Programowanie aplikacji mobilnych (3)

Moduły i funkcje sprzętowe oraz system plików

Smartfony z mobilnymi systemami operacyjnymi zawierają cały szereg różnorodnych modułów i układów, których aż szkoda nie wykorzystać w projekcie sterowania urządzeniem elektronicznym. Akcelerometr, odbiornik GPS, karta pamięci, kamera czy choćby mikrofon, to komponenty, które można niezwykle łatwo użyć w aplikacji. W artykule pokażemy, jak skorzystać z różnych modułów funkcjonalnych smartfonu czy tabletu i podpowiemy, do czego można ich użyć.

Jak poprzednio, skorzystamy z wcześniej zainstalowanej platformy Cordova i związanymi z nią narzędziami. Nie modyfikujemy żadnego z nich – wszystkie używamy w wersjach takich, jak dwa odcinki kursu temu. Skorzystamy natomiast z dodatkowych pluginów do Cordovy, które stopniowo będziemy dodawać.

Po zaznajomieniu się z tą częścią kursu, czytelnik powinien umieć skorzystać z:

- akcelerometru,
- odbiornika GNSS (np. GPS),
- systemu plików, a w tym karty pamięci w telefonie,
- kamery,
- mikrofonu.

Jako przykład wciąż będzie nam służyć aplikacja do sterownia pracą bramy, którą nieco zmodyfikujemy, aby znaleźć uzasadnienie dla użycia wymienionych funkcji. Być może będzie to trochę „naciągane" zastosowanie, ale poznane metody programowania czytelnicy będą umieli użyć w dowolnych przypadkach.

Nowa aplikacja

W naszej nowej aplikacji wykorzystamy dwa urządzenia mobilne. Jedno będzie, tak jak poprzednio, służyło za zdalny kontroler – swoisty interfejs do sterownika napędu bramy. Jednakże tym razem sam sterownik napędu też wyposażymy w telefon dołączony przez Ethernet (Wi-Fi). Zacznijmy od wykonania nowej aplikacji zdalnego kontrolera (inteligentnego pilota do bramy) i sterownika (napędu) poleceniami:

cordova create pilot pl.com.ep.android Pilot
cordova create naped pl.com.ep.android Naped

wydanymi w katalogu **C:\kursEP**. W powstałych katalogach będziemy przygotowywać aplikacje.

Pamiętajmy, by do obu projektów dodać odpowiednią platformę docelową – tu będzie to Android.

Akcelerometr

Wykorzystanie akcelerometru jest dosyć proste i wymaga skorzystania z pluginu **org.apache.cordova.device-motion**. Instalujemy go poleceniem:

cordova plugin add org.apache.cordova.device-motion

wydanym z katalogu **c:\kursEP\pilot**. W naszym przypadku pobrała się wtyczka w wersji 0.2.11.

Dostęp do akcelerometru odbywa się z użyciem globalnego obiektu **navigator.accelerometer**, który działa praktycznie pod każdym mobilnym systemem operacyjnym. Obiekt ten ma trzy funkcje:

- **navigator.accelerometer.getCurrentAcceleration()** – pozwala na pobieranie aktualnego wskazania akcelerometru,
- **navigator.accelerometer.watchAcceleration()** – pozwala na uruchomienie cyklicznego pobierania wskazania akcelerometru,
- **navigator.accelerometer.clearWatch()** – usuwa zlecenie cyklicznego pobierania wskazania.

Po pomyślnym zadziałaniu funkcji otrzymujemy obiekt (przekazany jako argument funkcji uruchamianej po sukcesie odczytu z akcelerometru), który zawiera cztery liczby: przyspieszenia w trzech osiach oraz moment czasowy, w którym zostały one zmierzone. W praktyce, w wypadku telefonu leżącego nieruchomo lub poruszającego się ze stałą prędkością (względem Ziemi), wskazywane będzie przyspieszenie grawitacyjne, a więc pozwala to na wykrywanie „dołu" w trójwymiarowej przestrzeni. Dzięki temu możemy użyć akcelerometru np. do orientowania obiektów prezentowanych na ekranie, względem ziemi. Oczywiście, akcelerometr może też zostać użyty jako moduł, którego dane przesyłane byłyby do innego komponentu urządzenia elektronicznego. Ponieważ jest całkiem dokładny, a jego wskazania można odczytywać dosyć często, może posłużyć np. do realizacji jakiegoś rodzaju nawigacji inercyjnej lub prostszej rejestracji ruchu przedmiotu sztywno przyłączonego do telefonu. Warto tylko zwrócić uwagę na fakt, że nawet stabilnie leżący na biurku telefon będzie wskazywał ciągłe drgania, które wynikają zarówno z drgań przenoszonych z otoczenia, jak i z szumów sensora.

Cóż możemy zrobić z akcelerometrem w naszym przypadku? Właściwie to trudno znaleźć zarazem praktyczne, jak i sensowne zastosowanie dla niego, ale wzorując się na pilotach do nowoczesnych telewizorów, możemy spróbować sterować bramą gestami ręki. Przyjmijmy, że aby otworzyć bramę, trzeba energicznie machnąć telefonem w lewo; aby zamknąć – machnąć w prawo. Od razu zaznaczamy jednak, że redakcja nie

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

Listing 1. Zawartość pliku index.html pilota, umożliwiającego sterowanie bramą gestami ręki, z użyciem akcelerometru

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="format-detection" content="telephone=no" />
        <meta name="msapplication-tap-highlight" content="no" />
        <!-- WARNING: for iOS 7, remove the width=device-width and height=device-height attributes. See 
https://issues.apache.org/jira/browse/CB-4323 -->
        <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1, minimum-scale=1, 
width=device-width, height=device-height, target-densitydpi=device-dpi" />
        <link rel="stylesheet" type="text/css" href="css/index.css" />
        <title>Pilot</title>
    </head>
    <body>
        <script type="text/javascript" src="cordova.js"></script>
        <script type="text/javascript" src="js/jquery-2.1.3.min.js"></script>
        <script type="text/javascript" src="js/index.js"></script>
        <div style="display:table;width:100%;height:100px;background-color:green;font-size:xx-large;text-
align:center">
            <div id="accelerometerStartButton" style="display:table-cell; vertical-align: middle;">MACHAJ</
div>
        </div>
        <DIV id="accelerometerStopData" style="width:100%height:50px;display:none"><DIV 
id="accelerometerResults"></DIV><BUTTON id="accelerometerCloseButton">OK</BUTTON></DIV>
    </body>
</html>
```

ponosi odpowiedzialności za zniszczenia wynikłe z machania telefonem.

W tym celu, podobnie jak w poprzedniej części kursu, tworzymy sobie duży przycisk, po którego naciśnięciu przejdziemy do sterowania bramą za pomocą gestów. Do przycisku (warstwa DIV o id równym **accelerometerStartButton**, utworzona w pliku **index.html**) przypisujemy w funkcji **app.assignButtons()** dwa zadania:

- pokazanie warstwy o id **accelerometerData**, w której dla podglądu będziemy wyświetlać aktualne wskazania akcelerometru (korzystamy z polecenia jQuery **.show()**),
- uruchomienie funkcji **navigator.accelerometer.watchAcceleration()**, która co 10 milisekund będzie sprawdzała wskazanie akcelerometru, wyświetlała je na warstwie **accelerometerData** i w razie potrzeby wysyłała odpowiednie polecenie do napędu bramy.

W warstwie **accelerometerData** umieściliśmy też mały przycisk **accelerometerStopButton**, którego naciśnięcie wstrzymuje monitorowanie stanu akcelerometru i ukrywa warstwę z wynikami. Aby poprawnie zatrzymać cykliczne odczytywanie akcelerometru, trzeba posłużyć się identyfikatorem zadania (**watchID**), zwróconym przez funkcję **navigator.accelerometer.watchAcceleration()**.

W efekcie, co 10 milisekund (o ile wydajność telefonu na to pozwala), odczytywany jest stan akcelerometru, a następnie wskazania wpisywane są na warstwę **accelerometerResults**, znajdującą się wewnątrz warstwy **accelerometerData**, z użyciem polecenia jQuery **.html()**. Dane z akcelerometru otrzymujemy w funkcji **app.onSuccess()**, w postaci obiektu z czterema parametrami: x, y, z, timestamp. Trzy pierwsze to przyspieszenia w trzech osiach, a wartość timestamp obejmuje znacznik czasu z chwili dokonania pomiaru. My korzystamy z osi X telefonu i sprawdzamy, czy wartość przyspieszenia w tej osi przekracza zadane granice. Trzeba uwzględnić fakt, że naturalne przyspieszenie ziemskie będzie wynosiło około 10 (±0,3) w osi, którą telefon skierowany jest w dół – np. w osi Z, gdy telefon leży płasko na biurku. Aby więc samo obrócenie telefonu bokiem do dołu nie wzbudzało mechanizmu bramy, za graniczne wartość przyspieszenia wybraliśmy 12 i –12. I rzeczywiście, wybranie takich wartości umożliwia wykrywanie energicznych ruchów ręką w lewo lub prawo w sposób dosyć jednoznaczny. Do wysłania komendy do napędu używamy ponownie napisanego wcześniej polecenia **app.sendGateCommand()**. Kod pliku HTML tego programu został umieszczony na **listingu 1**, a kod pliku JavaScriptowego – na **listingu 2**.

Odbiornik GNSS (GPS)

Jeśli udało nam się obsłużyć akcelerometr, skorzystanie z odbiornika GPS nie powinno stanowić żadnego wyzwania. Plugin Cordovy do obsługi geolokalizacji zawiera funkcje analogiczne do tych używanych w akcelerometrze, a różnice sprowadzają się jedynie do formatu i rodzaju danych wyjściowych. Nie będzie też problemu z wymyśleniem zastosowania dla satelitarnego pozycjonowania – załóżmy, że nasza brama znajduje się na naszym ranczo w Teksasie i chcielibyśmy, by za każdym razem, gdy podjeżdżamy do rancza naszym pickupem, brama się automatycznie otwierała. A gdy ranczo opuszczamy, by brama się bezpiecznie zamykała.

Zaczynamy od instalacji wtyczki do obsługi odbiorników GNSS w Cordovie:

cordova plugin add org.apache.cordova.geolocation

Powyższe polecenie wydajemy w katalogu **c:\kursEP\pilot**. W naszym przypadku pobrała się wtyczka w wersji 0.3.12.

Odczytu danych z odbiornika GPS dokonujemy cyklicznie, co 30 sekund, z użyciem funkcji **navigator.geolocation.watchPosition()**. Jeśli odczyt się powiedzie, porównujemy długość i szerokość geograficzną, z zadanymi ręcznie koordynatami naszej bramy. Przyjęliśmy, że jeśli znajdziemy się w obszarze o kształcie kwadratu (w rzeczywistości, gdyby przeliczyć na metry, nie będzie to kwadrat, gdyż stopnie odpowiadają różnym odległościom, zależnie od tego, na jakiej szerokości geograficznej są mierzone), o boku o długości 0,04°, którego to środek kwadratu zlokalizowany jest tam, gdzie brama, to nasz telefon ma wysłać polecenie otwarcia jej. Gdy wykroczymy za ten obszar, brama powinna zostać zamknięta. Co więcej, aby nie przesyłać niepotrzebnie poleceń do napędu bramy, zapamiętujemy ostatnio przesłane polecenie i wydajemy nowe tylko wtedy, gdy będzie ono inne niż ostatnio wysłane. Korzystamy w tym celu ze zmiennej **app.GPScommand()**. Zmodyfikowany fragment kodu JavaScript znalazł się na **listingu 3**, a w pliku **index.html** dopisaliśmy jedynie linijkę:

```
<DIV id="GPSdata"></DIV>
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

Listing 2. Zawartość pliku index.js pilota, umożliwiającego sterowanie bramą gestami ręki, z użyciem akcelerometru. Dla skrócenia, usunięto z niego standardowe komentarze Cordovy

```
var app = {
    initialize: function() {
        this.bindEvents();
    },
    bindEvents: function() {
        document.addEventListener('deviceready', this.onDeviceReady, false);
    },
    onDeviceReady: function() {
        app.receivedEvent('deviceready');
        app.assignButtons();
    },
    assignButtons: function() {
        $(document).ready(function() {
            $('#accelerometerStartButton').click(function() {
            $('#accelerometerData').show();
                var options = { frequency: 10 };
                this.watchID = navigator.accelerometer.watchAcceleration(app.onSuccess, app.onError, 
options);
            });
            $('#accelerometerStopButton').click(function(){
                navigator.accelerometer.clearWatch(this.watchID);
                $('#accelerometerData').hide();
            })
        });
    },
    watchID:null,
    onSuccess: function(acceleration) {
        $('#accelerometerResults').html('X: ' + acceleration.x + '<BR/>' +
          'Y: ' + acceleration.y + '<BR/>' +
          'Z: ' + acceleration.z + '<BR/>' +
          'T: ' + acceleration.timestamp);
            if (acceleration.x>12)
            app.sendGateCommand("zamknij");
            else if(acceleration.x<-12)
            app.sendGateCommand("otworz");
    },
    onError:function() {
        alert('Błąd akcelerometru!');
    },
    sendGateCommand: function(command) {
        var adres="http://192.168.0.6/brama.php";
        $.post( adres, { akcja: command, kod: device.uuid }, null,'json')
            .done(function( data ) {
                var response = "Polecenie " + command + " przesłane pomyślnie"
                if ('stan' in data) response+=". Stan bramy to: "+data.stan;
                alert(response);
                var d = new Date();
                var olddata=window.localStorage.getItem("data");
                if (olddata!=null)
                    info=olddata+command+" : "+d.toString()+"\n";
                else
                    info=command+" : "+d.toString()+"\n";
                window.localStorage.setItem("data",info);
            })
            .error(function( data ) {
                alert( "Nie udało się wysłać polecenia " + command);
            });
    },
    receivedEvent: function(id) {
        console.log(,Received Event: , + id);
    }
};
app.initialize();
```

Warto zwrócić uwagę na format danych, pozyskiwanych z odbiornika GPS. W naszym przykładzie skorzystaliśmy tylko z długości i szerokości geograficznej, ale moduł zwraca więcej parametrów. Oto wszystkie z nich:

- **latitude** – szerokość geograficzna wyrażona w stopniach, w postaci liczby rzeczywistej,
- **longtitude** – długość geograficzna wyrażona w stopniach, w postaci liczby rzeczywistej,
- **altitude** – wyrażona w metrach wysokość nad ziemią (elipsoidą Ziemi),
- **accuracy** – dokładność pomiaru długości i szerokości geograficznej, wyrażona w metrach,
- **altitudeAccuracy** – dokładność pomiaru wysokości, wyrażona w metrach,
- **heading** – kierunek poruszania się, wyrażony w stopniach liczonych zgodnie z ruchem wskazówek zegara, począwszy od północy.
- **speed** – aktualna szybkość poruszania się urządzenia względem ziemi, wyrażona w metrach na sekundę.

