Java EE 7 Recipes

Presented By: Josh Juneau
Author and Application Developer

About Me

Josh Juneau

Day Job: Developer and DBA @ Fermilab

Night/Weekend Job: Technical Writer

- Java Magazine and OTN
- Java EE 7 Recipes
- Introducing Java EE 7
- Java 8 Recipes

JSF 2.3 EG

Twitter: @javajuneau

Agenda

Resolve a series of real life scenarios using the features of Java EE 7. We will cover recipes that span across most of the Java EE specifications…

Before we start. . .

Old: J2EE

Modern: Java EE

Java EE of the Past

Difficult to Use

Verbose

Configuration

Few Standards

Progressive Improvements

More Productive

Less Configuration

More Standards

Java EE 7 Increases Productivity Even More and Introduces Standards for Building Modern Applications

Java EE 7 Increases Productivity

- CDI Everywhere
- JAX-RS Client API, Async Processing
- Bean Validation in EJBs and POJOs
- JSF Flows
- JMS 2.0 - Much Less Code

Java EE 7 Introduces Standards

- WebSockets
- JSON-P
- Batch API
- Concurrency Utilities for Java EE

Recipes!

Lots to cover…

Recipes!

Statement for Completeness:

Recipes are not meant to provide comprehensive coverage of the features. Rather, they are meant to get you up and running with the new functionality quickly, providing the details you need to know. To learn more details on any of the features covered, please refer to the online documentation.

Let's Cook!

JSF

- New features added in JSF 2.2, adding more flexibility
  - Faces Flows
  - Resource Library Contracts
  - HTML5 Support
  - File Upload Component
  - View Actions
  - Much More

Problem #1

You would like to invoke an action method the first time a particular view is loaded. Furthermore, you would like to perform navigation with the results of this method invocation.

Solution

Utilize a ViewAction, which will be invoked on the initial request by default.

How it Works

Load a view that contains the ViewAction metadata tag

```
<p:dataTable id="currentReservations" var="reservation"
             value="#{parkReservationController.currentReservations()}">

    <p:column headerText="Name">
        <h:link value="#{reservation.lastName}, #{reservation.firstName}" outcome="parkReservation">
            <f:param name="id" value="#{reservation.id}"/>
        </h:link>
    </p:column>
```

Specify the viewAction tag, within your JSF view as a child of <f:metadata>

```
<f:metadata>
    <f:viewParam name="id" value="#{parkReservationController.parkReservationId}"/>
    <f:viewAction action="#{parkReservationController.loadReservation}"/>
</f:metadata>
```

How it Works

Write the action method which is invoked via the ViewAction

```
 */
public String loadReservation(){
    reservation = ejbFacade.findById(parkReservationId);
    return null;
}

/**
```

By default, the view is loaded after the action has been invoked

How it Works

- Call actions on GET requests
- Specify the action to be invoked using the *action* attribute.
- View Actions do not operate on postback, by default (initial request), since they are commonly used in the initial view request…as in the example. Can be altered via the *onPostback* attribute.

How it Works

- Invoked in the INVOKE_APPLICATION phase by default, configurable via the viewAction tag *phase* attribute
- The return value of viewAction method can be a navigational response since view actions always force a redirect response.

Problem #2

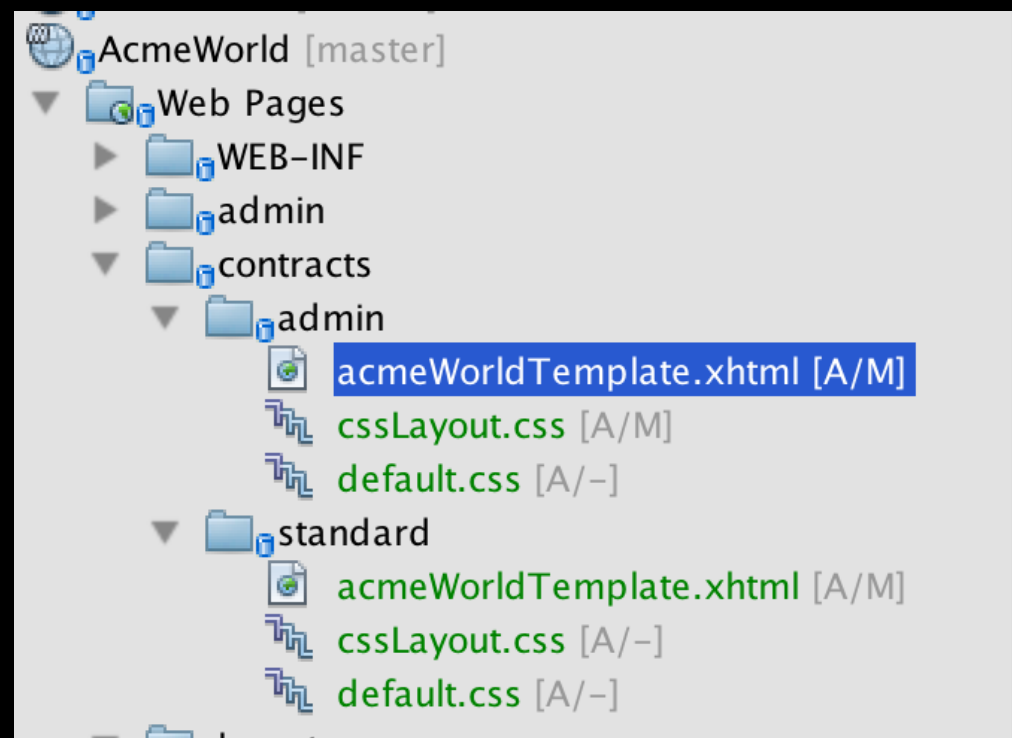
You wish to provide the ability to change your application look and feel based upon the different user privileges. For instance, administrators should have a green background, and users should see AcmeWorld blue.

