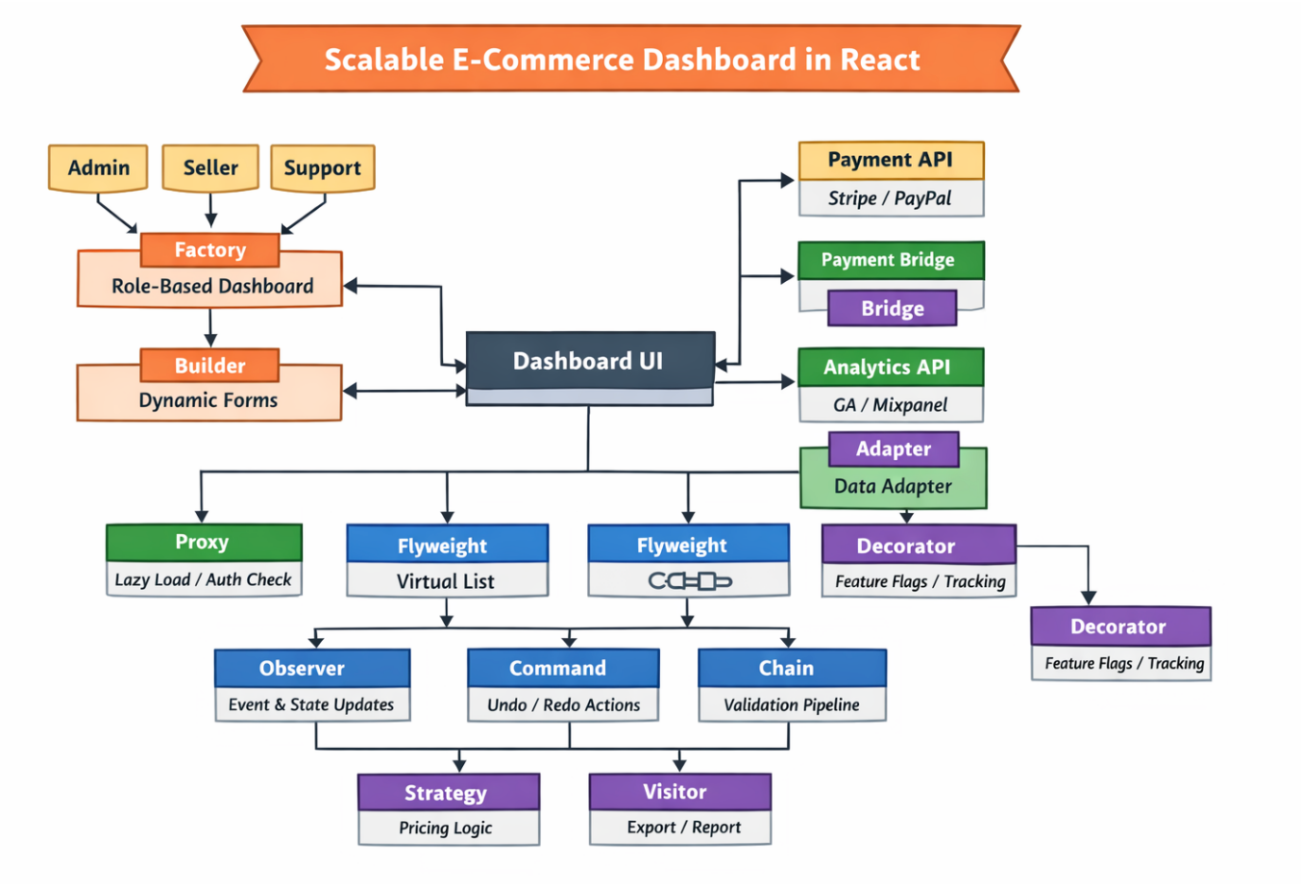
Build a scalable e-commerce Admin dashboard (like Amazon / Flipkart / Shopify) using React

The dashboard must support:

- Multiple user roles
- Multiple payment providers
- Multiple analytics providers
- Large product lists
- Feature toggles
- Undo/redo actions
- Future extensibility without rewriting components



Pattern Mapping inside the SAME use case

Creational Problems

Factory

Problem:
Render different dashboards based on role (`Admin`, `Seller`, `Support`) without `if/else` explosion.

Factory creates the correct dashboard component.

```
// DashboardFactory.tsx

export function DashboardFactory(role: string) {
  switch (role) {
    case "admin":
      return <AdminDashboard />
    case "seller":
      return <SellerDashboard />
    default:
      return <SupportDashboard />
  }
}
```

Builder

Problem:
Checkout forms and filters vary per country, tax rules, and feature flags.

