A tale of TypeScript conversion

Author: Vinicius Aurichio

Claps: 74

Date: Feb 13

Last fall Klaviyo completed its first ever acquisition, Napkin.io. While there were many differences between Napkinâ€™s frontend code and Klaviyoâ€™s, one in particular stood out: Napkin was using JavaScript, while Klaviyo uses TypeScript. From our experience we knew the benefits of using TypeScript over pure JavaScript, so after speaking to Napkinâ€™s founder we decided to migrate to TypeScript early on.

While there are many books and blog posts about how to convert a JavaScript code base to TypeScript, each conversion is unique. Reading about other peopleâ€™s experiences can help you plan your conversion, and thatâ€™s what this blog post is about. I will share the major decisions I made along the way, the rationale behind them and what went wrong.

Incremental vs Big Bang PR

You can either convert the entire code base to TypeScript at once or slowly adopt it, changing one file at a time. In theory, you can just change all file extensions from â€˜.jsâ€™ to â€˜.tsâ€™ and call it a day as TypeScript is a strict superset of JavaScript, but that does not give you much benefit.

Klaviyo originally had its frontend code in vanilla JavaScript. When we decided to move to TypeScript, we used the incremental adoption strategy. The codebase already had thousands of files owned by dozens of different teams and locking merges for a few weeks while the conversion happened was infeasible. Napkin, on the other hand, had only a couple hundred files and only a few developers working on it, so a big bang approach was viable.

Still, instead of writing one giant PR touching every single file to add types everywhere (and annoying my fellow developers with the most intractable PR review ever), I decided to split the work into a series of smaller, more focused conversions. After 8 PRs (not counting the reverts and bug fixes, more on that later) our entire codebase was fully converted to TypeScript.

Converting from the outside in or the inside out

Another decision you have to make is where to start. You can convert your leaf components first. You would want to do that because they likely have simple interfaces and you can use their types to guide the conversion of their parent components and the APIs that feed data to them. Alternatively you can start from the top level â€” your data fetching functions and contexts. You would want to do that to type your core building blocks first and adjust your inner components based on the inputs you are getting from the outside.

To illustrate both approaches, letâ€™s use the following JavaScript snippet:

```
export const getNapkin = async (user, { napkinId }) => {
  fetch(getNapkinUrl(napkinId), { headers: getAuthHeaders(user) })
}

export const NapkinCard = (napkin) => (
  <Card>
    <CardHeader title={napkin.name} icon={napkin.runtime} />
    <CardBody>{napkin.description}</CardBody>
  </Card>
)
```

We have a function that fetches the napkin data and a react component that displays a card using that data. If we want to convert this code inside out, we would type the *NapkinCard* component first and have something like this:

```
export const NapkinCard = (napkin: {
  name: string
  runtime: "python" | "node"
  description: string
}) => (
  <Card>
    <CardHeader title={napkin.name} icon={napkin.runtime} />
    <CardBody>{napkin.description}</CardBody>
  </Card>
)
```

The *NapkinCard* component now declares what it expects to be in the object it receives and when we go about typing our API, TypeScript will make sure we fulfill these expectations.

Now, letâ€™s say we do it the other way around and we start by typing the API.

```
type NapkinType = {
  uid: string
  name: string
  runtime: "python" | "node"
  description: string
  domain: string | null
  // all other fields
}

export const getNapkin: (
  user: User,
  { napkinId }: { napkinId: string }
) => NapkinType = async (user, { napkinId }) => {
  fetch(getNapkinUrl(napkinId), { headers: getAuthHeaders(user) })
}
```

Now, if we so wish, we can use the napkin object type directly in the card. This makes sense for the *NapkinCard* component as it is a specialized card that will only ever be used to render a napkin object. All properties of the napkin object are now visible to the *NapkinCard* component, even those that are not used by that component.

The code editorâ€™s autocomplete shows all fields available on the napkin object.

There are tradeoffs when passing the entire object down like this and you should make sure you understand them when using this approach. Note however that we can get the best of both worlds and destructure the object in the card argument while reusing the napkin type. There is no need to explicitly type the destructured values!

```
export const NapkinCard = ({name, runtime, description}: NapkinType) => ..
```

After perusing the codebase for a while I decided I wanted to have the most used object typed first. Unsurprisingly, in the Napkin codebase that object was the napkin object. Given its widespread use, it made sense to type it at the API layer and propagate the types down the component tree, so thatâ€™s why I chose to convert from the outside in.

Napkin did not have a separate folder for functions that make API calls, a common pattern and one that we use at Klaviyo, so my first step to get a typed API was to create thin wrapper functions that managed all the function parameters, built the URL, called the endpoint with the right HTTP verb, prepared headers, etc.Â[1] After creating a group of API endpoints (say, all endpoints under the /napkins path) I would convert all call sites to use the new functions. When enough data flowing into a component was coming through these typed interfaces, I would convert the component to TypeScript too. The now-available type information exposed subtle bugs in the components and also served as a guide to write the remaining types.

