Programowanie aplikacji mobilnych (8)

Debugowanie HTML + CSS + JavaScript

Debugowanie programów napisanych z użyciem Cordovy to zagadnienie bardzo złożone, ponieważ mamy do czynienia z kodem napisanym w wielu językach programowania, który ma pracować na różnych systemach operacyjnych, a te mają być uruchamiane na różnym sprzęcie. Co więcej, poszczególne systemy występują w różnych wersjach, które różnią się nie tylko zgodnością z bibliotekami Cordovy, ale też wsparciem dla funkcji języka JavaScript i styli CSS, opisujących wygląd aplikacji. Na szczęście da się to nieco podzielić na mniejsze zadania. Warto zacząć debugowanie od dopracowania kodu JS+HTML+CSS, a następnie – jeśli jest taka potrzeba – zająć się debugowaniem całej aplikacji, już pod kątem poprawności kodu Javy, Objective C, C#, czy innego języka, zależnego od wybranej platformy.

Wspominaliśmy już w pierwszej części kursu, że kompilacja aplikacji dla poszczególnych wersji Androida wymaga instalacji odpowiednich wersji bibliotek z użyciem Android Managera. Nazwom kodowym, takim jak np. Jelly Bean, czy Lollipop, odpowiadają grupy lub zakresy wersji (np. 4.1, 4.2 i 4.3), a poszczególnym wersjom lub ich zakresom – poziomy API. Przykładowo, Android Jelly Bean w wersji 4.2 wspiera API na poziomie 17. I tyle, by nas interesowało, gdybyśmy programowali Androida bezpośrednio w Javie. Jednakże my obraliśmy łatwiejsze i bardziej uniwersalne podejście, którego negatywne konsekwencje dotkną nas właśnie teraz. Napisany przez nas kod JavaScripcie, z użyciem języków HTML5 i CSS3 nie jest uruchamiany bezpośrednio w procesorze smartfonu, ani nawet w wirtualnej maszynie Javy, która jest środowiskiem wykonywalnym wszystkich aplikacji androidowych. Nasz kod korzysta z bibliotek Cordovy, które pracują na wirtualnej maszynie Javy, ale aby mógł się do nich odwołać, musi najpierw zostać przetworzony przez interpreter JavaScriptu, CSSa i HTMLa. Interpreter ten zawarty jest we wbudowanych w mobilnych systemach operacyjnych przeglądarkach internetowych. W Androidzie domyślną przeglądarką jest Chrome (a właściwie Android WebView, bazujący na silniku z tej samej rodziny), w iOSie jest to Safari, a w Windows Phone – Internet Explorer. Poszczególne przeglądarki różnią się tzw. silnikiem, czyli mechanizmem wyświetlania stron, ale ponieważ tego typu silników jest mniej niż przeglądarek, niektóre z systemów mobilnych korzystają z niemal tych samych silników. W przypadku Androida, a więc przeglądarki Chrome czy też WebView, silnikiem jest Webkit.

Debugowanie na komputerze

Jeśli więc chcemy debugować aplikację napisaną z myślą o Androidzie i zamierzamy zająć się najpierw kodem JavaScript+HTML+CSS, najlepiej posłużyć się tzw. „desktopową" przeglądarką internetową z tym samym silnikiem, co przeglądarka w naszym mobilnym systemie operacyjnym. Używanie dosyć zaawansowanych mechanizmów wnikania w kod, w jakie wyposażone są nowoczesne przeglądarki internetowe pozwala na dosyć wygodne analizowanie błędów.

Instalujemy więc przeglądarkę Google Chrome i możemy przystępować do debugowania. Ale jeśli chcemy być precyzyjni, warto sięgnąć po przeglądarkę z identyczną wersją Webkita, jak na naszym systemie mobilnym. W chwili pisania tego artykułu najnowsza, aktualna wersja Chrome ma już numer 46 i nawet, jeśli mamy również najnowszą wersję tej przeglądarki, zainstalowaną na telefonie komórkowym, to zazwyczaj niestety nie będzie to wersja, w oparciu o którą będzie pracować nasza aplikacja.

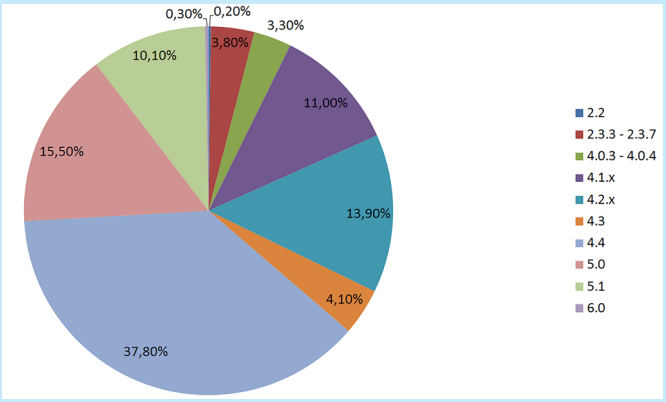
Wersje Androida i WebView

Program napisany z użyciem Cordovy nie uruchamia się bowiem w zewnętrznej przeglądarce w telefonie, ale w tzw. trybie WebView, a ten opiera się o silnik renderowania stron, którego wersja zależna jest od wersji całego systemu operacyjnego, a nie od wersji przeglądarki zainstalowanej na telefonie. Przykładowo, jeśli chcemy uruchomić aplikację na Androidzie Jelly Bean w wersji 4.1.1, to tryb WebView będzie korzystał z silnika Webkit w wersji 534.30. Natomiast przeglądarka Google Chrome, która korzysta z identycznej wersji Webkita nosi numer 12 i została wydana… w czerwcu 2011 roku! To niestety duże ograniczenie swobody programowania, gdyż od tego czasu powstały już 34 nowsze główne wersje Chrome, które nie tylko pozbawione są starych błędów i wprowadzają usprawnienia, ale też pozwalają na korzystanie z dodatkowych funkcji i parametrów języka CSS. Tak naprawdę dopiero Android KitKat, czyli 4.4, wprowadził istotną aktualizację Webkita w WebView, która została podniesiona do 537.36. W biurkowym Chrome zastosowano ją pierwszy raz w wersji 27 (połowa 2013 roku), a wszystkie dotychczasowe nowsze wersje Chrome pracują z delikatnie zmodyfikowanym silnikiem Webkit 537.36 Blink. Lista wersji Androida, wraz z podanymi wersjami silnika Webkit, stosowanego w trybie WebView, oraz odpowiadające im wersje PeCetowej przeglądarki Chrome została pokazana w **tabeli 1**.

Popularność Androida

Z powyższych rozważań wynika, że jeśli chcemy dopracować wygląd naszej aplikacji lub po prostu skorzystać z najnowszych bibliotek JavaScript czy zestawów styli CSS, musimy ograniczyć się do wspierania jedynie Androida w wersji 4.4 i nowszej. Czy to dobra decyzja? To zależy od przeznaczenia aplikacji i tego, kto ma z niej korzystać. Warto przy podejmowaniu decyzji zwrócić uwagę na fakt, że wersje 3.x były przeznaczona tylko na tablety oraz posłużyć się statystykami. W listopadzie 2015 roku, w chwili pisania tego artykułu, z Androida 4.4 KitKat korzystało 37,8% urządzeń mobilnych z systemem operacyjnym firmy Google. Z nowszego Androida 5.0 Lollipop lub 6.0 Marshmallow łącznie 25,63% (**rysunek 1**). Dosyć duży udział wciąż miał Android Jelly Bean – łącznie ponad 29%, który choć zawiera wiele usprawnień w stosunku do Androida Ice Cream Sandwich, pod względem silnika używanego w WebView niczym się nie różni. Dane te pochodzą ze sklepu Google i dotyczą całego świata. Aktualne informacje na temat popularności poszczególnych wersji Androida można znaleźć pod adresem: *https://goo.gl/pC78pv*.

Na pocieszenie można przytoczyć ogólne statystyki popularności systemów operacyjnych. Okazuje się bowiem, że o ile uda nam się dopracować aplikację pod względem



Rysunek 1. Popularność poszczególnych wersji systemów Android. Stan na początek listopada 2015 r.

