GeekBrains

TypeScript

Введение в TypeScript

TypeScript v4.3

На этом уроке

1. Узнаем о предпосылках появления языка TypeScript как альтернативы JavaScript.
2. Настроим и запустим первый код на TypeScript.
3. Рассмотрим некоторые преимущества TypeScript на примерах кода.
4. Настроим ESLint чтобы он мог работать с TypeScript синтаксисом.

Оглавление

Почему TypeScript?

Скрипт - последовательность команд, инструкций на языке программирования, использующихся для автоматизации рутинных задач. Изначально JavaScript и разрабатывался как небольшой скриптовый язык, который мог позволить выполнять простые операции на стороне браузера для улучшения пользовательского опыта. Его возможности были весьма скромными и использовались с такими целями как: показать уведомление пользователю (alert), завести таймер, сделать бегущую строку или падающие снежинки. Основная же работа сайта лежала на стороне сервера, в том числе и генерация HTML-разметки.

Со временем на стороне браузера стало выполняться всё больше и больше работы: валидация форм, создание стилизованных модальных окон, карусели, слайдшоу и так далее. Для того, чтобы упростить взаимодействие с JavaScript и обеспечить поддержку между разными браузерами стали появляться различные библиотеки такие как Prototype, MooTools и jQuery.

Так со временем всё больше работы стало выполняться на стороне клиента, появились различные фреймворки. Помимо этого JavaScript стал использоваться для написания серверной части, CLI утилит и даже для мобильных и настольных программ.

Несмотря на то, что JavaScript прибавил в своих возможностях, фундаментально в нём мало что поменялось. Таким образом уровень возможностей языка остался на уровне простого скриптового языка, а уровень задач, которые на нём решают возрос многократно. Писать и поддерживать современные, промышленные приложения на JavaScript крайне сложно.

Ровно по этой причине был создан язык TypeScript. Он призван привнести недостающие в JavaScript возможности и компенсировать его недостатки. При этом TypeScript в конечном итоге компилируется в JavaScript, что делает возможным запускать его в любом браузере и в Node.js.

О каких именно недостатках JavaScript идёт речь и как TypeScript помогает их решить для наглядности мы будем рассматривать на примерах. Но для этого нам сначала понадобится сделать минимальный сетап проекта.

Создание TypeScript проекта

Давайте начнём проект на TypeScript. Создадим папку проекта, например **my-project** и проинициализируем в нём **packacke.json**. Для этого в консоли выполните следующие команды:

```
mkdir my-project
cd my-project
npm init
```

Теперь нам необходимо установить TypeScript в наш проект в качестве зависимости. Для этого откройте консоль и наберите:

```
npm install --save-dev typescript
```

Данная команда создаст папку **node_modules**, содержащую установленный **typescript** и файл **package-lock.json**, фиксирующий версии установленных зависимостей. Помимо этого в файл **package.json** создастся секция **devDependencies** с указанной версией **typescript**.

Теперь можно открыть папку проекта в редакторе кода. В корне проекта создайте папку **src** с файлом **index.ts**, напишите в нём по традиции:

index.ts

```
console.log('Hello World!')
```

Прежде чем идти дальше, убедитесь, что структура вашего проекта выглядит следующим образом:

```
.
├── node_modules
│   ├── .bin
│   │   ├── tsc -> ../typescript/bin/tsc
│   │   └── tsserver -> ../typescript/bin/tsserver
│   └── typescript # внутри большое количество подпапок
├── package-lock.json
├── package.json
├── src
│   └── index.ts
```

Мы не можем без дополнительных действий запускать TypeScript код. Любой код на TypeScript мы должны предварительно транспилировать в JavaScript, после чего запустить уже преобразованный код.

Внимание! *Преобразование TypeScript кода в JavaScript правильнее называть “транспиляцией”, а не “компиляцией”. Однако во многих источниках, включая данную методичку, можно будет встретить оба термина для описания одного и того же процесса. Для получения подробностей обратитесь к разделу “Глоссарий”.*

Для того чтобы транспилировать наш **index.ts** в **index.js** необходимо настроить сборку. Откроем файл **package.json** и изменим его следующим образом:

package.json:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc src/index.ts --outDir dist --target es2015",
    "start": "node .",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Barin Britva",
  "license": "ISC",
  "devDependencies": {
    "typescript": "^4.2.4"
  }
}
```

Мы добавили команду **build**, которая использует TypeScript компилятор **tsc** по пути **node_modules/.bin/tsc**. Компилятор на вход получит наш файл **src/index.ts**, а результат будет направлен директивой **outDir** в папку **dist**, которая создастся автоматически. Параметр **target** устанавливает версию ECMAScript, в которую будет преобразован наш код.

Также мы создали команду **start**, которая запускает приложение. Конструкция **node .** прочитает значение **main** из **package.json** и запустит указанный файл.

Поэтому мы изменили опцию **main**, указав путь к главному исполняемому файлу в соответствии с тем, где он будет располагаться после выполнения сборки - **dist/index.js**.

Давайте теперь соберём и запустим наш код. Для этого выполним в консоли следующее:

