A marriage of the frontend and backend

Jonathan Fleckenstein April 1, 2023
https://github.com/fleck/CPOSC-2023
@fleck@hachyderm.io

Most applications have a HARD separation between the front end and backend

Pain points with current development techniques

- GraphQL has a decent amount of development overhead.
- REST isn’t typed by default.
- Logic is duplicated.
- End to end testing is difficult.
- Requires context switch if backend and front end languages differ.

A lot of applications don’t need a discreet front and and backend
Major frameworks are providing solutions bridge the gap

- Hotwire from the Rails crew
- LiveView from Phoenix
- Blazor in .NET land
- Remix, SvelteKit, and **Next.js** from the JavaScript folks

Terminology

Server side rendering

The UI is created by parsing HTML sent from the server.

- Initial page view is faster because the client can start building the UI as HTML streams in without waiting for JS to be parsed.
- Better for SEO.
- Battle tested approach to web development.
- Interactivity that doesn’t depend on server side data is slower.

Client side rendering

Most of the UI is created client side via a library like React.

- UI can respond immediately.
- Can reduce load on origin servers.
- Initial render can be slower.
- Adds complexity over traditional server side HTML.

Sharing logic

Many applications want to statically render HTML for better initial load perf which requires business logic server side, BUT, when users are interacting with our UI we want to give feedback ASAP which requires logic client side.

Calculation logic

http://localhost:3000/Change-Oil-in-a-Prius

Techniques to help share logic

- Pure functions operating on data allows easy and efficient code sharing.
- Data can be basically any structure that fits into JSON. E.g. no functions, Classes, etc. just arrays, objects and simple values.
- Pure functions allow compilers to ship only the code that’s needed to users saving bandwidth and improving app startup times. “Tree shaking” code is very hard when using `this` as it can be hard to statically analyze and determine if the code is ever used.

Next.js knows what code is used server side and client side and trims the respective bundles to only include the necessary code.

Unified client side and server side routing

Server side rendering provides a good experience for first time visitors.

Client side routing allows easier data prefetching and no re-parsing of JavaScript and CSS bundles between navigations.

Client side only sucks

Server side routing isn’t all roses

- Re-evaluate scripts on each navigation
- Re-parse styles across navigations
- Some limits on prefetching data for the next page
- Have to fetch full HTML representation of page, client side can just fetch leaner JSON
- Limits on page transitions

Client side routing only needs to fetch JSON and nearly instant page transitions

https://idealguides.com

End to end testing with code coverage

The holy grail of testing?

Potential advantages

- User focused testing.
- Less tests cover more code.

Problems with E2E testing

- Flaky
- CI can’t be gated by them as changes not related to the pull request often cause the tests to fail.
- Slow

Advancements in end to end testing

- Tools like Playwright are built with parallelization in mind.
- Tools can detect OS/CPU Architecture and simplify browser install.
- APIs to wait on elements to be visible and “ready to click” have advanced a ton.

Next.js makes end to end testing easier

- When we start our local dev server the frontend and backend are compiled at the same time.
- Next uses a fast rust based compiler to ensure speedy compilation of large codebases.
- Using the same language allows us to collect code coverage for the frontend and backend.

HTTP Calls With the Ergonomics of a Function Call

Pain points with GraphQL

- We already have a database schema, why recreate it all again as a GraphQL schema?
- Code generation step needed for type safety.
- Network level caching, e.g. cache control doesn’t work well.

Pain points with REST APIs

- No type safety on API calls.
- Over or under fetching data.
- Lack of convention, end up with a lot of ad hoc routes and query parameters.

- All of the existing IDE stuff works. E.g. go to definition, rename symbol, automated refactoring etc.

- Most codebases already have a “type gap” between the datastore and the application. Using a compile time API calls ensures that types flow through to front end.
- Ensure API changes are always in sync with callers.

tRPC

- Automatic typesafety, no code generation.
- Easy input sanitization.
- Works with vanilla React, Next.js, Express, Fastify, AWS Lambda, Solid, and Svelte.
- Minimum code added to client side bundle.

A query on the server

```
  indexers: publicProcedure
    .input(z.object({ hostnames: z.array(z.string()) }))
    .query(async ({ input }) => {
      const indexers = await db.indexer.findMany({
        where: { hostname: { in: input.hostnames } },
      })

      return indexers
    }),
```

