








# VOM CLIENT ZUM SERVER: WIE TANSTACK START DIE LÜCKE SCHLIESST UND LAUFZEITFEHLER ELIMINIERT SOWIE ENTWICKLUNG BESCHLEUNIGT

Leipzig.js Usergroup

# DIE MISSION HEUTE



## ZIEL: MODERNISIERUNG MEINER PERSÖNLICHEN BUDGET-APP

-  Ausgaben und Einnahmen tracken
-  Kategorisierte Transaktionen
-  User Authentication
-  Charts und Reports
-  PWA mit Offline-Support

Repository: [github.com/nogo/budget](https://github.com/nogo/budget)

# DAS EWIGE PROBLEM

## FRONTEND 🤝 BACKEND SYNCHRONISATION

```
1 // backend/api/transactions.ts
2 interface Transaction {
3   id: number
4   amount: number
5   categoryId: string // Ups, war das nicht number?
6   createdAt: Date    // Oder string?
7 }
8
9 // frontend/types/transaction.ts
10 interface Transaction {
11   id: string          // 🤖 Mismatch!
12   amount: number
13   category_id: number // 🤖 Andere Schreibweise!
14   created_at: string  // 🤖 Anderer Typ!
15 }
16
17 // Resultat: ✨ Laufzeitfehler in Produktion
```


# DAS EWIGE PROBLEM

## FRONTEND 🤝 BACKEND SYNCHRONISATION

```
1 // backend/api/transactions.ts
2 interface Transaction {
3   id: number
4   amount: number
5   categoryId: string // Ups, war das nicht number?
6   createdAt: Date    // Oder string?
7 }
8
9 // frontend/types/transaction.ts
10 interface Transaction {
11   id: string          // 🤖 Mismatch!
12   amount: number
13   category_id: number // 🤖 Andere Schreibweise!
14   created_at: string  // 🤖 Anderer Typ!
15 }
16
17 // Resultat: 🌟 Laufzeitfehler in Produktion
```

# DER KLASSISCHE WORKFLOW

## WIE WIR ES BISHER GEMACHT HABEN

1. **Backend-Entwickler:** Erstellt API Endpoint
2. **Dokumentation:** OpenAPI/Swagger (oft outdated)
3. **Frontend-Entwickler:** Rät die Datenstruktur
4. **Testing:** Findet Fehler zur Laufzeit
5. **Fix:** Manuelle Synchronisation
6. **Repeat:** Bei jeder Änderung 



**Zeitverlust: ~30% der Entwicklungszeit**

# WAS ICH MIR ANGESCHAUT HABE

## NEXT.JS APP ROUTER

```
// app/transactions/page.tsx
export default async function Page() {
  'use server' // 🤔 Server Component

  const data = await db.query(...)

  return (
    <div>
      <Button onClick={handleClick}> { /* ✖ Geht nicht! */ }
      Click me
    </Button>
  </div>
)
}
```

**Problem:** Ständiges Jonglieren zwischen 'use client' und 'use server'

# NEXT.JS: DIE DIREKTIVEN-HÖLLE

```
'use client' // Jetzt bin ich im Browser

import { ServerComponent } from './server' // ✗ Geht nicht!

export function ClientComponent() {
  const handleSubmit = async () => {
    'use server' // ✗ Nicht hier erlaubt!
    await db.save(...)
  }





  return <form onSubmit={handleSubmit}>...</form>
}
```

- Zu viele mentale Modelle
- Unnatürliche Trennung
- Fehleranfällig

# DIE ENTDECKUNG

## TANSTACK START - EIN ANDERER ANSATZ

*"Client-First, Full-Stack Capable"*

-  **Philosophie:** Der Client steht im Mittelpunkt
-  **Kernfeature:** Ende-zu-Ende Typsicherheit
-  **Technologie:** Server Functions ohne Direktiven
-  **DX:** TypeScript macht die Synchronisation



# WAS IST TANSTACK START?

## DIE TECHNISCHEN FACTS

- **Full-Stack Framework** für React (und Solid)
- **Built on:** Vite + Nitro + TanStack Router
- **Status:** Beta (aber produktionsreif)
- **USP:** Compile-Time Type Safety überall

```
// Eine Datei, zwei Welten - nahtlos verbunden
import { createServerFn } from '@tanstack/start'

const getTransactions = createServerFn()
  .handler(async () => {
    return await db.transaction.findMany() // Server
  })

// Client - TypeScript kennt automatisch die Struktur!
const transactions = await getTransactions()
```

