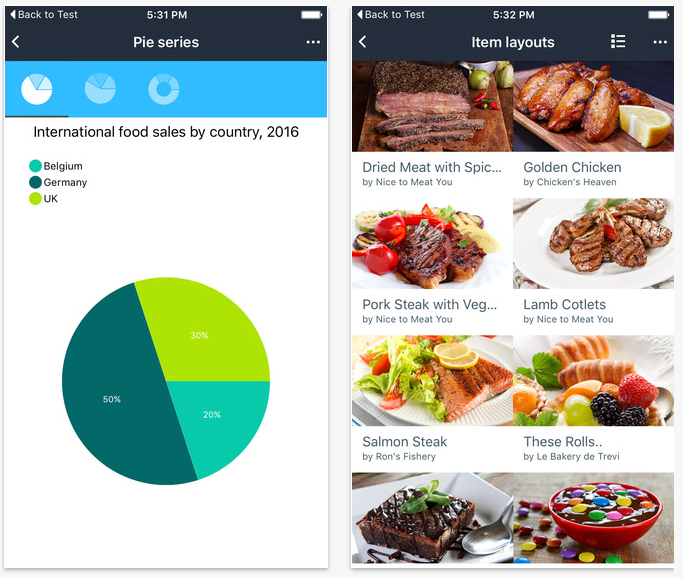
Das Beste aus allen Welten

Die Entwicklung mobiler Anwendungen steht in einem starken Spannungsfeld. Die Anforderungen an moderne Apps sind unter anderem eine ansprechende Ästhetik, ein plattformspezifisches Nutzererlebnis und natürlich bestmögliche Performance. Normalerweise werden hierzu eigenständige Apps für die beiden großen mobilen Betriebssysteme erstellt. Doch parallele Entwicklungen erzeugen gleichzeitig erhöhte Kosten. Eine Antwort darauf sind hybride Apps auf Basis von HTML und JavaScript. Doch leider sind durch die verwendeten Technologien Beschränkungen schwer zu vermeiden. Das Open-Source Framework NativeScript von Telerik schickt sich an, die letzten vorhandenen Grenzen einzureißen: echte native Apps auf Basis von JavaScript.

Durch NativeScript ist es möglich, direkt mit JavaScript Apps zu entwickeln. Diese Apps sind nicht mehr von Lösungen unterscheidbar, welche klassisch auf Basis von Objective-C bzw. Swift oder Java entwickelt worden sind. Sie verfügen über eine beachtliche Geschwindigkeit und verwenden die normalen Bedienelemente des jeweiligen mobilen Betriebssystems. Erfahren Sie in diesem Artikel die wichtigsten Fakten zu NativeScript und zu den Möglichkeiten, die sich durch die Kombination mit Googles Angular 2 aufzeigen.

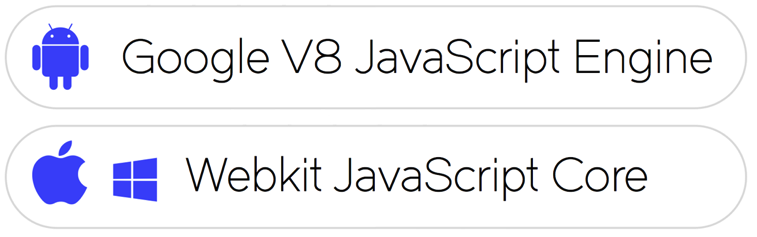
 Screenshot: Zwei NativeScript-Screens unter iOS

Was ist NativeScript?

NativeScript [https://goo.gl/LfT97E] ist ein Open-Source Framework zur Entwicklung von mobilen Apps. Neben der Programmiersprache JavaScript wird auch die JavaScript- Obermenge TypeScript direkt unterstützt. Aktuell stehen als Zielplattform sowohl Android als auch iOS zur Verfügung. Seit der der jüngsten Version 1.7 (März 2016) ist auch ein Support für universelle Windows-Plattform-Apps (UWP) hinzugekommen. Jene universellen Apps sind auf Windows Phone 10 und Windows 10 ausführbar. Telerik klassifiziert die neueste Zielplattform aber noch als "proof of concept". Zum aktuellen Zeitpunkt sind nur NativeScript-Apps für Android und iOS reif für den produktiven Einsatz.