Tabela 1. Wersje systemów Android, stosowanych w nich domyślnie silników Webkit w WebView i przeglądarek Chrome z najbardziej zbliżonym silnikiem Webkit

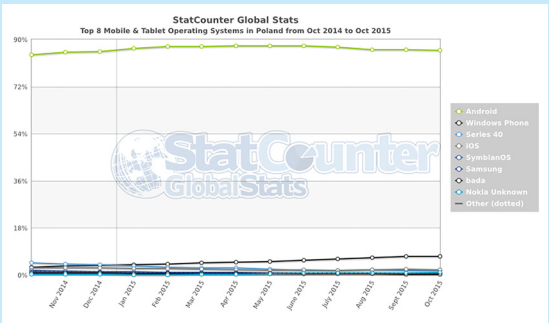
Wersja Androida		Wersja silnika Webkit	Wersja przeglądarki Chrome z najbardziej zbliżonym silnikiem	Wersja silnika Webkit w PeCetowym Chrome
Eclair	Android 2.1-update1	530.17	2.0.172	530
Froyo	Android 2.2	533.1	5.0.375	533
Froyo	Android 2.2.1	533.1	5.0.375	533
Froyo	Android 2.2.2	533.1	5.0.375	533
Froyo	Android 2.2.3	533.1	5.0.375	533
Gingerbread	Android 2.3.2	533.1	5.0.375	533
Gingerbread	Android 2.3.3	533.1	5.0.375	533
Gingerbread	Android 2.3.4	533.1	5.0.375	533
Gingerbread	Android 2.3.5	533.1	5.0.375	533
Gingerbread	Android 2.3.6	533.1	5.0.375	533
Gingerbread	Android 2.3.7	533.1	5.0.375	533
Honeycomb	Android 3.2.1	534.13	8.0.552	534.13
Ice Cream Sandwich	Android 4.0.1	534.30	12.0.742	534.30
Ice Cream Sandwich	Android 4.0.2	534.30	12.0.742	534.30
Ice Cream Sandwich	Android 4.0.3	534.30	12.0.742	534.30
Ice Cream Sandwich	Android 4.0.4	534.30	12.0.742	534.30
Jelly Bean	Android 4.1.1	534.30	12.0.742	534.30
Jelly Bean	Android 4.1.2	534.30	12.0.742	534.30
Jelly Bean	Android 4.2	534.30	12.0.742	534.30
Jelly Bean	Android 4.2.1	534.30	12.0.742	534.30
Jelly Bean	Android 4.2.2	534.30	12.0.742	534.30
Jelly Bean	Android 4.3	534.30	12.0.742	534.30
KitKat	Android 4.4.x	537.36	27.0.1453	537.36
Lollipop	Android 5.0.x	537.36	27.0.1453	537.36
Lollipop	Android 5.1	537.36	b.d.	
Lollipop	Android 5.1.1	537.36	b.d.	
Marshmallow	Android 6.0	537.36	b.d.	

zgodności z poszczególnymi wersjami Androida (czyli np. do zgodności z Webkitem 534.30, dzięki któremu aplikacja będzie się poprawnie wyświetlała na 96% telefonów z Androidem), to nasz program będzie zarazem poprawnie działał w przybliżeniu na 82% wszystkich urządzeń mobilnych w Polsce. Co prawda trudno znaleźć dokładne informacje na temat popularności poszczególnych wersji Androida w różnych krajach, ale dostępne są statystyki odnośnie wszelkich systemów mobilnych stosowanych w Polsce. W październiku 2015 roku niemal 86% wszystkich urządzeń mobilnych, jakie korzystają w naszym kraju z Internetu to urządzenia pracujące pod kontrolą Androida (**rysunek 2**). Udział w rynku Androida na Świecie wynosi natomiast około 63% (i rośnie), a na drugim miejscu jest iOS, który pracuje w około 23% urządzeń mobilnych korzystających z Internetu (**rysunek 3**). Aktualne dane na ten temat można znaleźć pod adresem *http://gs.statcounter.com*.

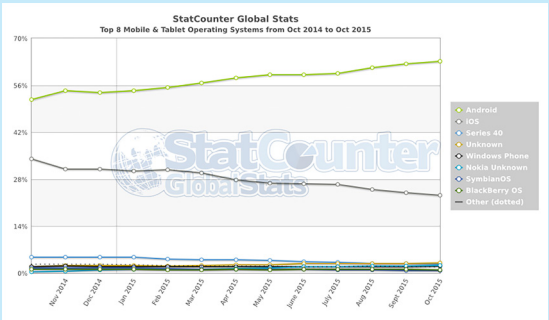
Starsze wersje Google Chrome

Niezależnie, na której wersji Webkita się skoncentrujemy, pewnym problemem może okazać się pobranie starej przeglądarki Chrome. Google już nie udostępnia ich na swoich stronach, a co więcej automatycznie aktualizuje każdą przeglądarkę, gdy tylko będzie to możliwe. Co gorsza, większość serwisów oferujących darmowe oprogramowanie, w przypadku Chrome linkuje bezpośrednio do Google, więc poszukiwanie starszych wersji Chrome może nastręczać problemów. W chwili pisania tego artykułu ratunkiem był chyba tylko serwis Oldversion.com. Pomimo, że próba wyszukiwania Chrome na tej stronie nie przynosi żadnych rezultatów, znajomość bezpośredniego linku do podstrony z tą przeglądarką już tak: *http://goo.gl/kPSCaH*. Teoretycznie dostępne tam pliki są puste, da się je faktycznie pobrać. Warto sobie zrobić kopię kluczowych dla WebView wersji Chrome: 2.0.172, 5.0.375, 8.0.552, 12.0.742 i 27.0.1453.

Pozostaje jeszcze problem instalacji różnych wersji Chrome na komputerze, co może być szczególnie niewygodne dla osób, które na co dzień korzystają z tej przeglądarki. Oczywiście można na zmianę instalować jedną, a później drugą wersję, po zakończeniu debugowania, ale to mało ergonomiczne. Pogłoski mówią, że Chrome nie powoduje



Rysunek 2. Popularność poszczególnych systemów operacyjnych na urządzeniach mobilnych w Polsce w okresie od października 2014 do października 2015 r.



Rysunek 3. Popularność poszczególnych systemów operacyjnych na urządzeniach mobilnych na Świecie w okresie od października 2014 do października 2015 r.

problemów, jeśli zainstaluje jego inną wersję na innym profilu użytkownika. Alternatywnym rozwiązaniem może być skorzystanie z drugiego systemu operacyjnego (choćby darmowego Linuksa) na potrzeby debugowania, albo np. instalacja systemu na wirtualnym komputerze, stworzonym w bezpłatnym VirtualBoxie, który i tak był potrzebny do instalacji emulatora Genymotion.

Co wybrać do debugowania?

Oczywiście można też po prostu korzystać z najnowszego Chrome, który nie tylko działa nieco szybciej niż starsze wersje, ale też jest wygodniejszy w użyciu, szczególnie jeśli chodzi o debugowanie. Zgodność ze starszymi wersjami Webkitu można opierać na zachowaniu zgodności z określoną wersją Chrome, poprzez ostrożne dobieranie używanych własności i jednostek w stylach CSS. Świetną pomocą będą zasoby serwisu *w3schools.com*. W podanej tam dokumentacji do języka CSS wskazane są dokładne wersje przeglądarek, od których obsługa poszczególnych deklaracji i parametrów została zaimplementowana. Jako przykład niech posłuży **tabelka 3**, w której zebrano informacje odnośnie obsługi różnych jednostek miar stosowanych w CSS, w różnych przeglądarkach.

Praktyka pokazuje jednak, że – szczególnie, jeśli korzysta się z zewnętrznych, nowoczesnych bibliotek – trudno uniknąć drobnych „wpadek". Najwygodniejszym rozwiązaniem wydaje się – podczas programowania aplikacji na Androida – korzystanie z nowej wersji Chrome w trakcie większości prac związanych z jej debugowaniem, a na koniec sprawdzenie jej przede wszystkim w wersji, która pracuje z silnikiem Webkit trybu WebView najstarszej wersji Androida, z którą zgodna ma być aplikacja oraz przetestowanie aplikacji na różnych symulatorach. Trzeba się też liczyć z tym, że gotowy program może po prostu wyglądać nieco inaczej na starszych telefonach, gdyż szkoda nie korzystać z różnych atrakcyjnych funkcji, jakie pojawiły się na przestrzeni lat w CSSie.

