# Architektura modularna

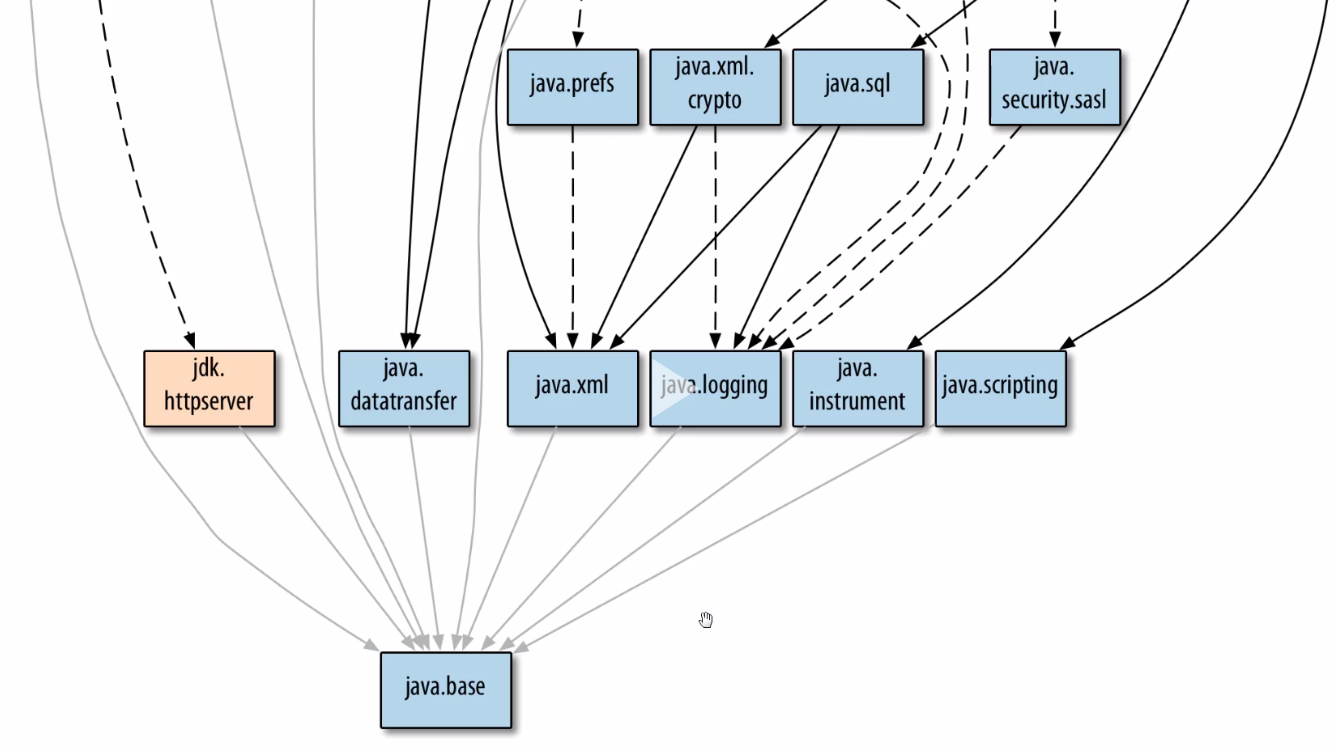
Od Javy 1.9 – natywne wsparcie dla modułów.

Złożony program można dzielić do plików jar albo do mikroserwisów.

