React with TypeScript all the way to production

You use React, TypeScript, and React Router to build an app from small components, then ship it to production with tests and good accessibility.

Mike Bild

Exercises: github.com/MikeBild/introduction-react

components, JSX, composition, styling

Thinking of components as functions · JSX is JavaScript, props are the contract · Pass content in instead of collecting flags · Styles belong with the component

Why components?

- UIs grow faster than any file
- Reuse replaces copies
- Small units are easier to understand and test

UI = f(props)

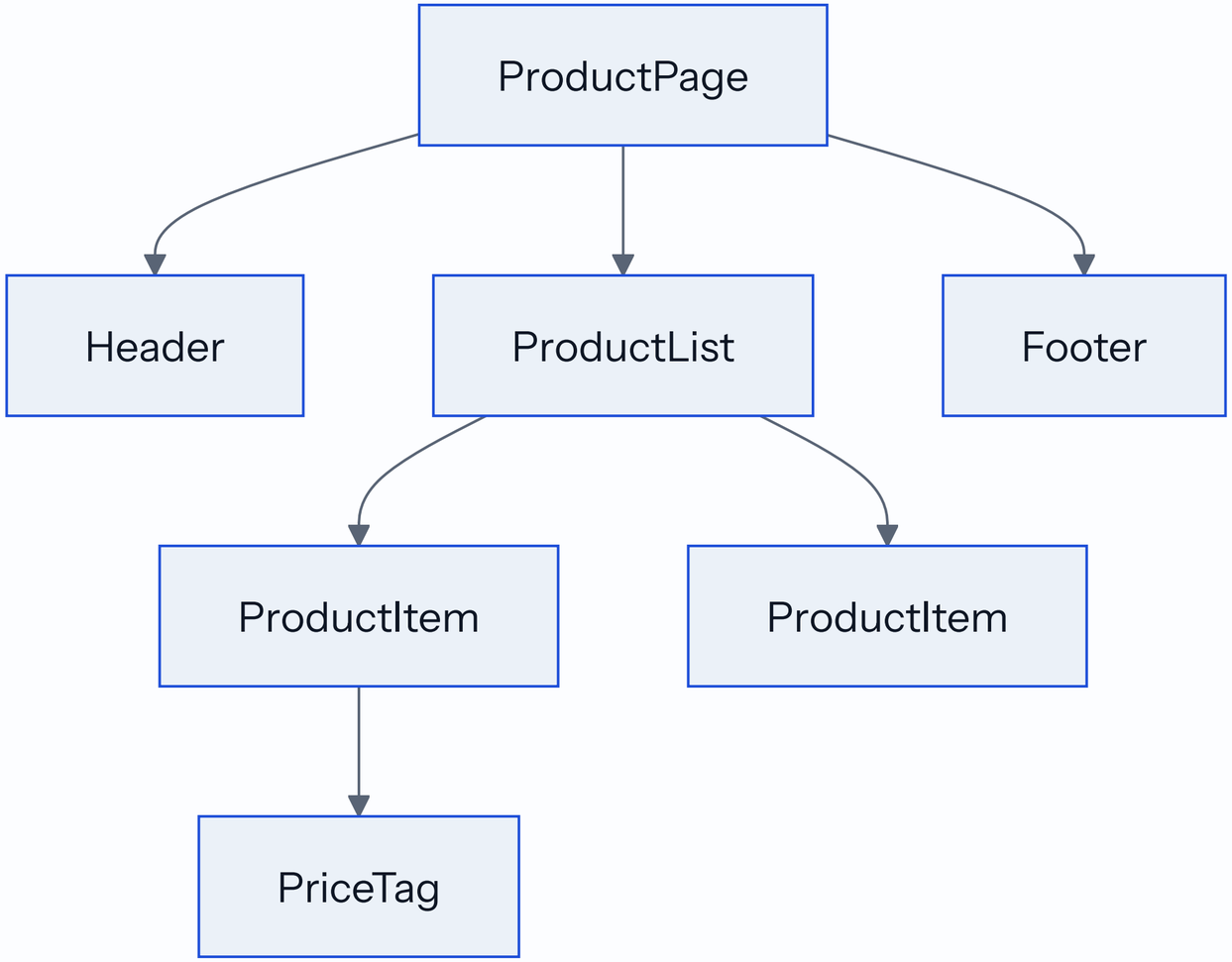
- Same props, same output
- Capitalized name, typed props
- One return value: JSX or null

```
type ProductCardProps = { name: string; priceCents: number; stock: number }

export const ProductCard = ({ name, priceCents, stock }: ProductCardProps) => (
  <article>
    <h2>{name}</h2>
    <PriceTag cents={priceCents} />
    {stock === 0 && <span> Ausverkauft</span>}
  </article>
)

// Nutzung
const hammerCard = <ProductCard name="Hammer" priceCents={1290} stock={0} />
```

First the component tree, then the code



Rendering stays pure

- Do not change anything during rendering, only describe
- Never mutate props, create new values
- Read Class Components only in legacy code

```
// Wrong: changes the props during rendering
const SortedNames = ({ items }: { items: string[] }) => {
  items.sort()
  return <ul>{items.map((item) => <li key={item}>{item}</li>)}</ul>
}

// Correct: new, sorted copy
const SortedNamesFixed = ({ items }: { items: readonly string[] }) => (
  <ul>{items.toSorted().map((item) => <li key={item}>{item}</li>)}</ul>
)
```

A component is a function

- Write components as functions from props to UI
- Sketch the component tree first, then code
- Keep rendering pure and mutate nothing

JSX is JavaScript

- JSX is compiled to function calls
- Vite translates JSX during the build
- Every tag is an object, not an HTML string

```
// What you write
const saveButton = <Button variant="primary">Speichern</Button>

// What it becomes (simplified)
const saveButtonCompiled = jsx(Button, {
  variant: 'primary',
  children: 'Speichern',
})
```

Curly braces are for expressions

- Values, calls, and ternaries are allowed
- if and for belong before the return
- Attributes are className and htmlFor; style is an object

```
export const CartRow = ({ item, productName }: { item: OrderItem; productName: string }) => {
  const totalCents = item.quantity * item.priceCents
  const quantityId = `quantity-${item.productId}`
  return (
    <article className="cart-row" style={{ opacity: totalCents > 0 ? 1 : 0.5 }}>
      <span>{productName}</span>
      <PriceTag cents={totalCents} />
      <label htmlFor={quantityId}>Menge für {productName}</label>
      <input id={quantityId} type="number" defaultValue={item.quantity} />
    </article>
  )
}
```

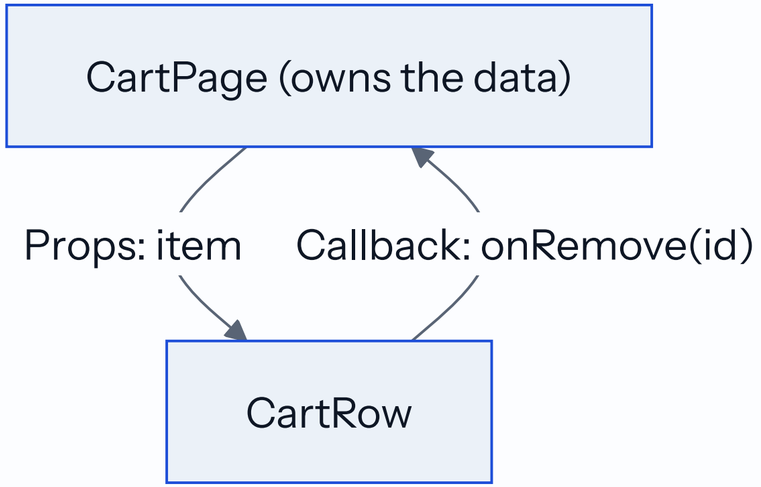
Props are a typed contract

- type describes what the component needs
- Union Types replace free strings
- Defaults go in the destructuring

```
type ButtonProps = {
  label: string
  variant?: 'primary' | 'secondary'
  isDisabled?: boolean
  onPress: () => void
}

export const Button = ({ label, variant = 'primary', isDisabled = false, onPress }: ButtonProps) => (
  <button className={variant} disabled={isDisabled} onClick={onPress}>
    {label}
  </button>
)
```

Data flows downward



Three typical beginner mistakes

- Props are read-only, even without readonly
- A return needs one root element or fragment
- any turns TypeScript off, unknown does not

Twenty props are a warning sign

- Each new variant adds a new flag
- The component suddenly knows every Use Case
- Tests and types grow with it

```
const orderCard = (
  <Card
    title="Bestellung"
    showIcon
    iconName="box"
    showFooter
    footerText="Details"
    footerLink="/orders/42"
    isHighlighted
  />
)
```

children accepts any content

- Everything between the tags ends up in children
- Type ReactNode allows text, elements, and lists
- The wrapper does not need to know the content

```tsx
import type { ReactNode } from 'react'

export const Panel = ({ children }: { children: ReactNode }) => (
  <section className="rounded border p-4">{children}</section>
)

// Nutzung
const orderPanel = (
  <Panel>
    <h2>Bestellung 42</h2>
    <OrderItems orderId="42" />
  </Panel>
)
```

Named slots replace flags

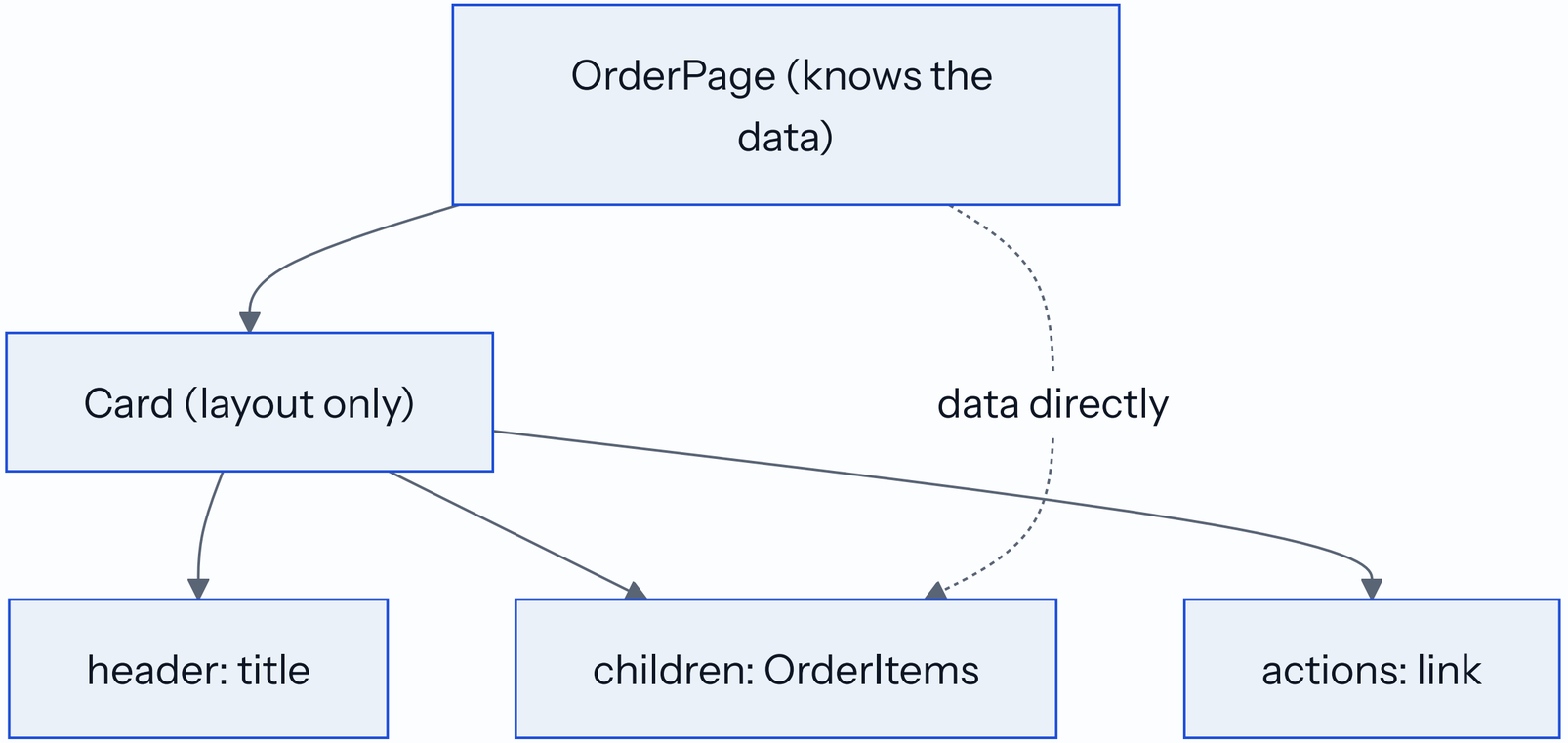
- Each area becomes a ReactNode prop
- Missing slots render nothing
- Domain logic stays with the caller

```
type CardProps = { header?: ReactNode; actions?: ReactNode; children: ReactNode }

export const Card = ({ header, actions, children }: CardProps) => (
  <section className="card">
    {header && <header>{header}</header>}
    <div>{children}</div>
    {actions && <footer>{actions}</footer>}
  </section>
)

const orderCard = (
  <Card header={<h2>Bestellung</h2>} actions={<a href="/orders/42">Details</a>}>
    <OrderItems orderId="42" />
  </Card>
)
```

