



integrata
cegos

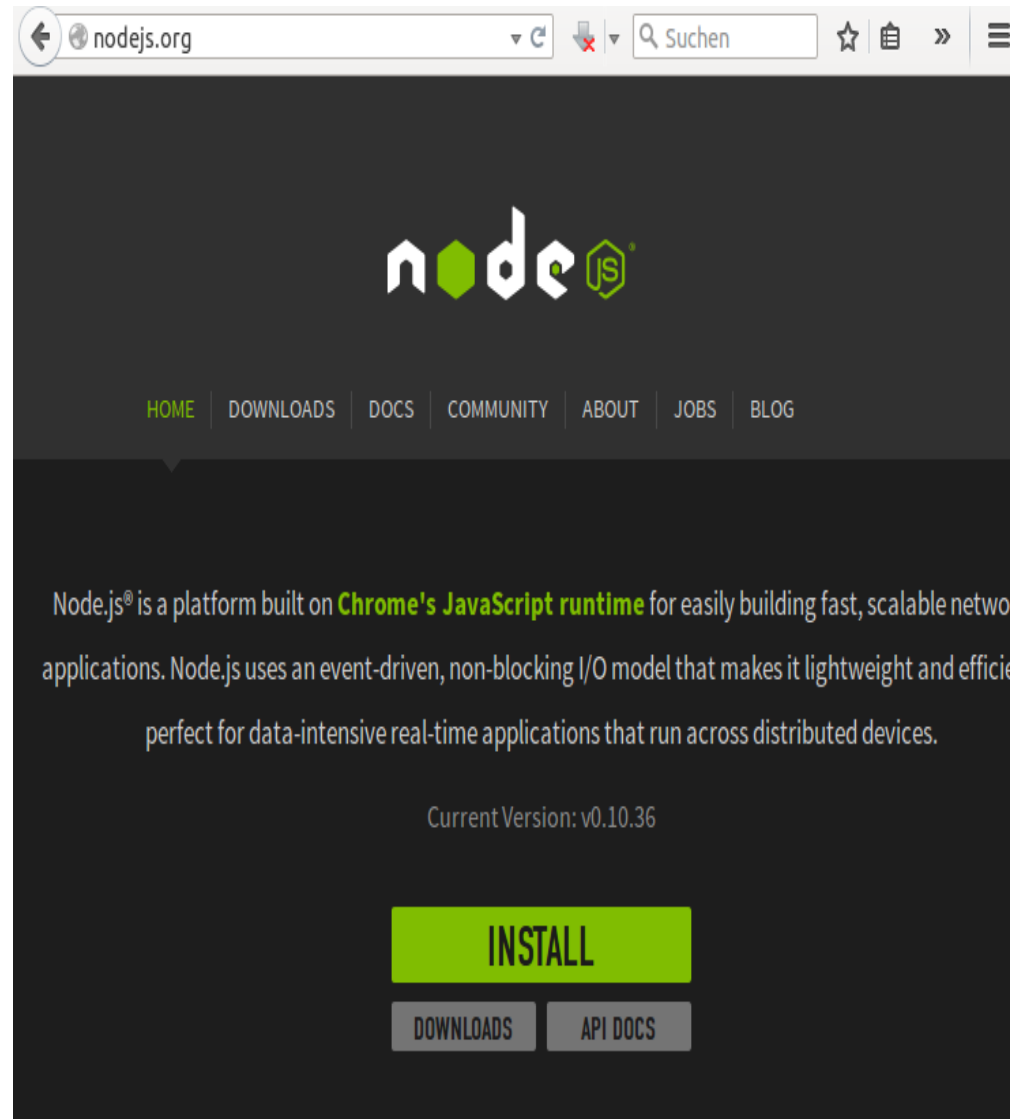
Node.js

Server-Implementierungen mit JavaScript

Node.js IN ACTION

Mike Cantelon
Marc Harter
T.J. Holowaychuk
Nathan Rajlich
FOREWORD BY Isaac Z. Schlueter

MANNING



- Dies ist ein Programmier-Seminar
 - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
 - Musterbeispiele werden zur Verfügung gestellt
 - Allgemeines Muster auf GitHub
 - <http://GitHub.com/Javacream/org.javacream.training.node>
- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
 - LPGL äquivalentes Lizenzmodell
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
- Konventionen
 - Befehle werden in `Courier-Schriftart` dargestellt
 - Dateinamen werden in *`Courier-Schriftart`* dargestellt
 - Links werden in `Courier-Schriftart` dargestellt

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

eMail: training@rainer-sawitzki.de

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.

Software-Entwicklung mit JavaScript	6
Das node-System	18
JavaScript Rekapitulation	46
Fortgeschrittene Programmierung	67
Node	90
Anwendungsprogrammierung	107
Datenzugriffe	118
Web Anwendungen	137

1

SOFTWARE-ENTWICKLUNG MIT JAVASCRIPT

1.1

AUSGANGSSITUATION UND PROBLEME

- Es gibt kein anerkanntes "JavaScript-Konsortium", das eine allgemeine Spezifikation definiert
- Es existieren viele Bibliotheken und Werkzeuge, die größtenteils von der Open Source-Community vertrieben werden
- Die Einsatzmöglichkeiten von JavaScript sind äußerst vielseitig
 - Im Browser
 - Auf dem Server
 - Als Abfragesprache für NoSQL-Datenbanken
 - Als Skript-Sprache für Produkte
 - In Embedded Systems

- JavaScript-Entwicklungsumgebungen bieten im Vergleich zu anderen Sprachen wie Java wenig Komfort
 - Code-Assists
 - Automatische Fehlererkennung
- Notwendig sind deshalb andere, kreative Ansätze
 - Linter
 - Testgetriebene Entwicklung
 - Benutzung von typisierten Sprachen, die JavaScript generieren

- Der Build-Prozess ist im JavaScript-Umfeld durch die Verbreitung auch relativ kleiner Frameworks und Produkte aufwändig
- Der Build-Prozess muss Abhängigkeiten
 - deklarieren
 - auflösen
 - laden und
 - zum Betrieb ausliefern
- Notwendig ist damit der Aufbau einer Build-Umgebung
 - Packaging Manager
 - Software-Repository
 - Dependency Management

- Das Testen ist durch die Verstrickung von JavaScript mit HTML, CSS und Browser nicht trivial
 - Unit-Tests, die ausschließlich JavaScript-Sequenzen testen, sind eher selten
 - Standalone JavaScript-Interpreter müssen hierfür benutzt werden
- Browser-Tests sind aufwändig und erschweren die Test-Automatisierung drastisch
 - Unterschiedliche Browser-Implementierungen der verschiedenen Hersteller müssen berücksichtigt werden
 - Tests müssen durch einen Tester mit einem UI-Recorder aufgezeichnet werden
 - Headless Browser ohne User Interface ermöglichen wenigstens eine rudimentäre Testautomatisierung

Alles nicht so einfach...

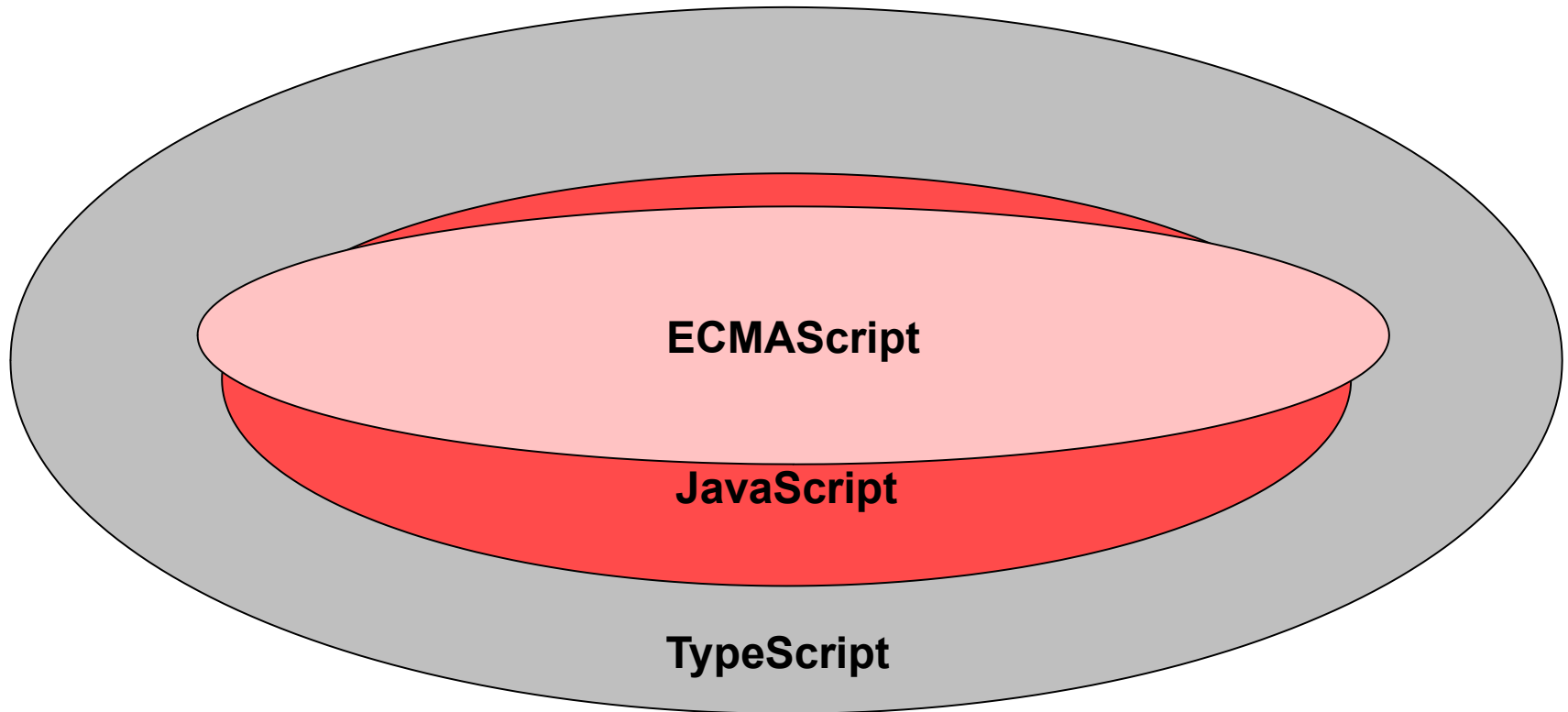


1.2

BEGRIFFE UND EINORDNUNG

- Alle Browser
- Google Chrome V8
- Node
 - basiert auf V8
- Java-Implementierungen
 - Rhino
 - Nashorn

- ECMAScript
 - Eine von der „European Computer Manufacturers Association“ spezifizierte Script-Sprache
 - Enthält elementare Syntax und Sprachkonstrukte
 - JavaScript ist ein Superset von ECMAScript
 - Vorsicht: Nicht alle JavaScript-Engines unterstützen den neuesten Stand von ECMAScript!
 - Siehe <http://en.wikipedia.org/wiki/ECMAScript>
- TypeScript
 - Eine von Microsoft entwickelte typisierte und Klassen-orientierte Programmiersprache
 - Ein Superset von JavaScript
- Andere Sprachen:
 - Coffescript
 - Go
 - ...



