Moduły w Javie 9

Java Platform Module System, znany wcześniej jako Project Jigsaw, znacząco wpłynął na organizację maszyny wirtualnej, zaś wdrożenie w bibliotekach i aplikacjach staje się faktem. W tym artykule opisuję, jakie problemy stały przed systemem JPMS i jak je rozwiązano, oddając w nasze ręce kompletny system modularnego środowiska Javy.

PIEKŁO WYBRUKOWANE KLASAMI

Dotychczas, by zlokalizować definicję klasy, Java używała *class path*, tj. podanej (w manifeście lub poprzez argument) listy ścieżek przeszukiwania. Każda z klas ładowanych do JVM musiała znajdować się w pliku dostępnym z tej listy. Zbiór wszystkich dostępnych tą drogą klas spłaszczany jest do sekwencji, z której – w kolejności wystąpienia – wybierana jest żądana nazwą definicja. Oczywiście, mówimy tu o domyślnym *classloaderze* aplikacji, zaś ambitny programista mógłby napisać własną implementację dobierającą klasy z większą precyzją. Koniec końców gubimy dodatkową charakterystykę, potencjalnie szerszą niż grupowanie pakietami.

JVM ładuje klasy na żądanie. Jeśli żaden *classloader* nie znajdzie klasy, maszyna rzuci wyjątkiem `NoClassDefFoundError`. Kompletność środowiska uruchomieniowego w tym sensie nie jest i nie może zostać zweryfikowana, przez co – w aplikacjach o złożonych zależnościach – trudno przewidzieć scenariusze prowadzące do katastrofy. W myśl praw Murphy'ego taki błąd objawi się podczas ważnej prezentacji, w ramach przetwarzania krytycznego *payloadu*, albo wyklika go strategiczny klient w zaciszu swojego biura piątkowego wieczoru.

Kolejnym problemem jest sytuacja, w której istnieją pod tą samą kwalifikowaną nazwą różne klasy, niekoniecznie ze sobą powiązane. Zawsze zostanie wybrana „pierwsza z brzegu" implementacja. Jeśli to nie ta, której oczekujemy, uruchomienie może się skończyć serią błędów zaciemniających źródło problemu.

Wszystko to definiuje stan zwany *classpath hell* i jest błędem konfiguracji. Odpowiedzią Javy 9 na ten problem jest spójny mechanizm zależności – system modułów, których powiązania i interfejs są jawnie zadeklarowane w deskryptorze modułu. Teraz JVM już podczas startu weryfikuje środowisko, zapewniając jego spójność, zaś w przypadku niespełnionej zależności zgłasza wyjątek `java.lang.module.FindException` zawierający wszystkie informacje niezbędne do skorygowania deskryptora.

ŚCISŁA HERMETYZACJA

Hermetyzacja przed Javą 9 sprowadzała się do kombinacji pakietów i modyfikatorów dostępu (*private*, *protected*, *package-private*, *public*). Praktyka wykazała, że istnieje model współdzielenia typów, w którym to nie wystarcza. Zwłaszcza w obliczu technik refleksji.

Dotychczasowy mechanizm nie pozwalał na ukrycie klasy przed światem i jednoczesne udostępnienie tej samej klasy do użytku we wszystkich pakietach biblioteki. Takie połączenie *protected* dla kodu z zewnątrz i *public* dla kodu z wewnątrz biblioteki. Między innymi dlatego, że nie było mechanizmu pozwalającego jasno określić granice tejże biblioteki. Pamiętajmy przy tym, że to, co nazywamy tutaj biblioteką w sensie logicznym, może składać się z wielu plików JAR.

Wszystkie pakiety *impl* czy *internal*, jakie spotykamy, są emanacją słabości hermetyzacji przed JPMS. Projektant biblioteki daje nam jasno do zrozumienia, że choć te klasy są publiczne, to nie należy ich używać, chociażby dlatego, że rości sobie prawo do ignorowania kompatybilności pomiędzy wersjami lub żeby ograniczyć możliwość przypadkowej ingerencji w danych.

Całą platformę podzielono na około 100 modułów, dzięki czemu wraz z aplikacją możemy dostarczyć w pełni sprawną maszynę wirtualną skrojoną na miarę, co jest istotne zwłaszcza na urządzeniach o mocno ograniczonych zasobach. Pełna modularyzacja JVM i aplikacji wpływa pozytywnie na jej wydajność, ponieważ graf zależności jasno precyzuje, które komponenty wchodzą w skład dystrybucji i warte są dalszego przetwarzania.

W celu dostosowania JVM, Java 9 wprowadza proces zwany linkowaniem i zupełnie nowe narzędzie: `jlink`. Znajdziemy je w katalogu *bin* instalacji JDK. Dzięki temu poleceniu przygotujemy obraz maszyny wirtualnej zintegrowany z wymaganymi przez aplikację modułami, a ponadto narzędzie wygeneruje skrypty uruchomieniowe dla różnych systemów. Linkowanie jest opcjonalnym procesem zaliczanym do zaawansowanych sztuczek, dlatego wykracza poza ramy artykułu. Polecam lekturę dokumentacji oraz publikacji z odnośników na końcu artykułu.

DESKRYPTOR MODUŁU

Długo wyczekiwanym rozwiązaniem wymienionych problemów jest Java Platform Module System, czyli system modułów. Moduł grupuje logicznie powiązane pakiety typów i ewentualne zasoby, takie jak obrazy czy pliki XML. Pakowany jest w dobrze znany format JAR, ale odróżnia go obecność w katalogu głównym archiwum skompilowanego pliku *module-info.java* czyli deskryptora modułu. Plik ten określa charakterystykę modułu: jego unikalną nazwę, zależności od innych modułów oraz jasno sprecyzowane API udostępniane przez moduł.

Ponieważ w przestrzeni nazw modułów aplikacji pod jedną nazwą może występować jeden moduł, nazwa ta musi być unikalna. Ponadto jeśli projektowany moduł będzie dostępny szerzej, np. jako fragment biblioteki, nazwa powinna być unikalna globalnie. Przyjętą praktyką jest konwencja nazewnicza *reverse DNS*, którą dobrze znamy choćby z nazewnictwa pakietów, np. `org.apache.logging`. Przestrzeń nazw modułów jest niezależna od pozostałych przestrzeni nazw, więc technicznie rzecz ujmując, moduł może nosić tę samą nazwę, co dostarczana przez niego klasa czy pakiet. Zdecydowanie lepiej jednak zachować globalną unikalność między tymi przestrzeniami, aby uniknąć wieloznaczności i pomyłek.

Przyjrzyjmy się deskryptorowi jednego z modułów platformy – modułu `java.prefs` dostarczającego API do obsługi prostej konfiguracji. Plik deskryptora wygląda tak: