

Angular 2

Gesamtinhaltsverzeichnis

1	Einführung	3
1.1	Installation und Setup	3
1.2	Ein erstes Projekt	4
2	Ein Angular Module	8
2.1	Dateien im Detail	8
2.2	Einfache Erweiterungen	11
3	UI-Programmierung	13
3.1	Data Binding	13
3.2	Events	13
4	Angular im Detail	15
4.1	Components	15
4.2	Services.....	16
4.3	Angular Modules.....	19
4.4	Direktiven	20
4.5	Pipes	21
5	Client-Server-Kommunikation	23
5.1	Exkurs REST	23
5.2	Der HttpClient.....	26
5.3	Routing und Navigation	27
6	Anhang.....	29
6.1	node.js.....	29
6.2	npm – Der Node Package Manager	31
6.3	Node-Modules	32
6.4	Einrichten von Typescript	36
6.5	Grundlagen der Programmierung.....	37
7	Stichwortverzeichnis	47
8	Weitere Informationen	51
8.1	Einige Hinweise	51
8.2	Literatur und Quellen	52

1 Einführung

1.1 Installation und Setup

1.1.1 JavaScript-Grundinstallation

- `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

1.1.2 Installation des Angular Command Line Interfaces

- `npm install -g @angular/cli`
- Damit ist der Angular-Projektassistent `ng` installiert
 - Anlegen eines neuen Projekts
 - `ng new <projektname>`
 - Erzeugen neuer Angular-Programmteile
 - Services
 - Direktiven
 - ...

Hinweis:

- Das Anlegen eines Angular-Projektes kann natürlich auch ohne dieses Werkzeug erfolgen

1.1.3 Kurzübersicht der ng-Kommandos

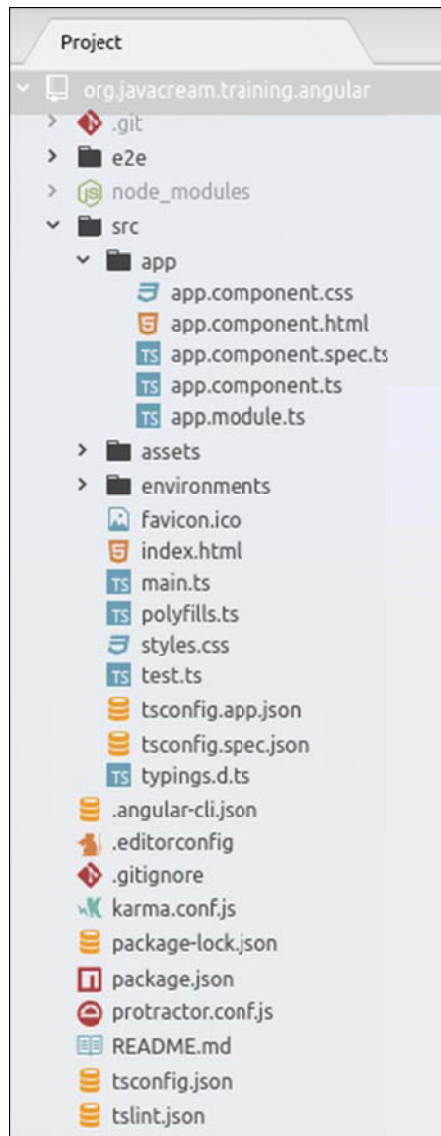
- Development server
 - `ng server` startet einen Server mit Browser-Sync auf <http://localhost:4200/>
- Code Scaffolding
- `ng generate component component-name`
 - `directive` | `pipe` | `service` | `class` | `guard` | `interface` | `enum` | `module`
- Build
 - `ng build <-prod>`
 - Artefakte werden in `dist/` abgelegt
- Unit Tests
 - `ng test` führt Karma-Tests aus
 - <https://karma-runner.github.io>
- End-to-End Tests
 - `ng e2e` führt Protractor-Tests aus
 - <http://www.protractortest.org/>

