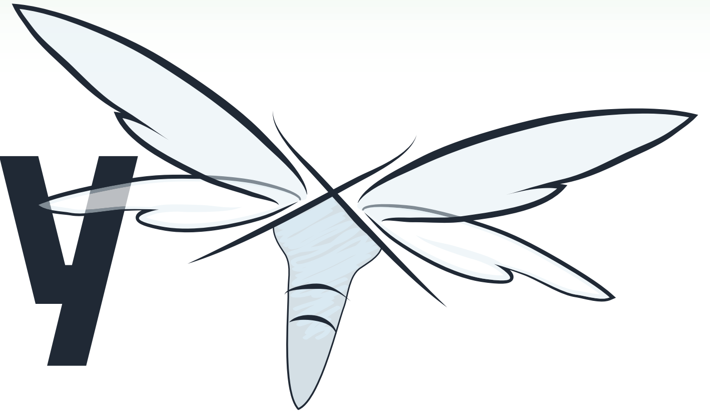
WildFly

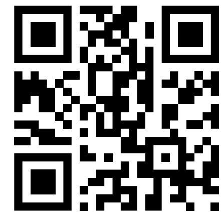


A STRONG HERITAGE

While the name may be new, WildFly is the eighth version of the project previously known as JBoss AS. Red Hat is committed to Java, and WildFly 8 will focus on quickly delivering an open-source implementation of Java EE 7.

Key features of WildFly:

- BLAZING FAST START–UP
- MEMORY DIET
- MODULAR DESIGN
- ELEGANT ADMINISTRATION
- COMMAND CENTRAL
- PATCHING MADE EASY
- STRICT COMPLIANCE
- PROJECT UNDERTOW



Learn more about the Wildfly project at wildfly.org

RED HAT® JBOSS® MIDDLEWARE | JBoss Community

DZone Refcardz

Brought to you by:
redhat

CONTENTS INCLUDE:

- API for WebSocket
- Batch Applications
- API for JSON Processing
- Concurrency Utilities
- JMS 2.0
- JSF 2.2... and More!

Java Enterprise Edition 7

By: Andrew Lee Rubinger and Arun Gupta

ABOUT THE PLATFORM

Enterprise software development is inherently complex; multiuser 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 which allow the programmer to write less and focus on delivering clean business logic.

To this end, 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 Refcard 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 which compose the platform, and provides additional integration support. The Java EE 7 platform added the following new APIs to previous version of the platform:

- 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 like Contexts and Dependency Injection 1.1, Bean Validation 1.1, Servlet 3.1, Java Persistence API 2.1, Java Server Faces 2.2, and Java Transaction API 1.2 were updated to provide a more coherent way of building enterprise applications.

Useful resources:

- Javadocs for the Java EE 7 API: http://docs.oracle.com/javaee/7/api/
- Technologies in Java EE 7: http://www.oracle.com/technetwork/java/javaee/tech/index.html
- Java EE 7 tutorial: http://docs.oracle.com/javaee/7/tutorial/doc/home.htm
- Java EE 7 samples: https://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 the 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 a programmatic way to develop, deploy, and invoke WebSocket endpoints.

Annotated endpoint 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);
       }
    }
}
```

Alternatively, programmatic endpoint can be defined as:

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

Client endpoint is annotated with @ClientEndpoint and can be connected to a server endpoint as:

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

Public API from javax.websocket:

Class Name	Description
Server Endpoint	Annotation that decorates a POJO as a WebSocket server endpoint
Client Endpoint	Annotation that decorates a POJO as a WebSocket client endpoint
PathParam	Annotate method parameters of an endpoint with URI-template
Session	Represents a conversation between two WebSocket endpoints
Encoders	Interface to convert application objects into WebSocket messages

Class Name	Description
Decoders	Interface to convert WebSocket objects into application messages
Message Handler	Interface to receive incoming messages in a conversation
OnMessage	Makes 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

Batch Applications 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, aka Job XML.

```
<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 operate on an input item and produce 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
Checkpoint Algorithm	Provides a custom checkpoint policy for chunk steps
Partition Mapper	Provides unique batch properties for each batch execution
Partition Reducer	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 lingua franca of the web for consuming and creating web services. A new API to parse and generate JSON is introduced 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 of the JSON.

The streaming API allows you to 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();
```

Object Model API allows you to 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 an 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 concurrency 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 Java 2 Platform, Standard Edition 7 in the `java.util.concurrent` package.

Four managed objects are provided:

```
ManagedExecutorService
ManagedScheduledExecutorService
ManageadThreadFactory
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 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>
```

Javadocs:

ManagedExecutorService	Manageable version of ExecutorService
ManagedScheduled ExecutorService	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	Is used to monitor the state of a task's Future

Definitions for variable argument lists are in the header file **cstdarg**. Here's how that works out in code:

```
#include <cstdarg>

void funct(int num, ... )
{
    va_list arg_list;
    va_start(arg_list, num);
    int i = va_arg (arg_list, int);
    int j = va_arg(arg_list, double);
      va_end(arg_list);
}
```

JAVA API FOR RESTFUL WEB SERVICES (JAX-RS) 2.0

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 resource as URI, a well-defined set of methods to access the resource, and multiple representation formats of a resource can be easily marked in a POJO via annotations.

JAX-RS 2 adds a new Client API that can be used to access web resources. Without this API, users must use a low-level HttpUrlConnection to access the REST endpoint.

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

Client API provides support for different HTTP verbs, support for custom media types by integration with entity providers, and dynamic invocation.

JAX-RS 2 allows an asynchronous endpoint that suspends the client connection if the response is not readily available and later resumes it when it is.

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

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);
```

