Бизнес логика

… чрез EJB технологията

Петьо Димитров

01 Април 2013

Съдържание на лекциите

Въведение в Java EE технологиите

Презентация на данните

Бизнес логика

Съхранение на данните

Съдържание

Същност на EJB технологията

Видове *bean*-ове

Услуги на EJB контейнера

Демонстрация

Същност на EJB

- ENTERPRISE JAVABEANS
- платформа за създаване на portable, reusable и scalable бизнес приложения използвайки Java
- server-side *компоненти* използвани за изграждане на бизнес логиката и кода за съхранение на данни
- POJO класове капсулиращи логиката на приложението и разчитащи на услугите на контейнера

Същност на EJB (продължение)

- компонентният характер на EJB-тата позволява:
  - повторно използване на бизнес логика (например един модул за таксуване на кредитни карти, може да се вика от различни сайтове на компанията)
  - добра организация на приложението
- услугите на EJB контейнера позволяват:
  - директно използване на тествана комплексна функционалност (като транзакции, сигурност, AOP)
  - фокусиране върху бизнес логиката
- анотациите дават лесен начин за декларативно дефиниране на поведение от разработчика

Значение на анотациите

- без анотации:

```
<enterprise-beans>
  <session>
    <ejb-name>HelloUserBean</ejb-name>
    <local>example.Hello</local>
    <ejb-class>example.HelloUserBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>
```

- с анотации:

```
@Local
public interface Hello {}
@Stateless
public class HelloUserBean implements Hello{}
```

История

- **1.0** (1998): основи на EJB компонентите и отговорностите на контейнера
- **1.1** (1999): xml дескриптори, RMI-IIOP, security модел
- **2.0** (2001): стандартизиране на архитектурата, съвместимост между производители и други Java API-та, улеснено създаване (без нишки, транзакции)
- **2.1** (2003): поддръжка за уеб услуги, таймери, messaging функционалност
- **3.0** (2006): сериозно опростяване с помощта на *анотации*
- **3.1** (2009): bean-ове без интерфейси, singleton, асинхронно извикване, глобални JNDI имена

Видове EJB-та

- *session* - извиква се от клиент за изпълнението на определена бизнес операция (например таксуване на кредитна карта), биват:
  - *stateful* - запазва състоянието между извикванията
  - *stateless* - не запазва състояние, независими операции
  - *singleton* - съществува една единствена инстанция
- *message-driven* - също имплементира бизнес логика, но се стартира чрез пращане на съобщение на messaging сървър; позволява асинхронност
- *entity* - POJO съответстващо на запис в дадена таблица и носещо ORM информация (в анотации)

Общи изисквания към EJB класовете

- POJO клас имплементиращ един или повече бизнес интерфейса дефиниращи операциите (защо?)*
- бизнес интерфейсът използва анотациите `@Local`, `@Remote` или `@WebService`
- не може да е abstract или final (защо?)
- трябва да има конструктор без параметри (защо?)
- не може имената на методите да започват с `ejb…()`

Stateless EJB (`@Stateless`)

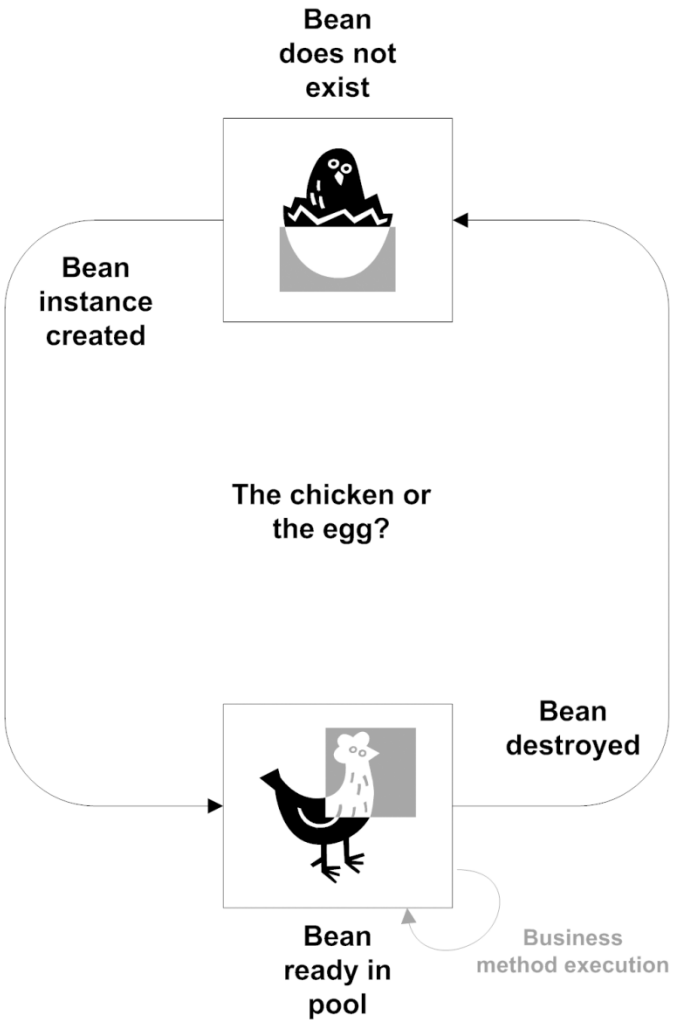
- не поддържа състояние между извикванията
- не се интересува от клиента
- работата се извършва с едно единствено извикване на метод
- може да има един или повече тематично свързани метода
- позволява pooling
- дава по-добро scalability
- *най-популярния вид*

Stateless EJB пример

```
@Local
public interface BidManager {
  Bid addBid(String bidderId,Long itemId,Double price);
  void removeBid(Long bidId);
  List<Bid> getBids(String userId);
}
@Stateless
public class BidManagerBean implements BidManager {
  @Override
  public Bid addBid(String bidderId, Long itemId, Double price) {
    Bid bid = createBid(bidderId, itemId, price);
    if (bid.getBidPrice().compareTo(bid.getItem().getInitialPrice()) < 0) {
      throw new IllegalArgumentException("Bid price lower than minimum...");
    }
    save(bid);
    return bid;
  }
  ...
}
```

Жизнен цикъл на stateless EJB

1. Създава се bean инстанцията използвайки конструктора без параметри.
2. Контейнерът инжектира ресурси.
3. Контейнерът складира инстанциите в pool.
4. При всяка нова заявка се взима свободен bean от pool-а (ако има свободен).
5. Изпълнява се избрания бизнес метод.
6. Връща се bean-ът обратно в басейна.
7. Унищожава се bean-ът.



