Autor

Zaawansowane programowanie na platformie Android

Część I: There is no app – kluczowe koncepty stojące za filozofią platformy Android

Artykuł jest pierwszym z serii tekstów poświęconych zaawansowanym aspektom tworzenia aplikacji na platformie Android. Na podstawie swoich doświadczeń w pracy z zespołami developerskimi postanowiłem poświęcić część pierwszą uporządkowaniu ogólnej filozofii Androida. Natomiast kolejne części będą poświęcone zaawansowanym technikom programistycznym.

POCZĄTKI

W momencie gdy zaczynamy development na nowej platformie, często eksperymentujemy, używając takich technik jak copy-paste. Jest to pomocne w początkowej fazie nauki, jednak przykłady kodu w tutorialach dla początkujących programistów mają charakter typowo dydaktyczny – "drzewa nie mogą przesłonić lasu". Powoduje to trochę wypaczony obraz tego, jak powinno wyglądać oprogramowanie na danej platformie. Nie inaczej jest z większością materiałów dotyczących Androida. Często w momencie, gdy już coś wiemy i już coś działa, a gonią nas terminy, trudno jest porzucić rozwijanie oprogramowania i jeszcze raz wrócić do dostępnych materiałów – do korzeni – aby zrozumieć, co tak naprawdę robimy. W przypadku Androida może być to tragiczne w skutkach, powodując między innymi przerośnięte i ciężkie do utrzymania Activity, które jest jednym z głównych komponentów aplikacji Androidowej, lub wycieki pamięci. Oczywiście rzutuje to negatywnie na jakość wytwarzanego oprogramowania.

Programując na Androida, spotykamy się co chwilę z jakimś bolesnym do obsłużenia mechanizmem, takim jak obsługa zmian konfiguracji czy konieczność zapisywania stanu. Pierwszą rzeczą, która przychodzi nam do głowy, jest *„obejdę to!"*, co niestety często nie jest możliwe i nie dość, że kończy się wybuchem naszej aplikacji, to dzieje się to nie wtedy, kiedy byśmy się tego spodziewali. Zrozumienie, jakimi prawami rządzi się cała platforma, może stać się krokiem do akceptacji tych niewygodnych mechanizmów.

DOSTĘPNE KLOCKI

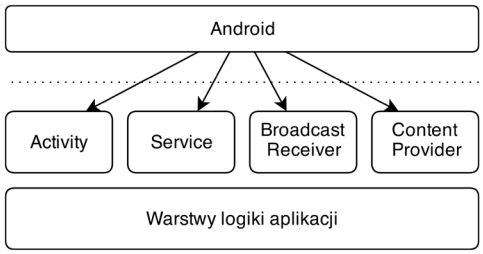
Gdy popatrzymy na warstwę aplikacji Androida z pewnego dystansu, to dostrzeżemy, że wszystkie aplikacje składają się z 4 rodzajów komponentów. Aplikacje dostarczone przez system rządzą się dokładnie takimi samymi prawami, co te tworzone przez niezależnych developerów. Wszystkie dostępne komponenty są od siebie całkowicie niezależne, a ich cykl życia kontrolowany jest przez system – na wiele rzeczy nie mamy wpływu.

Pierwszym komponentem, który poznaje programista Androida, jest **Activity**. Komponent ten zajmuje się warstwą prezentacji i powinien odpowiadać jednej czynności wykonywanej przez użytkownika, np. pisaniu maila. Można powiedzieć, że **Activity** spełnia rolę kontrolera zarządzającego widokami, które ukazują się oczom użytkownika. Niestety, bardzo łatwo jest wpaść w pułapkę i nadać **Activity** zbyt dużą odpowiedzialność, zamiast wydelegować pracę pozostałym dostępnym komponentom.

Często niedocenianym, a bardzo użytecznym klockiem budulcowym aplikacji jest **Service** – usługa, która powinna służyć do wykonywania dłużej trwających operacji, takich jak odtwarzanie muzyki czy operacje sieciowe. Bardzo użytecznym rodzajem serwisu, który może być wykorzystany do zadań typu "uruchom i zapomnij", jest **IntentService**, wykonujący zlecone mu zadania w innym wątku. Serwisy mogą również służyć do wystawiania części funkcjonalności na zewnątrz naszej aplikacji.

Kolejnym komponentem jest **BroadcastReceiver** – odbiorca rozgłoszeń, który powinien stanowić bramę do naszej aplikacji i delegować pracę innym komponentom (sam nie zawiera kluczowej logiki). Może on nasłuchiwać na różne zdarzenia systemowe, takie jak: *WIFI zostało włączone*, czy *system został uruchomiony*.

