



Java EE 5

in Action with

NetBeans

A critical exploration of Java EE 5's new productivity features, focusing on EJB 3.0, JPA, and NetBeans 5.5 tooling support

Oswaldo Pinali Doederlein

The traditional J2EE platform has long been regarded as powerful but difficult to learn and use: J2EE offers a massive number of high-end features and it's natural that these features have costs in complexity of architecture, APIs, and tools. Though NetBeans IDE has been offering strong support for J2EE development tasks for years, in an ideal world tools should not have to mask the inadequacies of frameworks. Frameworks should be well designed with respect to productivity – when this happens IDEs can be “unloaded” from the automation of dumb boilerplate code, and add an increasing number of higher-end productivity features without becoming, themselves, overloaded with complexity.

You will see in this article, which shows off both the new Java EE 5 functionality and NetBeans 5.5 features, that this is exactly the case for the next generation of enterprise Java development. We have a much easier, yet richer, framework, along with an increasingly powerful but still seamlessly integrated IDE, with effective support for the entire Java EE stack.

This article puts the major productivity enhancements of Java EE 5 through their paces. We'll look at annotation-driven APIs in general, EJB 3.0, the Java Persistence API (JPA), Dependency Injection, and application deployment. All this while going through a tutorial that develops a simple but complete application with NetBeans 5.5 and Glassfish/SJSAS, and looking at what NetBeans has to offer at the tool support side of Java EE 5. But differently from most tutorials that focus on teaching new APIs or techniques, we'll put some

time into analyzing the effectiveness of key Java EE features, extrapolating them to real-world projects.

Installing NetBeans and the Application Server

As this is NetBeans Magazine, I'll assume that you know how to download, install and configure the IDE and a compatible application server. At the time of writing, NetBeans offers support for SJSAS, GlassFish, JBoss, WebLogic, and JOnAS (with the JOnBAS plugin), besides Tomcat. This tutorial was developed with the Sun Java System Application Server 9.0 (a.k.a. Java EE 5 SDK or Java EE 5 RI), but it's application-server-portable. Except that the example application uses SJSAS's predefined *jdbc/sample* data source – so in other servers, you may have to configure a database and data source.

The JPM Application

The example application allows you to keep track of all Java platforms, composed of JSR specifications and APIs grouped by packages – deserving the pompous name “Java Platform Manager”. More specifically, the application manages a simple database, and is implemented with the JPA, EJB and JSF APIs.

Granted, due to space limitations I provide only a tiny amount of the potential functionality of such an app. But when you reach the end, I hope you'll be so impressed by the ease of development of Java EE 5 with NetBeans 5.5, that you'll enjoy coding the rest of it.

Creating the Projects

Start by selecting *File | New Project > Enterprise/Enterprise Application*. In the *Name and location* tab, set *Project Name* to *JPM*, change the base directory to an appropriate location, and accept defaults for the other options (see **Figure 1**). This creates the projects *JPM*, *JPM-ejb* and *JPM-war*. NetBeans adopts a project structure that mirrors the Java EE deployment structure, with one project for each deployable module (respectively the EAR, EJB-jar, and WAR modules).

You'll notice that the deployment model is similar to the traditional J2EE's. But it was simplified, starting with the descriptor files, most of which are now optional or much simpler. For example, though NetBeans creates a *JPM-ejb/Configuration Files/MANIFEST.MF* and fills



jcp.org/en/jsr/detail?id=244

Java EE 5 specification.

its *Class-Path* directive, this is often unnecessary. A Java EE 5 container will add all JARs inside an EAR to the classpath of all modules of that EAR without a *Class-Path* directive. For this tutorial, you may remove the *MANIFEST.MF* file.

Configuration by Exception

In Java EE 5, you can omit most configuration items for which the container can reasonably guess a default value. This is clearly the case of *MANIFEST.MF/Class-Path*.

Explicit configurations are still supported, both for compatibility and for cases where defaults are not good enough. For instance, if you need to force a specific class lookup order among several JARs, or if you want each module to import only the subset of JARs it really needs, then *Class-Path* is still your friend. But these scenarios should be rather the exception than the rule.

Even the traditional *ejb-jar.xml* is now optional. Look inside *JPM-ear/Configuration Files*. You won't find any EJB descriptor, not even after creating EJBs. On the other hand, you may still need proprietary descriptors. For example, with SJSAS/Glassfish, the *JPM/Configuration Files/sun-application.xml* file is required for settings such as security role mappings. As it turns out, proprietary settings cannot be moved from descriptors to code annotations. This is technically possible, but not a good idea. Annotations live in the source code, and proprietary annotations would break portability.

Notice that the Configuration by Exception feature is not the same as the tool-supported defaults we're more familiar with. For example, even with J2EE 1.4, NetBeans won't require that you write the *ejb-jar.xml* descriptor by hand. It provides a visual editor, and the

wizards and editor provide default settings, e.g. *Transaction Type=Container* for all EJBs. But the container still requires the XML descriptor with all these settings written down explicitly, thus polluting your projects with additional files and redundant settings, and making its maintenance harder.