Stateless EJB *callbacks*

- методи които се извикват в определена фаза от жизнения цикъл на bean-а (сходни с init() и destroy() методите при сървлети)
- обозначават се с анотации
- може да има няколко еднакви callback-а за един bean
- биват:
  - `@PostContruct` – извиква се в края на инициализирането на bean-а (след инжекцията на ресурси, стъпка 2) за осъществяване на комуникация с други системи
  - `@PreDestroy` – извиква се преди унищожението на bean-а (стъпка 7) за освобождаване на заети ресурси

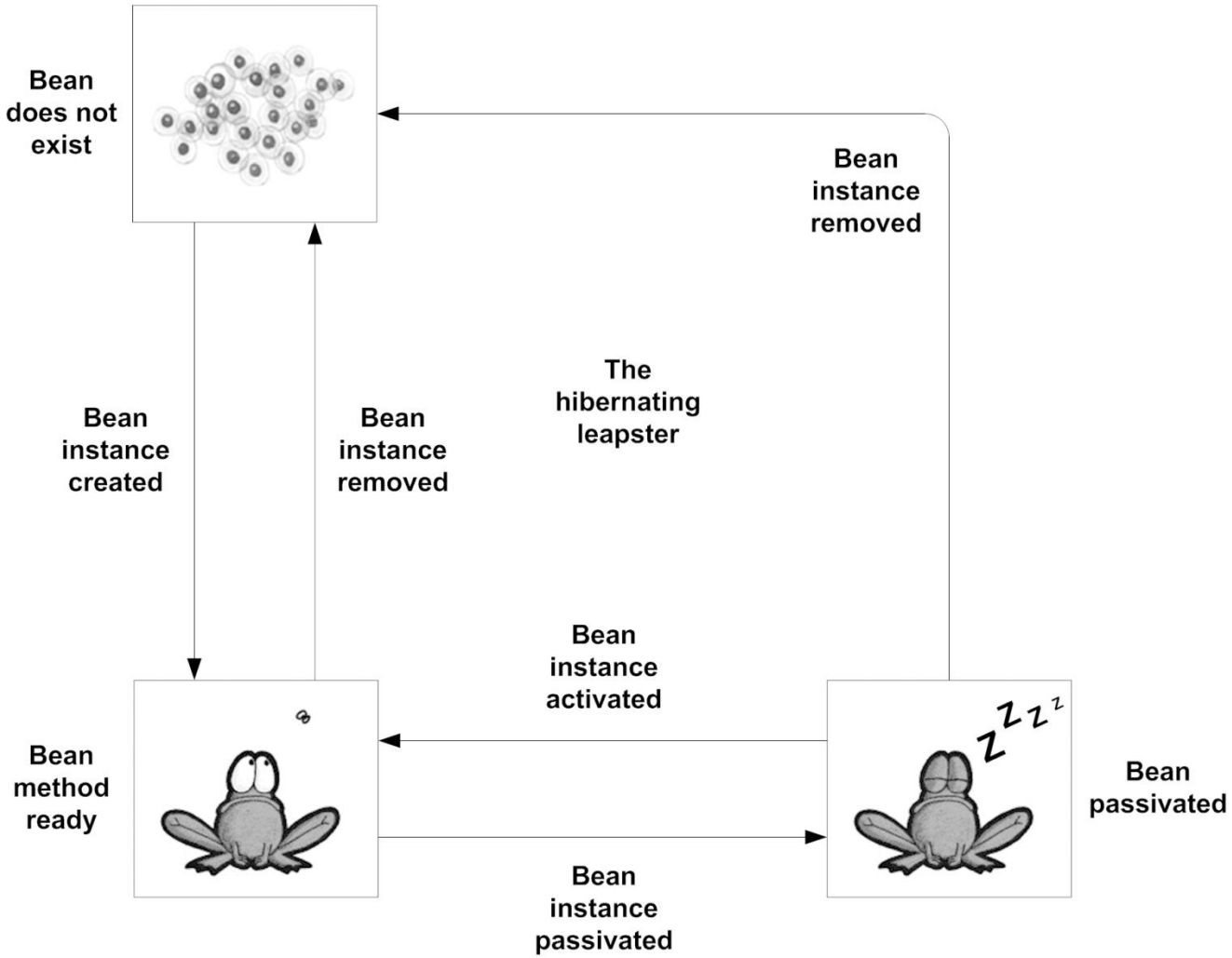
Stateful EJB (`@Stateful`)

- поддържа състояние между извикванията
- последователни заявки от един клиент се обработват от едно и също EJB
- пазенето на състояние пречи на използването на pooling
- използва се повече памет
- използва се passivation/ activation за намаляване на паметта
- използва се `@Remove` метод

Stateful EJB пример

```
@Stateful
public class PlaceOrderBean implements PlaceOrder {
  @EJB
  private ItemManager itemManager;
  private String bidderId;
  private Long itemId;
  private ShippingInfo shippingInfo;

  public void setBidderId(String bidderId) { this.bidderId = bidderId; }
  public void setItemId(Long itemId) { this.itemId = itemId; }
  public void setShippingInfo(ShippingInfo shippingInfo) { this.shippingInfo = shippingInfo;}

  @Remove
  public Long confirmOrder() throws OrderingException { ... }

  @PrePassivate
  private void prepare() {
    LOGGER.info("Passivaing stateful session bean of type " + getClass().getName());
  }

  @PostActivate
  private void restore() {
     LOGGER.info("Activating stateful session bean of type " + getClass().getName());
 }
}
```

Жизнен цикъл на stateful EJB



Stateful EJB *callbacks*

- поддържат се стандартните callback-ове (`@PostContruct` и `@PreDestroy`)
- дефинират се два нови callback-а свързани с управлението на състоянието:
  - `@PrePassivate` – извиква се преди *пасивирането* на bean-а (тогава се сериализират данните в bean-а и се складират на диска вместо в паметта); целта е да се освободят конекции/данни неподдържащи складиране
  - `@PostActivate` – извиква се след активацията на bean-а, с цел повторна инициализация на загубени конекции с външни системи

Singleton EJB (`@Singleton`)

