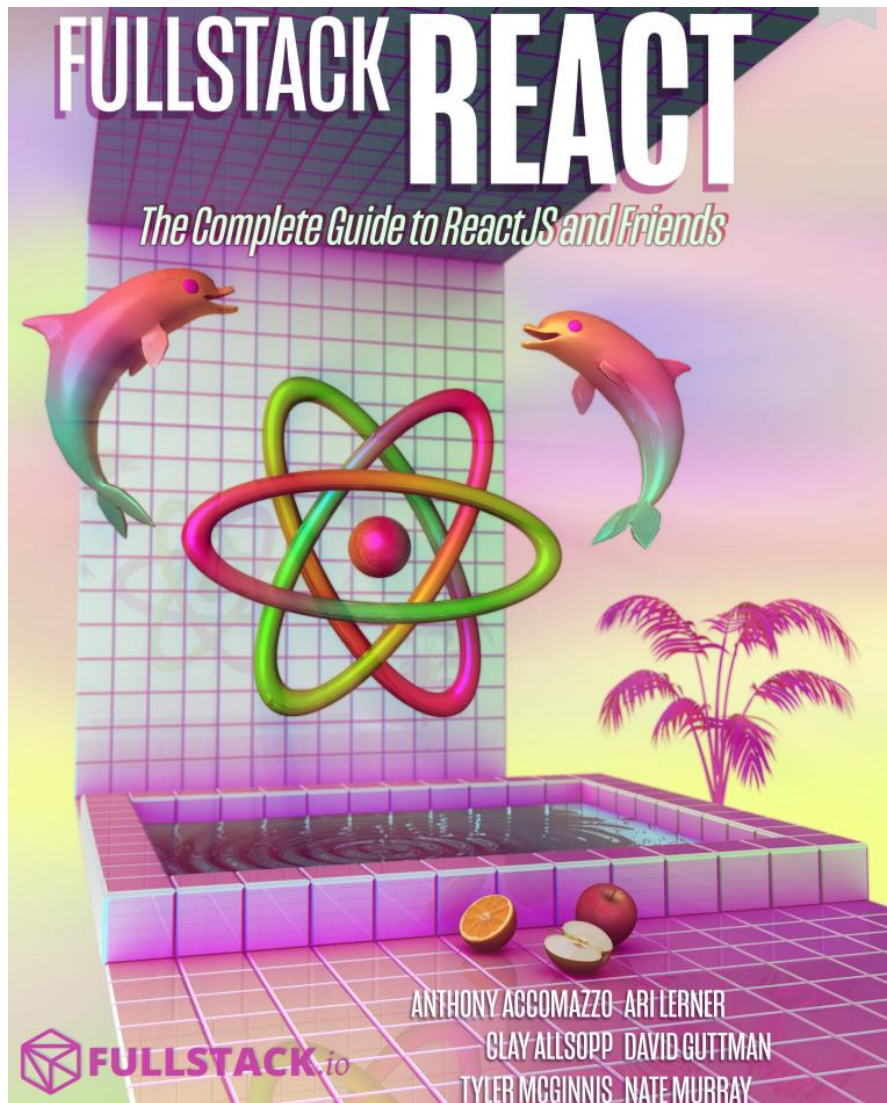


# React

Ein Seminar für die miovent AG



- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
  - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
  - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
  - Musterbeispiele werden zur Verfügung gestellt
  - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
    - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
  - Beispiele unter <https://GitHub.com/JavacreamTraining/org.javacream.training.react>
- Konventionen
  - Befehle werden in Courier-Schriftart dargestellt
  - Dateinamen werden in *kursiver Courier-Schriftart* dargestellt
  - Links werden in unterstrichener Courier-Schriftart dargestellt

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

eMail: [training@rainer-sawitzki.de](mailto:training@rainer-sawitzki.de)

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

Einführung	1
Programmierung	2
Web Anwendungen	3
Weitergehende Themen	4
Anhang Node und NPM	5
Anhang EcmaScript	6

1

# EINFÜHRUNG

1.1

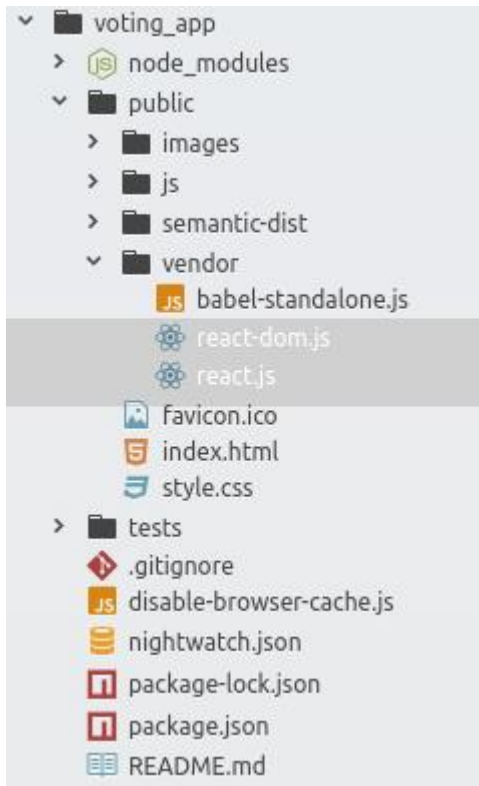
## **INSTALLATION UND SETUP**

- `node` und `npm` sind auf einem Entwicklerrechner zu installieren
  - Näheres hierzu im Anhang
- Damit steht ein ausgefeilter Buildprozess zur Verfügung
  - Verzeichnisstruktur und Projekt-Organisation
  - Automatische Transpilation
  - Browser-Update bei Änderungen an den Quellen
- Ein spezieller Editor ist nicht notwendig
  - Empfohlen wird Atom oder ähnliches



- React kann als npm-Dependency in ein Projekt eingebunden werden
- Alternativ kann React auch durch das Laden der notwendigen Skript-Dateien bereitgestellt werden
  - `react.js`
  - `react-dom.js`
- Zusätzlich kann auch der React-Projektgenerator zusätzlich installiert werden
  - `npm install -g create-react-app`

# Projektstruktur: npm-Projekt mit Laden der React-Skripte



```
{
  "name": "voting_app",
  "version": "1.1.0",
  "author": "Fullstack.io",
  "scripts": {
    "go": "open http://localhost:3000; npm run server",
    "e2e": "nightwatch",
    "test": "./node_modules/.bin/concurrently -k 'npm run server' 'npm run e2e'",
    "start": "npm run server",
    "server": "live-server public --host=localhost --port=3000 --middleware=./disable-browser-cache.js"
  },
  "private": true,
  "devDependencies": {
    "concurrently": "2.2.0",
    "live-server": "git://github.com/acco/live-server.git"
  }
}
```

```
{  
  "name": "first_react",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "react": "^16.2.0",  
    "react-dom": "^16.2.0",  
    "react-scripts": "1.1.0"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test --env=jsdom",  
    "eject": "react-scripts eject"  
  }  
}
```

1.2

## **EINE ERSTE ANWENDUNG**

- Eine typische React-Anwendung besteht aus
  - Einer index-Seite
    - Diese lädt alle notwendigen Skripte
    - Weiterhin definiert sie einen Bereich, in dem sich die React-Anwendung befinden wird

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Project One</title>
    <link rel="stylesheet" href="../semantic-dist/semantic.css" />
    <link rel="stylesheet" href="../style.css" />
    <script src="vendor/react.js"></script>
    <script src="vendor/react-dom.js"></script>
  </head>
  <body>
    <div class="main ui text container">
      <h1 class="ui dividing centered header">Popular Products</h1>
      <div id="content"></div>
    </div>
    <script src="../js/seed.js"></script>
    <script src="../js/app.js"></script>
  </body>
</html>
```

# Beispiel: index.html mit inline-Babel-Transpilation

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Project One</title>
    <link rel="stylesheet" href="../semantic-dist/semantic.css" />
    <link rel="stylesheet" href="../style.css" />
    <script src="vendor/babel-standalone.js"></script>
    <script src="vendor/react.js"></script>
    <script src="vendor/react-dom.js"></script>
  </head>
  <body>
    <div class="main ui text container">
      <h1 class="ui dividing centered header">Popular Products</h1>
      <div id="content"></div>
    </div>
    <script src="../js/seed.js"></script>
    <script
      type="text/babel"
      data-plugins="transform-class-properties"
      src="../js/app-complete.js"
    ></script>
  </body>
</html>
```

- Das Script definiert eine React-Komponente
- und lässt diese rendern
  - Dazu wird die Anbindung an ein HTML benötigt



# Beispiel: Eine simple React-Komponente

```
class ProductList extends React.Component {  
  render() {  
    return (  
      <div className='ui unstackable items'>  
        Hello, friend! I am a basic React component.  
      </div>  
    );  
  }  
}  
  
ReactDOM.render(  
  <ProductList />,  
  document.getElementById('content')  
);
```

2

## PROGRAMMIERUNG

2.1

## COMPONENTS

- Eine React-Komponente ist eine ECMA-Klasse, die eine `render`-Methode zur Verfügung stellt
- Der Rückgabewert der `render`-Methode ist eine Baum von DOM-Elementen, die die Komponente darstellen
  - Zur Vereinfachung kann dieser Baum als spezielles HTML-Fragment definiert werden
  - Hierfür wird JSX benutzt, eine JavaScript-Erweiterung
    - Diese wird von einem Transpiler in valides JavaScript übersetzt
- Diese Elemente werden einem so genannten Virtual DOM zugeordnet
- Das Virtual-DOM der Komponente wird durch React in ein HTML eingebunden
- Komplexere Komponenten enthalten zusätzliche Logik
  - Datenhaltung
  - Event-Handler
  - Seiten-Navigation

# Eine simple React-Komponente

```
class ProductList extends React.Component {  
  render() {  
    return (  
      <div className='ui unstackable items'>  
        Hello, friend! I am a basic React component.  
      </div>  
    );  
  }  
}  
  
ReactDOM.render(  
  <ProductList />,  
  document.getElementById('content')  
);
```

Notwendige  
Vererbungshierarchie

render-Funktion

JSX-Ausdruck

Darstellung der Komponente  
im HTML

Klassen-Name der  
Komponente

Id des HTML-Elements, das die  
Komponente aufnimmt

- Sollen mehrere Komponenten dargestellt werden, so können diese
  - In verschiedene Elemente der index.html gerendered
    - eher ungebräuchlich
  - oder einfach im JSX einer anderen Komponente benutzt werden
- Wichtig:
  - JSX wird nach JavaScript übersetzt
  - Deshalb können JSX-Ausdrücke selbstverständlich in
    - Kontrollstrukturen
    - Schleifen
    - Zuweisungen
    - Parametern
    - ...
  - benutzt werden!