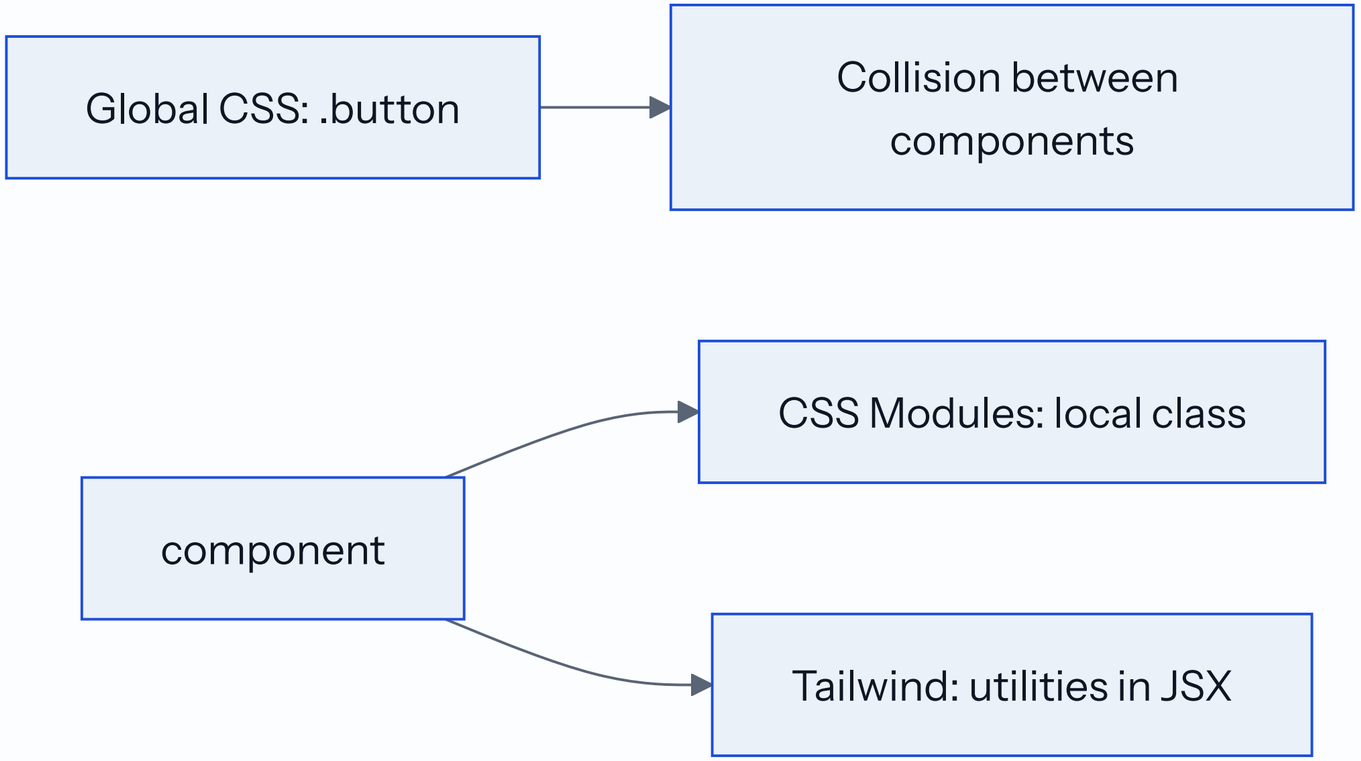
Data flows past the wrapper



Not every detail needs a slot

- Use fixed props for parts that stay the same
- Read Higher-Order Components only in legacy code
- Prefer children, slots, and Custom Hooks

Where should the CSS go?



CSS Modules make classes local

- The file ends in .module.css
- Class names become local and unique
- You keep writing normal CSS

```
// Button.module.css
// .button { padding: 0.5rem 1rem; border-radius: 0.375rem; }
// .primary { background: #1d4ed8; color: white; }

import styles from './Button.module.css'

export const Button = ({ label }: { label: string }) => (
  <button className={`${styles.button} ${styles.primary}`}>{label}</button>
)
```

Tailwind uses utility classes

- Utility classes are directly in JSX
- Spacing and colors come from a theme
- Only used classes end up in the build

```
export const Alert = ({ message }: { message: string }) => (
  <div role="alert" className="flex items-center gap-2 rounded-md bg-red-50 p-4 text-sm text-red-900 md:text-base">
    <span aria-hidden="true">!</span>
    {message}
  </div>
)
```

Variants belong in a map

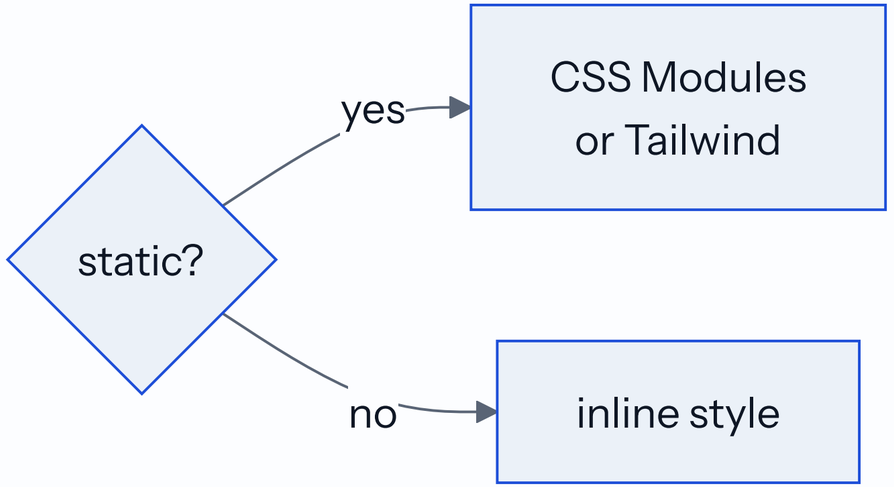
- Complete class names in a map
- Variant as a union type from the keys
- No composed class names

```
const variantClasses = {
  primary: 'bg-blue-700 text-white hover:bg-blue-800',
  secondary: 'bg-white text-slate-900 border border-slate-300',
  danger: 'bg-red-700 text-white hover:bg-red-800',
} as const

type ButtonProps = { label: string; variant?: keyof typeof variantClasses }

export const Button = ({ label, variant = 'primary' }: ButtonProps) => (
  <button className={`rounded-md px-4 py-2 ${variantClasses[variant]}`}>{label}</button>
)
```

One approach per project



state, events, lists, effects

State is a snapshot per render · Who owns the value in the input? · Keys are identity, not position · Do you really need this effect?

Why state?

- Local variables start over on every render
- Changes to variables do not trigger a render
- useState solves both

```
// Doesn't work: clickCount is 0 again on every render
const Counter = () => {
  let clickCount = 0
  return <button onClick={() => clickCount++}>{clickCount}</button>
}
```

State is a snapshot

- Each render sees its own values
- setState schedules a new render
- An updater function gets the latest value

```
const [count, setCount] = useState(0)

// Shows 1 afterward: both calls see count = 0
const incrementTwiceWrong = () => { setCount(count + 1); setCount(count + 1) }

// Shows 2 afterward: each updater gets the latest value
const incrementTwice = () => {
  setCount((current) => current + 1)
  setCount((current) => current + 1)
}
```

From click to new UI



New objects instead of mutation

- Updates create new objects and arrays
- Spread, map, and filter are your tools
- Same reference tells React: nothing changed

```
const [cart, setCart] = useState<readonly OrderItem[]>([])

// Wrong: same reference, no render
// cart.push(item); setCart(cart)

const addToCart = (product: Product) =>
  setCart((current) => [
    ...current,
    { productId: product.id, quantity: 1, priceCents: product.priceCents },
  ])

const removeFromCart = (productId: string) =>
  setCart((current) => current.filter((item) => item.productId !== productId))
```

State is tied to the position in the tree

- React remembers state per position in the tree
- The same component in the same place keeps it
- A component inside a component loses it

```
// Wrong: each render creates a new component type
export const SearchPage = () => {
  const [query, setQuery] = useState('')
  const SearchInput = () => <input value={query} onChange={(event) => setQuery(event.target.value)} />
  return <SearchInput />
}

// Correct: define at module level, values as props
const SearchField = ({ value, onChange }: { value: string; onChange: (next: string) => void }) => (
  <input value={value} onChange={(event) => onChange(event.target.value)} />
)
```

Where does state live?

- As local as possible
- As high as necessary
- Do not store derived values; compute them

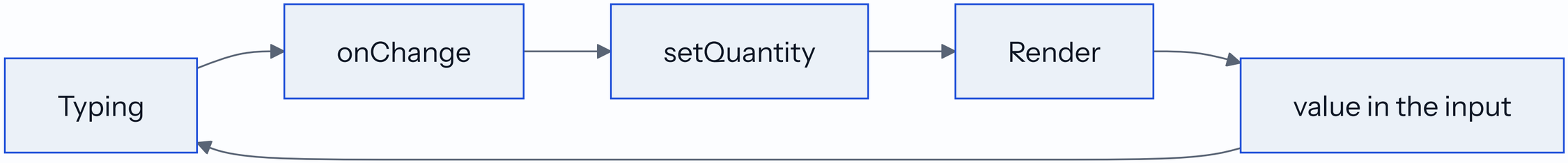
Pass event handlers, do not call them

- camelCase: onClick, onChange, onSubmit
- Reference instead of call: onClick={save}
- Arguments through an arrow function

```
type CartRowProps = { item: OrderItem; productName: string; onRemove: (productId: string) => void }

export const CartRow = ({ item, productName, onRemove }: CartRowProps) => (
  <li>
    {productName} × {item.quantity}
    <button type="button" onClick={() => onRemove(item.productId)}>
      Entfernen
    </button>
  </li>
)
```

In a controlled input, state decides



A form with typed events

- Type events with ChangeEvent and SubmitEvent
- Prevent page reload with preventDefault
- Disable the button instead of throwing an error

```
type Props = { stock: number; onAdd: (quantity: number) => void }

export const AddToCartForm = ({ stock, onAdd }: Props) => {
  const [quantity, setQuantity] = useState('1')
  const amount = Number(quantity)
  const handleSubmit = (event: SubmitEvent<HTMLFormElement>) => {
    event.preventDefault()
    onAdd(amount)
  }
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Menge <input value={quantity} onChange={(event) => setQuantity(event.target.value)} />
      </label>
      <button disabled={!(amount >= 1 && amount <= stock)}>In den Warenkorb</button>
    </form>
  )
}
```

Controlled or uncontrolled?

