ГЛАВА 1

Настройка проекта

Итак, вы хотите начать работу с TypeScript! Главный вопрос: с чего начать? Вы можете интегрировать язык в свои проекты разными способами, и все они немного различаются в зависимости от потребностей вашего проекта. JavaScript работает во многих средах исполнения. Точно так же существует множество способов настроить и TypeScript, чтобы он соответствовал вашим потребностям.

В этой главе рассматриваются все возможности внедрения в ваш проект языка TypeScript как дополнения к JavaScript, которое делает возможными базовое автозавершение и индикацию ошибок, а также настройки для полнофункциональных приложений на Node.js и в браузере.

Инструментарий JavaScript — область с безграничными возможностями (некоторые говорят, что новая цепочка сборки JavaScript выпускается каждую неделю, почти так же часто, как и новые фреймворки). Поэтому в данной главе больше внимания уделяется тому, что можно сделать с помощью одного только компилятора TypeScript, без каких-либо дополнительных инструментов.

TypeScript предлагает все необходимое для транспиляции, за исключением возможности создавать минимизированные и оптимизированные пакеты для распространения в Интернете. С этой задачей справляются такие сборщики, как ESBuild (https://esbuild.github.io/) или Webpack (https://webpack.js.org/). Кроме того, существуют конфигурации, содержащие другие транспиляторы, например Babel.js (https://babeljs.io/), которые могут прекрасно работать с TypeScript.

Сборщики и другие транспиляторы не будут рассматриваться в этой главе. Обратитесь к их документации по добавлению TypeScript и используйте информацию из данной главы для правильной настройки конфигурации.

Будучи проектом с более чем десятилетней историей, TypeScript содержит некоторые старые фрагменты, от которых не может просто избавиться ради совместимости. Поэтому в данной главе мы рассмотрим современный синтаксис JavaScript и последние разработки в области веб-стандартов.

Если вам все еще нужно использовать Internet Explorer 8 или Node.js 10, то, во-первых, извините, разрабатывать что-то для этих платформ очень сложно. А во-вторых, вы сможете собрать воедино компоненты для старых платформ, используя информацию из этой главы и официальной документации TypeScript (https://typescriptlang.org/).

1.1. Проверка типов JavaScript

Задача

Вы хотите выполнить базовую проверку типов для JavaScript, затратив наименьшее количество усилий.

Решение

Добавьте однострочный комментарий с `@ts-check` в начало каждого файла JavaScript, который хотите проверить. В правильных редакторах вы уже увидите красные волнистые линии, когда TypeScript сталкивается с чем-то не совсем подходящим.

Обсуждение

TypeScript был разработан как надмножество JavaScript, и каждый допустимый файл JavaScript также является допустимым файлом TypeScript. Это означает, что TypeScript очень хорошо справляется с определением потенциальных ошибок в обычном коде JavaScript.

Вы можете использовать его, если вам не нужна полноценная настройка TypeScript, но требуются базовые подсказки и проверки типов, чтобы упростить рабочий процесс разработки.

Наличие редактора или IDE — хорошее предварительное условие, если вы хотите только проверять типы в JavaScript.

Редактор, который отлично сочетается с TypeScript, — это Visual Studio Code (https://code.visualstudio.com/). VSCode был первым крупным проектом, в котором использовался TypeScript еще до выпуска этого языка.

Многие рекомендуют VSCode, если вы хотите писать на JavaScript или TypeScript. На самом деле любой редактор хорош, если в нем есть поддержка TypeScript. А в настоящее время большинство из них поддерживают его.

В Visual Studio Code при проверке типа JavaScript появляются красные волнистые линии, когда что-то не сходится (рис. 1.1). Это самый низкий порог входа в TypeScript. Система типов TypeScript имеет разные уровни строгости при работе с кодовой базой.

Сначала система типов будет пытаться *вывести* (infer) типы из кода JavaScript путем использования. Если в вашем коде есть строка наподобие этой:

```
let a_number = 1000;
```

то TypeScript правильно выведет `number` как тип `a_number`.

