KAFFEEKLATSCH

Das Magazin rund um Software-Entwicklung

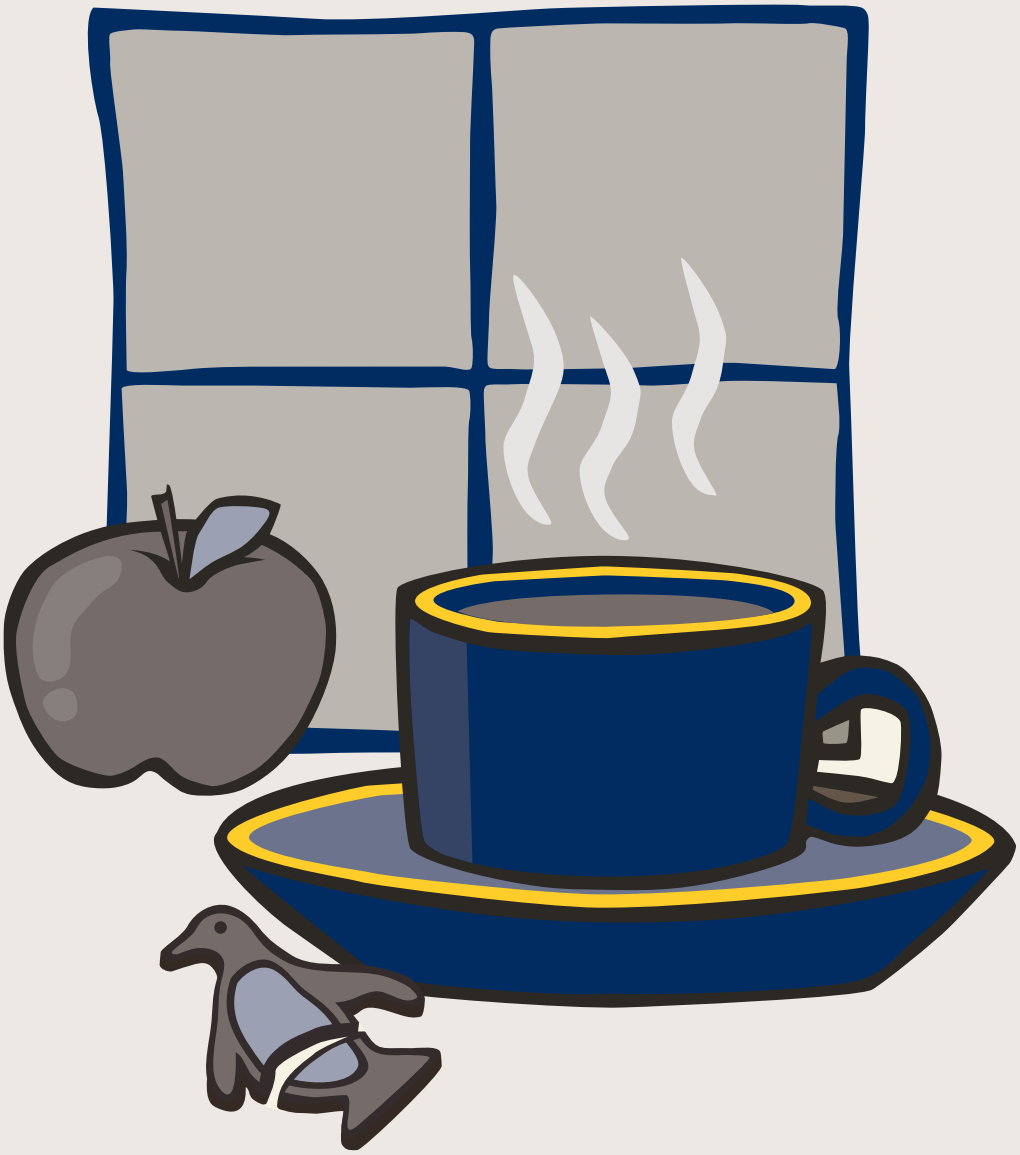
ISSN 1865-682X

04/2017

Crossplattform, aber Native!

