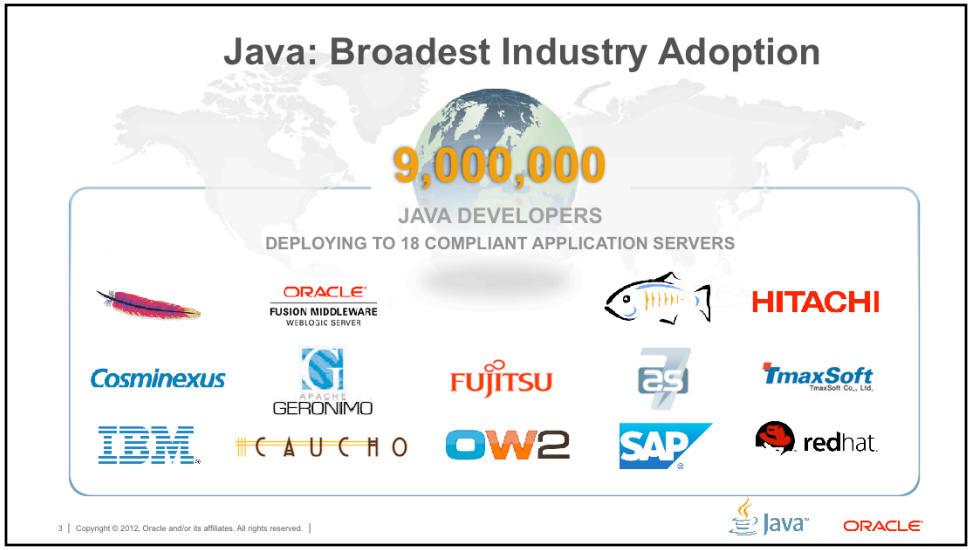

Java™


ORACLE
The Java EE 7 Platform
Productivity++ and HTML5
Arun Gupta
Java EE & GlassFish Guy
blogs.oracle.com/arungupta
@arungupta
MAKE THE
FUTURE
JAVA
Java



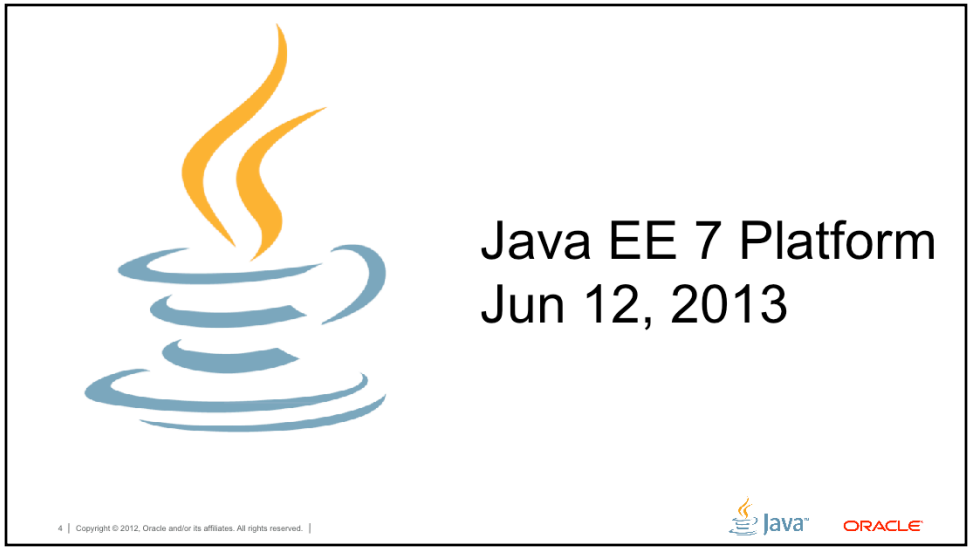
The talk is about Java EE 7, but lets take a quick look at Java EE 6 first.

But why does this matter?
More than 18 EE 6-compliant app servers
Strategic to almost every major technology vendor, including Oracle
EE 6 adopted quickly, anticipate EE 7 standard to continue trend
EE affects the way the world does business, and that's why we're so excited

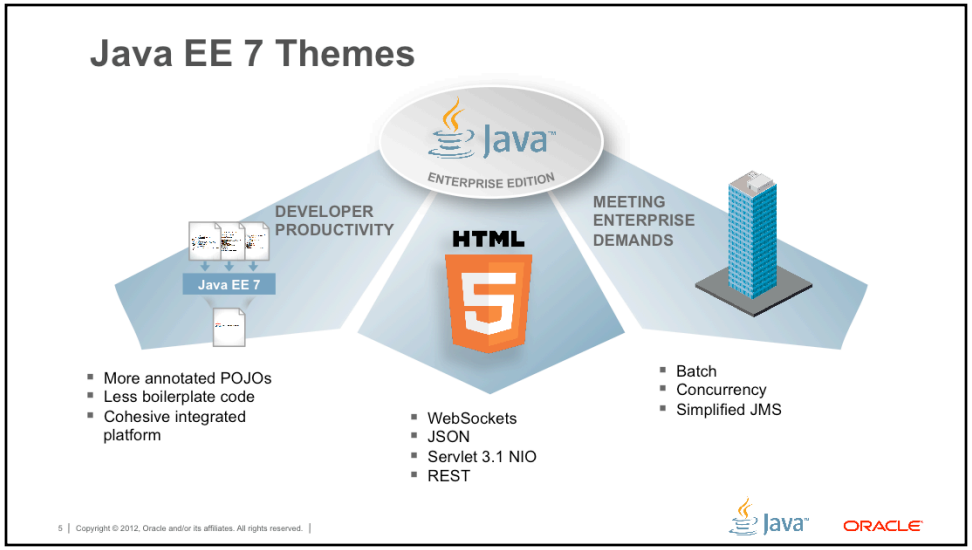
This has been the fastest roll out by Java EE application server vendors with a great variety of open source and commercial application servers. Java EE Web Profile allowed new vendors like Caucho and Apache to easily participate. This page shows the list of Java EE compliant vendors. The complete list is at: http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

This is the #1 development platform for enterprise applications and preferred choice for enterprise developers.

There is a mix of open source and commercial vendors. Java EE 6 has seen the fastest implementation of a Java EE release ever.



Java EE 7 was released on Jun 12, 2013. That means all 14 component specifications are final, the Reference Implementation is final, and TCK is available for licensees for building their compliant app servers.



There are 3 major themes of Java EE 7:

•Delivering HTML5 dynamic, scalable applications

- Reduce response time with low latency data exchange using WebSockets
- Simplify data parsing for portable applications with standard JSON support
- Deliver asynchronous, scalable, high performance RESTful Services and Async Servlets

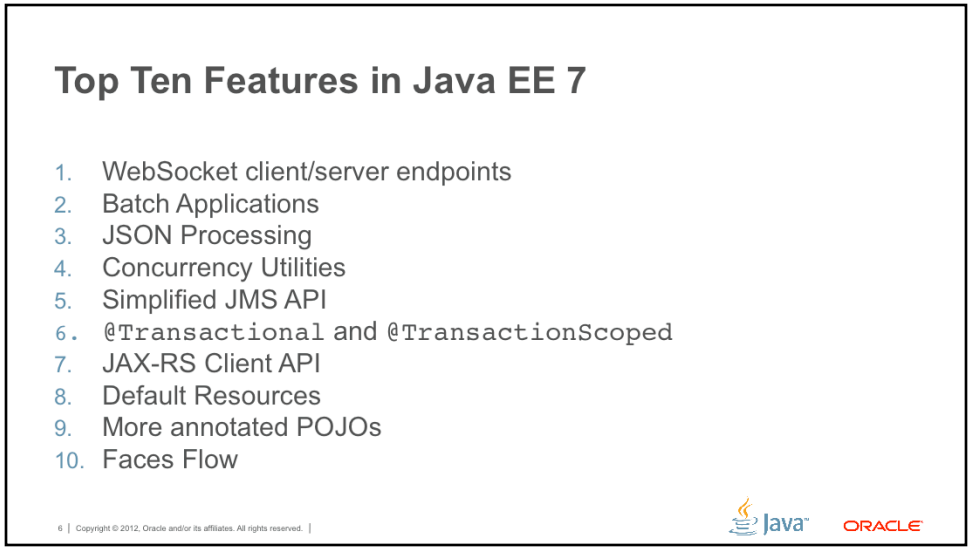
•Increasing developer productivity through simplification and new container services

