Blueprints for Using the Simplified Java™ EE 5 Programming Model

Smitha Kangath, Inderjeet Singh

Java BluePrints
Sun Microsystems Inc.

TS-1969

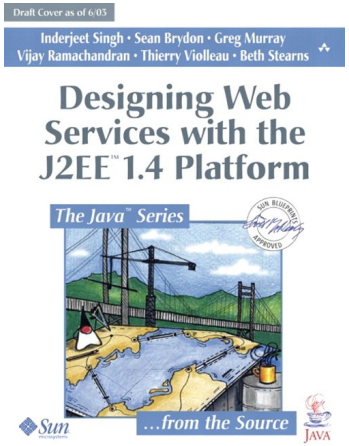
Goal of the Talk

Application Programming Model for Java™ EE 5 Platform

Sample the key features of the Java EE 5 Platform and see how it simplifies the development of Enterprise Applications and Web Services

Speaker's Qualifications

- Members of the Java BluePrints Program
  - http://blueprints.dev.java.net/
  - Java BluePrints Solutions Catalog
    - Topics: AJAX, JSF, Enterprise, Web Services
    - J2EE 1.4 and Java EE 5
  - Java Pet Store, *a new version showing Web 2.0 with Java EE 5*
  - Java Adventure Builder
  - Books
    - Designing Web Services with the J2EE 1.4 Platform
    - Designing Enterprise Applications with the J2EE Platform, 2nd Ed



Agenda

- POJO-based Programming
  - Annotations and Dependency Injection
- Key Changes in Component Models
  - EJB 3.0
  - Web tier
  - Web Services
- Programming Model for Java Persistence
  - With EJB 3.0
  - In Web-tier without EJBs

The J2EE™ Challenges

- Powerful and comprehensive
  - Supports lots of use-cases: Web applications, Web Services, messaging, database applications, etc.
  - Deployment Descriptors allow a lot of customization
- Challenges
  - Rigid enforcement of “good” design and patterns
  - Can be difficult to get started
  - Boring boilerplate code
  - Much Container configuration to get application to work
- Java EE 5 addresses these challenges

EoD through POJO based Programming

- Express Programmer's intentions in Java Code instead of in XML or other external configuration actions
  - Java Class is the main programming artifact
  - Annotations add new capabilities to the class
  - Lots of sensible defaults
  - Can be overridden by XML based deployment descriptors
  - Still fully compatible with J2EE 1.4

Annotations

- Based on Java SE 5 Annotations Support
- Annotations available for
  - Defining Web Services
  - Defining Enterprise Beans
  - Calling out EJB Lifecycle callbacks or Interceptors
  - Dependency Injection
  - Almost anything that you used to previously have a Deployment Descriptor entry for
    - Transaction Attributes
    - Security
- Look at JSR-250: Common Annotations

Annotations: Pros/Cons

- Annotations: Pros
  - Easier to write than .xml
  - Easier to understand than .xml
  - Fewer files to maintain
- Annotations: Cons
  - Only visible in source-code
  - Can't express all Java Platform, EE 5 metadata
  - Blurs lines between Java EE platform roles (e.g., Component Provider vs. Application Assembler)

Best Uses for Annotations

- Metadata that does not change often
- Metadata tied to component development time
- Examples
  - Structural metadata
    - e.g., @Stateless, @WebService, @Entity
  - Environment dependencies
    - e.g., @EJB, @Resource, @PersistenceContext
  - Callbacks
    - e.g., @PostConstruct, @Timeout, @Remove

Best Uses for Deployment Descriptors

- Overriding annotations and defaults
- Application assembler metadata
  - EJB Security Method Permissions
    - Typically not known until assembly/deployment time
    - Likely to change
    - Independent of business logic
  - Dependency linking info
    - e.g. cross-module ejb-link
- Metadata that has no corresponding annotation
  - e.g. EJB Default interceptors, EJB 2.x Entity Beans, Message Destinations

Guidelines for .xml Overriding

- Use sparingly
  - Overuse can make app difficult to understand/maintain
- Good with linking metadata
  - e.g., ejb-link, persistence-unit-name
- Keep in mind not all annotations are overridable
  - e.g., Session bean type (Stateful vs. Stateless) can't be overridden

Don’t Forget Spec-Defined Defaults!