Entwicklung mobiler Anwendungen mit NativeScript

von Werner Eberling



Crossplattform, aber Native!

Entwicklung mobiler Anwendungen mit NativeScript

von Werner Eberling

Bei der Entwicklung von mobilen Crossplattform-Anwendungen haben sich JavaScript-Ansätze bereits in vielen Varianten versucht. Mit NativeScript gibt es nun einen neuen Vertreter dieser Spielart, der den Bau von Anwendungen mit nativen Komponenten verspricht. Wie **NativeScript** funktioniert, wie damit aus einer Quelle Anwendungen für **Android** und **iOS** gebaut werden können und warum hier native wirklich nativ ist, wird in diesem Artikel beleuchtet.

Ob Sencha Touch, Cordova oder Xamarin, die Welt der mobilen Crossplattform-Ansätze ist vielfältig. In Anlehnung an die modernen Strömungen im Bereich Webentwicklung hat sich JavaScript als Sprache der Wahl hier inzwischen sehr weit verbreitet. Dies wurde auch dadurch gefördert, dass viele der bisherigen Ansätze auf WebViews als plattformübergreifende virtuelle Maschinen setzen. In diesen „Browsern ohne Browserfenster" werden dann die UI-Komponenten als Webkomponenten nachgebaut und per CSS so dargestellt, dass sie dem Original möglichst nahe kommen.

A new kid in town

Mit NativeScript [1] hat ein neuer Spieler diese Bühne betreten. Auch dieser Ansatz setzt auf JavaScript als Implementierungssprache, geht aber bezüglich Anwendungsausführung komplett andere Wege. Anstatt das JavaScript im abgespeckten Browser auszuführen und so zu tun, als ob native Komponenten verwendet würden, macht NativeScript genau das: Es verwendet direkt die nativen Komponenten und Darstellungsmechanismen der Zielplattform!

Um dies möglich zu machen, kommen auf den beiden von NativeScript unterstützen Plattformen Android und iOS spezielle JavaScript-Virtuelle-Maschinen (JSVMs) zum Einsatz. Konkret sind dies im Falle von Android Googles Chrome V8 [2], im Falle von iOS JavaScriptCore [3]. Die NativeScript-Anwendung wird in diesen JSVMs ausgeführt und erhält so Zugriff auf alle nativen APIs und Komponenten der Zielplattform. De facto wird hier also tatsächlich eine native Anwendung entwickelt, nur eben in einer anderen Sprache, als es die Zielplattform eigentlich vorsieht.

Wie in jedem VM-basierten Ansatz muss die Indirektion natürlich mit einer gewissen Einbuße an Anwendungsperformance bezahlt werden. Die Entwickler von NativeScript geben einen Performance-Nachteil von ca. 10% an [4], geloben aber gleichzeitig noch Besserung. In vielen Anwendungsfällen sollte dieser Nachteil aber vom Vorteil des Single-Source-Ansatzes deutlich aufgewogen werden.

Zwei Geschmacksrichtungen

NativeScript unterstützt sowohl die Verwendung von „purem" JavaScript (NativeScript Core) als auch das Zusammenspiel mit TypeScript und Angular (NativeScript with Angular). Da der Fokus in diesem Artikel auf NativeScript liegt, wird hier die Core-Variante mit reinem JavaScript betrachtet.

Probieren geht über Studieren

Wie sieht das Ganze nun im konkreten Anwendungsfall aus? Das Command-Line-Interface (CLI) von NativeScript basiert auf Node.js, sodass als allererstes eine entsprechende Node-Installation benötigt wird. Da hier echte, native Anwendungen erzeugt werden, werden neben dem eigentlichen Framework auch die Entwicklungsumgebungen der gewünschten Plattformen (Android und/oder iOS) sowie eine Reihe von Plug-ins zur automatisierten Nutzung des Android- bzw. iOS-SDK benötigt. Dies alles führt zu einem relativ umfangreicheren Installationsprozess, der im Detail auf der NativeScript-Webseite beschrieben ist [5].

