



Basics of programming 3

Java Enterprise Edition



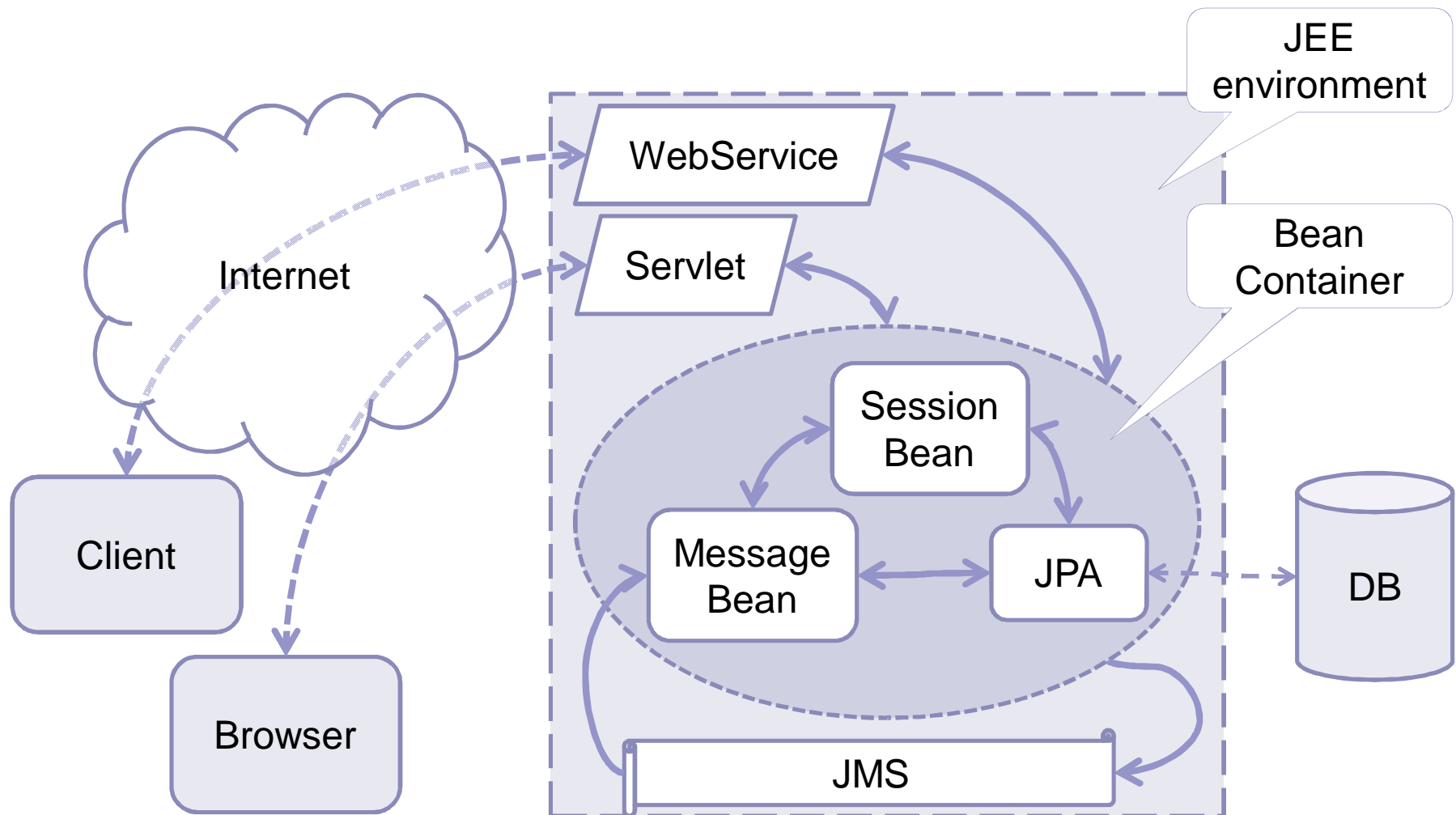
Introduction



Enterprise environment

- Special characteristics
 - continuous availability
 - component based
 - component life-cycle support needed
 - massive network support
 - detailed configuration support
- Java Enterprise Edition (JEE)
 - Java based environment
 - more than Java...

JEE architecture





JEE components

- Business logic

- Session beans

- implement a core functionality
 - stateful, stateless or singleton

- Message beans

- react to messages

- Database connection

- Entity beans

- Java Persistent API
 - connect to relational or NoSQL databases



JEE components

■ Interface

□ Web services

- standardized interface
- distributed communication
- stateless

□ Servlets

- connection to browsers
- produce web pages
- consume data from browsers
- JSP and JSF for ease-of-use



JEE components

■ Special connections

- Java Messaging Service (JMS)
 - handles, stores, delivers messages
 - connects to message beans
- Databases (DB)
 - usually relational
 - connection by JPA via hibernate etc.
 - OO – ER problems
 - associations
 - inheritance



JEE internals

■ Annotations rule

- connections between internal components
 - beans only specify class of parties (injection)
 - resources specified via JNDI
- business logic rules
 - concurrency
 - authorization
 - etc
- persistency
 - JPA is annotation-based
- web connections



