Zusammenfassung WEBAPP FS2018

Alex Neher

September 26, 2018

Inhalt

1 JavaScript - Basics

Javascript, auch ECMAScript genannt ist eine clientseitige web-Development Sprache für DOM- und CSS-Manipulation, AJAX oder EventHandling. Javascript kann in HTML-Code eingebunden werden, entweder inline über `<script></script>`-Tags, es kann mittels `<script src=path/to/file.js></script>` geholt werden oder direkt über EventHandler `<input type="checkbox" name="options" onchange="order.options.giftwrap = this.checked;">`.

1.1 Variablen definieren

Javascript hat keine Typisierung. Das heisst, Variablen können einfach mit dem Keyword `var` definiert werden, ohne dass ein Datentyp spezifiziert werden muss.

```
var a = 2; // a ist nun eine Nummer
a = 'Jetzt bin ich ein String';
a = false;
```

1.2 Funktionen

In Javascript werden Funktionen als Objekte behandelt. Sie können ebenfalls mit dem Keyword `var` erstellt werden. Jede Funktion hat ihren eigenen Kontext/Scope. Jede Funktion hat per Default zwei Parameter: `this` und `arguments`.

Der `this` Parameter gibt den Kontext der Funktion zurück. Dieser hängt davon ab, wie und wo die Funktion aufgerufen wird. `arguments` ist ein Array, in welchem alle mitgegebenen Argumente speichert.

```
//1. Über anonymes Function Literal
var add = function(a,b){
   console.log(arguments[0]) //gibt den Wert von a zurück
   return a+b;
}

//2. Über Function Literal mit Name
var sub = function sub(a,b){
   return a-b;
}

//3. Über Function Declaration
function mult(a,b){
   return a*b;
}

//4. Über Immediate Function Invocation
var TenDividedByTwo = function(a,b){return a/b;}(10,5);
console.log(TenDividedByTwo) //Output: 5
```

Funktionen können auch direkt in Objekten definiert werden

```
var person = {
   firstName = "Thomas",
   lastName = "Koller",

   printFullName: function(){
      console.log(this.firstName + " " + this.lastName);
   };
}
```

```
9
10 person.printFullName();
```

Mit dem 'apply'-Pattern können auch Funktionen von anderen Objekten aufgerufen werden

```
1  var anotherPerson = {
2     firstName = "Donald",
3     lastName = "Trump"
4  }
5
6  anotherPerson.printFullName() //error: undefined
7
8  aPerson.printFullName.apply(anotherPerson); //Output: Donald Trump
```

Wie bereits erwähnt, werden Funktionen als Objekte behandelt und haben ihren eigenen Kontext/Scope. Das heisst, von ausserhalb kann nicht direkt auf Variablen innerhalb einer Funktion zugegriffen werden, sondern nur über verschachtelte Funktionen. Dieses Konstrukt nennt man **Closure**.

```
1  var myCounter = (function(){
2     var value = 0;
3     return {
4        increment: function(inc){
5           value =+ inc;
6        },
7        getValue: function(){
8           return value;
9        }
10    };
11 }());
12
13 myCounter.increment(10) //value = 10
14 console.log(myCounter.value); //error: undefined
15 console.log(myCounter.getValue()); //10
```

1.3 Objekte

JavaScript kann auch objektbasiert programmiert werden. Es gibt grundsätzlich vier Möglichkeiten, Objekte zu instanziieren:

```
1  //1. über "var"
2  var bachelorModule = {
3     title: "Webapplication Development",
4     instructor: "Thomas Koller"
5  };
6
7  //2. über new und dem default-Konstruktor
8  var bachelorModule = new Object();
9
10 //3. über Object.create()
11 var bachelorModule = Object.create(Object.prototype); //ein leeres Objekt
12
13 //4. mit einem bereits bestehenden Objekt als Prototyp
14 var masterModule = Object.create(bachelorModule) //ist jetzt ein "Klon" von
       bachelorModule
```

Der Zugriff auf Properties funktioniert wie bei anderen objektorientierten Sprachen

```
console.log(bachelorModule.title); //Output: Webapplication Development
console.log(bachelorModule["instructor"]); //Output: Thomas Koller
```

Objekte sind dynamisch. Das heisst, es können zur Laufzeit noch Properties hinzugefügt oder entfernt werden:

```
//hinzufügen von Properties
bachelorModule.credits = 3;

//entfernen von Properties
delete bachelorModule.credits;

//Ebenfalls kann gecheckt werden, ob ein Property existiert
bachelorModule.hasOwnProperty("title"); //true
bachelorModule.hasOwnProperty("credits"); //false
```

JavaScript hat, wie auch andere objektorientierten Sprachen, Konstruktoren (die immer gross geschrieben werden)

```
function Name(vorname, nachname){
   this.vorname = vorname;
   this.nachname = nachname;
   this.birthDate = {
      year: 0,
      month: 0,
      day: 0
   }
}
```

Prototyp

Wie bereits im Kapitel 'Funktionen' beschrieben, können Funktionen direkt in Objekten definiert werden. Dann existieren sie jedoch nur für diese eine Objekt (z.B. dem Objekt aPerson). Wenn ich nun eine Funktion definieren will, die für alle `Name`-Objekte existiert, muss ich sie mithilfe dem `prototype`-Keyword definieren.

```
Name.prototype.hello = function(){
   console.log("Hello " + this.vorname);
}

var aPerson = new Name("Thomas", "Koller");
aPerson.hello();
```

Diese Methode funktioniert, da jedes Objekt ein Prototype hat. Wenn ein gesuchtes Property nicht im Objekt-eigenen Prototype gefunden, so wird rekursiv im Prototype des Prototype-Objekts gesucht, bis man ganz oben bei `Object` angekommen ist. Falls dort immer noch nichts gefunden wurde, wird `null` zurückgegeben.

Wenn ein Objekt mithilfe der `Object.create()`-Methode instanziiert wird, so hat das neu erstellte Objekt den Prototypen des mitgegebenen Objekts.

```
var obj1 = {
   a: 1
}

var obj2 = Object.create(obj1); //Der Prototyp von obj2 ist jetzt obj1
```

```
6
7 console.log(obj2.a) //Output: 1
```

`obj2` selbst hat kein 'a'-Property, es wird also eine Stufe höher gesucht, im Property von `obj2`, also `obj1`

1.4 Arrays

```
1  //Instanziierung von Arrays über var
2  var emptyArray = [];
3  var numberArray = [1, 3, 6];
4
5  //Instanziierung von Arrays über new
6  var anotherEmptyArray = new Array();
7  var arrayOfFive = new Array(5);
8
9  //Da JavaScript keine Typisierung kennt, sind auch gemsiche Arrays möglich
10 var mixedArray = ["String", 4, true];
11
12 //Es können auch Objekte in Arrays verpackt werden
13 var modulesArray = [
14    {title:"WEBAPP", instructor:"Koller"},
15    {title:"WEBTEC", instructor:"Infanger"}
16 ];
17
18 //Zugriff und hinzufügen von Array-Elemente erfolgt gleich wie bei bekannten Sprachen
19 console.log(numberArray[0]) //1
20 numberArray[3] = "new Element"
21
22 //Arrays müssen nicht zwingend ganz gefüllt sein
23 var sparseArray = new Array(1000);
24 console.log(sparseArray[500]); //undefined
```

1.5 JavaScript und JSON

Javascript erlaubt die direkte Konvertierung von Objekten zu JSON. Die Methode `JSON.stringify()` ruft die Methode `toJSON()` auf (falls diese existiert) und serialisiert das zurückgegebene Objekt.