Korzystając z danych odbiornika GPS można np. rejestrować ruch pojazdów i nawigować nimi do celu.

System plików

Obiecaliśmy pokazać, jak korzystać z kamery, ale aby przykład taki był kompletny, wypada wcześniej zademonstrować sposób używania systemu plików w urządzeniu mobilnym. I choć zagadnienie to wydaje się proste, wcale takie nie jest. Istnieje kilka sposobów na zapisywanie danych w telefonie, z czego każdy z nich cechuje się innymi ograniczeniami. Ponadto nawet jeśli dostęp dotyczy bezpośrednio systemu plików, jest on ograniczony przez uprawnienia – zabezpieczenia narzucone aplikacjom. Poszczególne programy mają swoje własne przestrzenie plików, a do tego trzeba jeszcze wziąć pod uwagę, że różne urządzenia mobilne w odmienny sposób dają dostęp do kart pamięci – o ile w ogóle je obsługują. W końcu, na plikach operujemy z poziomu języka HTML5, czyli tak jakby przez przeglądarkę internetową, co wprowadza dodatkowe utrudnienia.

W poprzedniej części kursu pokazaliśmy jak przechowywać dane w tzw. pamięci LocalStorage. Choć to bardzo wygodna i prosta w użyciu metoda, ma liczne ograniczenia. Jest powolna, a do tego dostępna przestrzeń jest względnie mała. W końcu można tam przechowywać tylko ciągi znaków, a nie dane binarne. Jeśli zaszłaby

potrzeba zapisu danych binarnych w LocalStorage, konieczne byłoby zastosowanie kodowania transportowego – np. Base64. A to tym bardziej zmniejsza ilość danych, które można zgromadzić.

W praktyce większość przeglądarek (to od nich zależy limit pamięci) pozwala na zapis około 5 MB danych w LocalStorage, ale wiele zależy od wersji przeglądarki. Android Browser 4.3 udostępnia tylko 2 MB, a Chrome 40 i IE 9, 10 czy 11 już po 10 MB, ale nie jest to przestrzeń gwarantowana. Ponieważ JavaScript stosuje w tym przypadku kodowanie UCS-2 (podobne do UTF16), każdy znak zapisywany w LocalStorage zajmuje 16 bitów. To oznacza, że w 5 MB pamięci możemy zapisać około 2,5 mln znaków. Jeśli jednak zastosujemy kodowanie Base64 do zapisu danych binarnych w LocalStorage, to zmarnujemy bardzo dużo pamięci, bo każde 6 bitów właściwych danych będzie zajmowało 16 bitów w LocalStorage, a więc jeden bajt zajmie ponad 21 bitów. To znowu oznacza, że w ten sposób, przy limicie 5 MB w LocalStorage, zmieścimy tylko 1,875 MB danych binarnych, zakodowanych w Base64 (pomijając już kwestie znaków końca linii).

Z powyższych rozważań chyba dosyć jasno wynika, że wszędzie tam, gdzie planowana jest jakaś rejestracja multimediów, warto skorzystać z systemu plików urządzenia mobilnego, którego wydajność nierzadko będzie kilkadziesiąt razy większa niż w przypadku LocalStorage, nawet jeśli robimy to z poziomu przeglądarki internetowej.

File API

Jak się dostać do plików? Najprostszym, a zarazem całkiem uniwersalnym sposobem jest skorzystanie z File API – zestawu obiektów i powiązanych z nimi metod, który wprowadzono w HTML5. Do manipulowania tymi obiektami służy język JavaScript, dzięki czemu nie musimy wykraczać poza narzędzia używane przy programowaniu z Cordovą.

Na File API składają się następujące interfejsy (udostępniane klasy obiektów):

- **FileList**, który reprezentuje tablicę plików.
- **Blob**, który pozwala odnosić się do danych w pliku w surowej postaci, zanim zostaną one przekonwertowane na formaty danych, stosowane w JavaScripcie.
- **File**, który pozwala pozyskać informacje na temat konkretnego pliku, takie jak jego nazwa, czy data modyfikacji.
- **FileReader**, który pozwala na odczyt danych z interfejsów **File** lub **Blob** oraz udostępnia zdarzenia do przechwytywania, związane z zakończeniem odczytu. Dzięki temu odczyt realizowany jest asynchronicznie.
- **URL scheme**, który pozwala na tworzenie wirtualnych adresów, które odnoszą się do danych istniejących aktualnie w pamięci przeglądarki, ale niekoniecznie zapisanych na serwerze. Można np. utworzyć plik w jakimś tymczasowym katalogu, niedostępnym dla przeglądarki poprzez protokół HTTP, i używając interfejsu **URL scheme** stworzyć link, który skorzysta z tego pliku (poprzez JavaScript) i będzie go przeglądarce udostępniał, jak każdy inny plik poprzez HTTP.

Oprócz powyższych istnieją też ich odpowiedniki synchroniczne, ale nie będziemy się nimi zajmować w tej części kursu.

Interfejsy te są częścią standardu obsługiwanego przez wszystkie nowoczesne przeglądarki, a więc będą działać też na praktycznie wszystkich mobilnych systemach operacyjnych. Problem w tym, że nie ma w nich narzędzi do zapisu do pliku. Te znajdują się w innych API, np. File System API, które oparte jest na FileWriter API. Niestety, nie jest to część oficjalnego standardu i choć **FileWriter** korzysta z File API, to organizacje standaryzujące wycofały się w kwietniu 2014 roku z rozwijania tej części HTML5 i nie zalecają stosowania API FileWriter (a więc też File System) do nowych projektów. W praktyce interfejsy te są zaimplementowane w pełni tylko w przeglądarce Chrome, a więc będą działać w aplikacjach mobilnych Cordovy na Androidzie. Istnieje

Listing 3. Zmodyfikowana funkcja app.onDeviceReady oraz funkcje dodane na potrzeby sterowania bramą w oparciu o geolokalizację

```
onDeviceReady : function() {
    app.receivedEvent('deviceready');
    app.assignButtons();
    $(document).ready(function() {
        this.watchGPS = navigator.geolocation.watchPosition(app.GPSSuccess, app.GPSError, { timeout: 3000 });
    });
},
GPSSuccess : function(position){
    var gate={
        lat:31.565592,
        lng:-97.507760,
        }
    $("#GPSdata").html('Szerokość: '+position.coords.latitude+'<br/>'+'Długość: ,+position.coords.
longitude+'<br/>');
    if (this.GPScommand!="otworz"){
        if ((position.coords.latitude<gate.lat+0.02)
            &&(position.coords.latitude>gate.lat-0.02)
            &&(position.coords.longtitude<gate.lng+0.02)
            &&(position.coords.longtitude>gate.lng-0.02)){
                this.GPScommand="otworz";
                app.sendGateCommand("otworz");
            }
    } else if (this.GPScommand!="zamknij"){
        if ((position.coords.latitude>gate.lat+0.02)
            &&(position.coords.latitude<gate.lat-0.02)
            &&(position.coords.longtitude>gate.lng+0.02)
            &&(position.coords.longtitude<gate.lng-0.02)){
                this.GPScommand="zamknij";
            app.sendGateCommand("zamknij");
        }
    }
},
GPSError : function(error) {
    alert('code: '+error.code+'\n'+'message: '+error.message+'\n');
},
GPScommand : null,
watchGPS : null,
```


Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

alternatywne API działające wyłącznie na Firefoksie – DeviceStorage API, czyli bezużyteczne w aplikacjach Cordovy, kompilowanych na Androida.

Cóż można zrobić w takiej sytuacji? Można ewentualnie sięgnąć po jeszcze inny sposób zapisu danych na urządzeniu mobilnym – np. IndexedDB lub WebSQL, ale są to mechanizmy bardziej bazodanowe, których użycie wymaga dodatkowej wiedzy i na razie nie będziemy ich opisywać. Ponadto żaden z nich nie jest w pełni kompatybilny z każdym systemem operacyjnym. Dlatego mimo wszystko wykorzystamy File API wraz z **FileWriterem** – tak zalecają twórcy Cordovy, a można dodać, że inżynierowie Google twierdzą, że nie zamierzają wycofywać wsparcia dla FileWriter API z przyszłych wersji swojej przeglądarki.

File System API i FileWriter API w praktyce

Spróbujmy zapisać coś do pliku, gdzieś w pamięci smartfona, a następnie postaramy się odczytać zapisane dane. Najpierw – dla uproszczenia – będzie to zwykły tekst, ale zapis danych binarnych nie będzie znacznie trudniejszy.

Proces zapisu, począwszy od uruchomienia aplikacji, będzie obejmował następujące kroki:

1. Wskazanie katalogu z plikiem.
2. Określenie nazwy pliku, a w razie potrzeby utworzenie go.
3. Otwarcie i znalezienie końca aktualnej treści.
4. Przygotowanie danych do zapisu.
5. Zapis przygotowanych danych.

Stworzymy oddzielną funkcję do przygotowania pliku, która będzie się uruchamiać po włączeniu aplikacji i oddzielną do zapisywania konkretnych treści, wywoływaną w razie potrzeby. Ponieważ w trakcie działania obu tych funkcji mogą wystąpić podobne błędy, związane z dostępem do systemu plików, przygotujemy też trzecią funkcję do obsługi tych błędów – w praktyce do wyświetlania stosownych komunikatów.

Po katalog z systemu plików sięgamy z użyciem polecenia:
window.resolveLocalFileSystemURL() z trzema argumentami:

- ścieżką katalogu,
- funkcją, która wykona się w wypadku sukcesu,
- funkcją, która wykona się w wypadku błędu.

Jako parametr pierwszej z funkcji przekazywany jest obiekt typu **DirectoryEntry** (zdefiniowany w File System API), który wskazuje na żądany katalog. Ma on kilka przydatnych metod, a w tym funkcję **getFile()**. Funkcja **DirectoryEntry.getFile()** przyjmuje cztery parametry:

- nazwę pliku,
- opcje (tablicę ze zmiennymi **create** i **exclusive**, które mogą przyjąć wartości **true** lub **false**),
- funkcję, która wykona się w przypadku sukcesu, a której parametrem będzie obiekt typu **FileEntry**,
- funkcję, która wykona się w przypadku błędu.

U nas, w przypadku sukcesu, uzyskany obiekt typu **FileEntry** przypisujemy do zmiennej **app.myFile**, którą sobie przy okazji dodatkowo zdefiniowaliśmy i która nam ułatwi dostęp z innych funkcji do znalezionego lub utworzonego pliku. Naturalnie, jeśli wywołamy **getFile()** z opcją **create** równą **true**, to w plik o podanej nazwie zostanie utworzony w danym katalogu, jeśli wcześniej taki nie istniał.

Powyższe fragmenty realizują pierwsze dwa z pięciu podanych wcześniej kroków i umieścimy je w jednej funkcji, wywoływanej po uruchomieniu aplikacji. Kroki 3-5 umieścimy w drugiej funkcji, w której skorzystamy z FileWriter API. Plik otwieramy do zapisu z użyciem metody zdefiniowanej dla obiektu typu **FileEntry**, czyli naszej zmiennej **app.myFile**. Metoda ta to **createWriter()**, która przyjmuje jako parametry dwie funkcje:

- pierwsza wywołuje się w przypadku sukcesu, a jej parametrem jest obiekt typu **FileWriter**, powiązany z wcześniej znalezionym obiektem typu **FileEntry**,
- druga uruchamiana jest w przypadku wystąpienia błędu.

FileWriter ma wiele ważnych właściwości, które warto wymienić:

- **readyState** – określa stan obiektu i pozwala sprawdzić, czy zapis do pliku aktualnie trwa, czy też się zakończył. Może przyjmować trzy wartości: **INIT**, **WRITING** lub **DONE**,
- **fileName** – nazwa pliku, którego dotyczy obiekt **FileWriter**,
- **length** – aktualna długość pliku,
- **position** – aktualna pozycja wskaźnika w pliku, czyli miejsce, od którego w razie potrzeby będą zapisywane dane,
- **error** – obiekt zawierający informację o ewentualnych błędach **FileWritera**,
- **onwritestart** – wskazanie funkcji, wywoływanej w przypadku rozpoczęcia zapisu do pliku,
- **onwrite** – wskazanie funkcji, wywoływanej w momencie pomyślnego zakończenia zapisu do pliku,
- **onabort** – wskazanie funkcji, wywoływanej w przypadku celowego przerwania zapisu (np. z użyciem polecenia **abort()**),
- **onerror** – wskazanie funkcji, wywoływanej w przypadku pojawienia się błędu zapisu,
- **onwriteend** – wskazanie funkcji, wywoływanej zarówno w przypadku pomyślnego, jak i błędnego zakończenia zapisu.

FileWriter ma też cztery ważne metody:

- **abort()** – wspomniana wcześniej funkcja przerywająca trwający zapis,
- **seek()** – funkcja ustawiająca wskaźnik w pliku do konkretnej pozycji (konkretnego bajta),
- **truncate()** – funkcja skracająca plik do określonej długości,
- **write()** – funkcja zapisująca dane do pliku (w oparciu o aktualną pozycję wskaźnika).

