

Java EEとアノテーション

日本Javaユーザグループ
河村 嘉之

1

本日のお題

- このセミナーの前半で紹介されたアノテーションが、実際にどのように使われているかを、Java EE 5 の重要なコンポーネントであるEJB3を中心にサーバサイドのコンポーネントを例に解説していきます。

2

EJB3.0

EJB3.0

- JSR-220
Enterprise JavaBeans 3.0

- Spec Lead

- Linda DeMichiel (Sun Microsystems, Inc.)
- Michael Keith (Oracle)

- Expert Group

- IBM, BEA, Apache Foundation, JBossなど

- Java EE 5でのEJBの仕様

- Java EEでの標準コンポーネント技術



4

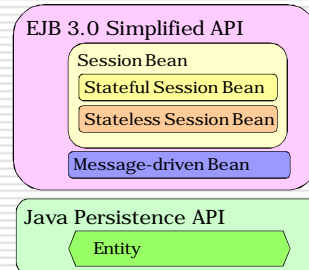
EJB3.0

- JSR220

- ドキュメントは3部構成
- EJB Core Contracts and Requirements (562ページ)
 - EJBコンテナに必要なことなどの定義
- Java Persistence API (256ページ)
 - 新しいPOJO Persistence API
 - EJBコンテナ外でも動作可能
- EJB 3.0 Simplified API (59ページ)
 - EJB開発の簡単化

5

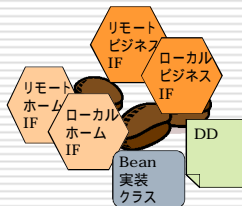
EJB3.0 構成



6

EJBの構成要素

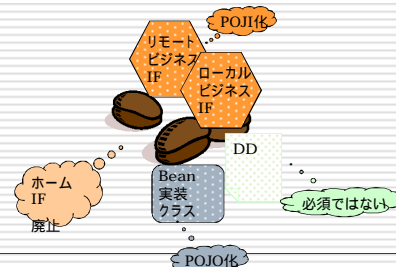
□ EJB2.1以前 Session Bean



7

EJBの構成要素

□ EJB3.0 Session Bean



8

EJB 3.0 Session Bean (Bean実装クラス)

```
package sample.ejb;

import java.util.ArrayList;
import java.util.List;

import javax.ejb.Stateless;

import sample.entity.Account;

@Stateless
public class AccountServiceBean implements AccountService {
    public boolean register(Account account) {
        return true;
    }

    public List<Account> find(String loginname) {
        return new ArrayList<Account>();
    }
}
```

AccountServiceBean.java

Stateless Session Bean

□ @Statelessアノテーション

- そのBeanがStateless Session Beanであることを示す。
- name属性: Beanの名前 ejb-jar内で一意でなければならない
- mappedName属性: このBeanをマッピングする製品固有の名前を指定する。(JNDI名?)
- description属性: このBeanの説明

10

Session Bean Lifecycle Callbacks

- EJB3.0では、ejbCreateのようなコールバックメソッドを作成する必要がなくなった。
- 代わりに、コールバックメソッドが必要なときは、以下のようなアノテーションをつけたメソッドを作成する。
 - @PostConstruct - EJB作成後に呼ばれる
 - @PreDestroy - EJB破棄前に呼ばれる
 - @PostActivate - EJB活性化後に呼ばれる (Statefulのみ)
 - @PrePassivate - EJB非活性化前に呼ばれる (Statefulのみ)

11

EJB 3.0 Session Bean (リモートインターフェース)

```
package sample.ejb;

import java.util.List;
import javax.ejb.Remote;
import sample.entity.Account;

@Remote
public interface AccountService {
    public abstract boolean register(Account account);
    public abstract List<Account> find(String loginname);
}
```

AccountService.java

EJB 3.0 Session Bean (ローカルインターフェース)

```
package sample.ejb;

import java.util.List;
import javax.ejb.Local;
import sample.entity.Account;

@Local
public interface AccountLocalService {
    public abstract boolean register(Account account);
    public abstract List<Account> find(String loginname);
}
```

AccountLocalService.java

Stateful Session Bean

```
package sample.ejb;

import java.util.ArrayList;
import java.util.List;

import javax.ejb.Stateful;

import sample.entity.Account;

@Stateful
public class SearchServiceBean implements SearchService {
    private List<Account> searchResult;
    .....
}
```

