Cross-Plattform-Apps mit NativeScript

Markus Schlichting, Karakun AG

Mobile Anwendungen für verschiedene Plattformen aus einer Codebasis anzubieten, ist nicht nur aus Kostensicht reizvoll. Die Fokussierung auf eine zentrale Technologie kommt auch dem Entwicklungsteam entgegen und nützt damit der Qualität. Mit NativeScript steht ein Framework bereit, mit dem Entwickler ihr vorhandenes Angular-/TypeScript-Know-how nutzen können, um aus der gleichen Codebasis iOS-und Android-Apps sowie Web-Applikationen bereitstellen zu können.

Das 2010 federführend von Google ausgerufene Paradigma „Mobile First" hat seine Spuren hinterlassen. Bei einer großen Anzahl neu aufgelegter Projekte wird zumindest von einer Mobile-Kompatibilität des webbasierten UI ausgegangen – wenn nicht sogar die parallele Implementierung einer zusätzlichen App angestrebt wird. Dies gilt insbesondere für Anwendungsfälle, die nicht ausschließlich beim Sachbearbeiter auf dem Desktop, sondern zum Beispiel auf dem Werksgelände, beim Kunden im Wohnzimmer oder im Führerhaus eines LKW ablaufen.

Rund 200 Milliarden App-Downloads im Jahr 2018 *[1]* allein in den öffentlichen Stores Google Play und Apple AppStore sprechen für sich: Mobiles und ihre Applikationen sind mindestens so allgegenwärtig, wie es prophezeit wurde.

Cross-Plattform

Aus dieser Situation heraus müssen Entwickler also drei Plattformen gleichzeitig bedienen: die beiden dominanten Mobile-Plattformen (Android und iOS) und das Web – denn die Frage, ob eine Mobile-App komplett durch eine PWA (Progressive Web App) ersetzt werden kann oder ob man voll auf native Apps setzen muss, ist noch lange nicht beantwortet.

```
<ActionBar title="SampleApp" class="action-bar">
</ActionBar>

<GridLayout rows="*" height="1500px">
        <StackLayout top="0" left="0" width="100%" height="100%">
            <ScrollView class="page">
                <StackLayout>
                    <CardView *ngFor="let item of data" class="card">
                        <app-card [item]="item"></app-card>
                    </CardView>
                </StackLayout>
            </ScrollView>
        </StackLayout>

</GridLayout>
```

Listing 1: Beispiel für UI-Deklaration via XML-Template

Aber wie können Entwickler dieser Anforderung möglichst effizient gerecht werden? Dass man mit drei Teams ebenso viele Apps parallel baut und zu gleichwertigen, auf die jeweilige Plattform optimierten Ergebnissen kommt, ist sehr zweifelhaft und in der Praxis oft widerlegt worden. Gleich mehrere Frameworks versprechen, zumindest das Erstellen der beiden mobilen Applikationen zu vereinheitlichen. NativeScript geht darüber hinaus und stellt in Aussicht, mit nur einer Codebasis alle drei Plattformen bedienen zu können.

NativeScript

Das seit 2015 verfügbare Open Source Framework NativeScript erlaubt es, native Anwendungen für iOS und Android aus einer Codebasis zu erstellen. Da diese Codebasis auf Webtechnologien aufbaut, kann der vorhandene Code auch zumindest in Teilen für eine Webanwendung genutzt werden. Die Vorteile bezüglich Produktivität und Wartbarkeit sind deutlich.

Anfang 2020 steht NativeScript in der Version 6.3 bereit und folgt einem klaren Versionsrhythmus: eine Major-Version pro Jahr, dazwischen vier Minor Releases. Als plattformunabhängige Programmiersprache wird JavaScript beziehungsweise TypeScript eingesetzt und es besteht die Möglichkeit, zusätzlich Frameworks wie Angular, Vue.js, Svelte *[2]* oder sogar React zur Erstellung der Programmlogik zu verwenden. Letzteres sogar trotz der Konkurrenz von ReactNative.

Dieser Artikel fokussiert sich auf die Verwendung von Angular, da dies der Technologie-Stack ist, der im Hinblick auf Code- und Know-how-Wiederverwendung überwiegend zum Einsatz kommt und auch vom NativeScript-Projekt selbst in den Vordergrund gestellt wird.

Funktionsweise

NativeScript verwendet also keine WebViews, um das UI mit DOM-Elementen darzustellen. Stattdessen nutzt es einen Glue-Layer, um aus Angular-Code native Steuerelemente des jeweiligen OS anzusteuern und anzuzeigen. Dies ist möglich, da das Angular Framework seit Version 2 vom DOM entkoppelt und somit nicht mehr auf Browser als Renderer beschränkt ist.

