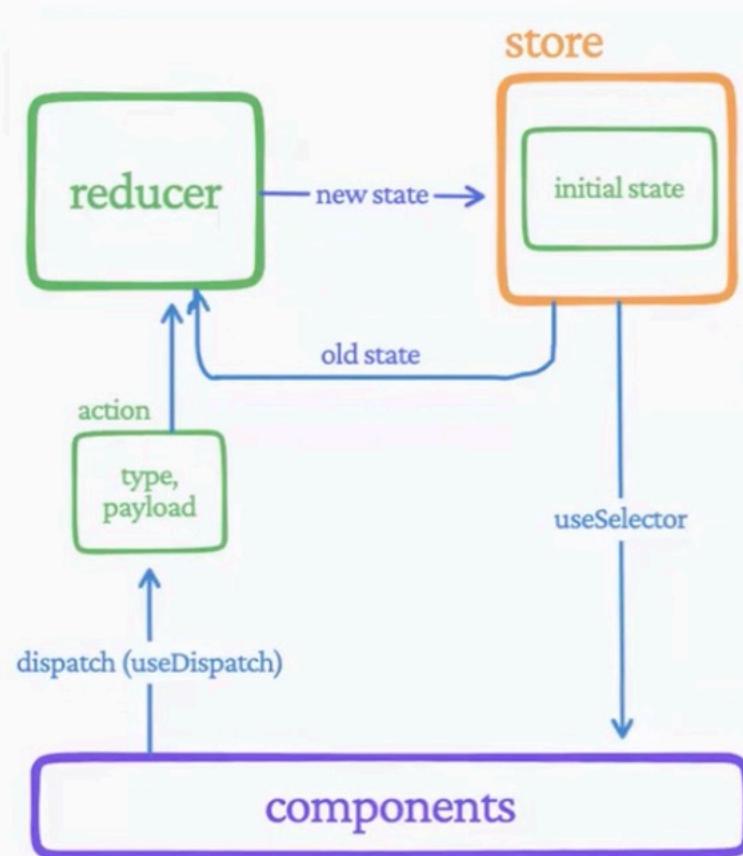
import { userReducer } from "./reducers/userReducer.js";
import { paymentReducer } from "./reducers/paymentReducer.js";
import { productReducer } from "./reducers/productReducer.js";

export const reducers = combineReducers({
  user: userReducer,
  payment: paymentReducer,
  product: productReducer,
});

