



# JavaScript

Anwendungsentwicklung



Cegos Group

inspire  
qualify  
change



# Inhalts- verzeichnis



Software-Entwicklung mit JavaScript



Workshop



Das node-System



Projektorganisation



Testen





Fortgeschrittene Programmierung



# Software-Entwicklung mit JavaScript



-  Ausgangssituation und Probleme
-  Begriffe und Einordnung



# Ausgangssituation und Probleme



# JavaScript : Softwareentwicklung

- 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: Eine untypisierte Sprache

- 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



# JavaScript: Buildprozess

- 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



# JavaScript: Testen

- 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...





# Begriffe und Einordnung



# JavaScript Engines

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

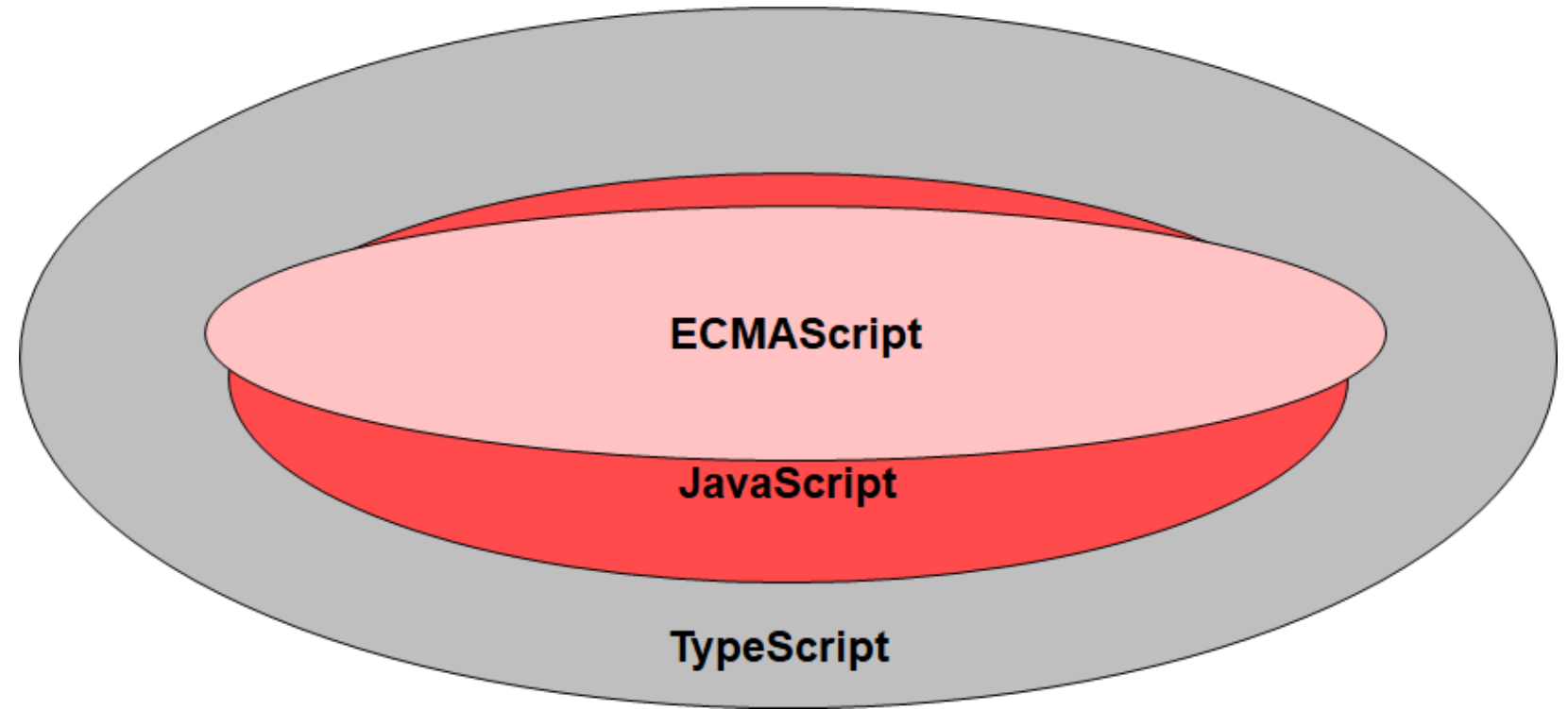


# Programmiersprachen

- 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
  - ...



# Übersicht: Programmiersprache n





# Werkzeuge

- 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



# Das node-System



node.js



npm – Der Node Package Manager



Node-Modules



Ein erstes Projekt



# node.js





## Was ist 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  
node-v6.11.4-x86.msi

  
Macintosh Installer  
node-v6.11.4.pkg

  
Source Code  
node-v6.11.4.tar.gz

- 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	



# Testen der Installation

- `node -v`
  - Ausgabe der Versionsnummer
- `node`
  - Starten der REPL zur Eingabe von JavaScript-Befehlen
- `node programm.js`
  - Ausführen der Skript-Datei *programm.js*



