KALEB GARNER
REACT SUMMIT 2026

KALEBGARNER.DEV

Mess to Modern

REFACTORING A REACT NIGHTMARE

A five-step playbook for turning an inherited 900-line React nightmare into a codebase you'd actually enjoy opening.

00 STARTING POINT
REACT SUMMIT 2026

This is our starting point.

- 900+ lines of React
- 5 useEffect calls
- 16 useState calls
- Fetched, transformed, validated, and rendered
- Nightmare to work on

Where do we go from here?

```
export default function DashboardPage() {
const [orders, setOrders] = useState([]);
const [filteredOrders, setFilteredOrders] = useState([]);
const [products, setProducts] = useState([]);
const [customers, setCustomers] = useState([]);
const [revenue, setRevenue] = useState(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);
const [statusFilter, setStatusFilter] = useState('all');
const [dateRange, setDateRange] = useState('30d');
const [isModalOpen, setIsModalOpen] = useState(false);
const [selectedOrder, setSelectedOrder] = useState(null);
const [currentUser, setCurrentUser] = useState(null);
const [sortField, setSortField] = useState('date');
const [sortDir, setSortDir] = useState('desc');
const [searchQuery, setSearchQuery] = useState('');
const [isSaving, setIsSaving] = useState(false);
useEffect(() => {
  setLoading(true);
  fetch('/api/orders')
  .then(res => res.json())
  .then(data => { setOrders(data); setLoading(false); })
  .catch(err => { setError(err); setLoading(false); });
}, []);
useEffect(() => {
  fetch('/api/products').then(res => res.json()).then(setProducts);
}, []);
useEffect(() => {
  fetch('/api/customers').then(res => res.json()).then(setCustomers);
}, []);
useEffect(() => {
  fetch('/api/users/me').then(res => res.json()).then(setCurrentUser);
}, []);
useEffect(() => {
  let result = orders;
  if (statusFilter !== 'all') {
  result = result.filter(o => o.status === statusFilter);
}
  if (searchQuery) {
    result = result.filter(o =>
      o.customerName.toLowerCase().includes(searchQuery.toLowerCase())
    );
}
result = [...result].sort((a, b) =>
sortDir === 'desc'
? b[sortField] > a[sortField] ? 1 : -1
: a[sortField] > b[sortField] ? 1 : -1
);
setFilteredOrders(result);
}, [orders, statusFilter, searchQuery, sortField, sortDir]);
}
```

```
const getRevenueByRange = (range) => {
const cutoff = new Date();
if (range === '7d') cutoff.setDate(cutoff.getDate() - 7);
if (range === '30d') cutoff.setDate(cutoff.getDate() - 30);
if (range === '90d') cutoff.setDate(cutoff.getDate() - 90);
return orders
.filter(o => new Date(o.date) >= cutoff)
.reduce((sum, o) => sum + o.total, 0);
};
const handleUpdateOrderStatus = async (id, status) => {
setIsSaving(true);
await fetch(`/api/orders/${id}`, {
method: 'PATCH',
body: JSON.stringify({ status }),
});

const data = await fetch('/api/orders').then(r => r.json());
setOrders(data);
setIsSaving(false);
};
return (
<div>
  {/* Header with user info */}
  {/* KPI cards (revenue, orders, customers) */}
  {/* Revenue chart */}
  {/* Orders table with sort, filter, search */}
  {/* Order detail modal */}
  {/* Low stock alerts */}
  {/* Error and loading states */}
</div>
);
```

What?

A flat **/components** folder with 40 files tells you nothing about what the app does. Before you move a single line of code, you need a map.

/components
DashboardPage.tsx
OrdersTable.tsx
OrderModal.tsx
RevenueChart.tsx
KPICard.tsx
KPICardNew.tsx
KPICardV2.tsx
FilterBar.tsx
Button.tsx
Button2.tsx
ButtonFinal.tsx
...

How?

Every feature gets its own folder. Components, hooks, types, all co-located. A single *index.ts* is the only thing the outside world can import from. The boundary is enforced by structure, not discipline.

```
/features
 /dashboard
  /components
    KPIGrid.tsx
    RevenueChart.tsx
    LowStockAlerts.tsx
  /hooks
    useDashboardMetrics.ts
index.ts

/shared
 /components
  Button.tsx
  Input.tsx
  Modal.tsx
```

Why?

Conway's Law says your system will mirror your org structure, usually by accident. Screaming Architecture says do it on purpose. Opening **/features/orders** should tell you exactly what that code does. **/components** tells you nothing.

```
// DashboardPage.tsx
export { OrdersTable } from './components/OrdersTable';
export { OrderModal } from './components/OrderModal';
export { useOrders } from './hooks/useOrders';
export type { Order, OrderStatus } from './types/orders.types';
```

What?