export const store = createStore(reducers);
```

## Ekstra: Redux State Lifecycle

1. Uygulama ilk ayağa kalktığında redux store initial değerler ile oluşturuluyor.
2. Componentler, useSelector() hook'u ile store'dan veri alıyor.
3. Reducer'a, useDispatch() hook'u ile tetiklediğimiz action(object) iletiliyor.
4. Reducer, eski state'i store'dan alıyor.
5. Reducer, yeni state'i action.type'a göre oluşturuyor.
6. Oluşan yeni state store'a aktarılıyor.



# Middleware Kavramı

Çalışmakta olan bir process'i (işlem, süreç) durdurarak araya giren(intercept) katmana middleware denir.



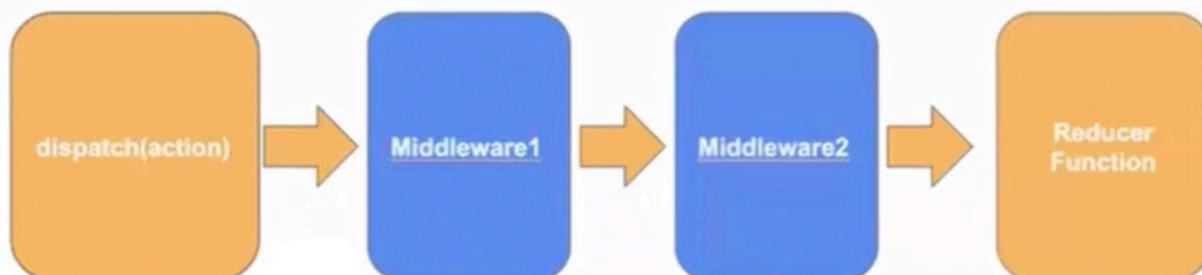
**Bir middleware:**

- Kod akışını durdurabilir
- Başka işlemler tetikleyebilir
- Kod akışını olduğu gibi devam ettirebilir

# Redux Middleware

Redux Middleware, **action dispatch ile reducer arasına giren katmandır.**

Birden fazla middleware aynı anda kullanılabilir. (Eklendikleri sırayla çalışırlar)



Redux Middleware, dispatch davranışının genişletilmesine yardımcı olur. Örn:

- Gerçekleşen action ve state değişimini loglayabiliriz.
- Asenkron işlemler gerçekleştirebiliriz. Bu sayede, tüm işlemleri tek merkezden yönetiriz.
- Yetki kontrolü yapabiliriz.

# Redux Middleware Kullanımı

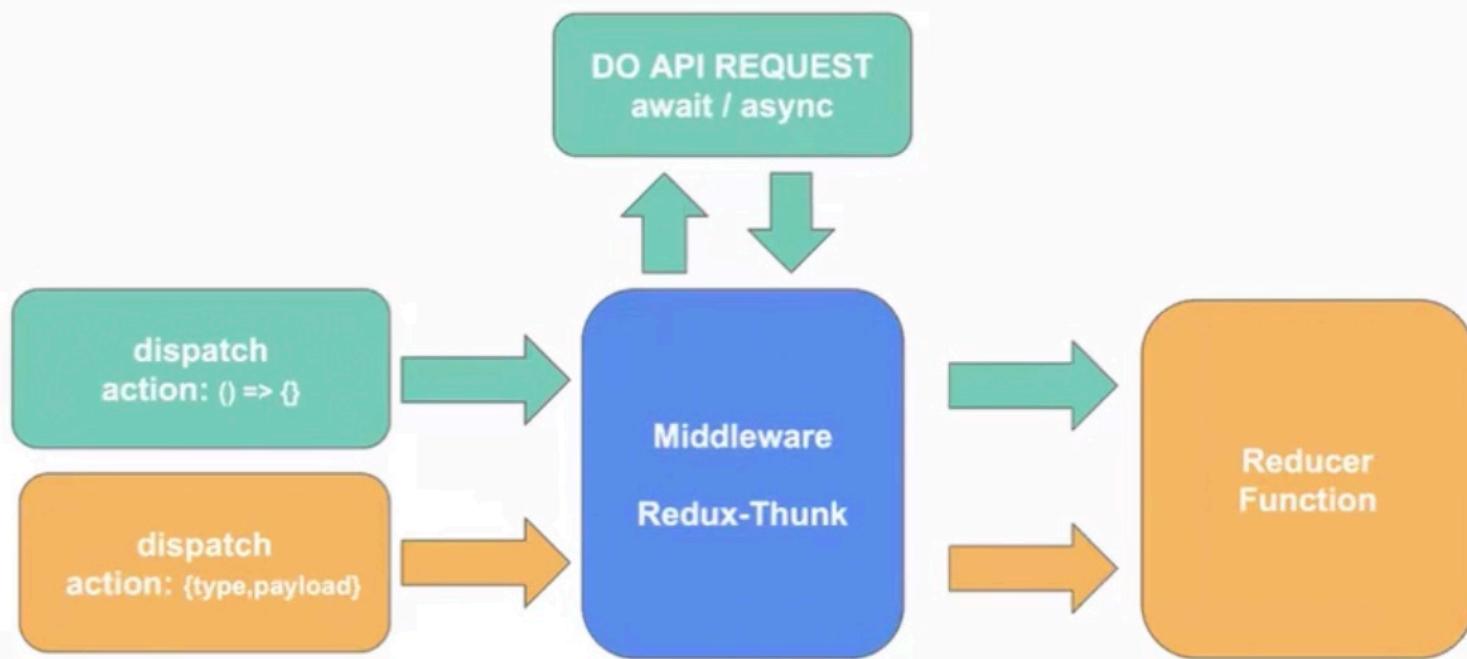
1. Middleware fonksiyonlarınımız argüman olarak `applyMiddleware(...)` metoduna çalışma sırasına göre veririz.
  2. `applyMiddleware`'ı de `createStore`'a 2. argüman olarak veririz.
- NPM registry'den custom middleware'ler bulup kullanabiliriz veya kendimiz custom middleware yazabiliriz.

```
import {  
  applyMiddleware,  
  legacy_createStore as createStore,  
} from "redux";  
  
import logger from "redux-logger";  
import customMiddleware from "../../middlewares/customMiddleware.js";  
  
export const store = createStore(reducers, applyMiddleware(customMiddleware, logger));
```

# Redux Thunk Middleware

Asenkron işlemler yapabilmemizi sağlayan özel bir middleware'dır.

- Genellikle bir **action** içinde **API Request** yapabilmek için kullanırız.
- Bir **action** fırlatıldığında, önce API request yapılır, bu esnada işlem bekletilir. Cevap geldiğinde ise işleme devam eder.



# Redux Thunk Middleware **Kullanımı**

1. Projeye yükle

```
▶ npm install redux-thunk
```

2. Middleware'lara ekle

```
import {applyMiddleware, legacy_createStore as createStore} from "redux";
import { thunk } from "redux-thunk";
export const store = createStore(reducers, applyMiddleware(thunk));
```

# Redux Thunk Middleware Kullanımı

- action creator'larda thunk function kullan.

```
export const fetchById = (id) => (dispatch) => {
  // yeni istek öncesi loading state'ini true yapıyoruz.
  dispatch({ type: FETCH_LOADING });

  // success veya error'da da loading false yapılır.

  axios
    .get("https://catfact.ninja/fact/" + id)
    .then(function (response) {
      dispatch({ type: FETCH_SUCCESS, payload: response.data });
    })
    .catch(function (error) {
      dispatch({ type: FETCH_ERROR, payload: error });
    })
};
```

```
const initialState = {
  favs: [],
  current: null,
  error: null,
  loading: true,
};
```

store'daki örnek state yapımız

```
case FETCH_SUCCESS:
  toast.success('İstek başarılı oldu...');
  return {
    ...state,
    loading: false,
    current: action.payload,
  };

case FETCH_LOADING:
  return {
    ...state,
    loading: true,
    current: null,
  };

case FETCH_ERROR:
  toast.error('Houston we have a problem!');
  return {
    ...state,
    loading: false,
    current: null,
    error: action.payload,
  };
};
```

reducer'daki caselerimiz

# Ekstra: Thunk Function

Bir fonksiyon başka bir fonksiyon tarafından oluşturuluyorsa ona **thunk function** denir.

```
export function logInUserAction(creds) {    // thunk action
  return function (dispatch, getState) {    // thunk function
    axios
      .post("/login", creds)
      .then((res) => {
        const loggedInAction = { type: USER_LOGGED_IN, payload: res.data.user };
        dispatch(loggedInAction);
      });
  };
}
```

Genellikle arrow yöntemi ile yazılır.

```
export const logInUserAction = (creds) => (dispatch, getState) => {
  axios
    .post("/login", creds)
    .then((res) => {
      const loggedInAction = { type: USER_LOGGED_IN, payload: res.data.user };
      dispatch(loggedInAction);
    });
};
```

# TanStack Query Nedir?

**Async State Management** yapabildiğimiz güçlü bir kütüphanedir.

- Caching:** Yapılan isteği hafızada tutarak, tekrar istenmesi durumunda bir daha request gerçekleştirmeden hafızadan kullanması
- Updating:** Cache'lenen datanın arka planda güncellenmesi, ne zaman güncellenmesi gerektiğini bilmesi.
- Performance Optimization:** Pagination veya Infinite scroll
- Data Sharing:** State manager gibi çekilen data componentler arası paylaşılabilir.

# TanStack Query Provider

```
import {  
  QueryClient,  
  QueryClientProvider,  
} from "@tanstack/react-query";  
  
export const queryClient = new QueryClient();  
  
function App() {  
  return (  
    <QueryClientProvider client={queryClient}>  
      <Main />  
    </QueryClientProvider>  
  );  
}  
;
```

# TanStack Query `useQuery`

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

export const Motivation = () => {
  const { isPending, error, data } = useQuery({
    queryKey: ["motivation"],
    queryFn: () =>
      axios
        .get("https://workintech-ng.onrender.com/motivation")
        .then((res) => res.data),
  });
}
```

```
if (isPending) return "Loading...";

if (error) return "An error has occurred: " +
  error.message;

return (
  <blockquote>
    <p>{data.word}</p>
    <cite>{data.author}</cite>
  </blockquote>
);
};
```

# queryKey, queryFn

1. **queryFn:** HTTP Requestini gerçekleştiren fonksiyondur.
  - o Bazı query'ler için parametre gerekebilir.
2. **queryKey:** Cacheleme için kullanılan anahtarıdır.
  - o requeste özel bir parametre varsa, o da queryKey array'ine eklenmelidir.