W celu skoczenia do końca pliku, tak aby można było do niego dopisywać dodatkową treść, należy skorzystać z polecenia **FileWriter.seek(FireWriter.length);**. A gdy mamy już przygotowane do zapisu dane, korzystamy z polecenia **FileWriter.write(data);**.

Pomiędzy dwoma powyższymi poleceniami trzeba przygotować dane. W tym celu korzystamy z interfejsu **Blob**, o którym wspomnieliśmy przy wymienianiu składników File API. Musimy utworzyć obiekt typu **Blob**, z zawartością, którą chcemy zapisać do pliku, a najprostszym na to sposobem jest skorzystanie z konstruktora, używając polecenia **new Blob(tresc, parametry);**. Treść musi być w postaci tablicy i jeśli chcemy po prostu zapisać tekst, który dotąd był przechowywany w postaci stringa, wydajemy polecenie np. następujące

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

Tabela 1. Popularne typy treści MIME	
Typ MIME	**Opis**
application/EDI-X12	dane w formacie EDI X12, zdefiniowane w RFC 1767
application/EDIFACT	dane w formacie EDI EDIFACT, zdefiniowane w RFC 1767
application/javascript	kod JavaScript, zgodny z RFC 4329
application/octet-stream	dowolne dane binarne, które nie pasują do żadnego innego formatu (zdefiniowano w RFC 2046)
application/ogg	multimedialny strumień, zgodny z RFC 3534
application/xhtml+xml	XHTML, zgodny z RFC 3236
application/x-shockwave-flash	plik Adobe Flash
application/json	treść w formacie JSON, zgodna z RFC 4627
audio/mpeg	plik audio w formacie MPEG, w tym MP3 (zgodny z RFC 3003)
audio/x-ms-wma	Plik Windows Media Audio, opisany w Microsoft KB 288102
audio/vnd.rn-realaudio	plik w formacie RealAudio
audio/x-wav	plik WAV
image/gif	grafika w formacie GIF (zgodna z RFC 2045 lub RFC 2046)
image/jpeg	grafika w formacie JPEG (zgodna z RFC 2045 lub RFC 2046)
image/png	grafika w formacie PNG
image/tiff	grafika w formacie TIFF, zgodna z RFC 3302
image/vnd.microsoft.icon	grafika w formacie ICO
multipart/mixed	wieloczęściowa wiadomość, np. e-mail, zgodny z RFC 2045 i RFC 2046
multipart/alternative	wieloczęściowa wiadomość, np. e-mail, zgodny z RFC 2045 i RFC 2046
multipart/related	E-mail zgodny z RFC 2387
text/css	kod CSS, zgodny z RFC2318
text/html	treść HTML, zgodna z RFC 2854
text/plain	zwykłe dane tekstowe, zgodne z RFC 2046 i RFC 3676
text/xml	treść XML, zgodna z RFC 3023
video/mpeg	wideo w formacie MPEG-1, zgodne z RFC 2045 i RFC 2046
video/mp4	wideo w formacie MP4, zgodne z RFC 4337
video/quicktime	wideo w formacie QuickTime
video/x-ms-wmv	wideo w formacie Windows Media Video, zdefiniowanym w Microsoft KB 288102

var blob = new Blob([str], parametry);. Parametry przekazywane są w postaci tablicy asocjacyjnej i obejmują dwie cechy:

- **type**, która określa rodzaj zawartości pliku,
- **endings**, która może przyjąć wartość **transparent** lub **native**, i w ten sposób określa sposób oznaczania końca linii.

Parametr **type** może przyjąć jedną z pośród różnych wartości, określanych mianem Internet Media Type, czyli tzw. MIME. Dla plików tekstowych będzie to **text/plain**, dla plików graficznych może to być np. **image/jpeg**, dla obiektów wideo będzie to np. **video/mpeg**, a dla dowolnych danych binarnych: **application/octet-stream**. Najbardziej popularne typu MIME zostały zebrane w **tabeli 1**, a pełną, aktualną ich listę można znaleźć pod adresem *http://goo.gl/lwt3sh*.

Wtyczki File i File System Roots

Mając powyżej opisaną wiedzę moglibyśmy już przystąpić do napisania kodu, zapisującego dowolny plik w pamięci telefonu, gdybyśmy tylko wiedzieli, gdzie dokładnie chcemy (i możemy go zapisać). Ponieważ aplikacja napisana w Cordovie, działa w nieco innym środowisku, niż zwykła przeglądarka internetowa, musimy skorzystać z wtyczki, która faktycznie pozwoli nam na dostęp do systemu plików telefonu, a ponadto umożliwi nam odnalezienie się w strukturze katalogowej, typowej dla danego systemu operacyjnego.

Zapis do pliku zrealizujemy we wspomnianym wcześniej sterowniku napędu bramy (projekt Naped). Wchodzimy do katalogu z projektem (**C:\KursEP\naped**) i instalujemy plugin **org.apache.cordova.file**. W naszym przypadku pobrała się wersja 1.3.3.

Wtyczka ta udostępnia szereg stałych, które odpowiadają ścieżkom w systemie plików, przeznaczonym do poszczególnych celów. Wartości stałych mają postać *file:///scieza/do/katalogu* i mogą zostać przekonwertowane do obiektu **DirectoryEntry** za pomocą wcześniej wymienionego polecenia **window.resolveLocalFileSystemURL()**. Lista stałych została podana w **tabeli 2**, przy czym niektóre z nich są specyficzne dla konkretnych systemów operacyjnych. W **tabeli 3** znalazły się informacje o ścieżkach odpowiadających poszczególnym stałym, w konkretnych systemach operacyjnych.

Warto też wspomnieć o pluginie **org.apache.cordova.file-system-roots**, który pozwala w nieco inny sposób na dostęp do niektórych katalogów systemu plików. Po zainstalowaniu tej wtyczki, w aplikacji pojawia się obiekt **cordova.filesystem**, który ma dwie podstawowe metody:

- **getFilesystem()** – pozwala na pobranie konkretnej ścieżki na podstawie jednej ze stałych, zależnych od systemu operacyjnego,
- **getDirectoryForPurpose()** – pozwala na pobranie ścieżki, która w danym systemie operacyjnym jest przeznaczona do określonego celu.

Potrzebne do określenia ścieżek stałe, używane przez plugin **org.apache.cordova.file-system-roots** w przypadku Androida to:

- **files**: katalog danych aplikacji w pamięci urządzenia,
- **files-external**: katalog danych aplikacji w pamięci zewnętrznej,
- **sdcard**: katalog główny pamięci zewnętrznej, jeśli jest zainstalowana,
- **cache**: katalog na dane tymczasowe, w pamięci wewnętrznej urządzenia,

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

- **cache-external**: katalog na dane tymczasowe, w pamięci zewnętrznej urządzenia,
- **root**: katalog główny systemu operacyjnego,
- **documents**: podkatalog Documents/ w ścieżce odpowiadającej stałej **files**.

