CS 407
Web Programming

Enterprise JavaBeans

Dr. Williams
Central Connecticut State University

Announcements

- No office hours tomorrow
- Final project presentations will begin on Monday
  - All material must be submitted to Blackboard
  - Reminder projects must be submitted to Blackboard by the beginning of class to count as being on time
  - Make sure you have the computer you are going to demo on ready to go (i.e. already have GlassFish running and project deployed)

Driving the business

- Entity beans
  - Capture model – representation of business objects and how they are stored
  - Capture state related business logic – state of the objects, rules for what is allowed
- Controller
  - Page flow
  - Simple data interaction
- EJB
  - Capture complex transaction business logic

To EJB or not to EJB

- EJBs are designed to:
  - Encapsulate complex business logic
    - Not the persistence mapping
    - But the rules for persistence
  - Transaction management
  - Security
- …but wait what about controller
  - Queries
  - Suitable for simple updates
  - However often if EJB created for complex updates all persistence put on EJB to simplify maintenance

Enterprise JavaBeans goals

- Easy to create applications freeing from low level details of:
  - Managing transactions
  - Threads
  - Load balancing
- Concentrate on business logic let framework handle managing data processing

EJB goals

- Aims to be standard way client/server applications to be built using Java language
  - Specifies server component
  - Requires definition of how client interfaces
    - Local
    - Remote
- CORBA compatible

How EJB client/server system works

- Basic parts of EJB system
  - EJB component
    - Executes in a EJB container, which executes within EJB server
    - Java class written by a developer than contains business logic
  - EJB container
    - Encapsulates component – provides services sucha s transactions, resource management, versioning, scalability, mobility, persistence, and security
  - EJB object (next page)

How EJB client/server system works

- Basic parts of EJB system
  - EJB component
  - EJB container
  - EJB object and the remote interface
    - Clients execute on the EJB object which implements the “remote interface”
    - Appears to be just and interface to class
    - Runs on client and remotely executes EJB component’s methods
    - Analogous to DVD remote

How EJB client/server system works

- Implementation of EJB object is code generated by code tools provided with EBJ container
- When client wants to create a server-side bean it uses JNDI to locate the class of the bean it wants
  - Calls create() and remote object created on server

Topics

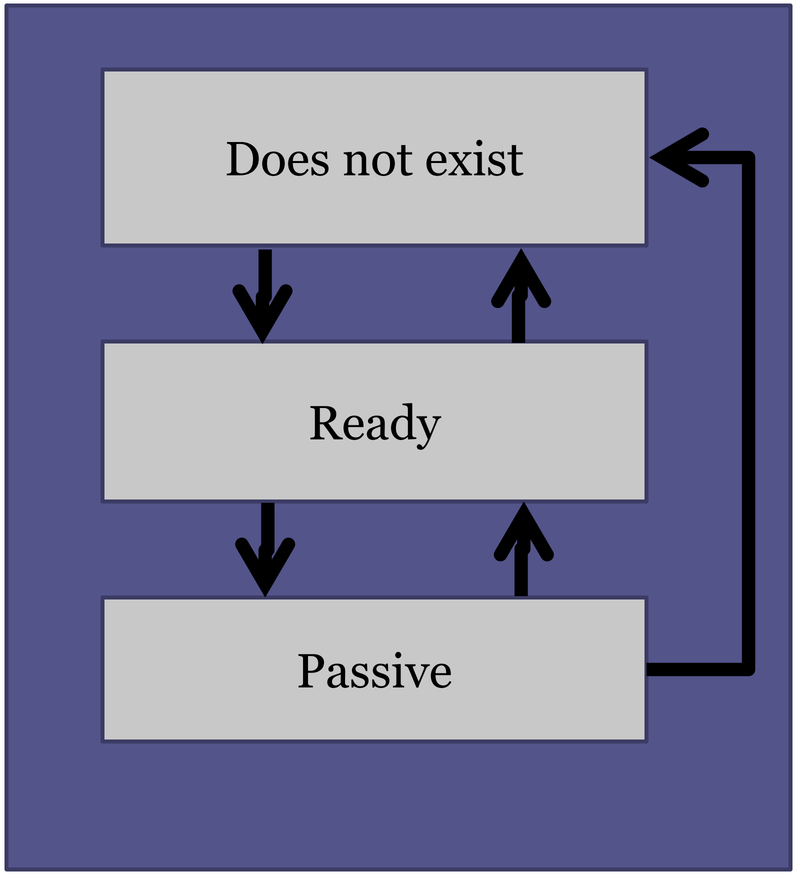
- Session beans
  - Session basics
  - DAO access pattern
  - Singleton session beans
  - Asynchronous methods
- Transactions in EJBs
  - Container managed
  - Bean-managed
- Life cycle
- Time service

Bean pool

- EJBs reside in application server in a pool similar to database connection pool
- Number initialized, when requested an initialized bean is returned
- When finished EJB is returned to pool for another process
- Pool properties
  - Initial and minimum pool size
  - Maximum pool size
  - Pool resize quantity
  - Pool idle timeout

Bean life cycle

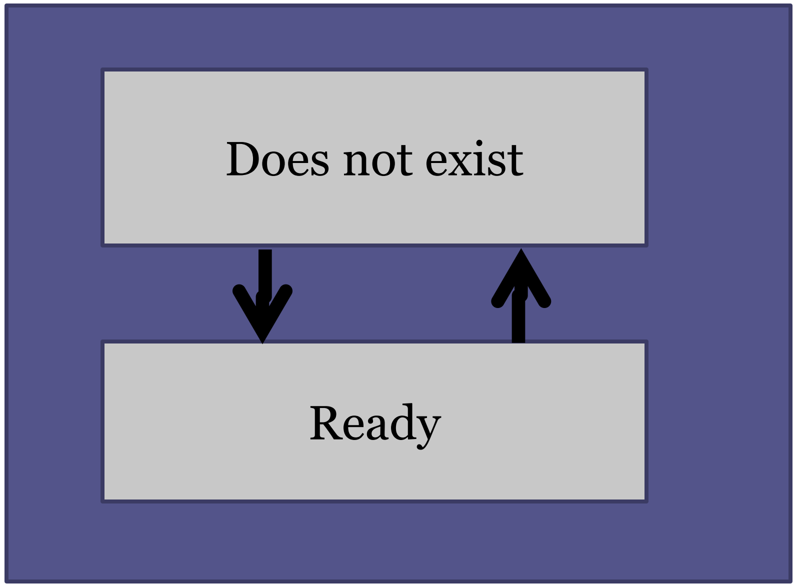
- Stateful session beans



- @PostActivate
- @PrePassivate
- @Remove
- @PreDestroy
- @PostConstruct

Bean life cycle

- Stateless session beans



- @PostActivate
- @PrePassivate
- @Remove
- @PreDestroy
- @PostConstruct

Session EJBs

- Logic related to verifying model rules enforced
  - Checks of states
  - Checks of consistency
  - Transactions that include multiple operations

Naming

- Note session EJBs does **not** mean alive for the session
- Stateful
  - Logic interaction across multiple calls
  - Allow variables to be kept
    - State of interaction
    - Pass request beans
- Stateless
  - Same purpose – complex transactions
  - State not kept – frees up resources

Creating the beans

```
import javax.ejb.Stateless
@Stateless
public class MyStatelessSessionBean
implements SimpleSession

import javax.ejb.Stateful
@Stateful
public class MyStatefulSessionBean
implements SimpleSession
```

Here there everywhere

- In addition to creating the implementing class need to create 1 or 2 interfaces
  - @Local
    - **Local business interface**
    - Added at declaration of interface
    - Interface for calling the EJB within same application server
  - @Remote
    - **Remote business interface**
    - Added at declaration of interface
    - Interface for calling the EJB from within another JVM or even across network

Using the EJB - same application server

```
public class SessionBeanLocalClient{
  @EJB
  private static SimpleSession sEJB;

  private void callEJB(){
     String msg = sEBJ.getMessage();
  }
}
```

