


Lecția 3: Scripturi și Module Node.js

<https://www.youtube.com/watch?v=qgRUr-YUk1Q>

Introducere

În browser, tot codul tau JavaScript rulează într-un singur thread și într-un singur fișier (sau fișiere incluse manual). Cu Node.js, intri într-o lume unde poți crea aplicații complexe, organizate în module reutilizabile, care rulează pe server și interacționează cu sistemul de operare. Să descoperim cum să creezi și să organizezi scripturi Node.js puternice și maintibile.

 **Setup pentru ES Modules:** În toate exemplele din această lecție folosim ES Modules (standardul modern). Pentru a rula aceste exemple, adaugă în `package.json`:

```
{ "type": "module" }
```

Sau folosește extensia `.mjs` pentru fișierele tale. ES Modules sunt standardul modern recomandat pentru proiecte noi.

Sistemul de Module în Node.js

Întrebări pentru Reflecție

- Cum organizai codul JavaScript complex în browser fără module?
- De ce este important să poți împărți codul în fișiere separate?
- Ce probleme rezolvă sistemul de module din Node.js?

Provocarea ta: Înțelegerea Module System

Obiectiv: Să stăpânești crearea și folosirea modulelor în Node.js

Concepte cheie:

- ES Modules (modern) vs CommonJS (legacy)

- import/export vs require() (încă găsit prin legacy code)
- Named exports vs default exports
- Module resolution și caching

💡 Prima explorare:

```
// În browser făceai așa: // <script src="utils.js"></script> // <script src
="app.js"></script> // // Toate variabilele în global scope - periculos! // Î
n Node.js modern cu ES Modules: // utils.js const add = (a, b) => a + b const
multiply = (a, b) => a * b // Export modern cu ES Modules: export { add, mult
iply } // sau export individual: // export const add = (a, b) => a + b // ap
p.js // Import modern cu ES Modules: import { add, multiply } from './utils.j
s' // sau import complet: // import * as utils from './utils.js' // Legacy Co
mmonJS (încă întâlnit în cod vechi): // module.exports = { add, multiply } //
export // const { add, multiply } = require('./utils.js') // import
```

De ce este ES Modules o abordare mai bună?

- Syntax mai curat și standardizat
- Tree shaking pentru bundlers
- Static analysis și better tooling
- Async loading support

🔧 Primul test - Crearea modulelor:

- Creează fișierele math-utils.js, app.js
- Exportă funcții pentru operații matematice (add, subtract)
- Importă acele funcții în app.js, scrie câteva teste pentru ele

✅ Soluția completă:

```
// math-utils.js const validateNumber = (value) => { if (typeof value !== 'nu
mber' || isNaN(value)) { throw new TypeError(`Value must be a valid number, g
ot ${typeof value}`) } } export const add = (a, b) => validateNumber(a) + val
idateNumber(b) export const subtract = (a, b) => validateNumber(a) - validat
eNumber(b)
```

```
// app.js import { add, subtract } as math from './math-utils.js' console.log('📦 Testing Math Utils Module\\n') // Testare operații de bază try { console.log('Addition:', math.add(5, 3)) console.log('Subtraction:', math.subtract(10, 4)) } catch (error) { console.error('Error in basic operations:', error.message) } // Testare validări console.log('\\nTesting Error Handling:') try { math.add(5, 'hello') // Ar trebui să arunce eroare } catch (error) { console.log('✅ Type validation works:', error.message) }
```

Nu uita, pentru ES Modules trebuie să specifice în package.json: `{ "type": "module" }` sau să folosești extensia `.mjs` pentru fișiere.