# Node und Browser-basierte Anwendungen

- 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



# npm – Der Node Package Manager



## Was ist npm?

- 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 Kommandos

- `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/>





# Node-Modules



# 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



## Die package.json

- 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

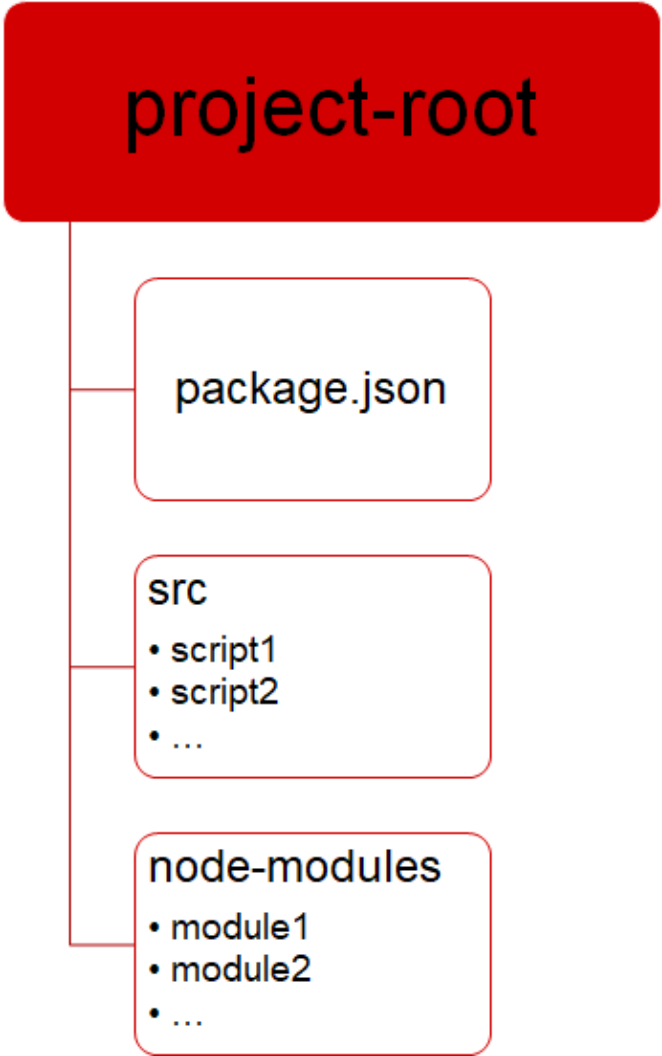


# Initialisierung eines Projekts

- 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



# Projektstruktur





## 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"
}
```



## Beispiel: Ein einfaches Node-Module

- Datei `index.js`

```
module.exports = {  
  log: function() {  
    console.log('Hello')  
  }  
}
```
- In der REPL

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



# Installieren von Abhängigkeiten

- 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





# Ein erstes Projekt



## Workflow zum Anlegen eines JavaScript-Projekts

- 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

The screenshot shows a code editor interface. On the left, a 'Project' sidebar displays a folder named 'javascript-tools' which contains a file named 'package.json'. The main editor area shows the content of 'package.json' with the following JSON structure:

```
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
```

The status bar at the bottom indicates 'package.json 1:1' and 'LF UTF-8 JSON 0 files'.



## Beispiel: Installieren des prototype- Packages

- 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 an IDE with two panels. The left panel, titled 'Project', displays a file tree for a project named 'javascript-tools'. The tree structure is as follows:

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

The right panel, titled 'package.json', shows the content of the selected file. The code is as follows:

```
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
```



## Programmierbeispiel: prototype\_demo.js

```
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!



# Projektorganisation



Vorbemerkung



Packages



Versionsmanagement



Server



Babel-Transpiler



TypeScript



Continuous Build



Linters



Webpack



# Vorbemerkung





# Zielsetzung und Vorgehensweise

- Ziel dieses Abschnittes ist es, eine komfortable und stabile Umgebung für ein JavaScript-Projekt zur Verfügung zu definieren
  - Das Ergebnis ist natürlich weder eindeutig noch 1 zu 1 auf andere Projekte übertragbar
- Dazu wird ein Projekt angelegt und Schritt für Schritt mit den notwendigen Features erweitert
  - Welche Bibliotheken und Techniken eingesetzt werden ist subjektiv
    - So wird beispielsweise als Packaging Manager `npm` vorgestellt, obwohl mit Facebooks `yarn` eine ernsthafte Alternative vorliegt
  - Die Auswahl berücksichtigt jedoch langjährige Erfahrungen und Best Practices



# Packages



## Dependencies und die package.json

- Das Installieren eines npm-Packages erfolgt sinnvoll mit der Option `save`
  - `npm install <package> -- save`
  - Damit wird die Dependency automatisch in die `package.json` eingetragen
  - Ab npm 5 wird diese Option automatisch benutzt
  - Ebenso wird ab npm Version 5 zusätzlich die Datei `package-lock.json` angelegt
    - Diese dient zur Optimierung und zur Behebung eines npm-Bugs, der nicht-reproduzierbare Module-Installationen verursachen konnte
- Nach der Installation können die Funktionen des installierten Moduls sofort im eigenen Projekt benutzt werden
  - Aber nur in der `node`-Umgebung, nicht im Browser
  - Dies wird später eingeführt
    - `browserify`
    - `webpack`



# Dependency Scope

- Zur Übersichtlichkeit werden Dependencies in unterschiedlichen Scopes verwaltet
  - Dependencies
    - Müssen bei einer Verteilung der erstellten Software mit ausgeliefert werden
  - Developer Dependencies
    - Werden nur während der Entwicklung benutzt und sind nicht Bestandteil der Distribution
- Beispiel:
  - Eine Abhängigkeit zu einer externen Bibliothek ist eine Dependency
  - Web Server, das Testframework oder Transpiler sind Developer Dependencies
- Die Unterscheidung wird beim Installieren des Packages getroffen
  - Dependency
    - `npm install --save`
  - Developer Dependency
    - `npm install --save-dev`