- Default values can be easier than annotations and .xml
  - EJB based transaction demarcation type
    - Default: Container managed transaction
  - EJB based transaction attribute
    - Default: TX_REQUIRED
  - Environment annotation name()
    - Default: Derived from class and field/method

Component Dependency Annotations

- For declaring environment dependencies
- For eliminating JNDI lookups
  - ejb-ref, resource-ref, service-ref, etc.
- Available on container-managed classes
  - Enterprise beans and interceptors, Servlets, Filters, ServletListeners, JSF Managed Beans, Web service endpoints and Handlers
  - Not for JSP, JSP beans, or other plain Java classes that are not available at deployment
- Declared at class, field, or method level
- Field/method level dependencies injected at runtime

annotation vs. .xml

```
@Resource(name=”Foo”) private DataSource ds;
```

```
<resource-ref>
  <res-ref-name>Foo</res-ref-name>
  <res-ref-type>javax.sql.DataSource</res-ref-type>
  <injection-target>
    <injection-target-class>com.acme.FooBean</...>
    <injection-target-name>ds</injection-target-name>
  </injection-target>
</resource-ref>
```

Dependency Injection

- Available for Fields as well as methods
- Available anywhere on the inheritence hierarchy
  - Follows normal language overriding rules
- @PostConstruct annotation available to provide initialization after injection

Injected Field/Method Access Modifiers

- Spec allows public, package, protected, private
- Which should you use?
  - Private is best
  - Injected data is typically internal to the .class
- Exception: Overriding of environment dependencies within a class hierarchy
  - Use sparingly
    - Tightly couples classes
    - Harder to understand/maintain

Which Is Best: Field, Method, or Class-level?

- Field-level: Easiest
  - e.g., @EJB Converter converter
  - Takes fewest characters to declare
  - Supports injection
- Method-level
  - Useful for logic tied to a specific dependency injection
  - But... Field-level + @PostConstruct would work too

Which Is Best: Field, Method, or Class-level? (Cont.)

- Class-level
  - Useful for dependency declaration WITHOUT injection
  - Declare environment dependency for use by non container-managed classes

FooBean.java:

```
@EJB(name="ejb/bar", beanInterface=Bar.class)
public class FooBean implements Foo { ... }
```

Utility.java:

```
Bar bar = (Bar) context.lookup("java:comp/env/ejb/bar");
```

Another Class-Level Dependency Example

- Stateful Session Bean creation
  - EJB 3.0 SFSBs are created as a side-effect of injection/lookup
  - Common need : many instances of same SFSB
  - Using field-based dependency + injection:

```
@EJB Cart cart1;
@EJB Cart cart2;
@EJB Cart cart3;
```

Too static :-(

Another Class-Level Dependency Example (Cont.)

- Alternative: class-level dependency + lookup

```
@EJB(name="ejb/Cart", beanInterface=Cart.class)
public class CartClient {
    ...
    Cart[] carts = new Cart[numCarts];
    for(int i = 0; i < carts.length; i++) {
     carts[i] = (Cart)
     ctx.lookup("java:comp/env/ejb/Cart");
   }
```

Concurrency and Injection

- Injection does not solve concurrency issues
- If an object obtained through lookup() is non-sharable, it's non-sharable when injected
- Be careful with Servlet instance injection

```
public class MyServlet ... {
    private @EJB StatelessEJB stateless;   // OK
    private @EJB StatefulEJB stateful; // dangerous!
```

Concurrency and Injection (Cont.)

- Most common issues: Stateful Session Beans, PersistenceContexts
- Recommended alternative: lookup() and store in HttpSession

```
@PersistenceContext(name="pc",
   type=EntityManager.class)
public class MyServlet ... {
    EntityManager em = ctx.lookup("java:comp/env/pc");
    httpSession.setAttribute("entityManager", em);
```

Performance and Injection

- Use of injection is unlikely to cause performance issues
- Injection is essentially a ctx.lookup() + one reflective operation
- Injection occurs at instance creation-time
  - Overhead of injection typically small compared to instance creation itself
  - Most lookups() resolved locally within server
  - Instances are typically long-lived/reused

Agenda

- POJO-based Programming
  - Annotations and Dependency Injection
- Key Changes in Component Models
  - EJB 3.0
  - Web tier
  - Web Services
- Programming Model for Java Persistence
  - With EJB 3.0
  - In Web-tier without EJBs

What Changed in EJB?