- Controlled: live validation, dependent fields
- Uncontrolled: defaultValue and FormData on submit
- Switch: never between the two

```
import type { SubmitEvent } from 'react'

const handleSubmit = (event: SubmitEvent<HTMLFormElement>) => {
  event.preventDefault()
  const formData = new FormData(event.currentTarget)
  const quantity = Number(formData.get('quantity'))
  onAdd(quantity)
}

// <input name="quantity" type="number" defaultValue="1" />
```

A clear pattern for events and forms

- Pass event handlers, do not call them
- Build controlled inputs with value plus onChange
- Decide controlled or uncontrolled per field

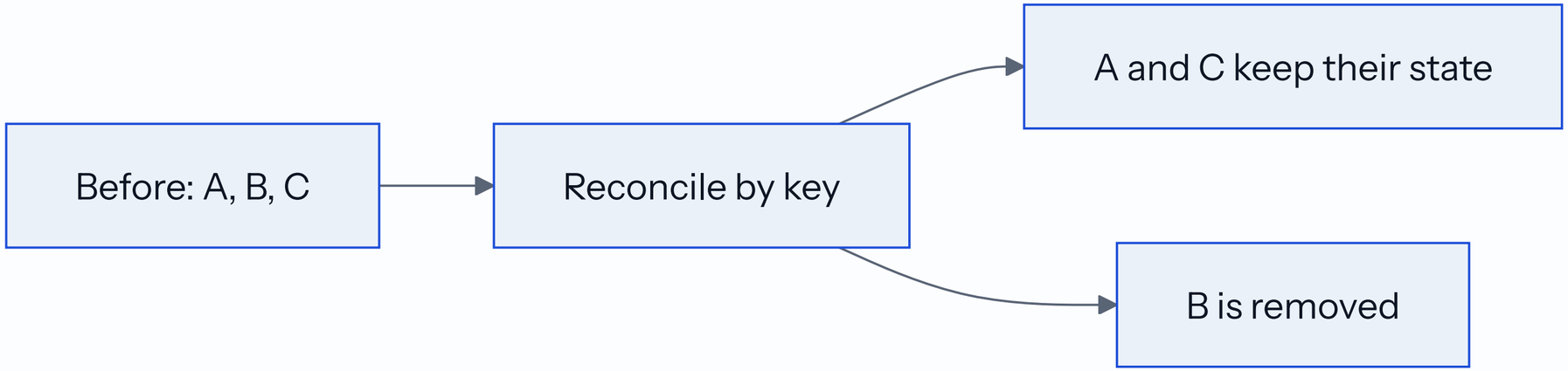
Arrays become elements

- Arrays become elements with map
- Every element needs a key
- Filter and sort before the return

```
export const ProductList = ({ products }: { products: readonly Product[] }) => {
  const availableProducts = products.filter((product) => product.stock > 0)
  return (
    <ul>
      {availableProducts.map((product) => (
        <li key={product.id}>{product.name}</li>
      ))}
    </ul>
  )
}
```

Keys are identity

- Unique among siblings
- Stable across all renders
- Index only for static lists



Conditions are normal expressions

- Ternary for either/or
- && only with real booleans
- null for empty output

```
export const CartSummary = ({ itemCount }: { itemCount: number }) => (
  <p>
    {itemCount > 0 ? `${itemCount} Artikel` : 'Warenkorb ist leer'}
    {/* Wrong: shows a 0 when 0 */}
    {itemCount && <span> - zur Kasse</span>}
    {/* Correct: real Boolean */}
    {itemCount > 0 && <span> - zur Kasse</span>}
  </p>
)
```

One status instead of many booleans

- One status field replaces isLoading and hasError
- TypeScript checks that every case is handled
- Early return keeps the component flat

```
type OrdersState =
  | { status: 'loading' }
  | { status: 'error'; message: string }
  | { status: 'success'; orders: readonly Order[] }

export const Orders = ({ state }: { state: OrdersState }) => {
  if (state.status === 'loading') return <p>Lade …</p>
  if (state.status === 'error') return <p role="alert">{state.message}</p>
  if (state.orders.length === 0) return <p>Noch keine Bestellungen</p>
  return <ul>{state.orders.map((order) => <li key={order.id}>Bestellung {order.id}: {order.status}</li>)}</ul>
}
```

A new key resets state

- Keys also work outside lists
- New key means new component with fresh state
- No effect that resets state by hand

```
// Without a key, the form keeps the inputs from the previous customer
export const CustomerDetails = ({ customerId }: { customerId: string }) => (
  <section>
    <h2>Kunde {customerId}</h2>
    <CustomerForm key={customerId} customerId={customerId} />
  </section>
)
```

Key from the data, status as a union type

- Key from the data, never random
- Conditions as expressions or early returns
- States as a union type instead of booleans

Do you need the effect?

- Calculate derived values during rendering
- Respond to events in the event handler
- Use effects only for external systems

```
// Unnecessary: additional state and one more render
const [fullName, setFullName] = useState('')
useEffect(() => setFullName(`${firstName} ${lastName}`), [firstName, lastName])

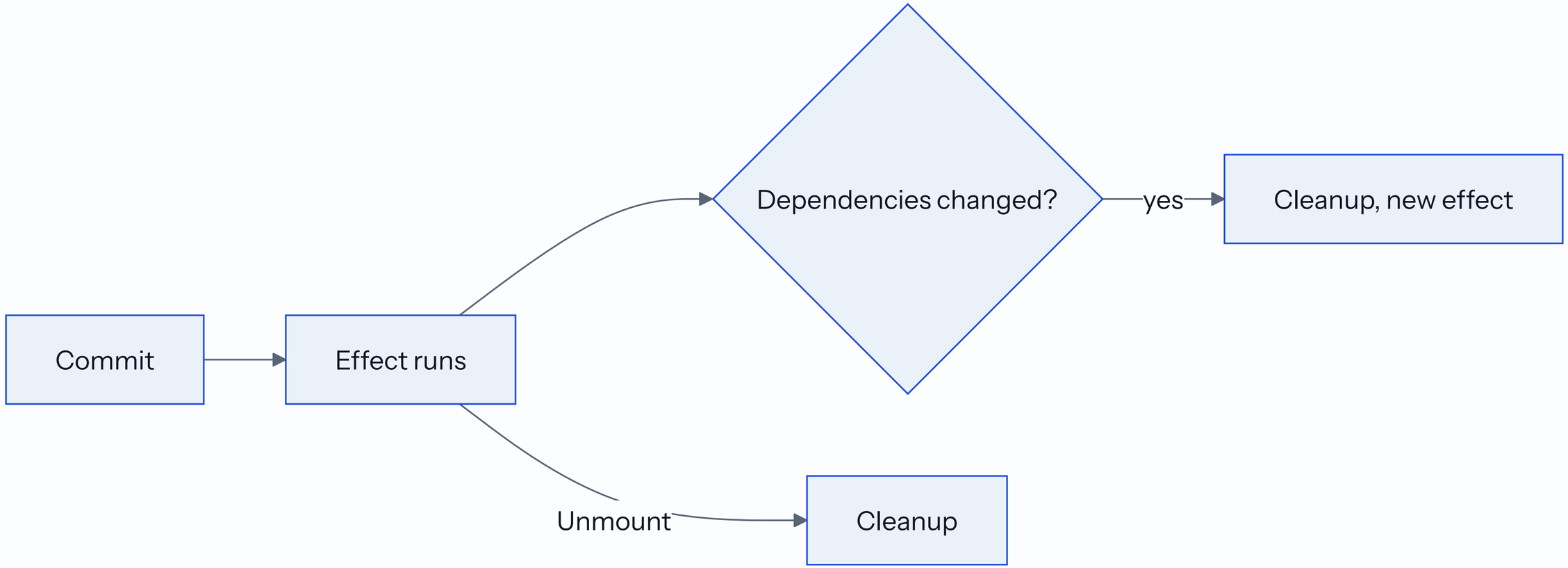
// Better: just calculate
const fullNameDerived = `${firstName} ${lastName}`
```

Effects synchronize with the outside world

- External systems are browser APIs, timers, network
- The effect runs after the commit
- The cleanup reverses it

```
export const OnlineBadge = () => {
  const [isOnline, setIsOnline] = useState(navigator.onLine)
  useEffect(() => {
    const updateStatus = () => setIsOnline(navigator.onLine)
    window.addEventListener('online', updateStatus)
    window.addEventListener('offline', updateStatus)
    return () => {
      window.removeEventListener('online', updateStatus)
      window.removeEventListener('offline', updateStatus)
    }
  }, [])
  return <span>{isOnline ? 'Online' : 'Offline'}</span>
}
```

Every effect starts and stops



Old responses must not win

- Every used value belongs in the array
- Outdated responses are discarded
- The exhaustive-deps lint rule is right

```
useEffect(() => {
  const controller = new AbortController()
  fetch(`/api/products/${productId}`, { signal: controller.signal })
    .then((response) => response.json())
    .then((product: Product) => setProduct(product))
    .catch((error: unknown) => {
      if (!controller.signal.aborted) setError(String(error))
    })
  return () => controller.abort()
}, [productId])
```

Read without reacting

- useEffectEvent always reads the latest values
- The function does not belong in the dependency array
- Call it only from effects, never during rendering

```
import { useEffect, useEffectEvent } from 'react'

export const ProductView = ({ productId, cartCount }: ProductViewProps) => {
  // Reads cartCount without the Effect reacting to it
  const logVisit = useEffectEvent((visitedId: string) => {
    trackVisit(visitedId, cartCount)
  })

  useEffect(() => {
    logVisit(productId)
  }, [productId])

  return <ProductDetails productId={productId} />
}
```

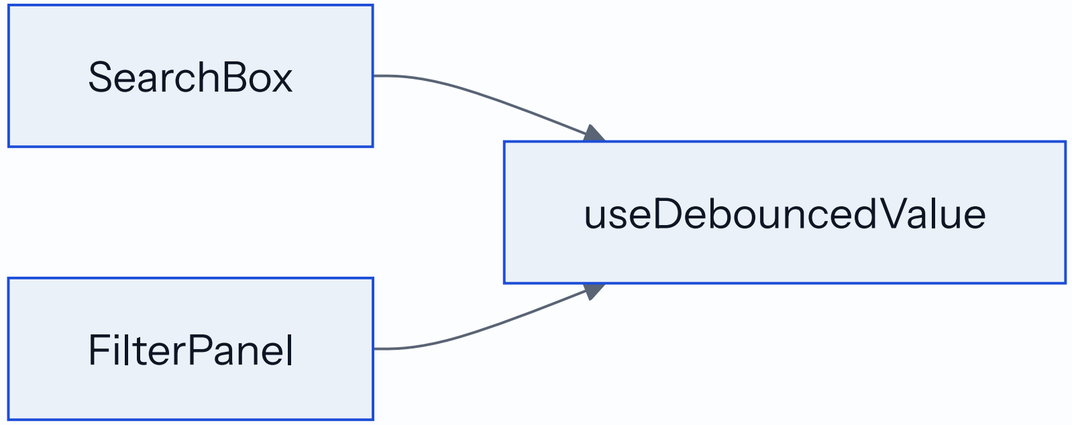
Ask first, then write the effect

- Ask about the external system before every effect
- Give every effect a matching cleanup
- Move recurring effects into custom hooks

custom hooks, context, reducer

Custom hooks share behavior, not state · Context is not a store for everything · Prevent impossible states with useReducer

The same logic, twice



A hook is a function

- Name starts with use
- Calls other hooks
- Each use has its own state

```
export const useToggle = (initialValue = false) => {
  const [isOn, setIsOn] = useState(initialValue)
  const toggle = useCallback(() => setIsOn((current) => !current), [])
  return { isOn, toggle } as const
}

// Nutzung
const { isOn: isMenuOpen, toggle: toggleMenu } = useToggle()
```