- сходен със stateless session EJB, но със **само една** инстанция за цялото приложение
- полезен за споделяне на данните между bean-ове
- позволява изпълнение на логика при стартиране на приложението или края му
- позволява паралелен достъп от множество нишки

Singleton EJB пример

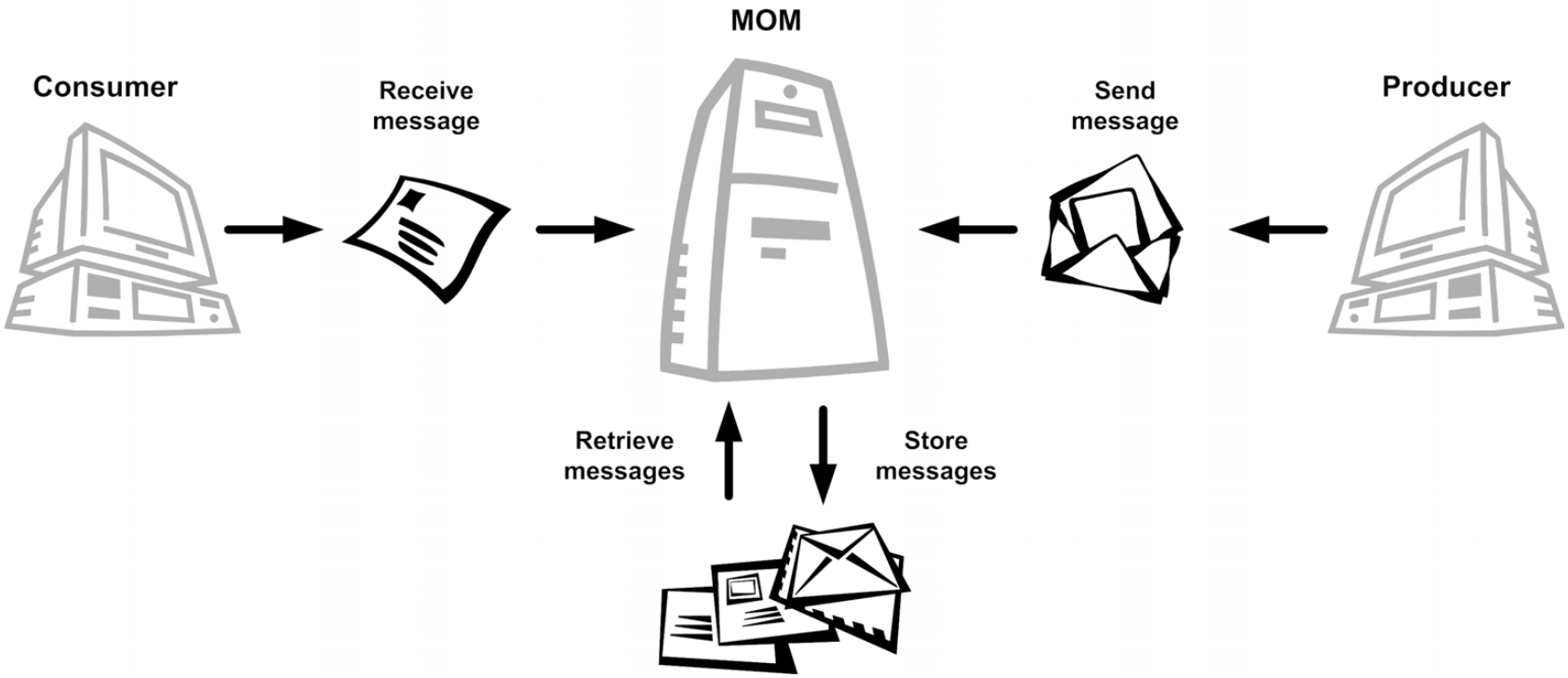
```
@Singleton
@Startup
@LocalBean
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class CommisionRateBean {

  private Double commisionRate;

  @PostConstruct
  private void initialize() { … }

  @Lock(LockType.READ)
  public Double getCommisionRate() { … }

}
```

Message-Driven EJB (`@MessageDriven`)

- консумира асинхронни съобщения идващи от MOM
- не се достъпва директно от клиенти
- сходен със stateless session EJB (не пази състояние, не се интересува от клиента и поддържа pooling)
- имплементира `MessageListener` интерфейса с метод `onMessage()`

Message-Oriented Middleware (MOM)

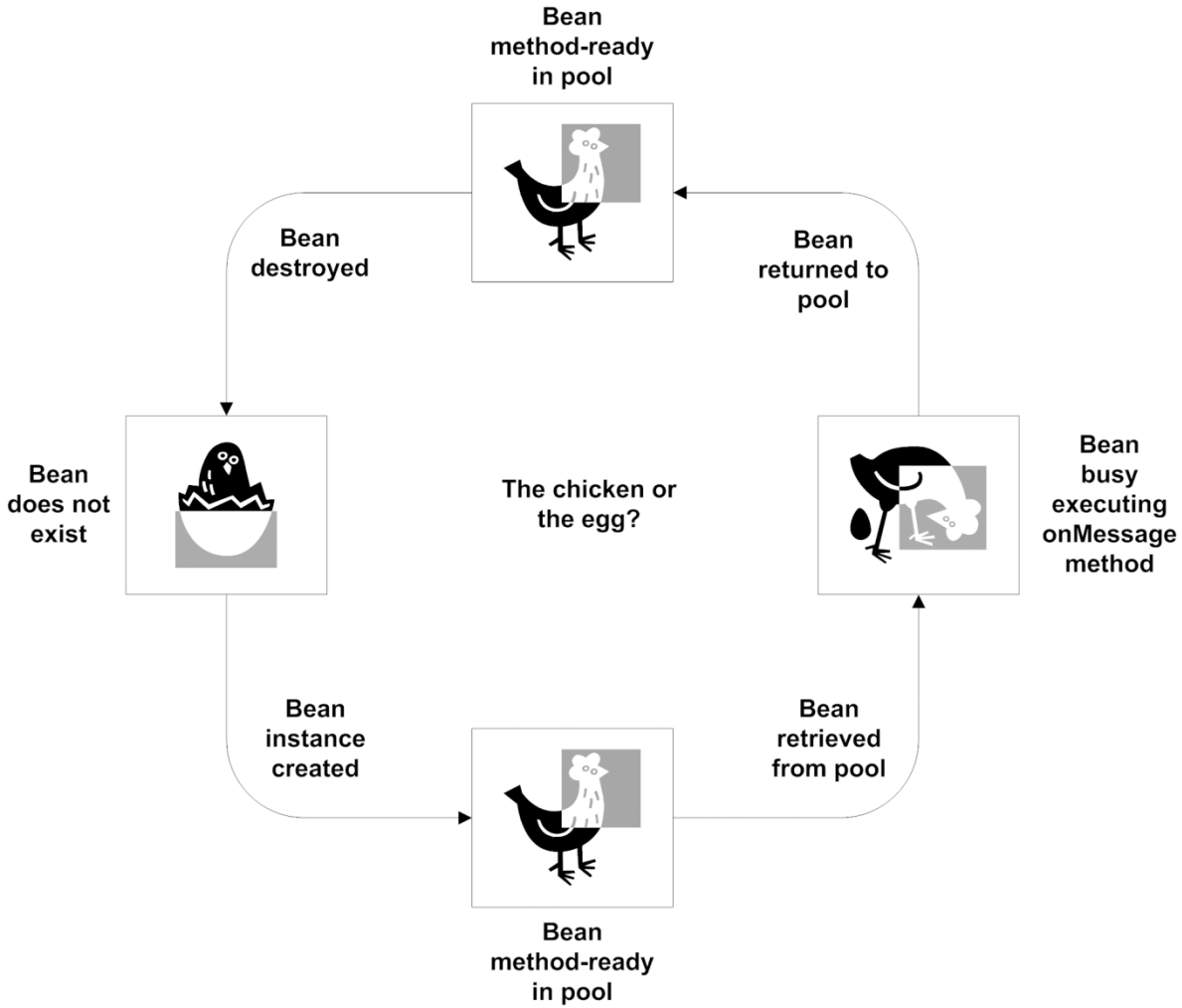
- асинхронно изпълнение и надеждност
- loose-coupling (пример)



