12. Backend (JEE/CDI)

Vilmos Bilicki PhD
University of Szeged
Department of Software Engineering
Overview

- JEE
- Bean Validation
- CDI
- EJB
J2EE 1.2
Servlet, JSP, EJB, JMS, RMI

J2EE 1.3
CMP, Connector Architecture

J2EE 1.4
Web Services
Mgmt, Deployment, Async Connector

Java EE 5
Ease of Development, EJB 3.0, JPA, JSF, JAXB, JAX-WS, StAX, SAAJ

Java EE 6
Pruning, Extensibility, Ease of Dev, CDI, JAX-RS

Java EE 7
JMS 2.0, Batch, Caching, TX Interceptor, WebSocket, JSON

JAX-RPC, CMP/ BMP, JSR 88

Web Profile
Servlet 3.0, EJB 3.1 Lite

JAX-RS 2.0
EJB 3.2 (JSR 345)

- Session Bean
  - Stateless
  - Statefull
  - Singelton

- Message Driven Bean
Connector 1.6

- Integrating resources
  - Secure
  - Transaction aware
- EIS – Enterprise Information System
JPA 2.1

- Entities
  - ORM
  - Lifecycle
- Queries

```java
@Entity public class Employee {
    @Id private Integer id;
    private String name;
    ...
    @ManyToOne
    private Department dept;
    ...
}

@Table(indexes= {@Index(columnList="NAME")
        @Index(columnList="DEPT_ID")})
```

JTA 1.2

- X/Open XA architecture
- TP monitor

```java
@Stateless
@TransactionManagement(BEAN)
public class ExampleBean {
    @Resource
    private SessionContext ctx;

    public void foo() {
        UserTransaction utx = ctx.getUserTransaction();
        // start a transaction
        utx.begin();
        // Do work
        // Commit it
        utx.commit();
    }
}
```
JMS 2.0

```java
@Inject
private JMSContext context;

@Resource(mappedName = "jms/inboundQueue")
private Queue inboundQueue;

public void sendMessage(String payload) {
    context.createProducer().send(inboundQueue, payload);
}
```
Concurrency Utilities

```java
public class TestServlet extends HTTPServlet {
    @Resource(name="concurrent/MyExecutorService")
    ManagedExecutorService executor;

    Future future = executor.submit(new MyTask());

    class MyTask implements Runnable {
        public void run() {
            ... // Task logic
        }
    }
}
```
Batch Applications

```java
@ReadItem
public Account readAccount() {
    // read account using JPA
}

@ProcessItem
public Account processAccount(Account account) {
    // calculate balance
}

@WriteItems
public void sendEmail(List<Account> accounts) {
    // use JavaMail to send email
}
```
Java API for JSON

```java
JsonObject jsonObject = new JsonBuilder()
    .beginArray("phoneNumber")
    .beginObject()
    .add("type", "home")
    .add("number", "408-123-4567")
    .endObject()
    .beginObject()
    .add("type", "work")
    .add("number", "408-987-6543")
    .endObject()
    .endArray()
    .build();
```

```java

{ "firstName": "John", "lastName": "Smith", "age": 25,
  "phoneNumber": [
    { "type": "home", "number": "212 555-1234" },
    { "type": "fax", "number": "646 555-4567" }
  ]
}

Iterator<Event> it = parser.iterator();
Event event = it.next(); // START_OBJECT
event = it.next(); // KEY_NAME
event = it.next(); // VALUE_STRING
String name = parser.getString(); // "John"
```
Java API for WebSocket

@Singleton
@WebSocketEndpoint(path="/chat")
public class ChatServer {
  Set<Session> peers = ...

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

  @WebSocketClose
  public void onClose(Session session) {
    peers.remove(session);
  }

  ...

  @WebSocketMessage
  public void message(String message, Session client)
      throws IOException {
    for (Session session : peers) {
      if (!session.equals(client)) {
        session.getRemote().sendObject(message);
      }
    }
  }

}
Bean Validation 1.1

- Validating the data is important in each layer

- The JSR-303 Bean Validation standard helps this:
  - Extensible annotation based model
  - Standards based API with runtime validation capabilities
  - Standard API for accessing the metadata
Defining the constrains

- Annotation or XML based field or property annotation

```java
public class Address {
    @NotEmpty @Max(50)
    private String street1;

    @Max(50)
    private String street2;

    @Max(9) @NotNull
    private String zipcode;
}
```

- Could be applied to each part of a class
  - getters
  - class
  - superclasses
  - interfaces
Example

- **Message**
  In the case of validation error we can specify messages

```java
public class Address {
    @NotEmpty(message="The street is mandatory")
    private String street1;
}
```

- **Object graph validation**
  It start with the root node

```java
public class Order {
    @OrderNumber private String orderNumber;
    @Valid @NotNull private Address delivery;
}
```
Custom constrain

- Steps

```java
@Target({METHOD, FIELD})
@Retention(RUNTIME)
@ConstraintValidator(OrderNumberValidator.class)
public @interface OrderNumber {
    String message() default "{error.orderNumber}";
    String[] groups() default {};
}
```

- Example:

```java
public class Order {
    @NotNull @OrderNumber private String number;
}
public class OrderNumberValidator implements Constraint<OrderNumber> {
    public void initialize(OrderNumber constraintAnnotation) {
        // no initialization needed
    }

    /**
     * Order number are of the form Nnnn-nnn-nnn when n is a digit
     * The sum of each nnn numbers must be a multiple of 3
     */
    public boolean isValid(Object object) {
        if (object == null) return true;
        if ( !(object instanceof String) )
            throw new IllegalArgumentException("@OrderNumber only applies to String");
        String orderNumber = (String) object;
        if (orderNumber.length() != 12) return false;
        if (orderNumber.charAt(0) != 'N'
            || orderNumber.charAt(4) != '-'
            || orderNumber.charAt(8) != '-'
        ) return false;

        try {
            long result = Integer.parseInt(orderNumber.substring(1, 4)) + Integer.parseInt(orderNumber.substring(5, 8)) + Integer.parseInt(orderNumber.substring(9, 12));
            return result % 3 == 0;
        }
        catch (NumberFormatException nfe) {
            return false;
        }
    }
}
Resource handling in JEE

- The JEE components interact with each other and may access variety of resources (email, LDAP, file, …)
- The goal of JNDI is to provide a white board for this interaction. It is based on a naming service
JNDI

Java Naming and Directory Interface

The most important component of the JEEecosystem (the glue)

It responsibility is:

- Storing the references of different object in a searchable format
- Interaction with other directory based systems

The most important layers/modules:

- JNDI API
- Naming manager
- JNDI SPI
- Service Provider
  - LDAP
  - COS
  - RMI
  - DNS
  - NIS
  - File System
  - Windows Reg.
  - Novell File Sys.
Naming/Directory

Naming:
- LDAP, DNS, COS, ...
- Binding
- References
- Context name - object
- Name systems

Directory:
- The object may have properties too
- Property identifier
- Directory service
- Directory Schema
- Search, ...
Example

Binding:
- bind, rebind, unbind, rename
  Fruit fruit = new Fruit("orange");
  ctx.bind("favorite", fruit);
  Fruit fruit = new Fruit("lemon");
  ctx.rebind("favorite", fruit);
  ctx.unbind("favorite");
  ctx.rename("report.txt", "old_report.txt");

Subcontext
- createSubcontext, destroySubcontext
  ctx.createSubcontext("new");
  ctx.destroySubcontext("new");
Contexts and Dependency Injection

- CDI – JSR 299
- Lifecycle management for stateful components bound well-defined contexts (+ new conversation context)
- A type-safe approach to dependency injection
- Interaction via an event notification facility
- Reduced coupling between interceptors and beans
- Decorators—interceptors better suited for solving business concerns
- Unified EL integration (named beans)
  An SPI for developing portable extensions for the Java EE platform
CDI services

- Managed beans, one can access with the help of EL expressions
- The injected objects could be decorated
- Interceptor handling facilities
- Event handling facilities
- New scope
Managed Bean (CDI Bean)

- Common bean definition
- Life cycle of instance managed by container
- Basic set of services
- Resource injection
- Lifecycle callbacks
- Interceptors
- Foundation on which other specs can build
JSR-299 theme

Loose coupling...

@InterceptorBinding
@Inject
@Observes
@Qualifier

@Produces @WishList
List<Product> getWishList()

Event<Order>

@UserDatabase EntityManager

...with strong typing
Loose coupling

- Decouple server and client
  - Using well-defined types and “qualifiers”
  - Allows server implementation to vary
- Decouple lifecycle of collaborating components
  - Automatic contextual lifecycle management
  - Stateful components interact like services
- Decouple orthogonal concerns (AOP)
  - Interceptors
  - Decorators
- Decouple message producer from message consumer
  - Events
Strong typing

- Eliminate reliance on string-based names
- Compiler can detect typing errors
  - No special authoring tools required for code completion
  - Casting virtually eliminated
- Semantic code errors detected at application startup
  - *Tooling can detect ambiguous dependencies*
What can be injected?

- Defined by the specification
  - Almost any plain Java class (managed beans)
  - EJB session beans
  - Objects returned by producer methods or fields
    - Java EE resources (e.g., Datasource, UserTransaction)
  - Persistence units and persistence contexts
  - Web service references
  - Remote EJB references
- SPI allows third-party frameworks to introduce additional injectable objects
- Annotations aligned with JSR-330
CDI Bean

- Set of bean types (non-empty)
- Set of qualifiers (non-empty)
- Scope
- Bean EL name (optional)
- Alternatives
- Set of interceptor bindings
- Bean implementation
Bean services with CDI

- @ManagedBean annotation not required (implicit)
- Transparent create/destroy and scoping of instance
- Type-safe resolution at injection or lookup
- Name-based resolution when used in EL expression
- Lifecycle callbacks
- Method interception and decoration
- Event notification
Injecting a bean

- The container is going to instantiate it and inject it based on the annotation

```java
import javax.inject.Inject;
public class Printer {
    @Inject Greeting greeting;
    ...
}
```
Injection

- **@javax.inject.Inject** is used:

```java
public class OrdersBean {
    @Inject private OrdersDao dao;
}
```

- The „dao“ field is called „injection point“.
- Types of the injection point:
  - Field
  - Constructor
  - Setter
  - Initializer
Injection points

```java
public class OrdersBean {
    @Inject private OrdersDao dao;

    @Inject public OrdersBean(OrdersDao dao) {}

    @Inject public void init(OrdersDao dao) {}

    @Inject public void setOrdersDao(OrdersDao dao){}
}
```
Injection metadata

It provides information about the injection point:

```java
@Produces Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint
        .getMember().getDeclaringClass());
}
```

```java
@Produces @HttpParam(""")
String getParamValue(ServletRequest request,
    InjectionPoint ip) {
    return request.getParameter(ip
        .getAnnotation(HttpParam.class)
        .value());
}
```
Multiple implementations

- Two scenarios:
  - Multiple implementations of same interface
  - One implementation extends another

- Which implementation should be selected for injection?

```java
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Informal {}

@Informal
public class InformalGreeting extends Greeting {
    public String greet(String name) {
        return "Hi, " + name + "!";
    }
}
```
Examples

@Qualifier //retention & target ommitted
public @interface Synchronous {}
Scope types and context

- Absence of scope - @Dependent
  - Bound to lifecycle of bean holding reference to it

- Servlet scopes
  - @ApplicationScoped
  - @RequestScoped
  - @SessionScoped

- JSF conversation scope - @ConversationScoped

- Custom scopes
  - Define scope type annotation (i.e., @FlashScoped)
  - Implement context API

- Scopes are not visible to client
- No coupling between scope and use of type Scoped beans are proxied for thread safety
Conversation context

- Request <= Conversation << Session

- Boundaries demarcated by application

- Optimistic transaction
  - Conversation-scoped persistence context
  - No fear of exceptions on lazy fetch operations
EL Using the EL names

- @Named

```java
import javax.inject.Inject;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@Named
@RequestScoped
public class Printer {
    @Inject @Informal Greeting greeting;
    
    <h:form id="greetme">
        <p><h:outputLabel value="Enter your name: " for="name"/>
        <h:inputText id="name" value="#{printer.name}"/></p>
        <p><h:commandButton value="Say Hello" 
            action="#{printer.createSalutation}"/></p>
        <p><h:outputText value="#{printer.salutation}"/></p>
    </h:form>
```
Producer method

- A method whose return value is an injectable object
- Used for:
  - Types which you cannot modify Runtime selection of a bean instance
  - When you need to do extra and/or conditional setup of a bean instance

```java
private int maxNumber = 100;
...
@Produces @MaxNumber int getMaxNumber() {
    return maxNumber;
}

@Inject @MaxNumber private int maxNumber;
```
Example

```java
//this class is within a bean archive
class ConnectionProducer {
    @Produces Connection createConnection() {
        // create and return jdbc connection
        return createConnection();
    }
    // when the object gets out of scope
    void dispose(@Disposes Connection con) {
        con.close();
    }
}
```
Events

- Completely decouples action and reactions
- Observers can use selectors to tune which event notifications are received
- Events can be observed immediately, at end of transaction or asynchronously
Firing an event

public class GroundController {
    @Inject @Landing Event<Flight> flightLanding;

    public void clearForLanding(String flightNum) {
        flightLanding.fire(new Flight(flightNum));
    }
}
An event observer