Using the EJB remotely

```
public class SessionBeanRemoteClient{
  private static SimpleSessionRemote
  sEJB;

  private void callEJB(){
     sEJB =
  (SimpleSessionRemote)InitialContext.lo
  okup("java:module/SimpleSession");}
}
```

Data Access Objects (DAOs)

- Because of
  - Encapsulated logic
  - Transaction control
- Often used as wrapper for JDBC and JPA
  - Provide EJB interface specifying all data access for model element
    - saveCustomer(Customer customer)
    - Customer getCustomer(Long custId)
    - deleteCustomer(Customer customer)

JPA interaction

- Slightly different
    - Rather than injecting instance of EntityManagerFactory get access directly to EntityManager
    - Built in transactions

```
@PersistenceContext
Private EntityManager eManager;

private void updateCustomer(
                Customer customer){
  eManager.merge(customer);
}
```

Accessing from controller

```
@ManagedBean
public class CustomerController{
@EJB
CustomerDaoBean customerDaoBean;
@ManagedProperty(value="#{customer})
Customer customer;

public String saveCustomer(){
  …
  customerDaoBean.updateCustomer(customer);
  …
}
```

Singleton session beans

- Single instance of the EJB session bean exists for the entire application server
- Useful for caching data
  - Initialize once
  - Entire application can use results
- Common use caching code tables

Singleton implementation

```
@Singleton
public class SingletonSessionBean
implements SingletonSBRemote{
  @PersistenceContext
  EntityManager entityManager;
  private List<UsStates> stateList;

@PostConstruct
public void init(){
  Query query = entityManager.createQuery("select us
                                  from UsStates us");
  stateList = query.getResultList();
}
```

Asynchronous methods

- Just like with AJAX, there are sometimes benefits to being able to continue to execute without blocking
  - Data intensive operations – data load
  - Long running calculation
  - Remote call
- EJB offers two ways of doing async methods

Asynchronous methods #1

- Option 1
  - Add @Asynchronous annotation above method and return void
  - Method is executed async calling method continues to run, no way (doesn’t care) to tell when operation completes or if successful
  - Logging appropriate for monitoring errors

Asynchronous methods #2

- Option 2
  - Add @Asynchronous annotation above method and return `Future` generic
  - Method is executed async calling method continues to run
  - Calling method has ability to check completion status of async operation and read value when complete

Asynchronous methods #2

```
public Future<Long> asyncMyFunction(){
   …
   return new AsyncResult<Long>(42L);
}
```

Generics for the two must match and ensure that expected type matches for calling function

Reading the Future

- The future class offers several methods to check interact with the async call
  - cancel(boolean mayInterruptIfRunning)
  - get()
  - get(long timeout, TimeUnit unit)
  - isCancelled()
  - isDone()

Transactions in EJBs

- One of the key strengths of EJBs is their integration with transactions
- By default all interactions with EJBs are transactional
- Two options:
  - Container managed
    - Minimal work
    - Many configuration modes
  - Bean managed
    - Maximum control

Complexities of transactions

- EJB can be called locally or remotely
- EJBs can be called from other EJBs
- What about other transactions
  - What if calling method already in transaction
    - Use current?
    - Create new?
    - What if a problem should happen?
  - What if not already in transaction

Container managed transaction types

- Specify how to handle client transaction interaction through TransactionAttributeTypes
  - MANDATORY
  - NEVER
  - NOT_SUPPORTED
  - **REQUIRED**
  - REQUIRES_NEW
  - SUPPORTS

Specifying transaction types

```
@TransactionAttribute(
   value=TransactionAttributeType.SUPPORTS)
```

This annotation can be added either before class declaration, or before individual method declaration

Rollback

- Specifying to roll back the transaction will roll back only as far as transaction it is a part of (i.e. it may or may not include rolling back calling side)

```
@Resource
private EJBContext = ejbContext;

public void doSomething(){
   …
   ejbContext.setRollbackOnly();
   …
}
```

