Zakres testu: Maven (fazy budowania, zależności, repozytoria), testy jednostkowe, dziennik zdarzeń (poziomy, podstawowe użycie), JDBC (sterownik, pule połączeń, podstawowe użycie), JPA (mapowanie , zasada działania (Persistence Context, synchronizacja z bazą przez instrukcje SQL)), podstawy JPQL), wzorzec DAO, pojęcie kontenera, serwer webowy vs aplikacyjny, wstrzykiwanie zależności, katalog JNDI, interceptory, komponenty serwlet (zastosowanie, cykl życia), komponenty EJB (sesyjne i MDB, zastosowanie, cykl życia, dostęp przez interfejsy lokalny, zdalny, bez interfejsu), komponenty CDI (zastosowanie, cykl życia), deklaratywna obsługa transakcji w EJB/CDI, wywołanie metod zgodnie z harmonogramem, metody asynchroniczne w EJB, JMS (mechanizm komunikacji poprzez wysyłanie komunikatów w modelu Point to Point oraz Publish/Subscribe), JAX-RS (odwzorowanie metod na żądania HTTP, odbieranie danych z ciała żądania lub URL, zwracanie odpowiedzi), model warstwowy aplikacji (warstwy danych, logiki, prezentacji)

==========[1. Wykład – Testy]==========

Czym są?

- oprogramowanie weryfikujące poprawność działania poszczególnych jednostek kodu (metod, obiektów, procedur).
- specjalnie napisane fragmenty kodu, które wywołują metody testowanych klas/modułów i sprawdzają czy zwracane wyniki/stan systemu są zgodne z oczekiwaniami.
- testy jednostkowe służą do sprawdzania poprawności implementacji, czy logika jej działania jest zgodna z oczekiwaniami.
- przy tworzeniu testów wykorzystuje się zazwyczaj biblioteki/narzędzia wspomagające ich tworzenie, automatyczne uruchamianie, tworzenie raportów wykonania – w środowisku Java popularne rozwiązania są oparte na bibliotekach JUnit lub Test NG.

Korzyści:

- wczesne wykrywanie błędów – testy jednostkowe mogą być zintegrowane z systemem budowania aplikacji oraz systemami pracy grupowej.
- kontrola poprawności działania po wprowadzeniu zmian/refaktoringu – ułatwienie wprowadzania zmian.
- ułatwienie integracji elementów aplikacji poprzez wcześniejsze sprawdzenie

ich działania
- dodatkowa dokumentacja w postaci przykładów poprawnego użycia testowanych elementów.
- dodatkowy element wspierający projektowanie – przypadki testowe określają wymagania odnoście działania testowanych elementów.

Wady (potencjalne):

- dodatkowy nakład pracy związany z pisaniem kodu testów – ale czas jest zazwyczaj odzyskiwany dzięki mniejszej liczbie błędów i ich wcześniejszemu wykrywaniu oraz szybszemu poprawianiu.
- konieczność modyfikacji testów przy zmianach API/działania systemu – ale zmienione api tez może być obarczone błędami, które zauktualizowane test są w stanie wykryć.
- poprawność wykonania testów jednostkowych nie daje 100% pewności, że kod jest wolny od błędów.

Praktyki:

- każdy z testów powinien weryfikować pojedynczą funkcjonalność danej klasy/metody – tak aby w przypadku niepowodzenia testu można było szybko stwierdzić, która funkcja nie jest poprawnie zaimplementowana.
- poszczególne testy powinny wykonywać się niezależnie od pozostałych, kolejność ich wykonania nie powinna mieć znaczenia.
- testy jednostkowe można tworzyć przed, w trakcie lub po utworzeniu testowanej implementacji.
- wskazane jest tworzenie testów możliwie wcześnie, aby przy tworzeniu/modyfikacji implementacji była możliwość ich wykonywania w celu możliwie wczesnego wykrycia ewentualnych błędów.

Wykorzystanie testów w procesie tworzenia oprogramowania: Test-Driven Development (TDD).

1) Utworzenie testu dla nowej funkcjonalności.
2) Uruchomienie testów – nowo dodany test powinien się udać.
3) Napisanie implementacji dla nowej funkcjonalności.
4) Uruchomienie testów – nowo dodany i wcześniejsze testy powinny się udać.
5) Refaktoryzacja implementacji.

6) Uruchomienie testów – nowo dodany i wcześniejsze testy powinny się udać.
7) Powtórzenie cyklu dla kolejnej funkcjonalności.

Asercje.

Asercja – warunek (predykat) umieszczony w kodzie, programista zakłada jego prawdziwość.

==========[2. Wykład – Maven]==========

Maven – repozytoria.

Zdalne – umieszczone na zewnętrznym serwerze; instalacje maven obsługują kilka domyślnych serwerów, z których pobierane są udostępnione publiczne biblioteki wymagane przez projekt; istnieje możliwość zdefiniowania własnego serwera repozytorium. Domyślnie repozytoria definiowane są w master pom'ie.
Lokalne – lokalny cache, który jest sprawdzany przed próbą pobrania archiwum ze zdalnego repozytorium; pobrane archiwa są w nim automatycznie zapisywane na wypadek przyszłych żądań. Repozytorium lokalne jest przechowywane w katalogu %user_home%/.m2/repository.

Maven – fazy budowania.

1) validate – walidacja projektu, sprawdzenie dostępności wymaganych elementów.
2) compile – kompilacja kodu.
3) test – wykonanie testów jednostkowych.
4) package – utworzenie wynikowego artefaktu, np. archiwum jar lub war.
5) integration-test – instalacja pakietu w środowisku do testów integracyjnych.
6) verify – weryfikacja wyników testów integracyjnych zbudowanego artefaktu.
7) install – skopiowanie utworzonego artefaktu do lokalnego repozytorium.
8) deploy – skopiowanie utworzonego artefaktu do zdalnego repozytorium, skąd może być pobrany z innych komputerów.

Maven – zależności.

Może chodzić o to, że w projekcie Maven biblioteki mogą być zależne od innych i powstaje taki graf zależności. Przykładowo biblioteka *mvn-basic* korzysta ze *spring-context*, która korzysta z *spring-aop*, która korzysta z *aopalliance*.

==========[3. Wykład – Dziennik zdarzeń]==========

Log – podstawowe użycie.

- obiekt dziennika (logger, log) jest zazwyczaj definiowany jako statyczne prywatne pole w klasie, które korzysta z dziennika lub jako statyczne pole chronione w klasie bazowej.
- instancja loggera jest tworzona z użyciem obiektu fabryki z wybranej biblioteki.
- każda instancja ma unikalną nazwę – zgodnie z konwencją jest to nazwa klasy poprzedzona nazwą pakietu; ponowne pobranie z fabryki obiektu o tej samej nazwie zwraca tę samą instancję.
- wpis do dziennika wykonywany jest przez wywołanie odpowiedniej metody, np.: *Logger.fine(String message)* – wpisanie tekstu message na poziomie FINE, *Logger.log(Level level, String message, Throwable e)* – wpisanie tekstu message oraz stosu wyjątku e na poziomie level.
- dodatkowo dziennik może uzupełnić wpis o takie informacje jak czas zdarzenia, nazwa użytego loggera, lokalizacja z której wykonano wpis: nazwa klasy/metody/pliku/numer linii.
- informacje o wyjątkach (błędach) najwygodniej wpisywać do dziennika w miejscu ich obsługi, unikamy wtedy wielokrotnego zgłaszania tego samego błędu w trakcie przechodzenia wyjątku przez aplikację.

Log – poziomy zdarzeń.

- poziom zdarzenia określa jego ważność/priorytet oraz rodzaj informacji (negatywna, pozytywna).
- najwyższy priorytet mają komunikaty dotyczące błędów: SEVERE w JDK Logging oraz FATA/ERROR w Log4j i Logback.
- w dalszej kolejności są ostrzeżenia WARNING: aplikacja działa ale są pewne problemy wpływające negatywnie na jej funkcjonalność, np.: nie znaleziono pliku z konfiguracją więc zastosowano domyślne ustawienia.
- niższy priorytet mają wpisy dla zdarzeń zachodzących w poprawnym przebiegu: INFO/CONFIG/FINE/FINER/FINEST w JDK Logging oraz INFO/DEBUG/TRACE w Log4j oraz Logback; im niższy priorytet takiego wpisu tym bardziej szczegółowe zadanie, którego dotyczy informacja.
- ogólnie im niższy priorytet tym więcej informacji jest wpisywanych na tym

poziomie do dziennika: są bardziej szczegółowe/dotyczą drobniejszych etapów.
- z tego względu poziom jest wykorzystywany do odfiltrowywania nadmiaru informacji.
- konfiguracja dziennika pozwala na ustalenie poziomu odcięcia – do dziennika zostaną wpisane tylko komunikaty z tego poziomu i te o wyższym priorytecie, pozostałe będą pomijane.
- na przykład: ustalenie poziomu odcięcia na INFO w JDK Logging, spowoduje, że komunikaty SEVERE, WARN i INFO będą wpisywane do dziennika, ale CONFIG, FINE, FINER i FINEST zostaną pominięte.

<table>
<tr><th>JDK Logging</th><th>Log4j, Logback, Common Logging</th><th>priorytet</th><th>rodzaj informacji</th></tr>
<tr><td>SEVERE

WARNING</td><td>FATAL
ERROR

WARN</td><td rowspan="2">najwyższy

najniższy</td><td>negatywna</td></tr>
<tr><td>INFO
CONFIG
FINE
FINER
FINEST</td><td>INFO
DEBUG
TRACE</td><td>pozytywna</td></tr>
</table>

==========[4. Wykład – JDBC]==========

Sterownik JDBC.

- stanowi implementacje API i odpowiada za komunikację z bazą danych.
- dołączany do aplikacji jako biblioteka jar lub umieszczony na serwerze aplikacyjnym.

Rodzaje sterowników JDBC.

- type 1 – przekazuje zadanie komunikacji z bazą do sterownika ODBC, implementacja przyjmuje formę pomostu/tłumacza pomiędzy API JDBC a interfejsem ODBC; najmniej wydajny rodzaj sterownika, stosowany przy braku innych typów.
- type 2 – napisany po części w Java i natywnym kodzie dla danej architektury sprzętowej/systemu operacyjnego. Wymaga zainstalowania dodatkowego

sterownika (zależnego od systemu), z którym komunikuje się sterownik Java dołączony do aplikacji.
- type 3 – zaimplementowany w całości w Javie, korzysta z warstwy pośredniej do komunikacji z bazą danych.
- type 4 – zaimplementowany w całości w Javie, komunikuje się bezpośrednio z bazą z użyciem specyficznego protokołu danej bazy; zazwyczaj najszybszy (najczęściej stosowany).

Pule połączeń JDBC.

- utworzenie połączenia jest czasochłonne, tworzenie nowego połączenia na każdą operacje w bazie może skutkować spadkiem wydajności.
- lepsze rozwiązanie to zastosowanie puli połączeń: implementacja DataSource może utrzymywać cały czas pewną liczbę zastawionych połączeń – pulę połączeń.
- w przypadku wywołania metody getConnection(), źródło połączeń sprawdza czy w puli jest dostępne wolne połączenie i jeśli tak to od razu je zwraca (nie tracimy czasu na zestawienie nowego połączenia, tylko od razu dostajemy gotowe).
- po zakończeniu pracy z połączeniem, wywołanie metody close() nie spowoduje zerwania połączenia z bazą, tylko zwrot do puli – te same fizyczne połączenie będzie dostępne dla kolejnych wywołań getConnection().

- konfiguracja puli połączeń określa takie parametry jak: minimalny rozmiar puli, maksymalny rozmiar puli, maksymalny czas oczekiwania na zwrócenie połączenia do puli, maksymalny czas przechowywania w puli niewykorzystanych połączeń.
- od strony aplikacji, korzystanie z puli jest przeźroczyste – aplikacja używa tego samego interfejsu DataSource.getConnection() do pozyskania połączenia.
- pula może być skonfigurowana jako zasób w serwerze aplikacyjnym, konfiguracja puli może być też dodana do samej aplikacji.
- jeśli aplikacja działa w środowisku serwera aplikacyjnego to dostęp do obiektu DataSource pozyskuje z katalogu JNDI lub otrzymuje referencje z użyciem mechanizmu Dependency Injection.

Przykładowe użycie JDBC

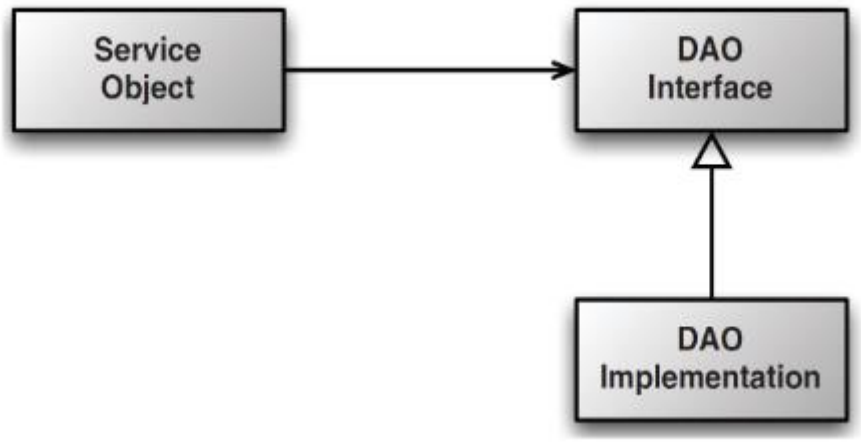
```java
public class UserDaoJdbcImpl implements UserDao {
    @Override
    public void save(User t) {
        final String SQL = "insert into user values (DEFAULT,?,?,?)";
        try (Connection conn = ConnectionFactory.getConnection();
             PreparedStatement statement = conn.prepareStatement(SQL, PreparedStatement.RETURN_GENERATED_KEYS)) {
                statement.setString( parameterIndex: 1, t.getLogin());
                statement.setString( parameterIndex: 2, t.getPassword());
                statement.setString( parameterIndex: 3, t.getEmail());
                statement.executeUpdate();
                try (ResultSet rs = statement.getGeneratedKeys()) {
                    if (rs.next()) {
                        t.setId(rs.getLong( columnIndex: 1));
                    }
                } catch (SQLException ex) {
                    throw new DataAccessException(ex);
                }

        } catch (SQLException ex) {
            throw new DataAccessException(ex);
        }
    }
```

```
# konfiguracja bazy h2 w trybie embedded z zapisem danych do pliku katalogu roboczym
jdbcUrl=jdbc:h2:file:./h2data;INIT=runscript from 'classpath:/schema.sql'
username=sa
password=sa
autoCommit=true
```

Wzorzec DAO (Data Access Object).

- pozwala na ukrycie implementacji dostępu do źródła danych.
- ułatwia podmianę źródła danych/bibliotek/implementacji realizujących dostęp do źródła.
- uwalnia operacje w warstwie logiki od szczegółów implementacji dostępu do danych.
- występuje często w parze z DTO (Data Transfer Object), który kapsułkuje przesyłane dane.



==========[5. Wykład – JPA]==========

Mapowanie.

Mapowanie – sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych o relacyjnym charakterze. Implementacja takiego odwzorowania stosowana jest m.in. w przypadku, gdy tworzony system oparty jest na podejściu obiektowym, a system bazy danych operuje na relacjach. Z ORM związany jest szereg problemów wydajnościowych.

- w programach obiektowych dane są reprezentowane w postaci obiektów i ich stanu.
- związki między obiektami są określone przez pola typu referencja lub kolekcje referencji.
- dodatkowo statyczne związki są modelowane z użyciem dziedziczenia.
- zapis obiektów i ich powiązań w relacyjnej bazie danych wymaga odpowiedniego odwzorowania modelu obiektowego na model relacyjny: klasy na tabele, pola na kolumny, obiekty na wiersze, referencje na klucze obce.
- niezgodność reprezentacji danych w postaci obiektowej i relacyjnej jest nazywana brakiem dopasowania „impedancji”.

- pewien problem stanowi również sposób komunikacji z relacyjną bazą danych za pomocą instrukcji SQL.
- operacje odczytu i zapisu obiektów musza być zamieniane na tekstowe instrukcje, dodatkowo przy odczycie jest konieczna transformacja wierszy wyniku na instancje obiektów.
- narzędzia ORM automatyzują zamianę operacji zapisu/odczytu na instrukcje SQL oraz transformację danych z postaci wierszowej na obiekty.
- ORM zapewnia mechanizmy do odwzorowywania klas reprezentujących dane i związków między klasami na tabele i kolumny w modelu relacyjnym.

Persistence Context.

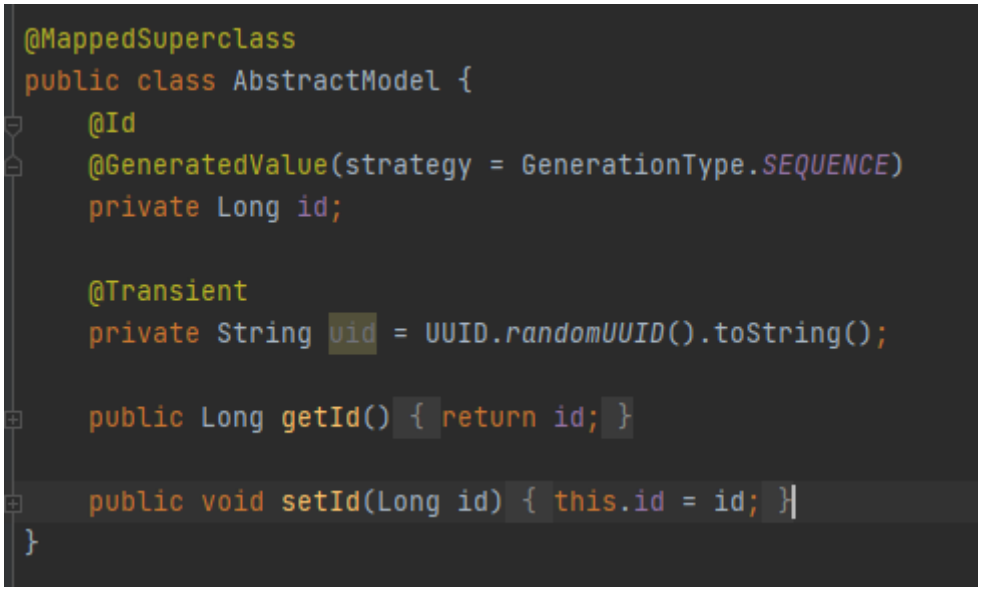
Persistence Context – zbiór obiektów encji, których stan jest synchronizowany z zawartością bazy.

Persistence Context – zasada działania

- W momencie przekazania obiektu w stanie NEW do metody EntityManager.persist lub merge jego referencja (dla persist) lub referencja do jego kopii (dla merge) jest dodawana do zbioru Persistence Context zarządzanego przez menadżera encji.
- Obiekty w Persistence Context są w stanie MANAGED.
- Persistence Context to zbiór referencji do obiektów encji, które w momencie zatwierdzania transakcji (lub nawet w trakcie jej trwania) zostaną zsynchronizowane z bazą danych, co oznacza, że zostaną wygenerowane i wysłane do bazy odpowiednie instrukcje SQL, których celem jest aktualizacja stanu bazy tak, żeby odpowiadał stanowi obiektów encji w Persistence Context.
- Gdy zbiór PersistenceContext jest usuwany wszystkie obiekty, które w nim się znajdują przechodzą w stan DETACHED.

==========[6. Wykład – JPQL]==========

Podstawy JPQL



```java
@NamedQueries({
        @NamedQuery(name = "Car.findAll", query = "SELECT c FROM Car c"),
        @NamedQuery(name = "Car.findAvailableCars", query = "SELECT c FROM Car c WH
})
@Entity
public class Car extends AbstractModel {
    public enum State { AVAILABLE, BOOKED, OCCUPIED}

    private String brand;
    private int numberOfSeats;
    private int numberOfDoors;
    private boolean airConditioning;
    private boolean manualGearbox;
    @Column(precision = 13, scale = 2)
    private BigDecimal price;
    @Enumerated
    private State state;
    @OneToMany(mappedBy = "car", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Reservation> reservations = new LinkedList<>();

//    public Car() { }

    public Car() {
```

```java
// Methods.
public void addReservation(Reservation reservation) {
    this.reservations.add(reservation);
    reservation.setCar(this);
}

public void removeReservation(Reservation reservation) {
    this.reservations.remove(reservation);
    reservation.setUser(null);
}
```

```java
@Stateless
public class CarDaoImpl implements CarDao {
    @PersistenceContext(name = "PU")
    private EntityManager entityManager;

    @Override
    public Car save(Car t) {
        if (t.getId() == null)
            entityManager.persist(t);
        t = entityManager.merge(t);
        return t;
    }

    @Override
    public void delete(Car t) {
        t = entityManager.merge(t);
        entityManager.remove(t);
    }

    @Override
    public Car findById(Long id) {
        return entityManager.find(Car.class, id);
    }

    @Override
    public List<Car> findAll() {
        return entityManager.createNamedQuery( s: "Car.findAll", Car.class).getResultList();
    }

    @Override
    public List<Car> findAvailableCars() {
        return entityManager.createNamedQuery( s: "Car.findAvailableCars", Car.class).getResultList()
    }
}
```

====================[7. Wykład – Kontenery EE]====================

Pojęcie kontenera.

Kontenery są składnikiem serwerów webowych lub aplikacyjnych, na których można uruchamiać aplikacje JEE. W zależności od rodzaju kontener:
- zarządza cyklem życia komponentów w trakcie działania aplikacji: tworzy i usuwa instancje komponentów, przenosi nieużywane komponenty z pamięci na dysk i przywraca je z dysku do pamięci w razie potrzeby.
- inicjalizuje komponenty w momencie ich tworzenia – inicjuje pola z referencjami do innych komponentów z których korzystają; udostępnia dane konfiguracyjne aplikacji, daje dostęp do API usług/technologii.
- zapewnia mechanizmy ułatwiające bezpieczne używanie komponentów przez wielu użytkowników.

- dodaje automatyczną obsługę transakcji, uwierzytelniania, autoryzacji, np.: metoda zostaje automatycznie objęta transakcją, przed jej rozpoczęciem transakcja jest tworzona, po jej zakończeniu zatwierdzana lub cofana jeśli zostanie wyrzucony wyjątek.
- wzbogaca język o dodatkowe techniki.

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web, enterprise bean, or application client component can be executed, it must be assembled into a Java EE module and deployed into its container.

Serwer webowy vs aplikacyjny

Serwery webowe udostępniają kontener serwletów i podzbiór technologii JEE umożliwiający tworzenie aplikacji webowych: Servlet, JSP, Expression Language, JASPIC, WebSocket.

Serwery aplikacyjne zawierają pełen komplet kontenerów JEE i technologii zdefiniowanych w danej wersji JEE (tzw. Full Profile)

- Web Server is designed to serve HTTP Content. App Server can also serve HTTP Content but is not limited to just HTTP. It can be provided other protocol support such as RMI/RPC
- Web Server is mostly designed to serve static content, though most Web Servers have plugins to support scripting languages like Perl, PHP, ASP, JSP etc. through which these servers can generate dynamic HTTP content.
- Most of the application servers have Web Server as integral part of them, that means App Server can do whatever Web Server is capable of. Additionally App Server have components and features to support Application level services such as Connection Pooling, Object Pooling, Transaction Support, Messaging services etc.
- As web servers are well suited for static content and app servers for dynamic content, most of the production environments have web server acting as reverse proxy to app server. That means while servicing a page request, static contents (such as images/Static HTML) are served by web server that interprets the request. Using some kind of filtering technique (mostly extension of requested resource) web server identifies dynamic content request and transparently forwards to app server

Komponenty serwlet – zastosowanie.

- zadaniem serwletu jest odpowiedź na żądania przychodzące przez sieć do aplikacji (przeważnie żądania HTTP).
- komponent serwlet obsługujący żądania http to instancja klasy, która dziedziczy z typu javax.servlet.http.HttpServlet.
- serwlet jest przypisywany do określonego adresu lub zakresu adresów URL w

obrębie aplikacji poprzez oznaczenie go adnotacją @WebServlet lub konfigurację w pliku web.xml dołączonym do aplikacji.
- kontener tworzy tworzy instancję serwletu w momencie instalacji aplikacji na serwerze lub przy pierwszym żądaniu http na adres obsługiwany przez serwlet. Jeśli do serwera z aplikacją zostanie wysłane żądanie http, kontener serwletów znajduje instancję obsługującą dany URL i wywołuje na niej metodę: void service(HttpServletRequqest req, HttpServletResponse resp).

Cykl życia serwletu.

1) Utworzenie instancji w momencie pierwszego żądania (domyślnie) lub przy starcie aplikacji.
2) inicjalizacja – init(ServletConfig config)
3) obsługa żądań – service(HttpServletRequest req, HttpServletResponse resp), domyślna implementacja woła jedną z poniższych metod zależnie od rodzaju żądania http: doGet, doPost, doPut, doDelete itp.
4) usunięcie – destroy()

==========[8. Wykład – EJB]==========

Wstrzykiwanie zależności (Dependency Injection)

W technice wstrzykiwania zależności kontener ustawia/wstrzykuje referencję do zasobu w polu danego komponentu podczas jego tworzenia. Zasoby mogą być wstrzykiwane tylko do komponentów, za których cykl życia odpowiada kontener (np.: serwlety, komponenty EJB, CDI, …). Jeśli kontener nie znajdzie wymaganego zasobu to zgłasza błąd w momencie uruchomienia aplikacji. Przykładowe adnotacje do wstrzykiwania: @EJB, @Resource, @PersistenceContext, @PersistenceUnit, @Inject

Katalog JNDI (Java Naming and Directory Interface)

JNDI określa jednolity interfejs dostępu do różnych implementacji usług katalogowych (LDAP, DNS, FS, Active Directory, …). Interfejs JNDI jest dostępny w Java SE.

Usługa katalogowa to rodzaj bazy danych, w której zasoby są rejestrowane pod nazwami o hierarchicznej strukturze (np.: nazwy domen, ścieżki do plików).

Serwery aplikacyjne zawierają implementację usługi katalogowej, w której rejestrowane są zasoby, np.: źródła połączeń do bazy, komponenty EJB, obiekty JMS, wartości parametrów z pliku deskryptora modułu. Dostęp do tej usługi jest wykonywany przez API JNDI.

Zasób nie może być przechowywany w usłudze katalogowej JNDI w postaci zserializowanego obiektu lub jako obiekt typu javax.naming.Reference, który pozwala na dynamiczne utworzenie/pozyskanie zasobu spoza katalogu w momencie jego pobrania.

Specyfikacja JEE definiuje 4 standardowe przestrzenie nazw w usłudze katalogowej serwera aplikacyjnego, które określają zakres widoczności umieszczonych w nich zasobów:
- java:global – przestrzeń współdzielona przez wszystkie aplikacje na serwerze, zasób w tej przestrzeni będzie widoczny we wszystkich aplikacjach.
- java:app – zasób widoczny tylko w aplikacji, w której został dodany.
- java:module – zasób widoczny tylko w module, w którym został dodany.
- java:comp – zasób widoczny w komponencie, dla którego został dodany.

Komponenty EJB sesyjne oraz zastosowanie.

- bezstanowy, nie przechowują stanu pomiędzy wywołaniami metod, idealne dla zadań, które można wykonywać metodą nie zmieniającą stanu obiektu, na którym została wywołana, np.: komponenty DAO – wywołanie operacji CRUD nie zmienia stanu komponentu DAO, tylko stan bazy lub zwraca dane z bazy.
- stanowy, reprezentują stan procesu, który zmienia się w czasie i nie powinien być współdzielony między klientami, np.: stan koszyka w trakcie procesu zakupów, stan konwersacji z klientem w trakcie sesji, stan przebiegu złożonej operacji, która składa się z etapów rozłożonych w czasie.
- singleton, zgodnie z wzorcem projektowym, kontener zapewnia istnienie tylko jednej instancji w ramach aplikacji. Zwykle służą do reprezentacji stanu współdzielonego dla całej aplikacji, np.: konfiguracja aplikacji, dynamiczne słowniki, cache.

Komponenty EJB MDB (Message-Driven Beans)

Są to komponenty sterowane wiadomościami, pozwalają na asynchroniczną komunikację poprzez wiadomości/komunikaty.

Zastosowanie EJB

- komponent działający po stronie serwera w kontenerze EJB.
- może zostać umieszczony w aplikacji webowej lub w osobnym module EJB, który jest częścią większej aplikacji lub być samodzielną aplikacją, która oferuje usługi dla innych.
- kontener EJB udostępnia dla komponentów EJB szereg usług, które znacznie upraszczają implementację wielu typowych operacji jak: obsługa transakcji, zdalne wywoływania metod, wywołania asynchroniczne metod, przechwytywanie wywołań metod, jednoczesna obsługa wielu użytkowników.
- komponenty EJB są skalowalne i umożliwiają rozproszenie aplikacji na wiele serwerów.
- komponenty EJB są również nazywane ziarnami EJB (EJB beans).

Cykl życia obiektów EJB.

Instancje EJB są tworzone przez kontener z użyciem konstruktora bezparametrowego. Po utworzeniu instancji, kontener wykonuje wstrzykiwanie zależności.
@PostConstruct – operacje do wykonania w trakcie inicjalizacji po wstrzyknięciu zależności.
@PreDestroy – operacje do wykonania przed usunięciem obiektu.
@PrePassivate (tylko dla @Stateful) – wykonanie operacji przed przejściem w stan pasywny, np.: zamknięcie otwartych plików/połączeń.
@PostActivate – (tylko dla @Stateful) – wykonanie operacji po wyjściu ze stanu pasywnego, np.: otwarcie plików, odtworzenie połączeń.

Komponent EJB – dostęp przez interfejsy

Publiczne metody instancyjne komponentu EJB, które klient może wywoływać są określone w JEE terminem metod biznesowych. Komponent EJB może również udostępniać swoje API przez typ interfejsowy, które implementuje klasa EJB. Wyróżnia się 2 rodzaje typów interfejsowych:

- Lokalne – służą do komunikacji w obrębie tej samej aplikacji, parametry i wartości zwracane przez metody przekazywane są klasycznie jak w zwykłym wywołaniu co jest najbardziej wydajne. Komunikacja bez pośrednictwa typu interfejsowego również ma charakter lokalny i jest tak samo wydajna. Serwer

może też wspierać lokalną komunikację między różnymi aplikacjami, które są na nim uruchomione.

- Zdalne – służą do komunikacji z obiektami EJB z innej aplikacji lub działającymi w innej maszynie wirtualnej/hoście. Parametry i wartości zwracane przez metody z tych interfejsów muszą być serializowane. Wywołanie metod z użyciem zdalnych interfejsów jest bardziej kosztowne dlatego nie należy ich używać jeśli jest możliwa komunikacja lokalna.

Interceptory

Interceptory to element programowania aspektowego, który pozwala na przechwycenie wywołania metody. W momencie wywołania przechwyconej metody sterowanie trafia do interceptora, który może:
- wykonać dodatkową logikę przed przekazaniem sterowania do przechwyconej metody.
- wykonać dodatkową logikę po powrocie z metody.
- zmienić argumenty przed przekazaniem sterowania do przechwyconej metody.
- zmienić zwróconą wartość.
- zablokować wywołanie: nie przekazać sterowania do przechwyconej metody i zwrócić ustaloną wartość lub wyrzucić wyjątek.

Na platformie JEE interceptory mogą być użyte do przechwytywania metod publicznych z komponentów EJB i CDI, metod callback cyklu życia, metod z adnotacją @Timeout wywoływanych przez Timer Service.

Deklaratywna obsługa transakcji w EJB.

Kontener EJB ułatwia stosowanie zarządzania transakcjami. W środowisku kontenera dostępne są dwa sposoby kontrolowania transakcji:
- transakcje zarządzane przez kontener (CMT), operacje tworzenia, zatwierdzania, cofania transakcji są automatycznie wykonywane przez kontener.
- transakcje kontrolowanie przez bean (BMT), operacje tworzenia, zatwierdzania, cofania transakcji są kontrolowane przez interfejs UserTransaction z Java Transaction API (JTA).

Transakcje w środowisku JEE mogą być:
-lokalne, obejmujące zestaw operacji na jednym zasobie (jednej bazie).
-rozproszone, obejmujące zestaw operacji na wielu zasobach (różne bazy danych, systemy kolejkowe…).

Wywołania metod zgodnie z harmonogramem.