Tabela 2. Wersje systemów Android, jakie używano na świecie na początku listopada 2015 roku. Wersje, które obejmowały 0,1% lub mniej użytkowników zostały pominięte

Wersja	Nazwa	Poziom API	Odsetek użytkowników
2.2	Froyo	8	0,2%
2.3.3 – 2.3.7	Gingerbread	10	3,8%
4.0.3 – 4.0.4	Ice Cream Sandwich	15	3,3%
4.1.x	Jelly Bean	16	11,0%
4.2.x		17	13,9%
4.3		18	4,1%
4.4	KitKat	19	37,8%
5.0	Lollipop	21	15,5%
5.1		22	10,1%
6.0	Marshmallow	23	0,3%

Tabela 3. Przykład obsługi poszczególnych sposobów podawania wymiarów w plikach CSS dla różnych przeglądarek. Podane numery określają wersje, w których obsługa danego sposobu została wprowadzona

Jednostka długości \ Przeglądarka	Chrome	Internet Explorer	Firefox	Safari	Opera
em, ex, %, px, cm, mm, in, pt, pc	1.0	3.0	1.0	1.0	3.5
ch	27.0	9.0	1.0	7.0	20.0
rem	4.0	9.0	3.6	4.1	11.6
vh, vw	20.0	9.0	19.0	6.0	20.0
vmin	20.0	9.0 (tylko pod nazwą vm)	19.0	6.0	20.0
vmax	26.0	Nie obsługuje	19.0	Nie obsługuje	20.0

Rozpoczęcie debugowania

Ponieważ koncentrujemy kurs na programowaniu pod kątem Androida, pokażemy jak debugować kod JavaScript, HTML i CSS aplikacji właśnie na potrzeby tej platformy. W praktyce można wskazać dwa sposoby debugowania tego kodu: rzeczywisty i powierzchowny (nie są to formalne określenia). Ta pierwsza metoda pozwala faktycznie sprawdzić, czy kod poprawnie działa na rzeczywistym urządzeniu. Druga natomiast jest szybsza i wygodniejsza do wprowadzania kosmetycznych poprawek, ale nie pozwala testować wielu zaawansowanych funkcji, a szczególnie tych, wymagających pracy konkretnych podzespołów elektronicznych urządzenia mobilnego.

Aby sensowne, rzeczywiste debugowanie JavaScriptu, HTMLa i CSSa w aplikacjach Cordovy było w ogóle możliwe, konieczne jest spełnienie kilku wymagań:

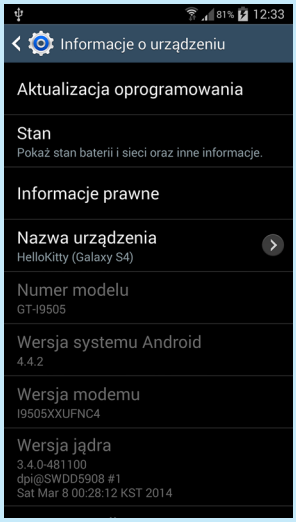
- Uruchomiona aplikacja powinna mieć możliwość debugowania.
- System, na którym jest uruchamiana aplikacja powinien pozwalać na jej debugowanie.
- Komputer, który łączy się z systemem, na którym jest uruchomiona debugowana aplikacja, powinien umożliwiać pracę w trybie debugowania.
- Oprogramowanie używane do debugowania musi obsługiwać używany system operacyjny.

Pierwszy z punktów mamy załatwiony – domyślnie kompilowane i budowane aplikacje, nawet w najnowszych wersjach Cordovy, mają włączoną funkcję debugowania. Natomiast aplikacji tego typu nie można wrzucać do sklepu Google Play. Publikowane aplikacje muszą być podpisane unikalnym podpisem wytwórcy oprogramowania (o czym napiszemy w jednej z dalszych części kursu), a aplikacje budowane z włączonym trybem debugowania, podpisywane są kluczem uniwersalnym.

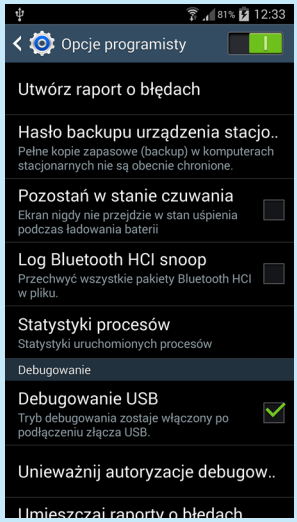
Aplikację, którą chcemy debugować instalujemy na urządzeniu mobilnym lub symulatorze takiego urządzenia, w taki sam sposób jak dotychczas. Jednakże system operacyjny urządzenia lub symulatora może domyślnie nie pozwalać na debugowanie, mimo że pozwala już na instalację aplikacji z nieznanych źródeł. Aby włączyć tę opcję należy upewnić się, że urządzenie pokazuje „Opcje programisty" (**rysunek 4**). Jeśli „Opcje programisty" nie są w ogóle pokazane, należy wejść w „Informacje o urządzeniu" i wielokrotnie kliknąć na pole „Wersja systemu Android" (**rysunek 5**). Następnie należy wrócić do poprzedniego menu i wejść w „Opcje programisty", po czym włączyć „Debugowanie USB" (**rysunek 6**). Pozostałe opcje programisty omówimy w kolejnej części kursu.

Teoretycznie wystarczy już teraz tylko podłączyć telefon do komputera lub włączyć symulator (np. Genymotion), a następnie przeglądarkę Google Chrome i rozpocząć debugowanie. Jednakże nierzadko problemem okazuje się niemożność wykrycia telefonu lub symulatora przez przeglądarkę. W Google Chrome, w pasku adresu, wpisujemy *chrome://inspect/#devices*. Jeśli telefon podłączony przez USB do komputera ma uruchomioną aplikację, która pozwala na debugowanie i sam telefon również na debugowanie przez USB pozwala, oczom użytkownika powinien pojawić się ekran mniej więcej taki, jak na **rysunku 7**. Wyświetlane są informacje o nazwie podłączonego urządzenia, a następnie nazwy uruchomionych aplikacji, które można debugować. Jeśli żadna taka aplikacja nie jest uruchomiona, urządzenie i tak powinno pojawić się na liście. Natomiast w informacjach o aplikacji znaleźć można jej nazwę – zarówno tę widzianą przez użytkownika, jak i zapisaną w odwrotnym formacie domenowym oraz rozdzielczość z jaką pracuje i pozycję ekranu, w której się zaczyna widok WebView. W naszym przypadku pozycja to 0,75, co oznacza

Rysunek 4. Opcje programisty dostępne na liście ustawień



Rysunek 5. Informacje o urządzeniu



Rysunek 6. Opcje programisty umożliwiają włączenie debugowania przez USB

że pierwsze 75 wierszy pikseli jest zajęte przez pasek stanu telefonu. Podany jest też plik HTML, w którym znajduje się podstawowy kod, jaki stworzyliśmy. Warto zwrócić uwagę na wersję WebView użytego do uruchomienia aplikacji. Przy każdej wylistowanej w ten sposób aplikacji znajduje się niebieski przycisk „inspect", którym przechodzi się do właściwego debugowania kodu HTML, JavaScript i CSS.

Tabela 4. Lista adresów, pod którymi można znaleźć sterowniki do debugowania różnorodnych androidowych urządzeń mobilnych

Producent	Adres strony ze sterownikami
Acer	http://goo.gl/OHJ2gb
Alcatel	http://goo.gl/VGE2c3
Asus	http://goo.gl/TYyTWX
Dell	http://goo.gl/V3Ue7q
Foxconn	http://goo.gl/ipznd8
Fujitsu	http://goo.gl/SjHaKC
Garmin-Asus	https://goo.gl/w6FDb2
Hisense	http://goo.gl/4cKyyg
HTC	http://goo.gl/Vvqb9J (należy wybrać zakładkę „suport" i wskazać urządzenie. Różne wersje regionalne strony będą zawierać różne sterowniki)
Huawei	http://goo.gl/vhsMex
Intel	http://goo.gl/hFLjQK
KT Tech	http://goo.gl/YWtW7a
Kyocera	http://goo.gl/cYqhZj
Lenovo	http://goo.gl/G82QZf
LGE	http://goo.gl/Bi4O4s
Motorola	http://goo.gl/x680Q2
MTK	http://goo.gl/J9REGM
Oppo	http://goo.gl/zAVwIQ
Pantech	http://goo.gl/4YwrN9
Pegatron	http://goo.gl/GWNoOo
Samsung	http://goo.gl/vnUhVD
Sharp	http://goo.gl/hm4ieW
SK Telesys	http://goo.gl/s0khhk
Sony	http://goo.gl/h0pYWF
Teleepoch	http://goo.gl/F2ytSQ
Toshiba	http://goo.gl/m5uhOJ
Yulong Coolpad	http://goo.gl/B35dBy
Xiaomi	http://goo.gl/GFFZtR
ZTE	http://goo.gl/YPAaec

Po lewej stronie znajduje się menu modułu *Inspect* przeglądarki Chrome. Pierwszą zakładką jest właśnie *Devices* (rysunek 7), na której pokazywane są wszystkie urządzenia, które można debugować. Zakładka *Pages* pozwala na przełączenie debugera do trybu mobilnego (mobile view), który omówimy później. Zakładka *Extensions* zawiera zainstalowane i uruchomione rozszerzenia. W naszym przypadku używamy jednego: „Allow-Control-Allow-Origin: *". Zainstalowaliśmy je by uniknąć jakichkolwiek problemów z aplikacjami, które w trakcie swojego działania odwołują się zapytaniami AJAXowymi do innych serwerów. Pozostałych zakładek nie będziemy omawiać.