- Simplify application architecture with a cohesive integrated platform
- Increase efficiency with reduced boiler-plate code and broader use of annotations
- Enhance application portability with standard RESTful web service client support

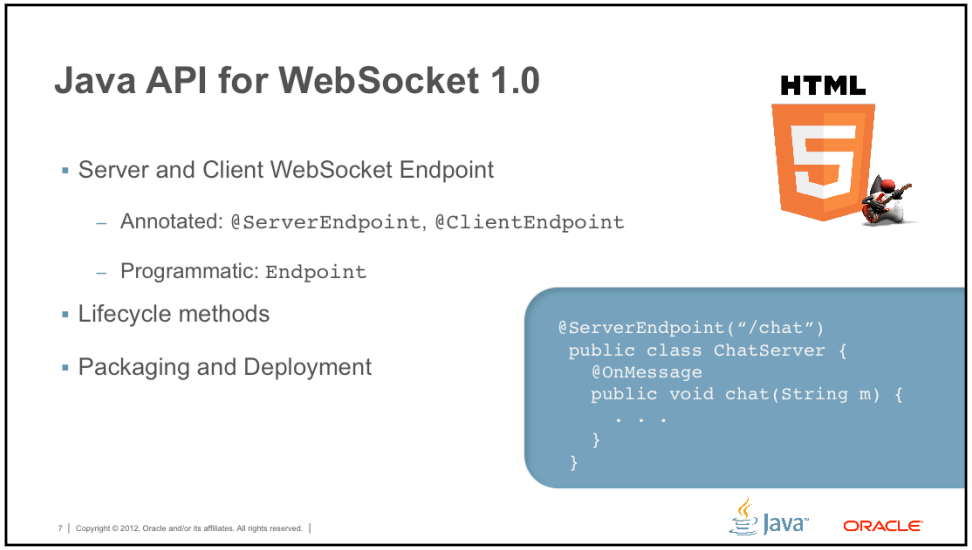
•And meeting the additional demands of the enterprise by adding new enterprise technologies.

- Break down batch jobs into manageable chunks for uninterrupted OLTP performance
- Easily define multithreaded concurrent tasks for improved scalability
- Deliver transactional applications with choice and flexibility

We'll be exploring all of these aspects in more detail in the context of the new Java EE 7 APIs as we



1. There is first class support for creating and deploying WebSocket endpoints. There is a standard W3C JavaScript API that can be used from browsers but this API also introduces a client endpoint.
2. In-built support for Batch applications allows to remove dependency on third-party frameworks.
3. Native support for JSON processing allows to make the application light-weight and getting rid of third party libraries.
4. Concurrency Utilities extends JSR standard Java SE Concurrency Utilities and add asynchronous capabilities to Java EE application components.
5. JMS API has been extremely simplified by leveraging CDI, Autocloseable, and other features of the language.
6. Deliver transactional applications with choice and flexibility, use @Transactional to enable transactions on any POJO.
7. JAX-RS added a new Client API to invoke a REST endpoint using a fluent builder API.
8. Default resources like JDBC DataSource, JMS ConnectionFactory, etc are added to simplify OOTB experience.
9. More annotations have been added to simplify devops experience such as @JMSDestinationDefinition that automatically creates a JMS destination.
10. JSF added Faces Flow that allows to create reusable modules to capture a flow of pages together.



The Java API for WebSocket, WebSocket 1.0, is a key technology for HTML5 support.
Enables highly efficient communication between client and server websocket endpoints over a single, bidirectional *and* full-duplex TCP connection, held for the duration of the client/server session.
WebSocket API offers a very convenient annotation-based approach to writing websocket endpoints.

Adding @ServerEndpoint annotation on a POJO converts it into a WebSocket server endpoint. No additional deployment descriptors are required. The URL at which the endpoint is published is included in the annotation. The POJO method that needs to be invoked is marked with @OnMessage and the payload of the message is automatically mapped to the method parameter.

The definition of a client endpoint is very similar, using instead the ClientEndpoint annotation.
For either case, there is also a programmatic API that allows you to drop down to dynamically control the configuration and the communication.
Using either the annotation or the programmatic approach, you can also define callback handlers for lifecycle events -- for example when a connection is opened, closed, or an error is received.

Websocket server and clients endpoints and associated classes and be conveniently packaged and deployed using the standard Java EE approach.

Java API for WebSocket 1.0

Chat Server

```
@ServerEndpoint("/chat")
public class ChatBean {
    static Set<Session> peers = Collections.synchronizedSet(…);

    @OnOpen
    public void onOpen(Session peer) {
        peers.add(peer);
    }

    @OnClose
    public void onClose(Session peer) {
        peers.remove(peer);
    }

    . . .
```


Java ORACLE

This is a slightly more elaborate sample that shows how you can build a canonical chat server using JSR 356 APIs.

As earlier, @ServerEndpoint is used to mark the POJO as a server endpoint and also specify the URL at which the endpoint is listening.

@OnOpen and @OnClose annotations mark the business methods that need to be invoked whenever a connection from a client is opened/closed. Session captures the other end of the conversation, client in this case. These methods add the client connecting or disconnecting to Set<Session>. So at any given time the list of connected client is always available.

Java API for WebSocket 1.0

Chat Server (contd.)

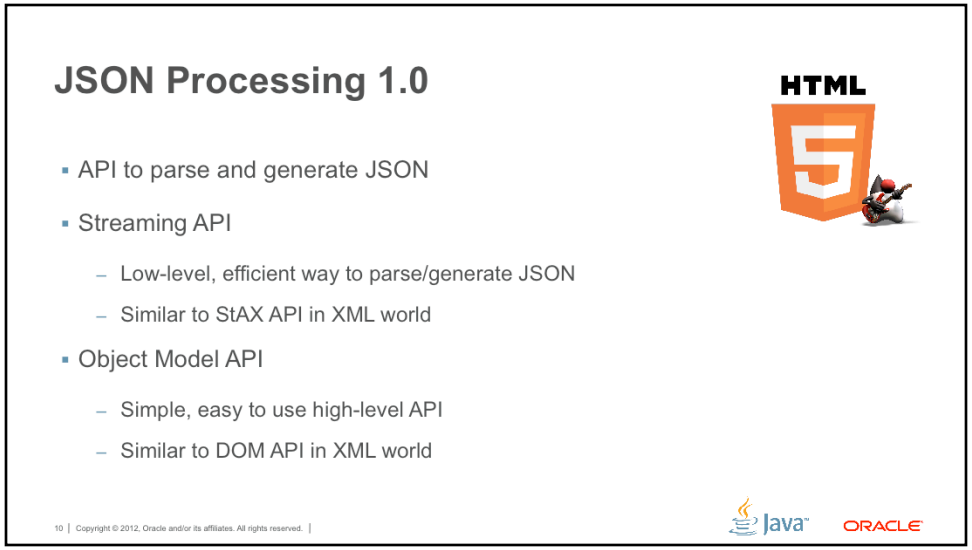
```
    . . .

    @OnMessage
    public void message(String message) {
        for (Session peer : peers) {
            peer.getRemote().sendObject(message);
        }
    }
}
```


Java ORACLE

@OnMessage indicates the business method that needs to be invoked whenever a WebSocket method is invoked. In this case, the received message is broadcasted to all the clients.