```java
public class GateServices {
    public void onIncomingFlight(
        @Observes @Landing Flight flight,
        Greeter greeter,
        CateringService cateringService) {
        Gate gate = ...;
        flight.setGate(gate);
        cateringService.dispatch(gate);
        greeter.welcome Visitors();
    }
}
```

Takes event API type with additional binding type

Additional parameters are injected by the container
Interceptors

- Interceptors handle orthogonal concerns
- Java EE 5 interceptors bound directly to component
  - @Interceptors annotation on bean type

What’s the problem?
- Shouldn’t be coupled to implementation
  - Requires level of indirection

Should be deployment-specific
- Tests vs production
- Opt-in best strategy for enabling

Ordering should be defined centrally
Interceptor wiring in JSR-299

- Define an interceptor binding type

```java
public
@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@interface Secure {
}
```
Interceptor wiring in JSR-299

Marking the interceptor implementation

```java
public
@Secure
@Interceptor
class SecurityInterceptor {

    @AroundInvoke
    public Object aroundInvoke(InvocationContext ctx)
        throws Exception {
        // ...enforce security...
        ctx.proceed();
    }
}
```
Interceptor wiring in JSR-299

- Applying interceptor to class with proper semantics

```java
public
@Secure
class AccountManager {

    public boolean transfer(Account a, Account b) {
        ...
    }
}
```
Interceptor wiring in JSR-299

- Applying interceptor to method with proper semantics

```java
public class AccountManager {

    public @Secure
    boolean transfer(Account a, Account b) {
        ...
    }
}
```
Interceptor wiring in JSR-299

- Multiple interceptors
- Interceptors referenced by binding type
- Specify binding type in /META-INF/beans.xml to activate

```java
public class AccountManager {
    @Transactional
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

```xml
<beans>
    <interceptors>
        <class>com.acme.SecurityInterceptor</class>
        <class>com.acme.TransactionInterceptor</class>
    </interceptors>
</beans>
```

Interceptors applied in order listed
Composite interceptor bindings

- Interceptor binding types can be meta-annotations

```java
public
@Secure
@Transactional
@InterceptorBinding
@Retention(RUNTIME)
@Target(TYPE)
@interface BusinessOperation {}
```

```java
public
@BusinessOperation
class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```
Stereotypes

- Common architectural patterns – recurring roles
- A stereotype packages:
  - A default scope
  - A set of interceptor bindings
  - The ability to that beans are named
  - The ability to specify that beans are alternatives
Annotation jam

```java
public
@Secure
@Transactional
@RequestScoped
@Named
class AccountManager {
    public boolean transfer(Account a, Account b) {
        ...
    }
}
```