Auf den ersten flüchtigen Blick scheint das Framework eine weitere Variante des hybriden Ansatzes zu sein. Als bekanntestes hybrides Framework ist hier Apache Cordova (ehemals PhoneGap genannt) zu nennen [https://goo.gl/QsTNul], welches den Einsatz von HTML und JavaScript zur App-Entwicklung salonfähig gemacht hat. Cordova lässt Web­anwendungen in einem Browser laufen und ermöglicht über Schnittstellen ebenso den Zugriff auf native Funktionen. Sofern Sie mit dem SPA-Framework AngularJS vertraut sind, steht Ihnen z.B. durch Einsatz des Ionic-Frameworks eine populäre und weit verbreitete Lösung zur Verfügung. Durch den Einsatz von vorbereiteten Themes und eigenem graphischen Geschick muten die fertigen Apps anschließend wie native Anwendungen an. Ein grundlegendes Problem ergibt sich jedoch prinzipiell immer. Eine hybride App ist und bleibt eine Website, welche nur den Anschein erweckt, dass es sich um eine native Anwendung handelt.

Im Gegensatz dazu reiht sich NativeScript in eine neue Disziplin ein. In dieser Disziplin geht es darum, JavaScript als vollwertige Programmiersprache für Apps zu etablieren. Weitere Frameworks, welche native Apps mit JavaScript ermöglichen, sind React Native von Facebook und Appcelerator Titanium. Bei allen drei Lösungen fällt der Umweg über HTML und den DOM schlicht weg. Die Frameworks ermöglichen die direkte Verwendung von nativen UI-Elementen aus der JavaScript-Umgebung heraus. Bei NativeScript für Android ist diese Umgebung Googles V8-Engine [https://goo.gl/GjcXrK]. Unter iOS sowie unter Windows kommt JavaScriptCore [http://goo.gl/8A5RVC] zum Einsatz.

Die verwendeten JavaScript Virtual Machines

Tipp: Sollten Sie skeptisch hinsichtlich der Performance sein, dann probieren Sie doch einfach die "Examples NativeScript" Apps im Google Play Store [https://goo.gl/aGzHvM] oder iTunes [https://goo.gl/EcF5zA] aus. Die App enthält eine Sammlung von graphischen Beispielen, welche die breiten Anwendungs­möglichkeiten von NativeScript demonstrieren.

Warum NativeScript?

Die technische Grundlage mag zwar spannend sein, doch im Projektalltag zählen praktische Gründe. Eine Reihe von Gegebenheiten spricht für den Einsatz von NativeScript.

Wiederverwendung von bestehende Skills: Das Erlernen einer neuen Programmiersprache zum Zwecke der App-Entwicklung ist anstrengend und aufwändig. Der Erwerb von Grundlagen einer Programmiersprache ist dabei noch das kleinere Problem. Der eigentliche Aufwand liegt im Detail. Es ist ein mühsamer und intensiver Prozess, bis ein Neueinsteiger tatsächlich alle Aspekte einer Programmierwelt kennt und sicher beherrschen kann. Während dieser Einarbeitung steht der Programmierer natürlich nicht mehr mit dem gewohnten Potential und der üblichen Kapazität zur Verfügung.

Wenn Sie hingegen schon einmal eine Anwendung für das Web entwickelt haben, so sind sie unweigerlich mit **JavaScript**, ggf. sogar mit TypeScript und auf jeden Fall mit **CSS** in Berührung gekommen. Damit steht Ihnen bereits ein großer Teil des notwendigen Wissens zur Verfügung. Das dedizierte Erlernen von Objective-C, Swift, Java und/oder C# entfällt. Sollten Sie sich zudem mit den neuesten Trends zur Webentwicklung beschäftigen, dann wird für Sie die Unterstützung von **Angular 2.0** von großem Interesse sein. Auf diese spannende Allianz von NativeScript und AngularJS werden wir im zweiten Teil dieses Artikels noch intensiver eingehen.

