FROM CLIENT TO SERVER: HOW TANSTACK START CLOSES THE GAP

AND ELIMINATES RUNTIME ERRORS AS WELL AS ACCELERATES DEVELOPMENT

Leipzig.js Usergroup

TODAY'S MISSION

🎯 GOAL: MODERNIZING MY PERSONAL BUDGET APP

- 📊 Track expenses and income
- 📈 Categorized transactions
- 🔐 User authentication
- 📊 Charts and reports
- 📱 PWA with offline support

Repository: github.com/nogo/budget

THE ETERNAL PROBLEM

FRONTEND 🤝 BACKEND SYNCHRONIZATION

```
// backend/api/transactions.ts
interface Transaction {
  id: number
  amount: number
  categoryId: string  // Oops, wasn't this number?
  createdAt: Date     // Or string?
}

// frontend/types/transaction.ts
interface Transaction {
  id: string          // 😱 Mismatch!
  amount: number
  category_id: number // 😱 Different naming!
  created_at: string  // 😱 Different type!
}

// Result: 💥 Runtime errors in production
```

THE ETERNAL PROBLEM

FRONTEND 🤝 BACKEND SYNCHRONIZATION

```typescript
// backend/api/transactions.ts
interface Transaction {
  id: number
  amount: number
  categoryId: string  // Oops, wasn't this number?
  createdAt: Date     // Or string?
}

// frontend/types/transaction.ts
interface Transaction {
  id: string          // 😱 Mismatch!
  amount: number
  category_id: number // 😱 Different naming!
  created_at: string  // 😱 Different type!
}

// Result: 💥 Runtime errors in production
```

THE CLASSIC WORKFLOW

HOW WE'VE BEEN DOING IT

1. **Backend Developer:** Creates API endpoint
2. **Documentation:** OpenAPI/Swagger (often outdated)
3. **Frontend Developer:** Guesses the data structure
4. **Testing:** Finds errors at runtime
5. **Fix:** Manual synchronization
6. **Repeat:** On every change 🔄

⏱️ **Time loss:** ~30% of development time

WHAT I LOOKED AT

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}> {/* ❌ Doesn't work! */}
        Click me
      </Button>
    </div>
  )
}
```

Problem: Constant juggling between 'use client' and 'use server'

NEXT.JS: DIRECTIVE HELL

```
'use client'  // Now I'm in the browser

import { ServerComponent } from './server' // ❌ Doesn't work!

export function ClientComponent() {
  const handleSubmit = async () => {
    'use server'  // ❌ Not allowed here!
    await db.save(...)
  }

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

- Too many mental models
- Unnatural separation
- Error-prone

THE DISCOVERY

TANSTACK START - A DIFFERENT APPROACH

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

- 🎯 **Philosophy:** The client is at the center
- 🔒 **Core Feature:** End-to-end type safety
- 🚀 **Technology:** Server Functions without directives
- ⚡ **DX:** TypeScript handles the synchronization

WHAT IS TANSTACK START?

THE TECHNICAL FACTS

- **Full-Stack Framework** for React (and Solid)
- **Built on:** Vite + Nitro + TanStack Router
- **Status:** Beta (but production-ready)
- **USP:** Compile-Time Type Safety everywhere

```
// One file, two worlds - seamlessly connected
import { createServerFn } from '@tanstack/start'

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

// Client - TypeScript automatically knows the structure!
const transactions = await getTransactions()
```

THE TEAM BEHIND IT

TANNER LINSLEY

Founder & Visionary

MANUEL SCHILLER

Core Maintainer

SEAN CASSIERE

Router Expert

BIRK SKYUM

Start Contributor

THE TANSTACK PHILOSOPHY

"COMPOSABILITY IS THE STRATEGY"

1. **Framework-Agnostic:** Core logic independent from UI
2. **Headless UI:** Logic without predefined components
3. **Modular:** Use only what you need

> *"UI frameworks are just rendering tools, not the basis of your app"*
> *— Tanner Linsley*

SERVER FUNCTIONS DEEP DIVE

THE POWERFUL FEATURE

```ts
// app/features/transactions.ts
import { createServerFn } from '@tanstack/start'
import { db } from './db'

// Define server function
export const createTransaction = createServerFn({ method: 'POST'})
  .validator(CreateTemplateSchema)
  .handler(async (data: { amount: number; categoryId: number }) => {
    // This code runs ONLY on the server
    const transaction = await db.transaction.create({
      data
    })
    return transaction
  })
```

CLIENT-SIDE: TYPE SAFETY

```tsx
// app/components/transaction-form.tsx
import { createTransaction } from '~/service/transactions'

export function TransactionForm() {
  const handleSubmit = async (e: FormEvent) => {
    // TypeScript knows EXACTLY the parameters and return types!
    const result = await createTransaction({
      amount: 100,
      categoryId: 1
      // typo: "test" ❌ TypeScript Error at compile-time!
    })

    // result is fully typed
    console.log(result.id, result.createdAt)
  }

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

No manual typing. No API documentation. It just works.

ROUTING WITH TYPE SAFETY

ERRORS AT COMPILE-TIME INSTEAD OF RUNTIME

```
// app/routes/transactions.$id.tsx
import { createFileRoute } from '@tanstack/react-router'

export const Route = createFileRoute('/transactions/$id')({
  loader: async ({ params }) => {
    // params.id is automatically string and required!
    return await getTransaction(params.id)
  },
  component: TransactionDetail
})

// Somewhere else in the code:
<Link to="/transactions/$id" params={{ id: 123 }}>
  {/* ❌ TypeScript Error: id must be string */}
</Link>
```

ROUTING WITH TYPE SAFETY

ERRORS AT COMPILE-TIME INSTEAD OF RUNTIME

```
// app/routes/transactions.$id.tsx
import { createFileRoute } from '@tanstack/react-router'

export const Route = createFileRoute('/transactions/$id')({
  loader: async ({ params }) => {
    // params.id is automatically string and required!
    return await getTransaction(params.id)
  },
  component: TransactionDetail
})

// Somewhere else in the code:
<Link to="/transactions/$id" params={{ id: 123 }}>
  {/* ❌ TypeScript Error: id must be string */}
</Link>
```

STREAMING SSR

PERFORMANCE WITHOUT COMPROMISES

```
export const Route = createFileRoute('/dashboard')({
  loader: async () => {
    // Critical data - load immediately
    const user = await getUser()

    // Slow data - stream it
    const analyticsPromise = getAnalytics() // Don't await!

    return {
      user,
      analytics: defer(analyticsPromise) // Will be streamed
    }
  }
})
```

Result: First Paint in <100ms, rest loads after

STREAMING SSR

PERFORMANCE WITHOUT COMPROMISES

```
export const Route = createFileRoute('/dashboard')({
  loader: async () => {
    // Critical data - load immediately
    const user = await getUser()

    // Slow data - stream it
    const analyticsPromise = getAnalytics() // Don't await!

    return {
      user,
      analytics: defer(analyticsPromise) // Will be streamed
    }
  }
})
```

Result: First Paint in <100ms, rest loads after

STREAMING SSR

PERFORMANCE WITHOUT COMPROMISES

```
export const Route = createFileRoute('/dashboard')({
  loader: async () => {
    // Critical data - load immediately
    const user = await getUser()

    // Slow data - stream it
    const analyticsPromise = getAnalytics() // Don't await!

    return {
      user,
      analytics: defer(analyticsPromise) // Will be streamed
    }
  }
})
```

Result: First Paint in <100ms, rest loads after

ISOMORPHIC LOADERS

ONE CODE, TWO WORLDS

```
const Route = createFileRoute('/transactions')({
  loader: async () => {
    // This code runs:
    // 1. On the server on first load (SSR)
    // 2. On the client during navigation (SPA)
    // Automatically optimized, no double fetching!

    return await getTransactions()
  }
})
```

COMPARISON WITH NEXT.JS:

- `getServerSideProps` (server only)
- `useEffect` + `fetch` (client only)
- Duplicate logic, duplicate error sources

TRADITIONAL ARCHITECTURE

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

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

PROBLEMS:

- Manually synchronize types
- Maintain API client
- Errors only at runtime

A UNIFIED CODEBASE

```
app/
├── routes/                    # Pages with loaders
│   ├── __root.tsx
│   ├── index.tsx
│   └── transactions.tsx
├── services/                 # Server Functions & logic
│   ├── transactions.ts # DB queries & business logic
│   └── categories.ts
└── db/
    └── schema.prisma     # Single source of truth
```

One project. One type definition. Zero synchronization.

CODE EXAMPLE: TRANSACTION FEATURE

```
// app/features/transactions.ts
import { createServerFn } from '@tanstack/start'
import { prisma } from '~/db'

export const getTransactions = createServerFn()
  .handler(async () => {
    return prisma.transaction.findMany({
      include: { category: true },
      orderBy: { date: 'desc' }
    })
  })

export const createTransaction = createServerFn({ method: 'POST' })
  .middleware([userRequiredMiddleware])
  .validator(TransactionSchema)
  .handler(async ({ data: transactionData }) => {
    // Validation, auth check, etc.
```

CODE EXAMPLE: TRANSACTION FEATURE