How simple ?

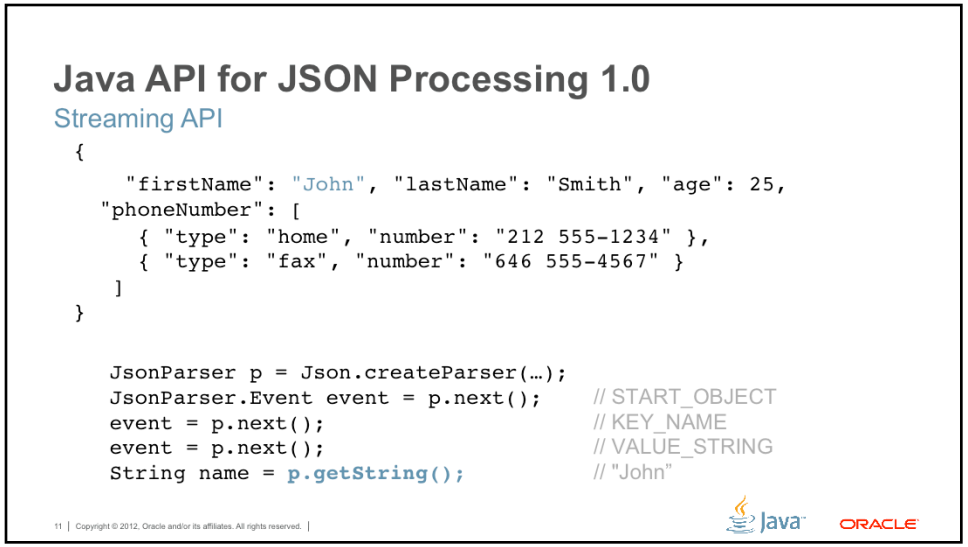
Wait for the attendees for an applause!



JSON is a key technology for data transfer within HTML5 applications, and certainly a lingua franca of the web.
With JSON 1.0, Java EE 7 adds new APIs to enable the parsing and generation of JSON text and objects.
There are 2 forms that these APIs take:
-A streaming API -- which is a low level, highly efficient event-driven API for the parsing and generation of JSON text, similar to StAX in the XML world. Streaming API generates events such as START_OBJECT, START_ARRAY, KEY_NAME, and VALUE_STRING.
-And, more conveniently, an easy-to-use, higher-level object-level API for mapping JSON text into JSON objects and arrays similar to the DOM API in the XML world.

Object Model API reads the entire JSON text in the memory and allows to cross-reference objects. If you just want to parse specific parts of JSON text then use Streaming API.

In both cases, read JSON data structure from a Reader/InputStream. Write to a Writer/OutputStream.

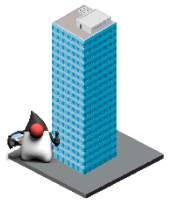


The slide show a simple JSON fragment. JSON is a collection of name/value pairs and an ordered list of values. This JSON fragment has several name/value pairs. The first such name/value pair has name “firstName” and value of “John”. Similarly two more name/value pairs. Then there is an array element with name “phoneNumber” and an ordered list of values.

The lower part of the slide shows a code fragment on how to create a JsonParser. It can read data from a Stream or a Reader. Iterating through the parser using next() allows to see the different events being emitted. The call to first next places the parser right after the opening {. Call to another next places the parser after “firstName”, and finally the third call to next() places it right after “John”. Calling getString() returns the current value.

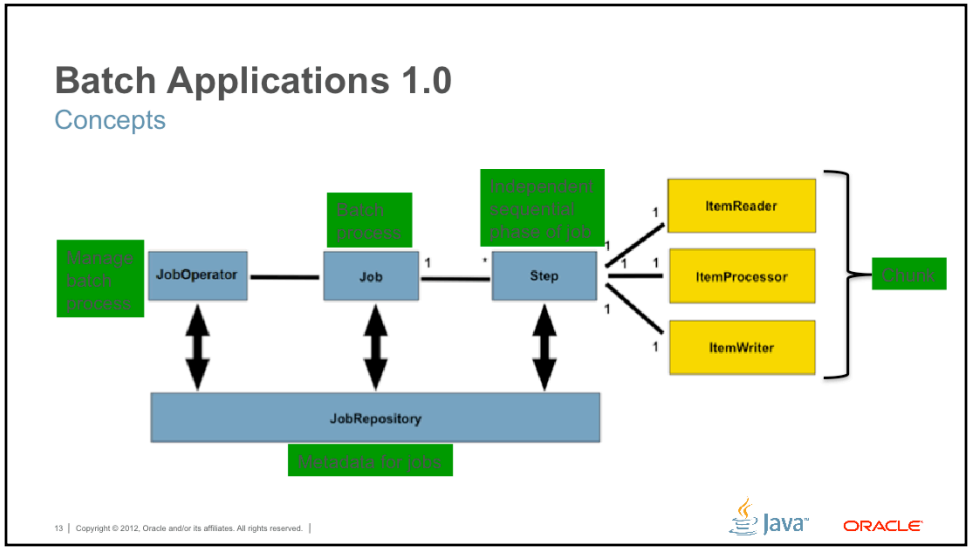
Batch Applications for Java Platform 1.0

- Suited for non-interactive, bulk-oriented, and long-running tasks
- Batch execution: sequential, parallel, decision-based
- Processing Styles
  - Item-oriented: Chunked (primary)
  - Task-oriented: Batchlet



Java ORACLE

The ability to execute batch jobs from Java EE is very important for many enterprise customers.
The new Batch Applications for Java API is therefore targeted at non-interactive, bulk-oriented, long-running tasks.
It allows you to customize the handling of these jobs in terms of their individual steps. You can specify sequential or parallel execution as well as decisions that direct the execution path.
The Batch API also provides for checkpointing and callback mechanisms.
An individual batch job step may itself be a "chunk", whereby you can specify separately the handling of the input, the processing, and the output of the individual items that are part of the chunk.
Or a step may be a "batchlet", which provides a more roll-your-own style of the step task's execution.



A Job is an entity that encapsulates an entire batch process. A Job will be wired together via a Job Specification Language.

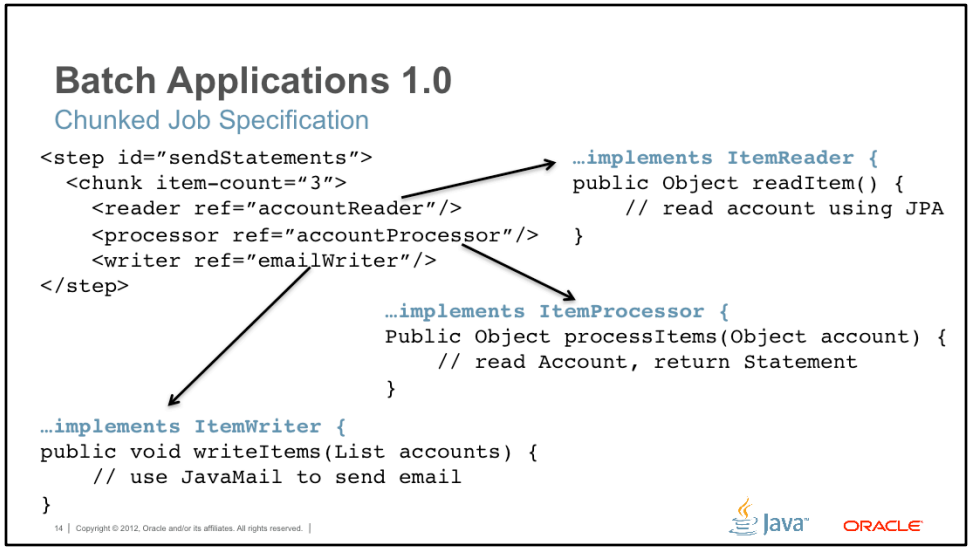
A Job has one to many steps, which has no more than one ItemReader, ItemProcessor, and ItemWriter. A job needs to be launched (JobOperator), and meta data about the currently running process needs to be stored (JobRepository).