Additionally, Java EE defaults are often much smarter than simple fixed values for missing properties. The automatic generation of classpaths is one example of non-trivial default logic. A more sophisticated example is DDL generation for persistent entities.

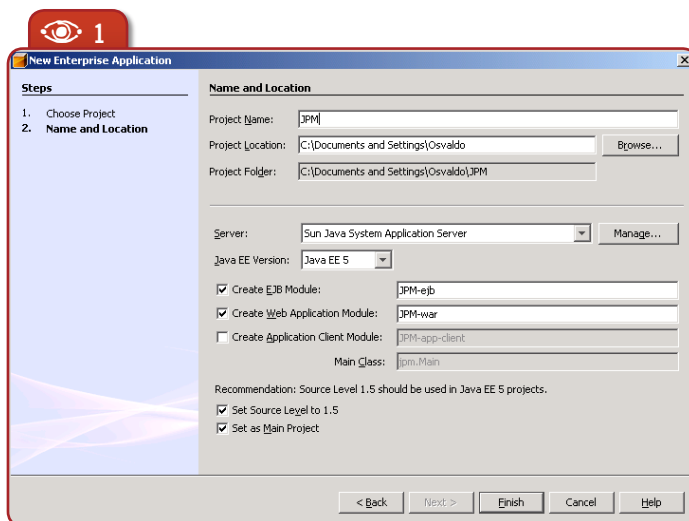
Implementing the Persistent Entities

Let's start coding the *JPM-ear* project, starting with the persistent entities. The application needs only three: Platform, JSR and Package. These will be persisted by the Java Persistence API (JPA).

The J2EE platform has been blamed for requiring a heavy load of bureaucratic artifacts even for simple tasks, so even a three-entity design like ours would be bloated with Entity Beans' home and bean interfaces, lifecycle methods and descriptors. Not to mention code that's not required (and not generated by IDEs) but is often needed in practice: a layer of POJOs to expose the Entity Beans to the outside world, DAOs to encapsulate the use of that code and convert between Entity Beans and POJOs, a Service Locator to provide resources like JNDI contexts, and so on.

This violates what I call **Complexity Scalability**. Simple problems should produce simple code, even if the framework supports much harder requirements like distributed transactions, clustering,

 **Figure 1.**
Creating
the Java EE
projects.



declarative security and other high-end features of EJB. A good framework is “pay as you go”, not requiring any code or configuration that’s not really necessary. As a corollary, a *great* framework will allow even complex requirements to map to relatively simple code: the bookkeeping overhead, measured as a proportion of the “useful” lines of code, should scale linearly.

So, if a very simple artifact (like a persistent entity) can be programmed with 30 lines of code and 3 lines of overhead (like descriptor, annotation, external O/R mapping, implementation of API interfaces, or equivalent), then the framework’s overhead is 10% and its complexity scalability looks good – since a simple task is performed with low overhead. For a much more complex artifact, like a thousand-line persistent class with many relationships, if the overhead is kept at 10% (100 lines), the complexity scalability is good; if it drops proportionally, e.g. to 3% (30 lines), it’s awesome.

It’s an interesting exercise to evaluate APIs and frameworks by this criterion. Take Hibernate, for example. The framework is regarded as easy to program, but its base overhead is high. The mapping files for “dumb” persistent classes (with no business logic) are often more complex than the classes themselves, because there’s no default logic at all. Every property and relationship, however trivial, must be explicitly mapped, and the mapping is highly redundant with the code. The same holds for Spring applications. These often contain so much XML metadata that one wonders how this is any easier than keeping the same information in Java code. (Note that I’m looking only at ease of program-


ming, not flexibility or other factors.)

In traditional J2EE, of course, we have the worst of the two worlds: an Entity Bean requires *both* a lot of code *and* a great deal of metadata. And its overhead scales linearly at best: complex apps require increased amounts of settings – for example, lots of resource references from beans to other beans, data sources, queues etc. – and the framework does nothing to avoid the need to declare settings explicitly. Thus complex apps easily end up with numerous, long descriptors.

The Persistence Unit

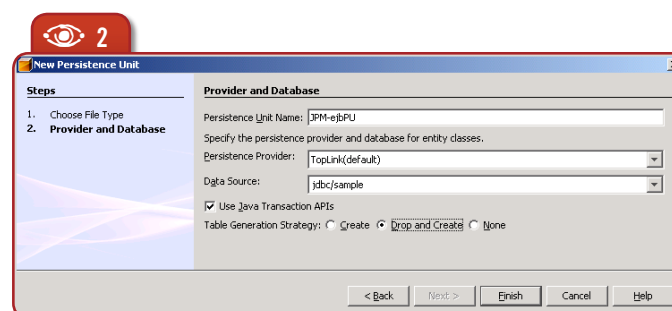
Before we can create any persistent entity, the JPA requires definition of a Persistence Unit. A P.U. is a set of JPA configurations (in other words, a descriptor). From the *JPM-ejb* project, run *New>File/Folder>Persistence>Persistence Unit* (**Figure 2**).