Revenue calculations, fetch logic, validation, all living inside the component. You cannot unit test any of it. You cannot reuse any of it. If the component breaks, everything breaks with it.

```
// DashboardPage.tsx
const getRevenueByRange = (range) => {
  const cutoff = new Date();
  if (range === '7d') cutoff.setDate(cutoff.getDate() - 7);
  if (range === '30d') cutoff.setDate(cutoff.getDate() - 30);
  if (range === '90d') cutoff.setDate(cutoff.getDate() - 90);
  return orders
    .filter(o => new Date(o.date) >= cutoff)
    .reduce((sum, o) => sum + o.total, 0);
};

useEffect(() => {
  setLoading(true);
  fetch('/api/orders')
    .then(res => res.json())
    .then(data => { setOrders(data); setLoading(false); })
    .catch(err => { setError(err); setLoading(false); });
}, []);
```

How?

useOrderFilters owns the filter and sort logic. *useOrders* owns the fetch. Each hook is independently testable with *renderHook*. The component calls one hook and gets everything it needs.

```
// hooks/useOrderFilters.ts
export function useOrderFilters(orders: Order[]) {
  const [statusFilter, setStatusFilter] = useState('all');
  const [searchQuery, setSearchQuery] = useState('');
  const [sortField, setSortField] = useState('date');
  const [sortDir, setSortDir] = useState<'asc' | 'desc'>('desc');

  const filtered = useMemo(() => {
    let result = orders;
    if (statusFilter !== 'all')
      result = result.filter(o => o.status === statusFilter);
    if (searchQuery)
      result = result.filter(o =>
o.customerName.toLowerCase().includes(searchQuery.toLowerCas
e())
      );
    return [...result].sort((a, b) =>
      sortDir === 'desc'
        ? b[sortField] > a[sortField] ? 1 : -1
        : a[sortField] > b[sortField] ? 1 : -1
    );
  }, [orders, statusFilter, searchQuery, sortField, sortDir]);

  return { filtered, statusFilter, setStatusFilter,
           searchQuery, setSearchQuery, sortField,
           setSortField, sortDir, setSortDir };
}
```

Why?

A component that fetches, transforms, validates, and renders is four things pretending to be one. When you extract into hooks, the component becomes a thin shell responsible for pure rendering.

```
// hooks/useDashboard.ts
function useDashboard() {
  const { orders, isLoading } = useOrders();
  const filters = useOrderFilters(orders);
  const { products } = useProducts();
  const { customers } = useCustomers();
  const { currentUser } = useCurrentUser();
  return { orders, filters, products, customers,
           currentUser, isLoading };
}

// DashboardPage.tsx
export default function DashboardPage() {
  const { orders, filters, products,
          customers, currentUser, isLoading }
    = useDashboard();

  if (isLoading) return <Spinner />;

  return (
    <div className="dashboard">
      <KPIGrid orders={orders} customers={customers} />
      <RevenueChart orders={orders} />
      <OrdersTable orders={filters.filtered} filters={filters} />
    </div>
  );
}
```

What?

filteredOrders is not state. It's a computation. But it's being stored, synced, and maintained like it's real data. This pattern creates bugs that are almost impossible to trace.

```
// DashboardPage.tsx
const [orders, setOrders] = useState([]);
const [filteredOrders, setFilteredOrders] = useState([]);
const [statusFilter, setStatusFilter] = useState('all');
const [searchQuery, setSearchQuery] = useState('');
const [sortField, setSortField] = useState('date');
const [sortDir, setSortDir] = useState('desc');

useEffect(() => {
  let result = orders;
  if (statusFilter !== 'all')
    result = result.filter(o => o.status === statusFilter);
  if (searchQuery)
    result = result.filter(o => o.customerName.includes(searchQuery));
  result = [...result].sort((a, b) => /* ... */);
  setFilteredOrders(result);
}, [orders, statusFilter, searchQuery, sortField, sortDir]);
```

How?