1.2 Ein erstes Projekt

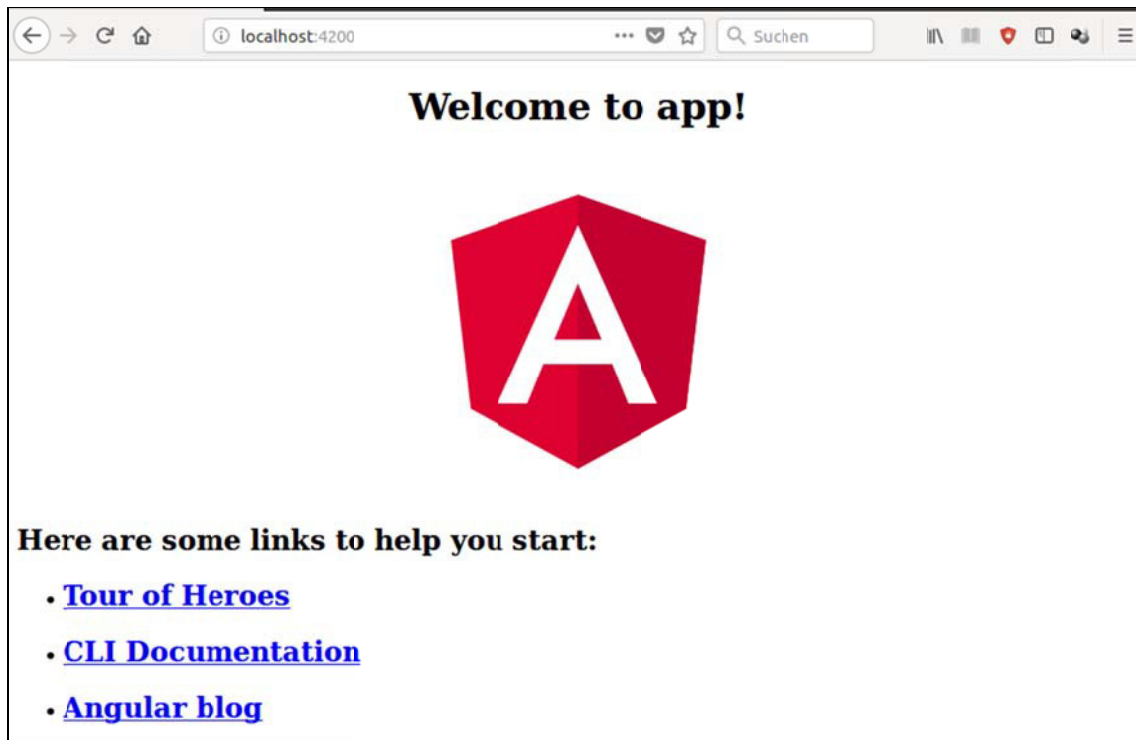
1.2.1 Anlegen des Projekts

- `ng new org.javacream.training.angular`
- Damit wird ein node-Projekt erzeugt
- Ebenso werden automatisch alle Abhängigkeiten installiert
 - Insbesondere TypeScript
 - Näheres zu TypeScript im Anhang

1.2.2 Projektstruktur



1.2.3 Die Anwendung unter localhost:4200



1.2.4 Entwicklungsprozess

- Jede Änderung an den Quellen führt zur Browser-Aktualisierung
- Beispiel:
 - Ändern des Seitentitels sowie der Willkommens-Nachricht
 - `src/index.html`
 - `src/app/app.component.html`
 - `src/app/app.component.ts`

1.2.5 Die aktualisierte Seite



2 Ein Angular Module

2.1 Dateien im Detail

2.1.1 Übersicht der Dateien eines Moduls

- `src/index.html`
 - Die Hauptseite
- `src/app/app.module.ts`
 - Definition des Moduls
- `src/app/app.component.ts`
 - Definition des dynamischen Anteils einer Komponente mit TypeScript
- `src/app/app.component.html`
 - Definition des HTML-Templates einer Komponente
- `src/app/app.component.css`
 - Stylesheet der Komponente
- `src/app/app.component.spec.ts`
 - Definition von Karma-Tests

2.1.2 Angular-Module: Die Index-Seite



The screenshot shows the `index.html` file in a code editor. The code is as follows:

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>org.Javacream.Training.Angular</title>
6   <base href="/">
7
8   <meta name="viewport" content="width=device-width, initial-scale=1">
9   <link rel="icon" type="image/x-icon" href="favicon.ico">
10 </head>
11 <body>
12   <app-root></app-root>
13 </body>
14 </html>
15
```

An annotation box with the text "Der Einstiegspunkt für Angular" (The entry point for Angular) has an arrow pointing to the `<app-root></app-root>` tag on line 12.

2.1.3 Angular-Module: Die Component



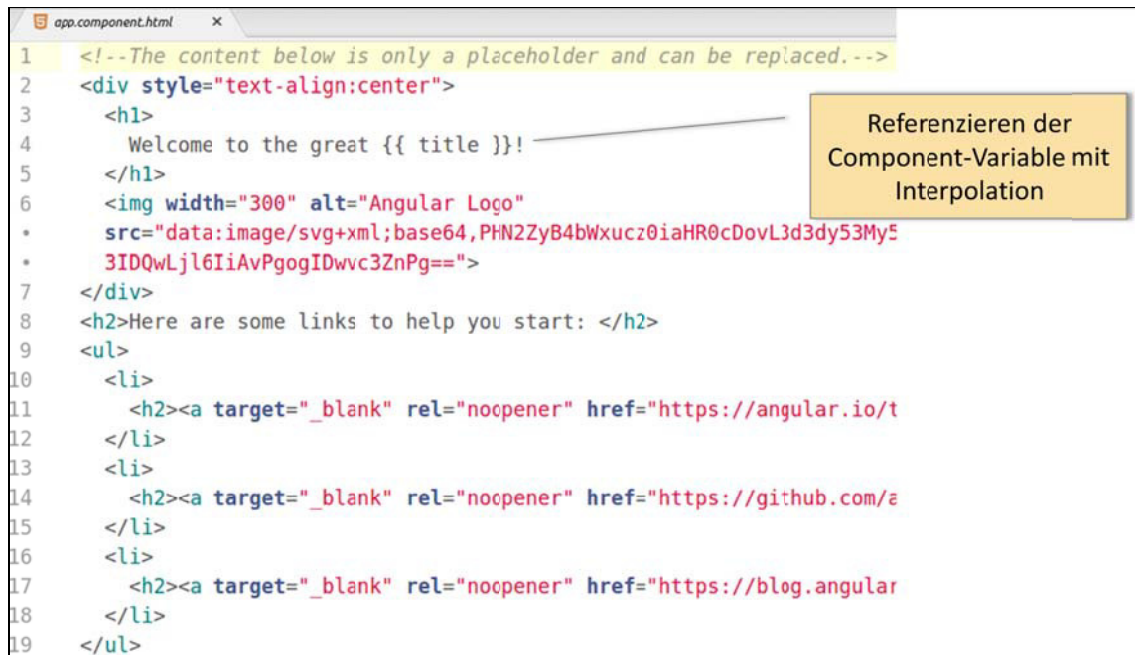
The screenshot shows the `app.component.ts` file in a code editor. The code is as follows:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'Javacream App';
10 }
11
```

Four annotations are present on the right side of the code editor, each with an arrow pointing to a specific line of code:

- "TypeScript-Import der Angular-Core-Bibliothek" points to line 1: `import { Component } from '@angular/core';`
- "Referenz auf den Einstiegspunkt" points to line 4: `selector: 'app-root',`
- "Referenz auf Template und CSS" points to line 5: `templateUrl: './app.component.html',`
- "Definition einer Variablen und Wert-Zuweisung" points to line 9: `title = 'Javacream App';`