A Step is a domain object that encapsulates an independent, sequential phase of a batch job.

JobOperator provides an interface to manage all aspects of job processing, including operational commands, such as start, restart, and stop, as well as job repository related commands, such as retrieval of job and step executions.

A job repository holds information about jobs currently running and jobs that have run in the past. The JobOperator interface provides access to this repository. The repository contains job instances, job executions, and step executions.

ItemReader is an abstraction that represents the retrieval of input for a Step, one item at a time.
ItemWriter is an abstraction that represents the output of a Step, one batch or chunk of items at a time.
ItemProcessor is an abstraction that represents the business processing of an item. While the ItemReader reads one item, and the ItemWriter writes them, the ItemProcessor provides access to transform or apply other business processing.



A Job is defined using Job Specification Language. For Java EE 7, this is defined using an XML document, called as Job XML.

Job XML typically consists of many steps, this contains one simple step. The step is either a chunk or batchlet – this one is chunk. This one is a chunk and has a reader, processor, and writer. The ref attribute refers to CDI resolvable bean name bundled with the archive. The item-count attribute defines the chunk size, i.e. the number of items processed at one time. The reader, processor, and writer work together for a chunk number of items at a time. All of this is done within a transaction, with automatic checkpointing.

Now lets take a look how reader, processor, and writer are implemented. This Job XML defines a simple use case of sending regular email statements.

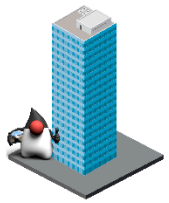
Reader implements ItemReader interface. The main method to override there is readItem which reads the item – this could be from any stream or database or JMS queue. For example, you may read Account information using JPA.

Processor implements ItemProcessor interface. processItem takes the item just read, applies business processing and returns a new object (possibly different type). This new object is now aggregated with the writer. For example, you may read an Account object and return a Statement object.

Writer implements ItemWriter interface. writeItems has a List parameter where all the aggregated items are available and writes it out. In our case, you will use JavaMail to send an email to List<Statement>.

Concurrency Utilities for Java EE 1.0

- Extension of Java SE Concurrency Utilities API
- Provide asynchronous capabilities to Java EE application components
- Provides 4 types of managed objects
  - `ManagedExecutorService`
  - `ManagedScheduledExecutorService`
  - `ManagedThreadFactory`
  - `ContextService`
- Context Propagation



Java™ ORACLE

Another new API in this release is the Concurrency Utilities for Java EE API.
This is an extension of the Java SE Concurrency Utilities API, for use in the Java EE container-managed environment so that the proper container-managed runtime context can be made available for the execution of these tasks—

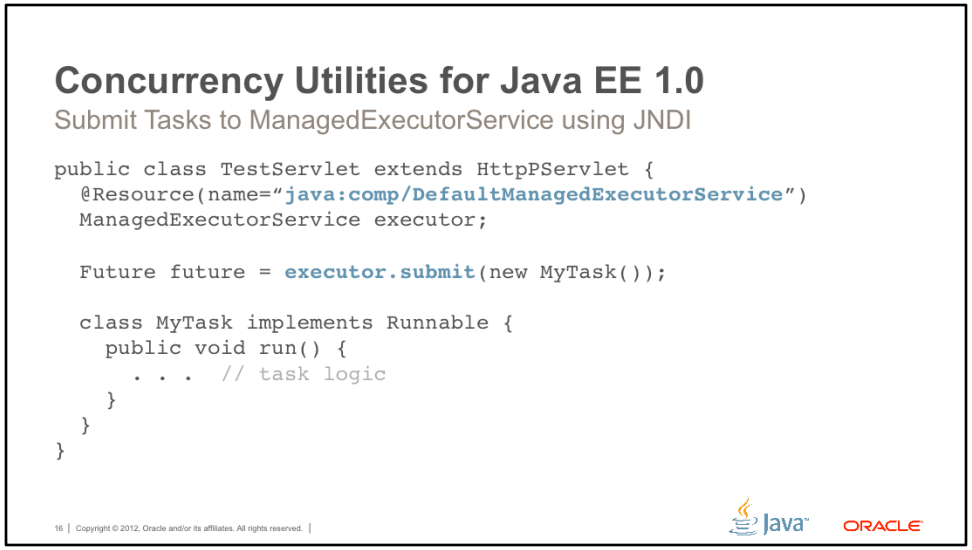
for example, naming (injection or JNDI), security, classloaders, etc.

[UTx API can be used within the concurrent tasks.]

This API provides asynchronous capabilities to Java EE application components, at a lower level than the existing asynch APIs (offered by EJB, Servlet, JAX-RS), and thus gives you a finer-grain level of control and configuration.

You can define your own runnable and callable tasks that can be invoked using these APIs.

The Concurrency Utilities API provides 4 types of managed objects.
The first 3 of these correspond to the Java SE concurrent APIs.
The ContextService allows you to convey a component's managed runtime context through to additional execution points where this context would not otherwise apply.



This code shows a simple Servlet.

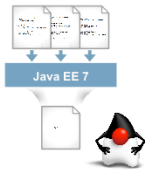
ManagedExecutorService can be obtained using @Resource. In this case, we are referring to the default ManagedExecutorService available via JNDI but these resources can be created in an application-server specific way. Now using the standard APIs, such as submit, a Runnable or Callable task can be easily submitted. The business logic of the task is implemented in run method.

All standard Java SE Concurrency APIs and design patterns are available for us. The big difference is that the threads created here are managed by the application server now. The JNDI, class loading, security context information is available to these tasks. This was not available earlier.

Java Message Service 2.0

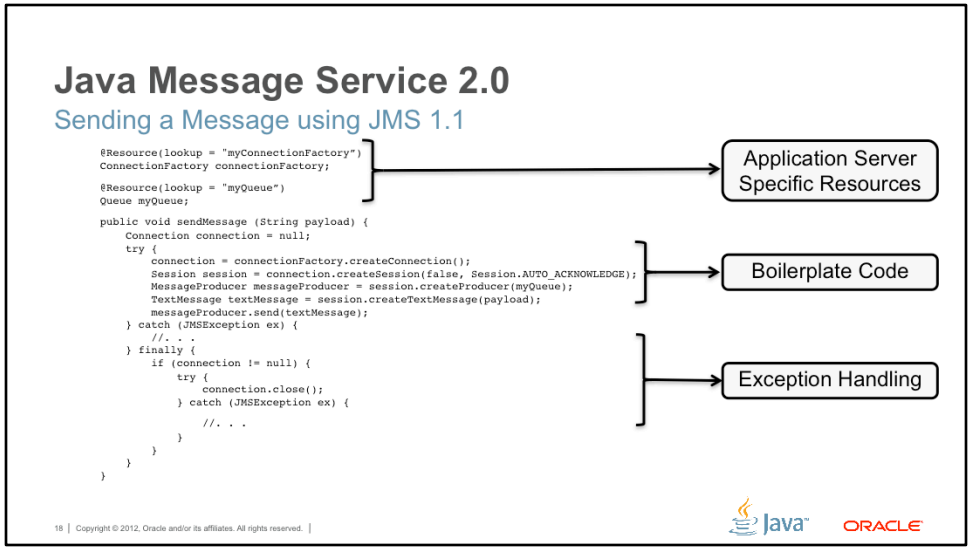
Get More from Less

- New `JMSContext` interface
- `AutoCloseable` `JMSContext`, `Connection`, `Session`, ...
- Use of runtime exceptions
- Method chaining on `JMSProducer`
- Simplified message sending



Java ORACLE

A big focus area of Java EE 7 is in increasing developer productivity by continuing our trend in offering simpler and easier-to-use APIs and in offloading developer tasks onto container services.
JMS in particular has undergone a major simplification with the JMS 2.0 release which is part of Java EE 7.
Sending and receiving a message using JMS 1.1 required a lot of boilerplate code.
JMS 2.0 has really fixed this with the addition of the new JMS simplified API and, in particular, the JMSContext interface.
The JMSContext combines in a single object the functionality of both the Connection and the Session in the earlier JMS APIs. You can obtain a JMSContext object by simply injecting it with the @Inject annotation.
There are many other simplifications in JMS as well -- to mention a few, AutoCloseable JMSContext, Session, Connection, JMSProducer and JMSConsumer objects
The use of runtime exceptions rather than checked exceptions
The use of method chaining for JMSProducers
Simplified message sending -- no need to create separate Message object, you can specify the message body as an argument to the send method instead.



