← QE Fundamentals & TypeScript

Playwright Basics →

Asynchronous JavaScript, TypeScript & Node.js

Table of Contents

Synchronous vs Asynchronous Programming

complete before the next one starts.

Example:

```
 console.log("First");
console.log("Second");
console.log("Third");

// Output (always in order):
// First
// Second
// Third
```

Characteristics:

- Predictable execution order
- Blocking operations
- Simple to understand
- Can cause performance issues with slow operations

Real-world Example:

```
 function calculateSum(numbers: number[]): number {
  let sum = 0;
  for (const num of numbers) {
    sum += num;
  }
  return sum;
}

const result = calculateSum([1, 2, 3, 4, 5]);
console.log("Sum:", result);
// This blocks until calculation is complete
```

Asynchronous Programming

Definition: Code can start an operation and continue executing other code while waiting for the operation to complete.

```
console.log("First");

setTimeout(() => {
  console.log("Second (after 1 second)");
}, 1000);

console.log("Third");

// Output:
// First
// Third
// Second (after 1 second)
```

Characteristics:

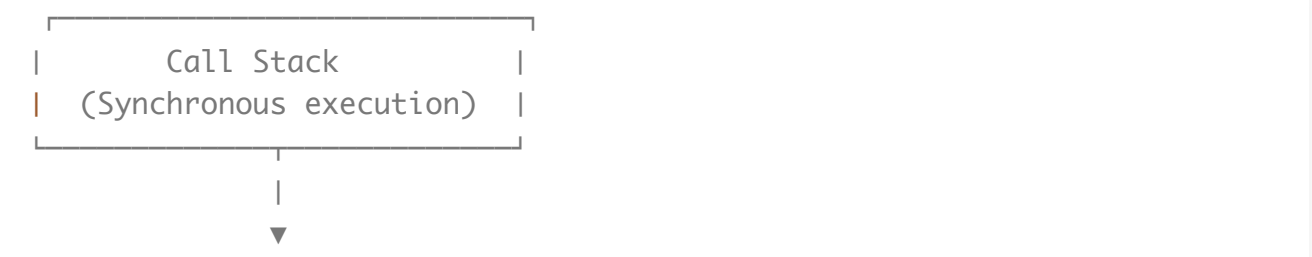
- Non-blocking operations
- Better performance for I/O operations
- More complex control flow
- Essential for modern web applications

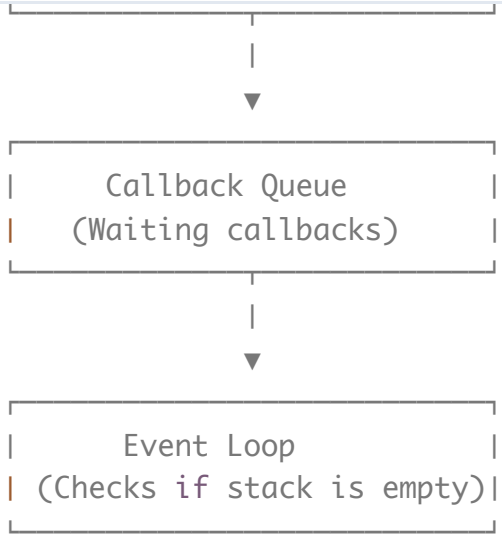
When to Use Async:

- Network requests (API calls)
- File system operations
- Database queries
- Timers and delays
- User input/events

The Event Loop

JavaScript uses an **event loop** to handle asynchronous operations:





How it works:

1. Synchronous code runs on the call stack
2. Async operations go to Web APIs
3. When complete, callbacks move to callback queue
4. Event loop checks if call stack is empty
5. If empty, moves callbacks from queue to stack

Callbacks and Callback Hell

What is a Callback?

A **callback** is a function passed as an argument to another function, to be executed later.

Simple Callback Example:

```
function greet(name: string, callback: () => void): void {
  console.log(`Hello, ${name}!`);
  callback();
}
```

```
greet("Alice", sayGoodbye);
// Output:
// Hello, Alice!
// Goodbye!
```

Async Callbacks

setTimeout Example:

```
 console.log("Start");

setTimeout(() => {
  console.log("This runs after 2 seconds");
}, 2000);

console.log("End");

// Output:
// Start
// End
// This runs after 2 seconds
```

File Reading Example:

```
 import fs from 'fs';

fs.readFile('data.txt', 'utf8', (error, data) => {
  if (error) {
    console.error("Error reading file:", error);
    return;
  }
  console.log("File content:", data);
});

console.log("Reading file...");
// "Reading file..." logs first
// Then file content logs when reading completes
```

Bad Example:

```
// Callback hell - hard to read and maintain
fs.readFile('user.txt', 'utf8', (err, userId) => {
  if (err) {
    console.error(err);
    return;
  }

  database.getUser(userId, (err, user) => {
    if (err) {
      console.error(err);
      return;
    }

    database.getUserPosts(user.id, (err, posts) => {
      if (err) {
        console.error(err);
        return;
      }

      posts.forEach(post => {
        database.getComments(post.id, (err, comments) => {
          if (err) {
            console.error(err);
            return;
          }

          console.log(`Post: ${post.title}, Comments: ${comments.length}`)
        });
      });
    });
  });
});
```