2.2

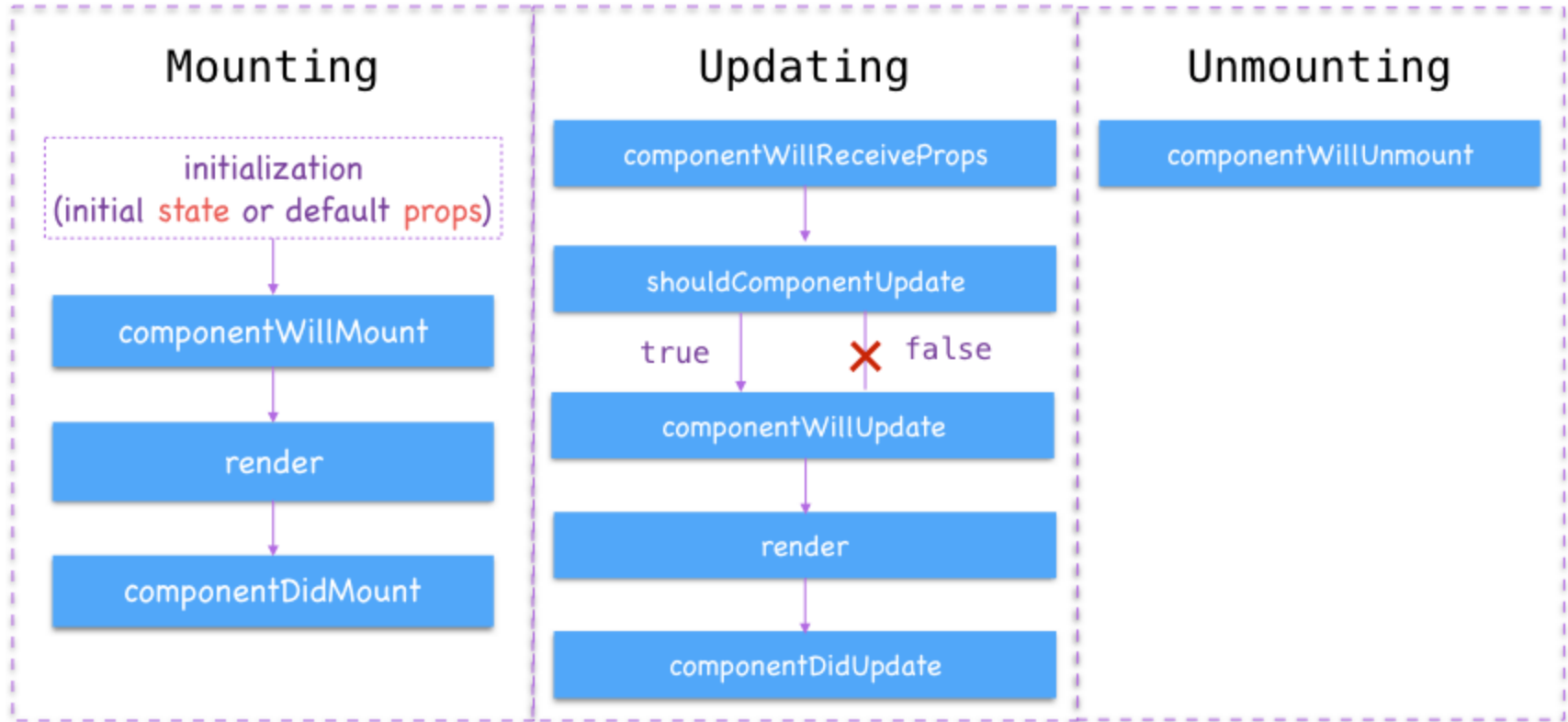
## PROPERTIES UND STATE

- Enthält eine Komponente eine andere Komponente, so kann diese die Sub-Komponente mit Daten versorgen
  - Properties
- Dazu wird das JSX-Element der Sub-Komponente mit Platzhaltern versehen
  - `placeholder = {expression}`
- Innerhalb der Komponente erfolgt der Zugriff über
  - `this.props.placeholder`
- Ein spezieller Placeholder = `key`
  - Dieser wird intern für eine eindeutige Referenzierung der Komponente benutzt



- Die Properties werden von der Parent-Komponente verwaltet
  - Die Sub-Komponente referenziert nur die Properties der Parent-Komponente
- Properties sind unveränderlich ("immutable")
  - Die Gründe dafür sind in der React-Architektur begründet
  - Werden Properties verändert, so wird die React-Anwendung einfach nicht funktionieren

- Im Gegensatz zu den Properties verwaltet eine Komponente ihren eigenen State
- Dieser darf verändert werden
  - Allerdings nicht ohne Berücksichtigung der React-Architektur
    - State-Änderungen werden stets durch neue Objekte oder Kopien des ursprünglichen State-Objekts signalisiert
    - und müssen innerhalb der Komponente mit `this.setState(newState)` signalisiert werden
- Die Properties einer Subkomponente können werden durch den State-definiert
  - und damit mit dem nächsten render-Zyklus dargestellt