```java
public
@Secure
@Transactional
@RequestScoped
@Named
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
@interface BusinessComponent {}
```
EJB Goals

- Standard component architecture for building distributed business applications in Java
- Interoperability between enterprise beans and other Java Platform Enterprise Edition components, as well as non-Java applications
- Compatible with other Java APIs and with CORBA protocols
- Follow the *Write Once, Run Anywhere* philosophy of Java - an enterprise bean can be developed once and then deployed on multiple platforms without recompilation or source code modification
- Define the “contracts” that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime
Reprise: 3-tiered architecture

- **Database**
- **LDAP**

**Application logic components**

**Client**

**Component middleware**

**Back-end tier**

**Middle tier**

**Front-end tier**
EJB 3-tiered architecture

- **Back-end tier**
  - Database
  - LDAP

- **Middle tier**
  - Application logic components
  - Component middleware
  - EJBs
  - Application server

- **Front-end tier**
  - Client
Java EE 3-Tier Architecture
EJBs as Components

- Enterprise Java Beans are components that provide middle-tier business logic.
- And interact heavily with the data layer of the application.
- EJB framework conforms to and at the same time induces a 3-tier architecture for distributed applications.
EJB as Component Model Framework

- Programming model
- Standardized interfaces
- Runtime environment
- Built-in component services (persistence, transactions, security, etc.)
- Meta-data
- Deployment facilities
EJB Specification

- EJB is an open specification - any vendor can develop a runtime environment that complies with the specification.
- EJB code intended to be portable across brands (assuming uses only services defined by the spec, not additional vendor facilities).
- **EJB specs** have evolved:
  - Originated with IBM 1997
  - Later adopted by Sun (1.0 1998, 1.1 1999)
  - Enhanced under Java community process (2.0 2001, 2.1 2003, 3.0 2006)
- EJB 3.0 is a major departure from earlier versions, but backwards compatible (old code works with 3.0 but not vice versa).
Enterprise Beans

- Body of code with fields and methods
- Encapsulates business data or business logic that operates on the enterprise’s data
- Instances are created and managed at runtime by a Container (application server)
- Client access is mediated by the bean instance’s Container - isolates the bean from direct access by client applications (and other beans)
Enterprise Bean Portability

- If an enterprise bean uses only the services defined by the EJB specification (and not additional facilities peculiar to the vendor), the bean can be deployed in any compliant EJB Container.
- Can be included in an assembled application without requiring source code changes or recompilation.
- Component services information, such as a transaction and security attributes, are separate from the enterprise bean class - this allows the services information to be managed by tools during application assembly and deployment.
- Can be customized at deployment time by editing the bean’s environment entries and/or deployment descriptor.
EJB Container

- Manages every aspect of an enterprise bean at runtime and implements component services.
- When a client application invokes a method on an enterprise bean, the container first intercepts the invocation to ensure persistence, transactions and access control are applied properly to every operation a client performs on the bean.
- An enterprise bean cannot function outside of an EJB container.
EJB Container

EJB Containers manage enterprise beans at runtime
The EJB object

- There should be no direct connection to the EJB object
- The EJB container receives the calls and delegates them to a selected bean instance (Instance Pooling)
- The EJB object acts as a glue
- It is generated by the container
- Implicit MiddleWare
Resource Management

- Containers manage many beans simultaneously
- To reduce memory consumption and processing, containers pool resources
- When a bean is not being used, a container may place it in a pool to be reused by another client
- Or evict it from memory (passivate) and only bring it back (activate) when its needed
- While its reference on the client remains intact
- When the client invokes a method on the reference, the container re-incarnates the bean to service the request
Business Data and Methods

- An *entity* bean (aka persistence entity) represents persistent business data stored in one row of a database table, and may add behavior specific to that data - but the methods are often just getters, setters and finders.

- *Session* beans implement business processes and interact with clients.

- *Message-driven* beans combine features of a session bean and a message listener, allowing a business component to receive messages (and event notifications) asynchronously.
Business Interfaces

- A “business interface” is required for both session and message-driven beans (and for entities prior to EJB 3.0)
- The business interface of a message-driven bean is defined by the messaging type used (typically `MessageListener`), not by the developer
Multiple Interfaces

- If a bean class implements only a single interface (not counting standard interfaces such as java.io.Serializable or any of the javax.ejb interfaces), it is deemed the “business interface” and is by default a local interface unless designated by a @Remote annotation.

- A bean class may have multiple interfaces, but one or more must be designated as a business interface by either a @Local or @Remote annotation.

- Remote business interfaces support remote clients running on a different JVM or machine, to which the bean’s location is transparent.

- If there are only local interfaces, all clients must run in the same JVM as the bean, and the location of the bean is not transparent.
Example

@Stateless @Remote
public class CalculatorBean implements Calculator {
    public float add (int a, int b) {
        return a + b;
    }
    public float subtract (int a, int b) {
        return a - b;
    }
}

public interface Calculator {
    public float add (int a, int b);
    public float subtract (int a, int b);
}
Remote and Local Interfaces

To allow remote access, must decorate the business interface with the @Remote annotation

```java
@Remote public interface InterfaceName { ... }
```

OR decorate the bean class with @Remote, specifying the business interface(s)

```java
@Remote(InterfaceName.class) public class BeanName implements InterfaceName { ... }
```

To build an enterprise bean that allows only local access, optionally annotate the business interface of the enterprise bean as @Local

```java
@Local public interface InterfaceName { ... }
```

OR specify the interface by decorating the bean class with @Local and specify the interface name