Message-Driven EJB пример

```
@MessageDriven(mappedName = "java:app/jms_ShippingQueue",
activationConfig = {
@ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")})
public class ShipOrderBean implements MessageListener {

  public void onMessage(Message message) {
    ObjectMessage objectMessage = (ObjectMessage) message;
    Order order = (Order) objectMessage.getObject();
    if (order.getStatus() != OrderStatus.READY_FOR_SHIPMENT) {
      throw new IllegalStateException("...");
    }
    // perform shipping ...
    notifyShippingSuccess(order);
    order.setStatus(OrderStatus.COMPLETE);
  }

}
```

Жизнен цикъл на message-driven EJB



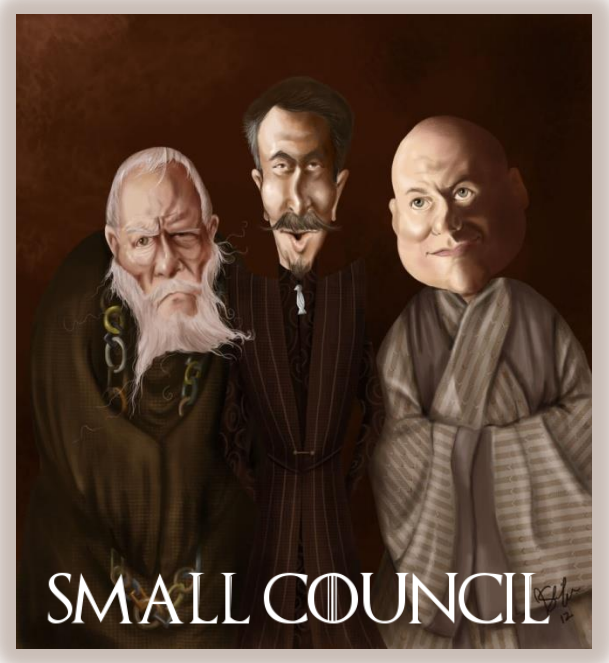
Entity (`@Entity`)

- Java клас представляващ една (или повече) таблици в базата
- носи информация за ORM мапинга
- всяка entity инстанция е POJO съдържащо данните от запис в таблицата (ите)

- *повече информация в лекцията за persistence*

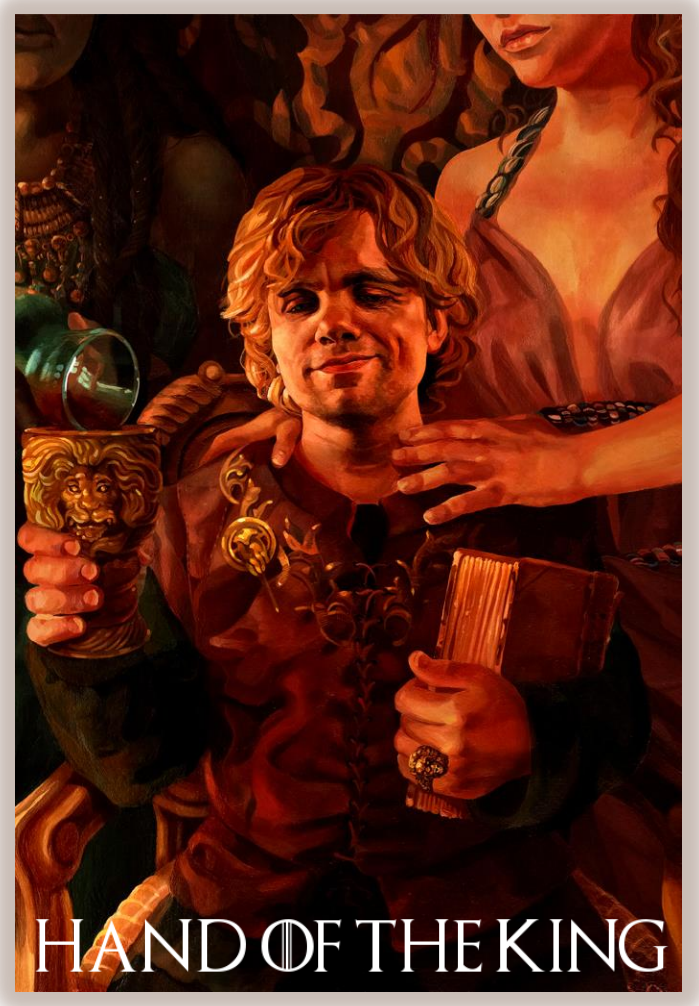
Контейнер услуги

- дават допълнителна стойност на приложението без нужда от писане на код (само конфигурация)
- примери:
  - **security**
  - **transactions**
  - **interceptors**
  - **scheduling**
  - thread-safety
  - persistence
  - state management
  - **dependency injection**
  - pooling
  - caching
  - messaging
  - remote access
  - web services