Wiederverwendung von bestehendem Code: Durch den Einsatz der Programmiersprache JavaScript bietet es sich an, bestehende Geschäftslogik oder Bibliotheken aus dem Internet weiter zu verwenden. Mit dem Repository **NPM** steht ein großer Fundus aus kurzen Schnipseln bis ganzen Bibliotheken zur Auswahl. Es gilt lediglich zu beachten, dass in NativeScript keinen DOM existiert. Das ist aber prinzipiell kein Problem, denn diese Restriktion gilt auch für alle üblichen Node.js-Pakete.

Wenn Sie zum Beispiel ein Datum formatieren wollen, dann können Sie dies mit der bekannten Bibliothek „moment“ tun. Nach einer Installation per „npm install moment“ steht Ihnen die Funktionalität wie üblich zu Verfügung:

// JavaScript

var moment = require("moment");  
var formattedTime = new moment().format("HH:mm:ss");

Verwendung eines NPM-Pakets

Es ist weiterhin möglich, bestehende **native Fremdbibliotheken** für Android und iOS anzusprechen. Das bedeutet, dass Sie nicht in der JavaScript- bzw. NativeScript Welt gefangen sind. Wenn es notwendig ist, können Sie auch sehr plattformspezifischen Code aufrufen. Von diesem Prinzip macht auch die Komponenten-Sammlung „Telerik UI for NativeScript“ [https://goo.gl/bNEFzx] Gebrauch. Die bestehenden Komponenten-Sammlungen sind hier vom Hersteller mit einem JavaScript-Wrapper vereinheitlicht worden.

Direkter Zugriff auf native APIs: Manchmal werden Sie einfach nicht drum herum kommen und Sie müssen doch tief in das Betriebssystem abtauchen. Für diese Fälle bietet NativeScript den direkten Zugriff auf native APIs aus JavaScript heraus an. Das ist ein großer Vorteil gegenüber React Native und Appcelerator, wo dies nicht so einfach möglich ist. Diesen Aspekt werden wir gleich noch einmal näher beleuchten.

