Chapter 9. Frontend and Backend Frameworks

While you could build every part of your application yourself from the ground up—the networking and database layers on the server, a user interface framework and state management solution on the frontend—you probably shouldn't. It's hard to get the details right, and luckily for us, lots of these hard problems on the frontend and backend have already been solved by other engineers. By taking advantage of existing tools, libraries, and frameworks to build things both on the frontend and the backend, we can iterate quickly and on stable ground when building our own applications.

In this chapter, we'll go through some of the most popular tools and frameworks that solve common problems on both the client and the server. We'll talk about what you might use each framework for, and how to safely integrate it into your TypeScript application.

Frontend Frameworks

TypeScript is a natural fit for the world of frontend applications. With its rich support for JSX and its ability to safely model mutability, TypeScript lends structure and safety to your application and makes it easier to write correct, maintainable code in the fast-paced environment that is frontend development.

Of course, all of the built-in DOM APIs are typesafe. To use them from TypeScript, just include their type declarations in your project's *tsconfig.json*:

```
{
  "compilerOptions": {
    "lib": ["dom", "es2015"]
  }
}
```

That will tell TypeScript to include *lib.dom.d.ts*—its built-in browser and DOM type declarations—when typechecking your code.

NOTE

The `lib` *tsconfig.json* option just tells TypeScript to include a set of specific type declarations when processing the code in your project; it won't emit any extra code, or generate any JavaScript that will exist at runtime. It won't, for example, make the DOM magically work in a NodeJS environment (your code will compile, but it will fail at runtime)—it's on you to make sure that your type declarations match up to what your JavaScript environment actually supports at runtime. Jump ahead to "Building Your TypeScript Project" to learn more.

With DOM type declarations enabled, you'll be able to safely consume DOM and browser APIs to do things like:

```
// Read properties from the global window object
let model = {
  url: window.location.href
}

// Create an <input /> element
let input = document.createElement('input')

// Give it some CSS classes
input.classList.add('Input', 'URLInput')

// When the user types, update the model
input.addEventListener('change', () =>
  model.url = input.value.toUpperCase()
)

// Inject the <input /> into the DOM
document.body.appendChild(input)
```

Of course, all of that code is typechecked and comes with the normal goodies like in-editor autocompletion. For example, consider something like this:

```
document.querySelector('.Element').value // Error TS233
                                         // not exist c
```

TypeScript will throw an error because the return type of `querySelector` is nullable.

While for simple frontend applications these low-level DOM APIs are enough and will give you what you need to do safe, type-guided programming for the browser, most real-world frontend applications use a framework to abstract away how DOM rendering and rerendering, data binding, and events work. The following sections will give some pointers on how to effectively use TypeScript with a few of the most popular browser frameworks.

React

React is among the most popular frontend frameworks today, and is a great choice when it comes to type safety.

The reason React is so safe is because React components—the basic building blocks of React applications—are both defined and consumed in TypeScript. This property is hard to find among frontend frameworks, and means that both component definitions and consumers are typechecked. You can use types to say things like "this component takes a user ID and a color" or "this component can only have list items as children." These constraints are then enforced by TypeScript, verifying that your components do what they say they do.

This safety around component definitions and consumers—the *view layer* of a frontend application—is killer. The view is traditionally the place where typos, missed attributes, mistyped parameters, and improperly nested elements cause programmers to collectively spend thousands of hours tearing their hair out and indignantly refreshing their browsers. The day you start typing your views with TypeScript and React is the day you double your and your team's productivity on the frontend.

A JSX primer

When using React, you define your views using a special DSL called *JavaScript XML* (JSX) that you embed straight into your JavaScript code. It sort of looks like HTML in your JavaScript. You then run your JavaScript through a JSX compiler that rewrites that funky JSX syntax into regular JavaScript function calls.

The process looks something like this. Say you're building a menu app for your friend's restaurant, and you list out a few items on the brunch menu with the following JSX:

```
<ul class='list'>
  <li>Homemade granola with yogurt</li>
  <li>Fantastic french toast with fruit</li>
  <li>Tortilla Española with salad</li>
</ul>
```

After running that code through a JSX compiler like Babel's `transform-react-jsx` plugin, you'll get the following output:

```
React.createElement(
  'ul',
  {'class': 'list'},
  React.createElement(
    'li',
    null,
    'Homemade granola with yogurt'
  ),
  React.createElement(
    'li',
    null,
    'Fantastic French toast with fruit'
  ),
  React.createElement(
    'li',
    null,
    'Tortilla Española with salad'
  )
);
```