# DAS TEAM DAHINTER



**TANNER LINSLEY**

Gründer & Visionär



**MANUEL SCHILLER**

Core Maintainer



**SEAN CASSIERE**

Router Experte



**BIRK SKYUM**

Start Contributor

# DIE TANSTACK PHILOSOPHIE

## "COMPOSABILITY IS THE STRATEGY"

1. **Framework-Agnostik:** Core-Logic unabhängig vom UI
2. **Headless UI:** Logik ohne vordefinierte Komponenten
3. **Modular:** Nutze nur was du brauchst

*"UI-Frameworks sind nur Rendering-Tools, nicht die Basis deiner App"*  
— Tanner Linsley

# SERVER FUNCTIONS DEEP DIVE

## DAS POWER-FEATURE

```
1 // app/features/transactions.ts
2 import { createServerFn } from '@tanstack/start'
3 import { db } from './db'
4
5 // Server Function definieren
6 export const createTransaction = createServerFn({ method: 'POST'})
7   .validator(CreateTemplateSchema)
8   .handler(async (data: { amount: number; categoryId: number }) => {
9     // Dieser Code läuft NUR auf dem Server
10     const transaction = await db.transaction.create({
11       data
12     })
13     return transaction
14   })
```

# CLIENT-SEITE: TYPISCHERHEIT

```
1 // app/components/transaction-form.tsx
2 import { createTransaction } from '~/service/transactions'
3
4 export function TransactionForm() {
5   const handleSubmit = async (e: FormEvent) => {
6     // TypeScript kennt EXAKT die Parameter und Return-Types!
7     const result = await createTransaction({
8       amount: 100,
9       categoryId: 1
10      // typo: "test" ❌ TypeScript Error zur Compile-Zeit!
11    })
12
13    // result ist vollständig typisiert
14    console.log(result.id, result.createdAt)
15  }
16
17  return <form onSubmit={handleSubmit}>...</form>
```

**Kein manuelles Typing. Keine API-Dokumentation. Es**

**funktioniert einfach.**

# ROUTING MIT TYPESICHERHEIT

## FEHLER ZUR COMPILE-ZEIT STATT RUNTIME

```
1 // app/routes/transactions.$id.tsx
2 import { createFileRoute } from '@tanstack/react-router'
3
4 export const Route = createFileRoute('/transactions/$id')({
5   loader: async ({ params }) => {
6     // params.id ist automatisch string und required!
7     return await getTransaction(params.id)
8   },
9   component: TransactionDetail
10 })
11
12 // Irgendwo anders im Code:
13 <Link to="/transactions/$id" params={{ id: 123 }}>
14   {/* ✖ TypeScript Error: id must be string */}
15 </Link>
```

# ROUTING MIT TYPESICHERHEIT

## FEHLER ZUR COMPILE-ZEIT STATT RUNTIME

```
1 // app/routes/transactions.$id.tsx
2 import { createFileRoute } from '@tanstack/react-router'
3
4 export const Route = createFileRoute('/transactions/$id')({
5   loader: async ({ params }) => {
6     // params.id ist automatisch string und required!
7     return await getTransaction(params.id)
8   },
9   component: TransactionDetail
10 })
11
12 // Irgendwo anders im Code:
13 <Link to="/transactions/$id" params={{ id: 123 }}>
14   {/* ❌ TypeScript Error: id must be string */}
15 </Link>
```



# STREAMING SSR

## PERFORMANCE OHNE KOMPROMISSE

```
1 export const Route = createFileRoute('/dashboard')({
2   loader: async () => {
3     // Kritische Daten - sofort laden
4     const user = await getUser()
5
6     // Langsame Daten - streamen
7     const analyticsPromise = getAnalytics() // Nicht awaiten!
8
9     return {
10      user,
11      analytics: defer(analyticsPromise) // Wird gestreamt
12    }
13  }
14 })
```

**Resultat:** First Paint in <100ms, Rest wird nachgeladen

# STREAMING SSR

## PERFORMANCE OHNE KOMPROMISSE

```
1 export const Route = createFileRoute('/dashboard')({
2   loader: async () => {
3     // Kritische Daten - sofort laden
4     const user = await getUser()
5
6     // Langsame Daten - streamen
7     const analyticsPromise = getAnalytics() // Nicht awaiten!
8
9     return {
10      user,
11      analytics: defer(analyticsPromise) // Wird gestreamt
12    }
13  }
14 })
```