```
npm run build
npm start
```

В качестве результата в консоли появится сообщение:

```
Hello World!
```

На данный момент этого будет достаточно. Мы ещё вернёмся к настройкам позже в этом уроке. А пока давайте приступим к знакомству непосредственно с языком TypeScript.

Знакомство с языком

Рассмотрим некоторые базовые возможности TypeScript, которые сразу помогут продемонстрировать его лучшие стороны. Для того, чтобы примеры были более наглядными, будем сравнивать аналогичный фрагмент кода, написанный на JavaScript и на TypeScript.

Код будет содержать небольшой набор данных книг и функцию, подбирающую пользователю подходящую книгу по жанру и количеству страниц.

Создадим временный файл **src/playground.js**:

playground.js

```
class Book {
    constructor (name, genre, pageAmount) {
        this.name = name
        this.genre = genre
        this.pageAmount = pageAmount
    }
}

const books = [
    new Book('Harry Potter', 'fantasy', 980),
    new Book('Lord of the Ring', 'fantasy', 1001),
    new Book('How to be productive', 'lifestyle', 500),
    new Book('Game of Thrones', 'fantasy', 999)
]

function findSuitableBook (genre, pagesLimit) {
    return books.find((book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    })
}
```

На первый взгляд в коде нет ничего необычного. С одной стороны так и есть. Теперь попробуем использовать функцию **findSuitableBook**. Для этого добавим ниже следующий код:

playground.js

```
console.log(findSuitableBook('fantasy', 1000))
console.log(findSuitableBook('fantasy', '1000'))
console.log(findSuitableBook('fantasy'))
console.log(findSuitableBook(1000, 'fantasy'))
console.log(findSuitableBook(1000))
console.log(findSuitableBook())
```

Несмотря на то, что корректным вариантом вызова является только первый, мы не получаем никаких сообщений об ошибках. Помимо того, что мы можем менять аргументы местами и передавать строку вместо числа, мы можем не передавать аргументы вовсе. JavaScript совершенно никак не реагирует на это.

Давайте выполним в консоли следующую команду, чтобы посмотреть на все результаты вызовов:

```
node src/playground.js
```

Мы увидим следующую картину:

```
Book { name: 'Harry Potter', genre: 'fantasy', pageAmount: 980 }
Book { name: 'Harry Potter', genre: 'fantasy', pageAmount: 980 }
undefined
undefined
undefined
undefined
```

Несмотря на некорректность второго варианта **findSuitableBook('fantasy', '1000')** он отработает как нужно за счёт приведения типов. Во время выполнения строка **'1000'** будет сконвертирована в число так как сравнивается с другим числом - это внутреннее устройство JavaScript. Можно сказать, что JavaScript “исправил” ошибку пользователя. Как же JavaScript “исправит” отсутствующие аргументы? Отсутствующим аргументам будет назначено значение **undefined**. Язык сам решит, какой результат должен быть при сравнении строки с **undefined** и математическом сравнении числа с **undefined**.

Разработчик на JavaScript может не заметить проблем в происходящем, так как привык к подобному поведению. Однако можно отметить как минимум два недостатка - плохая читаемость и неочевидное поведение кода.

Плохая читаемость заключается в том, что не прочитав код, мы не сможем понять типы аргументов и какие из них обязательные, а какие нет. Аргумент **genre** мог бы быть числом, равным ID жанра. А если в коде встретиться условие на проверку параметра **pagesLimit** перед его использованием это бы означало, что параметр можно не передавать. Таким образом, разрабатывая на JavaScript постоянно приходится перечитывать код, перед тем как его использовать.

Неочевидность поведения кода заключается в том, что разработчик никогда не знает, как именно отреагирует программа, потому что знать и учитывать каждую деталь внутреннего устройства JavaScript просто не возможно. Неочевидность приводит к сокрытию проблем, которые рано или поздно дадут о себе знать. А обнаружить причину и исправить её в подобных условиях достаточно трудная задача.

Добавим ещё один вызов функции:

playground.js

```
console.log(findSuitableBook().name)
```

До этого момента JavaScript сам решал все проблемы, скрывая их от нас и тем самым лишая нас возможности писать качественный код. Проверим, что будет теперь. Запустим выполнение кода как было показано ранее.

Теперь мы видим исключительную ситуацию, приложение упало со следующим сообщением:

```
console.log(findSuitableBook().name)
                               ^

TypeError: Cannot read property 'name' of undefined
```

Итак, JavaScript не смог придумать как взять поле из несуществующего значения и решил упасть. С запозданием мы узнаем, что в коде были проблемы. Всё тайное становится явным. Даже в небольшом кусочке кода мы столкнулись с не говорящим за себя, неочевидным кодом, который скрывает проблемы. Теперь рассмотрим, что может предложить TypeScript.

Скопируем код из **playground.js** в **index.ts**:

index.ts

