CONTENTS

JAVA ENTERPRISE EDITION 7

BY ANDREW LEE RUBINGER, ARUN GUPTA, AND LINCOLN BAXTER, III

ABOUT THE PLATFORM

Enterprise software development is inherently complex; multi-user systems open the door to concerns such as transactional integrity, security, persistence integration, and interaction between components. Very simply put, the mission of the Java Enterprise Edition is to enable an out-of-the-box set of configurable services and functionality, which allow the programmer to write less and focus on delivering clean business logic.

Java EE 7 is in fact an aggregate of many interoperable technologies designed to deliver a unified experience. Application Servers which are certified to the standards defined by the Java Community Process are intended to service applications written to the specifications within the platform.

For the sake of brevity, this reference will focus on the key APIs of Java EE 7 most relevant to modern development.

JAVA PLATFORM, ENTERPRISE EDITION 7 (JAVA EE 7)

JSR-342

This umbrella specification ties together the various subsystems that comprise the platform, and provides additional integration support where necessary. The Java EE 7 platform added the following new subsystems:

- Java API for WebSocket 1.0
- Batch Applications for the Java Platform 1.0
- Java API for JSON Processing 1.0
- Concurrency Utilities for Java EE 1.0

In addition, Java API for RESTful Web Services 2.0 and Java Message Service 2.0 went through significant updates. Several other technologies were updated to provide improved productivity and features:

- Contexts and Dependency Injection 1.1
- Bean Validation 1.1
- Servlet 3.1
- Java Persistence API 2.1
- Java Server Faces 2.2
- Java Transaction API 1.2

There are many useful resources for Java EE 7 available online, both on Oracle's website and in the Java EE 7 community samples GitHub repository:

- Javadocs for the Java EE 7 API: docs.oracle.com/javaee/7/api/
- Technologies in Java EE 7: oracle.com/technetwork/java/javaee/tech
- Java EE 7 tutorial: docs.oracle.com/javaee/7/tutorial/doc/home.htm
- Java EE 7 samples: github.com/javaee-samples/javaee7-samples

JAVA API FOR WEBSOCKET 1.0

JSR-356

WebSocket provides a full-duplex, bi-directional communication over a single TCP channel. This overcomes a basic limitation of HTTP where a server can send updates to the client and a TCP connection is maintained between client and server until one of them explicitly closes it. WebSocket is also a very lean protocol and requires minimal framing on the wire. Java API for WebSocket 1.0 is a new addition to the platform and provides an annotated and programmatic way to develop, deploy, and invoke WebSocket endpoints.

Annotated server endpoints can be defined as:

```
@ServerEndpoint(value="/chat",
  encoders=ChatMessageEncoder.class,
  decoders=ChatMessageDecoder.class)
public class ChatServer {
  @OnMessage
  public String receiveMessage(ChatMessage message,
  Session client) {
        for (Session s : client.getOpenSessions()) {
            s.getBasicRemote().sendText(message);
        }
    }
}
```

Client endpoints are defined with `@ClientEndpoint` and can be connected to a server endpoint via:

```
WebSocketContainer container =
    ContainerProvider.getWebSocketContainer();
String uri = "ws://localhost:8080/chat/websocket";
container.connectToServer(ChatClient.class, URI.create(uri));
```

Alternatively, programmatic server endpoints can be defined as:

```
public class ChatServer extends Endpoint {
  @Override
  public void onOpen(Session session, EndpointConfig ec) {
    session.addMessageHandler(...);
  }
}
```

RED HAT
DEVELOPERS

MAKE JAVA EE WORK for YOU

Software development can be complex, but you can simplify your development when Java EE 7 is combined with a helpful community and a full suite of Java based technologies. As a part of Red Hat Developers you can try your hand at new solutions and put your Java EE experience into action.

Learn more. Code more. Share more with the Red Hat Developers program.

Join now: developers.redhat.com

redhat.

