Shale and the Java Persistence Architecture

Craig McClanahan
Gary Van Matre

ApacheCon US 2006
Austin, TX

Agenda

- The Apache Shale Framework
- Java Persistence Architecture
- Design Patterns for Combining Frameworks
- Questions and Answers

The Apache Shale Framework

- Architected as extensions to the JavaServer Faces controller framework
- Key functional components:
  - View controller (application callbacks)
  - Dialog manager (scoped conversations)
  - Clay plug-in (alternate view handler)
  - Tiger extensions (annotations based config)
  - Remoting (AJAX back end support)
  - Struts compatibility functionality:
    - Tiles, Validator, Token

View Controller

- For our purposes, View Controller is the key touchpoint between the frameworks
- Based on JSF convention of “backing beans”
- Four application oriented callbacks:
  - **init()** -- called when view is created or restored
  - **preprocess()** -- called when about to process a postback
  - **prerender()** -- called when about to render this view
  - **destroy()** -- called after rendering, if init() was called
- Also supports init/destroy lifecycle events for request/session/application scoped data beans

View Controller and Model Tier

- Questions about backing beans (and view controllers):
  - Where is the business logic?
  - How is the model tier accessed?
- We will look into options after we explore JPA ...

Java Persistence Architecture

- Part of JSR-220 (Enterprise JavaBeans 3.0)
- Began as a simplification to entity beans
- Evolved into POJO based persistence technology:
  - Rich modelling capabilities, inheritance, polymorphism
  - Standardized object/relational mapping
  - Powerful query capabilities
- Scope expanded at the request of the community:
  - Into persistence technology for Java EE
  - To support out-of-container use is Java SE
  - To support pluggable persistence providers

JPA – Key Concepts

- Entities
- Persistence Units
- Persistence Contexts

JPA – Entities

- Plain old Java objects:
  - No required interfaces
  - Created using *new Foo()*
  - Support inheritance, polymorphism
  - Have persistent identity
  - May have both persistent and non-persistent state
- Usable outside the container:
  - Serializable
  - Can be used as a detached object

JPA – Entities

- Queryable via Java Persistence query language
  - Similar to SQL, but extended for O/R mapping
  - Dynamically constructed query strings
  - *Named Queries* embedded in entity classes
- Managed at runtime through the *Entity Manager* APIs

JPA – Entity Classes

```
@Entity
public class Customer {

    @Id private long id;

    private String name; // Non-persistent data

    @OneToMany List<Order> orders = new ArrayList();
    public List<Order> getOrders() { return orders; }

    public void addOrder(Order order) {
        getOrders().add(order);
    }

}
```

JPA – Persistence Unit

- Unit of persistence packaging and deployment
- Set of managed classes:
  - Entities
  - Related classes (primary keys, etc.)
- Defines scope for:
  - Queries
  - Entity relationships
- Object/relational mapping information:
  - Java language annotations and/or XML files
- Configuration information for provider:
  - **META-INF/persistence.xml** file

JPA – Persistence Unit

- Sample persistence.xml (shale-mailreader-jpa):

```
<persistence version="1.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="MailReaderJpa"
        transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.ejb.cmp3.
      EntityManagerFactoryProvider</provider>
    <non-jta-data-source>jdbc/mailreader
      </non-jta-data-source>
    <properties>
      <property name="toplink.ddl-generation"
                value="create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

JPA – Persistence Context

- Runtime application execution context
- Set of managed bean instances, belonging to a single persistence unit:
  - Entities that *have been* read from the database
  - Entities that *will be* written to the database
    - Including newly persistent entities
  - Persistent entity identity == Java identity
- Persistence context lifetime is either:
  - Single transaction scoped
  - Extended – spanning multiple transactions

JPA – Persistence Contexts

- Managed by either *container* or *application*
- Container managed persistence contexts:
  - Provide ease of use in Java EE environment
  - Propogated across components if using JTA
  - Obtained by injection or JNDI lookup
  - May be single-transaction or extended
- Application managed persistence contexts:
  - Use in either Java SE or Java EE environment
  - Obtained from *EntityManagerFactory*
  - Extended scope must be managed by app also
- Web tier supports both approaches

JPA – EntityManager API

- Entity lifecycle operations:
  - persist(), remove(), refresh(), merge()
- Finder operations:
  - find() (by primary key), getReference()
- Factory for query objects:
  - createQuery(), createNamedQuery(), createNativeQuery()
- Operations to manage persistence context:
  - Flush(), clear(), close(), getTransaction(), joinTransaction()

JPA – Persisting an Entity

```
@PersistenceContext
private EntityManager em;

...

public Order addNewOrder(Customer cust, Product prod) {

  Order order = new Order(prod);
  cust.addOrder(order);
  em.persist(order);
  return order;

}
```

JPA – Removing an Entity

```
@PersistenceContext
private EntityManager em;

...

public void dropCustomer(long custId) {

  Customer cust = em.find(Customer.class, custId);
  em.remove(cust);

}
```

JPA – EntityManagerFactory

- Factory for EntityManager instances
  - CreateEntityManager()
- Allows pluggable replacement of JPA implementation in Java SE environment
  - Container will have picked a particular implementation
- Injectable via @PersistenceUnit annotation

JPA – Other Issues

- Resource injection:
  - Applies to container-managed persistence contexts only
  - Performs annotation based dependency injection ...
    - @PersistenceContext, @PersistenceUnit, ...
  - On container-created objects only:
    - Servlet, Filter, Listener, *JSF Managed Bean*
  - Alternatives: JNDI lookup, Spring, programmatic access
- Thread safety:
  - EntityManager instance is **not** thread safe
  - EntityManagerFactory **is** thread safe

Overall Design Patterns

- Three basic patterns for integrating Shale/JSF and JPA:
  - “All in one” backing bean
  - POJO based business logic
  - Session EJB based business logic
- Plus a more radical alternative (JBoss Seam):
  - Backing bean **is a** stateful session bean

"All In One" Backing Bean

- Combines model access and business logic directly into backing bean class
- Suitable when required logic is extremely simple
- Characteristics:
  - EntityManager instance injected by container
  - Therefore, must be request scoped for thread safety
  - Event handler performs required data access
- Example: user login authentication

“All In One” Backing Bean

```
public class LogonBean {

  // EntityManager is injected by the container
  @PersistenceContext EntityManager em;

  // Properties for username/password (bound to components)
  private String username;
  public String getUsername() { return username; }
  public void setUsername(String username)
    { this.username = username; }

  private String password;
  public String getPassword() { return password; }
  public void setPassword(String password)
    { this.password = password; }

  ...
```

“All In One” Backing Bean

```
// Action method bound to submit button
public String authenticate() {
  try {
    Query query = em.createQuery
     (“select User u from Users where u.username =” +
      “:username and u.password = :password”);
      em.createNamedQuery(“User.findByUsernamePassword”);
    query.setParameter(“username”, username);
    query.setParameter(“password”, password);
    User user = (User) query.getSingleResult();
    ... store user in session to denote logged in status ...
    return “success”;
  } catch (NoResultException e) {
    ... store error message to be redisplayed ...
    return null; // Redisplay current page
  }
}

}
```

POJO Based Business Logic

- Often, business logic (and model tier access) is:
  - Nontrivial
  - Reusable
- Standard design pattern is to encapsulate in a separate Java class:
  - In Java-based apps, often a POJO
  - In webapps, typically stored in application scope
  - So, must deal with multithreaded access
- Let's look at such a business logic bean first ...

POJO Based Business Logic

```
public class Logic {
  // EntityManagerFactory is injected by the container
  @PersistenceUnit EntityManagerFactory emf;
  // Typical business logic method
  public Subscription createSubscription(Subscription subs) {
    EntityManager em = emf.createEntityManager();
    EntityTransaction et = null;
    try {
      et = em.getTransaction();  et.begin();
      User user = em.find(User.class, subs.getUser().getId();
      user.addSubscription(subs);
      em.persist(subs);  et.commit;
    } catch (Exception e) { ...
    } finally {
      if ((et != null) && et.isActive()) { et.rollback(); }
      em.close();
    }
  }
}
```

POJO Based Business Logic

- JSF can inject an instance of the business logic bean into backing beans for you as well:
  - Define a “logic” property of type Logic
  - Configure the business logic managed bean:

```
<managed-bean>
  <managed-bean-name>backing</managed-bean-name>
  <managed-bean-class>...<managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>logic</property-name>
    <value>#{logic}</value>
  </managed-property>
</managed-bean>
```

- Thus, backing beans have direct access to logic

Session Bean Business Logic

- In our business logic bean, we had to deal with transactions explicitly
  - Consider using a stateless session bean (EJB)
    - Transactions managed by the container
  - Also get other benefits:
    - No more thread safety concerns
    - Scalability (across server instances)
    - Participate in cross-resource transactions
- What adjustments would we have to make to our POJO example?

Session Bean Business Logic

- Define a business interface for our logic:

  public interface BusinessLogic { ... }
- Make our logic class implement the interface ...
- And add simple annotations:

  @Local @Stateless

  public class Logic implements BusinessLogic ...
- And we have just created a stateless session bean which can be injected into our backing beans:

  @EJB private Logic logic;
- We now have direct access to business methods

The JBoss Seam Alternative

- We won't have time to delve into all the details ...
- But Seam offers an interesting alternative:
  - Combine backing bean and business logic bean into one
    - Theory: the “reusable” business logic isn't always reusable
    - Theory: the backing bean's logic is just glue
  - While we are at it, we can use stateful session bean
    - Maintain conversational state across HTTP requests ...
    - Using the EJB container instead of HTTP sessions

Other Design Notes

- Binding UI component values:
  - Typical pattern: bind to properties of backing bean
    - In Struts, this would have been the form bean
    - Result: lots of copying values back and forth
  - Instead, make a JPA entity class a property of your backing bean, and bind to it directly
- Passing state between requests:
  - Typical pattern: pass primary keys around
  - Alternative: JPA entity instances can be *detached* and sent along with the JSF component state

Other Design Notes

- Creating bookmarkable URLs:
  - Shale's ViewController callbacks can help here
  - Two different HTTP accesses are supported:
    - GET: init() --> prerender() --> destroy()
    - POST: init() --> preprocess() --> action method --> prerender() --> destroy()
  - Can also access request parameters for keys
- We will see a worked out demonstration shortly

Demo

Today's News

- Shale has a brand new logo image:

- And a “powered by” logo:

- Congrats to Walied Amer, logo contest winner

Questions and Answers