Solution

Utilize JSF Resource Library Contracts to apply different theming across the various portions of your application, as needed.

Solution

Create a folder named "contracts" within the root of the web application, and then create separate folders for each contract. Place the applicable CSS files into each contract folder, along with a template file for each contract.



Solution

Add a <resource-library-contracts> element to the faces-config.xml file, and add a <contract-mapping> for each contract, along with a url pattern to map with each contract.

```
<resource-library-contracts>
    <contract-mapping>
        <url-pattern>/admin/*</url-pattern>
        <contracts>admin</contracts>
    </contract-mapping>
    <contract-mapping>
        <url-pattern>*</url-pattern>
        <contracts>standard</contracts>
    </contract-mapping>
</resource-library-contracts>
```

Solution

Modify each view to include the template, and faces will load the template corresponding to the contract specified in the faces-config.

```
<h:body>

    <ui:composition template="/acmeWorldTemplate.xhtml">

        <ui:define name="content">
            <f:view>
                <h:form id="parkReservationForm">
                    <h1>Create Reservation</h1>
```

How it Works

JSF 2.2 adds the ability to specify a different look and feel (contract) for different parts of one or more application.

Contracts should be placed into a directory named “contracts”, located at the root of the web directory.

How it Works

- Resources such as CSS, JS, template files, and images reside within the individual contract folders.
- Resource library contracts can be packaged into JAR files for use within different applications.
  - Contracts placed within META-INF/contracts
  - JAR placed within WEB-INF/lib

Problem #3

You are interested in mixing HTML5 and JSF components within the same application. In this example, we wish to create a user entry form with HTML5 elements wired to JSF managed beans. We'll also create a JSF view that utilizes HTML5 attributes within the JSF components.

Solution

Take advantage of the seamless HTML5 support that is offered by JSF 2.2+.

Create HTML5 forms that work seamlessly with the JSF Lifecycle.

Using HTML5 markup, specify JSF passthrough elements to access the JSF runtime.

```
xmlns:jsf="http://xmlns.jcp.org/jsf"
```

Solution

Create JSF forms, providing HTML5 attribute specification on JSF components via passthrough. In some cases, we might wish to use JSF components, but *enhance* them by specifying attributes that are available for use only via the component's HTML5 counterpart.

Use pass-through attributes to specify attributes on JSF components that should be ignored by the JSF runtime.

Solution

```
xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">
```

```
<p:fieldset>
    <label for="name">Name:</label>
    <p:inputText id="name" pt:placeHolder="Restaurant Name"
                 value="#{restaurantController.current.name}" required="true"/>
</p:fieldset>
```

How it Works

- To get started using HTML5 elements with JSF, add the jsf namespace for the pass-through elements to the page. The namespace can be added to any HTML5 element attribute, allowing that element to be processed by the JSF runtime.

- To be treated as a JSF component, at least one element must contain the jsf pass-through namespace

How it Works

Pass-through attributes are the converse of pass-through elements: a pass-through attribute is applied to a JSF component to signify that the specified attribute should be ignored by the JSF runtime and passed directly through to the browser.

A single passthrough attribute can be specified by nesting f:passThroughAttribute in a JSF component, or multiple attributes can be specified by nesting f:passThroughAttributes.

How it Works

```
<p:inputTextarea cols="30" rows="10" value="#{restaurantController.current.description}"
    <f:passThroughAttribute name="placeHolder" value="Enter a description..."/>
</p:inputTextarea>
```

Bean Validation

New Features in Bean Validation 1.1:

- Method Validation
- EL Expressions in Error Messages
- Dependency Injection of Bean Validation Components
- Integration with CDI
- Group Conversion

Problem #4

You wish to apply server side validation for an entity bean, such that the validation will be applied when a user is entering data into a form to populate the fields of that bean.

Solution

Apply the necessary bean validation constraints via the use of annotations on the fields of the entity class.

```
@Id
@Basic(optional = false)
@NotNull
@Column(name = "ID")
private BigDecimal id;
@NotNull
@Size(min=1, message="Please enter a first name")
@Column(name = "FIRST_NAME")
private String firstName;
@NotNull
@Size(min=1, message="Please enter a last name")
@Column(name = "LAST_NAME")
private String lastName;
@NotNull(message="You must include a trip start date")
@Temporal(TemporalType.DATE)
@Column(name = "TRIP_START_DATE")
private Date tripStartDate;
```

How it Works

- Place validation constraint annotations on a field, method, or class such as a JSF managed bean or entity class.
- When the JSF Process Validations phase occurs, the value that was entered by the user will be validated based upon the specified validation criteria. At this point, if the validation fails, the Render Response phase is executed, and an appropriate error message is added to the `FacesContext`. However, if the validation is successful, then the life cycle continues normally.