2.3

## JSX

- JavaScript wird um ein "DOM-Literal" erweitert
  - `let element = <div>Hello</div>`
  - Dieses Literal wird vom JSX-Transpiler in JavaScript übersetzt
    - `React.createElement(...)`
    - `element` ist ein `ReactElement`
- In diesem Literal können JavaScript-Expressions benutzt werden
  - `{expression}`
  - Es wird jedoch nur ein Subset unterstützt
    - beispielweise können keine Deklarationen erfolgen

- Jedes JavaScript-Objekt, dass ein `ReactElement` repräsentiert, kann im Dom-Literal benutzt werden
  - Component-Klasse
    - `class Person extends React.Component`
  - Funktionen
    - `Greeter = () => { return <div>Hello</div>}`
  - Referenzen
    - `const Greeting = <div>Hello</div>`
- Diese Objekte können dann selbst wiederum im Dom-Literal benutzt werden
  - `<Person />`
  - `<Greeter />`
  - `<Greeting />`
- Notwendige Konvention
  - HTML-Elemente beginnen mit einem Kleinbuchstaben
  - JavaScript-Elemente beginnen mit einem Großbuchstaben

- <https://reactjs.org/docs/jsx-in-depth.html>

2.4

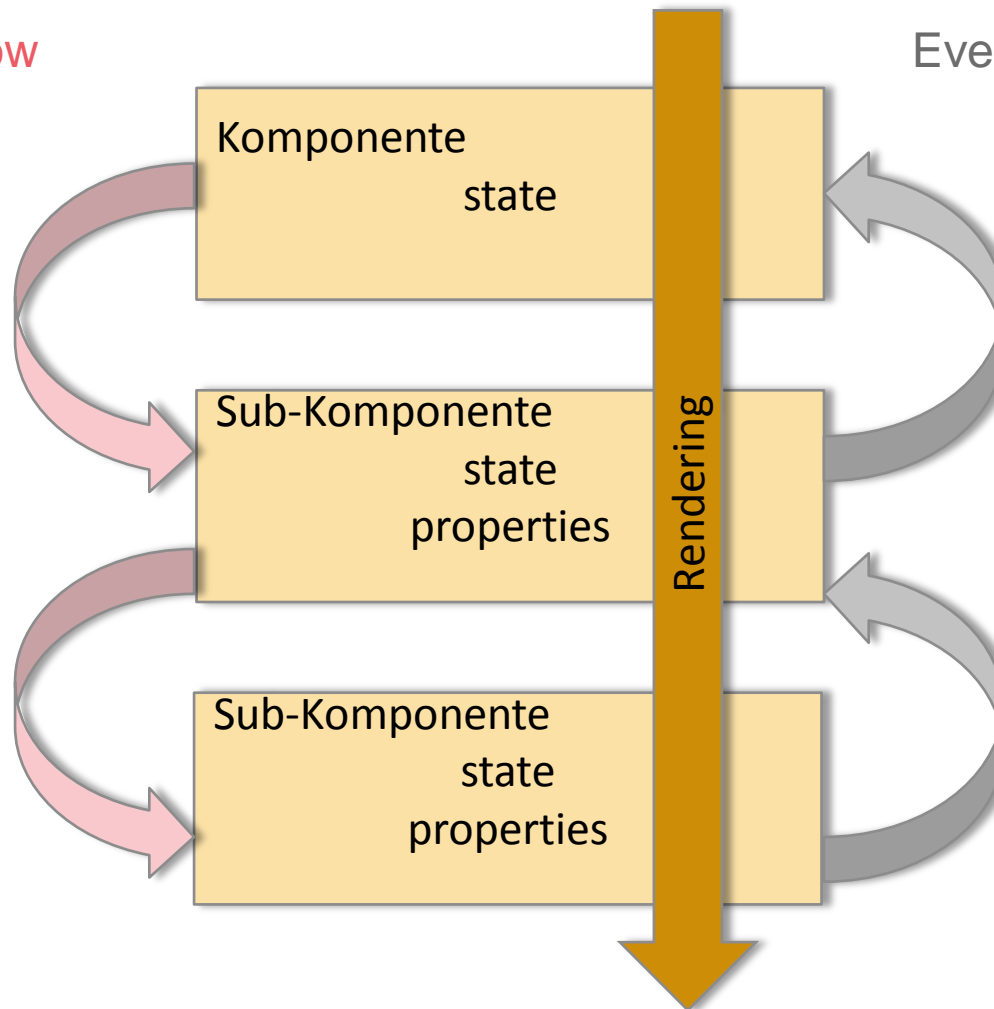
## EVENTS



- Events werden als JavaScript-Methoden im JSX an die HTML-Elemente gebunden
- In einer verschachtelten Komponenten-Hierarchie muss darauf geachtet werden, dass die Events an der richtigen Stelle zu Änderungen führen
  - Properties sind Immutable
  - Damit darf nur State geändert werden, so dass gegebenenfalls eine Event-Verarbeitung in der Hierarchie nach oben gereicht werden muss