So kann man also vorhandenes Know-how aus der Webentwicklung nutzen und die App in TypeScript erstellen – oder umgekehrt im Kontext von NativeScript aufgebautes Wissen für die Webentwicklung weiterverwenden.

Die UI-Elemente werden wie in *Listing 1* gezeigt mit XML-Templates beschrieben. Die verwendeten Widgets werden dabei von NativeScript auf native Steuerelemente gemappt *(siehe Abbildung 1)*. Das Styling erfolgt mit CSS, die Verwendung von SCSS ist in der Toolchain bereits vorbereitet. Eine vollständige Übersicht ist online zu finden *[3]*.

Der Einstieg

Den Einstieg in die Welt von NativeScript kann man auf zwei Wegen finden: dem NativeScript Playground *[4]* oder der Einrichtung der NativeScript CLI.

Beim NativeScript Playground handelt es sich um eine Web-IDE, mit der insbesondere kleine Beispiele und Prototypen schnell erstellt und ausprobiert werden können. Der springende Punkt des Playground ist, dass über eine dazugehörige App der im Playground erstellte Code direkt auf einem Android- oder iOS-Smartphone ausprobiert werden kann. Dafür erfolgt das Nachladen des Codes auf das Smartphone über einen im Playground angezeigten QR-Code.

Der für die Entwicklung vollständiger Apps nachhaltigere Weg ist die Einrichtung der NativeScript CLI auf dem eigenen Rechner. Ist diese einmal vorhanden, bietet das Kommando `tns create` einen Wizard zum Erzeugen eines neuen NativeScript-Projekts und die Möglichkeit, auf verschiedene Projektvorlagen aufzubauen. Diese Variante ist deshalb nachhaltiger, da man früher oder später im Zuge von Zertifikatverwaltung und Builds für Google Play und Apple AppStore auf den Funktionsumfang der CLI angewiesen ist.

Funktionale UIs

Für eine praktische App braucht es natürlich auch eine Verbindung zwischen Darstellung, Daten und Logik. Um dies zu implementieren, kommen die bekannten Angular-Paradigmen zum Einsatz. Angular-

Abbildung 1: Projektion generischer UI-Elemente auf native Widgets

Module, Components und Services können voll verwendet und Templates und Styles innerhalb der Komponenten genutzt werden. Für bekannte Basisbibliotheken, wie Router, Forms oder den HTTP-Client, gibt es für NativeScript angepasste Versionen, um der Funktionsweise außerhalb des Browsers gerecht zu werden. Diese verwenden die bekannten APIs, sodass dem direkten Einsatz, ohne ein abgewandeltes API adaptieren zu müssen, wenig im Wege steht.

Das Layout von UI-Komponenten ist nach wie vor eine anspruchsvolle Angelegenheit. Für NativeScript stehen eine Reihe von Layouts zur Verfügung, die diese Aufgabe für die beiden mobilen Plattformen vereinheitlichen. Diese reichen vom simplen StackLayout bis hin zu einer Implementierung von Flexbox, wie es aus dem Browser bekannt ist.

Zugriff auf native APIs

Die meisten Apps möchten von nativen APIs Gebrauch machen. NativeScript bietet für die am häufigsten verwendeten APIs wie beispielsweise UI-Komponenten, Layouts, FileSystem oder HTTP-Client Abstraktionen, sodass direkt im TypeScript Schnittstellen verfügbar sind, unter denen die OS-spezifischen APIs gekapselt werden. Für viele Schnittstellen, die nicht von NativeScript selbst abgedeckt werden, sind durch die Community gepflegte Plug-ins verfügbar (zum Beispiel „nativescript-fingerprint-auth" zur Verwendung biometrischer Merkmale für die Authentifizierung). Möchte man darüber hinaus direkt mit nativen Funktionen arbeiten, ist dies ebenfalls direkt aus TypeScript heraus möglich, da NativeScript Type Definitions für die APIs der Plattformen bereitstellt. Um den Code je Plattform entsprechend kapseln zu können, bietet NativeScript selbst Hilfsmittel, wie in *Listing 2* gezeigt. So kann man auch Features verwenden, die nur in Android, aber nicht auf iOS verfügbar sind.

Sehr nützlich ist es, dass auch bei den Steuerelementen, für die mit NativeScript eine Abstraktion für beide Plattformen bereitgestellt wird (zum Beispiel Button, Label, TextField etc.), jederzeit auf plattformspezifische Attribute und Funktionen zugegriffen werden kann.

