11. Designing Type-Safe APIs

So much of programming is about using the code of **other people**.

When we start a new project *"from scratch"*, we often end up weaving together frameworks, libraries, and tools built by hundreds and hundreds of people from all over the globe.

In a way, our ability to build successful projects depends on whether *their code* can be combined in new and interesting ways! **API design** is therefore an incredibly central problem.

The API, or *Application Programming Interface*, refers to how you interact with some piece of code — whether it's an HTTP endpoint, an object with properties and methods, or just a simple function.

So what makes for a great TypeScript API?

If you got to this chapter, chances are you're thinking that it comes down to one thing — ***writing good types*** ✨

That makes a lot of sense, types ***are*** the language of API design in TypeScript! The more precise the type, the more potential runtime errors can be caught ahead of time, so it seems evident that precise types improve the developer experience.

But from time to time, hovering over a seemingly harmless red squiggle forces us to reconsider this position... 🔎

```
Type 'ColumnDef<Shipment, unknown>' is not assignable to type
'ColumnDef<ExampleType, unknown>' with 'exactOptionalPropertyTypes: true'.
Consider adding 'undefined' to the types of the target's properties. Type
'AccessorKeyColumnDefBase<Shipment, unknown> &
Partial<StringHeaderIdentifier>' is not assignable to type
'ColumnDef<ExampleType, unknown>' with 'exactOptionalPropertyTypes: true'.
Consider adding 'undefined' to the types of the target's properties. Type
'AccessorKeyColumnDefBase<Shipment, unknown> &
Partial<StringHeaderIdentifier>' is not assignable to type
'AccessorKeyColumnDefBase<ExampleType, unknown> &
Partial<IdIdentifier<ExampleType, unknown>>' with 'exactOptionalPropertyTypes:
true'. Consider adding 'undefined' to the types of the target's properties.
Type 'AccessorKeyColumnDefBase<Shipment, unknown> &
Partial<StringHeaderIdentifier>' is not assignable to type
'AccessorKeyColumnDefBase<ExampleType, unknown>' with
'exactOptionalPropertyTypes: true'. Consider adding 'undefined' to the types
of the target's properties. Types of property 'accessorKey' are incompatible.
Type 'string | number | symbol | (string & {})' is not assignable to type
'(string & {}) | keyof ExampleType'. Type 'number' is not assignable to type
'(string & {}) | keyof ExampleType'.
```

😵

I bet you've experienced similar errors in your TypeScript journey. This one comes from `@tanstack/react-table`, a very popular React library for building complex tables. What went wrong here?

I can tell one thing: library authors do not intend to make you read giant walls of text that make you feel miserable. They only want to prevent you from shooting yourself in the foot by misusing their code. Yet, like everything in programming (in *life*?? 😬), it seems like precise types come with **tradeoffs**.

The truth is that **there is more to API design than just type safety**. Developers using our code will spend a significant portion of their time debugging and reading the type errors we throw at them. The good news is that TypeScript gives us a lot of ways to control this crucial aspect of the developer experience!

In this chapter, we will put ourselves **in the shoes of an open-source developer**. We are about to build a library together and focus on how we can guide our users to success through great error messages and helpful auto-complete suggestions!

The **principles** we are going to follow are simple:

1. *If it **type-checks**, it should work.*
2. ***Error messages** should be short and easy to understand.*
3. ***Auto-complete suggestions** should nudge users toward working code.*

Along the way, I'll share all the undocumented techniques I've collected over the years from the best open-source libraries out there to make TypeScript APIs truly delightful.

Let's get to it!

A tricky problem

We are building ***TrickTok*** 🛹 — a social network to share skateboarding trick videos! Of course, we chose TypeScript as our backend language. The product is simple: you can look at trick videos in a feed, open the details view, like them, and post comments.

To keep it simple, we initially built everything as a single request handler function, taking a `Request` and returning a `Response`:

```
async function requestHandler(req: Request): Promise<Response> {
  const { method, url } = req;

  if (method === "GET" && matches(url, "/tricks/:trickId")) {
    return trickVideoPage(req);
  }

  if (method === "POST" && matches(url, "/api/tricks/:trickId")) {
    return postTrick(req);
  }

  // ... more routes
}
```

And we created a handy `matches` function of the following type:

```
function matches(url: string, pattern: string): boolean {
  /* Returns true if `url` matches `pattern`. */
}
```

But we forgot something! Even though we want anyone to be able to watch a video, only **authenticated users** should be able to **post** new content.

No biggy, we start sprinkling some `if` statements here and there:

```
// ...
if (method === "POST" && matches(url, "/api/tricks/:trickId")) {
  //        👇
  if (!isAuthenticated(req)) return "⛔";
  return postTrick(req);
}
// Add this to all POST routes ...
```

🤔 Thinking about it, we should probably **validate** that clients are sending a well-formed request body. Let's add another `if`.

```
// ...
if (method === "POST" && matches(url, "/api/tricks/:trickId")) {
  if (!isAuthenticated(req)) return "⛔";
  //      👇
  if (!isValidVideo(req.body)) return "👎";
  return postTrick(req);
}
```

We'll try to remember to validate the body of all present and future routes too! 😅

Oh, and we'd like to have a dashboard page, only accessible to administrators. Let's add just one more `if`:

```
// ...

if (method === "GET" && matches(url, "/admin/dashboard")) {
  if (!isAuthenticated(req)) return "⛔";
  //      👇
  if (!isAdmin(req)) return "👮";
  return adminDashboard(req);
}
```

But as time passes and product requirements become more complex, our code starts to look more and more like a plate of spaghetti... 🍝 Is there something we could do about that?