2.1.4 Angular-Module: Das HTML-Template



The screenshot shows a code editor with a file named `app.component.html`. The code is as follows:

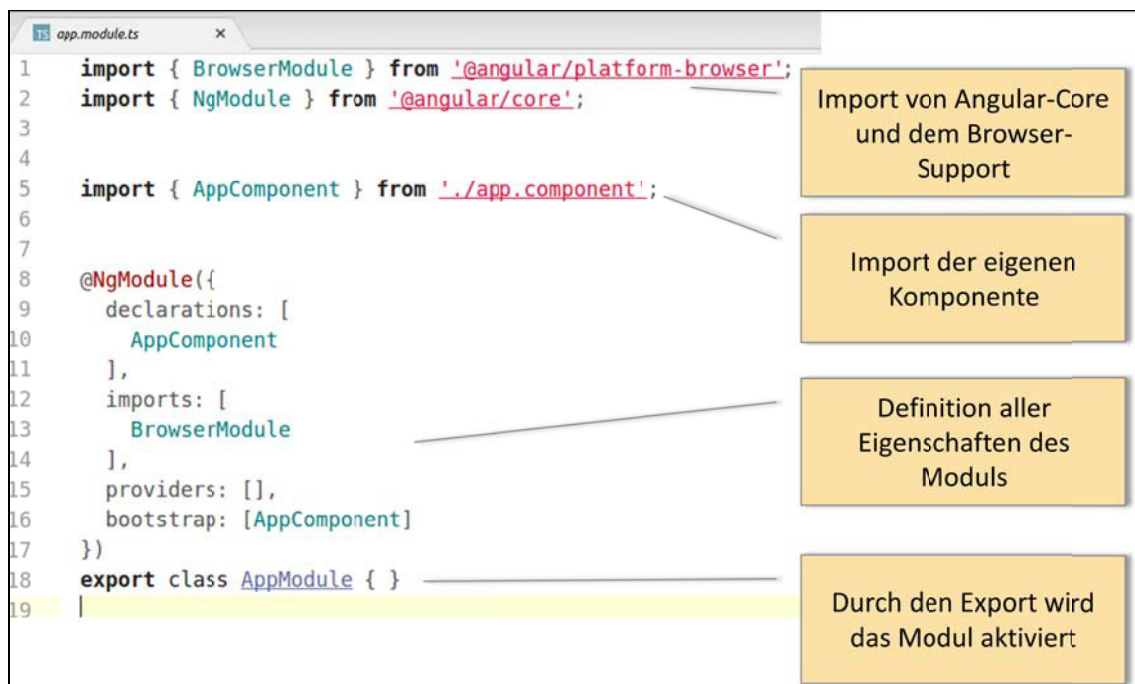
```

1  <!--The content below is only a placeholder and can be replaced.-->
2  <div style="text-align:center">
3    <h1>
4      Welcome to the great {{ title }}!
5    </h1>
6    
8  </div>
9  <h2>Here are some links to help you start: </h2>
10 <ul>
11   <li>
12     <h2><a target="_blank" rel="noopener" href="https://angular.io/t
13   </li>
14   <li>
15     <h2><a target="_blank" rel="noopener" href="https://github.com/a
16   </li>
17   <li>
18     <h2><a target="_blank" rel="noopener" href="https://blog.angular
19   </li>
20 </ul>

```

An annotation box on the right points to the `{{ title }}` interpolation in line 4, stating: "Referenzieren der Component-Variable mit Interpolation".

2.1.5 Angular-Module: Module-Deklaration



The screenshot shows a code editor with a file named `app.module.ts`. The code is as follows:

```

1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4
5  import { AppComponent } from './app.component';
6
7
8  @NgModule({
9    declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule
14   ],
15   providers: [],
16   bootstrap: [AppComponent]
17 })
18 export class AppModule { }
19

```

Annotations on the right explain the code:

- Lines 1-2: "Import von Angular-Core und dem Browser-Support"
- Line 5: "Import der eigenen Komponente"
- Lines 8-16: "Definition aller Eigenschaften des Moduls"
- Line 18: "Durch den Export wird das Modul aktiviert"

2.2 Einfache Erweiterungen

2.2.1 Mehrere Komponenten

- Auf einer Seite können auch mehrere Komponenten dargestellt werden
- Diese sind komplett voneinander isoliert
 - Der Datenaustausch zwischen verschiedenen Komponenten wird später beschrieben

2.2.2 Export von Benutzerdefinierten Typen

- Eine Komponente kann auch eigene Datentypen definieren und Exportieren
 - Ein Export ist nicht notwendig, so lange der Typ nicht anderen Modulen bekannt gemacht werden muss
- Ebenso können die Eigenschaften der Komponente komplexe Objekte sein
- Der Zugriff auf die Eigenschaften des komplexen Objekts erfolgen über die normale JavaScript-Syntax
 - Also über den `.`-Operator
 - Aber Vorsicht!
 - Die Deklarativen unterstützen nicht die komplette Syntax!
 - `if`
 - `for`
 - Deklarationen

2.2.3 Auflistungen

- Innerhalb der `{{ }}`-Interpolation wird das `for`-Konstrukt nicht unterstützt
- Allerdings gibt es eine spezielle `forEach`-Anweisung

```
<ul>
<li *ngFor="let person of people">
  <span>{{person.firstname}}</span> {{person.lastname}}
</li>
</ul>
```