Für einzelne Statements, die sich in solche if-Statements kapseln lassen, ist dieser Ansatz praktikabel, wartbar und damit akzeptabel. Für komplexeren Code (zum Beispiel für die Einbindung von umfangreicheren Komponenten), sollte man den Code in separate Dateien oder sogar ein Plug-in auslagern. In beiden Fällen erfolgt die Trennung der plattformspezifischen Implementierung durch Namenskonventionen für die Quellcodedateien, die Suffixe zur Trennung verwendet:

- EnhancedItemSwitch.ios.ts
- EnhancedItemSwitch.android.ts
- EnhancedItemSwitch.ts

Die Datei ohne Plattform-Suffix definiert dabei nur die Struktur (`declare class`), die Implementierung erfolgt in den Dateien mit spezifischem Suffix. Beim Erstellen der Artefakte für die jeweilige Plattform wird dann nur der Code mit dem passenden Suffix herangezogen.

Wirklich nativen Code, also Objective-C/Swift für iOS oder Java/Kotlin für Android, zu verwenden, ist natürlich auch möglich, indem man ihn als Library einbindet.

Testing

Automatisierte Tests sind aus der Softwareentwicklung nicht wegzudenken und werden natürlich auch bei der Entwicklung mit NativeScript voll unterstützt. Zum einen kann Logik und isolierte Teile des Codes mit den aus der Webentwicklung bekannten Tools (zum Beispiel Karma/Jasmine) getestet werden. Zur Ausführung wird ein Emulator gestartet (oder auf ein angeschlossenes Gerät zurückgegriffen), um den Code tatsächlich auf der JavaScript-VM der Zielplattform auszuführen und so während des Tests möglichst nahe an dem Zielsystem zu sein. Außerdem kann so auch Code, der auf plattformspezifische APIs zurückgreift, getestet werden.

Darüber hinaus möchte man gerade für Software mit einer so großen Zahl von Installationszielen wie mobilen Apps End-to-End-Tests (E2E) verwenden. Für NativeScript existiert mit nativescript-dev-appium *[5]* eine Integration für Appium, einem Open-Source-Testautomatisierungswerkzeug für Mobile Apps. Dieses bietet viele nützliche Features wie beispielsweise *findBy*-Methoden zum Lokalisieren von Elementen oder das einfache Ausführen von Actions (*tap, click, doubleTap, hold*) und Gesten (*scroll, swipe, drag*).

Um nicht selbst einen sehr großen Gerätepark pflegen zu müssen, kann man über die Vereinheitlichung der E2E-Tests mit Appium auf einschlägige Cloud Provider zurückgreifen, die sowohl virtualisierte als auch ferngesteuerte physikalische Hardware zum Testen bereitstellen.

Codesharing App und Web

Das vollmundige Versprechen von NativeScript ist es, aus einer Codebasis nicht nur die mobilen Anwendungen bereitstellen zu können, sondern auch eine Webapplikation. Dieses gilt vor allem für den Code, der nicht fest an den DOM gekoppelt ist, denn so gibt es keinen Grund, warum er nicht in beiden Kontexten funktionieren sollte.

Um von diesem Level der Code-Wiederverwendung profitieren zu können, steht mit `nativescript-schematics` *[6]* ein Schema bereit, das Entwickler beim Aufsetzen eines Projekts für alle drei Zielapplikationen optimal unterstützt. Zentrales Element dabei ist wiederum die Dateinamenskonvention, die ähnlich der für die Trennung von plattformspezifischem Code aufgebaut ist:

```
import * as platform from "@nativescript/core/platform";
if (platformModule.isAndroid) {
    base64String = android.util.Base64.encodeToString(data,android.util.Base64.NO_WRAP);
} else {
    base64String = data.base64EncodedStringWithOptions(0);
}
```

Listing 2: Einfache Verwendung plattformspezifischer APIs

- **name.component.ts** - web + NativeScript shared
- **name.component.html** - web UI
- **name.component.tns.html** - NativeScript UI
- **name.component.css** - web stylesheet
- **name.component.tns.css** - NativeScript stylesheet

Möchte man nun noch separaten Code für Android und iOS einbauen, kann man die .ts-Datei noch aufsplitten:

- **name.component.ts**
- **name.component.android.ts**
- **name.component.ios.ts**

Einen umfassenden Einblick und guten Einstieg bietet die Dokumentation online *[7]*.

Entwicklungsumgebung

Es stehen offiziell unterstützte Plug-ins für VSCode und IntelliJ IDEA/Webstorm zur Verfügung, mit denen die Entwicklung schon sehr gut unterstützt wird. Außerdem wird die Entwicklung durch standardmäßig aktives „Hot Reload" unterstützt, bei dem nach dem Start der App veränderte Dateien direkt auf dem Gerät (oder Emulator) neugeladen und getestet werden können.

Publishing

Apps werden natürlich dafür entwickelt, dass sie veröffentlicht und verwendet werden. Die Erfahrung aus dem Projektalltag hat gezeigt, dass gerade in diesem Prozess die Automatisierung möglichst vieler Schritte und deren Abbildung in einem CI-System (Jenkins, GitLab, TeamCity…) sehr hilfreich ist und sich schnell bezahlt macht. In diesem Zusammenhand ist *fastlane* *[8]* unbedingt zu empfehlen, da es viele Schritte vom Zertifikatshandling über die Signierung bis hin zum Upload in den App- oder Play Store sehr gut kapselt und handhabbar macht.

Fazit

NativeScript ist ein erwachsenes Framework, das durch die dahinterstehende Firma Telerik/Progress mit Unterstützung einer starken Community intensiv weiterentwickelt wird. Für Interaktion mit der Community gibt es einen belebten Slack-Channel *[9]* und auf Issues auf GitHub wird schnell reagiert.

Was man sich dennoch bewusst machen sollte: Der Technologie-Stack, auf den man sich einlässt, ist nicht zu unterschätzen *(siehe Abbildung 2)*. Im Zweifel sollte man bereit sein, tief in die jeweilige Technologie einzutauchen, um die gestellten Anforderungen in letzter Konsequenz erfüllen zu können.

Dennoch macht sich gerade durch die Wiederverwendung von Know-how, Tools und Komponenten aus der Webentwicklung die Arbeit mit NativeScript bezahlt. Sowohl für Entwickler, die mit einem attraktiven Werkzeug arbeiten, als auch für Anwender, die zeitgemäße Lösungen erhalten, und auch Teams, die für den Entscheid über einen zielführenden Technologie-Stack einen sehr guten Kandidaten bekommen, ist NativeScript empfehlenswert.

Quellen

[1] *https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/*

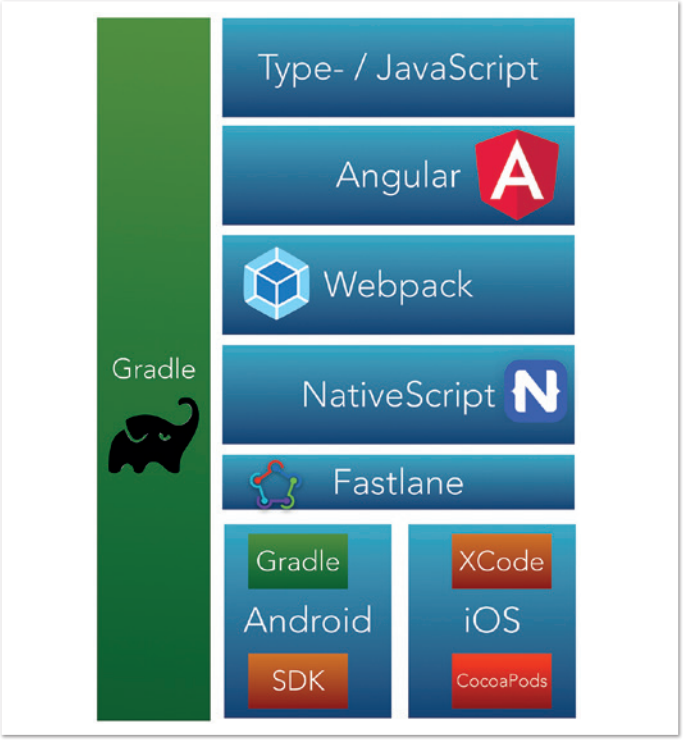

Abbildung 2: Technologie-Stack

[2] *https://svelte-native.technology/*
[3] *https://docs.nativescript.org/ui/styling*
[4] *https://play.nativescript.org/*
[5] *https://github.com/NativeScript/nativescript-dev-appium*
[6] *https://github.com/nativescript/nativescript-schematics*
[7] *https://docs.nativescript.org/angular/code-sharing/intro*
[8] *https://fastlane.tools/*
[9] *https://www.nativescript.org/slack-invitation-form*

Markus Schlichting
Karakun AG
markus.schlichting@karakun.com

Markus Schlichting ist Senior Software Engineer und Architekt bei Karakun AG. Software Engineering und -Architekturen, agile Methoden und Open-Source-Projekte zählen zu seinen Leidenschaften. Er liebt es, in den Bildschirmpausen Zeit mit seiner Familie zu verbringen oder beim Motorradfahren Frischluft zu tanken.