- Issues with EJB 2.1
  - Good component model, but required too much coding and concepts
    - Too many classes, interfaces, concepts
    - javax.ejb interfaces
    - Complex JNDI lookups
    - Awkward programming model
    - Deployment descriptors
    - Entity bean anti-patterns

What is Different in EJB 3.0?

- POJO-based Component Definition
  - No required Container interfaces
  - No required deployment descriptor
- Dependency Injection
- Decoupled Java Persistence from EJB components
- Simple lookups
- No required Deployment descriptor

Stateless Session Bean with J2EE

```
public class PayrollBean implements javax.ejb.SessionBean {
  SessionContext ctx;
  DataSource empDB;
  public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
  }
  public void ejbCreate() { empDB = (DataSource)
ctx.lookup("jdbc/empDb"); }

  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbRemove) {}

  public void setBenefitsDeduction(int empId, double deduction) {
    ...
    Connection conn = empDB.getConnection();
    ...
  }
...
}
```

Need Remote/Local, Home interfaces, Deployment descriptors

Stateless Bean Example with Java EE 5

```
@Stateless
public class PayrollBean implements Payroll
{
      @Resource DataSource empDB;
      public void setBenefitsDeduction(int empId,
                                       double deduction) {
         ...
        DataSource conn = empDB.getConnection();
      }
   ...
}
```

Dependency Injection in EJB

- Resources a bean depends upon are injected when bean instance is constructed
- References to
  - EJBContext
  - DataSources
  - UserTransaction
  - Environment entries
  - EntityManager
  - TimerService
  - Other EJB beans
  - ...

Dependency Injection

- Annotations
  - @EJB
    - References to EJB business interfaces
    - References to Home interfaces (when accessing EJB 2.1 components)
  - @Resource
    - Almost everything else
  - Number of annotations is simplified from EJB 3 specification early draft
- Injection can also be specified using deployment descriptor elements

Simplified Client View

- Session beans have plain Java language business interface
  - No more EJB(Local)Home interface
  - No more EJB(Local)Object interface
- Bean class implements interface
  - Looks like normal Java class to Bean developer
- Looks like normal Java interface to client

EJB 3.0 Client Example

```
// EJB 3.0 client view

@EJB ShoppingCart myCart;

    ...

    Collection widgets = myCart.startToShop("widgets");


    ...
```

Why Use EJB 3.0?

- Nothing in the platform is REQUIRED to be used
  - Use based on application requirements
- Benefits of EJB 3.0
  - Helps componentize and modularize code
  - Enforce good architecture
  - Good integration with Java Persistence
  - Greatly simplified concept

What Changed in the Web Tier?

- JSF 1.2 became part of the Java EE 5 platform
- Annotations support
  - No component defining annotations
  - Common annotations in container managed objects
    - JSF managed beans, servlets, filters, event listeners
    - Not in JSP. But available in JSP tag handlers or event listeners
- Web.xml not needed
  - Not needed if only JSP and Web service annotated classes
  - Still needed for JSF, servlets, security settings, etc.

Programming Model for Web Tier

- Traditional JSP/Servlets based applications
  - Use an MVC Framework
  - For Web 2.0, use an AJAX library; for example, Dojo
- JSF 1.2: Standardized MVC framework
  - Component-based
  - Unified expression language for JSP and JSF
  - Best used with a tool like Java Studio Creator
    - Also possible to write applications by hand
  - For Web 2.0, wrap AJAX functionality in reusable components

What Changed for the Web Services?