```typescript
// app/features/transactions.ts
import { createServerFn } from '@tanstack/start'
import { prisma } from '~/db'

export const getTransactions = createServerFn()
  .handler(async () => {
    return prisma.transaction.findMany({
      include: { category: true },
      orderBy: { date: 'desc' }
    })
  })

export const createTransaction = createServerFn({ method: 'POST' })
  .middleware([userRequiredMiddleware])
  .validator(TransactionSchema)
  .handler(async ({ data: transactionData }) => {
    // Validation, auth check, etc.
```

CODE EXAMPLE: TRANSACTION FEATURE

```ts
// app/features/transactions.ts
import { createServerFn } from '@tanstack/start'
import { prisma } from '~/db'

export const getTransactions = createServerFn()
  .handler(async () => {
    return prisma.transaction.findMany({
      include: { category: true },
      orderBy: { date: 'desc' }
    })
  })

export const createTransaction = createServerFn({ method: 'POST' })
  .middleware([userRequiredMiddleware])
  .validator(TransactionSchema)
  .handler(async ({ data: transactionData }) => {
    // Validation, auth check, etc.
```

THE ROUTE IMPLEMENTATION

```tsx
// app/routes/transactions.tsx
import { createFileRoute, useLoaderData } from '@tanstack/react-router'
import { getTransactions, createTransaction } from '~/features/transactions'

export const Route = createFileRoute('/transactions')({
  loader: () => getTransactions(),
  component: TransactionsPage
})

function TransactionsPage() {
  const transactions = Route.useLoaderData()
  // transactions is fully typed!

  const handleCreate = async (data: FormData) => {
    await createTransaction({...})
    // Automatic revalidation after mutation
  }
```

THE ROUTE IMPLEMENTATION

```tsx
// app/routes/transactions.tsx
import { createFileRoute, useLoaderData } from '@tanstack/react-router'
import { getTransactions, createTransaction } from '~/features/transactions'

export const Route = createFileRoute('/transactions')({
  loader: () => getTransactions(),
  component: TransactionsPage
})

function TransactionsPage() {
  const transactions = Route.useLoaderData()
  // transactions is fully typed!

  const handleCreate = async (data: FormData) => {
    await createTransaction({...})
    // Automatic revalidation after mutation
  }
```

THE ROUTE IMPLEMENTATION

```tsx
// app/routes/transactions.tsx
import { createFileRoute, useLoaderData } from '@tanstack/react-router'
import { getTransactions, createTransaction } from '~/features/transactions'

export const Route = createFileRoute('/transactions')({
  loader: () => getTransactions(),
  component: TransactionsPage
})

function TransactionsPage() {
  const transactions = Route.useLoaderData()
  // transactions is fully typed!

  const handleCreate = async (data: FormData) => {
    await createTransaction({...})
    // Automatic revalidation after mutation
  }
```

BENEFITS IN PRACTICE

WHAT HAS IMPROVED?

Before	**After with TanStack Start**
2 codebases	1 monorepo
Manual type sync	Automatic types
Runtime errors	Compile-time errors
API documentation	Code is documentation
Complex state sync	Built-in caching

🚀 **Development time:** -40% for new features

TANSTACK QUERY INTEGRATION

THE PERFECT DUO

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

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

Optimistic updates, caching, retry - all built-in!

TRADE-OFFS & LIMITATIONS

NOTHING IS PERFECT

👍 ADVANTAGES:

- Unbeatable type safety
- Fantastic DX
- High flexibility
- Streaming SSR

👎 DISADVANTAGES:

- **Beta status** - Breaking changes possible
- **Smaller community** than Next.js/Remix
- **Fewer tutorials/ examples** available

LIVE DEMO

GETTING STARTED

FIRST PROJECT IN 5 MINUTES

```
# Create project
npx create-start-app@latest my-app

# Install dependencies
cd my-app && bun install

# Development server
bun run dev
```

STARTER TEMPLATES:

- `basic` - Minimal setup
- `kitchen-sink` - All features

RESOURCES & COMMUNITY

WHERE TO LEARN MORE

📚 DOCUMENTATION

- tanstack.com/start
- Excellent guides & API docs

KEY TAKEAWAYS

WHAT YOU SHOULD REMEMBER

1. **Type Safety Matters** - Find errors at compile-time
2. **Client-First ≠ Client-Only** - Best of both worlds
3. **Server Functions** - The future of full-stack DX
4. **Beta doesn't mean unstable** - Production-ready with caution
5. **The right choice** - Not every tool for every problem

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

QUESTIONS?

LET'S DISCUSS!

DANILO KÜHN

I'm a developer

THANK YOU FOR YOUR ATTENTION!

🚀 **Happy Coding with TanStack**