14

Stateful Session Bean (1/2)

- @Statefulアノテーション
 - そのBeanがStateless Session Beanであることを示す。
 - name属性: Beanの名前 ejb-jar内で一意でなければならない
 - mappedName属性: このBeanをマッピングする製品固有の名前を指定する。(JNDI名?)
 - description属性: このBeanの説明

15

Stateful Session Bean (2/2)

- @Removeアノテーション
 - このStateful Session Beanを破棄する際に呼ばれるメソッドを指定する。
 - @PreDestoryを指定したメソッドが存在する場合は先にそれが呼び出されてから、@Removeメソッドを付与したメソッドが呼ばれる。
 - retainIfException属性: メソッドがアプリケーション例外を発生したとき、SFSBを削除しないことを指定する。デフォルトはfalse

16

Message-driven Bean

```
package sample.ejb;

import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        .....
    }
}
```

17

Message-driven Bean

- @MessageDrivenアノテーション
 - そのBeanがMessage-Driven Beanであることを示す。
 - name属性: Beanの名前 ejb-jar内で一意でなければならない
 - mappedName属性: このBeanをマッピングする製品固有の名前を指定する。(JNDI名?)
 - messageListenerInterface属性: Beanがインターフェースを実装していない場合や複数のインターフェースを実装しているときに、MessageListenerインターフェースを指定する。
 - description属性: このBeanの説明

18

Deployment Descriptor

無

ただし、Optional

19

Dependency Injection

□ 従来のJ2EE環境

- リソース、EJBなどを利用するとき、JNDIレポジトリを検索して、対象のオブジェクトを取得する必要があった。



□ Java EE 5以降

- 必要なリソースは、コンテナが自動的に検索し、Beanに注入する。

20

Simplified API Dependency Injection

□ @EJB

- EJBへの参照
- 変数の名前と型から注入するBeanをコンテナが判別する。
- このルールに合わない場合は、name属性でJNDI名、beanInterface属性でそのBeanのインターフェースを指定する

□ @PersistenceContext

- PUへの参照
- name属性で特定のPUを指定する

21

Simplified API Dependency Injection

□ @Resource

- リソースへの参照 (データソース、トランザクションオブジェクトなど)
- JNDIにバインドされているオブジェクトを注入する
- 変数の名前と型から注入するオブジェクトをコンテナが判別する。(JSR-250 Common Annotations API)
- このルールに合わない場合は、name/mappedName属性でJNDI名、type属性でそのオブジェクトの型を指定する

22

Dependency Injection

```
package sample.ejb;

@Stateless
public class AccountServiceBean implements AccountService {
    @PersistenceContext
    private EntityManager em;
    .....
}
```

AccountServiceBean.java (一部)

Persistence Unit

- Persistence Unitとは、以下を含む論理的なグループ
 - Entity Manager FactoryとEntity Managerおよびその設定情報
 - Entity Managerで管理されるクラス
 - クラスとDBをマッピングするメタデータ (アノテーションもしくはXML)
- Java EE環境では、EJB-Jar、War、Ear、アプリケーションクライアントJarでPUを0個以上定義することができる。
- PUは名前を持たなければいけない。
 - PUの名前は、アプリケーションで一意的でなければいけない

24

Persistence Unit

- PUは、persistence.xmlファイルによって定義される。
- persistence.xmlファイルは、クラスパス上のMETA-INFディレクトリに置かれる。
 - EJB-Jarファイル
 - WarファイルのWEB-INF/classesディレクトリ
 - WarファイルのWEB-INF/lib内のjarファイル
 - Earファイルのルートもしくはlibディレクトリ内のjarファイル
 - アプリケーションクライアントjarファイル