Um JSON wieder in ein Objekt zurückzuverwandeln, ruft man `JSON.parse()` auf.

```
1 var json = JSON.stringify(bachelorModule));
2 //Output: {"title":"WebApp","instructor":"Thomas Koller"}
3
4 var otherBachelorModule = JSON.parse(json);
```

2 Javascript - Advanced

2.1 Verarbeitung von JavaScript

JavaScript kann normal, asynchron oder deferred ausgeführt werden. Bei der normalen Verarbeitung wird das HTML-Parsing pausiert, das JS-Skript heruntergeladen, kompiliert und ausgeführt und erst anschliessend mit dem HTML-Parsing weitergemacht. Wenn man die asynchrone Verarbeitung wählt, wird das JS-Skript im Hintergrund heruntergeladen. Erst wenn das Skript heruntergeladen wurde, wird das HTML-Parsing pausiert und das Skript wird ausgeführt. Bei der deferred-Methode wird das Skript ebenfalls im Hintergrund heruntergeladen. Jedoch wird hier gewartet, bis das gesamte HTML-Parsing abgeschlossen ist, bevor das Skript ausgeführt wird.

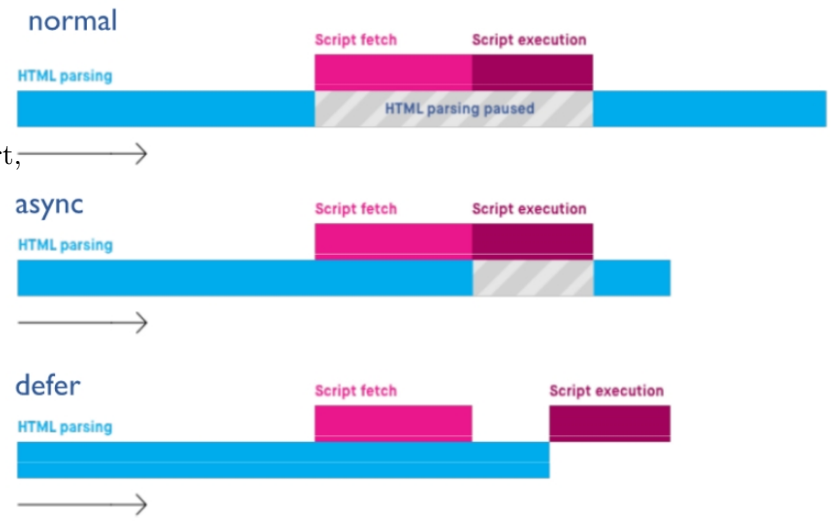


Abb. 2.1: Verschiedene Arten, wie JS verarbeitet werden kann

2.2 Verzögerte Ausführung

Mittels den Methoden `long setInterval(function f, unsigned long interval, any args)` und `long setTimeOut(function f, unsigned long timeout, any args)` kann die Ausführung der übergebenen Methode verzögert (setTimeOut) oder in einem definierten Intervall wiederholt (setInterval) werden. Die Ausführung der Methode `f` wird um `timeout` Millisekunden verzögert bzw. nach `interval` Millisekunden wiederholt.

2.3 DOM

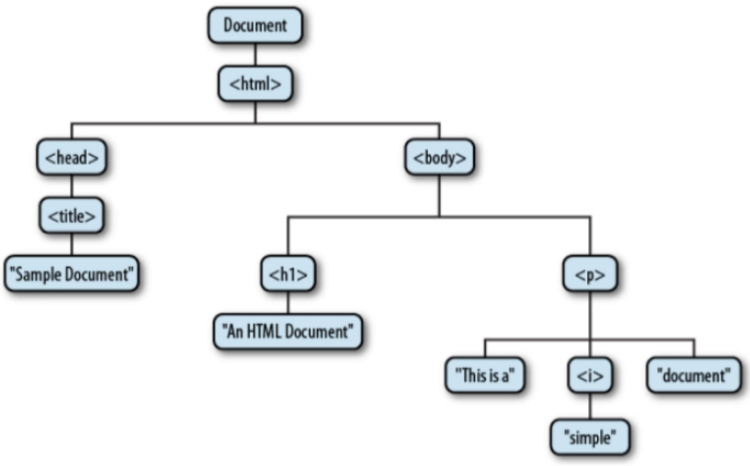


Abb. 2.2: Beispiel eines DOM-Baumes

DOM steht für *Document Object Model* und bezeichnet die Struktur einer HTML-Website. Alle Objekte der Website werden in der Baumstruktur des DOMs als Node abgespeichert. JavaScript kann direkt auf diese Nodes zugreifen z.B. wäre `"simple"` in Abb. 2.2 mittels `document.childNodes[0].childNodes[1].lastChild.firstChild.nextSibling.childNodes[0]` erreichbar. Da dies jedoch ein bisschen umständlich ist, werden im nächsten Kapitel einige einfachere Selektions-Methoden vorgestellt.

2.4 DOM-Manipulation

Wie bereits zu Beginn des Kapitels erwähnt, wird JavaScript unter anderem zur DOM-Manipulation verwendet. Um Elemente des DOM manipulieren zu können, muss dem Skript zuerst mitgeteilt werden, welches Element man manipulieren möchte. Dabei gibt es verschiedene Methoden:

Über ID (empfohlen)

```
1 var button = document.getElementById("button1")
```

```
1 <button id="button1">Click me!</button>
```

Über Namen

Diese Methode gibt eine NodeList zurück mit allen gefundenen Elementen

```
1 var buttons = document.getElementsByName("option_buttons")
2 console.log(buttons[0].tagName) //Output: button
```

```
1 <button name="option_button">Pizza Margharita</button>
2 <checkbox name="option_button">Pizza Diavola</checkbox>
```

Über Tags

Funktioniert gleich wie `getElementsByName` aber filtert nach HTML-Tags

```
1 var buttons = document.getElementsByTagName("button")
2 console.log(buttons[0].name) //Output: option_button
```

```
1 <button name="option_button">Pizza Margharita</button>
```

Über Klasse(n)

Gibt eine HTML-Collection zurück. Falls mehrere Klassen spezifiziert werden, muss das Element Mitglied *aller* Klassen sein

```
1 var buttons = document.getElementsByClassName("options")
2 console.log(buttons[0].name) //Output: option_button
```

```
1 <button class="options">Pizza Margharita</button>
```

Über CSS-Selektoren
Nur kompatibel mit HTML5. Gibt eine NodeList zurück.

```
var logs = document.querySelectorAll("#log>span)
```

```
<div id=span>
   <span>I'm selected</span>
</div>
```

3 JQuery

JQuery ist eines der inzwischen hunderten von JavaScript Frameworks. Genau wie JavaScript kann mit JQuery ebenfalls DOM- und CSS-Manipulation durchgeführt, sowie Browser Events und AJAX-Requests behandelt werden.

JQuery Funktionen werden mittels `jQuery()` bzw. `$()` aufgerufen. Da jede jQuery-Methode ein jQuery-Object zurückgibt, können beliebig viele Methoden aneinandergehängt werden (ob das leserlich ist, ist eine andere Frage):

```
$("p.details").css("background-color","yellow").show("fast");
```

Die obenstehende jQuery-Methode selektiert alle `p`-Elemente der Klasse `details` mit der Hintergrundfarbe Gelb und zeigt sie an mit einer schnellen Animation.

Man kann jQuery Methoden auf vier verschiedene Arten aufrufen:

1. Selektion
   `$(selector, (optional)context)`
2. Wrapper
   `$(document), $(window) etc.`
3. HTML-Elemente
   `var img = $("<img>", src:  url, css:...)`
4. Events
   `$(function)` ← ist equivalent zu `$(document).ready(function);`

jQuery verfügt nur über eine getter/setter Methode. Je nach dem, ob der Methode ein Parameter übergeben wird, fungiert sie als getter oder setter.

```
//Der Methode wird kein Parameter übergeben --> sie gibt den gefundenen Wert zurück
$("#icon").attr("src");

//Der Methode wird zusätzlich noch 'icon.gif' übergeben. --> Sie setzt den Wert
$("#icon").attr("src", "icon.gif");
```

Dasselbe gilt nebst DOM-Manipulation auch für CSS-Manipulation

```
//Kein Wert übergeben --> gibt die Wert von fontWeight zurück
$("h1").css("fontWeight");

//Es werden Parameter übergeben --> setzt den CSS-Style dementsprechend
$("h1").css({
   backgroundColor: "black",
```

```
   textColor: "green",
});
```

jQuery kann auch Text und HTML schreiben und lesen

```
//Gibt den Dokument-Titel zurück
var title = $("head title").text();

//Gibt das HTML des ersten 'h1'-Elementes zurück
var headline = $("h1").html();

//Setzt jede 'h1'-Heading zu 'TITEL'
$("h1").text(TITEL);
```