3 UI-Programmierung

3.1 Data Binding

3.1.1 Die ngModel-Direktive

- Bisher wurden die Werte der Komponente mit der `{{ }}`-Interpolation dargestellt
- Dieses Binding ist dynamisch (!)
 - Änderungen der Werte werden in der Oberfläche dargestellt
- Eigenschaften der Komponente können auch bidirektional gebunden werden
- Dazu dient die Direktive `ngModel`
 - Notwendig hierzu ist der Import des `FormsModule` aus `@angular/form`

3.1.2 Data Binding: Beispiel

```
<div style="text-align:center">
  <h2>Ui</h2>
  Mutable Data: {{data.mutableData}}
  <input type="text" [(ngModel)]="data.boundData"/>
</div>
```

3.2 Events

3.2.1 Event Handler

- Im Gegensatz zum Standard-Eventmodell im Browser können beliebige Funktionen mit Parametern als Handler aufgerufen werden
- Die Event-behandelnde Funktion ist Bestandteil der `Component`

3.2.2 Event Handler: Beispiel

```
export class UiAppComponent {  
    //...  
    selectionList: string[] =  
        ["Item1", "Item2", "Item3"]  
    setItem(item: string){  
        this.data.bindData = item  
    }  
}  
  
<ul>  
  <li *ngFor="let item of selectionList"  
    (click)="setItem(item)">  
    {{item}}</li>  
</ul>
```

4 Angular im Detail

4.1 Components

4.1.1 Component-Annotation

- Eine Component ist eine TypeScript-Klasse mit der `@Component`-Annotation
 - Annotationen enthalten Metadaten, die der Klasse zugeordnet werden
 - Diese müssen von einem Framework interpretiert werden
- Annotationen sind damit so etwas wie statische Eigenschaften einer Klasse
 - Allerdings sind diese syntaktisch nicht Bestandteil der Klassendeklaration
 - Damit ergibt sich eine saubere Trennung
- Eine typische Component definiert
 - Den Selector als Bezug zum Template
 - Die Lokation des Templates
- Vollständige Liste unter
 - <https://angular.io/api/core/Component>

4.1.2 Aufgabe der Component

- Die Eigenschaften der Component-Klasse dienen als Schnittstelle zum HTML-Code
 - Werte werden über Interpolation oder Direktiven gebunden
 - Funktionen dienen als Event-Handler
- Bei Bedarf implementiert die Component-Klasse Lifecycle-Funktionen
 - `ngOnChanges`
 - `ngOnInit`
 - <https://angular.io/guide/lifecycle-hooks>

4.1.3 Input einer Component

- Components können über das Template verschachtelt werden
 - Innerhalb des Templates wird ein Root-Element einer weiteren Angular-Component definiert
- Der Kind-Controller kann Input-Parameter definieren
 - Dazu wird eine Eigenschaft mit der `@Input`-Annotation versehen
- Diese werden im Template gesetzt

4.1.4 Beispiel: Input einer Child-Component

The screenshot shows an Angular component file named `app.component.html` and its corresponding class `PersonDetailComponent`. The template contains a `<div>` with `text-align:center` style, containing an `<h2>` and a ``. Inside the ``, there is a `<person-child-root>` element with `*ngFor="let person of people"` and a binding `[person]= person`. The class `PersonDetailComponent` has an `@Input()` property named `person` of type `Person`. Annotations on the right side explain the code:

- Neues Root-Element für die Kind-Component**: Points to the `<person-child-root>` element in the template.
- Angabe des Root-Elements**: Points to the `selector: 'person-child-root'` in the `@Component` decorator.
- Deklaration einer Input-Eigenschaft**: Points to the `@Input() person: Person;` in the class.

4.2 Services

4.2.1 Eigenschaften eines Services

- Services sind TypeScript-Klassen
- Diese werden ausschließlich vom Angular-Framework instanziiert
 - Diese Aufgabe übernimmt der Angular-Context
- Benötigt eine Component den Zugriff auf einen Service, so wird dieser automatisch gesetzt
 - Dieses Verfahren heißt Dependency Injection
- Insgesamt stellt Angular mit dem Service-Mechanismus ein Dependency-Injection-Framework zur Verfügung

4.2.2 Aufgaben eines Services

- Standard-Services des Angular-Frameworks stellen nützliche Routinen zur Verfügung
 - Beispielsweise wird die Client-Server-Kommunikation über den `HttpClient`-Service realisiert
- Eigene Services werden in der Anwendung häufig benutzt, um in einer sauberen Architektur ein Datenmodell zu definieren
 - Dieses wird dann von allen Components konsistent benutzt
 - Services ermöglichen damit eine Inter-Component-Kommunikation, die unabhängig von Input-Parametern sind
 - Wesentlich leichter wartbar

4.2.3 Ein simpler Service

- Die Erzeugung eines Services kann das Angular-CLI übernehmen
- `ng generate service <ServiceName>`
- Ein Service selbst ist eine `@Injectable`-Klasse
 - Damit kann diese Klasse im Konstruktor jeder anderen Angular-Klasse benutzt werden
 - Components
 - Andere Services
- Service-Klassen müssen als `provider` deklariert werden
 - Eigenschaft des Angular-Moduls oder
 - Eigenschaft des `@Component`-Annotation
 - Alternativen zur Deklaration der Klasse unter <https://angular.io/guide/dependency-injection#providers>

4.2.4 Beispiel: Service

```
1  import { Injectable } from '@angular/core';
2
3  @Injectable()
4  export class PeopleModelService {
5
6      constructor() { }
7      people: Person[] = [new Person("Gärtner", "Hans")]
8
9  }
10
11  providers: [
12      PeopleModelService
13  ],
14
15  export class ServiceAppComponent {
16      constructor(readonly peopleModel: PeopleModelService ){
17
18      }
19      people: Person[] = this.peopleModel.people
20  }
```

@Injectable macht diese Klasse dem Angular-Context bekannt

Im Modul oder der @Component-Annotation

Dependency Injection in einer Component

4.3 Angular Modules

4.3.1 Aufgaben des Moduls

- Das Angular-Module enthält die globale Konfiguration der Anwendung
- Dazu wird eine Klasse mit `@NgModule` annotiert
- Diese Annotation umfasst als Eigenschaften
 - `declarations`
 - Components, Directives, Pipes
 - Directives und Pipes werden später behandelt
 - `imports`
 - Definiert, welche Direktiven und Pipes den Templates des Modules zur Verfügung gestellt werden
 - `providers`
 - Alle `@Injectables`
 - insbesondere Services
 - `bootstrap`
 - Alle Elemente, die beim Start der Anwendung initialisiert werden müssen
 - Also insbesondere alle Components
 - Aber keine Kind-Komponenten
- Vollständige Dokumentation unter <https://angular.io/api/core/NgModule>

4.4 Direktiven

4.4.1 Direktiven

- Direktiven sind Erweiterungen von HTML, die vom Angular Framework gesucht und interpretiert werden
 - Während des Bootstrap-Vorgangs wird die Startseite der Anwendung nach Direktiven analysiert
- Erweiterung durch
 - HTML-Elemente
 - Attribute
- Was genau beim Auftreten einer Direktive passiert hängt von der Direktiven-Implementierung ab
 - `ngModel`
 - Data Binding
 - `ngFor`
 - Iteration

4.4.2 Direktiven des Angular-Frameworks

- Angular unterscheidet die folgenden Direktiven
 - Components sind Direktiven!
 - sie werden an ein HTML-Element gebunden
 - Structural Directives
 - Modifizieren das DOM der Anwendung
 - `ngFor` oder `ngIf` sind Beispiele hierfür
 - Attribute Directives
 - Ändern die Darstellung oder das Verhalten eines HTML-Elements
 - `ngStyle` als Beispiel
- Vollständige Liste unter <https://angular.io/api?status=stable&type=directive>

4.4.3 Eigene Direktiven

- Können simpel erstellt werden
- `ng generate directive <directive-name>`
- Die Direktive ist wiederum eine annotierte TypeScript-Klasse

- Der `selector` ist das neue HTML-Attribute oder -Element
- Im Konstruktor wird das DOM-Element, das manipuliert werden soll, injected

4.4.4 Eigene Directive: Beispiel

- Ein Beispiel ist unter <https://angular.io/guide/attribute-directives> zu finden

```
import { Directive, ElementRef } from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

4.5 Pipes

4.5.1 Was sind Pipes?

- Pipes formatieren das Ergebnis beispielsweise einer Interpolation
 - `{{dateOfBirth | date}}`
 - Hier wird das Geburtsdatum als Datum formatiert dargestellt
 - Pipes sowie der Operator `|` sind aus der Linux-Welt übernommen
- Vordefinierte Pipes umfassen
 - `date[:format[:timezone[:locale]]]`
 - `currency[:currencyCode[:display[:digitInfo[:locale]]]]`
 - `lowercase`
 - Vollständige Liste und Dokumentation unter <https://angular.io/api?status=stable&type=pipe>

5 Client-Server-Kommunikation

5.1 Exkurs REST

5.1.1 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

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

5.1.2 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

5.1.3 Elemente der http-Spezifikation II

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

5.1.4 MimeTypes

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

5.1.5 REST und http

- 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

5.1.6 Mapping REST - http

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

5.1.7 Konzeption eines RESTful Services: Neuanlage

- 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

5.1.8 Konzeption eines RESTful Services: Update

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

5.1.9 Konzeption eines RESTful Services: Delete

- 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`

5.2 Der HttpClient

5.2.1 Der Service HttpClient

- HttpClient ist ein Service
 - und steht damit via Dependency Injection zur Verfügung
- Das API
 - umfasst die Standard-http-Methoden
 - ist asynchron konzipiert

```
http.get('/endpoint').subscribe(data => {  
    // Read the result field from the JSON re-  
    sponse.  
    this.results = data['jsonAttribute'];  
});
```

5.3 Routing und Navigation

5.3.1 Arbeitsweise

- Routen definieren Pfade der Anwendung
- Jede Route verbindet einen Pfad mit einer Component
 - Auch Redirects können definiert werden
 - Die Pfade unterstützen Platzhalter
- Die Routen werden vom Angular-Module erzeugt und stehen für die gesamte Applikation zur Verfügung
 - dazu werden Klassen importiert
 - `import { RouterModule, Routes } from '@angular/router'`
- Routen-Pfade müssen nicht disjunkt sein
 - Der erste Treffer eines Pfads wird benutzt
 - Damit sind in der Routen-Definition speziellere Pfade vor allgemeinen zu platzieren

5.3.2 Beispiel: Routen-Definition

```
const appRoutes: Routes = [  
  { path: 'path1', component: Component1 },  
  { path: ' ', redirectTo: '/index', pathMatch: 'full' },  
  { path: '**', component: PageNotFoundComponent }  
];  
  
@NgModule({  
  imports: [  
    RouterModule.forRoot(  
      appRoutes,  
      { enableTracing: true } // <-- debugging purposes only  
    )  
  ],  
  ...  
})  
  
export class AppModule { }
```

5.3.3 Verlinken von Routen

- Dazu gibt es die `routerLink`-Direktive
- Die anzuzeigende Seite wird in einem `router-outlet`-Element angezeigt

5.3.4 Details zu den Routen

- Die `ActivatedRoute` kann in eine Component injiziert werden und liefert Zugriff auf die Routen-Definition, die zum Aufruf geführt hat
 - `url`
 - `paramMap`
 - ...
- Router-Events werden an interessierte Listener delegiert
 - `NavigationStart`
 - `RouteRecognized`
 - ...
- Details unter <https://angular.io/guide/router>

6 Anhang

6.1 node.js




6.1.1 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/>

6.1.2 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');
```