```
class Book {
    constructor (name, genre, pageAmount) {
        this.name = name
        this.genre = genre
        this.pageAmount = pageAmount
    }
}

const books = [
    new Book('Harry Potter', 'fantasy', 980),
    new Book('Lord of the Ring', 'fantasy', 1001),
    new Book('How to be productive', 'lifestyle', 500),
    new Book('Game of Thrones', 'fantasy', 999)
]

function findSuitableBook (genre, pagesLimit) {
    return books.find((book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    })
}

console.log(findSuitableBook('fantasy', 1000))
console.log(findSuitableBook('fantasy', '1000'))
```

```
console.log(findSuitableBook('fantasy'))
console.log(findSuitableBook(1000, 'fantasy'))
console.log(findSuitableBook(1000))
console.log(findSuitableBook())
console.log(findSuitableBook().name)
```

Сразу можно заметить, что в редакторе некоторые строки кода подчеркнулись красным. TypeScript сразу же обнаружил некоторые проблемы в коде. Попробуем собрать проект и запустить его:

```
npm run build
```

В консоли появятся ошибки. Ровно те же, которые были подчёркнуты в редакторе кода. Редактор кода отображает ошибки для улучшения пользовательского опыта. А вот появление ошибок во время выполнения сборки это ключевой момент. Такая сборка завершается с ненулевым статус-кодом. Разработчик не просто видит список ошибок, но и сам процесс завершается с ошибкой. Это важный момент потому, что команда сборки всегда выполняется во время деплоя проекта. Подобное поведение гарантирует, что код содержащий ошибки физически не может оказаться в продакшене. При этом мы пока не использовали ни одну возможность языка TypeScript.

Приведём код в порядок, чтобы сборка проходила успешно. Начнём с класса Book. С точки зрения JavaScript этот участок не содержит проблем. Однако, с точки зрения TypeScript присвоение свойств name, genre, pageAmount не может выполняться так как свойства не объявлены в классе. Нужно это исправить. При этом сразу ограничим типы значений, которые могут принимать эти свойства. Имя и жанр должны быть строкой, а количество страниц - числом.

index.ts

```
class Book {
    name: string
    genre: string
    pageAmount: number

    constructor (name: string, genre: string, pageAmount: number) {
        this.name = name
        this.genre = genre
        this.pageAmount = pageAmount
    }
}
```

Здесь мы впервые используем TypeScript синтаксис. Поставив двоеточие при объявлении свойств и аргументов конструктора, мы указываем их типы. В данном примере это string (строка) и number (число). Плотнее с системой типов мы познакомимся на следующем уроке.

На данном этапе главное понять, что таким образом мы пресекаем создание всевозможных ошибочных вариаций создания экземпляра книги. Все следующие строки кода содержат ошибки и не будут пропущены компилятором TypeScript:

```
new Book(),
new Book('Harry Potter'),
new Book('Harry Potter', 'fantasy')
new Book('Harry Potter', 'fantasy', '980'),
new Book(980, 'Harry Potter', 'fantasy'),
```

И это просто замечательно! Каждый раз когда разработчик совершает ошибку, он узнаёт об этом моментально. При этом он получает информацию о файле, строке и даже сути проблемы. Такую проблему можно просто и быстро локализовать.

У нас всё ещё остались ошибки в блоке вызова функции **findSuitableBook**. Их исправить достаточно легко. Для начала удалим все строки, которые помечены как ошибочные. Вместо семи строк у нас останется лишь три. Вот они:

index.ts

```
console.log(findSuitableBook('fantasy', 1000))
console.log(findSuitableBook('fantasy', '1000'))
console.log(findSuitableBook(1000, 'fantasy'))
```

Необходимо написать функцию таким образом, чтобы разработчику и компилятору было очевидно как она работает. Сейчас описать действие функции можно следующим образом: функция “найти подходящую книгу” принимает “жанр” и лимит страниц. Это звучит недостаточно подробно. Нам необходимо сделать так: функция “найти подходящую книгу” принимает “жанр” строкой и лимит страниц числом, а вернуть должна “книгу”. Давайте так и запишем:

index.ts

```
function findSuitableBook (genre: string, pagesLimit: number): Book {
    return books.find((book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    })
}
```

Теперь разработчику достаточно прочитать первую строчку функции (её сигнатуру), чтобы понять смысл того, что она делает. Компилятор же с лёгкостью отсекает оставшиеся неправильные варианты. Проверим, что всё работает как надо:

```
npm run build
npm start
```

На экране должно появиться следующее:

```
Book { name: 'Harry Potter', genre: 'fantasy', pageAmount: 980 }
```

Если же заглянуть в файл **dist/index.js**, то можно заметить, что код в нём один в один как был в нашем **playground.js**. Однако, он прошёл этап транспиляции из TypeScript, а это значит, что он обезопашен. Помимо этого, работать с ним никогда не придётся, работа производится в исходных **src/*.ts** файлах, а всё что находится в **dist/*.js** нужно лишь для исполнения.

Стоит отметить, что для JavaScript существует система, которая пыталась привнести в язык похожий опыт, а именно привнести прозрачность в отношении входных аргументов и возвращаемых значений функций. Реализация системы представляет собой специальный синтаксис комментариев JSDoc. Данный синтаксис поддерживается множеством редакторов кода. Так специальные комментарии выглядят для нашей функции **findSuitableBook**:

playground.js