Hooks encapsulate effects

- The effect and cleanup move into the hook
- The generic type stays with the caller
- The component sees only the result

```
export function useDebouncedValue<T>(value: T, delayMs = 300): T {
  const [debouncedValue, setDebouncedValue] = useState(value)
  useEffect(() => {
    const timeoutId = setTimeout(() => setDebouncedValue(value), delayMs)
    return () => clearTimeout(timeoutId)
  }, [value, delayMs])
  return debouncedValue
}

// Use in search
const debouncedQuery = useDebouncedValue(searchText, 400)
```

Hooks need a fixed order

- Call only at the top level
- Call only in components and custom hooks
- Always enable eslint-plugin-react-hooks

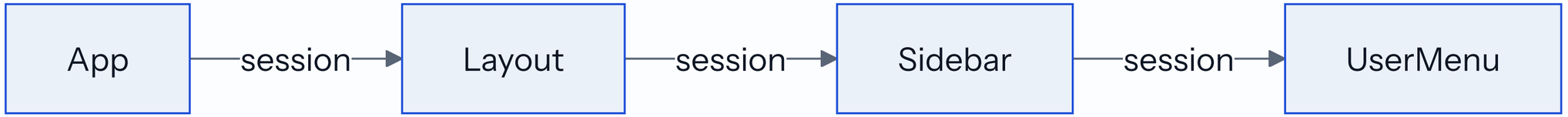


A good hook reads like a sentence

- Return an object for two or more values
- Choose the name by intent, not by technique
- Extract only logic that is truly repeated

Prop drilling through every level

- Values move through components that do not need them
- Every intermediate level must be changed
- Composition often helps even before context



Type-safe context with null as the default

- Default null instead of a fake object
- Access only through a custom hook
- Error message when the provider is missing

```
type Session = { userName: string; logout: () => void }

const SessionContext = createContext<Session | null>(null)

export const useSession = (): Session => {
  const session = useContext(SessionContext)
  if (!session) throw new Error('useSession braucht einen SessionProvider')
  return session
}
```

Providers cause re-renders

- Each value change renders all readers
- A new object per render counts as a change
- Frequently changing values get their own context

```
export const SessionProvider = ({ children }: { children: ReactNode }) => {
  const [userName, setUserName] = useState('Ada')
  const logout = useCallback(() => setUserName(''), [])
  const session = useMemo(() => ({ userName, logout }), [userName, logout])
  return <SessionContext value={session}>{children}</SessionContext>
}
```

Every state has its place

- Server data: hook or route loader
- URL state: router with search params
- Global UI values: context

Two mistakes in almost every project

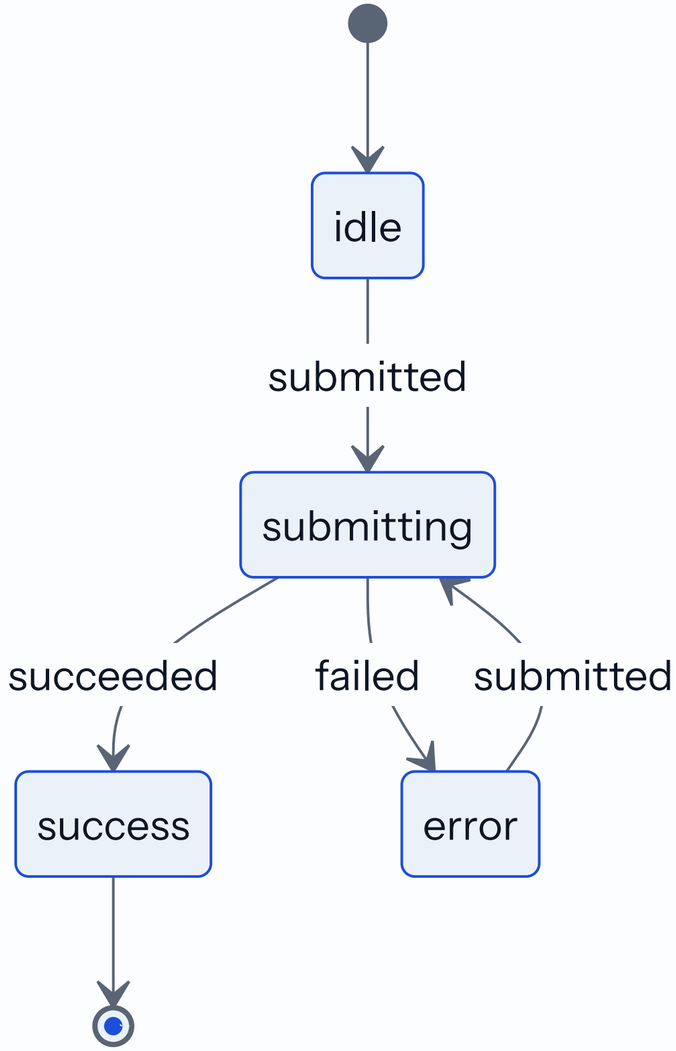
- One mega-context for everything
- Server state as a copy in context
- useReducer for complex transitions

When does useState tip over?

- Several booleans are connected
- Impossible combinations become possible
- Logic is spread across many event handlers

```
const [isSubmitting, setIsSubmitting] = useState(false)
const [hasError, setHasError] = useState(false)
const [isDone, setIsDone] = useState(false)
// What does isSubmitting and isDone at the same time mean?
```

Draw states and transitions first



The reducer translates the diagram

- State and actions are union types
- Old state plus action produces new state
- TypeScript checks that every action is handled

```
type CheckoutState =
  | { status: 'idle' | 'submitting' }
  | { status: 'success'; orderId: string }
  | { status: 'error'; message: string }

type CheckoutAction =
  | { type: 'submitted' }
  | { type: 'succeeded'; orderId: string }
  | { type: 'failed'; message: string }

const checkoutReducer = (state: CheckoutState, action: CheckoutAction): CheckoutState => {
  switch (action.type) {
    case 'submitted': return { status: 'submitting' }
    case 'succeeded': return { status: 'success', orderId: action.orderId }
    case 'failed': return { status: 'error', message: action.message }
  }
}
```

Side effects stay in the event handler

- dispatch is stable and safe to pass on
- Requests run in the event handler
- Rendering only reads the status

```
export const Checkout = ({ input }: { input: PlaceOrderInput }) => {
  const [state, dispatch] = useReducer(checkoutReducer, { status: 'idle' })
  const handleSubmit = async () => {
    dispatch({ type: 'submitted' })
    try {
      const order = await placeOrder(input)
      dispatch({ type: 'succeeded', orderId: order.id })
    } catch (error) {
      dispatch({ type: 'failed', message: String(error) })
    }
  }
  if (state.status === 'success') return <p>Bestellung {state.orderId}</p>
  return <button disabled={state.status === 'submitting'} onClick={handleSubmit}>Bestellen</button>
}
```

Redux is the same principle, just global

- Reducers can be tested without React
- Context plus reducer often replaces a store
- Server data is loaded by its own hook or loader

loading data, Suspense, routing, forms

Old answers must not overtake new ones · Where do you wait, where are errors caught? · The URL is state you can share · One schema for form, server, and type

Server data does not belong to you

- The server owns the data, you hold a copy
- Every request can fail or be overtaken
- Loading, error, and success are separate states

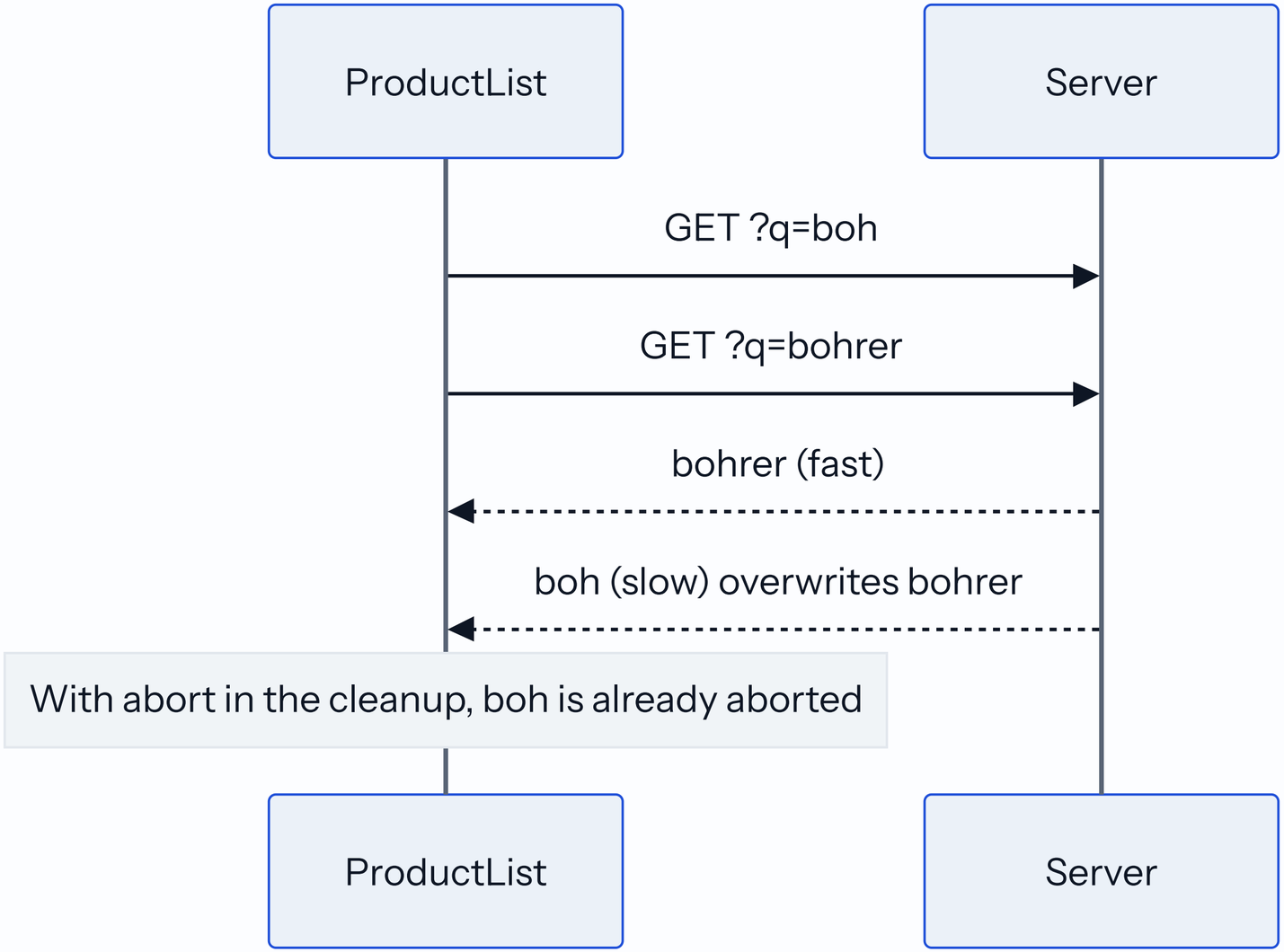
fetch returns unknown, not Product

- fetch does not throw on status 404 or 500
- The response is unknown until you check it
- The signal makes the request abortable

```
const isProduct = (value: unknown): value is Product =>
  typeof value === 'object' && value !== null && 'id' in value && 'priceCents' in value && 'stock' in value

export const fetchProducts = async (search: string, signal?: AbortSignal) => {
  const response = await fetch(`/api/products?q=${encodeURIComponent(search)}`, { signal })
  if (!response.ok) throw new Error(`HTTP ${response.status}`)
  const data: unknown = await response.json()
  if (!Array.isArray(data) || !data.every(isProduct)) throw new Error('Unerwartete Antwort')
  return data
}
```

