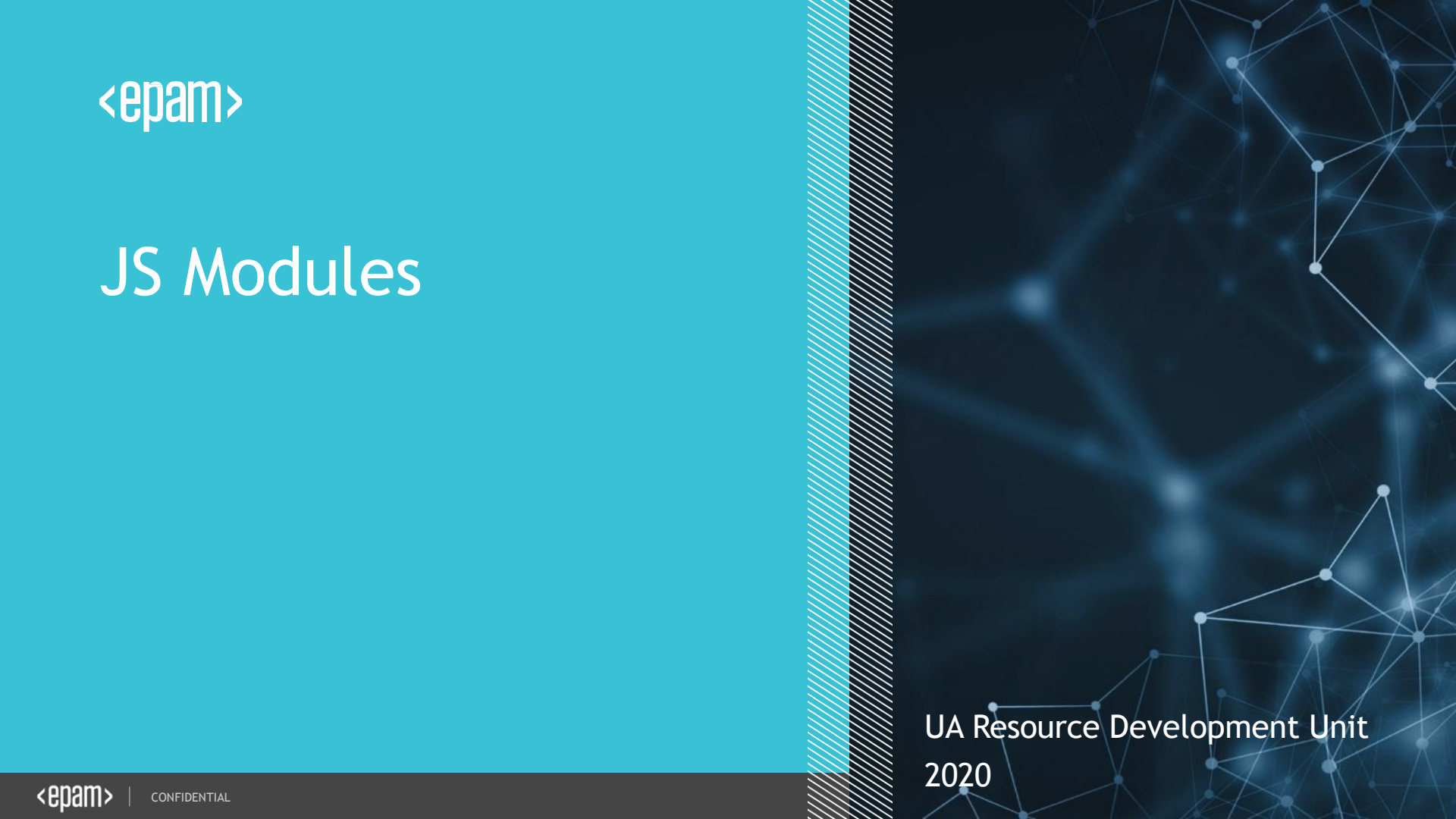

JS Modules
UA Resource Development Unit
2020

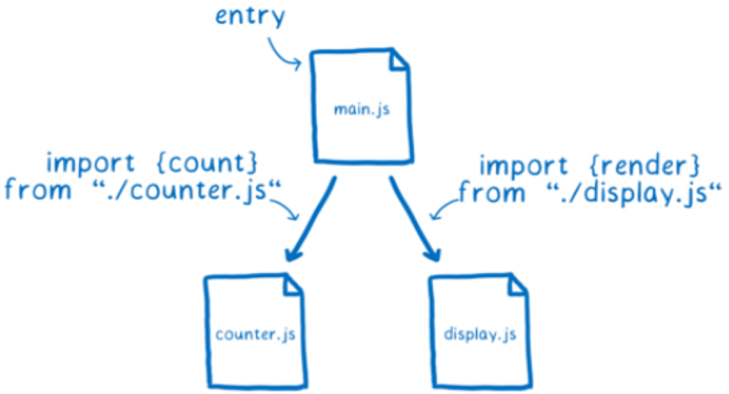
AGENDA

1. Introductory word
2. What are the modules?
3. Why we use it
4. Modules system type
5. Real examples

What are modules?

Small units of independent, reusable code

- Single-purpose
- Self-contained
- Promote to reuse



Why we use modules

Abstract code: Sometimes we don't need to know the actual implementation of some code, like in the library, we need to use it only

Encapsulation: to hide code inside the module, so we can't swap or upgrade it from another part of code

Reusability: avoid code duplication, DRY pattern

Define own Scope: Not-Polluting the Global Scope

Name collisions and conflicts

Easy to support: you can change code on demand

Easy to test

Modules approach in pure JS

Simple and popular workaround of Modules with JS

IIFE - Immediately Invoked Function Expressions

```
const a = 'test';
const foo = (
  () => {
    const a = 'foobar';
    return {
      bar: () => {
        console.log(a);
      }
    }
  }
)();
console.log(a); // test
foo.bar(); // foobar
```

This method allows us to:

- Encapsulate code inside IIEF
- Define variables inside the IIFE so they don't pollute the global scope

But we can't provide a mechanism for dependency management

Modules approach in pure JS

Module pattern

```
const Module = () => {
  const foo = 'Hello';
  const privateMethod = (str) => {
    console.log(`${foo} ${str}`);
    };
  return {
    bar: privateMethod,
  }
};

Module().bar( str: 'World'); // Hello World
```

This method allows us to:

- Encapsulate private members and methods
- Expose public methods on demand

But we can’t provide a mechanism for dependency management

Popular modules formats

CommonJS

Asynchronous module definition (AMD)

ES6 module format

Popular modules formats

Asynchronous module definition (AMD)

Load file in the async format

Also can load and build other types of file (html, css etc)

Now outdated prefer use common JS or native modules with webpack or babel+rollup