Mit dem `data()`-Attribut kann jQuery von jedem Element beliebig Daten lesen und schreiben. Der Syntax des `data()`-Attribut ist `jQuery.data(element, key, value)` wobei das `element` das DOM-Element ist, mit dem die Daten assoziiert sind, der `key` der Name der zu speichernden Daten und der `value` recht selbsterklärend der Wert der zu speichernden Daten ist.

```
//Daten speichern.
$.data(div, "test", {
   first: "firstText",
   pizza: "secondText"
});

//Daten abrufen und als Text im ersten bzw. letzten <span>-Element der Seite setzen
$("span:first").text( jQuery.data(div, "test" ).first);
$("span:last").text( jQuery.data(div, "test" ).pizza);
```

Wie bereits erwähnt, können mit jQuery auch EventHandler programmiert werden. Dabei folgen alle Browser derselben API (ausser alte IE-Versionen, aber das überrascht eigentlich niemanden).

```
$("#imageShrinker").click(
   function () {
      $("img").animate({height: 0})
   }
);
```

Die obenstehende Funktion setzt die Höhe jedes Bildes auf 0 wenn der Button mit der id `imageShrinker` geklickt wird. Es gibt eine vordefinierte Liste von Events, auf welche reagiert werden kann wie z.B. `click()`, `keypress()`, `load()`, `focus()` etc.

Das obere Codebeispiel deckt auch eine weitere Möglichkeit von jQuery ab: Es kann Animationen ausführen wie z.B. Elemente anzeigen und verschwinden lassen.

jQuery kann auch andere Websites oder remote-Scripts laden und ausführen.

```
//Eine externe Seite laden und im übergebenen Element anzeigen
$("#stats").load("status_report.html");

//Ein remote-Script laden
$.getScript("https://trustworthySite.ru.cn/js/totallyNotACryptoMiner.js");

//Ein remote-Script laden und anschliessend ausführen
$.getScript("https://trustworthySite.ru.cn/js/totallyNotACryptoMiner.js", function(){
   $(document).mineMonero();
```

```
});
```

jQuery kann ebenfalls JSON laden und parsen

```
$.getJSON("https://server.com/data.json", function(data){
   //geparste Daten aus data.json verarbeiten
});
```

Es können auch AJAX Request über jQuery gemacht werden

```
$.ajax({
   type: "GET", //GET-Request
   url: url, //aufzurufende URL
   data: null, //es sollen keine Daten mitgegeben werden
   dataType: "script", //die Antwort ist ein sofort auszuführendes Script
   success: callback //rufe diese Methode auf, wenn alles erledigt ist
});
```

4 Asyncronous Javascript

4.1 Callback

```
function doSomething(successCallback, failureCallback){
   if(a===1){
      successCallback(a);
   } else {
      failureCallback(new Error("failed"));
   }
}

//Aufruf mittels
doSomething(successCallback, failureCallback);
```

Das obenstehende Codebeispiel demonstriert die Verwendung von Callbacks. Callbacks an sich sind eine gute Sache, jedoch läuft man in die Gefahr, dass man eine sog. *Callback pyramid of hell* programmiert:

```
doSomething(function successCallback(result){
   processResult(result, function successProcessing(newResult){
      finalThing(newResult, function finalSuccess(finalResult){
         console.log("Final Result");
      }, failureCallback);
   }, failureCallback);
}, failureCallback);
```

Wie zu sehen ist, sind solche Konstrukte recht schwer zu lesen. Zudem haben Callback-Methoden keine returns und sie werfen keine Exceptions (aufgrund des fehlenden Callstacks). Die Lösung für das Problem sind:

4.2 Promises

```
var promise = doSomething(); //doSomething() gibt ein Promise zurück
promise.then(successCallback, failureCallback);
```

```
3
4  //oder, kompakter:
5  doSomething().then(successCallback, failureCallback)
6
7  //oder auch
8  doSomething()
9     .then(successCallback)
10    .catch(failureCallback);
```

Promises sind Platzhalter für das Resultat einer späteren, asynchronen Methode. Je nachdem, was das Resultat von `doSomething()` ist, wird entweder der `.then`-Teil (bzw. `successCallback`) oder der `.catch`-Teil (bzw. `failureCallback`) ausgeführt. Welcher der beiden Teile ausgeführt wird, kann jedoch erst zur Laufzeit bestimmt werden. Zudem können mehrere `.then()` aneinander gekettet werden, ohne dass eine Callback pyramid of hell daraus resultiert:

```
1  getSomeData()
2     .then(processData)
3     .then(storeData)
4     .then(finalizeData)
5     .catch(error)
```

Promises können folgendermassen erstellt werden:

```
1  var promise = new Promise(function(resolve, reject){
2     //Methodenausführung, die immer ausgeführt wird
3     if(//alles hat gefunzt){
4        resolve("Stuff's working");
5     } else{
6        reject("Stuff didn't work);
7     }
8  });
```

Ein Promise kann vier Zustände haben:

fulfilled: Promise wurde erfolgreich ausgeführt

rejected: Promise wurde nicht erfolgreich ausgeführt

pending: Promise wurde instanziiert aber noch nicht ausgeführt

settled: Promise wurde ausgeführt (gibt jedoch nicht zurück ob erfolgreich oder nicht)

5 TypeScript

TypeScript ist ein sog. **Superset** von JavaScript. Es ist eine Sprache, mit der objektorientiert in Klassen, Interfaces und Objekten programmiert werden kann. Jedoch kompiliert sie nach (ziemlich unleserlichem) JavaScript. Typescript unterstützt im Gegensatz zu JavaScript auch Types und somit auch Generics.

```
interface LabelledValue {
   label: string;
   value?: string //optionaler Wert
}
function printLabel(labelledObj: LabelledValue){
   console.log(labelledObj.label);
}

let myObj = {
   label: "Size 10 Object",
   value: "10"
}
printLabel(myObj);
```

TypeScript unterstützt auch verschiedene Sichtbarkeiten. So kann mittels `private` die Sichtbarkeit auf die eigene Klasse beschränkt werden. Wie in anderen OO-Sprachen kann mittels Setter- und Getter-Methoden immer noch auf das Attribut zugegriffen werden, sollte das denn vonnöten sein.

```
class BachelorModule{
   private _title: string;

   constructor(title: string){
      this._title = title;
   }

   get title(): string {
      return this._title;
   }

   set title(newTitle: string): void{
      this._title = newTitle;
   }
}

let WEBAPP = new BachelorModule("Web Applications");
console.log(WEBAPP._title);   //error
console.log(WEBAPP.title);  //OK
let newtit: string = "WEBAPP";
WEBAPP.title = newtit;     //OK
```

Mit TypeScript ist auch Vererbung möglich:

```
class BDAModule extends BachelorModule{
   private _expertName: string;

   constructor(name: string){
      super(name);
   }
}
```

6 Backend

6.1 PHP

PHP ist eine serverseitige Script-Sprache. Also im Gegensatz zu JavaScript, bei welcher alles auf dem Computer des Benutzers berechnet wird, wird bei PHP alles bereits auf dem Webserver berechnet und nur die Resultate an den Browser des Benutzers geschickt.

Ein PHP-Request läuft folgendermassen ab:

1. Der Client ruft eine PHP-Seite auf
2. Der Webserver leitet den Request an den PHP-Interpreter weiter
3. Der Interpreter verarbeitet die Seite und schickt das Resultat zurück an den Server
4. Der Server schickt das Resultat zurück an den Client.

PHP und HTML können in einer Datei gemischt werden.

```
<html>
   <head>
      <title>PHP-Stuff</title>
   </head>
   <body>
      <?php echo "Hello World"; ?>
   </body>
</html>
```

Variablen in PHP beginnen immer mit einem `$` und müssen nicht zwingend zuerst deklariert werden. Strings werden, wie gewohnt, entwerder mit Doppel- oder Einfachanführungszeichen geschrieben. Ebenfalls bereits bekannt aus anderen Sprachen ist das Abschliessen eines Statements mit dem ';'

PHP kann ebenfalls objektorientiert programmiert werden und hat Java/.NET ähnliche Kontrollstrukturen wie Schlaufen oder Konditionen. Externe PHP-Dateien können mittels `import` oder `include` importiert werden.

Falls eine Webapplikation eine SQL-Datenbank im Backend hat, wird meist über PHP darauf zugegriffen

```
include 'db_credentials.php'

//$host, $user, $pass und $db sind im File 'db_credentials.php' gespeichert
$mysqli = new mysqli($host, $user, $pass, $db);

//mittels '->' kann auf Methoden zugegriffen werden, hier also die Methode 'query'
    vom mysqli-Objekt
$result = $mysqli->query("SELECT * from personen");
```

Nebst plainText und HTML-Seiten kann PHP noch andere Content-Types zurücksenden, es muss jedoch als erstes im Header mittels `header("Content-Type:  whatever");` spezifiziert werden (z.B. JSON)

6.2 NodeJS

Eine weitere Möglichkeit fürs Backend, nebst PHP bietet **NodeJS**. NodeJS ist Teil des MEAN-Stacks (MongoDB, Express, Angular.js, Node.js) und ist eine JavaScript RunTime. NodeJS hat eine Package-Struktur, mit welcher Libraries mittels dem Package-Manager NPM installiert werden können.

NodeJS ist asynchron und Event-Basiert. Es unterstützt nur single-threaded Operationen, ist jedoch skalierbar und leicht (ausser man lädt. 2.9 Millionen npm-packages). NodeJS arbeitet, wie Angular auch, mit Modulen, die exportiert und importiert werden können:

```
1  //circle.js
2  exports.area=function(r){
3     return Math.PI * r * r;
4  }
5
6  //main.js
7  var circle = require('./circle.js');
8  console.log("The area of a circle with radius 4 is " + circle.area(4));
9
10 //circleModule.js
11 function Circle(){
12    this.r = 1;
13 }
14 //gleich wie bei JavaScript können einem Objekt mit 'prototype' neue Funktionen
       hinzugefügt werden.
15 Circle.prototype.area=function(){
16    return Math.PI * this.r * this.r;
17 };
18 Circle.prototype.setRadius=function(r){
19    this.r = r;
20 }
21 module.exports = new Circle();
22 //es kann auch nur der Konstruktur exportiert werden
23 module.exports = Circle;
24
25 //main.js
26 var circle2 = require('./circleModule.js');
27    //Falls nur der Konstruktor exportiert worden ist, muss jetzt noch folgendes
          gemacht werden
28    var c = new circle2();
29 circle2.setRadius(4); //bzw. c.setRadius(4)
30 console.log("The area of a circle with radius 4 is " + circle2.area());
```

NodeJS basiert auf dem sog. **Event-Loop** (siehe Abb. 6.1)

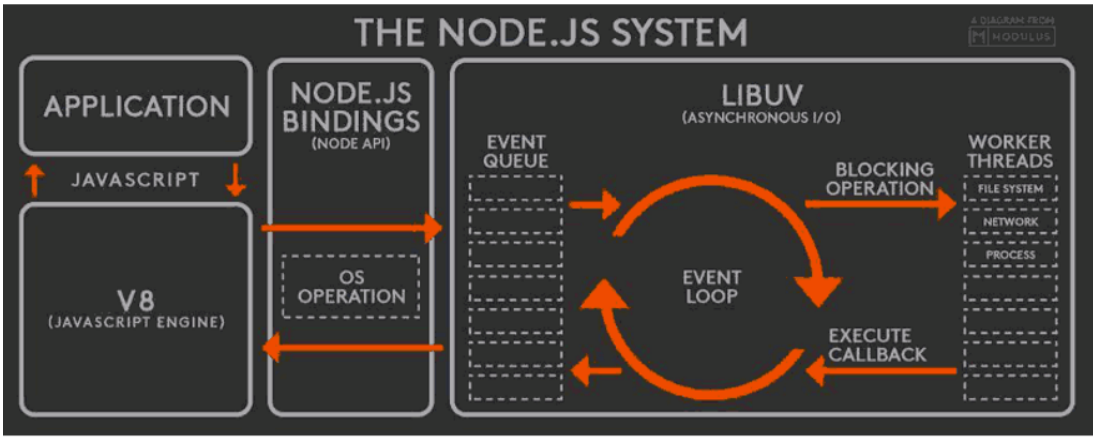


Abb. 6.1: NodeJS Event Loop

NodeJS unterscheidet zwischen **blocking** und **non-blocking** Operationen.

Blocking: Synchroner Aufruf. Warten auf Resultat einer (meist nicht JS-)Funktion.

Non-Blocking: Asynchroner Aufruf mit Callback-Methode.

```
//Synchron/Blocking
const fs = require('fs');
const data = fs.readFileSync('/file.md');
console.log(data);
doMoreStuff(); //Wird erst nach dem console.log ausgeführt

//Asynchron/Non-Blocking
const fs = require('fs');
fs.readFile('/file.md'), (err, data) => {
   if(err)
      throw err;
   console.log(data);
doMoreStuff(); //Wird vor dem console.log ausgeführt
```

6.3 Express

Express ist eine Middleware und ein NodeJS Framework. Das Ziel davon ist die Verarbeitung von Requests über mehrere Stufen, unter dem Single Responsibility Prinzip. Jede Middleware führt ihre Änderungen an den Request- oder Response-Objekten durch und leitet sie anschliessend weiter an die nächste Middleware.

```
//Loggen von jedem Request
var myLogger = function (req, res, next) {
   console.log('LOGGED'); next();
};
app.use(myLogger);
app.get('/', function (req, res) {
   res.send('Hello World!'); }
);
app.listen(3000);
```

7 Authentication

Es gibt drei Use-Cases für Authentication:

1. Authentifizierung für Zugriff auf (WebServer basierte) WebApps
2. Authentifzierung für Zugriff au Web Services
3. Anbindung an andere Provider wie Facebook oder Google

7.1 Basic Authentication

1. Client will auf URL zugreifen
2. Web Server fordert im Header Authentifizierung für einen Bereich an:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="RealmName"
```

3. Client schickt User/Passwort Base64-encoded im Header im Format User:Passwort

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

Jedoch ist die Basic Authentication nicht mehr sicher, da User/Passwort im Klartext übers Netzwerk geschickt werden.

7.2 Digest Authentication

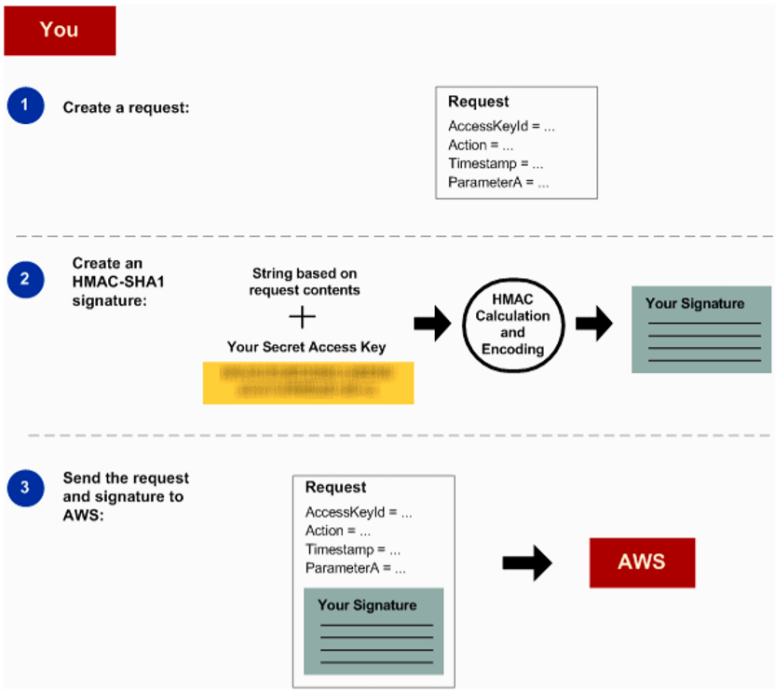
Die Userdaten werden nicht im Plain Text verschickt, sondern es wird nur deren Hashwert (MD5) berechnet und verschickt. Der Server schickt einen sog. *nonce* (Eindeutiger, einmaliger String) und *opaque* (String, der vom Server definiert wurde und wieder zurück geschickt wird).

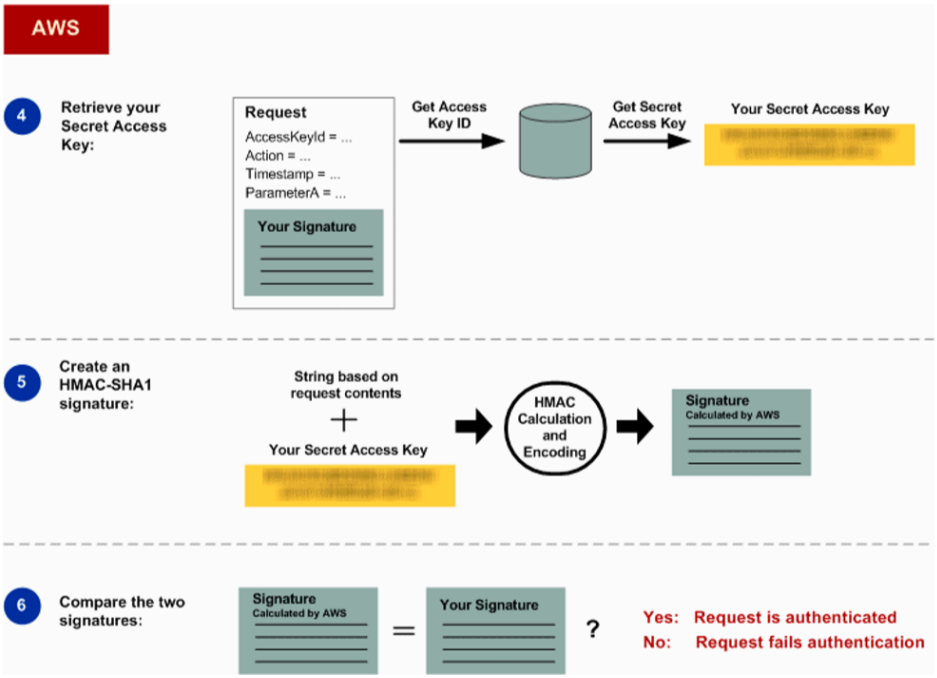
1. Client will auf URL zugreifen
2. Web Server fordert im Header Authentifizierung für einen Bereich an und shickt *nonce* und *opaque*-Werte mit
3. Client berechnet die Digest Werte daraus und fügt sie dem Header hinzu
4. Server prüft Digest. Wenn erfolgreich, wird entweder Zugriff gewährt oder nächste Authentifizierungsphase eingeleitet.

Digest Authentication ist zwar sicherer als Basic Authentication, ist aber immer noch nicht sicher verglichen zu einer Public Key Authentication, da es eigentlich nur das Passwort schützt. Alles andere ist immer noch öffentlich und kann abgehört werden.

7.3 Public Key Authentication

Die Authentifizierung geschieht über einen MAC (Message Authentication Code). Dieser Code wird über einen Privaten Access-Key und einem String berechnet





7.4 JWT (JSON Web Token)

JWTs werden benutzt, um JSON-Objekte sicher über das Netzwerk zu senden. Es besteht aus einem Header, einem Payload und einer Signatur.

7.4.1 Header

Beinhaltet den verwendeten Algorithmus

7.4.2 Payload

Beinhaltet die gesendeten Attribute (Claims). Diese können Private, Public oder Standard sein. Standard Claims sind z.B.

iss: Issuer

sub: Subject

aud: Audience

exp: Expiration Date

nbf: Not Before

iat: Issued At (Time)

jti: JWT ID

```
{"iss": "www.hslu.ch",
 "sub": "zakoller",
 "name": "Thomas Koller",
 "admin": true}
```

7.4.3 Signatur

Der Header und das Payload sind Base64 encoded und mit dem im Header angegebenen Protokoll verschlüsselt