- Transpiler
 - Erzeugen aus Script-Sprachen andere Scripte
 - TypeScript wird nach JavaScript transpiliert
- Software-Repositories und –Registries
 - Enthalten Produkte, Bibliotheken, ...
 - Identifikation über einen eindeutigen Namen sowie eine Versionsnummer
 - Zugriff über Netzwerk, primär Internet
- Dependency Management
 - Jede Software enthält eine Deklaration der von ihr benötigten Abhängigkeiten
 - Transitive Dependencies treten auf, wenn eine Dependency selbst wiederum Dependencies deklariert
- Packaging Manager
 - Lokale Installation von Software aus einem Software-Repository
 - Auflösung aller notwendigen Dependencies
 - auch transitiv

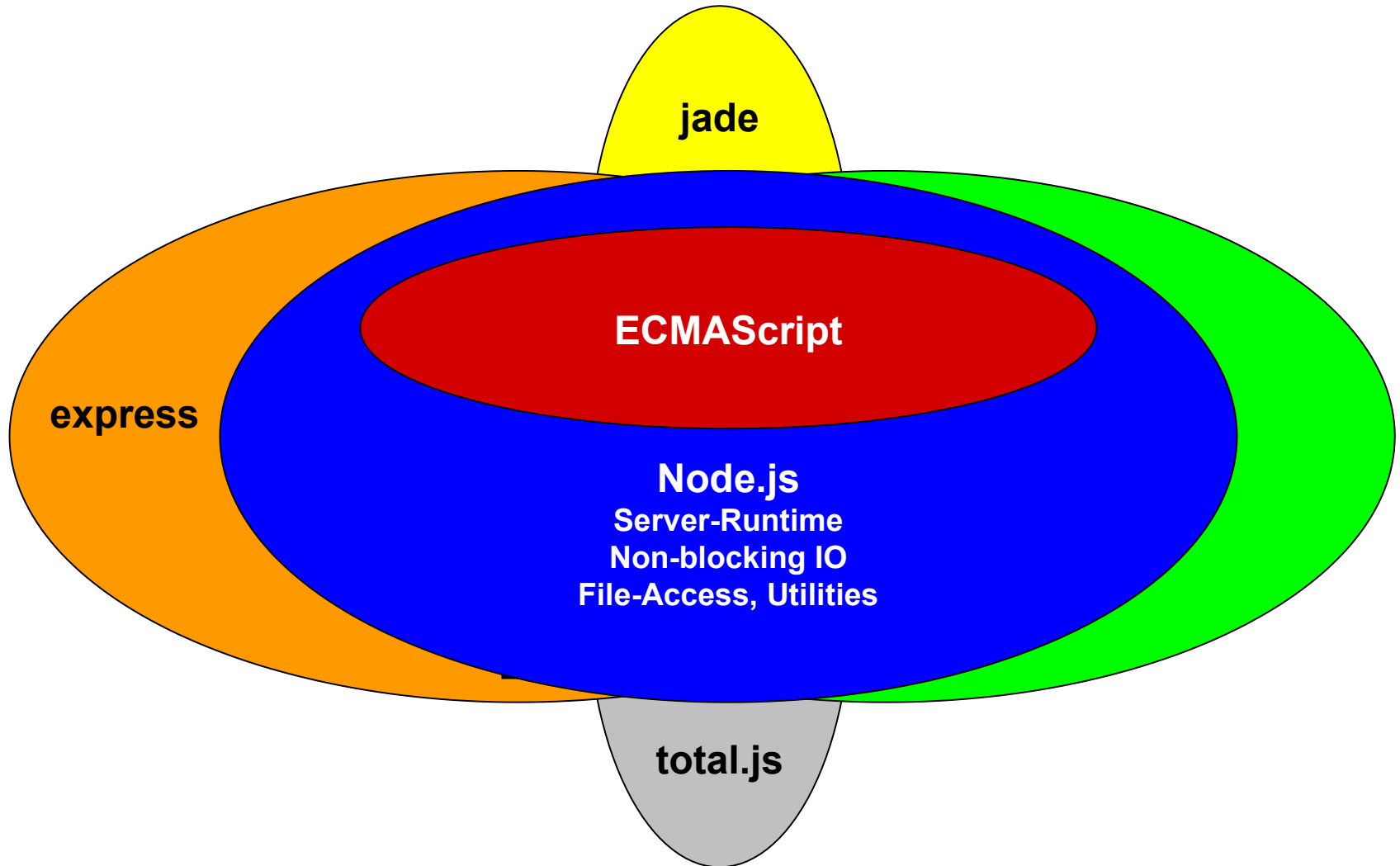
2

DAS NODE-SYSTEM

2.1

NODE.JS




- node.js ist ein Interpreter für Server-seitiges JavaScript
 - Auf Grundlagen der Google V8-Engine
- Mit node.js können damit keine Browser-Anwendungen betrieben werden
 - Keine UI, Keine User-Events
 - Kein Html-Dokument und damit kein DOM
 - Kein Browser-API
 - Window
 - Historie
 - ...
- Dafür stellt node.js eigene Bibliotheken zur Verfügung
 - Dateizugriff
 - Multithreading
 - Networking
 - ...
 - <https://nodejs.org/dist/latest-v8.x/docs/api/>



Beispiel: Ein kompletter http-Server

```
var http = require('http');
var fs = require('fs');
http.createServer(function handler(req, res) {
  var url = req.url;
  if (url.match(/.html/)) {
    res.writeHead(200, {
      'Content-Type' : 'text/html'
    });
  } else if ...
  var filename = "./static-content" + req.url;
  fs.createReadStream(filename).pipe(res);
}).listen(6061, '127.0.0.1');
```

Installation: node.js

LTS Recommended For Most Users	Current Latest Features	
 Windows Installer <small>node-v6.11.4-x86.msi</small>	 Macintosh Installer <small>node-v6.11.4.pkg</small>	 Source Code <small>node-v6.11.4.tar.gz</small>

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binaries (.tar.gz)

Linux Binaries (x86/x64)

Linux Binaries (ARM)

Source Code

32-bit		64-bit	
32-bit		64-bit	
64-bit			
64-bit			
32-bit		64-bit	
ARMv6	ARMv7		ARMv8
node-v6.11.4.tar.gz			

Additional Platforms

SunOS Binaries

Docker Image

Linux on Power Systems

Linux on System z

AIX on Power Systems

32-bit	64-bit
Official Node.js Docker Image	
64-bit le	64-bit be
64-bit	
64-bit	

- `node -v`
 - Ausgabe der Versionsnummer

- `node`
 - Starten der REPL zur Eingabe von JavaScript-Befehlen

- `node programm.js`
 - Ausführen der Skript-Datei *programm.js*

- Obwohl node.js nicht im Browser ausgeführt wird, wird es trotzdem gerne im Rahmen der Software-Entwicklung genutzt
- Hierzu wird node als Web Server eingesetzt, der die JavaScript-Dateien sowie die statischen Ressourcen (HTML, CSS, ...) zum Browser sendet
 - Mit Hilfe eines Browser-Sync-Frameworks triggern Änderungen von JavaScript-Dateien auf Server-Seite einen Browser-Refresh
 - <https://www.browsersync.io/>
 - Damit werden Änderungen ohne weitere Benutzer-Interaktion sofort angezeigt
 - Für eine agile Software-Entwicklung natürlich äußerst praktisch

2.2

NPM – DER NODE PACKAGE MANAGER

- Primär ein Packaging Manager
- `npm` ist Bestandteil der `node`-Installation
 - `npm -v`
- Die offizielle `npm` Registry liegt im Internet
 - <https://docs.npmjs.com/misc/registry>
 - Im Wesentlichen eine CouchDB
 - Laden der Software durch RESTful Aufrufe
 - Die `npm`-Registry ist aktuell die größte Sammlung von Software
- Unternehmens-interne oder private Registries können angemietet werden

- `npm` wird über die Kommandozeile angesprochen
 - eine grafische Oberfläche wird als separates Modul zur Verfügung gestellt
- **Hilfesystem**
 - `npm -h`
 - `npm <command> -h`
 - <https://docs.npmjs.com/>

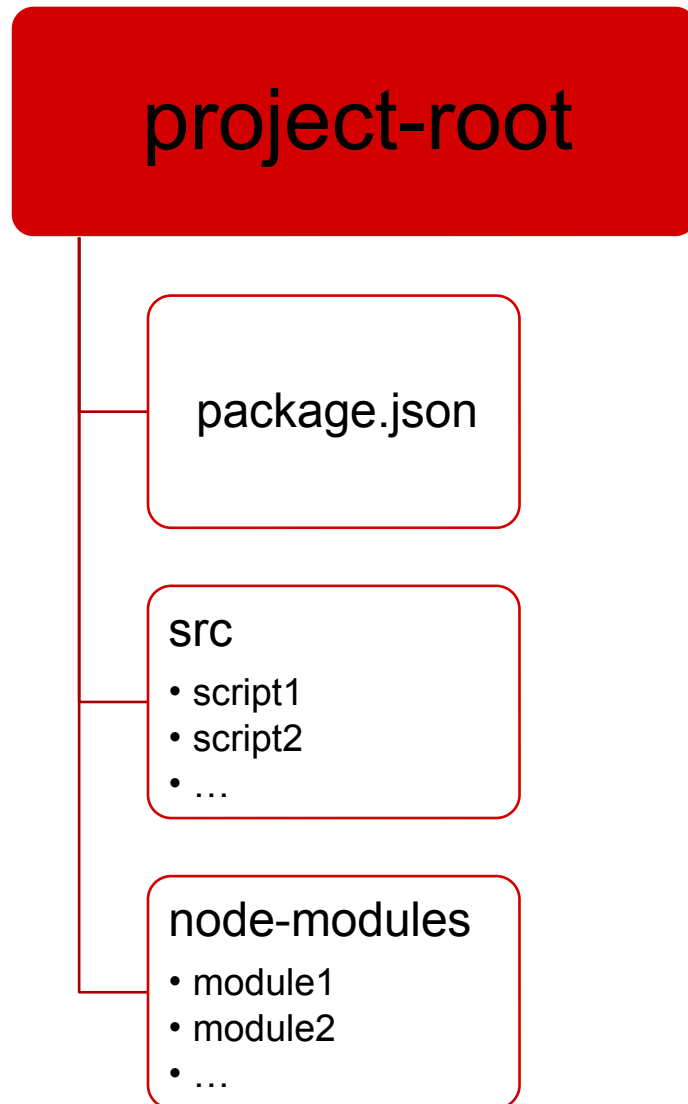
2.3

NODE-MODULES

- Jede via `npm` geladene Bibliothek wird als Node-Module konzipiert
- Jedes Modul besitzt
 - Eine Informationsdatei, die `package.json`, die das Projekt zusätzlich beschreibt
 - Abhängige Bibliotheken im Unterverzeichnis `node_modules`
 - Diese sind selbst ebenfalls Node-Module
 - Einen Entry-Point, in dem der Module-Entwickler das Fachobjekt seines Moduls erzeugt und exportiert
 - Dazu wird dem `module`-Objekt die Eigenschaft `exports` gesetzt
 - Zur Benutzung eines Moduls innerhalb eines Scripts dient der Node-Befehl `require`
 - Der Rückgabewert von `require` ist das vom Modul erzeugte und exportierte Fachobjekt

- Enthält die Projektinformation im JSON-Format
- Die Datei enthält
 - Den Projektnamen
 - Die aktuelle Versionsnummer
 - Meta-Informationen wie Autor, Schlüsselwörter, Lizenz
 - Dependencies
 - Ein `scripts`-Objekt mit ausführbaren Befehlen
 - Diese können mit `npm run <script>` ausgeführt werden

- Jedes `npm`-basierte Projekt ist ein neues Node-Module
- Initialisierung mit `npm init`
 - Dabei werden interaktiv die Informationen abgefragt, die zur Erstellung der initialen `package.json` benötigt werden



```
{  
  "name": "npm-sample",  
  "version": "1.0.0",  
  "description": "a simple training project",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [  
    "training"  
  ],  
  "author": "Javacream",  
  "license": "ISC"  
}
```

- **Datei *index.js***

```
module.exports = {  
  log: function() {  
    console.log('Hello')  
  }  
}
```

- **In der REPL**

```
var training = require('./index.js')  
training.log()
```

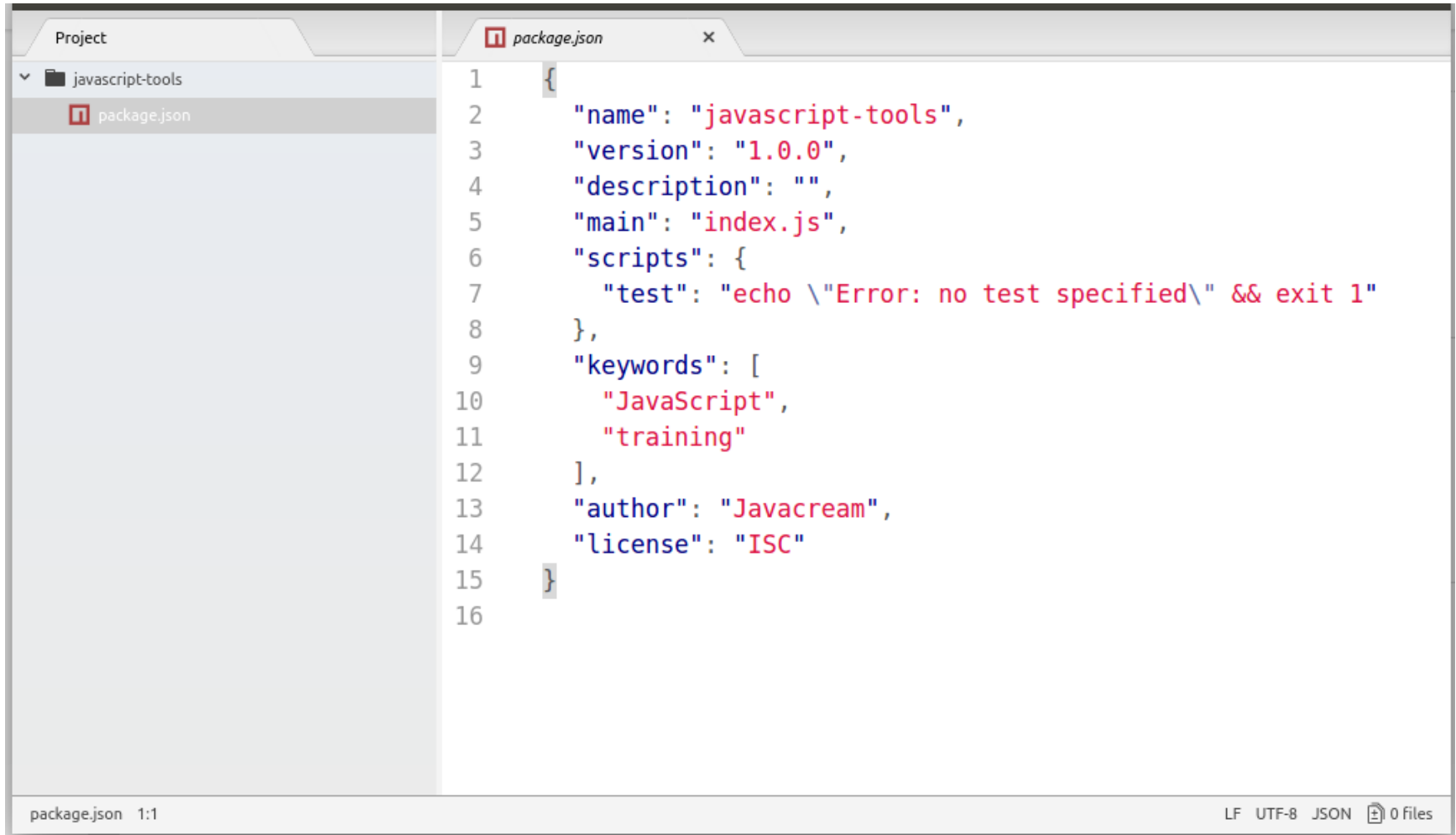
- Abhängigkeiten werden mit `npm install` von einer npm-Registry geladen
 - Ohne weitere Konfiguration wird dazu die Standard-Registry benutzt
 - Damit ist eine Internet-Verbindung notwendig
 - Es können aber auch Unternehmens-interne Repository-Server benutzt werden
 - z.B. Nexus
- Rechner-Registry
 - Die Abhängigkeiten werden auf dem Rechner abgelegt
 - Ab jetzt ist damit keine Internet-Verbindung mehr nötig
 - Orte:
 - lokale Ablage in einem Unterverzeichnis namens `node-modules`
 - Empfohlenes Standard-Verfahren zur Installation von Dependencies für eigene Software-Projekte
 - globale Ablage
 - Empfohlenes Standard-Verfahren zur Installation von allgemein verwendbaren Werkzeugen

2.4

EIN ERSTES PROJEKT

- Anlegen eines neuen Verzeichnisses
 - *javascript-tools*
- Initialisieren eines neuen Projekts mit `npm init`
- Öffnen des Projekt-Verzeichnisses in einem Editor
 - Atom oder ähnliches
- Hinweis:
 - Das Projekt ist natürlich noch komplett leer und damit sinnlos
 - Allerdings kann es bereits als vollständiges Node-Module betrachtet werden
 - Also beispielsweise in die npm-Registry hochgeladen werden

Ein einfaches Projekt



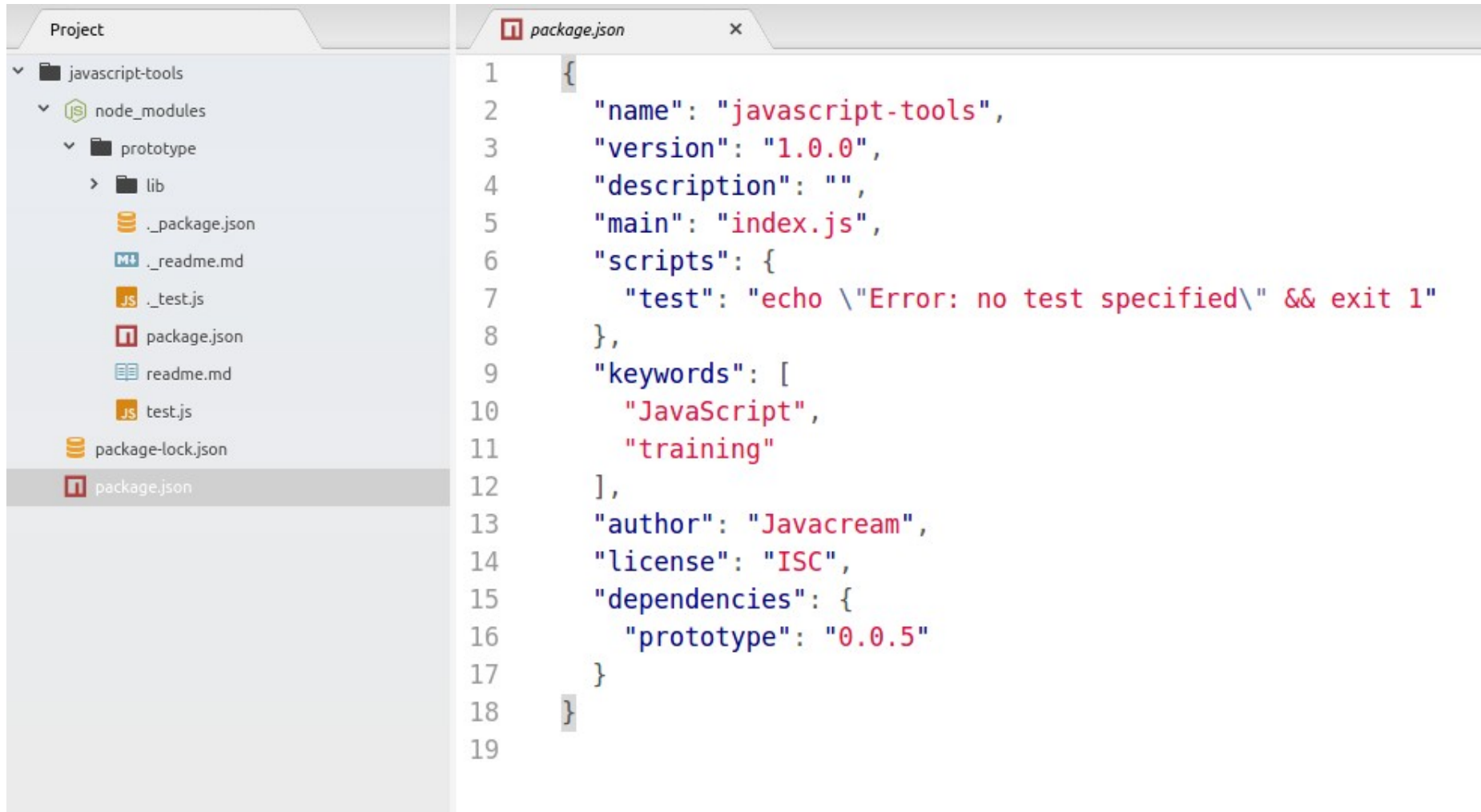
```
1  {
2    "name": "javascript-tools",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [
10      "JavaScript",
11      "training"
12    ],
13    "author": "Javacream",
14    "license": "ISC"
15  }
16
```

package.json 1:1

LF UTF-8 JSON 0 files

- Die prototype-Bibliothek stellt einige hübsche Objektorientierte Erweiterungen für JavaScript zur Verfügung
 - Immer noch recht weit verbreitet
 - aber seit ECMA2015 eigentlich obsolet
- `npm install prototype --save`

Das Projekt mit installiertem prototype



The screenshot shows a code editor with two panes. The left pane, titled 'Project', displays a file tree structure. The right pane, titled 'package.json', shows the content of the package.json file.

Project Structure:

- javascript-tools
 - node_modules
 - prototype
 - lib
 - ._package.json
 - ._readme.md
 - ._test.js
 - package.json
 - readme.md
 - test.js
 - package-lock.json
 - package.json

package.json Content:

```
1 {
2   "name": "javascript-tools",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "keywords": [
10    "JavaScript",
11    "training"
12  ],
13   "author": "Javacream",
14   "license": "ISC",
15   "dependencies": {
16     "prototype": "0.0.5"
17   }
18 }
19
```

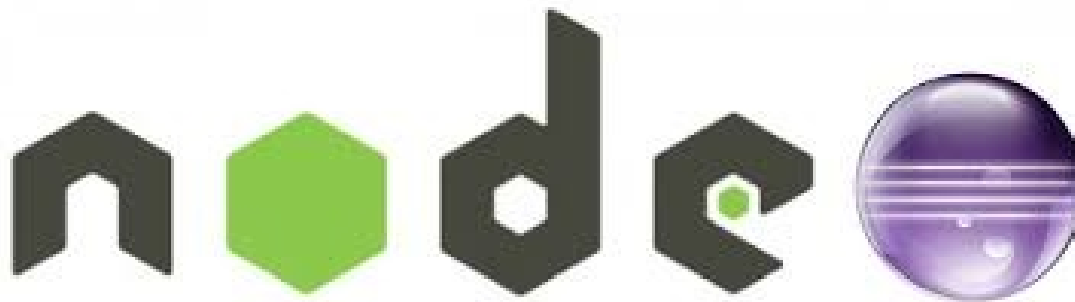
```
var prototype = require('prototype')
var Person = prototype.Class.create({
  initialize: function(name) {
    this.name = name;
  },
  greet: function() {
    return 'Hello ' + this.name + ' from prototype';
  }
});
var thommy = new Person('Thommy')
console.log(thommy.greet());
```

- Ausführung mit `node prototype_demo.js`
- Achtung: Dieses Beispiel läuft so nicht im Browser!
 - `require` ist ein `node`-Befehl und wird nicht im Browser unterstützt!

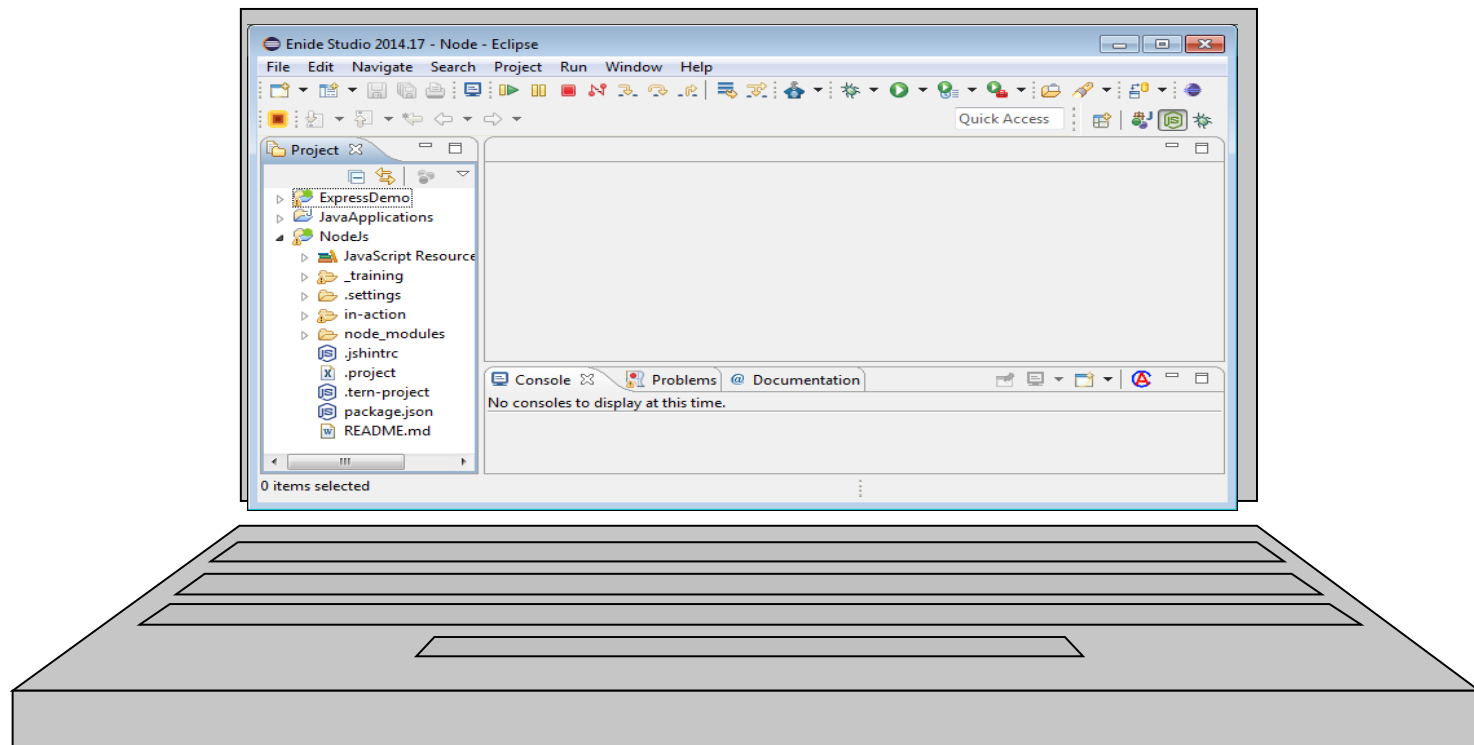
2.5

WERKZEUGE

- Node.js
 - Node-Runtime
 - Basierend auf Googles V8-Engine
 - npm
 - Node package manager
- Nodeclipse
 - Eclipse Node IDE
 - Download unter <http://www.nodeclipse.org/enide/>