# Versionsmanagemen t



# Versionskontrolle

- Wird in diesem Projektverzeichnis ein Modul installiert, so wird es im Unterverzeichnis `node-modules` abgelegt
- Damit enthält dieses Verzeichnis allerdings nur reproduzierbare Informationen, die redundant sind
  - Damit wird dieses Verzeichnis nicht unter Versionskontrolle gestellt!
- Die Projekt-Abhängigkeiten werden in der `package.json` abgelegt
  - Diese Datei ist selbstverständlich versioniert
  - Zusätzlich muss ab npm Version 5 die `package-lock.json` in die Versionsverwaltung aufgenommen werden

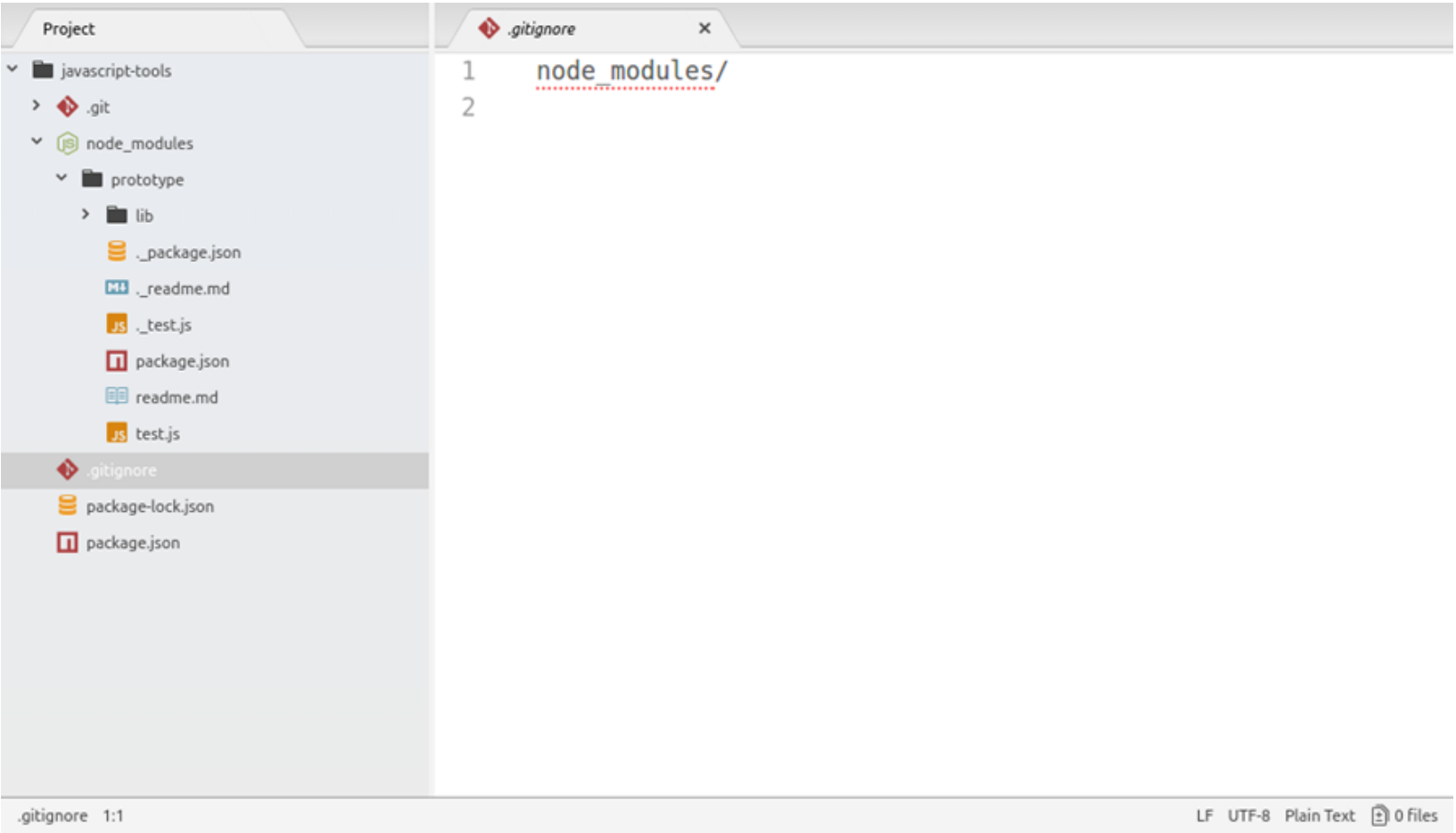


## Beispiel: Ein JavaScript-Projekt mit Git

- Im Projektverzeichnis Aufruf des Programms `git init`
- Erstellen einer Datei `.gitignore`
  - Darin wird `node-modules` eingetragen
- Ein Aufruf von `git status` zeigt nur die `package.json`, `package-lock.json` und `.gitignore` an
- Weiterer Workflow:
  - Hinzufügen von Dateien zum Repository
    - `git add *`
  - commit in die Versionsverwaltung
    - `git commit -m "commit message"`
  - Optional: Hochladen in ein Online-Repository
    - `git push`