The slow response arrives last



One hook per resource

- The effect starts and aborts
- The result remembers its search text
- If it does not match, the status is loading

```
export const useProducts = (search: string): LoadState<Product[]> => {
  const [result, setResult] = useState<Result<Product[]> | null>(null)
  useEffect(() => {
    const controller = new AbortController()
    fetchProducts(search, controller.signal)
      .then((data) => setResult({ search, status: 'success', data }))
      .catch((error) => {
        if (!controller.signal.aborted) setResult({ search, status: 'error', message: String(error) })
      })
    return () => controller.abort()
  }, [search])
  return result?.search === search ? result : { status: 'loading' }
}
```

When a server-state library is worth it

- Many views share the same data
- Data should stay fresh in the background
- An unstable network needs retries

Load, check, abort

- The loading function checks status and response shape
- The effect aborts in the cleanup
- The status is a Union Type instead of booleans

if loading everywhere

- Each component checks loading and error itself
- Many spinners appear one after another
- Error handling is inconsistent

```
export const OrderPage = ({ orderId }: { orderId: string }) => {
  const order = useOrder(orderId)
  if (order.status === 'loading') return <Spinner />
  if (order.status === 'error') return <p role="alert">{order.message}</p>
  // … only here does the actual Component begin
  return <OrderView order={order.data} />
}
```

use reads a Promise but does not cache it

- use(promise) suspends until the value is available
- The Promise comes from a cache, never from rendering
- One Map per resource is enough as a cache

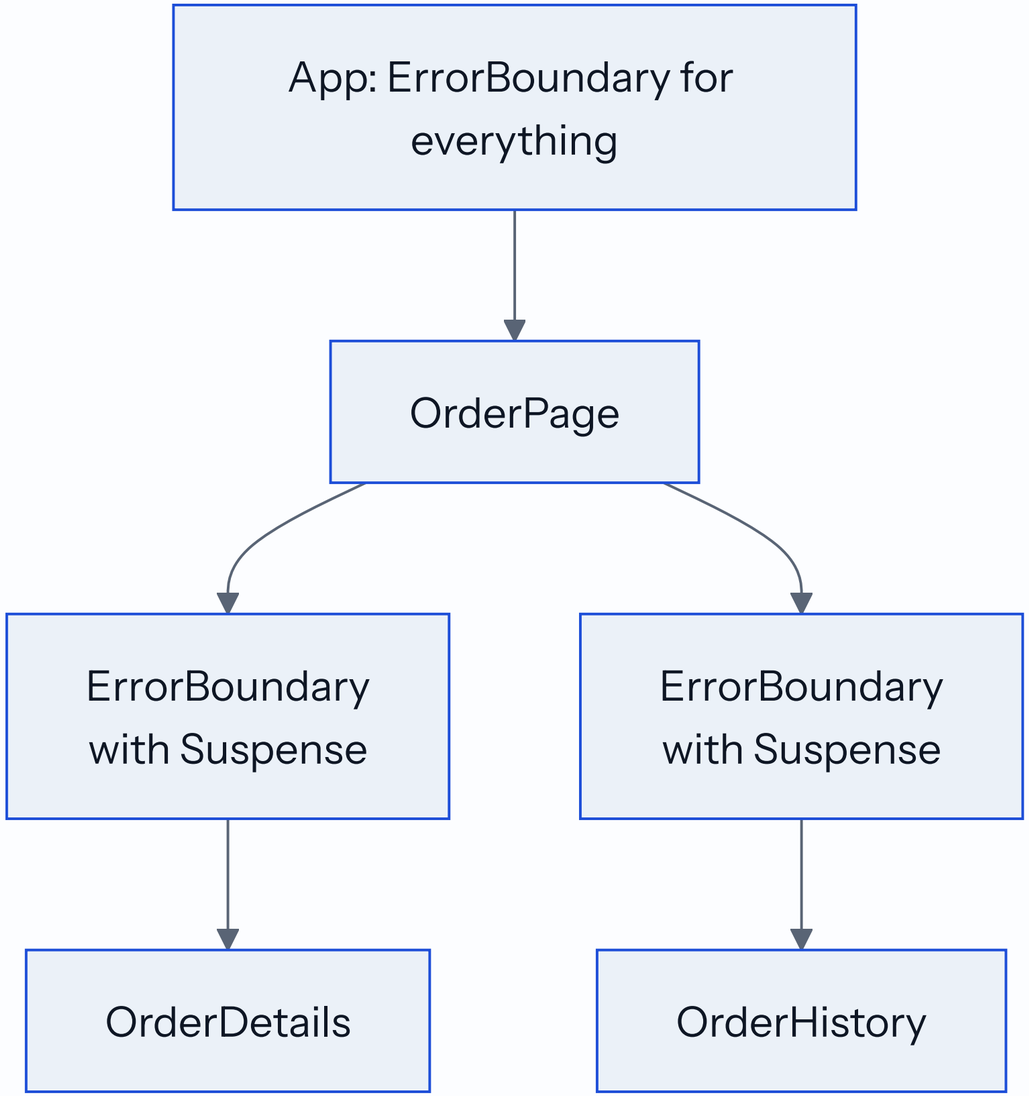
```
import { use } from 'react'

export const orderCache = new Map<string, Promise<Order>>()

const loadOrder = (orderId: string) => {
  let promise = orderCache.get(orderId)
  if (!promise) {
    promise = fetchOrder(orderId)
    orderCache.set(orderId, promise)
  }
  return promise
}

export const OrderDetails = ({ orderId }: { orderId: string }) => {
  const order = use(loadOrder(orderId))
  return <h2>Bestellung {order.id}: {order.status}</h2>
}
```

Boundaries belong at area borders



react-error-boundary instead of your own class

- React still needs a class component for boundaries
- react-error-boundary wraps it, so you do not write one
- The FallbackComponent receives the error and reset

```
import { ErrorBoundary, type FallbackProps } from 'react-error-boundary'

const OrderError = ({ error, resetErrorBoundary }: FallbackProps) => (
  <p role="alert">
    {error instanceof Error ? error.message : 'Bestellung nicht verfügbar'}
    <button onClick={resetErrorBoundary}>Erneut versuchen</button>
  </p>
)
```

Waiting and errors per area

- The Error Boundary wraps the Suspense Boundary
- onReset deletes the rejected Promise from the cache
- The component describes only the success case

```
export const OrderSection = ({ orderId }: { orderId: string }) => (
  <ErrorBoundary FallbackComponent={OrderError} onReset={() => orderCache.delete(orderId)}>
    <Suspense fallback={<Spinner />}>
      <OrderDetails orderId={orderId} />
    </Suspense>
  </ErrorBoundary>
)
```

Not every error lands in the boundary

- Boundaries do not catch errors in event handlers
- A single boundary at the top is too broad
- Routes are often good borders for boundaries

The URL is state

- Links can be shared and saved as bookmarks
- Filters and selections belong in the URL
- The router loads the data for the URL

Routes are objects, layouts have an Outlet

- createBrowserRouter describes all routes
- children render in the layout's Outlet
- RouterProvider comes from react-router/dom

```
import { createBrowserRouter } from 'react-router'
import { RouterProvider } from 'react-router/dom'

const router = createBrowserRouter([
  {
    path: '/',
    Component: ShopLayout,
    ErrorBoundary: RouteError,
    children: [
      { index: true, Component: ProductsPage },
      { path: 'orders', loader: ordersLoader, Component: OrdersPage },
      { path: 'orders/:orderId', loader: orderLoader, Component: OrderPage },
    ],
  },
])

createRoot(rootElement).render(<RouterProvider router={router} />)
```

The loader loads before rendering

- The loader receives params and request
- request.signal cancels stale navigations
- useLoaderData returns the loader’s type

```
import { useLoaderData, type LoaderFunctionArgs } from 'react-router'

export const orderLoader = async ({ params, request }: LoaderFunctionArgs) => {
  const order = await fetchOrder(params.orderId ?? '', request.signal)
  return { order }
}

export const OrderPage = () => {
  const { order } = useLoaderData<typeof orderLoader>()
  return (
    <h1>
      Bestellung {order.id}: {order.status}
    </h1>
  )
}
```

Search params are external input

- useSearchParams reads and writes the query
- Every value is validated and gets a default
- A new search param starts the loader again

```
const statuses = ['draft', 'placed', 'paid', 'shipped', 'cancelled'] as const

const readStatus = (value: string | null) =>
  statuses.find((status) => status === value) ?? 'all'

export const StatusFilter = () => {
  const [searchParams, setSearchParams] = useSearchParams()
  const status = readStatus(searchParams.get('status'))
  return (
    <select value={status} onChange={(event) => setSearchParams({ status: event.target.value })}>
      <option value="all">Alle</option>
      {statuses.map((value) => <option key={value}>{value}</option>)}
    </select>
  )
}
```

Each route catches its errors

- If the loader throws, the route's ErrorBoundary renders
- useRouteError returns the error
- Link takes you back without reloading the app

```
import { Link, isRouteErrorResponse, useRouteError } from 'react-router'

export const RouteError = () => {
  const error = useRouteError()
  const message = isRouteErrorResponse(error)
    ? `${error.status} ${error.statusText}`
    : 'Unerwarteter Fehler'
  return (
    <main role="alert">
      <h1>{message}</h1>
      <Link to="/orders">Zur Bestellübersicht</Link>
    </main>
  )
}
```

Common routing mistakes

- a tags reload the whole app
- Filters live twice, in the URL and state
- Data loads twice, in the loader and Effect

Validation rules in three places

- The same rules live in browser, server, and type
- Each copy goes stale differently
- Error messages sound different everywhere

One schema for browser and server

- The schema lives in a shared module
- The type comes from z.infer
- strictObject rejects unknown fields

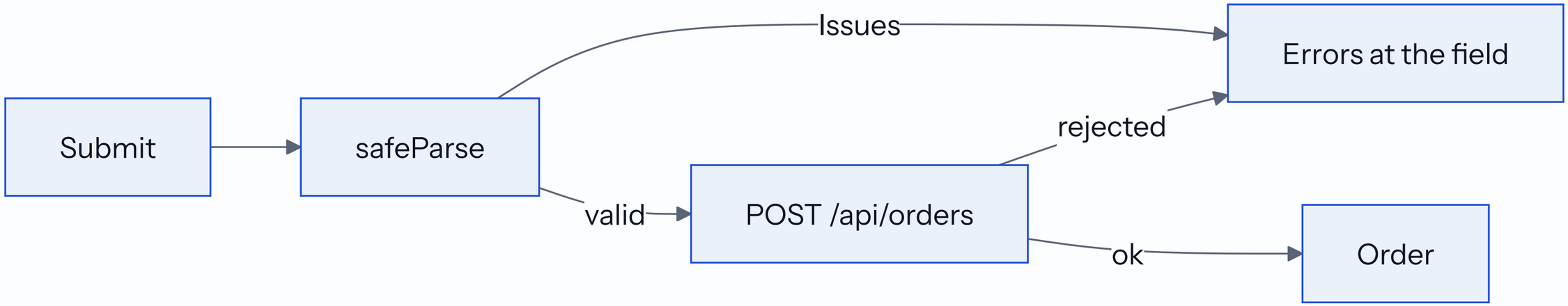
```
// shared/order-schema.ts - the backend uses the same module
import { z } from 'zod'

export const OrderItemInput = z.strictObject({
  productId: z.string().min(1),
  quantity: z.number().int().min(1),
})

export const PlaceOrderInput = z.strictObject({
  customerId: z.string().min(1),
  items: z.array(OrderItemInput).min(1),
})

export type PlaceOrderInput = z.infer<typeof PlaceOrderInput>
```

First safeParse, then the request



An issue path points to the field

- FormData returns text, but the schema wants numbers
- issue.path becomes a key like items.0.quantity
- toFieldErrors translates issues into sentences

```
const handleSubmit = (event: SubmitEvent<HTMLFormElement>) => {
  event.preventDefault()
  const formData = new FormData(event.currentTarget)
  const items = cart.map((item) => ({
    productId: item.productId,
    quantity: Number(formData.get(item.productId)),
  }))
  const result = PlaceOrderInput.safeParse({ customerId, items })
  if (!result.success) return setErrors(toFieldErrors(result.error.issues))
  setErrors({})
  void placeOrder(result.data)
}
```

Errors belong at the field

- The field key follows the path in the schema
- aria-invalid marks the field as invalid
- aria-describedby lets the message be read aloud

```
export const QuantityField = ({ item, index, errors }: QuantityFieldProps) => {
  const error = errors[`items.${index}.quantity`]
  const errorId = `${item.productId}-error`
  return (
    <div>
      <label htmlFor={item.productId}>Menge</label>
      <input id={item.productId} name={item.productId} defaultValue={item.quantity}
        aria-invalid={Boolean(error)} aria-describedby={error ? errorId : undefined} />
      {error && <p id={errorId}>{error}</p>}
    </div>
  )
}
```

No form that scolds on the first letter

- Marking red immediately on every keystroke
- Showing server errors only as a toast
- Validating only in the browser

accessibility, testing, design system

Accessibility from the start · Test what users see and do · Atomic Design gives the team a vocabulary · Generics preserve the caller’s type

Why from the start?

