Programowanie aplikacji mobilnych (6)

Pozostałe funkcje telefonu

W dotychczasowych częściach kursu programowania aplikacji mobilnych pokazaliśmy już większość przydatnych, podstawowych funkcji telefonu, które można wykorzystać w programach tworzonych przez elektroników. Niemniej zostało jeszcze trochę wtyczek do Cordovy, które mogą okazać się użyteczne, a które trudno było podczepić pod opisywane dotąd grupy funkcji. W tej części kursu omawiamy pozostałe, ciekawe pluginy Cordovy i związane z nimi funkcje.

Tym razem rozpoczniemy programowanie od stworzenia zupełnie nowego projektu. Załóżmy, że budujemy urządzenie elektroniczne, które do swojej pracy wymaga smartfonu, służącego m.in. za interfejs użytkownika. Pomijamy wszystkie kwestie związane z połączeniem urządzenia z telefonem i to, co dane urządzenie w praktyce miałoby robić, a zajmiemy się diagnostyką smartfonu i sygnalizowaniem problemów. Sprawimy więc, że smartfon będzie monitorował stan naładowania swojej baterii i w razie jego obniżenia, informował o tym użytkownika. Co więcej, będzie go informować nie tyle stosownym komunikatem, wyświetlanym na ekranie, ale za pomocą wiadomości SMS, wysyłanej pod numer telefonu z książki adresowej systemu operacyjnego. Będzie też mógł odczytywać wiadomości SMS i przyjmować w ten sposób polecenia, a w razie potrzeby, będzie przesyłał informacje o aktualnym stanie dostępu do sieci. Wszystkie informacje będzie podawał z dbałością o zapewnienie odpowiedniego formatu informacji, jaki obowiązuje w danym regionie. W tym celu zostanie wykorzystany odpowiedni plugin, który to ułatwia.

Pokażemy też, jak lepiej zapanować nad interfejsem użytkownika, tj. wpłynąć na to, czy wyświetlany ma być pasek stanu i czy pokazywana ma być klawiatura ekranowa, co może mieć znaczenie przy mniejszych ekranach i w tych aplikacjach, gdzie jest ona momentami celowo niepotrzebna.

Pluginy

Aby zaimplementować podane powyżej funkcje skorzystamy z szeregu wtyczek, których lista znajduje się w **tabeli 1**. Możemy je wszystkie doinstalować od razu po stworzeniu nowego projektu (dla przypomnienia, polecenie: **crodova create**), korzystając z komendy **cordova**

Tabela 1. Lista pluginów, przydatnych w realizacji zadań z tej części kursu. W tabeli podano stare i nowe nazwy pluginów. Warto zwrócić uwagę, że pod nowym adresem niektóre wtyczki są już dostępne w nowszych wersjach (przy czym numeracja wersji dla pluginów przeniesionych do nowego repozytorium została w większości przypadków zresetowana). W trakcie pisania kursu korzystano z wersji ze starego repozytorium, w którym znajdowało się 1076 pluginów, podczas gdy w nowym było ich jak dotąd tylko 393. Nowe repozytorium znajduje się pod adresem: *http://plugins.cordova.io/npm/index.html*

Nazwa skrócona pluginu	Stara nazwa	Wersja ze starego adresu	Nowa nazwa	Wersja z nowego adresu
Device Orientation	org.apache.cordova.device-orientation	0.3.11	cordova-plugin-device-orientation	1.0.1
Battery	org.apache.cordova.battery-status	0.2.12	cordova-plugin-battery-status	1.1.0
Contacts	org.apache.cordova.contacts	0.2.16	cordova-plugin-contacts	1.1.0
Globalization	org.apache.cordova.globalization	0.3.4	cordova-plugin-globalization	1.0.1
Network Information	org.apache.cordova.network-information	0.2.15	cordova-plugin-network-information	1.0.1
SoftKeyboard	org.apache.cordova.plugin.softkeyboard	1.0.3	brak - trzeba użyć starego adresu	n.d.
StatusBar	org.apache.cordova.statusbar	0.1.10	cordova-plugin-statusbar	1.0.1
Keyboard	com.ionic.keyboard	1.0.4	ionic-plugin-keyboard	1.0.5
SMS	com.rjfun.cordova.sms	1.0.3	brak - trzeba użyć starego adresu	n.d.
sms	com.jsmobile.plugins.sms	0.0.1	brak - trzeba użyć starego adresu	n.d.
Cordova SMS Plugin	com.cordova.plugins.sms	0.1.2	cordova-plugin-sms	1.0.4

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 87550, pass: rxoagj8

plugin add. W tabeli podano wersje pluginów pobrane w trakcie tworzenia tego kursu, przy czym cały czas kurs prowadzony jest w oparciu o Cordovę w wersji 4.1.2, czyli korzystającą ze starej nomenklatury wtyczek (patrz. poprzednia część kursu). Dla osób, które chcą już korzystać z nowej wersji Cordovy, podajemy adresy nowych wtyczek. Warto zauważyć, że niektóre z nich są dostępne pod nowymi adresami w uaktualnionych wersjach, choć widać, że Cordova sama zmieniła nie tylko adresy ale i numerację wersji wielu swoich pluginów.

