Integration Patterns

Wykonali:

- Filip Glinkowski
- Artur Gurak
- Jakub Jarecki
- Bartosz Knapik
- Tomasz Kot
- Emilia Kwolek

Integration Tier
Warstwy w Java EE

- Podział na warstwy w Java EE
  - Warstwa prezentacji
  - Warstwa biznesowa
  - Warstwa Integracji

- Współpraca pomiędzy warstwami pozwala na tworzenie aplikacji trójwarstwowych w których każda z warstw odpowiada za zdefiniowany zakres odpowiedzialności

Integration Tier
Środowisko biznesowe a powstanie warstwy integracji

Warstwa integracji powstała w celu oddzielenia zagadnień technicznych takich jak:

- Odczyt oraz zapis danych
- Komunikacja z bazą danych oraz innymi zasobami aplikacji (wewnętrznymi oraz zewnętrznymi)
- Logika aplikacji oraz jej złożoność

od **warstwy biznesowej** aplikacji

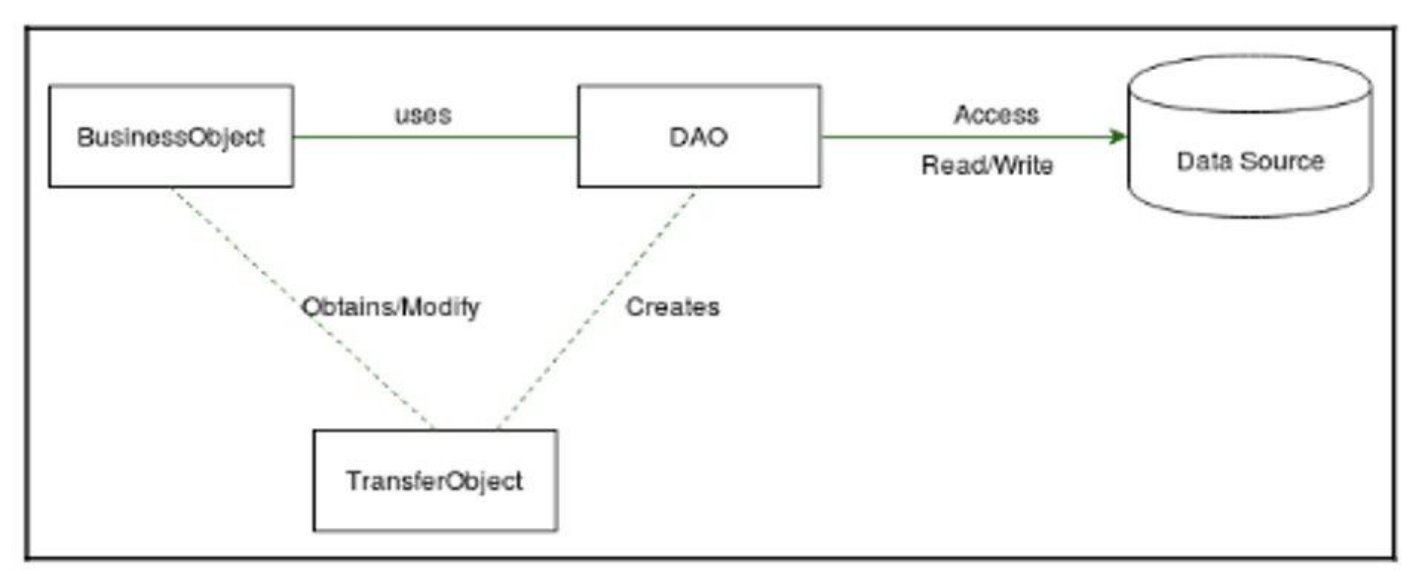
Data-access Object Pattern

W świecie biznesu, aplikacja zawsze musi być zintegrowana ze źródłem danych, aby móc odczytywać, zapisywać, usuwać i aktualizować dane. Takim źródłem danych może być na przykład relacyjna baza danych, baza danych NoSQL, LDAP (Lightweight Directory Access Protocol) lub system plików. Każdy typ źródła danych ma swoją strukturę i charakteryzuje się złożonością w zakresie łączenia się z nim, odczytywania i zapisywania danych. Operacje na danych powinny być oddzielone od logiki biznesowej.