Programmatic endpoints must also be registered in a `ServerApplicationConfig` to assign their external URL:

```
@Override
public class MyServerApplicationConfig implements
   ServerApplicationConfig {
  @Override
  public Set<ServerEndpointConfig> getEndpointConfigs(
     Set<Class<? extends Endpoint>> set) {
       return new HashSet<ServerEndpointConfig>() {
         {
           add(ServerEndpointConfig.Builder.create(
             ChatServer.class, "/chat").build());
         }
       };
     }
}
```

PUBLIC API FROM JAVAX.WEBSOCKET:

CLASS NAME	DESCRIPTION
@ServerEndpoint	Annotation that decorates a POJO as a WebSocket server endpoint
@ClientEndpoint	Annotation that decorates a POJO as a WebSocket client endpoint
@PathParam	Annotation that assigns parameters of an endpoint URI-template
Session	Represents a conversation between two WebSocket endpoints
Encoders	Interface to convert application objects into WebSocket messages
Decoders	Interface to convert WebSocket objects into application messages
MessageHandler	Interface to receive incoming messages in a conversation
@OnMessage	Annotation to make a Java method receive incoming message
@OnOpen	Decorates a Java method to be called when connection is opened
@OnClose	Decorates a Java method to be called when connection is closed
@OnError	Decorates a Java method to be called when there is an error

BATCH APPLICATIONS FOR THE JAVA PLATFORM 1.0

JSR-352

The Batch Applications specification allows developers to easily define non-interactive, bulk-oriented, long-running tasks in item-oriented and task-oriented ways. It provides a programming model for batch applications and a runtime for scheduling and executing batch jobs. It supports item-oriented processing using Chunks and task-oriented processing using Batchlets. Each job is defined using Job Specification Language, a.k.a. "Job XML," that must be created in the META-INF/batch-jobs/ folder (subdirectories of this folder are ignored).

Example `META-INF/batch-jobs/my-job.xml` file:

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  version="1.0">
    <step id="myStep">
        <chunk item-count="3">
            <reader ref="myItemReader"/>
            <processor ref="myItemProcessor"/>
            <writer ref="myItemWriter"/>
        </chunk>
```

Each item is read, processed, and aggregated for writing. `item-count` number of items are processed within a container-managed transaction.

Different elements define the sequence of a job:

- Step: Independent and sequential phase of a job
- Flow: Sequence of execution elements that execute together as a unit
- Split: Set of flows that execute concurrently
- Decision: Customized way of determining sequencing among step, flows, and splits

PUBLIC API FROM JAVAX.WEBSOCKET:

CLASS NAME	DESCRIPTION
BatchRuntime	Represents the JSR 352 runtime
JobOperator	Interface for operating on batch jobs
ItemReader	Batch artifact that reads item for chunk processing
ItemProcessor	Batch artifact that operates on an input item and produces an output item for chunk processing
ItemWriter	Batch artifact that writes to a list of item for chunk processing
Batchlet	Batch step for item-oriented processing
@BatchProperty	Annotation to define a batch property injected from Job XML
JobListener	Interface that intercepts job execution
StepListener	Interface that intercepts step execution
ChunkListener	Interface that intercepts chunk processing
CheckpointAlgorithm	Provides a custom checkpoint policy for chunk steps
PartitionMapper	Provide unique batch properties for each batch execution
PartitionReducer	Unit of work demarcation across partitions

JAVA API FOR JSON PROCESSING 1.0

JSR-353

JSON is a lightweight data-interchange format and is the common format of the web for consuming and creating web services data. A new API to parse and generate JSON is included in the platform, which reduces the dependency on third-party libraries. The API offers to produce/consume JSON text in a low-level streaming fashion (similar to StAX API for XML).

```
JsonParser parser = Json.createParser(new FileInputStream(...));
```

The event-based parser allows an application developer to ask for an event using `parser.next()`. This API returns a `JsonParser.Event`, which consists of values that indicate the start of an object, end of an object, and other similar events. This gives a more procedural control over processing the JSON.

The streaming API can generate a simple JSON object as:

```
StringWriter w = new StringWriter();
JsonGenerator gen = factory.createGenerator(w);
gen
    .writeStartObject()
    .write("apple", "red")
    .write("banana", "yellow")
    .writeEnd();
```

The API also offers a high-level Object Model API that provides immutable object models for JSON object and array structures. These JSON structures are represented as object models via the Java types `JsonObject` and `JsonArray`.