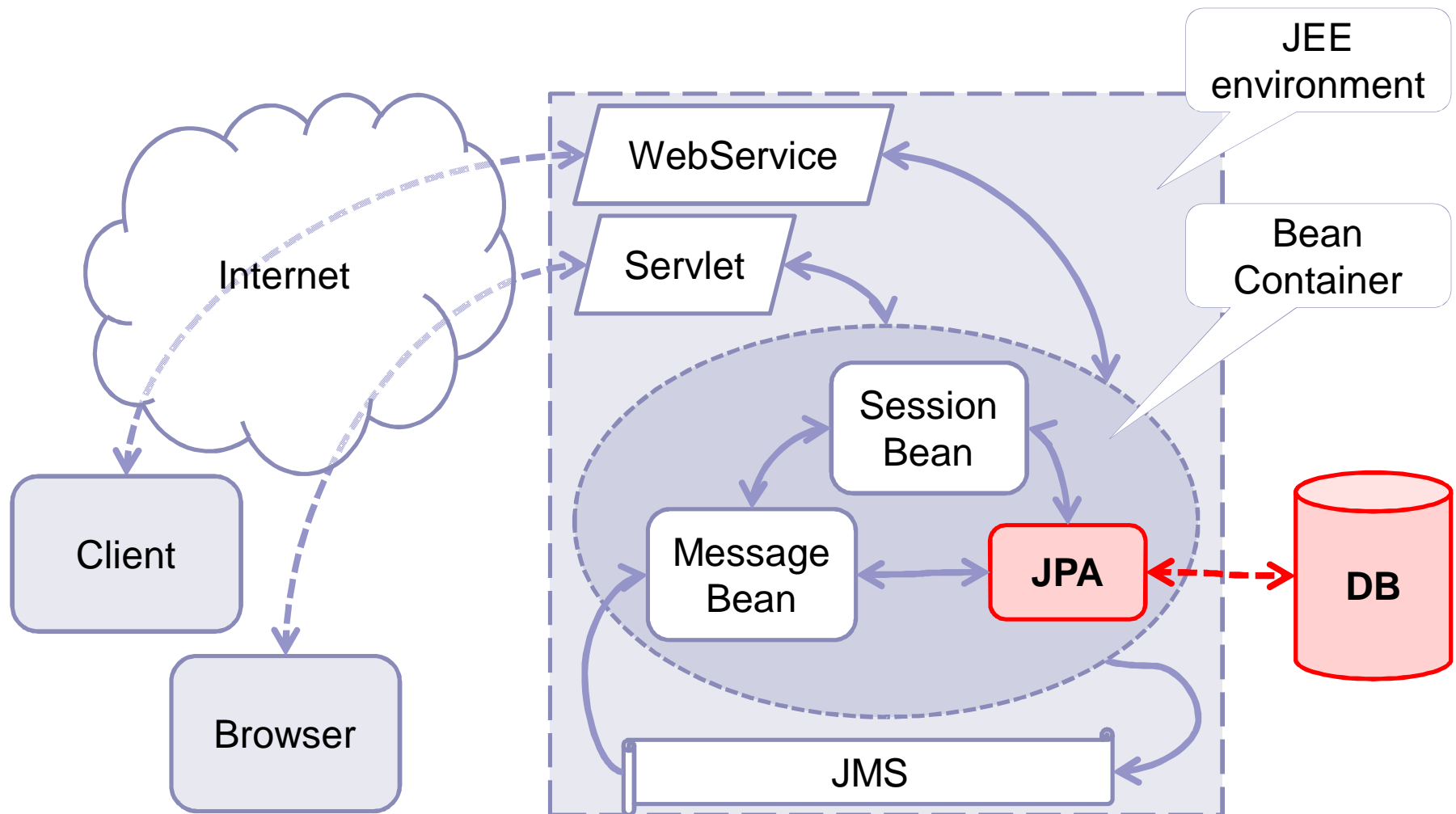
JEE internals

- Runtime enablement by container
 - creates components on demand
 - connects elements on demand
 - establishes DB and net connections on demand
- Container settings via configurations
 - vendor-specific settings
 - component deployment and server configuration separately



DB access

JEE architecture





Database access

- Database is separate component
 - SQL or NoSQL
- OO vs DB
 - different approaches
 - different optimizations
- Mappings
 - JDBC, JDO, JPA, etc



Database connection

- JDBC (Java DataBase Connectivity)
 - relation-based
 - uniform OO view on SQL concepts
 - java.sql, javax.sql packages
 - connection (commit, statement, query, parameters)
 - db metadata
 - resultset (record + record metadata)
 - wrapping for types
 - drivers for accessing different DBs



Database connection

- JDO (Java Data Objects)
 - direct mapping from OO to DB
 - ease of use
 - focus on domain object model
 - portability
 - run on multiple implementations
 - metadata highly portable
 - database independence
 - support many different kinds of transactional data stores
 - high performance
 - implementation can optimize data access



Database connection

- JPA (Java Persistence API)
 - OO to relational mapping
 - support for NoSQL available
 - configurable
 - inheritance
 - associations (multiplicity)
 - etc
 - provider
 - implements actual mapping
 - Hibernate, EclipseLink, OpenJPA, etc