Model wzorca DAO

Wzorzec obiektu dostępu do danych jest wzorcem używanym do ukrycia całego dostępu do źródeł danych, przed warstwą biznesową. Wzorzec ten oddziela całą logikę dostępu do danych oraz jej złożoność, od warstwy biznesowej. Jeśli będziemy chcieli zastąpić źródło danych innym, będziemy musieli jedynie zmodyfikować kod wzorca obiektu dostępu do danych. Modyfikacja ta nie będzie widoczna na poziomie biznesowym.

Schemat wzorca DAO



Implementacja DAO

Dobrą praktyką implementacji DAO, jest stworzenie abstrakcyjnego DAO (przykładowo “AbstractDao”), które będzie nadklasą wszystkich DAO, posiadającą metody z ogólną logiką, która może być używana przez wszystkie DAO.

```
import java.util.Optional;

public abstract class AbstractDao <T extends Entity>{

    //EntityManager that provide JPA functionalities
    @PersistenceContext
    protected EntityManager em;

    //Get the type of Subclass that implements Entity interface
    protected Class<T> getType() {
        ParameterizedType genericType = (ParameterizedType)
        this.getClass().getGenericSuperclass();
        return (Class<T>) genericType.getActualTypeArguments()[0];
    }

    //Find entity filtering by id.
    public Optional<T> findById ( T entity ){

        return Optional.ofNullable( em.find( (Class<T>)
        entity.getClass(), entity.getId() ) );

    }

    public Optional<T> persist (T entity ){

        em.persist( entity );
        return Optional.of( entity );
```

Implementacja DAO

Charakterystyczną cechą klas DAO jest tworzenie metod, które zwracają tylko encje lub listę encji. Jest to dobra praktyka, ponieważ znamy typ obiektu zwracanego przez tę metodę, a także porządkuje to kod, ponieważ wiemy, jaki typ wartości może zwrócić nasza metoda. W celu odczytania danych pracownika, możemy wywołać metody findAll, findByName i findById; aby zapisać dane pracownika, możemy wywołać metody persist i update; aby usunąć dane pracownika, możemy wywołać metodę delete.

```
@Stateless
public class EmployeeDao extends AbstractDao <Employee> {

    public List<Employee> findByName(String name ){

        return this.listWithNamedQuery("Employee.findByName",
                                       Collections.singletonMap(
                                       "name", name ) );

    }

    public List<Employee> findAll(){

        return this.listWithNamedQuery("Employee.findAll",
                                        Collections.emptyMap());

    }

}
```

Implementacja DAO

Należy zauważyć, że klasa biznesowa działająca na poziomie biznesowym, nie zna procesu odczytywania i zapisywania danych. Zna ona jedynie parametry metody, którą należy wywołać oraz zwracane przez nią dane. Dlatego możemy zastąpić źródło danych bez wpływu na logikę biznesową. Przykładowa klasa EmployeeBusiness wykorzystujący DAO do odczytu i zapisu danych.

```
@Stateless
public class EmployeeBusiness{

    @Inject
    protected EmployeeDao employeeDao;

public List<Employee> listByName( String name ){

    return employeeDao.findByName( name );

}

public boolean save ( Employee employee ){

    return employeeDao.persist( employee ).isPresent();

}

public List<Employee> listAll(){

    return employeeDao.findAll();

}

public Optional<Employee> findById(Employee employee ){
    return employeeDao.findById(employee);
}
```

Implementacja Encji JPA

Czym jest encja JPA

- Encja JPA jest klasą reprezentującą pewien widok lub tabele bazy danych
- Posiada ona swoje atrybuty pozwalające jednoznacznie określić daną encję
- Musi istnieć konstruktor nie posiadający żadnych argumentów
- Każdy obiekt klasy odpowiada tylko jednemu wierszowi widoku lub tabeli

Implementacja Encji JPA
Interfejs Entity

- Każda encja JPA implementowana jest w według następującej metody

```
public interface Entity < T > {

        public T getId() ;

}
```