- Starten der Enide mit `eclipse.exe`
- Das Projekt NodeJs ist bereits eingerichtet und geöffnet
 - Simple JavaScript-Beispiel
 - Ein erstes Node-Beispiel



3

JAVASCRIPT REKAPITULATION

3.1

SYNTAX

- Dieser Abschnitt soll knapp die wesentlichen Elemente von JavaScript erläutern
- Operatoren und Codebeispiele sind auch im Internet verfügbar
 - z.B. die w3school

- JavaScript hat Operatoren, Schleifen und Abfragen direkt aus C übernommen
 - Das hat auch Java so gemacht
 - Damit ist JavaScript hier näher an C als an Java!
- Die Objektorientierung in JavaScript funktioniert ganz anders als in Java
- Eine JavaScript-Engine ist jedoch Bestandteil einer Java-Standard-Installation
 - Technischer Hinweis: JavaScript wird nicht nach Bytecode kompiliert!

- Deklaration

`var meintest; bzw. var meintest = Wert;`

- Gültigkeit

- Innerhalb der deklarierenden Funktion

- Vorsicht: Im Detail den Closure-Effekt berücksichtigen!

- Global im gesamten Skript

- Vorsicht: Ein Weglassen der `var`-Anweisung ordnet die „Variable“ dem globalen Objekt zu
 - Dies ist in den allermeisten Fällen so nicht beabsichtigt!

- Referenzen

- Variablen sind stets Referenzen auf Objekte im Speicher

- Typisierung

- Variablen haben keinen Typ, Objekte schon

- `let` beschränkt den Gültigkeitsbereich einer Variable auf den deklarierenden Scope
 - Also beispielsweise einem Block einer Schleife
- `const` deklariert eine Konstante

```
■ if (Bedingung)
{
Anweisung1;
Anweisung2;
...
}
else
{
Anweisung7;
Anweisung8;
...
}
```

```
switch (Ausdruck) {  
  case FallA:{  
    Anweisung1; break;  
  }  
  case FallB:{  
    Anweisung2; break;  
  }  
  default:{  
    Anweisung4;  
  }  
}
```

- `while (Bedingung)`
 {
 Anweisung;
 Anweisung;
 }

- `do`
 {
 Anweisung;
 }
 while (Bedingung)

- `for(Initialisierung;Bedingung;Zähleranweisung){`
Code, der ausgeführt werden soll;
`}`
- `break`
 - Veranlasst das sofortige Verlassen der aktuellen Schleife
- `continue`
 - Springt zum nächsten Schleifendurchlauf

3.2

OBJEKTE

- Arrays sind Variablencontainer, die mehrere Variablen enthalten können
 - Auf einzelne Variable greift man über den Variablennamen und eine Nummer zu
- Als Literal-Kennzeichen für Arrays wird eine eckige Klammer benutzt, die Werte sind durch Kommas getrennt:

```
var theBeatles = ["John", "Paul", "George",  
"Ringo"];
```
- Der Index, über den auf ein Array-Element zugegriffen wird, steht in eckigen Klammern:

```
var john = theBeatles[0];
```
- Ein Array kann jederzeit mit weiteren Werten ergänzt werden bzw. die vorhandenen Einträge geändert werden
 - Der Zugriff erfolgt ebenfalls über den Index:

```
theBeatles[4] = "George Martin";
```

- **Simple Zuweisung:**

```
var theBeatles = {leadGuitar: "George", rhythmGuitar:  
"John", bass: "Paul", drums: "Ringo"};  
theBeatles.home = "Liverpool";
```

- **Als Eigenschaften können natürlich auch wieder komplexe Referenz-Typen benutzt werden, also zum Beispiel ein Array:**

```
theBeatles.albums = ["Please please me", /*usw*/, "Let  
it be"];
```

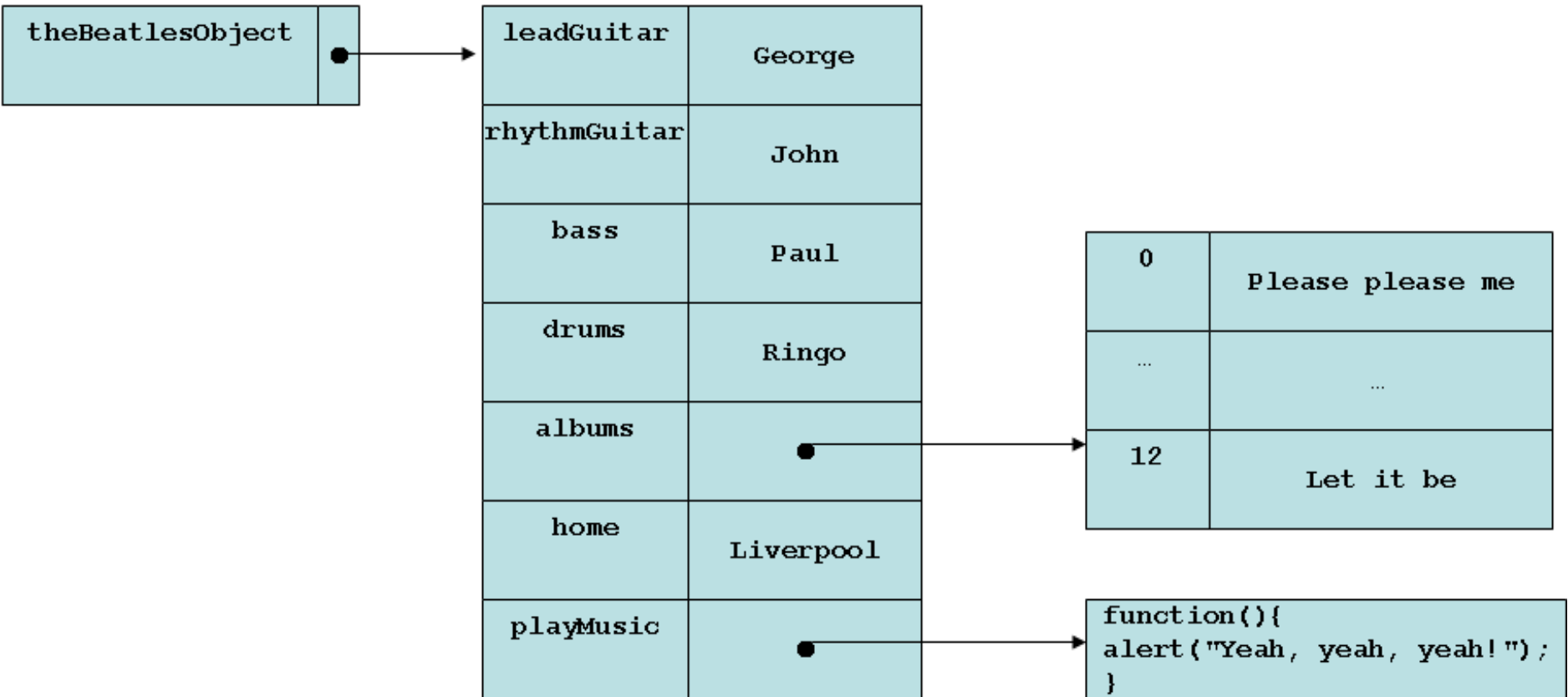
- **ein anderes Objekt:**

```
theBeatles.lifetime = {start: 1962, end: 1970};
```

- **oder eben eine Funktion:**

```
theBeatles.playMusic = function() {console.log("Yeah,  
yeah, yeah!");};
```

Objekte, Referenzen und Heap



- Jeder Aufruf von Programmlogik kann potenziell zu Fehlersituationen führen
- Die Auswertung einer Fehlersituation wird durch einen `try-catch`-Block erfolgen
 - Im `try-Block` steht die normale Anwendungs-Sequenz
 - Im `catch-Block` erfolgt die Fehlerbehandlung mit Zugriff auf ein Fehler-Objekt
 - Das Fehler-Objekt hat die Eigenschaften `message` und `name`, die eine genauere Analyse des Fehlers ermöglichen
- Mit `throw` kann ein beliebiges JavaScript-Objekt als Fehlerstruktur geworfen werden
 - Eine spezielle „Exception-Klasse“ ist nicht notwendig

```
function testExceptions() {  
    try {  
        throwException();  
    } catch (error) {  
        console.log("Caught error: message=" +  
error.message + ", name=" +  
error.name);  
    }  
    finally{  
        console.log("Always");  
    }  
}
```

3.3

FUNKTIONSOBJEKTE

- Eine Funktions-Definition der Form
 - `function myFunction() {...}`
- ist in Wahrheit völlig identisch zu:
 - `this.myFunction = function() {...}`
- Funktionen werden so dem globalen Objekt zugeordnet.
 - Einem Aufruf einer Funktion wird somit implizit die `window`-Referenz vorangestellt.
- Dieses Prinzip ist auch für Variablen gültig:
 - Ohne `var` wird implizit bei der Deklaration das globale Objekt vorangestellt
- Können Funktionen auch anderen Objekten zugeordnet werden?
 - Selbstverständlich
 - genau das passiert ja bei der Zuweisung einer Funktion an ein anderes Objekt im Rahmen der Objekt-orientierten Programmierung.

- Funktionen sind auch wiederum Objekte, die so wie alle anderen Objekte auch als Variable oder als Parameter benutzt werden können
- Zum Aufruf eines Funktions-Objekts existieren die beiden Methoden `apply` und `call`
 - Beide Funktionen bekommen als ersten Parameter die Referenz, die innerhalb der Methode als `this` benutzt werden soll
 - Die Funktionen unterscheiden sich nur in den weiteren Parametern
 - `apply` verlangt ein Array der Aufrufparameter
 - `call` benutzt eine beliebig lange Parameterliste

- Funktionen können auch innerhalb anderer Funktionen definiert werden
 - Dies haben wir auch bereits bei den Konstruktor-Funktionen gesehen
- Variablen der umhüllenden Funktion bleiben so lange gültig wie die innere Funktion
 - Dies nennt man allgemein „Closures“

- Ein potenziell gefährlicher Effekt von Closures darf nicht verschwiegen werden:
 - Durch die Verlängerung des Gültigkeitsbereiches kann die Garbage Collection die Objekte nicht mehr am Ende des Aufrufs sofort bereinigen
 - Insgesamt wird somit mehr Speicher benötigt
 - durch fehlerhafte Verwendung von Closures kann es sogar dazu führen, dass die JavaScript-Engine ihre Speicher-Ressourcen erschöpft
 - es kommt zu einem Speicherleck

4

FORTGESCHRITTENE PROGRAMMIERUNG

4.1

KLASSEN

- Die Konstruktor-Funktionen und der `new`-Operator sind in JavaScript notwendig, da es keine Klassen-Definitionen gibt
 - Eine Klasse ist ein abstraktes Template, aus dem Objekte erzeugt, besser: instanziiert werden
 - Jede Instanz einer Klasse hat damit einen durch die Klassen-Definition Satz von Eigenschaften
- Klassen sind in anderen Programmiersprachen wie Java und C# weit verbreitet
 - und sind bei Entwicklern sehr beliebt
- Workarounds sind möglich
 - Das "Module-Pattern" ist ein Beispiel hierfür
- Ab ECMAScript2015 werden Klassen eingeführt
 - Allerdings wird ES2015 noch bei weitem nicht von allen Browsern unterstützt
 - Zur Sicherheit: Transpilation!

```
class Book{
    constructor(isbn, title) {
        this.title = title;
        this.isbn = isbn;
    }
    get isbn() {
        return this.isbn;
    }
    get title() {
        return this.title;
    }
    set title(value) {
        this.title = value;
    }
    info() {
        return "Book: isbn=" + isbn + ", title=" + title;
    }
}
```

```
class SchoolBook extends Book{
    constructor(isbn, title, topic){
        super(isbn, title);
        this.topic = topic;
    }

    info(){
        return super.info + ", topic=" + topic;
    }
}
```