Nach erfolgreicher Installation kann über den Aufruf von *tns create <AppName>* eine neue NativeScript-Anwendung erzeugt werden. Das Ergebnis ist ein entsprechendes Projektverzeichnis mit dem in Abb. 1 dargestellten Layout.

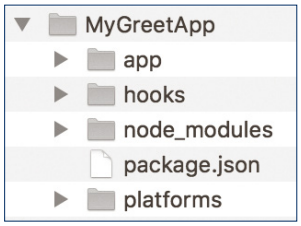


Abb 1: Toplevel-Struktur einer NativeScript Anwendung

Die eigentliche NativeScript-Anwendung befindet sich im Ordner app. Die weiteren Ordner enthalten u. a. die benötigten Abhängigkeiten (*node_modules*) sowie die plattformspezifischen Dateien, die für die Erzeugung der nativen Anwendungen benötigt werden (*platforms*).

In Anlehnung an die Beispiele aus [6] soll auch in diesem Artikel eine leicht abgewandelte Hello-World-Anwendung (*MyGreetApp*) dazu dienen, die wichtigsten Aspekte des Frameworks (Seitenaufbau, Navigation, Data-Binding und Anbindung von Business Logik) vorzustellen.

Seitenaufbau und Dateneingabe

Eine NativeScript-Anwendung besteht aus verschiedenen Seiten, die die einzelnen UI-Elemente enthalten. Diese Seiten werden über XML-Dateien definiert, die den Aufbau des Komponentenbaums der jeweiligen Seite beschreiben.

```
<Page loaded="onLoaded" class="page">
 <Page.actionBar>
   <ActionBar title="Eingabe" class="action-bar" />
 </Page.actionBar>
 <StackLayout class="p-20 text-center">
  <Label text="Wie heisst Du?" />
  <TextField hint="Dein Name" text="{{ name }}" />
  <Button text="weiter" tap="greet"
  class="btn btn-primary btn-active"/>
 </StackLayout>
</Page>
```

Listing 1: enter.xml

Die in XML verwendeten UI-Komponenten (hier: ActionBar, Label, TextField und Button) werden zur Laufzeit durch die nativen Komponenten der jeweiligen Plattform dargestellt. Somit passt sich die Anwendung automatisch dem Look-and-Feel der jeweiligen Umgebung an (vgl. Abb. 3 und 4). Das genaue Styling der einzelnen Komponenten kann durch CSS-Regeln beeinflusst werden. Diese können entweder anwendungsweit (*app.css*) oder pro Seite definiert werden (hier: *enter.css*). Zur Verwendung der Styles der jeweiligen Plattform können diese per Import aktiviert werden.

```
/* Light Theme */
@import 'nativescript-theme-core/css/core.light.css';
/* Platform Defaults */
@import url('~/platform.css');
/* Additional Styles*/

Page {
   background-color: white;
   font-size: 17;
}
TextField {
   margin: 10;
   padding: 10;
}
```

Listing 2: app.css (Ausschnitt)

Damit die Anwendung weiß, welche Seite sie beim Start darstellen soll, muss der entsprechende View noch in der zentralen *app.js*-Datei hinterlegt werden.

```
var application = require("application");
application.start({ moduleName: "views/enter" });
```

Listing 3: Registrierung des Start-Views in *app.js*

Anwendung im Simulator testen

Um die Anwendung im Android-Emulator oder im iOS-Simulator zu testen, genügt es, über die Kommandozeile den Befehl *tns run android* bzw. *tns run ios* auszuführen. Im Falle des Android-Emulators muss allerdings vorher ein passendes Geräteimage erstellt werden.

Solange der durch *tns run* gestartete Prozess läuft, werden Änderungen per Livesync direkt auf dem Emulator bzw. Simulator sichtbar. Es entfällt also das lästige „Re-Deployment“ der Anwendung.