6.1.3 Installation: node.js

	LTS Recommended For Most Users		Current Latest Features	
	 Windows Installer <small>node-v6.11.4-x64.msi</small>		 Macintosh Installer <small>node-v6.11.4.pkg</small>	
	 Source Code <small>node-v6.11.4.tar.gz</small>			
Windows Installer (.msi)	32-bit	64-bit		
Windows Binary (.zip)	32-bit	64-bit		
macOS Installer (.pkg)	64-bit			
macOS Binaries (.tar.gz)	64-bit			
Linux Binaries (x86/x64)	32-bit	64-bit		
Linux Binaries (ARM)	ARMv6	ARMv7	ARMv8	
Source Code	node-v6.11.4.tar.gz			
Additional Platforms				
SunOS Binaries	32-bit	64-bit		
Docker Image	Official Node.js Docker Image			
Linux on Power Systems	64-bit le	64-bit be		
Linux on System z	64-bit			
AIX on Power Systems	64-bit			

6.1.4 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*

6.1.5 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

6.2 npm – Der Node Package Manager

6.2.1 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

6.2.2 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/>

6.3 Node-Modules

6.3.1 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

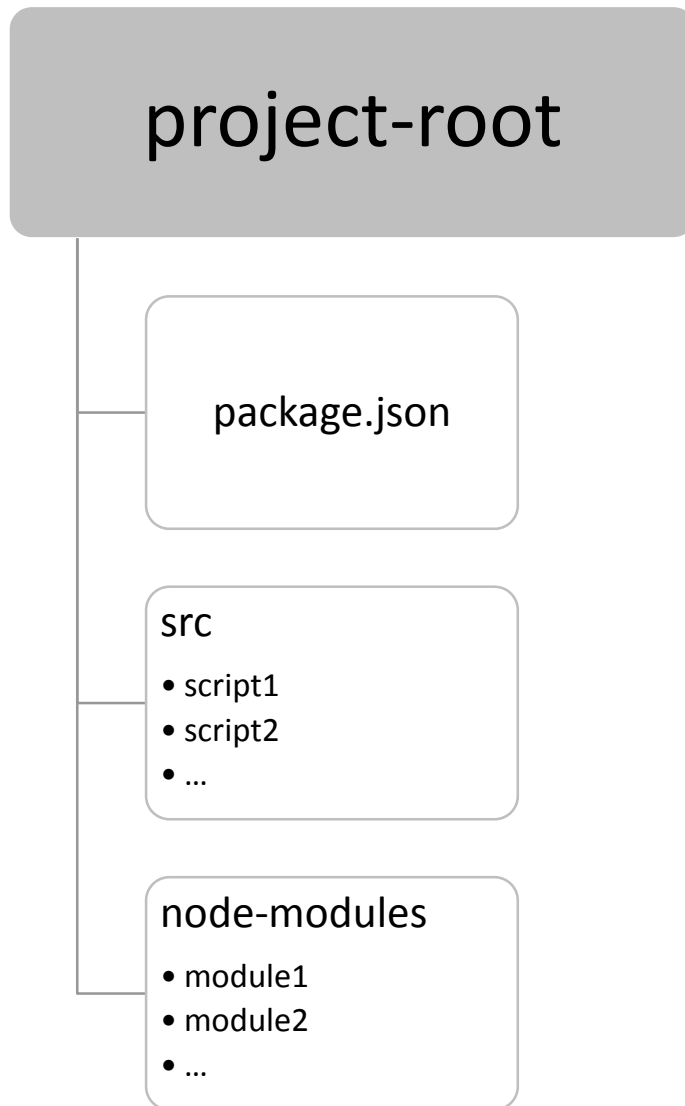
6.3.2 Die package.json

- 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

6.3.3 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

6.3.4 Projektstruktur



6.3.5 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"
}
```

6.3.6 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()
```

6.3.7 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

6.4 Einrichten von Typescript

6.4.1 Benötigte Komponenten

- TypeScript
 - `npm install typescript --save-dev`
- Lite-Server
 - `npm install lite-server --save-dev`
- Parallelisierung von npm-Kommandos
 - `npm install concurrently --save-dev`

6.4.2 Konfiguration des TypeScript-Compilers

- Initialisierung mit `tsc --init`
 - Erzeugt die Datei *tsconfig.json*
 - Darin werden alle möglichen Konfigurationen angelegt
 - wobei die allermeisten auskommentiert sind
- Für die folgenden Beispiele wird insbesondere der strict-Mode aktiviert

6.4.3 Scripts der package.json

```
"scripts": {  
  "serve": "lite-server",  
  "compile": "tsc --outDir ./dist -p .",  
  "compile-watch": "tsc -w --outDir ./dist -p .",  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "start": "concurrently \"npm run compile-watch\" \"npm run  
serve\""  
}
```

6.4.4 Benutzung

- Starten mit `npm start`
 - Startet den TypeScript-Transpiler
 - Startet den Lite Server mit Browser-Sync
 - Startet den Default-Browser und stellt die `index.html`-Seite dar
- Parallelisierung
 - Sämtliche Dateien mit der Endung `.ts` im Ordner `src` werden automatisch nach `dist` transpiliert
 - Änderungen der `.ts`-Dateien werden automatisch erkannt
 - Der Server aktualisiert den Browser mit den geänderten Informationen

6.5 Grundlagen der Programmierung

6.5.1 TypeScript ist JavaScript

- Jegliche JavaScript-Anweisung ist valides TypeScript

```
var message = "Hello World!"  
function printout(s) {  
  console.log("Hello World!")  
}  
printout(message)
```