```
/**
 * @param {string} genre
 * @param {number} pagesLimit
 * @returns {Book}
 */
function findSuitableBook (genre, pagesLimit) {
    return books.find((book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    })
}
```

Однако, данный подход имеет ограниченную эффективность по нескольким причинам. Во-первых, наличие или отсутствие комментариев целиком лежит на ответственности и внимательности разработчика. Во-вторых, несоблюдение описанных сигнатур не приводит к ошибкам, поэтому проблемы в коде могут продолжать оставаться незамеченными. В-третьих, подобные комментарии не являются частью языка, поэтому код может редактироваться, а комментарии оставаться без изменений, что приводит к ещё большей путанице.

Немного поговорим об объявлении типа возвращаемого значения. В примере выше в файле **index.ts** результат выполнения функции объявлен как **Book**. Это помогает в нескольких случаях. Во-первых,

улучшается читаемость, о чём мы упоминали ранее. Во-вторых, это делает невозможным возврат значения отличного от указанного. Например следующий код приведёт к ошибке:

```
function findSuitableBook (genre: string, pagesLimit: number): Book {
    return {
        name: 'Harry Potter',
       // потеряли поле genre
        pageAmount: 980
    }
}
```

Теперь доработаем функцию таким образом, чтобы она могла возвращать как один, так и несколько результатов. При этом по умолчанию функция будет возвращать множественный результат.

playground.js

```
/**
 * @param {string} genre
 * @param {number} pagesLimit
 * @returns {Book}
 */
function findSuitableBook (genre, pagesLimit, multipleRecommendations = true) {
    const findAlgorithm = (book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    }

    if (multipleRecommendations) {
        return books.filter(findAlgorithm)
    } else {
        return books.find(findAlgorithm)
    }
}

const recommendedBook = findSuitableBook('fantasy', 1000)
console.log(recommendedBook.name)
```

Мы добавили новый аргумент **multipleRecommendations**, который по умолчанию имеет значение **true**, изменили алгоритм поиска и оставили только корректный вызов функции. Здесь можно сразу заметить несколько вещей. Так как аргумент по умолчанию true, то это аффектит весь уже существующий код. При этом новый аргумент был потерян в JSDoc, а тип возвращаемого значения остался прежним - обычная практика. Поэтому код **console.log(recommendedBook.name)** остался без изменений и в данный момент приведёт к запросу поля **name** из массива. Снова неочевидное поведение с сокрытием проблем в коде.

Произведём аналогичные изменения в TypeScript коде:

index.ts

```
function findSuitableBook (
    genre: string,
    pagesLimit: number,
    multipleRecommendations = true
): Book {
    const findAlgorithm = (book: Book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    }

    if (multipleRecommendations) {
        return books.filter(findAlgorithm)
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    } else {
        return books.find(findAlgorithm)
    }
}

const recommendedBook = findSuitableBook('fantasy', 1000)
console.log(recommendedBook.name)
```

В данном случае мы получим ошибку компиляции из-за несоответствия описанного типа возвращаемого значения с реальным. Исправим это:

index.ts

```
function findSuitableBook (
    genre: string,
    pagesLimit: number,
    multipleRecommendations = true
): Book | Book[] {
    const findAlgorithm = (book: Book) => {
        return book.genre === genre && book.pageAmount <= pagesLimit
    }

    if (multipleRecommendations) {
        return books.filter(findAlgorithm)
    } else {
        return books.find(findAlgorithm)
    }
}

const recommendedBook = findSuitableBook('fantasy', 1000)
console.log(recommendedBook.name)
                            ~~~~
```

Мы заменили **Book** на **Book | Book[]**, что означает, что будет возвращена либо одна книга либо массив книг. На что компилятор тут же отреагировал другой ошибкой. Дело в том, что прежде чем брать **name** из книги, нужно убедиться, что это не массив книг. Доработаем код следующим образом:

index.ts

```
const recommendedBook = findSuitableBook('fantasy', 1000)

if (recommendedBook instanceof Book) {
    console.log(recommendedBook.name)
} else {
    console.log(recommendedBook[0].name)
}
```

Решение кроется в добавлении дополнительной проверки. В данном случае, мы проверили, является ли результат экземпляром класса **Book**. Как видите, TypeScript всегда найдёт ошибку и подскажет где её искать. Убедимся, что всё работает правильно:

```
npm run build
npm start
```

В консоли должно появиться следующее:

```
Harry Potter
```

Отлично! Можно удалить файл **src/playground.ts**, он нам больше не понадобится.

Пока мы рассмотрели лишь крошечную часть возможностей TypeScript. С каждым уроком мы будем узнавать всё больше о языке, а пока настроим проект для удобной дальнейшей работой над ним. И познакомимся с конфигурационным файлом **tsconfig.json**.

Расширенная настройка проекта

Настройка EditorConfig

Для того, чтобы форматирование всех создаваемых нами файлов было правильным, давайте настроим EditorConfig. EditorConfig - это инструмент, который регулирует некоторые базовые настройки создаваемых в редакторе файлов: кодировка, символ переноса строки и параметры табуляции. Это крайне удобный способ настройки вышеперечисленных параметров, который гарантирует однообразие при работе в разных кодовых редакторах и даже разных операционных системах.

Инструмент поддерживается огромным количеством сред разработки. В некоторых из них по умолчанию, в некоторых через установку плагина. Информация о поддержке и ссылки на плагины для всех редакторов можно найти на официальной странице утилиты.

Теперь собственно настройка. В коре проекта создайте файл **.editorconfig** и поместите туда следующий контент:

.editorconfig

```
root=true

[*]
charset = utf-8
end_of_line = lf
insert_final_newline = true

[*.{js,ts}]
indent_style = space
indent_size = 2

[{package.json,tsconfig*.json}]
indent_style = space
indent_size = 2
```

Данная конфигурация устанавливает для всех файлов в проекте кодировку **utf-8**, а также перенос строки **lf** и добавление пустой строки в конец файла при его сохранении как принято в Unix системах.

В экосистеме JavaScript существует общепринятый код стандарт, с которым мы познакомимся получше чуть позже. Он регламентирует два пробела в качестве символа табуляции. Поэтому мы устанавливаем эти настройки для файлов **package.json**, **tsconfig.json** и всех ***.js**, ***.ts** файлов.

Убедиться, что всё работает можно открыв файл **index.ts** и использовав символ табуляции. Так же при сохранении, в конце файла должна добавляться одна пустая строка. Убедитесь, что файл отформатирован правильно, если нет - внесите необходимые изменения.

Если всё работает и отформатировано как надо, можно двигаться дальше.

Базовая настройка tsconfig.json

До этого момента при вызове сборки мы передавали параметры прямо в команде командной строки. На практике такое встречается редко, так как обычно проект содержит достаточно большое количество опций, поэтому передавать их как аргументы может быть неудобным. Для этих целей существует конфигурационный файл **tsconfig.json**. Когда вызывается команда **tsc**, происходит поиск данного файла и чтение параметров из него.

На этом уроке мы познакомимся с опциями “первой необходимости”. А в дальнейшем рассмотрим более продвинутые настройки. В корне проекта создадим файл **tsconfig.json** и добавим в него следующий контент:

tsconfig.json

```
{
  "compilerOptions": {
    "outDir": "public/scripts",
    "target": "es2015",
    "module": "es2015",
    "moduleResolution": "node"
  },
  "files": [
    "src/index.ts"
  ]
}
```

Для большей наглядности проект будет разрабатываться для работы в браузере поэтому параметр **outDir** был изменён на **public/scripts**. Имя **public** общепринятое в вебе название для папки хранящей “ассеты” такие как скрипты, таблицы стилей, изображения и html-страницы.

Опция **target** уже была установлена нами ранее. Параметр определяет в какую версию стандарта ECMAScript будет скомпилирован исходный TypeScript код. Абсолютно нормальной практикой в веб-разработке писать код с использованием новейших спецификаций языка, которые могут ещё не поддерживаться всеми браузерами, а в процессе сборки проекта трансформировать код в более старые версии языка. Спецификация es2015 (ES6) имеет достаточно широкую поддержку в современных браузерах. Поэтому для начала давайте оставим **target** со значением **es2015**.

Опция **module** позволяет определить модульную систему, которая будет использоваться подключения файлов. Спецификация ES6 имеет свой синтаксис import и export для этих целей. Также в экосистеме JavaScript существуют и другие, более старые модульные системы такие как CommonJS и UMD. Мы можем смело использовать код стандарта ES6, но при этом переопределить модульную систему на другую. Чуть позже мы рассмотрим зачем это может быть нужно, но пока давайте установим **module** также в значение **es2015**.

Опция **moduleResolution** достаточно специфичная и в повседневной жизни практически не нужна. Однако, некоторые опции зависят друг от друга и одна опция может неявно изменить значение другой если её значение не было явно установлено. Такое поведение может привести к ошибкам при компиляции. Так **moduleResolution** при установке **module** в некоторые значения изменяется автоматически на устаревшее значение **classic**, которое использовалось в TypeScript. Для отмены подобных непредвиденных случаев давайте явно установим **moduleResolution** в актуальное значение **node**.

Файл для компиляции из аргумента команды **tsc** переместился в опцию **files**. Как видите опция принимает массив файлов, поэтому если необходимо транспилировать несколько не связанных между собой файлов, например, для разных страниц приложения, то их можно поместить здесь.

Следует отметить, что если файл **index.ts** содержит код подключения (import) других ***.ts** файлов, то они будут автоматически включены в сборку и указывать их в **files** нет необходимости.

Внимание! *В отличие от предыдущих опций, которые размещались в секции **compilerOptions**, директива **files** располагается на верхнем уровне.*

Теперь отредактируем package.json:

package.json