Zainstalowane pluginy wykorzystamy w następujący sposób – korzystając z wtyczki „**Battery**" będziemy monitorowali stan akumulatora. Jeśli spadnie on poniżej 20%, pobierzemy informację o orientacji przestrzennej urządzenia, korzystając z wtyczki „**Device Orientation**", a następnie sprawdzimy czy mamy dostęp do sieci (wtyczka „**Network Information**"). Jeśli tak, to zgromadzone informacje przetworzymy zgodnie z zasadami obowiązującymi w kraju posiadacza telefonu (wtyczka „**Globalization**"), wczytamy odpowiedni kontakt z książki adresowej (wtyczka „**Contacts**") i wyślemy wiadomość tekstową – z użyciem pluginu „**Cordova SMS plugin**". Uwaga: instalacja jednocześnie wtyczki **com.jsmobile.plugins.sms** i **com.cordova.plugins.sms** i ew. **com.rjfun.cordova.sms** uniemożliwia poprawne ich użycie, co wynika z wielokrotnej deklaracji tego samego obiektu.

Niezależnie zarejestrujemy reakcję na inne zdarzenie – na otrzymanie wiadomości tekstowej z poleceniem. W tym celu użyjemy najbardziej zaawansowanej z wtyczek do obsługi SMSów – „**SMS**" (**com.rjfun.cordova.sms**). Po otrzymaniu polecenia, dla celów demonstracyjnych, zareagujemy zmieniając widok klawiatury ekranowej (wtyczki „**Keyboard**" i „**SoftKeyboard**") oraz paska statusu („**StatusBar**"). Do dzieła.

Obsługa stanu baterii

Dodanie pluginów „**Battery**" sprawiło, że w systemie zostały zdefiniowane trzy nowe zdarzenia:

- **batterystatus**,
- **batterycritical**,
- **batterylow**.

Są one wyzwalane w momencie, gdy nastąpi zmiana stanu baterii, przy czym w przypadku zdarzenia **batterystatus**, wystarczy zmiana o 1% w dowolną stronę, podczas gdy w przypadku pozostałych zdarzeń, progi których przekroczenie powoduje wystąpienie zdarzenia są z góry ustalone przez system operacyjny.

Aby obsłużyć nowe zdarzenia, trzeba przypisać do nich akcję, korzystając z funkcji **window.addEventListener()**. Dla każdego z wymienionych zdarzeń, do funkcji je obsługującej przekazywany jest jeden obiekt z dwoma atrybutami:

- **level** – liczba z zakresu od 0 do 100, reprezentująca procentową wartość naładowania akumulatora (w przypadku niektórych systemów operacyjnych ta wartość może być niedostępna),
- **isPlugged** – wartość **true** lub **false**, informująca o tym czy urządzenie jest podłączone do zasilania.

Ponieważ w naszym przypadku chcemy wykrywać stan, w którym stopień naładowania akumulatora spadł poniżej 20%, skorzystamy z obsługi zdarzenia **batterystatus** i w tym celu tworzymy funkcję **app.onBatteryStatus()**.

Kompas

W poprzednich częściach kursu pokazaliśmy, jak korzystać ze wskazań akcelerometru i odbiornika GPS. Jednakże w niektórych urządzeniach mobilnych dostępny jest także kompas – niezależny, bazujący na polu magnetycznym Ziemi, sensor określający ustawienie telefonu w przestrzeni. O ile zgrubne obliczenie orientacji telefonu w przestrzeni jest możliwe, gdy urządzenie się porusza oraz dostępne są wskazania akcelerometru i odbiornika nawigacji satelitarnej, to w praktyce wygodniej jest skorzystać z wbudowanego kompasu.

Plugin **org.apache.cordova.device-orientation** wprowadza do systemu trzy nowe metody:

- **navigator.compass.getCurrentHeading()** – pozwala na pobranie aktualnego wskazania kompasu;
- **navigator.compass.watchHeading()** – pozwala na cykliczne pobieranie wskazań kompasu lub monitorowanie wskazania w celu wywołania wybranej funkcji o ile zmieni się ono wystarczająco znacząco;
- **navigator.compass.clearWatch()** – wstrzymuje cykliczne pobieranie lub monitorowanie wskazań kompasu.

W przypadku funkcji **navigator.compass.watchHeading()** twórcy zastosowali jedno mylące oznaczenie. Parametry tej funkcji są następujące:

- **compassSuccess** – nazwa funkcji wywoływanej w przypadku pomyślnego odczytania wskazań kompasu;
- **compassError** – nazwa funkcji wywoływanej w przypadku błędu podczas odczytu wskazań kompasu;
- **compassOptions** – opcjonalne parametry na które składa się obiekt o dwóch dopuszczalnych atrybutach:
  - **frequency** – podana w milisekundach wartość czasu (a nie częstotliwość) pomiędzy kolejnymi próbami odczytu wskazań kompasu;
  - **filter** – wartość minimalnej zmiany wskazania kompasu, która miałaby wywołać wywołanie funkcji **compassSuccess**.

Domyślna wartość parametru **compassOptions** zawiera atrybut **frequency** równy 100, a w przypadku podania wartości atrybutu **filter**, atrybut **frequency** jest zupełnie ignorowany.

Do funkcji wywoływanej po pomyślnym odczycie wskazań kompasu przekazywany jest obiekt **CompassHeading**, zawierający cztery atrybuty:

- **magneticHeading** – wartość odczytana bezpośrednio z kompasu, zawierająca się w zakresie od 0 do 359,99, przy czym 0 wskazuje północ;
- **trueHeading** – wartość odczytana z kompasu, skorygowana względem geograficznego położenia Bieguna Północnego. Również jest liczbą z zakresu od 0 do 359,99, ale może też przyjąć wartość ujemną, gdy ustalenie tego atrybutu nie jest możliwe;
- **headingAccuracy** – różnica pomiędzy wskazaniami **magneticHeading** i **trueHeading**;
- **timestamp** – wyrażony w milisekundach znacznik czasu, informujący o chwili pobrania danych z kompasu.