JPA example

table
specification

```
@Entity
@Table(name = "children")
public class Child implements java.io.Serializable {
    private Long id;
    private String name;
    private int age;
    private List<Mischief> ms = new ArrayList<Mischief>();
    public Child(){}
    @Id @GeneratedValue
    public Long getId() { return id; }
    @Column(name = "NAME", nullable = false, length=80)
    public String getName() { return name; }
    @ManyToMany @JoinTable(name = "child_mischief", joinColumns = {
        @JoinColumn(name = "mischief_id") },
        inverseJoinColumns = { @JoinColumn(name = "person_id") })
    public List<Mischief> getMischief() {
        return ms;
    }
}
```

attributes
(id is mandatory)

getters with
column specs

association specs



JPA association mapping

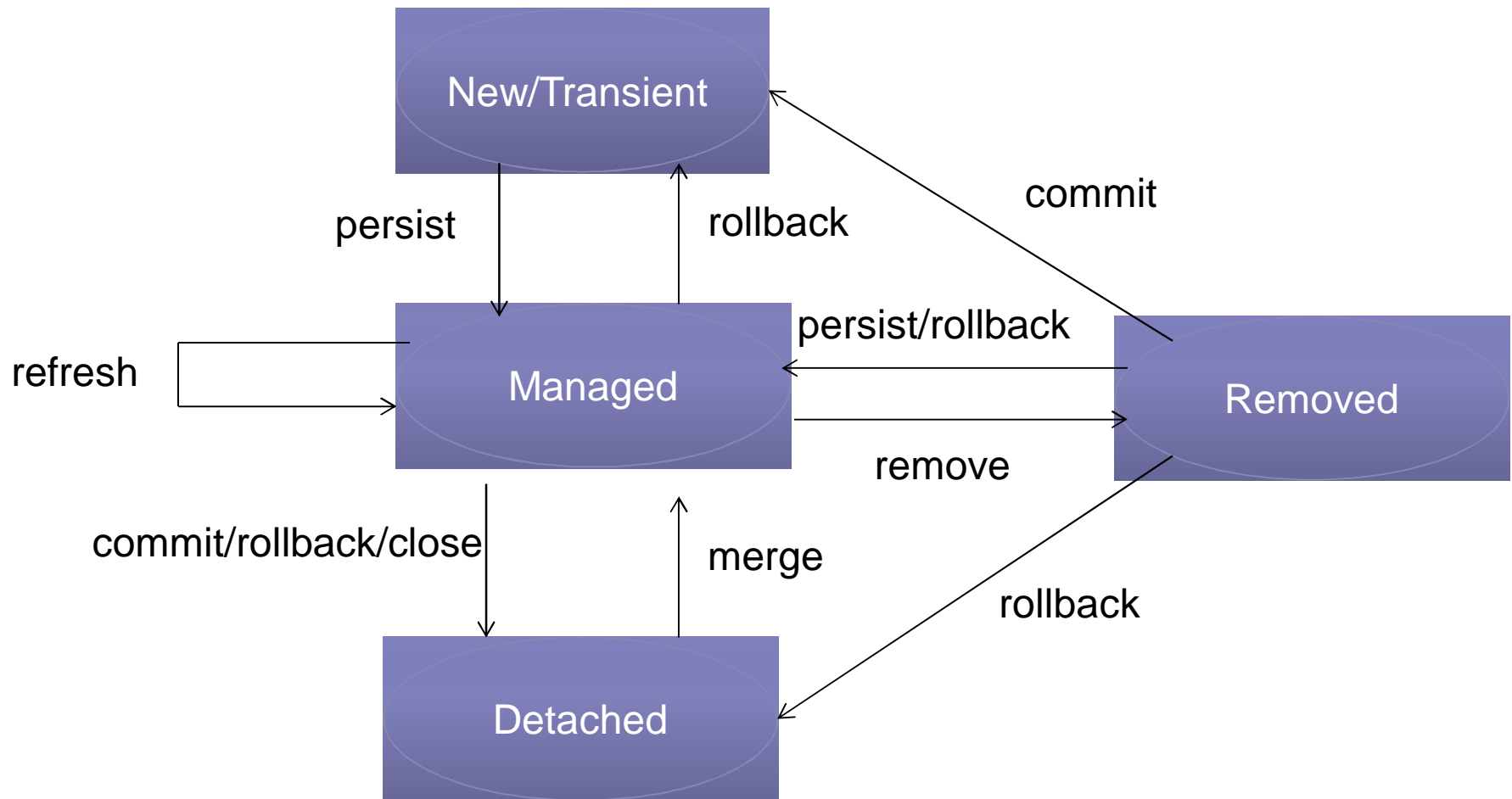
- ManyToOne
OneToMany
 - stored usually on *Many* endpoint
 - reverse navigation needs calculation
- OneToOne
 - on either end
- ManyToMany
 - join table is needed



JPA inheritance mapping

- Table Per Hierarchy
 - single table is required to map the whole hierarchy
 - extra column (discriminator) to identify the class
- Table Per Concrete class
 - tables only for concrete classes
 - duplicate column is added in subclass tables
- Table Per Subclass
 - tables are created for each class
 - inheritance by foreign key