Zod â€” a new best friend

When you add types to an API endpoint using TypeScript, you are coercing the type system into believing you. You are trusting that the data coming in is always going to have that shape and telling TypeScript to trust your knowledge. This means you might have the illusion of your code being type safe, when in reality it is not. The only way to know for sure that the returned values from the API match your expectations is to perform runtime validation and thatâ€™s exactly what Zod does for you.

While we have full control over Napkinâ€™s backend and could in theory figure out exactly what gets returned, some of the return values were not obvious even looking at the code. For example, values that were strings 99% of the time were sometimes undefined or null for a legacy record. On the other hand, we already knew that whatever was coming through the API was compatible with the frontend code as we were not observing crashes due to mismatched types.

These two factors combined â€” the need for runtime validation while knowing we did not need to crash the application in case of a mismatch â€” led me to the following solution:

```
export const parseOrPassThrough = <T extends ZodSchema>(
  payload: z.infer<T>,
  parser: T
```

```
): z.infer<T> => {
  try {
    return parser.parse(payload)
  } catch (e) {
    if (e instanceof ZodError) {
      if (ENV === "development") {
        console.log(e)
        console.log(payload)
      }
      Sentry.withScope((scope) => {
        scope.setLevel("warning")
        Sentry.captureException(e)
      })
      return payload as z.infer<T>
    } else {
      throw e
    }
  }
}
```

And it gets used like this:

```
export const getExamplesList = async () => parseOrPassThrough(
    await getData(examplesListUrl),
    GetExamplesListSchema
  )
```

Here, *GetExamplesListSchema* is a *ZodSchema* object that exposes the parse method called in *parseOrPassThrough*. The *getExamplesList* function will be typed with our desired schema, but we will be alerted in case one of our assumptions is violated. For a better developer experience, I also added extra logging for local development. When we get those alerts, we can then either make changes to the backend (e.g. when we are unintentionally returning multiple types from an endpoint), or adjust the types in the frontend when we notice we made wrong assumptions about the valid return types. Note that we have to pass through the raw value from the API when a validation error occurs, otherwise our application would crash.

Enable TypeScript strict mode and configure other linters

Once every file was converted to TypeScript it was time to see what I missed. Because TypeScript supports gradual adoption, it does not enforce certain aspects of type safety by default. By enabling strict mode it will yell at you if you are doing something potentially unsafe. In my case in particular, strict mode complained about multiple instances of implicit â€œanyâ€ type. When TypeScript is unable to determine the type of a variable it will give up and say that it is compatible with literally anything. There are two ways to address this error: make the implicit â€œanyâ€ explicit, or add types to the interface. In most cases I went with the latter option, but there were a few gnarly cases where I decided it was OK to type it as â€œanyâ€ and fix it later.

Thatâ€™s where the other linters come in. When you are ready to add an extra layer of strictness to your code, you can close escape hatches and enforce more rules. I used ESlint with the TypeScript plugin and enabled, among other rules, *no-explicit-any*. This does exactly what it says on the tin: you are not allowed to use â€œanyâ€ anymore.

Do not make any changes besides adding types to your code

I mentioned something about bug fixes and reverted PRs and thatâ€™s what caused them. Converting the code to TypeScript is a big enough change that it should not be bundled with other bug fixes or refactors. In some isolated instances it is OK to violate this rule to make typing easier. For example, you might want to convert a for loop to a .map/.reduce call, but you should only make changes you know for sure will not change the behavior of your code.

I violated this rule and caused two major incidents. In one instance, I introduced a stale closure that made it impossible for users to create new accounts. Not ideal when you are actively inviting users to use your product. In another, I tried to consolidate the type of a variable (it could be undefined, null, or a string and I tried to make it an optional string, i.e. disallow null). This changed the code execution in such a way that we were overwriting customerâ€™s code with the default boilerplate. We were able to fully recover the code for all affected users, but it would have been better if we didnâ€™t have to go through the recovery process in the first place.

Conclusion

Thatâ€™s my report on how I converted Napkinâ€™s frontend code to TypeScript. While it was inspired by heaps of blog posts and books, it was as unique as other conversions. My main takeaways were:

- There is no one-size-fits-all approach to make the conversion, but there are ample resources to help you decide what works best for you.
- Zod is great and everyone using TypeScript should check it out.
- Resist the urge to make small refactors alongside the conversion. Keep typing changes isolated to prevent unwanted behavior changes.

Useful Links

- Effective TypeScript (Oâ€™Reilly book)
- Migrating millions of lines of code to TypeScript (Stripe blog post)
- A simple guide for migrating from JavaScript to TypeScript (LogRocket blog post)

[1] A little note about other choices I had to make: most napkin endpoints were hit from exactly one place, so even though they were spread throughout the code there was not much duplication. While creating a function for each API endpoint helps us understand the API surface and allows for a more uniform interface, it also increases code complexity by adding layers of abstraction between the data source and its point of use.