Usługa dostępna dla komponentów EJB (z wyjątkiem stanowego) do tworzenia zadań uruchamianych automatycznie o określonej porze lub cyklicznie.
Są dwa sposoby użycia tej usługi:
- poprzez dodanie adnotacji javax.ejb.Schedule przy wybranych metodach komponentu EJB z przekazaniem harmonogramu przez atrybuty adnotacji.
- poprzez użycie adnotacji javax.ejb.Timeout na wybranej metodzie komponentu EJB i dynamiczne utworzenie harmonogramu z użyciem API TimerService.

Atrybuty adnotacji @Schedule

atrybut	wartość	domyślnie
second	0-59	0
minute	0-59	0
hour	0-23	0
dayOfWeek	0-7,Sun, Mon, Tue, Wed, Thu, Fri, Sat	*
dayOfMonth	1-31,Last,-7 - -1, [1st, 2nd, 3rd, 4th ...] [Sun Mon …]	*
month	1-12, Jan, Feb, ..., Dec	*
year	yyyy	*

Metody asynchroniczne w EJB.

Wywołanie metody asynchronicznej powoduje natychmiastowy powrót, przy czym wywołana metoda może być dalej przetwarzana na serwerze (jej wywołanie nie blokuje klienta). Można definiować metody komponentów sesyjnych EJB jako asynchroniczne poprzez adnotację @Asynchronous.

==========[9. Wykład – CDI]==========

Komponenty CDI – zastosowanie.

Komponenty CDI mogą być używane we wszystkich warstwach aplikacji. Przeważnie są używane przy implementacji warstwy prezentacji ze względu na dostępność takich cykli życia jak: czas obsługi żądania, czas istnienia sesji http, czas pozostawania na danej stronie JSF.

Oferowane usługi kontenera CDI to:
- zarządzanie cyklem życia obiektów z użyciem zakresów (scopes).
- wiązanie obiektów poprzez wstrzykiwanie zależności (typesafe dependency injection)
- możliwość wyboru różnych implementacji wstrzykiwanego interfejsu poprzez konfigurację.
- możliwość stosowania interceptorów.
- obsługa transakcji.
- luźne wiązanie obiektów z użyciem mechanizmu zdarzeń.

Cykl życia CDI

- obiekty CDI są tworzone i usuwane przez kontener.
- obiekt jest tworzony w momencie, gdy jest potrzebny i nie ma dostępnej instancji w aktywnym kontekście.
- jeśli obiekt jest już dostępny w aktywnym kontekście to jest używany.
- obiekt jest usuwany w momencie gdy kończy się jego zakres istnienia (scope).

Zakresy (scopes):
@Dependent – domyślny, czas istnienia ograniczony czasem istnienia obiektu, do którego został wstrzyknięty.
@RequestScoped – czas istnienia ograniczony do przetwarzania żądania.
@SessionScoped – czas istnienia ograniczony do czasu trwania sesji, wymagane Serializable.
@ApplicationScoped – czas istnienia do momentu zamknięcia aplikacji.
@CinversationScoped – czas istnienia do zakończenia konwersacji.
@ViewScoped – dodawany przez JSF 2.2

Deklaratywna obsługa transakcji w CDI.

W platformie EE od wersji 7 dla beanów CDI jest dostępna deklaratywna kontrola transakcji poprzez adnotacje @Transactional.

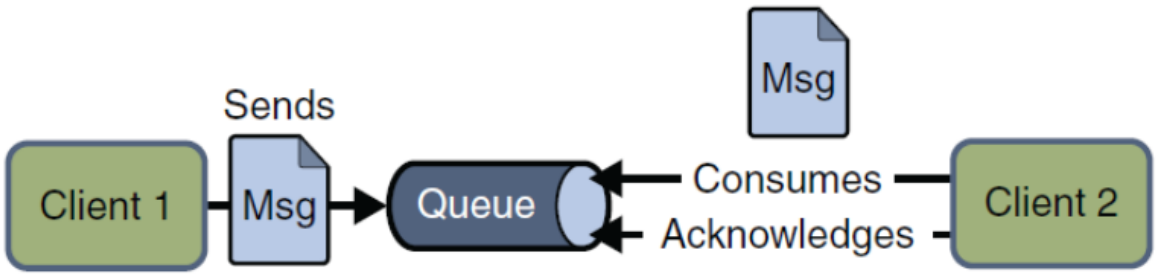
Atrybuty:
- value
  - REQUIRED: dołączenie do aktywnej transakcji lub utworzenie nowej, wartość domyślna.
  - REQUIRES_NEW: wstrzymanie aktywnej transakcji i utworzenie nowej.
  - SUPPORTS: dołączenie do aktywnej lub wykonanie bez.
  - MANDATORY: dołączenie do aktywnej lub wyjątek gdy brak.
  - NOT_SUPPORTED: wstrzymanie bieżącej i wykonanie bez t.
  - NEVER: wykonanie bez transakcji lub wyjątek gdy jest aktywna.
- rollbackOn (Class[]) – tablica klas wyjątków, których wyrzucenie ma spowodować cofnięcie transakcji.
- dontRollbackOn (Class[]) – tablica klas wyjątków, których wyrzucenie nie ma powodować cofnięcia transakcji.

Domyślnie transakcja jest cofana dla wyjątków unchecked.

==========[10. Wykład – JMS]==========

Java Message Service (JMS) – domena Point-to-point.

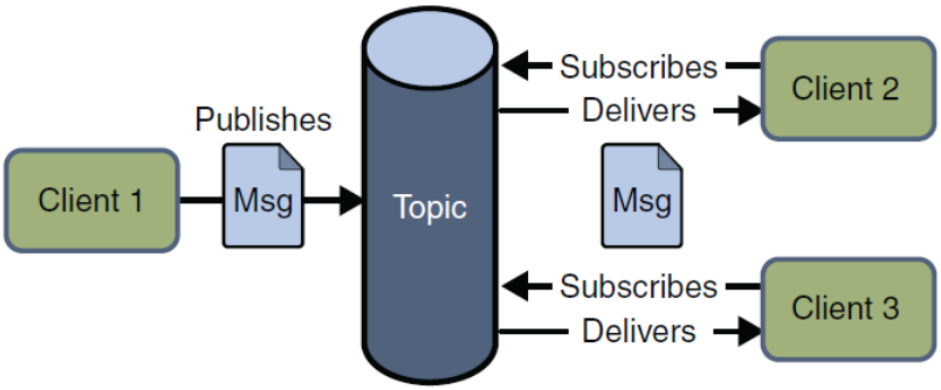
Oparta na koncepcji kolejek. Każda wiadomość jest adresowana do konkretnej kolejki i klient pobiera wiadomości z kolejki dla niego przeznaczonej. Wiadomości pozostają w kolejce, dopóki nie zostaną odebrane lub przedawnią się.



- każda wiadomość jest przekazywana tylko dla jednego odbiorcy na zasadzie first come first served.
- nie ma żadnej zależności czasowej pomiędzy nadawcą, a odbierającym.
- odbiorca potwierdza pomyślne dostarczenie wiadomości.

Java Message Service (JMS) – domena Publish/Subscribe.

- klient publikuje wiadomość do tematu, który działa jak forum lub tablica ogłoszeń.
- nadawca nadaje wiadomość, subskrybenci zapisani na temat ją otrzymują.
- dostawca zajmuje się dystrybucją wiadomości docierających do danego tematu – od wielu nadawców do wielu subskrybentów.
- temat zachowuje wiadomość dopóki nie rozdystrybuuje ich do aktualnych subskrybentów.
- każda wiadomość może mieć kilku odbiorców.
- istnieje zależność czasowa między nadawcami a typowymi subskrybentami – klient subskrybujący temat może odbierać wyłączeni te wiadomości, które zostały nadane w czasie jego subskrypcji (i aktywności).
- JMS pozwala również na tworzenie trwałych subskrypcji, które nie wymagają ciągłej aktywności subskrybenta.



====================[11. Wykład – JAX RS]=======================

Odwzorowanie metod na żądania http – JAX-RS.

```
@Path("/book")
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public class BookRestService {
    @Inject
    private BookDao bookDao;
    @Context
    private UriInfo uriInfo;
    @GET @Path("{id}")
    public Response getBook(@PathParam("id") String id) {
        Book book = bookDao.find(Integer.parseInt(id));
        return Response.ok(book).build();
    }
    @POST
    public Response createBook(Book book) {
        bookDao.save(book);
        URI bookUri = uriInfo.getAbsolutePathBuilder().path(book.getId().toString()).build();
        return Response.created(bookUri).build();
    }
    @PUT
    public Response updateBook(Book book) {
        bookDao.update(book);
        return Response.ok().build();
    }
    @DELETE @Path("{id}")
    public Response deleteBook(@PathParam("id") String id) {
        bookDao.remove(book.getId());
        return Response.noContent().build();
    }
}
```

Odbieranie danych z ciała żądania lub URL – JAX-RS.

```
@GET @Path("{id}")
public Response getBook(@PathParam("id") String id) {
    Book book = bookDao.find(Integer.parseInt(id));
    return Response.ok(book).build();
}
```

Schemat URI:

protokół://serwer:port/aplikacja/sciezka_globalna/sciezka_zasobu/parametry_sciezki?parametr_url=…

Zwracanie odpowiedzi – JAX-RS.

```
public class App {
   public static void main(String[] args) {
      URI uri = UriBuilder.fromUri("http://localhost:8080/appname/rest/book").port(8080).build();
      Client client = ClientBuilder.newClient();
      Book book = new Book(1,"title1",1.99);

      // dodanie książki (POST)
      Response response = client.target(uri).request().post(Entity.entity(book,
MediaType.APPLICATION_XML));
      System.out.println("Status code 1:"+response.getStatusInfo().getStatusCode());
      URI bookURI = response.getLocation();

      // pobranie (GET)
      response = client.target(bookURI).request().get();
      book = response.readEntity(Book.class);
      System.out.println("Status code 2:"+response.getStatusInfo().getStatusCode());
      System.out.println(book);

      // usunięce (DELETE
      String bookId = bookURI.toString().split("/")[6];
      response = client.target(uri).path(bookId).request().delete();
      System.out.println("Status code 3:"+response.getStatusInfo().getStatusCode());
   }
}
```

==========[12. Wprowadzenie]==========

Model warstwowy aplikacji.

Architektura warstwowa – podział złożonego systemu na warstwy, każda odpowiedzialna za określoną funkcjonalność i komunikująca się z warstwami sąsiednimi.

Warstwa prezentacji – odpowiedzialna za interakcję z użytkownikiem końcowym (wyświetlanie i wprowadzanie danych). W tej warstwie działają aplikacje klienckie takie jak przeglądarki internetowe.

Warstwa logiki – odpowiedzialna za przetwarzanie danych (żądań) od użytkownika. Tutaj też przygotowywane są dane wysyłane do warstwy prezentacji. W tej warstwie realizowane są wszelkiego rodzaju funkcjonalności biznesowe. Z drugiej strony warstwa logiki odpowiedzialna jest za komunikację z warstwą danych. Warstwa ta stanowi swego rodzaju pomost pomiędzy warstwą prezentacji a warstwą danych.

Warstwa danych – odpowiedzialna za przechowywanie danych. W tej warstwie mamy np. bazę danych.