10 Reasons why Java EE Development doesn't need to be Painful

Alexis Moussine-Pouchkine
GlassFish Team
Sun Microsystems
#1 - Use modern APIs: Java EE 5

Java EE Compatibility

In every industry, businesses face the challenge of accommodating ever greater demands for high-speed data access, diverse clients, and secure transactions without incurring extensive additional costs. To extend existing IT investments while meeting these demands, developers have consistently adopted the Java Platform, Enterprise Edition.

The de-facto standard for delivering secure, robust, scalable multi-platform applications and services, Java EE technology and its success is predicated on compatibility which brings Java technology’s mission of “Write Once, Run Anywhere” capability to the server. Developers can write applications to the Java EE specification — and companies can purchase such applications -- and be assured that they are portable across all the Java compatible, Enterprise Edition products available today.

Java EE 5 Compatible Implementations

- Apache Geronimo 2.1.2
- WebLogic Server 10.0
- IBM WASCE 2.0
- IBM WebSphere Application Server v7
- Apaxis Application Server (v5.0)
- Oracle Application Server 11
- SAP NetWeaver 7.1
- Sun GlassFish Enterprise Server 3.1
- TmaxSoft JEDS 8
```java
public class ShoppingCartBean implements javax.ejb.SessionBean {
    private SessionContext ctx;
    DataSource productDB;

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate() {
        Context initialContext = new InitialContext();
        productDB = (DataSource) initialContext.lookup("java:comp/env/jdbc/myDB");
    }

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

    public Collection startToShop (String productName) {
        Connection conn = productDB.getConnection();
        ...
    }
}

public interface ShoppingCart extends EJBObject {
    public Collection startToShop (String productName)
        throws RemoteException;
    ...
}

public interface ShoppingCartHome extends EJBHome {
    ShoppingCart create()
        throws CreateException, RemoteException;
}

Context initialContext = new InitialContext();
ShopCartHome myCartHome = (ShopCartHome)
    initialContext.lookup("java:comp/env/ejb/cart");
ShopCart myCart = myCartHome.create();

// Utilisation du composant
Collection widgets = myCart.startToShop("widgets")
...

// Code pour gérer javax.ejb.CreateException
...

<session>
  <ejb-name>ShoppingCartBean</ejb-name>
  <local-home>ShopCartHome</local-home>
  <local>ShopCart</local>
  <ejb-class>com.example.ShopCartBean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  <resource-ref>
    <res-ref-name>jdbc/myDB</res-ref-name>
    <res-ref-type>javax.sql.DataSource</res-ref-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</session>

<assembly-descriptor>
...
</assembly-descriptor>
```
@Remote
public interface ShopCart {
    public Collection startToShop(String productName);
    public void checkout();
}

@Stateful
public class ShopCartBean implements ShopCart {
    @Resource DataSource productDB;
    @Init
    public Collection startToShop(String productName) {
        Connection conn = productDB.getConnection();
        ...
    }
    @Remove public void checkout(){ ... }
}
package endpoint;
import java.rmi.*;

public class HelloServiceImpl
    implements HelloServiceSEI {

    public String sayHello(String param)
        throws java.rmi.RemoteException {
            return "Hello " + param;
        }
    }

package endpoint;
import java.rmi.*;

public interface HelloServiceSEI
    extends java.rmi.Remote {

    public String sayHello(String param)
        throws java.rmi.RemoteException;
    }

<?xml version='1.0' encoding='UTF-8' ?>
<webservice xmlns="http://java.sun.com/xml/ns/j2ee" version='1.1'>
    <webservice-description>
        <webservice-description-name>
            HelloService</webservice-description-name>
        <wadl-file>
            WEB-INF/wadi/HelloService.wadl</wadl-file>
        </webservice-description>
        <jaxrpc-mapping-file>
            WEB-INF/HelloService-mapping.xml
        </jaxrpc-mapping-file>
    
    <port-component xmlns:webservice="urn:HelloService/wsdil">
        <port-component-name>HelloService</port-component-name>
        <webservice-port>wsdl-port_namespace=HelloServiceSEI</webservice-port>
        <service-endpoint-interface>
            endpoint=HelloServiceSEI</service-endpoint-interface>
        <service-impl-bean>
            <service-link>WSServlet_HelloService</service-link>
        </service-impl-bean>
    </port-component>
</webservice>
<configuration
    xmlns=http://java.sun.com/xml/ns/jax-rpc/nconfig>
    <service name="HelloService" targetNamespace="urn:HelloService/wsdl"
        typeNamespace="urn:HelloService/types" packageName="endpoint">
        <interface name="endpoint:HelloServiceSEI"
            servantName="endpoint:HelloServiceImpl"/>
    </service>
</configuration>
import javax.jws.WebService;
@WebService
public class MySimpleWS {
    public String sayHello(String s) {
        return "Hello " + s;
    }
}
• Simple JAX-WS Web Service
#2 - No runtime restarts