# Das Projekt unter Versionskontrolle







# Server



## Server Packages

- In der `npm`-Registry sind eine ganze Reihe von Web Servern abgelegt
  - '2328 packages found for "http server"'
  - `http-server`
  - `lite-server`



## lite-server: Installation und Start

- `npm install lite-server --save-dev`
  - Ein Server für statische Inhalte
  - Hinweis:
    - Der Server installiert eine große Menge von Node-Modulen
- Erweitern der `package.json` um einen dev-Eintrag im `scripts`-Objekt

```
"serve": "lite-server"
```
- Start mit `npm run serve`
  - Standard-Port: 3000
- Aufruf des Servers aus dem Browser heraus mit `http://localhost:3000`
  - Hierzu muss noch eine Datei namens `index.html` im Projektverzeichnis erstellt werden



# Das Projekt mit installiertem lite- server

The screenshot shows a code editor with a project explorer on the left and two open files on the right. The project explorer shows a folder named 'javascript-tools' containing files: '.git', 'node\_modules', '.gitignore', 'index.html', 'package-lock.json', and 'package.json'. The 'package.json' file is selected. The editor shows the content of 'package.json' and 'index.html'.

```
1 {
2   "name": "javascript-tools",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "serve": "lite-server",
8     "test": "echo \"Error: no test specified\" && exit 1"
9   },
10  "keywords": [
```

```
1 <html>
2 <head>
3 <title>Simple Page</title>
4 </head>
5 <body>
6 <h1>Hello World!</h1>
7 </body>
8 </html>
9
```

File 0 Project 0 ✓ No Issues package.json 7:27 (1, 22) LF UTF-8 JSON 0 files

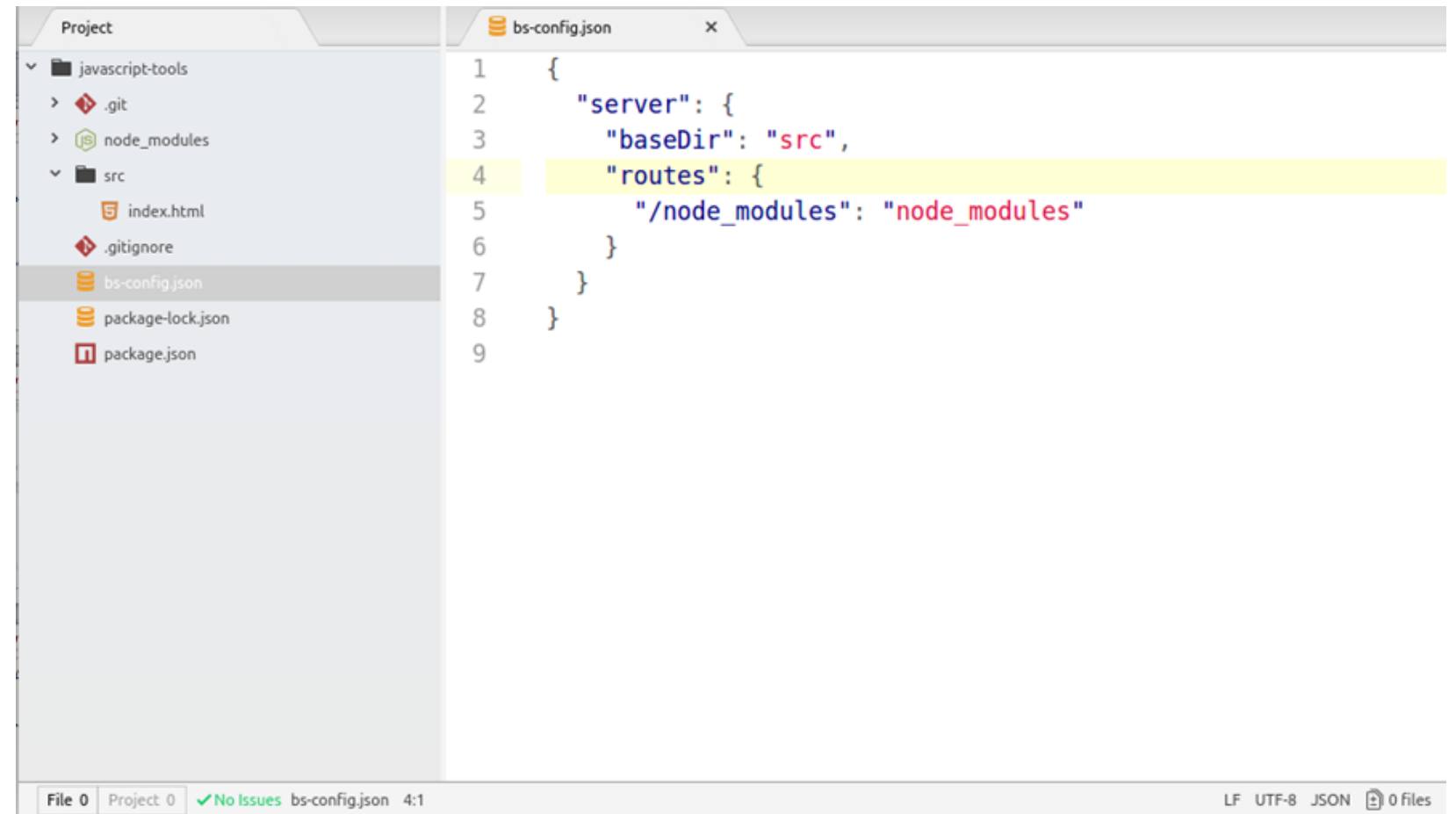


# lite-Server: Konfiguration

- Dokumentation unter <https://www.npmjs.com/package/lite-server>
- Der lite-server aktiviert automatisch Browsersync
  - Konfiguration in *bs-config.json*
    - Port
    - Server-Base-Directory, aus dem die Quellcodes (html, js, css...) gelesen werden
  - Dokumentation: <https://browsersync.io/docs/options/>