TSC FLAG: ESMODULEINTEROP

Because JSX compiles to a call to `React.createElement`, be sure to import the React library into each file where you use JSX so that you have a variable named `React` in scope:

```
import React from 'react'
```

Don't worry—if you forget, TypeScript will warn you:

```
<ul /> // Error TS2304: Cannot find name 'React'.
```

Also note that I've set `{"esModuleInterop": true}` in my *tsconfig.json* to support importing `React` without a wildcard (`*`) import. If you're following along, either enable `esModuleInterop` in your own *tsconfig.json*, or use a wildcard import instead:

```
import * as React from 'react'
```

The nice thing about JSX is you can write what looks a lot like normal HTML, then compile it automatically to a JavaScript engine–friendly format. As an engineer you only use a familiar, high-level, declarative DSL, and you don't have to deal with the implementation details.

You don't need JSX to work with React (you can write that compiled code directly and it'll work fine), and you can use JSX without React (the specific function call that JSX tags compile to— `React.createElement` in the previous example—is configurable), but the combination of React with JSX is magical, and makes writing views really fun, and really, really safe.

TSX = JSX + TypeScript

Files that contain JSX use the file extension *.jsx*. And TypeScript files that contain JSX use the *.tsx* extension. TSX is to JSX what TypeScript is to JavaScript—a compile-time safety and assistance layer to help you be more productive and produce code with fewer mistakes. To enable TSX support for your project, add the following line to your *tsconfig.json*:

```
{
  "compilerOptions": {
    "jsx": "react"
  }
}
```

The `jsx` directive has three modes at the time of writing:

`react`
: Compile JSX to a *.js* file using the JSX pragma (by default, `React.createElement`).

`react-native`
: Preserve JSX without compiling it, but do emit a file with a *.js* extension.

`preserve`
: Typecheck JSX but don't compile it away, and emit a file with a *.jsx* extension.

Under the hood, TypeScript exposes a few hooks for typing TSX in a pluggable way. These are special types on the `global.JSX` namespace that TypeScript looks at as the source of truth for TSX types throughout your program. If you're just using React, you don't need to go that low-level; but if you're building your own TypeScript library that uses TSX (and doesn't use React)—or if you're curious how the React type declarations do it—head over to Appendix G.

Using TSX with React

React lets us declare two kinds of components: function components and class components. Both kinds of components take some properties and render some TSX. From a consumer's point of view, they are identical.

Declaring and rendering a function component looks like this:

```
import React from 'react'
                          ❶

type Props = {
                          ❷
```

```
  isDisabled?: boolean
  size: 'Big' | 'Small'
  text: string
  onClick(event: React.MouseEvent<HTMLButtonElement>):
                                                    ❸

}

export function FancyButton(props: Props) {
                                                    ❹

  const [toggled, setToggled] = React.useState(false)
                                                    ❺

  return <button
    className={'Size-' + props.size}
    disabled={props.isDisabled || false}
    onClick={event => {
      setToggled(!toggled)
      props.onClick(event)
    }}
  >{props.text}</button>
}

let button = <FancyButton
                                                    ❻

  size='Big'
  text='Sign Up Now'
  onClick={() => console.log('Clicked!')}
/>
```

We have to bring the `React` variable into the current scope in order to use TSX with React. Since TSX is compiled to `React.createElement` function calls, that means we need to import `React` so that it's defined at runtime. ❶

We start by declaring the specific set of props we can pass to our `FancyButton` component. `Props` is always an object type, and is named `Props` by convention. For our `FancyButton` component, `isDisabled` is optional, while the rest of our props are required. ❷

React has its own set of wrapper types for DOM events. When using React events, be sure to use React's event types rather than regular DOM event types. ❸

A function component is just a regular function that has up to one

A function component is just a regular function that has up to one parameter (the `props` object) and returns a React-renderable type. React is permissive and can render a wide range of types: TSX, strings, numbers, booleans, `null`, and `undefined`.

We use React's `useState` hook to declare local state for a function component. `useState` is one of a handful of hooks available in React, which you can combine to create your own custom hooks. Note that because we passed the initial value `false` to `useState`, TypeScript was able to infer that the piece of state is a `boolean`; if we'd instead used a type that TypeScript wasn't able to infer—for example, an array—we would have bound the type explicitly (e.g., with useState<number[]>;([])).

We use TSX syntax to create an instance of `FancyButton`. The `<FancyButton />` syntax is almost identical to calling `FancyButton`, but it lets React manage the lifecycle of `FancyButton` for us.