- Specify a validation error message using the message attribute within the constraint annotation

Problem #5

You would like to validate a method's parameters or return value without writing an extra layer of code

Solution

Take advantage of method validation, available as of Bean Validation 1.1, and apply constraints to parameters or return values.

Solution

Parameter Constraints

```
public String createReservation(@NotNull
                                String firstName,
                                @NotNull
                                String lastName,
                                @Min(value=1)
                                @Max(value=15)
                                int     numberAdults,
                                @Min(value=0)
                                @Max(value=15)
                                int     numberChildren,
                                @Min(value=1)
                                int     numberDays,
                                Date    tripStartDate){
    ParkReservation newReservation = new ParkReservation();
    newReservation.setFirstName(firstName);
    newReservation.setLastName(lastName);
    newReservation.setNumAdults(numberAdults);
    newReservation.setNumChild(numberChildren);
    newReservation.setNumDays(numberDays);
    newReservation.setTripStartDate(tripStartDate);
    newReservation.setEnterDate(new Date());
    ejbFacade.create(newReservation);
    return "RESERVATION_CREATED";
```

Solution

Return Value Constraint

```
/**
 * Returns the number of people for a specified restaurant reservation.
 *
 * Constraint signifies that there must be at least one person included on the
 * restaurant reservation.
 * @return
 */
@Min(value=1)
private int reservationCount(RestaurantReservation res){
	return res.getNumberOfPeople().intValue();
}
```

How it Works

- Bean validation constraints can be placed on non-static methods and constructors, and on the return values of non-static methods
- To validate parameters for a method, constraint annotations can be placed directly on each parameter.
- If method validation fails, a *ConstraintValidationException* is thrown

How it Works

- It is possible to specify a constraint that applies to more than one parameter of a method, this is known as a *cross-parameter* constraint. Such constraints are applied at the method or constructor level.
- Constraint annotations that are placed on a method can also be used to validate the return value of that method.
- The *validationAppliesTo* element of the constraint annotation signifies the constraint target (*ConstraintTarget.RETURN_VALUE* or *ConstraintTarget.PARAMETERS*).

Problem #6

You would like to display the currently entered value within the bean validation error message.

Solution

Include the current value using expression language within the bean validation error message.

```
private int numChild;
@Min(value=1, message="You've selected ${validatedValue} for your reservation,
@Column(name = "NUM_DAYS")
private int numDays;
```

How it Works

- The current value can be embedded into the error message using the ${validatedValue} expression.
- Conditional logic with EL expression is possible in error messages.
- The validation engine makes formatter object available in the EL context (date formatting etc.)

CDI

- Improved alignment of JSF Component Model/ Scoping to CDI
- CDI Enabled By Default
- Add support for @AroundConstruct lifecycle callback for constructors
- Specify @Priority on interceptor binding
- Add event metadata
- Apply @Vetoed to ignore classes
- More…

Problem #7

You would like to access the a JSF Managed bean from within another JSF Managed bean. CDI 1.0 Recipe…

In this case, we want to check to ensure that a customer has booked a mandatory restaurant reservation with their park reservation.

Solution

Simply inject the resource into the managed bean of your choice. CDI can be used almost everywhere in Java EE 7 by default.

Solution

```
@Named(value = "restaurantReservationController")
@SessionScoped
public class RestaurantReservationController implements Serializable {
```

```
    @Inject
    RestaurantReservationController restaurantReservationController;
```

How it Works

CDI enabled by default, no need to include a beans.xml with your application.

- If no beans.xml is specified, the bean-discovery-mode attribute is set to 'annotated' is assumed
- Possible values for bean-discovery-mode are:

all, annotated, none

How it Works

As of Java EE 6, Managed beans can be annotated with @Named, and then injected into other CDI beans.

@ManagedBean and the javax.faces.bean scopes will be deprecated in a future release of Java EE, so @Named and the javax.enterprise.context scopes should be used instead.

Problem #8

You wish to mark a specific class as ignored by CDI.

We do not wish to have the entity classes of AcmeWorld managed by CDI.

Solution

Annotate the class with @Vetoed

```
@Vetoed
@Entity
@Table(name = "TRIP_RESERVATION_DETAIL")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "TripReservationDetail.findAll", query = "SELECT t FROM TripReserv
    @NamedQuery(name = "TripReservationDetail.findById", query = "SELECT t FROM TripReser
    @NamedQuery(name = "TripReservationDetail.findByReservationDate", query = "SELECT t F
    @NamedQuery(name = "TripReservationDetail.findByPark", query = "SELECT t FROM TripRes
public class TripReservationDetail implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
```

How it Works

- Marking a class with @Vetoed, means that it will be ignored by CDI. In other words, CDI will not process the class.
- The class will not be injectable.
- The class will not contain the life cycle of a contextual instance.
- When placed on package, all beans in the package are prevented from being managed.

Problem #9

You wish to specify the ordering of Interceptor bindings within an application.