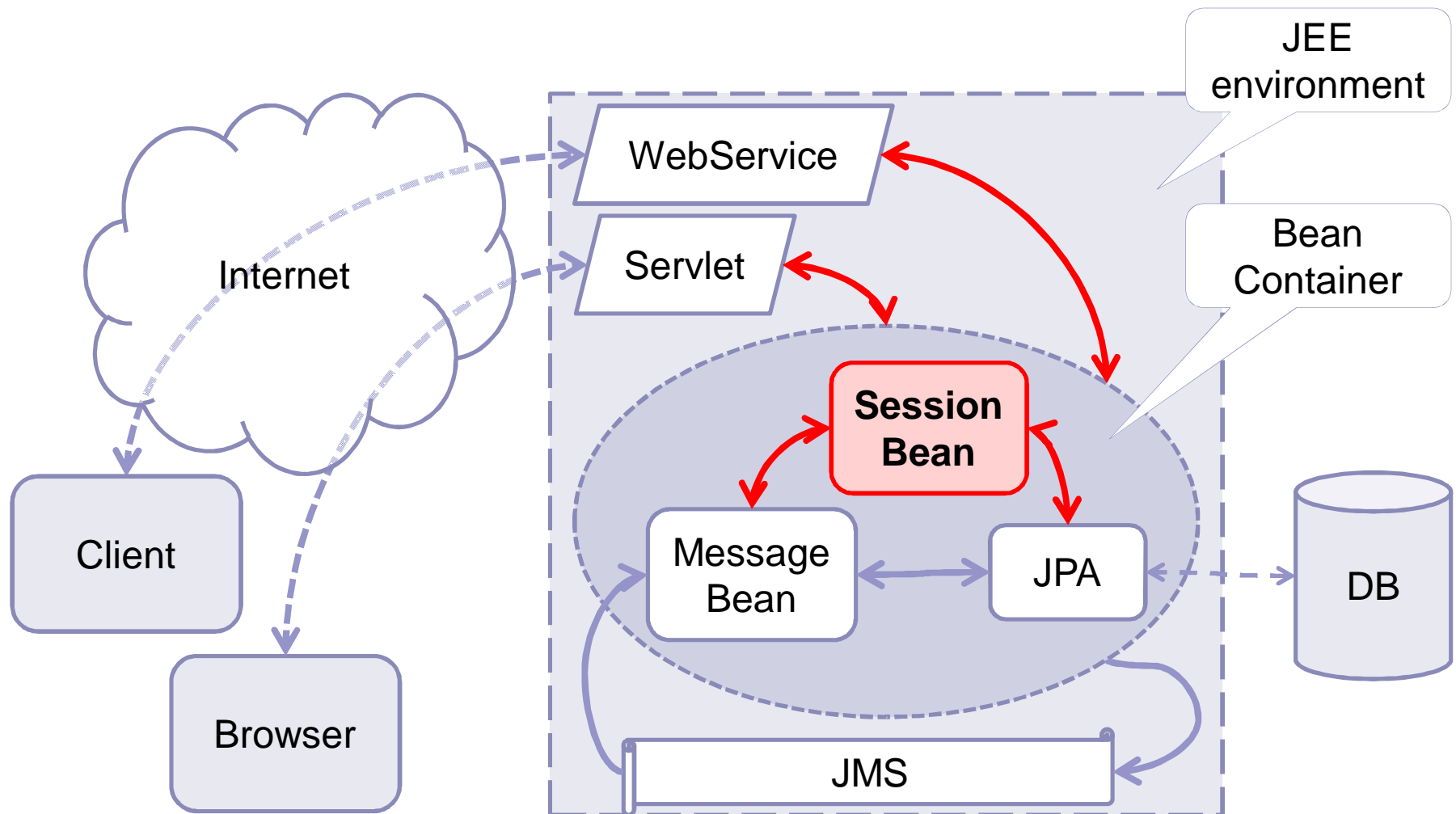
JPA Object lifecycle





Session Beans

JEE architecture





Session Bean

- JavaBeans

- are Java objects
- public setter-getter methods
 - heavy use of reflection

- Session Beans

- encapsulates business logic
- can be invoked programmatically
- can be stateful, stateless or singleton
- non-persistent



Stateful beans

- Attributes represent the state of a unique client/bean session
 - this state is called the **conversational state**.
 - state is retained for the duration of the session
- Similar to an interactive session
 - are not shared
 - can have only one client
 - when the client terminates, its session bean appears to terminate
- Can not implement a web service



Stateless beans

- Does not maintain a state with the client
 - attributes specific only during invocation
 - after invocation the client-specific state is not retained
 - stateless beans of the same type are considered similar
- Offers better scalability
 - no state, no state-maintaining overhead
- Can implement a web service
 - `@WebService`, `@WebMethod`



Singleton beans

- Instantiated once per application
 - exists for the lifecycle of the application
 - shared and concurrently accessed by clients
- Similar functionality to stateless session beans
 - but only single and continuous instance
- Can implement web service endpoints.
- Maintain their state between invocations
 - not across server crashes or shutdowns.