Data Flow

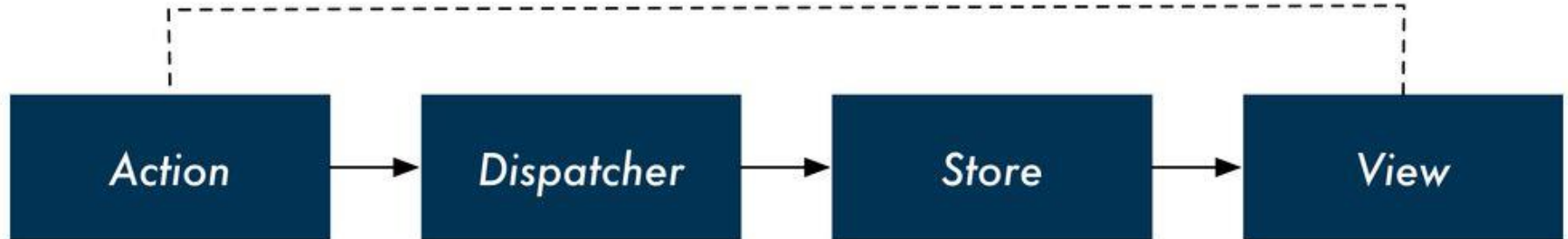
Event Flow



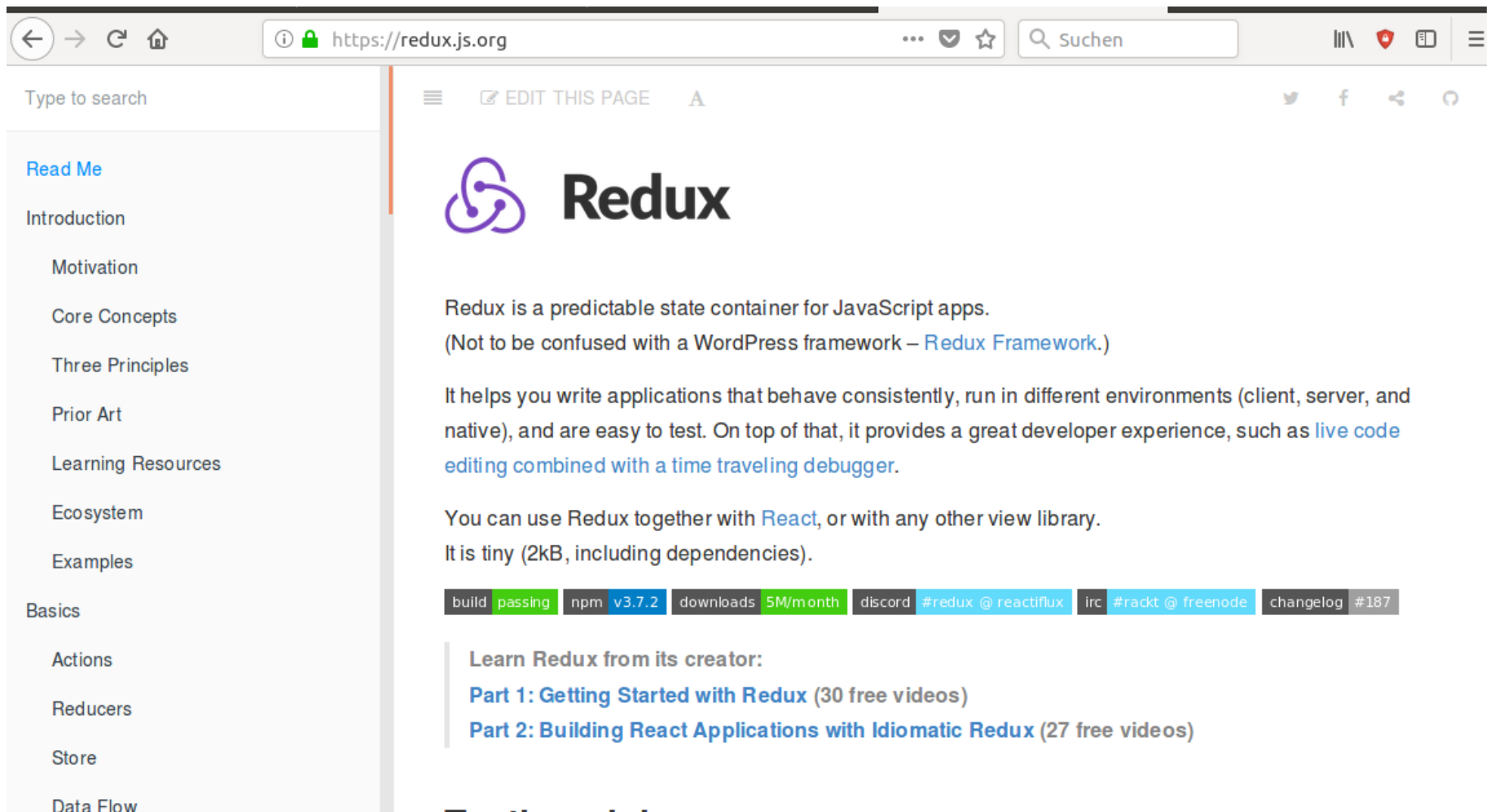
2.5

## FLUX UND REDUX

- Eine Erweiterung des klassischen MVC-Patterns
  - Der Store hält die Daten
    - Und entspricht dem ursprünglichen Model
  - Views repräsentieren den Zustand des Stores
    - der Store informiert die View über die Zustands-Änderungen
    - Dies entspricht 1 zu 1 der ursprünglichen View-Definition
  - Die View signalisiert Interaktionen durch das Versenden von Actions
    - Action-Objekte haben
      - Einen Typ
      - Beliebige weitere Parameter
  - Ein Dispatcher ruft leitet die Actions zum Store
    - und entspricht somit dem ursprünglichen Controller