- Significantly revised and simplified
- JAX-RPC 2.0 renamed to JAX-WS 2.0
  - Breaks compatibility with JAX-RPC 1.1
  - JAX-RPC 1.1 is also available
- Key Features
  - Simplified programming model with annotations and dependency injection
  - Uses JAXB 2.0 for type-mappings
  - Portable runtime artifacts

Key Features in Web Services

- Key Features (cont.)
  - Supports Fast-Infoset for high performance
  - JAX-WS supports REST services
    - Useful for AJAX Backends
  - Can generate annotated JAX-WS and JAXB code from WSDL and XSD

JAX-WS 2.0 New Architecture

- Multiple protocols
  - SOAP 1.1, SOAP 1.2, XML
- Multiple encodings
  - XML, MTOM/XOP, FAST Infoset (Binary XML)
- Multiple transports
  - HTTP
  - Others to be added in future releases

JAXB 2.0 Is Now Bi-Directional

- 1.0: Schema → Java only
  - JAXB is for compiling schema
  - Don't touch the generated code
- 2.0: Java → XML + schema compiler
  - JAXB is about persisting POJOs to XML
  - Annotations for controlling XML representation
  - Modify the generated code to suit your taste

Web Service Annotation Example

```
@WebService(name="Hello" serviceName="HelloService")
public class HelloWebService {

    @WebMethod
    public String sayHello(String s) {...}
    public void unpublished() {...}
}
```

```
public class HelloClient {
    @WebServiceRef (wsdlLocation=
    "http://localhost:8080/HelloService?WSDL")
    static hello.HelloService service;

    public static void main(String[] args) {
      hello.Hello wsPort = service.getHelloPort();
      System.out.println(wsPort.sayHello());
    }
}
```

Agenda

- POJO-based Programming
  - Annotations and Dependency Injection
- Key Changes in Component Models
  - EJB 3.0
  - Web tier
  - Web Services
- Programming Model for Java Persistence
  - With EJB 3.0
  - In Web-tier without EJBs

Java Persistence API

- Part of JSR-220, but a separate document
  - No reliance on EJB technology or EJB container
  - Usable in Web-only applications and Java SE
- POJO-based persistence
  - Lightweight domain objects—no overhead of container-managed components
  - Sensible default mappings
  - Complete query capabilities
- Key concepts - persistent entities, persistence unit, persistence context, entity manager

Persistent Entities

- Plain old Java objects
- No more interfaces required
- Supports use of new, inheritance
- Persistent properties with JavaBean style accessor methods or persistent instance variables
- Usable as “detached” objects in other application tiers—no more need for Data Transfer Objects
- Persistence, querying, and O/R mapping managed by the Java Persistence API

Persistent Entities (Cont.)

```
@Entity
public class Item implements java.io.Serializable {

    private String itemID;
    private String name;
    private String description;
    ...
    @Id
    public String getItemID() {
        return itemID;
    }
    public String getName() {
        return name;
    }
    ...
    public void setName(String name) {
        this.name = name; }
    ...
}
```

Persistence Unit

- Unit of packaging and deployment
- Set of related classes that map to a single database
- Defined by a persistence.xml file
- Includes O/R mapping metadata—metadata annotations or XML files

Persistence Context and EntityManager

- Persistence Context
  - Similar to transaction context, it's a scope
  - Entity instances are managed within the persistence context
  - A unique instance exists for any persistent entity identity
- EntityManager
  - API to manage the entity instance lifecycle
    - persist, remove, merge etc.
  - Operations to find entities by primary keys, to create Query objects, and to manage the persistence context
    - find, createQuery, close, getTransaction etc.

Types of Persistence Context

- Persistence Context lifetime maybe transaction-scoped or extended
- Transaction-scoped persistence context
  - bound to a JTA transaction—starts and ends at transaction boundaries
  - entities are detached from the persistence context when transaction ends
- Extended persistence context
  - spans multiple transactions
  - exists from the time the EntityManager instance is created until it is closed
- Defined when the EntityManager instance is created

Types of EntityManager

- Entity manager may be container-managed or application-managed
- Container-managed entity manager
  - Lifecycle managed by the Java EE container
  - May use transaction-scoped or extended persistence context
  - Extended persistence context is only available to stateful session beans
- Application-managed entity manager
  - Life cycle managed by the application
  - Also available in Java SE environments
  - Must use extended persistence context