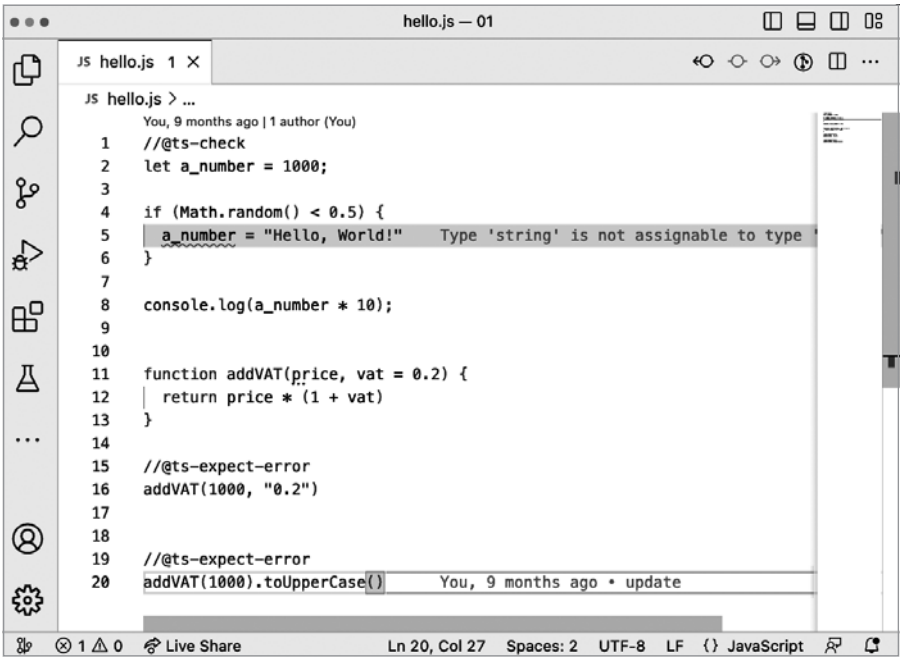


Рис. 1.1. Красные волнистые линии в редакторах кода: обратная связь первого уровня, если что-то в вашем коде не сходится

Одна из трудностей JavaScript состоит в том, что типы являются динамическими. Привязки через `let`, `var` или `const` могут менять тип в зависимости от использования[1].

Рассмотрим такой пример:

```
let a_number = 1000;

if (Math.random() < 0.5) {
  a_number = "Hello, World!";
}

console.log(a_number * 10);
```

Мы присваиваем число `a_number` и меняем привязку на `string`, если условие в следующей строке оценивается как `true`. Это не было бы большой проблемой, если бы мы не пытались умножить `a_number` в последней строке. Примерно в 50 % случаев данный пример приведет к нежелательному поведению.

[1] Объекты, имеющие привязку const, все еще могут изменять значения и свойства, а значит, и свои типы.

Здесь может помочь TypeScript. Добавив однострочный комментарий с `@ts-check` в самом верху нашего файла JavaScript, TypeScript активирует следующий уровень строгости: проверка типов в файлах JavaScript на основе информации о типе, доступной в файле JavaScript.

В нашем примере TypeScript обнаружит, что мы пытались присвоить привязке `string`, которую TypeScript вывел как число. В редакторе мы получим сообщение об ошибке:

```
// @ts-check
let a_number = 1000;

if (Math.random() < 0.5) {
  a_number = "Hello, World!";
// ^-- Type 'string' is not assignable to type 'number'.ts(2322)
}

console.log(a_number * 10);
```

Теперь мы можем приступить к исправлению кода, а TypeScript будет направлять нас.

Выведение типов для JavaScript имеет большое значение. В следующем примере TypeScript выводит типы, рассматривая такие операции, как умножение и сложение, а также значения по умолчанию:

```
function addVAT(price, vat = 0.2) {
  return price * (1 + vat);
}
```

Функция `addVat` принимает два аргумента. Второй является необязательным, так как для него установлено значение по умолчанию `0.2`. TypeScript выдаст предупреждение, если вы попытаетесь передать значение, которое не подходит:

```
addVAT(1000, "a string");
//           ^-- Argument of type 'string' is not assignable
//               to parameter of type 'number'.ts(2345)
```

Кроме того, поскольку мы используем операции умножения и сложения в теле функции, TypeScript понимает, что мы вернем число из этой функции:

```
addVAT(1000).toUpperCase();
//          ^-- Property 'toUpperCase' does not
//          exist on type 'number'.ts(2339)
```

В некоторых ситуациях требуется нечто большее, чем просто выведение типа. В файлах JavaScript вы можете аннотировать аргументы функций и привязки, используя аннотации типов JSDoc (https://jsdoc.app/). JSDoc — это соглашение о комментариях, которое позволяет вам описывать переменные и интерфейсы функций так, чтобы они были читабельны для людей и при этом их могли интерпретировать