```java
@Local(InterfaceName.class) public class BeanName implements InterfaceName { ... }
```
Enterprise Beans as Distributed Objects

- Business interfaces are types of Java RMI Remote interfaces.
- The `java.rmi.Remote` interface is used by distributed objects to represent the bean in a different address space (process or machine).
- An enterprise bean class is instantiated and lives in its container but can be accessed by client applications that live in other address spaces, using skeletons and stubs implemented by the container.
Stubs and Skeletons

1. Client invokes a method
2. Method invoked
3. Invoke on server
4. Communicate return value
5. Return results
Deciding on Local vs. Remote: Coupling

- Tightly coupled beans depend on one another
- For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled
- Tightly coupled beans are good candidates for local access
- Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access
Deciding on Local vs. Remote: Type of Client

- If an enterprise bean is accessed by application clients, then it should allow remote access.
- In a production environment, these clients almost always run on different machines than the Application Server.
- If an enterprise bean’s clients are web components or other enterprise beans, then the type of access depends on how you want to distribute your components.
Deciding on Local vs. Remote: Component Distribution

- Java EE applications are scalable because their server-side components can be distributed across multiple machines.
- In a distributed application, the web components may run on a different server than do the enterprise beans they access.
- Then the enterprise beans should allow remote access.
Deciding on Local vs. Remote: Performance

- Due to factors such as network latency, remote calls are often slower than local calls.
- On the other hand, if you distribute components among different servers, you may improve the application’s overall performance.
- Actual performance can vary in different operational environments.
Deciding on Local vs. Remote

- If you aren’t sure which type of access an enterprise bean should have, choose remote access, which gives more flexibility.
- In the future you can distribute your components to accommodate the growing demands on your application.
- It is possible for an enterprise bean to allow both remote and local access through different interfaces (the same business interface cannot be both a local and remote business interface).
EJB supported paradigms

- Implementing statless synchronous or asynchronous service
  - Web service
  - REST service
  - RMI service
  - JMS service

- Implementing statefull synchronous or asynchronous services
  - Web service
  - REST service
  - RMI service
EJB

Application Server

Remote Client

Remote Client

Local Client

EJB Container

BusinessLogic Tier

Session Beans

Message Driven Beans

Persistency Tier

JMS

JTA

JNDI

JDBC

RMI-IIOP

Threading

Pooling

Security

Web Services

Remote Client

JMS Provider

RDBMS

JPA Entities

Entity Manager

RDBMS
EJB Example
Entity handling

- EJB 1,2.0
  - Entity bean

- EJB 3.0, 3.1
  - Entity – JPA
  - = Hibernate
Session Bean

- Represents a single client (at a time) inside the Application Server
- Client invokes the session bean’s methods to execute business tasks
- When the client terminates, the session bean appears to have terminated and is no longer associated with the client
Stateful vs. Stateless

- There are two basic kinds of session bean: Stateless and Stateful
- *Stateful* session beans encapsulate business logic and state specific to a client
- Stateful beans are called "stateful" because they maintain conversational state between method invocations
- The state is held in instance variables (in memory) and is not persistent across executions
- The state disappears when the client removes the bean or terminates
Session bean client views

- Local
  ```java
  @Local
  public interface InterfaceName { ... }
  ```

- Remote
  ```java
  @Remote(InterfaceName.class)
  public class BeanName implements InterfaceName { ... }
  ```

- No Interface View
  ```java
  @Session
  public class MyBean { ... }
  ```
Session bean client views

- Web Service interface

```java
@Stateless
@WebService
public class HelloServiceBean {
    private String message = "Hello, ";

    public void HelloServiceBean() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```
SessionContext interface

- This is the environment provided by the container
  - `getCallerPrincipal`
  - `isCallerInRole`
  - `setRollbackOnly`
  - `getRollbackOnly`
  - `getUserTransaction`
  - `getTimerService`
  - `getBusinessObject`
  - `wasCancelCalled`
Paralell execution

- Statefull/Stateless session beans
  - Clients may start parallel execution
  - The container serializes the calls
  - The may be multiple bean instances, the calls to these instances are serialized
  - The interior of the bean could be not thread safe
  - The thread handling is forbidden
  - AccessTimeout
Singleton

- One by application
  - By default/JVM, for a given APP server it is true for cluster level also
  - It is live during the whole lifetime of the applications
  - It is for shared use

- Inicializálás:
  - @Startup (eager initialization)
  - @DependsOn

```java
@Singleton
public class B { ... }

@DependsOn("B")
@Singleton
public class A { ... }
```
Singelton parallellization

- Method level (Reentrant)
- Inside a given thread
  - In the case of write lock
    » The Read/write can continue without relasing the lock
  - In the case of read lock
    » The read can continue
    » In the case of write: IllegalLoopbackException

- Managed by container
  - Write/Read locks

- Bean managed
  - Synchronized/volatile

- Non singelton data member
  - Enabled but it is dangerous

- Timeout
  - AccessTimeout

@Lock(READ)
public class SomeClass {
    public void aMethod() { ... }
    public void bMethod() { ... }
    ...
}

@Singleton public class ABean extends SomeClass implements A {
    public void aMethod() { ... }
    @Lock(WRITE)
    public void cMethod() { ... }
    ...
}
Example

@Singleton
@ConcurrentManagement(BEAN)
public class DiscountRateBean {
    @PersistenceContext
    private EntityManager entityManager;
    private Rate rate;
    @PostConstruct
    private void init() {
        rate = entityManager.find(Rate.class, 1);
    }
    @PreDestroy
    private void destroy() {
        entityManager.merge(rate);
    }
    public synchronized void setRate(Rate rate) {
        this.rate = rate;
    }
    public synchronized Rate getRate() {
        return rate;
    }
}

@Startup
@Singleton
public class SharedBean implements Shared {
    private SharedData state;
    @PostConstruct
    void init() {
        // initialize shared data
    }
    ...
Stateful Session Beans

To conserve resources, stateful session beans may be *passivated* when not in use by the client.

Passivation means the bean's conversational-state is written to secondary storage (disk) and the instance is removed from memory.