Session bean example

`@Stateless`

```
public class ElfServiceBean {  
    private String message = "Naughty ";  
    public void ElfServiceBean() {}  
    public String sayHello(String name) {  
        return message + name + ".";  
    }  
}
```

Stateless
Session Bean

Stateful Session
Bean

`@Stateful`

```
public class SantaBean {  
    @EJB ElfServiceBean elf;  
    int n;  
    public void SantaBean() { n = 0; }  
    public String sayMore(String name) {  
        return "Hohoho, " + elf.sayHello(name) + " #" + n;  
    }  
}
```

state
maintained

Referring a
stateless bean



Further session bean features

■ Asynchronous access

- @Asynchronous (method or class)
- special rules (returns *Future*<*T*>, etc)

■ Transactions

- @TransactionalAttribute(*TYPE*)
 - Required: do in transaction, create if necessary
 - RequiresNew : do in transaction, create new
 - Mandatory : do in transaction, error if none
 - Supports : no transaction needed
 - NotSupported : do outside of transaction
 - Never : if in transaction throws exception



Further session bean features

■ Security (authorization)

- `@DeclareRoles` : specifies considered roles (class)
- `@RolesAllowed` : how may call method
- `@PermitAll` : no restriction
- `@RunAs` : execute method with role

■ Lifecycle management

- `@PostConstruct`, `@PreDestroy`
 - execute method after construction / before deletion
- `@PostActivate`, `@PrePassivate`
 - execute method after/before storing bean in secondary storage
- `@Remove` : delete bean after executing annotated method

Complex session bean example

```
@Stateless
@DeclareRoles({"Elf", "Santa", "Parent"})
public class ElfServiceBean {
    public void ElfServiceBean() {}

    @Asynchronous
    @RolesAllowed("Santa")
    @TransactionalAttribute(REQUIRES_NEW)
    public Future<Goodness> calcGoodness(String n) {...}

    @RolesAllowed({"Santa", "Parent"})
    @TransactionalAttribute(MANDATORY)
    public void addMischief(String n, Mischief m) { ... }
    ...
}
```

Stateless Session Bean

Considered Roles

Asynch execution

Admin may call only

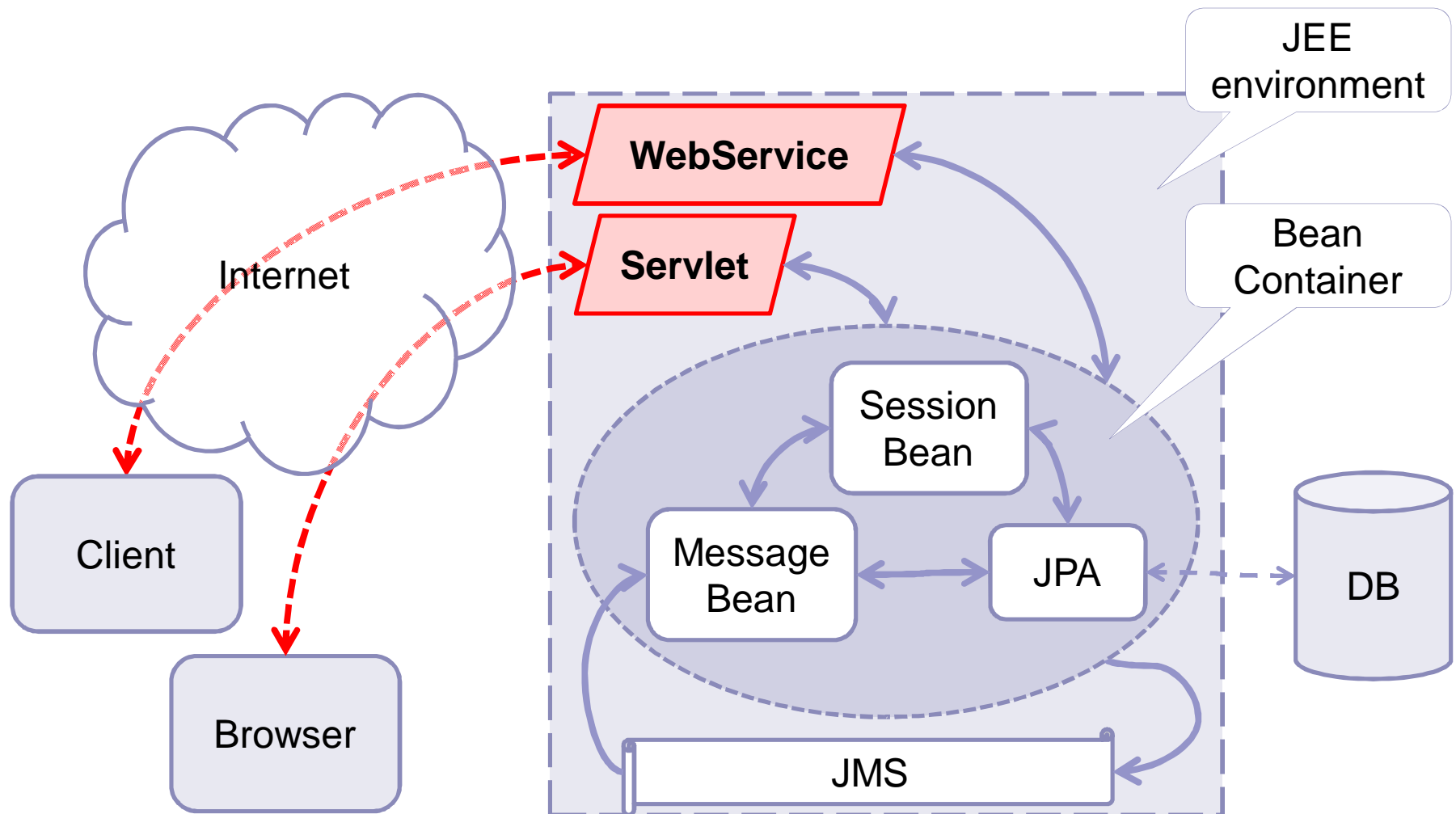
Create new transaction

Existing transaction is needed



Web connection

JEE architecture





Web connection

- Web pages
 - for browsers on client side
 - Servlet, JSP, etc generates web page
 - session handling - cookies
- Web service
 - remote method call
 - well defined protocol
 - SOAP, REST
 - protocol extensions (WS-*)