The query on the client

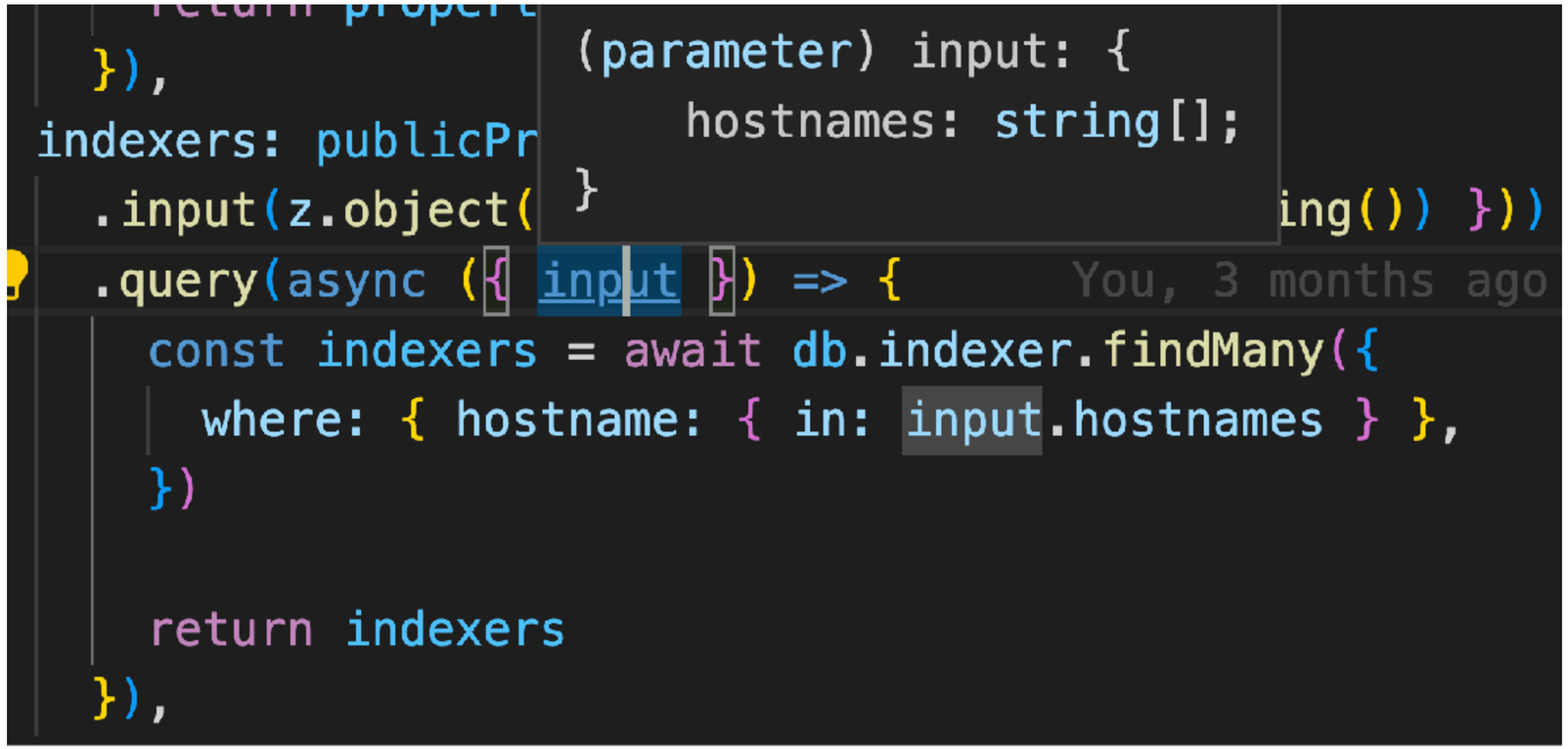
```
const { data: indexers = [] } = trpc.indexers.useQuery({
  hostnames: [...urls],
})
```

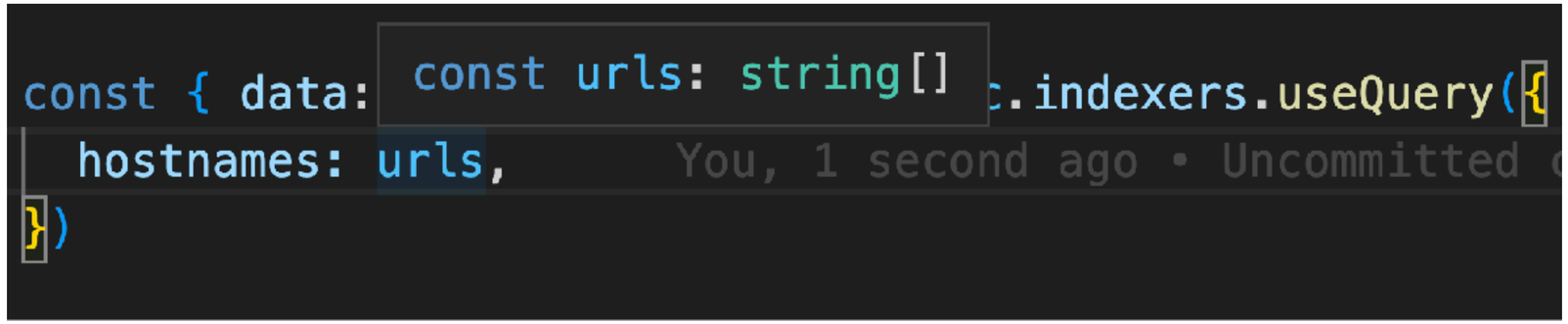
The return type on the server matches the type on the client

```
indexers: publicProcedure
  .input(z.object({ hostnames: z.array(z.string()) }))
  .query(async ({ input }) => {
    const indexers = await db.indexer.findMany({
      where: { hostname: { in: input.hostnames } },
    })
              const indexers: Indexer[]
    return indexers
  }),
```

```
              const indexers: Indexer[]
const { data: indexers = [] } = trpc.indexers.useQuery({
  hostnames: [...urls],
})
```

The server input type matches the client side arguments





Simplified deploys

The same server that’s serving the API endpoints is also severing the front end bundled code. So a single deploy will update everything!

Questions?