Implementacja Encji JPA

Przykład encji JPA dla kolumny *Employee*

- Poniżej przedstawiona została encja *Employee* wraz z atrybutami oraz konstruktorami. Odzwierciedlenie kolumn tabeli odbywa się poprzez adnotację ***@Column(name=" ")***

```
public class Employee implements Entity<Long> {
        @Id
        @GeneratedValue
        @Column(name = "id")
        private Long id;
        @NotNull
        @Column(name="name")
        private String name;
        @Column(name="address")
        private String address;
        @NotNull
        @Column(name="salary")
        private Double salary;
        public Employee( ) { }
        public Employee( String name, String address, Double salary){
                this.name = name;
                this.address = address;
                this.salary = salary;
        }
```

Domain-Store Pattern

Wzorzec projektowy, który pozwala dodać funkcjonalności do DAO. Umożliwia aplikacji wybór logiki persystencji w zależności od stanu obiektów.

Persystencja - określa zapisywanie danych na stałe gdzieś na zewnątrz programu. Najprostszym przykładem persystencji są pliki.

Domain-Store Pattern | Zalety i wady

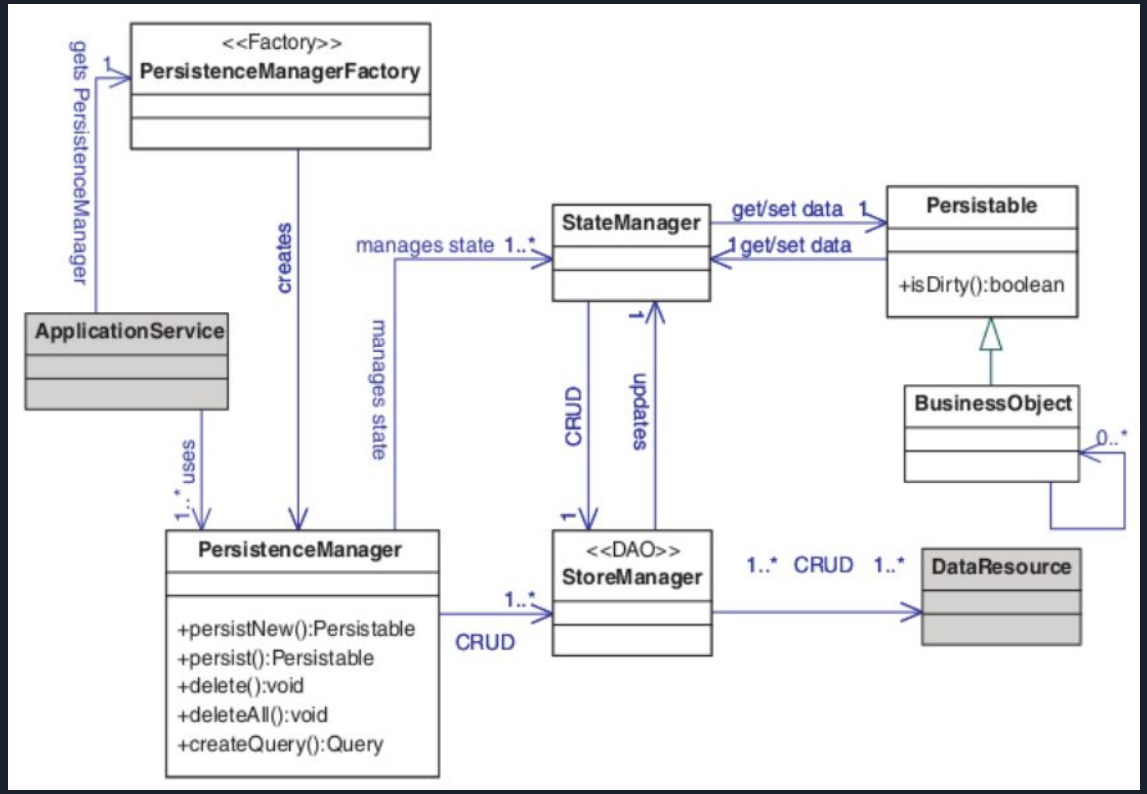
ZALETY:

- Oddziela logikę *persystencji* od modelu (logiki biznesowej)
- Umożliwia uruchamianie aplikacji w kontenerze sieciowym
- Transparentne rozwiązanie umożliwiające wybór mechanizmów persystencji w zależności od stanu obiektu

WADY:

- Złożona implementacja
- Rozbudowany model wykorzystujący dziedziczenie i złożone relacje pomiędzy jego elementami
- Częściowo wyparty przez JPA - głównie w przypadku zastosowania relacyjnej bazy jako źródła danych

Domain-Store Pattern | Diagram Encji



Domain-Store Pattern | Implementacja

`PersistenceManagerFactory` - wzorzec metody wytwórczej (factory), odpowiedzialny za tworzenie instancji klasy **`PersistenceManager`**. Jest singletonem, więc ma tylko jedną instancję w całej aplikacji. Główna funkcjonalność: implementacja metody **`getPersistenceManager()`**, która tworzy i zwraca instancję klasy **`PersistenceManager`**.

`PersistenceManager` - zarządza persystencją i zapytaniami o dane. Odpowiada za obsługę wszystkich trwałych procesów i zapytań. Posiada metody odpowiedzialne za zapis/odczyt obiektu, rozpoczęcie/zatwierdzenie/cofnięcie transakcji.

Domain-Store Pattern | Implementacja

`StoreManager` - działa jako obiekt dostępu do danych (DAO). Zwykle klasa wykorzystuje obiekty typu **data-access**, które umożliwiają przechowanie danych w odpowiednim obiekcie, zapis/usunięcie/ pobranie.

`StateManager` - jest to interfejs używany do tworzenia wszystkich implementacji stanów. Posiada obiekty klasy **`Employee`**; atrybut **`isNew`**, pozwalający na rozpoznanie czy dane są nowe; metodę **`flush()`**, która wywołuje zapis na źródle danych; metodę zwracającą obiekt klasy **`Employee`**.

Domain-Store Pattern | Implementacja

`TransactionFactory` - wzorzec metody wytwórczej do tworzenia instancji typu **Transaction**.

`Transaction` - klasa, która kontroluje cykl życia transakcji. Posiada: metodę **`init()`**, z adnotacją ***`@PostConstruct`*** (oznacza, że wywołana jest zaraz po konstruktorze), metody odpowiedzialne za rozpoczęcie/ zatwierdzenie/cofnięcie transakcji oraz metodę logiczną, która pozwala określić czy transakcja jest rozpoczęta.

Service-Activator pattern

Wzorzec odpowiadający na potrzebę zapytań asynchronicznych klienta do serwisu. Pozwala na nieblokowanie klienta, podczas oczekiwania na odpowiedź.

Przedstawione zostaną trzy rozwiązanie tego problemu:

- Java Message Service (JMS)
- metody asynchroniczne EJB
- zdarzenia asynchroniczne: producenci i konsumenci

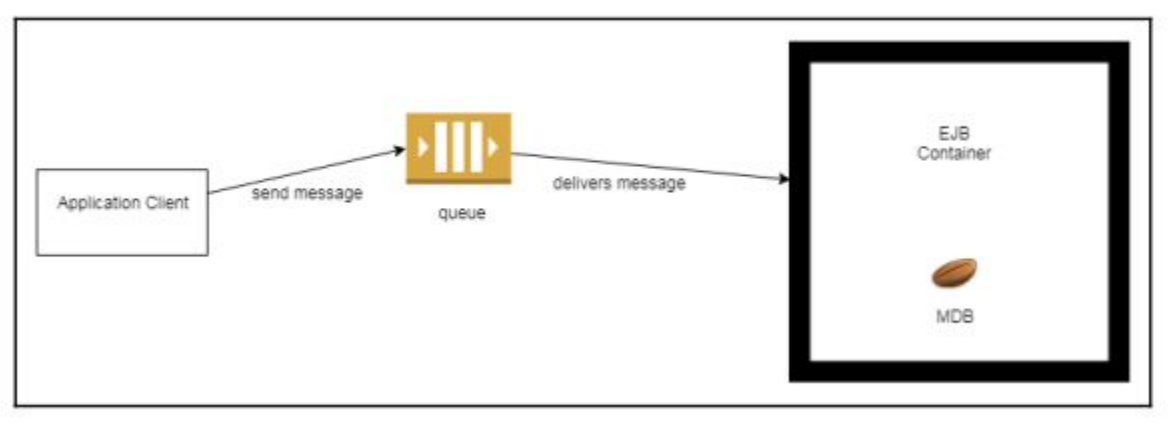
Java Message Service (JMS) and Message Oriented Middleware (MOM)

MOM - Architektura wspierająca wysyłanie i odbieranie wiadomości pomiędzy modułami aplikacji bądź systemami rozproszonymi.