4.2

COLLECTIONS

- Eine Map besteht aus key-value-Paaren
 - In anderen Sprachen als Dictionary oder assoziatives Array bezeichnet

```
map = new Map(); //oder mit Vorbelegung
map = new Map(['key1', 'value1'], ['key2', 'value2']);
map.set('key', 'value');
map.get('key');
map.size;
map.clear();
```

- **Iteration**

```
for (let key of map.keys()) {}
for (let value of map.values()) {}
```

- Eine Set besteht aus Unikaten
 - In anderen Sprachen als Dictionary oder assoziatives Array bezeichnet

```
var set = new Set();  
set.add("Hugo")  
set.add("Emil")  
set.add("Hugo")  
set.has("Hugo")  
set.size; //-> 2
```

4.3

VEREINFACHTE FUNKTIONSDEKLARATION

- Eine vereinfachte Schreibweise für Funktions-Definitionen
 - beispielsweise für Parameter-Übergabe

```
(res) => console.log(res + " at " + new Date())
```

4.4

PROMISES

- Promises sind Objekte, die ein potenziell zukünftiges Ergebnis liefern
 - "Ein Versprechen auf die Zukunft"
 - Das Ergebnis kann auch eine Fehlerstruktur sein
- Promise-Objekte halten einen Zustand:
 - Fulfilled
 - Ein Ergebnis konnte bestimmt werden
 - Rejected
 - Es wurde ein Fehler festgestellt
 - Pending
 - noch nicht fertig ausgeführt
- Promises sind ein Sprach-unabhängiges Entwurfsmuster (Design Pattern)
 - damit eine Spezifikation
 - Erste Erwähnung als "Promises/A"
 - <http://wiki.commonjs.org/wiki/Promises/A>

- Promises werden im Programm so benutzt, als wäre das Ergebnis bereits bekannt
 - Dem Promise-Objekt werden
 - success
 - error
 - und optional progress-Funktionen zugefügt

- Das Promise-API ordnet verschachtelte Callback-Funktionen als eine Sequenz von Funktionsaufrufen
- Dazu bietet das Promise-API eine Funktion then, die
 - eine Callback-Funktion als Parameter erwartet und
 - ein weiteres Promise-Objekt zurück liefert
 - Damit können then-Aufrufe verschachtelt werden, was die Lesbarkeit des Codes deutlich erhöht


```
function asyncFn() {  
  return new Promise(function(resolve, reject) {  
    setTimeout(() => resolve(4), 2000);  
  });  
}  
  
asyncFn().then(  
  (res) => { res += 2; console.log(res + " at " +  
    new Date()); }  
).then((res) => console.log(res + " at " + new  
Date()))
```

- Mit `async` `await` wurden in ES6 zwei neue Schlüsselwörter eingeführt, die die asynchrone Programmierung nochmals deutlich vereinfachen
- `async` annotiert Funktionen so, dass die JavaScript-Engine diese Funktion in einem separaten Thread ausführt
- In dieser Funktion dürfen dann blockierende `await`-Kommandos benutzt werden
 - Mehrere sind zulässig
 - Damit definiert die `await`s die zu synchronisierenden Aufrufe
 - Eine `async`-Funktion darf ein `Promise`-Objekt als Rückgabewert haben

```
async function asyncFn1() {  
  return new Promise(function(resolve, reject) {  
    setTimeout(function() { resolve('data'); }, 300);  
  });  
}  
  
async function asyncFn2(input) {  
  return new Promise(function(resolve, reject) {  
    setTimeout(function() {  
      resolve('processing ' + input); }, 200);  
    });  
}
```

```
async function sequence() {  
  let data = await asyncFn1();  
  let completeData = await asyncFn1(data);  
  console.log('Result: ' + completeData);  
}
```

```
sequence();  
console.log('Finished');
```

4.5

EXKURS TYPESCRIPT

- TypeScript
 - `npm install typescript --save-dev`
- Initialisierung mit `tsc --init`
 - Erzeugt die Datei `tsconfig.json`
 - Darin werden alle möglichen Konfigurationen angelegt
 - wobei die allermeisten auskommentiert sind
- Der TypeScript-Compiler übersetzt den TypeScript-Code nach JavaScript
 - Damit ist dies im strengen Sinne eine Transpilation
 - Der Compiler kann im Hintergrund auf Änderungen der TypeScript-Dateien lauschen und den Übersetzungsvorgang automatisch durchführen

```
"scripts": {  
  "compile": "tsc --outDir ./dist -p .",  
  "compile-watch": "tsc -w --outDir ./dist -p ."  
}
```

- Wesentliches Element ist die Einführung eines Typ-Systems
 - Explizite Typisierung

```
let name : string
let state : boolean
```
 - Type Inference
 - Hier wird der Typ durch die Zuweisung eines Wertes definiert

```
let name = "Hello"
let state = true
```
- Diese Typisierung wird von manchen Entwicklern als vorteilhaft gewertet
 - Code-Assistent in Entwicklungsumgebungen
 - Hier ist insbesondere die hervorragende Umgebung Visual Studio Code zu erwähnen
 - Fehler und Warnungen bereits zur Compile-Zeit

- TypeScript kann selbstverständlich auch für Node-Entwicklung benutzt werden
 - Im Endeffekt entsteht ja wieder pures JavaScript
- Node-Bibliotheken werden im Typsystem von TypeScript unterstützt
 - Beispiel: `npm install --save @types/express`

5

NODE

5.1

ÜBERSICHT

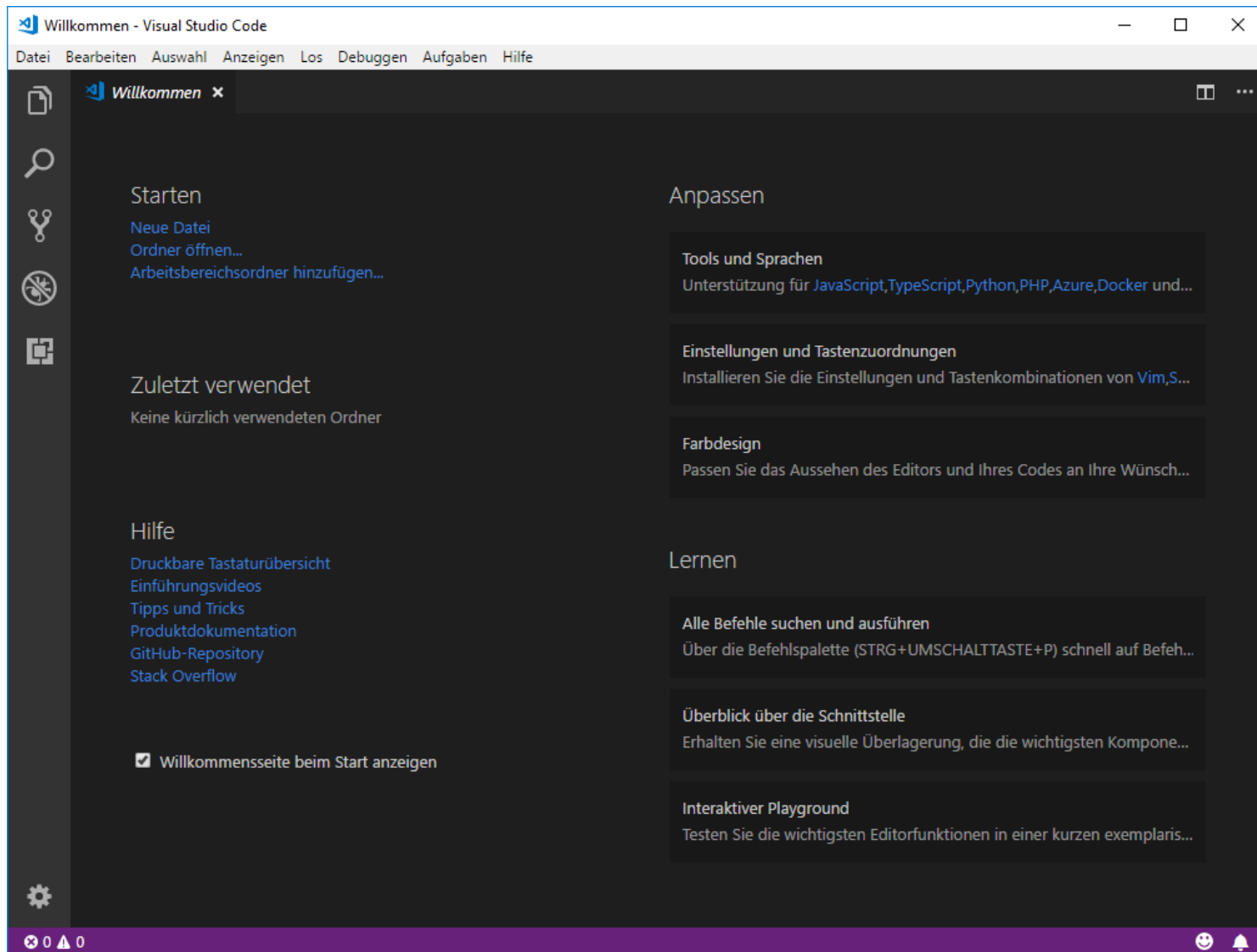
- Googles V8 JavaScript Engine
 - Auch benutzt von Google Chrome
 - Ausgefeilter Just In Time Compiler
 - Übersetzung in nativen Code
 - Optimierung und Tuning
 - Inkrementeller Garbage Collector



- JavaScript-Shell
 - Konsolen-basierte Ausführung von Node-Anweisungen
 - Autocomplete-Funktion

```
fs._stringToFlags      fs._toUnixTimestamp   fs.appendFile
fs.appendFileSync      fs.chmod               fs.chmodSync
fs.chown               fs.chownSync           fs.close
fs.closeSync           fs.createReadStream    fs.createWriteStream
fs.exists              fs.existsSync          fs.fchmod
fs.fchmodSync          fs.fchown              fs.fchownSync
fs.fdatasync           fs.fdatasyncSync       fs.fstat
fs.fstatSync           fs.fsync               fs.fsyncSync
fs.ftruncate           fs.ftruncateSync       fs.futimes
fs.futimesSync         fs.link                 fs.linkSync
fs.lstat               fs.lstatSync           fs.mkdir
fs.mkdirSync           fs.open                 fs.openSync
fs.read                fs.readFile             fs.readFileSync
fs.readSync            fs.readdir              fs.readdirSync
fs.readlink            fs.readlinkSync        fs.realpath
fs.realpathSync        fs.rename               fs.renameSync
fs.rmdir               fs.rmdirSync           fs.stat
fs.statSync            fs.symlink              fs.symlinkSync
fs.truncate            fs.truncateSync        fs.unlink
fs.unlinkSync          fs.unwatchFile          fs.utimes
fs.utimesSync          fs.watch                fs.watchFile
fs.write                fs.writeFile            fs.writeFileSync
fs.writeFileSync

> fs.
```