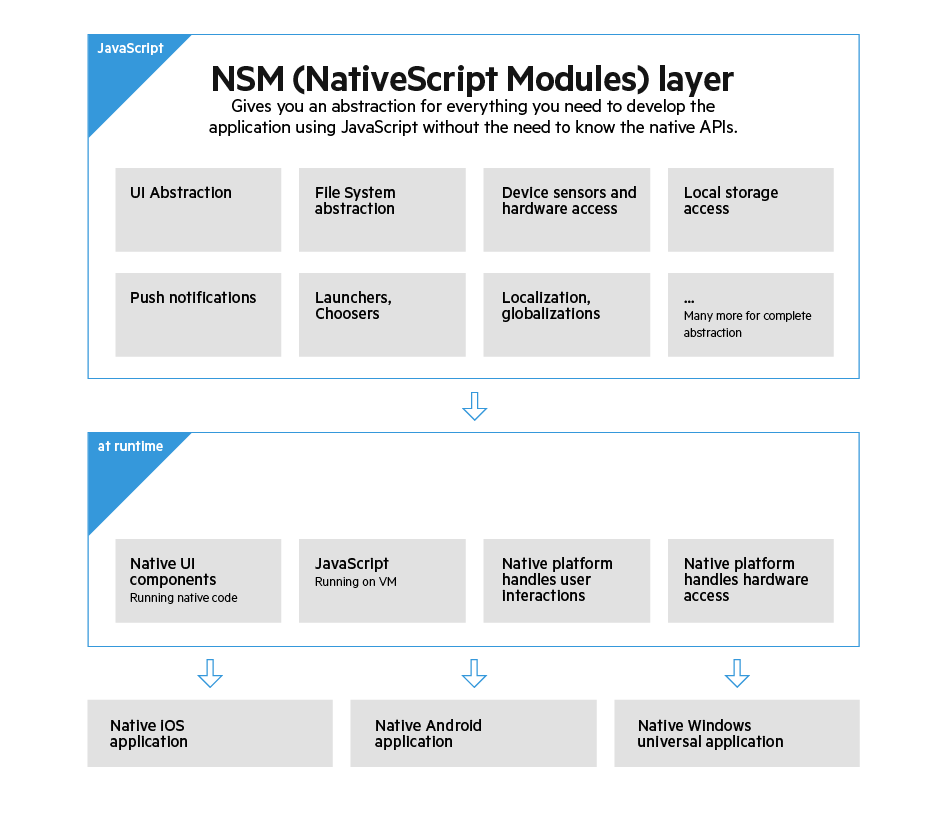
Open Source: Die Gretchenfrage in Sachen Software lässt sich bei NativeScript ohne Bauchschmerzen beantworten. Ja, das Framework ist Open-Source. Es steht unter der „Apache License, Version 2.0“ (ASLv2), welche die Kombination mit proprietärem Code erlaubt. Das NativeScript-Team selbst besteht aus rund 35 Köpfen. Das Team wiederum wird von einer aktiven Community unterstützt, welche auf Github [https://goo.gl/YHuNzn] die Fortentwicklung über Bug-Reports und Pull-Requests antreibt. Es ist problemlos möglich, einen kompletten NativeScript-Workflow mittels der „NativeScipt-CLI“ [https://goo.gl/BSiaUj] aufzubauen. Wenn Sie es jedoch etwas komfortabler wünschen oder kommerziellen Support benötigen, so gibt es von Telerik entsprechende kostenpflichtige Angebote. Lizenzpflichtig sind ebenso die erweiterte Komponenten-Sammlung „UI for NativeScript **Pro**“ und die integrierte Entwicklungsumgebung „Telerik Plattform by Progress“.

Hinter den Kulissen

Wie ist es möglich, unter Verwendung einer einzigen Code-Basis mehrere Plattformen anzusprechen? Die Grundlage hierfür bietet das NPM-Paket „**T**elerik **N**ative**S**cript Core Modules“ (kurz: „tns-core-modules“). Die darin enthaltenen Module bilden eine Abstraktionsschicht, welche spezifische Implantierungen für die unterstützten Plattformen enthalten.

„tns“ ist ebenso die Kurzschreibweise für den Kommandozeilenbefehl „nativescript“ der NativeScript-CLI.

Hier finden sich Module für die unterschiedlichen Aspekte der mobilen Entwicklung, von UI-Abstraktion zu Gerätesensoren bis hin zum Hardware-Zugriff.

  
Die Abstraktionsschicht in einer Übersicht

Da die NativeScript-Module in TypeScript geschrieben sind, bieten kompatible Entwicklungsumgebungen dank der Typ-Definition eine komfortable automatische Code-Vervollständigung und Syntax-Prüfungen an. Eigene Module können im selben Stil erstellt werden und das Software-Ökosystem erweitern (siehe hierzu [http://goo.gl/yxEGDB]).

UI-Abstraktion: Die Gestaltung von Oberflächen wird natürlich nicht durch eine Aneinanderreihung von JavaScript-Befehlen realisiert. Stattdessen wird die Oberfläche in einem spezifischen XML-Dialekt definiert. Tags wie „Button“, „TextField“, „DatePicker“ werden hierbei in die jeweiligen Oberflächenelemente umgewandelt:

<!-- XML -->

<Page>  
 <StackLayout>  
 <Label text="Name"></Label>  
 <TextField text="{{nameAttribute}}"></TextField>  
 <Button text="Press Me" tap="doSomething"></Button>  
 </StackLayout>  
</Page>"

Eine einfache Seite mit Text und Button

Beim „Build” der Anwendung wird jeder Tag durch das jeweilige native Äquivalent ersetzt. So wird etwa aus dem „TextField“ je nach Plattform das native Element.