If the client removes the bean or terminates, the session ends and the state disappears.

The client's reference to the bean is not affected by passivation: it remains alive and usable while the bean is passivated.

When the client invokes a method on a bean that is passivated, the container will activate the bean by instantiating a new instance and populating its conversational-state with the state previously written to secondary storage.
Stateless vs. Stateful

- Stateless session beans are made up of business methods that behave like functions: they operate only on the arguments passed to them when they are invoked (but can lookup state in a database or file).

- Stateless beans are called "stateless" because they are transient - they do not maintain a conversational state between method invocations.

- The bean’s instance variables may contain a state specific to the client during a single method invocation, but not retained when the method is finished.
Stateless Session Beans

- Each invocation of a stateless business method is independent from previous invocations.
- Because stateless session beans are "stateless" they tend to process requests faster and use less resources.
- All instances are equivalent – the EJB container can assign a pooled stateless bean instance to any client, improving scalability.
Session Bean Interfaces

- A client can access a session bean only through the methods in the bean’s business interface
- Can have more than one business interface
- A business interface can be either local or remote (or web service)
- Not required to implement any lifecycle methods, but may optionally do so and annotate as such (prior to EJB 3.0, all enterprise beans had to implement a “home” interface with lifecycle methods)
Lifecycle Methods

- The actual methods can have any names
- @PostConstruct: The container immediately calls the annotated method after a bean instance is instantiated
- @Init: Designates initialization methods for a stateful session bean
- @PrePassivate: Called before the container passivates a stateful bean instance
- @PostActivate: Called when a re-activated stateful bean instance is ready
- @Remove: Informs the container to remove the bean instance from the object pool after the method executes (not actually a callback)
- @PreDestroy: Called before the container destroys an unused or expired bean instance from its object pool
Lifecycle of a Stateful Session Bean

- Client initiates the lifecycle by obtaining a reference
- Container invokes the `@PostConstruct` and `@Init` methods, if any
- Now bean ready for client to invoke business methods
Lifecycle of a Stateful Session Bean

- While in ready state, container may passivate and invoke the `@PrePassivate` method, if any
- If a client then invokes a business method, the container invokes the `@PostActivate` method, if any, and it returns to ready stage
Lifecycle of a Stateful Session Bean

- At the end of the life cycle, the client invokes a method annotated `@Remove`
- The container calls the `@PreDestroy` method, if any
Lifecycle of a Stateless Session Bean

- A client initiates the life cycle by obtaining a reference
- The container invokes the `@PostConstruct` method, if any
- The bean is now ready to have its business methods invoked by clients
Lifecycle of a Stateless Session Bean

- Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods.
- At the end of the life cycle, the container calls the @PreDestroy method, if any.
JMS API

- Messaging Domains:
  - Publish/subscribe
  - Point-to-Point
Életciklusok

1. Dependency injection, if any
2. PostConstruct callback, if any

Does Not Exist

Ready

onMessage

PreDestroy callback, if any
Példa: Bean

```java
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/myQueue")
})
public class MyMessageBean implements javax.jms.MessageListener {

    public void onMessage(javax.jms.Message inMsg) {
        // implement the onMessage method
        // to handle the incoming message
        ....
    }
}
```
Container-Managed Transactions

- Container sets the boundaries of transactions, cannot use operations like `commit` or `rollback` within code.
- Container begins transaction immediately before enterprise bean method starts and commits just before method exits.
- **Transaction types:** Required, RequiresNew, Mandatory, NotSupported, Supports, Never.
Transactional Attributes

**Required** - If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

**RequiresNew** - If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction, starts a new transaction, delegates the call to the method, resumes the client’s transaction after the method completes; if the client is not associated with a transaction, the container starts a new transaction before running the method.

**NotSupported** - If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method; after the method has completed, the container resumes the client's transaction.

**Supports** - If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction; if the client is not associated with a transaction, the container does not start a new transaction before running the method.

**Mandatory** - If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction; if the client is not associated with a transaction, the container throws the `TransactionRequiredException`.

**Never** - If the client is running within a transaction and invokes the enterprise bean's method, the container throws a `RemoteException`.
Application-Managed Transactions

- The code in the session or message-driven bean explicitly marks the boundaries of the transaction.
- Useful for implementing multiple transactions within a single method or transactions that span multiple methods.
- Can use either Java Database Connectivity (JDBC) or the Java Transaction API (JTA).
- A JTA transaction can span updates to multiple databases from different vendors managed by the Java Transaction Service, but cannot support nested transactions.
- JTA supplies `begin`, `commit`, and `rollback` methods.
Using Transactions in Session Beans

- A stateless session bean with bean-managed transactions must commit or rollback before returning.