Popular modules formats

Asynchronous module definition (AMD)

```
// polyfill-vendor.js
define(function () {
    // polyfills-vendor code
});

// module1.js
define(function () {
    // module1 code
    return module1;
});

// module2.js
define(function (params) {
    var a = params.a;

    function getA(){
        return a;
    }

    return {
        getA: getA
    }
});
```

```
// app.js
define(['PATH/polyfill-vendor'] , function () {
    define(['PATH/module1', 'PATH/module2'] , function (module1, module2) {
        var APP = {};

        if(isModule1Needed){
            APP.module1 = module1({param1:1});
        }

        APP.module2 = new module2({a: 42});
    });
});
```

Popular modules formats

CommonJS

Load file in sync format

Work natively in NodeJS

In the browser by bundling up all of your dependencies prefer to use Browserify or Webpack

Popular modules formats

CommonJS example

```
const convertToLowerCase = (sentence) => {...};

module.exports = convertToLowerCase;
```

```
// For NPM packages
const moment = require('moment');
// For internal modules
const toLowerCase = require('./toLowerCase.js');
```

Best Practice:

Create separate file for each function/class/etc
Give him a direct name
Import all module on top of the file
if you have few elements to import/exports use destruction

Popular modules formats

CommonJS tricky example

```
(
  () => {
    const a = 5;
    return console.log(require( id: './foo')(5, a))
  }
)();
```

import file in the middle of function

```
module.exports.sum = (a, b) => {
  return a + b;
};

module.exports.minus = (a, b) => {
  return a - b;
};
```

```
const {sum, minus} = require( id: './foo');
```

Scripts vs Modules

When we use module in browser, we use syntax like this

<script type="module" src="foo.js"></script>

	Scripts	Modules
Default mode	Non-strict	Strict
Top level variables are	Global	Local to module
Value of this at top level	Window	Undefined
Executed	Synchronously	Asynchronously
Declarative imports (import statement)	No	Yes
File extension	.js	.js