The perfect routing library 📐

After brainstorming with our TrickTok colleagues, we came up with a neat little server-side routing library — which we soberly named `routing-library`.

You can create a route using the `route` function:

@/routes-1.ts

```
import { route } from "routing-library@v1";

export const home = route("GET /")
  .handle(() => `Tricks feed! 🛹`);

export const video = route("GET /tricks/:trickId")
  .handle(() => `Trick video page ▶️`);
```

Checking user permissions is easy. You only need to call `.isAuthenticated()` or `.isAdmin()`:

@/routes-2.ts

```
import { route } from "routing-library@v1";

export const details = route("GET /tricks/:trickId/details")
  .isAuthenticated() // returns ⛔ if unauthenticated
  .handle(() => `🎛️`);

export const admin = route("GET /admin/dashboard")
  .isAuthenticated()
  .isAdmin() // returns 👮 to non-admins
  .handle(() => `📈`);
```

You can validate `POST` data using `.validateBody()` and a predicate function:

@/routes-3.ts

```
import { route } from "routing-library@v1";

type Video = { title: string; url: string };

declare function isVideo(body: unknown): body is Video;

export const postTrick = route("POST /api/tricks")
  .isAuthenticated()
  .validateBody(isVideo) // 👈
  .handle(async ({ body }) => {
    await database.videos.create(body);
    return `✅ Sick trick!`;
  });
```

And once all your routes are ready, you can add them to a router, and start the server:

server.ts

```
import { router, route } from "routing-library@v1";
import { home, video } from "@/routes-1";
import { details, admin } from "@/routes-2";
import { postTrick } from "@/routes-3";
import { postComment } from "@/routes-4";

const server = router()
  .add(home)
  .add(video)
  .add(details)
  .add(admin)
  .add(postTrick)
  .add(postComment)
  .add(route("GET *").handle(() => "404 not found!"));
//                 👆 this is a wildcard matching everything.

server.start(3000);
```

The types of our library are really simple so far. We have a `router` function that returns a `Router` instance:

```ts
export declare function router(): Router;

type Router = {
  add: (route: Route) => Router;
  start: (port: number) => void;
};
```

and a `route` function returning a `Route` instance:

```ts
export declare function route(path: string): Route;

type Route = {
  isAuthenticated: () => Route;

  isAdmin: () => Route;

  validateBody: (
    predicate: (body: unknown) => boolean
  ) => Route;

  handle: (
    handler: (req: ParsedReq) => string | Promise<string>
  ) => Route;
};
```

The `ParsedReq` type that our `handler` function takes is created by parsing the native `Request` object we get when receiving a request. Here is what it contains:

```ts
type ParsedReq = {
  cookies: Record<string, string>;
  params: Record<string, string>;
  body: any; // 😬 Ouch, this isn't exactly type-safe.
};
```

> 💡 The JavaScript implementation is omitted for brevity.

Pretty excited about this library, we decide to publish it to `npm`, and guess what? The project is really **taking off** 🚀

But with the ever-increasing number of users, **GitHub issues** also start to pile up… 😱

Catching invalid inputs

After spending time sorting through our mailbox, a significant chunk of our issues seems to be coming from people ***misusing*** the library rather than from actual bugs.

📮 Our library only supports `GET`, `POST`, `PUT`, and `PATCH`, but some people attempt to use it with other HTTP methods:

```ts
import { route } from "routing-library@v1";

route("TRACE /my/api/endpoint"); // 💥
route("OPTION /my/api/endpoint"); // 💥
```

📮 A few people forgot to provide a method to their route:

```ts
import { route } from "routing-library@v1";

route("/oops/no/method"); // 💥
```

📮 And one person passed a path with spaces in it 🤦

```ts
import { route } from "routing-library@v1";

route("GET /oh no/spaces!"); // 💥
```

Could we improve our types to catch these problems?

Template literal types aren't good enough

Your first instinct might be to update the `route` function to take a Template Literal Type instead of a `string`:

```
type Method = "GET" | "POST" | "PUT" | "PATCH";

type RoutePath = `${Method} ${string}`;
//                    👆
//                                        👇
export declare function route(path: RoutePath): Route;
```

It's definitely better because it catches some of these problems:

```
import { route } from "routing-library@v1.1";

route("TRACE /my/api/endpoint"); // 🎉
route("/oops/no/method"); // 😱
```

But it unfortunately does not reject routes containing spaces:

```
import { route } from "routing-library@v1.1";

route("GET /oh no/spaces!"); // type-checks 😐
```

It would be kind of cool if we had a way to tell TypeScript that a value should ***not*** be assignable to a type. Maybe something that would look like this:

```
type RoutePath =
      ${Method} ${string}` &
      !`${string} ${string} ${string}`;
/*    👆
   Imaginary ! syntax 🦄
                        */
```

Unfortunately, negative types aren't a thing in TypeScript *yet*[1].

The other problem is that the compiler yields a pretty unwieldy error message…

```
Argument of type '"TRACE /my/api/endpoint"' is not assignable to parameter of
type '`GET ${string}` | `POST ${string}` | `PUT ${string}` | `PATCH
${string}`'.
```

It would much be clearer to display something like *"TRACE isn't a supported method"* instead. Is there anything we can do about this?

The type validator pattern

Here is the first trick on the agenda. In TypeScript, you can wrap any type parameter in a **type-level guard function** to modify it, or reject cases you do not want. Here is how.

