Chapter 12: Hoàn Thiện Dự Án - Build Tools, Linting, Testing và Best Practices

Mô tả tổng quát

Chapter này tổng kết các kiến thức đã học, tập trung vào các công cụ và quy trình để hoàn thiện một dự án TypeScript chuyên nghiệp. Chúng ta sẽ khám phá các công cụ xây dựng (build tools) nâng cao, cách thiết lập linting và formatting để đảm bảo chất lượng code, các phương pháp kiểm thử (testing) cơ bản, và các thực hành tốt nhất (best practices) khi làm việc với TypeScript.

Tiêu đề

Hoàn Thiện Dự Án: Build Tools, Linting, Testing và Best Practices

Tóm tắt lý thuyết chính

1. Build Tools (Công cụ xây dựng)

Các dự án TypeScript lớn thường cần các công cụ xây dựng mạnh mẽ hơn ngoài `tsc` (TypeScript Compiler).

`tsc` và các tùy chọn nâng cao

- `--watch` (`-w`): Theo dõi thay đổi trong các file và tự động biên dịch lại, giúp tiết kiệm thời gian trong quá trình phát triển.
- `--project <path>` (`-p <path>`): Chỉ định đường dẫn đến file `tsconfig.json`, hữu ích khi có nhiều cấu hình trong dự án.
- `--build` (`-b`): Dùng cho project references, xây dựng dự án và các dependency của nó một cách hiệu quả.
- `outFile`: Gộp tất cả output thành một file JavaScript duy nhất. Thường dùng với `amd` hoặc `system`, ít phổ biến với module ES6/CommonJS.

- `incremental: true` : Trong `tsconfig.json` , bật biên dịch tăng dần, chỉ biên dịch các file thay đổi, giúp tăng tốc độ.

Sử dụng Webpack hoặc Parcel với TypeScript

- **Webpack**: Module bundler mạnh mẽ, phổ biến trong các dự án phức tạp.
    - Cần **loader** như `ts-loader` hoặc `awesome-typescript-loader` để xử lý file `.ts` .
    - Có thể kết hợp với `babel-loader` và `@babel/preset-typescript` để tận dụng hệ sinh thái Babel (ví dụ: polyfills, minification).
    - Hỗ trợ tối ưu hóa code (minification, tree shaking), code splitting, và quản lý asset (CSS, images).
    - **Giải thích chi tiết**: Webpack cho phép tùy chỉnh cao, phù hợp với dự án lớn, nhưng cần cấu hình phức tạp. Ví dụ, tree shaking giúp loại bỏ code không dùng đến, giảm kích thước bundle.
- **Parcel**: Module bundler "zero-configuration", dễ sử dụng cho dự án nhỏ.
    - Tự động phát hiện và biên dịch TypeScript mà không cần loader riêng.
    - Tốc độ nhanh, phù hợp khi muốn bắt đầu nhanh mà không cần cấu hình phức tạp.
    - **Giải thích chi tiết**: Parcel lý tưởng cho các dự án đơn giản hoặc prototype, nhưng ít tùy chỉnh hơn Webpack.

`nodemon`

Công cụ tự động khởi động lại ứng dụng Node.js khi có thay đổi trong file. Thường kết hợp với `tsc -w` hoặc `ts-node` để chạy TypeScript trực tiếp.

```
# Chạy file JS đã biên dịch
nodemon dist/server.js

# Chạy TypeScript trực tiếp với ts-node
nodemon --exec ts-node src/server.ts
```

- **Giải thích chi tiết:** `nodemon` giúp tăng tốc phát triển bằng cách tự động reload server khi code thay đổi, giảm thao tác thủ công.

2. Linting và Formatting (Kiểm tra và Định dạng Code)

Đảm bảo code nhất quán, dễ đọc, và tuân theo các quy tắc chung.

ESLint

Công cụ linting phổ biến cho JavaScript/TypeScript, giúp phát hiện lỗi cú pháp, code smells, và đảm bảo tuân thủ style guide.