* **android.widget.EditText** (bei Android) bzw.
* **UITextField** (bei iOS).

Plattformspezifischer Code

Es gibt Situationen, in denen der Aufruf von plattformspezifischem Code notwendig wird. Das kann zum Beispiel der Fall sein, wenn eine Funktionalität tatsächlich nur auf der jeweiligen Plattform existiert, wenn eine native Fremdbibliothek eingebunden werden soll oder das gewünschte Feature tatsächlich einfach noch nicht über ein Core-Modul implementiert wurde. Hier kommt der Zugriff auf die „Native API“ ins Spiel. Die native Welt kann dabei so aufgerufen werden, als ob es sich um normale JavaScript-Methoden handeln würde.

Als Beispiel soll das Datum der letzten Modifikation einer Datei ermittelt werden. Diese Funktionalität wird für Android und iOS gänzlich anders umgesetzt.

// JavaScript

var fileManager = NSFileManager.defaultManager();  
var attributes = fileManager.attributesOfItemAtPathError(path);  
var lastModifiedDate = attributes.objectForKey(this.keyModificationTime);

Zugriff auf das Datum der letzten Modifikation unter iOS

// JavaScript

var javaFile = new java.io.File(path);  
var lastModifiedDate = new Date(javaFile.lastModified());

Zugriff auf das Datum der letzten Modifikation unter Android

Das Beste an der gezeigten Syntax ist die Tatsache, dass sowohl Namespaces, als auch Attribute und Typen sowie die gesamten Konventionen bei der Benennung dem Pendant aus der Android- bzw. iOS-Dokumentation entspricht. Dasselbe gilt für Fremd­bibliotheken. So lässt sich mit geringem Aufwand ein Code-Fragment aus den Dokumentationen oder dem Netz per Copy-and-Paste zum Laufen bringen. Hinter den Kulissen verwendet NativeScript „Reflection“, um eine Liste von APIs aufzubauen, die auf der aktuellen Plattform zur Verfügung stehen und zum globalen Gültigkeitsbereich hinzugefügt werden. Die Details zu der verwendeten Technik werden von Telerik in einem ausführlichen Artikel beschrieben [http://goo.gl/5c0vcE].

Styling

Bei der Gestaltung von Webseiten trennt man Struktur und Design per Cascading Style Sheets (CSS). Das direkte Styling von Elementen würde viel zu unübersichtlich sein und zu redundanten Deklarationen führen. Ganz ähnlich verhält es sich bei der Gestaltung einer NativeScript-App. Das Framework folgt hierbei den Spezifikationen von CSS und verwendet die bekannte Syntax, das Prinzip der kaskadierten Regeln und eine Auswahl an Selektoren und Deklarationen.

Mittels CSS lassen sich einfache Dinge wie Hintergrundfarbe und Schriftgröße bis hin zu komplexen Animationen realisieren.

/\* CSS \*/

.small-label {  
 font-size: 20;  
 color: #284848;  
 horizontal-align: center;  
}

Anpassung von Hintergrundfarbe und Schriftgröße per CSS

/\* CSS \*/

.button1:highlighted {  
 animation-duration: 1s;  
 animation-name: test;  
}  
  
@keyframes test {  
 from { transform: none; }  
 20% { transform: rotate(45); }  
 50% { transform: rotate(50) scale(1.2, 1.2) translate(50, 0); }  
 100% { transform: rotate(0) scale(1.5, 1.5) translate(100, 0); }  
}

Einen Button per CSS animieren

Das Styling per CSS ist bei NativeScript im Vergleich zum Browser jedoch sehr limitiert. Es handelt sich hierbei um eine übersichtliche Teilmenge des bekannten Browser-CSS. Dies liegt nicht zuletzt auch darin begründet, dass die verfügbaren Eigenschaften der gemeinsame Nenner der drei verfügbaren Plattformen sein müssen.