First, we introduce a type parameter `P`:

```
//                                  👇
export declare function route<const P extends string>(
  path: ...
): Route;
```

Notice the use of `const` to let TypeScript infer `P` as an *"exact"* string literal type.

Then, we wrap `P` with a **type validator**:

```
export declare function route<const P extends string>(
  path: ValidateRoute<P>,
  //          👆
```

```
): Route;
```

What does it look like? Just a series of conditions:

```
type ValidateRoute<Path extends string> =
    // In case it contains multiple spaces, reject path:
    Path extends `${Method} ${string} ${string}` ? never
    // The path is what we expect:
  : Path extends `${Method} ${string}` ? Path
    // In case the path doesn't have a method, reject it:
  : never;

type Method = "GET" | "POST" | "PUT" | "PATCH";
```

This function returns `never` whenever the `Path` is invalid, and returns it unchanged otherwise.

That's right, you can **infer** a type **while** conditionally **modifying it** in TypeScript. It's pretty cool because from now on, only valid uses of `route` will type check!

```
import { route } from 'routing-library@v1.2';

route("TRACE /my/api/endpoint"); // ❌
route("/oops/no/method");        // ❌
route("GET /oh no/spaces!");     // ❌

route("GET /");                  // ✅
route("POST /tricks");           // ✅
route("PUT /tricks/:id");        // ✅
```

What's not so hot however is the error message we get.

```
Argument of type 'string' is not assignable to parameter of type 'never'.
```

It's pretty cryptic. People unfamiliar with our internals will probably wonder what `never` has to do with their route.

But we aren't out of ideas yet!

Using string literals as error messages

One way or another, we have to wrestle with the fact that TypeScript's generic error messaging system **lacks** some **context** about our domain. An important thing it does not know is ***why*** this input is **invalid**. That's why the compiler does not produce a very insightful error message most of the time.

But we can hack ourselves around this limitation! A pretty common pattern in libraries nowadays is to use **String Literal Types** as error messages.

Concretely, we can define error messages as types:

```
type NoSpacesError = "Spaces aren't allowed in route paths.";
type InvalidPathError = "This path is invalid.";
```

And return them when we don't like the input:

```
type ValidateRoute<Path extends string> =
    Path extends `${Method} ${string} ${string}` ? NoSpacesError
  : Path extends `${Method} ${string}` ? Path    //    👆
  : InvalidPathError;
  //      👆
```

Here's the result:

```
import { route } from 'routing-library@v1.3';

route("GET /oh no/spaces!");     // ❌
route("PUT /tricks/:id");        // ✅
```

```
'"GET /oh no/spaces!"' is not assignable to parameter of type '"Spaces aren't
allowed in route paths."'
```

The reason we get an error is much clearer than in the previous version, though I'll admit this message is still a little bit misleading. It would be great if we could get rid of the `is not assignable` part, but that's not possible today.

Let's practice 🤹

Now it's your turn! Let's add error messages to catch more invalid inputs, nudge developers toward using our library correctly, and — most importantly — without opening an issue! 😅

Challenge | Solution — Format ✨ — Reset

```ts
/**
 * Add code branches covering the following invalid cases:
 * - No method was provided.
 * - An unsupported method was provided.
 * - A valid method was provided, but a path is missing.
 * Feel free to use the following types:
 */
type NoMethodError = "Method missing.";
type UnknownMethodError<M extends string> = `'${M}' isn't a supported method.`
type PathMissingError<T extends string> = `A path is missing after '${T}'.`;

type NoSpacesError = "Spaces aren't allowed in route paths.";
type InvalidPathError = "This path is invalid.";

type Method = "GET" | "POST" | "PUT" | "PATCH";

type ValidateRoute<Path> =
    Path extends `${Method} ${string} ${string}` ? NoSpacesError
  : Path extends `${Method} ${string}` ? Path
  : InvalidPathError;
```

Can we do even better? 💪

Do you know what's even nicer than a nice error message?

Auto-complete suggestions of course! It would be pretty cool if developers could just hit their `tab` key to fix their route.

We're in luck because the **Type Validator** pattern can also be used to generate lists of suggestions. The only thing we need to do is to return a **union of valid route paths** from our validator instead of an error message.

Let's start with something simple. If the route is invalid, we will return the following suggestions:

```ts
type Method = "GET" | "POST" | "PUT" | "PATCH";

type Suggestions = `${Method} /`;
//    ^? "GET /" | "POST /" | "PUT /" | "PATCH /"
```

To achieve this, we only need to replace the default error of our `ValidateRoute` type with our suggestions.

```ts
type ValidateRoute<Path extends string> =
    Path extends `${Method} ${string} ${string}` ? NoSpacesError
  : Path extends `${Method} ${string}` ? Path
  : Suggestions; // 👈
```

And here is the result:

```ts
import { route } from "router-library@v1.5";

route(""); // 👈 Type "get" or "post" and press tab
```

Nice!

Now, something that would be even cooler would be to suggest **prefixing** the string our user typed **with a method** if it is missing.

It is also pretty easy! Instead of returning a static list of suggestions, we only need to ***interpolate*** the current path in it:

```ts
type Suggestions<Path extends string> = `${Method} ${Path}`;
```

And update `ValidateRoute` accordingly:

```ts
type ValidateRoute<Path extends string> =
    Path extends `${Method} ${string} ${string}` ? NoSpacesError
  : Path extends `${Method} ${string}` ? Path
  : Suggestions<Path>; // 👈
```

Let's see if it works.

```ts
import { route } from "router-library@v1.6";

route("/"); // 👈 Type "/something" and press tab
```

We can now type our path directly and select the method we want to use from the list of suggestions. That's awesome 🙌

To infer or not to infer 💀

If you hover over the following `route` function call, you will notice that we have a little problem.

```ts
import { route } from "router-library@v1.6";

const x = route("GET /something");
/*         👆
    hover over `route`   */
```

The input string is now inferred as a **union** of **string literals**.

```ts
route<"GET /something" | "/something">(path: ...): Route
//                    👆
//     We would expect just "GET /something"
//     But we get a union instead...
```

Weird!