машины. TypeScript возьмет ваши аннотации и будет использовать их в качестве типов для системы типов:

```
/** @type {number} */
let amount;

amount = '12';
//       ^-- Argument of type 'string' is not assignable
//           to parameter of type 'number'.ts(2345)

/**
 * Adds VAT to a price
 *
 * @param {number} price The price without VAT
 * @param {number} vat The VAT [0-1]
 *
 * @returns {number}
 */
function addVAT(price, vat = 0.2) {
  return price * (1 + vat);
}
```

JSDoc также позволяет определять новые сложные типы объектов:

```
/**
 * @typedef {Object} Article
 * @property {number} price
 * @property {number} vat
 * @property {string} string
 * @property {boolean=} sold
 */

/**
 * Now we can use Article as a proper type
 * @param {[Article]} articles
 */
function totalAmount(articles) {
  return articles.reduce((total, article) => {
    return total + addVAT(article);
  }, 0);
}
```

Однако синтаксис может показаться немного неуклюжим. В рецепте 1.3 мы рассмотрим более удачные способы аннотирования объектов.

Если у вас есть кодовая база JavaScript, которая хорошо документирована с помощью JSDoc, то добавление одной строки поверх ваших файлов действительно поможет вам, если в коде что-то пойдет не так.

1.2. Установка TypeScript

Задача

Красных волнистых линий в редакторе недостаточно: вам нужны обратная связь с командной строкой, коды состояния, конфигурация и параметры для проверки типа JavaScript и компиляции TypeScript.

Решение

Установите TypeScript через основной реестр пакетов Node: NPM (https://npmjs.com/).

Обсуждение

TypeScript написан на языке TypeScript[1], скомпилирован в JavaScript и использует среду выполнения JavaScript Node.js (https://nodejs.org/) в качестве основной среды выполнения[2]. Даже если вы не пишете приложения на Node.js, инструментарий для ваших приложений JavaScript будет работать на Node. Поэтому убедитесь, что скачали Node.js с официального сайта (https://nodejs.org/), и ознакомьтесь с его инструментами командной строки.

Приступая к новому проекту, обязательно инициализируйте папку проекта с помощью нового файла `package.json`. Он содержит всю информацию для Node и его менеджера пакетов NPM, которая позволяет определить содержимое вашего проекта. Создайте новый файл `package.json` с содержимым по умолчанию в папке вашего проекта, используя инструмент командной строки NPM:

```
$ npm init -y
```



В книге вы будете знакомиться с командами, которые должны выполняться в вашем терминале. Для удобства эти команды будут приводиться в том виде, в каком они представлены в BASH или аналогичных оболочках, доступных для Linux, macOS или подсистемы Windows для Linux. Начальный знак $ — условное обозначение команды, но вам не нужно его вводить. Обратите внимание, что все команды работают и в обычном интерфейсе командной строки Windows, а также в PowerShell.

---

[1] В программировании можно создать новый язык на основе существующего. А потом транслировать программы, написанные на новом языке, в код на старом языке. — *Примеч. пер.*

[2] TypeScript работает и в других средах исполнения JavaScript, таких как Deno и браузер, но они не являются главными целями.

NPM — это менеджер пакетов Node. Он поставляется с интерфейсом командной строки, реестром и другими инструментами, позволяющими устанавливать зависимости. После инициализации файла `package.json` установите TypeScript из NPM. В данном случае он устанавливается как зависимость разработки, то есть TypeScript не будет добавлен, если вы собираетесь опубликовать свой проект как библиотеку в самом NPM:

```
$ npm install -D typescript
```

Вы можете установить TypeScript глобально, чтобы компилятор TypeScript был доступен везде, но я настоятельно рекомендую устанавливать TypeScript отдельно для каждого проекта. В зависимости от того, насколько часто вы взаимодействуете со своими проектами, у вас будут разные версии TypeScript, которые будут синхронизированы с кодом вашего проекта. Глобальная установка (и обновление) TypeScript может привести к повреждению проектов, с которыми вы давно не работали.



Если вы устанавливаете зависимости фронтенда через NPM, то вам понадобится дополнительный инструмент, позволяющий убедиться, что ваш код работает и в браузере, — сборщик. TypeScript не содержит сборщик, работающий с поддерживаемыми системами модулей, поэтому вам необходимо настроить соответствующий инструмент. Часто используются такие инструменты, как Webpack (https://webpack.js.org/) и ESBuild (https://esbuild.github.io/). Все инструменты предназначены для выполнения TypeScript. Или же вы можете перейти на полную нативную версию (см. рецепт 1.8).