```
JsonReader reader = Json.createReader(new FileInputStream(...));
JsonObject json = reader.readObject();
```

The Object Model API can generate a simple JSON object as:

```
Json.createObjectBuilder()
      .add("apple", "red")
      .add("banana", "yellow")
.build();
```

PUBLIC API FROM JAVAX.JSON:

JSON	FACTORY CLASS FOR CREATING JSON PROCESSING OBJECTS
JsonObject	Represents an immutable JSON object value
JsonArray	Represents an immutable JSON array
JsonWriter	Writes a JSON object or array structure to an output source
JsonReader	Reads a JSON object or array structure from an input source
JsonGenerator	Writes JSON data to an output source in a streaming way
JsonParser	Provides forward, read-only access to JSON data in a streaming way
JsonParser.EVENT	An event from JsonParser

CONCURRENCY UTILITIES FOR JAVA EE 1.0

JSR-236

Concurrency Utilities for Java EE provides a simple, standardized API for using concurrent programming techniques from application components without compromising container integrity, while still preserving the Java EE platform's fundamental benefits. This API extends the Concurrency Utilities API developed under JSR-166 and found in the Java 2 Platform, Standard Edition 7 in the `java.util.concurrent` package.

Four managed objects are provided:

- `ManagedExecutorService`
- `ManagedScheduledExecutorService`
- `ManagedThreadFactory`
- `ContextService`

A new default instance of these objects is available for injection in the following JNDI namespace:

- `java:comp/DefaultManagedExecutorService`
- `java:comp/DefaultManagedScheduledExecutorService`
- `java:comp/DefaultManagedThreadFactory`
- `java:comp/DefaultContextService`

A new instance of `ManagedExecutorService` can be created as:

```
InitialContext ctx = new InitialContext();
ManagedExecutorService executor = (ManagedExecutorService)ctx.
  lookup("java:comp/DefaultManagedExecutorService");
```

It can also be injected as:

```
@Resource(lookup="java:comp/DefaultManagedExecutorService")
ManagedExecutorService executor;
```

An `ExecutorService` obtained via any of these methods can then be used as expected:

```
executor.invokeAll(...);
executor.shutdown();
```

An application-specific `ManagedExecutor` can be created by adding the following fragment to web.xml:

```
    <resource-env-ref>
      <resource-env-ref-name>
        concurrent/myExecutor
      </resource-env-ref-name>
      <resource-env-ref-type>
        javax.enterprise.concurrent.ManagedExecutorService
      </resource-env-ref-type>
    </resource-env-ref>
```

PUBLIC API FROM JAVAX.ENTERPRISE.CONCURRENT:

ManagedExecutorService	Manageable version of ExecutorService
ManagedScheduledExecutorService	Manageable version of ScheduledExecutorService
ManagedThreadFactory	Manageable version of ThreadFactory

ContextService	Provides methods for creating dynamic proxy objects with contexts in a typical Java EE application
ManagedTaskListener	Used to monitor the state of a task's Future

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS)

JSR-339

JAX-RS provides a standard annotation-driven API that helps developers build a RESTful web service and invoke it. The standard principles of REST—such as identifying a resource via URI; defining a set of methods to access the resource; and allowing for multiple representation formats of a resource—can be easily marked in a POJO via annotations.

JAX-RS 2 adds a Client API that can be used to access web resources. The API provides support for standard HTTP verbs, custom media types by integration with entity providers, and dynamic invocation:

```
Author author = ClientBuilder
                    .newClient()
                    .target("http://localhost:8080/resources/authors")
                    .path("{author}")
                    .resolveTemplate("author", 1)
                    .request
                    .get(Author.class);
```

The Client API also permits retrieval of the response asynchronously by adding an `async()` method call before the method invocation.