Abb. 2: Eingabeseite unter Android

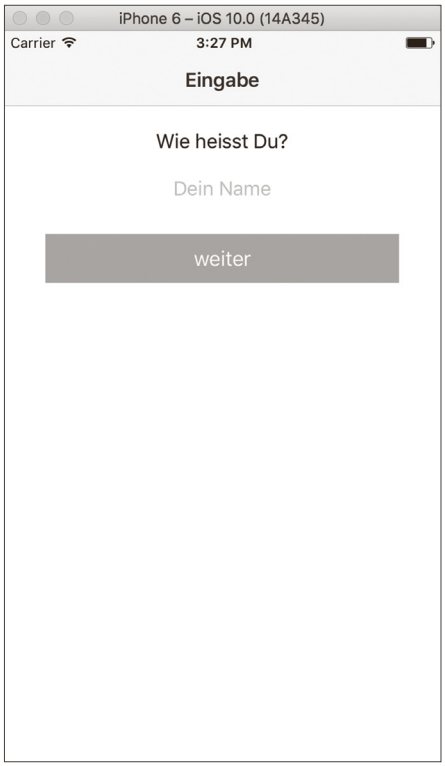


Abb. 3: Eingabeseite unter iOS

Navigation und Data-Binding

Die Anbindung der Oberfläche an die in JavaScript implementierte Businesslogik erfolgt über einen Code-Behind-Ansatz. Hierzu kann für jede Seitendefinition eine JavaScript-Datei mit identischem Namen angelegt werden (*enter.xml => enter.js*).

```
var frameModule = require("ui/frame");

exports.onLoaded = function(args) {
  page = args.object;
  page.bindingContext = localName;
};
exports.greet = function() {
   frameModule.topmost().navigate({
      moduleName: "views/greet",
      context: page.bindingContext
   });
};
```

Listing 4: Code behind in enter.js (Ausschnitt)

Die Bindung der in dieser Datei definierten Funktionen an die UI-Elemente erfolgt über die entsprechenden Event-Attribute der jeweiligen XML-Elemente:

```
<Page loaded="onLoaded" class="page">
…
<Button text="weiter" tap="greet" class=
"btn btn-primary btn-active"/>
```

Listing 5: Bindung von UI-Events an JS-Funktionen

Für die Verbindung von UI-Elementen an die passenden Datenstrukturen innerhalb der Businesslogik werden in NativeScript View-Models verwendet. Diese können entweder direkt in den JavaScript-Dateien der jeweiligen Seite definiert werden oder in einer eigenen JavaScript-Datei. Die Definition in einer eigenen Datei kommt vor allem für solche Modelle zum Einsatz, die zwischen mehreren Seite geshared werden.

Die Implementierung eines solchen View-Models erfolgt auf Basis des von NativeScript bereitgestellten Typs Observable.

```
var Observable = require("data/observable").Observable;

var localName = new Observable({
   name: "",
   reverse: ""
});

localName.addEventListener(Observable.
propertyChangeEvent, function (pcd) {

   if(pcd.propertyName === 'name'){
      localName.reverse = pcd.value.split('').reverse().join('');
   }
});
```

Listing 6:View-Model innerhalb von enter.js

Um in der Seitendefinition auf Attribute des Modells zugreifen zu können, muss dieses beim Laden noch an die Seite gebunden werden.

```
page.bindingContext = localName;
```

Sobald die Seite das Modell kennt, kann innerhalb der XML-Definition auf die einzelnen Attribute des Modells zugegriffen werden.

```
<TextField hint="Dein Name" text="{{ name }}" />
```