# Das Projekt mit konfiguriertem lite- server





# Babel-Transpiler



# Was ist "Transpilation"?

- Ein Transpiler erzeugt aus einer Script-Sprache eine andere
  - Ein Compiler arbeitet etwas anders
    - Beispiel: Der Java-Compiler erzeugt aus einer nicht direkt ausführbaren .java-Datei eine .class-Datei, die von der Java Virtual Machine ausgeführt werden kann
- Beispiele:
  - Der LESS Transpiler erzeugt CSS
  - grooscript erzeugt JavaScript aus Groovy



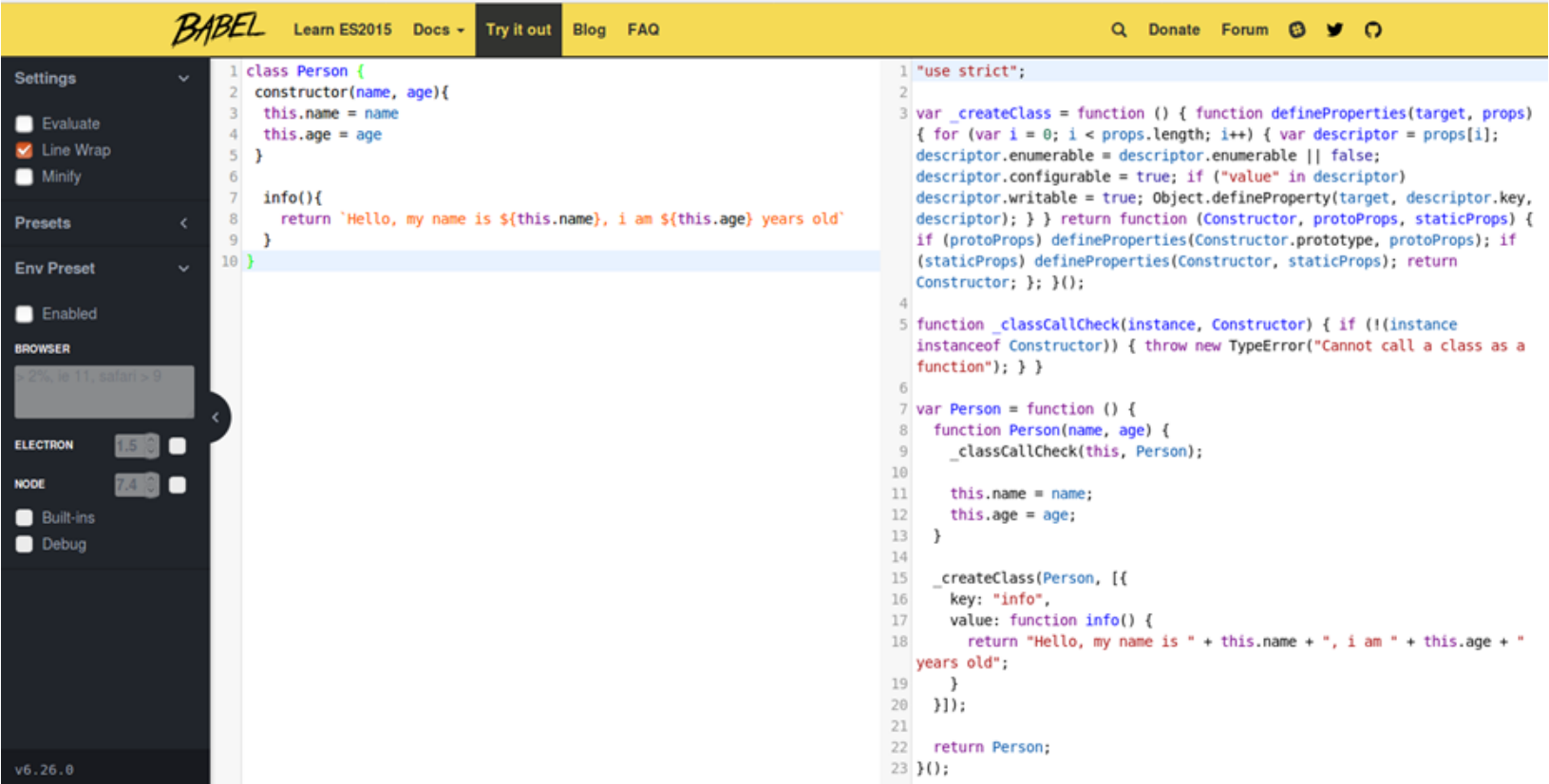


# Babel: Ein JavaScript Transpiler

- <https://babeljs.io/>
- Features:
  - Unterstützung von ECMAScript6
  - Statische und dynamische Transpilation
    - Bei der dynamischen Transpilation wird das Browser-kompatible JavaScript im Browser erzeugt
  - Erweiterbar durch Plugins