JMS 2.0 calls the JMS 1.1 API has Classic API and JMS 2.0 new API as simplified API.

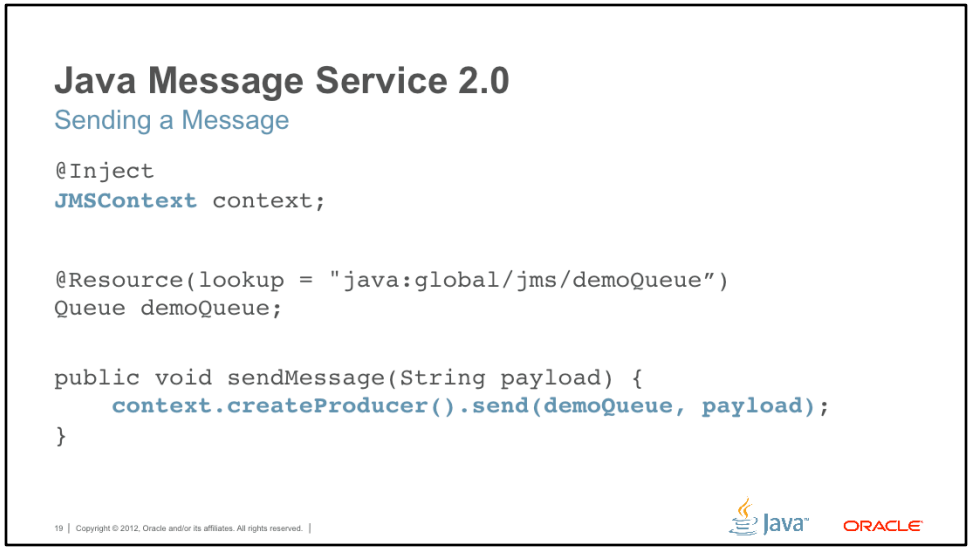
This code shows how to send a message using classic API. This is lot of boilerplate code to send a simple message and lets try to understand it.

First of all you need to create application server-specific resources such as ConnectionFactory and Destination – a Queue in this case.

Second you need to create a Connection from ConnectionFactory, Session from Connection, MessageProducer from Session, then a TextMessage and finally send a message. All this boilerplate code need to be written to just send a simple message.

Finally, all exceptions are checked and so there is lot of exception handling that needs to happen.

Lets take a look at how this can be done using simplified API.



This is the code used to send a message using simplified API.

First of all, JMSContext is injected using standard CDI syntax. Notice, no ConnectionFactory is specified here. In this case, a default JMS ConnectionFactory is used by and this is defined by the platform.

A Destination, a Queue in this case, is injected using @Resource. Even this annotation can be created using newly introduced @JMSDestinationDefintion annotation which would automatically create the destination.

Finally, the message is sent using method chaining. For example, create a producer using createProducer() and then calling send() method to send a message to a destination.

Really simple and clean. This also improves semantic readability of your code.

Encourage the audience to applause!

Java API for RESTful Web Services 2.0

- Client API
- Message Filters and Entity Interceptors
- Asynchronous Processing – Server and Client
- Common Configuration

Java ORACLE

JAX-RS 1.1 was a server-side API and introduced standard annotations to publish a REST endpoint. JAX-RS 2.0 now adds a client-side API to access a REST endpoint in a standard way. It provides a higher-level API than HttpURLConnection as well as integration with JAX-RS providers.

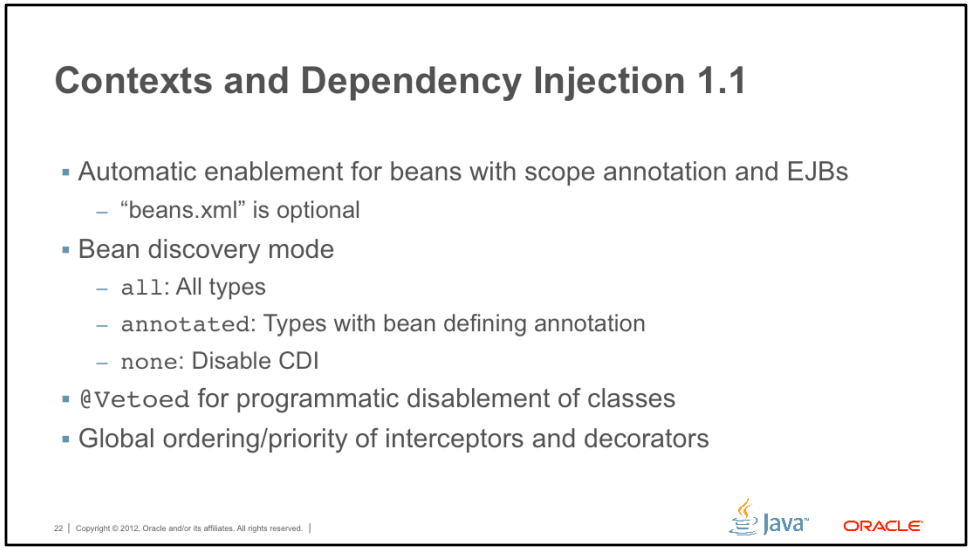
An instance of Client is required to access a Web resource using the Client API. The default instance of Client can be obtained by calling newClient on ClientBuilder.

A Web resource can be accessed using a fluent API in which method invocations are chained to build and ultimately submit an HTTP request.

Conceptually, the steps required to submit a request are the following: (i) obtain an instance of Client (ii) create a WebTarget (iii) create a request from the WebTarget and (iv) submit a request or get a prepared Invocation for later submission.

The benefits of using a WebTarget become apparent when building complex URIs, for example by extending base URIs with additional path segments or templates. Note the use of the URI template parameter {orderId}. The exact value of this template parameter is resolved using resolveTemplate() method.

The response to a request typed in the invocation of get() method. In this case, a String representation is returned back. This can also be a POJO where on-the-wire format is converted to POJO using JAX-RS entity providers.



CDI is making the platform lot more cohesive and is becoming a core component model.

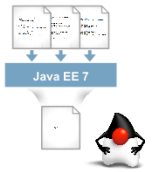
Also, CDI is now enabled by default in Java EE 7, without the requirement to specify a beans.xml descriptor.

If beans.xml exists with version 1.1, then a new attribute bean-discovery-mode can be specified on the top level <beans>. "all" value can be specified to enable injection all beans in the archive. "annotated" value is used to inject beans with a bean defining annotation – such as one with a CDI scope. And "none" can be used to completely disable CDI.

A new annotation @Vetoed is added to enable programmatic disablement of classes. This functionality was available in other frameworks and now in the standards.