In the AcmeWorld application, we have two interceptor classes (auditing reservation changes, logging new reservations), each with their own binding. We wish to prioritize these interceptor bindings such that the new reservation logging is ordered first.

Solution

Specify a priority for each interceptor binding using the @Priority annotation.

```
*/
@Interceptor
@NewLoggable
@Priority(200)
public class NewReservationLogger implements Serializable {
    @Inject
```

How it Works

@Priority can be used to specify the ordering of Interceptors, Decorators, and Alternatives.

The @Priority annotation requires an int value as an element. The lower the number, the higher the priority of the associated interceptor.

How it Works

Interceptor.Priority contains the following priority values:

APPLICATION (value 2000)
LIBRARY_AFTER (value 3000)
LIBRARY_BEFORE (value 1000)
PLATFORM_AFTER (value 4000)
PLATFORM_BEFORE (value 0)

Concurrency Utilities for Java EE

- Now we have a standard for developing multi-threaded and concurrent applications for the enterprise
- Built upon the Java SE java.util.concurrency classes foundation

Problem #10

You would like execute a process in the background while allowing the user to continue performing other tasks.

For instance, we wish to have the AcmeWorld application allow managers the ability to click a button to have the details of their all reservations sent to them, while the manager continues to navigate the site.

Solution

When the user clicks the button, send the task to a ManagedExecutorService on the application server for processing, and allow the user to continue their work.

Solution

```java
@Resource(name = "concurrent/__defaultManagedExecutorService")
ManagedExecutorService mes;
```

```java
AcmeReservationReport areport = new AcmeReservationReport("reservationReport",
                                    ejbFacade,
                                    reservationScheduleFacade,
                                    parkAdmissionFacade,
                                    parkFacade);
Future reportFuture = mes.submit(areport);
while (!reportFuture.isDone()) {
  // System.out.println("Running...");
}
if (reportFuture.isDone()) {
    System.out.println("Report Complete");
}
```

How it Works

- In Java SE 5, the java.util.concurrent package was introduced, providing a convenient and standard way to manage threads within Java SE applications
- The Concurrency Utilities for Java EE extends upon the SE implementation, making the concurrent resource injectable and placing the resources in the application server container
- All Java EE 7 Compliant Application Servers must contain default concurrent resources, in GlassFish these resources begin with _default, e.g: _defaultManagedExecutorService

How it Works

- Concurrency Utilities relies on JTA to maintain transaction boundaries
- Access a Concurrent Resource via JNDI lookup or Injection:

```
InitialContext ctx = new InitialContext();

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

How it Works

- A "Task" is a unit of work that needs to be executed in a concurrent manner
- Tasks must implement either *java.lang.Runnable* or *java.util.concurrent.Callable,* and optionally *ManagedTask*
- Tasks run within the same context of the component that submits the task
- To submit an individual task to a *ManagedExecutorService*, use the *submit* method to return a *Future* object
- To submit an individual task for execution at an arbitrary point, use the *execute* method

How it Works

```java
public class AcmeReservationReport implements Runnable, Serializable {

    String reportName;

    private ParkReservationFacade parkReservationFacade;
    private ReservationScheduleFacade reservationScheduleFacade;
    private ParkAdmissionFacade parkAdmissionFacade;
    private ParkFacade parkFacade;


    public AcmeReservationReport(String report,
                                 ParkReservationFacade parkReservationFacade,
                                 ReservationScheduleFacade reservationScheduleFacade,
                                 ParkAdmissionFacade parkAdmissionFacade,
                                 ParkFacade parkFacade) {
        this.reportName = report;
        this.parkReservationFacade = parkReservationFacade;
        this.reservationScheduleFacade = reservationScheduleFacade;
        this.parkAdmissionFacade = parkAdmissionFacade;
        this.parkFacade = parkFacade;
    }

    public void run() {
        // Run the named report
        if ("reservationReport".equals(reportName)) {
            runReservationReport();
        }
```

Problem #11

You are interested in spawning a thread to periodically execute a task in the background.

We wish to be alerted whenever a new reservation is placed, so we want to execute an alerter task in the background.

Solution

Spawn a server managed thread by passing a task via a ManagedThreadFactory.

Solution

```
@Resource(name="concurrent/__defaultManagedThreadFactory")
ManagedThreadFactory threadFactory;
```

```
public void initiateParkReservationAlerter(){
    reservationReportMessage = "Starting alerter thread...";
    Thread alerterThread = threadFactory.newThread(new ReservationAlerter());
    System.out.println(alerterThread);
    alerterThread.start();
    reservationReportMessage = "Check server log for output...";
}
```

How it Works

- A server managed Thread runs the same as any standard Thread...but in a managed fashion
- Thread class must implement Runnable
- Context can be passed to the Thread

EJB

- Transactional lifecycle callbacks in session beans can be opt-in
- Optional Passivation
- Explicit specification of remote or local interfaces
- Extended TimerService API
- JMS Alignment
- More…

Problem #12

You wish to mark the @PostConstuct and @PreDestroy lifecycle callback methods as transactional.

We want to start a new transaction when a reservation booking change is invoked by a user.

Solution

Opt into the transactional lifecycle callback by annotating the method with @TransactionAttribute