Communicating with browsers

■ Server to Client

- HTML page, images, JavaScript, etc
- generated by Servlet or JSP
 - servlet: Java code, output is content for browser
 - JSP: html-like code with commands
 - JSP is compiled to servlet internally

■ Client to Server

- HTTP/HTTPS protocol
 - get, post, put, delete, etc.
- HTML-form input
- AJAX, etc



Servlet

- Java object handling web communication
 - HTTPServlet for HTTP
 - *doGet, doPut, doUpdate, doDelete*, etc methods
 - access to session info
 - HttpSession
 - everything in Java
 - even printouts, result generation, etc.
- Configured via XML
 - access path in URL

Servlet example

```
public class HelloSanta extends HttpServlet {  
  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        String docType = "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD\"  
                        +\" HTML 4.0 Transitional //EN\">\n";  
        out.println(docType +  
                    "<HTML>\n<HEAD><TITLE>Santa's Page</TITLE></HEAD>\n" +  
                    "<BODY>\n<H1>Hohoho! </H1>\n</BODY></HTML>");  
    }  
}
```

HTTP protocol command
(GET method)

HTML source



JSP

- HTML-like language for dynamic web-pages
 - JSP elements
 - *HTML tags* - layout
 - *taglet tags* - special layout or logic (e.g. loops)
 - *Java code* - special logic (no support in taglib)
- Connection to rest of app via *managed beans*
 - managed beans generated in Java
 - almost POJO
 - managed beans accessed in JSP
 - bean: getter/setter methods

JSP example

```
<%@ page language="java" contentType="text/html" %>
<%@ page import="java.util.*" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html >
  <head><title>Santa's List</title></head>
  <body bgcolor="white">
    <h1>Children</h1>
    <table>
      <th>Name</th><th>Age</th><th>Goodness</th>
      <c:forEach items="{children}" var="current">
        <tr>
          <td><c:out value="{current.name}" /></td>
          <td><c:out value="{current.age}" /></td>
          <td><c:out value="{current.goodness}" /></td>
        </tr>
      </c:forEach>
    </table>
  </body>
</html >
```

basic structure in HTML

special tags for logic



Comparing Servlets and JSP

Characteristic	Servlet	JSP
Communication	same model	
Source lang	Java+println	HTML+tags
Auto recompilation	No	Yes
Approach	Imperative	Declarative
Readability	Hard	Easier
Layout check	Hard	Easier
Business Logic check	Easier	Hard



Web services

- Connection between applications
 - no direct UI dependency
 - usually over HTTP/HTTPS
 - might use JMS or other
 - extended protocols
 - encryption, authentication, authorization
 - transaction handling, atomicity
 - big data
 - XML or JSON based message formats
 - interface description is important
 - common data format is needed



Web services in Java

- Common protocols are language independent
 - mostly SOAP or REST
- Java mapping
 - JAX-WS
 - Java interface to/from WSDL
 - SOAP protocol conversion
 - JAX-RS
 - REST methods in Java
 - JAXB
 - Java data to/from XML representation
 - etc.

Web service example

```
@WebService public class Santa {  
    private String message = new String("Hohoho, ");  
    public void Santa() { }  
    @WebMethod public String sayHohoho(String name) {  
        return message + name + ".";  
    }  
}
```

Webservice specification

Method with Java
implementation

```
public class SantaClient {  
    @WebServiceRef(wsdl Location =  
        "META-INF/wsdl/localhost_8080/hs/SantaService.wsdl")  
    private static SantaService service;  
  
    public static void main(String[] args) {  
        hs.endpoint.Santa port = service.getSantaPort();  
        return port.sayHohoho(arg0);  
    }  
}
```