Filters and Entity Interceptors are extension points to 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.

```
@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
ClientResponseFilter	Interface implemented by client response filters
ContainerResponseFilter	Interface implemented by server request filters
ContainerRequestFilter	Interface implemented by server response filters

Class Name	Description
ConstrainedTo	Indicates the runtime context in which an JAX-RS provider is applicable
Link	Class representing hypermedia links
ReaderInterceptor	Interface for message body reader interceptor
WriterInterceptors	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:

```
@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 usage 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);
}
```

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
JMS Connection Factory	Annotation used to specify the JNDI lookup name of `ConnectionFactory`
JMS Destination Definition	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

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.

Java EE 7 platform enables default CDI injection for all beans that explicitly contain a CDI scope annotation and EJBs. A new attribute, "bean-discovery-mode" attribute is added to beans.xml:

```
<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: Only types with an explicitly declared CDI scope are considered for injection.
none: Disable CDI

By default, CDI interceptors are disabled and can be enabled and ordered via the @javax.interceptor.Interceptor.Priority annotation as shown:

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

This can be done for decorators and alternatives too.

In addition, CDI 1.1 also introduces the following:

- 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
- Allow easy access to the current CDI container
- `AfterTypeDiscovery` event, allowing extensions to register additional types after type discovery
- `@WithAnnotations` as a way of improving extension loading performance enchancements

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 all the way to the persistence layer.

New in the 1.1 release is:

- Method-level validation (validation of parameters or return values)
- Dependency injection for Bean Validation components
- Integration with Context and Dependency Injection (CDI)
- 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's how to 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;
}
```

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.

Non-Blocking I/O

Code Example

A traditional approach to reading data is to poll for available input in a loop:

```
public class TestServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws IOException, ServletException {
 ServletInputStream input = request.getInputStream();
       byte[] b = new byte[1024];
       int len = -1;
       while ((len = input.read(b)) != -1) {
          . . .
       }
    }
}
```

In the case that we're reading faster than data is available via the `read()` method, the calling `Thread` will block. An alternative approach is to leverage event listeners; using the new nonblocking additions to the Servlet API, this can be done as follows:

```
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 when 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
- Session fixation attack by adding `HttpServletRequest.changeSessionId` and a new interface `HttpSessionIdListener`. You can listen for any session id changes using these methods.
- Default security semantic for non-specified HTTP method in `<security-constraint>`
- Clarifying the semantics if a parameter is specified in the URI and payload
- 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 calling `getServletOutputStream` or `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 specification aims to provide an object view of backend storage in a transactionally-aware manner. By dealing with POJOs, JPA enables developers to perform CRUD operations without the need for manually tuning SQL.

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.createNamedStoredProce
dureQuery("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`, `CommonAbstractQuery` interfaces have been added to the API, and the AbstractQuery interface has been refactored.

A spec example of bulk update might 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 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 tools to help manage state and promote code reuse. JSF 2.2 introduces Faces Flow which provides an encapsulation of related pages and corresponding backing beans as a module. This module has well-defined entry and exit points assigned by the application developer. The newly introduced CDI scope @FlowScoped 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; //. . .
}
```

A new EL object for flow storage, #{flowScope}, is also introduced.

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

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 us to list arbitrary name/value pairs in a component that are passed straight through to the user agent without interpretation by the UIComponent or Renderer.

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

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:

`@javax.transaction.Transactional` annotation that enables 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.*:

User Transaction	Allows an application to explicitly manage application boundaries
Transactional	Provides the application the ability to declaratively control transaction boundaries on CDI-managed beans, as well as classes defined as managed beans
Transaction Scoped	Specifies a standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction

ABOUT THE AUTHORS

Andrew Lee Rubinger is an open-source engineer, developer advocate, Program Manager at Red Hat, and author of "Continuous Enterprise Development in Java" from O'Reilly Media. He's the creator of the ShrinkWrap project and founding member of the Arquillian Testing Platform community. Andrew frequently enjoys sharing his experience in testable development in conferences across the globe and via @ALRubinger on Twitter.

Arun Gupta is Director of Developer Advocacy at Red Hat, focusing on Red Hat JBoss Middleware. As a founding member of the Java EE team at Sun Microsystems, he spread the love for technology all around the world. At Oracle, he led a cross-functional team to drive the global launch of the Java EE 7 platform, including strategic planning and execution, content development, and the execution of marketing campaigns and programs. After authoring ~1,400 blogs at blogs.oracle.com/arungupta on different Java technologies, he continues to promote Red Hat technologies and products at blog.arungupta.me.

RECOMMENDED BOOKS

Hands-on guide to developing enterprise Java applications in a continuously test-driven fashion. Walks through the entire development process from bootstrapping to integration testing and deployment.
Buy Here

Introduces Java EE7 in detail, including chapters on WebSockets, Batch Processing, RESTful Web Services, JMS, and more.
Buy Here

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream"**, says PC Magazine.

Copyright © 2014 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome

refcardz@dzone.com

Sponsorship Opportunities

sales@dzone.com

$7.95

Version 1.0