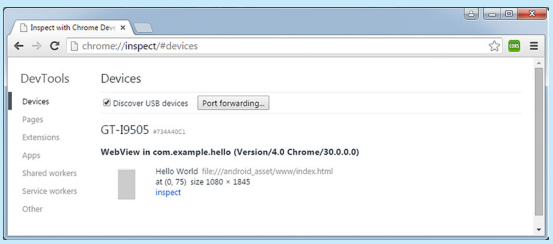
Jeśli podłączone urządzenie nie jest widoczne na liście w zakładce *Devices*, prawdopodobnie problemem jest brak sterowników. Można je pobrać ze strony Google lub ze strony producenta telefonu. Zalecamy skorzystanie przede wszystkim ze sterowników producenta sprzętu (w przypadku Samsunga znaleźć je można w dziale „Samsung Developers" *http://goo.gl/Q9aAI8*. Tymczasem u Google można ich szukać pod adresem *http://goo.gl/ps0BPC*.

Lista innych użytecznych adresów, pod którymi jest szansa znalezienia sterowników do debugowania urządzeń poszczególnych producentów została zebrana w **tabeli 4**. Jeśli urządzenie było widoczne na liście, ale przestało, warto zrestartować komputer i spróbować ponownie. Czasem pomaga też restart urządzenia mobilnego lub wyłączenie i ponowne włączenie w telefonie debugowania przez USB.

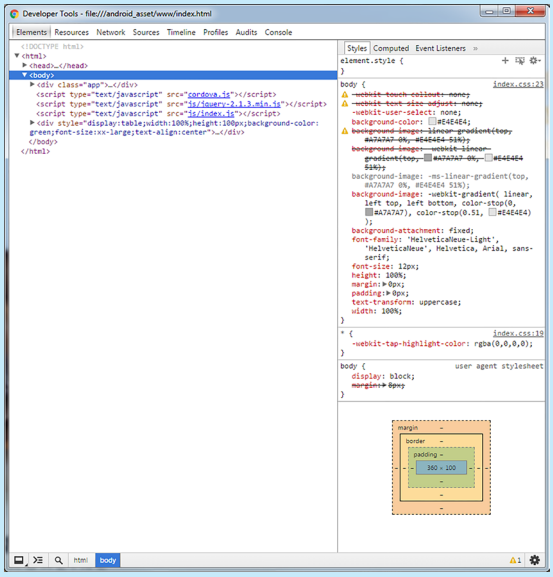
Debugowanie HTML i CSS

Jeśli projektowana aplikacja nie zawiera żadnych poważnych błędów, które powodowałyby jej wyłączanie się, można zacząć debugowanie od kodu HTML, CSS i JavaScript, czyli tego, który programista pisze, gdy tworzy aplikację w Cordovie. Jeśli błędy w programie powodują zupełną niestabilność aplikacji, wtedy konieczne staje się debugowanie języka natywnego, co w przypadku aplikacji tworzonych z użyciem platformy Cordova, najczęściej sprawdza się do debugowania kodu pluginów, a czasem także samej Cordovy. To zagadnienie omówimy w kolejnej części kursu, a tymczasem zaczniemy od najprostszego zadania, czyli debugowania kodu HTML i CSS, które odpowiadają za wygląd interfejsu użytkownika.

Zgodnie z tym, co napisano w pierwszych częściach kursu, aplikacje Cordovy są w praktyce stronami internetowymi, które odwołują się – poprzez biblioteki Cordovy – do zasobów sprzętowych urządzenia mobilnego. Dlatego właśnie do debugowania aplikacji używa się przeglądarki internetowej, gdyż w praktyce oznacza to debugowanie kodu strony internetowej. Poszczególne przeglądarki mają swoje własne i czasem bardzo potężne narzędzia do debugowania, umożliwiające nawet debugowanie animacji napisanych w postaci kodu CSS, czego świetnym przykładem są najnowsze Developer Tools z Mozilli Firefox i je również można wykorzystać do analizowania stworzonego kodu, ale w przypadku



Rysunek 7. Moduł Inspect przeglądarki Google Chrome – zakładka „Devices"



Rysunek 8. Zakładka Elements modułu debugera z widokiem zadeklarowanych styli

Androida, łączność z urządzeniem mobilnym i podgląd jego parametrów najlepiej zapewnia Google Chrome.

Naciśnięcie przycisku *Inspect* przy wybranej, uruchomionej na urządzeniu mobilnym aplikacji powoduje przejście do okna debugowania kodu HTML i CSS danego programu. W naszym przypadku zajmiemy się utworzoną w poprzedniej części kursu aplikacją wideo-domofonu.

Ekran debugera z Chrome jest podzielony na 8 zakładek:

- **Elements** – do przeglądania elementów strony internetowej.
- **Resources** – do przeglądania zasobów używanych przez stronę internetową.
- **Network** – do przeglądania ruchu sieciowego generowanego przez stronę internetową.
- **Sources** – do przeglądania kodu źródłowego plików strony.
- **Timeline** – do monitorowania zmian w pracy aplikacji w czasie.
- **Profiles** – do profilowania kodu, a więc podglądania sposobu wykonania kodu JavaScriptowego oraz wykorzystania stosu.
- **Audits** – do sprawdzania optymalności strony internetowej.
- **Console** – do posługiwania się konsolą JavaScript.

W zakładce **Elements** (**rysunek 8**) prezentowana jest aktualna struktura HTML strony. Nie musi to być dokładnie treść pliku źródłowego index.html, gdyż wyświetlany kod jest już po przetworzeniu. Jeśli w pliku źródłowym znajdowały się jakieś sprzeczne deklaracje, to w widoku zakładki Elements będą one już rozwiązane zgodnie z algorytmem rozwiązywania konfliktów danej przeglądarki. Ponadto widok ten zawiera elementy tworzone dynamiczne, np. za pomocą JavaScriptu i pozwala podejrzeć aktualny stan „na żywo" interpretowanego kodu HTML.

Poszczególne bloki kodu można rozwijać lub zwijać, by ułatwić przeglądanie treści. Co więcej, najechanie na dany blok myszką powoduje podświetlenie go i… zaznaczenie go wraz z dymkiem określającym jego parametry na urządzeniu mobilnym! Przykładowo najechanie na warstwę o identyfikatorze *scanButton* w kodzie aplikacji domofonu powoduje podświetlenie jej na ekranie telefonu (**rysunek 9**).

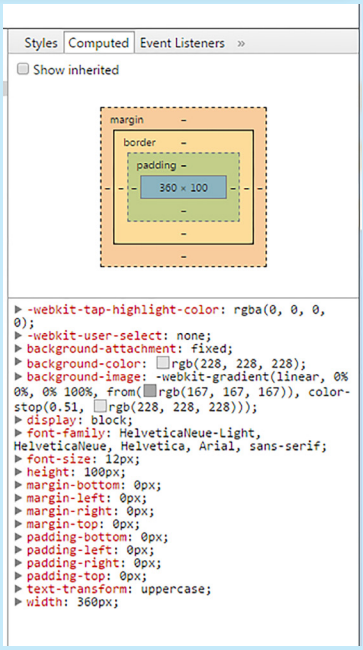
Na dole okna *Elements* znajduje się pasek, w którym pokazana jest hierarchia przynależności wybranego aktualnie elementu strony, przycisk wyszukiwania, przycisk widoku konsoli oraz – po prawej – informacja o otrzymanych komunikatach lub błędach JavaScript i przycisk ustawień. Większość z tych elementów powtarza się także przy innych zakładkach.

Prawa część okna *Elements* zawiera informacje o zadeklarowanych stylach, dotyczących zaznaczonego elementu. Wskazane są kompletne treści deklaracji CSS dla danego elementu, to w którym pliku są zapisane (o ile nie są wpisane ręcznie w kodzie HTML) oraz czy wynikają z przynależności

Nowości w Cordovie

Od momentu opracowania poprzedniej części kursu, w platformie Cordova pojawiło się kilka nowości. Aktualna wersja Cordovy i jej bibliotek to 5.4.1. Oprócz usunięcia błędów związanych z bezpieczeństwem, wprowadzono m.in. mechanizm automatycznego informowania o dostępności nowszej wersji Cordovy. Wprowadzono też obsługę Node v4 i v5 oraz nowe wersje platform na Androida, iOS i Windows 8.1/10. Cordova teraz automatycznie zmienia stare nazwy pluginów na nowe (o ile stare są zapisane w repozytorium tłumaczeń nazw). Warto dodać, że nowa platforma Cordova Android 5.0.0. obsługuje Androida Marshmallow, a więc wprowadzony w nim, zaawansowany mechanizm ustalania uprawnień dla aplikacji. W przypadku platformy Cordova Windows 4.2.0 poprawiono obsługę przycisku „wstecz". Zaktualizowano też większość podstawowych wtyczek. Użyteczne podczas ich wyszukiwania są nowe mechanizmy sortowania (po popularności, dacie publikacji i jakości) oraz filtrowania po systemie operacyjnym).