**Resultat:** First Paint in <100ms, Rest wird nachgeladen

# STREAMING SSR

## PERFORMANCE OHNE KOMPROMISSE

```
1 export const Route = createFileRoute('/dashboard')({
2   loader: async () => {
3     // Kritische Daten - sofort laden
4     const user = await getUser()
5
6     // Langsame Daten - streamen
7     const analyticsPromise = getAnalytics() // Nicht awaiten!
8
9     return {
10      user,
11      analytics: defer(analyticsPromise) // Wird gestreamt
12    }
13  }
14 })
```

**Resultat:** First Paint in <100ms, Rest wird nachgeladen

# ISOMORPHE LOADER

## EIN CODE, ZWEI WELTEN

```
const Route = createFileRoute('/transactions')({  
  loader: async () => {  
    // Dieser Code läuft:  
    // 1. Auf dem Server beim ersten Laden (SSR)  
    // 2. Auf dem Client bei Navigation (SPA)  
    // Automatisch optimiert, kein doppeltes Fetching!  
  
    return await getTransactions()  
  }  
})
```

## VERGLEICH NEXT.JS:

- `getServerSideProps` (nur Server)
- `useEffect` + `fetch` (nur Client)
- Doppelte Logik, doppelte Fehlerquellen

# TRADITIONELLE ARCHITEKTUR

```
frontend/  
├── api/  
│   └── client.ts  
├── types/  
│   ├── transaction.ts  
│   └── category.ts  
└── hooks/  
    └── useTransactions.ts
```

```
backend/  
├── routes/  
│   ├── transactions.js  
│   └── categories.js  
├── models/  
│   └── ...  
└── db/  
    └── queries.js
```

## PROBLEME:

- Types manuell synchronisieren
- API-Client pflegen
- Fehler erst zur Laufzeit

# EINE EINHEITLICHE CODEBASIS

```
app/
├── routes/                # Seiten mit Loadern
│   ├── __root.tsx
│   ├── index.tsx
│   └── transactions.tsx
├── services/             # Server Functions & Logik
│   ├── transactions.ts # DB-Queries & Business Logic
│   └── categories.ts
└── db/
    └── schema.prisma     # Single Source of Truth
```

**Ein Projekt. Eine Typdefinition. Null Synchronisation.**

# CODE-BEISPIEL: TRANSACTION FEATURE

```
1 // app/features/transactions.ts
2 import { createServerFn } from '@tanstack/start'
3 import { prisma } from '~/db'
4
5 export const getTransactions = createServerFn()
6   .handler(async () => {
7     return prisma.transaction.findMany({
8       include: { category: true },
9       orderBy: { date: 'desc' }
10    })
11  })
12
13 export const createTransaction = createServerFn({ method: 'POST' })
14   .middleware([userRequiredMiddleware])
15   .validator(TransactionSchema)
16   .handler(async ({ data: transactionData }) => {
17     // Validation, Auth-Check, etc.
```

# CODE-BEISPIEL: TRANSACTION FEATURE

```
1 // app/features/transactions.ts
2 import { createServerFn } from '@tanstack/start'
3 import { prisma } from '~/db'
4
5 export const getTransactions = createServerFn()
6   .handler(async () => {
7     return prisma.transaction.findMany({
8       include: { category: true },
9       orderBy: { date: 'desc' }
10    })
11  })
12
13 export const createTransaction = createServerFn({ method: 'POST' })
14   .middleware([userRequiredMiddleware])
15   .validator(TransactionSchema)
16   .handler(async ({ data: transactionData }) => {
17     // Validation, Auth-Check, etc.
```



# CODE-BEISPIEL: TRANSACTION FEATURE

```
1 // app/features/transactions.ts
2 import { createServerFn } from '@tanstack/start'
3 import { prisma } from '~/db'
4
5 export const getTransactions = createServerFn()
6   .handler(async () => {
7     return prisma.transaction.findMany({
8       include: { category: true },
9       orderBy: { date: 'desc' }
10    })
11  })
12
13 export const createTransaction = createServerFn({ method: 'POST' })
14   .middleware([userRequiredMiddleware])
15   .validator(TransactionSchema)
16   .handler(async ({ data: transactionData }) => {
17     // Validation, Auth-Check, etc.
```