Теперь, когда TypeScript установлен, инициализируйте новый проект TypeScript. Используйте для этого NPX: он позволяет запускать утилиту командной строки, установленную для вашего проекта.

С помощью команды:

```
$ npx tsc --init
```

можно запустить локальную версию компилятора TypeScript вашего проекта и передать флаг `init` для создания нового файла `tsconfig.json`.

Файл `tsconfig.json` — это основной файл конфигурации для вашего проекта TypeScript. Он содержит все необходимые настройки, позволяющие TypeScript понимать, как интерпретировать ваш код, как сделать типы доступными для зависимостей и нужно ли включить или выключить определенные функции.

По умолчанию TypeScript устанавливает эти параметры для вас:

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "esModuleInterop": true,
```

```
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }
}
```

Рассмотрим их подробнее.

Значение `es2016` параметра `target` означает, что если вы запустите компилятор TypeScript, то он скомпилирует ваши файлы TypeScript в синтаксис, совместимый с ECMAScript 2016. В зависимости от поддерживаемых браузеров или среды вы можете установить либо более свежую версию (версии ECMAScript называются по году выпуска), либо что-то более старое, например es5, для тех, кто вынужден поддерживать очень старые версии Internet Explorer. Конечно, я надеюсь, что вам не придется этого делать.

Значение `commonjs` параметра `module` позволит вам писать, используя синтаксис модуля ECMAScript. Но вместо того, чтобы переносить этот синтаксис в вывод, TypeScript скомпилирует его в формат CommonJS. Это означает, что

```
import { name } from "./my-module";

console.log(name);
//...
```

становится

```
const my_module_1 = require("./my-module");
console.log(my_module_1.name);
```

после компиляции. CommonJS была системой модулей для Node.js и стала очень распространенной из-за популярности Node. С тех пор в Node.js появились и модули ECMAScript, которые мы рассмотрим в рецепте 1.9.

Параметр `esModuleInterop`, равный `true`, гарантирует, что модули, не являющиеся модулями ECMAScript, будут приведены в соответствие со стандартом после импортирования. Параметр `forceConsistentCasingInFileNames` поддерживает использование файловых систем, чувствительных к регистру, помогая взаимодействовать с теми, кто использует файловые системы, нечувствительные к регистру. А параметр `skipLibCheck` предполагает, что установленные вами файлы определения типов (подробнее об этом позже) не содержат ошибок. Поэтому ваш компилятор не станет проверять их и будет работать немного быстрее.

Одной из самых интересных особенностей является параметр `strict`, устанавливающий строгий режим TypeScript. Если задать для него значение `true`, то TypeScript будет вести себя по-другому в некоторых областях. Это позволяет команде разработки TypeScript определять, как должна вести себя система типов.

Если в TypeScript будут внесены критические изменения из-за меняющегося представления о системе типов, то они будут добавлены в строгий режим. Это означает,

что ваш код может быть поврежден, если вы обновите TypeScript и всегда будете работать в строгом режиме.

Чтобы дать вам время адаптироваться к изменениям, TypeScript позволяет включать или выключать определенные функции строгого режима по отдельности.

В дополнение к настройкам по умолчанию я настоятельно рекомендую использовать еще две:

```
{
  "compilerOptions": {
    //...
    "rootDir": "./src",
    "outDir": "./dist"
  }
}
```

Благодаря этому коду TypeScript получает указание забирать исходные файлы из папки src и помещать скомпилированные файлы в папку dist. Такая настройка позволяет отделить сгенерированные файлы от созданных вами. Конечно, вам придется создать папку src; папка dist будет создана после компиляции.

А вот и компиляция. Выполнив настройку проекта, создайте файл `index.ts` в папке src:

```
console.log("Hello World");
```

Расширение `.ts` указывает на то, что это файл TypeScript. Теперь запустите его, введя в командной строке эту команду:

```
$ npx tsc
```

и посмотрите, как работает компилятор.

1.3. Поддержка типов «в стороне»

Задача

Вы хотите писать обычный JavaScript без дополнительных этапов сборки, но при этом получить некую поддержку редактора и необходимую информацию о типах для ваших функций. Однако вы не хотите определять сложные типы объектов с помощью JSDoc, как было показано в рецепте 1.1.

Решение

Сохраните файлы с определениями типов «в стороне» и запустите компилятор TypeScript в режиме «проверка JavaScript».