Angular 2

NativeScript wurde so geschaffen, dass es ein reines JavaScript-Framework ist, welches das User-Interface per XML definiert und per CSS formatiert. Mit Version 1.0 war im Mai 2015 die erste Ausgabe für den produktiven Einsatz gestartet.

NativeScript loves Angular

Etwa zur selben Zeit nahm auch die Entwicklung der zweiten Version von AngularJS ordentlich Fahrt auf. Angular 2 wurde als komplette Neuentwicklung konzipiert. Im Gegensatz zum Vorgänger ist man nicht mehr nur auf den Browser festgelegt. Die Architektur von Angular ist darauf ausgelegt, vollkommen unabhängig von der eingesetzten Umgebung zu sein. Dies ermöglicht es, mit Angular 2 sowohl für Web als auch Desktop und Mobile zu entwickeln (siehe hierzu [https://goo.gl/96Opxf]). Die Synergien aus beiden Projekten sind offenkundig und so arbeiten die Teams von Telerik und Google seit nun fast einem Jahr zusammen, um eine nahtlose und stabile Integration zu schaffen. Dabei ist es eine gute Fügung, dass beide Projekte auf TypeScript setzen.

Lassen Sie uns doch einfach als Beispiel einen Login-Screen mit Angular 2 umsetzen. Sollten Sie noch nie mit Angular 2 in Berührung gekommen sein, wird Ihnen der offizielle „5-Minuten-Schnellstart“ [https://goo.gl/01oMIUtyp] einen guten Überblick verschaffen.

Zunächst benötigen wir eine Klasse mit dem Namen „UserService“, welche die Logik für die Anmeldung beinhaltet und die Zugangsdaten zum Backend senden soll:

// TypeScript

import {Injectable} from "angular2/core";  
  
@Injectable()  
export class UserService {  
 login(userName: String, password: String) {  
 return doSomeMagicHereAndReturnPromise();  
 }  
}

Der UserService kapselt die eigentliche Login-Funktionalität

Die wichtigsten Grundbausteine in Angular 2 sind Komponenten. Komponenten definieren ganze Seiten, einzelne UI-Elemente und Routen. Eine Komponente hat immer ein Template, welches aber im Falle von NativeScript kein normales HTML beinhaltet. Angular 2 sorgt auch gleich noch per „Dependendy-Injection“ (DI) für die Verdrahtung von UserService und Komponente, so dass wir uns um die Initialisierung des UserService nicht zu kümmern brauchen:

// TypeScript

import {UserService} from "./user.service";  
@Component({  
 selector: "my-app",  
 providers: [UserService],  
 templateUrl: "pages/login/login.html"  
})  
export class LoginPage {  
 userName: String;  
 password: String;  
  
 constructor(private \_userService: UserService) {  
 this.username = "user@example.org";  
 this.password = "password";  
 }  
  
 login() {  
 this.\_userService.login(this.username, this.password)  
 .subscribe(  
 () => doSomethingOnSuccessfulLogin (),  
 (error) => alert("Unfortunately we could not find your account.")  
 );  
 }  
}

Die Komponente LoginPage verknüpft die Geschäftslogik mit dem UI

Die Komponente „LoginPage“ agiert als Bindeglied zwischen UserService und dem Template. Auf die Methode „login“ sowie auf die beiden Eigenschaften „username“ und „passwort“ können wir nun im Template zugreifen.