JMS to API zapewniające interfejs MOM (Message-oriented middleware) dla klientów, którzy chcą wykonać zadanie asynchronicznie. JMS stał się częścią EJB w specyfikacji EJB 2.0 i wprowadzono nowy moduł sesyjny: message-driven bean (MDB)

Message Driven Bean (MDB)

Bezstanowe ziarno, używane do nasłuchiwania zapytań dodawanych do kolejki JMS. Może implementować dowolny rodzaj wiadomości, choć najczęściej jest używany do obsługi komunikatów JMS



```
@MessageDriven (mappedName = "myQueue")
public class BeanMessage implements MessageListener
{
   @Override
   public void onMessage (Message message)
   {
        try {
               // przetwarzanie wiadomości
        } catch (JMSException ex) {
               // obsługa wyjątków
        }
   }
}
```

- Klasa publiczna, nie abstrakcyjna i nie finalna
- Konstruktor publiczny bez argumentów
- `mappedName` - nazwa serwisu JMS, który odbierze wiadomość
- Implementacja interfejsu `MessageListener` i metody `onMessage`

- Kontekst transakcji inny niż wysyłającego
- Może wysyłać wiadomości JMS
- Brak interfejsu służącego dostępowi do klienta

```
public class MessageSender {

  @Inject
  @JMSConnectionFactory("jms/connectionFactory")
  JMSContext context;

  @Resource(mappedName = "jms/myQueue")
  Destination queue;

  public void sendSomeMessage (String message) {
    context.createProducer().send(queue, message);
  }
}
```

`@JSMConnectionFactory` wskazuje która `ConnectionFactory` powinna być użyta do stworzenia `JMSContext`

```
@MessageDriven(
  activationConfig = { @ActivationConfigProperty(
  propertyName = "destinationType", propertyValue = "javax.jms.Queue")
})
public class EmailService implements MessageListener {
  @Resource
  private MessageDrivenContext mdc;

  public void onMessage (Message message) {
    try {
      String str = message.getBody (String.class);
    }
    catch (JMSException ex) {
      // obsługa wyjątków
      mdc.setRollbackOnly();
    }
  }
}
```

`@MessageDriven` zmienia zwykłą klasę w Message Driven Bean

`destinationLookup` i `destinationType` określa nazwę kolejki lub *tematu*

`MessageDrivenContext` może być wstrzyknięty do Message Driven Bean i umożliwić dostęp do jej kontekstu podczas uruchomienia programu - np. wykorzystać metodę `setRollbackOnly`

Specyfikacja EJB (Enterprise JavaBeans)

Idea EJB opiera się na tworzeniu komponentów (ziaren EJB), które mogą być osadzane na serwerze aplikacji (tzw. kontenerze EJB), który z kolei udostępnia je do wykonania lokalnie.

Zapewnia usługi takie jak transakcyjność, trwałość, rozproszenie, bezpieczeństwo i wielodostęp.

```
@Stateless
public class MyBean {

  @Asynchronous
  public void veryTimeConsumingProcess1
  (SomeFilterBean filter) { }

  @Asynchronous
  public Future veryTimeConsumingProcess2
  (SomeFilterBean filter) { }
}
```