Bean Validation 1.1

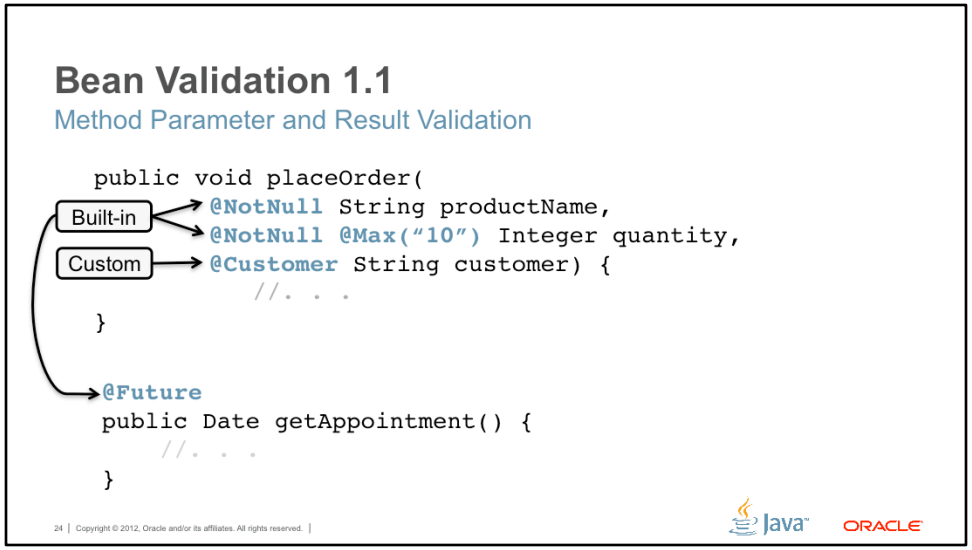
- Alignment with Dependency Injection
- Method-level validation
  - Constraints on parameters and return values
  - Check pre-/post-conditions
- Integration with JAX-RS



The first version introduced Bean Validation integration with just JPA and JSF.

Now Bean Validation 1.1 is more tightly aligned with the platform. For example, now validation constraints can be specified on POJO – either to validate the method parameters or return type. If the constraints specified on method parameters are not met, i.e. the pre-conditions are not met, then a ConstraintViolationException is thrown. Similarly if the constraint specified on the return type is not met then a ConstraintViolationException is thrown instead of returning the result. This ensures that post-conditions of invoking the method are met correctly.

BV 1.1 also introduces an integration with JAX-RS. So you can specify these constraints on your JAX-RS endpoints and the constraints are automatically verified.



This code shows two simple methods, one with a few parameters and void return type. Another method with no parameters and a Date return type.

In the first method, @NotNull is specified on the first two constraint ensuring that none of these parameters are null. The second parameter also has @Max indicating that this parameter cannot be more than 10. These are built-in constraints.

A custom constraint, such as @Customer, the one specified on the third parameter, can be specified. This custom constraint will define application-specific constraint logic.

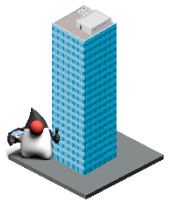
There is no need to call if/else within a method. This validation will be automatically triggered using standard CDI Interceptor Bindings and trigger ConstraintViolationException is the constraints are not met.

Similarly, for getAppointment() method, if the return value is not in future, per the current JVM timestamp, then a ConstraintViolationException is thrown as well.

In first case, if pre conditions is not met otherwise then the method is not invoked. In the second case, post conditions must be met for a clean invocation of the method.

Java Persistence API 2.1

- Schema Generation
  - `javax.persistence.schema-generation.*` properties
- Unsynchronized Persistence Contexts
- Bulk update/delete using `Criteria`
- User-defined functions using `FUNCTION`
- Stored Procedure Query



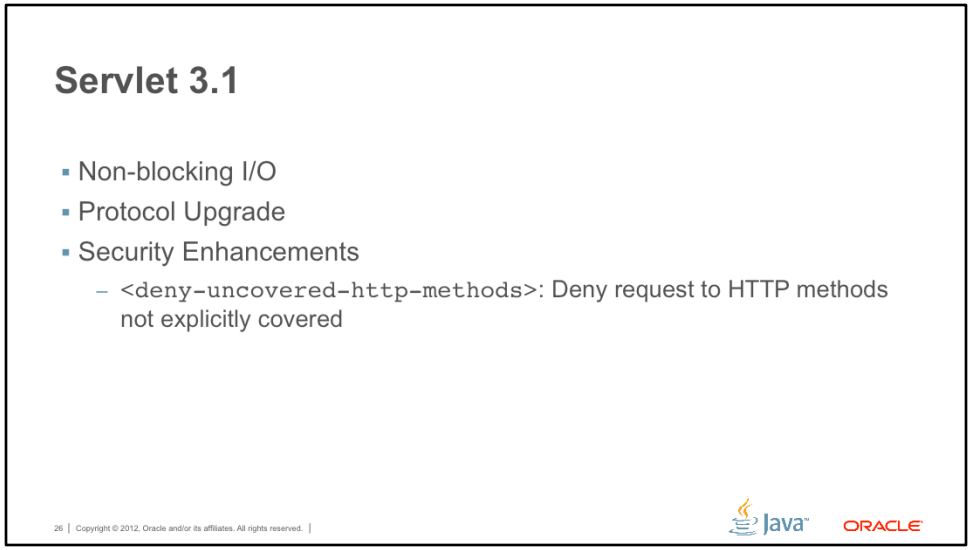
Java ORACLE

JPA allows to generate an Entity from a database table. JPA 2.1 adds new javax.persistence.schema-generation.* properties that allow to generate database schema or scripts from Entity classes. There is also a provision to load the database, i.e. DML, using these properties.

By default, a container-managed persistence context is of SynchronizationType.SYNCHRO- NIZED and is automatically joined to the current transaction. A persistence context of SynchronizationType.UNSYNCHRONIZED will not be enlisted in the current transaction, unless the EntityManager joinTransaction method is invoked. Similarly, by default, an application-managed persistence context that is associated with a JTA entity manager and that is created within the scope of an active transaction is automatically joined to that transaction. An application-managed JTA persistence context that is created outside the scope of a transaction or an application-managed persistence context of type SynchronizationType.UNSYNCHRONIZED will not be joined to that transaction unless the EntityManager joinTransaction method is invoked.

JPA 2 added typesafe Criteria API but that allowed only querying the database. JPA 2.1 updates the API to allow update/delete as well.

JPA 2.1 also adds the capability to invoke a stored procedure or user defined functions.



Until Servlet 3.0, any request processing, i.e. reading from a ServletInputStream is blocked. Servlet 3.1 introduces non-blocking I/O. Non-blocking request processing in the Web Container helps improve the ever increasing demand for improved Web Container scalability, increase the number of connections that can simultaneously be handled by the Web Container. Non- blocking IO in the Servlet container allows developers to read data as it becomes available or write data when possible to do so. Non-blocking IO only works with async request processing in Servlets and Filters. We'll take a look at real code in the next slide.

In HTTP/1.1, the Upgrade general-header allows the client to specify the additional communication protocols that it supports and would like to use. If the server finds it appropriate to switch protocols, then new protocols will be used in subsequent communication. The servlet container provides an HTTP upgrade mechanism. However the servlet container itself does not have knowledge about the upgraded protocol. The protocol processing is encapsulated in the HttpUpgradeHandler. Data reading or writing between the servlet container and the HttpUpgradeHandler is in byte streams. When an upgrade request is received, the servlet can invoke the HttpServletRequest.upgrade method, which starts the upgrade process. This method instantiates the given HttpUpgradeHandler class. The returned HttpUpgradeHandler instance may be further customized.

Last but not the least, Servlet 3.1 has made some improvements in security. By default, an HTTP method listed in security-constraint in web.xml is protected by the specified constraints. HTTP methods that are not specified in any of the constraints are called as "uncovered" HTTP methods. For example, if GET is specified in the constraint then POST, PUT, DELETE, and other HTTP methods are uncovered. Servlet

Servlet 3.1

Non-blocking I/O Traditional

```
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) {
      . . .
    }
  }
}
```


This code shows how data is read from ServletInputStream. This is done in a blocking way though, i.e. server thread waits for the client to complete sending data, or close the stream, or timeout. This is not very efficient, especially if the client is sending a large amount of data. The thread reading the data just waits for the client to complete.

