

React Projelerinde Temiz Kod Yazmanın Önemi

7 min read · Jul 12, 2026



Furkan Yorulmaz

Follow



Share

Kod, makineye ne yapması gerektiğini anlatmak için değil; altı ay sonraki sana ve ekip arkadaşına ne düşündüğünü anlatmak için yazılır.

Bir React projesine ilk başladığınızda her şey masum görünür. Birkaç bileşen, biraz useState, bir iki fetch. Kod akıyor, özellikler yetişiyor, demo çalışıyor. Sonra proje büyür. Bileşenler 400 satıra ulaşır, useEffect'ler birbirini tetiklemeye başlar, props zincirleri altı katman aşağı iner ve bir gün küçük bir buton rengini değiştirmek istediğinizde kendinizi üç farklı dosyada, “buraya dokunursam nereyi kırarım?” diye ter dökerken bulursunuz.

İşte temiz kodun önemini tam olarak o an anlarsınız. Ama o an geldiğinde iş işten geçmiş olur.

Ben bir frontend geliştirici olarak şunu öğrendim: **temiz kod bir lüks değil, bir yatırımdır.** Bugün beş dakika fazla harcıyıp bir bileşeni doğru parçalara ayırdığınızda, gelecekteki kendinize saatler kazandırabilirsiniz. React özelinde ise bu iş biraz daha kritik, çünkü React size özgürlük veriyor — ve büyük özgürlük, büyük karmaşa potansiyeli demek. Bu yazıda, React projelerinde temiz kodun neden bu kadar önemli olduğunu ve bunu pratikte nasıl uyguladığımı anlatacağım.

Temiz Kod Neden Sadece “Estetik” Değil?

“Temiz kod” deyince akla çoğu zaman güzel girintiler, düzenli boşluklar gelir. Ama o kısmı zaten Prettier hallediyor. Gerçek temiz kod, **okunabilirlik ve değiştirilebilirlik** ile ilgilidir.

Bir istatistik hep aklımda kalmıştır: kod okumakla geçirdiğimiz süre, kod yazmakla geçirdiğimiz sürenin en az 10 katıdır. Yani biz aslında yazar değil, çoğunlukla okuruz. Kendi kodumuzu, ekip arkadaşımızın kodunu, üç ay önce yazdığımız ve tanımadığımız kodu... Temiz kod, bu okuma sürecini işkence olmaktan çıkarıp akıcı bir deneyime çevirir.

React'te temiz kodun getirdiği somut faydalar:

- **Daha az bug.** Küçük, tek işi olan bileşenlerde hata yapmak zordur; hata çıksa bile nerede olduğunu bulmak kolaydır.
- **Daha hızlı geliştirme.** Yeni bir özellik eklerken mevcut yapıyı anlamak için saatler harcamazsınız.
- **Daha kolay ekip çalışması.** Kodunuzu başka biri (ya da gelecekteki siz) açtığında kafası karışmaz.
- **Daha az korku.** İyi yapılandırılmış bir kodda değişiklik yapmak korkutucu olmaktan çıkar.

1. Bileşenler Küçük ve Tek İşli Olmalı

React'te en sık yapılan hata, bir bileşene her şeyi tıktırmaktır. Veri çekme, state yönetimi, iş mantığı, koşullu render, sekiz farklı JSX bloğu... hepsi tek bir Dashboard bileşeninde. Buna genelde “God Component” (tanrı bileşen) denir ve hiç iyi bir şey değildir.

Tek Sorumluluk İlkesi (Single Responsibility Principle) burada da geçerli: bir bileşen tek bir şeyi yapmalı ve onu iyi yapmalı.