- Many people use a keyboard or screenreader
- Accessibility is mandatory for many products
- Retrofitting costs many times more

Semantics first

- button for actions, a for navigation
- Headings in a meaningful order
- Landmarks: header, nav, main, footer

```
// Bad: not focusable, no role, no keyboard
const fakeButton = <div className="btn" onClick={save}>Speichern</div>

// Good: everything built in
const saveButton = <button type="button" className="btn" onClick={save}>Speichern</button>

// Icon button needs an Accessible Name
const closeButton = (
  <button type="button" aria-label="Dialog schließen" onClick={onClose}>
    <XIcon aria-hidden="true" />
  </button>
)
```

Labels and errors belong to the field

- useId creates stable, unique IDs
- label connects to the field via htmlFor
- aria-describedby announces the error message

```
export const EmailField = ({ error }: { error?: string }) => {
  const inputId = useId()
  const errorId = `${inputId}-error`
  return (
    <div>
      <label htmlFor={inputId}>E-Mail</label>
      <input
        id={inputId}
        name="email"
        type="email"
        aria-invalid={Boolean(error)}
        aria-describedby={error ? errorId : undefined}
      />
      {error && <p id={errorId}>{error}</p>}
    </div>
  )
}
```

Focus must move in a meaningful way

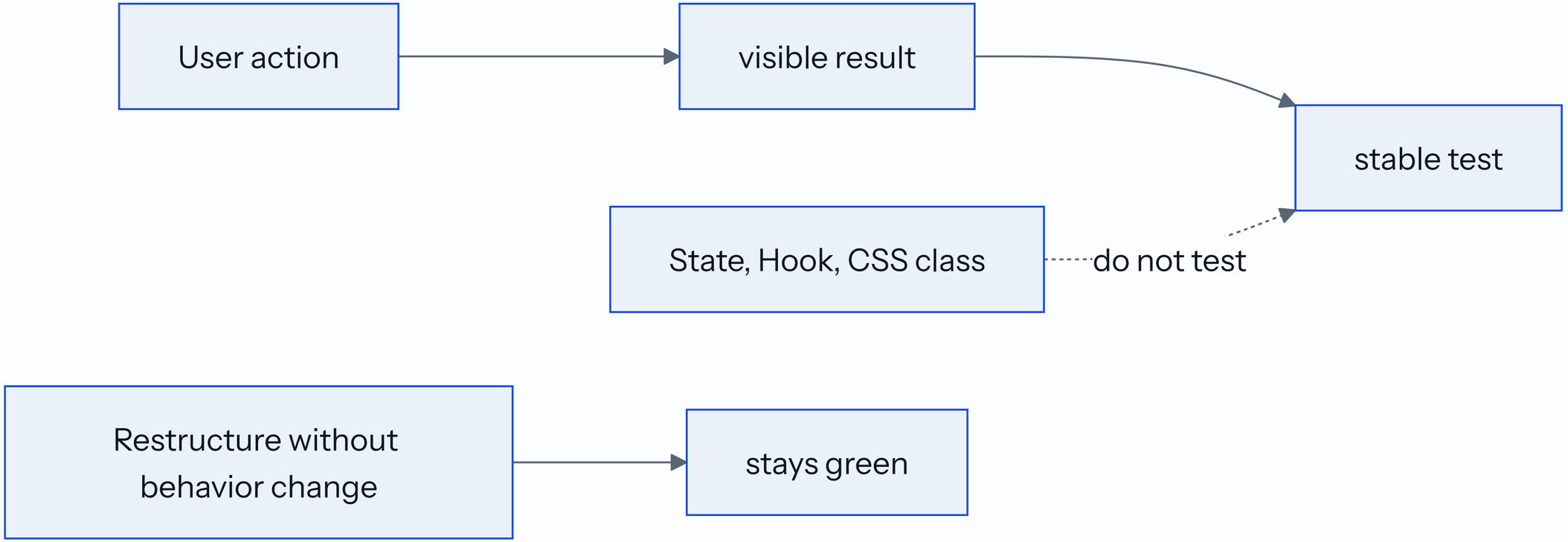
- After opening: focus into the dialog
- After closing: back to the trigger
- With the dialog element: both are built in

```
export const ConfirmDialog = ({ isOpen, onClose, children }: ConfirmDialogProps) => {
  const dialogRef = useRef<HTMLDialogElement>(null)
  useEffect(() => {
    const dialog = dialogRef.current
    if (!dialog) return
    if (isOpen && !dialog.open) dialog.showModal()
    if (!isOpen && dialog.open) dialog.close()
  }, [isOpen])
  return (
    <dialog ref={dialogRef} onClose={onClose}>
      {children}
    </dialog>
  )
}
```

Tools find a lot, but not everything

- eslint-plugin-jsx-a11y checks in the editor
- Tests with getByRole find gaps
- Keyboard and screenreader show the rest

What makes a good test?



Vitest uses the Vite configuration

- Vitest shares plugins and aliases with the build
- jsdom simulates the browser, jest-dom adds matchers

```ts
// vite.config.ts
import { defineConfig } from 'vitest/config'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  test: {
    environment: 'jsdom',
    setupFiles: ['./src/test/setup.ts'],
  },
})

// src/test/setup.ts
import '@testing-library/jest-dom/vitest'
import { cleanup } from '@testing-library/react'
import { afterEach, vi } from 'vitest'

afterEach(() => {
  cleanup()
  vi.unstubAllGlobals()
})
```

Tests operate the UI like users

- Find elements by role and accessible name
- Type and click with userEvent
- Check only the visible result

```
it('legt die gewählte Menge in den Warenkorb', async () => {
  const user = userEvent.setup()
  const onAdd = vi.fn()
  render(<AddToCartForm stock={5} onAdd={onAdd} />)

  const quantity = screen.getByRole('textbox', { name: 'Menge' })
  await user.clear(quantity)
  await user.type(quantity, '3')
  await user.click(screen.getByRole('button', { name: 'In den Warenkorb' }))

  expect(onAdd).toHaveBeenCalledWith(3)
})
```

findBy waits, a fetch stub responds

- findBy waits for the element
- vi.stubGlobal replaces fetch for one test
- The error path gets a response with status 500

```
const products = [{ id: 'p1', name: 'Hammer', priceCents: 1290, stock: 3 }]

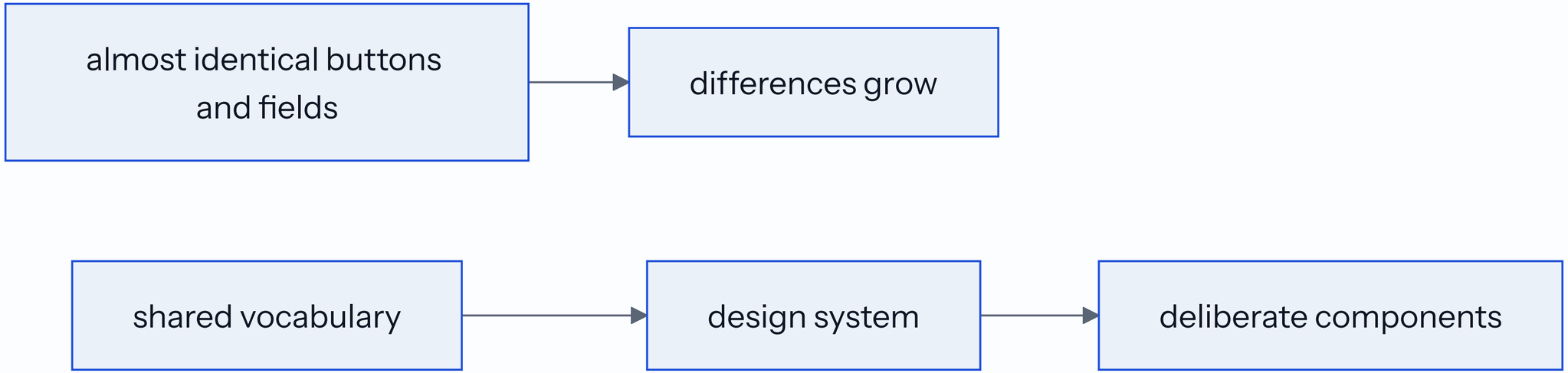
it('zeigt gefundene Produkte', async () => {
  vi.stubGlobal('fetch', vi.fn(async () => Response.json(products)))
  render(<ProductList search="ham" />)
  expect(await screen.findByText('Hammer')).toBeInTheDocument()
})

it('meldet Server Errors', async () => {
  vi.stubGlobal('fetch', vi.fn(async () => new Response(null, { status: 500 })))
  render(<ProductList search="ham" />)
  expect(await screen.findByRole('alert')).toHaveTextContent('HTTP 500')
})
```

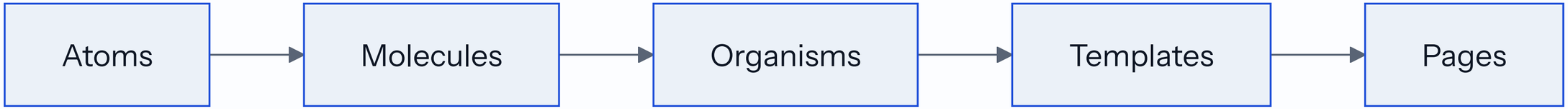
data-testid is the last resort

- Search for everything via data-testid
- Check internal hooks and state
- Update large snapshots unread

Why component design?



Five levels from atom to page

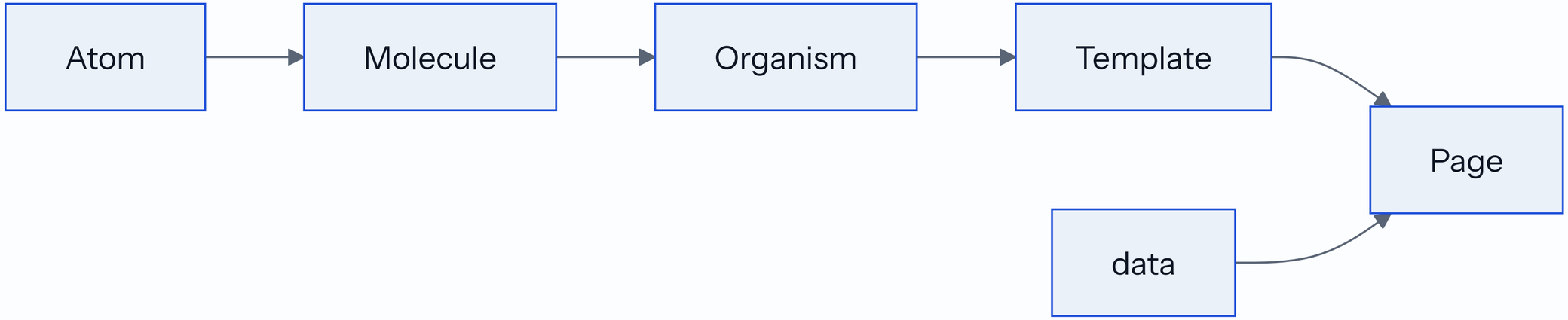


Atoms only know their props

```
type ButtonProps = { label: string; type?: 'button' | 'submit'; onClick?: () => void }
export const Button = ({ label, type = 'button', onClick }: ButtonProps) => (
  <button type={type} className="btn" onClick={onClick}>{label}</button>
)

export const SearchField = ({ onSearch }: { onSearch: (term: string) => void }) => {
  const [term, setTerm] = useState('')
  return (
    <form onSubmit={(event) => { event.preventDefault(); onSearch(term) }}>
      <Input value={term} onChange={setTerm} />
      <Button type="submit" label="Suchen" />
    </form>
  )
}
```

Only the Page knows real data



Vocabulary beats folder structure

- Assigning a level is secondary
- Storybook serves as a pattern library
- Feature components stay in the feature

One list for every type?