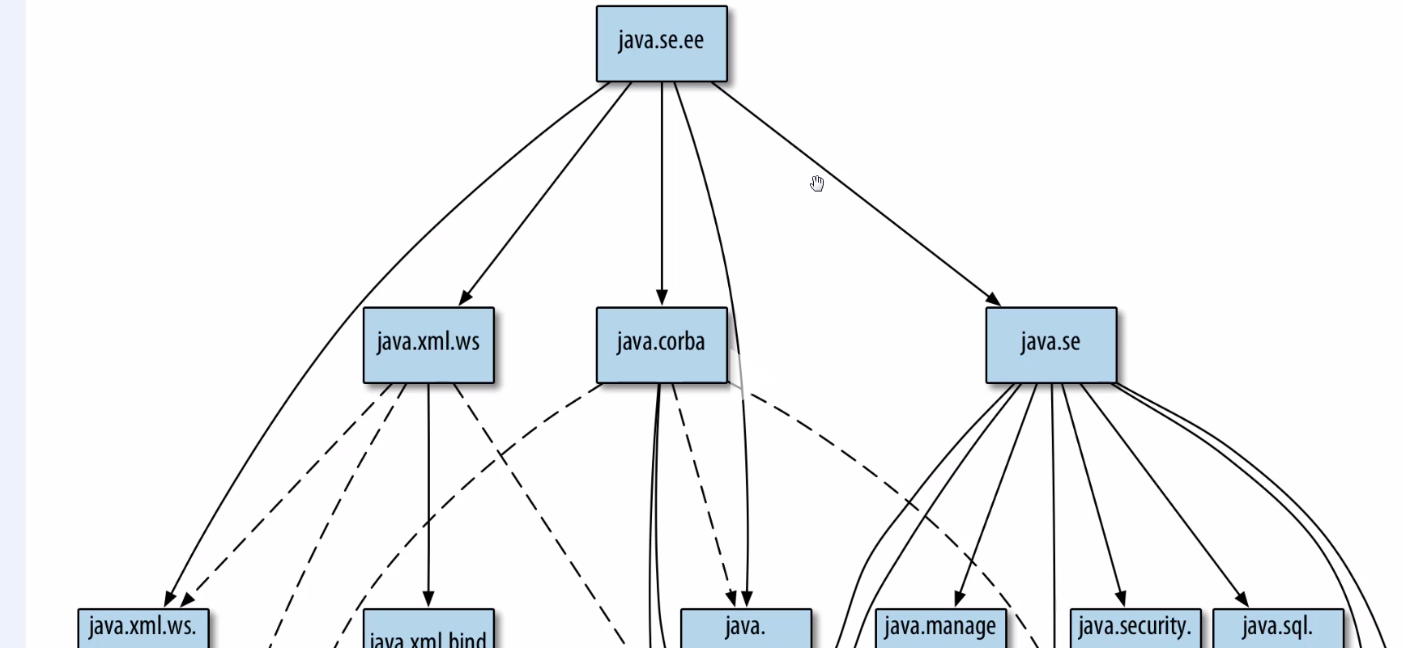
Mikroserwisy – kawałek wydzielonej funkcjonalności z naszego kodu. Zazwyczaj działa on niezależnie na jakimś komputerze w sieci i w razie potrzeby łączy się z główną aplikacją lub innymi mikroserwisami.

Aplikacja może być więc rozbita na kilka modułów, które wciąż są pakowane do jednego jara. Każdy z modułów ma określone rzeczy, które wystawia na zewnątrz i określone rzeczy, które potrzebuje od innych modułów.

# Modułowe SDK



Java.base to moduł po którym dziedziczą wszystkie moduły. (jest ich około 90)



# Pierwszy moduł

Każdy moduł musi posiadać plik **module-info.java**. Jest to plik, który zawiera info na temat modułu: jego nazwę, zależności, to co wystawia na zewnątrz, jego serwisy itp.

Nazwy modułów podobnie jak nazwy pakietów, muszą być unikalne.

1. W src tworzymy module-info, obowiązkowo pakiet, a w nim klasę.

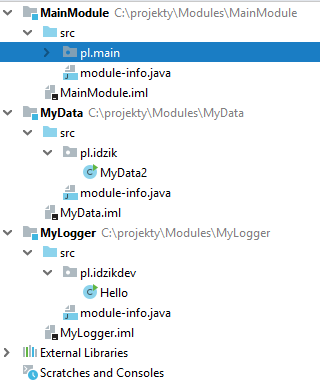
Tutaj jest lepiej opisane, bo na tym szkoleniu nie działa poprawnie.

<https://youtu.be/ZogBPXNy3uU>

Najlepiej jest robić moduły najpierw jako moduły IDEA (project settings->module->add)

a potem dodać do katalogu plik module-info.java, a zależność ustalić w tym samym oknie (..->module->dependencies) albo później w kodzie alt-enter

# Ustalanie zależności między modułami



**module** pl.main {  
 **requires** pl.idzik;  
 **requires** pl.idzikdev;  
}

Ponieważ jest exports

**module** pl.idzikdev {  
 **exports** pl.idzikdev;  
}

to możemy w klasie w module pl.main skorzystać z klasy Hello z modułu pl.idzikdev

**package** pl.main;  
  
**import** pl.idzikdev.Hello;  
  
**public class** MainModule {  
 Hello **hello**=**new** Hello();  
}

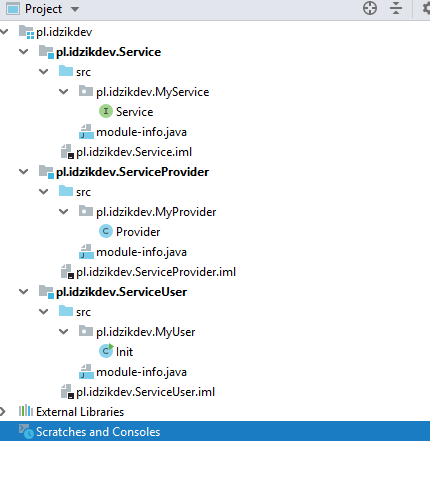
# Widoczność typów między modułami

Jeśli chcemy aby podrzędny pakiet był widoczny w innym module, to musimy napisać dla niego oddzielną dyrektywę exports.