5.2

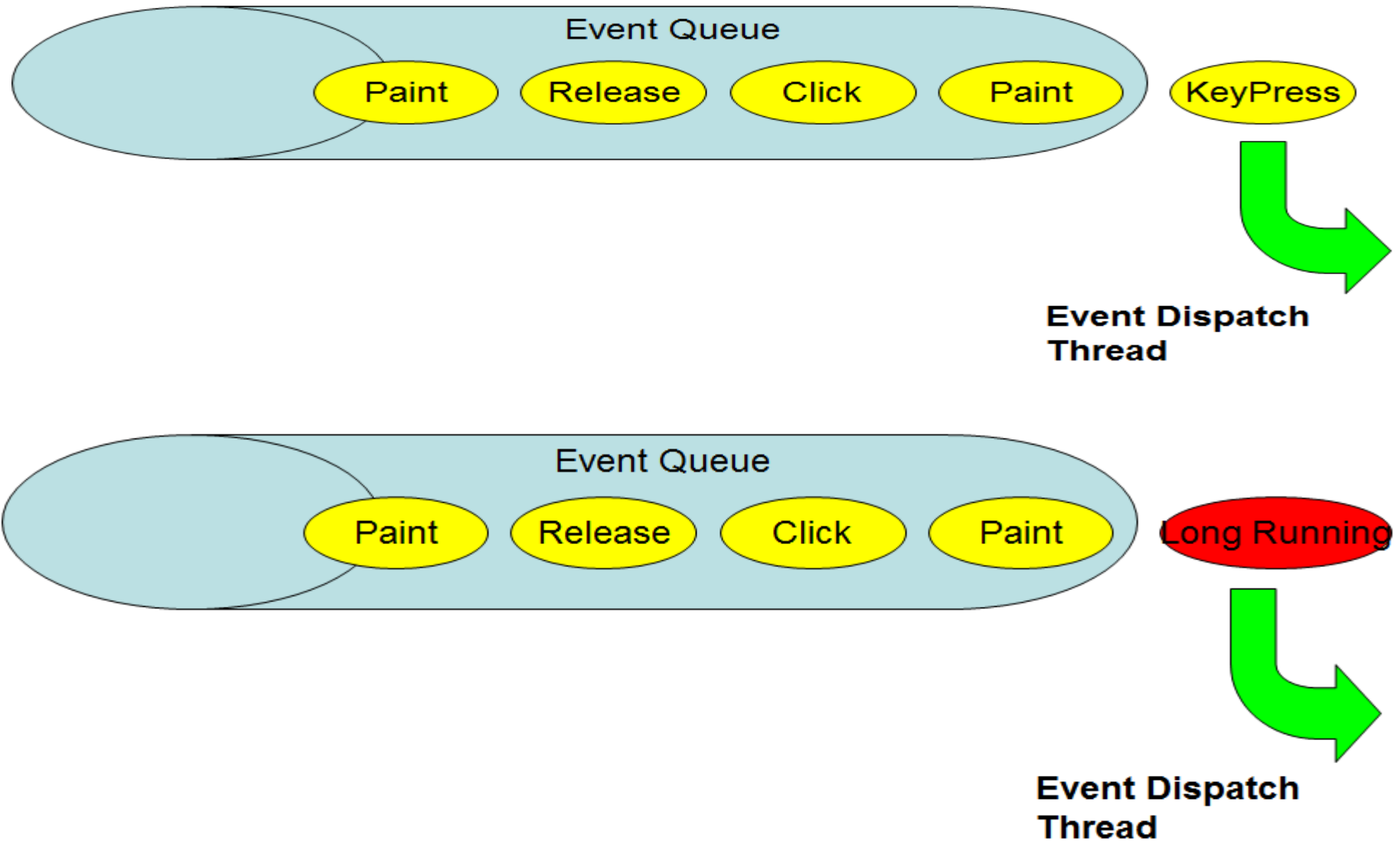
ARCHITEKTUR

- Eine geöffneter Server-Socket nimmt Requests entgegen
- Diese werden in einen Event umgewandelt und in einer Event-Queue abgelegt
 - Anschließend ist der Server-Socket sofort wieder frei
 - Requests können als immer angenommen werden
 - Limitierende Größe ist die Größe der Event-Queue
 - praktisch unbegrenzt
- Ein einzelner Thread arbeitet die Events sequenziell ab
 - Bei der Verarbeitung wird eine Callback-Funktion aufgerufen, die im Event hinterlegt ist
 - Durch die Verwendung eines einzelnen Threads werden konkurrierende Speicherzugriffe vermieden

- **Schnelle Ausführung**
 - **Keinesfalls blockierende/wartende Zustände!**
 - Sonst können keine weiteren Events verarbeitet werden
 - Der Server steht, obwohl die Event Queue weiter befüllt wird
 - Der Server ist „vergiftet“
- **Damit müssen sämtliche Ressourcen-Zugriffe ebenfalls durch ein Non-blocking API erfolgen**
 - Node stellt dafür geeignete Libraries zur Verfügung
 - File
 - Datenbank
 - Socket-Verbindungen zu anderen Systemen

- Ein traditioneller enthält einen Thread Pool
 - Beispiel: Ein Java-basierter Applikationsserver
 - Die maximale Größe des Thread Pools ist abhängig von der CPU-Leistung der Hardware
- Jedem Request wird ein freier Thread des Pools zugeordnet
 - Dieser übernimmt die Verarbeitung
 - Parallelisierung erfolgt durch Thread-Wechsel
 - Hierfür ist ein Rechen-intensiver Context-Switch notwendig

Im Vergleich: Der Browser arbeitet fast identisch!



- Non-blocking-Server sind hervorragend geeignet für
 - Streaming
 - http-Push (Comet)
 - Verarbeitung einer Vielzahl paralleler AJAX-Requests
- Vorsicht
 - Blockierende Vorgänge sind komplett zu vermeiden
 - Damit scheiden auch lang-dauernde komplexe Berechnungen aus
 - Durch das Single-Thread-Modell sind zwar keine *konkurrierende* Speicherzugriffe möglich, aber alle Callbacks arbeiten trotzdem auf einem *gemeinsam* genutzten Speicher
 - Damit muss für eine saubere Kapselung der Daten gesorgt werden

5.3

MODULE

Node.js

About these Docs

Usage & Example

Assertion Testing

Async Hooks

Buffer

C++ Addons

C/C++ Addons - N-API

Child Processes

Cluster

Command Line Options

Console

Crypto

Debugger

Deprecated APIs

DNS

Node.js v10.6.0 Documentation

[Index](#) | [View on single page](#) | [View as JSON](#) | [View another version ▼](#)

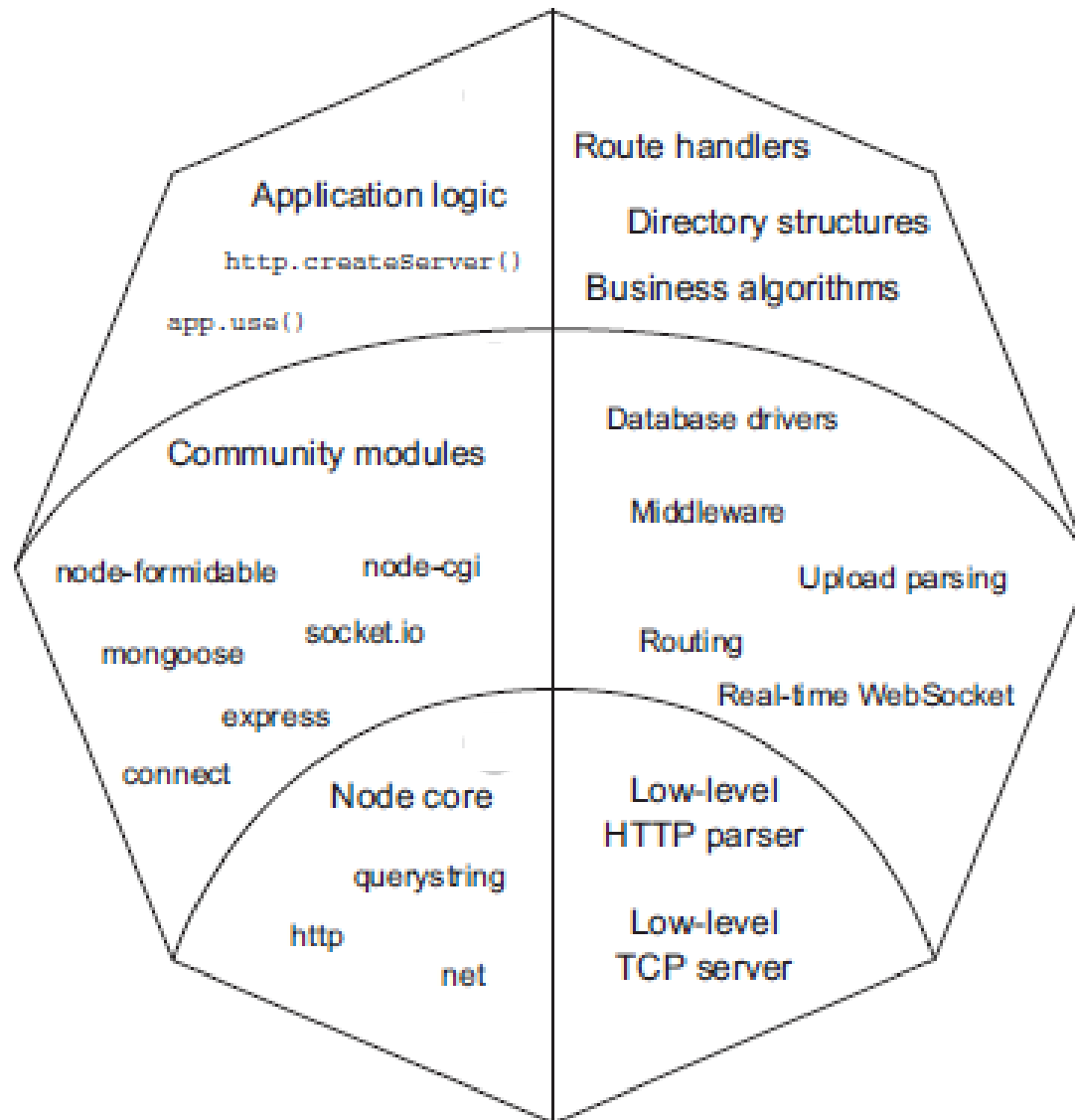
Table of Contents

- [About these Docs](#)
- [Usage & Example](#)
- [Assertion Testing](#)
- [Async Hooks](#)
- [Buffer](#)
- [C++ Addons](#)
- [C/C++ Addons - N-API](#)
- [Child Processes](#)
- [Cluster](#)
- [Command Line Options](#)
- [Console](#)
- [Crypto](#)
- [Debugger](#)
- [Deprecated APIs](#)

Modul	Beschreibung
Assert	Test-Assertions
Buffer	Native Speicherbereiche für Binärdaten
Child Processes	Aufruf von Betriebssystem-Kommandos
Cluster	Node auf Multi-Prozessor-Maschinen
Console	Synchrone und asynchrone Ausgabe
Debugger	V8-Debugger-API
DNS	Ansprechen von Domain-Servern
Events	Listener für interne Aktionen, z.B. Server-Connect
File System	Zugriff auf das Dateisystem
Globals	Globale Objekte
http/https	http-Server sowie Request- und Response-Klassen
Modules	System zu laden von Node-Modulen
Net	TCP-basierte Server und Clients
OS	Betriebssystem-nahe Befehle

Modul	Beschreibung
Path	Utilities zum Umgang mit Pfadangaben
Process	Das globale <code>process</code> -Objekt stellt eine Reihe von Funktionen zur Verfügung, die hier definiert sind
Query String	Utilities zur Erzeugung von Query-Strings
Readline	Öffnen einer Eingabe-Konsole
Streams	Lesende und Schreibende Datenströme
String Decoder	Decodierung und Encodierung von Strings in und nach Buffers
Timers	Scheduled Ausführung
TLS	Unterstützung von OpenSSL
UDP	Senden und Empfangen von UDP-Datagramme
URL	Parsen, Erzeugen und Auflösen von URLs
Utilities	Einige elementare Hilfsfunktionen
ZLib	ZIP-Verfahren

- Noch einige eher selten benutzte Bibliotheken
 - C/C++-Addons
 - Integration von C/C++-Bibliotheken
 - Domain
 - Gruppierung mehrere IO-Vorgänge
 - Punycode
 - Unterstützung des Punycode-Kodierungsverfahrens
 - REPL
 - Read-Evaluate-Print-Loop für Test-Skripte
 - TTY
 - Unterstützung des TTY-Protokolls
 - VM
 - Evaluierung und Ausführung von JavaScript



6

ANWENDUNGSPROGRAMMIERUNG

6.1

GRUNDLEGENDE ELEMENTE

- Die asynchrone Programmierung von Anwendungen benutzt Callback-Funktionen
 - und zwar exzessiv...
- **Beispiel: Lesen eines Verzeichnisses/einer Datei**

```
var fs = require('fs');
fs.readdir('.', function(er, data) {
  console.log(er);
  console.log(data);
});
console.log("continuing");
fs.readFile('./lib-fs.js', function(er, data) {
  console.log(er);
  console.log(data);
});
```

- Manchmal ist ein direktes Aufrufen einer Funktionalität nicht einfach oder eindeutig
 - Beispiel: Auslesen der Daten aus einem Socket
 - `var result = socket.read()`?
 - Das ist völlig falsch, da nicht non-blocking
 - `socket.read(readerFunction)`
 - Schon besser, aber wann und wie oft wird `readerFunction` aufgerufen?
 - Sobald eine Zeile komplett zur Verfügung steht oder
 - Sobald das erste Zeichen gelesen werden kann
- Events werden von einem `EventEmitter` in bestimmten Situationen geworfen
 - Die Identifikation des Events ist ein Typ, der über eine Zeichenkette definiert ist
 - Die Registrierung eines Listeners erfolgt in der `on`-Funktion
 - `socket.on('data', function(data) {...})`
 - Der `data`-Event wird gefeuert, sobald Daten komplett zur Verfügung stehen

