Introduction to Java EE

Java EE based 3-tier System Design Recommendations
(using JSF, EJB, JPA and CDI)
3-tier System Design Recommendations based on Java EE
3-tier Multilayer Reference Architecture based on Java EE

- **Client (Browser)**
- **Application Server**
  - Presentation: **JSF**
  - Business Logic: **EJB + CDI**
  - Data Access: **EJB + JPA**
- **Internet**
- **DBMS**
  - Tables/Triggers/Stored procedures
Java EE Practical Questions

- How many EJB components there should be?
- How granular EJB components should be?
- What type (stateless/stateful) EJB components should have?
- What type of Persistence Context should be used?
- How many EJB components should use a single web page?
- Should we create a separate EJB component for each web page?
Restriction Example: Data Access Tier (Java EE and .NET)

- EntityManager is not thread-safe
  - .Net EntityFramework ObjectContext is not thread-safe too
- One entity cannot be managed/shared by two EntityManagers
  - The same holds for .Net ObjectContext
- One relationship cannot be managed/shared by two EntityManagers
  - The same holds for .Net ObjectContext
- But all Web Applications ARE multi-threaded

1 Entity Framework FAQ: ObjectContext
Restriction Example: Data Access Tier (Java EE and .NET)

- You'll have problems, if in your system:
  - Two EntityManagers try to access one and the same Entity object:
    - For example, one EntityManager has loaded an entity, other tries to update it
  - One end of relationship between two entities is being managed by one EntityManager, other – by different EntityManager
    - For example: list of students is loaded by one EntityManager, but courses are being assigned to students by different EntityManager
  - Problems of these kinds occur when JSF pages belonging to a single use case use different EJB components
    - Approach “each JSF page must have dedicated EJB component” is problematic
Java EE Capabilities

- `@Stateless/@Stateful` – many instances, thread/memory/transaction management
- `@Singleton` – one instance, transaction management
- `@RequestScoped, ... , @ApplicationScoped` – life-cycle management
- `@TransactionAttribute` – transaction management
- `@PersistenceContext`:
  - `PersistenContextType` – short-term or long-term data cache
  - `SynchronizationType` – automatic or manual flushing to DB
- `@Version` – optimistic locking
- ... many others
2. Use Case Controller: Request Type Use Case - Annotations

- For stateful systems (state is being managed on server):
  - @Named
  - @Stateful
  - @RequestScoped
  - @PersistenceContext(type=EXTENDED, synchronization=SYNCHRONIZED)

- For REST type systems (no state management on server):
  - @Named
  - @Stateless
  - Use default @PersistenceContext (transactional and synchronized)
2. Request type Collaboration example
2. Use Case Controller: Conversation Type Use Case - Annotations

- For stateful systems (state is being managed on server):
  - @Named
  - @ConversationScoped
  - @Stateful
  - @PersistenceContext(type=EXTENDED, synchronization=UNSYNCHRONIZED)

- For REST type systems (no state management on server):
  - Not applicable, because conversation must be managed on client side!
    - Annotations the same as for request type use case
2. Conversation type Collaboration

Example

Diagram of University Management System with collaboration between User and Student.
3. Kinds of Requirements
3. Apply Design Principles: SoC and Cohesion with CDI + EJB

- **Use Case Controller** – one per use case:
  - Orchestrates the execution of sunny day/rainy day scenarios
  - Centralizes Persistence Context (cache) and Transaction management
  - => Stateful Component (CDI + EJB)

- **Localizable functional requirements**:
  - => Service (Stateless) Components (CDI + EJB)

- **Cross-cutting functional requirements**:
  - *Not-impacting* cross-cut functionality (Observer Pattern):
    - => Event-listening Components (CDI + EJB)
  - *Impacting* cross-cut functionality (**Decorator Pattern**):
    - => CDI Decorators

- **Cross-cutting non-functional requirements** (**Interceptor Pattern**):
  - => CDI Interceptors
3. Apply Design Principles: Asynchronous Communication (EJB)

- EJB (and CDI 2.0) provide means for asynchronous communication:

```java
@Asynchronous
public Future<String> longComputation() { ... }
```
3. Apply Design Principles: Java EE Capabilities/Restrictions

• Obey these rules:
  • Resulting business component tree must satisfy rules of Persistence Context propagation (see next slide)
  • For Service Components:
    • do not annotate them with @Named to prevent them to be used directly from web pages
    • prefer @Stateless with transactional unsynchronized @PersistenceContext
3. Persistence Context Propagation

Client \rightarrow Comp1

Comp1 \rightarrow Comp2
Comp1 \rightarrow Comp3

Comp2 \rightarrow Comp5
Comp2 \rightarrow Comp3

Comp5 \rightarrow Comp6
Comp5 \rightarrow Comp4

R \rightarrow Required
RN \rightarrow RequiresNew

Transaction and Persistence Context

Java EE based System Design Recommendations 16
3. Persistence context propagation rules (1)