# Beispiel: Try Out-Editor des Babel-Projekts





# Beispiel: Babel-Transpilation im Browser

```
1 <html>
2 <head>
3   <title>Simple Babel Page</title>
4 </head>
5 <body><script id="__bs_script__">
6   document.write("&lt;script async src='/browser-sync/browser-sync-client.js?v=2.18.13'&gt;&lt;/script&gt;".replace("HOST", location.hostname));
7 //]]&gt;&lt;/script&gt;
8
9
10 &lt;div id="output"&gt;&lt;/div&gt;
11 &lt;!-- Load Babel --&gt;
12 &lt;script src="https://unpkg.com/babel-standalone@6/babel.min.js"&gt;&lt;/script&gt;
13 &lt;!-- Your custom script here --&gt;
14 &lt;script type="text/babel"&gt;
15   const getMessage = () =&gt; "Hello World";
16   document.getElementById('output').innerHTML = getMessage();
17 &lt;/script&gt;
18 &lt;/body&gt;
19 &lt;/html&gt;
20</pre></div><div data-bbox="508 648 962 808" data-label="Diagram"><img alt="Diagram showing three components: ECMAScript, Laden von Babel, and Browsersync. Red lines connect the code snippets to these components: line 12 to 'Laden von Babel', line 14 to 'ECMAScript', and line 6 to 'Browsersync'."/><pre>graph LR; A[ECMAScript] --- B[Laden von Babel]; B --- C[Browsersync];</pre></div><div data-bbox="27 965 118 988" data-label="Page-Footer"><p>2.0.0820 © Javacream</p></div><div data-bbox="432 965 568 988" data-label="Page-Footer"><p>JavaScript Anwendungsentwicklung</p></div><div data-bbox="978 965 993 988" data-label="Page-Footer"><p>59</p></div>
```



## Babel: Installation über npm

- `npm install babel-cli --save-dev`
- In der `package.json` wird ein neues Skript eingetragen
  - `"babel-transpile": "babel src -d dist --ignore *.js"`
  - Hinweis: Das dist-Verzeichnis ist nicht Bestandteil der Versionsverwaltung
- Installation des Environments `env-preset`
  - Dieses bestimmt die nötigen Plugins automatisch
  - ECMA2015+ wird sofort unterstützt
  - `npm install babel-preset-env --save-dev`
- Konfiguration des Plugins in der `.babelrc`
  - `{ "presets": ["env"] }`



## Babel: Transpilation

- Aufruf der Transpilation über `npm run babel-transpile`
- Oder noch besser: `watch`-Option benutzen!
  - Nun bleibt der Babel-Transpiler aktiv und arbeitet bei jeder Änderung der Quelle
  - Und Browsersync aktualisiert sofort den Browser



# TypeScript



# TypeScript

- Package `typescript`
  - <https://www.npmjs.com/package/typescript>
  - Darin enthalten der "Compiler" `tsc`
- Benutzung
  - Globale Installation
    - `tsc src.ts`
  - Lokale Installation im Projekt
    - In der `package.json`

```
"scripts": {  
  "compile": "tsc *.ts",  
  ... }  
"
```
    - `npm run compile`
- Erzeugt werden vom Browser interpretierbare JavaScript-Dateien



# TypeScript: Beispiel

```
people.ts
1 class Person {
2   fullName: string;
3   constructor(public firstName, public lastName) {
4     this.fullName = firstName + " " + lastName;
5   }
6 }
7
8 function greet(person : Person) {
9   return "Hello, " + person.firstName + " " + person.lastName;
10 }
11
12 var user = new Person("Georg", "Metzger");
13
14 document.body.innerHTML = greet(user);
15
```

TypeScript

```
people.js
1 var Person = /** @class */ (function () {
2   function Person(firstName, lastName) {
3     this.firstName = firstName;
4     this.lastName = lastName;
5     this.fullName = firstName + " " + lastName;
6   }
7   return Person;
8 }());
9 function greet(person) {
10   return "Hello, " + person.firstName + " " + person.lastName;
11 }
12 var user = new Person("Georg", "Metzger");
13 document.body.innerHTML = greet(user);
14
```

Transpiliertes  
JavaScript





# Continuous Build



## Möglichkeiten des npm-script-Objekts

- Bisher wurde das `script`-Objekt der `package.json` nur für die Definition einzelner einfacher Skript-Aufrufe benutzt
- Es ist jedoch noch mehr möglich:
  - `pre` und `post`-Skripte
    - Diese werden vor bzw. nach dem Skript aufgerufen
    - Identifikation über Namenskonvention
      - `pre<script-name>` und `post<script-name>`
  - `npm` definiert einen Satz von Standard-Kommandos, die direkt, ohne `run`, aufgerufen werden können
    - z.B. `npm start`
  - Parallelisierung



## Parallelisierung mit concurrently

- Dazu wird das Package `concurrently` installiert
  - `npm install concurrently --save-dev`
- Ab nun kann als Script eine Liste von Skripten definiert werden, die parallel gestartet werden
  - `"start": "concurrently \"npm run babel-transpile\" \"npm run serve\""`
  - Beide hier gestarteten Skripte laufen permanent und überwachen ihre jeweiligen Ressourcen
    - Damit wird jede Änderung von Dateien
      - `.html`
      - `.js`
      - `.es`
  - registriert und durch Browsersync dargestellt