This is happening because the type checker is getting confused by our suggestions. Since we added them, it became unclear which branch of `ValidateRoute` TypeScript should use to infer the `Path` type:

```ts
//  Should TS infer `Path` from...
type ValidateRoute<Path extends string> =
    Path extends `${Method} ${string} ${string}` ? NoSpacesError
  : Path extends `${Method} ${string}` ? Path
  : `${Method} ${Path}`; //               👆
  //              👆                    This
  //         That branch?      or       branch?

function route<const P extends string>(
  path: ValidateRoute<P>
): Route;
```

TypeScript is like this friend, who's invited to three different parties on a Saturday night and ends up going to all of them — **it's not great at making choices!** Instead of picking a branch, it returns a union of all the possible ways to infer `Path`. That's a pretty big deal because this will prevent us from using `Path` to return a smarter `Route` type in the next section.

But here is the good news — TypeScript's standard library provides a **type helper** specifically designed to solve this problem! It's called `NoInfer`:

```
type ValidateRoute<Path extends string> =
  // ...
  : NoInfer<`${Method} ${Path}`>;
  //  👆
```

By wrapping our suggestions with `NoInfer`, we solve TypeScript's inference dilemma:

```
import { route } from "router-library@v1.7";

const x = route("GET /something");
/*          👆
    hover over `route`   */
```

`Path` is now inferred as `"GET /something"`! That's much better.

`NoInfer` is available since version 5.4. My recommendation is to always wrap suggestions with it to avoid any inference shenanigans.

```
type Suggestions<Path extends string> = NoInfer<
  `${Method} ${Path}` | `${Method} /`
>;
```

Type validator takeaways 📚

Here are the key points to keep in mind.

- Great errors and suggestions make a huge difference in terms of developer experience.
- The type validator pattern lets us **precisely control** what can and can't be passed to our functions and methods.
- String Literals can be used to make TypeScript display **custom error messages**.
- A type validator can also return a list of auto-complete **suggestions**.
- Don't forget to wrap your suggestions into a `NoInfer` to sidestep inference problems!

Pretty satisfied with ourselves, we decided to publish ✨ `routing-library@v2` ✨ to npm!

New release, new challenges

Our new release was very well received by the community. The number of users is rising to new heights!

The only problem is that it didn't completely prevent our pile of issues from growing. Our mailbox is as full as ever. Let's take a look at what's inside 📬

📮 The first category of issues comes from people complaining that their customers can send **incorrect payloads** to their endpoints.

```
import { route } from "routing-library@v2";

export const updateProduct =
  route("PUT /api/products/:id")
    .handle(async ({ body, params }) => {
      await database.products.update(params.id, {
        name: body.name,
        //        ^ 💥 runtime error!
        //             body is undefined.
        description: body.description,
        price: body.price,
      });
      return "✅";
    });
```

They should always use `.validateBody()` before calling `.handle()` for `POST`, `PATCH`, and `PUT`

routes, but they don't seem to know it exists!

🥊 The second category is worried that typing `body` as `any` would let them push **unsafe code** that might throw in production.

```ts
import { route } from "routing-library@v2";

export const updateProduct =
  route("PUT /api/products/:id")
    .validateBody(isProduct) // ✅ it's a product.
    .handle(async ({ body }) => {
      return body.tags.length
      //             ^ 💥
      //   tags doesn't exist on Product!
    });

type Product = { name: string; description: string; price: number };

declare function isProduct(body: unknown): body is Product;
```

They'd like us to ***infer*** a stricter type for `body`, and that's understandable! No open-source library should ever let an `any` escape to their user's codebase.

🥊 Lastly, a few people are unsuccessfully using `.validateBody()` on `GET` routes:

```ts
import { route } from "routing-library@v2";

export const updateProduct = route("GET /api/products/:id")
  .validateBody(isProduct)
  .handle(async ({ body }) => {
    return body.name;
    //         ^ 💥
    // body is always undefined at runtime 🤔
  });
```

Since `GET` requests don't have a body, there is no way this code can work 😬

We are facing a very common problem in API design — we have a **combination** of options that **doesn't make sense**. It's useful to support `GET` requests. It's useful to validate the body of a request. But it doesn't make sense to validate the body of a `GET` request! The problem is that our types currently allow those invalid combinations.

For our next release, we set out to fix these problems!

Tracking state with types

To prevent people from calling `.validateBody()` on `GET` requests, we would naively want to be able to write something like this:

```ts
type ValidateBody = (body: unknown) => boolean;

type Route = {
  /*       Imaginary type in scope ,˝ ✨
                    👇                    */
  validateBody: ThisIsAGetRoute extends true
    ? never
    : ValidateBody
  // ... other methods.
};
```

Here, we conditionally "remove" the `validateBody` method by assigning it to `never`.

To write this code though, we somehow need to know the HTTP method used by the current route. We need to **lift some state** to the **type level**.