Die zweite Seite

Der Aufbau der zweiten Seite der Greeting-Anwendung bringt nichts überraschend Neues. Auch hier erfolgen Data-Binding und Navigationslogik über den bereits vorgestellten Ansatz.

```
<Page loaded="onLoaded" class="page">
  <Page.actionBar>
    <ActionBar title="Gruss" class="action-bar" />
  </Page.actionBar>
  <StackLayout class="p-20 text-center">
    <Label text="{{ 'Hallo ' + name + '!!!'}}" />
    <Label text="{{ 'Dein Name r&#252;ckw&#228;rts:
    ' + reverse }}" />
    <Image src="~/images/Beany.png" width=
    "200px"></Image>
    <Button text="zur&#252;ck" tap="goBack"
    class="btn btn-primary btn-active"/>
  </StackLayout>
</Page>
```

Listing 7: greet.xml

Der aus der Seitendefinition angesprochene JavaScript-Code befindet sich in der Datei *greet.js*, auf die hier nicht im Detail eingegangen wird.

Natives Anwendungsverhalten

Wirklich interessant ist wieder das Zusammenspiel beider Seiten innerhalb der Anwendung. Wird diese unter Android oder iOS gestartet, passen sich sowohl die Navigationselemente als auch der Übergang zwischen den Seiten den jeweiligen Standards der Plattform an.

So enthält die Anwendung unter iOS einen Backlink in der Titelleiste (Abb. 4) und die Seiten werden beim Wechsel seitlich eingeschoben. Unter Android wird für die Zurück-Navigation die entsprechende (virtuelle) Taste des Devices genutzt, also kein Backlink im Titel dargestellt (Abb. 5). Auch die Seite erscheint Android-typisch durch Überblendung, also ohne den Effekt des seitlichen Einschiebens.

Abb. 4: Greeting-Seite unter iOs

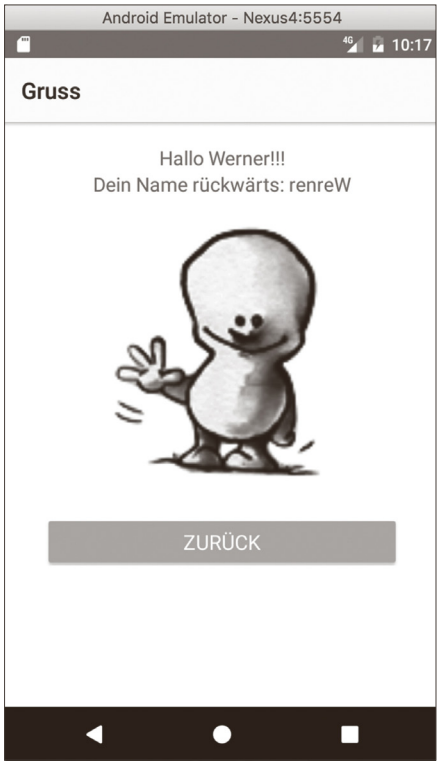


Abb. 5: Greeting-Seite unter Android

Zugriff auf Geräte-APIs

Auch der Zugriff auf die integrierten APIs der mobilen Endgeräte, auf denen die Anwendung läuft, ist mit NativeScript möglich. Grundsätzlich könnte über die Schnittstellen der virtuellen Maschinen direkt auf

die APIs zugegriffen werden. Zur Vereinheitlichung existieren aber spezielle NativeScript-Module, die als Adapter vor den speziellen APIs der jeweiligen Plattformen fungieren.

Sollen beispielsweise die Kontakte des Gerätes ausgelesen oder manipuliert werden, wird das Modul nativescript-contacts benötigt [7]. Dieses kann über die NativeScript-CLI der Anwendung hinzugefügt werden:

```
tns plugin add nativescript-contacts
```

Sobald das Modul installiert ist, kann per JavaScript auf die Kontakte des Gerätes zugegriffen werden.

```
var app = require( "application" );
var contacts = require( "nativescript-contacts" );

contacts.getAllContacts().then(function(args){
    console.log("getAllContacts Complete");
    console.log(JSON.stringify(args));
    /// Returns args:
    /// args.data: Generic cross platform JSON object, null if no
contacts were found.
    /// args.reponse: "fetch"
}, function(err){
    console.log("Error: " + err);
});
```

Listing 8: Beispiel zum Auslesen von Kontakten (aus [7])

Hierbei ist allerdings darauf zu achten, dass die Anwendung auch die entsprechenden Rechte besitzt. Da es sich hier um eine native Anwendung handelt, müssen diese sowohl im AndroidManifest, als auch in der entsprechenden pList für iOS definiert werden.