# Lint



## Lint: Definition

- Ein "Linter" prüft den Quellcode auf
  - die Einhaltung von Konventionen
  - die Umsetzung von Programmier- und Design-Richtlinien
  - die Verwendung von "Best Practices"
- Ursprünglich als Precompiler für C-Programme
  - `lint`
- Mittlerweile für viele Programmiersprachen verfügbar
  - allerdings nicht immer als Linter
    - Checkstyle oder SonarJ im Java-Umfeld
- Im JavaScript-Umfeld übernehmen Linter auch syntaktische Prüfungen
  - und damit typische Aufgaben eines Compilers
  - Beispiele
    - JSLint
    - JSHint



# JSLint

- Installation als `npm`-Package
  - `npm install jshint --save-dev`
- Konfiguration über `jshint.conf`
  - Einstieg über die `jshint.conf.example` der Distribution
  - Dokumentation unter <http://www.jshint.com/help.html>
- Integration in den Build-Prozess durch Script-Eintrag



# Das Projekt mit JSLint

The screenshot shows a code editor with a project structure on the left and a JSLint configuration file on the right.

**Project Structure:**

- javascript-tools
  - .git
  - dist
    - people.js
  - node\_modules
  - src
    - index.html
    - lintdemo.js
    - people-es.html
    - people.es
- .babelrc
- .gitignore
- bs-config.json
- jslint.conf (selected)
- package-lock.json
- package.json

**jslint.conf:**

```
1 {
2   "evil": false,
3   "indent": 2,
4   "vars": true,
5   "passfail": false,
6   "plusplus": false,
7   "predef": ["module", "require"]
8 }
```

The status bar at the bottom indicates: jslint.conf 1:1, LF, UTF-8, Plain Text, git+, master, 0 files.



# Webpack





# Node-Anwendungen im Browser

- Node-Anwendungen sind im Browser nicht lauffähig
  - Kein Filesystem-Zugriff
  - Keine Socket-Programmierung
  - Kein `require`
- Das Modul-Konzept ist jedoch auch auf Browser-Seite sinnvoll!
- Bestrebungen, ein Modul-Konzept im Browser einzuführen, sind im Gange
  - `import` und `export` von ES2015
- Aktuell werden während der Entwicklung die node-Programme für den Browser umgewandelt
  - Technisch werden alle `require`-Befehle analysiert und die Skripte zu einem Ganzen vereint
- Webpack ist hierfür ein etabliertes Framework
  - <https://webpack.js.org/>



# Installation und Konfiguration

- `npm install webpack --save-dev`
- Konfiguration in `webpack.config.js`

- Beispiel

```
const path = require('path');

module.exports = {
  entry: './src/prototype_demo.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```



# webpack

- <https://webpack.js.org/>
- **Installation als npm Package**
  - `npm install webpack --save-dev`



# Testen



Jasmine



Karma



Browser



# Jasmine



# Jasmine: Überblick

- <https://jasmine.github.io>
- Jasmine ist ein "Behaviour Driven Testing Framework"
  - [https://de.wikipedia.org/wiki/Behavior\\_Driven\\_Development](https://de.wikipedia.org/wiki/Behavior_Driven_Development)
- Tests sind unabhängig von einer Browser-Umgebung
- Klare Syntax
  - `describe` eröffnet eine neue Test-Suite
  - `it` startet eine Test-Spezifikation
  - `expect` formuliert innerhalb einer Spezifikation eine Annahme
  - Matchers vergleichen einen erwarteten Ist-Wert mit dem aktuellen Ergebnis eines Algorithmus
    - `toBe`
    - `toBeLessThan`
    - `toBeNull`
    - `not` zur Negation eines Matchers



## Jasmine: Installation und Benutzung

- `npm install jasmine --save-dev`
- Initialisierung mit `./node_modules/.bin/jasmine init`
  - Dadurch wird ein neues Verzeichnis spec angelegt
  - Darin die jasmine.json-Konfiguration
- Eintragen in die package.json
  - `"test": "jasmine"`
- Aufrufen der Tests nun mit `npm test`
  - Aktuelle Tests sind natürlich noch nicht vorhanden



## Ein trivialer Test

```
describe("A suite", function() {  
  it("contains spec with a valid expectation",  
    function() {  
      expect(true).toBe(true);  
    });  
  it("contains spec with an invalid expectation",  
    function() {  
      expect(true).toBe(false);  
    });  
});
```





# Jasmine: Ergebnis einer Suite

```
Failures:
1) A suite contains spec with an invalid
   expectation
   Message:
     Expected true to be false.
...

2 specs, 1 failure
```



# Karma



# Karma: Installation und Bedienung

- `npm install karma`
- Interaktive Initialisierung mit
  - `./node_modules/karma/bin/karma init karma.js`
- Abgefragt wird
  - Das zu verwendende Testing-Framework
    - jasmine
  - Wird Require.js benutzt?
    - Optional, Standard ist "no"
- Welche Browser sollen automatisch gestartet werden ?
- Lokation der JavaScript- und Test-Dateien
- Anschließend Exclusions
- Überwachung der Dateien, automatische Testausführung bei Änderungen



## Karma: Arbeitsweise

- Karma startet einen Web Server
- Karma startet den oder die angegebenen Browser
- Karma generiert eine Test-HTML-Seite
  - Diese lädt alle angegebenen Skripte
  - Weiterhin werden die Tests im Browser ausgeführt
- Die Ergebnisse werden dem Web Server zurück übermittelt
  - Standard-mäßige Protokollierung auf der Konsole



# Browser



## Der Phantom-Browser

- Eine andere Möglichkeit ist die Benutzung des Phantom-Browsers
  - <http://phantomjs.org/>
  - Im Wesentlichen ein Browser ohne User Interface
- Insbesondere wichtig für automatisierte Tests:
  - Der Start des Phantom-Browsers ist deutlich schneller als der Start eines vollwertigen Browsers
  - Weiterhin können die Tests auch auf Maschinen ohne grafisches Betriebssystem ausgeführt werden



# Fortgeschrittene Programmierung



Klassen



Promises



Scoped Variables und  
Konstanten



Collections



Vereinfachte  
Funktionsdeklaration



Generators und Proxies



# Klassen





# JavaScript und 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!



# Einfache Klassen