# Redux: Eine Implementierung des Flux-Patterns



The screenshot shows the Redux.js.org website in a web browser. The browser's address bar displays 'https://redux.js.org'. The website has a sidebar on the left with a search bar and a list of navigation links: 'Read Me', 'Introduction' (with sub-links for 'Motivation', 'Core Concepts', 'Three Principles', 'Prior Art', and 'Learning Resources'), 'Ecosystem', 'Examples', 'Basics' (with sub-links for 'Actions', 'Reducers', 'Store', and 'Data Flow'), and 'Data Flow'. The main content area features the Redux logo, the title 'Redux', and a description: 'Redux is a predictable state container for JavaScript apps. (Not to be confused with a WordPress framework – [Redux Framework](#).)'. It further explains that Redux helps write applications that behave consistently across different environments and provides a great developer experience with live code editing and a time-traveling debugger. It also mentions that Redux can be used with React or other view libraries and is very small (2kB). A status bar shows 'build passing', 'npm v3.7.2', 'downloads 5M/month', 'discord #redux @ reactiflux', 'irc #rackt @ freenode', and 'changelog #187'. At the bottom, it offers to 'Learn Redux from its creator' with links to 'Part 1: Getting Started with Redux (30 free videos)' and 'Part 2: Building React Applications with Idiomatic Redux (27 free videos)'.

3

## WEB ANWENDUNGEN

3.1

## **REST**



- 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.<sup>[1]</sup> 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

- 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

- 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

- 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

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

- Mit PUT
  - Der Client muss die Ressourcen-ID mit angeben
  - Rückgabe ist ein Statuscode „201: Created“
- Mit POST
  - Der Server entscheidet, ob er eine neue Ressource anlegen muss
  - Falls ja:
    - Statuscode „201: Created“
    - Gesetzter `Location`-Header mit URI der eben angelegten Ressource
    - Optional: Body enthält die angelegte Ressource

- Mit PUT
  - Statuscode „200: OK“ oder „204: No content“
  - PUT ist idempotent (!)
- Mit POST
  - POST wird für nicht-idempotente Updates benutzt



- Mit DELETE
  - Statuscode „200: OK“ oder „204: No content
  - PUT ist idempotent (!)
- Konzeptionell muss unterschieden werden:
  - Ein „echtes“ DELETE löscht die Ressource
  - Ein fachliches Löschen (z.B. Storno) ist eigentlich ein Update der Ressource
    - Ein überladen des http-DELETE ist für diese Zwecke jedoch durchaus legitim
      - `DELETE order/ISBN42?cancel=true`

3.2

## **ASYNCHRONE PROGRAMMIERUNG**

- Mit ES6 wurde das `fetch`-API eingeführt
  - basiert auf Promises

- Beispiel GET

```
return fetch(endpointUrl, {
  headers: {
    Accept: 'application/json',
  },
}).then(checkStatus)
  .then(parseJSON)
  .then(success);
}
```

- Beispiel PUT

```
fetch('/api/timers', {
  method: 'post',
  body: JSON.stringify(data),
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  },
}).then(checkStatus);
```

3.3

## ROUTING

- React Routing ermöglicht eine Navigation innerhalb einer Single Page Application
- Auf Grund der Benutzer-Anforderungen nicht ganz trivial:
  - Aktualisierung der im Browser dargestellten URL
  - Unterstützung von Browser-Historie und Back-Button

- Routing-Definitionen erfolgen deklarativ
  - Umhüllen der Oberflächen-Definition durch ein `<Router>`-Element
  - Definition von `<Route>`-Elementen, die auf Components verweisen
    - Eine Route besteht mindestens
      - aus einem Path
      - der Angabe einer Component
      - `<Route path="/about" component={About} />`