- **Cần cài đặt:**
  - `@typescript-eslint/parser`: Cho phép ESLint hiểu cú pháp TypeScript.
  - `@typescript-eslint/eslint-plugin`: Cung cấp các rule dành riêng cho TypeScript.
- **Cấu hình** `.eslintrc.js`:

```
module.exports = {
  parser: '@typescript-eslint/parser',
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/recommended',
    'plugin:@typescript-eslint/recommended-requiring-type-checking',
    'prettier' // Tắt các rule xung đột với Prettier
  ],
  parserOptions: {
    ecmaVersion: 2020,
    sourceType: 'module',
    project: './tsconfig.json' // Cần cho rule yêu cầu thông tin kiểu
  },
  rules: {
    '@typescript-eslint/no-explicit-any': 'warn',
    '@typescript-eslint/no-unused-vars': ['error', { argsIgnorePattern: '^_'
  },
  ignorePatterns: ['dist/', 'node_modules/']
};
```

- **Giải thích chi tiết**:
  - `eslint:recommended`: Bộ rule cơ bản của ESLint.
  - `@typescript-eslint/recommended`: Bộ rule khuyến nghị cho TypeScript.
  - `project: './tsconfig.json'`: Cho phép ESLint sử dụng thông tin kiểu từ TypeScript để kiểm tra chính xác hơn.
  - `ignorePatterns`: Bỏ qua thư mục không cần lint như `dist/` hoặc `node_modules/`.

Prettier

Công cụ định dạng code tự động, đảm bảo style nhất quán (khoảng cách, dấu ngoặc, xuống dòng, v.v.).

- **Tích hợp với ESLint**:
  - `eslint-config-prettier`: Tắt các rule ESLint xung đột với Prettier.
  - `eslint-plugin-prettier`: Chạy Prettier như một rule ESLint.
- **Cấu hình** `.prettierrc.js`:

```
module.exports = {
  semi: true,
  trailingComma: 'es5',
  singleQuote: true,
  printWidth: 80,
  tabWidth: 2,
  useTabs: false,
  bracketSpacing: true
};
```

- **Giải thích chi tiết**:
  - Prettier tự động format code khi lưu file, giảm tranh cãi về style.
  - Tích hợp với ESLint giúp vừa kiểm tra lỗi (ESLint) vừa định dạng (Prettier).

Tích hợp với VS Code

- Cài extension **ESLint** và **Prettier** cho VS Code.
- Cấu hình VS Code để tự động format khi lưu:

```
{
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.formatOnSave": true,
  "eslint.validate": ["javascript", "typescript"]
}
```

- **Giải thích chi tiết**: Tích hợp này giúp hiển thị lỗi linting và tự động format code ngay trong editor, cải thiện trải nghiệm phát triển.

3. Testing (Kiểm thử)

Kiểm thử đảm bảo code hoạt động đúng và dễ bảo trì.

Unit Testing với Jest hoặc Mocha + Chai

- **Jest**: Testing framework "all-in-one", phổ biến cho React và TypeScript.
  - Cần `ts-jest` để xử lý file `.ts`.
  - **Cấu hình** `jest.config.js`:

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  roots: ['<rootDir>/src'],
  testMatch: ['**/__tests__/**/*.(ts|tsx|js)', '**/?(*.)(spec|test).(ts|tsx|
  transform: {
    '^.+\\.(ts|tsx)$': ['ts-jest', { tsconfig: 'tsconfig.json' }]
  },
  moduleNameMapper: {
    '\\.(css|less|scss)$': 'identity-obj-proxy'
  },
```

```
  setupFilesAfterEnv: ['<rootDir>/src/setupTests.ts']
};
```

- **Mocha + Chai**: Linh hoạt, thường dùng cho dự án cần tùy chỉnh cao.
  - Cần `ts-node` để chạy TypeScript trực tiếp.
  - Cấu hình trong `package.json` hoặc file `mocha.opts`.
- **Giải thích chi tiết**:
  - Jest dễ dùng hơn cho beginner nhờ cấu hình đơn giản và tích hợp mock, snapshot testing.
  - Mocha phù hợp khi cần kiểm soát chi tiết quy trình test.

Ví dụ test case

```
// File: src/utils/math.ts
export function add(a: number, b: number): number {
  return a + b;
}

// File: src/utils/__tests__/math.test.ts
import { add } from '../math';