ES Modules vs CommonJS:

ES Modules (modern, recomandat):

- import/export syntax mai curat
- Static analysis pentru better tooling
- Tree shaking pentru optimizări
- Standard ECMAScript oficial

CommonJS (legacy, încă întâlnit):

- require()/module.exports syntax
- Dynamic loading la runtime
- Folosit în Node.js înainte de ES Modules
- Încă comun în ecosistem pentru compatibilitate

Pentru proiecte noi: folosește ES Modules

Pentru proiecte existente: migrează treptat sau rămâi pe CommonJS pentru consistență

Organizarea Codului în Module

Întrebări pentru Reflecție

- Cum decizi ce funcții să pui în același modul?
- Care este echilibrul ideal între module mici vs module complexe?
- Cum eviți dependențele circulare între module?

Provocarea ta: Arhitectura Modulelor

Obiectiv: Să înveți să organizezi codul în module logic separate

Structura unui proiect Node.js tipic:

```
project/ ├── src/ | ├── utils/ | | ├── file-helper.js | | └── string-helper.js  
s | ├── services/ | | ├── user-service.js | | └── email-service.js | ├── mode  
ls/ | | └── user.js | └── app.js ├── tests/ ├── package.json └── index.js
```

- Cum organizezi această structură?
- Ce responsabilități are fiecare folder?

Provocare practică - Biblioteca de String Utils:

Creează o bibliotecă modulară pentru prelucrarea string-urilor care demonstrează organizarea codului în module.

Obiectiv: Să implementezi o clasă StringHelper cu funcții utile și să o organizezi ca un modul reutilizabil.

Funcții de implementat:

1. `capitalize(str)` - Prima literă cu majusculă
2. `camelCase(str)` - Transformă în camelCase (ex: "hello world" → "helloWorld")
3. `slugify(str)` - Transformă în URL slug (ex: "Hello World!" → "hello-world")

Provocări tehnice:

- Gestionează edge cases (string gol, null, undefined)
- Gestionează caractere speciale și diacritice
- Adaugă validări pentru parametri
- Testează fiecare funcție

Soluția completă:

```
// src/utils/string-helper.js // Funcții utilitare pentru prelucrarea string-urilor
export const capitalize = (str) => { // Validarea input-ului
  if (!str || typeof str !== 'string') { return '' } // Capitalizarea primei litere
  return str.charAt(0).toUpperCase() + str.slice(1).toLowerCase() }
export const camelCase = (str) => { if (!str || typeof str !== 'string') { return '' } // Împarte string-ul după separatori și convertește în camelCase
  return str.toLowerCase().split(/[\\s\\-_]+)/ // Împarte după spații, cratime, underscore
  .filter((word) => word.length > 0) // Elimină cuvintele goale
  .map((word, index) => { // Prima cuvânt rămâne cu litere mici, restul se capitalizează
    return index === 0 ? word : capitalize(word) })
  .join('') }
export const slugify = (str) => { if (!str || typeof str !== 'string') { return '' }
  return (str.toLowerCase().trim() // Înlocuiește diacriticele cu echivalentele ASCII
    .normalize('NFD') .replace(/[\\u0300-\\u036f]/g, '') // Păstrează doar litere, cifre și spații
    .replace(/[\\^a-z0-9\\s]/g, '') // Înlocuiește spațiile multiple cu un singur spațiu
    .replace(/[\\s+/g, ' ') .trim() // Înlocuiește spațiile cu cratime
    .replace(/[\\s/g, '-') ) }
```

```
// tests/test-string-helper.js
import { capitalize, camelCase, slugify } from '../src/utils/string-helper.js'
console.log(`capitalize('hello'): "${capitalize('hello')}"`)
console.log(`capitalize('WORLD'): "${capitalize('WORLD')}"`)
console.log(`capitalize(null): "${capitalize(null)}"`)
console.log(`camelCase('hello world'): "${camelCase('hello world')}"`)
console.log(`camelCase('hello-world-test'): "${camelCase('hello-world-test')}"`)
console.log(`camelCase('hello_world_test'): "${camelCase('hello_world_test')}"`)
console.log(`slugify('Hello World!'): "${slugify('Hello World!')}"`)
console.log(`slugify('Café & Restaurant'): "${slugify('Café & Restaurant')}"`)
console.log(`slugify('DOM & Node.js'): "${slugify('DOM & Node.js')}"`)
```

Pattern-uri Avansate și Best Practices (opțional)

Întrebări pentru Reflecție

- Cum structurezi un proiect Node.js pentru scalabilitate?
- Ce pattern-uri de design sunt utile în Node.js?
- Cum testezi module-urile în mod eficient?