The only interesting option here is *Persistence Provider*: for SJSAS or GlassFish, select *TopLink(default)*. TopLink is an O/R mapping product from Oracle, but its Essentials version is open source. Users of other application servers may have to select a different provider. NetBeans comes preconfigured also for Hibernate and KODO.

 You can also ignore your application server’s default JPA provider and install a different one like Hibernate in SJSAS. This is a big advantage of Java EE 5: JPA supports “pluggable” persistence providers, instead of forcing you to go with each server’s built-in implementation like in EJB/CMP. The loosely-coupled design of JPA goes even further: you can use it in Java SE (without an application server), and you can make the Persistent Unit descriptor refer to external proprietary descriptors, like Hibernate’s *.hbm* files. This helps immensely in migrating legacy code. NetBeans supports this flexibility by offering multiple providers, and allowing you to configure new providers if necessary.

The resulting *Configuration Files/persistence.xml* file is a minimalist descriptor, only mandatory because it’s the only way to specify “global” properties like the persistence provider.

You could fill *persistence.xml* with the names of all persistent



 **Figure 2.** Creating a Persistence Unit.

classes, but this is not normally needed, as the default behavior is to automatically find all such classes (which is always good enough if your EJB module contains a single P.U.). Finally, the *persistence.xml* could include detailed mapping information, but this is also optional because we can do all the mapping with code annotations (or even more easily with default JPA behaviors).


The Platform entity

From the *JPM-ejb* project, select *New>Entity Class*. This brings up the dialog shown in **Figure 3**. Make *Class Name* = **Platform**, *Package* = **jpm.entity** and *Primary Key Type* = **String**. This wizard will generate the code in **Listing 1**. This is simply a POJO, but let's consider some important facts:

- The class uses some JPA annotations: **@Entity** makes it a persistent entity, **@Id** selects the attribute used for the primary key (PK), and **@GeneratedValue** declares that the value for this PK should be generated automatically.
- The entity must offer a default constructor, and JavaBeans-style getters and setters for its persistent attributes or relationships.


Only the code in boldface is mandatory. The rest – **hashCode()**, **equals()**, **toString()**, **@Override** annotations, implementation of **Serializable** – is just bonus from NetBeans. They are good programming practices, but not required by the JPA. We'll omit most of this non-required code to simplify future listings. The exception is **implements Serializable**: this is not required by the JPA but is necessary in our application because we want to transfer the entity objects through (possibly remote) EJB calls.

The entities will double as DTOs (data transfer objects)¹. In fact, one of the first advantages noticed when moving from a traditional ORM tool to a POJO-based one is that there's no need to write redundant POJOs / Value Objects / DTOs. You also don't need any code to convert between persistent and non-persistent objects, nor the structure (like the DAO pattern) to keep all that code organized.

 JPA only requires two kinds of classes to be serializable: classes that map composite PKs (with two or more columns), and the classes of attributes you may want to map to a LOB column. Anyway, you're better off if you follow the best practice that all POJOs should be serializable.

The *New Entity* wizard always generates the **@GeneratedValue** annotation, but our application uses only natural keys, so you must also

delete this annotation from the **id** attribute. In fact, **@GeneratedValue** is illegal here because only numeric PKs can be auto-generated.

 The entity's **hashCode()** and **equals()** methods are not necessary because the JPA will rely only on the PK for identification and hashing of entities. But if you use a custom class as primary key (which is required to support multi-column keys), your PK class must provide correct implementations of **hashCode()** and **equals()**.

JSR and Package entities

Using the same *New Entity* wizard, create two new classes: **JSR** (*Primary Key Type* = *int*) and **Package** (*Primary key type* = *String*). In the generated code, start by deleting the **@GeneratedValue** annotations. Next, rename the **id** attributes of these two classes to **JSR.number** and **Package.name**, respectively. Rename the getter and setters accordingly too.

You will notice that NetBeans balks at the new attribute names, showing a warning marker in the source editor with the hover message "*The column name of the persistent field is a SQL-99 keyword*". Yes, too bad – **number** and **name** are reserved SQL keywords, and this can cause problems in the generated tables because the JPA engine will map each attribute to a column with the same name, by default. Fix this by adding **@Column** annotations, e.g. **@Column(name="package_name")** to the **name** attribute, which will remove the warnings.

Relationships

Our **Platform** entity has a many-to-many relationship with the **JSR** entity (for example, JSR-224, the JAX-WS API, is part of both

¹ Remember, the term DTO implies a role: objects that can be transferred through a specific middleware; whereas POJO implies structure: objects whose implementation does not depend on specific APIs.

Java EE 5 and Java SE 6). Likewise, **JSR** has a one-to-many by-value aggregation relationship with **Package**. Both relationships are bidirectional.