Problems with Callback Hell:

- Hard to read (deeply nested)
- Difficult to debug
- Error handling repeated everywhere

Solution: Use Promises or async/await (covered next)

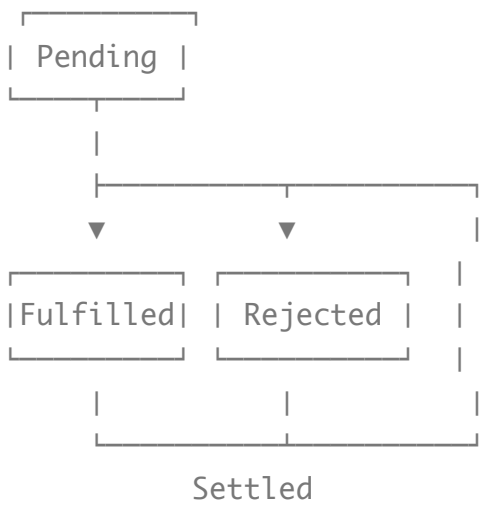
Promises

What is a Promise?

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

Promise States:

- **Pending**: Initial state, neither fulfilled nor rejected
- **Fulfilled**: Operation completed successfully
- **Rejected**: Operation failed



Creating Promises

Basic Promise:

```
const promise = new Promise<string>((resolve, reject) => {
  // Simulate async operation
```

```
    if (success) {
      resolve("Operation successful!");
    } else {
      reject(new Error("Operation failed!"));
    }
  }, 1000);
});
```

Promise that Resolves:

```
function fetchUserData(userId: number): Promise<User> {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const user: User = {
        id: userId,
        name: "Alice",
        email: "alice@example.com"
      };
      resolve(user);
    }, 1000);
  });
}
```

Promise that Rejects:

```
function divideNumbers(a: number, b: number): Promise<number> {
  return new Promise((resolve, reject) => {
    if (b === 0) {
      reject(new Error("Cannot divide by zero"));
    } else {
      resolve(a / b);
    }
  });
}
```

Consuming Promises

Using .then() and .catch():

```
    return user.name;
  })
  .then(name => {
    console.log("Name:", name);
  })
  .catch(error => {
    console.error("Error:", error);
  })
  .finally(() => {
    console.log("Operation complete");
  });
```

Chaining Promises:

```
interface User {
  id: number;
  name: string;
}

interface Post {
  id: number;
  userId: number;
  title: string;
}

function getUser(id: number): Promise<User> {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ id, name: "Alice" });
    }, 1000);
  });
}

function getUserPosts(userId: number): Promise<Post[]> {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve([
        { id: 1, userId, title: "First Post" },
        { id: 2, userId, title: "Second Post" }
      ]);
    }, 1000);
  });
}
```

```
  .then(user => {
    console.log("User:", user);
    return getUserPosts(user.id);
  })
  .then(posts => {
    console.log("Posts:", posts);
  })
  .catch(error => {
    console.error("Error:", error);
  });
```

Promise Methods

Promise.all() - Wait for all promises:

```
 const promise1 = fetchUserData(1);
const promise2 = fetchUserData(2);
const promise3 = fetchUserData(3);

Promise.all([promise1, promise2, promise3])
  .then(users => {
    console.log("All users:", users);
    // All users: [user1, user2, user3]
  })
  .catch(error => {
    console.error("At least one failed:", error);
  });
```

Promise.allSettled() - Wait for all, regardless of outcome:

```
 const promises = [
  fetchUserData(1),
  Promise.reject(new Error("Failed")),
  fetchUserData(3)
];

Promise.allSettled(promises)
  .then(results => {
    results.forEach(result => {
      if (result.status === 'fulfilled') {
        console.log("Success:", result.value);
```

```
    });
  });

// Output:
// Success: { id: 1, name: "Alice", ... }
// Failed: Error: Failed
// Success: { id: 3, name: "Alice", ... }
```

Promise.race() - First to complete wins:

```
const fast = new Promise(resolve => setTimeout(() => resolve("Fast"), 100)
const slow = new Promise(resolve => setTimeout(() => resolve("Slow"), 1000

Promise.race([fast, slow])
  .then(result => {
    console.log("Winner:", result); // "Fast"
  });
```

Promise.any() - First to resolve wins:

```
const promises = [
  Promise.reject(new Error("Error 1")),
  new Promise(resolve => setTimeout(() => resolve("Success 1"), 1000)),
  new Promise(resolve => setTimeout(() => resolve("Success 2"), 500))
];

Promise.any(promises)
  .then(result => {
    console.log("First success:", result); // "Success 2"
  })
  .catch(error => {
    console.error("All failed:", error);
  });
```

Utility Functions

Promisify a callback-based function:

```
// Convert callback-based to promise-based
const readFileAsync = promisify(fs.readFile);

// Now use with promises
readFileAsync('data.txt', 'utf8')
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Delay utility:

```
 function delay(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// Usage
delay(2000)
  .then(() => console.log("2 seconds passed"));
```

---

async/await

What is async/await?

async/await provides syntactic sugar over Promises, making asynchronous code look and behave like synchronous code.

Key Points:

- `async` keyword declares an async function
- `async` functions always return a Promise
- `await` keyword pauses execution until Promise resolves
- `await` can only be used inside `async` functions
- Much cleaner than promise chains

```
// Promise-based
function getUserDataPromise(id: number): Promise<User> {
  return fetchUserData(id)
    .then(user => {
      console.log("User:", user);
      return user;
    });
}

// async/await - cleaner
async function getUserDataAsync(id: number): Promise<User> {
  const user = await fetchUserData(id);
  console.log("User:", user);
  return user;
}
```

Async Function Always Returns Promise:

```
async function greet(name: string): Promise<string> {
  return `Hello, ${name}!`;
}

// Calling async function
greet("Alice").then(message => console.log(message));

// Or with await
const message = await greet("Alice");
console.log(message);
```

Sequential vs Parallel Execution

Sequential Execution (slow):

```
async function fetchAllUsersSequential() {
  console.time("Sequential");

  const user1 = await fetchUserData(1); // Wait 1 second
  const user2 = await fetchUserData(2); // Wait 1 second
```

```
  return [user1, user2, user3];
}
```

Parallel Execution (fast):

```
async function fetchAllUsersParallel() {
  console.time("Parallel");

  // Start all requests simultaneously
  const promise1 = fetchUserData(1);
  const promise2 = fetchUserData(2);
  const promise3 = fetchUserData(3);

  // Wait for all to complete
  const users = await Promise.all([promise1, promise2, promise3]);

  console.timeEnd("Parallel"); // ~1 second
  return users;
}
```

When to use sequential vs parallel:

- **Sequential**: When operations depend on each other
- **Parallel**: When operations are independent

Chaining with async/await

Replacing Promise Chains:

```
// Promise chains (harder to read)
function getUserWithPosts(userId: number) {
  return getUser(userId)
    .then(user => {
      console.log("User:", user);
      return getUserPosts(user.id);
    })
    .then(posts => {
      console.log("Posts:", posts);
      return posts;
    });
```

```
async function getUserWithPostsAsync(userId: number) {
  const user = await getUser(userId);
  console.log("User:", user);

  const posts = await getUserPosts(user.id);
  console.log("Posts:", posts);

  return posts;
}
```

Complex Async Flows

Multiple dependent operations:

```
interface User {
  id: number;
  name: string;
}

interface Post {
  id: number;
  title: string;
}

interface Comment {
  id: number;
  text: string;
}

async function getCompleteUserData(userId: number) {
  // Get user
  const user = await getUser(userId);

  // Get user's posts (parallel)
  const posts = await getUserPosts(user.id);

  // Get comments for all posts (parallel)
  const commentPromises = posts.map(post => getPostComments(post.id));
  const commentsArrays = await Promise.all(commentPromises);

  // Flatten comments
  const allComments = commentsArrays.flat();
```

```
    commentsCount: allComments.length,
    data: {
      posts,
      comments: allComments
    }
  };
}
```

Conditional async operations:

```
async function processUser(userId: number, includeDetails: boolean = false
  const user = await getUser(userId);

  if (includeDetails) {
    const posts = await getUserPosts(user.id);
    const followers = await getUserFollowers(user.id);

    return {
      ...user,
      posts,
      followers
    };
  }

  return user;
}
```

Error Handling in Async Code

Try-Catch with async/await

Basic Error Handling:

```
async function fetchUserSafe(userId: number): Promise<User | null> {
  try {
    const user = await fetchUserData(userId);
    return user;
```

```
  }
}
```

Multiple Operations:

```
async function getUserWithDetails(userId: number) {
  try {
    const user = await getUser(userId);
    const posts = await getUserPosts(user.id);
    const comments = await getUserComments(user.id);

    return { user, posts, comments };
  } catch (error) {
    if (error instanceof TypeError) {
      console.error("Type error:", error.message);
    } else if (error instanceof Error) {
      console.error("General error:", error.message);
    } else {
      console.error("Unknown error:", error);
    }
    throw error; // Re-throw for caller to handle
  }
}
```

Error Handling with Promise.all

All or nothing approach:

```
async function fetchMultipleUsers(userIds: number[]) {
  try {
    const promises = userIds.map(id => fetchUserData(id));
    const users = await Promise.all(promises);
    return users;
  } catch (error) {
    console.error("At least one user fetch failed:", error);
    return [];
  }
}
```

Handle errors individually:

```
      return await fetchUserData(id);
    } catch (error) {
      console.error(`Failed to fetch user ${id}:`, error);
      return null;
    }
  });

  const results = await Promise.all(promises);
  return results.filter(user => user !== null) as User[];
}
```

Custom Error Classes

Create custom errors:

```
class NetworkError extends Error {
  constructor(
    message: string,
    public statusCode: number,
    public endpoint: string
  ) {
    super(message);
    this.name = "NetworkError";
  }
}

class ValidationError extends Error {
  constructor(
    message: string,
    public field: string
  ) {
    super(message);
    this.name = "ValidationError";
  }
}

async function fetchData(url: string): Promise<any> {
  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new NetworkError(
```