Będziemy korzystali z funkcji **navigator.compass.getCurrentHeading()**, wywoływanej wraz ze spadkiem poziomu baterii do ustalonej wartości.

Monitorowanie stanu sieci

Gdy już pobierzemy informacje o orientacji urządzenia w przestrzeni, sprawdzimy czy możemy ją przesłać, a dokładniej, czy mamy połączenie z siecią. Zainstalowany plugin **org.apache.cordova.network-information** udostępnia jeden nowy obiekt oraz dwa zdarzenia.

Obiekt **navigator.connection** zawiera atrybut **type**, który przyjmuje jedną z następujących wartości:

- **Connection.UNKNOWN**,
- **Connection.ETHERNET**,
- **Connection.WIFI**,
- **Connection.CELL_2G**,
- **Connection.CELL_3G**,
- **Connection.CELL_4G**,
- **Connection.CELL**,
- **Connection.NONE**.

Sprawdzenie aktualnego stanu sieci wymaga jedynie odczytu wartości atrybutu **navigator.connection.type**. Wartość **Connection.UNKNOWN** pojawia się, gdy stan sieci jest nieznany, a **Connection.CELL**, gdy wiadomo że urządzenie jest podłączone do sieci, ale nie wiadomo w jaki sposób. Przykładowo, wartość **Connection.CELL** jest zwracana np. w systemach operacyjnych takich jak iOS, które nie dostarczają szczegółowych informacji o sieci.

Dwa nowe zdarzenia, definiowane przez plugin **org.apache.cordova.network-information** to:

- **offline**,
- **online**.

Są one wywoływane, gdy urządzenie straci lub uzyska dostęp do sieci, czyli gdy wartość **navigator.connection.type** przyjmie wartość **Connection.NONE** lub gdy zmieni wartość z **Connection.NONE** na inną.

Trzeba przy tym zaznaczyć, że plugin ten koncentruje się na obsłudze sieci internetowej, a nie podłączeniu do sieci komórkowej, co oznacza, że jeśli urządzenie ma dostęp do Wi-Fi, a nie ma do sieci operatora, atrybut **navigator.connection.type** przyjmie wartość **Connection.WIFI**.

Adekwatny format danych

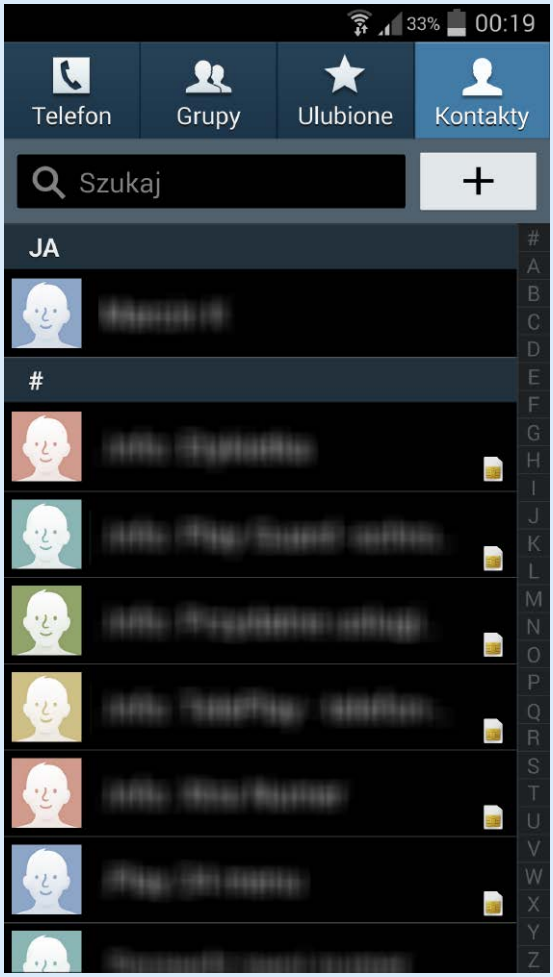
Być może część czytelników zastanawiała się kiedyś, dlaczego czasem w różnych publikacjach i urządzeniach, liczby podawane są z częściami dziesiętnymi oddzielonymi przecinkami, a czasem kropkami. Wynika to z różnych standardów przyjętych w poszczególnych krajach. O ile w Polsce do oddzielania części dziesiętnych liczb używamy przecinka, to w USA – kropki. To niejedyna różnica, pomiędzy zwyczajami zależnymi od regionu. Aby móc w łatwy sposób dostosować się do przyzwyczajeń użytkownika, obowiązujących w danym kraju, warto użyć wtyczki **Globalization**. Wprowadza ona szereg metod:

- **navigator.globalization.getPreferredLanguage()** – pobiera ustawienia języka, wprowadzone przez użytkownika urządzenia (np. en-US);
- **navigator.globalization.getLocaleName()** – pobiera ustawienia sposobu formatowania danych, tj. wskazanie kraju i języka, zgodnie z którego zasadami podawane będą liczby, daty, czas itp. Wartość ta może być inna niż uzyskana z metody **getPreferredLanguage()**, gdyż użytkownik może korzystać z telefonu np. w języku angielskim, a stosować polskie standardy, choć nie każde urządzenie na to pozwala;
- **navigator.globalization.dateToString()** – formatuje datę (podaną w postaci obiektu **Date**) do standardu zgodnego z ustawieniami;
- **navigator.globalization.stringToDate()** – tworzy obiekt typu **Date**, w oparciu o ciąg znaków, przy założeniu, że ciąg ten jest podany zgodnie z ustawieniami obowiązującymi w danym kraju;
- **navigator.globalization.getDatePattern()** – zwraca informacje o obowiązującym formacie daty, strefie czasowej, różnicy względem czasu Greenwich i o ewentualnym czasie letnim; Możliwe jest podanie rodzaju formatu czasu (krótki, długi, z datą itd.);
- **navigator.globalization.getDateNames()** – zwraca listę nazw dni tygodnia lub miesięcy, jakie stosowane są w danym kraju. Możliwe jest zażądanie nazw krótkich lub długich;
- **navigator.globalization.isDayLightSavingsTime()** – pozwala dowiedzieć się, czy wedle ustawień w danym urządzeniu, obowiązuje aktualnie czas letni;
- **navigator.globalization.getFirstDayOfWeek()** – informuje, który dzień tygodnia jest uważany w danym kraju za pierwszy, podając jego numer, przy czym jeśli zwrócona zostanie wartość 1, to jest to niedziela, a jeśli 2, to jest to poniedziałek itd.;
- **navigator.globalization.numberToString()** – tworzy ciąg znaków na podstawie liczby, zgodny z obowiązującym w danym kraju formatem;
- **navigator.globalization.stringToNumber()** – próbuje stworzyć liczbę, w oparciu o ciąg znaków, podany zgodnie z obowiązującym w danym kraju formatem;
- **navigator.globalization.getNumberPattern()** – pozwala pobrać obiekt z szeregiem atrybutów informujących o sposobie prezentowania liczb w danym kraju. Podawany jest nie tylko format zgodny ze standardem Unicode, ale też znak używany do formatowania ciągów znaków, liczba cyfr po przecinku, używana podczas wyświetlania liczb, sposób zaokrąglania, symbole stosowane do wskazywania liczb dodatnich, ujemnych, oddzielania części dziesiętnych oraz grupowania cyfr;
- **navigator.globalization.getCurrencyPattern()** – ostatnia z funkcji pozwala zdobyć informacje o walucie obowiązującej wedle ustawień danego urządzenia. Podawany jest nie tylko format prezentacji waluty, zgodnie ze standardem Unicode, ale też kod ISO4217 waluty, sposób zaokrąglania, grupowania cyfr itd.

W naszym przypadku użyjemy tylko funkcji **dateToString()** i **numberToString()**, by przygotować odpowiednią treść wiadomości do wysłania użytkownikowi.

Obsługa książki adresowej

Bardzo przydatną wtyczką jest **org.apache.cordova.contacts**, która pozwala na zapisywanie i wczytywanie kontaktów w książce adresowej urządzenia. Trzeba jednak zwrócić uwagę na fakt, że tworząc aplikację korzystającą z tego pluginów, wymagamy od użytkownika by zgodził się na udostępnienie nam informacji o wszystkich swoich kontaktach. Sposób wykorzystania tych



Fotografia 1. Zrzut ekranu umożliwiającego wybór kontaktu z użyciem funkcji navigator.contacts.pickContact()

danych zależy już od działania samej aplikacji, ale należy wziąć pod uwagę fakt, że w aplikacja mająca dostęp do tych cennych danych może być obiektem ataku osób, które chcą uzyskać informacje o kontaktach danego użytkownika.

Wtyczka pozwala na korzystanie z trzech nowych funkcji:

- **navigator.contacts.create()** – służy do tworzenia nowych kontaktów (które następnie można zapisać w książce adresowej);
- **navigator.contacts.find()** – pozwala pobierać informacje o kontaktach z książki adresowej, w oparciu o szereg filtrów;
- **navigator.contacts.pickContact()** – funkcja uruchamiająca systemowy ekran wyboru kontaktu z listy (**fotografia 1**), który następnie zostanie przekazany do aplikacji.

Do obsługi powyższych funkcji przygotowano szereg obiektów:

- **Contact** – podstawowy obiekt kontaktu, tworzony z użyciem funkcji **navigator.contacts.create()**, lub pobierany z książki adresowej. Zawiera liczne atrybuty, w tym będące obiektami poniższych typów;
- **ContactName** – obiekt zawierający informacje o imionach i tytułach kontaktu;
- **ContactField** – uniwersalne pole danych na temat kontaktu, w którym przechowywane mogą być takie informacje jak numer telefonu czy np. adres email. Atrybut **pref** tego obiektu pozwala sprawdzić, czy dany numer/adres jest domyślny (wartość **pref** wynosi wtedy **true**).
- **ContactAddress** – obiekt pozwalający na przechowywanie fizycznego adresu kontaktu; obejmuje atrybuty związane z typem adresu, nazwą ulicy, kodem pocztowym, państwem itp.;
- **ContactOrganization** – obiekt przechowujący informacje o organizacji, do której przynależy kontakt;
- **ContactFindOptions** – obiekt pomocniczy, podawany jako parametr funkcji **navigator.contacts.find()** i umożliwiający wyszukiwanie kontaktów w książce adresowej z użyciem filtra.
- **ContactError** – obiekt pomocniczy, zwracany w przypadku wystąpienia błędu związanego z obsługą książki adresowej. Zawiera tylko jeden atrybut: **code**, który może przyjąć którąś ze zdefiniowanych stałych. Pozwala rozpoznać rodzaj błędu, który wystąpił;

Warto jeszcze szczegółowo opisać obiekt **Contact**, który zawiera nie tylko atrybuty, ale i metody. Atrybuty obiektu **Contact** to:

- **id** – ciąg znaków, będący unikalnym identyfikatorem kontaktu,
- **displayName** – ciąg znaków, będący wyświetlaną użytkownikowi nazwą kontaktu,
- **name** – obiekt typu **ContactName**,
- **nickname** – ciąg znaków, zawierający pseudonim kontaktu,
- **phoneNumbers** – tablica obiektów typu **ContactField**, zawierająca numery telefonów kontaktu,
- **emails** – tablica obiektów typu **ContactField**, zawierająca adresy e-mail kontaktu,
- **addresses** – tablica obiektów typu **ContactAddress**, zawierająca adresy fizyczne kontaktu,
- **ims** – tablica obiektów typu **ContactField**, zawierająca numery komunikatorów internetowych, przypisanych do kontaktu,
- **organizations** – tablica obiektów typu **ContactOrganization**, zawierająca informacje o organizacjach, do których przynależy kontakt,
- **birthday** – obiekt typu **Date**, zawierający informacje o dacie urodzenia kontaktu,
- **note** – ciąg znaków, zawierających dodatkowe notatki na temat kontaktu,
- **photos** – tablica obiektów typu **ContactField**, zawierająca fotografie kontaktu,
- **categories** – tablica obiektów typu **ContactField**, zawierająca zdefiniowane przez użytkownika kategorie, do których przypisany jest dany kontakt,
- **urls** – tablica obiektów typu **ContactField**, zawierająca adresy stron internetowych, powiązanych z kontaktem.

Istnieją trzy metody obiektu **Contact**:

- **clone** – funkcja zwraca nowy obiekt typu **Contact**, zawierający pełną kopię obiektu typu **Contact**, podanego jako parametr, za wyjątkiem atrybutu id, który otrzymuje wartość **null**;
- **remove** – funkcja do usuwania kontaktu z książki adresowej;
- **save** – funkcja do dodawania nowego kontaktu do książki adresowej lub zapisywania zmian w dotychczas istniejącym kontakcie.

W naszym przypadku będziemy korzystać z funkcji **navigator.contacts.find()**, w celu pobrania danych kontaktu z książki adresowej, oraz z atrybutu **Contact.phoneNumbers**, by „zdobyć" numer kontaktu, do którego będziemy wysyłali wiadomość SMS.

Wysyłanie wiadomości tekstowych

Do wysyłania wiadomości SMS można użyć kilku różnych pluginów, które cieszą się podobną popularnością. My skorzystamy z domyślnego z nich, czyli **com.cordova.plugins.sms**. Wysłanie wiadomości tekstowej wymaga użycia funkcji **sms.send()**, która przyjmuje kolejno następujące pięć parametrów:

- **number** – numer telefonu, pod który ma być wysłana wiadomość,
- **message** – treść wiadomości do wysłania,
- **options** – obiekt zawierający opcje konfiguracyjne,
- **success** – nazwa funkcji wywoływanej w przypadku pomyślnego wysłania wiadomości,
- **error** – nazwa funkcji wywoływanej w przypadku wystąpienia błędu podczas wysyłania wiadomości.

Spośród powyższych dodatkowego wyjaśnienia wymaga parametr **options**. Ustawienie atrybutu **options.replaceLineBreaks** na **true** spowoduje zamianę znaków „\n" nowymi liniami w wysyłanej wiadomości. Ustawienie atrybutu **options.android.intent** na wartość „INTENT" spowoduje, że w przypadku Androida, wiadomość zostanie wysłana z użyciem domyślnej aplikacji systemowej do przesyłania wiadomości. Ustawienie pustego ciągu znaków jako wartości atrybutu **options.android.intent** spowoduje, że wiadomość zostanie wysłana bez otwierania żadnej dodatkowej aplikacji. Tak właśnie zrobimy w naszym przypadku.

Korzystając z wtyczki **com.cordova.plugins.sms** na Androidzie, należy się upewnić, że docelowa wersja API androidowego, wykorzystywanego w projekcie jest nie mniejsza niż 19. Parametr ten można ustawić w pliku **project.properties** w podkatalogu **platforms/android/** katalogu głównego projektu aplikacji.

Alternatywą do pluginów **com.cordova.plugins.sms** jest **com.jsmobile.plugins.sms**, w którym wysyłanie wiadomości odbywa się z użyciem funkcji **sms.sendMessage()**, która jako pierwszy parametr przyjmuje obiekt zawierający dwa atrybuty (**phoneNumber** z numerem telefonu adresata i **textMessage** z treścią wiadomości), a dwoma kolejnymi parametrami są funkcje wywoływane w przypadku pomyślnego wysłania wiadomości lub w przypadku wystąpienia błędu (odpowiednio).

Niezależnie od zastosowania jednej czy drugiej wtyczki, nie ma obecnie możliwości, by wiadomość w systemie iOS wysłać bez włączania domyślnej systemowej aplikacji do przesyłania SMSów. W przypadku Androida, SMS da się wysłać „w tle", przy czym w każdym przypadku będą one zapisane na liście wysłanych wiadomości, w domyślnym programie systemowym do obsługi SMSów.

Gotowy kod

Na **listingu 1** zaprezentowano kompletny kod JavaScript, realizujący opisany wcześniej algorytm. Treści kodu HTML nie trzeba zmieniać, gdyż cała aplikacja działa w praktyce bez jakiegokolwiek interfejsu użytkownika. Jedynie dla potwierdzenia wyświetlenia wiadomości, wyświetlane jest stosowne okno dialogowe.

Do obiektu **app** dodano pięć zmiennych, w których zapisywane są informacje o urządzeniu, w momencie ich pobierania. Szósta zmienna: **dataToSend** to obiekt zawierający informacje przetworzone do formatu, gotowego do wysyłki SMSem.

Obsługa zdarzenia związanego ze zmianą stanu baterii została dodana w funkcji **app.onDeviceReady()**, gdyż dopiero wtedy zdarzenie to jest rozpoznawane. Dodanie obsługi zdarzenia wcześniej spowodowałoby, że nie będzie ono wyzwalane.

Funkcja **app.onBatteryStatus()** pobiera informacje o aktualnym stanie baterii i sprawdza, czy poziom naładowania jest niższy niż 20%. Jeśli tak, to sprawdza, czy poziom ten już wcześniej został przekroczony, aby nie generować dodatkowych wiadomości wraz z każdym kolejnym spadkiem naładowania akumulatora o 1%. Jeśli poziom naładowania jest większy lub równy 20%, program zeruje zmienną informującą o uprzednim przekroczeniu stanu baterii. Aktualny stan jest zapisywany.

Następnie program stara się pobrać informacje z kompasu. Trzeba mieć jednak na uwadze, że wiele urządzeń nie ma wbudowanego takiego podzespołu – w naszym przypadku próba odczytu orientacji w przestrzeni kończyła się niepowodzeniem. Dlatego dla celów demonstracyjnych, wywołujemy polecenie **navigator.compass.getCurrentHeading()** z funkcją sukcesu bez dodatkowych parametrów oraz z funkcją niepowodzenia, która jest identyczna z funkcją sukcesu, tyle że z parametrami symulującymi dane pobrane z kompasu, wpisanymi na sztywno. Odpowiada za to linijka:

```
navigator.compass.getCurrentHeading(app.
getHeading,function(){app.getHeadi
ng({magneticHeading:310.15,timesta
mp:1437042176000});});
```

Funkcja **app.getHeading()** pobiera przekazane informacje o orientacji w przestrzeni i zapisuje je we wcześniej wspomnianych, dodatkowych zmiennych obiektu **app**. Następnie sprawdza stan sieci i jeśli urządzenie nie jest zupełnie odcięte od sieci komórkowej i Wi-Fi lub Ethernetu przewodowego, wywołuje polecenie **app.localize()**, którego celem jest przygotowanie danych do wysyłki.

W funkcji **app.localize()** użyto poleceń **navigator.globalization.numberToString()** i **navigator.globalization.dateToString()**, które są poleceniami asynchronicznymi, a więc nie zwracają przetworzonych wartości bezpośrednio, tylko poprzez wywołania kolejnych funkcji. W naszym przypadku oba polecenia wywołujemy z domyślnymi opcjami, a w funkcji sukcesu przypisujemy zmiennej **app.dataToSend** odpowiednie wartości atrybutów. Aktualną datę pobieramy ze znacznika czasu (milisekundowego), odczytanego w trakcie pobierania informacji z kompasu.

Następnie wywołujemy funkcję **app.sendToContact()**, która automatycznie wybiera z książki adresowej kontakt o nazwie wyświetlanej „administrator". Zaznaczamy przy tym, że interesuje nas tylko pierwszy kontakt z listy oraz przede wszystkim jego numer telefonu.

Do danych uzyskanego kontaktu odwołujemy się w funkcji **app.sendMessage()**, która pobiera stosowny

Listing 1. Kod Javascript, wysyłający komunikat SMSowy w momencie, gdy poziom zasilania spadnie poniżej 20%

```
var app = {
    battery:null,
    isPlugged:null,
    heading:null,
    headingDate:null,
    network:null,
    dataToSend:{battery:null,
        isPlugged:null,
        heading:null,
        network:null,
        date:null},
    initialize: function() {
    this.bindEvents();
},

bindEvents: function() {
    document.addEventListener(,deviceready', this.onDeviceReady, false);
},

onBatteryStatus: function(info) {
    console.log(„Poziom: „ + info.level + „ Zasilanie: „ + info.isPlugged);
    if (info.level<20){
        if (app.battery==null)
        {
            console.log(„pobieram dane baterii");
            app.battery=info.level;
            app.isPlugged=info.isPlugged;
            navigator.compass.getCurrentHeading(app.getHeading,function(){
                app.getHeading({magneticHeading:310.15,timestamp:1437042176000});
            });
        }
        }else app.battery=null;
    },
    getHeading: function(heading){
        app.heading=heading.magneticHeading;
        app.headingDate=heading.timestamp;
        app.network=navigator.connection.type;
        console.log(app.headingDate);
        console.log(app.network);
        if (navigator.connection.type!=Connection.NONE) app.localize();
    },
    localize: function(){
        navigator.globalization.numberToString(app.battery,function(number){
            app.dataToSend.battery=number.value;
        });
        app.dataToSend.isPlugged = (app.isPlugged) ? „podłączone" : „niepodłączone";
        navigator.globalization.numberToString(app.heading,function(number){
            app.dataToSend.heading=number.value;
        });
        console.log(app.headingDate);
        var fulldate = new Date(app.headingDate);
        navigator.globalization.dateToString(fulldate,function(date){
            app.dataToSend.date=date.value;
        });
        app.sendToContact();
    },
    sendToContact: function(){
        var fields=[navigator.contacts.fieldType.displayName];
        var options= new ContactFindOptions();
        options.filter="administrator";
        options.multiple=false;
        options.desiredFields=[navigator.contacts.fieldType.phoneNumbers];
        navigator.contacts.find(fields,this.sendMessage,this.onError,options);
    },
    sendMessage: function(contact){
        var number=contact[0].phoneNumbers[0].value;
        console.log(number);
        var message="Bateria poniżej 20%. Zasilanie „+ app.dataToSend.isPlugged
            + „. Kierunek: „ + app.dataToSend.heading
            + „. Odczyt z „ + app.dataToSend.date;
        console.log(message);
        var options={android:{intent:""}};
        sms.send(number,message,options,function(){
            alert(„wysłano wiadomość")
        },function(){
            alert(„błąd wysyłania wiadomości")
        });
    },
    onError: function(e){
        console.log(JSON.stringify(e));
    },
    onDeviceReady: function() {
        app.receivedEvent(,deviceready');
        window.addEventListener(,batterystatus', app.onBatteryStatus, false);
    },
    receivedEvent: function(id) {
        var parentElement = document.getElementById(id);
        var listeningElement = parentElement.querySelector(,.listening');
        var receivedElement = parentElement.querySelector(,.received');
        listeningElement.setAttribute(,style', ,display:none;');
        receivedElement.setAttribute(,style', ,display:block;');
        console.log(,Received Event: , + id);
    }
};

app.initialize();
```

numer telefonu i tworzy treść komunikatu, korzystając z przetworzonych wcześniej danych. Dodatkowo wskazujemy, że chcemy wysłać wiadomość w tle, po czym wyzwalamy polecenie wysłania SMSa.

W opisanym kodzie zastosowano kilka uproszczeń, których nie należy stosować w kodzie finalnym projektów na urządzenia mobilne. Po pierwsze nie wszędzie zadeklarowane funkcje wyzwalane w przypadku niepowodzenia działania poleceń. Po drugie, asynchroniczne polecenia przetwarzania liczb i dat tak, by były zgodne z lokalnymi ustawieniami, mogą się nie wykonać, zanim dojdzie do momentu wysyłania wiadomości. Polecenia te powinny znajdować się (zagnieżdżone) wewnątrz funkcji sukcesu kolejnych z nich, a komenda **app.sendToContact()** powinna być uruchamiana dopiero, gdy zakończone zostanie przetwarzanie ostatniej z danych. Mimo zastosowanego uproszczenia, program działa poprawnie, gdyż trochę czasu potrzebne jest na wczytanie kontaktu z książki adresowej, co sprawia że zanim dojdzie do formułowania treści komunikatu, potrzebne dane są już przygotowane.

W treści programu dodano też w kilku miejscach komendy logowania (polecenia **console.log()**) przebiegu działania aplikacji w konsoli systemowej JavaScriptu, co ułatwia debugowanie.

Odbieranie wiadomości tekstowych

Korzystając z innego, bardziej zaawansowanego pluginu do obsługi wiadomości SMS, można nie tylko wysyłać własne komunikaty, ale też odbierać je i przetwarzać listę wiadomości w telefonie. W tym celu wtyczka ta dostarcza osiem metod i definiuje jedno nowe zdarzenie:

- **sendSMS()** – funkcja umożliwiająca wysyłanie wiadomości pod jeden lub kilka numerów na raz.
- **listSMS()** – funkcja umożliwiająca przeglądanie listy wiadomości w oparciu o szereg filtrów. Filtr może obejmować rodzaj skrzynki pocztowej (odbiorcza, nadawcza, szkice, wszystkie itp.), może wskazywać tylko na wiadomości przeczytane lub nieprzeczytane, pochodzące z konkretnego numeru, o konkretnej treści lub o wybranym identyfikatorze; może też posłużyć do ładowania wiadomości stronami, poprzez podanie pozycji, od której mają być wyszukiwane i liczby wiadomości na stronę;
- **deleteSMS()** – funkcja pozwala na usuwanie wiadomości ze skrzynki, również z użyciem filtra. Dostępne opcje filtrowania są identyczne, jak w przypadku funkcji **listSMS()**, za wyjątkiem możliwości podziału na strony;
- **startWatch()** – funkcja rozpoczynająca monitorowanie skrzynki z wiadomościami (oraz przy okazji status połączenia Bluetooth). Jeśli zostanie uruchomiona, w razie nadejścia nowej wiadomości zostanie wywołane zdarzenie **onSMSArrive**;
- **stopWatch()** – funkcja wstrzymująca monitorowanie skrzynki z wiadomościami (a także status połączenia Bluetooth);
- **enableIntercept()** – funkcja ta pozwala uniemożliwić innym aplikacjom otrzymywanie nadchodzących wiadomości SMS;
- **restoreSMS()** – funkcja pozwala na przekazanie przechwyconej (po włączeniu **enableIntercept()**) wiadomości do systemowej skrzynki pocztowej;
- **setOptions()** – polecenie umożliwiające ustawienie dodatkowych opcji działania pluginu;
- **onSMSArrive** – zdarzenie wyzwalane w momencie nadejścia wiadomości, o ile wcześniej uruchomiono polecenie **startWatch()**;

Pasek stanu urządzenia

Zaskakująco rozbudowany okazuje się być plugin umożliwiający zmienianie stanu paska systemowego w urządzeniach mobilnych. Udostępnia on programiście aż dziewięć metod i jeden atrybut. Ponadto można go dodatkowo skonfigurować w ustawieniach projektu, co pozwala sterować wyglądem paska na etapie uruchamiania aplikacji. My jednak skoncentrujemy się na kontroli paska dopiero w trakcie pracy programu. Dostępne są następujące funkcje:

- **overlaysWebView()** – metoda ta pozwala w systemie iOS (i tylko tam) zdecydować, czy pasek ma być widoczny czy nie w trybie WebView;
- **styleDefault()**, **styleLightContent()**, **styleBlackTranslucent()**, **styleBlackOpaque()**, **backgroundColorByName()**, **backgroundColorByHexString()** – metody dla systemów iOS i Windows Phone, pozwalające na ustawienie domyślnego, jasnego, ciemnego-półprzeźroczystego, ciemnego-nieprzeźroczystego lub konkretnego koloru i stylu paska.
- **hide()** – funkcja ukrywająca pasek;
- **show()** – funkcja pokazująca pasek.

Atrybut **StatusBar.isVisible** pozwala sprawdzić, czy pasek jest aktualnie widoczny.

Kontrola klawiatury ekranowej

Czasem zachodzi sytuacja, gdy chcemy samodzielnie ukryć lub wyświetlić klawiaturę ekranową urządzenia mobilnego. Wtedy przydatna będzie jedna z wtyczek pozwalających na kontrolę prezentacji klawiatury. Podstawowy plugin to **org.apache.cordova.plugin.softkeyboard**, który zawiera tylko jedną funkcję **cordova.plugins.SoftKeyboard.show()** i która działa jedynie w systemie Android. Dlatego znacznie większą popularnością cieszy się wtyczka zespołu Ionic: **com.ionic.keyboard**, która zawiera cztery metody i działa w systemach Android, iOS i BlackBerry 10. Dostępne metody to:

Listing 2. Fragment kodu programu do przechwytywania nadchodzących wiadomości tekstowych

```
waitForMessage:function(){
    if (SMS) SMS.startWatch(function(){
        window.addEventListener(,onSMSArrive',app.readMessage);
    },null);
},
readMessage: function(e){
    var message = e.data;
    var info=”Otrzymano wiadomość z adresu „ + message.address + „. Jej treść to: „ + message.body;
    StatusBar.hide();
    cordova.plugins.Keyboard.show();
}
```

- **hideKeyboardAccesoryBar()** – metoda ta działa tylko w systemie iOS i pozwala ukryć pasek klawiatury z przyciskami „następny” „poprzedni” i „ok”;
- **close()** – metoda ukrywająca klawiaturę w każdym z obsługiwanych systemów operacyjnych;
- **disableScroll()** – metoda tylko dla systemu iOS, wyłączająca automatyczne przewijanie ekranu, w trakcie uzupełniania pól tekstowych;
- **show()** – metoda dla systemów Android i BlackBerry 10, umożliwiająca pokazanie klawiatury.

Powyższe polecenia wywołuje się stosując długi przedrostek, np. następująco:

```
cordova.plugins.Keyboard.
hideKeyboardAccessoryBar(true);
```

Wtyczka udostępnia też atrybut **cordova.plugins.Keyboard.isVisible**, informujący o tym, czy klawiatura jest aktualnie widoczna oraz definiuje dwa zdarzenia:

- **native.keyboardshow** – zdarzenie wyzwalane w momencie pokazania klawiatury, wraz z którym przekazywana jest informacja o podanej w pikselach wysokości klawiatury;
- **native.keyboardhide** – zdarzenie wyzwalane w momencie ukrycia klawiatury.

Przykładowy kod

Na **listingu 2** umieszczono fragment kodu programu, który pozwala na odbieranie wiadomości tekstowych oraz na kontrolę wyświetlania klawiatury ekranowej i paska systemowego. W domyślnym projekcie Cordovy wystarczy dodać linijkę:

```
app.waitForMessage();
```

wewnątrz funkcji **app.onDeviceReady()**. W przypadku dodania tego kodu do kodu z **listingu 1** trzeba pamiętać, że wtyczki **com.rjfun.cordova.sms**, **com.jsmobile.plugin.sms** i **com.cordova.plugins.sms** nie są ze sobą kompatybilne i w związku z tym należy najpierw odinstalować pozostałe, by użyć wtyczki **com.rjfun.cordova.sms**, wykorzystanej w przykładzie z listingu 2.

Uwaga – na niektórych urządzeniach mobilnych funkcja przechwytywania SMSów może działać wadliwie i powodować zawieszanie aplikacji.

Podsumowanie

Pokazaliśmy, jak korzystać z niektórych dodatkowych funkcji sprzętowych urządzeń mobilnych oraz jak stosować komunikację SMSową. Szczególnie interesująca wydaje się ta ostatnia funkcja, gdyż w połączeniu z wcześniej zdobytą już wiedzą, umożliwia tworzenie np. bluetoothowych bramek SMSowych, czy zdalne sterowanie urządzeniami sieciowymi z użyciem wiadomości tekstowych. O ile na danym telefonie wtyczka **com.rjfun.cordova.sms** działa poprawnie, można w niedrogi sposób zrealizować system telemetryczny do nadzoru odległych instalacji.

W kolejnej części kursu skoncentrujemy się na funkcjach związanych z multimediami, obsługą kamery, kodów graficznych, syntezą mowy i przesyłaniem plików, a jeśli wystarczy miejsca, rozpoczniemy temat debugowania aplikacji mobilnych.

Marcin Karbowniczek, EP

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 87550, pass: rxoaagj8