```
 *
 */
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
@PostConstruct
public void initialize() {
    System.out.println("Reservation booking begins...");
    bookingDetail = new ArrayList();
}
```

How it Works

By default, stateful session bean lifecycle callback methods are opt-in transactional.

The Lifecycle Callbacks will use the bean transaction management type, or use @TransactionAttribute to denote the Transaction Attribute Type.

How it Works

TransactionalAttribute Values:

- MANDATORY
- NOT_SUPPORTED
- REQUIRED
- REQUIRES_NEW
- SUPPORTED

Problem #13

You wish to return information about all active timers in your application.

In AcmeWorld, an active timer sends notifications to reservation holders that have not yet booked an obligatory restaurant reservation.

Solution

Call upon the TimerService getAllTimers() method to return a Collection of active Timer instances.

```
    */
    public Collection<Timer> obtainActiveTimers() {
        List<Timer> timerList = new ArrayList<>();
        TimerService timerService = sessionContext.getTimerService();
        Collection<Timer> timers = timerService.getAllTimers();

        return timers;
    }
}
```

How it Works

TimerService.getAllTimers is a newly added convenience API that returns all active timers associated with the beans in the same module in which the caller bean is packaged.

Includes both the programmatically-created timers and the automatically-created timers.

JPA

New Features in JPA 2.1:

- Stored Procedure support
- Custom Converters
- Schema Generation
- Unsynchronized Persistence Contexts
- More…

Problem #14

You wish to call upon a database stored procedure to perform a task within the database.

There is a database procedure that is used to create users within the Acme database (CREATE_USER).

Solution

Utilize the new StoredProcedureQuery to invoke the stored procedure. Utilize the EntityManager's createStoredProcedureQuery to invoke the stored procedure.

```
public void createUser(){
    StoredProcedureQuery spq = em.createStoredProcedureQuery("CREATE_USER");
}
```

Solution

Register any parameters by invoking the StoredProcedureQuery registerStoredProcedureParameter method and passing the required values.

```
spq.registerStoredProcedureParameter(1, String.class, ParameterMode.IN);
spq.setParameter(1, "JOSH");
spq.registerStoredProcedureParameter(2, String.class, ParameterMode.IN);
spq.setParameter(1, "JUNEAU");
```

Invoke the stored procedure by invoking the execute() method.

How it Works

In the past, we had to implement "hacks" to call native stored procedures, was also difficult to return values from stored procedures.

Invoke EntityManager's createNamedStoredProcedure or createStoredProcedure method, passing the string-based name of the stored procedure.

StoredProcedureQuery instance returned, which can be used to register parameters.

Invoke via StoredProcedureQuery execute()

How it Works

To return a value execute the stored procedure and then call upon its getOutputParameterValue() method.

For convenience, stored procedure can be registered on an Entity class via the @NamedStoredProcedureQuery.

```
@NamedStoredProcedureQuery(name="createUser", procedureName="CREATE_USER")
```

Problem #15

You wish to invoke some lifecycle callback methods to perform some additional tasks when a ParkReservation is created. We also wish to inject resources into the callback listener classes.

Solution

Create an Entity Listener, and perform the task using one of the life cycle callback methods: @PrePersist, @PostPersist, @PreUpdate, @PreRemove. Utilize CDI resources within the listener.

```java
public class ParkReservationListener {

    @Resource(name="jndi/AcmeMail")
    javax.mail.Session mailSession;

    @PrePersist
    public void prePersist(String name){
        System.out.println("A reservation is being made for " + name);
        // use the mail session
    }
```

```java
@EntityListeners(ParkReservationListener.class)
public class ParkReservation implements Serializable {
```

How it Works

Life cycle callback implementations get even easier, as we can now use CDI injection within entity listeners.

JSON-P

The Java API for JSON Processing is a new standard for generation and processing of JavaScript Object Notation data.

JSON-P provides an API to parse, transform, and query JSON data using the object model or the streaming model.

JSON is often used as a common format to serialize and deserialize data in applications that communicate with each other over the Internet.

JSON-P

The javax.json package contains a reader interface, a writer interface, and a model builder interface for the object model.

The javax.json.stream package contains a parser interface and a generator interface for the streaming model.

JSON-P

Object Model API: Like DOM API, Object Model API uses builder pattern to model JSON objects as a tree structure to represent data in memory. The model provides the JsonReader interface for consuming JSON objects and JsonObjectBuilder and JsonArrayBuilder to produce JSON objects.

JSON-P

Streaming API: This API is similar to SAX API for XML and used for parsing JSON text in a streaming fashion. Low level and event-based. Less memory intensive, and therefore more suitable for processing larger amounts of data.

Problem #16

You would like to build a JSON object model using Java code.

We wish to create a list of current reservations using JSON-P.

Solution

Utilize JSON Processing for the Java EE Platform to build a JSON object model containing all of the current reservations.

Solution