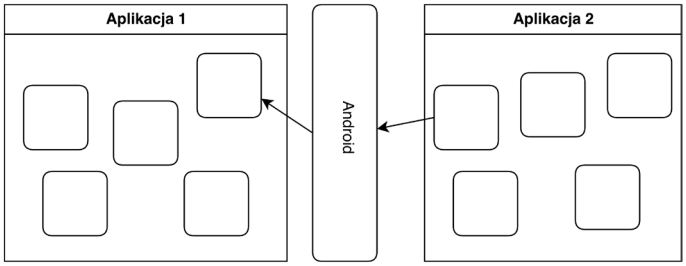
Istnieje jeszcze **ContentProvider** – zarządzający treścią aplikacji. Stanowi on fasadę na dane, które przechowuje nasza aplikacja.



Rysunek 1. Komponenty a system

Użycie każdego z wymienionych powyżej komponentów powinno stanowić klej pomiędzy logiką pisanej aplikacji a systemem oraz być najcieńszą możliwie warstwą odwołującą się do niezależnej od platformy logiki domenowej, modelu czy innych warstw (Rysunek 1). Dzięki temu możliwe będzie skuteczne testowanie jednostkowe kodu bez konieczności zaślepiania zależności Androidowych.

Aktywacja komponentów



Rysunek 2. Odwoływanie się do komponentów z innych aplikacji

Jak przedstawiono na Rysunku 2, we wszystkich interakcjach pomiędzy komponentami pośredniczy system. Daje to niesamowite możliwości, ponieważ dzięki temu zaciera się granica aplikacji. Chcąc pobrać kontakty poprzez system, odwołujemy się do ContentProvidera z aplikacji kontakty. Kiedy potrzebujemy avataru użytkownika, często nie obchodzi nas, skąd będzie on pochodził – z naszego Activity, gdzie użytkownik go narysuje, z aparatu, czy też z galerii telefonu. Każdą interakcję (z Activity, Servicem i BroadcastReceiverem), którą chcemy wykonać, zgłaszamy do systemu jako intencję – prośbę, którą system próbuje wykonać. Bytem, który definiuje prośbę, jest obiekt klasy **Intent**. Na jej podstawie system szuka komponentu, który będzie w stanie obsłużyć nasze żądanie, i aktywuje go. Intent może zawierać argumenty dla komponentu, takie jak URI zasobu do wyświetlenia czy jakiekolwiek inne prymitywne dane. W zależności od tego, jaki komponent chcemy uruchomić, wywołujemy jedną z poniższych metod:

Listing 1: Przykłady wysyłania intent

```
context.startActivity(intent);
context.startService(intent);
context.sendBroadcast(intent);
```

W związku z tym, że aktywowaniem komponentów aplikacji zajmuje się system, każda implementacja komponentu musi zostać uwzględniona w manifeście aplikacji (AndroidManifest.xml) tak, aby system wiedział o jej istnieniu.

System nie daje absolutnie żadnych ograniczeń co do komponentów, z jakich składać ma się aplikacja. Dzięki temu możemy np. stworzyć aplikację bez interfejsu użytkownika (bez żadnego Activity), która udostępnia API innym aplikacjom poprzez wystawione serwisy. Punkt wejścia do aplikacji też jest umowny – może być to ikona (lub wiele ikon) w launcherze systemowym, ale też zdarzenie systemowe odebrane przez BroadcastReceiver, który może wygenerować powiadomienie i pokazać interfejs użytkownika.

Gdy chcemy, aby komponent mógł być aktywowany na żądanie, możemy zdefiniować mu filtr intencji (**IntentFilter**). Kiedy dany intent jest przez niego akceptowany – komponent może być aktywowany przez system. Poniżej znajduje się przykład kodu manifestu, który deklaruje dwa różne Activity jako punkty wejścia do aplikacji. Dzięki odpowiedniemu filtrowi intencji znajdą się one w launcherze systemowym.

Odpowiedni komponent jest wyszukiwany przez system na podstawie minimalnie dwóch elementów zawieranych przez intencję: akcji i kategorii. Launcher wyświetli wszystkie Activity w systemie, które pasują do akcji **`action.MAIN`** i kategorii **`category.LAUNCHER`**.

Listing 2: Filtry intencji

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
   ...
   <application ...>
      <activity
         android:name="pl.example.SocialPostsActivity"
         android:label="Social Posts">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.
            LAUNCHER" />
         </intent-filter>
      </activity>
      <activity
         android:name="pl.example.SocialMessangerActivity"
         android:label="Social Messanger">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.
            LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

Filtry intencji mogą zawierać zaawansowane kryteria, takie jak fragmenty adresów URI i typy MIME. Dzięki temu w łatwy sposób możemy stworzyć aplikację np. dla serwisu internetowego, która będzie uruchamiać się po kliknięciu linka z oczekiwanym hostem. Poniżej przykład filtra intencji, który przechwytuje wszystkie żądania uruchomienia strony http://example.com/.