# DIE ROUTE DAZU

```
1 // app/routes/transactions.tsx
2 import { createFileRoute, useLoaderData } from '@tanstack/react-router'
3 import { getTransactions, createTransaction } from '~/features/transactions'
4
5 export const Route = createFileRoute('/transactions')({
6   loader: () => getTransactions(),
7   component: TransactionsPage
8 })
9
10 function TransactionsPage() {
11   const transactions = Route.useLoaderData()
12   // transactions ist vollständig typisiert!
13
14   const handleCreate = async (data: FormData) => {
15     await createTransaction({...})
16     // Automatisches Revalidation nach Mutation
17   }
```

# DIE ROUTE DAZU

```
1 // app/routes/transactions.tsx
2 import { createFileRoute, useLoaderData } from '@tanstack/react-router'
3 import { getTransactions, createTransaction } from '~/features/transactions'
4
5 export const Route = createFileRoute('/transactions')({
6   loader: () => getTransactions(),
7   component: TransactionsPage
8 })
9
10 function TransactionsPage() {
11   const transactions = Route.useLoaderData()
12   // transactions ist vollständig typisiert!
13
14   const handleCreate = async (data: FormData) => {
15     await createTransaction({...})
16     // Automatisches Revalidation nach Mutation
17   }
```

# DIE ROUTE DAZU

```
1 // app/routes/transactions.tsx
2 import { createFileRoute, useLoaderData } from '@tanstack/react-router'
3 import { getTransactions, createTransaction } from '~/features/transactions'
4
5 export const Route = createFileRoute('/transactions')({
6   loader: () => getTransactions(),
7   component: TransactionsPage
8 })
9
10 function TransactionsPage() {
11   const transactions = Route.useLoaderData()
12   // transactions ist vollständig typisiert!
13
14   const handleCreate = async (data: FormData) => {
15     await createTransaction({...})
16     // Automatisches Revalidation nach Mutation
17   }
```

# DIE VORTEILE IN DER PRAXIS

## WAS HAT SICH VERBESSERT?

### Vorher

2 Codebasen

Manuelle Type-Sync

Runtime Errors

API Dokumentation

Complex State Sync

### Nachher mit TanStack Start

1 Monorepo

Automatische Types

Compile-Time Errors

Code ist Dokumentation

Eingebautes Caching



**Entwicklungszeit: -40% für neue Features**

# TANSTACK QUERY INTEGRATION

## DAS PERFEKTE DUO

```
// Server Function
const updateTransaction = createServerFn({ method: 'POST' })
  .validator(UpdateSchema)
  .handler(async ({ data }) => {
    return prisma.transaction.update({...})
  })

// Client mit TanStack Query
const mutation = useMutation({
  mutationFn: updateTransaction,
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ['transactions'] })
  }
})
```

**Optimistic Updates, Caching, Retry - alles built-in!**

# TRADE-OFFS & LIMITIERUNGEN

## NICHTS IST PERFEKT



### VORTEILE:

- Unschlagbare Typsicherheit
- Fantastische DX
- Hohe Flexibilität
- Streaming SSR



### NACHTEILE:

- **Beta Status** - Breaking Changes möglich
- **Kleinere Community** als Next.js/Remix
- **Weniger Tutorials/Beispiele** verfügbar

**LIVE DEMO**



# GETTING STARTED

## IN 5 MINUTEN ZUM ERSTEN PROJEKT

```
# Projekt erstellen
npx create-start-app@latest my-app

# Dependencies installieren
cd my-app && bun install

# Development Server
bun run dev
```

### STARTER TEMPLATES:

- `basic` - Minimales Setup
- `with-auth` - Mit Clerk
- `kitchen-sink` - Alle Features

# RESOURCES & COMMUNITY

## WO IHR MEHR ERFAHRT



### DOKUMENTATION

- [tanstack.com/start](https://tanstack.com/start)
- Exzellente Guides & API Docs

# KEY TAKEAWAYS

## WAS IHR MITNEHMEN SOLLTET

1. **Type Safety Matters** - Fehler zur Compile-Zeit finden
2. **Client-First  $\neq$  Client-Only** - Das Beste aus beiden Welten
3. **Server Functions** - Die Zukunft der Full-Stack DX
4. **Beta heißt nicht instabil** - Produktionsreif mit Vorsicht
5. **Die richtige Wahl** - Nicht jedes Tool für jedes Problem

*"The best framework is the one that solves YOUR problems"*

# FRAGEN?

## LET'S DISCUSS!



**DANILO KÜHN**

I'm a developer

## DANKE FÜR EURE AUFMERKSAMKEIT!



# Happy Coding mit TanStack Start!