```
HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Das gesamte Token ist nachher einfach ein String mit allen drei Teilen in Base64-encoding, die einzelnen Teile durch '.' getrennt:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ3d3cuaHNsdS5jaCIsInN1YiI6Inpha29sbGV
yIiwibmFtZSI6IlRob21hcyBLb2xsZXIiLCJhZG1pbiI6dHJ1ZX0.gPO7XZYZ7Au83MHRuIMtq41LOl8fWYBG
4-ylb4f2c4
```

JWT ist kleiner als XML, die Signatur lässt sich sehr einfach berechnen und das es JSON ist, ist das Parsing auch recht einfach. Zudem ist es self-contained, benötigt also keine zusätzlichen Pakete etc.

7.5 OAuth 2.0

OAuth 2.0 ist der Nachfolger des veralteten OAuth 1.0 Protokolls. Es ermöglicht die Authentifizierung über Drittparteien, man kann sich also z.B. bei einem Fotodruckservice mit seinem Google- Facebook- oder GitHub-Account anmelden.

Beim Authentifizierungs- und Autorisierungsprozess gibt es grundsätzlich vier verschiedene Rollen:

Client: 3rd Party Anwendung, die auf die Ressourcen zugreifen möchte.

Resource Server: Server, auf welchem die Ressourcen gespeichert sind.

Resource Owner: Besitzer der Ressource auf dem Resource Server.

Authorization Server: Authentifiziert Resource Owner und stellt die Tokens für den Client zur Verfügung

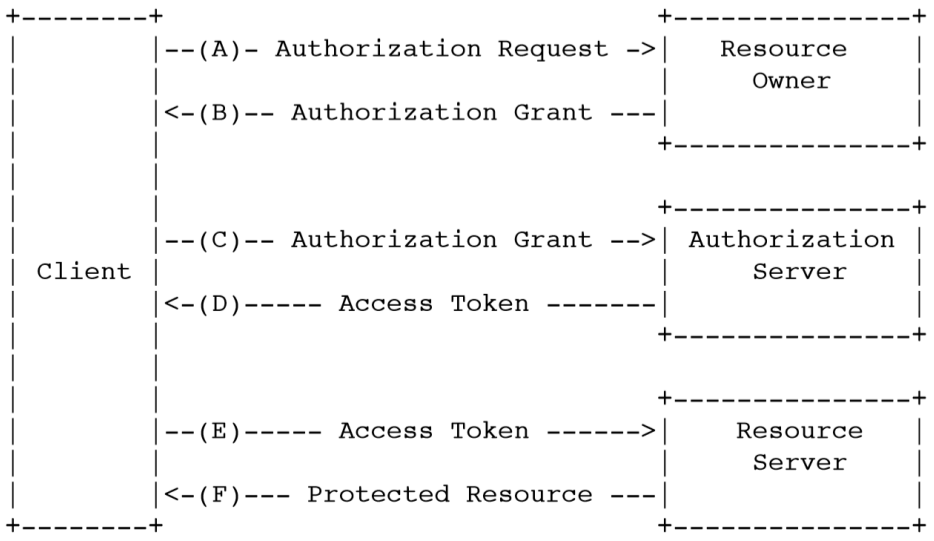


Abb. 7.1: Basic Flow wenn ein Client auf eine Ressource zugreifen will

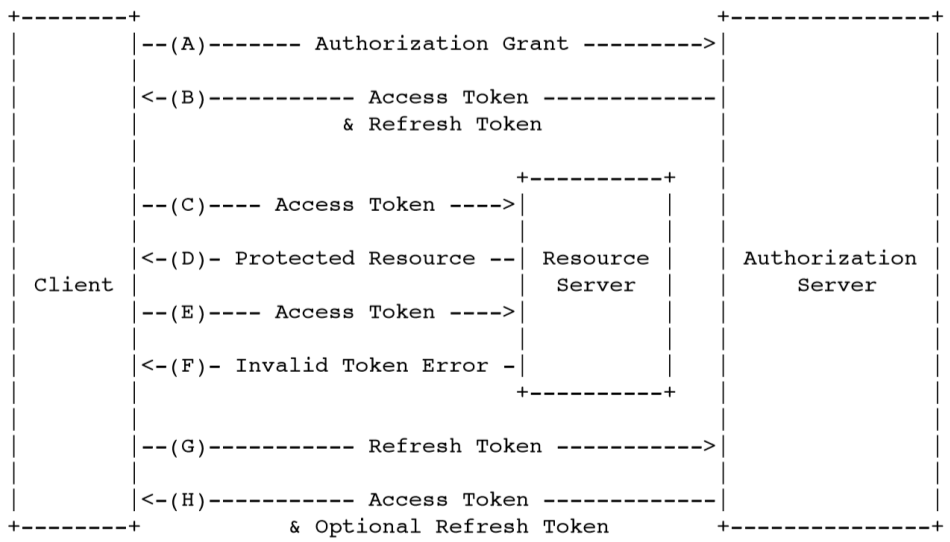
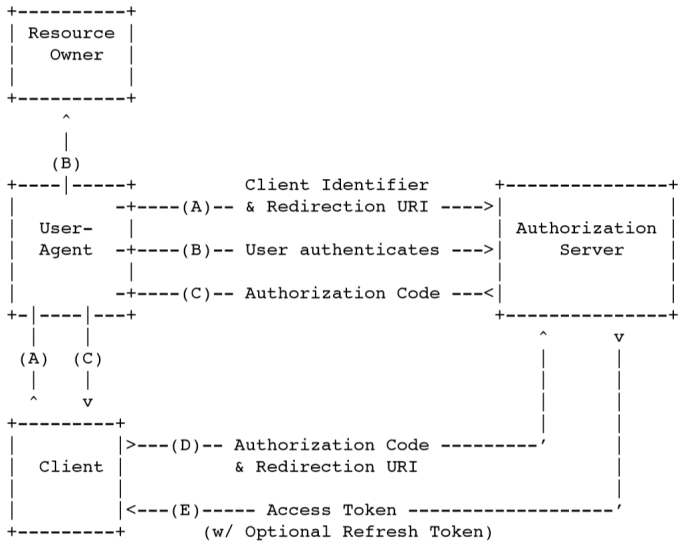


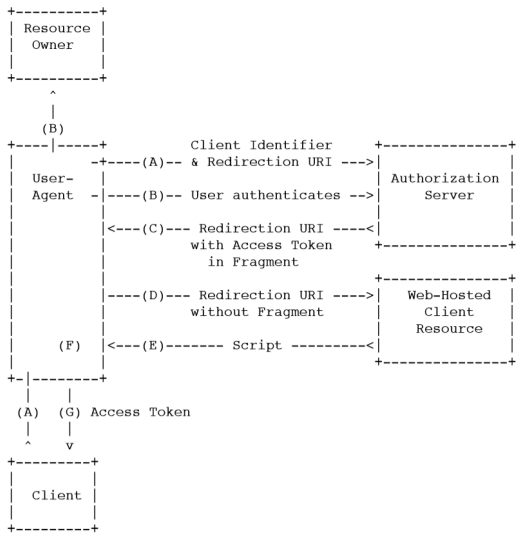
Abb. 7.2: Wenn das Token abgelaufen ist, muss der Resource Owner dem Client wieder Zugriff gewähren über den Authorization Server

Es gibt verschieden Möglichkeiten, wie ein Client ein solches Token erhalten kann:

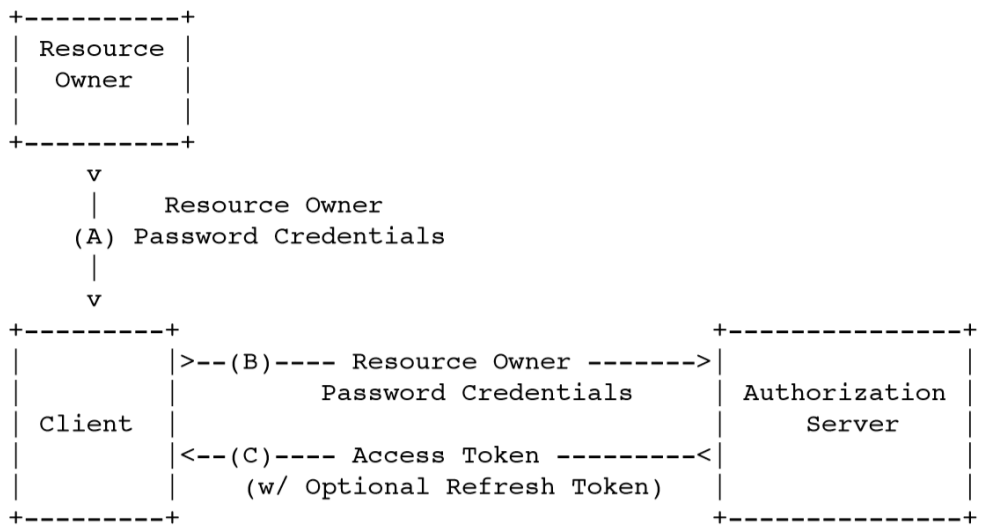
Authorization Code:



Implicit:



Resource Owner Password :



8 Single Page Application

8.1 Definition

"A Web app that fits on a single web page providing a fluid UX by loading all necessary code with a single page load"

Eine SPA sit eine Rich Client Application im Browser, die keinen Page Reload benötigt, um eine neue Ressource zu laden (→ die URL ändert sich nicht oder nur hinter dem #). Zudem besteht sie aus reinem HTML und JS, benötigt also keine Plugins wie z.B. Flash o.ä.

Man kann trotz fehlendem Page Reload den '←'-Button benutzen, um zur vorherigen Ressource zurückzukehren. Ebenfalls können einzelne Ressourcen effektiv gebookmarked werden.

Aufgrund des fehlenden Page Reloads funktioniert eine SPA, einmal aufgerufen, auch offline.

Die Vorteile der SPA lassen sich mit den **3R** zusammenfassen:

- **R**each
- **R**ich User Experience
- **R**educed Round Tripping

8.2 Komponenten

Routing: Das clientseitige Routing ermöglicht eine Navigation auf der Seite ohne Page Reload. Zudem muss der Backbutton und die Browserhistory (inkl. Bookmarks) wie gewohnt funktionieren. Vor HTML 5 wurde das mit einem Anchor (z.B. `/#/home`) gelöst. HTML 5 bietet dafür eine eigene API mit der sich auch die Browserhistory ansteuern lässt.

Template-Rendering: Beim Template-Rendering wird das Model (TypeScript) über den Controller mit der View (HTML) verknüpft. Durch die Entkoppelung der beiden wird eine bessere Testbarkeit garantiert und durch das Binding reduziert sich der Code für das Updaten der Daten (die vom Model abhängen) in der HTML-View.

Daten Zugriff: Die Daten werden meist über eine REST-API mittels einem XMLHTTP-Request vom Server bezogen (siehe später) und können bei Bedarf z.B. mittels LocalStorage gecached werden.