- Syntaktische Fehler werden jedoch vom TypeScript-Compiler erkannt

6.5.2 TypeScript ist nicht ECMAScript

- Die OOP-Konzepte von ECMAScript sind prinzipiell den Konstrukten in TypeScript sehr ähnlich
 - aber nicht identisch
 - Eine ECMA-Klasse mit Attributen ist keine valide TypeScript-Klasse

6.5.3 Unterstützte Operatoren

- TypeScript unterstützt die aus JavaScript bekannten Operatoren
 - Mathematisch
 - Logisch
- Ebenfalls unterstützt wird der Punkt-Operator zum Zugriff auf Eigenschaften eines Objekts
- Schleifen
 - `for`
 - `while`
- Abfragen
 - `if-else`
 - `switch`

6.5.4 Was ist Typisierung?

- Ein Typ definiert einen Satz von Eigenschaften und Funktionen
- Jede Variable hat einen Typen, der sich nach der Deklaration nicht mehr ändern kann
 - Der Compiler prüft dies
 - Statische Typisierung
- Diese Einschränkung ist für Programmierer häufig vorteilhaft
 - Moderne Entwicklungsumgebungen prüfen die Typisierung bereits während der Eingabe
 - Ein Satz typischer Programmierfehler wird damit bereits frühzeitig erkannt
 - Ebenso beschränkt die Typisierung die möglichen Aufrufe auf einer Variablen, so dass die Entwicklungsumgebung Vorschläge unterbreiten kann
 - Code Assists vermindern damit die Tipparbeit gewaltig

6.5.5 Deklaration von Variablen

```
let | const<name>  
const name  
let state
```

6.5.6 Basis-Typen in TypeScript

- `boolean`
 - Ein logischer Wert, also `true` oder `false`
- `number`
 - Eine Ganz- oder Kommazahl
- `string`
 - Eine Zeichenkette

6.5.7 Typisierte Deklaration von Variablen

- 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
```

- Contextual Type
 - Auch bei Zuweisungen versucht TypeScript, untypisierte Deklarationen zu erkennen

```
window.onmousedown = function(mouseEvent) {  
    console.log(mouseEvent.button); //<- Error,  
};
```

6.5.8 Type Assertions

- Umwandlung des `any`-Typen in einen speziellen Typen

```
const value :any  
let message:string = <string>value  
let message2 : string = value as string
```


6.5.9 Container

- `array`
 - Eine Liste
- `tuple`
 - Eine feste Menge von anderen Basis-Typen
- `enum`
 - Eine feste Menge von Werten

6.5.10 Spezielle Typen

- `null`
 - Eine Eigenschaft ist nicht gesetzt
- `undefined`
 - Eine Eigenschaft oder Funktion ist nicht vorhanden
 - Damit unterschiedlich zu `null`
- `any`
 - Eine untypisierte Variable, die jeden Wert zugewiesen bekommen kann
 - Damit ist bei Bedarf auch eine untypisierte Programmierung auch in TypeScript möglich
- `void`
 - Eine Funktion, die keinen expliziten `return`-Wert aufweist
- `never`
 - Der Rückgabotyp einer Funktion, die kein implizites oder explizites `return`-Statement aufweist
 - Endlose Ausführung oder garantiertes Werfen einer `Exception`

6.5.11 Namespaces

- Namespaces gruppieren Deklarationen
 - Diese sind nur innerhalb des Namespaces direkt ansprechbar
 - Damit wird die Wahrscheinlichkeit von Namenskollisionen vermieden
- Deklarationen werden mit Hilfe des Schlüsselworts `export` anderen Namespaces zur Verfügung gestellt
- Aus einem anderen Namespace müssen die Variablen mit dem Namespace angesprochen werden
 - "Qualifizierte Namen"

6.5.12 Beispiel: Namespaces

```
namespace Namespace1{  
    export let message = "Hello from namespace1"  
}  
  
namespace Namespace2{  
    console.log(Namespace1.message);  
}
```

6.5.13 Module

- Ein Modul exportiert Deklarationen auf Top-Level-Ebene
- Exportierte Deklarationen können importiert werden
- Zur Unterstützung von Modulen unterstützt der TypeScript-Compiler unterschiedliche Optionen:
 - ES
 - commonjs

6.5.14 Interface: Definition

- Ein TypeScript-Interface definiert eine Signatur bestehend aus Eigenschaften
 - Einfache Attribute
 - Funktionen
- Eigenschaften können Optional sein
 - An den Namen der Eigenschaft wird ein `?` ergänzt
- Unveränderbare Eigenschaften werden mit `readonly` deklariert
- Das Interface wird als Typ benutzt
 - Das hierfür benutzte Objekt muss der Struktur des Interfaces entsprechen
 - Dies prüft der Compiler

6.5.15 Interface: Beispiel

```
interface Person{
    lastname: string
    readonly firstname: string
    address?: string
    formattedName():string
}

let p:Person = {
    lastname: "Sawitzki",
    firstname: "Rainer",
    formattedName: function(){
        return this.firstname + " " + this.lastname
    }
}
```

6.5.16 Interfaces: Vererbung

- Interfaces können in einer Vererbungshierarchie benutzt werden
 - Schlüsselwort `extends`
- Das Sub-Interface erbt die Struktur des Super-Interfaces

6.5.17 Interfaces: Beispiel Vererbung

```
interface Worker extends Person{
    company: string
    work(): string
}

let worker:Worker = {
    company: "Integrata",
    lastname: "Sawitzki", firstname: "Rainer",
    formattedName: function(){
        return this.firstname + " " + this.lastname
    },
    work: function(){
        return "working at " + this.company
    }
}
```

6.5.18 Klassen: Benutzerdefinierte Datentypen

- Klassen definieren wie Interfaces eine Struktur
 - Die Attribute einer Klasse
- Im Gegensatz zu Interfaces können Klassen aber auch Funktionen implementieren
 - Die Methoden einer Klasse
- Instanzen einer Klasse werden jedoch durch einen Konstruktor-Aufruf erzeugt
 - Dazu dient der `new`-Operator
 - Der Konstruktor selbst ist eine spezielle Methode ohne Rückgabertyp
 - `constructor(params)`

6.5.19 Klassen: Beispiel