Start declaring a **List<JSR> jsrs** attribute in **Platform**. Again NetBeans will show a warning: “*The multi-valued entity relation is not defined*”. If you left-click the attribute name or select the warning marker and right-click, NetBeans will offer two corrections for the problem. Select *Create bidirectional “ManyToOne” relationship*. This shows the wizard in **Figure 4**.

A bidirectional relationship requires that the entity in the other end, **JSR**, point back to **Platform**. Accept the *Create new field* option and *Name = platforms*. You must add the getters and setters manually for

the new **Platform.jsrs** and **JSR.platforms** relationships. It's useful also to have a constructor that receives only the PK and another receiving all attributes. The result should be close to **Listing 2**, where are highlighted the parts introduced after

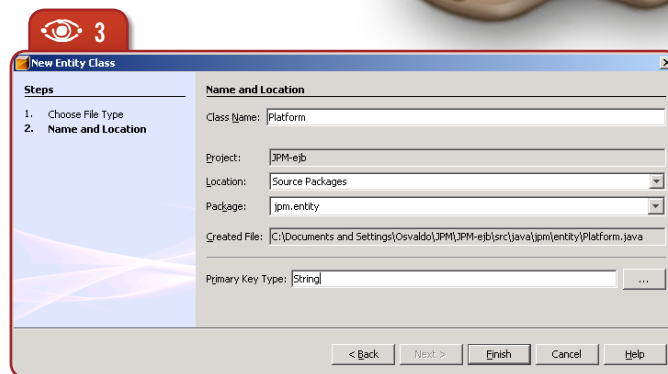


Figure 3. Creating an Entity Class.

Listing 1. Original code emitted for the Platform entity class (generated comments omitted).

```
package jpm.entity;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Platform implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private String id;
    public Platform() {
    }
    public String getId() {
        return this.id;
    }
    public void setId(String id) {
        this.id = id;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.id != null ? this.id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Platform)) {
            return false;
        }
        Platform other = (Platform)object;
        if (this.id != other.id && (this.id == null || !this.id.equals(other.id))) return false;
        return true;
    }
    @Override
    public String toString() {
        return "jpm.entity.Platform[id=" + id + "];"
    }
}
```



Figure 4.
Creating a
relationship.

creating the original entity code skeleton.

Note that the relationship wizard annotates `Platform.jsrs` with `@ManyToMany`, and the new `JSR.platforms` with `@ManyToMany(mappedBy="jsrs")`. This `mappedBy` property is intended to avoid ambiguities, should `Platform` contain multiple collections whose element-type is `JSR`. In our code this is not the case, so you could get rid of that `mappedBy`, unless you think it makes the code easier to read (in this case, you could add `mappedBy="platforms"` to `Platform.jsrs`'s annotation).

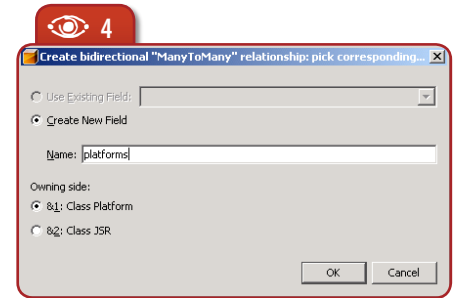
Now you can finish the `JSR` class by creating another bidirectional relationship with `Package`. This relationship will have a `OneToMany` cardinality, so `JSR` will have an attribute of type `List<Package>` but `Package` will only need a simple reference to a `JSR`. Listings 3 and 4 show the final code for these classes. I added a couple of tweaks to the relationship annotations (which will be explained later).

Listing 2. The complete `Platform` class.

```
package jpm.entity;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.persistence.*;
import static javax.persistence.FetchType.*;
import static javax.persistence.CascadeType.*;

@Entity
public class Platform implements Serializable
{
    @Id
    private String id;
    @ManyToMany(fetch=EAGER, cascade={
        MERGE, PERSIST, REFRESH})
    private List<JSR> jsrs;
    public Platform () {
    }
    public Platform (String id) {
        setId(id);
    }
    public Platform (String id, JSR... jsrs) {
        this(id);
        setJsrs(new ArrayList<JSR>(Arrays.asList(jsrs)));
    }
    public String getId () {
        return this.id;
    }
    public void setId (String id) {
        this.id = id;
    }
    public List<JSR> getJsrs () {
        return jsrs;
    }
    public void setJsrs (List<JSR> jsrs) {
        this.jsrs = jsrs;
    }
}
```



Testing the Deployment

The application's persistent entities are complete, so let's test them. In the `JPM` project run `Deploy project`. In a few seconds you should see messages reporting success in the console windows.

The deployment operation will also create the necessary tables, because we have set the Persistence Unit's `Table Generation Strategy to Drop and Create` (or `Create`). You can check this by connecting to the project's data source. If you're using the recommended JavaDB sample database in `SJSAS`, this will be predefined in `Runtime>Databases>jdbc:derby://localhost:1527/sample [app on APP]`. Just run this node's `Connect` command, log in with the password `app`, and you'll see the created tables. Notice, in particular, that an associative table, `PLATFORM_JSRS`, is created to map the `ManyToMany` relationship between `Platform` and `JSR`.