That's it. TypeScript enforces that:

- JSX is well formed. Tags are closed and properly nested, and tag names aren't misspelled.
- When we instantiate a `<FancyButton />` we pass all required—plus any optional—props to `FancyButton` (`size`, `text`, and `onClick`), and that the props are all correctly typed.
- We don't pass any extraneous props to `FancyButton`, just the ones that are required.

A class component is similar:

```
import React from 'react'

import {FancyButton} from './FancyButton'

type Props = {

  firstName: string
  userId: string
}

type State = {
```

```
  isLoading: boolean
}

class SignupForm extends React.Component<Props, State>
                                                    ❹

  state = {
                                                    ❺
    isLoading: false
  }
  render() {
                                                    ❻
    return <>
                                                    ❼
      <h2>Sign up for a 7-day supply of our tasty
          toothpaste now, {this.props.firstName}.</h2>
      <FancyButton
        isDisabled={this.state.isLoading}
        size='Big'
        text='Sign Up Now'
        onClick={this.signUp}
      />
    </>
  }
  private signUp = async () => {
                                                    ❽

    this.setState({isLoading: true})
    try {
      await fetch('/api/signup?userId=' + this.props.us
    } finally {
      this.setState({isLoading: false})
    }
  }
}

let form = <SignupForm firstName='Albert' userId='13ab9
                                                    ❾
```

❶ Like before, we import `React` to bring it into scope.

❷ Like before, we declare a `Props` type to define what data we need to pass in when creating an instance of `<SignupForm />`.

❸ We declare a `State` type to model our component's local state.

To declare a class component, we extend the `React.Component`

To declare a class component, we extend the `React.Component` base class.

We use a property initializer to declare default values for local state.

Like with function components, a class component's `render` method returns something renderable by React: TSX, a string, a number, a boolean, `null`, or undefined.

TSX supports fragments using the special `<>...</>` syntax. A fragment is a nameless TSX element that wraps other TSX, and is a way to avoid rendering extra DOM elements in places where you need to return a single TSX element. For example, a React component's `render` method needs to return a single TSX element; to do that, we could have wrapped our code with a `<div>` or any other element, but that would have incurred unnecessary overhead during rendering.

We define `signUp` using an arrow function, to make sure that `this` in the function doesn't get re-bound.

Finally, we instantiate our `SignupForm`. Like when instantiating function components, we could have directly `new`-ed it with `new SignupForm({firstName: 'Albert', userId: '13ab9g3'})` instead, but that would mean that React couldn't manage the `SignupForm` instance's lifecycle for us.

Notice how we mix and match value-based (`FancyButton` , `SignupForm`) and intrinsic (`section` , `h2`) components in this example. We put TypeScript to work to verify things like:

- That all required state fields were defined either in the `state` initializer, or in the constructor
- That whatever we access on `props` and `state` actually exists, and is of the type we think it is
- That we don't write to `this.state` directly, because in React, state updates have to go through the `setState` API
- That calling `render` really returns some JSX

With TypeScript you can make your React code safer, and become a better, happier person as a result.

NOTE

We didn't use React's `PropTypes` feature, which is a way to declare and check props' types at runtime. Since TypeScript is already checking types for us at compile time, we don't need to do it again.

Angular

Contributed by Shyam Seshadri

Angular is a more fully featured frontend framework than React, and comes with support not just for rendering views but also for sending and managing network requests, routing, and dependency injection. It's built from the ground up to work with TypeScript (in fact, the framework itself is written in TypeScript!).

Central to the way Angular works is the Ahead-of-Time (AoT) compiler built into Angular CLI, Angular's command-line utility, that grabs the type information you gave it with your TypeScript annotations and uses that information to compile your code down to regular JavaScript. Instead of calling TypeScript directly, Angular applies a whole bunch of optimizations and transformations to your code before ultimately delegating to TypeScript and compiling it down to JavaScript.

Let's see how Angular uses TypeScript and its AoT compiler to make writing frontend applications safe.

Scaffolding

To initialize a new Angular project, start by globally installing Angular CLI using NPM:

```
npm install @angular/cli --global
```

Then, use Angular CLI to initialize a new Angular application:

```
ng new my-angular-app
```

Follow the prompts, and Angular CLI will set up a bare-bones Angular application for you.

In this book we won't go into depth on how an Angular application is structured, or how to configure and run it. For detailed information, head over to the official Angular documentation.