Open in app ↗

Sign up

Sign in

Medium



Search



```
const [user, setUser] = useState(null);
```

```
const [posts, setPosts] = useState([]);
```

```
const [loading, setLoading] = useState(true);
```

```
useEffect(() => {
```

```
fetch(`/api/users/${userId}`).then(r => r.json()).then(setUser);

fetch(`/api/users/${userId}/posts`).then(r => r.json()).then(setPosts);

setLoading(false);

}, [userId]);

if (loading) return <div>Yükleniyor...</div>;

return (

  <div>

    <div className="avatar">

      <img src={user.avatar} alt={user.name} />

      <h1>{user.name}</h1>

      <p>{user.bio}</p>

    </div>

    <div className="posts">

      {posts.map(post => (

        <div key={post.id} className="post">

          <h3>{post.title}</h3>

          <p>{post.excerpt}</p>

        </div>

      ))}

    </div>

  </div>

);
```

```
}
```

Bu bileşen kullanıcı bilgisini de çekiyor, gönderileri de çekiyor, ikisini de render ediyor. Şimdi bunu okunabilir, parçalara ayrılmış haline çevirelim:

```
function UserProfile({ userId }) {  
  
  return (  
  
    <div>  
  
      <UserCard userId={userId} />  
  
      <UserPosts userId={userId} />  
  
    </div>  
  
  );  
  
}  
  
function UserCard({ userId }) {  
  
  const { data: user, loading } = useUser(userId);  
  
  if (loading) return <Spinner />;  
  
  return (  
  
    <div className="avatar">  
  
      <img src={user.avatar} alt={user.name} />  
  
      <h1>{user.name}</h1>  
  
      <p>{user.bio}</p>  
  
    </div>  
  
  );  
  
}
```

```
function UserPosts({ userId }) {  
  
  const { data: posts, loading } = useUserPosts(userId);  
  
  if (loading) return <Spinner />;  
  
  return (  
  
    <div className="posts">  
  
      {posts.map(post => <PostCard key={post.id} post={post} />)}  
  
    </div>  
  
  );  
  
}
```

Fark ortada. Her bileşen artık tek bir işi yapıyor, bağımsız test edilebiliyor, bir yerde bug olursa nereye bakacağınız belli. PostCard'ı başka bir sayfada da kullanmak isterseniz sadece taşıyorsunuz.

Peki bir bileşen ne zaman “çok büyük” olur? Kesin bir kural yok ama pratik bir işaret: JSX'e bakmak için sürekli aşağı yukarı kaydırıyorsanız, ya da bir bileşeni açıklamak için “şunu yapıyor ve şunu yapıyor ve şunu...” demek zorunda kalıyorsanız, bölme vakti gelmiştir.

2. Mantığı Custom Hook'lara Taşıyın

Yukarıdaki örnekte bir sihir vardı: veri çekme mantığını useUser ve useUserPosts gibi custom hook'lara taşıdım. Bu, React'te temiz kodun belki de en güçlü aracı.

Custom hook'lar, iş mantığını görsel mantıktan ayırmanızı sağlar. Bileşen “neyin gösterileceğine” odaklanırken, hook “verinin nasıl geleceğine” odaklanır.

```
function useUser(userId) {  
  
  const [data, setData] = useState(null);  
  
  const [loading, setLoading] = useState(true);  
  
  const [error, setError] = useState(null);
```

```
useEffect(() => {  
  
  let cancelled = false;  
  
  setLoading(true);
```

Get Furkan Yorulmaz's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

☐ Remember me for faster sign in

```
fetchUser(userId)  
  
  .then(user => { if (!cancelled) setData(user); })  
  
  .catch(err => { if (!cancelled) setError(err); })  
  
  .finally(() => { if (!cancelled) setLoading(false); });  
  
return () => { cancelled = true; };  
  
}, [userId]);  
  
return { data, loading, error };  
  
}
```

Bunun güzelliği çok yönlü. Bir kere yazdığınız bu hook'u istediğiniz kadar bileşende kullanabilirsiniz. Ayrıca loading, error ve cancelled gibi ince ayrıntıları tek bir yerde doğru yapıp geçersiniz — her bileşende tekrar tekrar aynı useEffect temizlik mantığını yazma derdi ortadan kalkar. Bileşeniniz ise artık tertemiz kalır; sadece hook'u çağırıp sonucu gösterir.