Listing 3: Filtr intencji uruchamiający przeglądarkę

```
<intent-filter>
   <action android:name="android.intent.action.VIEW" />
   <category android:name="android.intent.category.DEFAULT" />
   <category android:name="android.intent.category.BROWSABLE" />
   <data android:scheme="http" />
   <data android:host="example.com" />
</intent-filter>
```

MUR CHIŃSKI

Jedynym sposobem na przekazanie danych (argumentów) do komponentu jest umieszczenie ich w Intencie. Niestety, przekazując argumenty, jesteśmy ograniczeni tylko do typów prymitywnych lub obiektów **Serializable** i **Parcelable** (szybka serializacja Androidowa). Jest to problem nie do obejścia.

Aby Intent mógł dotrzeć do systemu, a potem do kolejnego komponentu, musi czasem przekroczyć barierę proces-proces. Jest to swoisty Mur Chiński, gdyż pamięć pomiędzy procesami nie jest współdzielona. Intent musi zostać zserializowany, a następnie zdeserializowany po drugiej stronie. Wysyłając intencję pokazania zawartości strony pod danym adresem URL, zserializowany adres musi przejść pomiędzy procesami z aplikacji do przeglądarki internetowej.

Porządek w kodzie

Stworzenie Intentu wymaga podania akcji (do której dopasowywany jest IntentFilter) lub bezpośrednio komponentu w postaci nazwy klasy. Intent zawiera słownik argumentów, w którym kluczem każdego argumentu jest String. Komponent, który dostaje Intent, wypakowuje z niego argumenty. Aby nie dzielić odpowiedzialności pakowania i rozpakowywania danych pomiędzy różne klasy i nie zaśmiecać przestrzeni nazw stałymi (kluczami argumentów), dobrą praktyką jest stworzenie statycznej metody tworzącej Intent ze spakowanymi danymi lub nawet aktywującej dany komponent. Przykład takiego rozwiązania znajduje się we fragmencie kodu poniżej.

Listing 4: Tworzenie intent ze spakowanymi danymi

```
public class ArticleDetailsActivity extends Activity {
   private static final String ARG_ARTICLE_ID = "ARG_ARTICLE_ID";

   public static void startForArticle(Context ctx, long articleId) {
      Intent intent = new Intent(ctx, ArticleDetailsActivity.class)
         .putExtra(ARG_ARTICLE_ID, articleId);
      ctx.startActivity(intent);
   }

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      long articleId = getIntent().getLongExtra(ARG_ARTICLE_ID);
      showArticleContent(articleId);
   }
}
```

Cała odpowiedzialność pakowania i rozpakowywania argumentów należy do **`ArticleDetailsActivity`**. Wyświetlenie takiego Activity jest bardzo czytelne i sprowadza się do wywołania metody **`startForArticle`**.

Listing 5: Wyświetlenie activity

```
public class ArticleListActivity extends Activity {
   //...
   void onAricleSelected(long articleId) {
      ArticleDetailsActivity.startForArticle(this, articleId);
   }
}
```

ULOTNY STAN

Z punktu widzenia użytkownika Android jest systemem jedynym w swoim rodzaju, ponieważ trzyma w tle wszystkie aplikacje, które użytkownik uruchomił od startu urządzenia. Oczywiście jest to iluzja, gdyż pamięć urządzenia jest skończona, a to programista musi zadbać o to, aby użytkownik miał takie wrażenie.

Praktycznie każdy z komponentów, który implementujemy, może być zniszczony przez system. Activity, które jest długo nieużywane w tle, może zostać w dowolnym momencie zniszczone przez system razem z procesem naszej aplikacji. Pamięć jest wówczas dealokowana, a gdy użytkownik wraca do aplikacji, jest ona startowana od początku. Podobnie usługa działająca w tle może w każdym momencie zostać zabita przez system – praktycznie nad niczym nie mamy kontroli.

Rzeczą karygodną w zachowaniu aplikacji jest gubienie danych użytkownika, na przykład w połowie napisanego maila, gdy użytkownik otrzyma rozmowę telefoniczną. Android umożliwia zapisanie danych, które zostaną przechowane po zabiciu aplikacji. Służy do tego metoda `onSaveInstanceState(Bundle state)`, którą można nadpisać, implementując Activity. Pojawia się jednak kolejny problem. Podobnie jak Intent, obiekt typu Bundle, do którego zapisujemy stan, może zawierać tylko obiekty typów prymitywnych i serializowalnych. Dzieje się tak dlatego, że stan aplikacji również musi przekroczyć Mur Chiński (barierę proces-proces), gdyż przechowywany jest w pamięci procesu systemowego. Po zapisaniu stanu aplikacja może zostać bezpiecznie zabita, a pamięć całego procesu zdealokowana. Po ponownym uruchomieniu paczka ze stanem przekracza barierę system-aplikacja i możemy zapisany stan odtworzyć.