```
Author author = ClientBuilder.newClient().target(...).async().
   get(Author.class);
```

JAX-RS enables the creation of server-side endpoints using annotations. To enable JAX-RS, create a class that extends Application and specify the root server endpoint URL using the `@ApplicationPath` annotation:

```
@ApplicationPath("/webapi")
public class MyApplication extends Application {
}
```

JAX-RS 2 introduces asynchronous endpoints that may be defined to suspend the client connection if the response is not readily available and later resume when it is. Here, an Executor is used to perform work in a separate thread:

```
@Path("authors")
public class AuthorEndpoint {
    @GET
    public void getAll(@Suspended final AsyncResponse ar) {
        executor.submit(new Runnable() {
            public void run() {
               ar.resume(authorsDatabase.getAll());
            }
        ));
    }
}
```

A synchronous endpoint that uses `@PathParam` to identify a single book. Note that this endpoint's `@Path` will be combined with the `@ApplicationPath`; this results in a complete endpoint path of "`/webapi/books/{id}`":

```
@Path("books/{id}")
public class BookEndpoint {
    @GET
    public Book get(@PathParam("id") final int bookId) {
       return booksDatabase.get(bookId);
    }
}
```

Filters and Entity Interceptors are extension points that customize the request/response processing on the client and the server side. Filters are mainly used to modify or process incoming and outgoing request or response headers. A filter is defined by implementing `ClientRequestFilter`, `ClientResponseFilter`, `ContainerRequestFilter`, and/or `ContainerResponseFilter`.

```
public class HeaderLoggingFilter implements ClientRequestFilter,
  ClientResponseFilter {
    // from ClientRequestFilter
    public void filter(ClientRequestContext crc) throws IOException {
        for (Entry e : crc.getHeaders().entrySet()) {
             … = e.getKey();
             … = e.getValue();
        }
    }

    // from ClientResponseFilter
    public void filter(ClientRequestContext crc,
  ClientResponseContext crc1) throws IOException {
        ...
    }
}
```

Entity Interceptors are mainly concerned with marshaling and unmarshaling of HTTP message bodies. An interceptor is defined by implementing `ReaderInterceptor` or `WriterInterceptor`, or both.

```
public class BodyLoggingFilter implements WriterInterceptor {
    public void aroundWriteTo(WriterInterceptorContext wic) {
        wic.setOutputStream(...);
        wic.proceed();
    }
}
```

JAX-RS 2 also enables declarative validation of resources using Bean Validation 1.1. In addition to validating any `@PathParam` or `@QueryParam` method parameters, validation constraints may also be placed on the JAX-RS resource class fields:

```
@Path("/names")
public class Author {
    @NotNull @Size(min=5)
    private String firstName;
    ...
}
```

PUBLIC API SELECTION FROM JAVAX.WS.RS PACKAGE:

CLASS NAME	DESCRIPTION
AsyncInvoker	Uniform interface for asynchronous invocation of HTTP methods
ClientBuilder	Entry point to the Client API
Client	Entry point to build and execute client requests
ClientRequestFilter	Interface implemented by client request filters

CLASS NAME	DESCRIPTION
ClientResponseFilter	Interface implemented by client response filters
ContainerRequestFilter	Interface implemented by server request filters
ContainerResponseFilter	Interface implemented by server response filters
ConstrainedTo	Indicates the runtime context in which a JAX-RS provider is applicable
Link	Class representing hypermedia links
ReaderInterceptor	Interface for message body reader interceptor
WriterInterceptor	Interface for message body writer interceptor
NameBinding	Meta-annotation used for name binding annotations for filters and interceptors
WebTarget	Resource target identified by a URI

JAVA MESSAGE SERVICE 2.0

JSR-343

Message-oriented middleware (MOM) allows sending and receiving messages between distributed systems. JMS is a MOM that provides a way for Java programs to create, send, receive, and read an enterprise system's messages.

A message can be easily sent as the following example:

```
@Stateless
@JMSDestinationDefinition(name="...", interfaceName="javax.jms.
  Queue")
public class MessageSender {
    @Inject
    JMSContext context;
    @Resource(mappedName=...)
    Destination myQueue;

    public void sendMessage() {
        context.sendProducer().send(myQueue, message);
    }
}
```

The newly introduced, simplified API is very fluent, uses runtime exceptions, is annotation-driven, and makes use of CDI to reduce the boilerplate code.

A message can be received as:

```
public void receiveMessage() {
    String message = context.createConsumer(myQueue).
  receiveBody(String.class, 1000);
}
```

JMS 2.0 queues and topics may now be configured at deployment time using the `@JMSDestinationDefinitions` annotation. Queues may be injected using the `@Resource` annotation:

```
@JMSDestinationDefinitions(
        value = {
            @JMSDestinationDefinition(name = "java:/queue/my-queue",
                    interfaceName = "javax.jms.Queue",
  destinationName = "my-queue"),
            @JMSDestinationDefinition(name = "java:/topic/my-topic",
                    interfaceName = "javax.jms.Topic",
  destinationName = "my-topic")
        }
)
@Stateless
public class MessageSender {
    @Resource(lookup = "java:/queue/my-queue")
    private Queue queue;
}
```

PUBLIC API FROM JAVAX.JMS:

JMSContext	Main interface to the simplified API
JMSProducer	Simple object used to send messages on behalf of JMSContext
JMSConsumer	Simple object used to receive messages from a queue or topic
@JMSConnectionFactory	Annotation used to specify the JNDI lookup name of ConnectionFactory
@JMSDestinationDefinition	Annotation used to specify dependency on a JMS destination
QueueBrowser	Object to look at messages in client queue without removing them

CONTEXTS AND DEPENDENCY INJECTION FOR JAVA (CDI 1.1)

JSR-346

The Java Contexts and Dependency Injection specification (CDI) introduces a standard set of application component management services to the Java EE platform. CDI manages the lifecycle and interactions of stateful components bound to well-defined contexts. CDI provides typesafe dependency injection between components and defines: interceptors and decorators to extend the behavior of components; an event model for loosely coupled components; and an SPI allowing portable extensions to integrate cleanly with the Java EE environment.

The Java EE 7 platform, by default, enables CDI injection for all Java classes that explicitly contain a CDI scope or EJB annotation. The following are examples of beans that would be automatically enabled for injection using default settings.

```
@Dependent // CDI scope annotation
public class ExampleBean {
}
@Stateful // EJB annotation
public class ExampleBean {
}
```

To configure non-default bean discovery settings, a new "bean-discovery-mode" attribute is added to beans.xml (located in /WEB-INF/beans.xml in web applications, and /META-INF/beans.xml in JAR libraries):

```
<beans
        xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
        bean-discovery-mode="all">

</beans>
```

This attribute can take the following values:

- **all:** All types in the archive are considered for injection
- **annotated (default):** Only types with an explicitly declared CDI scope are considered for injection
- **none:** Disable CDI

By default, CDI interceptors are disabled. They can be enabled and ordered via the `@javax.interceptor.Interceptor.Priority` annotation as shown (the same can be done for decorators and alternatives):

```
@Priority(Interceptor.Priority.APPLICATION+10)
@Interceptor
@Logging
public class LoggingInterceptor {
    //. . .
}
```

In addition, CDI 1.1 also introduces the following features:

- Support for `@AroundConstruct` lifecycle callback for constructors
- Binding interceptors to constructors
- Interceptor binding moved to the interceptors spec, allowing for reuse by other specifications
- Support for decorators on built-in beans
- `EventMetadata` to allow inspection of event metadata
- `@Vetoed` annotation allowing easy programmatic disablement of classes
- Many improvements for passivation capable beans, including `@TransientReference` allowing instances to be retained only for use within the invoked method or constructor
- Scope activation and destruction callback events
- `AlterableContext`, allowing bean instances to be explicitly destroyed
- Class exclusion filters to beans.xml to prevent scanning of classes and packages
- `Unmanaged`, allowing easy access to non-contextual instances of beans
- Easy access to the current CDI container
- `AfterTypeDiscovery` event, allowing extensions to register additional types after type discovery
- `@WithAnnotations`, a way of improving extension loading performance

BEAN VALIDATION 1.1

JSR-349

Expanded in Java EE 7, the Bean Validation Specification provides for unified declaration of validation constraints upon bean data. It may be used to maintain the integrity of an object at all levels of an application: from user form input in the presentation tier (JSF) all the way to the persistence layer (JPA).

New in the 1.1 release is:

- Method-level validation (validation of parameters or return values)
- Dependency injection for Bean Validation components
- Group conversion in object graphs
- Error message interpolation using EL expressions
- Support for method and constructor validation (via CDI, JAX-RS, etc.)
- Integration with CDI (`Validator` and `ValidatorFactory` injectable instances, `ConstraintValidator` instances being CDI beans and thus accept `@Inject`, etc.)

Here we show how we may apply Bean Validation constraints in a declarative fashion to ensure the integrity of a User object.