- Copies for products, customers, orders
- any and casts at the call site
- No path that scales

```
// Typinformation geht verloren
type ListProps = { items: any[]; renderItem: (item: any) => ReactNode }
```

T comes from the caller

- The component gets a type parameter T
- getKey returns the key for each item
- The caller gets its own type back

```
type ListProps<T> = {
  items: readonly T[]
  getKey: (item: T) => string
  renderItem: (item: T) => ReactNode
}

export function List<T>({ items, getKey, renderItem }: ListProps<T>) {
  return <ul>{items.map((item) => <li key={getKey(item)}>{renderItem(item)}</li>)}</ul>
}

// T is inferred as Customer
const customerList = (
  <List items={customers} getKey={(customer) => customer.id} renderItem={(customer) => customer.email} />
)
```

keyof allows only real columns

- Typos in column names fail at compile time
- The constraint with extends guarantees the id

```
type Column<T> = {
  key: keyof T & string
  header: string
  render?: (row: T) => ReactNode
}

type DataTableProps<T extends { id: string }> = {
  rows: readonly T[]
  columns: readonly Column<T>[]
}
```

Each column renders its value

- Without a renderer, the value appears as text
- With render, the column controls the display

```
export function DataTable<T extends { id: string }>({ rows, columns }: DataTableProps<T>) {
  return (
    <table>
      <tbody>
        {rows.map((row) => (
          <tr key={row.id}>
            {columns.map((column) => (
              <td key={column.key}>{column.render?.(row) ?? String(row[column.key])}</td>
            ))}
          </tr>
        ))}
      </tbody>
    </table>
  )
}
```

Variants exclude each other

- Some props are mutually exclusive
- A Discriminated Union replaces optional fields
- TypeScript enforces valid combinations

```
type ActionProps =
  | { kind: 'button'; label: string; onPress: () => void }
  | { kind: 'link'; label: string; href: string }

export const Action = (props: ActionProps) =>
  props.kind === 'link'
    ? <a href={props.href}>{props.label}</a>
    : <button type="button" onClick={props.onPress}>{props.label}</button>
```

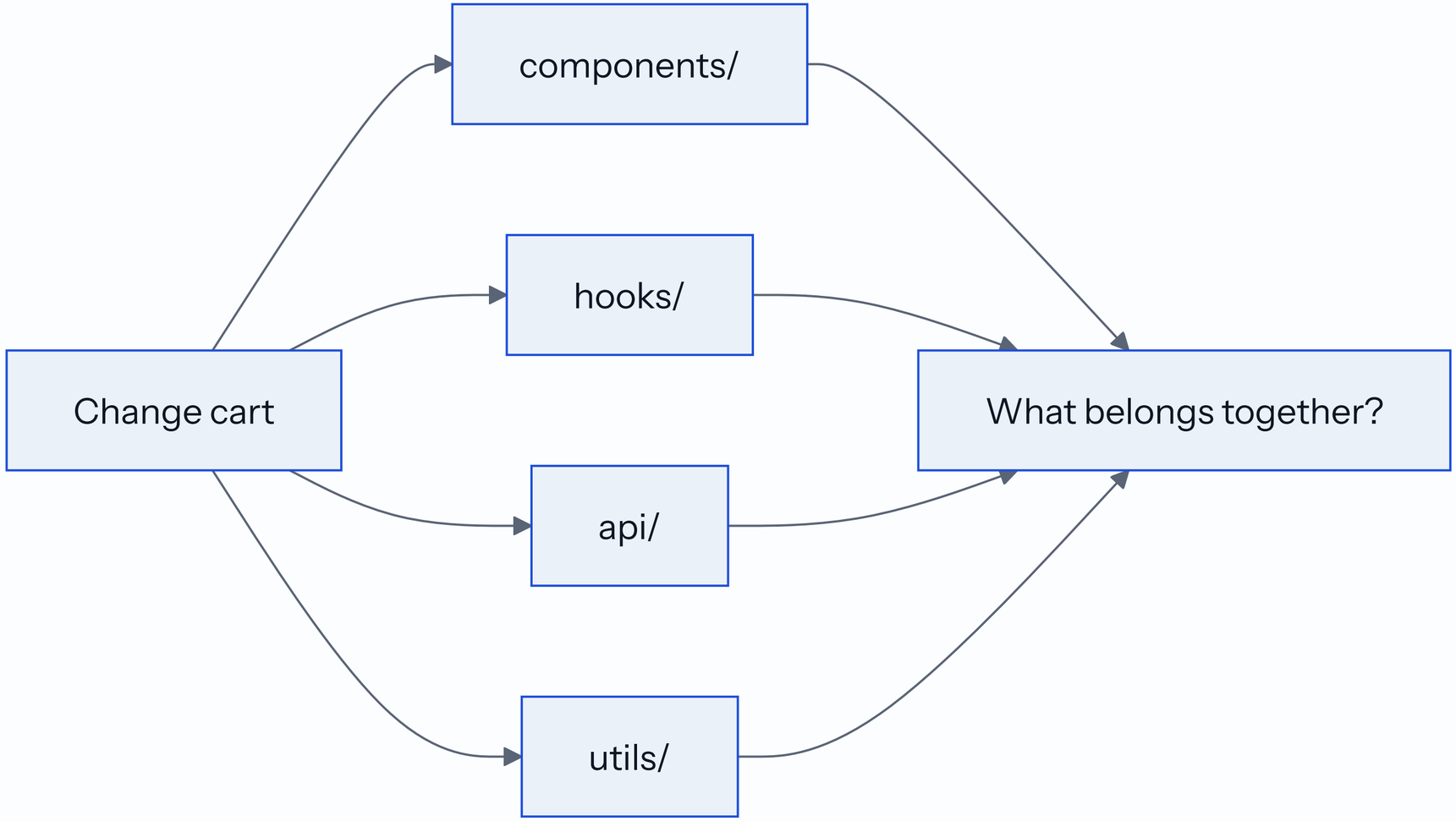
Too generic is also wrong

- Nobody understands five type parameters
- Casts with as hide real errors
- Not every kind of flexibility needs generics

architecture, performance, production

Organize by feature, verify boundaries · Measure first, then optimize · Optimistic UI needs a way back · Production starts with the build

Folders by technology do not scale

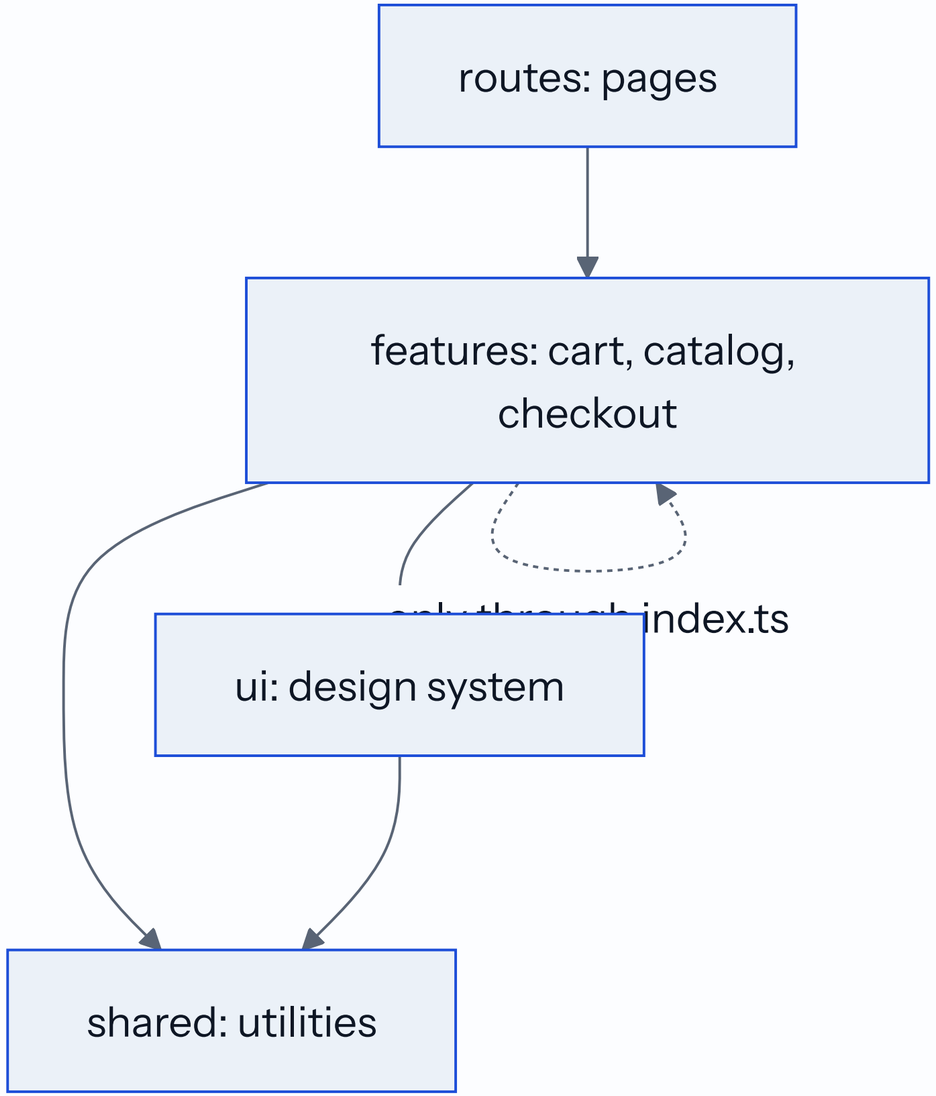


Organize by feature

- Everything for one feature is colocated
- index.ts is the public API
- shared and ui stay domain-free

```
src/
  app/              Router, Provider, Entry Point
  routes/           Pages compose features
  features/
    cart/
      api.ts          fetch functions and loaders
      CartSummary.tsx
      useCart.ts
      index.ts        Public API
    catalog/
  ui/               Design System: Button, Card, DataTable
  shared/           domain-free utilities
```

Dependencies point downward



The design system knows no domain data

- Components do not call an API
- Design tokens live in the Tailwind theme
- Slots and union types shape the API

```
// ui/Card.tsx – knows nothing about orders
export const Card = ({ header, actions, children }: CardProps) => (
  <section className="rounded-lg border border-border bg-surface p-4">
    {header && <header className="mb-2 font-semibold">{header}</header>}
    {children}
    {actions && <footer className="mt-4 flex gap-2">{actions}</footer>}
  </section>
)

// features/orders/OrderCard.tsx – domain logic
export const OrderCard = ({ order }: { order: Order }) => (
  <Card header={`Bestellung ${order.id}`}>{order.items.length} Positionen</Card>
)
```

Atomic Design or feature folders?