- Let k1 and k2 be EJB component instances; k1 injects k2 (with the help of @EJB or @Inject)

- If both components have declared @PersistenceContext:
  - Transactional → Transactional ⚫, propagates
  - Transactional → Extended ⫻, error
  - Extended → Transactional ⚫, propagates
  - Extended →_{new} Extended ⚫, inherits
    - k1 injects k2 with the help of @Inject and k2 is a newly born instance
  - Extended →_{old} Extended ⫻, error
    - k1 injects k2 with the help of @Inject, but k2 was born long before k1 and already has its own EntityManager object
3. Persistence context propagation rules (2)

- Let k1 and k2 be EJB component instances, k1 injects k2 (with the help of @EJB or @Inject)
- If both components have declared @PersistenceContext:
  - Unsynchronized → Unsynchronized 🧑‍💻, propagates
  - Synchronized → Synchronized 🧑‍💻, propagates
  - Synchronized → Unsynchronized 🧑‍💻, propagates
  - Unsynchronized → Synchronized 😞, error
Example 1: Request Type Use Case
Example 1: Request Type Use Case

```java
@Named @RequestScoped @Stateful
class UseCaseController {
    @PersistenceContext(type=EXTENDED,
                        synchronization=SYNCHRONIZED)
    private EntityManager em;

    @Inject
    private Service1 service1;
    @Inject
    private Service2 service2;

    public String formProcessingMethod() {
        service1.someMethod(); // em gets propagated
        service2.someMethod(); // em gets propagated
    }
}
```
Example 1: Service Component

```java
@Stateless
public class Service1 {
    @Inject
    private StudentCRUD studentCRUD;

    public void someMethod(...) {
        ... 
        studentCRUD.createStudent(s);
        ...
    }
}
```
Example 1: Data Access Component

@Stateless
public class StudentCRUD {
  @PersistenceContext(type=TRANSACTIONAL,
                      synchronization=UNSYNCHRONIZED)
  private EntityManager em;

  public void createStudent(Student s) {
    em.persist(s);
  }
}

• This is an example of reusable Data Access Component:
  • It may be reused in other use case implementations
    • no matter the type of use case!
  • In case of accidental direct use by web tier client – no changes to DB would be done!
    • It may be protected from such a use by using transaction attribute MANDATORY
Example 2: Conversation Use Case

- **Use Case – Register New Student**

- **Steps:**
  1. Fill student form
  2. Fill study journal form
  3. Confirm entered data
  4. Observe outcomes: success or failure

- This is just an example: actually this use-case could be implemented with fewer steps.
Example 2 – Use Case Controller (1)

@Named @ConversationScoped @Stateful
public class RegisterNewStudentController {
  @PersistenceContext(type=PersistenceContextType.EXTENDED,
                      synchronization=SynchronizationType.UNSYNCHRONIZED)
  private EntityManager em;

  @Inject
  private Conversation conversation;

  //--- State being manipulated during the conversation
  // student and course are bound to HTML forms.
  private Student student = new Student();
  public Student getStudent() { return student; }

  private Course course = new Course();
  public Course getCourse() { return course; }

  //--- END of state
  ...

public void createStudent() {
    conversation.begin();
    em.persist(student); // changes are not flushed to DB!
}

public void createCourse() {
    em.persist(course); // changes are not flushed!
}

public void end() {
    try {
        course.getStudentList().add(student);
        student.getCourseList().add(course);
        em.flush(); // try to catch DB problems
        em.joinTransaction();
        conversation.end();
    } catch (OptimisticLockException ole) {
        // handle optimistic locking conflicts
    }
}
Example 2 - Discussion

- This is just an example: relationships can be managed in any step of the conversation

- You are allowed to call:
  - `em.persist(...);`
  - `em.merge(...);`
  - `em.remove(...);`
  - `em.refresh(...);`
  - `em.find(...) and em.createQuery(...);`

  Changes to entities are being recorded within the cache of EntityManager, and are flushed to the database as soon as `em.joinTransaction()` is called.

- But you **cannot** call (TransactionRequiredException):
  - `em.flush(...);`
  - `em.lock(...);`
Example 3: Conversation Use-Case with Several Components
@Named @ConversationScoped @Stateful
class RegisterNewStudentController {
  @PersistenceContext(type=PersistenceContextType.EXTENDED,
                      synchronization=SynchronizationType.UNSYNCHRONIZED)
  private EntityManager em;
  private Conversation conversation;

  private Student student = new Student();
  public Student getStudent() { return student; }

  private Course course = new Course();
  public Course getCourse() { return course; }

  @Inject private StudentCRUD studentCRUD; // service component
  public void createStudent() {
    conversation.begin();
    studentCRUD.persistStudent(student);
  }

  ...
}