W systemie iOS dostępne stałe to:

- **library**: katalog Library/ aplikacji,
- **library-nosync**: katalog Library/ aplikacji, niepodlegający synchronizacji z iCloud,
- **documents**: katalog Documents/ aplikacji,
- **documents-nosync**: katalog Documents/ aplikacji, niepodlegający synchronizacji z iCloud,
- **cache**: katalog Cache/ aplikacji,
- **app-bundle**: katalog, w którym znajduje się główny plik aplikacji,
- **root**: katalog główny systemu operacyjnego.

Metoda **getDirectoryForPurpose()** korzysta natomiast ze stałych:

- **data**,
- **documents**,
- **cache**,
- **temp**,
- **app-bundle**

oraz pozwala na przekazanie parametrów informujących, czy żądamy dostępu do katalogu synchronizowanego oraz czy ma to być katalog dostępny tylko dla danej aplikacji.

Właściwy kod zapisu do pliku

To już wszystko, czego potrzeba do zapisania danych w systemie plików aplikacji mobilnej. Dla testu tworzymy prosty kod JavaScript w pliku **index.js** projektu **Naped**, definiując:

- funkcję **app.prepareFile()**, która lokalizuje i ew. tworzy plik po uruchomieniu aplikacji,
- funkcję **app.writeFile(str)**, która zapisuje konkretny łańcuch znaków do pliku,
- funkcję **app.fail(e)** do obsługi błędów związanych z systemem plików,
- zmienną **app.myFile**, zawierającą obiekt typu **FileEntry**, w którym będziemy dokonywać zapisu.

Powstały kod (jedynie nowe funkcje oraz funkcja **app.onDeviceReady()**) znalazł się na **listingu 4**. Do tego należy wytworzyć odpowiedni plik **index.html**, w którym załadujemy bibliotekę jQuery (i skopiujemy ją w odpowiednie miejsce) oraz umieścimy jakiś przycisk o identyfikatorze **saveButton**. Kod JavaScriptu będzie powodował zapis określonej treści do pliku, właśnie po naciśnięciu tego przycisku. Należy też pamiętać o dodaniu platformy android do projektu, jeśli nie zrobiliśmy tego wcześniej, po czym można skompilować kod i uruchomić kod.

Ponieważ skorzystaliśmy ze stałej **cordova.file.dataDirectory**, na Androidzie uruchomiona aplikacja, po każdym naciśnięciu przycisku „zapisz", zapisuje do pliku */data/data/pl.com.ep.android/files/plik.txt* słowo „nacisniento" wraz z aktualną datą.

Odczyt z pliku

Gdybyśmy w dowolnym momencie chcieli odczytać wcześniej zapisany plik, musimy posłużyć się interfejsem **FileReader**, będącym częścią File API. Szczęśliwie jest to zestaw funkcji kompatybilny z praktycznie każdym mobilnym systemem operacyjnym.

FileReader działa podobnie do **FileWritera**. Potrzebujemy mieć obiekt identyfikujący plik, a ponieważ będziemy chcieli odczytywać ten sam plik, do którego dokonywaliśmy zapisu, skorzystamy z już stworzonego obiektu **app.myFile**, inicjowanego podczas uruchamiania aplikacji. Dodamy dodatkową funkcję **app.readFile()**, która będzie uruchamiana po naciśnięciu nowego przycisku (dopisanego do **index.html**).

Dotąd korzystaliśmy z metody **FileEntry.createWriter()**, ale obiekt **FileEntry** ma oprócz tego jedną inną metodę i cztery atrybuty. Kompletny zestaw metod i atrybutów **FileEntry** to:

- **createWriter()** – pokazana wcześniej metoda do otwierania pliku do zapisu,

Tabela 2. Stałe zdefiniowane w ramach pluginu org.apache.cordova.file, określające ścieżki w systemach plików różnych mobilnych systemów operacyjnych, wraz z informacją, na których platformach są dostępne. Stałe te są właściwościami dostępnymi w ramach obiektu cordova.file

Stała	Opis	Dostępność		
		Android	iOS	BB10
applicationDirectory	Katalog, w którym zainstalowana jest aplikacja. Tylko do odczytu	TAK	TAK	TAK
applicationStorageDirectory	Katalog główny indywidualnego środowiska aplikacji. Wszystkie zapisane w tym miejscu dane są dostępne tylko dla konkretnej aplikacji	TAK	TAK	TAK
dataDirectory	Podkatalog na dane, umieszczony w katalogu głównym indywidualnego środowiska aplikacji. Zapisywane tam dane są trwale przechowywane	TAK	TAK	TAK
cacheDirectory	Katalog na dane tymczasowe, które w razie potrzeby mogą być samodzielnie usunięte przez system operacyjny, ale nie muszą. To dobre miejsce do przechowywania większych ilości danych, potrzebnych w trakcie aktualnego działania programu, które to dane nie będą później potrzebne	TAK	TAK	TAK
externalApplicationStorageDirectory	Przestrzeń udostępniona dla aplikacji, zlokalizowana na nośniku zewnętrznym	TAK	NIE	NIE
externalDataDirectory	Przestrzeń udostępniona na dane aplikacji, zlokalizowana na nośniku zewnętrznym	TAK	NIE	NIE
externalCacheDirectory	Przestrzeń na dane tymczasowe aplikacji, zlokalizowana na nośniku zewnętrznym	TAK	NIE	NIE
externalRootDirectory	Katalog główny pamięci zewnetrznej - najczęściej karty SD	TAK	NIE	TAK
tempDirectory	Katalog na dane tymczasowe, które mogą być usunięte przez system. Niemniej aplikacja powinna usuwać je samodzielnie, jeśli nie są potrzebne	NIE	TAK	NIE
syncedDataDirectory	Katalog na pliki przeznaczone do synchronizacji - np. poprzez Apple iCloud	NIE	TAK	NIE
documentsDirectory	Katalog na pliki aplikacji, które mogą być używane przez inne programy (np. pliki PDF)	NIE	TAK	NIE
sharedDirectory	Katalog na pliki dostępne dla wszystkich aplikacji	NIE	NIE	TAK

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

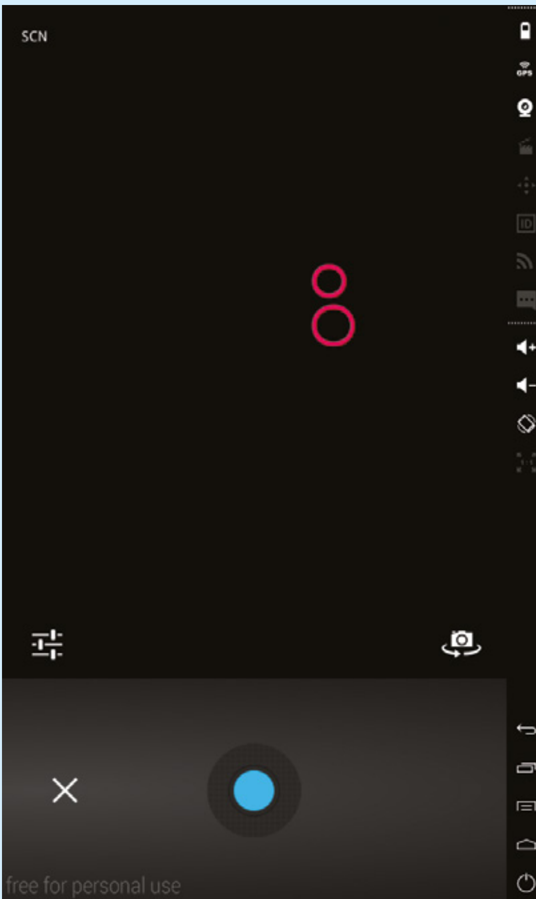
Tabela 3. Ścieżki odpowiadające poszczególnym stałym obiektu cordova.file, wraz z informacjami o sposobie ich traktowania przez system