4

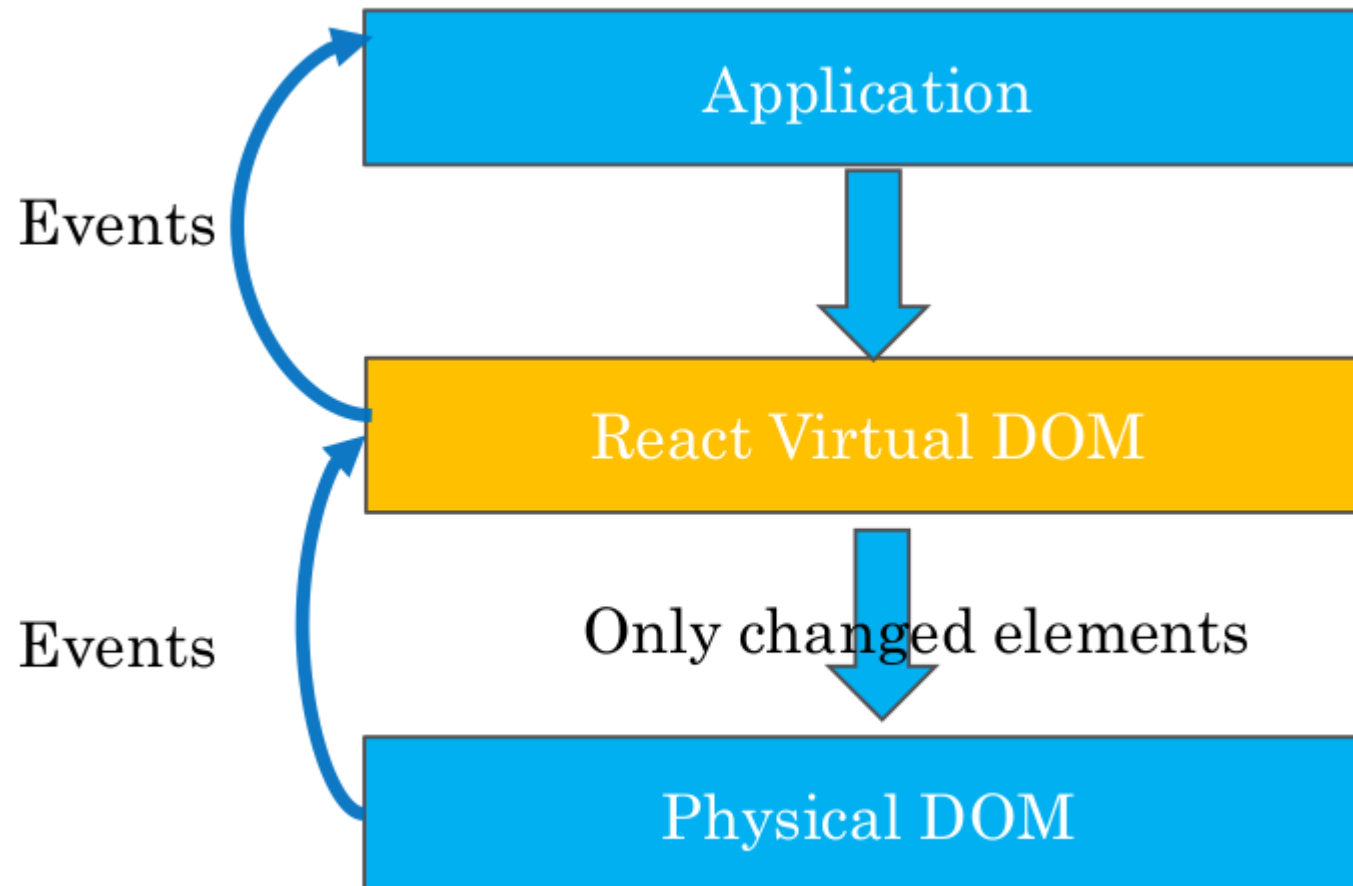
## VERSCHIEDENES

4.1

## VIRTUAL DOM



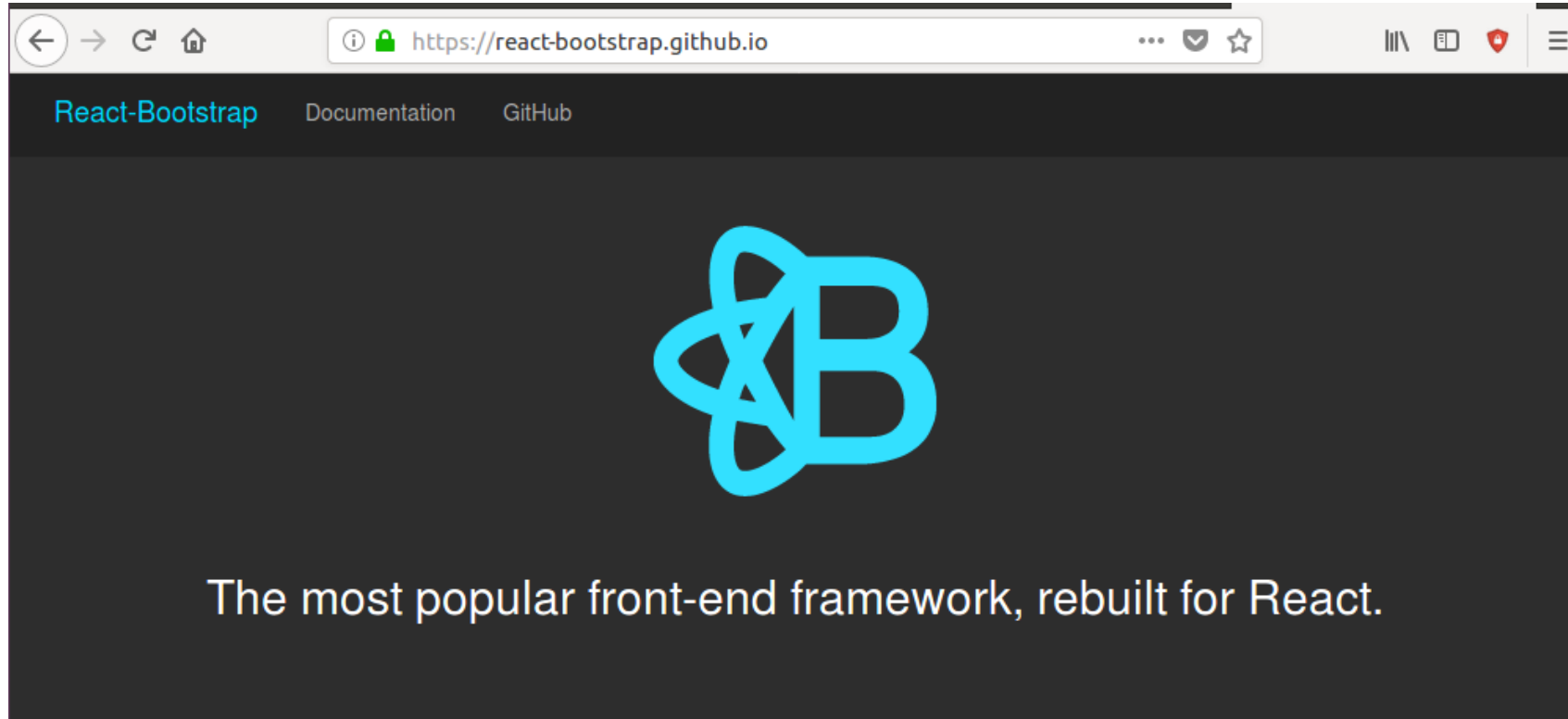
- React verwaltet ein Virtuelles DOM
  - Eigenes In-Memory-Model des Browser-DOMs
- Das Programm manipuliert nur das Virtual DOM
  - Änderungen erfolgen damit rein im Speicher und sind damit sehr performant
- React implementiert einen effizienten Algorithmus, um Änderungen des Virtual DOM in das Browser-DOM zu übertragen
- Auch Änderungen des Browser-DOMs werden registriert und in das Virtual DOM übernommen