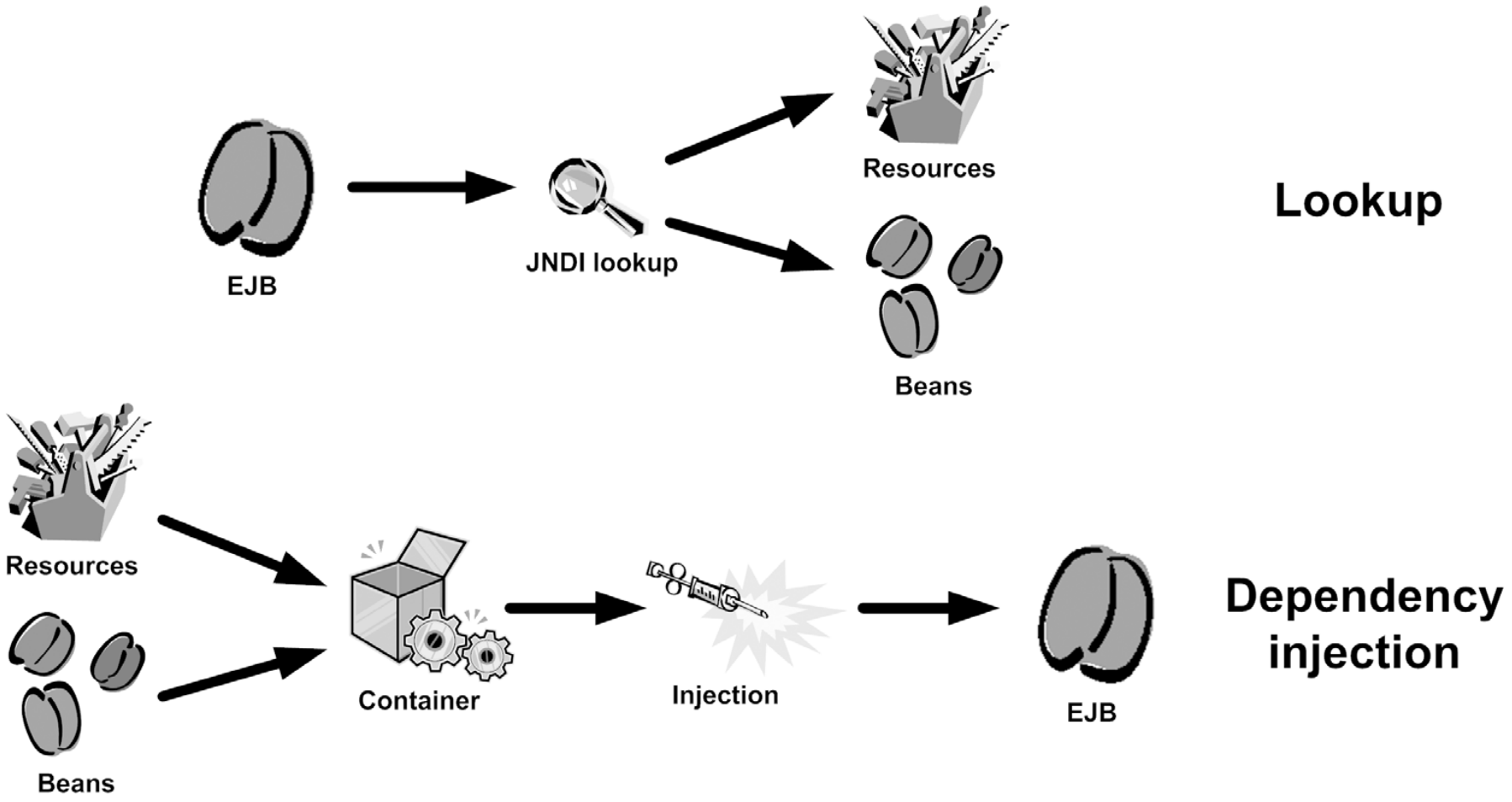


Dependency Injection (DI)

- цели да направи връзките между компоненти максимално *loosely coupled*
- компонент А извиква Б само чрез интерфейса му и никога не създава инстанция от Б директно
- реалните връзки се правят чрез конфигурация, а не в кода
- връзките се правят от контейнера → DI или IoC
- позволява лесна подмяна на имплементацията



Dependency Injection vs. JNDI lookup



Dependency Injection vs. JNDI lookup

- създаване чрез конструктор:
```
BidManager manager = new BidManagerBean();
```
- взимане чрез JNDI:
```
InitialContext context = new InitialContext();
BidManager manager =
(BidManager)context.lookup("java:comp/env/BidMana
ger");
```
- получаване чрез DI:
```
@WebServlet("/actions/add-bid")
public class AddBidServlet extends HttpServlet {
  @EJB
  private BidManager placeBid;
  …
}
```

Dependency Injection анотации

- анотациите могат да се дефинират на върху член променливи (field), или върху setter методи (property)

- видове:
  - `@Resource` – за инжектиране на JDBC, JMS, имейл, таймер услуга, EJBContext, environment променливи или други ресурси
  - `@EJB (@EJBs)` – за инжектиране на други bean-ове
  - `@WebServiceRef (@WebServiceRefs)` – за инжектиране на уеб услуга

Сигурност (security)

- позволява декларативно да се зададе *authentication* (верифициране на самоличността) и *authorization* (определяне на правата) на потребител
- дефинират се роли (roles) и се задава достъпа им до определена функционалност
- ролите се мапват на групи и потребители на сървъра при деплой и конфигурация
- възможно е да се правят проверки в кода вместо декларативно

Метафора за сигурност

пример: охраната на различни събития има следните security опции при допускане на цивилни:

- да се допускат всички (вход в универсален магазин)
- да не допуска никой (вход в казарма)
- да допуска потребителите в даден списък (вход в посолство – само ако си в списъка за деня)

- при дадено извикване на метод, може потребителят да се представи за друг (*Бонд, Джеймс Бонд…*)

Security пример

```
@DeclareRoles("BIDDER", "ADMIN")
@Stateless
public class BidManagerBean implements
BidManager {

    @RolesAllowed("BIDDER")
    public void addBid(...) {...}

    @RolesAllowed("ADMIN")
    public void removeBid(...) {...}

    @PermitAll
    public List<Bid> getBids(Item item) {...}
}
```