Bean-managed transactions

- Container managed handles most cases, especially with all transaction type options
- Bean-managed
  - Primary reason need to incorporate more than one transaction (ex. next page)
  - You don’t necessarily want to rollback if an exception occurs, with container managed this occurs
    - (Note you should usually avoid this situation occurring anyway)

Bean managed example

```
@TransactionManagement(value =
     TransactionManagementType.BEAN)
public class daoBeanMgmt implements…
  @Resource
  private UserTransaction transaction;

  public void saveMultiple(List<Customer> customers){
    for (Customer customer: customers){
      try{
        userTransaction.begin();
        entityManager.persist(customer);
        userTransaction.commit();
      }catch{
        // log but no roll back
      }
    }
  }
```

Allows 1-many fail while still saving the rest

EJB timer service

- Create and manage timers
- Create schedules for kicking off events
- Can be used with stateless session beans (and message-driven beans not covered here)

- Allows method of EJB to be called at regular intervals
  - Polling
  - Indexing
  - Clean up

Timer service – client initiated timers

- Setting up timers:

```
@Resource
TimerService timerService

public void startTimer(Serializable info){
  Timer timer = timerService.createTimer(new Date(),5000,info);
}
```

Arguments

- 1st – When to start initial expiration
- 2nd – interval duration
- 3rd – How to identify timer

- Other methods for setting up schedule expression timers (more later)
- **Key aspect must be invoked by client initially to start timer process**

Timer service - client initiated timers

```
public void stopTimer(Serializable info){
  Timer timer;
  Collection timers = timerService.getTimers();
  for (Object object : timers){
    timer (Timer)object;
    if (timer.getInfo().equals(info)){
      timer.cancel();
      break;
    }
  }
}

timerEJBExample.startTimer("Timer 1");
timerEJBExample.stopTimer("Timer 1");
```

Timer also provides methods to check time remaining

Acting on time events

- Remember stateless session bean
- Resource of TimerService
- Callbacks for timer events specified with
`@Timeout`
  - Can be specified for multiple methods on the EJB

```
private static Logger logger =
Logger.getLogger(MyTimerBean.class.getName());

@Timeout
public void logMessage(Timer timer){
  logger.info("Message triggered by: " +
timer.getInfo());
}
```

Scheduled jobs

- Creating timers in previous way requires client to invoke bean before timer is started
  - Result not useful for kicking off at application startup
- Calendar-based EBJ timers
  - Provide same basic functionality in terms of setting up timer schedule
  - Timer bound to bean at initiation
  - As soon as app server starts up scheduled timer initiates

Calendar-based EJB timer expressions

- @Schedule annotation used to bind and specify schedule
- Can have multiple schedules each bound to different methods

```
@Schedule(hour = "20", minute = "10")
public void logMessage(Timer timer){
  logger.info("Message was triggered at: " +
System.currentTimeMillis());
}
```

- Method will get called at 8:10pm everyday

Power of schedule syntax

Within @Schedule(xxx="4",yyy="Mon-Fri")

- dayOfMonth – default is all "*"
  - "3" third day of month
  - "Last" last day of month
  - "-2" two days before end
  - "1st Tue" first Tuesday of the month
- dayOfWeek – default is all "*"
  - "3" every Wednesday
  - "Thu" every Thursday
  - "Tue, Sat" Tuesday and Saturday
  - "Mon-Fri" Monday thru Friday

Power of schedule syntax cont.

- hour – default is “0”
  - “14” 2pm
- minute – default is “0”
  - “10” 10 minutes after the hour
- month – default is all “*”
  - “2” February
  - “March” March
- second – default is “0”
  - “5” 5 seconds after the minute
- timezone – default is local timezone
  - “America/New York”
- year – default is all “*”
  - “2010” Four digit year

Power of schedule

- In addition to being able to specify specific times can also set up intervals
  - @Schedule(hour = “*/12”)
  - Method will be called every 12 hours (all divided by 12)
  -
- JEE version of Unix *cron* jobs