Using TypeScript Modules

GETTING STARTED WITH TYPESCRIPT MODULES

Dan Wellman
LEAD UI DEVELOPER

Everything you ever wanted to know about TypeScript modules.

Required Experience

You may have used modules or TypeScript before

Required Experience

You may have used modules or TypeScript before

Perhaps you haven’t used either modules or TypeScript at all

Required Experience - None

All concepts will be explained in full!

Course Aims

Build a solid foundation of knowledge

Course Aims

Build a solid foundation of knowledge

Reinforce existing knowledge and fill in any gaps

Master TypeScript Modules

Overview

Getting started with TypeScript modules:

- What is a module exactly?
- Why and when should I use them?
- How to setup a TypeScript project
- How to compile TypeScript

Next up, when and why should we use modules?

Why and When to Use TypeScript Modules

When to Use TypeScript Modules

If you're already using TypeScript

Use TypeScript modules always with TypeScript

Medium to large JavaScript projects

When to Use TypeScript Modules

You're likely to use modules in every TypeScript project

When to Use TypeScript Modules

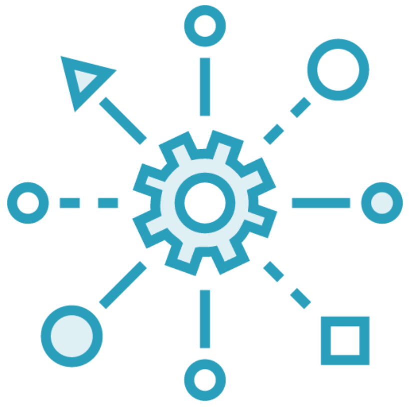
You're likely to use modules in every TypeScript project

Popular modern frameworks use modules

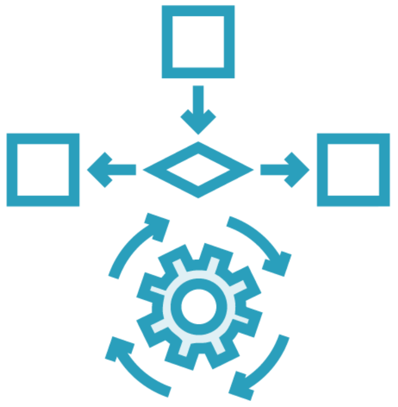
Why You Should Use TypeScript Modules

Front-end applications are increasingly complex

Why You Should Use TypeScript Modules



Front-end applications are increasingly complex



Organisation and structure are critical

Reuse Is Key

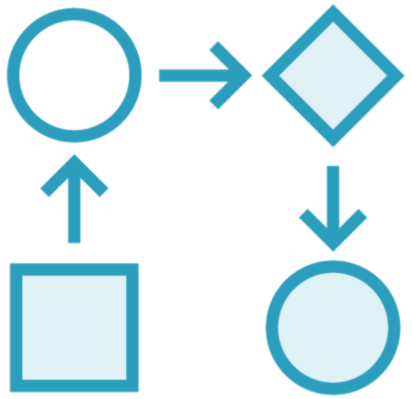
Reuse Is Key

DRY

DRY

Don’t repeat yourself

Structure code effectively



Make code more portable

Importing and Exporting

Easily make code in a module public or private

Importing and Exporting

Easily make code in a module public or private

Avoid repetition and boiler-plate code

Strict Mode

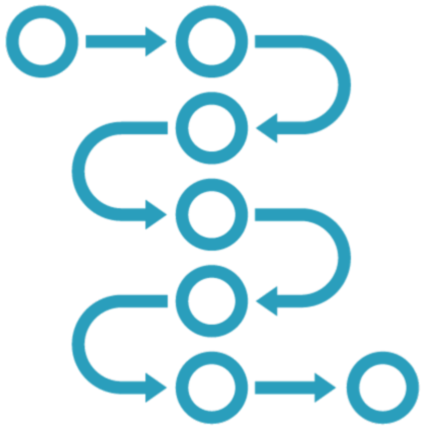
Modules are always evaluated in strict mode

Strict Mode

Modules are always evaluated in strict mode

Early versions of JavaScript had lots of bugs

Strict Mode



Strictly opt-in

Add `use strict` to every JavaScript file

Avoid even more repetition!

Next up, what makes a module, a module?

What Makes a Module a Module?

```
import { x } from ‘./some-module’;
export x;
```