```
public class User {
    @NotNull
    @Size(min=1, max=15)
    private String firstname;

    @NotNull
    @Size(min=1, max=30)
    private String lastname;

    @Pattern(regexp="\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b")
    public String email;
}
```

Custom constraints can be created to provide reusable annotations with specific validation definitions:

```
@Pattern(regexp = "[a-zA-Z0-9-]+")
@Constraint(validatedBy = {})
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
public @interface Username {
    String message() default "Invalid username: must only contain
  letters";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

To enable internationalization and localization, custom constraint messages may reference resource bundle properties:

```
String message() default "{validation.invalid_username}";
```

Custom constraints may also use custom `ConstraintValidators`. This is specified using the `validatedBy` attribute:

```
@Constraint(validatedBy = UsernameValidator.class)
```

JAVA SERVLET 3.1

JSR-340

Servlet technology models the request/response programming model, and is commonly used to extend HTTP servers to tie into server-side business logic. In Servlet 3.1, the specification has been expanded over previous versions to include support for non-blocking I/O, security, and API enhancements.

To register a servlet in Java EE 7, annotate an implementation of HttpServlet with @WebServlet (the web descriptor WEB-INF/web.xml is no longer required):

```
@WebServlet("/report")
public class MoodServlet extends HttpServlet {
    ...
}
```

NON-BLOCKING I/O

Servlet 3.1 introduces a non-blocking I/O (NIO) API, which can be used to read input as it becomes available from the request via a `ReadListener`:

```
AsyncContext context = request.startAsync();
ServletInputStream input = request.getInputStream();
input.setReadListener(new ReadListener(){...});
```

The `ReadListener` interface defines three callbacks:

- `onAllDataRead()` Invoked when all data for the current request has been read
- `onDataAvailable()` When an instance of the `ReadListener` is registered with a `ServletInputStream`, this method will be invoked by the container the first time it is possible to read data
- `onError(Throwable t)` Invoked when an error occurs processing the request

SECURITY ENHANCEMENTS

- Applying run-as security roles to `#init` and `#destroy` methods
- Added `HttpSessionIdListener` to enable detection of session fixation attacks, and enabled deterrence of session fixation attacks by enabling changing the session ID via `HttpServletRequest.changeSessionId()`
- Addition of `<deny-uncovered-http-methods/>` to web.xml in order to block HTTP methods not covered by an explicit constraint

MISCELLANEOUS ADDITIONS TO THE API

- `ServletResponse.reset()` clears any data that exists in the buffer as well as the status code, headers. In addition, Servlet 3.1 also clears the state of `getServletOutputStream()` and `getWriter()`
- `ServletResponse.setCharacterEncoding(...)` sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8.
- Relative protocol URL can be specified in `HttpServletResponse.sendRedirect`. This will allow a URL to be specified without a scheme. That means instead of specifying "http://anotherhost.com/foo/bar.jsp" as a redirect address, "//anotherhost.com/foo/bar.jsp" can be specified. In this case the scheme of the corresponding request will be used.

JAVA PERSISTENCE 2.1

JSR-338

Most enterprise applications will need to deal with persistent data, and interaction with relational databases can be a tedious and difficult endeavor. The Java Persistence API (JPA) specification aims to provide an object view of backend storage in a transactionallyaware manner. By dealing with POJOs, JPA enables developers to perform CRUD operations without the need for manually tuning SQL.