Annotations: Intelligence and Limitations

As this article intends also to make a critical review of Java EE 5's ease-of-use features, we can't stop here, implying that all persistence-related work will always be as easy as in our simple example.


Many developers starting to use annotation-enabled APIs think that the new APIs just shuffle things around – metadata that used to be written in XML descriptors is

now written in code annotations but they have to be written just like before, right?

Not really. As you can see in our example, we *don't* have to create annotations for all metadata previously in descriptors. For example, there are no annotations declaring that **JSR.description** is a persistent property, nor declaring the mapped table for each entity. And we only needed two **@Column** annotations for attributes whose names would map to illegal SQL identifiers. We could have been more verbose, writing code like:

```
@Entity
@Table(name="T_PLATFORM", schema="JPM")
public class Platform implements Serializable {
    ...
    @Column(name="PK_ID", nullable=false,
            length=512).
    private String id;
    ...
}
```

But these annotations are all optional. If they are missing, the JPA will look at the names and types of classes and attributes, so a class named **Platform** becomes a table named **PLATFORM**; a **String** attribute becomes a **VARCHAR** column, and so on. (If you want an entity class to contain an attribute *not* mapped to any column, tag it with the **@Transient** annotation).

 In principle, a framework based in descriptors, like EJB 2.1 or Hibernate, could also infer default configurations from the application classes. But the tight coupling of code and annotations, plus the availability of standard and built-in APIs and tools to manipulate annotated classes, make this strategy more robust, efficient and easy to implement.

Listing 3. The JSR class.

```
package jpm.entity;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.persistence.*;
import static javax.persistence.FetchType.*;
import static javax.persistence.CascadeType.*;

@Entity
public class JSR implements Serializable
{
    @Id
    @Column(name="num")
    private int number;
    private String description;
    @ManyToMany(mappedBy="jsrs")
    private List<Platform> platforms;
    @OneToMany(fetch=EAGER, cascade=ALL)
    private List<Package> packages;
    public JSR () {
    }
    public JSR (int id) {
        setNumber(id);
    }
    public JSR (
        int id, String description, Package... packages) {
        this(id);
        setDescription(description);
        setPackages(new ArrayList<Package>(
            Arrays.asList(packages)));
    }
    public int getNumber () {
        return this.number;
    }
    public void setNumber (int number) {
        this.number = number;
    }
    public String getDescription () {
        return description;
    }
    public void setDescription (String description) {
        this.description = description;
    }
    public List<Platform> getPlatforms () {
        return platforms;
    }
    public void setPlatforms (List<Platform> platforms) {
        this.platforms = platforms;
    }
    public List<Package> getPackages () {
        return packages;
    }
    public void setPackages (List<Package> packages) {
        this.packages = packages;
        if (packages != null) for (
            Package p: packages) p.setJsrs(this);
    }
}
```

Our sample application is also simplified by completely automatic DDL generation. Even if you have to write some annotations detailing types, table and column names, nullability and other properties, you

Listing 4. The Package class.

```

package jpm.entity;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Package implements Serializable
{
    @Id
    @Column(name="package_name")
    private String name;
    @ManyToOne
    private JSR jsr;
    public Package () {
    }
    public Package (String name) {
        setName(name);
    }
    public String getName () {
        return this.name;
    }
    public void setName (String name) {
        this.name = name;
    }
    public JSR getJsR () {
        return jsr;
    }
    public void setJsR (JSR jsr) {
        this.jsr = jsr;
    }
}

```

never have to collect these pieces into a **CREATE TABLE** statement. Thanks to automatic schema generation and SJSAS's embedded JavaDB database, we could write the whole app without even knowing that a relational database is being used! These features allow you to go far in your coding before you start worrying about "real-world" issues.

From Hello World to the Real World

Now, how realistic is the persistence code in this tutorial? In real applications you often have to comply with strict rules for database schemas; you may have to use legacy tables, or follow a style guideline that says something like "all tables should have the prefix 'T_'. In the average case, you'll be forced to tag every entity class with a **@Table** annotation, and every property with **@Column** or related annotations (**@JoinColumn** and **@JoinedColumns**, not used in this article). In the worst case, you may have to give up on the DDL generation feature, for example because your **CREATE TABLE** statement must be highly tuned with fine-grained (and usually proprietary) clauses, like physical allocation or partitioning options.

So in the worst case, our entity classes may be relatively polluted

by many annotations, or else we'll have to write DDLs and manually keep them in sync with the classes. But this doesn't mean the JPA's default behaviors are useless, but only that you pay as you go. The JPA (and other annotation-enabled APIs in Java EE 5) requires a programming effort that's proportional to the complexity of the problem being solved, and not artificially inflated by bookkeeping artifacts that must be written whether you need them or not.

