DIE NODE.JS CORE APIS

HALLO, ICH BIN DER BASTI 👋

- Sebastian Springer
- React & Node.js
- München

AGENDA

- FS-API
- Module, Pakete & Co.
- Events
- CLI
- Crypto

NODE ARCHITEKTUR UND MODULSYSTEM

MODULARER AUFBAU

- V8 & C++ Libraries
- Node Bindings
- Core Modules
- externe Pakete (z.B. NPM)
- eigene Applikation

CORE MODULES

- Zugriff auf Systemschnittstellen
- Stabilitätsindex

NODE.JS UND TYPESCRIPT

NODE.JS UND TYPESCRIPT

- TS entwickeln, kompilieren und ausführen
- Externe Bibliotheken: z.B. tsx mit `node --import=tsx index.ts`
- Strip Type: automatisch Typen entfernen

FILESYSTEM API

FILESYSTEM API

- Dateien lesen und schreiben
- Verzeichnisse verwalten (Erstellen, Lesen, Löschen)
- Datei- und Verzeichnisinformationen (Status abfragen, Existenz prüfen)
- Datei- und Verzeichnisoperationen (Umbenennen, Verschieben, Kopieren, Löschen)
- Streams (Lesen, Schreiben)
- Pfadverwaltung (Symbolische Links)

SYNC, ASYNC ODER PROMISES?

- Sync: für Skripte, Config loading beim Start oder einfache Automatisierung. Blockieren die Ausführung.
- Async mit Callback: Nicht blockierend, können aber schwer lesbaren Code erzeugen (Callback Hell)
- Promises: Beste Wahl für moderne Applikationen. Sauberer Code mit async/await und Errorhandling
- Performance: ähnliche I/O-Performance. Synchron hat etwas weniger overhead, aber blockiert.
- Empfehlung: durchgängig Promises nutzen, Node setzt immer mehr auf Promises

OPTIMIERUNGEN

- Streams für große Dateien
- High Water Mark bei Streams
- Chunking bei Schreibvorgängen
- Leseoperationen cachen

STREAMING FÜR GROSSE DATEIEN

GEWÖHNLICHES LESEN (READFILE)

- lädt die gesamte Datei
- Hoher Speicherverbrauch
- Verarbeitung startet erst, wenn die gesamte Datei gelesen ist
- Gefahr von Speicherüberlauf
- Einfacher Code

kleine bis mittelgroße Dateien, wenn einfacher Code wichtiger ist als Speicherverbrauch

STREAMING FÜR GROSSE DATEIEN

STREAMING

- lädt kleiner chunks (standard 64KB)
- Geringer Speicherverbrauch
- Verarbeitung beginnt mit dem ersten Chunk
- Gut für große Dateien oder begrenzten Speicher
- Komplexerer Code

große Dateien, begrenzter Speicher, schnelle Verarbeitung

HIGHWATERMARK BEI STREAMS

Größe des internen Puffers für Streams

- Readable Streams: wenn highwaterMark erreicht ist, wird das Lesen pausiert, bis alle Daten aus dem Puffer konsumiert wurden
- Writable Streams: wenn highwaterMark erreicht ist, gibt write false zurück, um zusignalisieren dass nicht mehr geschrieben werden soll
- zu kleine HighWaterMark: häufige Pausierung und Wiederaufnahme
- zu große HighWaterMark: erhöhter Speicherverbraucht

CHUNKING BEI SCHREIBVORGÄNGEN

- Weniger RAM-Verbrauch: Große Dateien werden in kleinere Chunks aufgeteilt.
- Bessere Performance: Verarbeitung beginnt mit den ersten Chunks.
- Backpressure-Handling: Kontrollierter Datenfluss bei Streaming.
- Effizientere Fehlerbehandlung: Nur betroffene Chunks müssen neu übertragen werden.
- Parallelisierung: Chunks können parallel verarbeitet werden.

