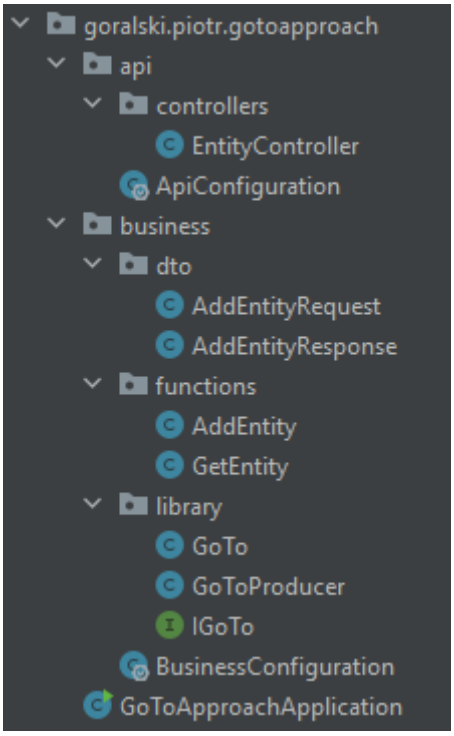
GoTo

Główną koncepcją programowania GoTo jest pozbycie się wstrzykiwania zależności. Korzystając z przedstawionej w tym artykule architektury nie jesteśmy ograniczeni kontekstami aplikacji i możemy je dowolnie definiować.

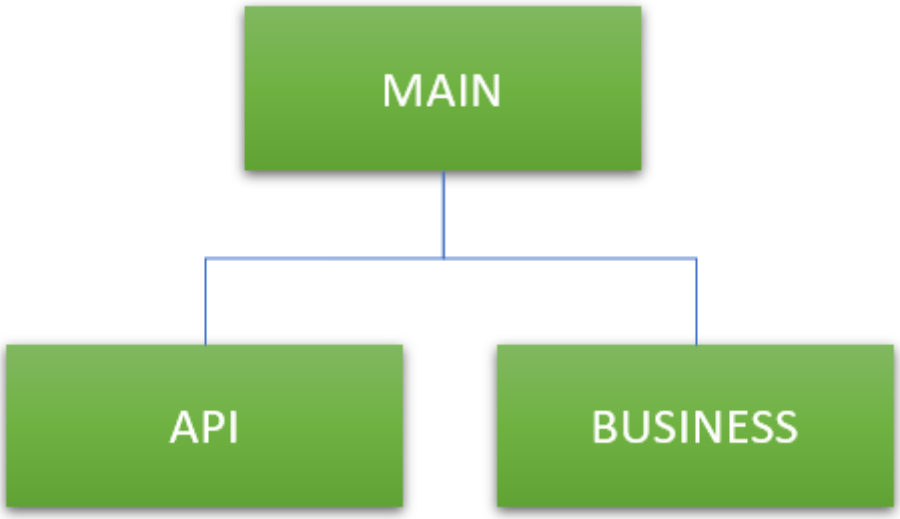
Przykład zastosowania

Końcowa struktura projektu będzie prezentować się w następujący sposób.



Aplikacja będzie składać się z 3 kontekstów:

- Głównego kontekstu aplikacji
- Kontekstu API
- Kontekstu BUSINESS



Definiowanie kontekstu odbywa się następująco

```
@SpringBootConfiguration
@PropertySource("classpath:goToApproachApplication.properties")
public class GoToApproachApplication {

    public static void main(String[] args) {
        new SpringApplicationBuilder()
                .sources(GoToApproachApplication.class).web(WebApplicationType.NONE)
                .child(ApiConfiguration.class).web(WebApplicationType.SERVLET)
                .sibling(BusinessConfiguration.class).web(WebApplicationType.NONE)
                .run(args);
    }

}
```

Kontekst API będzie zawierał kontroler obsługujący przykładową encję. Kontroler nie będzie posiadał żadnych powiązanych komponentów co czyni implementacje prostszą. Implementacja kontrolera prezentuje się następująco.

```java
@RestController
@RequestMapping
public class EntityController {

    @GetMapping("/entity/{id}")
    public String getEntity(@PathVariable String id) {
        return new GoToProducer<>(GetEntity.class).execute(id);
    }

    @PostMapping("/entity")
    public AddEntityResponse createEntity(@RequestBody AddEntityRequest request) {
        return new GoToProducer<>(AddEntity.class).execute(request);
    }

}
```

By uruchomić daną akcję korzystamy z przygotowanej klasy GoToProducer. Klasa korzysta z generycznych typów by uniemożliwić pomyłkę przy uruchamianiu konkretnej akcji. Metoda execute jest odpowiedzialna za stworzenie obiektu odpowiedniej klasy, wykonania przypisanej do niej akcji oraz zwrócenie wyniku. Dzięki klasie GoToProducer programista nie musi za każdym razem obsługiwać całego procesu pobierania klasy, uruchamiania odpowiedniej metody oraz zwracania wyniku. Klasa GoToProducer posiada poniższą implementację.

```java
public class GoToProducer<T extends GoTo<R> & IGoTo<V, R>, V, R>{

    2 usages
    private final Class<T> function;

    2 usages
    public GoToProducer(Class<T> function) { this.function = function; }

    2 usages
    public R execute(V data) {
        try {
            Constructor<T> constructor = function.getConstructor(data.getClass());
            GoTo<R> object = constructor.newInstance(data);

            return object.getResult();
        } catch (Exception e) {
            throw new RuntimeException(e.getCause());
        }
    }
}
```