Ścieżka	cordova.file.*	Zapis	Trwałość	Czyszczenie	Synchronizacja	Prywatność
Android						
file:///android_asset/	applicationDirectory	NIE	n.d.	n.d.	n.d.	TAK
/data/data/<app-id>/	applicationStorageDirectory	TAK	n.d.	n.d.	n.d.	TAK
cache	cacheDirectory	TAK	TAK	TAK	n.d.	TAK
files	dataDirectory	TAK	TAK	NIE	n.d.	TAK
<sdcard>/	externalRootDirectory	TAK	TAK	NIE	n.d.	NIE
Android/data/<app-id>/	externalApplicationStorageDirectory	TAK	TAK	NIE	n.d.	NIE
cache	externalCacheDirectry	TAK	TAK	NIE	n.d.	NIE
files	externalDataDirectory	TAK	TAK	NIE	n.d.	NIE
iOS						
/var/mobile/Applications/<UUID>/	applicationStorageDirectory	NIE	n.d.	n.d.	n.d.	TAK
appname.app/	applicationDirectory	NIE	n.d.	n.d.	n.d.	TAK
Documents/	documentsDirectory	TAK	TAK	NIE	TAK	TAK
NoCloud/	dataDirectory	TAK	TAK	NIE	NIE	TAK
Cloud/	syncedDataDirectory	TAK	TAK	NIE	TAK	TAK
Caches/	cacheDirectory	TAK	TAK	TAK	NIE	TAK
tmp/	tempDirectory	TAK	NIE	TAK	NIE	TAK
BlackBerry 10						
file:///accounts/1000/appdata/<app id>/	applicationStorageDirectory	NIE	n.d.	n.d.	n.d.	TAK
app/native	applicationDirectory	NIE	n.d.	n.d.	n.d.	TAK
data/webviews/webfs/temporary/local__0	cacheDirectory	TAK	NIE	TAK	n.d.	TAK
data/webviews/webfs/persistent/local__0	dataDirectory	TAK	TAK	NIE	n.d.	TAK
file:///accounts/1000/removable/sdcard	externalRemovableDirectory	TAK	TAK	NIE	n.d.	NIE
file:///accounts/1000/shared	sharedDirectory	TAK	TAK	NIE	n.d.	NIE

Listing 4. Funkcje związane z zapisem danych do pliku

```
onDeviceReady : function() {
    app.receivedEvent(’deviceready’);
    app.prepareFile();
    $(’#saveButton’).click(function() {
        app.writeFile(”nacisnieto”);
    })
},

myFile : null,
prepareFile : function() {
    window.resolveLocalFileSystemURL(cordova.file.dataDirectory, function(
            dir) {
        dir.getFile(”plik.txt”, {
            create : true
        }, function(file) {
            app.myFile = file;
        });
    }, app.fail);
},

fail : function(e) {
    var msg = ’’;

    switch (e.code) {
    case FileError.QUOTA_EXCEEDED_ERR:
        msg = ’Brak miejsca’;
        break;
    case FileError.NOT_FOUND_ERR:
        msg = ’Nie znaleziono’;
        break;
    case FileError.SECURITY_ERR:
        msg = ’Brak dostępu’;
        break;
    case FileError.INVALID_MODIFICATION_ERR:
        msg = ’Niedostępna zmiana’;
        break;
    case FileError.INVALID_STATE_ERR:
        msg = ’Nieprawidłowy stan’;
        break;
    default:
        msg = ’Nieznany błąd’;
        break;
    }
    alert(‚Error: ‚ + msg);
},

writeFile : function(str) {
    if (!app.myFile)
        return;
    var log = str + ” [” + (new Date()) + ”]\n”;
    app.myFile.createWriter(function(fileWriter) {
        fileWriter.seek(fileWriter.length);
        var blob = new Blob([ log ], {
            type : ’text/plain’
        });
        fileWriter.write(blob);
    }, app.fail);
},
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9



Rysunek 1. Interfejs wykonywania fotografii z użyciem wtyczki org.apache.cordova.media-capture, w wypadku uruchomienia go na symulatorze Genymotion, na komputerze bez kamery

- **file()** – metoda, która w praktyce otwiera plik do odczytu, która zwraca obiekt typu **File** funkcji wywoływanej przy pomyślnej próbie wywołania tej metody,
- **fullPath** – pełna ścieżka do pliku,
- **isDirectory** – wartość **true** albo **false**, określająca czy obiekt **FileEntry** jest katalogiem,
- **isFile** – wartość **true** albo **false**, określająca czy obiekt **FileEntry** jest plikiem,
- **name** – nazwa pliku.

Do odczytu używamy funkcji **file()**, a jako jej pierwszy argument definiujemy funkcję w której tworzymy nowy obiekt klasy **FileReader**, korzystając z polecenia:

```
var reader = new FileReader();
```

Obiekt klasy **FileReader** ma metody i właściwości w dużej mierze analogiczne do tych z **FileWritera**:

- **error** – zawiera informację o błędzie, który wystąpił podczas próby odczytu pliku
- **readyState** – informacja o aktualnym stanie odczytu (**EMPTY**, **LOADING** lub **DONE**),
- **result** – treść wczytanego pliku, dostępna jedynie po pomyślnym zakończeniu operacji odczytu,
- **onabort** – wskazanie funkcji, wywoływanej w przypadku przerwania wczytywania pliku,
- **onerror** – wskazanie funkcji, wywoływanej w przypadku wystąpienia błędu podczas odczytu pliku,
- **onload** – wskazanie funkcji, wywoływanej po pomyślnym odczycie pliku,
- **onloadstart** – wskazanie funkcji, wywoływanej po rozpoczęciu wczytywania pliku,
- **onloadend** – wskazanie funkcji, wywoływanej po zakończeniu (pomyślnym lub nie) wczytywania pliku,
- **onprogress** – wskazanie funkcji, wywoływanej w przypadku wczytywania treści typu **Blob**,
- **abort()** – metoda do przerywania odczytu,
- **readAsArrayBuffer()** – metoda do wczytywania danych do formatu **ArrayBuffer**,
- **readAsBinaryString()** – metoda do wczytywania pliku jako danych binarnych,
- **readAsDataURL()** – metoda pozwalająca na wczytanie pliku do postaci danych w formacie „data:" (zakodowanych z użyciem Base64), użytecznych w niektórych przypadkach w HTML5,
- **readAsText()** – funkcja do wczytywania danych z pliku jako ciągu znaków.

Ponieważ dane w pliku zapisywaliśmy w postaci tekstowej, wczytamy je również jako tekst, funkcją **FileReader.readAsText()**. Musimy tylko wcześniej zdefiniować, co się stanie, po zakończeniu odczytu i w tym celu przypisujemy do atrybutu **FileReader.onloadend** funkcję wyświetlającą załadowany tekst w postaci komunikatu systemowego **alert()**. Korzystamy przy tym z atrybutu **FileReader.result**, w którym znajduje się treść wczytanego pliku. W efekcie powstała funkcja **app.readFile()**, zdefiniowana w następujący sposób:

```
readFile: function(){
    app.myFile.file(function(file){
        var reader = new FileReader();
        reader.onloadend = function(e){
            alert(this.result);
        };
        reader.readAsText(file);
    },app.fail);
},
```

Prosta obsługa kamery i mikrofonu

Na koniec tej części kursu spróbujemy pobrać obraz lub nagranie z kamery (i mikrofonu) telefonu, a następnie skorzystać z poznanej wiedzy i dostać się do materiału z poziomu systemu plików. W tym celu użyjemy prostego w obsłudze pluginu **org.apache.cordova.media-capture**. Po jego zainstalowaniu (w naszym przypadku wersja 0.3.6) w aplikacji otrzymujemy do dyspozycji obiekt **navigator.device.capture** oraz powiązane z nim metody i właściwości:

- **capture.captureAudio()** – funkcja rozpoczynająca nagrywanie dźwięku,
- **capture.captureImage()** – funkcja rozpoczynająca robienie zdjęć,
- **capture.captureVideo()** – funkcja rozpoczynająca nagrywanie wideo,
- **MediaFile.getFormatData()** – pobiera informacje na temat format zarejestrowanych multimediów,
- **supportedAudioModes** – lista formatów audio, obsługiwanych przez urządzenie,
- **supportedImageModes** – lista formatów obrazów, obsługiwanych przez urządzenie,
- **supportedVideoModes** – lista formatów wideo, obsługiwanych przez urządzenie.

Jak to zwykle bywa, jako parametry wymienionych wyżej funkcji przekazuje się nazwy funkcji do wywołania w razie powodzenia lub porażki próby pobrania danego rodzaju multimediów. Ponadto przekazuje się opcje, mówiące o liczbie nagrań lub zdjęć, które mają zostać wykonane,

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9

Listing 5. Funkcje związane z rejestracją obrazów i audio

```
onDeviceReady : function() {
    app.receivedEvent('deviceready');
    app.prepareFile();
         ('#saveButton').click(function(){
        app.writeFile("nacisnieto");
    });
    $('#readButton').click(function(){
        app.readFile();
    });
    $('#audioButton').click(function(){
        app.captureAudio();
    });
    $('#imageButton').click(function(){
        app.captureImage();
    });
    $('#videoButton').click(function(){
        app.captureVideo();
    });
},

captureSuccess : function(mediaFiles) {
    var i, path, len;
    for (i = 0, len = mediaFiles.length; i < len; i += 1) {
        path = mediaFiles[i].fullPath;
        if (path.substr(-4)==".jpg"){
            window.resolveLocalFileSystemURL(path, function(fileEntry){
                fileEntry.file(function(file){
                    var reader  = new FileReader();
                    reader.onloadend = function  () {
                        $("#myImage").prop('src',reader.result);
                    }
                    if (file) reader.readAsDataURL(file);
                    else $("#myImage").prop('src','');
                },app.fail);
            }, app.fail);
        }
    }
},

captureError : function(error) {
    alert(‚Error code: ‚ + error.code, null, 'Capture Error');
},
captureAudio : function(){
    navigator.device.capture.captureAudio(app.captureSuccess, app.captureError, {limit:1});
},
captureImage : function(){
    navigator.device.capture.captureImage(app.captureSuccess, app.captureError, {limit:1});
},
captureVideo : function(){
    navigator.device.capture.captureVideo(app.captureSuccess, app.captureError, {limit:1});
},
```

a w przypadku formatów audio i wideo, także o limicie długości tych nagrań. Do funkcji wywoływanych w przypadku powodzenia rejestracji obrazu lub dźwięku, przekazywane są obiekty klasy **MediaFile**, które odnoszą się do zarejestrowanych multimediów i pozwalają na operowanie nimi.

Niestety, prostota funkcji pluginu **org.apache.cordova.media-capture** sprawia, że nie da się za jego pomocą swobodnie, elastycznie rejestrować obrazy i dźwięki. Wtyczka wymusza skorzystanie z interfejsu graficznego, w którym użytkownik samodzielnie wybiera moment i długość rejestrowanego dźwięku czy obrazu i w razie czego może powtórzyć nagranie. Interfejs ten, uruchomiony na symulatorze Genymotion, został przedstawiony na **rysunku 1**. Co więcej, pozyskiwane materiały są automatycznie zapisywane w konkretnym katalogu – w naszym przypadku jest to ścieżka *file:/storage/emulated/0/*, w której albo bezpośrednio tworzone są pliki, albo podkatalogi z plikami.

Gotowy plik możemy np. wyświetlić w aplikacji, korzystając z funkcji **FileReader.readAsDataURL()**, dodając w pliku **index.html** obiekt **IMG** bez ścieżki

```
<IMG SRC="" id="myImage" height="200"/>
```

i dopisując następujący fragment kodu do funkcji obsługującej pomyślnie wykonane zdjęcie:

```
window.resolveLocalFileSystemURL(path,
function(fileEntry){
    fileEntry.file(function(file){
        var reader  = new FileReader();
        reader.onloadend = function  () {
          $("#myImage").prop('src',reader.
result);
        }
        if (file) reader.readAsDataURL(file);
        else $("#myImage").prop('src','');
    },app.fail);
}, app.fail);
```

gdzie **path** to zmienna zawierająca pełną ścieżkę pliku graficznego. W powyższy sposób, za pomocą funkcji jQuery **.prop()** podmieniamy aktualną wartość atrybutu SRC obiektu o id **myImage** (czyli naszego IMG) na zakodowane w formacie Base64 dane, bezpośrednio wyświetlane przez przeglądarkę jako plik graficzny JPEG.

Do czego może być przydatne takie rejestrowanie obrazów? W naszej aplikacji napęd bramy może za pomocą zainstalowanej wtyczki żądać zdjęć lub nagrań wideo osób, otwierających bramę i zapisywać je do celów archiwizacji.

Kod funkcji JavaScriptowych, związanych z rejestracją multimediów z użyciem wtyczki **org.apache.cordova.media-capture** pokazano się na **listingu 5**.

Podsumowanie

W niniejszej części kursu pokazaliśmy, jak skorzystać z niektórych podzespołów telefonu. Nie starczyło miejsca na pokazanie dalszych możliwości manipulacji na plikach, takich jak np. ich przenoszenie lub usuwanie, ale postaramy się to zrobić w przyszłych częściach kursu. Natomiast już w najbliższej części będziemy chcieli przybliżyć możliwe sposoby komunikacji urządzenia mobilnego z otoczeniem, w tym poprzez takie interfejsy, jak np. Bluetooth.

Marcin Karbowniczek, EP

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 75421, pass: tkuyg3b9