Rysunek 9. W trakcie debugowania z użyciem przeglądarki Google Chrome, elementy podświetlane kursorem w widoku zakładki Elements są zaznaczane także na podłączonym przez USB urządzeniu mobilnym



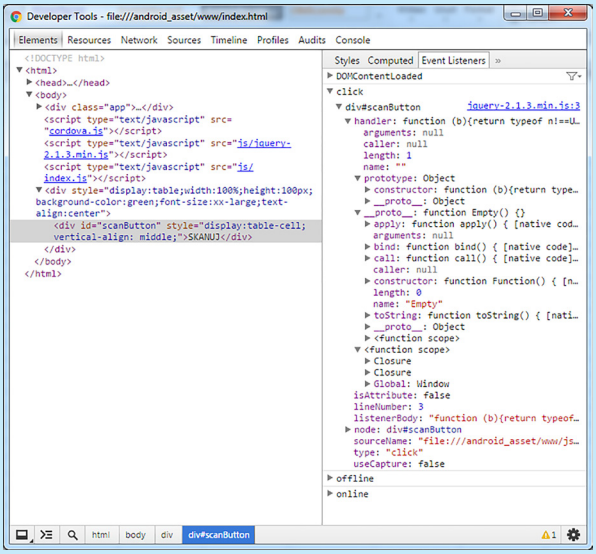
Rysunek 10. Widok wyliczonych wartości styli elementów strony internetowej

do elementu położonego wyżej w hierarchii. Deklaracje przekreślone wskazują, że zostały zastąpione przez inne, ważniejsze (np. dotyczące danego obiektu bardziej bezpośrednio). Na rys. 8 widać, że deklaracja wielkości marginesu dla wszystkich elementów podrzędnych blokowi body została nadpisana inną deklaracją marginesu. Zasady dotyczące hierarchii ważności są opisane w dokumentacji języka CSS. Deklaracje ze znakami wykrzykników wskazują, że może z nimi występować jakiś problem, choć w praktyce tego typu niezgodności w CSS warto diagnozować w praktyce i jeśli wszystko wygląda dobrze, to nie trzeba ich starać się poprawiać.

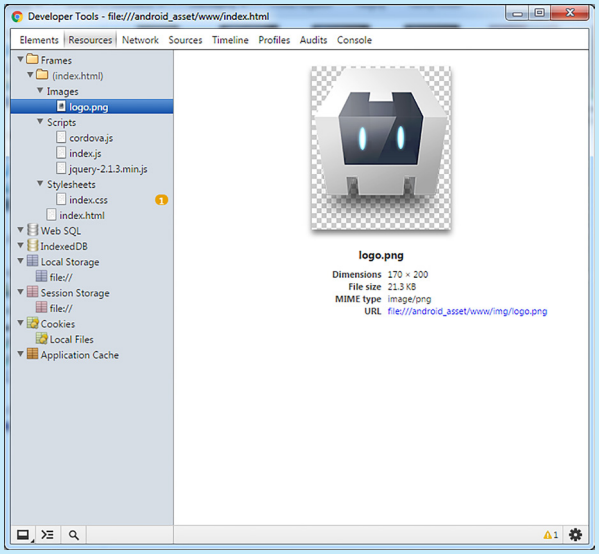
W prawej dolnej części ekranu *Elements* znajduje się szereg koncentrycznych prostokątów. Pokazują one wymiary zaznaczonego obiektu w sposób blokowy. W samym środku pokazywane są aktualna szerokość i wysokość wewnętrzna elementu, a następnie grubość odstępu wewnątrz ew. ramki obiektu (tzw. padding), grubość ramki (border) i szerokość marginesu na zewnątrz ramki (margin).

Klikając na widok *Computed* (**rysunek 10**), po prawej stronie ekranu, otrzymujemy podsumowanie styli, w którym wyświetlają się aktualnie wyliczone wartości, powstałe w oparciu o wszystkie adekwatne deklaracje. W obu widokach (zarówno *Styles*, jak i *Computed*) można ręcznie zmieniać deklaracje w funkcjonującej aplikacji, sprawdzając jak wpłynie to na jej wygląd. By jednak zmiany zostały zapisane, trzeba je następnie samodzielnie wprowadzić do plików źródłowych i przekompilować aplikację Cordovy.

Trzeci widok – *Event Listeners* – ułatwia dekodowanie JavaScriptu, gdyż przechowuje informacje o zdarzeniach przypisanych do poszczególnych obiektów HTML. To bardzo pomocne narzędzie. Na **rysunku 11** wyraźnie widać,



Rysunek 11. Widok zdarzeń przypisanych do wybranego elementu strony internetowej



Rysunek 12. Lista zasobów i sposób prezentacji informacji o pliku graficznym

że do warstwy *scanButton*, która służy jako przycisk, jest przypisane zdarzenie *click*, mimo że w samym kodzie HTML w ogóle go nie zadeklarowano. Zostało ono przypisane przez JavaScript, już po uruchomieniu aplikacji, co pozwala upewnić się, że fragment kodu przypisujący akcje do przycisków został poprawnie napisany. Rozwinięcie zdarzeń daje dostęp do dalszych informacji na temat miejsca i sposobu dokonania przypisania.

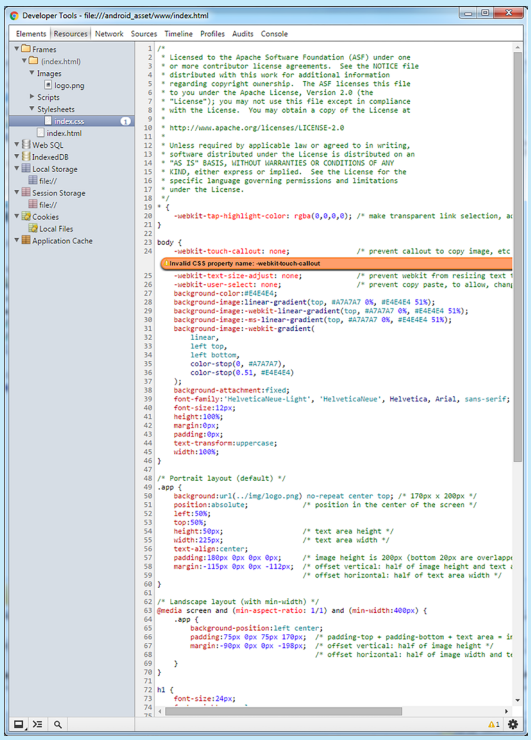
Przeglądanie zasobów

Każda strona internetowa, a więc i aplikacja Cordovy, składa się z szeregu plików, które w naszym przypadku zostały później skompresowane do postaci paczki w formacie APK. W trakcie uruchamiania aplikacji, paczka ta jest rozpakowywana do pamięci urządzenia mobilnego. Wśród plików, które w sobie mieści są też wszystkie dołączone pliki potrzebne do poprawnego wyświetlenia strony internetowej – zarówno skrypty, pliki styli, jak i graficzne. Zakładka *Resources* narzędzia debugującego w Chrome pozwala przeglądać te zasoby, a dokładniej te, do których odniesienia pojawiły się w trakcie działania aplikacji. Widać to na **rysunku 12**, na którym po lewej stronie znajduje się lista wykorzystywanych zasobów; jest tam m.in. plik graficzny *logo.png*, do którego odniesienie znajdowało się w pliku *index.html*, ładowanym przy starcie aplikacji. Wybranie danego zasobu z listy powoduje wyświetlenie go po prawej stronie. W przypadku plików graficznych prezentowana jest miniaturka oraz parametry zasobu (wymiary, rozmiar). W przypadku plików styli CSS prezentowany jest zawarty w nich kod, wraz z zaznaczonymi ewentualnymi linijkami, które mogą powodować błędy (**rysunek 13**).

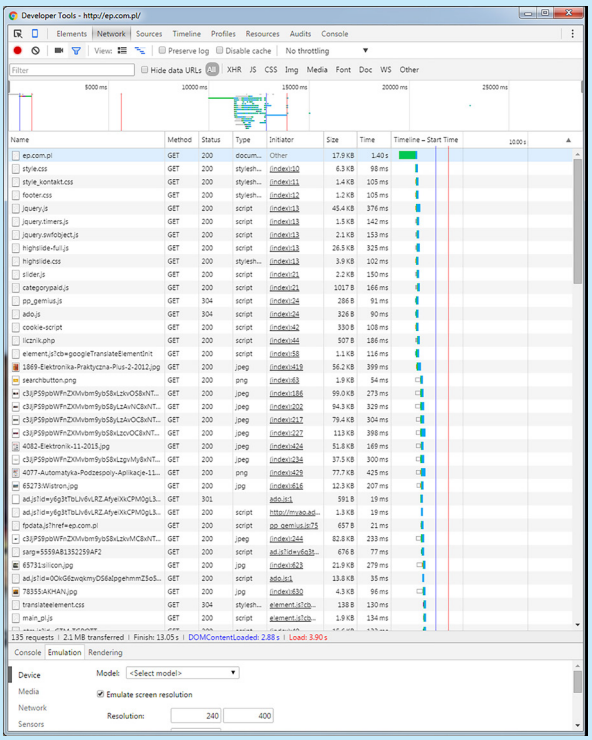
Zakładka przeglądania zasobów pozwala sprawdzić, czy zapisane w aplikacji pliki są dokładnie takie, jak powinny być oraz, ułatwia znajdowanie w nich błędów.

Ruch sieciowy

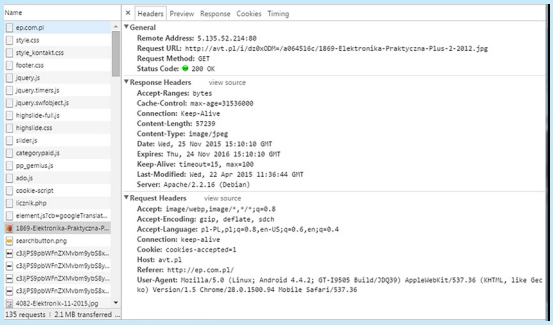
Nieocenioną pomoc niesie zakładka *Network*, która umożliwia przeglądanie ruchu sieciowego, generowanego przez



Rysunek 13. Lista zasobów i wyświetlony plik stylu CSS, wraz z wskazaniem problematycznych deklaracji



Rysunek 14. Zakładka „Network" dla sytuacji, w której ładowana jest strona internetowa Elektroniki Praktycznej



Rysunek 15. Szczegóły żądania zasobu

stronę internetową. W przypadku Cordovy prezentowane są wszystkie zewnętrzne żądania HTTP wysyłane przez program. Zazwyczaj odnoszą się one do załadowania zewnętrznych zasobów, takich jak grafiki lub do wyzwolenia konkretnych operacji na zewnętrznych serwerach – tak jak to robiliśmy w pierwszych częściach kursu. Omawiana zakładka jest tak jakby selektywnym *snifferem* sieciowym, wyposażonym dodatkowo w dosyć zaawansowane algorytmy przetwarzania danych.

Do jej opisania posłużymy się **rysunkiem 14**, na którym pokazaliśmy, co się dzieje w sieci, gdy wywołujemy stronę internetową Elektroniki Praktycznej (gdybyśmy np. chcieli ją wyświetlić w oknie aplikacji).

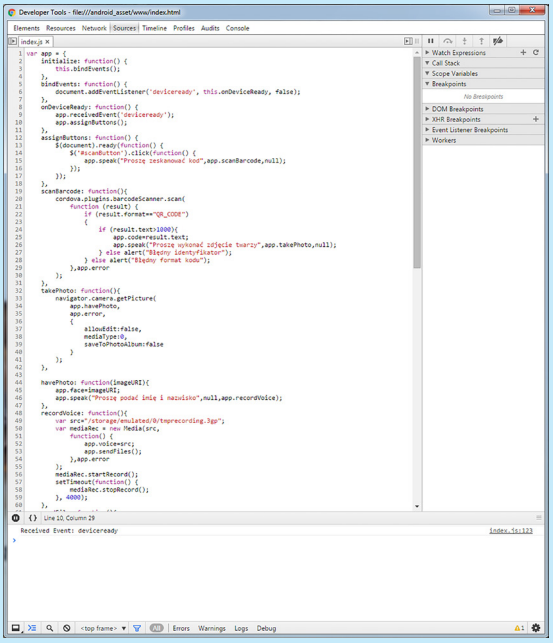
Na górze ekranu, zaraz pod przyciskami, znajduje się wykres przebiegu całego ruchu. Kolorowe paski reprezentują przebieg procesu ładowania poszczególnych żądanych elementów. W głównej części ekranu znajduje się lista żądanych zasobów. Na liście umieszczono nazwy żądanych plików wraz z ewentualnymi treściami zapytań GET, metodą stosowana dla danego żądania (najczęściej GET lub POST), status odpowiedzi serwera, typ otrzymanego pliku, miejsce, skąd pochodziło żądanie, rozmiar pobranego pliku, czas pobierania danego zasobu oraz przebieg tego procesu w czasie.

Debugując aplikację, która komunikuje się z zewnętrznym serwerem możemy przede wszystkim sprawdzić, czy żądanie do zewnętrznego serwera jest w ogóle wysyłane i czy serwer je poprawnie przyjął, a także znaleźć numer linijki w pliku, w którym polecenie żądania zostało zapisane. Ale to nie wszystko, bo klikając na żądany zasób na liście można wyświetlić szczegóły danej operacji (**rysunek 15**). Prezentowane są m.in. nagłówki żądania, w tym dokładny adres URL i adres IP, sposób kodowania znaków, jeśli ma znaczenie itd. Dostępne są też: podgląd zasobu albo dokładna treść odpowiedzi w przypadku plików tekstowych, wskazane są ciasteczka, które przesłano w trakcie żądania oraz dokładne szczegóły czasowe procesu, wraz z możliwością uzyskania wyjaśnień odnośnie do ich znaczenia. Ten ostatni element pozwala na dopracowywanie wydajności aplikacji poprzez sprawdzenie, w których momentach musi ona czekać na zewnętrzne serwery.

Na dole listy żądań znajduje się podsumowanie, obejmujące m.in. sumaryczny czas ładowania. Jest też możliwość włączenia emulacji zaawansowanych parametrów żądania (**rysunek 14**), ustalając np. sposób w jaki aplikacja będzie się przedstawiać zewnętrznemu serwerowi, jednak w przypadku debugowania aplikacji mobilnych funkcja ta będzie rzadko potrzebna. Bardziej przydatne może okazać się filtrowanie żądań, poprzez wskazanie określonego czasu, w którym interesują nas wywołania lub wybór rodzaju żądanych zasobów. Dla aplikacji mobilnych pomocne może być użycie funkcji ograniczania szybkości transferu. Trzeba mieć na uwadze, że użytkownicy mogą korzystać z programu np. będąc gdzieś w górach, gdzie dostęp do Wi-Fi ani 3G nie jest powszechny. W takich sytuacjach będą np. łączyli się przez GPRS. Dostępna na górze ekranu (również rysunek 14) lista z domyślną etykietą „No Throttling" pozwala wybrać rodzaj symulowanego łącza, co spowoduje, że następne zdalne wywołanie będzie miało ograniczony transfer, tak by nie przekroczyć parametrów wskazanego interfejsu internetowego.

Źródła

Zakładka źródła (*Sources*) to element najbardziej przypominający klasyczny debuger. Ponieważ uruchamiany kod aplikacji nie jest tak naprawdę skompilowany, a jedynie



Rysunek 16. Zakładka źródła

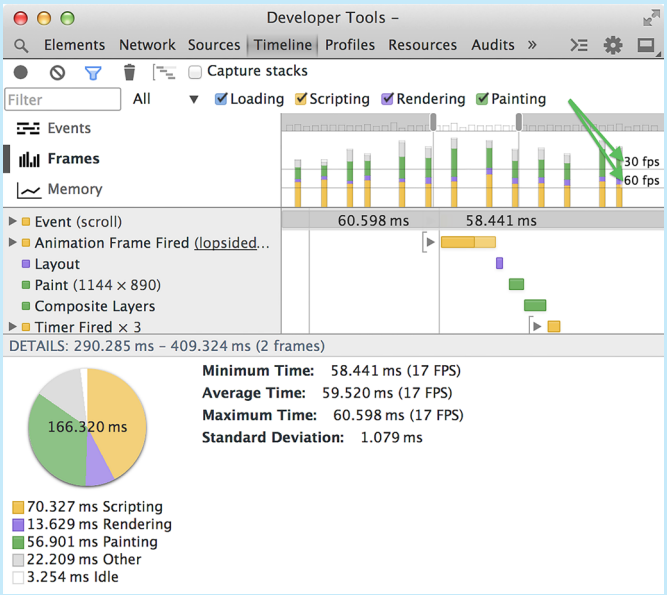
interpretowany, możliwe jest przeglądanie źródeł, tak samo jak podczas debugowania programów uruchamianych ze zintegrowanych środowisk deweloperskich.

Zakładka źródła (**rysunki 15** i **16**) pozwala na wstawianie punktów wstrzymania wykonywania kodu (breakpoints), monitorowanie wartości zmiennych w pamięci, przeglądanie stosu itd. Domyślnie na dole ekranu widoczny jest podgląd konsoli.

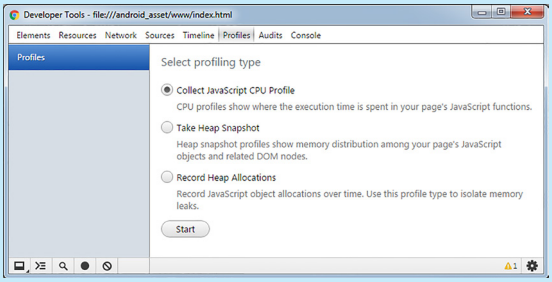
Profilowanie i audyt

Kolejne dwie zakładki: *Timeline* i *Profiles* pozwalają na optymalizację szybkości działania strony internetowej. W przypadku aplikacji Cordovy, pierwsza z zakładek ma małe zastosowanie. Pozwala zorientować się, ile czasu zajęły kolejne operacje potrzebne do przygotowania widoku strony, przy czym dotyczy to nie tyle ruchu sieciowego, co czasu potrzebnego na przetworzenie skryptów, wyliczenie wynikowych wartości styli elementów oraz wyświetlenie odpowiedzi. Prawie każdy z fragmentów procesu przygotowania widoku da się oddzielnie obejrzeć (**rysunek 17**).

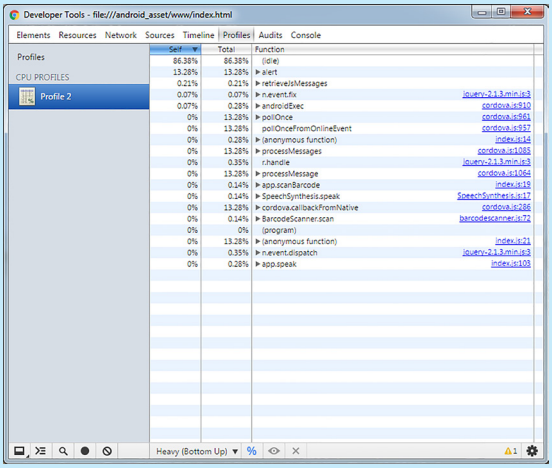
W przypadku aplikacji Cordovy znacznie więcej da się wywnioskować z zakładki profili. Pozwala ona na uruchomienie trzech akcji (**rysunek 18**):



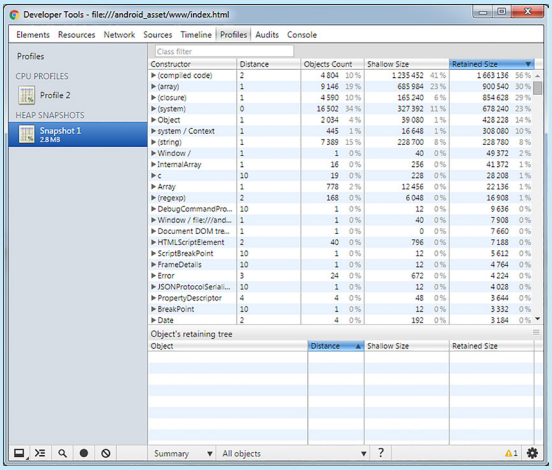
Rysunek 17. Zakładka Timeline z szczegółowym widokiem czasu wykonywanych operacji



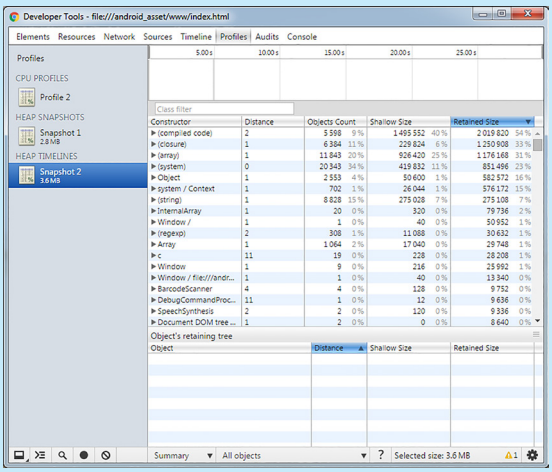
Rysunek 18. Zakładka Profiles



Rysunek 19. Monitorowanie czasu poświęconego przez procesor na wykonywanie poszczególnych funkcji JavaScript



Rysunek 20. Przeglądanie stanu stosu



Rysunek 21. Monitorowanie akcji wykonywanych na stosie

- Monitorowania czasu pracy procesora, poświęcanego poszczególnym wywoływanym funkcjom JavaScript (**rysunek 19**).
- Monitorowania stanu stosu (**rysunek 20**).
- Przeglądania operacji wykonywanych na stosie (**rysunek 21**).

Najbardziej przydatne do optymalizacji aplikacji będzie monitorowanie czasu wykonywania poszczególnych funkcji. Widok można posortować wg odsetka zajmowanego czasu, albo wyświetlić przebieg wykonywanych operacji na wykresie czasowym. Podane są nazwy plików i numery linijek, w których znajduje się wykonywana funkcja.

Zrzut stosu jest dosyć obszerny, a każda z pozycji może być rozłożona na szereg podelementów i elementów zależnych, co w efekcie sprawia, że przeglądanie takiego raportu jest skomplikowane. Podobnie z monitorowaniem dostępu do stosu. Wynika to przede wszystkim z faktu, że aplikacji Cordovy działają wraz z dużą liczbą bibliotek, a operacje napisane w JavaScripcie są następnie przenoszone na język natywny systemu operacyjnego – w tym wypadku Javę.

Dodatkowym narzędziem, udostępnionym przez Google w Chrome jest zakładka **Audits**, w ramach której programista otrzymuje szereg wskazówek odnośnie działania stworzonej aplikacji (**rysunek 22**). Podawane są sugestie, jak przyspieszyć jej działanie m.in. poprzez usuwanie nieużywanych deklaracji CSS, wykorzystanie pamięci cache, łączenie ze sobą plików JavaScript, czy też odgórne zdefiniowanie wymiarów grafik, aby przeglądarka mogła szybciej dokonywać obliczeń. Część z tych zaleceń nie ma zbytnio zastosowania do aplikacji Cordovy, a raczej tylko do stron internetowych, ale niektóre wskazówki będą naprawdę pomocne, szczególnie w końcowym etapie tworzenia aplikacji.

Konsola JavaScript

Ostatnia zakładka pozwala wyświetlić duży podgląd widoku konsoli (**rysunek 23**). Możliwe jest filtrowanie prezentowanych tam komunikatów w zależności od stopnia ich ważności. Podgląd konsoli jest bardzo przydatny na etapie debugowania, gdyż można do niej przesyłać dowolne informacje diagnostyczne. Używając polecenia *console.log()* podaje się treść, która będzie przekazana do konsoli JavaScript, zamiast być wyświetlana użytkownikowi. Podgląd konsoli dostępny jest też w dolnej części niektórych innych ekranów debugera.

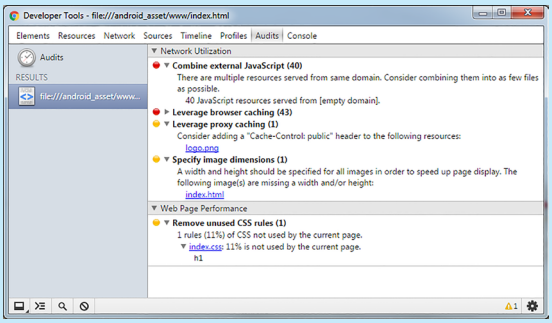
Szybkie debugowanie interfejsu użytkownika

Opisana powyżej metoda debugowania jest precyzyjna, gdyż pozwala przetestować aplikację na rzeczywistym urządzeniu, dzięki wyjątkowej funkcji debugowania przez USB, dostępnej w Google Chrome. Niestety, jeśli projektujemy interfejs użytkownika i chcemy go dopracować do perfekcji z dokładnością do pojedynczych pikseli, często konieczne jest wiele iteracji wprowadzania zmian i sprawdzania wyglądu programu. Jeśli każda z iteracji musiałaby wiązać się z kompilacją, która nierzadko trwa około minuty, to byłoby to zadanie bardzo czasochłonne. Dlatego w takich sytuacjach warto skorzystać z metody debugowania, którą nazwaliśmy powierzchowną. Pozwala ona znaleźć wiele błędów, szczególnie w składni JavaScript i w wartościach deklaracji w CSS, bez konieczności każdorazowego rekompilowania kodu.

Aby skorzystać z tej możliwości, należy potraktować tworzoną aplikację tak, jak każdą inną stronę internetową. Edytowany przez nas plik *index.html* (np. *C:\kursEP\domofon\www\index.html*), w którym znajduje się główny kod HTML aplikacji należy otworzyć w przeglądarce internetowej. Wybór przeglądarki podyktowany jest tymi samymi warunkami, co w przypadku wcześniej opisanej procedury debugowania. Prezentujemy w skrócie trzy narzędzia.

Firefox FireBug – jest to popularne narzędzie do debugowania stron internetowych, dostępne jako dodatek w przeglądarce Mozilla Firefox (**rysunek 24**). Pozwala szybko przejrzeć treść strony w sposób podobny jak pokazywaliśmy w Google Chrome, ale jest nieco mniej skomplikowany. Ma też oddzielną zakładkę do ciasteczek. Nie da się na nim debugować przez USB aplikacji z Androida, więc można go używać tylko do robienia drobnych poprawek, takich jak dobieranie kolorów i ustawianie elementów interfejsu użytkownika.

Google Chrome mobile view – to tryb, w którym Google Chrome symuluje wyświetlanie strony internetowej na wybranym urządzeniu mobilnym. Użytkownik może



Rysunek 22. Automatyczny audyt aplikacji

samodzielnie określić rozdzielczość ekranu urządzenia, jego orientację oraz sposób, w jaki identyfikuje się serwerowi. Można też wybrać któreś z predefiniowanych urządzeń i zastosować skalowanie używane w widoku WebView tego urządzenia. Przykładowo, na **rysunku 25** pokazano stronę widoku aplikacji tak, jak będzie wyglądać na telefonie Samsung Galaxy S4 w orientacji pionowej. Widok mobilny włącza się w debugerze Google Chrome poprzez kliknięcie na ikonkę smartfonu. Pozostałe funkcje debugera są mniej więcej takie same, jak opisano wcześniej.

Mozilla Developer Tools – to potężne narzędzie, szczególnie w jego ostatniej wersji. Również pozwala na włączenie trybu mobilnego (**rysunek 26**), ale nie służy do debugowania programów Androidowych przez USB. Atrakcyjną funkcją tego narzędzia jest możliwość debugowania animacji stworzonych w CSSie, co widać akurat na załączonej ilustracji. Pulsująca dzięki klasie *blink* warstwa *deviceready* wygląda jak zapętlona 3-sekundowa animacja. Mozilla Developer Tools pozwala na wstrzymywanie, przyspieszanie czy odwracanie tej animacji.

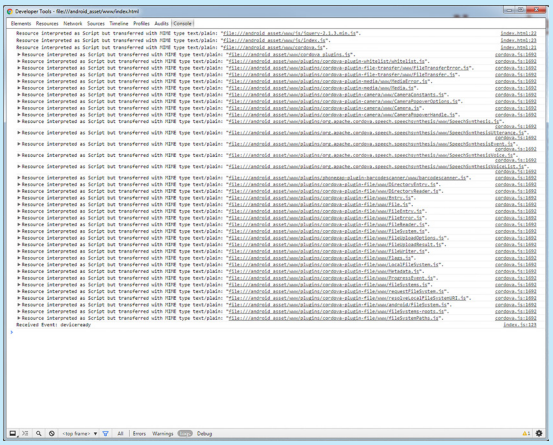
W trakcie debugowania nieskompilowanych aplikacji Cordovy, a jedynie ich plików HTML, CSS i JavaScript, część funkcji nie będzie działała. Przede wszystkim zdarzenie *deviceready* zostanie wywołane, funkcje odnoszące się bezpośrednio do sprzętu nie będą działać, a niektóre ładowane biblioteki mogą nie być znajdywane. Ten ostatni problem można rozwiązać poprzez tymczasowe skopiowanie ich pod do odpowiednich katalogów, w których są wyszukiwane lub poprzez debugowanie plików nie w podkatalogu *www* katalogu głównego aplikacji, ale w podkatalogu *platforms\android\assets\www* katalogu głównego aplikacji. Trzeba jednak pamiętać, że dokonywane tam zmiany nie zostaną automatycznie zapisane – będą zastąpione podczas kompilacji plikami z wcześniej wspomnianego katalogu *www*.

Alternatywą jest doinstalowanie do projektu platformy **browser**, która pozwala na uruchamianie aplikacji w przeglądarce internetowej, a następnie np. debugowanie ich. Niestety, wiele dodatków (ok. 95%) nie wspiera tej platformy, w związku z czym nie rozwiąże to wielu problemów. Naszym zdaniem, w przypadku zaawansowanych projektów korzystnie jest po prostu tymczasowo przekopiować potrzebne pliki i dopisać funkcję, która dla celów debugowania będzie zastępowała akcje wykonujące się po nadejściu zdarzenia **deviceready**, tak by można było wyświetlić w okienku przeglądarki wymagane widoki do debugowania. Naturalnie, po dopracowaniu wyglądu, a przed kompilacją kodu, funkcje te oraz dodatkowe pliki należy usunąć.

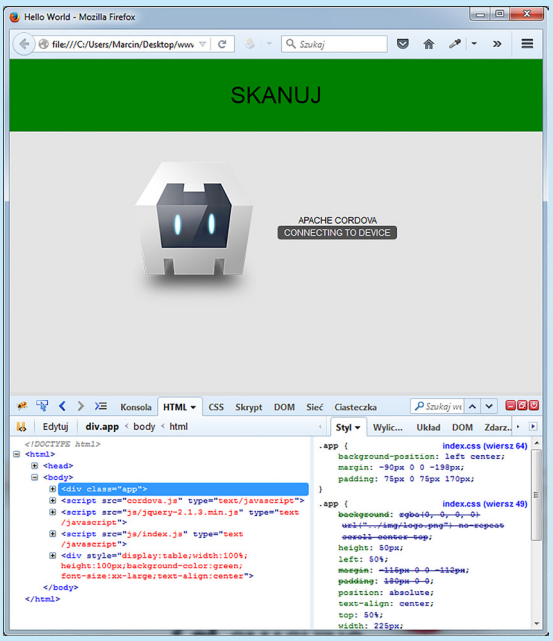
Podsumowanie

Opisane w niniejszej części kursu narzędzia pozwalają dogłębnie przetestować interfejs użytkownika aplikacji i w praktyce powinny wystarczyć większości osób piszących aplikacje Cordovy szczególnie, jeśli mają być kompilowane pod Androida. W kolejnej części kursu pokażemy jednak, jak debugować kod Java na Androidzie, co będzie przydatne dla osób piszących bardziej zaawansowane aplikacji, wymagające poprawiania Cordovy lub pisania własnych pluginów. Wiedza z kolejnej części kursu pomoże też wykrywać błędy w pluginach, dzięki czemu programista – nawet, jeśli nie zna Javy i nie zamierza samodzielnie modyfikować wtyczek – będzie mógł zidentyfikować, która wtyczka, na którym etapie powoduje problem i zdecydować o jej ewentualnej zamianie na alternatywną.

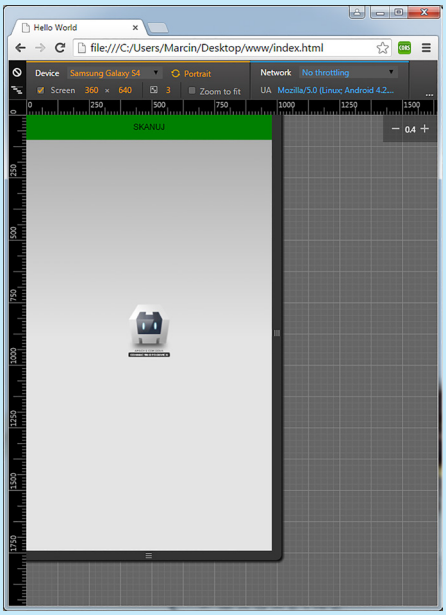
Marcin Karbowniczek, EP



Rysunek 23. Okienko konsoli JavaScript



Rysunek 24. Mozilla Firefox z dodatkiem FireBug, w trakcie debugowania pliku index.html aplikacji domofonu



Rysunek 25. Google Chrome w trybie widoku mobilnego pozwala symulować wygląd stron internetowych na różnych urządzeniach mobilnych