describe('Math utility functions', () => {
  describe('add function', () => {
    it('should return the sum of two positive numbers', () => {
      expect(add(2, 3)).toBe(5);
    });
    it('should return the sum of a positive and a negative number', () => {
      expect(add(5, -2)).toBe(3);
    });
    it('should return zero when adding zero', () => {
      expect(add(7, 0)).toBe(7);
    });
  });
});
```

- **Giải thích chi tiết**:
  - `describe` nhóm các test case liên quan.
  - `it` định nghĩa từng test case với mô tả rõ ràng.
  - `expect` kiểm tra kết quả đầu ra của hàm.

4. TypeScript Best Practices (Thực hành tốt nhất)

- **Bật `strict` mode**: Trong `tsconfig.json`, đặt `strict: true` để bật các kiểm tra kiểu nghiêm ngặt (`strictNullChecks`, `noImplicitAny`, v.v.), giúp phát hiện lỗi sớm.
- **Ưu tiên `unknown` hơn `any`**: `unknown` buộc kiểm tra kiểu trước khi sử dụng, an toàn hơn `any`.
- **Khai báo kiểu tường minh cho API**: Dùng `interface` hoặc `type` để định nghĩa kiểu rõ ràng cho dữ liệu từ API hoặc module.
- **Tận dụng Utility Types**: Sử dụng `Partial`, `Readonly`, `Pick`, `Omit` để tạo kiểu an toàn và ngắn gọn.
- **Code dễ đọc, dễ bảo trì**: Đặt tên biến/hàm/class rõ ràng, chia nhỏ code thành module/hàm với mục đích cụ thể.
- **Dùng `readonly`**: Đánh dấu thuộc tính không thay đổi để tăng tính bất biến.
- **Tránh lạm dụng type assertions (`as Type`)**: Chỉ dùng khi chắc chắn về kiểu, tránh tắt kiểm tra của TypeScript.
- **Dùng ESLint và Prettier**: Đảm bảo code nhất quán và tuân thủ quy tắc.
- **Viết Unit Tests**: Đảm bảo code đúng và dễ refactor.
- **Cập nhật TypeScript và `@types` packages**: Tận dụng tính năng mới và bản vá lỗi.
- **Giải thích chi tiết**:

- `strict` mode giúp code an toàn hơn nhưng có thể cần thêm công sức xử lý `null` / `undefined`.
- Utility Types giảm code lặp và tăng tính linh hoạt.
- Type assertions nên dùng cẩn thận, ví dụ khi làm việc với thư viện không có type definitions.

5. Ôn tập và Q&A

Học viên có thể đặt câu hỏi về bất kỳ chủ đề nào trong khóa học, chia sẻ kinh nghiệm hoặc thảo luận các tình huống thực tế khi áp dụng TypeScript.

Code ví dụ chính

Ví dụ `package.json` scripts

```
{
  "scripts": {
    "build": "tsc",
    "build:watch": "tsc -w",
    "start": "node dist/server.js",
    "dev": "nodemon --exec ts-node src/server.ts",
    "lint": "eslint . --ext .ts,.tsx",
    "lint:fix": "eslint . --ext .ts,.tsx --fix",
    "format": "prettier --write \"src/**/*.ts\"",
    "test": "jest",
    "test:watch": "jest --watch"
  }
}
```

Danh sách bài tập

1. Trắc nghiệm: Công cụ phân tích tĩnh code

Tiêu đề: Hiểu về Linting
Mô tả: Chọn công cụ phù hợp cho phân tích tĩnh code.
Câu hỏi: Công cụ nào thường được sử dụng để phân tích tĩnh code TypeScript, tìm lỗi tiềm ẩn, code smells, và vấn đề về style?

- A. Jest
- B. Webpack
- C. ESLint
- D. Node.js
  Đáp án: C
  Giải thích: ESLint là công cụ phân tích tĩnh code, kiểm tra cú pháp, style, và lỗi tiềm ẩn. Jest dùng cho testing, Webpack cho bundling, và Node.js là runtime.

2. Thực hành: Cài đặt ESLint và Prettier

Tiêu đề: Thiết lập môi trường Linting và Formatting
Mô tả:

- Cài đặt ESLint, `@typescript-eslint/parser`, `@typescript-eslint/eslint-plugin`.
- Cài đặt Prettier, `eslint-config-prettier`, `eslint-plugin-prettier`.
- Tạo file `.eslintrc.js` và `.prettierrc.js`.
- (Tùy chọn) Cấu hình VS Code để format khi lưu và hiển thị cảnh báo ESLint.
- Viết code TypeScript có lỗi style/linting và kiểm tra ESLint/Prettier.

Gợi ý cài đặt:

```
npm init -y
npm install typescript --save-dev
npm install eslint @typescript-eslint/parser @typescript-eslint/eslint-plugi
npm install prettier eslint-config-prettier eslint-plugin-prettier --save-de
npx eslint --init
```

- **Giải thích chi tiết:** Bài tập giúp học viên làm quen với quy trình thiết lập môi trường linting/formatting, từ cài đặt đến tích hợp editor.

3. Thực hành: Cài đặt Jest và viết Unit Test

Tiêu đề: Viết Unit Test cơ bản
Mô tả:

- Cài đặt Jest, `ts-jest`, `@types/jest`.
- Tạo file `jest.config.js`.
- Viết hàm `capitalize(str: string): string` (viết hoa chữ cái đầu).
- Viết file test `capitalize.test.ts` với 2-3 test case.
- Chạy test bằng `npm test`.

Ví dụ hàm `capitalize`:

```
export function capitalize(str: string): string {
  if (!str) return '';
  return str.charAt(0).toUpperCase() + str.slice(1);
}
```

Ví dụ test:

```
import { capitalize } from '../utils';

describe('capitalize function', () => {
  it('should capitalize the first letter of a string', () => {
    expect(capitalize('hello')).toBe('Hello');
  });
  it('should return empty string for empty input', () => {
    expect(capitalize('')).toBe('');
  });
  it('should not change already capitalized string', () => {
    expect(capitalize('Hello')).toBe('Hello');
```

```
  });
});
```

- **Giải thích chi tiết**: Bài tập giúp học viên làm quen với Jest, viết test case, và kiểm tra tính đúng đắn của hàm.

4. Thảo luận: Lợi ích của `strictNullChecks`

Tiêu đề: Thảo luận về `strictNullChecks`
Mô tả:

- **Lợi ích**: Giảm lỗi `Cannot read property '…' of null/undefined`, buộc xử lý `null` / `undefined` tường minh, tăng an toàn code.
- **Thách thức**: Cần thêm code để kiểm tra `null` / `undefined`, đặc biệt khi làm việc với API hoặc code JavaScript cũ.
- **Giải thích chi tiết**: Thảo luận giúp học viên hiểu tầm quan trọng của kiểm tra kiểu nghiêm ngặt và cách xử lý các trường hợp đặc biệt.

5. Code Review: Áp dụng Best Practices

Tiêu đề: Áp dụng Best Practices
Mô tả: Review đoạn code sau và đề xuất cải thiện dựa trên best practices:

```
function processData(data: any, options: any) {
  let result;
  if (options.type === 'string') {
    result = String(data).toUpperCase();
  } else if (options.type === 'number') {
    result = Number(data) * options.factor;
  }
  return result;
}

class UserProfile {
  id;
  name;
```

```
  email;
  constructor(id, name, email) {
    this.id = id;
    this.name = name;
    this.email = email;
  }
  updateProfile(newData) {
    this.name = newData.name ? newData.name : this.name;
    this.email = newData.email ? newData.email : this.email;
  }
}
```

Đề xuất cải thiện:

1. **Tránh `any` trong `processData`:**
   - Định nghĩa interface cho `data` và `options`.
   - Dùng `unknown` nếu kiểu chưa rõ, kết hợp type guard để kiểm tra.
2. **Khai báo kiểu tường minh cho `UserProfile`:**
   - Thêm kiểu cho các thuộc tính và tham số constructor.
   - Dùng `readonly` cho `id` vì không nên thay đổi.
3. **Định nghĩa kiểu cho `newData` trong `updateProfile`:**
   - Dùng interface để khai báo kiểu của `newData`.
4. **Kiểm tra `null` / `undefined` tường minh:**
   - Dùng `strictNullChecks` để bắt lỗi sớm.

Code cải thiện:

```
interface ProcessOptions {
  type: 'string' | 'number';
  factor?: number;
}
```

```
function processData(data: unknown, options: ProcessOptions): string | numbe
  if (options.type === 'string') {
    return typeof data === 'string' ? data.toUpperCase() : String(data).toUp
  } else if (options.type === 'number') {
    return typeof data === 'number' && options.factor ? data * options.facto
  }
}

interface UserProfileData {
  name?: string;
  email?: string;
}

class UserProfile {
  readonly id: string;
  name: string;
  email: string;

  constructor(id: string, name: string, email: string) {
    this.id = id;
    this.name = name;
    this.email = email;
  }

  updateProfile(newData: Partial<UserProfileData>) {
    this.name = newData.name ?? this.name;
    this.email = newData.email ?? this.email;
  }
}
```

- **Giải thích chi tiết:**
  - Interface `ProcessOptions` và `UserProfileData` làm rõ cấu trúc dữ liệu.
  - `unknown` thay vì `any` trong `processData` tăng an toàn.
  - `Partial<UserProfileData>` cho phép `newData` có các thuộc tính tùy chọn.
  - `??` thay vì toán tử `?` để xử lý `null` / `undefined` ngắn gọn hơn.

Kết luận

Chapter này cung cấp kiến thức toàn diện về cách hoàn thiện dự án TypeScript với build tools, linting, testing, và best practices. Học viên được khuyến khích áp dụng vào dự án thực tế và thảo luận để củng cố kiến thức.