That only requires introducing a single type parameter:

```ts
type Route<M extends Method> = {
  /*       👆
      The method is
      in scope now!
                    */
  validateBody: M extends "GET"
    ? never
    : (isValid: (body: unknown) => boolean) => Route<M>
```

```
  // ... other methods.
};
```

We now need to pass this extra parameter when creating our `Route`. Let's update the `route` function:

```
export declare function route<const P extends string>(
  path: ValidateRoute<P>
): Route<GetMethod<P>>;
//           👆

type GetMethod<P> =
  // we pattern-match on the path
  P extends `${infer M extends Method} ${string}`
    ? M
    : never;
```

That's it! Now it should no longer be possible to call `.validateBody()` on a `GET` route.

Let's make sure our code works as we expect:

```
import { route } from "routing-library@v2.1";

route("GET /products")
  .validateBody(isProduct) // ❌ shouldn't type-check
  .handle(() => "...");

route("POST /products")
  .validateBody(isProduct) // ✅ should type-check
  .handle(() => "...");
```

Yay! 🎉

We just extended the Builder Pattern that we've been using from the beginning **with a type parameter**. It may not look like much, but I find this pattern so interesting that I gave it a name: the ***Type Builder Pattern***!

The type builder pattern

First, let's step back for a quick recap of what the builder pattern is about. The core idea is to **avoid** passing every single option **up front**, like this:

```
// example of passing an options object:
const chart = createChart({
  type: "barChart",
  colorPalette: "warm",
  yScale: "linear",
});

chart.render([1, 2, 3]);
```

And instead pass these options by chaining methods on what is usually called a *builder object*:

```
// example of using the builder pattern:
const chart = createChart()
  .type("barChart")
  .colorPalette("warm")
  .yScale("linear");

chart.render([1, 2, 3]);
```

At the value level, the main benefit of this approach is to decouple the way you create an object from what it actually looks like under the hood. Granted, these two flavors of API design don't make a huge difference here.

It's when you add types to the mix that things start to get more interesting: the builder pattern enables us to **update** the ***type*** of our builder object every time we chain a method. It means we can **dynamically change our API** in function of the options that have been passed so far!

Dynamic APIs with type builders

Let's get back to our router. Only half of the job is done: we still need to make calling `.validateBody()` mandatory for non-GET routes, but how?

Well, we need some more **type-level state**! When calling `.handle()` we would like to know whether `.validateBody()` has been previously called.

We could do that by introducing a **second** type parameter:

```
type Route<M extends Method, IsValid extends boolean> = {
  // ...     👇                👆
  handle: IsValid extends true
    ? (fn: HandlerFn) => Route<M, IsValid>
    : never;
  // ...
};
```

The new `IsValid` parameter would allow us to turn the `handle` method **on** and **off**. It would start at `false` and when calling `.validateBody()`, we would flip it to `true`:

```
type Route<M extends Method, IsValid extends boolean> = {
  // ...
  validateBody: M extends "GET"
    ? never
    : (isValid: (body: unknown) => unknown) => Route<M, true>;
  // ...                                                  👆
};
```

And that would work just fine!

```
import { route } from "routing-library@v2.2";

route("POST /products")
  .handle(() => "..."); // ❌

route("POST /products")
  .validateBody(isProduct)
  .handle(() => "..."); // ✅
```

But this approach **does not scale**.

We are not done yet with our Type Builder pattern let me tell you! There are plenty of features we would like to add, and we want a solution that doesn't force us to thread new type parameters into our `Route` type all over our codebase every time we feel like tracking some more type-level information!

My favorite approach is to **wrap** all parameters into a single **object** type:

```
type RouteStateConstraint = {
  method: Method;
  isValid: boolean;
};

type Route<State extends RouteStateConstraint> = {
  //        👆
  // ...
};
```

We use a ***type constraint*** to describe the shape of our state, and each parameter becomes a property!

We can then access these properties using the `["prop"]` syntax:

```
type Route<State extends RouteStateConstraint> = {
  // ...              👇
  validateBody: State["method"] extends "GET"
    ? never
    : // ...
  // ...
};
```

To update the state, we can reach for the `Assign` type helper that we discovered way back in Chapter 3.

```
// Our trusty `Assign` helper:
```

```ts
type Assign<A, B> = Omit<A, keyof B> & B;

type Route<State extends RouteStateConstraint> = {
  // ...
  validateBody: State["method"] extends "GET"
    ? never
    : (predicate: ...) => Route<
        Assign<State, { isValid: true }>
        /*                 👆
              We assign `isValid` to `true`.     */
      >;
  // ...
};
```

`Assign` works exactly like `{ ...a, ...b }`, but for type-level objects.

Let's also update `handle`:

```ts
type Route<State extends RouteStateConstraint> = {
  // ...             👇
  handle: State["isValid"] extends false
    ? never
    : (handler: ...) => Route<State>;
  // ...
};
```

As well as our `route` function:

```ts
type GetDefaultRouteState<M extends Method> = {
  method: M;
  isValid: M extends "GET" ? true : false;
  /*                   👆
              GET routes don't need
                 validation!                */
};

export function route<const P extends string>(
  path: ValidateRoute<P>,
): Route<GetDefaultRouteState<GetMethod<P>>>;
//              👆
```

▼🤔 **Can I take a look at the full code?**

Here you go!

routing-library@v2.3.ts

```ts
/**
 * Our route function type:
 */

export declare function route<const P extends string>(
  path: ValidateRoute<P>,
): Route<InitialRouteState<GetMethod<P>>>;

type RouteStateConstraint = {
  method: Method;
  isValid: boolean;
};

type InitialRouteState<M extends Method> = {
  method: M;
  isValid: M extends "GET" ? true : false;
} & {};

type Route<State extends RouteStateConstraint> = {
```

Let's give this a try!

```ts
import { route } from "routing-library@v2.3";

route("POST /products")
  .validateBody(isProduct) // 👈 Try commenting this line.
  .handle(() => "...");
```

```
route("GET /products")
  // .validateBody(isProduct) // 👈 Try UN-commenting this line.
  .handle(() => "...");
```

It's working like a charm!

Another benefit of wrapping parameters in an object is that a much nicer type gets displayed on hover:

```
import { route } from "routing-library@v2.3";

export const postRoute = route("POST /products");
//   hover here 👆
```

Since our parameters have names now, their purposes become much more obvious.

```
// Our new display type ✨
const postRoute: Route<{
  method: "POST";
  isValid: false;
}>;
```

Neat!

Improving our errors

Alright, now that we know how to dynamically update our APIs, I think we should do a quick pass at improving our error messages:

```
This expression is not callable. Type 'never' has no call signatures.
```

It's clearly not up to our new standards! 😠 Fortunately, we already know how to improve them — use **String Literals** as **error messages**!

Let's create two new errors:

```
type GETBodyError =
  ".validateBody( ... ) isn't available on GET routes.";

type UnvalidatedBodyError<M extends Method> =
  `You must call .validateBody( ... ) before .handle( ... ) on ${M} routes.`;
```

And use them in our `Route` type:

```
type Route<State extends RouteStateConstraint> = {
  // ...
  validateBody: State["method"] extends "GET"
    ? GETBodyError
    : // ...
  // ...
  handle: State["isValid"] extends false
    ? UnvalidatedBodyError<State["method"]>
    : // ...
};
```

And here is what we get 👇

```
import { route } from "routing-library@v2.4";

route("POST /products")
  // .validateBody(isProduct)
  .handle(() => "...");

route("GET /products")
  .validateBody(isProduct)
  .handle(() => "...");
```

We are once again hacking our way around TypeScript errors to provide developers with **more actionable feedback** about their code!

```
handle: "You must call .validateBody( ... ) before .handle( ... ) on POST routes."
```

```
validateBody: ".validateBody( ... ) isn't available on GET routes."
```

Since we replaced the whole method with an error, notice that these messages get displayed exactly where you would expect — **under the method name**. This should help people understand what's wrong much faster.

▼🤔 **Is there a way to completely filter out invalid methods instead?**

As you probably noticed, we aren't *actually* removing invalid methods from our objects, just turning them into errors. The slight downside with this technique is that invalid methods **will still show up** in **suggestions** when accessing route methods:

```ts
import { route } from "routing-library@v2.4";

const someRoute = route("GET /products");

someRoute; // 👈 try typing `.` after `someRoute`
           //    and select `validateBody`.
```

It's kind of too bad to invite users to select `validateBody` even though we know it will error.

If we wanted to actually remove these properties from our object, we could use the `Omit` built-in helper.

```ts
//                                                  👇
type Route<State extends RouteStateConstraint> = Omit<
  {
    validateBody: // ...
    handle: // ...
  },
  KeysToOmit<State>
  //   👆
>;
```

The tricky bit is generating the list of methods we ***don't*** want in function of our state. The simplest is to `|` together several conditional types:

```ts
type KeysToOmit<State extends RouteStateConstraint> =
  | (State["method"] extends "GET" ? "validateBody" : never)
  | (State["isValid"] extends false ? "handle" : never);
```

As you know, `never` will get filtered out from the resulting union, leaving only the keys of the methods we want to omit.

Let's give it another try:

```ts
import { route } from "routing-library@v2.4-using-omit";

const someRoute = route("GET /products");

someRoute; // 👈 try typing `.` after `someRoute`
```

`validateBody` no longer shows up in our list of suggestions! 🎉

But here is the downside:

```ts
import { route } from "routing-library@v2.4-using-omit";

route("POST /products")
  // hover here
```

```
  //  👇
  .handle(() => "...");
```

Since TypeScript no longer knows about `handle`, we can't show a nice little error message when it's being used incorrectly anymore. We get this instead:

```
Property 'handle' does not exist on type 'Route<InitialRouteState<"POST
/products">>'
```

It's a missed opportunity to explain ***why*** `handle` cannot be called in this case.

Heh, tradeoffs.

Type builder takeaways 📚

Here is what you should remember:

- The Type Builder pattern allows us to guide users by **dynamically changing our APIs**.
- We can track **type-level state** and modify it on every method.
- We can read this state to conditionally remove methods or change their parameters.
- Custom error messages work great for methods too.
- Using an **object type** is the most scalable way to keep type-level state.

Wrapping several type parameters into a single object is something to keep in mind, even outside of the builder pattern. In his great talk "5 Years of Building React Table", **Tanner Linsley** refers to this pattern as ***the "Generic bag"*** and says it reduced the number of lines of code of `react-table`'s type definitions by **70%**. Impressive!

Challenges 🕹️

As we approach the end of this chapter, I'll let you in the driver's seat 😎 Let's make some additional improvements to our `routing-library` before we press the publish button!

Challenge 1/3

Let's start by improving the type of our `body` parameter. If developers pass a **type predicate function** to `.validateBody` like this one:

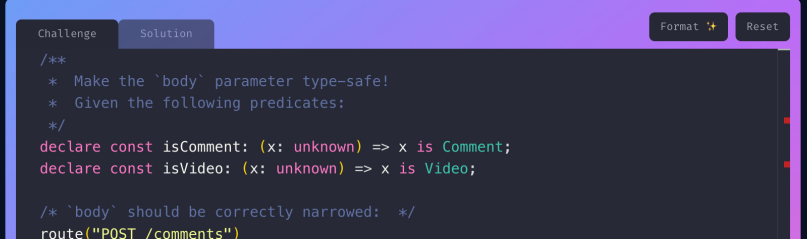
```
const isComment = (body): body is Comment => {
  /*                          👇
      isComment is a type predicate because
      its return type looks like `value is Type`.

      ... return true if `body` is a comment.     */
};
```

We should pass a `body` of type `Comment` to the handler function:

```
route("POST /comments")
  // We pass it here 👇
  .validateBody(isComment)
  .handle(({ body }) => {
    //        👆 should be of type `Comment`.
  });
```

Keep in mind that you can always take a peek at the solution if you get stuck 🙂

Let's go!



```
  .validateBody(isComment)
  .handle(({ body }) => {
    type test = Expect<Equal<typeof body, Comment>>;
    return "✅";
  });

route("PUT /video/:id")
  .validateBody(isVideo)
  .handle(({ body }) => {
    type test = Expect<Equal<typeof body, Video>>;
    return "✅";
  });

/**
 * Modify the `route` function and its methods
 * to keep track of the type of `body`! 👇
 */
```

Challenge 2/3

Let's close another GitHub issue. At the beginning of this chapter, I showed how people could use `.isAuthenticated()` and `.isAdmin()` to check user **permissions** on their routes:

```
import { route } from "routing-library@v2.4";

export const details = route("GET /tricks/:trickId/details")
  .isAuthenticated() // returns ⛔ if unauthenticated
  .handle(() => `🎛️`);

export const admin = route("GET /admin/dashboard")
  .isAuthenticated()
  .isAdmin() // returns 👮 to non-admins
  .handle(() => `📈`);
```

Welllll... there are still a few problems with these two methods.

The `.isAdmin()` method requires the user to be authenticated to work, so we should make sure that `.isAuthenticated()` always gets called before we can call `.isAdmin()`.

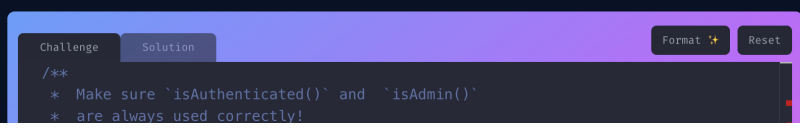
```
import { route } from "routing-library@v2.4";

const admin = route("GET /admin/dashboard")
  .isAdmin() // 💥 This throws
  .handle(() => `...`);

const admin = route("GET /admin/dashboard")
  .isAuthenticated()
  .isAdmin() // ✅ but this is ok!
  .handle(() => `...`);
```

In addition, nothing stops you from calling `.isAuthenticated()` or `.isAdmin()` **repeatedly** currently, even though it doesn't make any kind of sense:

```
import { route } from "routing-library@v2.4";

export const admin = route("GET /admin/dashboard")
  // this type-checks 😬
  .isAuthenticated()
  .isAuthenticated()
  .isAuthenticated()
  .isAdmin()
  .isAdmin()
  .isAdmin()
  // We should be safe by now 😅
  .handle(() => `...`);
```

Let's make sure:

- `.isAuthenticated()` and `.isAdmin()` can be called **exactly once**.
- `.isAdmin()` is allowed **only if** `.isAuthenticated()` has been called first.



```
 */

route("GET /tricks/:trickId/details")
  .isAuthenticated()
  // @ts-expect-error ❌ already authenticated.
  .isAuthenticated()

route("GET /admin/dashboard")
  // @ts-expect-error ❌ not authenticated.
  .isAdmin()

route("GET /admin/dashboard")
  .isAuthenticated()
  .isAdmin()
  // @ts-expect-error ❌ already admin.
  .isAdmin()

route("GET /admin/dashboard")
  .isAuthenticated()
  .isAdmin()
  .handle(() => `📈`); // ✅
```

Challenge 3/3 🧠

Before we feel ready to release `routing-library@v3` to our users, there's **one final improvement** we'd like to make.

TrickTok 🛹 is rapidly growing, and the team is shipping features like crazy. That means our router is starting to have a ***lot*** of routes:

```
const server = router()
  // ... ui routes
  .add(home)
  .add(feed)
  .add(trickVideo)
  .add(userProfile)
  // ... api routes
  .add(newTrickVideo)
  .add(newComment)
  // ... 200 other routes later ...
  .add(notFound);

server.start(3000);
```

One day, one of our coworkers trying to add a new route tells us that we might have reached the limit. She's building an endpoint to retrieve the followers of a skater:

```
// Our new route
//       👇
const listFollowers =
  route("GET /api/skater/:name/followers").handle(handler);

// server.ts
const server = router()
  // ... all our routes ...
  .add(listFollowers)
  .add(notFound);

server.start(3000);
```

But when she tries to fetch data from this new endpoint, it doesn't behave as expected:

```
// client.ts
const fetchFollowers = (name: string) =>
  fetch(`/api/skater/${name}/followers`);

await fetchFollowers("yuto")
```

She receives an error message coming from who knows where saying that no skater named `"yuto/followers"` exists in the database!

After a few minutes of debugging, it turned out the router was calling a completely different route:

```
/* The router was calling
```

```
    this route 👇           */
const skaterProfile =
  route("GET /api/skater/:name").handle(handler)

const listFollowers =
  route("GET /api/skater/:name/followers").handle(handler);

const router = router()
  // ...
  .add(skaterProfile) //    👈 Which is added
  // ...                        before `listFollowers`
  .add(listFollowers)
  .add(notFound);
```

Since route parameters like `:name` are wildcards, they can be assigned to any string, including `"yuto/followers"`! The new `listFollowers` route was being ***shadowed*** by `skaterProfile`.