- A stateful session bean using JTA transactions retains its association with a transaction across multiple client calls, even if the database connection is opened and closed.

- A stateful session bean using JDBC transactions loses its transaction association if the connection is closed.
Examples

![Diagram of EJB Server and related components]

- Client
- EJB Server
- Components: X, Y
- Databases: A, B, C
- Queue: A
- Databases: B, C

![Diagram of EJB Server and related components]

- Client
- EJB Server
- Components: X, Y
- Databases: A, B, C
- Queue: A
- Databases: B, C

![Diagram of EJB Server and related components]

- Client
- EJB Server
- Components: X
- Databases: B, C

![Diagram of EJB Server and related components]

- Client
- EJB Server
- Components: X
- Database: A
- Database: B
# Transactions scope

<table>
<thead>
<tr>
<th>Transaction Attribute</th>
<th>Client’s Transaction</th>
<th>Business Method’s Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>None</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>RequiresNew</td>
<td>None</td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>Mandatory</td>
<td>None</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>NotSupported</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>None</td>
</tr>
<tr>
<td>Supports</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>Never</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>T1</td>
<td>Error</td>
</tr>
</tbody>
</table>

```java
// Bean-1
method-A()
  bean-2.method-B()

// Bean-2
method-B()
```

```python
TX?
```
Example

```java
@Stateless public class MySessionBean implements MySession {

    @TransactionAttribute(REQUIRED)
    public void someMethod(...) {
        java.sql.Connection con1;
        java.sql.Connection con2;
        java.sql.Statement stmt1;
        java.sql.Statement stmt2;

        // obtain con1 and con2 connection objects
        con1 = ...;
        con2 = ...;

        stmt1 = con1.createStatement();
        stmt2 = con2.createStatement();

        // Perform some updates on con1 and con2. The container
        // automatically enlists con1 and con2 with the container-
        // managed transaction.
        stmt1.executeQuery(...);
        stmt1.executeUpdate(...);

        stmt2.executeQuery(...);
        stmt2.executeUpdate(...);

        stmt1.executeUpdate(...);
        stmt2.executeUpdate(...);

        // release connections
        con1.close();
        con2.close();
    }
}
```
Resource methods

- setRollback
- getRollback
- afterCompletion
- SessionSynchronization
  - afterBegin
  - afterCompletion
  - beforeCompletion
Security

- Goal:
  - Help the developer
  - Help for the Deployer/System administrator managing the roles and rules

- Elements:
  - Roles
  - Methods decorations
  - Runtime context (getCallerPrincipal)
Security

- Security view
  - Defining roles
    - @RolesAllowed, @Permission, @DenyAll
  - The calling context is propagated (inter-enterprise passing)
- Run-as – new context but saves the old one (getCallerPrincipal) (security-identity)
- It is configured by the deployer
- Resource) handling
- Logging: java.security.Exceptions

```java
@Stateless public class EmployeeServiceBean
    implements EmployeeService{
    ...
    }
```

```java
@RolesAllowed("admin")
public class SomeClass {
    public void aMethod () {...
    public void bMethod () {...
    ...
}

@Stateless public class MyBean extends SomeClass implements A {
    @RolesAllowed("HR")
    public void aMethod () {...
    public void cMethod () {...
    ...
}}
```
Timers

“Every minute of every hour of every day”

@Schedule(minute="*", hour="*")

“Every Monday, Wednesday, and Friday at 30 seconds past noon”

@Schedule(second="30", hour="12", dayOfWeek="Mon, Wed, Fri")

“Every five minutes within the hour”

@Schedule(minute="*/5", hour="*")

“Every other hour within the day starting at noon on the 2nd Tuesday of every month.”

@Schedule(hour="12/2", dayOfMonth="2nd Tue")

// Generate account statements at 1 a.m. on the 1st of every month
@Schedule(hour="1", dayOfMonth="1")
public void generateMonthlyAccountStatements() { ... }

@Singleton
public class CacheBean {
    Cache cache;

    // Setup an automatic timer to refresh
    // the Singleton instance cache every 10 minutes
    @Schedule(minute="/10", hour="*", persistent=false)
    public void refresh() {
        // ...
    }
}

public void sendLunchNotification() { ... }
public interface javax.ejb.Timer {

    public void cancel();

    public long getTimeRemaining();

    public java.util.Date getNextTimeout();

    public javax.ejb.ScheduleExpression getSchedule();

    public javax.ejb.TimerHandle getHandle();

    public java.io.Serializable getInfo();

    public boolean isPersistent();

    public boolean isCalendarTimer();
}
Overview

- JEE
- Bean Validation
- CDI
- EJB