```
{
  "name": "my-project",
  "version": "1.0.0",
  "scripts": {
    "build": "tsc",
    "start": ""
  },
  "devDependencies": {
    "typescript": "^4.2.4"
  }
}
```

Для немного почистим лишнее для удобства. Удим пустой **description**, ненужный больше **main** и избыточные для нас сейчас **author** и **licence**. Так же удалим скрипт **test**, скрипт **start** временно оставим пустым и изменим команду **build** просто на **tsc**.

Запустим сборку и убедимся, что создаётся файл **public/scripts/index.js**. Папку **dist** от прошлой сборки удалим.

Настройка запуска в браузере

Для начала приведём код к более “натуральному” виду. В реальных проектах не пишут всё в одном файле. Вы делим сущность **Book** в отдельный файл **book.ts**, а набор книг в **book-collection.ts**.

book.ts

```
export class Book {
  name: string
  genre: string
  pageAmount: number

  constructor (name: string, genre: string, pageAmount: number) {
    this.name = name
```

```
    this.genre = genre
    this.pageAmount = pageAmount
  }
}
```

book-collection.ts

```
import { Book } from './book.js'

export const books = [
  new Book('Harry Potter', 'fantasy', 980),
  new Book('Lord of the Ring', 'fantasy', 1001),
  new Book('How to be productive', 'lifestyle', 500),
  new Book('Game of Thrones', 'fantasy', 999)
]
```

index.ts

```
import { Book } from './book.js'
import { books } from './book-collection.js'

function findSuitableBook (
  genre: string,
  pagesLimit: number,
  multipleRecommendations = true
): Book | Book[] {
  const findAlgorithm = (book: Book) => {
    return book.genre === genre && book.pageAmount <= pagesLimit
  }

  if (multipleRecommendations) {
    return books.filter(findAlgorithm)
  } else {
    return books.find(findAlgorithm)
  }
}

const recommendedBook = findSuitableBook('fantasy', 1000)

if (recommendedBook instanceof Book) {
  console.log(recommendedBook.name)
} else {
  console.log(recommendedBook[0].name)
}
```

Внимание! *Стандарт ES6 требует указания расширения файла при импорте. Это обусловлено тем, что кроме *.js файлов могут импортироваться скрипты написанные на других языках, например WebAssembly (*.wasm).*

Справедливо будет заметить, что в импорте используется расширение **js**, а не **ts**, не смотря на то, что на самом деле подключаемые файлы имеют расширение **ts** в исходной кодовой базе. Использовать расширение **ts** в конструкции **import** нельзя - TypeScript умеет обрабатывать данную ситуацию.

Соберём проект - **npm run build**.

Пришло время запустить наш скомпилированный код в браузере. Для этого в папке **public** создадим файл **index.html** со следующим содержимым:

index.html

```
<!DOCTYPE html>
<head>
    <script type="module" src="/scripts/index.js"></script>
</head>
```

Внимание! *По причине того, что мы используем импорты ES6, при подключении тэга script необходимо установить атрибут type равный modle. Без этого возникнет ошибка при исполнении скрипта.*

Так как начальный слэш в адресе скрипта **/scripts/index.js** означает, что скрипт нужно искать на том же сервере, то открыть **index.html** просто как файл недостаточно. Нам нужно запустить сервер. Для этого установим следующий пакет:

```
npm install --save-dev http-server
```

Добавим команду запуска сервера в **package.json**:

package.json

```
{
  "name": "my-project",
  "version": "1.0.0",
  "scripts": {
    "build": "tsc",
    "start": "http-server -p 3000"
  },
  "devDependencies": {
    "http-server": "^0.12.3",
    "typescript": "^4.2.4"
  }
```

```
}
```

Запустим сервер выполнив **npm start**. Утилита **http-server** будет “хостить” содержимое папки **public** на сервере по адресу **http://localhost:3000**. Откроем данный адрес в браузере, откроем консоль и убедимся, что видим сообщение “Harry Potter”.

Настройка tsconfig.json для разных окружений

Мы убедились, что всё настроено верно и сборка запускается. Так как нам предстоит много редактировать код, то было бы здорово пересобирать проект автоматически при изменении файлов, а не запускать сборку вручную. Это крайне удобно для разработки, однако для совершенно не нужно для создания production сборки. Поэтому нам необходимо создать две команды сборки **build** для production окружения и **build:dev** для development.

Каждая из этих команд будет использовать свой **tsconfig.json** с несколько отличающимися параметрами и удалять результат предыдущей сборки перед началом нового процесса.

Для начала внесём изменения в **package.json**:

package.json

```
{
  "scripts": {
    "build": "rm -rf public/scripts && tsc",
    "build:dev": "rm -rf public/scripts && tsc -p tsconfig-dev.json",
    "start": "http-server -p 3000"
  },
  "devDependencies": {
    "http-server": "^0.12.3",
    "typescript": "^4.2.4"
  }
}
```

Внимание! *Если вы работаете на Windows команды **build** и **build:dev** необходимо указать следующим образом: **rd -r public\\scripts 2>null & tsc** и **rd -r public\\scripts 2>null & tsc -p tsconfig-dev.json** соответственно.*

Команда **rm -rf public/scripts** удалит папку с ранее скомпилированным кодом и сразу после этого запустит процесс сборки. Команда **build** как и раньше использует файл **tsconfig.json**, а **build:dev** использует аргумент **-p** чтобы указать путь к другому конфигу - **tsconfig-dev.json**. Создадим его:

tsconfig-dev.json