```
JsonObjectBuilder builder = Json.createObjectBuilder();
JsonArrayBuilder reservationArray = Json.createArrayBuilder();
StringBuilder jsonStr = new StringBuilder();
try(StringWriter sw = new StringWriter()){
    for(ParkReservation reservation:reservations){
        JsonObjectBuilder reservationBuilder = Json.createObjectBuilder();
        reservationBuilder.add("id", reservation.getId())
                .add("firstName", reservation.getFirstName())
                .add("lastName", reservation.getLastName())
                .add("numAdults", reservation.getNumAdults())
                .add("numChild", reservation.getNumChild())
                .add("numDays", reservation.getNumDays())
                .add("tripStart", reservation.getTripStartDate().toString());
        reservationArray.add(reservationBuilder);

    }
    builder.add("reservations", reservationArray);
    JsonObject result = builder.build();
```

How it Works

JSON defines only two data structures: objects and arrays. An object is a set of name-value pairs, and an array is a list of values. JSON defines seven value types: string, number, object, array, true, false, and null.

How it Works

Java EE includes support for JSR 353, which provides an API to parse, transform, and query JSON data using the object model or the streaming model

Make use of JsonObjectBuilder to build a JSON object using the builder pattern.

Call upon the Json.createObjectBuilder() method to create a JsonObjectBuilder.

How it Works

Utilize the builder pattern to build the object model by nesting calls to the JsonObjectBuilder add() method, passing name/value pairs.

```
JsonObjectBuilder add(String name, BigDecimal value)
JsonObjectBuilder add(String name, BigInteger value)
JsonObjectBuilder add(String name, boolean value)
JsonObjectBuilder add(String name, double value)
JsonObjectBuilder add(String name, int value)
JsonObjectBuilder add(String name, JsonArrayBuilder builder)
JsonObjectBuilder add(String name, JsonObjectBuilder builder)
JsonObjectBuilder add(String name, JsonValue value)
JsonObjectBuilder add(String name, long value)
JsonObjectBuilder add(String name, String value)
JsonObjectBuilder addNull(String name)
```

How it Works

It is possible to nest objects and arrays.

The JsonArrayBuilder class contains similar add methods that do not have a name (key) parameter. You can nest arrays and objects by passing a new JsonArrayBuilder object or a new JsonObjectBuilder object to the corresponding add method.

Invoke the build() method to create the object.

Problem #17

You've generated a JSON Object Model, now you wish to write it to a file.

Solution

Write a JSON Object Model using a JsonWriter, which can be created from the javax.json.Json class.

```
JsonObject result = builder.build();
try(JsonWriter writer = Json.createWriter(sw)){
    writer.writeObject(result);
}
jsonStr.append(sw.toString());
System.out.println(jsonStr);
```

How it Works

The JsonWriter is instantiated by passing a Writer as an argument.

Call upon the JsonWriter.writeObject() method, and pass the jsonObject you wish to write out.

Close the JsonWriter when finished.

Problem #18

You wish to parse some JSON using Java.

Solution

Use the JsonParser, which is a pull parser that allows us to process each record via iteration.

```
public void parseJson(String fileName){
    try {
    InputStream is = new FileInputStream(fileName);
    JsonParser parser = Json.createParser(is);

    while(parser.hasNext()){
        Event event = parser.next();
        switch (event) {
        case KEY_NAME:
            parser.getString();
            break;
        case VALUE_STRING:
```

How it Works

Parser can be created on a byte or character stream by calling upon the Json.createParser() method.

Iterate over the JSON object model and/or array, and parse each event accordingly.

How it Works

JSON Events

START_OBJECT

END_OBJECT

END_ARRAY

KEY_NAME

VALUE_STRING

VALUE_NUMBER

VALUE_TRUE

FALUE_FALSE

VALUE_NULL

WebSockets

The Java API for WebSocket provides support for building WebSocket applications.

WebSocket is an application protocol that provides full-duplex communications between two peers over the TCP protocol.

Problem #19

You wish to create a WebSocket endpoint that can be used to receive messages asynchronously.

Solution

Create a WebSocket endpoint by annotating a POJO class using @ServerEndpoint, and providing the desired endpoint path.

Create a message receiver method, and annotate it with @OnMessage

Solution

```java
@ServerEndpoint(value = "/chatEndpoint", encoders = ChatEncoder.class, decoders = ChatDecoder.class)
public class ChatEndpoint {

    @OnOpen
    public void open(final Session session) {
        System.out.println("Chat Session Opened: " + session.getId());
    }

    @OnMessage
    public void onMessage(final Session session, final ChatMessage chatMessage) {

        try {
            for (Session s : session.getOpenSessions()) {
                if (s.isOpen()) {
                    s.getBasicRemote().sendObject(chatMessage);
                }
            }
        } catch (IOException | EncodeException e) {
            System.out.println("Chat Exception: " + e);
        }
    }
}
```

How it Works

Create a WebSocket endpoint by annotating a class with @ServerEndpoint, and passing the value attribute with a desired path for the endpoint URI.

@ServerEndpoint(value="/chatEndpoint")

URI: ws://server:port/application-name/path

How it Works

Method annotated @OnOpen is invoked when the WebSocket connection is made.

Method annotated @OnMessage is invoked when a message is sent to the endpoint. It then returns a response.

How it Works

Specify optional Encoder and Decoder implementations to convert messages to and from a particular format.

How it Works

Example of a Decoder:

```
public class ChatDecoder implements Decoder.Text<ChatMessage> {
    @Override
    public void init(final EndpointConfig config) {
    }

    @Override
    public void destroy() {
    }

    @Override
    public ChatMessage decode(final String textMessage) throws DecodeException {
        ChatMessage chatMessage = new ChatMessage();
        JsonObject obj = Json.createReader(new StringReader(textMessage))
                .readObject();
        chatMessage.setMessage(obj.getString("message"));
        chatMessage.setSender(obj.getString("sender"));
        chatMessage.setMessageDate(new Date());
        return chatMessage;
    }

    @Override
    public boolean willDecode(final String s) {
        return true;
    }
```

Problem #20

You want to send a message to your new WebSocket Endpoint.

We want to add a chat service to the AcmeWorld application.

Solution

Utilize JavaScript to open a connection to the WebSocket endpoint. This will invoke the @OnOpen method.

```
var serviceLocation = "ws://localhost:8080/AcmeWorld/chatEndpoint";
```

```
function connectToChatserver() {
        wsocket = new WebSocket(serviceLocation);
        wsocket.onmessage = onMessageReceived;
}
```

Solution

Handle messages received and messages sent via JavaScript calls to the endpoint.

```
function onMessageReceived(evt) {
        //var msg = eval('(' + evt.data + ')');
        var msg = JSON.parse(evt.data); // native API
        var $messageLine = $('<tr><td class="received">' + msg.received
                        + '</td><td class="user label label-info">' + msg.sender
                        + '</td><td class="message badge">' + msg.message
                        + '</td></tr>');
        $chatWindow.append($messageLine);
}
function sendMessage() {
        var msg = '{"message":"' + $message.val() + '", "sender":"'
                        + $nickName.val() + '", "received":""}';
        wsocket.send(msg);
        $message.val('').focus();
}
```

How it Works

- The javax.websocket.server package contains annotations, classes, and interfaces to create and configure server endpoints.
- The javax.websocket package contains annotations, classes, interfaces, and exceptions that are common to client and server endpoints.
- Open a WebSocket connection by connecting to the endpoint using JavaScript. The WebSocket object takes two arguments, the first is the endpoint location, and the second is an optional string of parameters.

How it Works

Once connected, send the messages using JavaScript to construct the JSON object from the message content, and then send that to the WS endpoint.

Any encoders will intercept the message and put it into proper format.

@OnMessage will be invoked, processing the message accordingly, and sending response.

How it Works

When the response is sent, the encoder is invoked, sending the message back in JSON format.

```
@Override
public String encode(final ChatMessage chatMessage) throws EncodeException {
    return Json.createObjectBuilder()
                .add("message", chatMessage.getMessage())
                .add("sender", chatMessage.getSender())
                .add("received", chatMessage.getMessageDate().toString()).build()
                .toString();
}
```

credit to Mika Kops: http://www.hascode.com/2013/08/creating-a-chat-application-using-java-ee-7-websockets-and-glassfish-4/ for a nice client-side chat implementation

JMS 2.0 - Java EE Simplified

Single biggest benefit of JMS 2.0 is better productivity

Problem #21

You are running an online reservation system again, and:

- You would like to send a message to a user when he or she creates a reservation.
- Let's also send any notifications to other systems that are affected by the reservation.

Solution

Utilize JMS to create and send a message to a JMS queue when a reservation is placed.

(You have choice between classic or simplified API...let's see both)

Solution

```
@Resource(name = "jms/acmeConnectionFactory")
private ConnectionFactory connectionFactory;
@Resource(lookup = "jms/Queue")
Queue inboundQueue;
```

```
try(JMSContext context = connectionFactory.createContext();){
    context.createProducer().send(inboundQueue, reservation.getLastName());
    System.out.println("Message sent to queue...");
} catch (JMSRuntimeException jre){
    // Do something
}
```

How it Works

- JMSContext object simplifies the API
- Everything implements AutoCloseable
- Non-transaction sessions are default...no need to specify parameters
- JMS provider automatically coerces message
- JMSRuntimeException instead of checked exceptions

Problem #22

You would like to consume a message from a Queue and do something with it...

Solution

Implement a JMS Message Consumer (again, your choice of classic or simplified API)

Solution

```java
@Resource(name = "jms/acmeConnectionFactory")
private ConnectionFactory connectionFactory;
@Resource(lookup = "jms/Queue")
Queue inboundQueue;
```

```java
try{
    System.out.println("Receiving messages from queue...");
    JMSContext context = connectionFactory.createContext();
    JMSConsumer consumer = context.createConsumer(inboundQueue);
    // No need to cast
    messageBody = consumer.receiveBodyNoWait(String.class);
    System.out.println("Message:" + messageBody);
    // Do something with message...
```

How it Works

- Utilize JMSContext to create a consumer
- Use consumer to receive message body

Problem #23

You wish to send a message after a designated delay period

Solution

Set a delay by specifying setDeliveryDelay

Solution