```
      );
    }

    const data = await response.json();

    if (!data.id) {
      throw new ValidationError("Missing ID field", "id");
    }

    return data;
  } catch (error) {
    if (error instanceof NetworkError) {
      console.error(`Network error at ${error.endpoint}: ${error.statusCod
    } else if (error instanceof ValidationError) {
      console.error(`Validation error on field ${error.field}`);
    } else {
      console.error("Unknown error:", error);
    }
    throw error;
  }
}
```

Retry Logic

Retry failed operations:

```
async function fetchWithRetry<T>(
  fetchFn: () => Promise<T>,
  maxRetries: number = 3,
  delay: number = 1000
): Promise<T> {
  let lastError: Error;

  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      console.log(`Attempt ${attempt}/${maxRetries}`);
      return await fetchFn();
    } catch (error) {
      lastError = error as Error;
      console.error(`Attempt ${attempt} failed:`, error);

      if (attempt < maxRetries) {
        console.log(`Retrying in ${delay}ms...`);
```

```
  }

  throw new Error(`Failed after ${maxRetries} attempts: ${lastError!.messa
}

// Usage
const userData = await fetchWithRetry(() => fetchUserData(1), 3, 2000);
```

Timeout Pattern

Add timeout to async operations:

```
function timeout<T>(promise: Promise<T>, ms: number): Promise<T> {
  return Promise.race([
    promise,
    new Promise<T>((_, reject) =>
      setTimeout(() => reject(new Error("Operation timed out")), ms)
    )
  ]);
}

// Usage
try {
  const user = await timeout(fetchUserData(1), 5000);
  console.log("User:", user);
} catch (error) {
  if (error.message === "Operation timed out") {
    console.error("Request took too long");
  } else {
    console.error("Other error:", error);
  }
}
```

Node.js Fundamentals

What is Node.js?

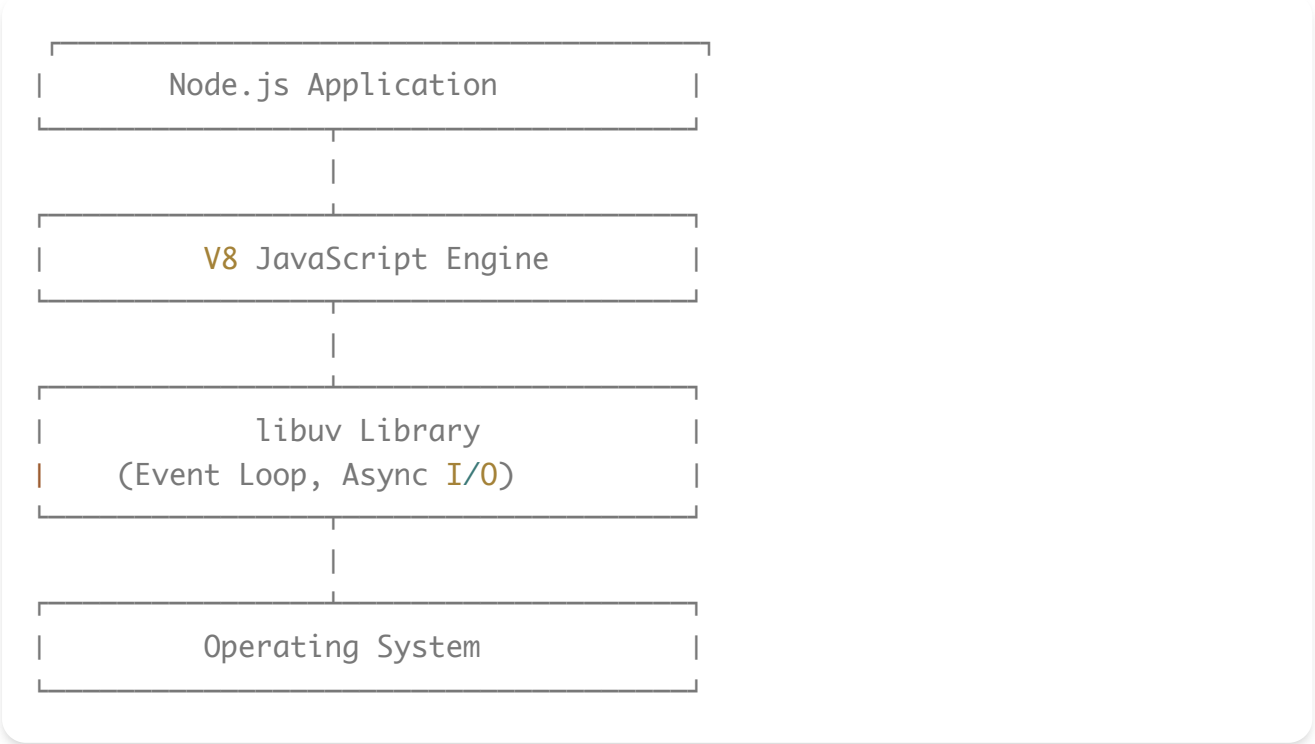
Key Features:

- **Event-driven**: Uses an event loop
- **Non-blocking I/O**: Asynchronous operations
- **Single-threaded**: But can handle many connections
- **Fast**: V8 engine compiles JavaScript to machine code
- **npm ecosystem**: Largest package ecosystem

Use Cases:

- REST APIs
- Real-time applications (chat, gaming)
- Microservices
- Command-line tools
- Test automation frameworks
- Build tools

Node.js Architecture