Implementing the Session Bean

Now that we're done with persistence, the next step is to create a Session Bean that exposes methods to manipulate and query the persistent entities.

From the *JPM-ejb* project, run *New>Session Bean*. Make *EJBname=JavaPlatformManager* and *Package=business*. Accept defaults for *Session Type=Stateless* and *Create Interface=Local*, but check *Create Interface=Remote*. The wizard will create the skeletal sources for the Session Bean. These include the class **JavaPlatformManagerBean**, and the interfaces **JavaPlatformManagerLocal** and **JavaPlatformManagerRemote**. You'll just need to fill the blanks to obtain the code in **Listings 5** and **6**.

Listing 5 shows the local interface for the **JavaPlatformManager** bean. We have a **@Local** annotation, but we don't need to extend the interface **EJBLocalObject** anymore. So we have a POJI (*Plain Old Java Interface*).

The **JavaPlatformManagerRemote** interface (not listed) is identical, except that **@Local** is replaced by **@Remote**. When you have both local and remote interfaces

^{1,2} Using a single Java interface and annotating it with both **@Local** and **@Remote** could be even easier, but EJB 3.0 does not support this further simplification.

for a Session Bean, EJB 3.0 allows you to refactor all common methods to another interface, which is extended by both the Local and the Remote interfaces². This is a big improvement from EJB 2.x, where this refactoring and type unification was impossible because the remote interface was required to declare all methods with **throws RemoteException** (EJB 2.0: 7.10.5) but the local interface shouldn't use this exception (EJB 2.0: 7.10.7, 18.3.8, 18.6).

Listing 6 is the full implementation of our Stateless Session Bean, qualified as such by the **@Stateless** annotation. Notice how this class implements both the local and remote interfaces. In EJB 3.0, we finally have the complete strong-typing that EJB has lacked since its first version.

Now let's analyze the bean implementation. The **@PersistenceContext** annotation does dependency injection for the **EntityManager**, a JPA object that stands behind a Persistence Unit. When the **JavaPlatformManagerBean** class is instantiated by the container, the **EntityManager em** attribute will be initialized before any method can be invoked.

The method **listPlatforms()** runs a query that returns all **Platform** objects that were found in the database. These objects will have their **jsrs** collections populated, because we added the option **fetch = EAGER** to the relationship's annotation. The **EAGER** op-



 **Listing 5.** The Session Bean's Local interface.

```
package jpm.session;

import java.util.List;
import javax.ejb.Local;
import jpm.entity.Platform;

@Local
public interface JavaPlatformManagerLocal
    extends JavaPlatformManager {
    void createPlatform (Platform plat);
    List<Platform> listPlatforms ();
}
```

 **Listing 6.** The Session Bean's implementation.

```
package jpm.session;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.*;
import jpm.entity.Platform;

@Stateless
public class JavaPlatformManagerBean
    implements jpm.session.JavaPlatformManagerRemote, jpm.session.
        JavaPlatformManagerLocal
{
    @PersistenceContext private EntityManager em;
    public void createPlatform (Platform plat) {
        em.merge(plat);
    }
    public List<Platform> listPlatforms () {
        return em.createQuery(
            "select p from Platform p").getResultList();
    }
}
```



Java EE 5 SDK/
SJSAS, the Java
EE 5 Reference
Implementation.

tion in our example, forces a **Platform** entity's associated **JSRs** to be read when the **Platform** is read (possibly through a single query with an outer join, for better performance – a critical optimization if you need the associated objects most of the time you need the parent object). Likewise, the **JSR.packages** collections will also come initialized.

JPA versus EJB

The **fetch = EAGER** option is also important for correctness in the example application, because we return the **Platform** objects through a (possibly remote) EJB invocation. Entity instances associated with an **EntityManager** are managed by the JPA. If we used **fetch = LAZY** (the default), the query could return **Platform** objects with uninitialized **jsrs** collections. These would be populated on demand, upon first invocation of **getJsrs()**, but this only works for the original, JPA-managed collections.

When you transfer these objects through an EJB call, the application at the other side of the call receives different objects (reconstructed through cloning or serialization/deserialization). The new objects are not bound to an **EntityManager**. So when the receiver invokes **getJsrs()**, the JPA engine won't do its magic of intercepting the call and fetching the collection on demand. In fact, the receiver will often be an application – like our *JPM-war* – that doesn't contain a persistence unit, and hasn't even access to the entities' data source.

This issue is well-known to users of other POJO-based ORMs like Hibernate, and the solution is the same: eager fetching. (And just like in Hibernate, instead of tagging the relationship with a **fetch** option that affects all loads, you can write queries with a **FETCH JOIN** clause that gives finer control over when this loading strategy should be used.)

Who moved my lifecycle?