Lets see how this can be efficiently solved using the simple “suspend and resume” design pattern.

Servlet 3.1

Non-blocking I/O: doGet

```
AsyncContext context = request.startAsync();
ServletInputStream input = request.getInputStream();
input.setReadListener(
    new MyReadListener(input, context));
```


Java ORACLE

Servlet 3.1 introduces a new interface ReadListener. This class represents a call-back mechanism that will notify implementations as HTTP request data becomes available to be read without blocking.

Application creates an implementation of ReadListener and set it on the ServletInputStream. Remember this works only in asynchronous Servlets.

Servlet 3.1

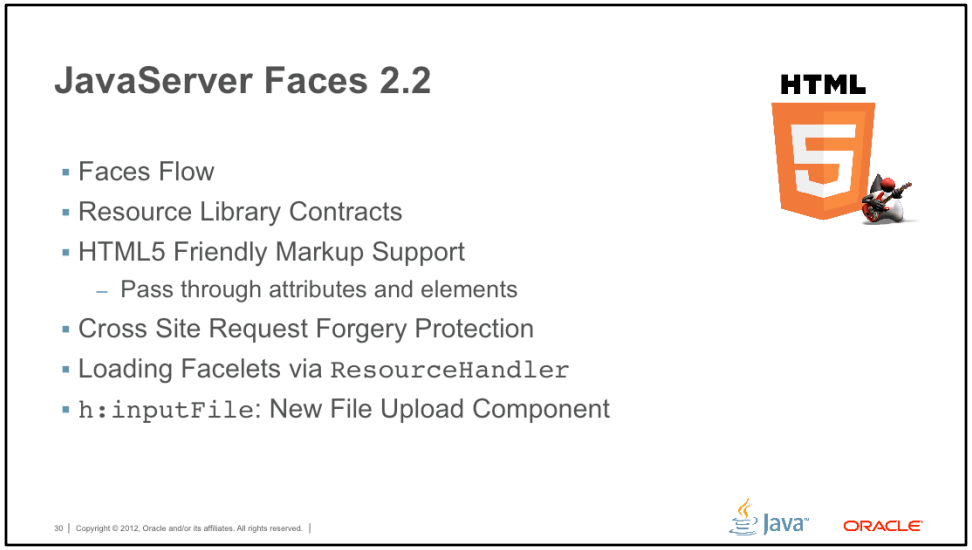
Non-blocking read

```
@Override
public void onDataAvailable() {
  try {
    StringBuilder sb = new StringBuilder();
    int len = -1;
    byte b[] = new byte[1024];
    while (input.isReady() && (len = input.read(b)) != -1) {
      String data = new String(b, 0, len);
      System.out.println("--> " + data);
    }
  } catch (IOException ex) {
    . . .
  }
}
```


Java™ ORACLE

onDataAvailable method in ReadListener is called when the data is available.

The server thread can process other requests in the meanwhile.



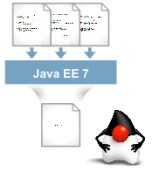
Faces Flow provides an encapsulation of related views/pages with an application defined entry and exit points. For example, a check out cart can consist of cart page, credit card details page, shipping address page, and confirmation page. All these pages, along with required resources and beans, can be packaged together as a module which can then be reused in other applications. Each flow has a well-defined entry and exit point that have been assigned some application specific meaning by the developer. Usually the objects in a faces flow are designed to allow the user to accomplish a task that requires input over a number of different views. In our case, the navigation between pages for selecting items, entering shipping address, credit card details, and confirmation page would make a flow. All the pages and objects that deal with the checking out process can be composed as modules. An application thus become a collection of flows instead of just views.

JavaServer Faces 2 introduced Facelets as the default View Declaration Language. Facelets allows to create templates using XHTML and CSS that can be then used to provide a consistent look-and-feel across different pages of an application. JSF 2.2 defines Resource Library Contracts that allow facelet templates to be applied to an entire application in a reusable and interchangeable manner.

Previously, all attributes used for JSF components could only be those that were understood by JSF itself. With JSF 2.2, JSF now adds HTML5-friendly markup support with its new pass-through attributes and elements.
These allow you to list arbitrary name/value pairs in a component that are passed through directly to the browser without interpretation by the JSF UIComponent or Renderer. HTML5 adds a series of new attributes for existing elements. These attributes include things like the type attribute for input elements,

Java Transaction API 1.2

- `@Transactional`: Define transaction boundaries on CDI managed beans
- `@TransactionScoped`: CDI scope for bean instances scoped to the active JTA transaction

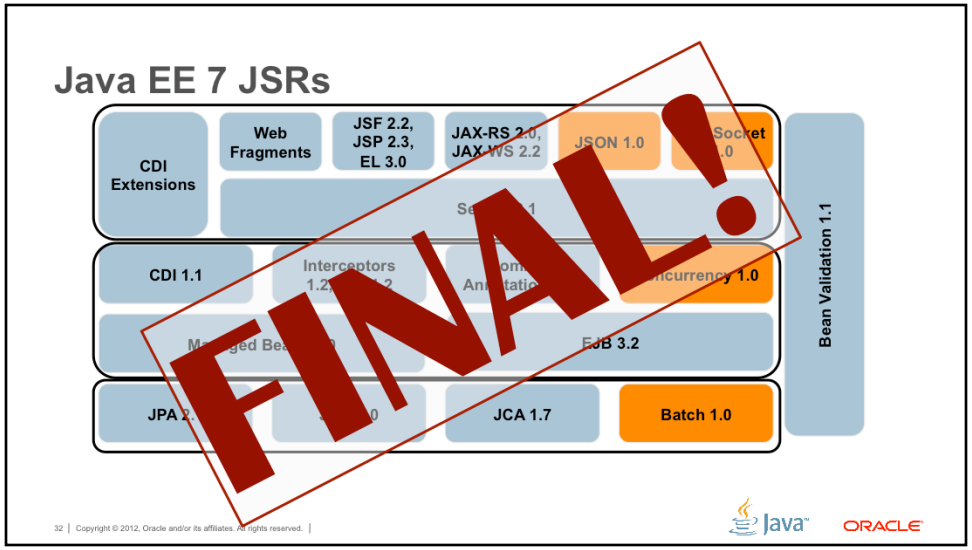


Java ORACLE

A significant change in JTA allows to convert any POJO into a transactional object.

The javax.transaction.Transactional annotation provides the application the ability to declaratively control transaction boundaries on CDI managed beans, as well as classes defined as managed beans by the Java EE specification, at both the class and method level where method level annotations override those at the class level.

The javax.transaction.TransactionScoped annotation provides the ability to specify a standard CDI scope to define bean instances whose lifecycle is scoped to the currently active JTA transaction. This annotation has no effect on classes which have non-contextual references such those defined as managed beans by the Java EE specification .



Different components can be logically divided in three different tiers: backend end, middle tier, and web tier. This is only a logical representation and the components can be restricted in a different tier based upon application's requirement.

JPA and JMS provide the basic services such as database access and messaging. JCA allows connection to legacy systems. Batch is used for performing non-interactive bulk-oriented tasks. Managed Beans and EJB provide a simplified programming model using POJOs to use the basic services.

CDI, Interceptors, and Common Annotations provide concepts that are applicable to a wide variety of components, such as type-safe dependency injection, addressing cross-cutting concerns using interceptors, and a common set of annotations. Con- currency Utilities can be used to run tasks in a managed thread. JTA enables Trans- actional Interceptors that can be applied to any POJO. CDI Extensions allow you to extend the platform beyond its existing capabilities in a standard way.