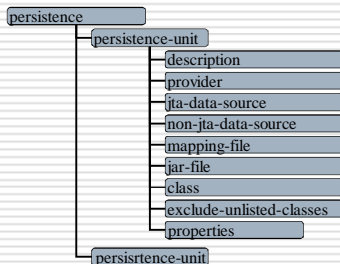
25

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  version="1.0">
  <persistence-unit name="em1">
    <jta-data-source>jdbc/SampleDB</jta-data-source>
    <properties>
      <property name="toplink.platform.class.name"
        value="oracle.toplink.essentials.platform.database.DerbyPlatform"/>
      <property name="toplink.ddl-generation"
        value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

persistence.xml

persistence.xml



27

persistence.xml

- トップレベルエレメントはpersistence、persistenceエレメントは複数のpersistence-unitエレメントで構成される。
- persistence-unitは、name属性とtransaction-type属性を持つ。
 - name属性は、PUを特定するために用いられる一意の名前である
 - transaction-type属性は、値としてJTAもしくはRESOURCE_LOCALをとる。
 - JTAを指定するとJTAデータソースが与えられるものとする。Java EE環境でのデフォルト
 - RESOURCE_LOCALを指定すると、非JTAデータソースが与えられるものとする。Java SE環境でのデフォルト

28

persistence.xml

- persistence-unitの子エレメント
 - description
このPUの情報
 - provider
永続プロバイダーのクラス
 - jta-data-source, non-jta-data-source
永続化に利用するデータソースのJNDI名を指定。
 - mapping-file, jar-file, class, exclude-unlisted-classes
オブジェクトとデータベースのマッピングを記述したファイル、PUに含まれるjarやPUに含まれる/含まれないクラスを定義する。Java SE環境では、これらを特定する必要がある。
 - properties
永続プロバイダーに特化した設定情報を指定。
 - Hibernate: Reference Manual Chap.3 SessionFactory Configuration
http://www.hibernate.org/hib_docs/reference/en/html/session-configuration.html
 - TopLink: TopLink JPA Extensions Reference
<http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-extensions.html#persistence-xml>

29

PersistenceContext

```
package sample.ejb;
.....
@Stateless
public class AccountServiceBean implements AccountService {
  @PersistenceContext
  private EntityManager em;
  .....
  public boolean register(Account account) {
    try {
      em.persist(account);
      return true;
    } catch (Exception ex) {
      ex.printStackTrace();
      return false;
    }
  }
}
AccountServiceBean.java (一部)
```

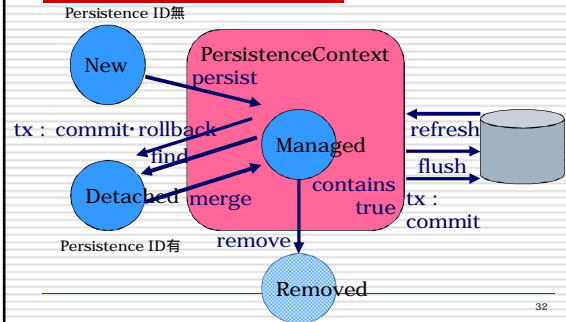
EntityManager

- EntityManager
 - PUに関連付けられた、Entityを管理するためのオブジェクト
- EntityManager API
 - persist() - EntityをDBに格納する*
 - remove() - EntityをDBから削除する*
 - refresh() - EntityのデータをDBから再取得する*
 - merge() - DetachされたEntityの情報とDBの情報を同期する*
 - find() - 主キーでEntityを取得する
 - createQuery() - Entityを取得する質問文を作成する
 - createNamedQuery() - 名前付けされた質問文を作成する
 - contains() - Entityが存在するかをチェックする
 - flush() - PersistenceContextにあるデータをDBに強制的に同期する。

*TransactionContext内で呼び出さなくてはいけない。

31

Entityインスタンスのライフサイクル



32

PersistenceContextの生存期間

- PersistenceContextの生存期間は2種類
 - Transaction-Scope (PersistenceContextType.TRANSACTION)
 - Extended-Scope (PersistenceContextType.EXTENDED)
 - type属性で指定

33

PersistenceContextの生存期間

- Transaction-Scope
 - トランザクションのスコープと同じ。トランザクション生成時に作成され、トランザクション終了時に消滅
 - デフォルト
 - 取得されたEntityは、トランザクション終了後にdetachedとなる。
- Extended-Scope
 - 複数のトランザクションにまたがるスコープ。トランザクションが終了してもPersistenceContextは消滅せず、closeを呼んだときに消滅する。
 - 取得されたEntityは、トランザクション終了後もPersistenceContextの管理下となる。

34

JPA QL API呼び出し

```
package sample.ejb;
.....
@Stateless
public class AccountServiceBean implements AccountService {
    @PersistenceContext
    private EntityManager em;
    .....
    public List<Account> find(String loginname) {
        Query query = em.createQuery(
            "SELECT a FROM Account AS a WHERE a.loginname = :loginname");
        query.setParameter("loginname", loginname);
        return query.getResultList();
    }
}
AccountServiceBean.java (一部)
```

JPA QL

- SQLに似た文法
- 検索文
 - SELECT句で検索対象のオブジェクトを指定する
 - オブジェクトや数値の集合を戻り値にとれる
 - SUM, AVG, COUNT, MAX, MINといった簡単な集約関数が利用可
 - FROM句でどのEntityを取得するか指定
 - Order AS のようにEntityの変数を指定する
 - INNER JOIN, OUTER JOINなどを指定することも可能
 - WHERE句で取得する条件を指定
 - PATH式でEntityの子-孫要素を指定することも可能
 - ?1のように位置パラメータで入力値を指定することも可能
 - 位置パラメータは、1始まり
 - :nameのように名前パラメータで入力値を指定することも可能
 - =, <, >, >=, <=, <> (Not Equal) で条件を比較
 - AND, OR, NOTが利用可能。INで集合を指定することもできる
 - LIKEでパターンマッチの指定が可能。可変部は%で指定

36

JPA QL

- 検索文(続き)
 - GROUP BY, HAVING, ORDER BYなども指定できる
- 削除および更新文
 - DELETE文、UPDATE文によってEntityの一括更新が可能
- 詳しい仕様は、Java Persistence API仕様4章に記載

37

Entity

- POJOで定義
 - 通常のJavaBeans
 - インスタンスフィールド(private/protected) + アクセスメソッド(public)
 - 各フィールドは、自動的に永続化
- ×finalなクラス、どのメソッド、永続化される変数もfinalではない
- Serializableインターフェースを実装
- 引数を持たないコンストラクターが必要(public/protected)
- クラスレベルの@Entityアノテーションを定義

38

Entity

```
package sample.entity;
.....
@Entity
public class Account
    implements Serializable {
    private String loginname;
    private String password;

    public Account() {
    }

    public Account(String loginname,
                    String password) {
        this.loginname = loginname;
        this.password = password;
    }

    @Id
    public String getLoginname() {
        return loginname;
    }
    public void setLoginname(
        String loginname) {
        this.loginname = loginname;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(
        String password) {
        this.password = password;
    }
}
```

Account.java

39

Entity 高度な設定

- データベーステーブルとの関連付け
 - デフォルトではEntityと同じ名前
 - @Tableアノテーションを用いて設定することができる
 - name: テーブル名
 - schema: スキーマ名
 - catalog: カタログ名
 - uniqueConstraints: + ユニーク制約 (テーブルDDL生成時に利用される)

```
@Entity
@Table(
    name="UserAccount"
    uniqueConstraints=
        @UniqueConstraint(columnNames={"loginname"}))
public class Account implements Serializable {
.....
}
```

40

Entity 高度な設定

- 主キーフィールドの指定
 - @Idアノテーションを用いてどのフィールドが主キーかを指定

```
@Entity
public class Account implements Serializable {
    private String loginname;
    @Id
    public void setLoginname(String loginname) {
        this.loginname = loginname
    }
    public void getLoginname() {
        return loginname
    }
    .....
}
```

41

Entity 高度な設定

- 主キーの自動生成
 - @GeneratedValueアノテーションを利用
 - strategy: キーの生成方法を指定。デフォルトはAUTO
 - TABLE (データベーステーブルを用いてユニークキーを生成)
 - SEQUENCE, IDENTITY (データベースのSequence, Identityカラムを利用)
 - AUTO (Persistence Providerが自動的に方式をピックアップ)
 - generator: SEQUENCE, TABLEを指定した場合にKey Generatorの名前を指定する。デフォルトは、Persistence Providerが提供するものを利用する)

```
@Entity
public class Account implements Serializable {
    private long accountId;
    @Id
    @GeneratedValue(
        strategy=GenerationType.AUTO
    )
    public void setAccountId(long accountId) {
        this.accountId = accountId;
    }
    .....
}
```