Just like for JPA, the “pay as you go” rule also holds for programming EJB 3.0 components. For example, our session bean didn't need to implement any of the traditional lifecycle methods, such as **ejbActivate()**. But that does not mean that lifecycle control is gone from Java EE. You can still hook into events like creation or activation, but these are now specified by annotations rather than interfaces. For example:

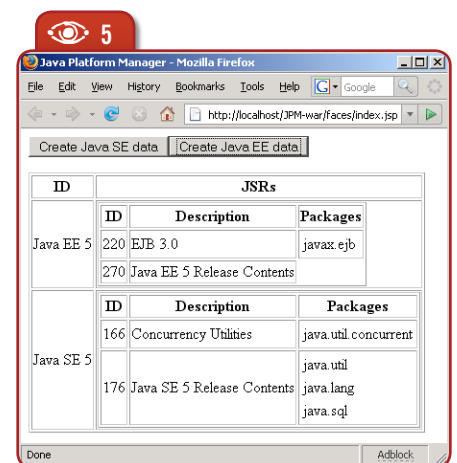
```
@PostActivate
public void postActivate (InvocationContext ctx)
{
    // Actions to execute after the activation
}
```

Annotations are finer-grained than interfaces like **SessionBean**: you can implement only one of the lifecycle methods, ignoring all others.

Differently from the JPA entities, however, when programming EJBs you'll find that most optional methods and annotations are rarely necessary, even in large, real-world applications. In my experience with good old J2EE, I rarely had to implement any of these lifecycle methods (well, if you don't count paranoid-logging implementations that just report “Bean XYZ was Activated”).



Figure 5.
The Web
GUI.



So the EJB 3.0 programming model results in much tighter code, even for realistic applications.

Implementing the Web client

Our application's back-end is ready. All that's missing is a GUI. I'm not much of a GUI programmer, but life has taught me that most people won't believe my code does

anything useful otherwise, so I'll provide a simple web GUI with the subproject *JPM-war*.

Start by configuring the NetBeans project to enable JSF. In the *JPM-war* project's properties page, select *Frameworks>Add*, and pick *JavaServer Faces*. This will populate the project with the minimum required configurations (including sample JSPs we won't use; delete those).

The Web GUI contains a single JSP that allows



 **Listing 7.** The Web GUI's view implementation, *index.jsp*.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
  <title>Java Platform Manager</title>
</head>
<body>
  <f:view>
    <h:form id="CreateTestData">
      <h:commandButton value="Create Java SE data" action="#{JPMClient.createJavaSE}"/>
      <h:commandButton value="Create Java EE data" action="#{JPMClient.createJavaEE}"/>
    </h:form>
    <h:dataTable value="#{JPMClient.allPlatforms}" var="platform" border="1">
      <h:column>
        <f:facet name="header">
          <h:outputText value="ID"/>
        </f:facet>
        <h:outputText value="#{platform.id}"/>
      </h:column>
      <h:column>
        <f:facet name="header">
          <h:outputText value="JSRs"/>
        </f:facet>
        <h:dataTable value="#{platform.jsrs}" var="jsr" border="1">
          <h:column>
            <f:facet name="header">
              <h:outputText value="ID"/>
            </f:facet>
            <h:outputText value="#{jsr.number}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Description"/>
            </f:facet>
            <h:outputText value="#{jsr.description}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Packages"/>
            </f:facet>
            <h:dataTable value="#{jsr.packages}" var="pack" border="0">
              <h:column>
                <h:outputText value="#{pack.name}"/>
              </h:column>
            </h:dataTable>
          </h:column>
        </h:dataTable>
      </h:column>
    </h:dataTable>
  </f:view>
</body>
</html>
```

 glassfish.dev.java.net

Project
GlassFish.

you to create sample **Platform** objects, and shows the ones already present in the database. **Listing 7** shows the source for this JSP, which is built with JSF component taglibs. **Figure 5** shows the page in action.

The JSP requires a Managed Bean – a POJO that is registered in the `WEB-INF/faces-config.xml` descriptor, so JSF-based pages can access the bean's properties and invoke their methods. Create it with `New>File/Folder>Web>JSF Managed Bean`; set `Class Name` to `JPMClient` and `Package` to `jpm.web`, accepting other defaults. This bean will provide the following services to our JSP:

- The property `allPlatforms` returns all **Platforms** (including child **JSR** objects and **Packages**), that the page will dump as nested HTML tables with JSF's `<h:dataTable>` components.
- The methods `createJavaSE()` and `createJavaEE()` are event handlers for JSF's `<h:commandButton>` component's **action** events. Each of these buttons triggers the insertion of one sample **Platform** object in the database (after which the page will be updated).

Listing 8 shows a simple controller implementation. There's no need of JNDI code to look up the Session Bean: just annotate it with `@EJB`, and dependency injection will do the rest. You don't even have to handle `RemoteException`, so the invocations to this EJB are

straightforward even though we're using a remote interface.

Looking again at **Listing 6**, check out the implementation of the EJB's `createPlatform()` method. You only have to pass the **Platform** object to `EntityManager.persist()`. The child **JSR** and **Package** objects are persisted by reachability. This happens because we declared the relationship `Platform.jsrs` with the option `cascade = {MERGE, PERSIST, REFRESH}`, and `JSR.packages` with `cascade = ALL`.

The `cascade` rule indicates which persistence operations are propagated by reachability to associated objects. For `Platform.jsrs` we used **MERGE** (synchronization of the object in memory with persistent records), **PERSIST** (creation) and **REFRESH** (reloading of persistent state to the object in memory). For `JSR.packages` we used **ALL**, which combines all the former with **DELETE** (dele-

 **Listing 8.** The Managed Bean.