```
1  //cachen von Informationen
2  storelocalStorage.setItem("lastname","Smith");
3
4  //Abrufen der gecachten Information
5  retrievedocument.getElementById("result").innerHTML =
       localStorage.getItem("lastname");
```

Code-Modularisierung: Durch die Modularisierung des Codes ergibt sich eine klare Struktur, die sich viel besser Warten und Testen lässt. → Separation of Concern

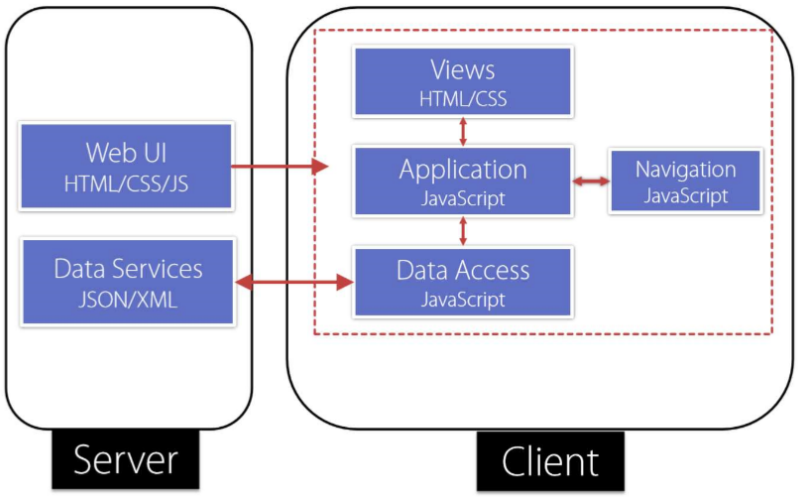


Abb. 8.1: Visualisierung der verschiedenen SPA-Komponenten

9 Angular

Angular ist ein Framework, um SPAs für CRUD Anwendungen zu bilden. Es basiert auf Typescript und besticht insbesondere durch die Benutzung von Dependency Injection und einem performanten 2-Way Binding.

9.1 Komponenten

Angular ist komponentenbasiert. Das heisst, die ganze Angular-Applikation ist eine einzige Hauptkomponente. Diese Hauptkomponente wird wiederum in kleinere Komponenten unterteilt, die wiederum Unterkomponenten haben etc. So entsteht schlussendlich ein sog. Komponentenbaum.

Daten werden durch Components (Logik) angezeigt. Jede Komponente enthält entweder direkt ein Template oder einen Verweis auf ein Template (Ansicht, in HTML). Verbunden sind diese zwei Dateien über Event- und Property-Binding.

Das Event-Binding bedeutet, dass in HTML Events geschehen, die dann Funktionen in den Komponenten aufrufen. Solch ein Event kann zum Beispiel das Drücken eines Knopfes sein:

```
1 <button (click)="showResults()">Resultate</button>
```

Die Komponenten wiederum können über das Property-Binding Inhalte im Template verändern, womit sich die Ansicht für den Benutzer verändert. So kann zum Beispiel ein Button-Text von der Komponente aus geändert werden:

```
1 <input type="submit" value="{{buttonText}}">
```

9.2 Module

Jede Angular Applikation wird über Module organisiert. Jede Angular-Applikation hat mind. ein sog. "Root Module", das die Hauptkomponente zur Verfügung stellt. Angular selbst bietet verschiedenste Module zur Verwendung an (`@angular/core, /forms`, etc). Aufbau eines solchen Moduls:

```
import {NgModule} from "@angular/core";
import { CommonModule } from "@angular/core";
import { FormsModule } from "@angular/forms";

@NgModule({
   imports:   [CommonModule, FormsModule],
   declarations: [ContactComponent, HighlightDirective, AwesomePipe],
   exports:   [ContactComponent],
   providers:    [ContactService]
})

export class ContactModule{
   //stuff here
}
```

Unter `imports:` werden alle importierten Module aufgelistet. `declarations:` beinhaltet alle Komponenten, die dieses Modul beinhaltet

Solche Module werden exportiert und können wiederum von anderen Modulen importiert werden, um auf Funktionen dieser Module zugreifen zu können.

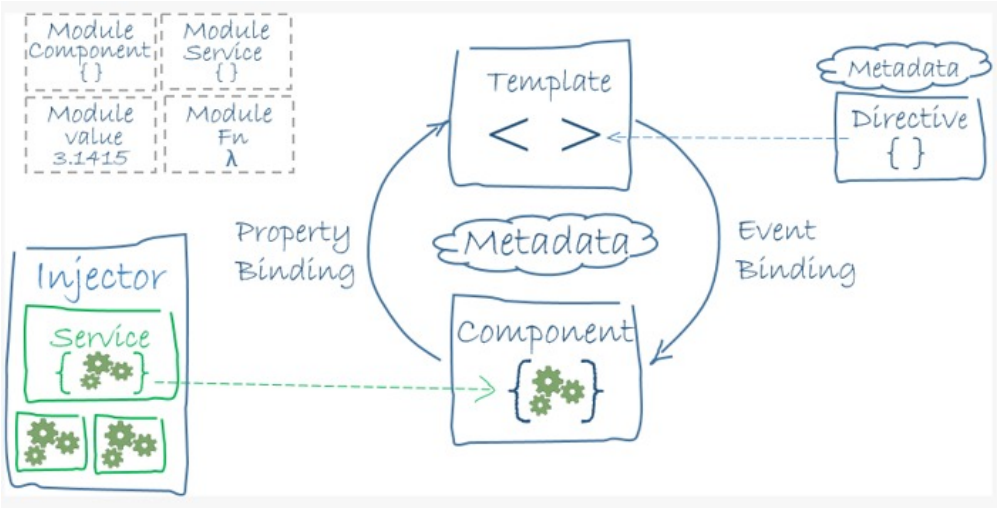


Abb. 9.1: Visualisierung der Funktionsweise von Angular 5

9.3 Services

Ein Service ist prinzipiell nichts anderes als eine Klasse, die mittels Dependency Injection in eine andere Klasse injected werden kann. (Siehe Dependency Injection). Services sind normalerweise Container für Werte, Funktionen oder Features. Sie haben kein Interface und holen z.B. Daten vom Server, validieren user Input und loggen Events.

9.4 Directives

Komponenten Direktiven mit einem HTML-Template

Strukturelle Direktiven: verändern das Layout (DOM), Bsp: `*ngFor, *ngIf`

Attribut Direktiven: Verändern Erscheinung oder Verhalten eines existierenden Elements. Bsp: `<input [(ngModel)]="hero.name">`

9.5 Providers

Angular ist dumm und kann mit Services und ihren Injectoren nicht einfach so umgehen. Deshalb musst du für jeden Service Provider spezifizieren, die den Services sagen, wie und wo die Werte der Dependencies geholt werden müssen.

9.6 Dependency Injection

Dependency Injection kann mit dem Prinzip des Singletons verglichen werden: Falls noch keine Instanz existiert, wird sie erstellt. Wenn die Instanz irgendwo in der Applikation existiert, wird von überall her auf diese einzelne Instanz zugegriffen, sollte ein Modul eine Methode dieser Instanz benötigen.

Nehmen wir das Beispiel eines Autos:

```
export class Auto{
   public motor: Motor;

   constructor(){
      this.motor = new Motor();
   }
}
```

Dieses Prinzip ist bereits bekannt aus diversen objektorientierten Programmiersprachen. Was wäre aber, wenn wir nun keinen neuen Motor erstellen müssten, sondern einen bereits existieren Motor einfach importieren könnten?

```
import { Motor } from '../Motor/';

export class Auto{
   public motor: Motor;

   constructor(public motor: Motor){

   }
}
```

Wir haben nun einfach den Service `Motor` importiert, das bereits irgendwo in der Applikation existiert und müssen somit keinen neuen Motor erstellen. Das ist Dependency Injection. Zu beachten ist nur, dass man in der Klasse `Motor.ts` noch spezifiziert, dass dieser Motor als Injectable/Service genutzt werden kann. Das macht man, indem man vor dem `export class Motor` noch `@Injectable()` einfügt.

10 REST

REST steht für Representational State Transfer und ist ein zustandloses Client-Server Modell, das auf HTTP basiert. Zustandlos heisst, der Server speichert keine Kontextinformationen, sondern der Kontext muss bei jeder Anfrage mitgeschickt werden. REST eignet sich gut zum cachen von Informationen.

REST besitzt eine einheitliche Schnittstelle, die auf vier Eigenschaften aufbaut:

Identifikation von Ressourcen Jede Ressource ist eindeutig identifizierbar mittels ihrer URL. Umgekehrt wird jede Information, die eine URL/URI besitzt als Ressource gekennzeichnet.

Manipulation der Ressourcen durch 'Representationen' Alle Ressourcen können unterschiedliche Repräsentationen haben. Je nach dem, welche Repräsentation der Client verlangt, kann der Server JSON, XML oder HTML formatierte Ressourcen ausliefern. Der Client kann also folgende Anfrage schicken:

```
1 GET /notes/1234 HTTP/1.1
2 Host: www.hslu.ch
3 Accept: application/xml, application/json
```

und erhält die Antwort entweder im XML- oder JSON-Format Auch möglich ist ein `Content-Type: application/xml, application/json` anstelle von `Accept:`

Beschreibende Meldungen REST-Nachrichten verwenden die HTTP-Standartmethoden wie PUT, GET etc.

Hypermedia as the Engine of Application State (HATEOS) Der Client der REST-Schnittstelle navigiert ausschliesslich über URLs, welche vom Server bereitgestellt werden.

Wie bereits bei Punkt eins erwähnt, wird alles, was eine URI besitzt als Ressource bezeichnet. Etwas allgemeiner kann man sagen, alles was für den Client interessant sein könnte ist eine Ressource und erhält somit eine URI.

Bei REST-Methoden (was schlussendlich nichts anderes als HTTP-Methoden sind) gibt es **safe** und **idempotente** Methoden. Safe-Methoden sind solche, die Daten *nicht verändern*, also keine Schreiboperationen durchführen. Idempotente Methoden sind Methoden, die die Daten auch bei mehrfacher Durchführung nicht verändern. So ist eine `GET`-Methode zum Beispiel sowohl safe wie auch idempotent. Denn mit der `GET`-Methode liest man Daten und die Antwort auf den 329839876432-ten `GET`-Request auf diese Ressource wird dieselbe sein wie die Antwort auf den ersten Request.

Methode	Safe	Idemptotent
`GET`	✔	✔
`POST`	✗	✗
`PUT`	✗	✔
`DELETE`	✗	✔
`HEAD`	✔	✔
`OPTIONS`	✗	✔
`PATCH`	✗	✗

Untenstehende Bilder sagen eigentlich alles über HTTP-Statuscodes aus, was man wissen muss:

HTTP status ranges in a nutshell:

1xx: hold on

2xx: here you go

3xx: go away

4xx: you fucked up

5xx: I fucked up

11 Mobile Web-Apps

Es gibt fünf Möglichkeiten, wie eine Web-App aufs Handy kommen kann:

1. **Web-App:** Im Browser mittels HTML, CSS, JS
2. **Hybrid:** WebApp in einen native Wrapper 'verpackt'. Kann als native App installiert werden und greift auf native Ressourcen zu (z.b. Cordova)
3. **Cross-Compiled:** In Sprache X geschreibene App, die nach Java/Kotlin/Swift/Binär kompiliert wird und somit native ausführbar ist (z.b. Xamarin mit C#)
4. **JIT-compiled:** App wird in JS geschrieben und 'Just In Time' (JIT) auf dem JS-Engine des Handys kompiliert wird. Kann wie Hybrid-App auf native Ressourcen zugreifen (z.b. NativeScript)
5. **Native App:** WebApp wird ins Native portiert/neu geschrieben (Java/Kotlin für Android, Swift für iOS).

WebApps haben im Gegensatz zu nativen Apps sowohl Vor- wie auch Nachteile

Vorteile

- Eine Code-Basis für mehrere OS
- Kein App-Store, Download oder Installation notwendig
- App kann jederzeit veröffentlicht, geupdated und verändert werden
- Vorhandene Web-App kann zu Mobile App erweitert werden

Nachteile

- Läuft ausschliesslich im Browser
- Kein Ressourcen-Zugriff (Kamera, Kontakte etc.)
- App nicht im App-Store → Benutzer müssen sie finden

11.1 Ionic

Ionic ist ein Angular-Framework, das die Mobile WebApp-Entwicklung vereinfacht. Es bietet bereits UI-Komponent-Templates, APIs etc.

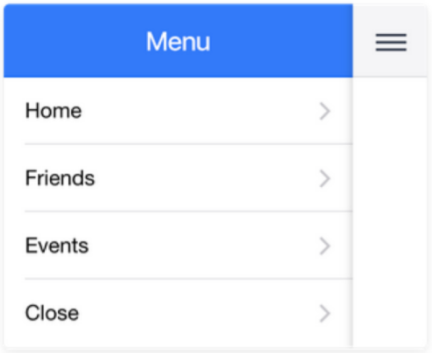
Da es ein Angular-Framework ist, basiert es auf TypeScript und npm.

Man kann mit zwei Zeilen Code bereits eine funktionierende (zwar leere, aber egal) WebApp haben:

```
//ionic und cordova installieren
npm install -g ionic cordova

//eine neue WebApp mit sidemenu erstellen
ionic start myApp sidemenu

//App starten
cd ./myApp
ionic serve
```



Die App wird auf `localhost:8100` aufgeschalten und kann im Browser angeschaut und getestet werden.

Die verschiedenen Pages sind Angular-Module, die in TypeScript geschrieben sind und liegen im subdirectory `src/app/pages/`. Jede Page besteht (normalerweise) aus einem HTML-File (view), einer TypeScript-File(Logik dahinter) und einem (S)CSS-File für das Layout und Design.

Jede Page muss zudem im File `src/app/app.module.ts` registriert werden, damit die App fehlerfrei funktioniert.

Die HTML-Seite ist normalerweise dreigeteilt:

```
<ion-header>
<!-- Hier kommt die Kopfzeile der App -->
</ion-header>
<ion-content>
<!-- Hier kommt der Inhalt der Seite -->
</ion-content>
<ion-footer>
<!-- Hier kommt die Fusszeile der App-->
</ion-footer>
```

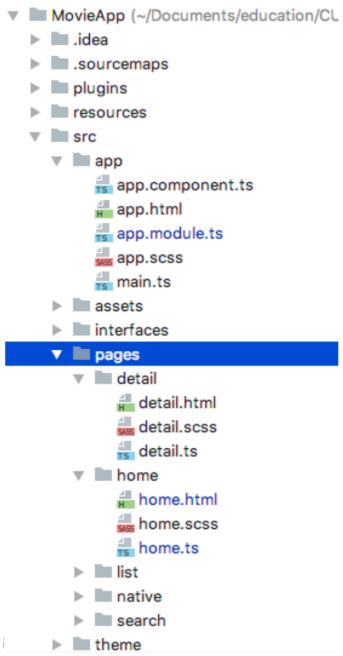


Abb. 11.1: Struktur einer Ionic-WebApp am Bsp. der MovieApp

Wie bereits erwähnt ist Ionic ein Angular-Framework auf und unterstützt somit auch Dependency Injections. Nachfolgend ein kleines Beispiel, wie man Daten asynchron über eine HTTP-GET Request mittels einer Callback-Methode (`.subscribe`) und einem Observable (`movieJson`) abrufen kann, über ein definiertes Interface (`Movie`) in ein Objekt parsen und anschliessend weiterverarbeiten kann.

```
import { HttpClient } from '@angular/common/http';
import { Movie } from '../../interfaces/Movie';
...
constructor( ... public httpClient: HttpClient ...) {
   ...
```

```
   let movieJson = this.httpClient.get('http://...');
   movieJson.subscribe(data => {
      let movie: Movie = <Movie>data;
      if(movie.Response == 'True'){
         //daten weiterverarbeiten
      }
   }
```

11.2 Cordova

Cordova ist eigentlich nichts anderes als ein Wrapper, der eine in Ionic geschriebene WebApp in eine Native App verpackt. Mithilfe bestimmter Angular-Pakete kann Cordova auch auf native Ressourcen wie die Kamera, Galerie oder Kontakte des Handys zugreifen.

Eine Ionic-App mit Cordova Integration (bei der Erstellung einer Ionic-App wird zu Beginn jeweils gefragt, ob man Cordova bereits integrieren will) kann ganz einfach von der CMD gestartet werden:

```
//Für Android. Zuerst jedoch einen Android-Emulator starten
ionic cordova run android --emulator

//für iOS. Zuerst Emulator starten
ionic cordova platform add ios
cordova run ios --emulator

//Alternativ kann die App auch direkt auf einem über USB oder WiFi angeschlossenem Gerät gesartet wereden
ionic cordova run android

//mit dem parameter --live-reload können ausserdem live reloads der App aktiviert werden, so dass die App sofort neu kompiliert wird, sobald Änderungen am Code gespeichert wurden
ionic cordova run android --live-reload
```