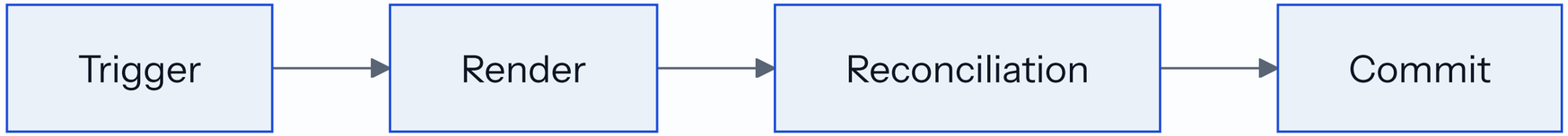
- Slice ui by Atomic Design: atoms, molecules, organisms
- Templates are layouts, pages live in routes
- Slice features by domain, not by levels

Decisions belong in short ADRs

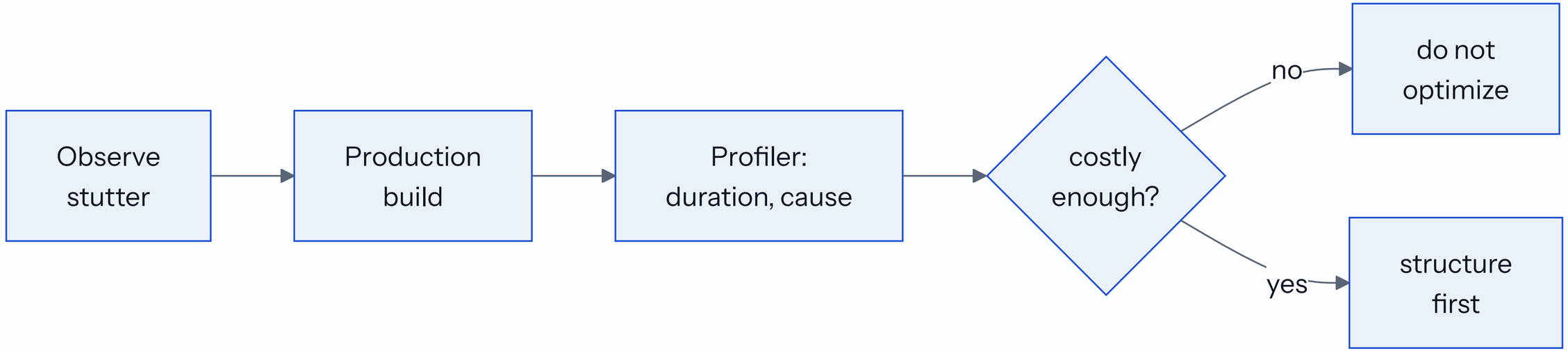
- Write ADRs with context, decision, and consequences
- Automate rules instead of debating them
- Let the structure grow with you, not all at the start

When does React render?

- Its own state changes
- The parent component renders
- A context it reads changes



Measure first



Structure before memo

- Move state down
- Pass expensive parts in as children
- Virtualize long lists

```
// Before: every keystroke renders the whole page
// After: only SearchBox has the state
const SearchBox = () => {
  const [searchText, setSearchText] = useState('')
  return <input value={searchText} onChange={(event) => setSearchText(event.target.value)} />
}

export const ProductPage = () => (
  <Layout sidebar={<SearchBox />}>
    <ExpensiveProductGrid />
  </Layout>
)
```

Memoization is handled by the compiler

- memo skips renders when props are the same
- useMemo and useCallback keep references stable
- React Compiler adds both at build time

```
const ProductRow = memo(({ product, onSelect }: ProductRowProps) => (
  <tr onClick={() => onSelect(product.id)}>
    <td>{product.name}</td>
  </tr>
))

const ProductTable = ({ products }: { products: readonly Product[] }) => {
  const [selectedId, setSelectedId] = useState<string | null>(null)
  const handleSelect = useCallback((id: string) => setSelectedId(id), [])
  return <tbody>{products.map((product) => <ProductRow key={product.id} product={product} onSelect={handleSelect} />)}</tbody>
}
```

Inputs stay urgent

- useDeferredValue for expensive derivations
- useTransition for non-urgent updates
- Smooth inputs despite expensive lists

```
export const ProductSearch = ({ products }: { products: readonly Product[] }) => {
  const [searchText, setSearchText] = useState('')
  const deferredSearchText = useDeferredValue(searchText)
  const isStale = searchText !== deferredSearchText
  return (
    <>
      <input value={searchText} onChange={(event) => setSearchText(event.target.value)} />
      <div style={{ opacity: isStale ? 0.6 : 1 }}>
        <ProductResults products={products} query={deferredSearchText} />
      </div>
    </>
  )
}
```

Hide without losing state

- Activity hides instead of unmounting
- State and DOM are preserved
- Effects pause while the part is hidden

```
import { Activity } from 'react'

export const ProductTabs = ({ tab }: { tab: 'details' | 'reviews' }) => (
  <>
    <Activity mode={tab === 'details' ? 'visible' : 'hidden'}>
      <ProductDetails />
    </Activity>
    <Activity mode={tab === 'reviews' ? 'visible' : 'hidden'}>
      <ProductReviews />
    </Activity>
  </>
)
```

Every form needs the same states

- Loading state and error state always get added
- Double clicks and outdated responses happen
- Boilerplate repeats in every form

useActionState returns state and pending

- action on the form receives FormData
- The return value becomes the new state
- isPending comes without your own useState

```
async function placeAction(_previous: unknown, formData: FormData) {
  const body = JSON.stringify(toPlaceOrderInput(formData))
  const response = await fetch('/api/orders', { method: 'POST', body })
  return { error: response.ok ? null : 'Bestellung abgelehnt' }
}

export function PlaceOrderForm() {
  const [state, formAction, isPending] = useActionState(placeAction, { error: null })
  return (
    <form action={formAction}>
      <CartFields />
      <button disabled={isPending}>Bestellen</button>
      {state.error && <p role="alert">{state.error}</p>}
    </form>
  )
}
```

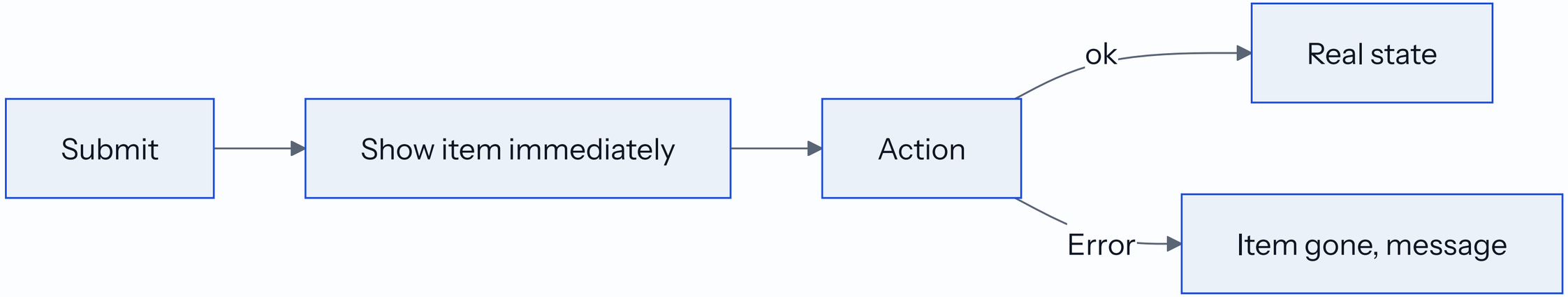
useOptimistic shows the expected result right away

- The expected value appears immediately
- It applies only while the Action is running
- After that, the real state counts again

```
const [optimisticItems, addOptimisticItem] = useOptimistic(
  items,
  (current: readonly OrderItem[], newItem: OrderItem) => [...current, newItem],
)

const addAction = async (product: Product) => {
  const newItem = { productId: product.id, quantity: 1, priceCents: product.priceCents }
  addOptimisticItem(newItem)
  try {
    const savedItems = await addItemToDraft(orderId, newItem)
    startTransition(() => setItems(savedItems))
  } catch {
    setError('Position nicht gespeichert')
  }
}
// <form action={() => addAction(product)}> … optimisticItems rendern
```

On error, the item disappears



Optimistic only where almost everything succeeds

- Line items in a draft work well; payment does not
- After success, the state from the server counts
- Idempotent endpoints can handle duplicate requests

vite build writes VITE_ values into the bundle

- vite build creates hashed, minified files
- Only VITE_ variables end up in the client
- A check function aborts startup when something is missing

```
// src/env.ts
const readEnv = (name: 'VITE_API_URL' | 'VITE_RELEASE') => {
  const value: unknown = import.meta.env[name]
  if (typeof value !== 'string' || value === '') throw new Error(`${name} fehlt`)
  return value
}

export const env = {
  apiUrl: new URL(readEnv('VITE_API_URL')).href,
  release: readEnv('VITE_RELEASE'),
}
```

Less JavaScript, faster startup

- Split code by route
- Load rarely used parts with lazy
- Analyze the bundle regularly

```
const ReportsPage = lazy(() => import('./features/reports/ReportsPage'))

export const ReportsRoute = () => (
  <Suspense fallback={<PageSkeleton />}>
    <ReportsPage />
  </Suspense>
)
```

Errors need release and route

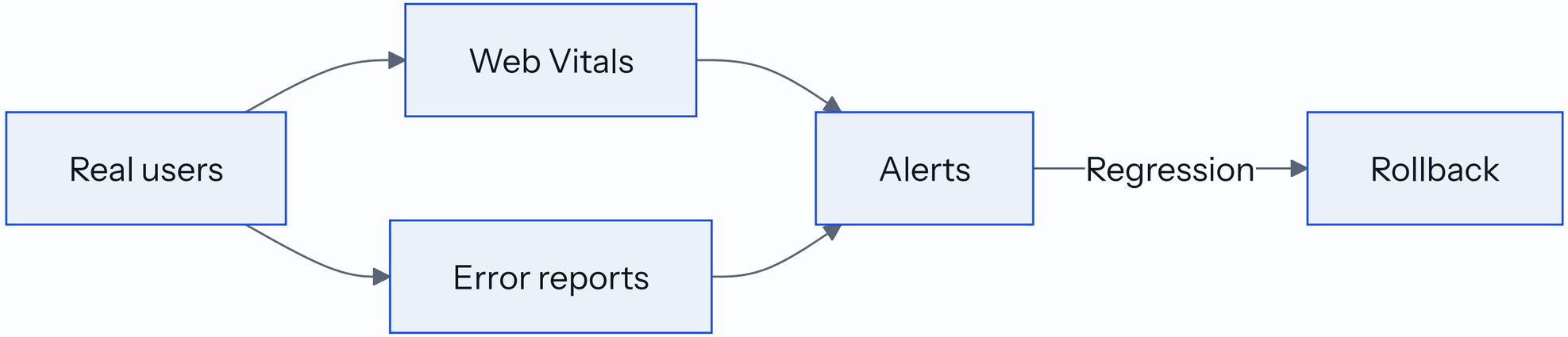
- createRoot reports errors from all Boundaries
- Global handlers catch the rest
- Every report includes release and route

```
const reportError = (error: unknown, source: string) =>
  navigator.sendBeacon('/api/client-errors', JSON.stringify({
    message: String(error), source, release: env.release, url: location.pathname,
  }))

window.addEventListener('error', (event) => reportError(event.error, 'window'))
window.addEventListener('unhandledrejection', (event) => reportError(event.reason, 'promise'))

createRoot(rootElement, {
  onCaughtError: (error) => reportError(error, 'boundary'),
  onUncaughtError: (error) => reportError(error, 'uncaught'),
}).render(<RouterProvider router={router} />)
```

Measure real users, not the laptop



Rollback beats hotfix

- Check the build, validate configuration
- Capture errors and Web Vitals with release
- Plan rollback within minutes

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen