# Svelte 5 and SvelteKit with TypeScript

You build the workshop store with Svelte 5, runes, and SvelteKit 3, from the first component to a tested Node server.

Mike Bild

Exercises: github.com/MikeBild/introduction-svelte

# Compiler and runes

Why doesn’t Svelte need a virtual DOM? · $state makes state visible and deeply reactive · Compute with $derived, act with $effect

# A compiler instead of a runtime library

- A .svelte file is source code for a compiler
- The compiler sees which markup depends on which value
- It generates code that changes exactly those DOM locations

# The analysis happens once, during the build



# A first component

- $state makes quantity reactive
- $derived recalculates the total
- onclick is a regular attribute

```
<script lang="ts">
  let quantity = $state(1)
  const priceCents = 4990
  let total = $derived(quantity * priceCents)
</script>

<button onclick={() => quantity--} disabled={quantity <= 1}>-</button>
<span>{quantity}</span>
<button onclick={() => quantity++}>+</button>
<p>Summe: {total / 100} €</p>
```

# Only what you mark is reactive

- Logic, markup, and scoped CSS live in one file
- Runes like $state make reactivity visible in code
- The compiler reports misspelled Runes immediately

```
npx sv create werkstatt-shop
cd werkstatt-shop
npm run dev
npm run check
```

# Svelte calculates during the build, not in the browser

- Dependencies at build time instead of diffing at runtime
- Mark state explicitly with Runes
- Find errors early with npm run check

# Runes are compiler keywords

- Start with a dollar sign and are not imported
- Look like function calls, but are language syntax
- Work in .svelte and .svelte.ts files

# Deep state in the cart

- Objects and arrays in $state become a proxy
- push and quantity += 1 both trigger an update
- Newly inserted objects also become reactive

```
export class Cart {
  items = $state<OrderItem[]>([])

  add(product: Product) {
    const item = this.items.find((entry) => entry.productId === product.id)
    if (item) item.quantity += 1
    else this.items.push({
      productId: product.id, quantity: 1, priceCents: product.priceCents,
    })
  }
}
```

# $state.raw only replaces, never changes

- No proxy, less work for large lists
- An update only happens on a new assignment
- Good for data that comes from the server as a whole

```
<script lang="ts">
  import type { Product } from '#lib/shop.js'

  let products = $state.raw<Product[]>([])

  async function reload() {
    const response = await fetch('/api/products')
    products = await response.json()
  }
</script>

<button onclick={reload}>Neu laden ({products.length})</button>
```

# $state.snapshot at the system boundary

- Deep state is a proxy, but some APIs want a plain object
- $state.snapshot returns a static copy
- Typical before structuredClone, logging, or sending

```
const copy = $state.snapshot(cart.items)
const backup = structuredClone(copy)
console.log(copy)
```

# Three variants, one rule of thumb

- $state as the default for local and shared state
- $state.raw for large data that you only replace
- $state.snapshot when external code expects an object

# Values from values

- $derived only depends on other state
- Svelte remembers the dependencies and recalculates as needed
- The domain function totalCents stays pure TypeScript

```
export class Cart {
  items = $state<OrderItem[]>([])
  total = $derived(totalCents(this.items))
  count = $derived(this.items.reduce((sum, item) => sum + item.quantity, 0))
}
```

# $derived.by for multiple lines

- One expression is not enough, but a function is
- The result stays a derived value without a side effect
- Search term and filter determine the results list

```
<script lang="ts">
  let { products }: { products: Product[] } = $props()
  let query = $state('')
  let onlyInStock = $state(false)

  let visible = $derived.by(() => {
    const q = query.trim().toLowerCase()
    return products.filter(
      (p) => p.name.toLowerCase().includes(q) && (!onlyInStock || p.stock > 0),
    )
  })
</script>
```

# $effect is the exit to the outside world

- Runs after mounting and whenever its dependencies change
- Only what the effect reads synchronously is a dependency
- The returned function cleans up

```
<script lang="ts">
  let { orderId }: { orderId: string } = $props()
  let status = $state<OrderStatus>('placed')

  $effect(() => {
    const events = new EventSource(`/orders/${orderId}/events`)
    events.onmessage = (event) => (status = event.data as OrderStatus)
    return () => events.close()
  })
</script>

<p>Status: {status}</p>
```

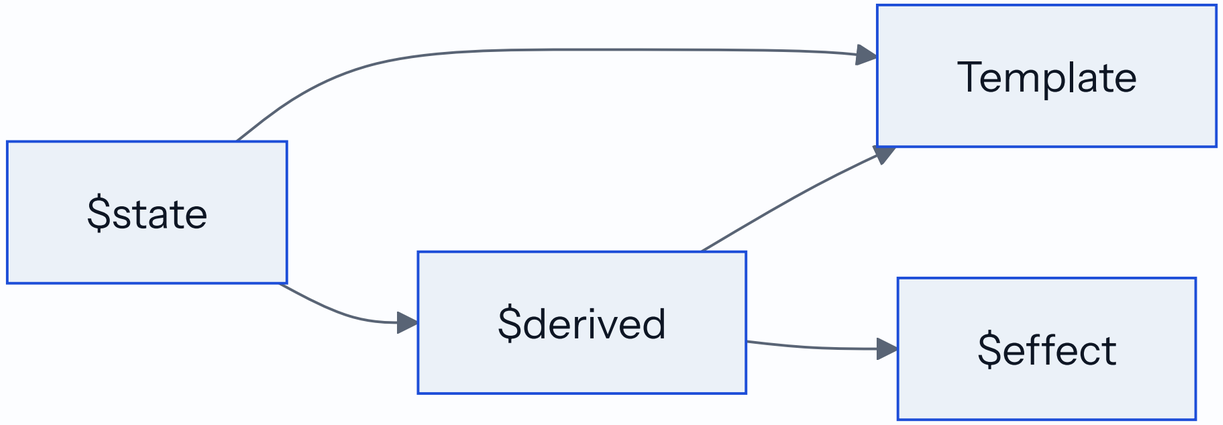
# The most common mistake with $effect

- An effect that only sets state is $derived in disguise
- It runs only after rendering, so the value is briefly stale
- $derived has no intermediate state

```
// not like this: update state via effect
let total = $state(0)
$effect(() => {
  total = cart.total
})

// instead: derive
let totalDerived = $derived(cart.total)
```

# State flows, effects stay at the edge



# Props, forms, snippets

Props in, callbacks out · One form, one schema, errors at the field · Snippets pass markup along like values

# Data comes in via $props

- A component reads props with let { … } = $props()
- You describe types as a regular TypeScript object
- Props are reactive and change with the caller

```
<script lang="ts">
  import type { Snippet } from 'svelte'
  import { euro, type Product } from '#lib/shop.js'

  type Props = {
    product: Product
    onadd: (product: Product) => void
    children?: Snippet
  }
  let { product, onadd, children }: Props = $props()
</script>
```

# Events are callbacks now

- DOM events like onclick are attributes like any other
- Component events are functions passed as props, such as onadd
- createEventDispatcher and on: are considered deprecated

```
<article>
  <h3>{product.name}</h3>
  <p>{euro(product.priceCents)}</p>
  {@render children?.()}
  <button onclick={() => onadd(product)} disabled={product.stock === 0}>
    In den Warenkorb
  </button>
</article>
```

# Keyed each matches by ID

- (product.id) makes the each block keyed
- Svelte matches entries by ID, not by position
- The caller decides what happens on click

```
<script lang="ts">
  import ProductCard from '#lib/components/ProductCard.svelte'
  import { getCart } from '#lib/cart.svelte.js'
  import type { PageProps } from './$types'

  let { data }: PageProps = $props()
  const cart = getCart()
</script>

{#each data.products as product (product.id)}
  <ProductCard {product} onadd={(p) => cart.add(p)} />
{/each}
```

# $bindable enables two-way binding

- Only props with $bindable may be bound by the caller via bind:
- The child reports changes without its own callback
- bind:value keeps the field and state in sync

```
<!-- QuantityInput.svelte -->
<script lang="ts">
  let { value = $bindable(1), max }: { value?: number; max: number } = $props()
</script>

<input type="number" min="1" {max} bind:value />

<!-- Aufrufer -->
<QuantityInput bind:value={quantity} max={3} />
```

# Data in, events out

- Read and type props with $props
- Report events as callback props
- Use bindings only for form fields with $bindable

# Bindings keep the inputs

- bind:value and bind:checked keep the field and $state in sync
- A field with type="number" returns a number
- Without an initial value, a select uses the first option

```
<script lang="ts">
  let productId = $state<string>()
  let quantity = $state<number | null>(1)
</script>

<select bind:value={productId}>
  {#each products as product (product.id)}
    <option value={product.id}>{product.name}</option>
  {/each}
</select>
<input type="number" min="1" bind:value={quantity} />
```

# The schema knows the rules

- The server form action uses the same module
- The type is created with z.infer
- strictObject rejects unknown fields

```
// src/lib/schema.ts
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>
```

# Validate first, then report

- onsubmit is an attribute like onclick
- preventDefault prevents the page reload
- safeParse returns data or issues without throwing

```
<script lang="ts">
  function handleSubmit(event: SubmitEvent) {
    event.preventDefault()
    const result = PlaceOrderInput.safeParse({
      customerId: 'c-1',
      items: [{ productId, quantity }],
    })
    if (!result.success) {
      errors = toFieldErrors(result.error.issues)
      return
    }
    errors = {}
    onorder(result.data)
  }
</script>

<form onsubmit={handleSubmit} novalidate>…</form>
```

# Errors belong at the field

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

```
<script lang="ts">
  let quantity = $state<number | null>(1)
  let errors = $state<Record<string, string>>({})
  let quantityError = $derived(errors['items.0.quantity'])
</script>

<label for="quantity">Menge</label>
<input id="quantity" type="number" min="1" bind:value={quantity}
  aria-invalid={quantityError ? 'true' : undefined}
  aria-describedby={quantityError ? 'quantity-error' : undefined} />
{#if quantityError}<p id="quantity-error">{quantityError}</p>{/if}
```

# Good forms do not scold

- Validate on submit, then optionally live per field
- Show errors at the field, not in a toast
- Always make the server validate for itself

# Snippets replace slots

- A snippet is reusable markup with parameters
- Content between the tags arrives as the prop children
- Named snippets are simply additional props

# Render children

- The card accepts children?: Snippet
- {@render children?.()} shows the content or nothing
- The caller writes the content between the tags

```
<!-- ProductCard.svelte -->
<article>
  <h3>{product.name}</h3>
  {@render children?.()}
</article>

<!-- Aufrufer -->
<ProductCard {product} onadd={(p) => cart.add(p)}>
  {#if product.stock === 0}<small>Ausverkauft</small>{/if}
</ProductCard>
```

# Snippets with parameters

- Snippet<[Product]> describes the parameter
- The table defines the structure
- {:else} in the each block shows the empty state

```
<script lang="ts">
  import type { Snippet } from 'svelte'
  import type { Product } from '#lib/shop.js'

  type Props = { products: Product[]; row: Snippet<[Product]>; empty?: Snippet }
  let { products, row, empty }: Props = $props()
</script>

<table><tbody>
  {#each products as product (product.id)}
    <tr>{@render row(product)}</tr>
  {:else}
    <tr><td>{@render empty?.()}</td></tr>
  {/each}
</tbody></table>
```

# The caller provides the markup

- Snippets between the tags become props with the same names
- The parameter replaces the earlier let:
- Each page fills the same table differently

```
<ProductTable {products}>
  {#snippet row(product: Product)}
    <td>{product.name}</td>
    <td>{euro(product.priceCents)}</td>
  {/snippet}
  {#snippet empty()}Keine Produkte gefunden{/snippet}
</ProductTable>
```

# Structure inside, content outside

- Build layout components with external content
- Group repeated markup in one file
- Pass data outward by parameter

# Shared state and SvelteKit through production

Where does the cart live that every page needs? · Routes are files, data comes through load · Write with form actions, even without JavaScript · Test Svelte the way users use it · Which adapter takes the shop to production?

# Props through every layer do not scale

- Product list, cart, and header need the same cart
- Passing props through every layer quickly gets confusing
- Runes also work outside components

# The cart as a class

- State, derived values, and methods in one place
- Never reassign exported $state; only change properties
- Testable without a component

```
// src/lib/cart.svelte.ts
export class Cart {
  items = $state<OrderItem[]>([])
  total = $derived(totalCents(this.items))
  count = $derived(this.items.reduce((sum, item) => sum + item.quantity, 0))

  remove(productId: ProductId) {
    this.items = this.items.filter((item) => item.productId !== productId)
  }
}
```

# Context sets once and reads everywhere

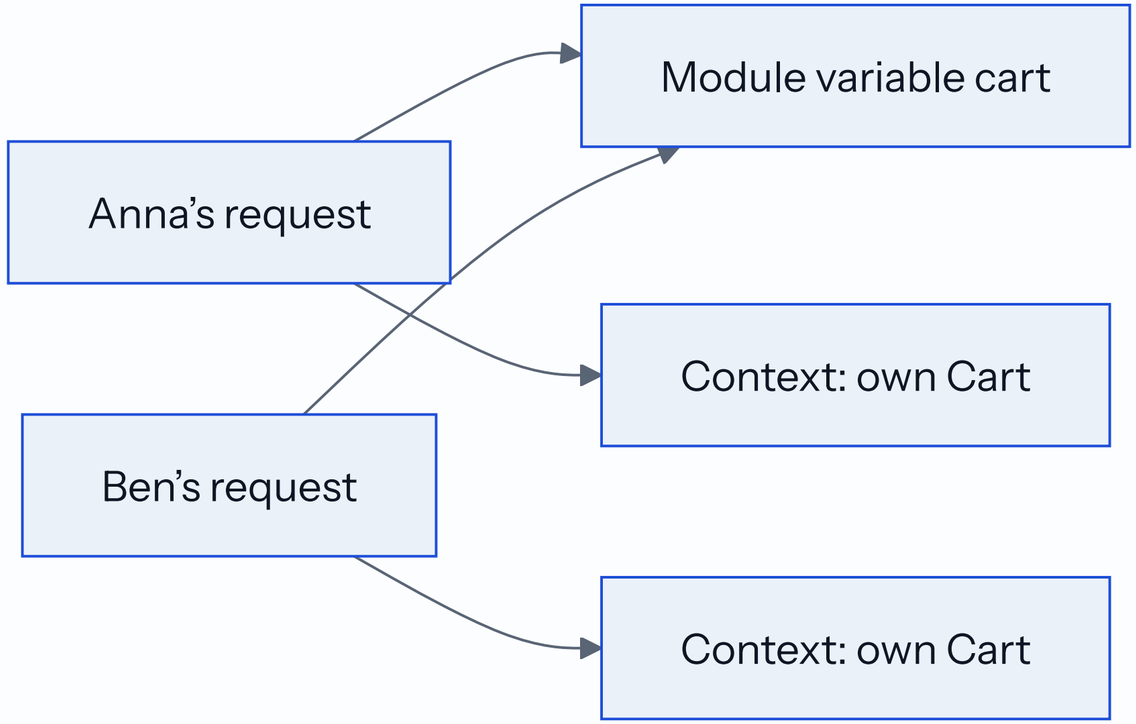
- createContext returns a typed pair of get and set
- The layout calls set; every component below it calls get
- get throws if no ancestor has called set

```
<!-- src/routes/+layout.svelte -->
<script lang="ts">
  import { Cart, setCart } from '#lib/cart.svelte.js'
  import type { LayoutProps } from './$types'

  let { children }: LayoutProps = $props()
  setCart(new Cart())
</script>

{@render children()}
```

# Module variables share all requests



# Stores remain for streams

- Stores are not obsolete; $store continues to work
- readable fits browser events and external sources
- fromStore builds a bridge from Stores to Runes

```
import { readable } from 'svelte/store'

export const online = readable(true, (set) => {
  const update = () => set(navigator.onLine)
  update()
  window.addEventListener('online', update)
  window.addEventListener('offline', update)
  return () => {
    window.removeEventListener('online', update)
    window.removeEventListener('offline', update)
  }
})
```

# Every state has its place

- Local: $state in the component
- Shared per person: class in .svelte.ts plus Context
- From outside: readable Store or class with $effect

# SvelteKit turns components into an application

- File-based routing with layouts
- server-side rendering and hydration
- Adapters for Node, serverless, and static pages

```
src/routes/
  +layout.svelte              cart via context
  products/
    +page.server.ts           load: products
    +page.svelte              product list
  orders/
    +page.server.ts           load and Form Action
    +page.svelte              form and overview
    [id]/+page.server.ts      load: one order or 404
```

# SvelteKit 3 configures in the Vite plugin

- No more svelte.config.js, everything is in vite.config.ts
- #lib instead of $lib, a subpath import from package.json
- $app/state instead of $app/stores

```
// vite.config.ts
import adapter from '@sveltejs/adapter-node'
import { sveltekit } from '@sveltejs/kit/vite'
import { defineConfig } from 'vite'

export default defineConfig({
  plugins: [sveltekit({ adapter: adapter() })],
})
```

# load provides the page data

- +page.server.ts runs only on the server
- +page.ts runs on the server and client
- The result comes as the data prop, typed through ./$types

```
// src/routes/products/+page.server.ts
import { db } from '#lib/server/db.js'
import type { PageServerLoad } from './$types'

export const load: PageServerLoad = async () => {
  return { products: await db.products() }
}
```

# What does not exist becomes a 404

- params comes from the folder name [id]
- error throws, so no return is needed
- The second parameter is a string

```
// src/routes/orders/[id]/+page.server.ts
import { error } from '@sveltejs/kit'
import { db } from '#lib/server/db.js'
import type { PageServerLoad } from './$types'

export const load: PageServerLoad = async ({ params }) => {
  const order = await db.order(params.id)
  if (!order) error(404, 'Diese Bestellung gibt es nicht. Wähle eine aus der Bestellliste.')
  return { order }
}
```

# page from $app/state is reactive

- page is a reactive object, not a store
- page.params, page.url, and page.data are available everywhere
- $app/stores no longer exists in SvelteKit 3

```
<script lang="ts">
  import { page } from '$app/state'
  import OrderStatus from '#lib/components/OrderStatus.svelte'
  import type { PageProps } from './$types'

  let { data }: PageProps = $props()
</script>

<h1>Bestellung {page.params.id}</h1>
<OrderStatus orderId={data.order.id} />
```

# Form actions accept POST requests

- A form with method="POST" sends data to actions
- It works without JavaScript, and use:enhance improves it
- Report errors with fail, success for example with redirect

# The action validates with the same schema

- FormData returns text, the schema expects numbers
- PlaceOrderInput is the same schema as in the browser
- The server reads prices itself; the client sends only IDs

```
export const actions = {
  default: async ({ request }) => {
    const form = await request.formData()
    const input = PlaceOrderInput.safeParse({
      customerId: form.get('customerId'),
      items: [{ productId: form.get('productId'), quantity: Number(form.get('quantity')) }],
    })
    if (!input.success) return fail(400, { error: 'Wähle ein Produkt und eine Menge ab 1.' })
    // then: create order
  },
} satisfies Actions
```

# fail for expected cases, redirect for success

- return fail(409, ...) reports missing stock
- redirect(303, ...) throws and redirects
- error is reserved for data that does not exist

```
// in the action, after validation
const order = await db.placeOrder(input.data)
if (!order) return fail(409, { error: 'Zu wenig auf Lager. Wähle eine kleinere Menge oder ein anderes Produkt.' })
redirect(303, `/orders/${order.id}`)
```

# fail appears as form on the page

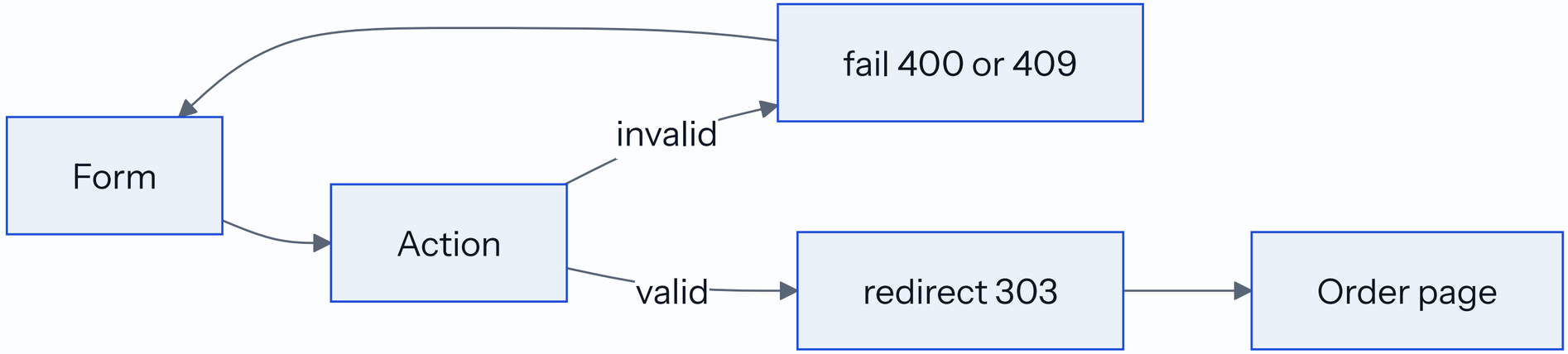
- The page receives the data from fail as the form prop
- role="alert" lets screen readers announce the message
- use:enhance keeps the status from fail

```
<script lang="ts">
  import { enhance } from '$app/forms'
  import type { PageProps } from './$types'

  let { form }: PageProps = $props()
</script>

<form method="POST" use:enhance>
  <input type="hidden" name="customerId" value="c-1" />
  <label>Produkt <input name="productId" value="p-1" /></label>
  <label>Menge <input name="quantity" type="number" min="1" value="1" /></label>
  <button>Bestellen</button>
  {#if form?.error}<p role="alert">{form.error}</p>{/if}
</form>
```