```
{
```

```
    "extends": "./tsconfig.json",
    "compilerOptions": {
        "watch": true,
        "inlineSourceMap": true,
        "inlineSources": true
    }
}
```

Здесь мы используем опцию **extends**, чтобы унаследовать настройки из оригинального файла конфигурации. Это позволяет избежать дублирования и поддержки двух конфигов. Мы лишь переопределяем те опции, которые нас интересуют. В данном случае, мы указываем опцию **watch** со значением **true**, чтобы сказать компилятору, что мы хотим наблюдать за изменениями исходных файлов и запускать сборку каждый раз при их изменении.

Но отличия production и development окружений на этом не заканчиваются. Мы добавили ещё два флага. Для того, чтобы понять зачем они нужны нам необходимо открыть страницу нашего приложения и обратить внимание на строку “Harry Potter” в консоли. Там можно заметить, что сообщение создано строкой номер 19 файла **index.js**. Другими словами, браузер показывает происхождение сообщения в уже скомпилированном коде. Это будет не удобно в дальнейшем при поиске и исправлении ошибок. Для этих целей мы и добавили флаги **inlineSourceMap** и **inlineSources**. Они включают в скомпилированный код его исходный код, что позволяет браузеру ссылаться на оригинальный код во время дебаггинга.

Давайте посмотрим как это выглядит и проверим как это работает на практике. Выполните команду **npm run build:dev**. Первое, что мы можем заметить, консоль не вернула нам управление, вместо этого мы видим сообщение вида:

```
[6:23:22 PM] Starting compilation in watch mode...

[6:23:23 PM] Found 0 errors. Watching for file changes.
```

Компилятор следит за исходными файлами. Откроем файл **book-collection.ts**, добавим “and the Philosopher\'s Stone” к названию книги “Harry Potter” и сохраним файл. Теперь откроем скомпилированный файл **book-collection.js** и проверим его содержимое.

Видно, что название книги изменилось и нам для этого не пришлось запускать сборку вручную. Также можно заметить большую строку, начинающуюся с символов “//# sourceMappingURL” - это и есть карта исходного кода. Для того, чтобы проверить её в действии обновим страницу приложения и посмотрим на сообщение “Harry Potter and the Philosopher's Stone”. Теперь мы видим, что сообщение ссылается на исходный файл **index.ts** строка 21.

Итак, мы завершили настройку нашего проекта на данном этапе. В конце курса мы вернёмся к данному вопросу, чтобы узнать больше всевозможных опций TypeScript.

Настройка ESLint

Но прежде чем сделать следующие шаги в направлении TypeScript, подготовим рабочее окружение по всем стандартам индустрии. Мы уже настроили EditorConfig для того, чтобы все создаваемые нами файлы соответствовали определённым правилам. ESLint - это интсрумент контроля качества кода. Он позволяет описать список правил для форматирования кода, используемых языковых конструкций и так далее. Это позволяет писать код в едином стиле. При несоблюдении этих правил, код будет подчёркиваться в редакторе. Так же проверку можно запускать из консоли.

Хорошей практикой считается запуск проверок на git хук pre-commit и перед сборкой проекта в процессе деплоя. Так как это не относится к теме нашего курса, здесь мы на этом останавливаться не будем. Только настроим проверки в IDE и команды для запуска через консоль.

В IDE WebStrom есть поддержка ESLint по умолчанию, для Visual Studio Code необходимо установить плагин. Плагин существует и для других редакторов кода. Установите его если работаете в другой IDE.

Существует большое количество готовых конфигураций. Несколько из них являются основными принятыми стандартами в индустрии, например Airbnb JavaScript Style Guide или JavaScript Standard Style. Но они для нашего учебного проекта будут избыточны. В комплекте с ESLint идёт стандартный конфиг с минимальными настройками, которым мы и воспользуемся.

Помимо самого ESLint нам понадобятся дополнительные пакеты для поддержки TypeScript синтаксиса. Установим все необходимые зависимости:

```
npm install --save-dev eslint @typescript-eslint/parser
@typescript-eslint/eslint-plugin
```

Для того, чтобы IDE начала валидировать код, необходимо создать конфигурационный файл **.eslintrc** в корне проекта:

.eslintrc

```
{
    "root": true,
    "parser": "@typescript-eslint/parser",
    "plugins": [
      "@typescript-eslint"
    ],
    "extends": [
      "eslint:recommended",
      "plugin:@typescript-eslint/eslint-recommended",
```