```
class SimplePerson{
  name:string
  height:number
  constructor(name:string, height:number){
    this.name = name
    this.height = height
  }
  sayHello():string{
    return "Hello, my name is " + this.name
  }
}

let simplePerson = new SimplePerson("Mustermann", 188)
console.log(simplePerson.sayHello())
```

6.5.20 Klassen im Detail: Attribute und Methoden

- Methoden können überschrieben werden
 - Eine Subklasse implementiert die selbe Signatur einer Methode wie die Superklasse
 - Die Aufrufe von überschriebenen Methoden werden zur Laufzeit ausgewertet
 - Polymorphie
 - Der Zugriff auf eine Methode der Superklassen-Hierarchie ist mit der Referenz `super` möglich

6.5.21 Klassen im Detail: Kapselung

- TypeScript unterstützt für Attribute und Methoden das Prinzip der Kapselung
 - `public`
 - `protected`
 - `private`

6.5.22 Klassen: readonly

- `readonly`-Attribute sind möglich
- Verkürzter Konstruktor durch "Parameter properties"
 - `constructor(readonly attr:type)` deklariert und setzt ein Attribut

6.5.23 Klassen: getter und setter

- getter- und setter-Methoden
 - Diese definieren ein "Pseudo-Attribut"
 - Beim lesenden oder schreibenden Zugriff werden die korrespondierenden Methoden aufgerufen

6.5.24 Beispiel: getter und setter

```
class PersonWithGetterAndSetter {  
    private _name: string;  
  
    get name(): string {  
        console.log("reading name")  
        return this._name;  
    }  
  
    set name(newName: string) {  
        console.log("setting name")  
        this._name = newName;  
    }  
}  
  
let p = new PersonWithGetterAndSetter ();  
p.name = "Bob Smith";  
console.log(p.name);
```

6.5.25 Klassen: Vererbung

- Auch Klassen unterstützen das Konzept der Vererbung
- Methoden einer Klasse können auch abstrakt sein
 - Analog zu Definition einer Interface-Funktion
 - Eine Klasse, die mit `new` instanziiert werden soll darf keine abstrakten Methoden enthalten
- Ein Interface kann von einer Klasse erben
 - Allerdings darf die Klasse keine nicht-abstrakten Methoden enthalten
- Eine Klasse kann eine Schnittstelle implementieren
 - Schlüsselwort `implements`

7 Stichwortverzeichnis

Angular-Module	
Das HTML-Template	11
Die Component	10
Die Index-Seite	10
Module-Deklaration	11
Anlegen des Projekts	4
Arbeitsweise	27
Aufgabe der Component	15
Aufgaben des Moduls	19
Aufgaben eines Services	17
Auflistungen	12
Basis-Typen in TypeScript	39
Beispiel	
Ein einfaches Node-Module	35
Ein einfaches Projekt	35
Ein kompletter http-Server	30
getter und setter	45
Input einer Child-Component	16
Namespaces	41
Routen-Definition	27
Service	18
Benötigte Komponenten	36
Benutzung	37
Component-Annotation	15
Container	40
Das http-Protokoll	23
Data Binding	
Beispiel	13
Deklaration von Variablen	39
Der Service HttpClient	26
Details zu den Routen	28
Die aktualisierte Seite	7
Die Anwendung unter localhost	
4200	6
Die ngModel-Direktive	13
Die package.json	33
Direktiven	20
Direktiven des Angular-Frameworks	20
Eigene Directive	
Beispiel	21
Eigene Direktiven	20
Eigenschaften eines Services	16
Ein simpler Service	17
Elemente der http-Spezifikation	23
Elemente der http-Spezifikation II	24

Entwicklungsprozess	6
Event Handler	13
Beispiel	14
Export von Benutzerdefinierten Typen	12
Initialisierung eines Projekts	33
Input einer Component	16
Installation	
node.js	30
Installation des Angular Command Line Interfaces	3
Installieren von Abhängigkeiten	35
Interface	
Beispiel	42
Definition	41
Interfaces	
Beispiel Vererbung	42
Vererbung	42
JavaScript-Grundinstallation	3
Klassen	
Beispiel	43
Benutzerdefinierte Datentypen	43
getter und setter	44
readonly	44
Vererbung	45
Klassen im Detail	
Attribute und Methoden	44
Kapselung	44
Konfiguration des TypeScript-Compilers	36
Konzeption eines RESTful Services	
Delete	26
Neuanlage	25
Update	25
Kurzübersicht der ng-Kommandos	4
Mapping REST - http	25
Mehrere Komponenten	12
MimeTypes	24
Module	41
Namespaces	40
Node Modules	32
Node und Browser-basierte Anwendungen	31
npm Kommandos	32
Projektstruktur	5, 34
REST und http	24
Scripts der package.json	37
Spezielle Typen	40
Testen der Installation	31
Type Assertions	39
TypeScript ist JavaScript	37
TypeScript ist nicht ECMAScript	38
Typisierte Deklaration von Variablen	39

Übersicht der Dateien eines Moduls	9
Unterstützte Operatoren	38
Verlinken von Routen	28
Was ist node.js?	29
Was ist npm?	31
Was ist Typisierung?	38
Was sind Pipes?	21

8 Weitere Informationen

8.1 Einige Hinweise

- 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
- Konventionen
 - Befehle werden in `Courier-Schriftart` dargestellt
 - Dateinamen werden in *`Courier-Schriftart`* dargestellt
 - Links werden in unterstrichener `Courier-Schriftart` dargestellt

8.2 Literatur und Quellen