Genel kural: bir useEffect veya state mantığını birden fazla yerde tekrar ediyorsanız, o bir custom hook adayıdır.

3. İsimlendirme: Kodun Kendini Anlatması

“Bilgisayar bilimlerinde sadece iki zor problem vardır: cache geçersiz kılma ve isimlendirme.” — Phil Karlton

Şaka gibi ama gerçek. İyi isimlendirme, yorum satırına olan ihtiyacı büyük ölçüde ortadan kaldırır. Kod kendini anlatmalı.

data, item, handleClick, temp, flag gibi isimler size hiçbir şey söylemez. Bunun yerine niyeti anlatan isimler seçin:

// Kötü

```
const [d, setD] = useState([]);
```

```
const handleClick = () => { ... };
```

// İyi

```
const [pendingOrders, setPendingOrders] = useState([]);
```

```
const handleCheckoutSubmit = () => { ... };
```

React'e özel birkaç isimlendirme konvansiyonu var, bunlara uymak kodunuzu diğer geliştiriciler için anında okunur kılar:

- Bileşenler **PascalCase**: UserCard, CheckoutButton
- Hook'lar use ile başlar: useAuth, useCart
- Olay işleyicileri handle ile başlar: handleSubmit, handleInputChange
- Boolean değişkenler soru gibi okunur: isLoading, hasError, canEdit
- Prop olarak geçen fonksiyonlar on ile başlar: onClose, onSelect

Bu küçük tutarlılıklar birikince, kodunuz bir dile dönüşür ve o dili konuşan herkes onu akıcı okur.

4. Prop Drilling'den Kaçının

Bir state'i, ihtiyaç duyan bileşene ulaştırmak için onu beş katman boyunca props olarak aşağı taşımaya **prop drilling** denir. Ara bileşenler o veriyi kullanmasa bile

sadece “taşıyıcı” olarak onu geçirmek zorunda kalır. Bu hem yorucu hem de kırılğan.

```
// Prop drilling kabusu
```

```
<App theme={theme}>
```

```
<Layout theme={theme}>
```

```
<Sidebar theme={theme}>
```

```
<Menu theme={theme}>
```

```
<MenuItem theme={theme} /> { /* Nihayet burada kullanılıyor */ }
```

Çözümler duruma göre değişir. Basit, global durumlar için (tema, dil, oturum açmış kullanıcı) **Context API** birebir:

```
const ThemeContext = createContext();
```

```
function App() {
```

```
  const [theme, setTheme] = useState('dark');
```

```
  return (
```

```
    <ThemeContext.Provider value={{ theme, setTheme }}>
```

```
      <Layout />
```

```
    </ThemeContext.Provider>
```

```
  );
```

```
}
```

```
// İhtiyaç duyan bileşen, aradaki katmanları hiç rahatsız etmeden alır:
```

```
function MenuItem() {
```

```
  const { theme } = useContext(ThemeContext);
```

```
  return <li className={theme}>...</li>;
```

```
}
```

Daha karmaşık, çok sayıda güncelleme içeren uygulama durumu için ise **Redux Toolkit** veya **Zustand** gibi bir state yönetim kütüphanesi devreye girer. Ama burada bir uyarı: her şeyi global state'e koymayın. Sadece o bileşene ait, dışarıyla ilgisi olmayan bir durum (örneğin bir modal'ın açık/kapalı olması) yerel useState ile kalmalı. Global state'i çöp kutusu gibi kullanmak, prop drilling'den daha büyük bir kaosa yol açar.

5. Sihirli Değerler ve Tekrar Yerine Sabitler

Kodunuzun içine serpiştirilmiş açıklamasız sayılar ve string'ler — “magic values” — bir bakım kabusudur. Altı ay sonra if (status === 3) satırına baktığınızda 3'ün ne anlama geldiğini kim hatırlayacak?

```
// Kötü
```

```
if (user.role === 2 && order.status === 'PRC') { ... }
```

```
// İyi
```

```
const ROLES = { ADMIN: 1, EDITOR: 2, VIEWER: 3 };
```

```
const ORDER_STATUS = { PENDING: 'PENDING', PROCESSING: 'PRC', DONE: 'DONE' };
```

```
if (user.role === ROLES.EDITOR && order.status === ORDER_STATUS.PROCESSING) { ... }
```

Aynı şey tekrar eden mantık için de geçerli. Aynı doğrulama kontrolünü ya da aynı biçimlendirme fonksiyonunu üç yerde kopyalıyorsanız, onu tek bir utils fonksiyonuna alın. **DRY (Don't Repeat Yourself)** ilkesi, değişiklik gerektiğinde onu tek bir yerde yapmanızı sağlar — üç ayrı yeri güncellemeyi unutup bug üretmezsiz.

6. Klasör Yapısı da Temiz Kodun Parçası

Temiz kod sadece dosyanın içinde değil, dosyaların düzeninde de yaşar. Projeye yeni katılan biri, klasör yapısına bakarak “neyin nerede olduğunu” tahmin edebilmeli.

Bir dosyanın türüne göre gruplamak (tüm bileşenler components'te, tüm hook'lar hooks'ta) küçük projelerde iş görür. Ama proje büyüdükçe **özelliğe göre gruplama**

(feature-based) çok daha ölçeklenebilir olur:

src/

features/

auth/

components/

hooks/

authSlice.js

authApi.js

cart/

components/

hooks/

cartSlice.js

components/ // paylaşılan, jenerik bileşenler (Button, Modal)

hooks/ // paylaşılan hook'lar

utils/

Bu yapıda bir özelliğe dair her şey (bileşeni, mantığı, state'i, API çağrıları) bir arada durur. cart ile ilgili bir şeyi değiştireceğinizde, tüm dosya ağacını dolaşmak yerine tek bir klasöre gidirsiniz.

Ama Dikkat: Aşırıya Kaçmayın

Temiz kod önemli, ama her şeyde olduğu gibi burada da denge var. Yeni öğrenenlerin sık düştüğü bir tuzak, **erken soyutlama (premature abstraction)**. Henüz iki kez tekrar etmemiş bir kodu, “belki ileride gerekir” diye erkenden bir hook’a ya da jenerik bir bileşene çevirmek, çoğu zaman gereksiz karmaşa yaratır.

Pratik bir kural olan “**AHA — Avoid Hasty Abstractions**” (Aceleci Soyutlamalardan Kaçın) burada işe yarar: yanlış bir soyutlama, biraz kod tekrarından çok daha

maliyetlidir. Kodu önce çalışır ve okunur yazın; tekrar eden desen üçüncü kez ortaya çıktığında soyutlamayı yapın. O zaman zaten deseni gerçekten anlamış olursunuz.

Aynı şekilde, her bir satırlık fonksiyonu ayrı dosyaya bölmek de temizlik değil, dağınıklıktır. Amaç bileşen sayısını maksimize etmek değil; **bir insanın kodu rahatça takip edebilmesi**.

React size bir çerçeve değil, bir kütüphane veriyor — yani mimariyle ilgili kararların çoğunu size bırakıyor. Bu özgürlük, temiz kod disiplini bir zorunluluk haline getiriyor. Kuralları React sizin için koymuyor; onları siz koymak zorundasınız.

Temiz kod yazmak, ne yapmadan önce mükemmel olmayı beklemek ne de her satırı cilalamaktır. Sadece küçük, tutarlı alışkanlıklardır: bileşenleri küçük tut, mantığı hook'lara taşı, isimlerine özen göster, tekrardan kaçın ve niyetini kodun kendisiyle anlat. Bu alışkanlıklar birikince, projeniz büyüdükçe boğulan değil, nefes alan bir kod tabanına sahip olursunuz.

Sonuçta yazdığınız kodun en önemli okuyucusu, altı ay sonra o dosyayı tekrar açacak olan sizsiniz. Ona iyi davranın. 😊

Mutlu ve temiz kodlamalar.

[Follow](#)

Written by Furkan Yorulmaz

0 followers · 1 following