Hinzu kommt, dass unter Android seit Version 6 (Marshmallow) Rechte explizit angefordert werden müssen. Aber auch hierfür gibt es ein entsprechendes NativeScript-Plug-in [8], das dafür sorgt, dass auch durch diese Plattformspezifika der Single-Source-Ansatz nicht gebrochen wird.

Auslieferung

Ist die Anwendung fertig zur Auslieferung, so kann über den Aufruf *tns build <plattform> --release* der Bau eines deploybaren und in den Store upload-fähigen Artefakts angestoßen werden. Für das AndroidArtefakt muss hier zusätzlich noch der Pfad auf einen Keystore angegeben werden, der den privaten Schlüssel zum Signieren der Anwendung enthält. Wird der Schalter *--release* nicht angegeben, so wird lediglich eine Entwicklungsversion mit dem Postfix *debug* erstellt, die nicht in den Play- oder AppStore hochladbar ist.

Fazit

Der Ansatz JavaScript als Sprache für crossplattforme Entwicklung zu verwenden ist nicht neu. Bisher brachten die vorhandenen Ansätze aber die Beschränkungen des WebViews mit sich, die sich vor allem im Bereich der Performance und beim Look-and-Feel bemerkbar machten.

Durch die Verwendung des nativen Renderings adressiert NativeScript genau diese beiden Bereiche und punktet damit besonders bzgl. der Haptik der Anwendung. Ob der durch den VM-Ansatz immer noch vorhandene Performance-Nachteil in komplexeren Anwendungen ein Problem darstellt, bleibt abzuwarten. Dennoch ist NativeScript ein Schritt in die richtige Richtung und bei einfachen Anwendungen, die sowohl für Android als auch für iOS verfügbar sein sollen, sicher einen Versuch wert.

Das Beispiel auf GitHub

Die Anwendung, aus der die Codebeispiele entnommen sind, ist auf GitHub verfügbar [9].

Referenzen

[1] NativeScript *Build amazing iOS and Android apps with technology you already know,* https://www.nativescript.org

[2] Chrome V8 *Google's high performance, open source, JavaScript engine* https://developers.google.com/v8

[3] Developer *JavaScriptCore* https://developer.apple.com/reference/javascriptcore

[4] Telerik Blog *NativeScript: How does it work* http://developer.telerik.com/featured/nativescript-works

[5] NativeScript *Installation Guide* http://docs.nativescript.org/start/quick-setup

[6] *„Die rote oder die blaue Pille? Zwei mobile Webframeworks im Vergleich"*, Java Spektrum 01/2014, Sigs Datacom, Troisdorf 2014

[7] NativeScript *Modul: Contacts* https://www.npmjs.com/package/nativescript-contacts

[8] NativeScript *Modul: Permissions* https://github.com/NathanaelA/nativescript-permissions

[9] GitHub *MyGreetApp* https://github.com/wern/NativeScript_GreetSample

Werner Eberling ist Principal Consultant und Technical Lead bei der MATHEMA Software GmbH in Erlangen. Seit 1999 beschäftigt er sich mit verteilten Systemen. Sein Fokus liegt dabei auf CORBA und der JEE. Neben seiner Projekttätigkeit hält er Technologie-Trainings und ist als Sprecher auf nationalen und internationalen Konferenzen anzutreffen. Sein aktuelles Steckenpferd ist die Anwendungsentwicklung für mobile Endgeräte bzw. deren Anbindung an Enterprise-Systeme.

Werner Eberling ist u. a. Autor des Buches "Enterprise JavaBeans 3.1 – Das EJB-Praxisbuch für Ein- und Umsteiger", erschienen im Hanser Verlag.