```
// Available everywhere without import
console.log("Hello");
console.error("Error message");
console.table([{ name: "Alice", age: 30 }]);

// Global variables
console.log(__dirname);  // Current directory path
console.log(__filename); // Current file path
console.log(process.cwd()); // Current working directory

// Process object
console.log(process.version);    // Node.js version
console.log(process.platform);   // Operating system
console.log(process.env);        // Environment variables
```

2. Built-in Modules:

```
// Import built-in modules
import path from 'path';
import fs from 'fs';
import os from 'os';
import http from 'http';

// Path module
const filePath = path.join(__dirname, 'data', 'users.json');
const ext = path.extname(filePath); // .json
const base = path.basename(filePath); // users.json

// OS module
console.log(os.platform());  // darwin, linux, win32
console.log(os.cpus().length); // Number of CPU cores
console.log(os.totalmem());  // Total memory
console.log(os.freemem());   // Free memory
```

Process Management

Command Line Arguments:

```
console.log(process.argv);
// [
//   '/path/to/node',
//   '/path/to/script.js',
//   'arg1',
//   'arg2'
// ]

const args = process.argv.slice(2); // Get user arguments
console.log("Arguments:", args);
```

Exit Codes:

```
 // Exit with success
process.exit(0);

// Exit with error
process.exit(1);

// Exit handlers
process.on('exit', (code) => {
  console.log(`Process exiting with code: ${code}`);
});

process.on('SIGINT', () => {
  console.log('Received SIGINT. Gracefully shutting down...');
  process.exit(0);
});
```

Node.js Modules

CommonJS (require/exports)

Exporting:

```
}

function subtract(a, b) {
  return a - b;
}

const PI = 3.14159;

// Export individual items
module.exports.add = add;
module.exports.subtract = subtract;
module.exports.PI = PI;

// Or export object
module.exports = {
  add,
  subtract,
  PI
};
```

Importing:

```
// app.js
const math = require('./math');

console.log(math.add(5, 3));      // 8
console.log(math.subtract(10, 4)); // 6
console.log(math.PI);              // 3.14159

// Destructuring
const { add, subtract } = require('./math');
console.log(add(2, 3)); // 5
```

ES6 Modules (import/export)

Note: To use ES6 modules in Node.js, set `"type": "module"` in package.json or use `.mjs` extension.

Exporting:

```
}

export function subtract(a: number, b: number): number {
  return a - b;
}

export const PI = 3.14159;

// Default export
export default class Calculator {
  multiply(a: number, b: number): number {
    return a * b;
  }
}
```

Importing:

```
// app.ts
import Calculator, { add, subtract, PI } from './math.js';

console.log(add(5, 3));      // 8
console.log(subtract(10, 4)); // 6
console.log(PI);              // 3.14159

const calc = new Calculator();
console.log(calc.multiply(4, 5)); // 20

// Import everything
import * as math from './math.js';
console.log(math.add(2, 3)); // 5
```

Module Pattern for Test Automation

page.ts (Page Object):

```
import { Page } from '@playwright/test';

export class LoginPage {
  constructor(private page: Page) {}
```

```
  async login(email: string, password: string) {
    await this.page.fill('#email', email);
    await this.page.fill('#password', password);
    await this.page.click('button[type="submit"]');
  }
}
```

test.spec.ts:

```
 import { test, expect } from '@playwright/test';
import { LoginPage } from './pages/login.page';

test('User can login', async ({ page }) => {
  const loginPage = new LoginPage(page);
  await loginPage.navigateTo();
  await loginPage.login('user@example.com', 'password');
  await expect(page).toHaveURL('/dashboard');
});
```

File System Operations

Reading Files

Async (Recommended):

```
 import fs from 'fs/promises';

// Read file
async function readFile(filePath: string): Promise<string> {
  try {
    const data = await fs.readFile(filePath, 'utf8');
    return data;
  } catch (error) {
    console.error('Error reading file:', error);
    throw error;
  }
```

```
const content = await readFile('data.txt');
console.log(content);
```

Sync (Blocking - Avoid in production):

```
import fs from 'fs';

try {
  const data = fs.readFileSync('data.txt', 'utf8');
  console.log(data);
} catch (error) {
  console.error('Error:', error);
}
```

Callback-based (Old way):

```
import fs from 'fs';

fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log(data);
});
```

Writing Files

Write file:

```
import fs from 'fs/promises';

async function writeFile(filePath: string, content: string) {
  try {
    await fs.writeFile(filePath, content, 'utf8');
    console.log('File written successfully');
  } catch (error) {
    console.error('Error writing file:', error);
  }
```

Append to file:

```
async function appendToFile(filePath: string, content: string) {
  try {
    await fs.appendFile(filePath, content, 'utf8');
    console.log('Content appended');
  } catch (error) {
    console.error('Error appending:', error);
  }
}

await appendToFile('log.txt', 'New log entry\n');
```

Working with Directories

Create directory:

```
async function createDirectory(dirPath: string) {
  try {
    await fs.mkdir(dirPath, { recursive: true });
    console.log('Directory created');
  } catch (error) {
    console.error('Error creating directory:', error);
  }
}

await createDirectory('test-results/screenshots');
```

Read directory:

```
async function listFiles(dirPath: string): Promise<string[]> {
  try {
    const files = await fs.readdir(dirPath);
    return files;
  } catch (error) {
    console.error('Error reading directory:', error);
    return [];
  }
```

```
console.log('Files:', files);
```

Delete file/directory:

```
async function deleteFile(filePath: string) {
  try {
    await fs.unlink(filePath);
    console.log('File deleted');
  } catch (error) {
    console.error('Error deleting file:', error);
  }
}

async function deleteDirectory(dirPath: string) {
  try {
    await fs.rm(dirPath, { recursive: true, force: true });
    console.log('Directory deleted');
  } catch (error) {
    console.error('Error deleting directory:', error);
  }
}
```

File Stats and Checks

Check if file/directory exists:

```
async function fileExists(filePath: string): Promise<boolean> {
  try {
    await fs.access(filePath);
    return true;
  } catch {
    return false;
  }
}

const exists = await fileExists('data.txt');
console.log('File exists:', exists);
```

Get file stats:

```
    return {
      size: stats.size,
      isFile: stats.isFile(),
      isDirectory: stats.isDirectory(),
      created: stats.birthtime,
      modified: stats.mtime
    };
  } catch (error) {
    console.error('Error getting file info:', error);
    return null;
  }
}

const info = await getFileInfo('data.txt');
console.log('File info:', info);
```

JSON File Operations

Read JSON file:

```
interface User {
  id: number;
  name: string;
  email: string;
}

async function readJsonFile<T>(filePath: string): Promise<T> {
  const content = await fs.readFile(filePath, 'utf8');
  return JSON.parse(content) as T;
}

const users = await readJsonFile<User[]>('users.json');
console.log('Users:', users);
```

Write JSON file:

```
async function writeJsonFile<T>(filePath: string, data: T) {
  const content = JSON.stringify(data, null, 2);
  await fs.writeFile(filePath, content, 'utf8');
```

```
  { id: 1, name: 'Alice', email: 'alice@example.com' },
  { id: 2, name: 'Bob', email: 'bob@example.com' }
];

await writeJsonFile('users.json', users);
```

---

npm and Package Management

What is npm?

npm (Node Package Manager) is the default package manager for Node.js.

Features:

- Install packages
- Manage dependencies
- Run scripts
- Publish packages

package.json

Create package.json:

```
npm init          # Interactive setup
npm init -y       # Use defaults
```

Example package.json:

```
{
  "name": "test-automation-project",
  "version": "1.0.0",
  "description": "Test automation framework",
  "main": "index.js",
  "scripts": {
```

```
    "report": "playwright show-report"
  },
  "keywords": ["automation", "testing"],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "@playwright/test": "^1.40.0"
  },
  "devDependencies": {
    "typescript": "^5.3.0",
    "@types/node": "^20.10.0"
  }
}
```

Installing Packages

Install production dependency:

```
npm install playwright
npm i playwright           # Short form
```

Install dev dependency:

```
npm install --save-dev typescript
npm i -D typescript        # Short form
```

Install globally:

```
npm install -g typescript
```

Install specific version:

```
npm install playwright@1.40.0
```

Install from GitHub:

Managing Dependencies

Update packages:

```
npm update            # Update all packages
npm update playwright # Update specific package
```

Remove packages:

```
npm uninstall playwright
npm un playwright     # Short form
```

List installed packages:

```
npm list              # All packages
npm list --depth=0    # Top-level only
npm list -g --depth=0 # Global packages
```

Check outdated packages:

```
npm outdated
```

npm Scripts

Define scripts in package.json:

```
{
  "scripts": {
    "start": "node dist/index.js",
    "build": "tsc",
    "test": "playwright test",
    "test:smoke": "playwright test --grep @smoke",
    "test:regression": "playwright test --grep @regression",
    "lint": "eslint src/**/*.ts",
    "format": "prettier --write src/**/*.ts"
```

Run scripts:

```
 npm run build
npm test              # Shortcut for npm run test
npm run test:smoke
```

Pre and Post Scripts:

```
 {
  "scripts": {
    "pretest": "npm run lint",
    "test": "playwright test",
    "posttest": "npm run report"
  }
}
```

package-lock.json

Purpose:

- Locks exact versions of dependencies
- Ensures consistent installs across environments
- Should be committed to version control

Commands:

```
 npm ci    # Clean install (uses package-lock.json)
npm install # May update package-lock.json
```

Environment Variables

What are Environment Variables?

Common Uses:

- API keys
- Database credentials
- Environment-specific URLs
- Feature flags

Using .env Files

Install dotenv:

```
npm install dotenv
```

Create .env file:

```
API_URL=https://api.example.com
API_KEY=your_secret_key_here
DB_HOST=localhost
DB_PORT=5432
DB_NAME=testdb
ENVIRONMENT=development
```

Load environment variables:

```
import dotenv from 'dotenv';

// Load .env file
dotenv.config();