Popular modules formats

ES6 module format

Key information

Declare module like <script type="module" src="foo.js"></script>

Module download like defer script, they save order

Use strict by default

In a module, “this” is undefined

A module code is evaluated only the first time when imported

Each module has its own top-level scope

Async works on inline scripts

```
<script type="module">
  a = 5; // error
</script>
```

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

```
<script type="module">
  // The variable is only visible in this module script
  let user = "John";
</script>

<script type="module">
  alert(user); // Error: user is not defined
</script>
```

Popular modules formats

ES6 module export syntax

```
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
// export a constant
export const MODULES_BECAME_STANDARD_YEAR = 2015;
// export a class
export class User {
  constructor(name) {
    this.name = name;
  }
```

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export { sayHi, sayBye }; // a list of exported variables
```

We can export all type of data

with curly braces we can export all functionality in one row

Popular modules formats

ES6 module import syntax

1

```
// 🗀 main.js
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

We can import function with destruction js

```
// 🗀 main.js
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

But if there's a lot to import, we can import everything as an object using import * as <obj>

```
// 🗀 say.js
...
export {sayHi as hi, sayBye as bye};
```

```
// 🗀 main.js
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

We can also use <as> to import or export under different names.

Popular modules formats

ES6 module export default syntax

```
// 🗀 user.js
export default class User { // just add "default"
  constructor(name) {
    this.name = name;
  }
}
```

```
// 🗀 main.js
import User from './user.js'; // not {User}, just User

new User('John');
```

```
// 🗀 user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

In practice, there are mainly two kinds of modules:
- Module that contains a library, pack of functions;
- Module that declares a single entity, e.g. a module user.js exports only class User;

Popular modules formats

ES6 module export default syntax

```
// 🗀 main.js
import {default as User, sayHi} from './user.js';

new User('John');
```

Modules provide special export default (“the default export”) syntax to make “one thing per module” way look better.

```
// 🗀 main.js
import * as user from './user.js';

let User = user.default; // the default export
new User('John');
```

We can combine two types of exports in one file

Also we can use <*> and create alias for all file

Popular modules formats

Dynamic imports

```
import ... from getModuleName(); // Error, only from "string" is allowed
```

```
if(...) {
  import ...; // Error, not allowed!
}

{
  import ...; // Error, we can't put import in any block
}
```

```
// 🗀 say.js
export function hi() {
  alert(`Hello`);
}

export function bye() {
  alert(`Bye`);
}
```

```
let {hi, bye} = await import('./say.js');

hi();
bye();
```

The <import(module)> expression loads the module and returns a promise that resolves into a module object that contains all its exports. It can be called from any place in the code.
import(module)

Popular modules formats

Dynamic imports

!Dynamic imports work in regular scripts, they don’t require script type="module"

```
import('./say.js')
  .then( onfulfilled: ({ hi, bye }) => hi())
  .catch( onrejected: err => console.log(err));
```

Popular modules formats

Build tools

In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as Webpack and deploy to the production server. One of the benefits of using bundlers – they give more control over how modules are resolved, allowing bare modules and much more, like CSS/HTML modules.

Build tools do the following:

Take a “main” module, the one intended to be put in <script type="module"> in HTML.

Analyze its dependencies: imports etc.

Build a single file with all modules (or multiple files, that’s tunable), replacing native import calls with bundler functions, so that it works. “Special” module types like HTML/CSS modules are also supported.

In the process, other transforms and optimizations may be applied:

- Unreachable code removed
- Unused exports removed (“tree-shaking”).
- The resulting file is minified (spaces removed, variables replaced with shorter name etc).
- Modern, bleeding-edge JavaScript syntax may be transformed into older one with similar functionality using Babel.

Links

How to start: https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/

http://addyosmani.com/writing-modular-js

http://wiki.commonjs.org/wiki/Modules

https://github.com/amdjs/amdjs-api/blob/master/AMD.md

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

https://drive.google.com/file/d/1oFit-QO82s2corPlzNUpywl5A1p_bpne/view - lecture code examples

Tools:

http://webpack.js.org

http://requirejs.org

http://browserify.org

https://github.com/cujojs/curl

https://github.com/ModuleLoader/es6-module-loader

FE Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

serhii_shcherbak@epam.com

With the note [FE Online UA Training Course Feedback]

Thank you.

Q&A

UA Frontend Online LAB