```
package jpm.web;

import java.util.*;
import javax.ejb.EJB;
import jpm.entity.JSR;
import jpm.entity.Platform;
import jpm.entity.Package;
import jpm.session.JavaPlatformManagerRemote;

public class JPMClient {
    @EJB private JavaPlatformManagerRemote jpm;
    public void createJavaSE () {
        jpm.createPlatform(new Platform("Java SE 5",
            new JSR(176, "Java SE 5 Release Contents",
                new Package("java.lang"), new Package("java.util"), new Package("java.sql")),
            new JSR(166, "Concurrency Utilities", new Package("java.util.concurrent"))));
    }
    public void createJavaEE () {
        jpm.createPlatform(new Platform("Java EE 5",
            new JSR(270, "Java EE 5 Release Contents"),
            new JSR(220, "EJB 3.0", new Package("javax.ejb"))));
    }
    public List<Platform> getAllPlatforms () {
        return jpm.listPlatforms();
    }
}
```

tion of the persistent data).

The default is not to cascade any operation. You should of course use the most appropriate option for each kind of relationship. Here, **Platform**↔**JSR** is a simple association, so you could remove a **JSR** from a **Platform** without destroying the **JSR**, which is an independent entity (there are some **JSRs** that are part of multiple **Platforms**, and others that aren't part of any). But **JSR**↔**Package** is an aggregation by value. If you delete a **JSR**, its **Packages** should be deleted too.

Testing the complete application

Redeploy the application and load the following URL in your browser: `http://localhost/JPM-war/faces/index.jsp`. (The URL will be different if you've installed the application server to another HTTP port than 80 or used a different name for the *JPM-war* project). Click both buttons; after each click a new **Platform** will be inserted and displayed in the updated page. Don't click any button twice, as this would cause a PK violation in the database.

Our job is not done, however, before running the *Verify project* command on all projects. This submits each project to a massive number of Java EE-specific validations (in the results window I recommend selecting *Display = Failures and Warnings only*). Make sure you're clean on these validations to ensure both correctness and portability.

Look ma, no JNDI!

You may have noticed that we not only avoided the JNDI APIs, but also didn't even

have to specify JNDI names. This is because EJB 3.0 generates default JNDI names for all beans and even for resource references that are implicit in dependency injection annotations like `@EJB`.

You can ignore these defaults and force explicit JNDI names; for example, `@Stateless(name="ejb/session/JPM")` and `@EJB(name="ejb/session/JPM")`. But once again EJB 3.0 default behavior here is widely applicable. I've seen few EJB applications in which JNDI names are complex enough as not to allow use of Java EE 5 defaults. For example, for the *JPMClient.jpm* resource reference, the system will generate the default JNDI name `java:comp/env/jpm.web.JPMClient/jpm`, which, by combining the fully-qualified class name and field name, should be unique enough not to cause clashes.

Conclusions

Java EE 5 is a major ease-of-programming release, built around many novel design ideas: configuration by exception, POJO-oriented programming, annotation-driven APIs, dependency injection, a state-of-the-art persistency API, and a streamlined component model without hard-line rules like throwing **RemoteException**.

Nevertheless, Java EE is still a large, complex framework with a boatload of features – and good IDE support still makes a big difference. NetBeans 5.5 offers a full range of Java EE-specific support: code-generation wizards; visual descriptor editors; code completion, validation and automatic fixes for annotations; tight integration with application servers and building of Java EE deployment packages, as well as JSP debugging, support for both Struts and JSF, and many useful utilities like database browsing or HTTP monitoring. And we didn't even get started with the *really* advanced features, like SOA/BPEL tools in the Enterprise Pack. All these features come out-of-the-box, without need of dealing with third-party plug-ins.

The recent advances in the Java EE platform and the NetBeans IDE should not only make enterprise-grade development simpler, but also easier to learn and more incremental. Many tedious and error-prone tasks of the past are now gone, have been rationalized, or are highly automated by powerful IDE support. This translates directly to higher productivity for all developers and a gentler learning curve for beginners. ☺



Osvaldo Pinali Doederlein

(*opinali@gmail.com*) is a software engineer and consultant, working with Java since 1.0beta. An independent expert for the JCP, he has served for JSR-175 (Java SE 5), and is the Technology Architect for Visionaire Informatica, holding a MSc in Object Oriented Software Engineering. Osvaldo is a contributing editor for Java Magazine and blogs at *weblogs.java.net/blog/opinali*.