// Access variables
const apiUrl = process.env.API_URL;
const apiKey = process.env.API_KEY;

console.log('API URL:', apiUrl);
```

Multiple Environment Files

- `.env.development` - Development
- `.env.staging` - Staging
- `.env.production` - Production

Load specific env file:

```
 import dotenv from 'dotenv';
import path from 'path';

const environment = process.env.NODE_ENV || 'development';
const envFile = `.env.${environment}`;

dotenv.config({ path: path.resolve(process.cwd(), envFile) });

console.log('Environment:', environment);
console.log('API URL:', process.env.API_URL);
```

Type-Safe Environment Variables

Create config.ts:

```
 interface Config {
  apiUrl: string;
  apiKey: string;
  dbHost: string;
  dbPort: number;
  environment: 'development' | 'staging' | 'production';
}

function loadConfig(): Config {
  const requiredVars = ['API_URL', 'API_KEY', 'DB_HOST'];

  for (const varName of requiredVars) {
    if (!process.env[varName]) {
      throw new Error(`Missing required environment variable: ${varName}`)
    }
  }

  return {
    apiUrl: process.env.API_URL!,
```

```
    environment: (process.env.ENVIRONMENT as Config['environment']) || 'de
  };
}

export const config = loadConfig();
```

Usage:

```
import { config } from './config';

console.log('API URL:', config.apiUrl);
console.log('Environment:', config.environment);
```

Security Best Practices

1. Never commit .env to version control:

Create .gitignore:

```
.env
.env.*
!.env.example
node_modules/
```

2. Create .env.example:

```
API_URL=https://api.example.com
API_KEY=your_api_key_here
DB_HOST=localhost
DB_PORT=5432
```

3. Use secrets management in CI/CD:

```
# GitHub Actions example
jobs:
  test:
    runs-on: ubuntu-latest
```

```
    steps:
      - uses: actions/checkout@v2
      - run: npm test
```

Practice Exercises

Exercise 1: Promise Practice

Task: Create a User API Simulator

```
interface User {
  id: number;
  name: string;
  email: string;
}

// Simulate API delay
function delay(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// Simulate fetching user (implement this)
function fetchUser(id: number): Promise<User> {
  // Your code: Return a promise that resolves after 1 second
  // with a user object
}

// Simulate fetching multiple users (implement this)
function fetchMultipleUsers(ids: number[]): Promise<User[]> {
  // Your code: Use Promise.all to fetch multiple users
}

// Simulate API that sometimes fails (implement this)
function fetchUserUnreliable(id: number): Promise<User> {
  // Your code: 50% chance to succeed, 50% chance to reject
}

// Test your functions
async function testUserAPI() {
```

```
    console.log("User:", user);

    // Test 2: Fetch multiple users
    const users = await fetchMultipleUsers([1, 2, 3]);
    console.log("Users:", users);

    // Test 3: Handle failures
    const unreliableUser = await fetchUserUnreliable(1);
    console.log("Unreliable user:", unreliableUser);
  } catch (error) {
    console.error("Error:", error);
  }
}

testUserAPI();
```

Exercise 2: async/await Chain

Task: Build a Data Processing Pipeline

```
interface Product {
  id: number;
  name: string;
  price: number;
  category: string;
}

// Mock functions to implement
async function fetchProducts(): Promise<Product[]> {
  // Simulate fetching products from API
  // Return array of 10 products after 1 second delay
}

async function filterExpensiveProducts(products: Product[]): Promise<Produ
  // Filter products with price > 50
  // Add 500ms delay to simulate processing
}

async function groupByCategory(products: Product[]): Promise<Map<string, P
  // Group products by category
  // Add 300ms delay to simulate processing
}
```

```
  // Calculate total price for each category
  // Add 200ms delay to simulate processing
}

// Main pipeline function
async function productPipeline() {
  try {
    console.time('Pipeline');

    // Implement the pipeline
    // 1. Fetch products
    // 2. Filter expensive products
    // 3. Group by category
    // 4. Calculate totals

    console.timeEnd('Pipeline');
    console.log('Results:', /* your results */);
  } catch (error) {
    console.error('Pipeline error:', error);
  }
}

productPipeline();
```

Exercise 3: File Operations

Task: Build a Log File Manager

```
 import fs from 'fs/promises';
import path from 'path';

class LogManager {
  private logDir: string;
  private logFile: string;

  constructor(logDir: string = './logs') {
    this.logDir = logDir;
    this.logFile = path.join(logDir, 'app.log');
  }

  async initialize(): Promise<void> {
    // Create log directory if it doesn't exist
    // Your code here
```

```
    // Format: [TIMESTAMP] [LEVEL] Message
    // Example: [2024-01-15T10:30:00.000Z] [INFO] Application started
    // Append to log file
    // Your code here
  }

  async readLogs(): Promise<string[]> {
    // Read all log entries as an array
    // Each line is one entry
    // Your code here
  }

  async filterLogs(level: 'INFO' | 'WARN' | 'ERROR'): Promise<string[]> {
    // Return only logs of specified level
    // Your code here
  }

  async clearLogs(): Promise<void> {
    // Delete log file
    // Your code here
  }

  async archiveLogs(): Promise<void> {
    // Rename current log file to app_TIMESTAMP.log
    // Create new empty log file
    // Your code here
  }
}

// Test the LogManager
async function testLogManager() {
  const logger = new LogManager();

  await logger.initialize();
  await logger.log('INFO', 'Application started');
  await logger.log('WARN', 'High memory usage detected');
  await logger.log('ERROR', 'Database connection failed');

  const allLogs = await logger.readLogs();
  console.log('All logs:', allLogs);

  const errorLogs = await logger.filterLogs('ERROR');
  console.log('Error logs:', errorLogs);

  await logger.archiveLogs();
```

Exercise 4: Environment Configuration

Task: Create a Configuration Manager

```
import dotenv from 'dotenv';

interface AppConfig {
  environment: 'development' | 'staging' | 'production';
  server: {
    host: string;
    port: number;
    apiUrl: string;
  };
  database: {
    host: string;
    port: number;
    name: string;
    username: string;
    password: string;
  };
  features: {
    enableLogging: boolean;
    enableDebug: boolean;
    maxRetries: number;
  };
}

class ConfigManager {
  private config: AppConfig | null = null;

  load(environment?: string): void {
    // Load appropriate .env file based on environment
    // Parse all environment variables
    // Validate required fields
    // Your code here
  }

  get(): AppConfig {
    if (!this.config) {
      throw new Error('Config not loaded');
    }
    return this.config;
```

```
    // Validate all required fields are present
    // Return true if valid, throw error if invalid
    // Your code here
  }

  print(): void {
    // Print configuration (hide sensitive data like passwords)
    // Your code here
  }
}

// Create .env.development file with these variables:
// ENVIRONMENT=development
// SERVER_HOST=localhost
// SERVER_PORT=3000
// API_URL=http://localhost:3000/api
// DB_HOST=localhost
// DB_PORT=5432
// DB_NAME=testdb
// DB_USERNAME=admin
// DB_PASSWORD=secret
// ENABLE_LOGGING=true
// ENABLE_DEBUG=true
// MAX_RETRIES=3

// Test the ConfigManager
const config = new ConfigManager();
config.load('development');
config.print();

const appConfig = config.get();
console.log('Server port:', appConfig.server.port);
```

Exercise 5: Async CLI Tool

Task: Build an Async Command-Line Tool

```
 import fs from 'fs/promises';
import readline from 'readline';

class AsyncCLI {
  private rl: readline.Interface;
```

```
    output: process.stdout
  });
}

async ask(question: string): Promise<string> {
  return new Promise(resolve => {
    this.rl.question(question, resolve);
  });
}

async run(): Promise<void> {
  console.log('=== Async File Manager ===');
  console.log('Commands: read, write, list, exit');

  while (true) {
    const command = await this.ask('\n> ');

    switch (command.trim().toLowerCase()) {
      case 'read':
        await this.handleRead();
        break;
      case 'write':
        await this.handleWrite();
        break;
      case 'list':
        await this.handleList();
        break;
      case 'exit':
        console.log('Goodbye!');
        this.rl.close();
        return;
      default:
        console.log('Unknown command');
    }
  }
}

private async handleRead(): Promise<void> {
  // Ask for filename
  // Read file content
  // Display content
  // Your code here
}

private async handleWrite(): Promise<void> {
```

```
    // Confirm success
    // Your code here
  }

  private async handleList(): Promise<void> {
    // Ask for directory path
    // List all files
    // Display with file sizes
    // Your code here
  }
}

// Run the CLI tool
const cli = new AsyncCLI();
cli.run().catch(console.error);
```

Summary

In Day 2, you learned:

âœ… **Asynchronous Programming**

- Synchronous vs asynchronous execution
- Event loop and non-blocking I/O
- When to use async programming

âœ… **Callbacks**

- What callbacks are
- Callback hell problem
- Why callbacks are difficult to maintain

âœ… **Promises**

- Promise states (pending, fulfilled, rejected)
- Creating and consuming promises

âœ… **async/await**

- Modern asynchronous syntax
- Sequential vs parallel execution
- Cleaner than promise chains
- Error handling with try-catch

âœ… **Error Handling**

- Try-catch with async/await
- Custom error classes
- Retry logic
- Timeout patterns

âœ… **Node.js Fundamentals**

- What is Node.js
- Global objects and built-in modules
- Process management
- Command-line arguments

âœ… **Modules**

- CommonJS (require/exports)
- ES6 modules (import/export)
- Module patterns for test automation

âœ… **File System**

- Reading and writing files
- Directory operations
- JSON file handling
- Async file operations

- Dependency management
- npm scripts
- package.json configuration

âœ… **Environment Variables**

- .env files
- dotenv package
- Multiple environments
- Security best practices

Next Steps

- Practice async/await extensively
- Build async utilities for test automation
- Complete all exercises
- Create a small Node.js CLI tool
- Experiment with different npm packages

Key Takeaways for Test Automation

- Use async/await for all async operations
- Handle errors properly with try-catch
- Use Promise.all for parallel operations
- Store configuration in environment variables
- Use fs/promises for file operations
- Leverage npm scripts for test execution

End of Day 2 Documentation