# The path of an order



# Three levels, three tools

- Test domain logic like totalCents and Cart without a component
- Test components with Vitest, jsdom, and Testing Library
- Call load and Form Actions as normal functions

# You test Runes without a component

- The test file ends in .svelte.test.ts, so it understands Runes
- Cart calculates the total and count itself
- No rendering, no DOM, fast tests

```
// src/lib/cart.svelte.test.ts
import { expect, it } from 'vitest'
import { Cart } from '#lib/cart.svelte.js'

it('zählt Menge und Summe mit', () => {
  const cart = new Cart()
  cart.add(hammer)
  cart.add(hammer)
  expect(cart.count).toBe(2)
  expect(cart.total).toBe(2 * hammer.priceCents)
})
```

# Vitest knows Svelte through the Vite plugin

- The same vite.config.ts for app and tests
- jsdom as the environment for component tests
- resolve.conditions browser loads the browser variant of Svelte

```
// vite.config.ts
export default defineConfig({
  plugins: [sveltekit({ adapter: adapter() })],
  test: {
    projects: [
      { extends: true, test: { name: 'server', environment: 'node',
        include: ['src/**/*.test.ts'], exclude: ['src/**/*.svelte.test.ts'] } },
      { extends: true, resolve: { conditions: ['browser'] },
        test: { name: 'client', environment: 'jsdom',
          include: ['src/**/*.svelte.test.ts'] } },
    ],
  },
})
```

# Test what users see and do

- render mounts the component with props
- getByLabelText and getByRole instead of CSS selectors
- userEvent types and clicks like a person

```
it('bestellt das gewählte Produkt', async () => {
  const user = userEvent.setup()
  const onorder = vi.fn()
  render(CheckoutForm, { products: sampleProducts, onorder })

  await user.selectOptions(screen.getByLabelText('Produkt'), 'Akkuschrauber')
  await user.clear(screen.getByLabelText('Menge'))
  await user.type(screen.getByLabelText('Menge'), '2')
  await user.click(screen.getByRole('button', { name: 'Bestellen' }))

  expect(onorder).toHaveBeenCalledWith({
    customerId: 'c-1', items: [{ productId: 'p-4', quantity: 2 }],
  })
})
```

# Tests that survive refactoring

- Check domain logic and classes without a UI
- Use components by role and label
- Observe callback props with vi.fn()

# The adapter defines the target platform

- adapter-node creates a standalone Node server
- adapter-static creates only files for a CDN
- adapter-auto detects known platforms at build time

# npm run build, then node build

- The output is in build/ and needs only production packages
- PORT sets the port; the default is 3000
- build/handler.js contains the handler for custom servers

```
npm run build
node build
PORT=8080 node build
```

# Express 5 as an option in front

- Custom routes like /healthz come before the handler
- app.use(handler) forwards everything else to SvelteKit
- Node.js 26 starts server.ts without a build step

```
// server.ts
import express from 'express'
import { handler } from './build/handler.js'

const app = express()
app.disable('x-powered-by')
app.get('/healthz', (_req, res) => {
  res.json({ ok: true })
})
app.use(handler)

app.listen(Number(process.env.PORT ?? 3000))
```

# Behind a proxy, the origin matters

- Form Actions reject foreign origins with 403
- adapter-node assumes https and the Host header
- PROTOCOL_HEADER and HOST_HEADER read X-Forwarded headers

```
PROTOCOL_HEADER=x-forwarded-proto \
HOST_HEADER=x-forwarded-host \
PORT=3000 node build
```

# One image, two stages

- The first stage builds with all packages
- The second contains only build/ and production packages
- The container runs as the node user, not as root

```
FROM node:26-slim AS build
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:26-slim
WORKDIR /app
ENV NODE_ENV=production
COPY package.json package-lock.json ./
RUN npm ci --omit=dev
COPY --from=build /app/build ./build
USER node
CMD ["node", "build"]
```

# Questions?

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

Workshops für Teams, die Architekturentscheidungen selbst treffen