Container-Managed Entity Manager Example

- Dependency injection of EntityManager with the @PersistenceContext annotation or a JNDI lookup

```
@Stateless
public class CatalogFacadeBean implements CatalogFacade{

  @PersistenceContext(unitName="PetstorePu")
  private EntityManager em;
  ...
  public List<Category> getCategories(){
      List<Category> categories =  em.createQuery("SELECT
      c FROM Category c").getResultList();
      return categories;
  }
  ...
}
```

Application-Managed Entity Manager Example

- Dependency injection of EntityManagerFactory with the @PersistenceUnit annotation

```
public class CatalogFacade implements
ServletContextListener {

  @PersistenceUnit(unitName="PetstorePu")
   private EntityManagerFactory emf;
   ...
   public List<Category> getCategories() {
       EntityManager em = emf.createEntityManager();
       List<Category> categories = em.createQuery("SELECT
       c FROM Category c").getResultList();
       em.close();
       return categories;
   }
   ...
}
```

Transactions with Entity Manager

- JTA entity manager
  - Transactions are controlled through JTA
  - Container-managed entity manager always does JTA transactions
- Resource-local entity manager
  - Transactions are controlled through the EntityTransaction API
  - Application-managed entity manager can be either JTA or resource-local
- Transactional type is defined in persistence.xml

Entity Operations

- Persisting an entity

```
@PersistenceContext(unitName="PetstorePu")
private EntityManager em;

...

Item item = new Item(itemID, name, description, price);
em.persist(item);
```

- Finding and removing an entity

```
Item item = em.find(Item.class, itemID);
em.remove(item);
```

Query API

- To query and retrieve entities
- Static and dynamic queries
- Named parameter binding and pagination control
- Queries are defined in Java Persistence Query Language or native SQL
- Named Queries

An Example Using Queries

```
@PersistenceContext(unitName="PetstorePu")

private EntityManager em;

...

public List<Product> getProducts(String catID){

    Query query = em.createQuery("SELECT p FROM Product p
        WHERE p.categoryID LIKE :categoryID");
    List<Product> products = query.setParameter
        ("categoryID", catID).getResultList();

}
```

Refactoring Using Named Queries

- Defining named queries

```
@NamedQuery(
  name="Item.getItemsPerProduct",
  query="SELECT i FROM Item i WHERE i.productID LIKE :
  pID")

@Entity
public class Item implements java.io.Serializable  {
...
}
```

- Using named queries

```
Query query = em.createNamedQuery
("Item.getItemsPerProduct");
query.setParameter("pID",prodID);
List<Item> items = query.getResultList();
```

Native SQL vs. Java Persistence Query Language

- Native SQL
  - Returns raw data – field values for the entity
  - Complex SQL for navigating relationships
- Java Persistence Query Language
  - Returns entities
  - Relationships can be navigated using a “.”
  - Similar to SQL - small learning curve

Value List Handler

- Common design pattern
- Helper method in Java Persistence Query Language to get chunks of data

```
public List<Item> getItemsVLH(String prodID, int start,
int chunkSize){
    ...
    Query query = em.createQuery("SELECT i FROM Item i
        WHERE i.productID = :pID");
    List<Item>  items = query.setParameter("pID",prodID).
        setFirstResult(start).setMaxResults(chunkSize).
        getResultList();
    em.close();
    return items;
}
```

O/R Mapping Metadata

- Physical mapping annotations
  - tables, columns etc. eg.,@Column, @Table
- Logical mapping annotations
  - Relationship modeling annotations eg.,@OneToOne
- Relationship mappings can be One-to-One, One-to-many, Many-to-one, and Many-to-many
- Relationships may be unidirectional or bidirectional

O/R Mapping Metadata: Example

```
@Entity

@Table(name="Customer")

public class Customer implements Serializable {
   private String name;
   private Collection<Order> orders;
   ...
   @Column(name="CUST_NAME")
   public String getName() {
       return name;
   }

   @OneToMany
   public Collection<Order> getOrders() {
       return orders;
   }
   ...

}
```