```
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;
    }
}
```



# Vererbung

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



# Scoped Variables und Konstanten



# let und const

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



# Collections



# Map

- 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()) {}
```



# Set

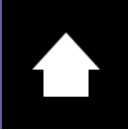
- Eine Set besteht aus Unikaten
  - In anderen Sprachen als Dictionary oder assoziatives Array bezeichnet

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





# Vereinfachte Funktionsdeklaration



# Arrow-Syntax für Funktionen

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

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



# Generators und Proxies



# Generators

- 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

- 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);
```



# Promises



# Was sind Promises?

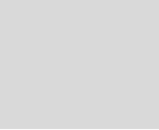
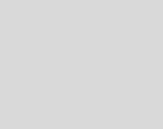
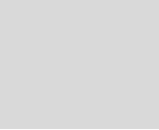
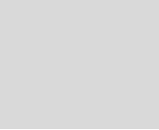
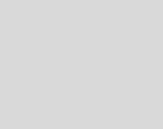
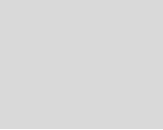
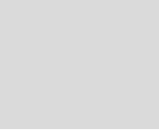
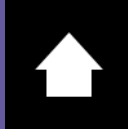
- Promises sind Objekte, die ein potenziell zukünftiges Ergebnis liefern
  - "Ein Versprechen auf die Zukunft"
    - Das Ergebnis kann auch eine Fehlerstruktur sein
- Promise-Objekte halten einen Zustand:
  - Fulfilled
    - Ein Ergebnis konnte bestimmt werden
  - Rejected
    - Es wurde ein Fehler festgestellt
  - 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: Benutzung

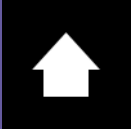
- Promises werden im Programm so benutzt, als wäre das Ergebnis bereits bekannt
  - Dem Promise-Objekt werden
    - success
    - error
    - und optional progress-Funktionen zugefügt





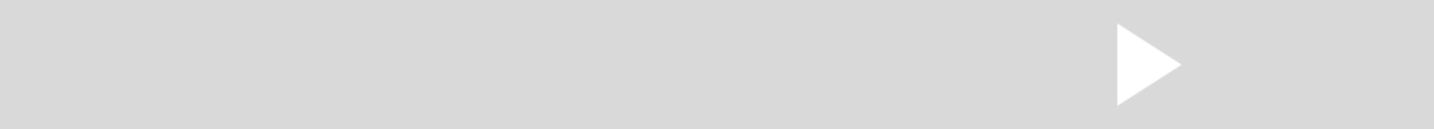
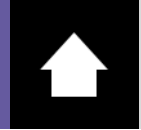
# Das Promise-API

- 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



# Beispiel: Promise

```
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()))
```



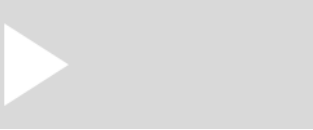
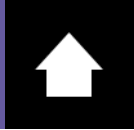
# async await

- 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



# Beispiel: async await

```
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');
```



# Workshop



Vorgabe



Vorgehensweise



# Vorgabe



# Workshop: Server

- Für den Workshop wird ein fertiger Server zur Verfügung gestellt
- Dieser realisiert einen RESTful Web Service zur Verwaltung von Personen
  - Struktur der Person
    - `id`, eine Ganzzahl
    - `firstname` und `lastname`, Zeichenketten
    - `gender`, ein Einzelzeichen
    - `height`, eine Ganzzahl
  - Operationen
    - POST zum Anlegen
    - GET für Leseoperationen
    - PUT zum Aktualisieren
    - DELETE zum Löschen
- Pfad
  - `people`
    - Anlegen, Aktualisieren, Liste aller vorhandenen Personen
  - `people/{id}`
    - Suche nach ID, Löschen mit ID





## Workshop: Browser-Frontend

- Hier soll ein simples User Interface abgebildet werden, dass dem Benutzer die vom Server realisierten Services zur Verfügung stellt
  - Dieses User-Interface muss definitiv nicht hübsch gestaltet werden
  - Ob mehrere Seiten benutzt werden oder alles auf einer Seite dargestellt wird bleibt dem Entwickler vorbehalten



# Vorgehensweise



## Schritt für Schritt

- Es wird `npm` benutzt
- Als Sprache wird JavaScript benutzt, allerdings sollen ECMA6+-Features genutzt werden können
  - Babel als Transpiler
- Es wird ein agiler Entwicklungsprozess benutzt werden
  - Lite Server
- Zur Client-Server-Kommunikation wird prototype eingesetzt
  - Hier insbesondere die AJAX-Komponente
- Begleitend zur Implementierung sollen Tests implementiert werden
  - Jasmine
  - Optional mit Karma