42

Entity 高度な設定

- 複合キー
 - @IdClassアノテーションで主キークラスを指定

```
@Entity
@IdClass(sample.entity.AccountPK.class)
public class Account implements Serializable {
    @Id String firstname;
    @Id String lastname;
    .....
}
```

43

Entity 高度な設定

- カラムの関連付けの高度な設定
 - @Columnアノテーションを利用
 - name: 対象となるカラム名 デフォルトはフィールド名
 - unique: ユニーク制約 デフォルトはfalse
 - nullable: null制約 デフォルトはtrue
 - insertable: 挿入可能か デフォルトはtrue
 - updatable: 更新可能か デフォルトはtrue
 - columnDefinition: DDLのフラグメント
 - length: カラムの長さ デフォルトは255

```
@Entity
public class Account implements Serializable {
    private String password;
    @Column(name="SECRET_KEY", nullable=false)
    public void setPassword(String password) {
        this.password = password;
    }
    .....
}
```

44

Entity 高度な設定

- Entityのバージョンを保持するフィールドを設定
 - @Versionアノテーションを利用
 - Optimistic Lockに利用

```
@Entity
public class Account implements Serializable {
    private int version;
    @Version
    public void setVersion(int version) {
        this.version = version;
    }
    .....
}
```

45

Entity 高度な設定

- 日付型を保持するフィールドを設定
 - @Temporalアノテーションを利用
 - フィールドの型がjava.util.Date, java.util.Calendarの時は必須
 - TemporalType: マッピングする型を指定
 - DATE: java.sql.Date
 - TIME: java.sql.Time
 - TIMESTAMP: java.sql.Timestamp

```
@Entity
public class Account implements Serializable {
    private Date birthday;
    @Temporal(TemporalType.DATE)
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
    .....
}
```

46

Entity 高度な設定

- 永続化しないフィールドを設定
 - @Transientアノテーションを利用

```
@Entity
public class Account implements Serializable {
    private int total;
    @Transient
    public void setTotal(int total) {
        this.total = total;
    }
    .....
}
```

47

Entity間の関連

- 以下の種類のEntity間の関連付けが可能
 - 1対1
 - 1対n / n対1
 - n対n
- 参照先が1の場合は、参照先のオブジェクト型のフィールドを作成。
- 参照先がnの場合は、Collection型のフィールドを作成。Genericsを利用する事も可能
- 生成・変更・削除処理をCascadeさせることもできる(CascadeType)

48

Entity間の関連 (1対1)

- 外部キーによる関連付け
- 参照元
 - 参照先のオブジェクト型のフィールドを定義
 - @OneToOneアノテーションを関連するフィールドに付ける
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
 - optional: true null可 false null不可
 - @JoinColumnアノテーションで外部キーの情報を設定する
 - name: 外部キーを保持するカラム
 - referencedColumnName: 参照先のカラム, デフォルトでは参照先の主キー
 - unique: 外部キーにユニーク制約を付けるか? デフォルトはfalse
 - nullable: 外部キーにnullを取れるか? デフォルトはtrue
 - insertable: 外部キーは挿入可能か? デフォルトはtrue
 - updatable: 外部キーは更新可能か? デフォルトはtrue
 - columnDefinition: DDL生成時のSQLのスニペット

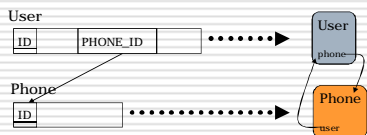
49

Entity間の関連 (1対1)

- 外部キーによる関連付け
- 参照先
 - 参照元のオブジェクト型のフィールドを定義
 - @OneToOneアノテーションを関連するフィールドに付ける
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
 - optional: true null可 false null不可
 - mappedBy: 参照元のフィールド名

50

Entity間の関連 (1対1)



51

Entity間の関連 (1対1)

外部キーによる関連付け

```
package phonebook.entity;

@Entity
public class User implements
    java.io.Serializable {
    .....
    @OneToOne
    @JoinColumn(
        name="PHONE_ID",
        unique=true, updatable=false
    )
    public Phone getPhone() {
    }

    public void setPhone(Phone phone) {
        this.phone = phone;
    }
}

package phonebook.entity;

@Entity
public class Phone implements
    java.io.Serializable {
    .....
    @OneToOne(mappedBy=phone)
    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }
}
```

52

Entity間の関連 (1対1)

- 同じ主キーを利用することによる関連付け
- 参照元
 - 参照先のオブジェクト型のフィールドを定義
 - @Idアノテーションで主キーフィールドを指定する
 - @OneToOneアノテーションを関連するフィールドに付ける
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
 - optional: true null可 false null不可
 - @PrimaryKeyJoinColumnアノテーションで同じ主キーを利用することを設定する
 - name: 外部キーを保持するカラム
 - referencedColumnName: 参照先のカラム, デフォルトでは参照先の主キー
 - columnDefinition: DDL生成時のSQLのスニペット

53

Entity間の関連 (1対1)

- 同じ主キーを利用することによる関連付け
- 参照先
 - 参照元のオブジェクト型のフィールドを定義
 - @Idで主キーフィールドを指定



54

Entity間の関連 (1対1)

主キーによる関連付け

```

package phonebook.entity;

@Entity
public class User
    implements java.io.Serializable {
    ....
    @Id long id;
    ....
    @OneToOne
    @PrimaryKeyJoinColumn
    public Phone getPhone() {
        return phone;
    }

    public void setPhone(Phone phone) {
        this.phone = phone;
    }
}
    
```

```

package phonebook.entity;

@Entity
public class Phone
    implements java.io.Serializable {
    ....
    @Id long id;
    ....
}
    
```

55

Entity間の関連 (1対n)

参照元

- Collection型のフィールドを作成する。(Genericsを利用して対象のオブジェクトの型を指定してもよい)
- @OneToManyアノテーションを関連するフィールドに付ける
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
 - mappedBy: 参照元のフィールド名

56

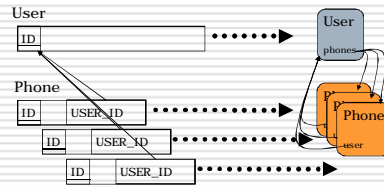
Entity間の関連 (1対n)

参照先

- 参照元オブジェクト型のフィールドを定義
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
 - optional: true null可 false null不可
- @JoinColumnアノテーションで外部キーの情報を設定する
 - name: 外部キーを保持するカラム
 - referencedColumnName: 参照先のカラム。デフォルトでは参照先の主キー
 - unique: 外部キーにユニーク制約を付けるか? デフォルトはfalse
 - nullable: 外部キーにnullを取れるか? デフォルトはtrue
 - insertable: 外部キーは挿入可能か? デフォルトはtrue
 - updatable: 外部キーは更新可能か? デフォルトはtrue
 - columnDefinition: DDL生成時のSQLのスニペット

57

Entity間の関連 (1対n)



58

Entity間の関連 (1対n)

1対n: 1側

```

package phonebook.entity;

@Entity
public class User
    implements java.io.Serializable {
    ....
    @OneToMany(
        cascade = CascadeType.ALL,
        fetch = FetchType.EAGER,
        mappedBy = "user"
    )
    public Collection<Phone> getPhones() {
        return phones;
    }

    public void
    setPhones(Collection<Phone> phones)
    {
        this.phones = phones;
    }
}
    
```

1対n: n側

```

package phonebook.entity;

@Entity
public class Phone
    implements java.io.Serializable {
    ....
    @ManyToOne
    @JoinColumn(name = "userId")
    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }
}
    
```

59

Entity間の関連 (n対n)

参照元

- Collection型のフィールドを定義(Genericsを用いて参照先のオブジェクトを特定することも可能)
- @ManyToManyアノテーションを関連するフィールドに付ける
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
- @JoinTableアノテーションで関連をマップするテーブルの情報を設定する
 - name: テーブル名
 - schema: スキーマ名
 - catalog: カタログ名

60

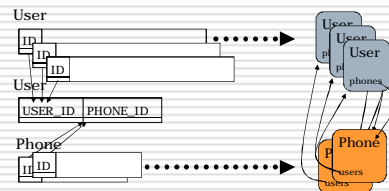
Entity間の関連 (n対n)

□ 参照先

- Collection型のフィールドを定義 (Genericsを用いて参照元のオブジェクトを特定することも可能)
- @ManyToManyアノテーションを関連するフィールドに付ける
 - cascade: 処理の伝播を設定する
 - fetch: 関連する項目の取得のタイミングを指定する
 - mappedBy: 参照元のフィールド名

61

Entity間の関連 (n対n)



62

Entity間の関連 (n対n)

```

package phonebook.entity;
@Entity
public class User implements
    java.io.Serializable {
    .....
    @ManyToMany
    @JoinTable(name="USER_PHONES")
    public Collection<Phone> getPhones() {
        return phones;
    }
    public void
    setPhones(Collection<Phone> phones)
    {
        this.phones = phones;
    }
}

package phonebook.entity;
@Entity
public class Phone implements
    java.io.Serializable {
    .....
    @ManyToMany(mappedBy="phones")
    public Collection<User> getUsers() {
        return users;
    }
    public void
    setUsers(Collection<User> users) {
        this.users = users;
    }
}
    
```

63

Entity Lifecycle Callbacks

- EJB3.0では、ejbCreateのようなコールバックメソッドを作成する必要がなくなった。
- 代わりに、コールバックメソッドが必要なときは、以下のようアノテーションをつけたメソッドを作成する。
 - @PrePersist - Entityを永続化する前に呼ばれる
 - @PostPersist - Entityを永続化した後に呼ばれる
 - @PreRemove - Entityを削除する前に呼ばれる
 - @PostRemove - Entityを削除した後に呼ばれる
 - @PreUpdate - Entityを更新する前に呼ばれる
 - @PostUpdate - Entityを更新した後に呼ばれる
 - @PostLoad - Entityをロードした後に呼ばれる

64

トランザクション

```

@Stateless
public class AccountServiceBean implements AccountService {
    .....
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public boolean register(Account account) {
        try {
            em.persist(account);
            return true;
        } catch (Exception ex) {
            ex.printStackTrace();
            return false;
        }
    }
    .....
}
AccountServiceBean.java
    
```

トランザクション設定項目

- javax.ejb.TransactionAttributeType
- デフォルトはREQUIRED

Type	トランザクション	
	実行中	なし
NOT_SUPPORTED	中断、メソッド終了後再開	何もしない
REQUIRED	現在のトランザクションに参加	新しいトランザクションを開始
SUPPORTS	現在のトランザクションに参加	何もしない
REQUIRES_NEW	現在のトランザクションを中断し、新しいトランザクションを開始	新しいトランザクションを開始
MANDATORY	現在のトランザクションに参加	例外を送出
NEVER	例外を送出	何もしない

66

Security

- EJBのセキュリティモデル
 - ロールモデルのセキュリティモデル
 - ロールに対して、プリンシパル(ユーザなど)を割り当てる。
 - ベンダ独自のDD
 - デプロイ時
 - メソッドへのアクセスをロールベースで許可・却下する。

- @RolesAllowed: 指定されたロールの実行を許可する
- @PermitAll: すべての呼び出しを許可する
- @DenyAll: すべての呼び出しを却下する
- @RunAs: 指定されたロールとしてメソッドを実行する

67

Security

```
@Stateless
public class PhoneBookBean implements PhoneBook {
    @PermitAll
    public User loadUser(long id) {
        return manager.find(User.class, id);
    }

    @RolesAllowed("admin")
    public void addUser(User user) { manager.persist(user);
    }
}
```

68

Interceptor

```
package sample.ejb;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class ProfileInterceptor {
    @AroundInvoke
    public Object profileMethod(InvocationContext inv)
        throws Exception {
        String name = inv.getTarget().getClass().getName();
        String methodName = inv.getMethod().getName();
        long start = System.currentTimeMillis();
        Object o = inv.proceed();
        long end = System.currentTimeMillis();
        System.out.println(name + "." + methodName + " took "
            + (end - start) + " ms.");
        return o;
    }
}
```

ProfileIterceptor.java

InvocationContext

- InvocationContext
AroundInvokeメソッドなどに渡されるコンテキスト情報
呼び出し先のメソッドの情報などにアクセスできる

メソッド一覧

- getContextData – コンテキスト情報を取得する
- getMethod – ターゲットメソッドを取得する。
- getParameters – メソッドに渡されるパラメータを取得する。
- getTarget – ターゲットのインスタンスを返す。
- proceed – ターゲットのメソッドを実行する。
- setParameters – ターゲットメソッドに渡すパラメータを変更する。

70

Interceptorを織り込むBean

```
package sample.ejb;

.....

@Stateless
@Interceptors({ ProfileInterceptor.class})
public class AccountServiceBean
    implements AccountService {
    .....
}
```

AccountServiceBean.java (一部)

Java EE 5におけるWebサービス

- JSR-181
Web Services Metadata for the Java Platform
 - バージョン2.0 (2005/2/27)
 - Spec Lead
 - Stuart Edmondston (BEA)
 - Brian Zotter (BEA)
 - Expert Group
 - BEA, IBM, Motorola, Nokia Corporation, Oracle, SAP, Sunなど
- アノテーションを用いて、Webサービスとして公開するメソッドを特定する。
- 前提となる仕様
 - JSR-109 Web Services for Java EE
 - JSR-224 JAX-WS
 - JSR-175 A Metadata Facility for Java Programming Language

72

Webサービス

```
package sample.ejb;
.....
@Stateless
@Interceptors({ProfileInterceptor.class})
@WebService
public class AccountServiceBean implements AccountService {
    @WebMethod
    public boolean register(Account account) {
        .....
    }
}
```

AccountServiceBean.java (一部)

Webサービス

- サービス実装Bean
 - publicかつfinal, abstractではないクラス
 - クラスレベルアノテーション@WebServiceを定義する
 - どのメソッドも@WebMethodアノテーションが宣言されない場合は、すべてのpublicなメソッドがサービスとして公開される
- エンドポイントインターフェース
 - エンドポイントインターフェースは必須ではない
 - クラスレベルで@WebServiceアノテーションを定義する。
 - portName, serviceName, endpointInterface属性は含んではいけない
 - java.rmi.Remoteインターフェースを継承する必要はない
- Webメソッド
 - publicでなければいけない
 - java.rmi.RemoteExceptionをthrowする必要はない、

74

Webサービス (アノテーション)

- @WebServiceアノテーション
 - そのBeanがWebサービスであることを示す
 - name属性: 名前, デフォルトはクラス名が利用される
 - serviceName属性: サービス名,
 - wsdlLocation属性: 定義済みのWSDLの場所を指定する,
 - endpointInterface属性: エンドポイントインターフェースのクラス名を特定する,
- @WebMethodアノテーション
 - そのメソッドがサービスとして公開されることを示す,
 - action属性: soap actionを指定する,
 - exclude属性: trueを指定するとそのメソッドは公開されない, デフォルトはfalse
- その他(WSDL関連, 引数や戻り値のマッピング)に関する詳細は, JSR-181 4章Web Services Metadataを参照

75

WebレイヤでのDependency Injection

- Webレイヤで Dependency Injection が利用可能
 - Servlet API 2.5 (JSR 154 Maintenance Release)
 - @EJB, @Resourceなどのアノテーションが利用可能
 - コンテナによって、自動的に参照が解決される,

76

WebレイヤでのDependency Injection

JSFのManagedBeanにEJBを注入

```
public class AccountBean {
    @EJB
    private AccountService accountService;
    .....
    public String register() {
        if (accountService.register(
            new Account(loginname, password, birthday))) {
            return "SUCCESS";
        } else {
            return "FAIL";
        }
    }
}
```

AccountBean.java

WebレイヤでのDependency Injection

- PersistenceContext利用時の注意
 - WebレイヤからEJB3 Entityを使いたい,
 - EJBレイヤではPersistenceContextを利用する,
 - しかし、PersistenceContextはThreadSafeではないので、Servletで利用するには注意が必要,
 - 解決策
 - PersistenceUnitを利用する,
- ```
public XXXServlet extends HttpServlet {
 @PersistenceUnit private EntityManagerFactory emf;
 public void service(HttpServletRequest req,
 HttpServletResponse res) throws Exception {
 EntityManager em = emf.createEntityManager();
 em.persist(entity);
 em.close();
 }
}
```

78

## WebレイヤでのDependency Injection

---

### □ 注意事項

- Dependency InjectionがサポートされたのはServletバージョン2.5以降
- web.xmlのweb-app要素のversion属性は、2.5としておく必要がある。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5">

</web-app>
```

web.xml(一部)

79

## まとめ

---

### □ Java EE 5

- EJB3を中心にアノテーションの利用が浸透

### □ 今後のJava EE

- Web層などより広範囲なアノテーションの利用が進むものと考えられる

80