Builder constructs forms step-by-step.

```
class FormBuilder {
  private fields: string[] = []

  addEmail() {
    this.fields.push("email")
    return this
  }

  addPassword() {
    this.fields.push("password")
    return this
  }

  build() {
```

```
    return this.fields
  }
}
const loginForm = new FormBuilder()
  .addEmail()
  .addPassword()
  .build()
```

Prototype

Problem:
Product cards are mostly identical; only price, badge, or CTA changes.

Clone base product card configs.

```
const baseProductCard = {
  showPrice: true,
  showBadge: false
}

const saleProductCard = {
  ...baseProductCard,
  showBadge: true
}
```

Structural Problems

Adapter

Problem:
Different backend services return product, order, and user data in incompatible formats.

Adapter normalizes data before UI consumes it.

```
// Adapter

export class ProductAdapter {
  static adapt(apiProduct: any) {
    return {
      id: apiProduct.product_id,
      name: apiProduct.product_name,
      price: apiProduct.cost
    }
```

```
  }
}
const product = ProductAdapter.adapt(apiResponse)
```

Bridge

Problem:
UI should work with **Stripe, Razorpay, PayPal** without rewriting components.

Payment UI abstraction bridged to provider implementations.

```
// Abstraction

interface PaymentProvider {
  pay(amount: number): void
}
// Implementations
class StripePayment implements PaymentProvider {
  pay(amount: number) {
    console.log("Stripe:", amount)
  }
}

class RazorpayPayment implements PaymentProvider {
  pay(amount: number) {
    console.log("Razorpay:", amount)
  }
}
function Checkout({ provider }: { provider: PaymentProvider }) {
  return <button onClick={() => provider.pay(100)}>Pay</button>
}
```

Proxy

Problem:
Product images, charts, and feeds should:

- Lazy load
- Check permissions
- Show skeletons

Proxy controls access before rendering.

```
function ProductProxy({ isAllowed }: any) {
  if (!isAllowed) return <div>Not Authorized</div>
  return <HeavyProductList />
}
```

Decorator

Problem:
Add analytics, feature flags, and error tracking to components without modifying them.

Decorators enhance behavior dynamically.

```
const withAnalytics = (Component: any) => (props: any) => {

  console.log("Tracked")

  return <Component {...props} />

}
```

Flyweight

Problem:
Thousands of products render in grids causing memory and performance issues.

Shared product metadata + virtual rendering.

```
const ProductMetaStore: any = {}

export function getProductMeta(type: string) {
  if (!ProductMetaStore[type]) {
    ProductMetaStore[type] = { icon: "📦", color: "blue" }
  }
  return ProductMetaStore[type]
}
```

Behavioral Problems

Observer

Problem:
Multiple widgets must update when:

- Order status changes
- Inventory updates
- WebSocket events arrive

Observer notifies subscribers.

```
class EventBus {
  private listeners: Function[] = []

  subscribe(fn: Function) {
    this.listeners.push(fn)
  }

  publish(data: any) {
    this.listeners.forEach(fn => fn(data))
  }
}
eventBus.subscribe(updateInventory)
```

Command

Problem:
Admin actions need:

- Undo / redo
- Bulk operations
- Action logs

Each UI action becomes a command.

```
interface Command {

  execute(): void

  undo(): void

}
```

Chain of Responsibility

Problem:
Checkout validation must pass through:

- Auth check
- Inventory check
- Coupon validation
- Fraud detection

Each handler decides to process or pass.

```
class Handler {
  next?: Handler
  setNext(handler: Handler) {
```

```
    this.next = handler
    return handler
  }
  handle(data: any) {
    this.next?.handle(data)
  }
}
authHandler.setNext(stockHandler).setNext(couponHandler)
```

Strategy

Problem:
Pricing, discount, and tax rules vary by country and campaign.

Select pricing strategy at runtime.

```
interface PricingStrategy {
  calculate(price: number): number
}
class IndiaPricing implements PricingStrategy {
  calculate(price: number) {
    return price * 1.18
  }
}
function getFinalPrice(strategy: PricingStrategy, price: number) {
  return strategy.calculate(price)
}
```

Visitor

Problem:
Need to add new operations later:

- Analytics tracking
- Accessibility scanning
- Export to CSV

Without changing existing components.

Visitors operate on component structure.

```
interface Visitor {
  visitProduct(product: any): void
}
class AnalyticsVisitor implements Visitor {
```

```
  visitProduct(product: any) {
    console.log("Track", product.id)
  }
}
product.accept(new AnalyticsVisitor())
```