# Serwisy

Interfejs może deklarować jakiś serwis czyli dostarczać interfejs. Inne moduły mogą być providerami dla tego serwisu czyli dostarczać jego implementacje. Inne moduły mogą być klientami tego serwisu czyli korzystać z niego.

Utworzymy teraz moduł **Service**, dostarczyciela klas tego service przez moduł **ServiceProvider**, a używać tego będzie moduł **ServiceUser.**



**SERWIS**

**package** pl.idzikdev.MyService;  
  
**public interface** Service {  
 String getString();  
}

**module** pl.idzikdev.Service {  
 **exports** pl.idzikdev.MyService;  
}

**PROVIDER**

**package** pl.idzikdev.MyProvider;  
  
**import** pl.idzikdev.MyService.Service;  
  
**public class** Provider **implements** Service {  
 @Override  
 **public** String getString() {  
 **return "Hello"**;  
 }  
}

**module** pl.idzikdev.ServiceProvider {  
 **requires** pl.idzikdev.Service;  
 **provides** pl.idzikdev.MyService.Service **with** pl.idzikdev.MyProvider.Provider;  
}

**USER**

**package** pl.idzikdev.MyUser;  
  
**import** pl.idzikdev.MyService.Service;  
  
**import** java.util.ServiceLoader;  
  
**public class** Init {  
 **public static void** main(String[] args) {  
 ServiceLoader<Service> services=ServiceLoader.*load*(Service.**class**);  
 services.findFirst().ifPresent(s-> System.***out***.println(s.getString()));  
 }  
}

**module** pl.idzikdev.ServiceUser {  
 **requires** pl.idzikdev.Service;  
 **requires** pl.idzikdev.ServiceProvider;  
 **uses** pl.idzikdev.MyService.Service;  
}

JESZCZE RAZ MODUŁY razem

**module** pl.idzikdev.**Service** {  
 **exports** pl.idzikdev.MyService;  
}

**module** pl.idzikdev.**ServiceProvider** {  
 **requires** pl.idzikdev.Service;  
 **provides** pl.idzikdev.MyService.Service **with** pl.idzikdev.MyProvider.Provider;  
}

**module** pl.idzikdev.**ServiceUser** {  
 **requires** pl.idzikdev.Service;  
 **requires** pl.idzikdev.ServiceProvider;  
 **uses** pl.idzikdev.MyService.Service;  
}

Serwisy – zastosowanie Factory Pattern

Poprzednie podejście nie jest zbyt efektywne. Trzeba dużo wiedzieć: np. który serwis czego używa, który co implementuje. Ciężko wymagać takiej wiedzy od klienta.

Zastosujemy nowe podejście do interfejsów. Zadeklarujemy w serwisie zmienną statyczną i pobierzemy jego implementację. Zrobimy tak, aby nasz interfejs (nasz serwis) sam z siebie korzystał.

Zabieramy **uses** pl.idzikdev.MyService.Service;

Wstawiamy do Service

Po zmianach

**public interface Service** {  
 String getString();  
 **static** Service getInstance(){  
 ServiceLoader<Service> services=ServiceLoader.*load*(Service.**class**);  
 Optional<Service> first=services.findFirst();  
 **return** first.orElseThrow(()->**new** RuntimeException(**"ERROR"**));  
 }  
}

**module** pl.idzikdev.Service {  
 **exports** pl.idzikdev.MyService;  
 **uses** pl.idzikdev.MyService.Service;  
}

**module** pl.idzikdev.ServiceProvider {  
 **requires** pl.idzikdev.Service;  
 **provides** pl.idzikdev.MyService.Service **with** pl.idzikdev.MyProvider.Provider;  
}

**public static void** main(String[] args) {  
 Service instance=Service.*getInstance*();  
 System.***out***.println(instance.getString());  
}

**module** pl.idzikdev.ServiceUser {  
 **requires** pl.idzikdev.Service;  
}

czyli w zasadzie ServiceProvider nie jest nam już potrzebny.

# Budowa modułów z użyciem mavena

Przydałoby się teraz uniezależnić te moduły od IDE i znaleźć sposób na opisanie zależności np. z Hibernate czy innymi.