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

export const Motivation = () => {
  const { isPending, error, data } = useQuery({
    queryKey: ["motivation", page],
    queryFn: () => (page) =>
      axios
        .get(
          "https://workintech-ng.onrender.com/products?page=" + page)
        .then((res) => res.data),
  });
}
```

# useMutation

1.

Cache'deki verilerimizden biri değişeceği zaman kullanılır.

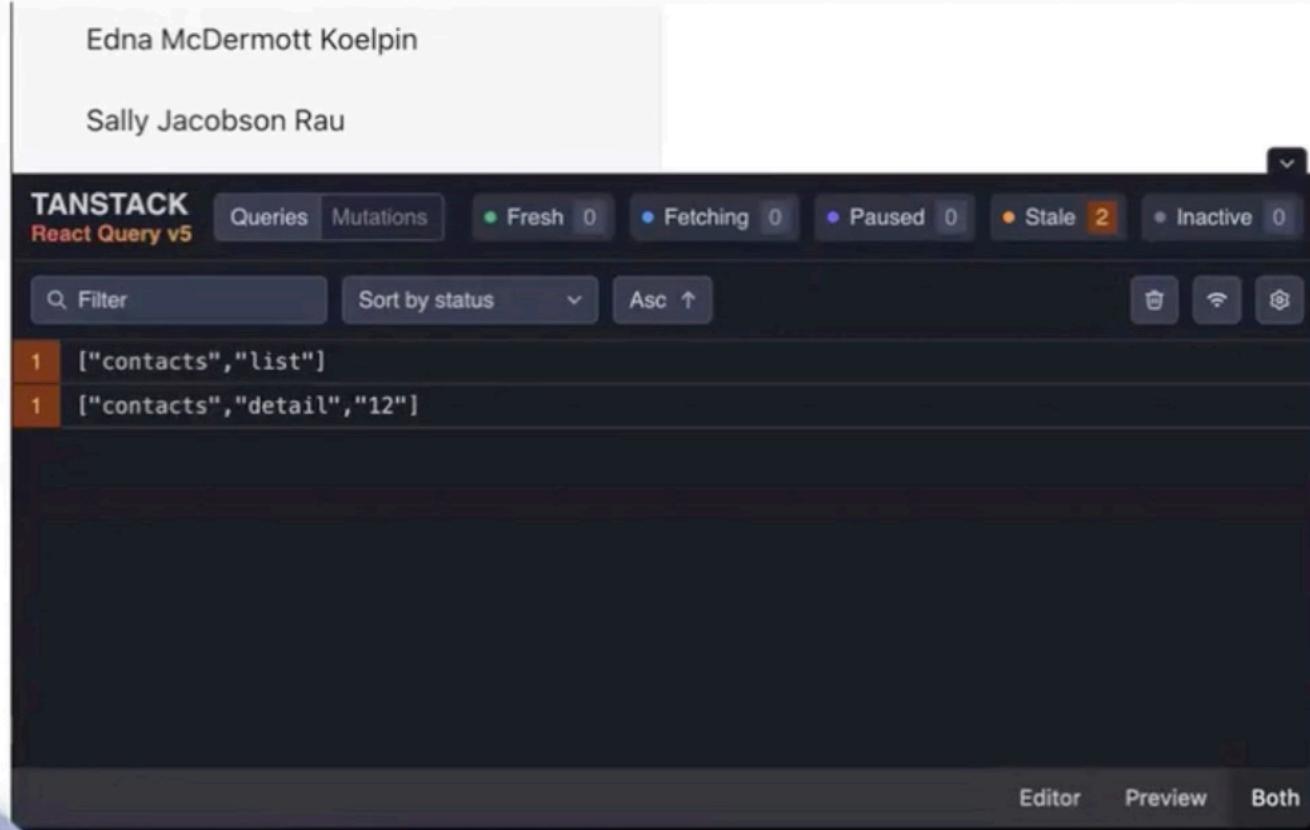
- **QueryClient** instance'ı oluşturulur.
- **invalidateQueries**'e queryKey array'i vererek bu query'nin artık invalid olduğunu söyleriz.
- onMutate, onSuccess, onError, onSettle event'leri vardır.

```
const queryClient = new QueryClient();
const mutation = useMutation({
  mutationFn: createProduct,
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ["products"] });
    history.push("/products");
  },
});

const productSubmitHandler = (e) => {
  e.preventDefault();
  mutation.mutate(newProduct);
};
```

# DevTools Eklentisi ile TanStack Query Takibi

@tanstack/react-query-devtools ile bütün server state işlemleri görsel olarak kolayca takip edebiliriz.



The screenshot shows a dark-themed code editor with a cursor pointing at the end of a line of code. The code uses the `<ReactQueryDevtools>` component to provide a developer tool interface for managing the query client. The code is as follows:

```
<QueryClientProvider client={queryClient}>
  <Main />
  <ReactQueryDevtools initialIsOpen={false} />
</QueryClientProvider>
```