```
      "plugin:@typescript-eslint/recommended"
    ]
}
```

Данный конфиг “научит” ESLint понимать TypeScript синтаксис и применит самые стандартные правила рекомендованные командой ESLint.

Нормальной практикой является дорабатывать конфиг под свои нужды в случае необходимости. Так как стандартный конфиг имеет минимум настроек, добавим туда некоторые опции:

.eslintrc

```
{
  "root": true,
  "parser": "@typescript-eslint/parser",
  "plugins": [
    "@typescript-eslint"
  ],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended"
  ],
  "rules": {
    "quotes": ["error", "single"],
    "indent": ["error", 2]
  }
}
```

В секции **rules** мы добавили пару обязательных условий - должны использоваться одинарные кавычки и два символа пробела для отступов. С полным списком правил можно ознакомиться на официальном сайте ESLint и в репозитории плагина TypeScript ESLint.

Теперь IDE должна реагировать на использование двойных кавычек или символы табуляции. Попробуйте использовать четыре пробела вместо двух или двойные кавычки вместо одинарных. Если код в этих местах подчеркнулся, значит, всё настроено верно. Не спешите исправлять эти ошибки, сделаем это с использованием ESLint. Сначала добавим скрипты линтинга для командной строки. Для этого добавьте в секцию **scripts** файла **package.json** ещё две команды:

package.json

```
"scripts": {
  "build": "rm -rf public/scripts && tsc",
  "build:dev": "rm -rf public/scripts && tsc -p tsconfig-dev.json",
  "start": "http-server -p 3000",
  "lint": "eslint src --ext .js --ext .ts",
  "lint-fix": "eslint src --ext .js --ext .ts --fix"
}
```

Запустив **npm run lint** мы должны увидеть все ошибки в консоли. А запустив **npm run lint-fix** выполнится автоматический фикс ошибок. Те ошибки, которые не исправляются автоматически необходимо править вручную.

Итак, мы полностью готовы к дальнейшей работе.

Практическое задание

В процессе прохождения курса мы создадим приложение-агрегатор по аренде недвижимости. Приложение будет представлять собой форму с выбором дат заезда и выезда, выбором поставщиков услуг, указанием максимальной суммы за сутки и городом. Приложение отображает список результатов с возможностью забронировать апартаменты или добавить их в избранное. Апартаменты можно сортировать по цене и расстоянию от пользователя.

1. Возьмите заготовку проекта. Настройте отступы, кодировку и символы окончания строк с помощью EditorConfig. Создайте две конфигурации сборки TypeScript для production и development окружений. Поменяйте расширения файлов с ***.js** на ***.ts** у файлов в папке **src**. Подключите и настройте ESLint. Скомпилируйте и запустите проект в браузере. Убедитесь, что видите на экране блоки пользователя, поисковой формы и результатов поиска.
2. Найдите функцию **renderUserBlock** и доработайте её следующим образом. Функция должна принимать три аргумента имя пользователя, ссылка на его аватар и количество элементов в избранном. Аргументы должны иметь подходящие для них типы. Аргументы должны использоваться для соответствующих целей. Убедиться, что следующая логика работает верно, а если нет, то внести правки. Если количество избранных объектов меньше одного, то нужно выводить сообщение “ничего нет” и иконка сердечка должны быть не закрашена. Иначе иконка сердечка закрашена и рядом просто выводится количество избранных объектов. Все стили уже присутствуют, необходима только логика.
3. Найдите функцию **renderSearchFormBlock** и доработайте её следующим образом. Функция должна принимать дату въезда и дату выезда. При этом минимальная дата, которую можно выбрать это дата сегодняшнего дня, а максимальная дата - последний день следующего месяца. Будем считать это ограничениями сервиса. По умолчанию поля заполняются следующим образом. Дата въезда это следующий день от текущей даты. Дата выезда - плюс два дня от даты въезда.

Глоссарий

Транспиляция — преобразование программы, при котором используется исходный код программы, написанной на одном языке программирования в качестве исходных данных, и производится

эквивалентный исходный код на другом языке программирования. Термин применяется при переводе между разными языками программирования, работающими примерно на одном уровне абстракции.

Компиляция — преобразование программы, при котором используется исходный код программы, написанной на одном языке программирования в качестве исходных данных, и производится эквивалентный исходный код на другом языке программирования. Термин применяется при переводе между разными языками программирования, работающими на разных уровнях абстракции. Обычно с языка высокого уровня на язык низкого уровня.

Сборка — процесс получения информационного продукта из исходного кода. Чаще всего включает компиляцию и компоновку, выполняется инструментами автоматизации.

Деплой (развёртывание программного обеспечения) — совокупность действий, направленных на перевод исходного кода в рабочее состояние на конкретном сервере

Дополнительные материалы

1. Официальный сайт JSDoc
2. Редактор Visual Studio Code
3. Плагин EditorConfig для Visual Studio Code
4. Плагин ESLint для Visual Studio Code
5. Статья “TypeScript: Раскладываем tsconfig по полочкам. Часть 1”
6. Книга Борис Чёрный “Профессиональный TypeScript”

Используемые источники

1. Официальный референс на TSConfig
2. Официальная документация JSDoc
3. Документация MDN