4.2

## **REACT BOOTSTRAP**

- Bootstrap ist ein etabliertes JavaScript-Framework
  - Bestandteile
    - CSS-Gridsystem
    - Theming
    - Komponenten-Bibliothek
- Bootstrap basiert intern auf jQuery
- React Bootstrap ist ein Fork unter Benutzung von React



4.3

## **REACT AUF DEM SERVER**

- React DOM kann auch auf dem Server ausgeführt werden
  - Dieser sendet dem Browser fertige HTML-Seiten
- Konsequenzen
  - Entlastung des Clients
  - React-Anwendungen können auch ohne aktiviertes JavaScript im Browser verwendet werden
  - Search Engine Optimizations (SEO)

4.4

## TESTEN MIT JEST



- JEST ist ein Unit-Test-Framework auf Basis von Jasmine
- Erweiterungen umfassen
  - Automatisches Erkennen von Test-Skripten
  - Automatische Ausführung der erkannten Tests
  - Asynchrone Methoden können einfach sequenziell verwendet werden
  - Mit jsdom wird ein Fake DOM für den Test bereitgestellt
    - Ausführung des Tests damit in einer simplen Umgebung ohne Browser möglich
  - Parallelisierung von Test-Läufen
  - Mocking-Framework

```
describe('My test suite', () => {  
  it('`true` should be `true`', () => {  
    expect(true).toBe(true);  
  });  
  it('`false` should be `false`', () => {  
    expect(false).toBe(false);  
  });  
});
```

4.5

## **REACT NATIVE – EINE ÜBERSICHT**

- iOS und Android unterstützen sowohl Browser-basierte als auch native Anwendungen
  - Native Apps werden für die Zielplattform speziell kompiliert und über einen Store installiert
- Cross-Platform-Applications werden in einem einzigen Sourcecode definiert und in die nativen Plattformen übersetzt
  - Wird der Web Stack HTML, CSS und JavaScript benutzt, spricht man auch von Hybrid Applications

- React Native ist eine Hybrid-Sprache
- Allerdings definiert React Native einen separaten Satz von Komponenten
  - Diese entsprechen der Schnittmenge der gemeinsam verfügbaren Elemente auf iOS und Android
  - `<View>`
  - `<Text>`
  - `<ListView>`
  - `<WebView>`
  - ...

5

**ANHANG**

5.1

## **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	
 <b>Windows Installer</b> <small>node-v6.11.4-x86.msi</small>	 <b>Macintosh Installer</b> <small>node-v6.11.4.pkg</small>	 <b>Source Code</b> <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

5.2

## **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/>

5.3

## **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



# Beispiel: Ein einfaches Projekt

```
{  
  "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

6

## ECMAScript



6.1

## **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;  
    }  
}
```

6.2

## **SCOPED VARIABLES UND KONSTANTEN**

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

6.3

## **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
```

6.4

## **VEREINFACHTE FUNKTIONSDEKLARATION**

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

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

6.5

## **GENERATORS UND PROXIES**

- Generators sind spezielle Funktionen, die den Kontrollfluss an die aufrufende Funktion zurück delegieren
  - Dafür wird die `yield`-Funktion eingeführt

```
function* sampleGenerator() {  
  print('First');  
  yield("Hugo");  
  print('Second');  
};  
//...  
  
let gen = sampleGenerator();  
print(gen.next());  
print(gen.next());
```

- Proxies erweitern ("dekorieren") bereits vorhandene Funktionen und Objekte
- Dieses Design-Pattern ist in untypisierten Sprachen sehr einfach umzusetzen

```
var handler = {  
    get: function (target, name)  
        return Reflect.get(target, name) },  
    apply: function (receiver, ...args) {  
        print("applying...")  
    }  
};  
//...  
obj = new Proxy(obj, handler);
```

6.6

## 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);  
    });  
}
```

# Beispiel: async await

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

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