PARALLELE VERARBEITUNG

PARALLELE VERARBEITUNG

- Node.js bietet asynchrone Methoden im fs-Modul, die parallele Dateizugriffe ohne Blockierung ermöglichen
- Bei großen Dateimengen kann die Methode Promise.all() verwendet werden, um mehrere Dateien gleichzeitig zu verarbeiten
- Mit Worker Threads ermöglichen echte parallele Berechnung zusätzlich zur I/O-Parallelität
- Streaming-Ansätze mit fs.createReadStream() und Pipe können die Verarbeitung großer Dateien optimieren
- Techniken wie Pool-Limitierung schützen vor Überlastung durch zu viele parallele Operationen
- Achtung bei der Verwendung der Promise-basierten Methoden und async/await - hier kann es zu sequenzieller Abarbeitung kommen

FS-EXTRA

FS-EXTRA

- Zusätzliche Methoden: Erweiterung des fs-Moduls um Methoden wie copy, move, remove, ensureDir und mehr
- Promise-Unterstützung: Alle Methoden unterstützen Promises (automatisch, wenn kein Callback übergeben wird)
- Fehlervermeidung (EMFILE): nutzt graceful-fs, um Fehler wie EMFILE (zu viele offene Dateien) zu verhindern.
- Abwärtskompatibilität: Es ist ein Drop-in-Ersatz für das native fs-Modul
- Einfachere API: Funktionen wie ensureDir (rekursives Erstellen von Verzeichnissen) und writeJson/readJson (Arbeiten mit JSON-Dateien) vereinfachen häufige Aufgaben erheblich.

CUSTOM MODULES UND PAKETMANAGEMENT

DAS MODULSYSTEM

- ECMAScript Module als neuer Standard
- Node Erkennt das Modulsystem automatisch
- CommonJS funktioniert nach wie vor

ARBEITEN MIT KERNMODULEN

- `node:` Präfix für Kernmodule (optional)
- Standardimport: `import fs from 'node:fs'`
- Named Imports: `import { writefile } from 'node:fs/promises'`
- Gruppierung der Import-Statements (Kernmodule, externe Pakete, eigene Dateien)

MODULSCHNITT

- Ein Modul, eine Verantwortet: Jedes Modul hat genau eine definierte Aufgabe.
- Konsistente Benennung: Einheitliche Namenskonventionen für Dateinamen und Strukturen.
- Hierarchische Verezichnisstruktur: Entweder technisch oder fachlich strukturieren.
- Übersichtliche Struktur: Maximal 10 Dateien pro Verzeichnis.
- Barrel Exports: Exporte mehrerer Submodule zusammenfassen.

WAS KANN ICH EXPORTIEREN?

- Funktionen: Wiederverwendbare Code-Blöcke
- Klassen: Objekt-Blueprints mit Eigenschaften und Methoden
- Objekte (und Arrays): Sammlungen von Schlüssel-Wert-Paaren
- Konstanten & Variablen: Werte und Datenstrukturen
- Primitive Datentypen: Strings, Numbers, Booleans etc.

WAS KANN ICH EXPORTIEREN?

- Default Export: Für die Hauptfunktionalität eines Moduls
- Named Exports: Für mehrere verwandte Funktionen/Objekte
- Konsistent bleiben: Entweder CommonJS oder ES Modules im Projekt verwenden
- Nur exportieren, was nach außen sichtbar sein soll

SEITENEFFEKTE IN MODULEN

- Top-Level-Code wird beim Import ausgeführt
- Einmalige Ausführung: wird einmalig beim ersten Import ausgeführt
- Seiteneffekte nutzen: Initialisierung (DB), Konfiguration (Frameworks), Registrierung (Events)
- Probleme: erschwertes Testen, Ausführungsreihenfolge, schwer nachvollziehbarer Code

SEITENEFFEKTE IN MODULEN

- Seiteneffekte minimieren
- Explizite Initialisierungsfunktionen exportieren statt automatischer Seiteneffekte
- Initialisierungscode klar kennzeichnen (z.B. mit Kommentaren)

ACHTUNG, SCHMUTZIGER HACK: URL-Parameter nutzen, um den Cache zu umgehen

EIGENE PAKETE

1. Projekt initialisieren (package.json, .gitignore)
2. package.json füllen (name, version, main, scripts, author, license
3. Code schreiben
4. Testen z.B. vitest
5. Dokumentation (README.md)
6. Veröffentlichen (npm login & npm publish)
7. Versionierung: Semantic Versioning (in der package.json)

EVENTS IN NODE

EVENTS IN NODE

- EventEmitter: Herzstück des Events-Moduls; alle Objekte, die Events auslösen können, erben von EventEmitter
- Wichtige Methoden: .on() (Event-Listener registrieren), .emit() (Event auslösen), .once() (einmaliges Ausführen), .removeListener() (Listener entfernen)
- Event-Priorität: Listener werden in der Reihenfolge ausgeführt, in der sie registriert wurden; mit .prependListener() kann ein Listener an den Anfang der Warteschlange gesetzt werden
- Error-Events: Besondere Behandlung des 'error'-Events; wenn kein Listener registriert ist, stürzt die Anwendung bei einem Error-Event ab
- MaxListeners: Standardmäßig sind 10 Listener pro Event erlaubt, kann mit .setMaxListeners() angepasst werden, um Memory Leaks zu vermeiden
- Zahlreiche Core-Module erben von EventEmitter - HTTP, fs, stream, net und andere setzen auf dem Events-Modul auf

EVENTEMITTER VS. ASYNCITERATOR

EVENTEMITTER VS. ASYNCITERATOR

- Funktionsprinzip: Wandelt Node.js EventEmitter-Events in einen asynchronen Iterator um, der mit for await...of durchlaufen werden kann
- Queue-Mechanismus: Speichert eingehende Events in einer Warteschlange, wenn kein ausstehender Promise existiert
- Promise-basierte Steuerung: Blockiert die Iteration, bis neue Events eintreffen, durch Verwendung von Promises und Resolvers
- Resource-Management: Der finally-Block stellt sicher, dass der Event-Listener korrekt entfernt wird, um Speicherlecks zu vermeiden
- Asynchrones Iteration: Ermöglicht die synchron wirkende Verarbeitung asynchroner Events mittels for await...of-Syntax

CLI APPLIKATIONEN

CLI APPLIKATIONEN

- CLI-Anwendungen mit Node.js für Automatisierung wiederkehrender Aufgaben
- Standardisierung von projektspezifischen Workflows
- Plattformübergreifende Basis für CLI-Tools, die auf Windows, macOS und Linux funktionieren
- Maßgeschneiderte Lösungen für spezifische Anforderungen erstellen, die in Standard-Tools nicht abgedeckt werden
- Modularer Aufbau: CLI-Tools schrittweise erweitern und an wachsende Anforderungen anpassen

CLI APPLIKATIONEN

1. Projekt initialisieren: `npm init -y`
2. Applikation erstellen
3. Applikation ausführbar machen: bin in package.json
4. Testen: `npm link` (entfernen: npm unlink -g)

BIBLIOTHEKEN

- commander, yargs: Option Parsing
- oclif: CLI Framework
- inquirer: Interaktive CLI
- chalk: Farbige Ausgabe
- ora: Ladeanimationen
- boxen: Boxen um Text
- figlet: ASCII Art
- blessed: Terminal UI

READLINE

- Das Readline-Modul ermöglicht zeilenweise Verarbeitung von Streams wie stdin oder Dateien.
- Einfache Schnittstelle für Kommandozeileneingaben und interaktive Programme ohne externe Bibliotheken.
- Seit Node.js v17 gibt es eine Promise-basierte Version (readline/promises).
- readline.question() fragt Benutzereingaben ab, verarbeitet mit Callbacks oder Promise.
- createInterface() initialisiert das Interface, wobei input- und output-Streams (typischerweise process.stdin und process.stdout) angegeben werden.
- Nach der Verwendung sollte das Interface mit close() geschlossen werden, um Ressourcen freizugeben.

DIE CRYPTO API

DIE CRYPTO API

- Die Web Crypto API in Node.js bietet eine browserkompatible Implementierung, erleichtert die Entwicklung von Anwendungen, die sowohl im Browser als auch auf dem Server laufen.
- Sie ersetzt ältere, unsichere Implementierungen und bietet standardisierte Methoden für kryptographische Operationen.
- Die API arbeitet nativ mit TypedArrays (z.B. Uint8Array) für effiziente Verarbeitung binärer Daten und optimierten Speicherverbrauch.
- Kryptographische Operationen werden als Promises implementiert, was asynchrone Verarbeitung ohne Callback-Hölle ermöglicht.
- Unterstützt moderne Algorithmen für Hashing (SHA-256, SHA-512), Verschlüsselung (AES-GCM, AES-CBC), Signierung (ECDSA, RSA-PSS) und Schlüsselgenerierung.
- Ermöglicht die Entwicklung von End-to-End-Verschlüsselungslösungen mit konsistentem Code für Client und Server.

ECHTE ZUFALLSZAHLEN

- crypto.getRandomValues(): kryptografisch sichere Methode für Zufallszahlen, deutlich sicherer als Math.random().
- Nutzt TypedArrays wie Uint8Array, Uint16Array oder Uint32Array, um Zufallszahlen in unterschiedlichen Größen und Wertebereichen zu erzeugen.
- Für Anwendungsfälle, die Zufallsbytes als Strings benötigen (z.B. für IDs oder Tokens), lassen sich die erzeugten Zufallswerte leicht in hexadezimale oder Base64-Strings konvertieren.
- crypto.randomUUID(): erzeugen von UUIDs (Version 4), die auf kryptografisch sicheren Zufallszahlen basieren.

DIGITALE SIGNATUREN

- Die Web Crypto API in Node.js bietet Funktionen für digitale Signaturen, um die Integrität und Authentizität von Daten zu gewährleisten.
- Unterstützt moderne Signaturalgorithmen wie ECDSA und RSA-PSS, ECDSA ist gut wegen hoher Sicherheit bei kleiner Schlüsselgröße.
- Ermöglicht das Generieren von Schlüsselpaaren, das Signieren von Daten und die Verifikation von Signaturen mit einem Promise-basierten Interface.
- Digitale Signaturen werden als TypedArrays (meist Uint8Array) verarbeitet, was eine effiziente Verarbeitung ermöglicht.
- Die browserkompatible Implementierung erlaubt denselben Code für Signaturoperationen im Browser und in Node.js zu verwenden.

VERSCHLÜSSELUNG

- Die Web Crypto API bietet moderne Verschlüsselungsmethoden in Node.js, unterstützt symmetrische Algorithmen wie AES-GCM und AES-CBC sowie asymmetrische Verfahren wie RSA-OAEP.
- AES-GCM ist der empfohlene symmetrische Modus, da er Vertraulichkeit, Authentizität und Integrität der Daten gewährleistet.
- Die API nutzt TypedArrays und Promises für effiziente Verarbeitung und asynchrone Programmierung.
- Umfassende Funktionen zur sicheren Schlüsselverwaltung: Generieren, Importieren, Exportieren und Speichern von Schlüsseln.
- Verwendung eines zufällig generierten Initialization Vector (IV) ist wichtig für sichere Verschlüsselung.
- Browserkompatible Implementierung ermöglicht End-to-End-Verschlüsselungslösungen mit demselben Code im Browser und auf dem Server.