With a list of routes as long as ours, it's getting increasingly difficult to know if a new route will be shadowed by an existing one. Could we solve this problem **with types**? Could we somehow detect unreachable routes and reject them with a nice error message?

Ideally, we would like **equivalent paths** to be rejected:

```
const router1 = router()
  .add(route("GET /user/:username").handle(handler))
  .add(route("GET /user/:name").handle(handler));
  //     👆 ❌ shouldn't type-check

const router2 = router()
  .add(route("GET /user/:username").handle(handler))
  .add(route("PUT /user/:name").handle(handler));
  //          👆 ✅ using a different method is allowed.
```

As well as **unreachable** routes that are being **shadowed** by another one.

```
const router1 = router()
  .add(route("GET /user/:username").handle(handler))
  .add(route("GET /user/:username/followers").handle(handler));
  //     👆 ❌ shouldn't type-check

const router2 = router()
  // ✅ Reversing the order fixes it.
  .add(route("GET /user/:username/followers").handle(handler))
  .add(route("GET /user/:username").handle(handler));
```

But how do we go about implementing this? 🤔

Part 1 — Detecting unreachable routes

Since this challenge is significantly more complicated than previous ones, **let's break it down into two parts**.

The first thing we will clearly need is a way to **detect** if a route is shadowing another one:

```
type IsShadowing<Path1, Path2> = //  ...  ?
```

This helper should return `true` if `Path1` shadows `Path2`, and `false` otherwise:

```
type T1 = IsShadowing<
  "GET /user/:username",
  "GET /user/:username/followers"
>; // => true

type T2 = IsShadowing<
  "GET /user/:username/followers",
  "GET /user/:username"
>; // => false
```

So let's implement this `IsShadowing` helper as a first step!

▼ SPOILER 🧯 additional hints to implement IsShadowing.

In order to detect shadowing, you can use the **assignability** of **Template Literal Types**!

If you replace route parameters like `:username` with the `string` type, then detecting if `Path1` shadows `Path2` is as simple as checking if `Path2` is assignable to `Path1`:

```
type UserProfile = `GET /user/${string}`;
type Followers = `GET /user/${string}/followers`;

type IsShadowing<Path1, Path2> = Path2 extends Path1 ? true : false;

type test1 = IsShadowing<UserProfile, Followers>; // => true
type test2 = IsShadowing<Followers, UserProfile>; // => false
```

Challenge | Solution — Format ✨ — Reset

```
/**
 * Implement `IsShadowing`:
 */
export type IsShadowing<Path1, Path2> = TODO;

/**
 * Does shadow:
 */
type res1 = IsShadowing<"GET /", "GET /">;
type test1 = Expect<Equal<res1, true>>;

type res2 = IsShadowing<"GET /user/:name", "GET /user/:username">;
type test2 = Expect<Equal<res2, true>>;

type res3 = IsShadowing<"GET /user/:name", "GET /user/:name/profile">;
type test3 = Expect<Equal<res3, true>>;

/**
 * Does not shadow:
 */
```

Part 2 — Rejecting unreachable routes

Now that we know how to detect unreachable routes, let's update our router to reject them with the following error:

```
type UnreachableRouteError<ShadowPath extends string> =
  `This route is being shadowed by '${ShadowPath}'.`;
```

Note that we also need to find the "shadower" route to display its path in our error message. This will make debugging so much easier for our users 🥰

```
This route is being shadowed by '/api/skater/:name'.
```

Let's go!

Challenge | Solution — Format ✨ — Reset

```
/**
 * Return the following error message from `router.add()`
 * if the route is unreachable.
 */
type UnreachableRouteError<ShadowPath extends string> =
  `This route is being shadowed by '${ShadowPath}'.`;

/**
 * Remember that `IsShadowing` is in scope:
 */
type IsShadowing<Path1, Path2> =
  ParamsToStrings<Path2> extends ParamsToStrings<Path1> ? true : false;

namespace testCases {
  const handler = () => "...";

  const router1 = router()
    .add(route("GET /", handle(handler)))
```

```
  // @ts-expect-error ❌
  .add(route("GET /").handle(handler));

const router2 = router()
  .add(route("GET /user/:username").handle(handler))
  // @ts-expect-error ❌
  .add(route("GET /user/:name").handle(handler));
```

Conclusion

I hope to have convinced you that **API design is a critical part of programming**. It is thanks to the careful design of thousands of library authors that millions of us, developers, were able to build these invaluable apps and products that everyone takes for granted today.

Even though **type safety** is a huge aspect of API design, **it is not the only one**! If our smart types yield errors that no one can understand, we failed to design the great abstraction we were envisioning.

During this chapter, we've seen how to use string literal types to get the compiler to display our own **custom error messages**. We've also discovered how this same technique could be used to provide **auto-complete suggestions**. Finally, we learned to use the ***Type Builder Pattern*** to infer as much information as possible, dynamically change our APIs, and guide our users toward working code!

Happy typing ✌️

Footnotes

1. We went pretty close to a world in which TypeScript would have had negative types. This interesting proof of concept from the TypeScript maintainer Wesley Wigham was opened in 2019 but was unfortunately never merged, losing to higher-priority features. Maybe someday! ↩

⟸ Previous
10. TypeScript Assignability Quiz

Next ⟹
12. Conclusion