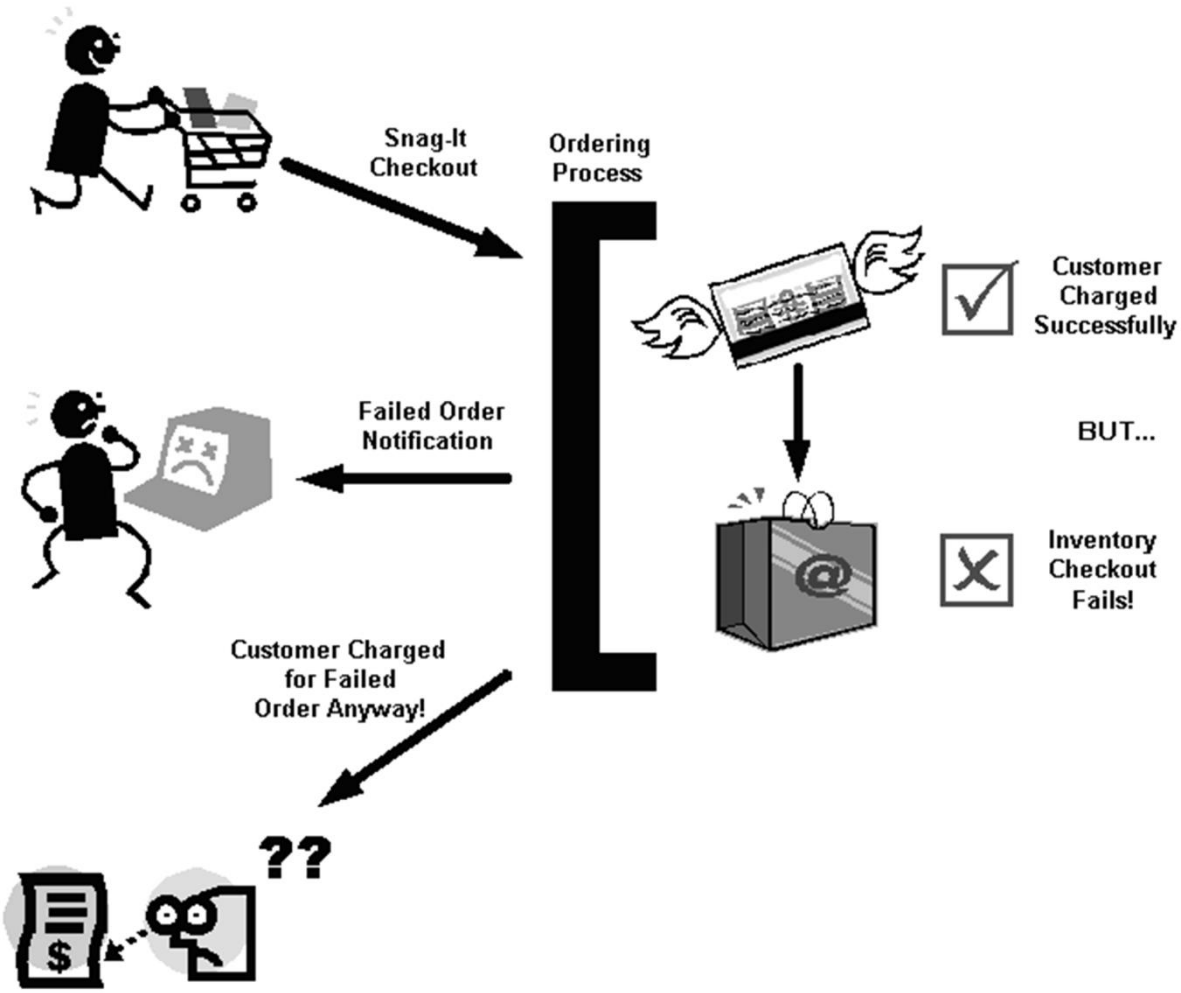
Security анотации

- `@DeclareRoles` – задава списък от роли известни за приложението
- `@PermitAll` – дава достъп на всички роли в приложението до даден метод
- `@DenyAll` – забранява достъпа на всички роли в приложението до даден метод
- `@RolesAllowed` – задава конкретни роли имащи достъп до даден метод
- `@RunAs` – позволява промяна на ролята на текущия потребител

Транзакции (transaction)

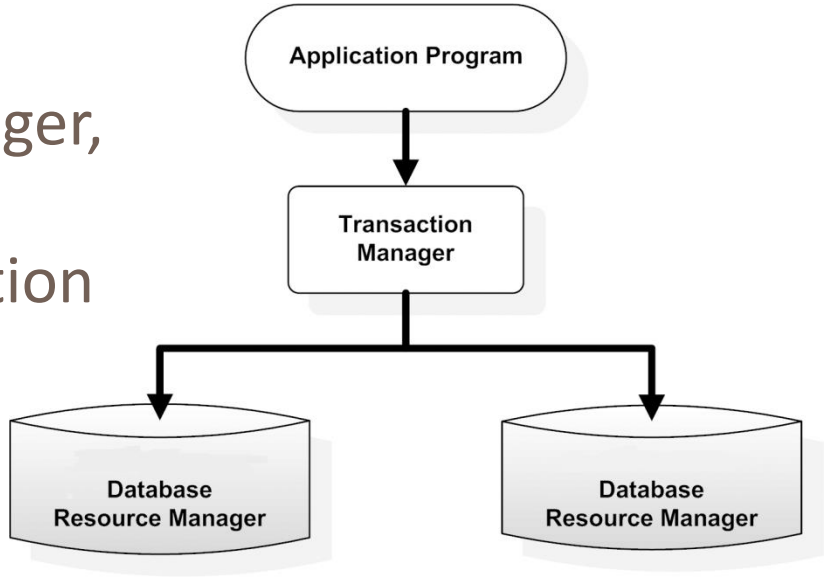
- групират набор от операции и гарантират всичко или нищо (например местене на пари между сметки)
- спазват ACID принципите
- могат да обхващат една система или няколко транзакционални системи
- разчита на JTA (Java Transaction API)

Транзакции – практически пример



Видове транзакции

- спрямо обхват:
  - локални (един ресурс и manager, single commit)
  - глобални (N ресурси, Transaction Manager + N * Resource Managers, 2PC)



- спрямо контрол:
  - CMT – транзакции управлявани от контейнера (чрез декларирани анотации)
  - BMT – транзакции управлявани от потребителя (чрез `UserTransaction;ut.begin(); ut.commit(); ut.rollback()`)

Анотации за CMT транзакции

- `@TransactionManagement` – задава дали транзакцията е CMT или BMT
- `@TransactionAttribute` – декларира се за отделните методи на bean-а; видове:
  - REQUIRED* – използва се съществуваща или създава нова транзакция
  - REQUIRES_NEW – създава се нова транзакция (ако вече има транзакция, тя се спира временно)
  - SUPPORTS – ако има транзакция се използва, иначе не
  - NOT_SUPPORTED – ако има транзакция се спира временно
  - MANDATORY – задължително трябва да има съществуваща транзакция (но не се създава нова)
  - NEVER – задължително трябва да няма транзакция

Метафора за CMT транзакции

- идея:
  - **транзакцията** е **фотоапарат**
  - **промените** в транзакцията са натрупаните **снимки**
  - **локалната** транзакция е **картичка**
  - **бизнес методите** където искаме да я използваме са различни **места/мероприятия**, които посещаваме
  - **настъпването на грешка** (rollback) е **унищожаване на фотоапарата** (и загуба на всички снимки/промени)

- случаи:
  - ***required*** → *туристически обект:* ако имаш фотоапарат го използваш, ако нямаш си купуваш картичка

Метафора за CMT транзакция

- ***requires_new*** → *музей:* трябва да прибереш фотоапарата и да си купиш картичка
- ***supports*** → *разходка по улиците на града:* ако имаш фотоапарат правиш снимки, а ако нямаш не правиш (не се продават картички)
- ***not_supported*** → *музикален концерт*: оставяш фотоапарата навън и след това си го взимаш обратно
- ***mandatory*** → *за да се включиш в клуба на фотографите*: трябва предварително да си се сдобил с фотоа
- ***never*** → *в квартала на червените фенери*: **не** трябва да носиш фотоапарат/снимаш, иначе идва един "чичко" и то казва да го прибереш

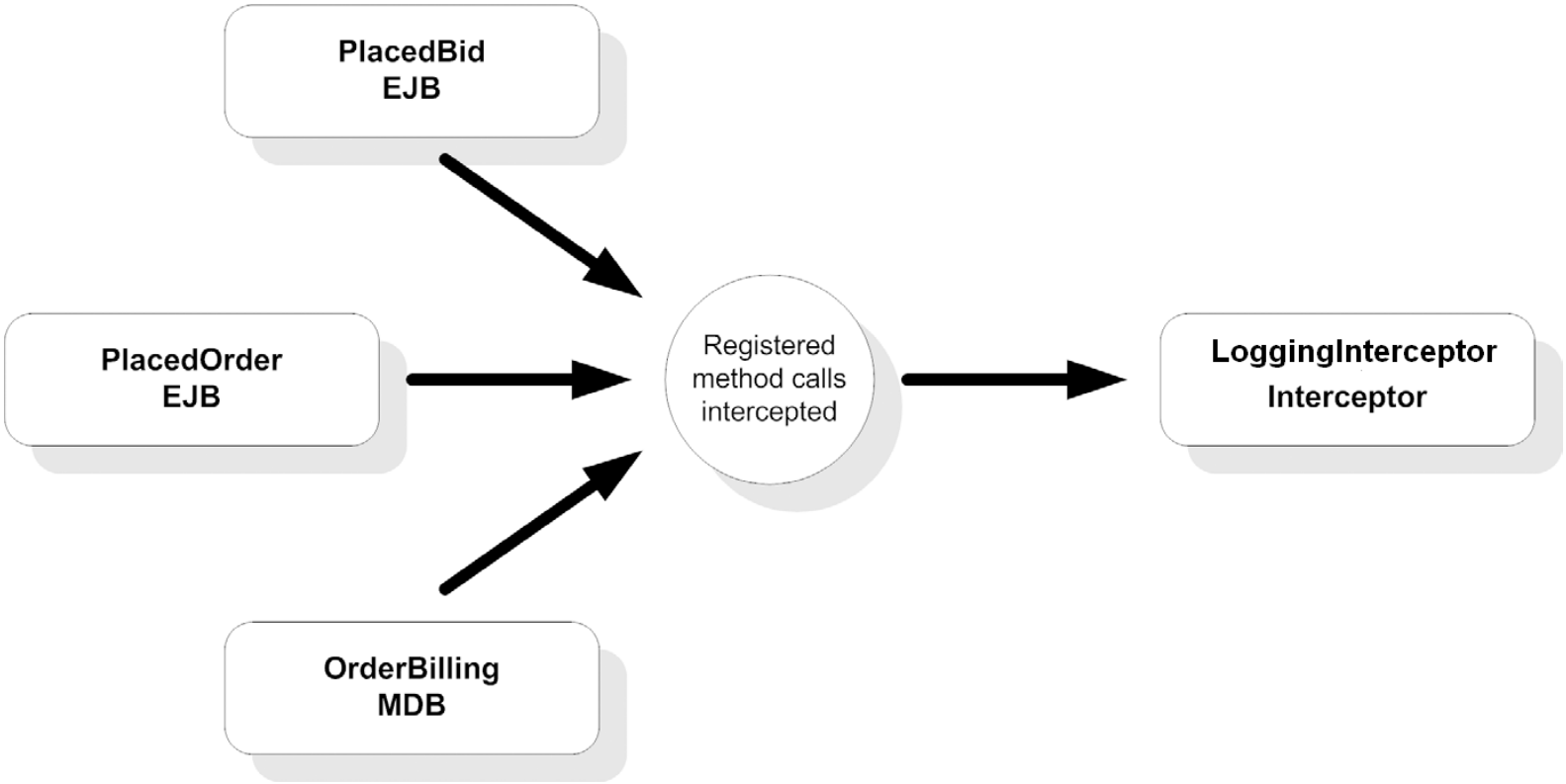
Транзакции - пример

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class ItemManagerBean implements ItemManager {