The Primary Distinction of Modules

`import` **and/or** `export` **statements**

Modules are always evaluated in strict mode

Old code converted to a module may not function correctly

Module Features

Modules are never evaluated in the global scope

Module Features

Modules are never evaluated in the global scope

The window object is available to all modules

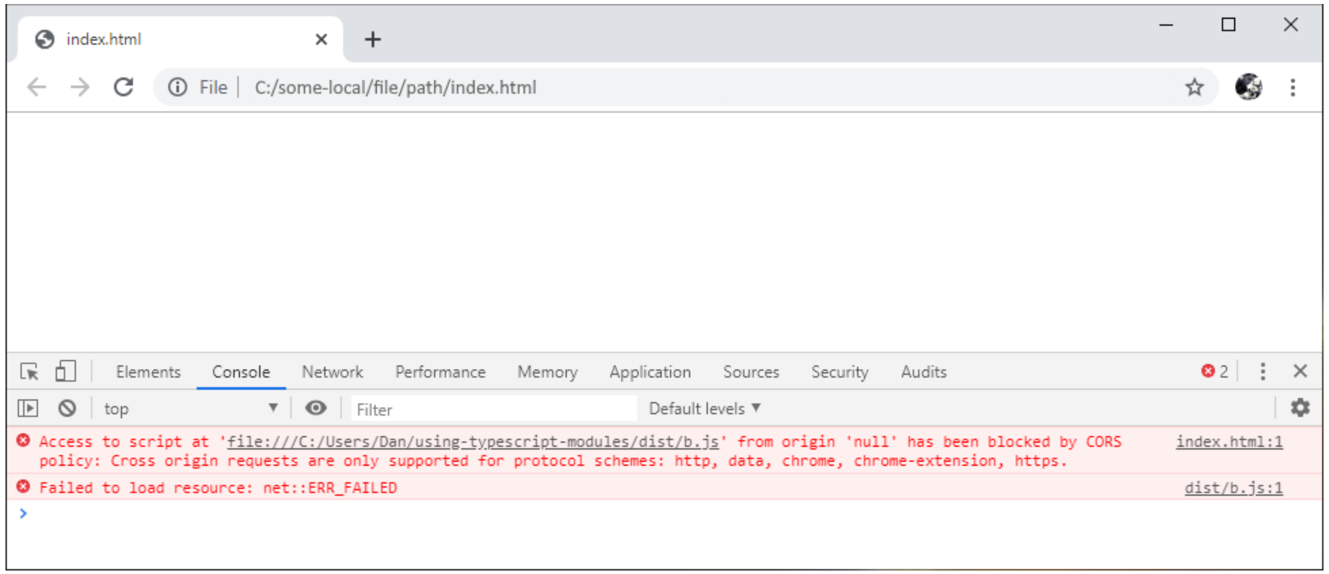
Module Features

Modules are never evaluated in the global scope

The window object is available to all modules

Only exported code can be imported

Modules cannot be loaded locally



```
Access to script at 'file:///C:/Users/Dan/using-typescript-modules/dist/b.js' from origin 'null' has been blocked by CORS    index.html:1
policy: Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, https.
Failed to load resource: net::ERR_FAILED                                                                                  dist/b.js:1
```

A local web-server is needed for testing

We will set one up shortly

Modules are always deferred

```
<sript src="some-file.js" deferred></script>
```

The script is loaded, but not executed until the ready event fires

Modules are loaded asynchronously and executed on ready

Next up, internal vs. external modules

Internal vs. External Modules

TypeScript used to have the concept of internal and external modules

External Module

External Module

External modules always used import and/or export

External Module

External modules are now just modules

External Module

The concept no longer exists in TypeScript

Named Internal Module

Named Internal Module

A module created with the `module` keyword

```
module MyNamedInternalModule {

}
```

Named Internal Module

```
module MyNamedInternalModule {
  const privateVar = ‘private’;

}
```

Named Internal Module

Members are scoped to the module

```
module MyNamedInternalModule {
  const privateVar = ‘private’;
  export const publicVar = ‘public’;
}
```

Named Internal Module

Values can be exported

```
module MyNamedInternalModule {
  const privateVar = ‘private’;
  export const publicVar = ‘public’;
}
console.log(MyNamedInternalModule.publicVar); //public
```

Named Internal Module

Values can be exported

```
module MyNamedInternalModule {
  const privateVar = ‘private’;
  export const publicVar = ‘public’;
}
console.log(MyNamedInternalModule.publicVar); //public
```

Named Internal Module

Now referred to as **namespaces**

```
namespace MyNamespace {
  const privateVar = ‘private’;
  export const publicVar = ‘public’;
}
console.log(MyNamespace.publicVar); //public
```

Namespace

Exactly equivalent to the previous example

```
namespace MyNamespace {
  const privateVar = ‘private’;
  export const publicVar = ‘public’;
}
console.log(MyNamespace.publicVar); //public
```

Namespace

Preferred syntax

```
namespace MyNamespace {
  const privateVar = ‘private’;
  export const publicVar = ‘public’;
}
console.log(MyNamespace.publicVar); //public
```

Namespace

Can be useful, but we won’t cover them again in this course

Implicit Internal Module

Implicit Internal Module

A module attached to the global scope

```
const globalVar = ‘global’;
```

Implicit Internal Module

Attached to the global scope

```
const globalVar = ‘global’;
```

Implicit Internal Module

globalVar is available everywhere

Avoid implicit internal modules where possible

Next up, creating a TypeScript project

Creating a TypeScript Project for Modules

Next up, compiling TypeScript

Compiling TypeScript

Compilers include:

- TSC – included with TypeScript
- Babel
- Webpack
- Gulp

Summary

In this module we covered the fundamentals:

- Why and when to use modules
- Import/export statements make a file a module
- External/internal modules are now modules and namespaces
- How to create and configure a TypeScript project
- How to compile TypeScript to JavaScript