W rezultacie wykonania powyższego kodu zostanie wykonana akcja GetEntity dla pierwszej metody kontrolera, bądź akcja AddEntity dla drugiej metody kontrolera. Jako parametr, metoda execute przyjmuje dane wejściowe do akcji, które będą dostępne podczas przetwarzania danego procesu. Akcje są zdefiniowane w kontekście BUSINESS, w innym niż miejsce z którego wywołujemy akcje.

Implementacja akcji AddEntity została przedstawiona poniżej

```
public class AddEntity extends GoTo<AddEntityResponse> implements IGoTo<AddEntityRequest, AddEntityResponse> {

    public AddEntity(AddEntityRequest data) { super.proceed(task(data)); }

    2 usages
    @Override
    public AddEntityResponse task(AddEntityRequest data) {
        //PROCEED LOGIC
        System.out.println("AddEntity - proceeding - " + data.getRequest());

        return new AddEntityResponse("AddEntity result");
    }

}
```

Akcja musi dziedziczyć po klasie GoTo oraz implementować interfejs IGoTo który przyjmuje 2 generyczne typy. Pierwszym typem jest klasa której obiekt przedstawia dane przekazane do akcji. Obiekt danej klasy jest przyjmowany przez konstruktor i metodę task która odpowiada za logikę danej akcji. Drugim typem jest klasa której obiekt zostanie zwrócony z danej akcji. W konstruktorze wywołujemy metodę proceed z nadklasy GoTo która przyjmuje parametr który jest wynikiem działania metody task odpowiadającej za logikę klasy. Implementacja klasy GoTo prezentuje się następująco.

```
public class GoTo<T> {

    2 usages
    private T result;

    2 usages
    public void proceed(T result) { setResult(result); }

    1 usage
    public T getResult() { return result; }

    1 usage
    public void setResult(T result) { this.result = result; }

}
```

Klasa GoTo przyjmuje generyczny typ który będzie określał typ atrybutu result. Atrybut result jest zwracany przez klasę GoToProducer do pierwotnego miejsca wywołania metody.

Interfejs IGoTo zawiera tylko 1 metodę task, dzięki której zmuszamy programistę do dostarczenia logiki klasy oraz zapewniamy poprawne mapowanie typów. Implementacja interfejsu IGoTo prezentuje się następująco.

```java
public interface IGoTo<T, V> {

    2 usages  2 implementations
    V task(T request);

}
```

Klasy AddEntityRequest oraz AddEntityResponse są klasami POJO i nie wpływają na przedstawione rozwiązanie. Ukazują one wyłącznie możliwość operowania na własnych klasach i pokazują stabilność typologiczną rozwiązania. W akcjach można korzystać z typów prostych oraz dowolnych typów złożonych. Implementacja akcji GetEntity pokazuje że w prosty sposób możemy zdefiniować akcję dla typu String który jest przekazywany jako parametr oraz jest zwracany jako rezultat operacji.

```java
public class GetEntity extends GoTo<String> implements IGoTo<String, String> {

    public GetEntity(String data) { super.proceed(task(data)); }

    2 usages
    @Override
    public String task(String data) {
        //PROCEED LOGIC
        System.out.println("GetEntity - proceeding - " + data);

        return "GetEntity result";
    }

}
```

Aplikacja pokazuje w jak łatwy sposób możemy wywoływać poszczególne akcje oraz jak prosto jest definiować nowe akcje. Tworzone akcje mogą znajdować się w dowolnym kontekście aplikacji ponieważ klasa GoToProducer korzysta z mechanizmu refleksji dzięki czemu omija bariery standardowych rozwiązań. Mimo zastosowania mechanizmu refleksji rozwiązanie jest stabilne i odporne na błędy. Nie można wywołać akcji z błędnymi parametrami, a typ zwracanego wyniku jest znany dzięki czemu unikamy konieczności rzutowania typów.

Próba wykonania akcji z parametrem błędnej klasy skutkuje błędem składniowym i nie pozwala na kompilację rozwiązania.

Uważam że przedstawione podejście ma liczne zalety oraz może być stosowane w projektach dowolnego typu oraz wielkości.

Krótka analiza innych rozwiązań:

Zamiast powyższej implementacji moglibyśmy stworzyć klasę serwisu z metodami getEntity oraz addEntity, lecz klasa musiała by się znajdować w tym samym kontekście aplikacji. W przypadku podejścia programowania zdarzeniami i wykorzystania klasy Spring Event nie jesteśmy w stanie zwrócić wyniku z wydarzenia oraz w dalszym ciągu jesteśmy ograniczeni kontekstem aplikacji. Więc podejście programowania zdarzeniami wydaje się najgorsze z możliwych.