Provocarea ta: Pattern-uri Profesionale

Obiectiv: Să aplici best practices pentru cod production-ready

💡 EventEmitter Extension Pattern:

```
// src/core/event-emitter.js import EventEmitter from 'events' class CustomEventEmitter extends EventEmitter { constructor() { super() this.setMaxListeners(20) // Previne memory leaks } emitAsync(event, ...args) { return new Promise((resolve) => { this.emit(event, ...args) resolve() }) } emitWithTimeout(event, timeout, ...args) { setTimeout(() => { this.emit(event, ...args) }, timeout) } }
```

Ce face acest pattern:

- Extinde funcționalitatea EventEmitter-ului built-in din Node.js
- Adaugă metode custom pentru evenimente asincrone și cu timeout
- Păstrează toate funcționalitățile originale (on, emit, off, etc.)

Când să-l folosești:

- Când ai nevoie de comunicare event-driven în aplicație
- Pentru decuplarea componentelor (publisher-subscriber pattern)
- În aplicații real-time (chat, notificări, streaming)
- Când vrei să adaugi funcționalitate extra la evenimente standard

💡 Singleton Pattern pentru configurație:

```
// src/config/app-config.js
class AppConfig {
  constructor() {
    if (AppConfig.instance) {
      return AppConfig.instance
    }
    this.config = {
      database: {
        host: process.env.DB_HOST || 'localhost',
        port: process.env.DB_PORT || 5432,
      },
      api: {
        baseUrl: process.env.API_URL || '<http://localhost:3000>',
      },
    }
    AppConfig.instance = this
  }

  get(key, defaultValue) {
    const keys = key.split('.')
    let value = this.config
    for (const k of keys) {
      value = value?.[k]
      if (value === undefined) return defaultValue
    }
    return value
  }

  set(key, value) {
    const keys = key.split('.')
    const lastKey = keys.pop()
    let target = this.config
    for (const k of keys) {
      if (!target[k]) target[k] = {}
      target = target[k]
    }
    target[lastKey] = value
  }
}

// Folosire: AppConfig va fi mereu aceeași instanță
const config1 = new AppConfig()
const config2 = new AppConfig()
console.log(config1 === config2) // true
```

Ce face acest pattern:

- Garantează că o clasă are doar o singură instanță în întreaga aplicație
- Oferă un punct global de acces la configurația aplicației
- Încarcă configurația o singură dată la prima utilizare

Când să-l folosești:

- Pentru configurația aplicației (database settings, API keys, etc.)
- Pentru conexiuni la baza de date (connection pools)
- Pentru cache-uri globale sau state management
- Când ai nevoie de un obiect unic shared între module

💡 Factory Pattern pentru servicii:

```
// src/factories/service-factory.js import EmailService from '../services/email-service.js' import SMSService from '../services/sms-service.js' import PushNotificationService from '../services/push-service.js' class ServiceFactory { static services = new Map() static createService(type, config) { switch (type) { case 'email': return new EmailService(config) case 'sms': return new SMSService(config) case 'push': return new PushNotificationService(config) default: throw new Error(`Unknown service type: ${type}`) } } static getService(type, config) { // Cache pattern: returnează instanța existentă sau creează una nouă if (!this.services.has(type)) { const service = this.createService(type, config) this.services.set(type, service) } return this.services.get(type) } static clearCache() { this.services.clear() } } // Folosire: const emailService = ServiceFactory.getService('email', { apiKey: 'xxx' }) const smsService = ServiceFactory.getService('sms', { provider: 'twilio' })
```