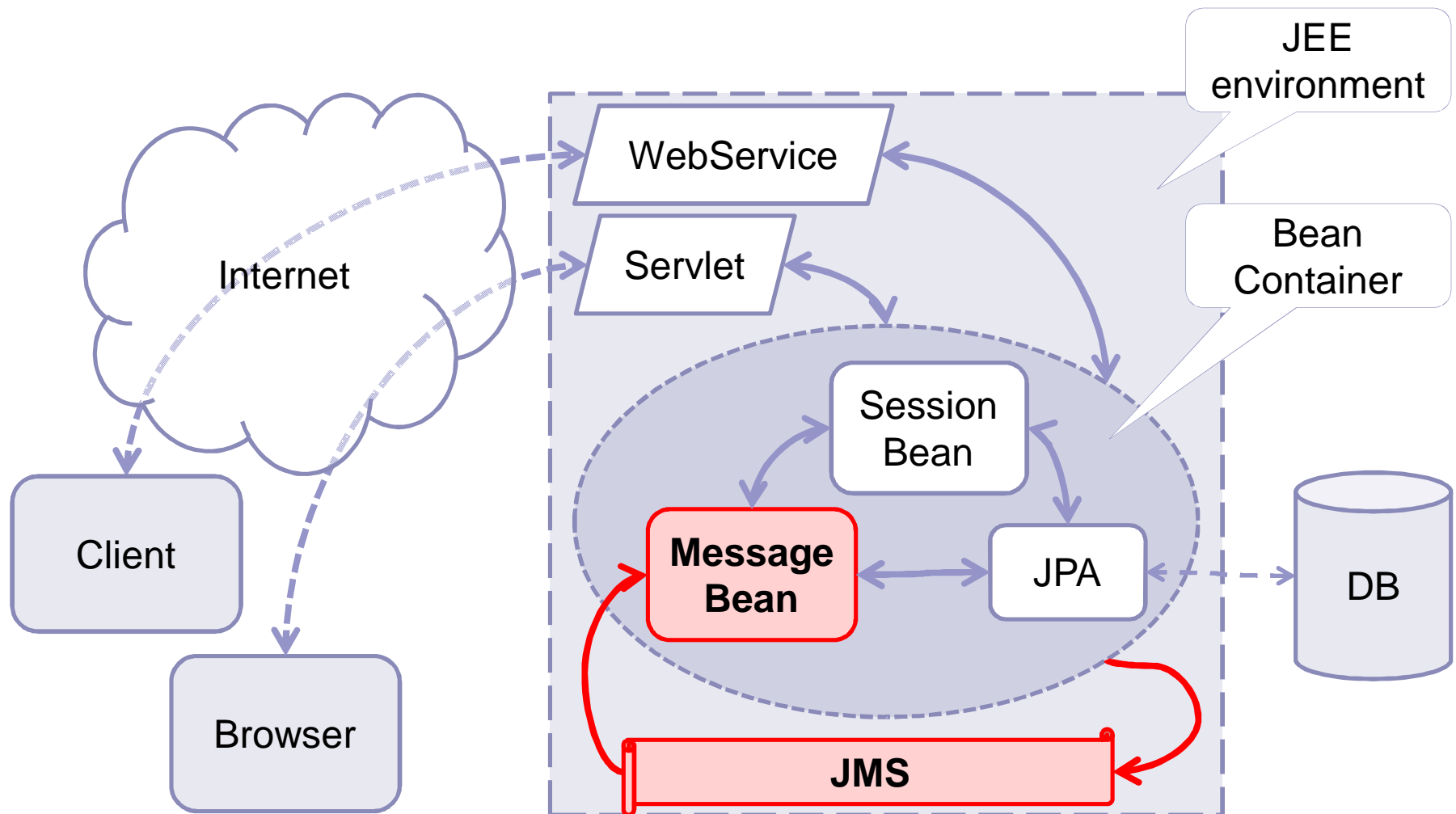
Server URL

Calling server side



Messaging

JEE architecture





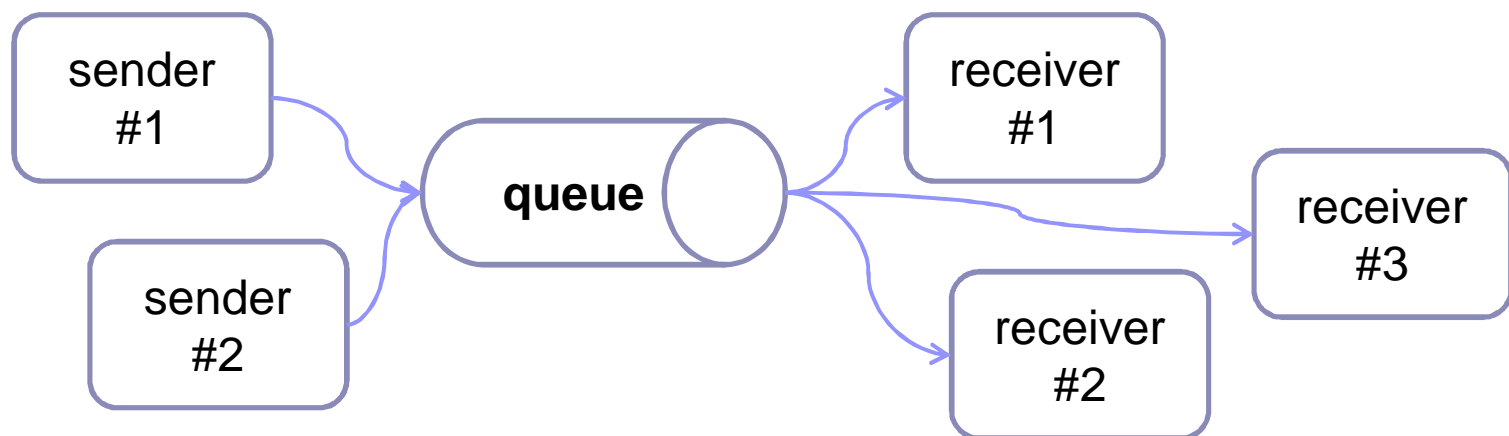
Java Messaging Service (JMS)

- Decoupled communication
 - Messages are sent between peers
 - via queue or topic
 - Producer (sender) creates message
 - Consumer (receiver) consumes message
- Messaging domains
 - describe message handling models
 - point-to-point: single recipient
 - publish-subscribe: multiple recipients