Components

Let's build an Angular component. Angular components are like React components, and include a way to describe a component's DOM structure, styling, and controller. With Angular, you generate component boilerplate with Angular CLI, then fill in the details by hand. An Angular component consists of a few different files:

- A template, which describes the DOM a component renders
- A set of CSS styles
- A component class, which is a TypeScript class that dictates your components' business logic

Let's start with the component class:

```
import {Component, OnInit} from '@angular/core'

@Component({
  selector: 'simple-message',
  styleUrls: ['./simple-message.component.css'],
  templateUrl: './simple-message.component.html'
})
export class SimpleMessageComponent implements OnInit {
  message: string
  ngOnInit() {
    this.message = 'No messages, yet'
  }
}
```

For the most part, this is a pretty standard TypeScript class, with just a few differences that bring out how Angular leverages TypeScript. Namely:

- Angular's lifecycle hooks are available as TypeScript interfaces—just declare which ones you `implement` (`ngOnChanges` , `ngOnInit` ,

etc.). TypeScript then enforces that you implement methods that comply with the lifecycle hooks you want. In this example we implemented the `OnInit` interface, which requires that we implement the `ngOnInit` method.

- Angular makes heavy use of TypeScript decorators (see "Decorators") to declare metadata related to your Angular components, services, and modules. In this example, we used a `selector` to declare how people can consume our component, and we used `templateUrls` and `styleUrl` to link an HTML template and CSS stylesheet to our component.

TSC FLAG: FULLTEMPLATETYPECHECK

To enable typechecking for your Angular templates (you should!), be sure to enable `fullTemplateTypeCheck` in your *tsconfig.json*:

```
{
  "angularCompilerOptions": {
    "fullTemplateTypeCheck": true
  }
}
```

Note that `angularCompilerOptions` isn't specifying options for TSC. Rather, it defines compiler flags specific to Angular's AoT compiler.

Services

Angular comes with a built-in dependency injector (DI), which is a way for the framework to take care of instantiating services and passing them in as arguments to components and services that depend on them. This can make it easier to instantiate and test services and components.

Let's update `SimpleMessageComponent` to inject a dependency, `MessageService`, responsible for fetching messages from the server:

```
import {Component, OnInit} from '@angular/core'
import {MessageService} from '../services/message.servi

@Component({
  selector: 'simple-message',
  templateUrl: './simple-message.component.html',
```

```
  styleUrls: ['./simple-message.component.css']
})
export class SimpleMessageComponent implements OnInit {
  message: string
  constructor(
    private messageService: MessageService
  ) {}
  ngOnInit() {
    this.messageService.getMessage().subscribe(response
      this.message = response.message
    )
  }
}
```

Angular's AoT compiler looks at the parameters that your component's `constructor` takes, plucks out their types (e.g., `MessageService`), and searches the relevant dependency injector's dependency map for a dependency of that specific type. It then instantiates that dependency (`new`-ing it) if it hasn't been instantiated yet, and passes it into the `SimpleMessageComponent` instance's constructor. All of this DI stuff is pretty complicated, but it can be convenient as your application grows and you have multiple dependencies you might use depending on how the app is configured (e.g., `ProductionAPIService` versus `DevelopmentAPIService`) or when testing it (`MockAPIService`).

Now let's take a quick look at how to define a service:

```
import {Injectable} from '@angular/core'
import {HttpClient} from '@angular/common/http'

@Injectable({
  providedIn: 'root'
})
export class MessageService {
  constructor(private http: HttpClient) {}
  getMessage() {
    return this.http.get('/api/message')
  }
}
```

Whenever we create a service in Angular, we again use TypeScript decorators to register it as something that is `Injectable`, and we define whether it is provided at the root level of the application or in a submodule. Here, we

registered the service `MessageService`, allowing us to inject it anywhere in our application. In the constructor of any component or service, we can just ask for a `MessageService` and Angular will magically take care of passing it in.

With how to safely use these two popular frontend frameworks out of the way, let's move on to typing the interface between your frontend and your backend.

Typesafe APIs

Contributed by Nick Nance

Regardless of which frontend and backend frameworks you decide to use, you'll want a way to safely communicate across machines—from client to server, server to client, server to server, and client to client.

There are a few competing tools and standards in this space. But before we explore what they are and how they work, let's think about how we might build our own solution, and what benefits and drawbacks it might have (we are engineers, after all).

The problem we want to solve is this: though our clients and servers might be 100% typesafe—bastions of safety—at some point they'll need to talk to each other over untyped network protocols like HTTP, TCP, or some other socket-based protocols. How might we make this communication typesafe?