Automatic Generation of Primary Keys

- Different strategies – TABLE, SEQUENCE, IDENTITY, AUTO

```
@Entity
public class Item implements java.io.Serializable {
    ...
    @TableGenerator(name="ITEM_ID_GEN", table="ID_GEN",
    pkColumnName="GEN_KEY",valueColumnName="GEN_VALUE”,
    pkColumnValue="ITEM_ID", allocationSize=1)

    @GeneratedValue(strategy=GenerationType.TABLE,
    generator="ITEM_ID_GEN")
    @Id
    public String getItemID() {
        return itemID;
    }
    ...
}
```

Java Persistence in the Web Tier

- Java Persistence was designed to be used without requiring EJBs
  - Can be used in the Web tier
  - Can be used in Java SE environments
- Web-only application may have
  - Application-managed or container-managed entity manager
  - Transaction-scoped or extended persistence context
  - Application-managed transactions

What Is Wrong with This Code?

```
public class CatalogServlet extends HttpServlet {

    @PersistenceContext(unitName="PetstorePu")
    EntityManager em;
    ...
    public void doGet( HttpServletRequest req,
        HttpServletResponse resp) throws ServletException,
        IOException {
        ...
        Item item = new Item(itemID, name, description,
            price);
        em.persist(item)
        ...
    }
}
```

This Code Is Not Thread-Safe

- PersistenceContext is injected just once during the entire lifecycle of the application
- Concurrent requests coming to the servlet will access the same PersistenceContext object
- PersistenceContext is NOT a thread-safe object!

A Better Way of Using Java Persistence in Web Tier

- Dependency injection of EntityManagerFactory with the @PersistenceUnit annotation

```
public class CatalogServlet extends HttpServlet {
    @PersistenceUnit(unitName="PetstorePu")
    EntityManagerFactory emf;
    ...
    public void doGet( HttpServletRequest req,
       HttpServletResponse resp) throws ServletException,
       IOException {
       EntityManager em = emf.createEntityManager();
       Item item = new Item(itemID, name, description,
                                price);
       em.persist(item);
       em.close();
       ...
    }
}
```

A Better Way of Using Java Persistence in Web Tier (Cont.)

- JNDI lookup to obtain the entity manager

```
@PersistenceContext(name="PetstorePu")
public class CatalogServlet extends HttpServlet {
    ...
    public void doGet( HttpServletRequest req,
       HttpServletResponse resp) throws ServletException,
       IOException {
       ...
       EntityManager em = (EntityManager) new
       InitialContext().lookup("java:comp/env/PetstorePu");
       Item item = new Item(itemID, name, description,
           price);
       em.persist(item)
       ...
    }
}
```

Managing Transactions in the Web Tier

- JTA transactions

```
@Resource UserTransaction utx;
...
public void addItem(Item item){
        try{
            utx.begin();
            em.joinTransaction();
            em.persist(item);
            utx.commit();
        } catch(Exception exe){
            ...
        } finally {
            em.close();
        }
}
```

Managing Transactions in the Web Tier (Cont.)

- Resource-local transactions

```
public void addItem(Item item){
        try{
            em.getTransaction().begin();
            em.joinTransaction();
            em.persist(item);
            em.getTransaction().commit();
        } catch(Exception exe){
            ...
        } finally {
            em.close();
        }
}
```

Facade Pattern

- Centralizes requests to the domain
- Handles and encapsulates transactions, entity managers, etc.
- May need to do dependency injections—use container-managed classes
- May return detached entities
- May aggregate calls to multiple entities
- May aggregate multiple calls to the entity manager, such as a find and then merge

Summary

- Use annotations and defaults to define components and external dependencies
- Use DD entries to override
- Use EJB 3.0 for simplified components
- Use JSF components for drag-and-drop Web application development
- Use JAX-WS (with integrated JAXB 2.0) for creating Web services use Java Persistence for OR mappings

If You Only Remember One Thing…

Java EE 5 Dramatically Simplifies The Programming Model for Enterprise Web Applications and Web Services

Q&A

Blueprints for Using the Simplified Java™ EE 5 Programming Model

Smitha Kangath, Inderjeet Singh

Sun Microsystems Inc.

TS-1969