- Zwei Möglichkeiten
 - Seriell
 - Realisierung durch verschachtelte Callbacks
 - Dies kann sehr schnell sehr unübersichtlich werden
 - Parallel
 - Realisierung durch Sequenzen von Callback- oder Listener-Registrierungen
- Framework-Lösungen vorhanden
 - Flow Control Framework, z.B. „nimble“
 - <http://caolan.github.io/nimble/>

6.2

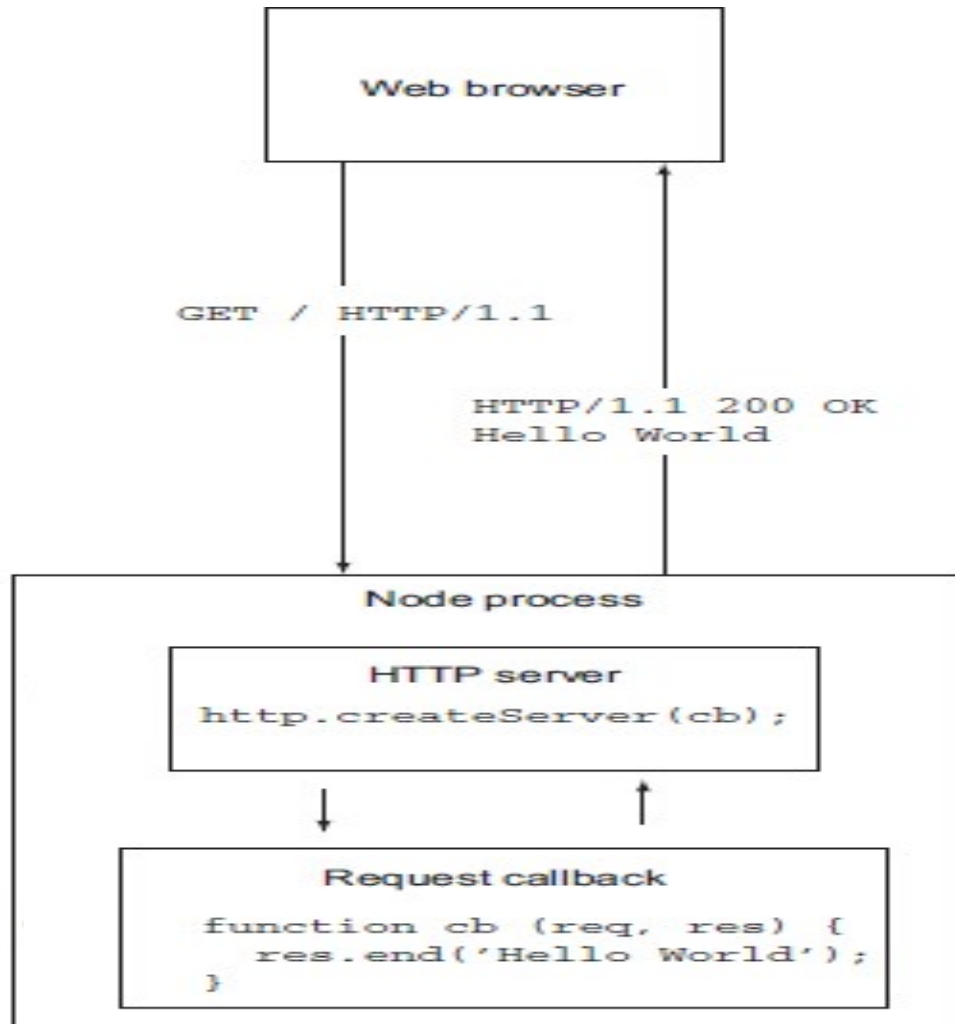
BEISPIELE FÜR NODE-SERVER

- Vereinfachung:

- Der Client schickt exakt eine Zeile

```
var net = require('net');
net.createServer(function(socket) {
  socket.on('data', function(data) {
    console.log("callback: " + data.toString());
    socket.write('Hello World!\r\n');
    socket.end();
  });
  socket.on('end', function() {
    console.log("ended socket");
  });
}).listen(1337);
console.log('listening on port 1337');
```

- Dafür dient das Modul `http`
- Der http-Server hat ein etwas anderes API als der TCP-Server
 - Die Callback-Funktion bekommt zwei Parameter
 - Request
 - Response
 - Request- und Response unterstützen das http-Protokoll
 - Allerdings eher rudimentär
 - Request und Response unterscheiden zwischen dem Header und dem Body
 - Es fehlt ein komfortables API um beispielsweise Query-Parameter auszulesen
 - Zusätzliche Bibliotheken füllen diese Lücke:
 - z.B. `express.js`



```
var http = require('http');
var fs = require('fs');
http.createServer(function handler(req, res) {
  var url = req.url;
  if (url.match(/.html/)) {
    res.writeHead(200, {
      'Content-Type' : 'text/html'
    });
  } else if ...
  var filename = "./static-content" + req.url;
  fs.createReadStream(filename).pipe(res);
}).listen(6061, '127.0.0.1');
```

- Eingabe- und Ausgabeströme können direkt durch eine Pipe miteinander verbunden werden

Beispiel: Ein File-Server

```
var net = require('net');  
var fs= require('fs');  
net.createServer(function(socket) {  
  socket.on('data',function(data) {  
    var filename = data.toString();  
    fs.createReadStream(filename).pipe(socket);  
  });  
}).listen(1338);
```

7

DATENZUGRIFFE

7.1

IN MEMORY DATEN-ABLAGE

- Das ausgeführte Node-Skript kann selbstverständlich Variablen deklarieren
 - Diese halten Informationen global
- Vorsicht mit asynchroner Programmierung!
 - Während einer Request-Verarbeitung hat der asynchrone Callback exklusiven Zugriff auf die Variable
 - Es gibt ja nur einen einzigen Thread...
 - Aber der Wert der Variable kann beim „nächsten“ Callback-Aufruf ein unbestimmter Wert sein
 - Der Wert, den der zufällig vorher aufgerufene Callback hinterlassen hat

- Sollen Daten für Requests eines Clients im Speicher des Servers gehalten werden wird eine Session aufgebaut
- Zur Identifikation des Clients muss irgendwie eine Client-ID benutzt werden
 - Für Web-Anwendungen wird dazu die „Session-ID“ benutzt
 - häufig in einem Cookie abgelegt
- Sessions werden von Node nicht direkt unterstützt
 - Zusätzliche Node-Frameworks füllen diese Lücke
 - z.B. das später behandelte Module `connect`

7.2

DATEISYSTEM

- Der Zugriff auf das Dateisystem erfolgt durch das `fs`-Modul
- Die darin definierten Methoden sind fast selbsterklärend
 - repräsentieren eine Unix/Linux-Sicht auf das Filesystem

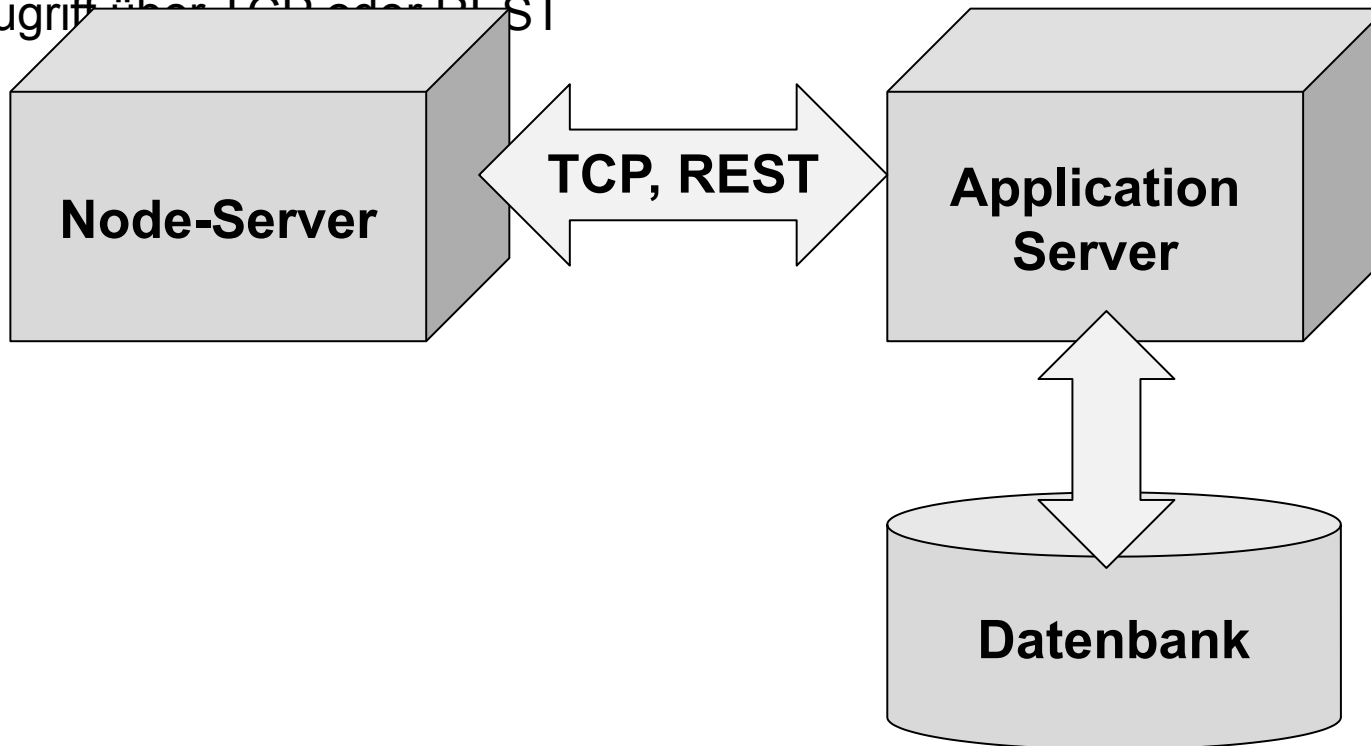
- `appendFile(filename, data)` - node
- `appendFile(filename, data, callback)` - node
- `appendFile(filename, data, encoding)` - node
- `appendFile(filename, data, encoding, callback)` - node
- `appendFileSync(filename, data)` - node
- `appendFileSync(filename, data, encoding)` - node
- `chmod(path, mode)` - node
- `chmod(path, mode, callback)` - node
- `chmodSync(path, mode)` - node
- `chown(path, uid, gid)` - node
- `chown(path, uid, gid, callback)` - node
- `chownSync(path, uid, gid)` - node
- `close(fd)` - node
- `close(fd, callback)` - node
- `closeSync(fd)` - node
- `createReadStream(path)` : stream.Readable - node
- `createReadStream(path, options)` : stream.Readable - node
- `createWriteStream(path)` : stream.Writable - node
- `createWriteStream(path, options)` : stream.Writable - node
- `exists(path)` - node
- `exists(path, callback)` - node
- `existsSync(path)` : bool - node
- `fchmod(fd, mode)` - node
- `fchmod(fd, mode, callback)` - node

- Innerhalb von Dateien können praktisch beliebige komplexe JavaScript-Objekte abgelegt werden
 - Häufig als JSON-Dokument
- Zur Organisation der Daten werden Verzeichnisse und Dateinamen benutzt
 - Der Pfad zur Datei ist der eindeutige „Primary Key“
 - Auch Sessions können im Dateisystem abgelegt werden
- Die Ausfallsicherheit und ein benötigter Cluster-Betrieb müssen durch Administration des Dateisystems gewährleistet werden
 - Dateisicherung
 - Shared Directories
 - Benutzung von Content Management Systemen
 - beispielsweise Alfresco

7.3

ANSPRECHEN EINES EXTERNEN SERVERS

- Mit Node können auch Clients programmiert werden
- Diese greifen über ein unterstütztes Protokoll auf einen externen Server zu
 - Dieser kann sich dann um die Datenhaltung kümmern
 - Zugriff über TCP oder REST



```
var net = require('net');  
var client = net.connect({port : 1337}, function() {  
    console.log('client connected');  
    client.write('world!\r\n');  
});  
client.on('data', function(data) {  
    console.log(data.toString());  
    client.end();  
});
```