- *Don't lose your (developer) train of thought*
- *App servers shouldn't require to restart*
- *At least in development, use an app server runtime that:*
  - Does crash on PermGen OOM on redeploys
  - Doesn't require restart upon new or modified JNDI, and new or modified configuration (i.e. log levels)
- *Tomcat and others have made great progress in the past few releases*
#3 - Fast startup

- Sometimes, you do have to (re)start
- What is acceptable?
  - 2 minutes?
  - 30 seconds?
  - 5 seconds?
  - < 1 sec?
• AppServer Startup
#4 - Tooling

- **Compile on Save**
  - In Eclipse, now also in NetBeans (6.5)
  - Cut down on time to test changes
- **Save/Reload for JSPs**
  - Relatively easy given dynamic nature of JSP
- **Deploy on change**
  - Any Java EE artifact (not just JSPs)
  - Available in NetBeans 6.5 for various runtimes (GlassFish, Tomcat, ...)

• Tooling Demo
#5 - Preserve HTTP Session (across redeployments)

- Getting (back) to where you want to test the app is what takes most time

- **GlassFish v3 Prelude offers this**
  - Serialize sessions in memory, re-inject in new app

- **Not an IDE trick:**
  - `$ asadmin redeploy --properties keepSessions=true --name myap myapp.war`

- **JavaRebel offers similar “ZeroTurnaround” features**
Ease of Development for SOAP Web Services

- No XML, strong typing wherever possible

JAX-WS helps achieve this but one could use:

- Testing tools: soapUI, auto-generated client, ...
- Declarative QoS. No need to learn WS-*

Quality Of Service

- Optimize Transfer Of Binary Data (MTOM)
- Reliable Message Delivery
- Secure Service

Advanced ...
- Declarative QoS Web Services
• **Ease of Development for RESTful Web Services (JAX-RS)**
  - Create resources from entities, from data(bases), patterns (singleton, container-item, ...)
  - Testing clients (JavaScript, Ajax)
#7 - Tuning can be easy

- **Some stress testing could be done early**
- **Application top-down profiling**
- **GlassFish “Call-Flow”**
  - Integrated by default in GlassFish v2
  - Simple check-box to enable, no restart, no config
Jetty, JBoss, and now GlassFish offer an API to drive the runtime (start/stop, config, deploy).

Testing is an obvious use-case:

```java
@BeforeClass
public static void setUpClass () {
    AppServer glassfish = new AppServer(8080);
    glassfish.deployWar(warFile);
}

@Test
public void pingApplication () {
    ...
}
```

Other use-cases (mainly ISVs)
DEMO

- Embeddeable GlassFish
Essentially a portable DTrace
- A safe (read, not write), low-overhead, probe-based dynamic tracing tool

Annotations to define static troubleshooting interceptors ("probe points")
- Examples: @OnMethod, @OnTimer, @OnEvent, @OnExit, @OnError, @OnLowMemory

Some restrictions
- Cannot create objects, throw or intercept Exceptions, use loops, use synchronized statements, ...
// import all BTrace annotations
import com.sun.btrace.annotations.*;

// import statics from BTraceUtils class
import static com.sun.btrace.BTraceUtils.*;

//@BTrace annotation tells that this is a BTrace program
@BTrace
public class HelloWorld {

    //@OnMethod annotation tells where to probe.
    // In this example, we are interested in entry
    // into the Thread.start() method.
    @OnMethod(
        clazz=":java.lang.Thread",
        method="start"
    )
    public static void func() {
        // println is defined in BTraceUtils
        // you can only call the static methods of BTraceUtils
        println("about to start a thread!");
    }
}

% btrace <pid> <btrace-script>
#10 - CRUD in two steps

- Entities from Database
  - Standard EJB 3 + JPA
- Web pages from Entities
  - JSF pages for CRUD operations (Create, Read, Update, Delete)
• 2-step CRUD application
All of the above in one product

(Download) Size also matters

http://glassfish.org
With the proper tools and APIs, plain Java EE development can be painless and very productive.

It's not 2002 anymore!

Don't let your Rails/PHP/etc... friends call you fat :-)

Thanks for your attention!

alexis.mp@sun.com
http://blogs.sun.com/alexismp