```
try(JMSContext context = connectionFactory.createContext();){
    context.createProducer().setDeliveryDelay(1000).send(inboundQueue, reservation.getLastNam
    System.out.println("Message sent to queue...");
} catch (JMSRuntimeException jre){
    // Do something
}
```

How it Works

- Specify setDeliveryDelay on the message producer, and then send message normally
- The message is sent after the specified number of milliseconds

JAX-RS

New Features in JAX-RS 2.0

Client-Side API

Async Support

Filters and Interceptors

More…

Problem #24

Your application contains a web service for obtaining information via a specified URL. You want to generate a client to request the web service information and do something with the response. In this case, the web service can be used to return reservation information per a specified reservation ID.

Solution

Make use of the new JAX-RS Client API to build a client application

```
Client client = ClientBuilder.newClient();

WebTarget webTarget =  client.target("http://localhost:8080/AcmeWorld/rest/reservationRest");
Response res = webTarget.request("application/xml").get();

setReservationRestXml(res.readEntity(String.class));
```

How it Works

Obtain an instance of javax.ws.rs.client.Client by calling ClientBuilder.newClient().

Set any necessary properties on the client. Properties are name/value pairs that an be passed via setProperty(). Also may register Feature or Provider classes.

Call upon a target and return WebTarget.

How it Works

Target can be further qualified by invoking path() method and passing the next sequence in URI path:

client.target("http://someservice").path("1");

Zero or more paths can be appended…

Include dynamic content by wrapping in {}:

client.target("…").path("{dynamic}")

.pathParam("dynamic","my dynamic value")

How it Works

Lots more for the client API:

- Obtain a response (Different media types)
- Return entities
- Invoke clients at later time
- WebTarget Injection

Problem #25

You wish to execute a long-running task within a RESTful web service.

Solution

Make use of the AsyncResponse, allowing the task to be performed in an asynchronous manner, returning a response once the task has completed.

Solution

```java
@GET
@Path("allAsync")
public void getAllReservationsAsync(@Suspended final AsyncResponse async) {
    // This is for example purposes only...recreating long-running process
    try {
        Thread.sleep(2000);
    } catch (InterruptedException ex) {
    }
    // Long running process
    List<ParkReservation> result = parkReservationFacade.findAll();
    Response response = Response.ok(result).build();
    async.resume(response);

}
```

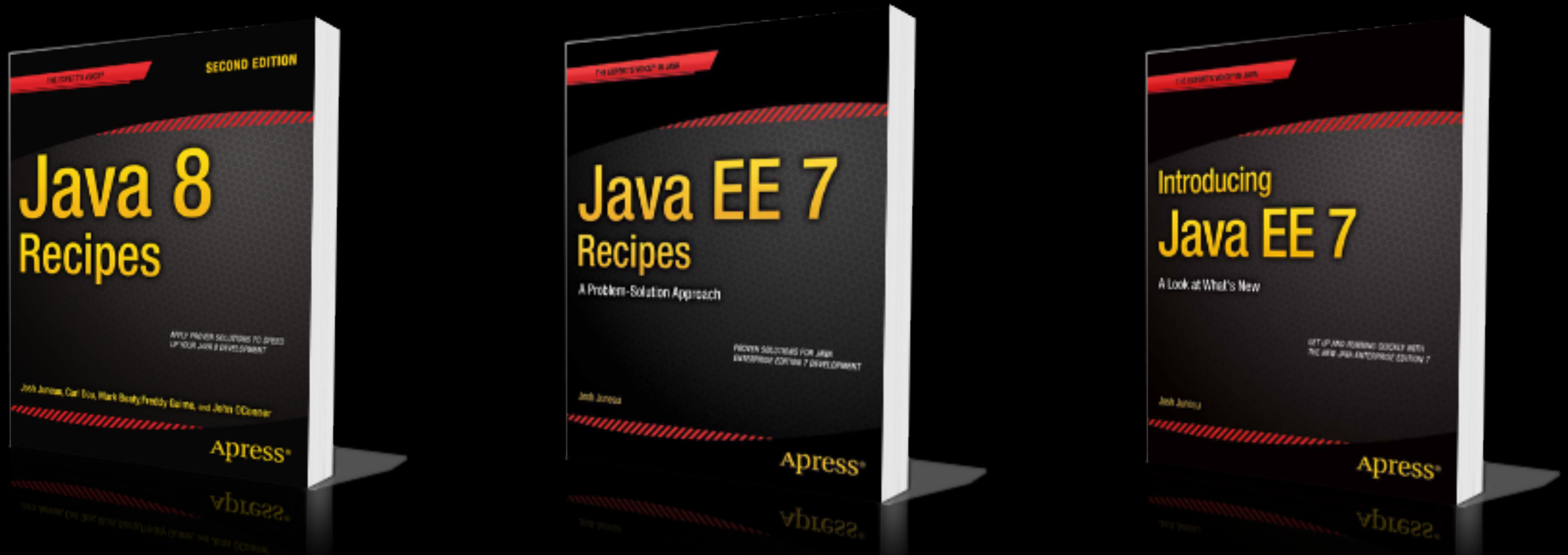
How it Works

- Pass AsyncResponse as argument to the REST method
- @Suspended
- Separate threads…client thread returns, response returned in a separate thread
- Runtime knows when method completes

We've Covered A Lot

…but there is a lot more to cover!!!!

Learn More



Code Examples: https://github.com/juneau001/AcmeWorld

Contact on Twitter: @javajuneau