7.4

RELATIONALE DATENBANKEN

- Die Node-Distribution enthält Treiber-Software für die gängigen relationalen Datenbank-Systeme
 - MySQL
 - Postgres
 - Oracle
- Nach Installation der Treiber können gegen die Datenbank SQL-Statements abgesetzt werden

```
var mysql = require('mysql');  
var db = mysql.createConnection({  
  host : '127.0.0.1',  
  user : 'root',  
  password : 'unilog',  
  database : 'test'  
});  
  
db.query(statement, function(err) {  
  ...  
})
```



integrata
cegos

Persistente Daten

NoSQL Datenbanken

- Key-Value-Stores speichern beliebige Daten ab
 - Values sind beispielsweise
 - Dokumente
 - JSON
 - XML
 - HTML
 - Images
 - Office-Dokumente
 - ...
 - Identifikation der Values nur über den Key möglich
 - Keine Joins zwischen Daten
- Beispiele
 - Redis
 - Riak

- Ablage der Daten als Dokumente
 - Meistens im JSON-Format
- Auch hier erfolgt die Identifikation eines Dokuments über eine ID
 - Allerdings können hier auch Abfragen auf Basis der Dokumenten-Struktur erfolgen
 - Joins werden über Verlinkungen realisiert
- Beispiele
 - MongoDB
 - CouchDB

- Key-Value-Stores bieten einen RESTful Zugriff
 - Anbindung von Node damit über einen simplen http-Client möglich
- Bei den Dokumenten-orientierten Datenbanken wird eher ein Treiber-Ansatz bevorzugt
 - Das REST-API ist durch die Unterstützung von Abfragen wesentlich komplexer
 - Für den Zugriff auf die MongoDB vereinfacht das `mongoose`-Module den Zugriff

```
var mongodb = require('mongodb');
var server = new mongodb.Server('127.0.0.1', 27017, {});
var client = new mongodb.Db('mydatabase', server, {w :
1});
client.open(function(err) {
    ...
})
client.collection('test_insert', function(err,
collection) {
    collection.insert({
        "title" : "I like cake",
        "body" : "It is quite good."
    })
    console.log('Document ID is: ' + documents[0]._id);
});
});
```


8

WEB ANWENDUNGEN

8.1

EXKURS: HTTP UND RESTFUL

Das http-Protokoll

- Eine umfassende Spezifikation des w3w-Konsortiums
 - Siehe <http://en.wikipedia.org/wiki/Http>

Hypertext Transfer Protocol

From Wikipedia, the free encyclopedia
(Redirected from [Http](#))

The **Hypertext Transfer Protocol (HTTP)** is an [application protocol](#) for distributed, collaborative, [hypermedia](#) information systems.^[1] HTTP is the foundation of data communication for the [World Wide Web](#).

[Hypertext](#) is structured text that uses logical links ([hyperlinks](#)) between [nodes](#) containing text. HTTP is the protocol to exchange or transfer hypertext.

The standards development of HTTP was coordinated by the [Internet Engineering Task Force](#) (IETF) and the [World Wide Web Consortium](#) (W3C), culminating in the publication of a series of [Requests for Comments](#) (RFCs), most notably [RFC 2616](#) [↗](#) (June 1999), which defines HTTP/1.1, the version of HTTP in common use.

Contents [\[hide\]](#)

- 1 Technical overview
- 2 History
- 3 HTTP session
- 4 Request methods
 - 4.1 Safe methods
 - 4.2 Idempotent methods and web applications
 - 4.3 Security
- 5 Status codes
- 6 Persistent connections
- 7 HTTP session state
- 8 Encrypted connections
- 9 Request message
- 10 Response message
- 11 Example session
 - 11.1 Client request

Internet protocol suite

Application layer

BGP • DHCP (DHCPv6) • DNS • FTP •
HTTP • IMAP • IRC • LDAP • MGCP •
 NNTP • NTP • POP • RPC • RTP • RTSP •
 RIP • SIP • SMTP • SNMP • SOCKS • SSH •
 Telnet • TLS/SSL • XMPP • *more...*

Transport layer

TCP • UDP • DCCP • SCTP • RSVP •
more...

Internet layer

IP (IPv4 • IPv6) • ICMP • ICMPv6 • ECN •
 IGMP • IPsec • *more...*

Link layer

ARP/InARP • NDP • OSPF • Tunnels (L2TP)
 • PPP • Media access control (Ethernet •
 DSL • ISDN • FDDI • DOCSIS) • *more...*

V • T • E

Elemente der http-Spezifikation

- Definition von URIs
 - Pfad
 - Parameter
- http-Request und http-Response
 - Daten-Container mit Header und Body
 - Encodierung
- Umfassender Satz von Header-Properties
 - Content-Length
 - Accepts
 - Content-Type

Elemente der http-Spezifikation II

- http-Methoden
 - PUT
 - GET
 - POST
 - DELETE
 - OPTIONS
 - HEAD
- Statuscodes für Aufrufe
 - 404: „Not found“
 - 204: „Created“
 - ...

- Definition der Datentypen des Internet
 - Nicht zu verwechseln mit einem XML-Schema
 - Ein MimeType ist „nur“ eine strukturierte Zeichenkette
 - Eigene Erweiterungen sind möglich

Der Representational State Transfer

- Doktorarbeit von Roy Fielding, 2000
- Siehe <http://en.wikipedia.org/wiki/REST>

Representational state transfer

From Wikipedia, the free encyclopedia
(Redirected from [REST](#))

"REST" redirects here. For other uses, see [Rest](#).

Representational state transfer (REST) is a software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed [hypermedia](#) system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.^{[1][2]}

The term *representational state transfer* was introduced and defined in 2000 by [Roy Fielding](#) in his doctoral dissertation at [UC Irvine](#).^{[1][3]} REST has been applied to describe desired web architecture, to identify existing problems, to compare alternative solutions, and to ensure that protocol extensions would not violate the core constraints that make the web successful. Fielding used REST to design [HTTP 1.1](#) and [Uniform Resource Identifiers \(URI\)](#).^{[4][5]}

The REST architectural style is also applied to the development of [web services](#)^[6] as an alternative to other distributed-computing specifications such as [SOAP](#).

Contents [\[hide\]](#)

- 1 History
- 2 Software architecture
 - 2.1 Components
 - 2.2 Connectors
 - 2.3 Data
- 3 Architectural properties
- 4 Architectural constraints
 - 4.1 Client-server
 - 4.2 Stateless
 - 4.3 Cacheable
 - 4.4 Layered system
 - 4.5 Code on demand (optional)
 - 4.6 Uniform interface

- Die Arbeitsweise des Internet wird abstrahiert
 - Was sind Ressourcen?
 - Wie werden Ressourcen identifiziert?
 - Was sind Ressourcen-Operationen?
 - Wie werden Services beschrieben?
 - Was sind Stateless Operationen?
 - Voraussetzungen und Umsetzung von Caching-Mechanismen
 - Optional: Übertragung von Skript-Logik auf den Client
- Grundlegendes Konzept sind die verlinkten HyperText-Dokumente
- Nicht überraschend: „Das Internet ist ein Beispiel für die Implementierung eines REST-basierten Systems“

- REST hat mit http prinzipiell nichts zu tun
 - REST ist eine abstrakte Architektur
 - http ist ein konkretes Kommunikationsprotokoll
- Aber
 - http passt als Kommunikations-Protokoll der „Referenz-Implementierung“ Internet natürlich perfekt zum REST-Stil

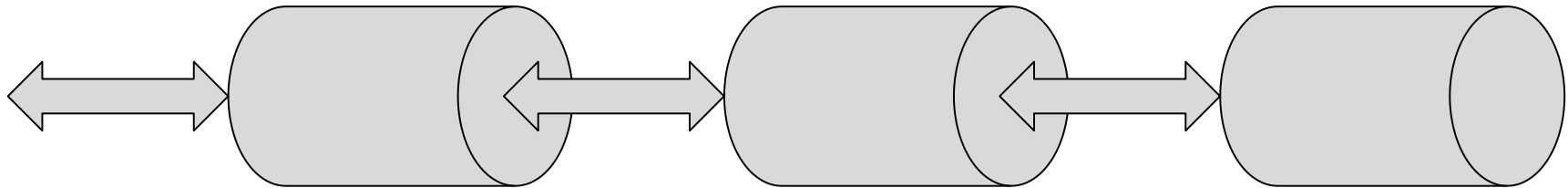
Mapping REST - http

- http Methoden und Ressourcen-Operationen
 - PUT
 - Neu-Anlegen einer Ressource
 - Aktualisierung
 - GET
 - Lesen einer Ressource
 - POST
 - Aktualisierung
 - Neuanlage
 - DELETE
 - Löschen

8.2

CONNECT

- `connect` ist eine Bibliothek, mit deren Hilfe eine http-basierte Middleware realisiert werden kann
 - Im Vergleich zu einem einfachen http-Server ist das Programmier-Modell
 - einfacher
 - aber auch mächtiger
 - Realisierung des http-Protokolls
 - Selbst eine minimale `connect`-Applikation antwortet mit korrekten Status-Codes
 - Ein Dispatcher verbindet die Komponenten miteinander und koordiniert die Aufrufe
 - Eine Liste von `connect`-Components realisiert die konkrete Verarbeitung einer Anfrage
 - Jede Komponente
 - implementiert ihre spezielle Aufgabe
 - und delegiert optional an die nächste Komponente weiter
 - Einige Standard-Komponenten werden bereits zur Verfügung gestellt



- Eine Komponente ist eine JavaScript-Funktion mit drei Parametern
 - Der Request
 - Der Response
 - Die nächste Komponente
 - Dieser Parameter ist für die letzte Komponente der Liste optional
 - Kompatibilität zu vorhandenen http-Server-Implementierungen ist gegeben
- Die Komponente selber wird vom Dispatcher benutzt
 - `app.use('component1').use('component2')`
- Durch Mounting werden Komponenten einem URL-Path zugeordnet
 - `app.use('admin', 'component1').use('admin', 'component2');`
 - `app.use('user', 'component3').use('user', 'component4')`

- Die Standard-Komponenten bieten Funktionalitäten, die für die Bedürfnisse von Web-Applikationen zugeschnitten sind
- Beispiele:
 - Erweiterung des Request- und Response-Objekts
 - Schutz vor typischen Hacker-Angriffen

- `cookieParser()`
 - `req.cookies` und `req.signedCookies`
- `bodyParser()`
 - `req.body` und `req.files`
- `query()`
 - `req.query`
- `session()`
 - `req.session`
 - Die Session ist persistent
- `static()`
 - Liefert statischen Content
- `directory()`
 - Verzeichnis-Browsing

- `limit()`
 - Maximalgröße des Requests
- `basicAuth()`
 - HTTP Basic Authentifizierung
- `csrf()`
 - Schutz vor Cross-Site Request Forgery Angriffen

- `compress()`
 - GZIP des Responses
- `logger()`
- `favicon()`
 - Rückgabe des Favicons für die Adresszeile des Browsers
- `methodOverride()`
 - Erzeugt in Abhängigkeit vom Request PUT- und DELETE-Requests für RESTful-Aufrufe
- `vhost()`
 - Benutzt Komponenten in Abhängigkeit des (virtuellen) Hosts
- `errorHandler()`
 - Im Fehlerfall Rückgabe des Stack-Traces an den Client

8.3

EXPRESS

- **express baut auf connect auf**
 - Damit werden alle Komponenten unterstützt
- **Hinzu kommt**
 - Direkte Unterstützung der http-Methoden
 - Funktionen `get()`, `post()`, ...
 - Eine Templating-Sprache
 - Embedded JavaScript (EJS)
 - Auch andere Templating-Sprachen können benutzt werden
 - Seiten-Navigation mit Routings
 - Ein Projekt-Generator
 - Erzeugen eines vollständigen Projekt-Rumpfs
 - Integriert in das Nodeclipse-Plugin

- Im Gegensatz zu bisher werden nun dynamische Seiten gerendered
 - Diese Seiten werden `views` genannt
- Das Basis-Verzeichnis der Views wird konfiguratativ gesetzt
 - `app.set('views', __dirname + '/views');`
- Das eigentlich Rendering wird über eine Route angestoßen
 - `app.get('/', routes.index);`
 - `app.get('/users', user.list);`
 - Hierbei ist der erste Parameter der URL-Pfad, der zweite eine Routing-Funktion

```
exports.index = function(req, res){  
  res.render('index', { title: 'Express' });  
};
```

- Formulare
- File Upload und File Download
- Authentifizierung
- Komplexe Routen mit Platzhaltern
 - Grundlage für die Realisierung eines RESTful Web Services
- Template Sprachen
 - Embedded JavaScript
 - Jade
 - Hogan
- Beispiele hierzu im Praktikum