Point-to-point model

■ Characteristics

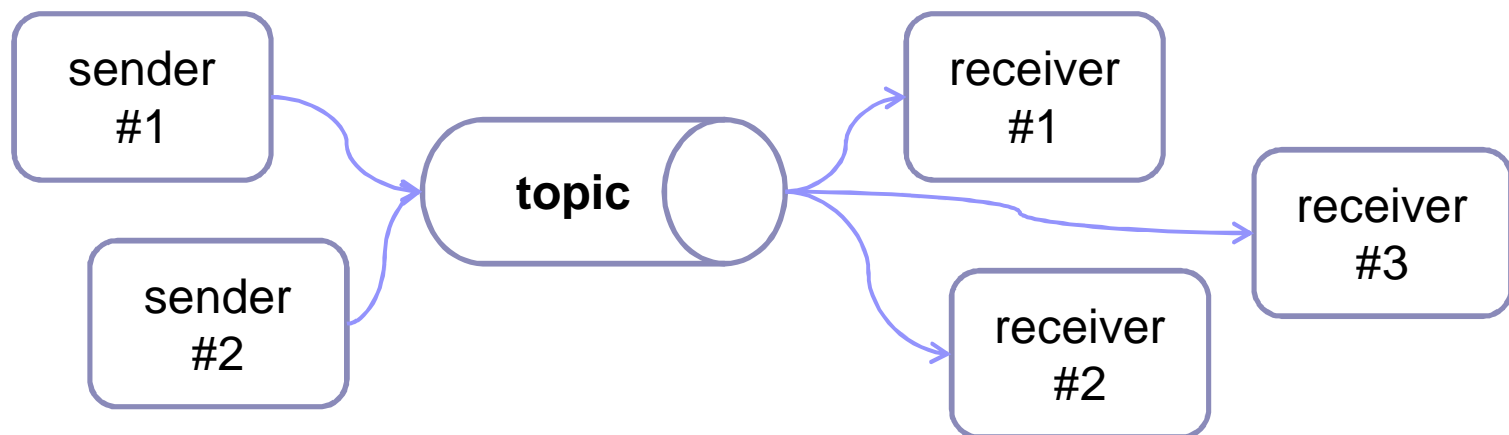
- each message has only one receiver
- sender and receiver have no timing dependencies
- receiver acknowledges successful processing



Publish-subscribe model

■ Characteristics

- each message can have multiple receivers
- publishers and subscribers have timing dependency
 - messages can be consumed only after subscription
 - message predating subscription are lost to subscriber





Message consumptions

- Consumption is decoupled
 - senders and receivers don't wait for each other
- Timing models for consumers
 - Synchronous (pull)
 - explicit fetch by calling the *receive* method on queue/topic
 - block or time out
 - Asynchronous (push)
 - message listener is registered
 - JMS delivers by calling the listener's *onMessage* method

JMS example (receiver)

message bean

queue name

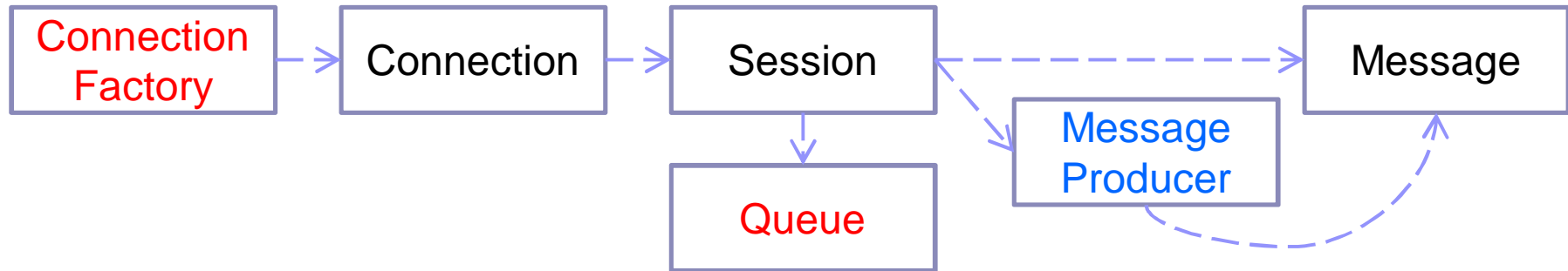
```
@MessageDriven(mappedName = "jms/deeds", activationConfig = {
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue")
})
public class DeedMessage implements MessageListener {
    @EJB E1fServiceBean e1f;
    public DeedMessage() {}
    public void onMessage(Message message) {
        try {
            TextMessage textMsg = (TextMessage)message;
            String text = textMsg.getText();
            storeReceipt(e1f.addDeed(textMsg));
        } catch (Exception e) {e.printStackTrace();}
    }
}
```

business logic

comm model

push method

JMS example (sender)



```
@Resource(mappedName="jms/ConnectionFactory")  
private static ConnectionFactory connectionFactory;  
  
@Resource(mappedName="jms/deeds") private static Queue queue;
```

```
connection = connectionFactory.createConnection();  
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);  
messageProducer = session.createProducer(queue);  
  
message = session.createTextMessage();  
message.setText("Isidor broke the window");  
messageProducer.send(message);
```



Configuration



Configuration

- Container manages resources
- Lookup
 - By name: *JNDI*
 - By type: *Dependency injection*
- Setup
 - Annotations
 - Deployment descriptor
 - XML based
 - vendor specific features