Pierwszą rzeczą, która przychodzi do głowy, jeśli chodzi o obejście mechanizmu zapisywania stanu Activity, jest postawienie Singletona, który będzie zawierał stan. Niestety jest to daremny wysiłek, ponieważ gdy aplikacja zostanie zabita, wszystkie dane przepadną. Serializacja danych odbywa się leniwie tylko wtedy, gdy jest ona potrzebna (przed zabiciem procesu), dlatego aby przetestować zapisywanie stanu każdego wysłanego w tło Activity, należy uruchomić opcję „Nie zachowuj działań" w ustawieniach developerskich na urządzeniu.

Zabijanie procesów

Z punktu widzenia użytkownika urządzenia z systemem Android aplikacje typu **task killer** nie mają zbyt dużego sensu, gdyż system automatycznie sprząta dawno nieużywane aplikacje z pamięci. Można tym wręcz zaszkodzić i stracić wartościowe dane aplikacji, gdyż wtedy mechanizm zapisywania stanu nie jest wykonywany podczas zabijania procesu.

Autor

Serializable vs. Parclable

Trzeba nadmienić, iż standardowa Javowa serializacja obiektów (poprzez implementację interfejsu Serializable) działa na Androidzie nawet kilkunastokrotnie wolniej, niż serializacja przez implementację interfejsu Parcelable. Dzieje się tak z powodu nieoptymalnie zaimplementowanego mechanizmu refleksji w Dalviku – wirtualnej maszynie Javy na Androidzie. Przy małej ilości danych nie stanowi to problemu, jednak przy większej będzie to znacząco opóźniało start aplikacji. Niestety implementacja obiektów Parcelable wymaga napisania sporej ilości dosyć bezsensownego kodu, ale na szczęście istnieją wtyczki (do Android Studio i IntelliJ IDEA), które umożliwiają wygenerowanie kodu parcelacji.

I JESZCZE TEN OBRÓT EKRANU

Jednym z pierwszych dużych problemów, które napotykamy, programując aplikacje na Androida, jest gubienie stanu Activity przy obrocie ekranu. Android dostarcza developerom znakomity system zarządzania zasobami w zależności od konfiguracji urządzenia. Dzięki temu systemowi możemy transparentnie wspierać urządzenia o różnych gęstościach, rozdzielczościach i rozmiarach ekranu. Do konfiguracji należy między innymi również informacja o wybranym języku i o orientacji ekranu.

Użytkownik zmieniając język systemu, oczekuje, że język zmieni się we wszystkich aplikacjach, które były uruchomione. Aby tego dokonać, Android niszczy wszystkie Activity, zapisując ich stan, i odtwarza je, ładując nowe zasoby. Dokładnie ten sam mechanizm tyczy się zmiany orientacji ekranu. Na jej podstawie załadowane mogą zostać całkowicie inne zasoby niż do tej pory.

Gdy rzetelnie zapisujemy i odtwarzamy stan Activity (co i tak powinniśmy robić ze względu na to, że aplikacja może zostać zabita przez system), zapisywanie stanu przy zmianie konfiguracji będzie działało, ponieważ używa dokładnie tego samego mechanizmu. Nie należy oszukiwać, dodając magiczne flagi do manifestu, chyba że dokładnie wiemy, co robimy.

UŻYTKOWNIK JEST NAJWAŻNIEJSZY

Na pierwszy rzut oka widać, że Android został zaprojektowany bardziej z myślą o użytkowniku niż o programiście, co jest bardzo słusznym podejściem.

Aplikacja w typowym tego słowa znaczeniu nie istnieje (threre is no app). Zbiór komponentów, który instalujemy na urządzeniu, staje się pluginem, integralną częścią systemu. Android ma całkowitą kontrolę nad każdym aspektem jego działania – od aktywowania poszczególnych komponentów przez ich zabijanie i zapisywanie stanu. Większość fragmentów niewygodnego API wynika właśnie z komunikacji międzyprocesowej, która musi być wewnętrznie używana, aby mechanizm mieszania aplikacji mógł działać. Należy pamiętać, że aplikacje piszemy dla użytkownika i aby zapewnić mu najlepsze możliwe doświadczenia z jej używania, nie należy walczyć z mechanizmami platformy.

Michał Charmas michal@charmas.pl

Trener w firmie Bottega IT Solutions i programista zajmujący się aplikacjami mobilnym na platformie Android. Pasjonat tego systemu praktycznie od momentu jego powstania. Szczególny nacisk kładzie na jakość tworzonych aplikacji zarówno pod względem architektury oraz czystości kodu, jak i designu. Do jego pozostałych zainteresowań zawodowych należą Python, JS oraz Scala. Prywatnie porwany przez muzykę.