Ce face acest pattern:

- Creează obiecte fără să specifice clasa exactă în cod
- Centralizează logica de creare a obiectelor similare
- Poate implementa caching pentru a reutiliza instanțele

Când să-l folosești:

- Când ai multe clase similare (EmailService, SMSService, etc.)
- Pentru plugin systems sau extensibility
- Când vrei să ascunzi complexitatea creării obiectelor
- Pentru dependency injection și testare mai ușoară

💡 Barrel Pattern pentru exporturi clean:

```
// src/index.js - punct central de export export { default as services } from './services/index.js' export { default as utils } from './utils/index.js' export { default as factories } from './factories/index.js' // src/services/index.js - barrel pentru servicii export { default as EmailService } from './email-service.js' export { default as SMSService } from './sms-service.js' export { default as UserService } from './user-service.js' // src/utils/index.js - barrel pentru utils export { capitalize, camelCase, slugify } from './string-helper.js' export { formatDate, parseDate } from './date-helper.js' export { validateEmail, validatePhone } from './validators.js'
```


Ce face acest pattern:

- Agregă exporturile din multiple module într-un singur punct de entry
- Simplifică importurile pentru consumatorii bibliotecii
- Oferă control centralizat asupra API-ului public

Când să-l folosești:

- Pentru biblioteci și package-uri NPM
- Când ai o structură de foldere complexă
- Pentru a ascunde implementarea internă și expune doar API-ul public
- Când vrei să faci refactoring fără să afectezi consumatorii

Testarea modulelor:

```
// tests/string-helper.test.js import StringHelper from '../src/utils/string-helper.js' // Testing framework simplu fără dependențe externe import assert from 'assert' const runTests = () => { console.log('Running StringHelper tests...') // Test capitalize assert.strictEqual( StringHelper.capitalize('hello'), 'Hello', 'capitalize should work for simple strings' ) // Cum testezi edge cases? // - Empty strings // - null/undefined // - Special characters // Cum organizezi testele pentru coverage complet? console.log('All tests passed!') } // Cum integrezi testele în workflow-ul de dezvoltare? // Ce script NPM adaugi pentru teste?
```

Exerciții practice

Sistem de Logging Simplu

Creează un sistem de logging organizat în 2 module simple pentru a practica organizarea codului.

Pași ghidați:

Pasul 1: Creează `logger.js` - modulul principal

```
const log = (level, message) => { // TODO: Implementează logica de bază }
```

Pasul 2: Creează `formatter.js` - formatarea mesajelor

```
const formatMessage = (level, message) => { const timestamp = new Date().toISOString() // TODO: Returnează mesajul formatat // Format: "[2024-01-15T10:30:00.000Z] INFO: Mesajul tau" }
```

Pasul 3: Integrează modulele în `app.js`

```
import { log } from './logger.js' // Testează cu diferite nivele: 'INFO', 'WARN', 'ERROR' log('INFO', 'Aplicația a pornit') log('WARN', 'Atenție: memorie puțină') log('ERROR', 'Eroare în conectarea la baza de date')
```

Cerințe:

- ✓ Mesajele să aibă timestamp
- ✓ Să afișeze nivelul de log (INFO, WARN, ERROR)
- ✓ Să coloreze diferit în consolă:
 - INFO: verde
 - WARN: galben
 - ERROR: roșu

💡 Hint pentru culori:

```
const colors = { INFO: '\\x1b[32m', // verde WARN: '\\x1b[33m', // galben ERROR: '\\x1b[31m', // roșu reset: '\\x1b[0m', // reset culoare } console.log(`${colors.INFO}This is a green message`)
```

🎯 **Obiectiv:** Să înveți să organizezi cod în module și să practici import/export.