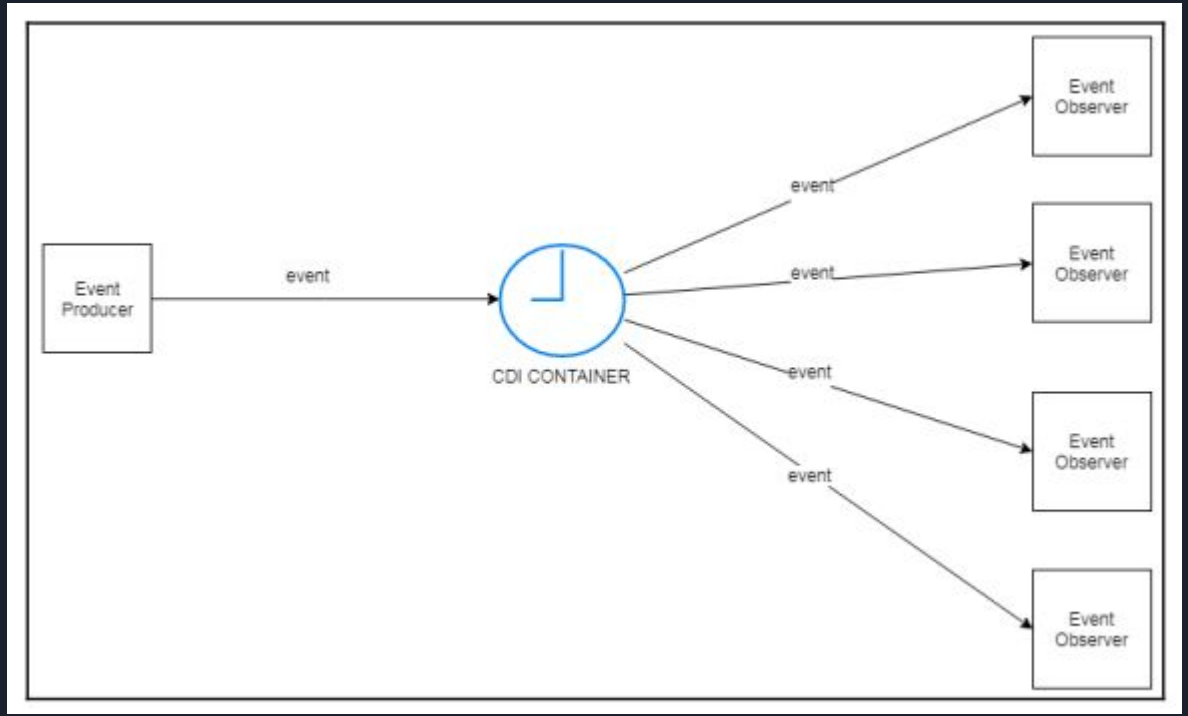
- adnotacja `@Asynchronous` czyni pojedyncze metody lub całe klasy asynchroniczne
- obiekt `Future` umożliwia sprawdzenie stanu asynchronicznego procesu - zwrot bez blokowania wątku klienta

```
@javax.ejb.Stateless
public AcademicServiceBean {
 @javax.ejb.Asynchronous
 public Future requestTestReview(Test test) {
  // ...
 }
}

@Named
@SessionScope
public TestController {
 @Inject
 private AcademicServiceBean academicBean;
 private Future statusTestReview;

 public void requestTestReview(){
  Test testToBeReviewed = ...;
   this.statusTestReview = academicBean.requestTestReview(testToBeReviewed);
 }
 public Future<TestReview> checkTestReviewStatus() {
  // ...
 }
}
```

Metoda asynchroniczna może zwracać Future w celu sprawdzania rezultatu

Producent i konsument

Mechanizm składający się z producentów i konsumentów zdarzeń, czyli komponentu tworzącego wiadomości i komponentu je odbierającego.



```
public class SeminarProducer {
  @Inject private Event<Seminar> seminarEvent;
  public void sendEmailProcess(Date date, String title,
  String description) {
    Seminar seminar = new Seminar(date, title, description);
    seminarEvent.fireAsync(seminar);
  }
}
```

Przykład jednego producenta i dwóch obserwatorów. Obserwatorzy asynchronicznie czekają na wydarzenie i odpowiednio na nie reagują.

```
public class SeminarServiceBean {
  public void inviteToSeminar (@ObservesAsync Seminar seminar) {
    // send email for the college students inviting for the seminar
  }
}

public class StatisticControlingBean {
  public void generateStatistic (@ObservesAsync Seminar seminar) {
    // create some statistic data
  }
}
```

Miłego programowania!