If a value can be calculated from existing state, calculate it with useMemo. When state has multiple related pieces, use a discriminated union so impossible states become impossible to represent. The type system enforces it.

```
const filteredOrders = useMemo(() => {
  let result = orders;
  if (statusFilter !== 'all')
    result = result.filter(o => o.status === statusFilter);
  if (searchQuery)
    result = result.filter(o =>
o.customerName.toLowerCase().includes(searchQuery.toLowerCase
())
    );
  return [...result].sort((a, b) =>
    sortDir === 'desc'
      ? b[sortField] > a[sortField] ? 1 : -1
      : a[sortField] > b[sortField] ? 1 : -1
  );
}, [orders, statusFilter, searchQuery, sortField, sortDir]);

type OrderFetchState =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success'; data: Order[] }
  | { status: 'error'; error: Error };
```

Why?

useReducer gives you a single place where state transitions are defined and readable. *useReducer* for local complex state. **Zustand** for state that lives outside the component tree. Don't reach for a store by default if state doesn't leave the tree, it doesn't need one.

```
type Action =
  | { type: 'FETCH_START' }
  | { type: 'FETCH_SUCCESS'; data: Order[] }
  | { type: 'FETCH_ERROR'; error: Error };

function ordersReducer(
  state: OrderFetchState,
  action: Action
): OrderFetchState {
  switch (action.type) {
    case 'FETCH_START':
      return { status: 'loading' };
    case 'FETCH_SUCCESS':
      return { status: 'success', data: action.data };
    case 'FETCH_ERROR':
      return { status: 'error', error: action.error };
    default:
      return state;
  }
}
```

What?

useEffect manages its own loading state, its own error state, its own cache. They drift. They go stale independently. One gets updated, the others don't. This is server state being managed on the client side.

```
// DashboardPage.tsx
useEffect(() => {
  fetch('/api/orders')...
}, []);

useEffect(() => {
  fetch('/api/products')...
}, []);

useEffect(() => {
  fetch('/api/customers')...
}, []);

useEffect(() => {
  fetch('/api/users/me')...
}, []);

const handleUpdateOrderStatus = async (id, status) => {
  await fetch(`/api/orders/${id}`, { method: 'PATCH', ... });
  const data = await fetch('/api/orders').then(r => r.json());
  setOrders(data);
};
```

Why?

orders.queries.ts is the single source of truth for every component that needs orders. *invalidateQueries* replaces the manual refetch entirely. Same key, one network request. No duplication. No drift.

```
// queries/orders.queries.ts
export const ordersQuery = {
  queryKey: ['orders'],
  queryFn: (): Promise<Order[]> =>
    fetch('/api/orders').then(r => r.json()),
};

export const updateOrderStatusMutation = () => ({
  mutationFn: ({ id, status }: { id: string; status:
OrderStatus }) =>
    fetch(`/api/orders/${id}`, {
      method: 'PATCH',
      body: JSON.stringify({ status }),
    }),
  onSuccess: () =>
    queryClient.invalidateQueries({ queryKey:
['orders'] }),
});

// DashboardPage.tsx
const { data: orders, isLoading } =
useQuery(ordersQuery);
const { mutate: updateStatus } =
  useMutation(updateOrderStatusMutation());
```

How?

Server state and client state are different things that need different tools. React Query doesn't just clean up the code, it changes the mental model from "fetch and store" to "cache and sync." When the API changes, you change one file.

/queries

orders.queries.ts
// DashboardPage imports from here
// OrdersSidebar imports from here
// SalesReport imports from here

products.queries.ts
// DashboardPage imports from here
// InventoryPage imports from here

customers.queries.ts
...

What?

Nobody builds a 12-prop component on purpose. You add **showDeleteButton** once. Then **showExportButton**. Then **isEditing**. Every new use case adds a new prop. The interface grows. The component becomes untestable.

```
<OrderModal
  isOpen={isModalOpen}
  order={selectedOrder}
  onClose={() => setIsModalOpen(false)}
  onUpdateStatus={handleUpdateOrderStatus}
  isSaving={isSaving}
  customers={customers}
  products={products}
  currentUser={currentUser}
  isEditing={!!selectedOrder}
  showDeleteButton={
    !!selectedOrder && currentUser?.role === 'admin'
  }
  showExportButton={currentUser?.role !== 'viewer'}
  onExport={handleExport}
/>
```

How?

The compound component pattern lets each piece do exactly one thing. *OrderModal.Header* and *OrderModal.Details* assembled at the call site. The caller controls the layout. The pieces don't need to know about each other.

```
// OrderModal.tsx
function OrderModal({ children, onClose }) {
  return <Modal onClose={onClose}>{children}</Modal>;
}

OrderModal.Header = ({ order }) => (
  <ModalHeader
    title={order ? `Order #${order.id}` : 'New Order'}
    badge={order?.status}
  />
);

OrderModal.Details = ({ order, customers }) => (
  <OrderDetails order={order} customers={customers} />
);

// DashboardPage.tsx
<OrderModal onClose={closeModal}>
  <OrderModal.Header order={selectedOrder} />
  <OrderModal.Details order={selectedOrder} customers=
{customers} />
</OrderModal>
```

Why?

For anything requiring accessibility Radix has already solved keyboard navigation, focus trapping, and ARIA. You own the styling. **Radix** owns the behavior.

```
import * as DropdownMenu from '@radix-ui/react-dropdown-menu';

const ORDER_STATUSES = ['pending', 'processing', 'shipped', 'delivered'];

<DropdownMenu.Root>
  <DropdownMenu.Trigger asChild>
    <Button variant="ghost">Status: {order.status}</Button>
  </DropdownMenu.Trigger>
  <DropdownMenu.Content>
    {ORDER_STATUSES.map(status => (
      <DropdownMenu.Item
        key={status}
        onSelect={() => updateStatus({ id: order.id, status })}
      >
        {status}
      </DropdownMenu.Item>
    ))}
  </DropdownMenu.Content>
</DropdownMenu.Root>
```

06 ENDING POINT
REACT SUMMIT 2026

This is where we started.

- 900+ lines of React
- 5 ***useEffect*** calls
- 16 *useState* calls
- Fetched, transformed, validated, and rendered
- Nightmare to work on

```
export default function DashboardPage() {
const [orders, setOrders] = useState([]);
const [filteredOrders, setFilteredOrders] = useState([]);
const [products, setProducts] = useState([]);
const [customers, setCustomers] = useState([]);
const [revenue, setRevenue] = useState(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);
const [statusFilter, setStatusFilter] = useState('all');
const [dateRange, setDateRange] = useState('30d');
const [isModalOpen, setIsModalOpen] = useState(false);
const [selectedOrder, setSelectedOrder] = useState(null);
const [currentUser, setCurrentUser] = useState(null);
const [sortField, setSortField] = useState('date');
const [sortDir, setSortDir] = useState('desc');
const [searchQuery, setSearchQuery] = useState('');
const [isSaving, setIsSaving] = useState(false);
useEffect(() => {
  setLoading(true);
  fetch('/api/orders')
  .then(res => res.json())
  .then(data => { setOrders(data); setLoading(false); })
  .catch(err => { setError(err); setLoading(false); });
}, []);
useEffect(() => {
  fetch('/api/products').then(res => res.json()).then(setProducts);
}, []);
useEffect(() => {
  fetch('/api/customers').then(res => res.json()).then(setCustomers);
}, []);
useEffect(() => {
  fetch('/api/users/me').then(res => res.json()).then(setCurrentUser);
}, []);
useEffect(() => {
  let result = orders;
  if (statusFilter !== 'all') {
  result = result.filter(o => o.status === statusFilter);
}
  if (searchQuery) {
    result = result.filter(o =>
      o.customerName.toLowerCase().includes(searchQuery.toLowerCase())
    );
}
result = [...result].sort((a, b) =>
sortDir === 'desc'
? b[sortField] > a[sortField] ? 1 : -1
: a[sortField] > b[sortField] ? 1 : -1
);
setFilteredOrders(result);
}, [orders, statusFilter, searchQuery, sortField, sortDir]);
}
```

```
const getRevenueByRange = (range) => {
const cutoff = new Date();
if (range === '7d') cutoff.setDate(cutoff.getDate() - 7);
if (range === '30d') cutoff.setDate(cutoff.getDate() - 30);
if (range === '90d') cutoff.setDate(cutoff.getDate() - 90);
return orders
.filter(o => new Date(o.date) >= cutoff)
.reduce((sum, o) => sum + o.total, 0);
};
const handleUpdateOrderStatus = async (id, status) => {
setIsSaving(true);
await fetch(`/api/orders/${id}`, {
method: 'PATCH',
body: JSON.stringify({ status }),
});

const data = await fetch('/api/orders').then(r => r.json());
setOrders(data);
setIsSaving(false);
};
return (
<div>
  {/* Header with user info */}
  {/* KPI cards (revenue, orders, customers) */}
  {/* Revenue chart */}
  {/* Orders table with sort, filter, search */}
  {/* Order detail modal */}
  {/* Low stock alerts */}
  {/* Error and loading states */}
</div>
);
```

06 ENDING POINT
REACT SUMMIT 2026

This is where we ended.

- 35 lines of React
- Pure rendering, logic stays where it belongs
- Organized and structured
- Fun to work in again

```tsx
// DashboardPage.tsx
export default function DashboardPage() {
  const {
    orders,
    filters,
    products,
    customers,
    currentUser,
    isLoading,
  } = useDashboard();

  if (isLoading) return <Spinner />;

  return (
    <div className="dashboard">
      <KPIGrid orders={orders} customers={customers} />
      <RevenueChart orders={orders} />
      <OrdersTable
        orders={filters.filtered}
        filters={filters}
      />
      <LowStockAlerts products={products} />
    </div>
  );
}
```

Thank you

Thank you for taking the time to listen!