Web services using JAX-RS and JAX-WS, JSF, JSP, and EL define the programming model for web applications. Web Fragments allow automatic registration of third- party web frameworks in a very natural way. JSON provides a way to parse and generate JSON structures in the web tier. WebSocket allows to setup a bi-directional full-duplex communication channel over a single TCP connection.

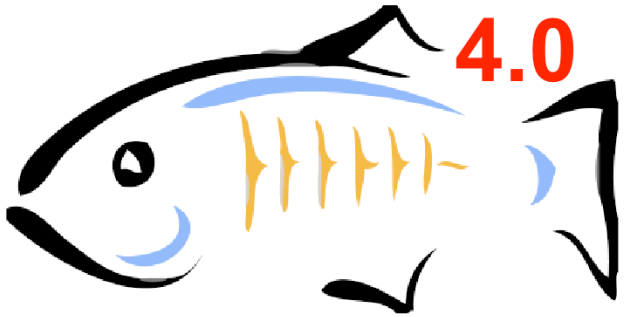
Bean Validation provides a standard means to declare constraints and validate them across different technologies.

Java EE 7 SDK can be downloaded from oracle.com/javaee.

Wait a moment, repeat the URL and encourage attendees to bookmark the URL and spread the word.

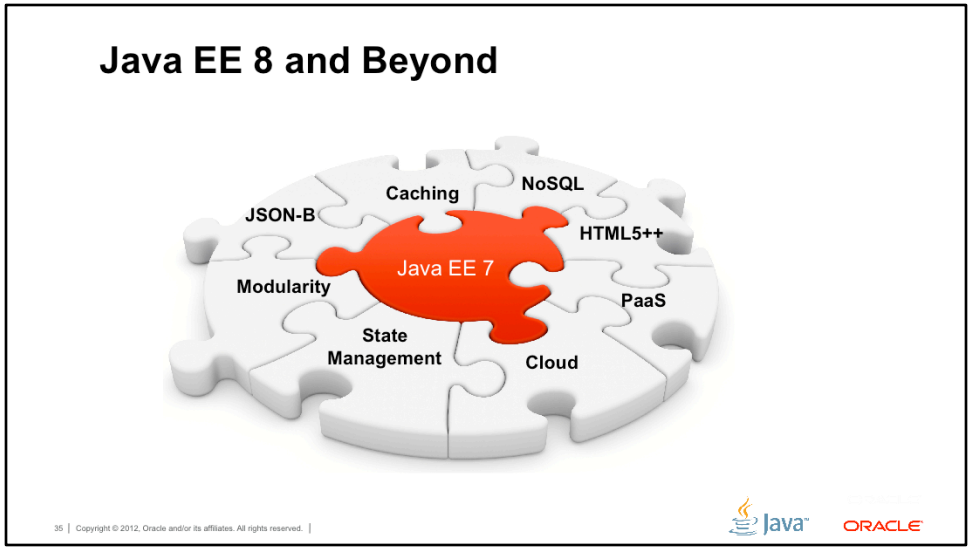
GlassFish Server Open Source Edition 4.0 Web Profile and Full Platform can be downloaded from glassfish.org.

Java EE 7 Implementation



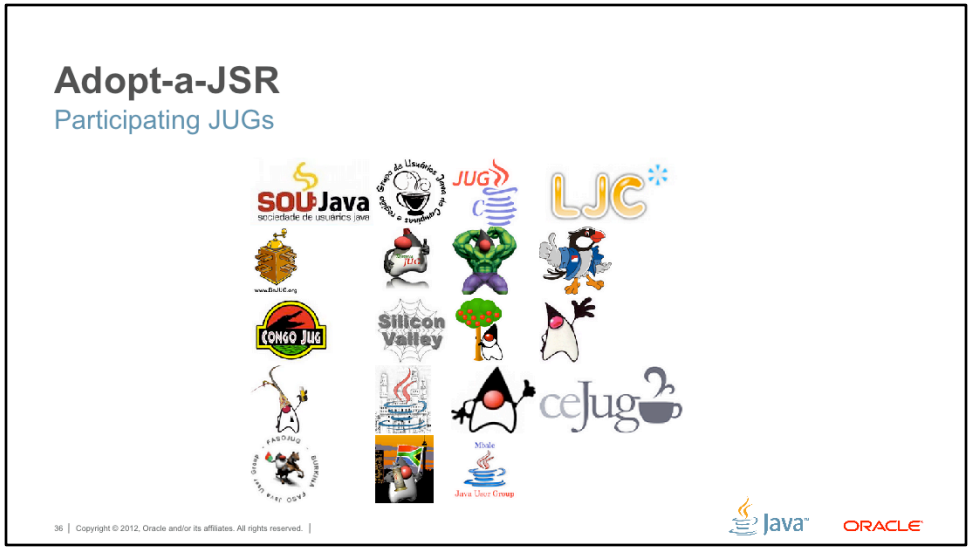
download.java.net/glassfish/4.0/promoted/

Java ORACLE



This diagram shows some of the target items for Java EE 8. This is only a thought process at this time and everything is completely subject to discussion with the Java EE Expert Group. Nothing is committed at this time.

JSON-B will provide a support for binding POJOs to JSON structure.
JSR 107 will provide support for caching.
Support for modular Java EE applications will be built upon Java SE modularity.
Standards-based PaaS/Cloud will definitely be looked upon.
Thin Server Architecture or HTML5++ will be looked upon.
State Management, NoSQL, and several other options.



Adopt-a-JSR is an initiative started by JUG leaders to encourage JUG members to get involved in a JSR, in order to increase grass roots participation. This allows JUG members to provide early feedback to specifications before they are finalized in the JCP. The standards in turn become more complete and developer-friendly after getting feedback from a wide variety of audience.

The JUGs organized hackathons, created sample applications, presented on different Java EE 7 topics at conferences, filed bugs on implementation. Overall they led to a much better Java EE 7. This release was not possible without support of different JUGs. They kept us honest.

Here are some specific examples on how different JUGs were involved:

London Java Community (LJC) organized a WebSocket and JSON Hack Day. The event was sold out within 2 hours and had 17 people on the waiting list. The event started with a presentation on explaining the APIs in Java API for WebSocket (JSR 353) and Java API for JSON Processing (JSR 353). The attendees designed a Market Ticker application. All the presentation material and source code was shared publicly. LJC also created projects (cdiex-palindrom-jsf and cdiex-datastore) to test CDI 1.1 specification.
Chennai JUG is building a multi-player game that can be played across the Internet. The application uses Java API for WebSocket 1.0 (JSR 356), Java API for JSON Processing 1.0 (JSR 353), Java Persistence API 2.1 (JSR 338), JavaServer Faces 2.2 (JSR 344) and Batch Applications for Java Platform (JSR 352) and/or Enterprise JavaBeans 3.2 (JSR 345) and so provide a holistic experience of building a Java EE 7 application. The energetic JUG meets regularly using G+ hangouts and in the neighborhood coffee shops

Call to Action

- **Specs**: javaee-spec.java.net
- **Implementation**: glassfish.org
- **Blog**: blogs.oracle.com/theaquarium
- **Twitter**: @glassfish
- **NetBeans**: wiki.netbeans.org/JavaEE7


Java EE 7
GlassFish
HTML5
Developer Productivity
Innovation
JMS2
Default Resources
NamedStoredProcedureQuery
Concurrency
Batchlet
ManagedExecutorService
@ClientEndpoint
@ServerEndpoint
Default CDI
Batch
JSON
JAX-RS2
Non-blocking I/O
JsonArray
JsonObject
Chunk
WebSocket
@Transactional
Expression Language
ClientBuilder
@JMSDestinationDefinition
@AroundConstruct
@FlowScoped

The preceding material is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.


Java™