A good starting point could be a typesafe protocol like the one we developed in "Typesafe protocols". It might look something like this:

```
type Request =
  | {entity: 'user', data: User}
  | {entity: 'location', data: Location}

// client.ts
async function get<R extends Request>(entity: R['entity
  let res = await fetch(/api/${entity})
  let json = await res.json()
  if (!json) {
    throw ReferenceError('Empty response')
  }
  return json
```

```
}

// app.ts
async function startApp() {
  let user = await get('user')  // User
}
```

You could build corresponding `post` and `put` functions to write back to your REST API, and add a type for each entity your server supports. On the backend, you'd then implement a corresponding set of handlers for each type of entity, reading from your database to return to the client whatever entity it asked for.

But what happens if your server isn't written in TypeScript, or if you aren't able to share your `Request` type between the client and server (leading to the two getting out of sync over time), or if you don't use REST (maybe you use GraphQL instead)? Or what if you have other clients to support, like Swift clients on iOS or Java clients on Android?

That's where typed, code-generated APIs come in. They come in a lot of flavors, each with libraries available in a bunch of languages (including TypeScript)—for example:

- Swagger for RESTful APIs
- Apollo and Relay for GraphQL
- gRPC and Apache Thrift for RPC

These tools rely on a common source of truth for both server and clients—data models for Swagger, GraphQL schemas for Apollo, Protocol Buffers for gRPC—which are then compiled into language-specific bindings for whatever language you might be using (in our case, that's TypeScript).

This code generation is what prevents your client and server (or multiple clients) from getting out of sync with each other; since every platform shares a common schema, you won't run into the case where you updated your iOS app to support a field, but forgot to press Merge on your pull request to add server support for it.

Diving into the details of each framework is out of scope for this book. Pick one for your project, and head over to its documentation to learn more.

Backend Frameworks

When you build an application that interacts with a database, you might start with raw SQL or API calls, which are inherently untyped:

```
// PostgreSQL, using node-postgres
let client = new Client
let res = await client.query(
  'SELECT name FROM users where id = $1',
  [739311]
) // any

// MongoDB, using node-mongodb-native
db.collection('users')
  .find({id: 739311})
  .toArray((err, user) =>
    // user is any
  )
```

With a bit of manual typing you can make these APIs safer and get rid of most of your `any`s:

```
db.collection('users')
  .find({id: 739311})
  .toArray((err, user: User) =>
    // user is any
  )
```

However, raw SQL APIs are still fairly low-level, and it's still easy to use the wrong type, or forget a type and accidentally end up with `any`s.

That's where *object-relational mappers* (ORMs) come in. ORMs generate code from your database schema, giving you high-level APIs to express queries, updates, deletions, and so on. In statically typed languages, these APIs are typesafe, so you don't have to worry about typing things correctly and manually binding generic type parameters.

When accessing your database from TypeScript, consider using an ORM. At the time of writing, Umed Khudoiberdiev's excellent TypeORM is the most complete ORM for TypeScript, and supports MySQL, PostgreSQL, Microsoft

SQL Server, Oracle, and even MongoDB. Using TypeORM, your query to get a user's first name might look like this:

```
let user = await UserRepository
  .findOne({id: 739311}) // User | undefined
```

Notice the high-level API, which is both safe (in that it prevents things like SQL injection attacks) and typesafe by default (in that we know what type `findOne` returns without having to manually annotate it). Always use an ORM when working with databases—it's more convenient, and it will save you from getting woken up at four in the morning because the `saleAmount` field is `null` because you updated it to `orderAmount` the night before and your coworker decided to run your database migration for you in anticipation of your pull request landing while you were out, but then around midnight your pull request failed even though the migration succeeded, and your sales team in New York woke up to realize that all your clients' orders were for exactly `null` dollars (this happened to… a friend).

Summary

In this chapter we've covered a lot: directly manipulating the DOM; using React and Angular; adding type safety to your APIs with tools like Swagger, gRPC, and GraphQL; and using TypeORM to safely interact with your database.

JavaScript frameworks change at a rapid pace, and by the time you read this, the specific APIs and frameworks described here may be on their way to becoming museum exhibits. Use your newfound intuition for *what problems typesafe frameworks solve* to identify places where you can take advantage of someone else's work to make your code safer, more abstract, and more modular. The big idea to take away from this chapter isn't what the best framework to use in the year 2019 is, but what sorts of problems can be better solved with frameworks.

With the combination of typesafe UI code, a typed API layer, and a typesafe backend, you can eliminate entire classes of bugs from your application, and sleep better at night as a result.