To activate JPA, a META-INF/persistence.xml file must be present in the JAR or WAR in which persistence functionality is to be used; however, a default data-source is now provided, simplifying testing and development.

Example META-INF/persistence.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
       http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="MyPU" transaction-type="JTA">
    </persistence-unit>
</persistence>
```

NEW IN JPA 2.1 IS:

Support for Stored Procedures

We may use the new `@NamedStoredProcedureQuery` annotation atop an entity to define a stored procedure:

```
@Entity
@NamedStoredProcedureQuery(name="newestBlogsSP",
  procedureName="newestBlogs")
public class Blog {...}
```

From the client side, we may call this stored procedure like:

```
StoredProcedureQuery query = EntityManager.
  createNamedStoredProcedureQuery("newestBlogsSP");
query.registerStoredProcedureParameter(1, String.class,
  ParameterMode.INOUT);
query.setParameter(1, "newest");
query.registerStoredProcedureParameter(2, Integer.class,
  ParameterMode.IN);
query.setParameter(2, 10);
query.execute();
String response = query.getOutputParameterValue(1);
```

Bulk Update and Delete via the Query API

`CriteriaUpdate`, `CriteriaDelete`, and `CommonAbstractQuery` interfaces have been added to the API, and the `AbstractQuery` interface has been refactored.

A spec example of a bulk update may look like:

```
CriteriaUpdate<Customer> q = cb.createCriteriaUpdate(Customer.class);
Root<Customer> c = q.from(Customer.class);
q.set(c.get(Customer_.status), "outstanding")
  .where(cb.lt(c.get(Customer_.balance), 10000));
```

Entity Listeners Using CDI

Listeners on existing entity events `@PrePersist`, `@PostPersist`, `@PreUpdate`, and `@PreRemove` now support CDI injections, and entity listeners may themselves be annotated with `@PostConstruct` and `@PreDestroy`.

Synchronization of Persistence Contexts

In JPA 2, the persistence context is synchronized with the underlying resource manager. Any updates made to the persistence context are propagated to the resource manager. JPA 2.1 introduces the concept of unsynchronized persistence context. Here is how you can create a container-managed unsynchronized persistence context:

```
@PersistenceContext(synchronization=SynchronizationType.
  UNSYNCHRONIZED) EntityManager em;
```

JAVA SERVER FACES 2.2

JSR-344

JavaServer Faces (JSF) is a user interface (UI) framework for the development of Java web applications. Its primary function is to provide a component-based toolset for easily displaying dynamic data to the user. It also integrates a rich set of patterns to help manage state and promote code reuse.

JSF 2.2 is activated automatically when any JSF *.xhtml file (referred to as a view) is detected in either the web-root directory, or the WEB-INF/resources/ directory. View files may be placed in subdirectories.

Example /catalog/item.xhtml page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

    <f:metadata>
        <f:viewParam id="id" name="item" value="#{catalog.itemId}"/>
        <f:viewAction action="#{catalog.loadItem}"/>
    </f:metadata>
    <h:head>
        <title>Catalog Index</title>
    </h:head>
    <h:body>
        You requested catalog item: <h:outputText value="#{catalog.
  item.name}"/>
    </h:body>
</html>
```

The above page requires additional functionality for the Expression Language (EL) references, such as:

- viewParam value=”#{catalog.itemId}”,
- viewAction action=”#{catalog.loadItem}”
- outputText value=”#{catalog.item}”

Below is an example of a CDI managed bean (referred to as a backing bean) that provides the required functionality for the view:

```
@Named("catalog")
@RequestScoped
public class CatalogViewPage {
  private int id;
  private Item item;

  public String loadItem() {
    item = // load the item (which contains a ‘name’ field).
    return null;
  }

  // Getters and setters...
}
```

JSF 2.2 introduces Faces Flows, which provide an encapsulation of related pages and corresponding backing beans as a module. This module has well-defined entry and exit points defined by the application developer. Each flow resides in its own folder within the web-root directory, and contains a *-flow.xml file. Flow XML files may be empty if no configuration is required.

Example flow layout:

```
src/main/webapp/
    catalog/
        item.xhtml
    purchase/
        purchase-flow.xml
        1-cart.xhtml
        2-checkout.xhtml
        3-confirm.xhtml
```

The newly introduced CDI `@FlowScoped` annotation defines the scope of a bean in the specified flow. This enables automatic activation/passivation of the bean when the scope is entered/exited:

```
@FlowScoped("flow1")
public class MyFlow1Bean {
    String address;
    String creditCard; //. . .
}
```

Anew EL object for flow storage, `#{flowScope}`, is also introduced.

```
<h:inputText id="input" value="#{flowScope.value}" />
```

Flows may also be defined as CDI classes with producer methods annotated with `@FlowDefinition`:

```
class CheckoutFlow implements Serializable {
    @Produces
    @FlowDefinition
    public Flow defineFlow(@FlowBuilderParameter FlowBuilder
  flowBuilder) {
        ...
    }
}
```

JSF 2.2 defines Resource Library Contracts, a library of templates and associated resources that can be applied to an entire application in a reusable and interchangeable manner. A configurable set of views in the application will be able to declare themselves as template-clients of any template in the resource library contract.

JSF 2.2 introduces passthrough attributes, which allow listing of arbitrary name/value pairs in a component that are passed straight through to the user agent without interpretation by the UIComponent or Renderer.

Example passthrough attribute:

```
<html ... xmlns:p="http://xmlns.jcp.org/jsf/passthrough"
...
   <h:form prependId="false">
   <h:inputText id="itemId" p:type="number" value="#{catalog.id}"
             p:min="1" p:max="1000000" p:required="required"
             p:title="Enter an Item ID">
        ...
```

This will cause the following output HTML to be generated:

```
<input id="itemId" type="number" value="1" min="1" max="1000000"
       required="required"  title="Enter an Item ID">
```

PUBLIC API FROM JAVAX.FACES.*:

Flow	Runtime representation of a Faces Flow
@FlowScoped	CDI scope that associates the bean to be in the scope of the specified Flow

JAVA TRANSACTION API 1.2

JSR-344

The Java Transaction API enables distributed transactions across multiple X/Open XA resources such as databases and message queues in a Java application. The API defines a high-level interface, annotation, and scope to demarcate transaction boundaries in an application.

The `UserTransaction` interface enables the application to control transaction boundaries programmatically by explicitly starting and committing or rolling back a transaction.

```
@Inject UserTransaction ut;

ut.begin();
…
ut.commit();
```

`ut.rollback()` can be called to rollback the transaction.

JTA 1.2 introduces the `@javax.transaction.Transactional` annotation that enables one to declaratively control transaction boundaries on POJOs. This annotation can be specified at both the class and method level, where method-level annotations override those at the class level.

```
@Transactional
class MyBean {
   . . .
}
```

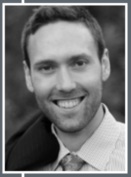
All methods of this bean are executed in a transaction managed by the container. This support is provided via an implementation of CDI interceptors that conduct the necessary suspending, resuming, etc.

JTA 1.2 also introduces a new CDI scope `@TransactionScoped`. This scope defines a bean instance whose lifecycle is scoped to the currently active JTA transaction.

PUBLIC API FROM JAVAX.TRANSACTION.*:

UserTransaction	Allows an application to explicitly manage application boundaries
@Transactional	Annotation that provides the application the ability to declaratively control transaction boundaries on CDI-managed beans, as well as classes defined as managed beans
@TransactionScoped	Annotation that specifies a standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction

ABOUT THE AUTHOR



LINCOLN BAXTER, III (@lincolnthree) is the Chief Editor of Red Hat Developers, and has worked extensively on JBoss open-source projects—most notably as creator & project lead of JBoss Forge, author of Errai UI, and Project Lead of JBoss Windup.

He is a founder of OCPsoft, the author of PrettyFaces and Rewrite, the leading URL-rewriting extensions for Servlet, Java EE, and Java web frameworks; he is also the author of PrettyTime, social-style date and timestamp formatting for Java. When he is not swimming, running, or playing competitive Magic: The Gathering, Lincoln is focused on promoting open-source software and making technology more accessible for everyone.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0