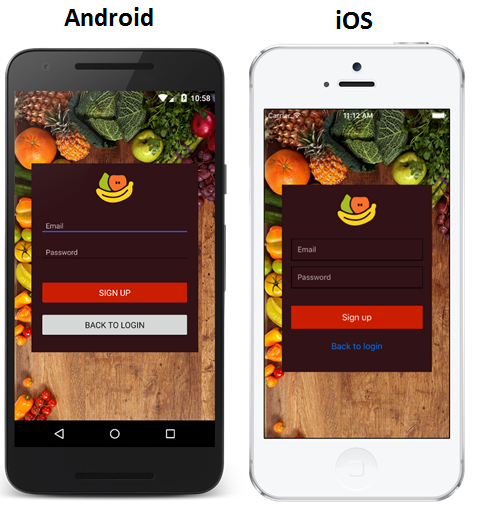
Zum Schluss müssen wir nur noch einen Screen im Template definieren. Der bereits vorgestellte XML-Dialekt zur Definition der Oberfläche kommt auch hier zum Einsatz. Die [eckigen] und (runden) Klammern sind der Template-Syntax von Angular 2 geschuldet. Es handelt sich um so genannte „Bindings“, welche als Verknüpfung mit der Komponente dienen. Da die Verwendung dieser Klammern kein valides XML ergeben würde, deklarieren wir das Markup lieber als HTML statt als XML. Praktisch gesehen ergibt sich hieraus aber kein wirklicher Unterschied. Die Bindings hauchen der Oberfläche Leben ein. Wird in dem Textfeld die Eingabe geändert, so wird die Änderung sofort in der Komponente publik gemacht. Eine Betätigung des Buttons ruft anschließend die Login-Methode auf.

<!-- HTML -->

<StackLayout>  
 <Image src="res://logo\_login" stretch="none" horizontalAlignment="center"></Image>  
 <TextField hint="Email" [(ngModel)]="username"></TextField>  
 <TextField hint="Password" secure="true" [(ngModel)]="password"></TextField>  
 <Button [text]="Sign up" (tap)="login()"></Button>  
 <Button [text]="Back to login"></Button>  
</StackLayout>

Die Komponente LoginPage verknüpft die Geschäftslogik mit dem UI

Nach der Anwendung von etwas zusätzlichem Styling ergibt sich ein fertiger Dialog. Fertig! Schon steht die erste Seite unserer Anwendung.

Die finale Login-Seite unter Android und iOS

Das hier gezeigte Beispiel ist ein angepasster Ausschnitt aus der offiziellen Demo-App „Groceries“, mit der man eine Einkaufsliste zusammenstellen und teilen kann. Die App kann einmal mit purem NativeScript [http://goo.gl/Vb9WYE] und einmal in Kombination mit Angular 2 [https://goo.gl/hBTDfk] nachprogrammiert werden. Hier wird auch die Installation der notwendigen Abhängigkeiten – vor allem das iOS SDK und das Android SDK – detailliert beschrieben.

Fazit

Der Einstieg in die App-Entwicklung mit NativeScript ist für einen Webentwickler nicht sonderlich schwer. Angular 2 gibt der Anwendung eine solide Architektur vor und sorgt dafür, dass Geschäftslogik und Oberflächenelemente sauber getrennt bleiben. NativeScript wiederrum liefert die Mechanismen, um mit nativen APIs und nativen UI-Elementen zu interagieren. Man erhält das Beste aus allen Welten: JavaScript bzw. TypeScript als Programmiersprache, Angular 2 als Oberflächenframework und das native Look-and-Feel, welches schlussendlich dem Anwender gefallen wird.



**Johannes Hoppe** ist selbstständiger IT-Berater, Softwareentwickler und Trainer für .NET und AngularJS. Er ist Leiter der .NET User Group Rhein-Neckar und unterrichtet als Lehrbeauftragter der DHBW Mosbach. Johannes schreibt über seine Trainings und Vorträge in seinem Blog (<http://blog.johanneshoppe.de/>). Demnächst wird sein Buch zu Angular 2 beim dpunkt.verlag erscheinen.

Twitter-Account: @JohannesHoppe



**Sebastian Witalec** ist Technical Evangelist bei Progress und hat über acht Jahre Erfahrung in der Softwareentwicklung. Sein Fokus liegt seit einigen Jahren auf der Cross-Plattform-Mobile-Entwicklung. Dadurch lernte er Apache Cordova und Native Script kennen. Er lebt in London und arbeitet dort eng mit verschiedenen Entwicklercommunitys zusammen.

Twitter-Account: @sebawita