        @TransactionAttribute(TransactionAttributeType.REQUIRED)
        public Item addItem(String title, ...) throws
ItemCreationException {
                ...
        }

        public void deleteItem(Long itemId) { ...}


        @TransactionAttribute(TransactionAttributeType.SUPPORTS)
        public List<Item> getItems(String userId) {...}
}
```

Прихващане на съобщения (interceptor)

- interceptor-ите предлагат опростени AOP възможности за EJB-та
- прихващат извикването на бизнес методи
- позволяват групирането на обща логика срещана в много класове (cross-cutting concerns)
- често се използва се за по-прости задачи – logging, tracing и auditing

Прихващане на съобщения



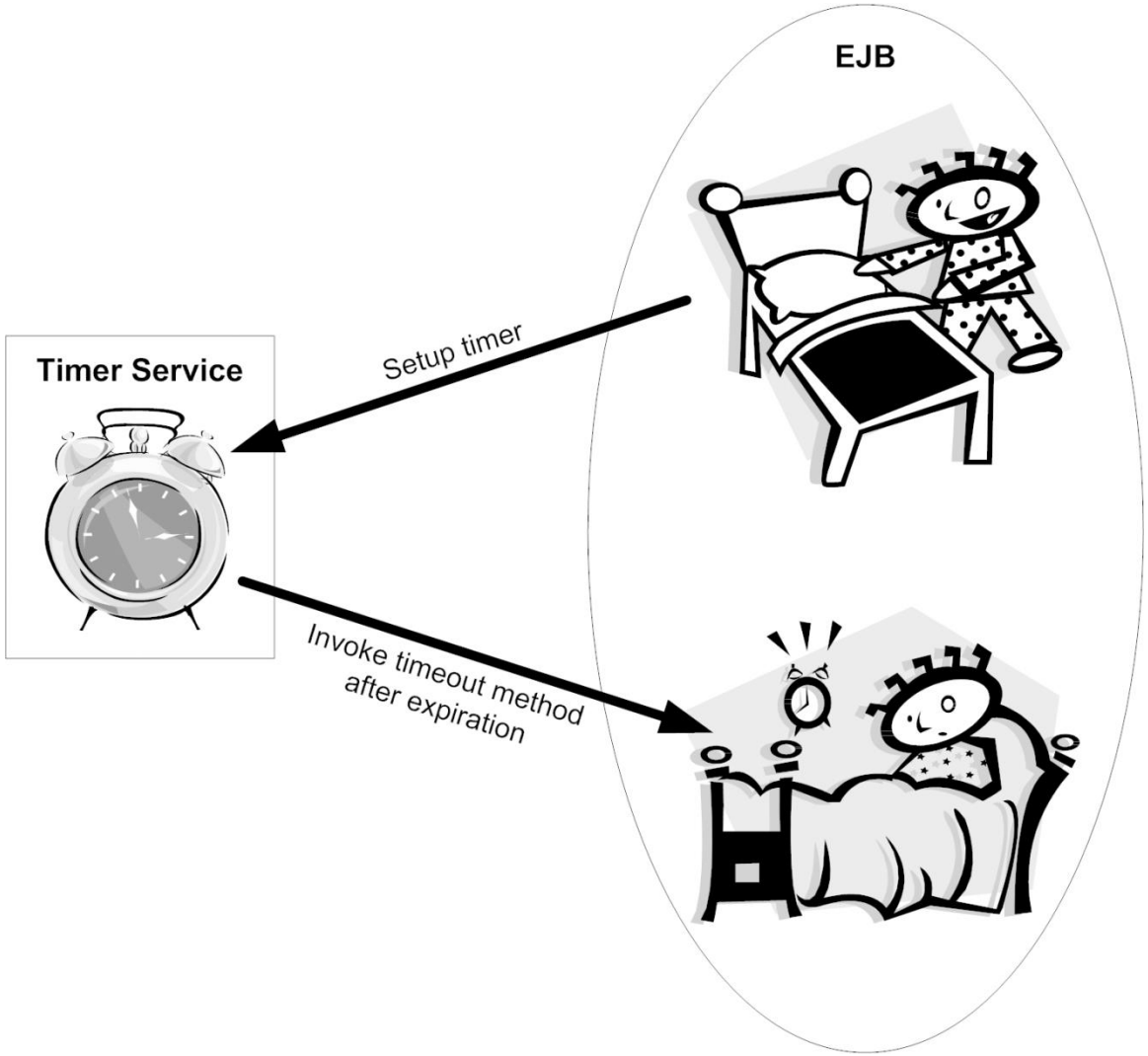
Прихващане на съобщения - пример

```
@Stateless
public class BidManagerBean implements BidManager {
      ...
      @Interceptors(LoggingInterceptor.class)
      public void addBid(String bidderId, Long itemId, ...) {
      ...
      }
}

public class LoggingInterceptor {
      @AroundInvoke
      public Object logMethodEntry(InvocationContext
invocationContext) throws Exception {
            System.out.println("Entering method: " +
invocationContext.getMethod().getName());
            return invocationContext.proceed();
      }
}
```

Задаване на график (scheduling)

- таймери позволяващи изпълнението на периодични задачи или задачи за определен момент в бъдещето
- поддържат се само в stateless и message-driven bean-ове
- използват `@Timeout` и `@Schedule` анотациите
- поддържат транзакции (ако някоя от задачите пропадне всичко се rollback-ва)

Принцип на работа на таймер



Таймер - пример

```
@Stateless
public class ItemManagerBean implements ItemManager {

  @Resource
  private TimerService timerService;

  public Item addItem(String title, Date bidEndDate...) {
    Item item = createItem(title, initialPrice, bidEndDate, description);
    validateItem(item);
    ...
    timerService.createTimer(bidEndDate, item.getItemId());
    return item;
  }

  @Timeout
  private void bidExpired(Timer timer) {
    Long itemId = (Long) timer.getInfo();
    // determine winner
  }

}
```

```
//декларативен cron синтаксис
@Schedule(dayOfWeek = "0-5")
public void sendNewsletter() {…}
```

Демонстрация

Предимства на EJB

- лесна за използване технология
- интегрирана с други Java технологии (JMS, RMI, JSF, JAAS, и други)
- open source стандарт
- широка поддръжка от производители
- стабилна и зряла технология
- поддръжка на сложни възможности като clustering, load balancing и failover

Out of scope

- XML синтаксис за конфигуриране на EJB-та
- Особености на EJB: ejb object, home, EJBContext, Asynchronous извиквания, scheduling синтаксис, използване на EJB 2.1
- Отдалечен достъп: RMI и уеб услуги
- JNDI = Java Naming and Directory Interface

Въпроси

Благодаря за вниманието